# Task Planning for High-Level Robot Control

Christian Dornhege

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

Technische Fakultät
Albert-Ludwigs-Universität Freiburg im Breisgau

Betreuer: Prof. Dr. Bernhard Nebel

Dezember, 2014

# Task Planning for High-Level Robot Control

Christian Dornhege

Dissertation zur Erlangung des akademischen Grades Doktor der Naturwissenschaften
der Technischen Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau

| | |
|---|---|
| Dekan: | Prof. Dr. Georg Lausen |
| Erstgutachter: | Prof. Dr. Bernhard Nebel |
| Zweitgutachter: | Prof. Dr. Wolfram Burgard |
| Tag der Disputation: | 11.05.2015 |

# Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Problem der Entscheidungsfindung für Roboter. Die Entscheidung, welche Aktion in der aktuellen Situation notwendig ist, um ein gegebenes Ziel zu erreichen, ist grundlegend für intelligente Systeme. Dabei ist es notwendig, dass die Vielfalt der in einem aktuellen Robotiksystem vorhandenen Fähigkeiten sinnvoll kombiniert werden. Dazu benutzen wir Handlungsplanung. Automatisierte Planungssysteme benötigen als Eingabe lediglich eine Beschreibung der Fähigkeiten des Roboters und des Ziels. Aufgrund dieser Beschreibungen erzeugt ein Planer für beliebige Situationen eine Aktionsfolge. Um die Stärken solcher Systeme in einem Roboter einzusetzen, müssen verschiedene Probleme gelöst werden. Zum einen ist es notwendig, dass die Ausdruckskraft der Planungssprache ausreichend ist, um Aufgaben und Fähigkeiten sowohl auf symbolischer, als auch auf geometrischer Seite zu beschreiben. Zum anderen muss solch ein Handlungsplaner sinnvoll in das Gesamtsystem eingebunden sein, so dass dieser als zentrale Entscheidungskomponente mit dem restlichen System zusammen arbeitet.

Der erste Teil der Arbeit zielt darauf korrekte Pläne für Planungsaufgaben in der Robotik zu erzeugen. Moderne Planer akzeptieren hauptsächlich symbolische Eingaben oder nur stark eingeschränkte numerische Probleme. Diese sind nicht ausreichend, um Fragen wie "Kann der Roboter diese Tasse jetzt greifen?" zu beantworten. Allerdings gibt es für solche Fragestellungen bereits Robotikalgorithmen, die diese korrekt und effizient lösen. Daher verfolgen wir den Ansatz solche externen Algorithmen in den Planungsprozess zu integrieren. Dies wird durch das Konzept von *semantic attachments* (semantische Anhänge) gewährleistet. Die Idee dabei ist, dass verschiedene logische Fragestellungen innerhalb des Planers durch externe Algorithmen beantwortet werden. Dies ist für den Planer transparent, so dass diese Techniken direkt auf aktuelle Planungssysteme übertragen werden können. Solche Anhänge sind in Form von Modulen implementiert, die beliebige Algorithmen über eine Schnittstelle aufrufen. Wenn während der Planung zum Beispiel ein Prädikat `greifbar` ausgewertet werden muss und dieses einen Anhang besitzt, dann wird zuerst das zugeordnete Modul ausgeführt. Der Wahrheitswert des Prädikats ist dann durch das Ergebnis des Modulaufrufs definiert. Dadurch lassen sich beliebige externe Algorithmen in einen Handlungsplaner integrieren. Die daraus erzeugten Pläne erlauben es zum Beispiel Manipulationspläne zu erzeugen, die symbolisch und geometrisch korrekt sind.

Ein weiteres Problem besteht darin, dass klassische Beschreibungen von Planungsproblemen nur endlich viele Objekte und damit Möglichkeiten eine Aktion zu parametrisieren zulassen. Eine Aktion, wie zum Beispiel "Stelle die Tasse auf den Tisch", kann

deshalb nicht die Entscheidung, wohin genau dies geschehen soll, widerspiegeln. Aktuelle Ansätze gehen dieses Problem an, indem nur endlich viele Optionen betrachtet werden. Ist darunter keine, die zur Lösung des Gesamtproblems führt, schlägt die Planung fehl. Das Hauptproblem besteht hier daran, dass jede vollständige Lösung unendlich viele Möglichkeiten untersuchen muss, was im Allgemeinen unentscheidbar ist. Dies ist auch für praktisch angewandte Systeme ein Problem. Unser Ansatz generiert daher neue Aktionsparametrisierungen durch einen beliebigen externen Algorithmus als Teil der Planung. Der Planer selbst kann entscheiden, ob noch weitere Möglichkeiten an diesem Punkt der Suche getestet werden müssen, oder die Suche vorerst an anderer Stelle fortgesetzt wird. Da dies prinzipiell auch unendlich viele Möglichkeiten sein können, war es notwendig einen neuen Suchalgorithmus zu entwickeln, der solche Probleme lösen kann, solang diese semi-entscheidbar sind. In Vergleichen mit klassischen Planungsalgorithmen zeigt dieses neue Verfahren in Manipulationsplanungsaufgaben deutlich bessere Laufzeiten und löst mehr Probleme.

Der zweite Teil der Arbeit beschäftigt sich mit der Anwendung dieser Planungsalgorithmen auf Robotersysteme. Zuerst werden dazu die Eigenschaften von Planungsproblemen in realen Umgebungen definiert. Diese sind ein ungenaues Wissen über die vorhandenen Objekte, sowie über deren Eigenschaften, eine unsichere Aktionsausführung, sowie externe Ereignisse. Um auf diese Dinge reagieren zu können, verwenden wir das Prinzip des *continual planning* (kontinuierliche Planung). Dazu wird der Planer in eine Schleife eingebunden. In jedem Schritt wird nur eine Aktion ausgeführt. Danach wird der aktuelle Zustand neu geschätzt und geprüft, ob der aktuelle Plan dafür noch zum Ziel führt. Ist dies nicht der Fall wird neu geplant. Dies ermöglicht es einen klassischen Planer auf einem Roboter einzusetzen, der mit den Eigenschaften der echten Welt zurecht kommen muss. Allerdings waren dafür einige Vereinfachungen dieser Eigenschaften notwendig. Dies führt zu einigen Annahmen über die Eigenschaften, die Anwendungsszenarien wie Haushaltsrobotik haben. Wir untersuchen diese insbesondere im Hinblick darauf, unter welchen Annahmen wir dennoch garantieren können, dass das Ziel erreicht wird. Konkret ist die wichtigste dabei, dass der Roboter sich nicht in eine Sackgasse manövrieren kann, aus der kein Weg mehr zum Ziel führt. Die Implementierung eines komplexen Robotiksystems im Bereich der mobilen Manipulation zeigt, dass dieses System in der echten Welt zuverlässig funktioniert.

Des weiteren untersuchen wir das Problem von Umgebungssuche für mehrere Roboter in 3d. Hier besteht das Ziel darin eine bekannte drei-dimensionale Umgebung mit den Sensoren der Roboter einmal vollständig zu erfassen. Da wir keine Annahmen über den Arbeitsbereich der Roboter, deren Sensoren, oder die Umgebung machen, ist das Problem schon von der Robotikseite her schwierig. In einem ersten Schritt berechnen wir daher mit einem Monte-Carlo Algorithmus eine endliche Menge von Blickpunkten, die in der Gesamtheit die Zielumgebung abdecken. Das verbleibende Problem besteht darin, eine Teilmenge von Blickpunkten auszuwählen, die immer noch die gesamte Umgebung erfasst und gleichzeitig sich von mehreren Robotern in minimaler Zeit abarbeiten lässt. Wir formulieren dies als klassisches Planungsproblem. Um dieses

zu lösen, wenden wir gierige (greedy) Algorithmen mit vergleichsweise geringen Laufzeiten, sowie einen Planer an. Des weiteren zerlegen wir das Gesamtproblem in zwei Teilprobleme, die sich als *set cover* und *traveling salesman* Problem darstellen lassen. Diese können wir wiederum mit einem Planer lösen. Eine ausführliche Evaluation in Simulation und auch mit echten Robotern zeigt, dass dieser Ansatz zu guten Lösungen führt und in der Praxis funktioniert.

Das Ziel dieser Arbeit ist es Handlungsplanungstechniken für die Entscheidungsfindung von mobilen Robotern zu entwickeln. Ein Hauptergebnis ist daher unser Planungssystem Temporal Fast Downward with Modules (TFD/M), das auf einem klassischen Planer aufbaut. Die dafür entwickelten Techniken sind essentiell, um korrekte Pläne zu erhalten und diese zur Aktionsauswahl des Roboters zu benutzen. Dabei ist es wichtig ein domänenunabhängiges System zu entwickeln, dass sich allgemeingültig anwenden lässt. Dies wird durch diverse Applikationen auf verschiedenen Robotern und Szenarien demonstriert.

# Abstract

Making intelligent decisions is an essential capability of any robotic system. A reasoning component that solves this problem must combine all available skills of a robot to solve a complex task. This thesis investigates task planning as such a reasoning mechanism. The strength of automated planning lies in the fact that a planner only requires a description of a robot's skills and the goal to reach. From this it computes action sequences for arbitrary situations. To apply planning techniques to robotics we must ensure that these are able to deal with the continuous and geometric nature of real-world tasks.

Therefore the first half of the thesis describes planning tasks that integrate external reasoners into the planning process. We consider planning operators to have a symbolic and geometric aspects. While planners deal very well with the former, symbolic task descriptions are not expressive enough for the latter. We introduce the concept of *semantic attachments* that connect a symbolic predicate like "is the cup graspable" with an external reasoner that computes this query. These allow us, for example, to describe and plan for mobile manipulation tasks soundly.

Another issue is that we cannot describe geometrical choices such as where to place an object on a table as planning tasks are required to be finite. To solve this we generate different instantiations of planning operators reflecting different options during the planning process by an external procedure. This made it necessary to develop a new search algorithm for planning with infinite branching factors. This new algorithm outperforms classical search algorithms on manipulation planning tasks.

The second half of the thesis investigates techniques for integrating task planning into robotic systems. First, we formulate properties of real-world tasks, e.g., unexpected action outcomes or uncertainty about the world. To tackle these we follow the concept of continual planning, where the planner is embedded in an observation, monitoring and replanning loop. We state what kind of simplifications we make to be able to apply our planner to real-world scenarios and specifically address under which assumptions such a system is guaranteed to reach a desired goal. We demonstrate this by implementing a complex mobile manipulation system.

Finally we investigate multi-robot coverage search in 3d. Here, the robots have to observe a known three-dimensional environment as quickly as possible with their sensors. This is a challenging robotics task especially in 3d scenarios that also contains a task planning problem. First, we generate a set of high-quality view poses. A subset of these have to be visited in a short amount of time to cover the search area. We introduce greedy and planning based methods to solve this problem and compare these algorithms in simulation and real-world experiments.

The main result of this thesis is our task planner Temporal Fast Downward with Modules (TFD/M) that extends a classical planner with the aforementioned planning techniques. This is used in our continual planning infrastructure that allows to embed this planner into a robotic system. Our evaluation in simulation and real-world experiments shows that task planning is a viable solution for high-level decision making in robotics. The techniques developed in this thesis are essential for that.

# Acknowledgements

A PhD thesis is a big endeavour. My scientific work would not have been possible without the contributions of many people that I wish to thank hereby.

First, I want to thank my advisor Bernhard Nebel that gave me the unique opportunity to follow each and every idea that I came up with. During my thesis I had the chance to work with many excellent colleagues and co-authors, among these are Alexander Kleiner, Andreas Hertle, Andreas Kolling, Armin Hornung, Bastian Steder, Bernhard Nebel, Christian Becker-Asano, Cyrill Stachniss, Dali Sun, Eduardo Meneses, Gabi Röger, Giorgio Grisetti, Johannes Aldinger, Julien Hué, Kai Wurm, Manuela Ortlieb, Marc Gissler, Maren Bennewitz, Matthias Teschner, Matthias Westphal, Michael Brenner, Michael Ruhnke, Patrick Eyerich, Rainer Kümmerle, Robert Mattmüller, Stefan Wölfl, Thomas Keller and Wolfram Burgard.

Every roboticist knows that as soon as you integrate your algorithms into a real-world robot as part of a large system the trouble starts. Finishing many awesome projects to a deadline, especially as part of a competition like RoboCup is always hard and can only be achieved in a team that brings the necessary motivation to work through nights and weekends. For keeping the spirits up I'd like to thank everyone that worked with me on any robotics project. A possibly incomplete list of these people consists of Alexander Kleiner, Andreas Hertle, Bastian Steder, Daniel Meyer-Delius, Felix Endres, Jörg Stückler, Jürgen Hess, Johann Prediger, Kai Wurm, Marc Gissler, Matthias Luber, Matthias Westphal, Michael Brenner, Michael Ruhnke, Michael Schnell, Moritz Göbelbecker, Rainer Kümmerle, Tobias Bräuer and Vittorio Ziparo. From these I'd like to especially thank Alexander Kleiner for showing me the ropes of robotics and Andreas Hertle for joining me in many fun endeavours robotics or otherwise.

Finally, I'd like to thank my family and friends for supporting me along the way and understanding when the robots demand more than a nine to five work schedule.

# Contents

# Chapter 1

# Introduction

Intelligent robots must be able to make rational decisions. This can affect low-level controls like moving an arm slightly more to the left or a high-level command to pick up a cup. The former are necessary to implement individual skills, while the latter combine these skills to solve a complex task. In this thesis we address the high-level control problem, i.e., what action should a robot execute to eventually reach a desired goal. A central question therefore is: What method is suited for high-level control?

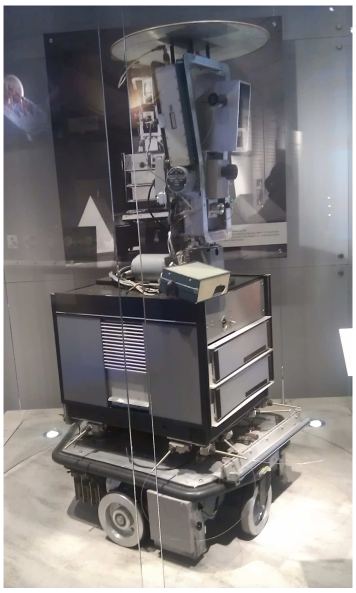


Figure 1.1: One of the first autonomous robots "Shakey" used a task planner. It is now on display at the Computer History Museum in Mountain View, California.

Task planning as the high-level decision making component dates back to the early days of robotics. In fact one of the first robots "Shakey" shown in Figure 1.1 inspired the inception of the STRIPS planning formalism that is still relevant today (Fikes and Nilsson, 1971). Here the world is modeled by a symbolic abstraction based on first-order logic and operators in the planning language directly map to executable action routines on the robot. Based on a description of the current world model the planner produces a sequence of operators—the plan—that leads to a goal state. Exemplary planning tasks use operators like `goto` a location, `push` a box, `turn on` a light or `climb` on a box. Note, that the planner was seen as a necessary tool for intelligent robots in contrast to considering robotics as an application domain for planning research. Of course both views go hand in hand.

This brings us to another question: Why isn't task planning ubiqutous in state of the art robotics applications? The robot "Shakey" could not turn on a light switch or climb on a box. In fact these are still not simple tasks for modern robots. The actions the planner wanted the robot to do were just not available and the skills that robots from decades ago possessed did not warrant the application of a planner to solve a task. McDermott (1992) in his work on *Robot Planning* says: "Nowadays nobody works on this problem any more. As stated, the problem turned out to be too hard and too easy. It was too hard because it was intractable. It was too easy because action sequences are not adequate as a representation of a real robot's program. [. . . ] Perhaps we should build robots that have their own need to plan before we think about robot planning again." This is exactly what happened. Robotics research focused on the more urgent needs like driving from A to B. Planning research addressed solving large planning tasks from other domains.

As long as there are only single or few skills to combine hand-scripting an executive might actually be sufficient. Other methods to explicitly define action choices like state machines can also be used to achieve a desired behavior and dependent on the application domain there might be different solutions that work. One could understand a planning-based approach as a state machine that assigns the correct action leading to the goal in any state. However, the main problem is to define this, which requires to solve the planning problem in any state. This is usually easy, when it is straight forward to determine what the next action is to be. It is important to understand what planning does well and therefore what kind of problems planning solves. McDermott (1992) also concludes ". . . that there is such a thing as robot planning, an enterprise at the intersection of planning and robotics". The interaction of multiple different skills is a strong suit of symbolic planning. An automated planner finds solutions for arbitrary situations that arise, handles generic goals and is able to accept task descriptions in a generic domain-independent way. The more skills must be combined in one system, the more these are dependent on each other, the more general the application scenarios and tasks are to be, the greater is the advantage and also the need to use an automated reasoning approach like task planning. Moreover, when there is more than one way to reach the goal, planning does not only pick one solution, but also solves optimization problems to return the best.

Nowadays robots possess a multitude of versatile skills. We have mobile manipulators with 6-dof arms, accurate sensors like laser scanners or RGBD cameras, and adequate computation power. On the algorithmic side simultaneous localization and mapping (SLAM), motion planning in high-dimensional configuration spaces, navigation, 3d perception and object recognition are all available. More specialized skills also exist, for example, towel folding, cooking pancakes, baking cookies, or solving a Rubik's cube. Of course these are not perfect, but robust enough to be applied in general. This makes it possible to combine these into a robot system that solves complex tasks. Creating such systems is certainly an interesting and relevant problem. One that task planning is well suited for.

So, what exactly are the issues when using a task planner to control the high-level decision making of a robot? First, resulting plans must be executable on the robot. This means that the underlying model that a symbolic planner reasons on should be an accurate representation of the continuous real world. In other words plans must be sound. Second, the planner must be integrated into the robot system as the executive. Not only is it necessary to execute the symbolic actions in the real world, but we also have to provide an initial state that represents the current world in a manner that the planner understands. Moreover, we need to consider that a real-world system does not behave exactly like it is modeled in a planning formulation.

A purely symbolic planning approach is not expressive enough to capture the geometric nature of real-world systems. For example, if a symbolic operator such as "pick up the cup from the table" is applicable also depends on the fact if there exists an inverse kinematics solution that moves the manipulator arm to the cup. If the planner asks the robot to execute this action, it will not be sure to succeed. However, we do have algorithms that can test this correctly. The challenge is to integrate external reasoners into the planning process, ideally keeping the generality of a domain-independent planning approach. This is usually refered to as *integrated task and motion planning.* [1] Such a planner is able to express symbolic and geometric semantics and leads to sound and thus applicable plans.

If we have such a generic planner, the second part of the problem is to apply that on a robot. A system acting in the real world is still prone to execution failures or unexpected outcomes. A closed loop control method that integrates the planner as the decision making component is necessary as a blind open loop execution is unlikely to be adequate. Perception actions that update the robot's knowledge about the world also need to be issued when necessary, while the system must handle the results of these. Nevertheless with all possible scenarios and robots that one can imagine acting in the real world is still challenging.

In this thesis we address several aspects of planning systems for robotics and the application of planning for solving various different problems. First, we present our approach to integrated task and motion planning called *semantic attachments.* Prior work on integrated task and motion planning was mainly used to solve specific problems, i.e., a domain-independent integrated planner did not exist. We addressed this shortcoming by basing our planner on the state of the art domain-independent planner Temporal Fast Downward (Eyerich et al., 2009). The result of this is the planner Tem-

[1]Although commonly named "integrated task and motion planning" we consider this an integration of arbitrary generic reasoners into planning that is not limited to motion planning.

poral Fast Downward with Modules (TFD/M). This also provides another advantage. Not only do we gain a generic system, but we also incorporate the advances from planning research such as an efficient state representation and the search guidance heuristic. Our idea uses the concept of *semantic attachments* to integrate geometric reasoning in a symbolic planner. Different aspects of a planning task's semantics are not expressed on a logical level, but provided by external reasoners that are called by the planner during planning. In contrast to approaches that verify such geometric constraints after symbolic planning or produce symbols for geometric relations before planning an integrated approach ideally computes costly geometric facts only if they are needed.

We have developed three kinds of semantic attachments that provide external semantics. They are used when checking operator applicability, providing numerical effects, or computing an operator's cost. Our planning tasks are described in PDDL/M, which is a slight adaption to the well known Planning Domain Definition Language (PDDL). Besides this language extension we also introduce a generic interface for implementing semantic attachments. We define these planning tasks and the interface and describe how a search-based planner is enhanced with semantic attachments.

Another problem for robotics planning tasks is action parametrization. To use a skill like "place an object" during planning it must be determined which object to place where. This relates to the problem of what is given to the planner as an input and what does the planner need to find out as part of its algorithm. Ideally we want to specify only what we clearly need to state. For the case of placing an object the planner needs to know what objects there are and what surfaces they can be placed on, but not where exactly to put them. This is a *choice* that the planner should be able to figure out.

State of the art approaches either relieve the planner of this by just passing complete parametrizations or somehow limit the planner's options to explore such choices to a fixed number of tries. The reason for this is that planning tasks with a high or unlimited number of choices are extremely hard to solve as these influence the branching factor of an underlying search process. We introduced a new generic interface for generating action parameters by an external procedure. In contrast to other approaches it is not required to limit the possible parametrizations to a fixed number. The planner itself asks for more when it deems this necessary for the search to succeed. As such tasks are undecidable in general we also present a new search algorithm that deals with this issue efficiently.

Next, we look into robotics applications of planning in general. For applying a planner to real-world tasks we follow the approach of continual planning, where the classical planner is embedded in a closed loop. Such a monitoring, planning, and execution system is commonly chosen. Using generic interfaces for action and perception we integrate the planner into the overall robotics system. An important factor is that such a system must react to unexpected action outcomes and perception from real-world data. We investigate in detail what guarantees this approach provides and state

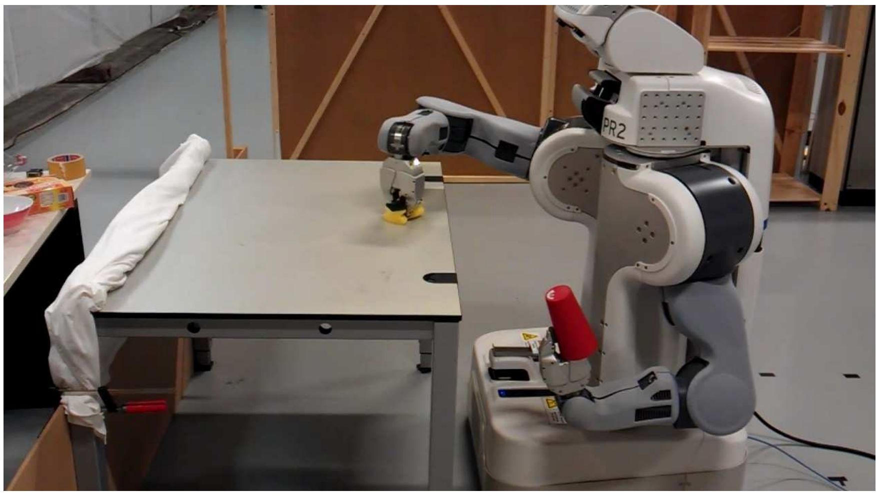


Figure 1.2: The mobile manipulation robot PR2 cleaning a table.

the assumptions that we make for this.

We implement a complex mobile manipulation system on the principle of continual planning using a classical planner with semantic attachments. An example scene is shown in Figure 1.2. This serves as a testbed and evaluations show that our assumptions are reasonable for real-world scenarios like service robotics. We also introduce techniques for efficiency in integrated planning systems and perform a quantitative evaluation of this system. Furthermore, we apply our continual planning system on multiple robots and scenarios to demonstrate that we have a truly generic and versatile system.

Finally, we investigate multi-robot coverage search in 3d. Here, multiple robots are to search a known environment as quickly as possible. Covering a three dimensional environment with sensors completely is a challenging robotics problem that has not been widely addressed. We use a generic problem formulation with no specific requirements on the sensor model, the robot's reachable space or the environment itself. Our solution first produces a set of high-quality view poses that covers the environment. For this we use a sampling-based approach, where the sampling space is explicitly computed as part of the algorithm. The resulting view volumes are then partitioned into individual parts.

Besides the geometric problem of finding good observation poses in a three dimensional world this task contains a hard combinatorial problem. What the first step leaves us with is the following problem: Find the fastest multi-robot trajectory, where each view part is covered by a view at least once. We formulate this as a task planning problem. However, large scenarios with tens to hundreds of poses and thousands of parts are not within the reach of modern planning systems. We therefore decompose this problem into two parts: A set cover problem, and a (multi-)traveling salesman problem. These can individually be expressed and solved as a planning task. As these are known problems we also investigate the use of a dedicated traveling salesman solver. We evaluate this system on small to large scale simulation tasks and in a real-world multi-robot scenario.

The remainder of this thesis is structured as follows. First, we explicitly state the scientific contribution. Then, Chapter 2 gives foundations for classical planning and introduces a running example for a robotics planning task. Planning tasks with semantic attachments are defined in Chapter 3, where we also show how to integrate these into our planner. Chapter 4 addresses planning tasks with infinite branching factors and our novel search algorithm. In Chapter 5 we state properties of real-world planning tasks and describe under which assumptions we can solve these using a classical planner in a continual planning loop. Here, we also show applications of this system to various robot scenarios. Chapter 6 introduces the problem of multi-robot coverage search in 3d and shows our solution as a combination of a geometric and a combinatorial planning problem. Finally, we conclude in Chapter 7.

## 1.1 Scientific Contribution and Publications

In general there have been the following major contributions. We introduced *semantic attachments* that constitute a generic methodolgy for integrating task and motion planning (Chapter 3). We formulated partially groundable planning tasks and developed a new search algorithm that can plan with an infinite branching factor (Chapter 4). Both are implemented in the integrated task and motion planner TFD/M that is the result of multiple iterations of applications and has therefore continuously been enhanced. We addressed continual planning for real-world applications. In Chapter 5 we formulated such tasks from a planning perspective and stated how these can be solved with a classical planner. We determined under what explicit assumptions our approach works. Chapter 5 also introduces new algorithms aimed at efficiency in integrated task and motion planning. Furthermore, in Chapter 6 we have developed a novel approach to the coverage search problem in 3d with multiple robots, where planning is a viable solution. These insights were contributed in the form of publications in workshops, conferences and journals that we list here in chronological order.

- C. Dornhege, A. Kleiner, A. Hertle, and A. Kolling. Multi-robot coverage search in 3d. *Journal of Field Robotics*, 2014. To appear.
- A. Hornung, S. Böttcher, J. Schlagenhauf, C. Dornhege, A. Hertle, and M. Bennewitz. Mobile manipulation in cluttered environments with humanoids: Integrated perception, task planning, and action execution. In *International Conference on Humanoid Robots (HUMANOIDS)*, 2014.
- A. Hertle, C. Dornhege, T. Keller, R. Mattmüller, M. Ortlieb, and B. Nebel. An experimental comparison of classical, FOND and probabilistic planning. In *German Conference on Artificial Intelligence (KI)*, 2014.

- C. Dornhege, A. Hertle, and B. Nebel. Lazy evaluation and subsumption caching for search-based integrated task and motion planning. In *IROS Workshop on AI-based robotics*, 2013a.

- C. Dornhege, A. Kleiner, and A. Kolling. Coverage search in 3d. In *International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2013b. (Best Paper Award Finalist)

- B. Nebel, C. Dornhege, and A. Hertle. How much does a household robot need to know in order to tidy up your home? In *AAAI Workshop on Intelligent Robotic Systems*, 2013.

- C. Dornhege and A. Hertle. Integrated symbolic planning in the TidyUp-robot project. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI II*, 2013.

- C. Dornhege and A. Kleiner. A frontier-void-based approach for autonomous exploration in 3d. *Advanced Robotics, 27(6)*, 2013.

- K. M. Wurm, C. Dornhege, C. Stachniss, B. Nebel, and W. Burgard. Coordinating heterogeneous teams of robots using temporal symbolic planning. *Autonomous Robots 34(4):277–294*, 2013.

- A. Hertle, C. Dornhege, T. Keller, and B. Nebel. Planning with semantic attachments: An object-oriented view. In *European Conference on Artificial Intelligence (ECAI)*, 2012.

- C. Dornhege and A. Kleiner. A frontier-void-based approach for autonomous exploration in 3d. In *International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2011.

- M. Westphal, C. Dornhege, S. Wölfl, M. Gissler, and B. Nebel. Guiding the generation of manipulation plans by qualitative spatial reasoning. *Spatial Cognition and Computation: An Interdisciplinary Journal 11(1):75–102*, 2011.

- K. M. Wurm, C. Dornhege, P. Eyerich, C. Stachniss, B. Nebel, and W. Burgard. Coordinated exploration with marsupial teams of robots using temporal symbolic planning. In *International Conference on Intelligent Robots and Systems (IROS)*, 2010.

- M. Gissler, C. Dornhege, B. Nebel, and M. Teschner. Deformable proximity queries and their application in mobile manipulation planning. In *Symposium on Visual Computing (ISVC)*, 2009.

- C. Dornhege, M. Gissler, M. Teschner, and B. Nebel. Integrating symbolic and geometric planning for mobile manipulation. In *International Workshop on Safety, Security and Rescue Robotics (SSRR)*, 2009b.

- C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel. Semantic attachments for domain-independent planning systems. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009a.

## 1.2 Collaborations

The work in this thesis resulted in multiple scientific publications. As common in state of the art research this scientific work was performed in collaboration with other researchers. The consistent use of “we” within this thesis reflects this fact. As it is not possible from the study of these publications to identify each author’s individual contribution I will now list all publications contained in this thesis that I co-authored, relate them to the following chapters, and explicitly state my contributions therein.

Throughout any development on the planner TFD/M I have worked on the design and implementation of every feature. The first concept for integrating semantic attachments in Temporal Fast Downward presented in Dornhege et al. (2009a) was created with P. Eyerich, T. Keller and B. Nebel and implemented with P. Eyerich and T. Keller, where I mainly contributed the translation component. This work along with the following applications and extensions is described in Chapter 3. The experimental evaluation was also performed with P. Eyerich, T. Keller and B. Nebel. For the resulting manipulation planner described in Dornhege et al. (2009b); Gissler et al. (2009) all authors, i.e., M. Gissler, M. Teschner, B. Nebel and I conceived the algorithm and evaluation. M. Gissler and I implemented the algorithm and executed the experiments, where I implemented the motion and the task planner. The algorithm from Westphal et al. (2011) was designed by M. Westphal, S. Wölfl and me and implemented mainly by M. Westphal and me, where I implemented the guidance heuristic based on the qualitative planner from M. Westphal. The experiments were performed by M. Westphal, M. Gissler and me and evaluated by all co-authors. Together with A. Hertle and T. Keller I contributed to the design of the planning language OPL presented in Hertle et al. (2012), the translation algorithm of OPL to PDDL and the evaluation of the OPL interface.

I designed, implemented and evaluated the algorithm for searching planning tasks with infinite successors shown in Chapter 4. Using this algorithm A. Hornung, A. Hertle and I designed the planning domain for the work presented in Hornung et al. (2014) and we applied the planning architecture from Chapter 5 to humanoid Nao robots. Chapter 5 is mainly based on the next three papers that illustrate several aspects of planning for real-world tasks. The planning formulation for the Nao robots was based on the domain designed by A. Hertle and me for the *TidyUp* mobile manipulation scenario presented in Dornhege and Hertle (2013). Here, the robot implementation and integration was performed by F. Endres, A. Hertle and J. Hess and me. The simplifications making this planning domain applicable to real-world tasks were formalized in Nebel et al. (2013) with B. Nebel and A. Hertle. The algorithms presented in Dornhege et al. (2013a) were created and implemented by A. Hertle and me, where I implemented lazy module evaluation and subsumption caching (see Section 5.5). I also performed the experimental evaluation of this work. In addition Chapter 5 also includes the following related work and application examples. The comparison of planning methodologies and their evaluation in Hertle et al. (2014) was designed by A. Hertle, T. Keller, R. Mattmüller, M. Ortlieb, B. Nebel and me. The integration of cost modules (see Chapter 3) in the work presented in Wurm et al. (2010) was performed by P. Eyerich and me. K. Wurm and I implemented the exploration algorithms for the work in Wurm et al. (2010, 2013), where I implemented the planning variant. K. Wurm, C. Stachniss and I designed the experimental evaluation and K. Wurm and I executed the experiments.

For the topic of multi-robot coverage search addressed in Chapter 6 the candidate generation algorithm presented in Dornhege and Kleiner (2011, 2013) was designed, implemented and evaluated by A. Kleiner and me. The formulation and algorithm for multi-robot coverage search in Dornhege et al. (2013b, 2015) as well as the evaluation was designed by A. Kleiner, A. Kolling, A. Hertle and me. The implementation and experiments were performed by A. Hertle, A. Kleiner and me.

# Chapter 2

# Foundations

In this chapter we introduce fundamental concepts relevant for task planning in a robotics context. The following chapters will build on these concepts extending the basic capabilities and applying the resulting formulation to high-level robot control. First, we define classical planning tasks. Then we describe how such planning tasks are formulated in the Planning Domain Definition Language (PDDL), where we focus on a domain formulation for robotics. We will use this domain as a running example throughout this work.

## 2.1 Classical Planning Tasks

There are many formulations of classical planning tasks in the literature. Our definitions are based on those for PDDL planning tasks (Fox and Long, 2003). A planning task consists of a planning domain and planning problem. The domain describes how a robot's world model is constructed and the generic actions that can be executed are modeled as schematic operators. A planning problem instantiates a world model and the possible actions for a specific situation. Therefore the domain is usually given as part of a system and problems are generated during runtime from sensor data reflecting the current world and goals.

First, we define the planning domain.

**Definition 1.** *A planning domain is a tuple $(P, \mathcal{O})$, where $P$ is a finite set of predicate symbols, each with an associated arity, and $\mathcal{O}$ is a finite set of schematic operators.*

Planning tasks use first-order logic formulae. The terms of our logic are constant and variable symbols drawn from a finite set $\mathcal{V}_{\mathcal{F}}$. No interpreted first-order function symbols are allowed. We call these terms *classical terms*. The predicate symbols together with terms form atomic formulae and expressions of our language.

**Definition 2.** *The atomic formulae are formed from an $n$-ary predicate symbol from $P$ with $n$ classical terms.*

Before addressing planning operators in Definition 5 we define a specific planning problem for a planning domain and the resulting state space.

**Definition 3.** *A planning problem is a tuple $(I_L, \phi_G, O_F)$, where $I_L$ is the initial state, $\phi_G$ is a first-order formula describing the goal and $O_F$ is a finite set of objects.*

We refer to a planning problem with its respective planning domain as a planning task. Its state space is defined by the grounded representation as follows. We call $O_F$ the set of *PDDL objects*, which is used for grounding *classical terms*. Grounding is the syntactic procedure that replaces each term with an object from $O_F$. Constant terms are replaced by a given representation in $O_F$. Variable terms are grounded by replacing all variables in an $n$-ary predicate with all possible object combinations from $O_F$ leading to many possible grounded instantiations. Boolean fluents are formed from grounding predicates. The grounded predicates form the set of Boolean fluents $\mathcal{F}_\mathcal{L}$. A state of a planning task is represented by the values of Boolean fluents. We now define a state for a problem instance.

**Definition 4.** *A state* s *is a function that maps each Boolean fluent from $\mathcal{F}_\mathcal{L}$ to $\{true, false\}$, i.e.:*

$$s : \mathcal{F}_\mathcal{L} \rightarrow \{true, false\}$$

*The set of all possible assignments for $\mathcal{F}_\mathcal{L}$ is the set of states $S$.*

$\mathcal{F}_\mathcal{L}$ is finite as it is grounded from a finite set $O_F$. $I_L$ is a variable assignment that represents the initial state.

**Definition 5.** *A schematic operator is a tuple $(\phi, e, c)$, where $\phi$ is a first-order formula—the precondition, $e$ an effect and $c \in \mathbb{R}_{>0}$ the operator cost. First-order formulae are formed over the atomic expressions from $P$ with the usual logical connectives. An effect is defined by the finite application of the following rules:*

- *$e$ is called a simple effect if $e$ is a predicate literal (i.e., a predicate or negated predicate).*
- *$e_1 \wedge \ldots \wedge e_n$ is a conjunctive effect for effects $e_1 \ldots e_n$.*
- *$\forall x : e$ is a universal effect for a variable symbol $x \in \mathcal{V}_\mathcal{F}$ and an effect $e$.*

*We call the set of free variables of an operator the* ***parameters*** *of an operator. The free variables of an operator $(\phi, e, c)$ are the free variables of its precondition and its effect, i.e., $free(\phi) \cup free(e)$, where the $free(e)$ is defined as:*

$$free(e) = \begin{cases} free(\psi) & \text{if } e = \psi \text{ or } e = \neg\psi \text{ is a simple effect} \\ \bigcup_{i=1}^{n} free(e_i) & \text{if } e = e_1 \wedge \ldots \wedge e_n \text{ is a conjunctive effect} \\ free(e') \setminus \{x\} & \text{if } e = \forall x : e' \text{ is a universal effect} \end{cases}$$

*A* ***grounded operator*** *is an operator that has no parameters.*

We refer to the cost of an operator $o$ as $cost(o)$. If the cost is not explicitly given, unit costs are used, i.e., $c = 1$. A grounded operator describes a transition between two states. Grounding an operator is performed by instantiating each of its parameters with an object from $O_F$ for variables in $\mathcal{V}_{\mathcal{F}}$ leading to $|O_F|^m$ grounded operators for a schematic operator with $m$ parameters. All free variables in the precondition and effect are thereby substituted by concrete objects. We now define when an operator $o$ can be applied in a state $s$—or $applicable(o, s)$—and how the successor state $s' = app(s, o)$ resulting from applying $o$ in $s$ is derived.

**Definition 6.** *An operator $o = (\phi, e, c)$ is applicable in state $s$ iff $o$ is grounded and the interpretation of its precondition $\phi$ in the state $s$ given by $\phi^I$ is true. Applying an operator $o$ in a state $s$ results in the successor state $s'$ given by $s' = app(s, o)$.*

We now give this interpretation $I$ of $\phi$ in a state $s$. Mostly the interpretation of first-order formulae $\phi^I$ is straight-forward. We use standard first-order semantics for the usual logical connectives and quantifications and focus on the semantics of the atomic expressions, i.e., if $\phi$ is a grounded Boolean fluent $p \in \mathcal{F}_{\mathcal{L}}$. In this case $s \models \phi^I$ iff $s(p) = true$.

To define the semantics of successor states, i.e., the function $app(s, o)$ we use PDDL semantics (Fox and Long, 2003). First, all simple effects in conjunctive effects are recursively collected into the set of effects $E$ to apply. Universal effects $\forall x : e$ therefore are also treated as conjunctive effects, where $x$ is substituted for all objects in $O_F$ resulting in one effect for each object in $O_F$. Next, if there is a negative and positive literal for the same Boolean fluent, the negative literal is removed. The resulting state $s' = app(s, o)$ for a Boolean fluent $p$ of applying $o$ in $s$ is then:

$$s'(p) = \begin{cases} true & \text{if } p \in E \\ false & \text{if } \neg p \in E \\ s(p) & \text{otherwise} \end{cases}$$

With the notion of applicability and successor states, we can now define the state space for a planning task.

**Definition 7.** *The **State Space for a Planning Task** is a directed graph $G = (S, E)$, where the vertices are all possible states in $S$ over $\mathcal{F}_{\mathcal{L}}$, and there is an edge from a state $s$ to $s'$, iff there exists a grounded operator $o$, so that $o$ is applicable in $s$, i.e., $applicable(o, s)$ and the result of applying $o$ in $s$ is $s'$, i.e., $s' = app(s, o)$.*

Based on the state space we formulate the planning problem for classical planning tasks as a search problem in the state space.

**Definition 8.** Plan *is a search problem in the state space for a planning task, where $s_0$ is the state defined by $I_L$. Find a finite path in the state space from $s_0$ to any state that fulfills $\phi_G$ or show that none exists.*

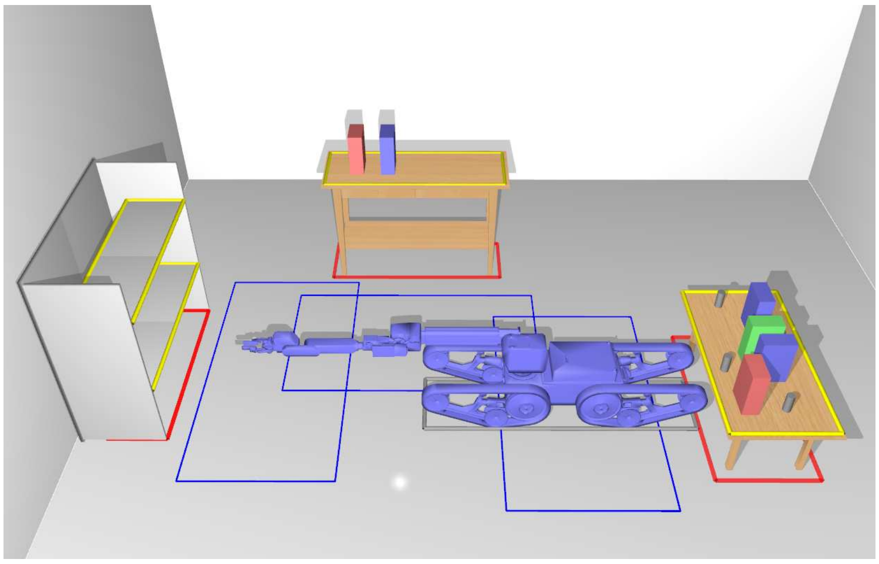


Figure 2.1: An example of a mobile manipulation scenario. The robot can pick up and put down objects with its manipulator arm and drive to tables and shelves to move objects from one place to another.

The solution of a planning task is a plan.

**Definition 9.** *A plan is the sequence of operators* $o_1, \dots, o_n$ *along a path, where* $applicable(o_i, s_{i-1})$ *and* $app(o_i, s_{i-1}) = s_i$ *for all* $i \in \{1, \dots, n\}$*. A plan must lead to a goal state, i.e.,* $s_n \models \phi_G$*.*

The cost of a plan is $\sum_{i=1}^{n} cost(o_i)$.

## 2.2 Robot Planning Tasks in PDDL

The goal of this section is to give a complete PDDL example for a robotic planning task. This shows how the definitions of the previous section are used in a real world example. Although the planner that we use supports temporal planning with concurrent actions we restrict the domain to non-concurrent actions as executing multiple robot skills like driving and grasping at the same time is not possible without affecting each other. This task will also serve as a running example for the next chapters. We model a mobile manipulation domain as shown in Figure 2.1. Mobile manipulation tasks require a robot to bring objects from one place to another, e.g., to clean up. Manipulation skills allow the robot to pick up and put down objects and a navigation component drives the robot base to different locations.

We start with formulating the planning domain. The first step is to give the predicate symbols with arities. This is an important part of any domain formalization as it defines what constitutes the state and thus describes our model of the world. The preamble of the domain states, which PDDL features we require. We use typed objects. This allows us to restrict the grounding to use only objects of a certain type for variables. Without this we would need to explicitly specify that, for example, there should not be a pick-up action for lifting a static object like a table.

```
(define (domain mobile-manipulation)
    (:requirements :strips :typing)

    (:types
        grasp
        static
        pose
        movable - pose
    )
```

There are four different types in this domain. A `grasp` specifies how an object is to be held, e.g., a side grasp. A `static` is an object that cannot be moved like a table or a shelf. A `pose` is used for anything that has a pose, while `movable` is a sub-type of pose and describes objects to be manipulated like a cup or a bottle.

Next, we will look at the predicate symbols from Definition 1 and how they form the predicates from Definition 2. They are given in the `:predicates` section. PDDL variables used therein are preceded by a question mark and can have an optional type. For example `?x - movable` states that the variable `x` only holds objects of type `movable`.

```
(:predicates
    (on ?x - movable ?y - static)
    (grasped ?x - movable ?g - grasp)
    (is-grasp-for ?g - grasp ?o - movable)
    (handempty)
    (at ?s - static)
)
```

The `on` predicate states on which static object a movable object is placed. An object can also be `grasped` in a certain grasp. Which grasps are possible for an object is given by `is-grasp-for`. If no object is grasped, `handempty` is true. `at` is true for a static object if the robot base is located near this object. The second part of a domain description contains the schematic operators (Definition 5). First, we look at the `pick-up` action.

```
(:action pick-up
    :parameters (?x - movable ?y - static ?g - grasp)
    :precondition (and
        (at ?y)
        (on ?x ?y)
        (handempty)
        (is-grasp-for ?g ?x)
        )
    :effect (and
        (not (on ?x ?y))
        (not (handempty))
        (grasped ?x ?g)
        )
)
```

The `pick-up` as well as the `put-down` operator both have three parameters: the object `?x` to pick or place, the static object `?y`, where the movable object is placed on, and the grasp to use. For the `pick-up` action to be applicable the robot needs to be `at` the static object, the object to pick needs to be `on` the static `?y` and the robot cannot already hold an object, i.e., `handempty` is true. The last condition `is-grasp-for` states that this grasp can be used for the object. The result of applying the `pick-up` action is that `?x` is not on `?y` any more, but `grasped`. Therefore `handempty` is now false.

```
(:action put-down
    :parameters (?x - movable ?y - static ?g - grasp)
    :precondition (and
        (at ?y)
        (grasped ?x ?g)
        (is-grasp-for ?g ?x)
        )
    :effect (and
        (not (grasped ?x ?g))
        (handempty)
        (on ?x ?y)
        )
)
```

The `put-down` action inverts the effects of the `pick-up` action. Thus now the object being `grasped` is the precondition, while the effect leads to the object being not `grasped` anymore and `handempty` is true.

```
(:action drive-base
    :parameters (?s - static ?g - static)
    :duration (= ?duration 5)
    :precondition (and
        (at ?s)
        (not (at ?g))
        )
    :effect (and
        (not (at ?s))
        (at ?g)
        )
)
```

Finally, the `drive-base` action moves the robot base from a start location `?s` to a goal location `?g`. To be able to do so, the robot must be `at ?s` and not already `at ?g`. As an effect of the action the robot is now `at ?g`, but not `at ?s` any more. The cost of this operator is explicitly given in the `duration` statement as 5. This concludes the domain for our mobile manipulation scenario. While the domain usually stays fixed during execution, the problem definition (Definition 3) changes given the current situation and goal. We give one example problem here with a single object in the scene of Figure 2.1.

```
(define (problem p)
    (:domain mobile-manipulation)
    (:objects cereal_box_6 - movable
        table2 table3 shelf1 shelf2 - static
        side_left side_right top - grasp)
```

First, the object set $O_F$ is given. Here we have one movable object `cereal_box_6` and four static ones: the two tables as well as the two stacked shelves to the left. Note that `movable` is a sub-type of `pose` and thus the `cereal_box_6` can also be used anywhere, where a pose is accepted. It might be more intuitive to think of movable objects also having the properties of a pose. Three different grasps are available. Next, the initial state is defined.

```
(:init
    (handempty)
    (is-grasp-for top cereal_box_6)
    (is-grasp-for side_left cereal_box_6)
    (is-grasp-for side_right cereal_box_6)
    (at table3)
    (on cereal_box_6 table2)
)
```

This represents $I_L$ and states that the robot is `at table3` and its hand is empty. The cereal box is on `table2` and all three grasps are possible. Although the `is-grasp-for` predicates are constant throughout the plan it is often still necessary to formulate a domain this way for robotics tasks and not compile this information out beforehand. The reason is that those might be determined during execution from sensor data and are then written to the problem. The planner therefore only uses the correct grasps. Finally the goal statement requires `cereal_box_6` to be in either of the shelves.

```
(:goal (or
        (on cereal_box_6 shelf1)
        (on cereal_box_6 shelf2)
    )
)
```

This problem together with the domain above states a classical planning task for a mobile manipulation scenario. With respect to robotics tasks there are some limitations. All conditions and effects are purely symbolic. Although there is symbolic information like a specific grasp, we cannot express if the arm can be moved into a collision-free configuration that grasps an object. It is also not clear, where an object is to be placed when it is `on` a table. The actual placement pose influences not only if it can be placed, but also any further actions. In addition this planning task assumes a deterministic execution and full knowledge of the scenario. A requirement unlikely to be fulfilled for real-world tasks. We will address all these shortcomings in the following chapters.

## 2.3 Forward-Chaining State Space Search

There are multiple different ways to solve the planning problem, i.e., finding a plan or proving that none exists. Many planners—including the one developed in this thesis—use a search-based approach. Therefore, we will shortly introduce the basic concepts of how such an approach works. The purpose of this section is not a complete overview of state of the art search techniques for planning, but to give an intuitive understanding of state space search that will illustrate how the techniques presented in the following chapters integrate with the search procedure. The underlying principle is that we use a forward-chaining—that is from the initial to a goal state—search in the state space graph given by Definition 7.

An important distinction between planning and other search algorithms is that this state space graph is too large to be explicitly built. It is only partially constructed during the search. However, we do need to represent a state and the grounded operators. In other words: We need a grounded representation. Although it is possible to create this online during planning this is often built as an explicit preprocessing step called *grounding*. There exist efficient algorithms to do so. We use the method

**Algorithm 1** Heuristic Best-First Search

```
 1: open ← $PriorityQueue(\emptyset)$
 2: closed ← $\emptyset$
 3: current ← $s_0$
 4: current_g ← 0
 5: best_g[current] ← 0
 6: while True do
 7:     if current $\notin$ closed or current_g < best_g[current] then
 8:         closed ← closed $\cup$ {current}
 9:         best_g[current] ← current_g
10:         if current $\models \phi_G$ then
11:             return SOLVED
12:         end if
13:         GENERATE_SUCCESSORS(open, current, current_g)
14:     end if
15:     next_ok, current, current_g ← FETCH_NEXT_STATE(open)
16:     if not next_ok then
17:         return NO_PLAN_FOUND
18:     end if
19: end while
```

introduced by Helmert (2009). The result of this is a more concise representation than the simple replacement of all variables with objects from $O_F$ as described in Section 2.1, where instantiations for unreachable fluents or operators are not performed. Any grounding procedure leads to a set of grounded fluents representing the state and a set of grounded operators, here called $\mathcal{P}$, that are used by the search.

Algorithm 1 shows the basic search algorithm. The priority queue stores pairs of the parent state and operator to apply, thus implicitly representing a successor of the parent state, as well as the g-value of the parent. The g-value is the cost estimate from the initial state to this state. In each step the current state—starting with the initial state—is added to the closed set (l. 8) and expanded by computing successors (l. 13). We discard states that have been closed unless their cost is lower than the closed state's cost (l. 7). GENERATE_SUCCESSORS is shown in Algorithm 2 and inserts all applicable operators into the open queue. The priority for the open queue is determined using heuristic estimates from a given heuristic guidance function. FETCH_NEXT_STATE also shown in Algorithm 2 takes the next state from the open queue. It computes this state explicitly by $app(state, op)$ and its g-value as well as a flag that states if a state could be taken. The search procedure continues until a goal state is found (l. 11) or no new state could be fetched as the open queue ran empty (l. 17).

**Algorithm 2** Successor Generation and Fetching

```
1: function GENERATE_SUCCESSORS(open, parent, parent_g)
2:     for all $o \in \mathcal{P}$ do
3:         if applicable($o$, parent) then
4:             open.insert((parent, $o$, parent_g),
5:                 computePriority(parent, $o$, parent_g))
6:         end if
7:     end for
8: end function
1: function FETCH_NEXT_STATE(open)
2:     if open.empty() then
3:         return False, None, $\infty$
4:     end if
5:     state, op, g $\leftarrow$ open.pop()
6:     return True, app(state, op), g + cost(op)
7: end function
```

# Chapter 3

# Semantic Attachments

Real-world planning problems often require several sub-problems to be solved. The high-level tasks a robot is supposed to perform, e.g., fetching a bottle of water from the kitchen, must be planned for. In a robotics context it is usually also necessary to plan for lower-level aspects like robot movements or the manipulation of objects. While the former problem can be expressed using a symbolic planning formalism as shown in Chapter 2, navigation and path planning is beyond the scope of symbolic planners. In fact, specialized planners that are better suited for these problems are available.

It makes, of course, a lot of sense to decompose a complex real-world planning problem into different simpler subtasks. However, the planners have to be combined in the right way. A commonly used method is a hierarchical *top-down* combination. First, at the highest level, the symbolic planner creates a symbolic plan. Then the actions are refined using low-level planners, e.g., the path planner and the manipulation planner. The assumption here is that the abstract symbolic description is expressive enough to permit a successful refinement and execution of the generated actions. However, very often this is not true. In such cases, the early commitment of the symbolic planner may lead to failures on the lower levels.

Instead of such a top-down approach, hierarchical composition can also be achieved in a bottom-up manner, where all information possibly relevant to the symbolic planner is precomputed by the lower level reasoners beforehand. This, however, may be very costly if there are too many facts, which often is the case. For example, the precomputation of all trajectories between all pairs of poses of a manipulator arm for all possible configurations of objects is extremely time consuming. Furthermore, most of the generated information will turn out to be irrelevant to the task at hand. In addition the state space in robotics problems might not be finite and thus a precomputation of all information is impossible.

Therefore, in this work, we propose an approach that integrates high and low-level planning more tightly and in which a low-level reasoner provides information to the high-level planner *during* the planning process, but is only evoked if relevant to the high-level planner. Contrary to the hierarchical decomposition and combination, a particular choice on the symbolic level can lead the low-level planner to detect failure and cause backtracking immediately.

To integrate information about special-purpose reasoning into symbolic planning we use what we call *semantic attachments*[1] within a planning domain description. Some of the predicate symbols of the domain description can have such a semantic attachment, meaning that the truth values for corresponding atomic ground formulas are specified by an *external mechanism*. We also include numerical fluents in the state. There exist semantic attachments for effects on numerical fluents and operator cost, which are computed by an external mechanism as well.

Semantic attachments can easily be added to a planning language like, in our case, PDDL. Based on that, we describe a general framework for integrating these extensions into forward-chaining state-space planners, which are particularly suited to our task since they search over complete world states. External modules can then access these states in order to compute conditions and effects for their special-purpose behaviors. While similar mechanisms have been used before, in particular in domain-specific contexts (Konolige and Nilsson, 1980; Bacchus and Kabanza, 2000; Orkin, 2006), our work (Dornhege et al., 2009a) appears to have been the first that extends PDDL rendering this feature available for domain-independent planners.

The rest of this chapter is structured as follows. In the next section, we give a number of motivating examples. Then we comment on related work and put other approaches in perspective to ours. Afterwards we define planning tasks with semantic attachments and how PDDL is extended to PDDL/M. Based on that, we describe our implementation of semantic attachments in the planning system *Temporal Fast Downward*. We evaluate the performance of semantic attachments and demonstrate the applicability of the concept for the examples that motivated this work. Finally, we conclude by summarizing what is possible with this formalism and give an outlook on how this system is extended and applied throughout the further chapters in this thesis.

## 3.1 Motivating Examples

For many real-world problems, it is hard to find a symbolic abstraction that guarantees that for every symbolic plan an executable plan can be refined. In this chapter, we consider two such problems, namely a logistics domain with complex truck packing problems and the robot manipulation domain introduced in Chapter 2.

### 3.1.1 Transport Domain

The *logistics* domain has been a standard benchmark for several years at the International Planning Competition. It models a common logistics problem, where trucks

[1] *Semantic attachment* is a term coined by Weyhrauch (1980) to describe the attachment of an interpretation to a predicate symbol using an external procedure.

deliver packages to different locations. In the original formulation, each truck can pick up only one package. With the introduction of numerical fluents, it became possible to model truck capacities and package sizes in the *transport-numeric* domain, allowing trucks to load multiple packages. This is represented by adding up package sizes and permitting to load additional packages as long as the sum stays below a truck's capacity.

Although more realistic than not representing capacities at all, summing up volumes is obviously not sufficient for checking whether a set of packages can be loaded into a truck, since the package geometries are not considered. For example, Figure 3.1 shows that it is impossible to pack two equally sized cubes into a cube with twice the volume. Moreover, it demonstrates that the volume approximation is not even close to reality.

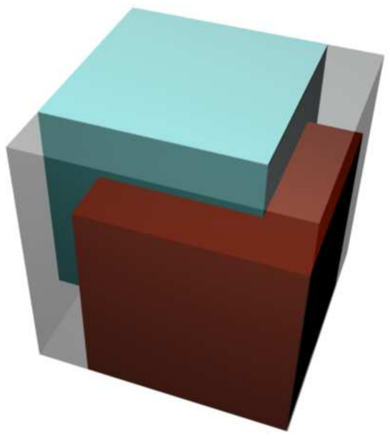


Figure 3.1: The two smaller cubes have half the volume of the outer one, but they obviously do not fit together into the outer cube.

Clearly, it is beyond the capabilities of a symbolic planner to solve the three-dimensional packing problem. However, there exist specialized algorithms for solving this NP-hard problem exactly or approximately. Such a reasoner could be integrated into the planner by attaching it semantically to the precondition of an action.

### 3.1.2 Robot Manipulation Domain

Similar to the *logistics* domain, the *blocks-world* domain has been a benchmark in the planning area for a long time. It can be considered as a highly abstract version of a robot manipulation problem. Nowadays large tasks are easily solved by symbolic planners. Unfortunately, however, the domain is so abstract that it has hardly anything to do with reality. For example, gripper poses or potential collisions of the gripper with other objects do not play a role at all.

A concrete domain is the symbolic robot manipulation domain introduced in Section 2.2. Figure 3.2 shows a simple example task for this domain. There are multiple objects with different shapes and sizes and a manipulator arm. A realistic model for moving an object to another place entails various problems. A chosen grasp must be reachable given the kinematic constraints of the manipulator. There must not be any collisions of the robot arm or the grasped object with any other object in the scene when grasping it, placing it and on a trajectory between grasping and placing. This can also mean that another object must be moved away first to avoid collisions or that the object needs to be re-grasped between picking it up and putting it down at a target location.

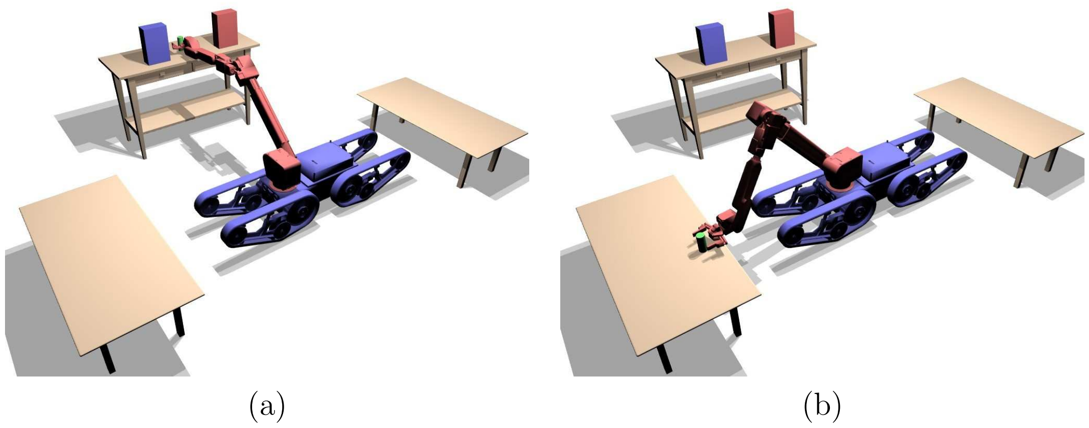


Figure 3.2: Visualization of the manipulation domain with (a) an initial state and (b) a goal state.

Again, solving such a task using only a symbolic planner is clearly impossible. We need a manipulation planner that computes collision-free trajectories for picking and placing objects as a sub-component of the symbolic planner. Such an embedded planner checks the precondition of whether a grasp or place action is possible. Furthermore, it also updates the internal model of the environment, i.e., the new object pose and arm configuration, so that possible collisions in subsequent states are detected.

## 3.2 Related Work

There exist some planning systems that exploit domain-specific knowledge, such as SHOP2 (Nau et al., 2003), TLPlan (Bacchus and Kabanza, 2000), or the TALplanner (Kvarnström and Doherty, 2000). However, in contrast to semantic attachments the main intention is to create a more efficient search process instead of providing a more precise domain semantics. This is achieved by "outsourcing" of hard problem-specific computations during planning, for example by isolating optimization sub-problems from planning problems (Fox and Long, 2001). The work by Srivastava and Kambhampati (1999) on decomposing a general planning problem into a resource and a planning problem is also relevant here. However, they specifically investigate the relation between resource and planning problems while we propose a general framework for combining different kinds of planning and reasoning.

In the past, semantic attachments have already been used in some domain-specific planning systems for computing specific action preconditions (Konolige and Nilsson, 1980; Orkin, 2006). The mechanism we propose is similar to a feature in TLPlan (Bacchus and Kabanza, 2000) called "computed predicates". These are implemented in the formula evaluator, which in this case is viewed as an interpreter of a language. Botea et al. (2003) use this planner to solve instances of an abstracted Sokoban domain, while the subproblems are solved externally. This domain-specific decomposition can thus be represented on the planner level.

The main differences to our approach (Dornhege et al., 2009a) are that most other approaches aim at domain-dependent search control, that the planning state cannot be queried via call-back functions, and that only the semantics of conditions can be externally computed, i.e., it is not possible to specify externally computed numerical effects. Another important aspect is that we developed a domain-independent interface with a suitable PDDL dialect that easily allows to apply our planner in various systems.

Motivated by SAT modulo theories more recent work proposes planning modulo theories as a domain-independent formulation of external semantics (Gregory et al., 2012). In contrast to our work, they do not provide external semantics for distinct aspects, but external semantics are given for a complete first-order theory. In practice this means that, e.g., function symbols that give theory-specific results can be computed externally. They give sets of objects and operations thereon as an example.

On the robotics side the integration of AI planning with robotics systems has also been considered. In the area of robotic planning, the work by Cambon et al. (2004, 2009) also aims at the integration of geometric and symbolic planning. They use special propositions that map to subspaces of the geometric configuration space in a symbolic planner. Symbolic plans are then used as a heuristic to guide a custom geometric search algorithm. Our manipulation planner build on semantic attachments (Dornhege et al., 2009b) follows a different concept. Instead of the symbolic planner being used in the manipulation planner, here the motion planner is a part of the symbolic planner.

Burbridge and Dearden (2013) do use a symbolic and a geometric planning solution, although both are not integrated. Their approach can be considered to be an iterated application of a top-down decomposition for a finite number of times. Other concepts for integrating task and motion planning rely on hierarchical task network (HTN) planners. Wolfe et al. (2010) use a HTN that on the lowest level of the hierarchy computes geometric solutions. A more generic formulation uses external computations in HTN to compute "shared predicates" that are shared in the sense that they have a meaning for the symbolic planner and for a "geometric task planner" (de Silva et al., 2013). An important aspect of their work is that they explicitly address backtracking in the symbolic and geometric domain. Another concept for integrating external reasoners is

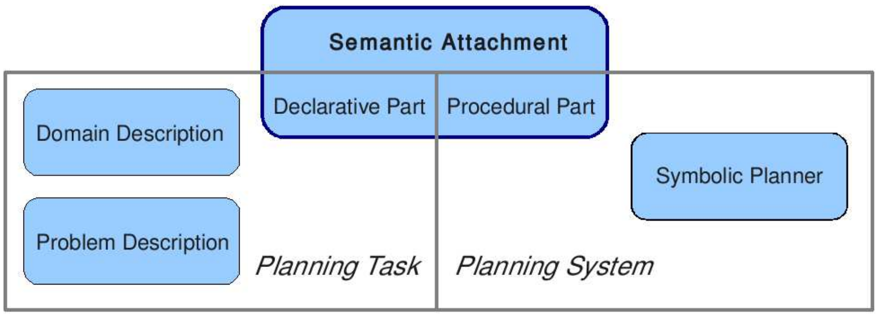


Figure 3.3: Semantic attachments consist of a *declarative* and a *procedural* part. The declarative part is contained within the task description and states, which variable valuations in a planning task are to be computed by the procedural part instead of a simple lookup.

presented by Kaelbling and Lozano-Pérez (2011, 2013). They search backwards from a goal state. Instantiations of some objects, such as intermediate placement locations, are provided by external "suggesters" that, for example, only produce locations so that a swept volume is kept free for subsequent operators in the plan.

## 3.3 Semantic Attachments

We address the shortcomings of symbolic planning formulations by a concept that we call *semantic attachments*. Semantic attachments are external reasoning modules (in the following just called *modules*) that compute valuations of variables used by a planner at run-time. Thereby they enhance the semantics of what is possible to express in a symbolic planner, while the domain-independent planner itself is mostly unaffected by this extension. Instead of looking up values or updating them through state transitions as usual in Strips-like languages, a function call provides the necessary information. Under the hood of the module, though, complex computations can be performed that transcend the capabilities of the planner.

In order to integrate semantic attachments into a planner we propose the architecture shown in Figure 3.3. Semantic attachments consist of a *declarative part* that describes their use in the planning domain, i.e., their symbolic use in preconditions, costs and effects of planning operators. Additionally, they have a *procedural part* that is attached to the declarative part and is the actual algorithm for computing the value in question. This part is implemented as a function and directly included into the planner as a shared library. The current planner state can be accessed through callback functions. The next section gives more details about the implementation.

We propose three kinds of semantic attachments that can be part of operators: *Condition checkers* test whether some complex operator precondition given by a predicate is satisfied. *Effect applicators* compute changes to any number of numerical state variables. *Cost modules* compute the cost of an operator. When speaking of the declarative part of these modules, i.e., their use as preconditions and effects of planning operators, we will speak of *module predicates* in the case of condition checkers and *effect modules* in the case of effect applicators.

### 3.3.1 Definitions

In order to actually use semantic attachments in classical planning, it is necessary to include semantic attachments in the definitions of classical planning tasks given in Section 2.1. Additionally the description language for planning tasks PDDL is slightly extended to *PDDL/M*—an acronym for PDDL with modules. PDDL planning tasks are described with different levels that optionally enable additional expressiveness. Besides the usual first-order logic based symbolic formulation we additionally include numerical state variables. Although our planner TFD/M supports conditions and effects on numerical state variables, we skip these definitions as numerical PDDL conditions and effects are not sufficient to express robotics tasks. In PDDL/M numerical fluents are an essential part of the planner state as the external modules are most relevant when complex *numeric* computations need to be performed. As such besides the symbolic values they take numerical values as inputs and also provide resulting numerical effects. We will now restate the definitions from Section 2.1 extended with numerical fluents and semantic attachments, but focus the descriptions on the differences and additions for planning tasks with semantic attachments. We also give PDDL/M examples for defining modules based on the PDDL task from Section 2.2. First, we define the planning domain.

**Definition 10.** *A planning domain is a tuple $(P, MP, F, EM, \mathcal{O})$, where $P$ is a finite set of predicate symbols, $MP$ is a finite set of module predicate symbols, $F$ is a finite set of function symbols, $EM$ is a finite set of effect module symbols—all pairwise disjoint and with respective arities. $\mathcal{O}$ is a finite set of schematic operators.*

The additions here are the introduction of *function symbols* that represent numerical values, *module predicate symbols* that represent *condition checkers* and *effect module symbols* that represent *effect applicators*. Together with the *classical terms*, i.e., constant and variable symbols, we can now form *module predicates* and *effect modules*.

**Definition 11.** *The atomic formulae are formed from an $n$-ary predicate symbol from $P$ with $n$ classical terms or from an $n$-ary module predicate symbol from $MP$ with $n$ classical terms. In addition to atomic formulae we define* Primitive Numerical Expressions *(PNE). A PNE is formed from an $n$-ary function symbol from $F$ and $n$ classical terms. Finally there are* effect modules. *An effect module consists of an $n$-ary effect module symbol from $EM$, $n$ classical terms and a $k$-dimensional vector of PNEs that we call the affected fluents.*

In PDDL primitive numerical expressions are declared in the `:functions` section exactly like PDDL predicates are declared in the `:predicates` section.

```
(:functions
    (q0) (q1) (q2) (q3) (q4) (q5) (q6)

    (robot-x)
    (robot-y)
    (robot-th)

    (x ?p - pose)
    (y ?p - pose)
    (z ?p - pose)
    (qx ?p - pose)
    (qy ?p - pose)
    (qz ?p - pose)
    (qw ?p - pose)
)
```

There are three blocks of numerical state variables. `q0`–`q6` are the current configuration parameters of the—in this case— 7-dof manipulator. `robot-x`, `robot-y` and `robot-th` give the pose of the robot base in the 2d plane. Finally, there is a generic pose for objects, where `x`, `y`, `z` describe the position and `qx`, `qy`, `qz`, `qw` are parameters of the orientation quaternion.

A PDDL/M domain contains an additional `:modules` section that declares the three types of modules similar to the way predicates and PNEs are declared in PDDL. In this section, each semantic attachment has its own entry, a *condition checker* and *cost module* consisting of four, and an *effect applicator* of five mandatory parts. All start with a unique identifier. In the case of condition checkers and effect applicators this is the respective symbol; for cost modules this is a name to reference the module. Next follows a possibly empty list of parameters, similar to a function or predicate entry in their respective sections. Only for effect applicators we then list the $k$ PNEs that are set by the module. Then the type of module is declared by a keyword and finally the function and library name defining the procedural part is given. This is illustrated by the following modules section for the exemplary mobile manipulation domain.

```
(:modules
    ...
    (checkTransfer ?target - movable ?place - static ?grasp - grasp
        conditionchecker
        checkTransfer@libtrajectoryModule.so)
    (applyTransfer ?target - movable ?place - static ?grasp - grasp
        (q0) (q1) (q2) (q3) (q4) (q5) (q6)
        (x ?target) (y ?target) (z ?target)
        (qx ?target) (qy ?target) (qz ?target) (qw ?target)
        effect applyTransfer@libtrajectoryModule.so)
    (costDrive ?s - static ?g - static cost
        costDrive@libtrajectoryModule.so)
)
```

The first two modules are used in the `put-down` action. They represent a transfer motion, i.e., a manipulator movement, where a grasped object is transfered to a placement location. First, the symbol names of the module predicate and effect applicator are given as `checkTransfer` and `applyTransfer`. The following three parameters match those used in the `put-down` operator. The final statements of the form `checkTransfer@libtrajectoryModule.so` state that this module is implemented by the function `checkTransfer` in the dynamic library with the name `libtrajectory Module.so`. When and how such functions are applied will be explained later in Definitions 15–17 and Section 3.4. The condition checker `checkTransfer` here computes *if* a collision-free motion plan exists to place the object `?target` at `?place`, while the effect applicator `applyTransfer` computes the pose, where the object will be placed afterwards and the resulting configuration of the manipulator arm. The configuration parameters of the manipulator are stored in the state's numerical fluents `q0`–`q6` and the 6-dof object pose is given by the fluents `x, y, z` and `qx, qy, qz, qw` for the `?target` object. The cost module is applied in the `drive-base` operator. It computes the path cost to drive the robot base from the location `?s` to `?g`.

Before addressing planning operators in Definition 14 we define a specific planning problem for a planning domain and the resulting state space.

**Definition 12.** *A planning problem is a tuple $(I_L, I_N, \phi_G, O_F)$, where $I_L$ and $I_N$ are the logical and numerical initial state, $\phi_G$ is a first-order formula describing the goal and $O_F$ is a finite set of objects.*

The planning problem defines the initial state, a goal formula and a finite object set $O_F$ for grounding as in Definition 12, where the initial state is now specified for Boolean and numerical fluents. As before the state space of a planning task is defined by the grounded representation. The object set is again used to ground the predicates, but PNEs are also grounded. The grounded predicates form the set of Boolean fluents $\mathcal{F}_{\mathcal{L}}$ and the grounded PNEs form the set of numerical fluents $\mathcal{F}_{\mathcal{N}}$. Both together define a state, which is a function $s$ that maps each Boolean fluent to $\{true, false\}$ and each numerical fluent to a value in $\mathbb{R}$. Note that module predicates—although being predicates in first-order formulae—are *not* part of the Boolean fluents and therefore not contained in a state. We now define a state for a problem instance.

**Definition 13.** *A state* s *is a function that maps each Boolean fluent from $\mathcal{F}_\mathcal{L}$ to $\{true, false\}$ and each numerical fluent from $\mathcal{F}_\mathcal{N}$ to $\mathbb{R}$, i.e.:*

$$s : \mathcal{F}_\mathcal{L} \to \{true, false\}$$

*and*

$$s : \mathcal{F}_\mathcal{N} \to \mathbb{R}$$

*The set of all possible assignments for $\mathcal{F}_\mathcal{L}$ and $\mathcal{F}_\mathcal{N}$ is the set of states $S$.*

Given that we include numerical fluents in the state there are infinitely many states. However, $\mathcal{F}_\mathcal{L}$ and $\mathcal{F}_\mathcal{N}$ are still finite as they are grounded from a finite set $O_F$, so that in turn the descriptor for a single state is finite and—for a given planning task—of constant size. $I_L$ and $I_N$ are the variable assignment that represents the initial state. An initial state in PDDL now also contains initial values for numerical fluents, e.g., for our problem from Chapter 2:

```
(:init
    (handempty)
    (is-grasp-for top cereal_box_6)
    (is-grasp-for side_left cereal_box_6)
    (is-grasp-for side_right cereal_box_6)
    (at table3)
    (on cereal_box_6 table2)

    (= (robot-x) 0)
    (= (robot-y) 0)
    (= (robot-th) 0)

    (= (q0) 0)
    (= (q1) 1.570796327)
    (= (q2) 0)
    (= (q3) 0)
    (= (q4) 0)
    (= (q5) 0)
    (= (q6) 62)

    (= (x cereal_box_6) -0.8)
    (= (y cereal_box_6) 1.4)
    (= (z cereal_box_6) 0.298)
    (= (qx cereal_box_6) 0)
    (= (qy cereal_box_6) 0)
    (= (qz cereal_box_6) 0.7071067812)
    (= (qw cereal_box_6) 0.7071067812)
)
```

The second half of the `:init` statement after the first six entries initializes all numerical fluents to the correct values representing the scene in Figure 2.1. The definition of an operator is slightly adapted to include semantic attachments.

**Definition 14.** *A schematic operator is a tuple $(\phi, e, cost)$, where $\phi$ is a first-order formula—the precondition, $e$ is an effect, and cost a function that maps from a state $s$ to $\mathbb{R}_{>0}$—the operator cost. First-order formulae are formed over the atomic expressions from $P$ and $MP$ with the usual logical connectives. An effect is defined by the finite application of the following rules:*

- *$e$ is called a simple effect if $e$ is a predicate literal (i.e., a predicate or negated predicate). Note that this excludes module predicate literals.*
- *$e$ is called a module effect if $e$ is an effect module $ea$.*
- *$e_1 \wedge \ldots \wedge e_n$ is a conjunctive effect for effects $e_1 \ldots e_n$.*
- *$\forall x : e$ is a universal effect for a variable symbol $x \in \mathcal{V}_{\mathcal{F}}$ and an effect $e$.*

*We call the set of free variables of an operator the **parameters** of an operator. The free variables of an operator $(\phi, e, cost)$ are the free variables of its precondition and its effect, i.e., $free(\phi) \cup free(e)$, where the $free(e)$ is defined as:*

$$free(e) = \begin{cases} free(\psi) & \text{if } e = \psi \text{ or } e = \neg\psi \text{ is a simple effect} \\ free(ea) & \text{if } e = ea \text{ is an effect module} \\ \bigcup_{i=1}^{n} free(e_i) & \text{if } e = e_1 \wedge \ldots \wedge e_n \text{ is a conjunctive effect} \\ free(e') \setminus \{x\} & \text{if } e = \forall x : e' \text{ is a universal effect} \end{cases}$$

*$free(ea)$ for an effect module are all variables used in the terms of $ea$ and all variables used in terms of any of the affected fluents of $ea$. A **grounded operator** is an operator that has no parameters.*

There are three differences to the classical planning operators given in Definition 5. The operator cost is now a function. This function might be constant as in classical planning tasks. In this case we use the same PDDL statements as before. Otherwise the cost function is given by a cost module (see Definition 17). We added module predicates to the first-order formulae and effect modules as a module effect to an operator's effect. As a consequence of this when grounding an operator we also ground the module predicates and effect modules. For effect modules not only variables in the $n$ terms of the module are grounded, but also all variables within the affected fluents are grounded as well. In addition to $\mathcal{F}_{\mathcal{L}}$ and $\mathcal{F}_{\mathcal{N}}$ this leads to grounded module predicates and grounded effect modules. Within PDDL domain descriptions we signal the use of a module predicate or effect applicator by square brackets. The `put-down` action for our mobile manipulation task with semantic attachments is an example of this notation. Here the operator cost is constant. The module predicate `checkTransfer` and the effect module `applyTransfer` matching the module definitions above are used.

```
(:action put-down
    :parameters (?x - movable ?y - static ?g - grasp)
    :duration (= ?duration 5.0)
    :precondition (and
        (at ?y)
        (grasped ?x ?g)
        (is-grasp-for ?g ?x)
        ([checkTransfer ?x ?y ?g])
        )
    :effect (and
        (not (grasped ?x ?g))
        (handempty)
        (on ?x ?y)
        ([applyTransfer ?x ?y ?g])
        )
)
```

The semantics of applicability and operator application have to be adapted to account for semantic attachments. This is the crucial part, where semantic attachments come into play. If a grounded operator $o$ can be applied in a state $s$—or *applicable*$(o, s)$— is given by the interpretation $I$ of the precondition $\phi$ in a state $s$. We use standard first-order semantics for the usual logical connectives and quantifications and focus on the semantics of the atomic expressions. The difference to first-order formulae in classical planning is that we added module predicates. If $\phi$ is a grounded Boolean fluent $p \in \mathcal{F}_\mathcal{L}$, then $s \models \phi^I$ iff $s(p) = \mathit{true}$. However, if $\phi$ is a grounded module predicate the semantics are defined by external functions—the procedural part of the condition checker.

**Definition 15.** *A* ***condition checker*** *is a function $f_{cc}$ attached to an n-ary module predicate symbol, where $f_{cc} : O_F^n \to (S \to \{\mathit{true}, \mathit{false}\})$ maps from an n-dimensional vector of objects to a function that maps from a state to true or false.*

First, note that *condition checkers* are attached to module predicate symbols, and not to module predicates. Thus the definition is generic applied to a domain independent of an actual problem instance. For a grounded operator each $n$-ary module predicate was grounded with $n$ objects from $O_F$ that we collect in a vector $\mathbf{x}$. Applying the condition checker to $\mathbf{x}$ results in a function $g = f_{cc}(\mathbf{x})$ for the grounded module predicate that maps $S \to \{\mathit{true}, \mathit{false}\}$. This is the function defining the semantics of the grounded module predicate in a state from $S$. Therefore, if $\phi$ is a grounded module predicate $s \models \phi^I$ iff $g(s) = \mathit{true}$. This means that condition checkers are evaluated on the current state, and thus not part of the set of logical fluents $\mathcal{F}_\mathcal{L}$. As such module predicates must not appear as an effect in any operator. In contrast to derived predicates in PDDL (Fox and Long, 2003) condition checkers are computed from the logical *and* numerical state and thus are able to model complex relations.

The semantics of successor states, i.e., applying $o$ in $s$ given by the function $app(s, o)$ is defined by PDDL semantics for the parts of an effect that are defined in PDDL, that is: universal effects, conjunctive effects and simple effects (see Fox and Long (2003)). The actual state update is performed by simple effects and module effects. Module effects affect the numerical fluents of the state in $\mathcal{F}_\mathcal{N}$ and thus are independent from simple effects that change the Boolean fluents in $\mathcal{F}_\mathcal{L}$. The semantics of module effects are defined by *effect applicators* that are attached to effect module symbols. An example is shown in the `put-down` operator above.

**Definition 16.** *An* ***effect applicator*** *is a function $f_{ea}$ attached to an $n$-ary effect module symbol, where $f_{ea} : O_F^n \to \left(S \to \mathbb{R}^k\right)$ maps from an $n$-dimensional vector of objects to a function that maps from a state to a $k$-dimensional numerical vector.*

Recall that effect modules are formed from $n$ terms and a $k$-dimensional vector of PNEs. Similar to module predicates, for a grounded operator the $n$-ary effect module was grounded with a vector $\mathbf{x}$ of $n$ objects. To apply a module effect, first we call $h = f_{ea}(\mathbf{x})$ to get an effect applicator function for a grounded effect module. When applying a module effect with function $h$ in a state $s$, we call $\mathbf{y} = h(s)$, where $\mathbf{y}$ is a $k$-dimensional vector in $\mathbb{R}^k$. Let $\mathbf{a}$ be the $k$-dimensional vector of affected fluents from $\mathcal{F}_\mathcal{N}$ for this effect module. The resulting state $s'$ for these numerical fluents in $\mathbf{a}$ is given by:

$$s'(\mathbf{a}[i]) = \mathbf{y}[i] \text{ for } i \in \{1, \dots, k\}$$

Numerical fluents $f$ that were not affected by any effect module in an operator stay unchanged, i.e., $s'(f) = s(f)$. The state update is ambiguous if multiple module effects affect the same PNE. As there is no reasonable ordering between different PNE values as, e.g., delete-first for Boolean effects, we disallow assigning to the same affected fluent within the same operator.

**Example** Consider the grounded operator `put-down bottle1 table2 top` that places a bottle on a table. The object vector $\mathbf{x}$ for both modules here is (`bottle1, table2, top`)$^T$. Grounding for this operator lead to a grounded effect module

```
(applyTransfer bottle1 table2 top
    (q0) (q1) (q2) (q3) (q4) (q5) (q6)
    (x bottle1) (y bottle1) (z bottle1)
    (qx bottle1) (qy bottle1) (qz bottle1) (qw bottle1)
    effect applyTransfer@libtrajectoryModule.so)
```

Here $k = 14$ and the affected fluents $\mathbf{a}$ are (`(q0), (q1), (q2), (q3), (q4), (q5), (q6), (x bottle1), (y bottle1), (z bottle1), (qx bottle1), (qy bottle1), (qz bottle1), (qw bottle1)`)$^T$.

A state update for applying this effect applicator in a state $s$ is computed as follows:

- First, call $h = f_{ea}((\texttt{bottle1, table2, top})^T)$.
- Next, call $\mathbf{y} = h(s)$, which must result in a 14-dimensional vector $\mathbf{y}$.

Let us assume the result of that computation is to place the bottle at the position $(0.4, -0.2, 0.75)^T$. For brevity we will not consider all 14 values here although in practice each one must be set.

- The respective entries are: $\mathbf{y}[7] = 0.4$, $\mathbf{y}[8] = -0.2$, $\mathbf{y}[9] = 0.75$.
- From $\mathbf{a}$ we have $\mathbf{a}[7] =$ `(x bottle1)`, $\mathbf{a}[8] =$ `(y bottle1)`, $\mathbf{a}[9] =$ `(z bottle1)`.
- Therefore the resulting state $s'$ for those fluents is $s'($`(x bottle1)`$) = 0.4$, $s'($`(y bottle1)`$) = -0.2$, $s'($`(z bottle1)`$) = 0.75$.

The third type of module is a *cost module*. As the name suggests it is used to define an operator's cost by an external function.

**Definition 17.** *A* ***cost module*** *is a function $f_{cost}$ attached to a schematic operator $o$ with $n$ parameters, where $f_{cost} : O_F^n \to (S \to \mathbb{R}_{>0})$ maps from an $n$-dimensional vector of objects to a function that maps from a state to $\mathbb{R}_{>0}$.*

A cost module is used to provide the cost function of an operator from Definition 14. Similar to *condition checkers* and *effect applicators* this function takes an object vector to allow domain writers to implement generic functions for any problem instance. In contrast to the two other module types the function has the same parameters as the operator. The function $f_{cost}$ is applied in a similar way when grounding a schematic operator $o$ with a vector of objects $\mathbf{x}$. We get a new function $cost = f_{cost}(\mathbf{x})$. This function $cost : S \to \mathbb{R}_{>0}$ is the cost function of the grounded operator and determines the operator's cost as $cost(s)$ when applied in a state $s$. As an example consider the `drive-base` operator.

```
(:action drive-base
    :parameters (?s - static ?g - static)
    :duration (= ?duration [costDrive ?s ?g])
    :precondition (and
        (at ?s)
        (not (at ?g))
        ([checkDrive ?s ?g]))
    :effect (and
        (not (at ?s))
        (at ?g)
        ([applyDrive ?s ?g]))
)
```

The cost of this operator is defined by the *cost module* `costDrive` that, for example, calls a path planner to compute the length of the path from the pose of `?s` to the pose of `?g`. Likewise, `checkDrive` will compute if such a path exists and `applyDrive` will set the resulting pose at `?g`. To prevent code duplication these modules are therefore usually implemented in the same dynamic library.

Semantic attachments only affect the notion of applicability, successor derivation and cost of individual operators. Therefore the state space for a planning task with semantic attachments, the resulting planning problem, and the notion of a plan (Definitions 7–9) are similar to Chapter 2. We will quickly restate how the state space for a planning task looks like as that is relevant for the discussion on soundness and completeness.

**Definition 18.** *The **State Space for a Planning Task** is a directed graph $G = (S, E)$, where the vertices are all possible states in $S$ over $\mathcal{F}_\mathcal{L}$ and $\mathcal{F}_\mathcal{N}$, and there is an edge from a state $s$ to $s'$, iff there exists a grounded operator $o$, so that $o$ is applicable in $s$, i.e., $applicable(o, s)$ and the result of applying $o$ in $s$ is $s'$, i.e., $s' = app(s, o)$.*

The most noteworthy difference to classical planning tasks is that the state space is now defined over Boolean *and* numerical fluents and that effect applicators set or change numerical fluents. As these take values in $\mathbb{R}$ the state space is no longer finite. The planning problem is the search problem in this graph to either find a path from the initial state $s_0$ to any goal state—the plan—or show that no such path exists.

**Definition 19.** Plan *is a search problem in the state space for a planning task, where $s_0$ is the state defined by $I_L$ and $I_N$. Find a finite path in the state space from $s_0$ to any state that fulfills $\phi_G$ or show that none exists.*

## 3.3.2 Soundness and completeness

One important question when using semantic attachments is how these affect soundness and completeness of a planner. Since arbitrary code can be used in the modules, it cannot be guaranteed that the planner terminates when calling a semantic attachment. However, under some reasonable assumptions soundness and completeness can still be investigated. All semantic attachments are modeled as functions that given some objects return functions that in turn map from a state to a module specific result. We require all these functions to always terminate and return a deterministic value. This means for identical parameter values and states the result of any semantic attachment must be identical.

In other words, condition checkers are nothing else than a concise representation of derived predicates. Similarly we can view effect applicators as a concise representation of part of the deterministic transition function that for classical planning tasks is specified by the PDDL effects. Effect applicators in particular should not contain any mechanism for making *choices* between different outcomes, such as selecting a location for placing an object, when called repeatedly with the same inputs. Cost modules are understood as providing another means of specifying the cost of an operator in a state.

If these conditions are met, one can analyze soundness and completeness of a planner extended by a semantic attachment mechanism *relative* to the semantic attachments: Assuming that all semantic attachments implement the intended meaning are the returned plans correct and will the planner find a solution if there exists one? Before we argue later, what guarantees our implementation of semantic attachments provides, we now consider what is possible theoretically.

First, we define what it means that an effect applicator *changes* numerical fluents or the opposite only *sets* numerical fluents.

**Definition 20.** *We say an effect applicator* ***changes*** *numerical fluents if the result of a module effect depends on* any *numerical fluent in $\mathcal{F}_\mathcal{L}$. If this is not the case, i.e., the result only depends on symbolic information, we say that an effect applicator* ***sets*** *numerical fluents.*

One important factor is if the *reachable* state space is finite. Changing numerical fluents causes a possibly infinite state space for a planning task. This leads to the following result.

**Theorem 21.** Plan *is undecidable when effect applicators* change *numerical fluents.*

*Proof.* Planning with numerical state variables is undecidable (Helmert, 2002). Helmert reduces the planning problem to finding solutions for Diophantine equations, which are polynomial equations with multiple variables, in the natural numbers. Planning tasks with semantic attachments also model Diophantine equations as they are more expressive than PDDL's numerical expressions and effects. We create one grounded operator per variable of the equation that just increases this variable by one with an *effect applicator*. This enumerates the natural numbers $\mathbb{N}$. A *condition checker* computes if the equation holds and only then allows the transition to a goal state. $\square$

When restricting effect applicators to only set numerical fluents, we retain a finite reachable state space. Let $\mathcal{P}$ be the finite set of grounded operators. Each grounded predicate in $\mathcal{F}_\mathcal{L}$ can be true or false, so that the number of distinct symbolic states is bounded by $2^{|\mathcal{F}_\mathcal{L}|}$. Each operator in $\mathcal{P}$ might lead to a new assignment of numerical fluents. However, the number of distinct assignments to numerical fluents is still finite and bounded by $2^{|\mathcal{F}_\mathcal{L}|} \cdot |\mathcal{P}|$—one assignment per operator per state. In the worst case, we thus get a large, but finite state space. Such a state space can be searched with a forward-search procedure like breadth-first-search that decides Plan in this case. Given that the number of operators is finite applying breadth-first-search to the generic case when effect applicators *change* numerical fluents leads to a procedure that enumerates the state space and therefore is semi-decidable. Restricting effect applicators to only set numerical fluents does have practical applications. The case of placing an object is formulated to always result in the same geometric placement.

A `push` action, for example, would violate this restriction as multiple pushes lead to different placements.

Although these results might seem unsatisfactory at first, this does not prohibit such planning tasks from being applied to real-world systems. As long as we have a search procedure that only produces sound plans and does not prune away any plans, computation time is the limiting factor. Considering that complex geometric operations might be performed in each search step even fully exploring a finite state space is infeasible. For real-world applications it is therefore important to find existing plans quickly. A negative result due to a timeout also often occurs even for finite state spaces. We will address these properties for our implementation in the next section.

## 3.4 Implementation

A PDDL/M planner must evaluate semantic attachments at runtime, i.e., it must call the external modules and use the computed results during the planning process. A sound implementation must consider the following points.

- How do we acquire the actual function to be called from the domain-independent formulation of a module, i.e., how are module calls grounded?
- How is the current planner state passed to module calls?
- When are modules to be called and how does the module specific return value influence the planning process?
- How do we integrate actual external code into the planning process?

This section aims to show how these can be addressed for forward-chaining state space search planners. We present the extension of the planner Temporal Fast Downward (TFD) to TFD/M—Temporal Fast Downward with Modules. First we shortly introduce the concepts TFD is built upon and then we will show how the aspects listed above are implemented in TFD/M.

Temporal Fast Downward (Eyerich et al., 2009) is a domain-independent state space search planner built on top of the classical planning system *Fast Downward* (Helmert, 2006). It extends the original system supporting durative actions as well as numerical and object fluents. TFD searches in so called *time stamped states* to model concurrent aspects from durative actions. We do not consider concurrent actions for robotics tasks and therefore prohibit any concurrent application of operators in the search. This results in a standard progression search in the state space without time stamps. TFD was chosen because of its native support for numerical fluents, which are essential to geometric computations. One distinguishing feature is that the input consisting of propositional atoms is automatically translated into an encoding using *multi-valued* variables.

(Temporal) Fast Downward solves a planning task in three phases: As a first step, the PDDL planning task is *translated* from its Strips-like encoding into a representation similar to SAS+ (Bäckström and Nebel, 1995), using finite-domain variables instead of binary predicates. Afterwards, in a *knowledge compilation* step, data structures used by the search heuristic are generated. Finally, a best-first state space *search*, guided by a numeric temporal variant of the context-enhanced additive heuristic (Helmert and Geffner, 2008), is performed. Since the internal representation of TFD is significantly different from PDDL, enabling both the planner and the modules to access and manipulate the planning state is not trivial. The most significant extensions to TFD occur in the translation and search phases, which we describe in the following.

**Translation** In the translation phase of Temporal Fast Downward, the task is converted into a finite-domain representation (FDR). In order to generate an appropriate FDR description from PDDL/M tasks, we adapt the method of Helmert (2009). Roughly, this process consists of the following phases: First, a normalization that leaves all conditions as conjunctions of literals is executed. During this phase a finite number of additional operators and axioms, which for this discussion we can treat as zero-cost operators, might be introduced. Note that this means that conditions within the search procedure are thus conjunctions of literals, which is quite important on multiple points throughout this thesis. Next, an invariant synthesis generates mutual exclusion (mutex) invariants that describe, which propositions may never be true at the same time. These propositions can be combined into a single FDR variable. The following grounding phase, by means of a relaxed reachability analysis, results in the sets of grounded predicates and grounded numerical fluents as well as the grounded operators. Finally the FDR task generation computes from the mutex invariants a translation table that matches each grounded predicate to an FDR variable and the value from the variable's domain, where this grounded predicate is true. This translation table is then used to translate all conditions and effects to their respective FDR representations.

Since module predicates are exactly the same as normal predicates regarding their treatment in first-order formulae, besides different semantics in their interpretation, the normalization phase is virtually unchanged. We only note, which predicates are module predicates. During the invariant generation we cannot make assumptions about mutex groups for module predicates. Therefore for each grounded module predicate we introduce a distinct FDR variable with two valuations: one for true and one for false.

The instantiations of grounded predicates, numerical fluents and operators during the grounding phase are extended to module calls. Grounded module predicates are instantiated exactly like grounded predicates. When an operator is grounded the operator's parameter instantiations are also used to instantiate grounded cost modules and grounded module effects if a *cost module* or *effect applicators* appear in the operator.

```
typedef double (*conditionCheckerType)(
    const ParameterList & parameterList,
    predicateCallbackType predicateCallback,
    numericalFluentCallbackType numericalFluentCallback,
    int relaxed);
typedef int (*applyEffectType)(
    const ParameterList & parameterList,
    predicateCallbackType predicateCallback,
    numericalFluentCallbackType numericalFluentCallback,
    int relaxed, vector<double> & writtenVars);
```

Figure 3.4: Main part of the PDDL/M C++-Interface. Two types of module call signatures are used: conditionCheckerType and effectApplicatorType. Cost modules share the same function signature with condition checkers.

Here, it is important to also ground the affected numerical fluents as they might not appear anywhere else in the planning task and otherwise might be compiled away.

This is also the point, where we make the transition from a generic module call to a problem specific instantiation. Recall that the first step to applying any semantic attachment is to produce a specific function for the *grounded* module by calling $h_{type} = f_{type}(\mathbf{x})$ for any module type. Due to the grounding we now have the vector of objects $\mathbf{x}$. For each grounded module call of any type we instantiate an individual reference to the respective module including the grounded parameters. This represents the instantiated calling function for the module call. For *cost modules* we add this reference to the grounded operator. For *condition checkers* the module call reference is attached to the grounded module predicate. For *effect applicators* a reference to the module call is added to the grounded operator's effect list as the module effect.

When the FDR task is generated each grounded module predicate is assigned to an FDR variable that represents its truth value. Here we add a direct link between that FDR variable and the grounded module call reference. When module effects are generated, the variable references from the translation table are also applied to the affected fluents, so that they can be referred to in the translated task. Finally, we need to address the fact that semantic attachments reason about PDDL, i.e., they are unaware of any underlying representation such as the FDR. Therefore we also write out the translation table as part of the generated FDR task. This is used to map the current planner state stored as an FDR to PDDL.

**Module Interface** Technically, modules are implemented as dynamically loaded shared libraries. This allows arbitrary modules to be included at run time, i.e., independent of the planner, so that the domain specification and the planner are kept separate. However, it is necessary to have a common interface. As the search component of TFD/M is written in C++, the interface shown in Figure 3.4 is also specified in C++. Similar interfaces can be designed for other programming languages. It is also possible to relay calls to external components by inter process communication using robot middleware such as ROS (Quigley et al., 2009).

*Cost modules* and *condition checkers* use the `conditionCheckerType` and *effect applicators* use the `applyEffectType`. All kinds of semantic attachment share the same input data. First, for a grounded module call, the vector of objects $\mathbf{x}$ is passed in as `parameterList`. Each module call is also passed the state via a callback interface (see below). Cost modules and condition checkers return a floating point value that is either the actual cost, or for condition checkers results in a Boolean decision that is interpreted as true iff the floating point value is below a predefined threshold `INFINITE_COST`. Effect applicators are passed a reference to a list of numeric values—the vector $\mathbf{y}$—that is to be filled with the new values for the affected fluents.

Additionally, all module calls can be invoked with a *relaxed* flag requesting a relaxed computation that can be significantly faster. This invocation might be used anywhere within the planner, most notably within heuristic computations. This is an advantage that implementers can provide for efficiency, but not a limitation to applicability. If no relaxation that makes sense is available, the full module computation is a trivially sound, although not faster, relaxed implementation. In other words: It is safe to ignore this flag.

**State Callback** The purpose of the state callback interface is to provide the current planner state to the semantic attachments. The interface has to bridge the internal representation of a state as a FDR with the module's interface in PDDL. There are two callback functions shown in Figure 3.5: One for grounded predicates and one for grounded numerical fluents. A module that needs to acquire the current state's valuation of relevant predicates or numerical fluents passes a list of these that contains the symbol name, the grounded parameters, and undefined valuations to the callback. The planner uses the translation table computed during the PDDL to FDR translation to look up, which variable in the planner state in FDR corresponds to the PDDL name in the list. For numerical fluents this variable contains the valuation and it is set correspondingly. For Boolean fluents, i.e., grounded predicates, the lookup table also contains a value. If the state variable has this value, the fluent is true, otherwise it is false. One can also pass in an empty list, in which case the planner will report the complete state back. This is relevant, e.g., for collision checking, when the module does not even know which objects exist and thus could not query for a specific object.

**Search** Temporal Fast Downward performs forward-chaining search in the state space as described in Section 2.3. The internal representation of a state contains

```
typedef bool (*predicateCallbackType)(
    PredicateList* &predicateList);
typedef bool (*numericFluentCallbackType)(
    NumericFluentList* &numericFluentList);
```

Figure 3.5: The function signatures for the state callback interface accept a list of predicates or numerical fluents. When called the planner provides the current valuations for each requested fluent.

a valuation of all state variables in an FDR and the real-valued accumulated cost to reach this state also called the g-value. Most parts of the search algorithm are unchanged as we only adapt the semantics of specific aspects. This affects mainly the successor generation. First, testing for applicability means evaluating the precondition formula in the current state. Whenever the truth value of a variable with an attached link to a *condition checker* module call is requested, we execute this module call instead of doing a state look-up and return the resulting truth value. The same holds when evaluating the goal formula for a state. Next, when applying effects to generate a successor state the effect list might contain module effects, which leads to an *effect applicator* call. The resulting numerical values are directly applied to the list of variables in the effect applicator call as those already refer to FDR variables generated during the translation. Finally, when computing the cost of the operator we call the *cost module* if one was attached to the operator.

We employ a minor optimization to minimize computation time. Recall that any condition is a conjunction of literals. We sort all variables with an attached condition checker call behind purely symbolic variables. Therefore if any symbolic literal evaluates to false, the condition is determined to be false without calling costly external modules.

**Example Translation** We illustrate how a PDDL/M planning task is translated into the internal grounded FDR formulation on the example of the `put-down` operator for placing an object `beer011` that is held in a `side` grasp onto a static object known as `crate_001`. Let us review how the condition checker and effect applicator as well as the `put-down` operator are defined in PDDL/M.

```
(checkTransfer ?target - movable ?place - static ?grasp - grasp
    conditionchecker checkTransfer@libtrajectoryModule.so)
(applyTransfer ?target - movable ?place - static ?grasp - grasp
    (q0) (q1) (q2) (q3) (q4) (q5) (q6)
    (x ?target) (y ?target) (z ?target)
    (qx ?target) (qy ?target) (qz ?target) (qw ?target)
    effect applyTransfer@libtrajectoryModule.so)
```

```
(:action put-down
    :parameters (?x - movable ?y - static ?g - grasp)
    :duration (= ?duration 5.0)
    :precondition (and
        (at ?y)
        (grasped ?x ?g)
        (is-grasp-for ?g ?x)
        ([checkTransfer ?x ?y ?g])
        )
    :effect (and
        (not (grasped ?x ?g))
        (handempty)
        (on ?x ?y)
        ([applyTransfer ?x ?y ?g])
        )
)
```

The first part generated is the translation table mapping PDDL predicates to FDR variables.

```
begin_pddl_translation
grasped 2 beer011 side 0 0
...
end_pddl_translation
```

Here, the `grasped` predicate symbol was grounded with the two objects `beer011` and `side`. The next two numbers give the variable-value pair, in which this grounded predicate is true, i.e., when the FDR variable 0 has the value 0. A predicate callback for `grasped beer011 side` will therefore retrieve the value of variable 0 from the current state and compare it to the value 0. The condition checker and effect applicator module calls are grounded from the operator instance `put-down beer011 crate_001 side`.

```
begin_modules
checkTransfer@libtrajectoryModule.so
  3 ?x movable beer011 ?y static crate_001 ?g grasp side 45
applyTransfer@libtrajectoryModule.so
  3 ?x movable beer011 ?y static crate_001 ?g grasp side
  me-0 14 31 32 33 34 35 36 37 42 43 44 39 40 41 38
...
end_modules
```

The first entry states that the condition checker call to the function `checkTransfer` in `libtrajectoryModule.so` when called with three parameters that have the values `beer011, crate_001, side` determines the truth value of the FDR variable 45. The second entry refers to the effect applicator call `applyTransfer` in the same library with the same parameters. This is known as the module effect `me-0` that sets 14 affected fluents. The next 14 numbers give the FDR variables that hold those. The following listing gives an excerpt of the translated operator.

```
01  begin_operator
02  put-down beer011 crate_001 side
03  2
04  0 0
05  45 0
06  4
07  0 0 1 0 0 0 -1 1
08  0 0 0 2 -1 0
09  0 0 0 4 -1 0
10  0 0 0 me-0
11  end_operator
```

Here, we focus on the relevant aspects. Lines 03–05 define the precondition. There are two literals. Variable 0 must be 0, i.e., the object must be grasped in a side grasp (see the translation above) and variable 45 must be 0. The modules section above states that variable 45 is linked to a condition checker call. The four effects are listed in lines 07–10, one per line. The first three effects are symbolic effects, where the last number triple states that, e.g., for line 07 variable 0 is set from -1, which is a marker for any value, to 1. This means that the object is not grasped any more (variable 0 would be 0). Instead of a number triple line 10 ends in `me-0`, which means that the module effect `me-0` defined above will be applied in this case.

**Soundness and Completeness** We cannot overcome the theoretical limitations illustrated in Section 3.3.2. Here we consider how the implementation affects those properties and what guarantees we have. These are always understood under the assumption that external modules satisfy the requirements specified previously, i.e., assuming that they terminate and deterministically compute values that are regarded as "correct". For soundness this means that an implementation must not produce any plans that are not in the state space (i.e., invalid plans). For affecting completeness this means that any plan in the state space is eventually found by the search. In that case, if the state space is finite the search is decidable, otherwise it is only semi-decidable.

It is fairly obvious that semantic attachments, as implemented in TFD/M, do not affect soundness of the planning algorithms. *Effect applicators* virtually define how a correct state transition looks like in the presence of semantic attachments, whereas *condition checkers* only restrict the options of the planner, but do not alter them. As we are using a forward-chaining search, every operator application has been proven to be valid and successors are a direct result of applying an operator in the parent state.

Semantic attachments themselves do not affect completeness, since *condition checkers* only rule out transitions that are considered "incorrect". For *effect applicators* we assume that choices are uniquely determined by the current planning state, so that we cannot lose possible plans through "unfortunate" effect selection in the module.

The search algorithm itself might not enumerate every possible plan as its expansion strategy depends on the search guidance heuristic. A simple breadth-first search would give us such guarantees. However, in practice planning tasks might become large and require a more goal-driven search to find plans in reasonable time. In that case one must evaluate if such a behavior is sufficient for the desired application.

## 3.5 Evaluation

In this section we present three experiments. The first experiment is an adaptation of a standard benchmark domain that does not add any new features, but provides insight on the computational overhead solely caused by module calls itself. The second experiment shows a new variant of the logistics domain that respects the geometry of packages when determining if a package can be loaded. In the third experiment a geometric manipulation planning domain is generated, in which semantic attachments check for collision-free motion plans in pick and place scenarios. For the first two experiments we enabled concurrent actions. All experiments have been run on a standard desktop computer with an Intel Core2Duo E6400 CPU and 2 GB of RAM.

### 3.5.1 Computational Overhead

The first experiment is designed to show the overhead introduced by the module calls alone. This consists of performing the actual function call and using the state callback interface. As an example we chose the *crew-planning* domain of the International Planning Competition (IPC) from 2008. The reason is that it contains numerous operators that all have one predicate in common, namely the predicate `available`, showing if a crew member is available for executing a task.

We wrote a condition checker that models this predicate by executing a callback to the symbolic planner, requesting the truth value of the `available` predicate in the current state and returning its truth value. Essentially the module does not do anything different and does not perform any extra calculations. This means the semantics and thus the reachable state space of the original domain formulation are unchanged, so that the results are comparable.

| # | TFD | TFD/M | % | # | TFD | TFD/M | % |
|---|---|---|---|---|---|---|---|
| 01 | 0.01 | 0.01 | 0 | 16 | 0.61 | 0.78 | 28 |
| 02 | 0.01 | 0.02 | 100 | 17 | 0.73 | 0.96 | 32 |
| 03 | 0.01 | 0.02 | 100 | 18 | 0.85 | 1.10 | 29 |
| 04 | 0.04 | 0.05 | 25 | 19 | 1.89 | 2.38 | 26 |
| 05 | 0.08 | 0.10 | 25 | 20 | 3.19 | 4.06 | 27 |
| 06 | 0.14 | 0.18 | 29 | 21 | 2.47 | 3.12 | 26 |
| 07 | 0.16 | 0.24 | 50 | 22 | 0.16 | 0.19 | 19 |
| 08 | 0.18 | 0.24 | 33 | 23 | 0.12 | 0.14 | 17 |
| 09 | 0.29 | 0.37 | 28 | 24 | 0.20 | 0.26 | 30 |
| 10 | 0.59 | 0.75 | 27 | 25 | — | — | — |
| 11 | 0.47 | 0.61 | 30 | 26 | 1.50 | 1.89 | 26 |
| 12 | 0.58 | 0.76 | 31 | 27 | — | — | — |
| 13 | 0.05 | 0.08 | 60 | 28 | 3.82 | 4.71 | 23 |
| 14 | 0.08 | 0.12 | 50 | 29 | 5.74 | 7.21 | 26 |
| 15 | 0.06 | 0.07 | 17 | 30 | 5.55 | 6.89 | 24 |

Table 3.1: Results of the first experiment (runtimes are in seconds). TFD indicates the original domain formulation, while TFD/M lists runtimes that include module calls. The % columns show the runtime increase from the former to the latter.

In our experiment we ran TFD/M on the original version of the domain, and then compared runtimes with the modified version that adds the module call. Table 3.1 shows the planning time until the first plan was found (for a timeout of 30 minutes and a memory limit of 1 GB). As expected, the runtime for the module version of the domain is higher. However, it can be seen that the relative overhead is independent of the problem size, thus scaling properties of the planner are not affected.

To put the observed overhead into perspective it should be noted that in actual problems it is not the module call itself that takes a majority of the runtime, but the module's calculations. These, however, are necessary to provide sound semantics. The increase in runtime is as anticipated, as we replaced a predicate check that is usually implemented as an integer comparison by a function call that in turn creates a callback to the requested predicate. Additionally, we chose a harsh domain for this experiment as the crew planning domain calls this module in almost every operator.

### 3.5.2 Transport Logistics

The second experiment presents a full implementation of a PDDL/M task that uses non-trivial semantic attachments. We follow the transport example proposed in the motivation section. Our custom domain models a classic logistics task where trucks are allowed to carry multiple packages with one crucial adaptation: The `pick-up-package`

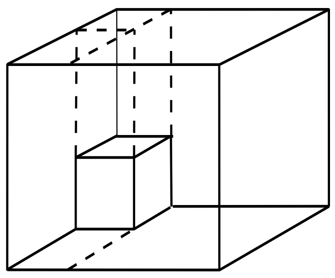


Figure 3.6: Recursive packing of rectangular objects: Once a package has been placed in a corner, three new rectangular containers emerge from the remaining space into which the remaining packages are recursively packed in the same manner.

operator's precondition contains a semantic attachment implemented as a condition checker. The module `canLoad` is a packing algorithm that we shortly describe. The algorithm needs to solve the three dimensional bin-packing problem which, even for one bin, is already NP-hard (Martello et al., 2000). As our main focus is implementing a correct, but not necessarily optimal solution, we therefore use a heuristic packing algorithm.

Our implementation follows a recursive approach of packing a set of rectangular packages into one rectangular container. First, the largest package that fits the container is placed in a corner. Second, the remaining space is partitioned into three new rectangular containers as shown in Figure 3.6. Third, the set of remaining packages is recursively packed into the remaining containers, starting with the smallest. If no unpacked packages remain, a packaging has been found and the module returns true. To make the planning algorithm complete, the exact method for three dimensional bin-packing (Martello et al., 2000) could be used. In that case our or any other simplified solution is an obvious choice for a search guidance heuristic, although it should be noted that it does not provide a relaxation.

The implemented attachment combined with the transport-modules domain adapted to PDDL/M was run on examples based on the transport-numeric domain of the International Planning Competition 2008. Results are shown in Table 3.2 and indicate the time in seconds until a valid plan was found. As in the first example, a timeout of 30 minutes and a memory limit of 1 GB was set.

These results only indicate the feasibility of the approach. A direct comparison with the original transport-numeric domain without semantic attachments does not make sense as we changed the domain semantics. Our approach is not optimal as a better packing algorithm could fit more packages into a truck. However, it is sound as it provides an actually working packaging. For practical problems this is an important factor. Besides the domain formulation of transport-numeric we also based the example problems in Table 3.2 on the transport-numeric problems. We discovered that the package sizes used in the original problems mostly lead to trivial solutions

| # | Trucks | Packages | Locations | Runtime |
|---|---|---|---|---|
| 01 | 2 | 2 | 5 | 0.01 |
| 02 | 2 | 4 | 10 | 0.36 |
| 03 | 3 | 6 | 15 | 0.81 |
| 04 | 3 | 8 | 20 | 1.70 |
| 05 | 3 | 10 | 25 | 33.86 |
| 06 | 4 | 12 | 30 | 27.47 |
| 07 | 4 | 14 | 35 | 146.62 |
| 08 | 4 | 18 | 45 | 244.45 |

Table 3.2: Results of the transport logistics experiment (runtimes in seconds).

as packing multiple packages was rarely possible. Therefore we adapted the package sizes to present a challenge to the planner. This further shows the necessity of being able to provide correct semantics in a planning task formulation.

### 3.5.3 Manipulation Planning

In this experiment we implement a manipulation planner by integrating a motion planner into our symbolic planner using semantic attachments. The system produces motion plans for each `pick-up` and `put-down` action and thus the results are directly executable on real robots. The purpose of this experiment is on the one hand to demonstrate that such a system is possible with semantic attachments, and on the other hand to investigate the feasibility in practice in particular with regard to computation times.

The PDDL/M domain consists of a `pick-up` and a `put-down` operator as described in the examples before. These are implemented by computing if a collision-free motion plan could be found in the case of the *condition checker* and the result of the *effect applicator* is given by the goal configuration in that motion plan. The motion planner itself is a randomized roadmap planner (Kavraki et al., 1996; Dornhege et al., 2009b; Gissler et al., 2009).

We evaluate our manipulation planning system by conducting several experiments of increasing difficulty in two environments. The scenarios target different challenges for integrated task and motion planning. The first environment illustrated in Figure 3.7 consists of the robot surrounded by three tables. Here various movable items such as bottles or cereal boxes are to be placed at target locations. While this is simple from the motion planning side the focus is on the problem that target locations might be blocked by other objects. A straight-forward pick up and place at the target location does not work in these cases. The planner is forced to detect such situations and plan for them accordingly by moving blocking objects out of the way first.

The second scenario shown in Figure 3.8 is motivated by a grasping dexterity

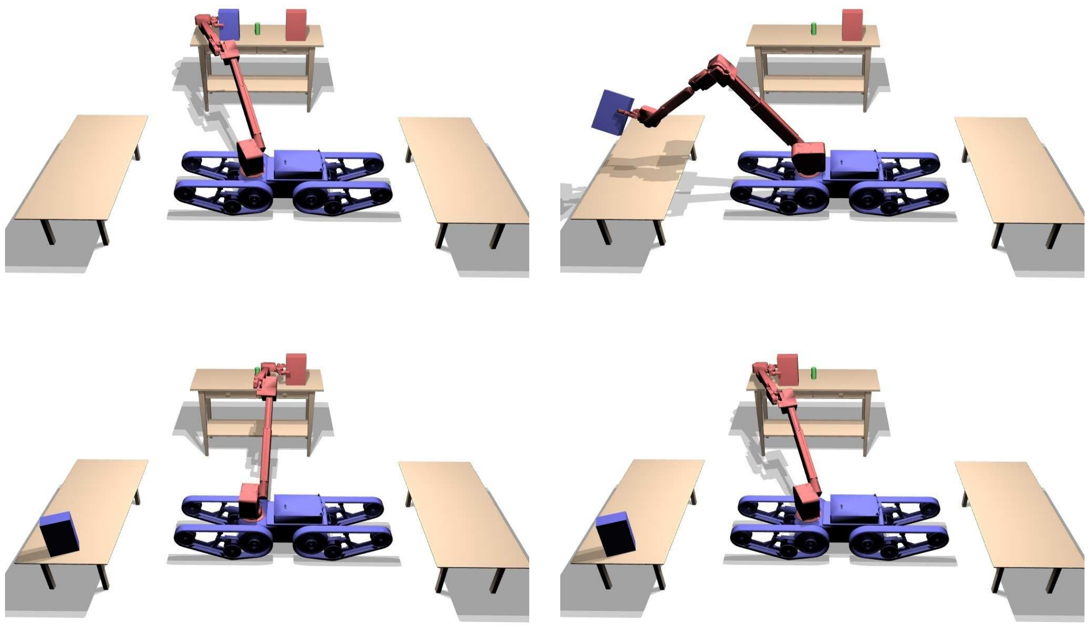


Figure 3.7: Execution of a manipulation plan in the first environment. The task is to place the red box to the left of the table, where the blue box is located. Therefore, it first has to remove the blue box from that position (upper left) and place it somewhere else (upper right and lower left).

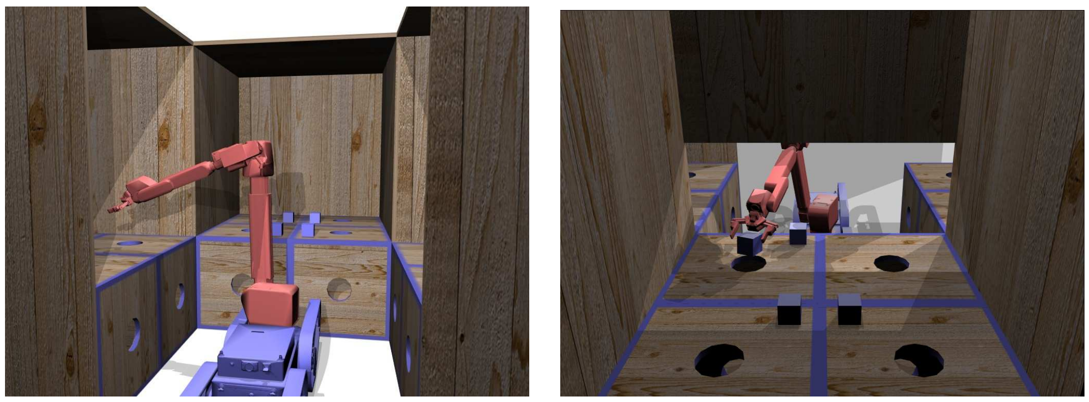


Figure 3.8: The grasping dexterity test, where small cubes have to be put into holes. Left: Scenario overview with open shelves. Right: Inside view of one shelf, when a vertical board is mounted.

test (Jacoff and Messina, 2007). Small cubes of about 8 cm side length must be picked up and placed in holes of 15 cm in diameter. These holes are located in the front and rear of a shelf that is 1.2 meters wide. In principle this is a pick and place scenario. The focus here is on the difficulty for motion planning. The shelves as shown in Figure 3.8 left are open and allow the robot to easily reach locations. When a vertical board is mounted above this shelf (see Figure 3.8 right) the shelf becomes a confined space and robot motions are significantly more constrained.

We separate problem instances for the first environment in three classes: Class I contains simple pick and place tasks, Class II requires to move one object out of the way and in Class III multiple objects have to be moved. For the second environment we evaluate the same problem instances with and without the vertical board present. The limiting factor for all instances was runtime (memory consumption for the hardest problems was below 100 MB), which is dominated by the motion planner's computations. We executed 13 runs for each instance with different seeds and give average runtimes for the first environment in Table 3.3 and for the second environment in Table 3.4.

The results for the tables scene indicate that even replacing multiple objects still leads to reasonable runtimes. It is obvious that problems, where objects have to be moved away (Class II and III), are harder. Here even the smallest problems require more time than most of the tasks in Class I. This shows that runtimes are not only influenced by the number of objects, but also by the plan length and thus the search depth required to find a plan. The more actions have to be planned for, the more alternatives there are to investigate for the planner. These kind of problems are especially hard as the symbolic information does not encode any knowledge about when two objects can be placed on the same table. This depends on the actual poses and shapes of the objects and thus has to be discovered by the semantic attachments in geometric computations. It also shows that a purely symbolic abstraction will fail to produce correct plans in these cases.

Results for the grasping dexterity test show that especially the problem instances requiring to grasp the rear cubes (3–5, 10, 11) are more challenging. These are still solved quickly when there is no board obstructing the way. In the more confined setting with the vertical board, reaching the rear of the shelves poses a hard problem for the motion planner, which is evident in the runtimes for those problem instances. The most difficult problem is instance 11 that places the two rear cubes in holes at the left and right compartment, so that the manipulator has to be moved in and out of a shelf four times. This shows that external computations have a large influence on the performance of the combined planning system.

| Class I | Runtime [s] |
|---|---|
| 01 | 3.48 ± 1.23 |
| 02 | 6.08 ± 3.49 |
| 03 | 3.44 ± 1.61 |
| 04 | 1.47 ± 0.12 |
| 05 | 3.77 ± 0.97 |
| 06 | 3.98 ± 3.01 |
| 07 | 4.75 ± 2.36 |
| 08 | 5.27 ± 2.71 |
| 09 | 63.83 ± 7.67 |
| 10 | 5.66 ± 7.50 |
| 11 | 12.48 ± 14.74 |
| 12 | 3.30 ± 0.96 |
| 13 | 5.80 ± 2.40 |

| Class II | Runtime [s] |
|---|---|
| 01 | 24.32 ± 8.63 |
| 02 | 24.95 ± 9.25 |
| 03 | 91.87 ± 14.01 |
| 04 | 30.26 ± 9.74 |

| Class III | Runtime [s] |
|---|---|
| 01 | 37.33 ± 6.85 |
| 02 | 15.50 ± 2.52 |
| 03 | 78.55 ± 45.61 |

Table 3.3: Results for the tables scene. Problem instances are separated in three classes: Simple pick-and-place tasks (Class I), problems that require replacing another object to reach the goal configuration (Class II), and problems that require replacing multiple objects (Class III).

| Problem | without board [s] | with board [s] |
|---|---|---|
| 01 | 0.06 ± 0.01 | 0.06 ± 0.01 |
| 02 | 0.06 ± 0.00 | 0.06 ± 0.00 |
| 03 | 0.17 ± 0.01 | 59.46 ± 41.92 |
| 04 | 0.17 ± 0.00 | 67.96 ± 46.87 |
| 05 | 11.22 ± 9.50 | 207.66 ± 143.61 |
| 06 | 0.12 ± 0.01 | 0.12 ± 0.00 |
| 07 | 0.39 ± 0.01 | 0.12 ± 0.00 |
| 08 | 0.23 ± 0.00 | 0.24 ± 0.01 |
| 09 | 0.23 ± 0.01 | 0.24 ± 0.00 |
| 10 | 1.51 ± 0.01 | 162.00 ± 52.99 |
| 11 | 54.79 ± 21.00 | 978.35 ± 1105.81 |

Table 3.4: Results for the grasping dexterity scenario. Problem instances have been evaluated with and without the vertical board present.

## 3.6 Conclusion

Planning occurs in many real-world problems. However, often the purely symbolic nature of classical AI planning tasks is insufficient to describe such problems accurately. This is most notable in *robotics* applications where causal, symbolic reasoning must be tightly entwined with numeric computations, and where both may directly influence each other. The separation of symbolic and geometric reasoning into distinct phases is an abstraction that only works under the assumption that the symbolic planning problem can be isolated from other reasoning tasks, which the planner is not designed to solve.

We believe that interfacing symbolic planners with non-symbolic reasoners such as manipulation planners or path planners *during* the planning process is imperative for their use in robotics. Therefore we have presented an approach to integrate external reasoning mechanisms, so-called semantic attachments, directly into a planner. An important aspect of this work is that the interface is domain-independent. This allows domain designers to use domain-independent planners, and extend them with domain-specific sub-solvers where necessary. We have specified a suitable extension of PDDL to model them, and have described criteria under which soundness and completeness of planners are maintained when they are extended with semantic attachments.

Our initial results show that it is feasible to implement complex robotic planning systems such as manipulation planning (Dornhege et al., 2009b; Gissler et al., 2009) with semantic attachments, where we also investigated specific search guidance heuristics (Westphal et al., 2011). This transfers to many other applications, for example multi-robot coordination (Wurm et al., 2010, 2013). We will address further robotics applications in Chapter 5. This planning system has been improved in several ways. We have developed a cleaner and more efficient interface to access the planner state (Hertle et al., 2012). The efficiency with regard to solving more complex tasks by better suited heuristics is the topic of Section 4.3.4. For real-world tasks we also developed strategies to prevent costly external computations as much as possible (see Chapter 5).

There is one important restriction in PDDL/M and our implementation: In general, there may be many options for how to achieve a module effect. For example, a manipulation planner may find several poses at which it could place an object. Currently, we only permit modules that return exactly one result. In the next chapter we will enable the planner to branch over an initially unknown, and possibly infinite number of outcomes online.

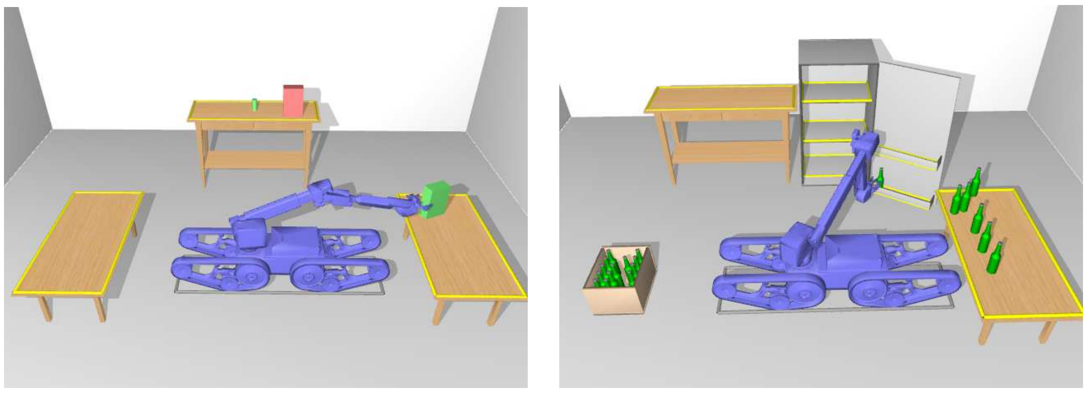
# Chapter 4

# Partially Groundable Planning Tasks

Integrated task and motion planning with semantic attachments focuses on correct geometric modeling of symbolic actions. However, from the planner's perspective actions are still considered to be purely logical. There is one operator to "put the cup on the table". This hides the geometric decision, where exactly to put the cup that is left to some part of the geometric reasoning infrastructure. The decision to put the cup to a specific position does influence any following operators, for example by blocking access to other objects. If we want to capture a robot's capabilities, we need to model a new kind of decision—the decision to try another geometric position. In the words of planning this models multiple different instantiations leading to multiple branches for the symbolic operator. The problem is that there are a large or even infinite number of possibilities. Classical planning systems do not handle such cases well for practical and theoretical reasons. Neglecting to consider multiple geometric outcomes, however, prunes away a large number of solutions or possibly the only solution.

For practical applications different solutions exist that mitigate this problem by using a classical planning formulation with a finite number of successors. A straightforward solution instantiates a fixed number of operators that correspond to different geometrical choices. This enables searching multiple possibilities. Another approach moves the geometric selection to the geometric reasoning. When an operator is to be applied a geometric reasoner tries different geometric solutions and returns the best solution. This approach provides a clean separation between the geometric and symbolic parts of the planning process. However, a bad choice for the geometric outcome might prove disadvantageous later on. Any advanced method for choosing a good outcome cannot guarantee that this is suited for subsequent operators unless it performs planning by itself. A solution in the spirit of integrated task and motion planning computes possible geometric choices during the planning process dynamically. The advantage is that such instances are only computed when needed, although there still can only be a fixed number. We will review related work in more detail in Section 4.1.

All these solutions face the problem that ultimately the number of geometric choices for an operator in classical planning is finite. Consider the example in Figure 4.1. The complex scenario on the right requires the robot to move bottles from the crate on the



**Large number of positions**

slow/likely solvable slow/likely solvable

**Small number of positions**

fast/likely solvable fast/hard to solve

Figure 4.1: This figure compares an easy scenario (left) with a more complex scenario (right). Choosing a large number of geometric positions makes it likely that both scenarios are solved, although a small number is sufficient for the simpler task.

left into the fridge door, which requires regrasping and precise positioning. A large number of geometric positions must be investigated to find a plan for this setting. The goal of the simpler scenario on the left is to move all objects from the center table to the right one. This is likely to be solved with a small number of positions as there is no constraining geometry. A larger number in this case would only increase search time. Ideally one wants to have a small search space for simple tasks that can be searched fast and a larger more complex one when necessary. However, how many placements must be considered depends on many factors such as what algorithm is used to select geometric positions and how complex the scenario is. In the case of a Monte-Carlo algorithm chance is also a factor. These things are hard to predict in general. Thus, one must balance between gaining faster solutions considering less possibilities at the risk of pruning out the only feasible solution. The problem is that the decision how many successors to generate is made at either the geometric or the symbolic level. An integration of both is not straight-forward. On the practical side this means that backtracking needs to be interleaved between both levels and must influence each other. On the theoretical side we must consider that there might be an infinite number of branches, something that classical planning systems do not support. Even for a large finite number it is not desirable to compute all possibilities at once for efficiency reasons.

We approach this problem by integrating branching as part of the planning process. The main idea is that the planner asks for additional successors if it deems that necessary or continues the search elsewhere with the possibility to come back later. As a model for planning tasks with infinite numbers of successors we use *partially groundable planning tasks*. A classical planning task as in Chapters 2 and 3 is specified using schematic operators and a given finite object set. The state space to be searched is defined by the *grounded* representation, where all parameters in each schematic operator are replaced with any possible combination of objects. In our new formalization this object set is not required to be finite, thus possibly leading to an infinite number of operators and an infinite branching factor. As a result of this explicitly grounding all operators is impossible. For efficiency we use two different object sets, where one must be finite. The finite set of objects enables to *partially ground* operators, i.e., ground some parameters of an operator. Consider the example of a `put-down` operator. The parameters for the object to be placed and the surface to place it on, e.g., cup and table, are given from the finite set. The exact pose on the surface is an additional parameter chosen from an infinite set (e.g., $\mathbb{R}^2$). While we can have one explicit operator for each cup and table in the task, one explicit operator for each pose is clearly impossible. Partially groundable planning tasks are formalized in Section 4.2.

In our implementation the non-finite object set is given implicitly by a new kind of semantic attachment: a *grounding module*. Repeated calls to an external function during planning generate different objects and thus represent multiple branches from a schematic operator. As for other semantic attachments the interface is generic, so that any geometric reasoner that provides different instances for a schematic operator can be used. However, the decision to generate additional operators is moved to the planner. In Section 4.3 we provide a novel search algorithm based on the principle of heuristic search that handles these decisions even for planning tasks with infinite numbers of successors. As we deal with huge state spaces and large or infinite branching factors we aim for satisficing instead of optimal plans. In Section 4.4 we evaluate our algorithms in realistic problem settings, show the limits of our new algorithm and compare its performance to the classical search adaption.

## 4.1 Related Work

The problem of searching continuous spaces and choices that we address in this chapter is commonly found in motion planning. Even for the specific case of motion planning this is a hard problem and many state of the art solutions are Monte-Carlo algorithms that sample the collision-free configuration space of the robot to build a graph of collision-free motions called the *roadmap*. Besides collision-free robot motions it is also possible to plan for manipulation tasks. A manipulation graph is build that is composed of transit and transfer motions (Alami et al., 1995; Latombe, 1991). Transit motions move the manipulator without any objects, while transfer motions move grasped objects. Both are planned using a motion planner. A specific problem here is to generate the configurations that are to be connected with these paths if they are not given as an input. Simeon et al. (2004) address this by sampling this subspace explicitly and thus building up the manipulation graph as part of the planning process. Their work is able to deal with continuous numbers of grasps and object placements. Beyond pick and place motions Barry et al. (2013) integrate additional motions like pushing objects into their DARTT planner.

Those approaches are still purely geometric solutions. For generic robot planning this is not sufficient as symbolic information is required, for example, to reason about beliefs (Kaelbling and Lozano-Pérez, 2013). If we generalize the setting to integrated task and motion planning, we find the problem of determining suitable places as different geometric interpretations of symbolic actions. One option is to only plan on the symbolic level and thus give an underlying controller more freedom to adapt to the real-world scenario (Kresse and Beetz, 2012). However, if plans are constructed from a purely symbolic description of the world, geometric constraints and dependencies that were not considered might prevent any such plan from being executable. This is especially true for longer, more complex plans. If geometric choices are considered, one must determine how backtracking is to be performed either in the symbolic plan or on geometric possibilities. Kaelbling and Lozano-Pérez (2011) use *suggesters* that produce a number of possible instantiations for operator parameters during planning. Another possibility is to plan symbolically, but search a number of geometric possibilities when trying to apply each operator (Leidner et al., 2012), so that an operator is only applicable when some geometrically valid configuration is found. Burbridge and Dearden (2013) separate this process by first producing a symbolic plan and then searching for a number of geometric instantiation of this plan. If no such instantiation is found, additional symbolic plans are produced. De Silva et al. (2013) use a *geometric task planner* that solves geometric problems like “make this book graspable” within an HTN planner. The symbolic and geometric parts are connected by shared predicates. They deal with a discrete number of possibilities. Backtracking geometrical possibilities here is not only done for the currently planned action. Geometric instantiations for earlier actions can be adapted if necessary as long as their symbolic effect is not changed.

In contrast to our approach all of those solutions separate at some point between a geometric and symbolic backtracking phase that might be interleaved. We consider geometric and symbolic possibilities as different choices to take by an integrated planner. Additionally our proposed algorithm is able to deal with infinitely many choices. Most classical planning formulations assume a finite number of objects and thus cannot be used without setting some predefined limit. This is mainly due to the fact that planning with an infinite number of objects has been shown to be undecidable (Erol et al., 1995). The problem of limiting the number of geometric possibilities or just choosing a single one can be mitigated by a good selection strategy. For example, Leidner and Borst (2013) use reachability analysis to choose good placements for the robot base. Stulp et al. (2012) define *Action-Related Places* as a probability distribution over robot poses peaked at high success locations for grasping. Such approaches surely push the capabilities of a system, but are still limited by the complexity of the solvable tasks.

The issue of planning tasks, where the grounded representation is not finite is therefore not widely addressed in classical planning. Somewhat similar to this is the problem of domains that are too large to ground in practice. Relevance grounding addresses this by grounding only objects that were learned to be relevant for a certain task (Lang and Toussaint, 2009). Their notion of partial grounding—more precisely partially grounded models—is different from ours as they produce a grounded task that is grounded from a partial set of objects. We consider a partially grounded operator to be an operator where not all parameters have been grounded. The same notion as ours is used for *Partially Grounded Planning as Quantified Boolean Formula* (Cashmore et al., 2013). Here SAT-based planning is used, but tasks are only partially grounded. The ungrounded parameters are quantified, so that quantified Boolean formula are produced that are solved with a QBF solver. The efficiency of this method relies on the fact that the quantified objects are behaving similarly. For our setting this is not the case as we are looking at geometric possibilities that are explicitly different. However, a similar idea is used by the heuristic described in Section 4.3.4 as in the symbolic abstraction of a geometric plan, there is no difference between geometric options. In the related field of answer set programming for distributed systems grounding on-the-fly is also relevant, especially when not all information necessary for grounding might be known beforehand (Dao-Tran et al., 2012).

## 4.2 Partially Groundable Planning Tasks

In this section we formalize partially groundable planning tasks with infinite branching factors. First, we define such tasks and state the resulting planning problem. Then we consider the consequences that this formulation has regarding decidability and possible implementations.

### 4.2.1 Definitions

We adapt the definitions from Section 3.3.1 for PDDL/M planning tasks with semantic attachments and focus descriptions on the differences. The major difference to these planning tasks is that we introduce an infinite set of objects that leads to states with infinite successors. Semantic attachments on numerical variables are a necessary step to provide meaningful semantics for planning tasks with numerical variables and unbounded branching factors. [1] Therefore partially groundable planning tasks are an extension of planning tasks with semantic attachments.

[1]A classical planning task with $K$ grounded Boolean fluents has a fixed state space size of $2^K$ and thus at most $(2^K)^2$ different transitions. Defining more than $(2^K)^2$ different operators must have redundant and thus unnecessary operators.

The planning domain is the same as in Definition 10, i.e., a tuple of a finite set of predicate symbols $P$, a finite set of module predicate symbols $MP$, a finite set of function symbols $F$, a finite set of effect module symbols $EM$, and a finite set of schematic operators $\mathcal{O}$. However, for building terms we use a two-sorted logic. Terms are constant and variable symbols drawn from two disjoint sets $\mathcal{V}_{\mathcal{F}}$ and $\mathcal{V}_{\infty}$. We call the first sort *classical terms* as before and the latter are called *module terms*. Module terms are only used together with modules defining semantic attachments. This is reflected when atomic formulae and expressions of our language are formed from domain symbols and terms.

**Definition 22.** *The atomic formulae are formed from an n-ary predicate symbol from $P$ with $n$ classical terms or from an n-ary module predicate symbol from $MP$ with $n$ classical or module terms. In addition to atomic formulae we define* Primitive Numerical Expressions *(PNE). A PNE is formed from an n-ary function symbol from $F$ and $n$ classical terms. Finally there are* effect modules. *An effect module consists of an n-ary effect module symbol from $EM$, $n$ classical or module terms and a k-dimensional vector of PNEs that we call the affected fluents.*

Note that predicates and PNEs are only formed from classical terms, while module predicates and effect modules are formed from classical and module terms. The state space is derived from the grounded formulation of a planning domain for a specific problem. Therefore, we slightly adapt the problem definition to account for grounding of module terms.

**Definition 23.** *A planning problem is a tuple $(I_L, I_N, \phi_G, O_F, O_\infty)$, where $I_L$ and $I_N$ are the logical and numerical initial state, $\phi_G$ is a first-order formula describing the goal, $O_F$ is a finite set of objects, and $O_\infty$ a set of objects.*

A second object set $O_\infty$ was added. In contrast to $O_F$ we do not require $O_\infty$ to be finite. To define a planning task by its grounded representation we now have two sets of objects corresponding to the two sorts of terms. $O_F$ is used in the same way as the object set in PDDL planning tasks for grounding *classical terms*. $O_\infty$ is used to ground *module terms*. We call $O_\infty$ the *module objects*. The motivation for the separation of $\mathcal{V}_{\mathcal{F}}$ and $\mathcal{V}_{\infty}$ instead of allowing all variable symbols to lead to infinitely many groundings is that we can still explicitly ground some—in practical descriptions almost all—variables using efficient grounding algorithms. As there might be infinitely many objects for a variable in $\mathcal{V}_{\infty}$, for $\phi_G$ we only allow quantification over variables from $\mathcal{V}_{\mathcal{F}}$.

A state of a planning task consists of the sets of Boolean fluents $\mathcal{F}_{\mathcal{L}}$ and numerical fluents $\mathcal{F}_{\mathcal{N}}$, i.e., grounded predicates and PNEs. Grounding here is performed exactly the same as for classical planning tasks. Note that as predicates and PNEs both only contain classical terms, they are grounded from $O_F$ only and thus $\mathcal{F}_{\mathcal{L}}$ and $\mathcal{F}_{\mathcal{N}}$ are finite. The notion of a state is the same as for planning tasks with semantic attachments.

**Definition 24.** *A state* s *is a function that maps each Boolean fluent from $\mathcal{F}_\mathcal{L}$ to $\{true, false\}$ and each numerical fluent from $\mathcal{F}_\mathcal{N}$ to $\mathbb{R}$, i.e.:*

$$s : \mathcal{F}_\mathcal{L} \to \{true, false\}$$

*and*

$$s : \mathcal{F}_\mathcal{N} \to \mathbb{R}$$

*The set of all possible assignments for $\mathcal{F}_\mathcal{L}$ and $\mathcal{F}_\mathcal{N}$ is the set of states $S$.*

As $\mathcal{F}_\mathcal{L}$ and $\mathcal{F}_\mathcal{N}$ are the same as for classical planning tasks the set of states is also the same as before. However, the *reachable* state space defined by the possible transitions given by the grounded operators is different.

**Definition 25.** *A schematic operator is a tuple $(\phi, e, cost)$, where $\phi$ is a first-order formula—the precondition, $e$ is an effect, and cost a function that maps from a state $s$ to $\mathbb{R}_{>0}$—the operator cost. First-order formulae are formed over the atomic expressions from $P$ and $MP$ with the usual logical connectives, wherein quantification is only performed over variables from $\mathcal{V}_\mathcal{F}$. An effect is defined by the finite application of the following rules:*

- *$e$ is called a simple effect if $e$ is a predicate literal (i.e., a predicate or negated predicate). Note that this excludes module predicate literals.*
- *$e$ is called a module effect if $e$ is an effect module $ea$.*
- *$e_1 \wedge \ldots \wedge e_n$ is a conjunctive effect for effects $e_1 \ldots e_n$.*
- *$\forall x : e$ is a universal effect for a variable symbol $x \in \mathcal{V}_\mathcal{F}$ and an effect $e$.*

*We call the set of free variables of an operator the* ***parameters*** *of an operator. The free variables of an operator $(\phi, e, cost)$ are the free variables of its precondition and its effect, i.e., $free(\phi) \cup free(e)$, where the $free(e)$ is defined as:*

$$free(e) = \begin{cases} free(\psi) & \text{if } e = \psi \text{ or } e = \neg\psi \text{ is a simple effect} \\ free(ea) & \text{if } e = ea \text{ is an effect module} \\ \bigcup_{i=1}^{n} free(e_i) & \text{if } e = e_1 \wedge \ldots \wedge e_n \text{ is a conjunctive effect} \\ free(e') \setminus \{x\} & \text{if } e = \forall x : e' \text{ is a universal effect} \end{cases}$$

*$free(ea)$ for an effect module are all variables used in the terms of $ea$ and all variables used in terms of any of the affected fluents of $ea$. A* ***grounded operator*** *is an operator that has no parameters.*

Besides the requirement that first-order formulae use only quantifications over variables in $\mathcal{V}_{\mathcal{F}}$, a schematic operator is the same as for planning tasks with semantic attachments in Definition 14. The differences appear when grounding schematic operators. As an operator's precondition might contain module predicates and its effect might be formed over effect modules, it is possible that there are parameters of an operator in $\mathcal{V}_{\infty}$.

Grounding a schematic operator is performed by instantiating each of its parameters with an object from $O_F$ for variables in $\mathcal{V}_{\mathcal{F}}$ or with an object from $O_{\infty}$ for variables in $\mathcal{V}_{\infty}$. All possible object combinations then lead to the set of grounded operators. The free variables in the precondition, effect, affected fluents in effect modules, and cost modules are substituted by concrete objects. While $\mathcal{F}_{\mathcal{L}}$ and $\mathcal{F}_{\mathcal{N}}$ are only grounded with objects in $O_F$, this leads to grounded module predicates, grounded effect modules and cost modules that might have objects from $O_{\infty}$. Therefore if $O_{\infty}$ is not finite there are infinitely many grounded operators.

Grounded operators describe transitions between two states. The notion of operator applicability and operator application to form a successor state as well as operator cost remains the same as for planning tasks with semantic attachments. However, as module predicates and effect modules are defined over classical and module terms we have to adapt the definitions of condition checkers and effect applicators. The adaptions to the module definitions are straight-forward as we include module objects together with PDDL objects in the definitions. In addition cost modules attached to an operator are also not restricted to classical terms.

**Definition 26.** *A **cost module** is a function $f_{cost}$ attached to a schematic operator $o$ with $n$ parameters, where $f_{cost} : (O_F \cup O_{\infty})^n \to (S \to \mathbb{R}_{>0})$ maps from an $n$-dimensional vector of objects to a function that maps from a state to $\mathbb{R}_{>0}$.*

Operator parameters from $O_F$ and $O_{\infty}$ a passed on to any cost module definition. The external functions implementing any module type are able to accept arbitrary objects from $O_F$ or $O_{\infty}$ and assign proper semantics. The inclusion of module terms in module predicates is reflected in the definition of condition checkers for partially groundable planning tasks.

**Definition 27.** *A **condition checker** is a function $f_{cc}$ attached to an $n$-ary module predicate symbol, where $f_{cc} : (O_F \cup O_{\infty})^n \to (S \to \{true, false\})$ maps from an $n$-dimensional vector of objects to a function that maps from a state to true or false.*

The interpretation of a grounded module predicate in a first-order formula by its attached condition checker is defined exactly as before. The definition of an effect applicator is modified in the same way.

**Definition 28.** *An **effect applicator** is a function $f_{ea}$ attached to an $n$-ary effect module symbol, where $f_{ea} : (O_F \cup O_{\infty})^n \to \left(S \to \mathbb{R}^k\right)$ maps from an $n$-dimensional vector of objects to a function that maps from a state to a $k$-dimensional numerical vector.*

Again objects from $O_F$ and $O_\infty$ can be passed to the attached external function. As for condition checkers we do not change the semantics of applying module effects defined by effect applicators. Thus, operator applicability and the definition of a successor state are the same. Therefore also the state space and the resulting planning problem are defined like before. Let us review the definition of the state space from Definition 18.

**Definition 29.** *The **State Space for a Planning Task** is a directed graph $G = (S, E)$, where the vertices are all possible states in $S$ over $\mathcal{F}_\mathcal{L}$ and $\mathcal{F}_\mathcal{N}$, and there is an edge from a state $s$ to $s'$, iff there exists a grounded operator $o$, so that $o$ is applicable in $s$, i.e., applicable$(o, s)$ and the result of applying $o$ in $s$ is $s'$, i.e., $s' = app(s, o)$.*

Although we do have the same definition of a graph, this graph looks different. The reason is that there might be infinitely many grounded operators. This makes it possible that there are infinitely many edges connected to a state. An infinite number of outgoing edges makes it impossible to explicitly enumerate all successors and thus classical search algorithms cannot be applied any more.

### 4.2.2 Decidability and Solution Concepts

We have defined planning tasks that model real-world problems with geometric choices. Before we come to an implementation we first look into the theoretical limitations that an infinitely large search space and infinite branching factors cause—especially regarding decidability and completeness. Two things differ from purely symbolic planning tasks: Numerical state variables and the cardinality of the object set $O_\infty$, which does not need to be finite. The results of Erol et al. (1995) indicate that infinite entities in a planning task lead to undecidability. We now address this explicitly for our planning tasks.

**Theorem 30.** *For partially groundable planning tasks* PLAN *is undecidable. This holds even when effect applicators only* set *numerical fluents.*

We have already seen in Section 3.3.2 that planning tasks with numerical state variables and semantic attachments are undecidable. Therefore we will address only the case when effect applicators *set* numerical fluents, i.e., the result of an effect applicator does not depend on any numerical fluent in the state.

*Proof.* We again use Diophantine equations to construct a planning task that models polynomial equations with multiple variables in the natural numbers (Helmert, 2002). We create one schematic operator per variable of the equation that just sets the value of this variable provided by an effect applicator. With $O_\infty = \mathbb{N}$ there is one grounded operator for each number in $\mathbb{N}$. A *condition checker* computes if the equation holds and only then allows the transition to a goal state. Any solution to a Diophantine equation with $m$ variables thus is represented by a plan that consists of $m$ grounded operators in an arbitrary order each setting its variable's value to the one of the solution followed by one `go-to-goal` operator with the condition checker. □

We already know that we cannot provide a solution concept that proves that no plan exists for planning tasks with semantic attachments. However, for partially groundable planning tasks even a complete procedure that finds a plan if one exists is not easily attainable. For a finite or countable $O_\infty$ we can enumerate successors as before. For uncountable object sets this is not possible any more. Using a sampling-based Monte-Carlo procedure, with some assumptions we gain the weaker property of *probabilistic completeness* that is known from randomized motion planning. An algorithm is *probabilistically complete* if given that a solution exists the probability of finding that solution goes to one as computation time goes to infinity. In other words: If an algorithm continues to search for a solution, it will eventually be found.

We consider a classical graph search algorithm that uses a FIFO queue. If we expand a state $s$ with a finite number of successors, we add all these to the queue. Otherwise, we add only a single successor state. In addition we add a marker that contains $s$ to the queue. Whenever such a marker is taken from the queue, we add the next successor of $s$ to the queue and again reinsert this marker into the queue. When $O_\infty$ is countable the next successor is given by a bijection to $\mathbb{N}$. For uncountable $O_\infty$ the successors are sampled using a function on $O_\infty$. This sampling function must guarantee that along each plan $s_0, \dots, s_n$ for $i \in \{0, \dots, n-1\}$ the probability of sampling $s_{i+1}$ in $s_i$ is greater than zero. Note that this is not a property that is trivially given for infinitely large sampling sets.

**Theorem 31.** *For countable $O_\infty$ this procedure finds a plan in finite time if one exists. For uncountable $O_\infty$ the procedure is probabilistically complete.*

*Proof.* We first establish that the increase of the queue size between subsequent visits of a state $s_i$, i.e., when reinserting a marker to $s_i$, is bounded. Let $k$ be the largest number of successors for any state that has a finite number of successors. If the search queue has N entries, after visiting all $N$ entries, the queue will have a maximum size of $N \cdot \max(k, 2)$ as each state expansion will have either added at most $k$ successors or added one successor and a marker.

Now assume there exists a finite path $s_0, \dots, s_n$ through the state space that is not found by this procedure. This means there must exist some state $s_i, 0 \leq i < n$ in the path that is visited by our procedure, where $s_{i+1}$ is never visited. If $s_i$ had a finite number of successors, we would have added all successors including $s_{i+1}$ to the queue and as it is a FIFO queue $s_{i+1}$ will be expanded after a finite number of steps, which contradicts the assumption that $s_{i+1}$ is never visited.

If the number of successors is not finite, each time when we visit $s_i$ we add another successor of $s_i$ and a marker to revisit $s_i$ to the queue. If $s_i$ had a countable number of successors, let $s_{i+1}$ be the $m$-th successor of $s_i$, where $m$ is an arbitrarily large integer. We need to visit $s_i$ $m$ times until $s_{i+1}$ is in the search queue. As the queue size increase is bounded for each visit we will eventually reach $s_{i+1}$, contradicting that $s_{i+1}$ is never visited.

If the successors of $s_i$ are uncountable, the probability to sample $s_{i+1}$ as a successor of $s_i$ on the path is greater than zero. If we visit $s_i$ $m$ times, the probability of not sampling $s_{i+1}$ is $(1-p)^m$. As the queue size increase is bounded if we search infinitely long the probability to not sample $s_{i+1}$ is $\lim_{m\to\infty}(1-p)^m$. With $p > 0$ and $(1-p) < 1$ it follows that the probability to not sample $s_{i+1}$ is zero. This means that $s_{i+1}$ will be sampled, added to the search queue and then visited, which again contradicts the original assumption. $\square$

Although we have determined that there is no procedure that decides Plan, it is possible to give semi-decidable or probabilistically complete algorithms depending on the cardinality of the object set $O_\infty$. However, these are uninformed search algorithms with a FIFO queue. A guided search is preferable for complex real-world tasks. In practice there is also the problem of how $O_\infty$ is represented. We will address these issues in the next section.

## 4.3 Planning with an Infinite Branching Factor

In this section we introduce planning algorithms that solve *partially groundable planning tasks* efficiently and show how these are implemented in our planner TFD/M. We extend the implementation for semantic attachments from Section 3.4. There are three aspects that have to be adapted for partially groundable planning tasks. First, we describe how we actually model such tasks in PDDL/M. We introduce *grounding modules* that implicitly define a grounded representation and allow us to generate grounded operators from $O_\infty$ during planning. Next, we address how a partially grounded formulation is computed in the *translation* phase. Finally, we describe two *search* algorithms that deal with infinite branching factors based on the principle of heuristic state space search.

### 4.3.1 Grounding Modules

We need to be able to specify—that is write down—a planning task. PDDL problem descriptions explicitly give the object set for grounding. Given that in our case $O_\infty$ might be infinite, this is impossible. However, in our formulation of a planning task, we still give the finite object set $O_F$ explicitly. In fact, we utilize the definitions of PDDL objects for $O_F$. All variable symbols and objects in a standard PDDL description represent variables and objects from $\mathcal{V}_\mathcal{F}$ and $O_F$, respectively. Variables from $\mathcal{V}_\infty$ and $O_\infty$ are given in a different way. The restriction that variable symbols from $\mathcal{V}_\infty$ are only allowed in modules is thus handled syntactically. For $\mathcal{V}_\infty$ and $O_\infty$ we extend PDDL/M by grounding modules. An example illustrating this for the `putdown` operator is shown in Figure 4.2.

```
(:action put-down
    :parameters (?x - movable ?y - static ?g - grasp)
    :grounding ([determinePutdownPose])
    :duration (= ?duration 5.0)
    :precondition (and
        (at ?y)
        (grasped ?x ?g)
        (is-grasp-for ?g ?x)
        ([checkTransfer ?x ?y ?g])
        )
    :effect (and
        (not (grasped ?x ?g))
        (handempty)
        (on ?x ?y)
        ([applyTransfer ?x ?y ?g])
        )
)
```

Figure 4.2: The `put-down` operator with a grounding module. There are four parameters: `?x ?y ?g` in $\mathcal{V}_{\mathcal{F}}$ and one from $\mathcal{V}_{\infty}$ computed by `determinePutdownPose`. This fourth parameter is implicitly added to the condition checker and effect applicator module calls `checkTransfer` and `applyTransfer`.

```
(:modules
    (determinePutdownPose grounding
        determinePutdownPoseSampling@libtrajectoryModule.so)
        ...
)
```

```
typedef std::string (*groundingModuleType)(
    const ParameterList & parameterList,
    predicateCallbackType predicateCallback,
    numericalFluentCallbackType numericalFluentCallback,
    int relaxed, const void* statePtr);
```

Figure 4.3: Example of the planner interface for grounding modules. Top: Declaration of a grounding module named `determinePutdownPose` implemented in the function `determinePutdownPoseSampling`. Bottom: The C++ interface to be implemented returning the name of a new object from $O_\infty$. The `statePtr` is used to determine if the module is called in the same state.

**Definition 32.** *A* ***grounding module*** *is a function $gm_o : O_F^m \to (S \to O_\infty \cup \{\bot\})$ that is attached to a schematic operator $o$, where $m$ is the number of the operator's parameters in $\mathcal{V}_\mathcal{F}$ and $\bot \notin O_\infty$. The function maps from parameter instantiations from $O_F$ to a function that for a given state in $S$ returns an object in $O_\infty$ or $\bot$ if there are no more objects.*

Grounding modules are attached to operators that have parameters in $\mathcal{V}_\infty$. They implicitly represent $O_\infty$. We currently allow one grounding module per operator. One could still represent $k$ parameters in $\mathcal{V}_\infty$ by using a new object set $O'_\infty = \times_{i=1}^{k} O_\infty$. However, for the scenarios that we considered this was not necessary. Similar to the other module types a grounding module is implemented for a schematic operator independent of the—in this case partial—grounding for a concrete problem to allow a generic implementation.

A grounding module produces objects from $O_\infty$ and returns $\bot$ once all objects are enumerated (for a finite $O_\infty$). Although it is impossible to enumerate a set $O_\infty$ that is not countable, it is still possible to generate objects from the set, e.g., by sampling. A grounded operator is derived from a schematic operator by first instantiating the parameters from $\mathcal{V}_\mathcal{F}$ using a classical grounding algorithm as shown in the next section. Then the grounding module is called for this partially grounded operator. If $\bot$ is returned, it is not possible to ground the operator in this state any more. Otherwise, the parameter from $\mathcal{V}_\infty$ is instantiated. Multiple groundings for the same partially grounded operator are produced by calling the same module repeatedly and represent different branches. [2] Note that in our implementation grounding modules are dependent on the current state enabling to produce different subsets of $O_\infty$ for different states. This is not necessary, but allows a module implementation to quickly exclude impossible groundings, e.g., a putdown action for a certain table does not need to produce positions on another table that will be unreachable anyways. The declarative and procedural parts of the grounding module interface are shown in Figure 4.3. $\perp$ is represented by the empty string.

### 4.3.2 Partial Grounding

The input to partial grounding is a planning task as defined in Section 4.2. The result is a planning task, where all operators are partially grounded. This means that for each schematic operator all parameters from $\mathcal{V}_\mathcal{F}$ were grounded, so that only variables from $\mathcal{V}_\infty$ are left as operator parameters. We perform this step for the same reasons as in classical planners: Operating on (partially) grounded operators is more efficient than using schematic operators. Advanced grounding procedures also enable to prune inapplicable operators before the search.

In TFD/M partial grounding is performed as part of the translation procedure described in Section 3.4. The main point of interest here is the instantiation of grounded atoms and operators. The algorithm produces an over-approximation of the reachable state space beginning with the atoms given in the initial state. If we relax operator conditions, this still results in an over-approximation and thus we retain a sound grounding process. To adapt the grounding algorithm to partially groundable planning tasks we only instantiate variables from $\mathcal{V}_\mathcal{F}$ when computing possible groundings of schematic operators exactly as described before thus effectively generating partially grounded operators. This causes a problem for applicability tests and operator application that are only defined for grounded operators. By our definition the parameters in $\mathcal{V}_\infty$ only appear in *module predicates*, *module effects* or *cost modules*. Therefore we relax every condition in an operator by removing any condition checker from any conjunction. [3] As only condition checkers use the numerical fluents set by effect applicators we can also remove any module effects. *Cost modules* are also not considered here as they are irrelevant for reachability. One can view the partial grounding process as grounding being applied to an abstraction of the state set towards the symbolic part of a state. The set of schematic operators for a planning task is given by $\mathcal{O}$. We denote the partially grounded operators as $\mathcal{P}$.

[2] To allow the function to produce different outputs a parameter $n \in \mathbb{N}$ is added implicitly. This represents the number of groundings already performed by the module for a state.

[3] Normalization is performed before this step, so that all conditions are conjunctions (see Helmert (2009)).

**Algorithm 3** Best-First Search on Partially Grounded Planning Tasks

```
1: open ← $\mathit{PriorityQueue}(\emptyset)$
2: closed ← $\emptyset$
3: current ← $s_0$
4: current_g ← 0
5: best_g[current] ← 0
6: while True do
7:     if current $\notin$ closed or current_g < best_g[current] then
8:         closed ← closed $\cup$ {current}
9:         best_g[current] ← current_g
10:        if current $\models \phi_G$ then
11:            return SOLVED
12:        end if
13:        GENERATE_SUCCESSORS(open, current, current_g)
14:    end if
15:    next_ok, current, current_g ← FETCH_NEXT_STATE(open)
16:    if not next_ok then
17:        return NO_PLAN_FOUND
18:    end if
19: end while
```

### 4.3.3 Searching Partially Grounded Planning Tasks

We use a variant of forward-chaining heuristic best first search in the state space as described in Section 2.3. The search procedure is shown in Algorithm 3. In each step the current state—starting with the initial state—is closed and expanded by computing successors. We discard states that have been closed unless their cost is lower than the closed state's cost. GENERATE_SUCCESSORS computes successors and inserts them into the open queue that is sorted using heuristic estimates. FETCH_NEXT_STATE takes the next state from the open queue. It returns that state and its g-value as well as a flag that states if a state could be taken. This procedure continues until a goal state is found or no new state could be fetched as the open queue ran empty.

This base procedure is not different from classical search. However, we have to adapt the successor generation as it is impossible to compute all successors. Successor generation also has to ground partially grounded operators on the fly using grounding modules before applicability tests can be performed. We present two different algorithms: *Ground N* (Algorithm 4) and *Ground Single Reinsert* (Algorithm 5). Note that in the open queue successor states are only represented implicitly by their parent state and the operator to be applied.

**Algorithm 4** Ground N

```
Parameters: $\mathrm{N} \in \mathbb{N} \cup \{\infty\}$
 1: function GENERATE_SUCCESSORS(open, parent, parent_g)
 2:     for all $o \in \mathcal{P}$ do
 3:         if $grounded(o)$ and $applicable(o$, parent) then
 4:             open.insert((parent, $o$, parent_g),
 5:                 COMPUTEPRIORITY(parent, $o$, parent_g))
 6:         else if not $grounded(o)$ then
 7:             for $i \in \{1, \ldots, N\}$ do
 8:                 ground_param ← $gm_o(grounded\_params(o))$(parent)
 9:                 if ground_param = $\perp$ then
10:                     break
11:                 end if
12:                 $o_g = o[v^o_\infty$ = ground_param]
13:                 if $applicable(o_g$, parent) then
14:                     open.insert((parent, $o_g$, parent_g),
15:                         COMPUTEPRIORITY(parent, $o_g$, parent_g))
16:                 end if
17:             end for
18:         end if
19:     end for
20: end function
```

**Ground N** Ground N shown in Algorithm 4 generalizes the successor generation from classical search to partially groundable planning tasks. The idea is to ground up to $N$ operators and discard the rest, where $N$ is a configurable parameter. Applicable grounded operators are inserted into the open queue as in classical search (ll. 3-5). If an operator is not grounded, we try to ground an instance up to $N$ times (l. 7) by calling the associated grounding module $gm_o$ (l. 8). Here *grounded_params(o)* returns the vector of objects from $O_F$ that $o$ was partially grounded with. If the operator cannot be grounded any more, we stop (ll. 9-11). We say the operator is *grounded out*. Otherwise we produce a grounded operator $o_g$ by assigning the variable $v^o_\infty$ to the ground_param (l. 12). $v^o_\infty$ is the free variable in the partially grounded operator $o$. If the grounded operator is applicable, it is inserted into the open queue (ll. 13-16). This procedure is quite similar to classical search as always all possible operators are added. For a finite object set $O_\infty$ one can set $N = \infty$ and all successors are produced. In that case we get the same behavior as classical search with the difference that the full task is not grounded explicitly, which might be preferable when states have many geometrically different successors.

**Algorithm 5** Ground Single Reinsert

```
1: function GENERATE_SUCCESSORS(open, parent, parent_g)
2:     for all $o \in \mathcal{P}$ do
3:         if grounded($o$) then
4:             if applicable($o$, parent) then
5:                 open.insert((parent, $o$, parent_g),
6:                     COMPUTEPRIORITY(parent, $o$, parent_g))
7:             end if
8:         else
9:             open.insert((parent, $o$, parent_g),
10:                COMPUTEPRIORITY(parent, $o$, parent_g))
11:        end if
12:    end for
13: end function
```

**Ground Single Reinsert** The problem with *Ground N* is that it discards possible successors. As we cannot generate all successors we must balance between generating more successors (branching) or looking at other states (expanding) and interleave both. The idea is to produce only one possible successor per partially grounded operator at each step, but not close this state if there are potentially more. This is motivated by the algorithm presented in Section 4.2.2. However, we produce one successor *per partially grounded operator* and not one per state. Following the principle of heuristic search we leave the decision to revisit a state and produce more possible successors or to expand other states to the heuristic.

The successor generation for Ground Single Reinsert is shown in Algorithm 5. Grounded operators are handled the same as with *Ground N*. They are added to the open queue if applicable (ll. 4-7). In the other case *Ground Single Reinsert* simply adds the partially grounded operator to the open queue (ll. 9-10) without an applicability check. A partially grounded operator in the open queue serves as a marker to produce possible successors by grounding this operator once it is retrieved. Computing a heuristic estimate for this entry is impossible as it does not represent an actual state. This, however, is mitigated by the fact that we use *deferred evaluation* for heuristic computations (Richter and Helmert, 2009). *Deferred evaluation* takes the heuristic of the state's parent as the state's estimate. The original intention of the technique is that per expansion step only one heuristic needs to be computed. In our case this enables us to compute heuristic estimates for grounded and partially grounded operators alike.

The successor generation inserts partially grounded operators in the open queue. Therefore we need to adapt the FETCH_NEXT_STATE procedure to handle those correctly. Algorithm 6 shows this process. First, we check that the queue is not empty and return failure otherwise (ll. 2-4). Then, we retrieve the state and operator from

**Algorithm 6** Fetch Next State

```
Parameters: $N \in \mathbb{N} \cup \{\infty\}$
function FETCH_NEXT_STATE(open)
    if open.empty() then
        return False, None, $\infty$
    end if
    state, op, g $\leftarrow$ open.pop()
    if not grounded(op) then
        ground_param $\leftarrow$ $gm_{op}(grounded\_params(op))$(state)
        if ground_param $= \perp$ then
            return FETCH_NEXT_STATE(open)
        end if
        $op_g$ = op[$v^o_\infty$ = ground_param]
        if num_groundings(state, op) $<$ N then
            open.insert((state, op, g), computePriority(state, op, g))
        end if
        if applicable($op_g$, state) then
            op $\leftarrow$ $op_g$
        else
            return FETCH_NEXT_STATE(open)
        end if
    end if
    return True, app(state, op), g + cost(op)
end function
```

the queue along with the g-value estimation of this state (l. 5). Let us first consider the case of a grounded operator. The function returns `True` and computes the successor state by applying the operator to its parent. The g-value is computed by adding the operator cost to the parent's g-value (l. 21). This is the same behavior as in classical search.

For a partially grounded operator, first we call the grounding module (l. 7). If the operator is grounded out, it can be discarded and we continue with the next state from the open queue by recursively calling FETCH_NEXT_STATE (ll. 8-10). Otherwise, we build the grounded operator $op_g$ (l. 11). Now two things happen: We reinsert the same state and partially grounded operator into the open queue—again as a marker (l. 13). This happens only if we produced less than $N$ grounded operators before in this state. The number of grounded operators generated from a partially grounded operator in a state is maintained in *num_groundings*. $N$ in this case serves the same purpose as for *Ground N*—to limit the number of possible successors from a state. However, in this case, one can set $N = \infty$ even if $O_\infty$ is not finite as only a single grounded operator is added in each step. The second part handles the grounded operator $op_g$. If it is applicable, it is used as the operator for this planning step (ll. 16, 21). Otherwise, we continue with the next state from the queue as before (l. 18).

**Example Run** Figure 4.4 shows an example run using *Ground Single Reinsert* on the scenario shown in Figure 4.5. The task is to move a bottle from a crate into the lower shelf of the fridge door. The robot must regrasp the object in order to avoid collisions of the gripper with the fridge. The example run uses $N = \infty$, i.e., there is no limit on the possible successors of an operator. To understand the behavior of *Ground Single Reinsert* it is important to note the numbers before transitions, which show the order, in which all events took place. These are either insertion into the priority queue or the time of insertion and subsequent removal from the queue (e.g., $36 \rightarrow 38$). Note that for partially grounded operators there are multiple entries as they are reinserted into the queue. Values in parentheses are the priorities.

After initializing with the initial state the only applicable operator `pick-up beer011 crate011 top` is inserted into the priority queue and immediately removed and applied ($2 \rightarrow 3$). The `put-down` operator is only partially groundable. After inserting into the priority queue different grounded operators are tried by the planner. Here the planner alternates between grounding from `put-down beer011 fridge_door_shelf1 top` and `put-down beer011 table3 top` as their heuristic estimates are equal (2) and priorities of reinserted states are higher due to discounting (see Section 4.3.4). As it is geometrically impossible to place the bottle directly in the fridge this continues until a pose on the table is found ($23 \rightarrow 25$). Two grasps for re-grasping the object are applied, a `side` and `top` grasp. The partially grounded operators originating from the two new states are inserted into the queue. When the first grounded operator for `put-down beer011 fridge_door_shelf1 side` succeeds, a goal state is found

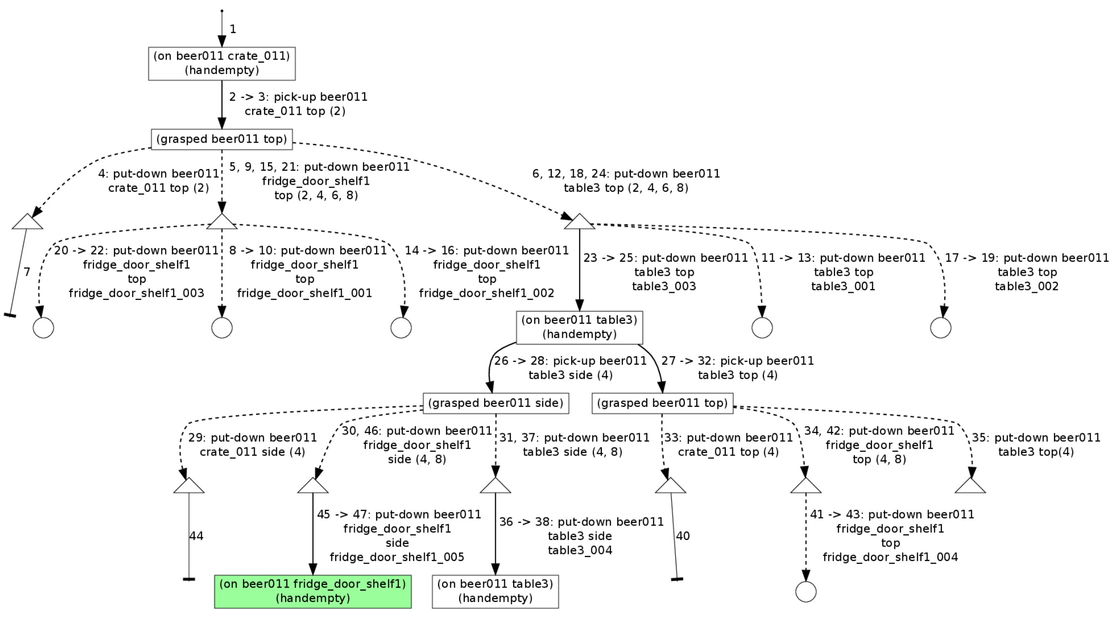

1
(on beer011 crate_011)
(handempty)
2 -> 3: pick-up beer011 crate_011 top (2)
(grasped beer011 top)
4: put-down beer011 crate_011 top (2)
5, 9, 15, 21: put-down beer011 fridge_door_shelf1 top (2, 4, 6, 8)
6, 12, 18, 24: put-down beer011 table3 top (2, 4, 6, 8)
7
20 -> 22: put-down beer011 fridge_door_shelf1 top fridge_door_shelf1_003
8 -> 10: put-down beer011 fridge_door_shelf1 top fridge_door_shelf1_001
14 -> 16: put-down beer011 fridge_door_shelf1 top fridge_door_shelf1_002
23 -> 25: put-down beer011 table3 top table3_003
11 -> 13: put-down beer011 table3 top table3_001
17 -> 19: put-down beer011 table3 top table3_002
(on beer011 table3)
(handempty)
26 -> 28: pick-up beer011 table3 side (4)
27 -> 32: pick-up beer011 table3 top (4)
(grasped beer011 side)
(grasped beer011 top)
29: put-down beer011 crate_011 side (4)
30, 46: put-down beer011 fridge_door_shelf1 side (4, 8)
31, 37: put-down beer011 table3 side (4, 8)
33: put-down beer011 crate_011 top (4)
34, 42: put-down beer011 fridge_door_shelf1 top (4, 8)
35: put-down beer011 table3 top(4)
44
45 -> 47: put-down beer011 fridge_door_shelf1 side fridge_door_shelf1_005
36 -> 38: put-down beer011 table3 side table3_004
40
41 -> 43: put-down beer011 fridge_door_shelf1 top fridge_door_shelf1_004
(on beer011 fridge_door_shelf1)
(handempty)
(on beer011 table3)
(handempty)

Figure 4.4: An example run with *Ground Single Reinsert*. Rectangles are states. The goal is green. Solid arrows show applications of grounded operators. Dashed arrows to triangle marks show partially grounded operators. Outgoing arrows from these are the possible successors. They are solid if applicable, and dashed otherwise. Lines ending in a bar indicate that this operator is grounded out.

(45 → 47). If this was not the case, the planner would continue producing grounded operators for this choice, but also for all other `put-down` operators from each state as they are still in the priority queue and none have been grounded out.

**Soundness and Completeness** Soundness and completeness of the implementation are only guaranteed under the assumptions and conditions stated in Section 4.2.2. For soundness we need to establish that no incorrect plans are produced. For a search-based algorithm this means that any closed state must be the result of applying a series of applicable operators. This is ensured in both algorithms. For Ground N we only insert grounded and applicable operators in the open queue. The successor generation of Ground Single Reinsert puts operators in the open queue that are either grounded and applicable or not fully grounded. The latter case is handled in FETCH_NEXT_STATE. Lines 15–19 guarantee that the operator used in this step is applicable. If it wasn't, FETCH_NEXT_STATE will be called again (l. 18) until either an applicable successor for a state is found or the queue runs empty.

For completeness we need to ensure that a plan is eventually found. This only holds for Ground Single Reinsert with $N = \infty$. With a finite $N$ any successors represented by a partially grounded operator are not visited after $N$ have been generated. Ground N cannot be run with $N = \infty$ on infinite $O_\infty$ as in that case successor generation does not terminate. Ground Single Reinsert works slightly different as the algorithm proposed in Section 4.2.2. However, the same arguments still hold: The increase on the queue size in every step is still bounded. The difference here is that one additional queue entry is added per partially grounded operator. As partial grounding uses the finite object set $O_F$ this number is still finite.

Another noteworthy restriction is that this only holds with a FIFO queue, i.e., breadth-first-search. In general heuristic best-first search does not make such guarantees. For planning tasks with semantic attachments we argued that this still works in practice shown by the experimental data. This not true any more as partially groundable planning tasks are an extension to classical planning tasks with numerical fluents that, with an infinite number of successors, goes beyond adding more expressive semantics. We will address this issue in the following section.

### 4.3.4 Efficient Search Techniques

The aforementioned search algorithms find plans for partially groundable planning tasks. However, the practical problems that we face are likely to produce a large number of successors leading to a large branching factor. Moreover, the geometrical solvers take significantly longer to evaluate than symbolic tests. Therefore we aim for satisficing solutions. A fast and well guided search is crucial for good performance. We address this by introducing a discounting scheme that devalues large numbers of successors from the same state. In addition, we introduce a search heuristic on the assumption that the symbolic part of a planning task provides good guidance information.

**Discounting** In contrast to *Ground N*, which always produces all (up to $N$) successors, *Ground Single Reinsert* gives us the possibility to balance between branching and expanding. We use the search heuristic to make this decision. However, the search heuristic is not well informed about the geometric model that appears as a black box. The heuristic estimate for a reinserted partially grounded operator might again be the best entry in the open queue leading to a large number of successors from this state. To prevent this we employ a discounting function to force the planner to look at different operators. The idea is that this balances the heuristic estimate for a state with the fact that the previous successors from this state have not lead to the goal—otherwise the search would have ended.

The function computePriority computes the heuristic for the state $h(s)$ and passes this through a discounting function that depends on the number of possible successors that have been generated from a partially groundable operator *op* in this state given by *num_groundings*. We use a linear discounting function and greedy search. Thus, computePriority$(s, op, g)$ computes $(1 + \mathit{num\_groundings}(s, op)) \cdot h(s)$ for partially grounded operators and $h(s)$ otherwise.

**Module-Abstracted Search Heuristics** The heuristic in our planner employs an adaption of the context-enhanced-additive heuristic (Eyerich et al., 2009) to semantic attachments. The heuristic can be used independent of the module calls for semantic attachments operating only on the symbolic part of a state. As only condition checkers and effect applicators use variables from $\mathcal{V}_\infty$ partially grounded operators can also be evaluated and applied on the symbolic part of a state. Thus during heuristic evaluation we do not need to ground operators. Although this heuristic already works for planning tasks with semantic attachments, we consider an even better informed heuristic.

In classical planning often the heuristic computation itself takes major parts of the search time. In our case this is different as geometric computations in modules dominate the computation time. This gives light to a new heuristic that we name *module-abstracted heuristic*. The module-abstracted heuristic performs a full classical search in the state space formed by the symbolic part of the state and partially grounded operators. As a heuristic for this search we use the planner's classical search heuristic. The value computed by this heuristic is the length of the plan found by the search. Although using a full search to compute a heuristic estimate might sound infeasible at first, keep in mind that the state space is considerably smaller as there is only one transition per partially grounded operator. Also applicability tests and operator applications now work with purely logical states and do not call costly geometric computations.

In fact what we gain by this heuristic is a combination of the efficiency of a top-down abstraction for task and motion planning and the soundness of an integrated approach. A top-down abstraction would run a search without considering geometric properties and produce concrete geometric instantiations only when executing the plan. This is fast, but can obviously fail. By using an abstract plan only as the search heuristic we follow its guidance, but retain the ability to backtrack.

## 4.4 Evaluation

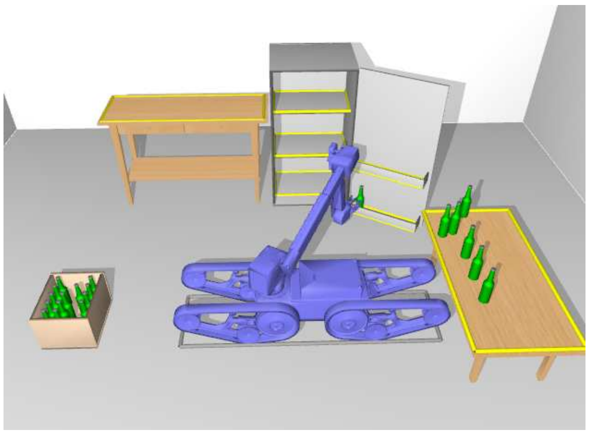


Figure 4.5: This figure shows the scenario used in the first two experiments. Yellow rectangles show the surfaces that the robot can put objects on, i.e., all tables and the shelves in the fridge and fridge door.

We evaluate our algorithms on manipulation planning scenarios shown in Figure 4.5 and Figure 4.11. In the first two experiments we compare the behavior of Ground Single Reinsert and Ground N when successors are computed by sampling or from a fixed discretization. The third experiment investigates the performance of the module-abstracted heuristic on a mobile manipulation task. The planning tasks are designed to be challenging and thus contain intricate manipulation or mobile manipulation that necessitates to generate many successors. The integrated planner is built on our previous work (Dornhege et al., 2009b) and in this case uses a Rapidly Exploring Random Tree for trajectory planning of the manipulator. We also enabled partial state caching and lazy module evaluation, which are introduced in Section 5.5.

### 4.4.1 Sampling-based successor generation

In this experiment we focus on the behavior of Ground Single Reinsert and Ground N when successors are generated by sampling. In particular we are interested in how well both algorithms scale with the increased complexity for larger tasks with more objects. Here it is especially important how dependent they are on setting a suitable value of $N$.

The domain is modeled with two actions: `pick-up` moves the arm towards an object and grasps it in a side or top grasp iff there exists a motion plan that allows the robot to do so without collisions with the environment or other objects. `put-down` moves the arm towards a specific putdown pose on a static object (e.g., a table) and releases the object iff there exists a collision-free motion plan. The placement pose for an object is chosen by a *grounding module* that uniformly samples poses on the target surface.

The scenario is shown in Figure 4.5. Initially all objects are stored in the crate at the left and can only be grasped with a top grasp. We define four tasks with different goals. For the *Table* task all objects are to be positioned on the table at the right. The *Fridge* task requires all objects to be stored anywhere in the fridge. The *Shelf* task positions all objects in the shelves within the fridge, but not in the door. Finally, the goal of the *Door* task is to put objects in the lower shelf of the door. Especially the last task forces intricate trajectory planning to avoid collisions of the object with the shelf and of the gripper with the upper shelf. Also to position a maximum number of objects in the small lower shelf the poses need to be as close together as possible while still allowing the gripper to fit between the objects to put them down. Besides the *Table* task all other settings require regrasping from a top grasp lifting the object from the crate to a side grasp to put an object into the fridge.

There are up to 20 different problems for each task corresponding to 1–20 objects to be placed at the goal surface(s). We used values of $1, 5, 10, 25, 50$ and—if applicable—$\infty$ for N and ran the planner with Ground Single Reinsert and Ground N on all tasks and all problems. As pose sampling is a Monte-Carlo algorithm we executed each of the 20 problems on each of the tasks for each N with ten different random seeds. The tasks were executed on a single core of an Intel Core-i7-3930K. We set the memory limit to 4 GB and a timeout of 1800 seconds. For each run we record the time until the first plan is found and its length or note a failure if no plan was found before the timeout. We show the results where at least one algorithm with one N value succeeded to produce a plan.

We use two ways of comparing both algorithms. A direct comparison uses the same N value for each algorithm. Ground Single Reinsert with $N = \infty$ cannot be compared to Ground N. Thus in this case we compare it to an oracle algorithm that chooses the best run among different N values from each Ground N run. Note that to create this algorithm it is necessary to run Ground N for all N values. This is unfavorable for Ground Single Reinsert. It serves as a comparison of Ground Single Reinsert without a pre-given N, i.e., $N = \infty$ to Ground N under the assumption that we could predict the best N—which is hard to do in practice.

| | Ground N | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 1 | 5 | 10 | 25 | 50 | $\infty$ | 1 | 5 | 10 | 25 | 50 |
| 01 | 3 | 10 | 10 | 10 | 10 | 10 | 5 | 10 | 10 | 10 | 10 |
| 02 | 9 | 10 | 9 | 10 | 10 | 10 | 7 | 10 | 10 | 10 | 10 |
| 03 | 10 | 10 | 10 | 10 | 5 | 10 | 9 | 10 | 10 | 10 | 10 |
| 04 | 9 | 10 | 10 | 7 | 4 | 10 | 10 | 10 | 10 | 10 | 10 |
| 05 | 7 | 9 | 8 | 3 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 06 | 10 | 9 | 4 | 2 | 0 | 10 | 9 | 9 | 10 | 9 | 9 |
| 07 | 10 | 8 | 3 | 0 | 0 | 10 | 9 | 10 | 9 | 10 | 10 |
| 08 | 7 | 2 | 0 | 0 | 0 | 9 | 10 | 8 | 6 | 10 | 8 |
| Total | 65 | 68 | 54 | 42 | 29 | 79 | 69 | 77 | 75 | 79 | 77 |

Fridge

| | Ground N | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 1 | 5 | 10 | 25 | 50 | $\infty$ | 1 | 5 | 10 | 25 | 50 |
| 01 | 5 | 10 | 10 | 10 | 10 | 10 | 4 | 10 | 10 | 10 | 10 |
| 02 | 8 | 10 | 9 | 8 | 6 | 10 | 7 | 10 | 10 | 10 | 10 |
| 03 | 8 | 8 | 9 | 4 | 0 | 10 | 9 | 10 | 10 | 10 | 10 |
| 04 | 10 | 4 | 9 | 0 | 0 | 9 | 8 | 10 | 10 | 10 | 9 |
| 05 | 9 | 4 | 2 | 0 | 0 | 10 | 10 | 9 | 9 | 10 | 10 |
| 06 | 9 | 0 | 0 | 0 | 0 | 9 | 8 | 8 | 8 | 7 | 9 |
| 07 | 5 | 0 | 0 | 0 | 0 | 9 | 9 | 9 | 8 | 7 | 9 |
| 08 | 6 | 0 | 0 | 0 | 0 | 9 | 9 | 8 | 9 | 8 | 8 |
| 09 | 0 | 0 | 0 | 0 | 0 | 5 | 2 | 5 | 6 | 4 | 8 |
| 10 | 1 | 0 | 0 | 0 | 0 | 4 | 4 | 5 | 5 | 5 | 7 |
| 11 | 1 | 0 | 0 | 0 | 0 | 3 | 4 | 5 | 6 | 6 | 4 |
| 12 | 0 | 0 | 0 | 0 | 0 | 4 | 1 | 6 | 5 | 5 | 5 |
| 13 | 0 | 0 | 0 | 0 | 0 | 4 | 2 | 0 | 2 | 3 | 4 |
| 14 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Total | 62 | 36 | 39 | 22 | 16 | 97 | 77 | 95 | 98 | 96 | 103 |

Shelf

| | Ground N | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 1 | 5 | 10 | 25 | 50 | $\infty$ | 1 | 5 | 10 | 25 | 50 |
| 01 | 6 | 10 | 10 | 10 | 10 | 10 | 8 | 10 | 10 | 10 | 10 |
| 02 | 8 | 10 | 10 | 10 | 10 | 10 | 9 | 10 | 10 | 10 | 10 |
| 03 | 10 | 10 | 10 | 10 | 10 | 10 | 5 | 10 | 10 | 10 | 10 |
| 04 | 10 | 7 | 2 | 1 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 05 | 9 | 6 | 9 | 5 | 1 | 10 | 10 | 10 | 10 | 10 | 10 |
| 06 | 10 | 9 | 10 | 1 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 07 | 9 | 8 | 6 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 08 | 6 | 2 | 2 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 09 | 9 | 7 | 3 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 10 | 10 | 10 | 10 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 11 | 9 | 9 | 1 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 12 | 6 | 2 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 13 | 10 | 7 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 14 | 9 | 8 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 15 | 7 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 16 | 5 | 2 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 17 | 8 | 5 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 18 | 3 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 19 | 8 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 20 | 0 | 0 | 0 | 0 | 0 | 10 | 8 | 10 | 10 | 10 | 10 |
| Total | 152 | 112 | 73 | 37 | 31 | 200 | 190 | 200 | 200 | 200 | 200 |

Table

| | Ground N | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 1 | 5 | 10 | 25 | 50 | $\infty$ | 1 | 5 | 10 | 25 | 50 |
| 01 | 3 | 10 | 10 | 10 | 10 | 10 | 0 | 10 | 10 | 10 | 10 |
| 02 | 8 | 10 | 10 | 10 | 10 | 10 | 7 | 10 | 10 | 10 | 10 |
| 03 | 8 | 10 | 10 | 9 | 4 | 10 | 10 | 10 | 10 | 10 | 10 |
| 04 | 9 | 0 | 0 | 0 | 0 | 10 | 9 | 10 | 10 | 10 | 10 |
| 05 | 10 | 6 | 0 | 0 | 0 | 10 | 8 | 8 | 10 | 10 | 10 |
| 06 | 1 | 0 | 0 | 0 | 0 | 4 | 1 | 3 | 2 | 6 | 5 |
| Total | 39 | 36 | 30 | 29 | 24 | 54 | 35 | 51 | 52 | 56 | 55 |

Door

Table 4.1: This table shows how often a plan was found for each task and problem instance. Ten runs with different random seeds were executed for each task, problem, algorithm and choice of N.

**Discussion** Table 4.1 shows the coverage for all settings, problem instances and both algorithms depending on N. As expected coverage goes down when the number of objects is higher and the tasks are more complex. The Table scenario is the easiest as it does not require regrasping. Here all 20 instances are solved by Ground Single Reinsert. The lower shelf in the Door task only fits up to six objects and thus becomes very constrained with six objects. The Shelf and Fridge tasks give the planner increasingly more options to put objects. This results in less solved tasks overall for the larger Fridge task as the search space gets larger. The scaling of computation time with problem size is visualized in Figure 4.6. [4]

If we consider the coverage in Table 4.1 for Ground N with larger N values ($N > 5$), we observe that in most instances an increasing N for Ground N leads to more failures. We can also see this in the plot of computation times in Figure 4.7. In the top row we compare individual runs with both algorithms for the same N value. Different N values are colored from the smallest in blue to the largest in red. Entries are farther away from the diagonal with increasing N. This means that the performance in comparison to Ground Single Reinsert decreases with increasing N. Some entries noted as "no plan" are drawn off limits. In these cases the reason in not a timeout, but a completely explored search space within the time limit. However, this happens

[4]Plots in this chapter were created with Matplotlib (Hunter, 2007).

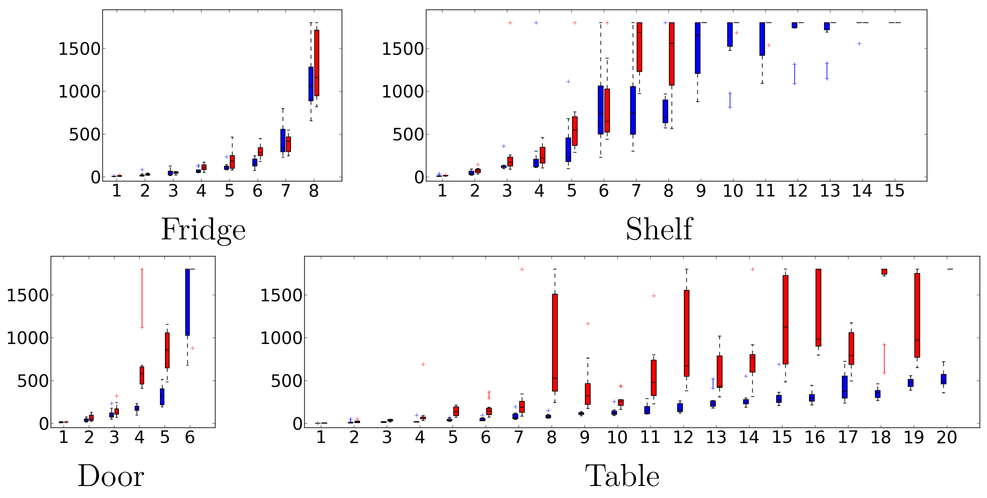


Figure 4.6: This figure illustrates the scaling of computation time in seconds with problem size. Ground Single Reinsert (blue) with $N = \infty$ is compared to the oracle choice for Ground N (red).

for small N values, here drawn in light blue. Which value of N is best depends on the problem size. For smaller problems, e.g., for Fridge problem 03 $N = 25$ still works, for problem 05 $N = 5$ is best, and from 06 on $N = 1$ is best. We see two opposing effects here. A smaller N crops the search space and thus might prune away the solution. On the other hand a smaller N leads to a faster search, so that as long as the solution path is preserved the chance to run into a timeout is lower. The more complex a task is, the more important it is to limit the branching factor. The main issue here is that we cannot determine the best N before planning time. This is dependent on the complexity of the task.

In contrast Ground Single Reinsert behaves differently. We again see that too small values of N prune away solution paths. However, there is no negative effect for choosing larger values of N. In fact $N = \infty$ still leads to competitive performance. The reason is that Ground Single Reinsert does not produce all successors, but determines by itself, when more are needed. Thus the limit of N is often not reached. This already is a beneficial property, but more importantly the performance in comparison to Ground N is better not only for the same values of N, but also when using $N = \infty$.

To emphasize this behavior the bottom row in the plots for the computation times (Figure 4.7) also contains a comparison of Ground Single Reinsert with $N = \infty$ to the oracle algorithm that chooses the best run among different N values for Ground N. Even against this oracle Ground Single Reinsert performs well. Note also that especially for the larger problems of the Shelf and Table tasks many runs for Ground N result in a timeout for any N, where Ground Single Reinsert ($N = \infty$) still finds a solution. In the Table task the performance difference is most noticeable, although it

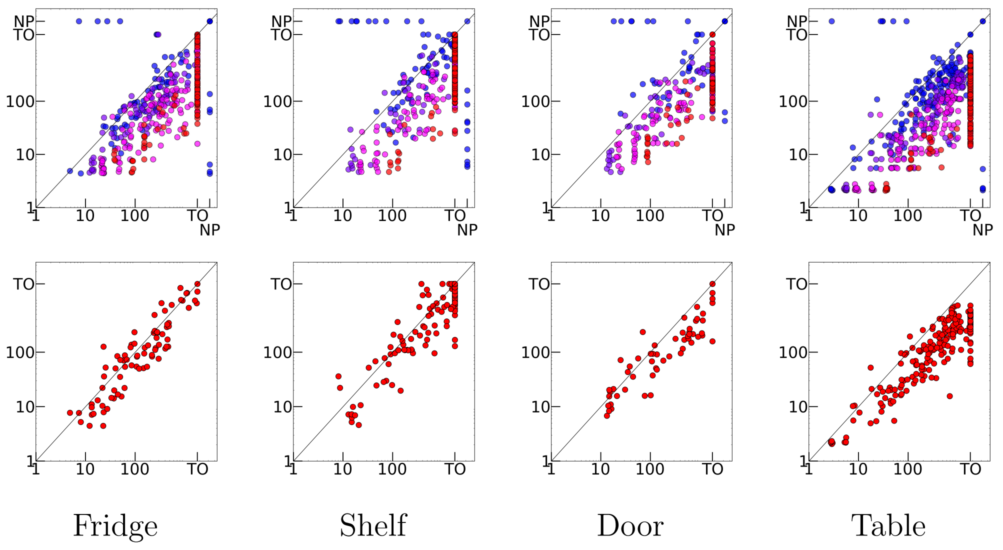


Figure 4.7: This figure plots the computation time in seconds for sampling-based successor generation. Ground N is plotted on the x-axis and Ground Single Reinsert on the y-axis. The top row compares individual runs for the same N. The bottom row shows Ground Single Reinsert ($N = \infty$) and the oracle choice of Ground N. Entries at *TO* are timeouts. Entries drawn at *NP* signal that no plan could be found, because the search space was completely explored before the timeout.

is the easiest task. The different behavior of both algorithms is similar to the difference between breadth-first and depth-first search. Ground N always produces all successors, while Ground Single Reinsert only produces one and then continues to explore. This leads to a deeper exploration of the search space, which is advantageous for the Table and Shelf tasks with many objects. As a result of this plan lengths (see Figure 4.8)

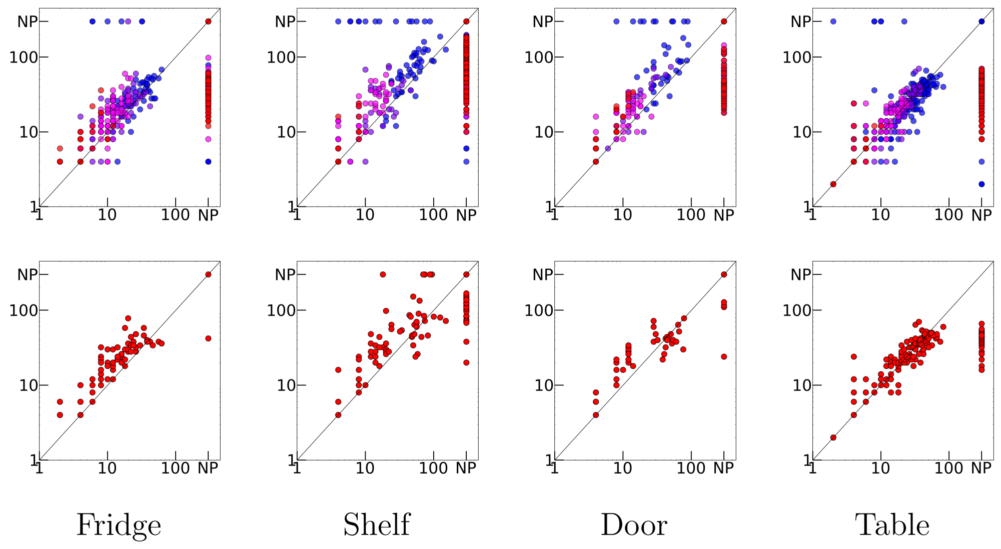


Figure 4.8: This figure plots the plan lengths for sampling-based successor generation. Ground N is plotted on the x-axis and Ground Single Reinsert on the y-axis. The top row compares individual runs for the same N. The bottom row shows Ground Single Reinsert ($N = \infty$) and the oracle choice of Ground N. Entries drawn at *NP* signal that no plan could be found.

tend to be higher for Ground Single Reinsert. Here "no plan" entries for plan lengths do not represent an actual plan, but are shown for completeness. The effect is most prominent for shorter plans and decreases when the plans are longer. It should also be noted that there are more "no plan" entries for Ground N on the larger problems especially in the Shelf and Table tasks.

## 4.4.2 Discretization-based successor generation

In this experiment we investigate successor generation based on a fixed discretization on the same scenario as before. We use the same domain and tasks shown in Figure 4.5 and focus on the differences to the previous experiment. We discretize the possible positions on each surface with 10, 25 and 50 centimeters and for directions we considered 45 and 90 degree steps. We denote a specific discretization by position and direction as, e.g., 25–90. We set $N = \infty$ for both algorithms. Here, which discretization is used defines how many successors there are and thus a finer discretization producing more successors has the same effect as choosing a larger N in the previous experiment. Courser discretizations, however, might not have a solution. As the discretization is deterministic no sampling is included and we executed each task once for all possible discretizations and both algorithms.

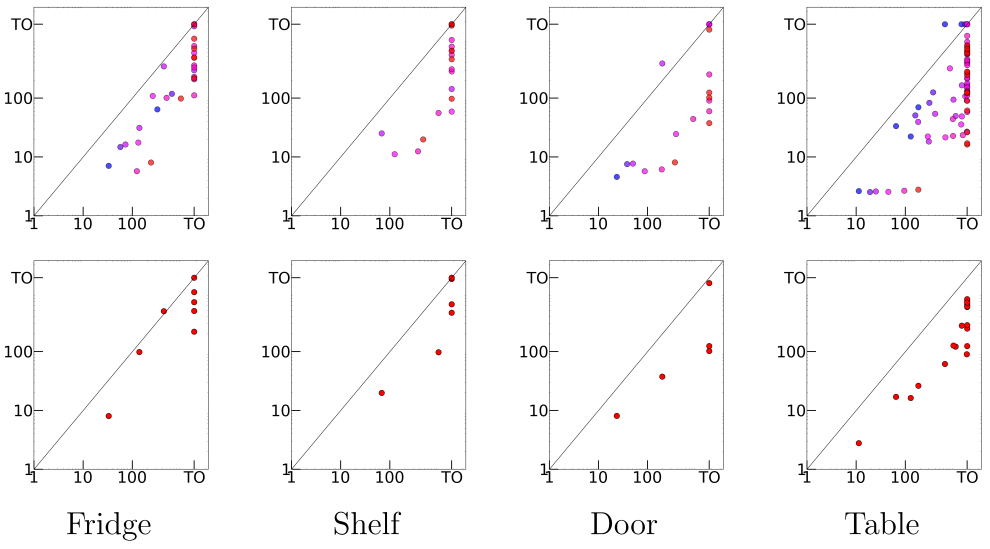


Figure 4.9: This figure plots the computation time in seconds when using a fixed discretization. Ground N is plotted on the x-axis and Ground Single Reinsert on the y-axis. The top row compares individual runs for the same discretization. The bottom row shows Ground Single Reinsert and the oracle choice of Ground N. Entries at *TO* are timeouts.

As a first note, given that the number of successors here is fixed, although large, it is theoretically possible to fully ground the planning task before search instead of using grounding modules. As a preliminary experiment we tried to do so, but ran into memory limits even for smaller tasks and coarse discretizations. Thus, planning tasks with large numbers of successors already benefit from using grounding modules even when just Ground N is used. Keep in mind that even the roughest discretization of 50 cm and 90 degrees produces dozens of successors for placing an object. Therefore we only consider results when using grounding modules with Ground N and Ground Single Reinsert.

**Discussion** We record the same information as in the previous experiment. Table 4.2 shows the computation time for all settings, problem instances and both algorithms depending on the discretization. If we consider how many tasks were solved, we see that the coverage in comparison to sampling in the previous experiment is worse,

| | Ground N | | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 |
| 01 | **23.6** | 38.2 | 50.2 | 87.3 | 194.1 | 360.0 | **4.6** | 7.6 | 7.7 | 5.7 | 6.1 | 8.1 |
| 02 | - | - | **199.0** | 379.2 | 848.2 | - | - | - | 388.0 | **24.5** | 44.2 | 37.6 |
| 03 | - | - | - | - | - | - | - | - | - | - | **59.9** | 124.2 |
| 04 | - | - | - | - | - | - | - | - | - | - | **90.8** | 102.3 |
| 05 | - | - | - | - | - | - | - | - | - | - | **252.4** | 1459.7 |

Door

| | Ground N | | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 |
| 01 | - | - | **68.1** | 124.9 | 372.3 | 474.1 | - | - | 25.1 | **11.2** | 12.5 | 19.9 |
| 02 | - | - | - | - | **973.4** | - | - | - | 142.9 | 59.0 | **55.9** | 97.5 |
| 03 | - | - | - | - | - | - | - | - | - | - | **284.0** | 1711.9 |
| 04 | - | - | - | - | - | - | - | - | - | - | **309.8** | 457.5 |
| 05 | - | - | - | - | - | - | - | - | - | - | 637.6 | **637.1** |
| 06 | - | - | - | - | - | - | - | - | - | - | **750.1** | - |
| 07 | - | - | - | - | - | - | - | - | - | - | **543.2** | - |
| 08 | - | - | - | - | - | - | - | - | - | - | **974.6** | - |

Shelf

| | Ground N | | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 |
| 01 | **33.0** | 56.8 | 72.3 | 131.9 | 123.1 | 238.5 | 7.1 | 14.8 | 16.3 | 17.6 | **5.7** | 8.0 |
| 02 | 320.9 | 634.0 | **138.4** | 258.7 | 494.5 | 965.4 | 64.4 | 118.3 | **31.2** | 109.0 | 101.2 | 98.4 |
| 03 | - | - | **435.0** | - | - | - | - | - | 346.3 | 230.2 | **111.6** | 486.0 |
| 04 | - | - | - | - | - | - | - | - | 297.5 | - | **216.8** | 218.0 |
| 05 | - | - | - | - | - | - | - | - | - | - | **359.9** | 490.6 |
| 06 | - | - | - | - | - | - | - | - | - | 588.2 | **209.0** | 1024.8 |
| 07 | - | - | - | - | - | - | - | - | 1662.6 | - | **331.2** | 693.1 |
| 08 | - | - | - | - | - | - | - | - | - | - | **767.1** | - |

Fridge

| | Ground N | | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 | 50-90 | 50-45 | 25-90 | 25-45 | 10-90 | 10-45 |
| 01 | **11.3** | 19.1 | 25.1 | 45.0 | 94.9 | 182.9 | 2.6 | **2.5** | 2.6 | 2.6 | 2.7 | 2.8 |
| 02 | **128.3** | 158.7 | 181.5 | 285.6 | 928.4 | - | 22.3 | 50.9 | 39.4 | 22.3 | 22.8 | **16.3** |
| 03 | **64.4** | 306.5 | 401.4 | 801.3 | 919.4 | - | 33.5 | 83.2 | 54.1 | 320.3 | 44.1 | **17.0** |
| 04 | **183.5** | 365.8 | 298.9 | 648.7 | 1466.6 | - | 69.9 | 125.2 | **18.3** | 21.5 | 23.5 | 26.3 |
| 05 | **635.5** | 1632.4 | 1781.6 | 1359.9 | - | - | - | - | **26.9** | 35.7 | 58.0 | 61.6 |
| 06 | 1380.6 | - | **1049.3** | 1402.0 | - | - | - | - | 49.7 | **49.0** | 130.4 | 121.4 |
| 07 | - | - | **947.4** | 1647.6 | - | - | - | - | **94.3** | 107.2 | 153.7 | 126.5 |
| 08 | - | - | **1770.5** | - | - | - | - | - | 90.8 | 178.3 | 111.1 | **90.1** |
| 09 | - | - | - | - | - | - | - | - | 167.0 | 150.0 | 133.1 | **124.1** |
| 10 | - | - | **1398.6** | - | - | - | - | - | **165.1** | 214.3 | 407.2 | 275.9 |
| 11 | - | - | - | - | - | - | - | - | - | - | **221.5** | 246.3 |
| 12 | - | - | - | - | - | - | - | - | - | - | **256.6** | 281.8 |
| 13 | - | - | - | - | - | - | - | - | - | - | **391.7** | 598.9 |
| 14 | - | - | - | - | - | - | - | - | - | - | **551.2** | 608.2 |
| 15 | - | - | - | - | - | - | - | - | - | - | 905.4 | **572.6** |
| 16 | - | - | - | - | - | - | - | - | - | - | **457.5** | 721.4 |
| 17 | - | - | - | - | - | - | - | - | - | - | 785.2 | **646.9** |
| 18 | - | - | - | - | - | - | - | - | - | - | **432.8** | 582.2 |
| 19 | - | - | - | - | - | - | - | - | - | - | 1144.9 | **778.2** |
| 20 | - | - | - | - | - | - | - | - | - | - | **364.3** | 682.4 |

Table

Table 4.2: This table shows the computation times for the different settings when fixed discretizations are used. Timeouts are marked by "-". Best times among the same discretization between Ground N and Ground Single Reinsert are shown light gray. Best overall times for the same task are displayed dark gray. Best times for different discretizations for the same algorithm are bold.

especially for Ground N. The reason for this is that successors here are produced deterministically from the discretization. Thus, when the search generates successors for different states it will always generate the same successors in contrast to sampling, where successors are produced randomly. This could be an advantage as when good successors are generated, they will be generated again. On the other hand bad successors will also be generated repeatedly. The results indicate that—at least for such challenging tasks, where a majority of the possible successors are not applicable—a sampling-based approach is better. This affects Ground N more than Ground Single Reinsert as Ground Single Reinsert does not need to produce all possibly bad successors for a state. Instead it expands other states before returning. One undesirable property of Ground N is that the discretization that results in the most solved problems is dependent on the task. Coarser discretizations are faster to explore, but might not be solvable. Finer discretizations take too long to search. Ground Single Reinsert handles finer discretizations very well. Almost all tasks are solved even when the finest discretization is used. Ground Single Reinsert with 10-90 solves all tasks in all scenarios.

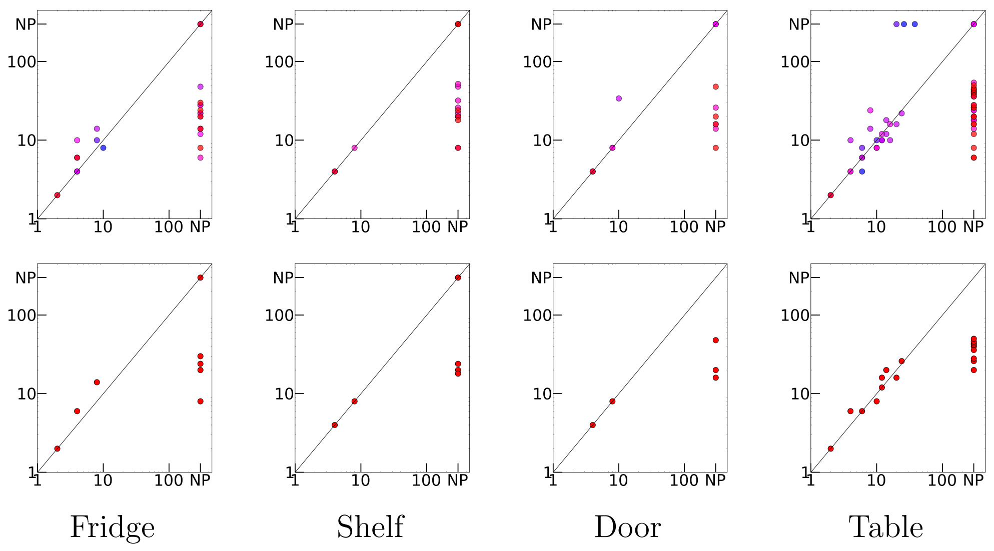


Figure 4.10: This figure plots the plan lengths when using a fixed discretization. Ground N is plotted on the x-axis and Ground Single Reinsert on the y-axis. The top row compares individual runs for the same discretization. The bottom row shows Ground Single Reinsert and the oracle choice of Ground N. Entries drawn at *NP* signal that no plan could be found.

We compare computation times in Figure 4.9. The oracle algorithm for Ground N here chooses the best run among all discretizations. This is compared with Ground Single Reinsert using the finest discretization available, i.e., all possible successors.

Different discretizations are colored from the coarsest in blue to the finest in red. Here, even in the comparison to the oracle choice the same problem instance is almost always solved faster with Ground Single Reinsert. Comparing plan lengths in Figure 4.10 the advantage for Ground N is not as pronounced as for the sampling case. One reason for this is that many tasks are just not solved by Ground N. Overall we can say that for complex tasks, where not many successors are applicable, and especially, when they are produced deterministically, Ground Single Reinsert is superior.

### 4.4.3 Module-Abstracted Search Heuristics

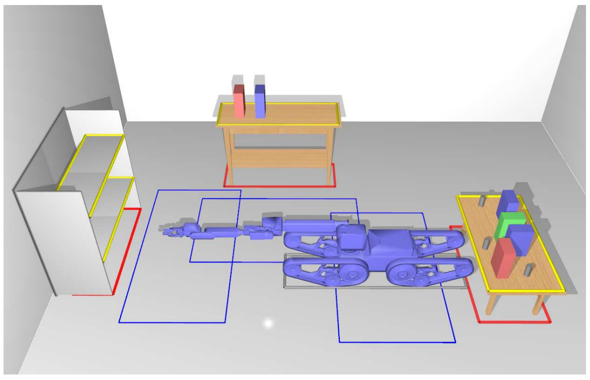


Figure 4.11: This figure shows the scenario used in the third experiment. Yellow rectangles show the surfaces that the robot can put objects on, i.e., the tables and the shelves on the left.

Search guidance heuristics have to provide a good balance between how expensive they are to compute and how informed they are. The module-abstracted heuristic actually performs classical planning albeit on a task that is simpler than the full domain with geometric information. As a base line we use the context-enhanced-additive heuristic denoted as CEA (Eyerich et al., 2009), which is the default for our planner TFD/M and was used in the previous experiments. The goal of this experiment is to investigate the performance of module-abstracted search heuristics in comparison to this base line.

We use a mobile manipulation scenario shown in Figure 4.11 with two grounding modules. The `pick-up` and `put-down` operators are the same as before. Placements are chosen with the sampling-based grounding module. In addition a `drive-base` operator moves the robot base to another pose (i.e., position and orientation) near a table or cupboard. The poses are sampled uniformly in the blue rectangles, where the robot base (gray rectangle) must not intersect any object's surface area (red rectangles). The robot pose is defined to be the point at the manipulator base, e.g., in Figure 4.11 the

| | Ground N | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 1 | 5 | 10 | 25 | 50 | $\infty$ | 1 | 5 | 10 | 25 | 50 |
| 01 | 14 | 20 | 20 | 20 | 20 | 20 | 15 | 20 | 20 | 20 | 20 |
| 02 | 19 | 20 | 20 | 20 | 17 | 20 | 16 | 20 | 20 | 20 | 20 |
| 03 | 9 | 0 | 0 | 0 | 0 | 3 | 10 | 8 | 7 | 9 | 7 |
| 04 | 12 | 0 | 0 | 0 | 0 | 13 | 10 | 16 | 15 | 17 | 15 |
| 05 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| 06 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 07 | 6 | 0 | 0 | 0 | 0 | 2 | 4 | 7 | 7 | 6 | 6 |
| 08 | 4 | 0 | 0 | 0 | 0 | 5 | 7 | 9 | 8 | 7 | 3 |
| 09 | 1 | 0 | 0 | 0 | 0 | 1 | 4 | 2 | 6 | 6 | 4 |
| 10 | 6 | 0 | 0 | 0 | 0 | 5 | 10 | 3 | 7 | 9 | 7 |
| Total | 73 | 40 | 40 | 40 | 37 | 69 | 78 | 85 | 90 | 94 | 82 |

CEA

| | Ground N | | | | | Ground Single Reinsert | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | 1 | 5 | 10 | 25 | 50 | $\infty$ | 1 | 5 | 10 | 25 | 50 |
| 01 | 13 | 20 | 20 | 20 | 20 | 20 | 18 | 20 | 20 | 20 | 20 |
| 02 | 15 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 03 | 16 | 20 | 20 | 20 | 20 | 19 | 16 | 19 | 19 | 19 | 19 |
| 04 | 6 | 20 | 20 | 20 | 20 | 9 | 10 | 12 | 7 | 13 | 8 |
| 05 | 11 | 9 | 15 | 8 | 5 | 16 | 16 | 13 | 13 | 15 | 12 |
| 06 | 10 | 6 | 4 | 0 | 0 | 11 | 11 | 11 | 10 | 10 | 9 |
| 07 | 13 | 12 | 8 | 0 | 0 | 11 | 11 | 10 | 12 | 13 | 13 |
| 08 | 15 | 13 | 6 | 0 | 0 | 18 | 20 | 18 | 17 | 18 | 14 |
| 09 | 10 | 17 | 6 | 0 | 0 | 18 | 14 | 16 | 19 | 15 | 13 |
| 10 | 9 | 9 | 1 | 0 | 0 | 12 | 12 | 13 | 15 | 14 | 14 |
| Total | 118 | 146 | 120 | 88 | 85 | 154 | 148 | 152 | 152 | 157 | 142 |

Module-Abstracted

Table 4.3: This table shows the number of solved runs out of 20 for each problem using the context-enhance-additive heuristic (CEA) in comparison to the module-abstracted heuristic.

robot is positioned at the very left of the rightmost rectangle. There is one additional operator `arm-to-drive-pose` that makes sure that the robot does not navigate with an extended arm. This domain is more challenging as the previous ones as the ability to position or grasp an object now not only depends on the grasp and (chosen) object pose, but also on the robot's pose that directly influences the reachability. Here, well informed heuristic guidance is even more important than in the previous scenarios.

The task is to place all boxes in the shelf on the left and to move four glasses (gray cylinders) from the right table to the center table. Besides the mobile manipulation scenario itself the challenge for the planner is to determine a valid sequence. Until the two boxes on the center table are moved it is harder to determine collision free poses for placing the glasses. Also, if the boxes and glasses on the right table are graspable depends on the order, in which they are removed from the table and the robot's pose. For example, one of the glasses here is placed behind the green box and thus is not graspable until that is removed. Problems are numbered with an increasing number of objects. Problem #01 and #02 only address the two boxes on the center table. Problems #03 - #06 also include the four boxes on the right table and #07 - #10 additionally include the glasses on the right table. For each of the 10 problems we executed Ground N and Ground Single Reinsert with values of $1, 5, 10, 25, 50$ and if applicable $\infty$ for N. Each run was executed with the context-enhanced-additive heuristic and the module-abstracted heuristic on 20 random seeds each. The tasks were executed on a single core of an Intel Core-i7-3930K. We set the memory limit to 4 GB and a timeout of 1800 seconds.

**Discussion** Again we record the time until the first plan was found and its length or note a failure if no plan was found. Table 4.3 shows the coverage for CEA and the module-abstracted heuristic. The module-abstracted heuristic solves more tasks in total than CEA for any search algorithm and N. With the exception of Ground Single Reinsert in problem #04 this tendency also shows for individual problems. The behavior of problems #03 and #04 is different, which can also be seen as Ground N

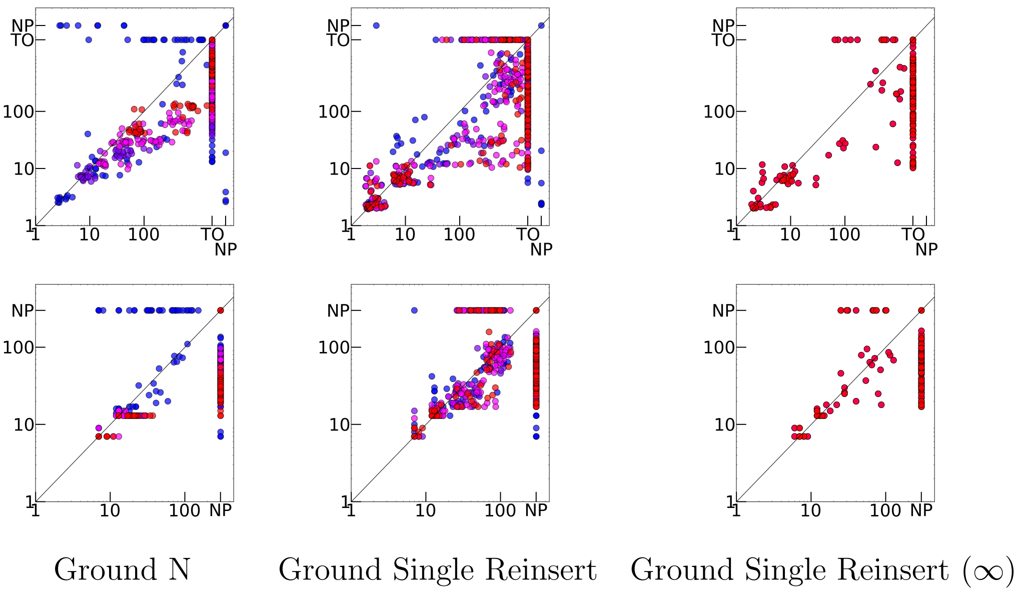


Figure 4.12: This figure compares the module-abstracted heuristic with the context-enhanced-additive (CEA) heuristic. CEA is plotted on the x-axis and the module-abstracted heuristic on the y-axis. The top row shows computation times in seconds, the bottom row plots plan lengths. Entries at *TO* are timeouts. Entries drawn at *NP* signal that no plan could be found.

here has more coverage than Ground Single Reinsert. These are the problems, where two boxes need to be moved from the right table in addition to the two boxes on the center table. The only successful plan here is to find a good robot position that allows to grasp each box directly as other objects cannot be repositioned. This is unfavorable to the more explorative nature of Ground Single Reinsert or the more goal-directed behavior in the module-abstracted heuristic as in this case enumerating all possibilities early on is better.

We plot computation times and plan lengths for the same N value and random seed in Figure 4.12 comparing CEA and the module-abstracted heuristic for each search algorithm. For Ground Single Reinsert fixed N values and $N = \infty$ are displayed separately. Again, different N values are colored from the smallest in blue to the largest in red and "no plan" entries are shown when the search space was explored before the timeout. For smaller problems that are solved in about 10 seconds or less neither heuristic shows better planning times than the other. On more difficult problems there is a considerable advantage for the module-abstracted heuristic. Here, also the larger number of unsolved problems for CEA is clearly visible. Plan lengths shown in the bottom row of Figure 4.12 are comparable for both heuristics. Overall the guidance provided by module-abstracted heuristics for these mobile manipulation tasks leads to better coverage and faster solution times than CEA.

## 4.5 Conclusion

We addressed planning tasks with an infinite branching factor. Such problems arise in real-world domains like robotics when geometric choices are made. We formulated these kind of tasks as *partially groundable planning tasks*. Classical search algorithms cannot be applied any more as they do not terminate with an infinite number of successors. We presented two generic search algorithms for planning tasks with an unlimited branching factor. Ground N extends classical planning to infinite successors by setting a fixed limit for each state. As this requires to pre-set a large enough limit to reliably find plans, we introduced Ground Single Reinsert that dynamically makes the decision to branch or explore.

We evaluated both algorithms on robot manipulation scenarios. Our evaluation shows that Ground Single Reinsert provides better coverage and leads to faster planning times than Ground N, even when compared to an oracle choice of N. The motivation here was to relieve a user from making this choice and leave this to the algorithm. Ground Single Reinsert does have another advantage. The number of necessary branches is determined on a per state basis as part of the search in comparison to a fixed limit. Therefore even on a fixed discretization we have shown favorable performance for Ground Single Reinsert. With uniform sampling we only considered a simple mechanism for successor generation. More advanced methods for producing geometric choices, e.g., using a capability map for robot placement (Leidner et al., 2014) easily fit in our framework and thus can be integrated to lead to further improvements.

Finally, we evaluated module-abstracted heuristics for integrated task and motion planning that search on the symbolic abstraction of the full geometric state. Our results show that this heuristic works well for mobile manipulation tasks. Search guidance is an important topic for symbolic-geometric planning. Techniques from motion planning like RRT-planning have been applied to symbolic planning (Alczar et al., 2011) and thus might lead to even better search mechanisms in the future.

# Chapter 5

# Real-World Applications using Continual Planning

In the previous chapters we showed how we plan for robotics tasks. A robotic system must also integrate the planner to achieve goal-directed behavior. This means that interfaces to the robot's action and perception capabilities are provided. Perception is necessary as the current state of the world represents the initial state for the planner. The action interfaces make it possible to actually execute a plan. The planning algorithms that we described are based on the semantics of classical planning, i.e., they assume that the inital state completely and correctly describes the current world state, that each action execution deterministically leads to the stated effects, and that the current state in the world does not change from anything else, but the robot's actions. These assumptions are clearly violated for a robot acting in the real world.

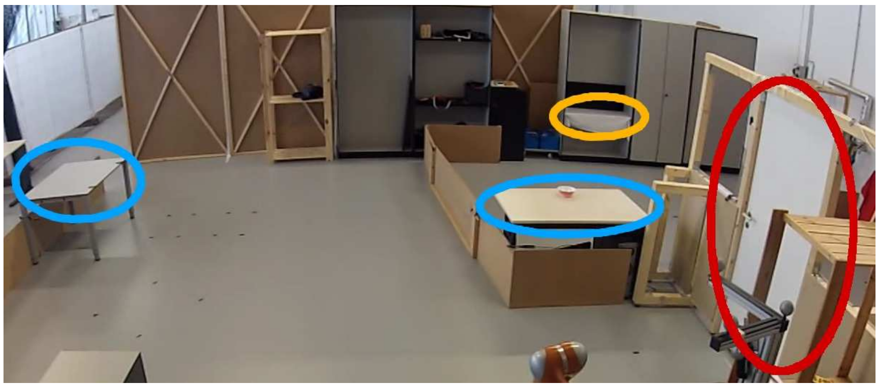


Figure 5.1: Overview of an experimental environment for a household scenario. The test scenario contains two rooms separated by a door (red), tables (blue) and a shelf (yellow) in the back.

Consider the example of a household domain in Figure 5.1. We want a robot to tidy up by bringing all objects on the tables into the shelf and wiping all tables with a sponge. This is all the information that is necessary to state this task. In particular one should not need to specify, which objects there are or where they are located. The robot should determine this by planning for appropriate sensing actions. Furthermore, an action execution might fail, e.g., a `pick-up` action not grasping an object leading to a different state than predicted. Also external events can happen: For example, a human closes the door after walking through it. A robot must at least to some degree be able to deal with all these properties of real-world tasks.

We follow the approach of continual planning. The planner is embedded in a closed perception-reasoning-action loop. Instead of blindly following a plan, the current state is estimated after each action and replanning is triggered if necessary. This procedure continues until the goal is reached or no plan can be found. Replanning mitigates the shortcomings of the robot's skills or of the expressiveness of the planning formalism always adapting the current plan to the observed situation.

The principle of continual planning is also applied in other robotic systems. How well such a system performs depends on various factors, for example, the robotics algorithms like manipulation or navigation skills, the expressiveness of the planning algorithm, or the domain formulation itself. In this chapter we formalize the properties of real-world planning tasks and show how these can be addressed by continual planning with an embedded classical planner. Here we look in particular at identifying the underlying assumptions and as a result of this the limitations of such a system.

Another aspect of real-world systems is speed. Planning with semantic attachments is a time consuming process. Even classical planning is PSPACE-complete and more expressive formalisms for partially-observable, nondeterministic domains are 2-EXP-complete (Rintanen, 2004). The results from previous chapters have shown that reasonable computation times are possible in practice. However, faster planning times are beneficial for real-world systems. We introduce two techniques that specifically address planning tasks with semantic attachments. First, a novel lazy evaluation algorithm prevents costly geometric computations as much as possible without losing soundness. Second, we investigate a variety of caching techniques that answer requests to a semantic attachment by previously computed results within the current plan computation, but also across different calls to the planner.

One goal of our integrated planning system was to provide a domain-independent planner with semantic attachments. This means that the same planner can be used in different problem domains. To demonstrate the versatility of our approach we apply our continual planning system to various different tasks and robots. In addition the application to these scenarios also demonstrates that our assumptions are realistic and thus allow the system to work in the real-world. We particularly focus on a complex tidy up example with the PR2 mobile manipulation robot. Here, we also evaluate the performance of the aforementioned lazy evaluation and caching techniques.

The remainder of this chapter is structured as follows. In the next section we will formalize real-world planning tasks. In Section 5.2 we review other approaches and formalisms. Section 5.3 illustrates our continual planning architecture and we show how properties of real-world tasks are addressed in Section 5.4. Section 5.5 introduces efficient planning techniques. The application scenarios and evaluation is presented in

Section 5.6 before we conclude in Section 5.7.

## 5.1 Real-World Planning Tasks

In order to control a robot with the help of a planning system, an adequate planner would be one that is able to deal with:

- open domains (with an unlimited number of objects)
- uncertain initial states
- sensing operations with partial observability
- nondeterministic (or probabilistic) effects of actions
- exogenous events

These properties are certainly present in common robotics domains such as service robotics, which is our main target application. One example is a simple task like "clean the dinner table". This is an *open domain* as the robot does not even know which objects to clean. If it doesn't know which objects there are, it obviously can't be sure about their *initial state*. It has to find out by using *sensing actions*. However, usually the robot's sensors will only detect objects in front of the robot. Thus the state is only *partially observable*. Failure to grasp an object is just one possible *nondeterministic* outcome of a grasping action. Finally someone might put a dirty glass on the table while the robot is already cleaning. This is an *exogenous event* as it is not directly caused by the robot.

Although this seems like a hard problem from a planning point of view, this is only an issue if we want to provide strong guarantees to our robot behavior. To what extend such guarantees are necessary is dependent on the application domain. For a household robot we can be more lenient than for a Mars lander or a medical robot. We concur with the assessment of Kaelbling and Lozano-Pérez (2011) that "there are few catastrophic or entirely irreversible outcomes". We believe that a good measure for the simplifications and resulting robot behavior is under what assumptions a human would address and solve such tasks.

It seems as a bit of an overkill that a robot should be able reason about multiple possible situations. A service robot certainly is not supposed to solve whodunit puzzles or diagnose the failure of a dish washer. Furthermore, it also seems a bit over-cautious to plan for all contingencies in advance, given that a household domain (as many others) is quite forgiving concerning wrong choices or guesses. Also humans plan in most cases without considering all possibilities. Requiring safe plans under all possible outcomes might even hinder the robot when trying to move any object. There is no plan that ensures that a glass can be brought into the kitchen without breaking it.

Almost everyone has done that at some time. In any case planners that consider nondeterministic effects can only do so within the effects that have been described in the planning input.

So, as many others have proposed and done, we use a *continual planning* approach, where we plan for one way to solve the planning problem at hand, and replan if anything does not work out according to plan. We now discuss related work for solving real-world planning problems before we explain our procedure. Afterwards, we will address two important questions:

1. How to compile a particular feature in the original planning task description away?

2. Under what conditions can we expect that the approach guarantees that we can reach the goal?

## 5.2 Related Work

Robot planning tasks could be, for instance described using the planning language NPPDL (Bertoli et al., 2003), an extension of PDDL that is able to deal with uncertainty, nondeterminism, partial observability and sensing, but not open domains. It should be noted that this language does not support an explicit knowledge or belief modality. Rather all preconditions, effect conditions, and goal specifications are assumed to be implicitly in the scope of a modal belief operator.

A number of planners have been developed that deal with nondeterminism, partial observability, and sensing. For instance the myND planner solves fully observable nondeterministic planning tasks using LAO* search (Mattmüller et al., 2010; Hansen and Zilberstein, 2001), the planning system MBP, which is based on BDDs (Bertoli et al., 2001), the planning systems POND and CAltAlt (Bryce et al., 2006; Bryce, 2006), a lazy approach to representing belief sets (Hoffmann and Brafman, 2005), and a compilation approach to solve nondeterministic planning problems using classical planning (Kuter et al., 2008). However, given the size of the search space in our domain, it is questionable from the performance data in the paper by Bryce et al. (2006), if such planners will be able to generate plans in reasonable time especially when geometric computations are to be included.

We believe, planning approaches that do not try to solve the entire problem offline are more adequate for robotics. One approach is, for example, to interleave planning for nondeterministic partially observable domains with execution as proposed by Bertoli et al. (2004). While often there is a simple plan-execute-monitor loop, there are also other approaches more tightly integrating planning and execution (Ambros-Ingerson and Steel, 1988; Brenner and Nebel, 2009; Knoblock, 1995). Konecny et al. (2014) specifically address the problem of execution monitoring for robotics with causal, temporal, and categorical models. An interesting aspect of their approach is that they produce lifted plans, so that unpredicted events do not necessarily cause execution failures. For example, if a mug to be grasped was removed, but another equivalent is detected at the same location the execution system switches to use the other mug. Our monitoring system also does not fail on any unpredicted event albeit relying on another principle (Dornhege and Hertle, 2013).

Under which conditions such approaches are feasible and how expensive the verification of such a condition would be has not been investigated thoroughly, though. This is one main point of the work presented in this chapter. We not only describe the challenges in real-world planning tasks and state how we simplify these to make them applicable to robotics, but also discuss our assumptions (Nebel et al., 2013).

State of the art systems for high-level robot control often require an explicit specification of the desired robot behavior, for example in a state machine (Bohren et al., 2011). In recent years, robot control by automatic planning is considered again as a realistic means to achieve autonomy. The reason for that is that automatic planning has become much more efficient, as demonstrated by recent events such as the *International Planning Competitions*.

Current approaches can deal with incomplete knowledge and beliefs while integrating geometric reasoning into the planning process. Notably, Kaelbling and Lozano-Pérez (2013) have developed a robot planning system that integrates task and motion planning in belief space. They also use a replanning approach which accounts for wrong choices in the planning process and implicitly for execution failures. To prevent becoming stuck in dead ends they require the property that a domain should be reversible. Another interesting approach related to our work is the work by Gaschler et al. (2013), which demonstrates how Petrick's belief-space planner can be used in a real world environment.

Efficient planning techniques specifically for integrated task and motion planning have also been addressed as computing motion plans as part of a symbolic plan is a costly process. Srivastava et al. (2013) interleave task and motion planning and feed symbolic facts determined by motion planners back into the classical planner to avoid committing early to instantiations of continuous operators. Another way is to *plan in the now* (Kaelbling and Lozano-Pérez, 2011) using abstract versions of operators for plan steps far enough in the future. Here, Kaelbling and Lozano-Pérez use a hierarchical regression planner and once the prefix of the computed plan is refined to the lowest level of the hierarchy they start execution immediately. Other approaches focus on reuse of solutions: In their combined task and motion planner Wolfe et al. (2010) use subtask-specific irrelevance to reuse trajectories that have been computed before. This concept is similar to what we call partial state caching, which we extend even further to subsumption caching (Dornhege et al., 2013a).

In the context of classical planning Eyerich et al. investigated subsumptions of planning operators (Eyerich et al., 2008), although without connecting this concept to geometric constraints. Related to our lazy module evaluation is a technique from classical planning. Heuristic estimates can be approximated by the heuristic value of the parent state, so that computationally expensive heuristics are never computed for states in the search queue (Richter and Helmert, 2009). We use a similar concept to avoid geometric computations for queued states.

## 5.3 Continual Planning

The architecture of our continual planning system is shown in Figure 5.2. The left side shows the generic implementation of the closed loop system. In each iteration the current state is merged with observations of the world. Then, the current plan (initially empty) is monitored, i.e., it is determined, if executing the plan leads to the goal, given the current state. If that is not the case, a new plan is produced. The first action of the plan is executed and removed from the current plan and the loop continues until the goal is reached or no plan can be found.

We use our planner TFD/M to perform both monitoring and planning. For monitoring, in addition to the domain description and the current state we also pass the current plan. In that case the planner performs a search that is at each step restricted to apply only the operator listed in the plan if possible. If the goal is reached, monitoring is successful. This serves multiple purposes. First, as we use the exact same algorithms, we guarantee that the results of monitoring and planning agree. In particular when semantic attachments are used we therefore can be sure that the next action is still applicable. We also gain a generic goal test: If the empty plan fulfills the goal, we are in a goal state. Finally it is important to note that we only compute if the current plan still leads to a goal state, but not if the exact same state progression takes place that was used to produce this plan initially. This is important for real-world planning tasks as monitoring will often fail otherwise. For example, the perception might discover a new object, thus leading to a different state space. If this object is irrelevant for the robot to reach its goal, we can still follow the plan. Another example is that the observed pose for an object changes slightly without preventing any motion plan from being executable.

The continual planning loop is implemented independent from a specific robotic system. While semantic attachments provide a domain-independent way to integrate robot-specific semantics into the planning process we also need to integrate robot-dependent perception and action execution into our system. We therefore provide two interfaces: One to estimate and update the symbolic state from real-world observations and another to execute symbolic actions. Both interfaces are provided by a plugin-based architecture. Multiple *StateEstimator* plugins can be defined that retrieve different aspects of the current state from the world. Each action in the domain description requires a corresponding *ActionExecutor* plugin. Besides actually executing the actions, *ActionExecutor* plugins also define the effects of an action updating the state depending on the actual action outcome. For example, failing to grasp an object will not blindly set the `grasped` predicate to true as specified in the domain.

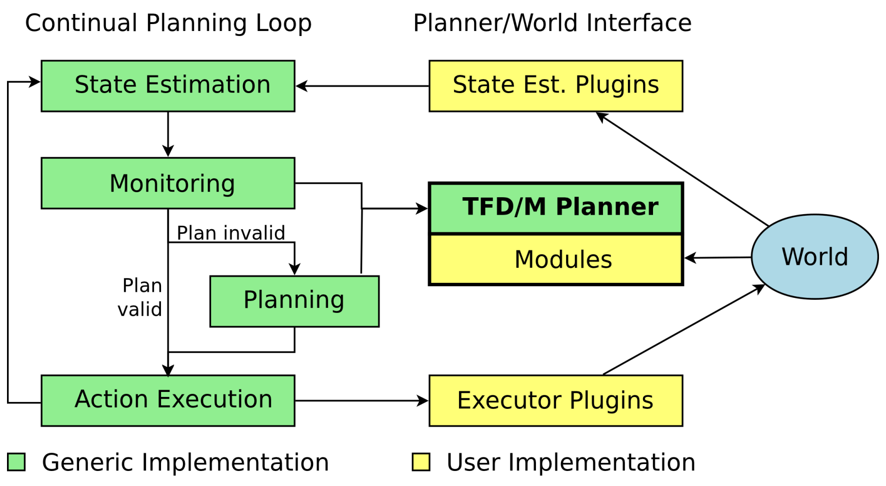


Figure 5.2: This figure gives an overview of the planner's integration in the continual planning loop.

Dealing with such a nondeterministic outcome is then left to monitoring or replanning. This also allows us to implement and plan for an explicit sensing action like performing object detection. Such an action inserts newly perceived objects into the state. Replanning will then be triggered automatically if the objects are relevant for the goal.

## 5.4 Simplifications and Guarantees

An ideal planner would consider all aspects of real-world planning tasks. It is clear that we lose optimality and completeness with our approach. However, often enough, we might be able to still guarantee that the goal can be reached. We now describe how we compile each aspect into a classical planning formulation when the planner runs in a continual planning loop. We address open domains, partial observability and nondeterministic action outcomes. From the perspective of the planner exogenous events can be modeled as nondeterministic outcomes. For the planner it does not matter if a possible unexpected outcome was caused by the robot or by something else.

### 5.4.1 Expanding Universes Instead of Open Universes

The first issue we want to address is the mismatch between the *domain closure assumption* in most planning systems and the fact that it seems rather unrealistic to assume that all relevant objects are known from the beginning. To formulate such tasks semantically, one would assume an infinite (open) domain, where for all types of objects, we have a—possibly infinite—set of such objects, whereby for most of them we do not know anything about them, e.g., where they are located. Such infinite domains lead to the problem that the planning task may become undecidable and that grounding a first-order specification of the planning domain into a propositional logic theory does not work any more.

Relevant objects unknown at planning time cause two issues. These objects can appear in the goal or might be necessary to reach the goal. An example for the first case is a task like "Bring all dirty dishes into the dish washer". Here it is not specified what these objects are or even how many there are. The second case models situations, where tools (e.g., a hammer) that are needed to solve a task are not given in the input.

One way out could be to introduce new objects only when they are detected by the robot when observing its environment thus expanding the universe during execution. After such an introduction of a new object, one can use replanning to deal with the changed domain. This approach can deal with all the objects we encounter in a household or similar environment. This is indeed the strategy we choose. However, one must make sure that these objects are actually encountered during execution. Consider the first example "Bring all dirty dishes into the dish washer". If the robot does not know any dirty objects, the goal formula is true. This is solved by introducing knowledge-based goal formulae that are still expressible even with the limitations from the next section. We now say "Make sure that you know all dirty dishes and bring them into the dish washer". This forces the robot to visit all possible locations and perform a sensing action that will then add all detected objects to the task.

This does not completely solve the issue as such tasks are undecidable, but it leaves us with a weaker assumption. We assume that our knowledge about unknown objects increases monotonously, i.e., once we performed a sensing action no additional objects can appear by executing this action again in the same state and thus once we performed all possible sensing actions we know all objects.

The second case concerns objects that are necessary to reach the goal, but not known to the planner. Note that this is different from the situation, where a planner knows that such an object exists, but not where it is. We actually do have a solution for these cases: *Grounding modules* introduced in the previous chapter enable the planner to create objects during planning if it deems this necessary. Although our intention there was directed at generating objects like placement poses, for the planner a "placement object" or a hammer are both the same—symbolic objects. Such a solution is of course still affected by the limitations discussed in the previous chapter.

### 5.4.2 Limited Uncertainty Through Kleene's Strong Three-Valued Logic

Uncertainty about the current state is semantically usually represented by a *belief set*, the set of states that are believed to be possible. In NPDDL, uncertainty is introduced by the specification of the initial situation and by nondeterminism in action effects.

So, for example, it can be that in a particular situation "it is believed that the cup is broken or the bottle is full". If the robot now observes that the cup is not broken, it concludes that the bottle must be full. This kind of knowledge-based reasoning is, of course, necessary when we want to solve puzzles or diagnose the failure of a machine from observations. However, we do not expect our robots to do so, at least not within the planning process. We are more interested in representing what a robot currently knows about the world or not. So, a simpler way to represent uncertainty might be enough.

Most of the time, we are just uncertain about the value of a fluent and do not consider the connections between different fluents. So, one could extend the value domain (may it be Booleans or many-valued fluents) by a value *unknown*. The evaluation of logical formulae can then be based on *Kleene's strong three-valued logic* (Kleene, 1950). This logic does just what you would expect when combining known with unknown truth values. Of course, such a representation cannot represent that "it is believed that the cup is broken or the bottle is full", or more generally, any disjunctive knowledge. We have reduced the expressivity from uncertainty about formulae to uncertainty about fluent values. This prohibits the robot from doing knowledge-based reasoning and thus if we use such a representation to approximate a given belief set, we lose completeness. For example, we would over-approximate the above statement about the cup and the bottle by "it is unknown whether the cup is broken" and "it is unknown whether the bottle is full". So finding out that the cup is not broken does not allow the robot to make further conclusions.

We thus do not assume that the robot knows and reasons about hidden variables. This means that the only way to find out about the value of a fluent is to know it either from the beginning, to learn about it by *sensing* its value, or by setting the value as an effect of an action. Thus, if a fluent's value is not known initially or set by an action there must be an observation action to directly observe this fluent—at least for all fluents that are necessary to solve a task. In practice we assume that all fluents are observable once the robot is spatially close enough to the location where the fluents can be sensed.

Furthermore, when the actions of the robot have nondeterministic effects, we assume that the robot can only change fluents that are observable after the action execution. In other words, by monitoring the outcomes of actions, the robot can determine the true effects of an action. As a result of this we have now only uncertainty due to the initial state and the degree of uncertainty (as measured by the number of fluents that are *unknown*) shrinks monotonically.

With these two assumptions, which are admittedly quite strong, the representational move of using Kleene's strong three-valued logic instead of general belief sets does not sacrifice completeness, but of course, optimality. It also means that we have reduced the search space from double exponential (belief sets = set of sets of state) to exponential (the set of states). The simplification to a three-valued logic is performed explicitly in the planning domain, i.e., there is no "known"-operator that can be applied to formulae. This means that the possible expressions for uncertainty are evident to a domain writer from the syntax and thus there is no need to determine if a given planning task can be represented.

### 5.4.3 Continual Replanning Instead of Conditional Planning

While we have reduced the search space considerably, there is still the problem that after sensing the value of a fluent (be it after an action execution or after a sensing action), we may have to branch according to the sensed value. What we have done is to simplify the nondeterministic planning problem under partial observability to a nondeterministic planning problem under full observability, where the nondeterminism is created by the outcomes of an action followed by a monitoring action or by possible fluent values of pure sensing actions. In terms of computational complexity, this is a reduction from 2-EXP (Rintanen, 2004) to EXP (Littman, 1997). This is also what is behind Petrick's knowledge-based approach to planning that has been applied to robotics tasks (Petrick and Bacchus, 2002; Gaschler et al., 2013).

The classical way to deal with this problem is to generate *conditional plans* that can branch and perhaps loop. Alternatively and equivalently, a policy (a mapping from states to actions) is created. In this context, one distinguishes *weak plans*, *strong plans*, and *strong cyclic plans*. Weak plans are sequences of actions whereby nondeterministic outcomes are chosen arbitrarily. This is the same model as plans in classical planning with the underlying assumption that one can choose an outcome. Strong plans are winning strategies against any possible nondeterministic outcomes. Strong cyclic plans are strategies that guarantee that we never leave the set of states from which the goal is reachable—again independent of any nondeterministic outcomes.

Strong and strong cyclic plans have the desirable property that they guarantee that eventually the goal will be reached. It would be nice to create a plan taking into account, for example, all possible states of all doors and all possible positions of all cups. However, this does not seem to be worth the computational effort. Furthermore, for real-world planning tasks one might often not be able to guarantee the existence of a strong or strong cyclic plan or these plans might be overly cautious and thus lead to inefficient robot behavior.

In fact, many robotics domains do not require hard guarantees and are quite forgiving concerning non-optimal or even wrong choices. So, our straight-forward approach is to develop a plan for the *intended* outcomes (which have to be marked as such), monitor the success of the plan (which we do anyway in order to deal with uncertainty) and replan if things do not advance as predicted. So, we propose, as many others before us to replace conditional planning by classical planning with replanning.

A replanning approach using a classical planner such as FF (Hoffmann and Nebel, 2001) has been shown to outperform probabilistic planners on many probabilistic planning problems. The reason was that in most cases, the planning domains were *probabilistic uninteresting* (Yoon et al., 2007) meaning that the probabilities had very little impact. Many robotics domains probably have a similar property of being *non-deterministically uninteresting*. This can mean two things: Either there is no bad choice that cannot be repaired any more or a safe strategy like a strong cyclic plan does not exist as any action required to reach the goal might fail (e.g., when picking up a cup there is always a chance to drop it). Given that we follow our approach the main question is now, under which conditions do we not lose soundness and completeness. Before we can answer this question, we have to define what soundness and completeness in a replanning context compared to nondeterministic planning actually means. Intuitively we need to determine when continual planning with a classical planner gives the same guarantees as a strong cyclic plan.

*Completeness* means that if there is strong cyclic plan, then the classical planner is able to generate a successful straight-line plan for each state that could be reached in the strong cyclic plan. This is trivial to achieve provided the classical planner is complete as the conditions for a classical plan are weaker than for a strong plan.

*Soundness* means that the classical planner generates only straight-line plans that are traces of a possible execution of a strong cyclic plan. In particular, this implies that if there is no strong cyclic plan, then the classical planner must not generate a straight-line plan. Moreover, the planner shall never generate a plan that leads to states that are not reachable by a strong cyclic plan. These are severe restrictions that do not seem to be easily satisfiable. In particular, in order to meet them, one has to essentially solve the nondeterministic planning problem.

However, it is possible to specify sufficient conditions on the topology of the search space under which soundness and completeness are satisfied when using a classical planner. The main point is that we never want to "paint ourselves into a corner" or more precisely end up in a *dead end* in the search space. We define a dead end as a state from where no weak plan reaches the goal. A simple guarantee that dead ends are avoided would be to assume the following: There exist no dead ends that are reachable by a weak plan from the initial state. This is the assumption that we use.

Note that this condition is weaker than the condition posed by Kaelbling and Lozano-Pérez (2011), who required to have *reversibility* in their domain, meaning that one has strong connectivity in the search space. Instead, we only assume that we are guaranteed to reach the goal, although not necessarily on the path that was originally planned. The difference is that under our assumption one can, for example, throw away trash irreversibly, which is not possible under the *reversibility* restriction.

We now have a sufficient condition that guarantees soundness and completeness for nondeterministic planning tasks when using a continual planning approach with a classical planner. To guarantee these properties we must show that a planning task fulfills this condition. Although it is easier to test than performing nondeterministic planning it is still as complex as classical planning (Nebel et al., 2013).

**Theorem 33.** *Checking whether there exists a dead end reachable by a weak plan from a given initial state in the search space of a propositional nondeterministic planning task is PSPACE-complete.*

Another problem arises when we have planning tasks that do not fulfill this condition. As it is only sufficient, but not necessary, in a planning task that has dead ends reachable by a weak plan from the initial state strong or strong cyclic plans can still exist. Unfortunately it is not possible to use the dead end test during execution to guarantee an action selection avoiding dead ends. This is just the question of whether there exists a strong plan for the environment forcing us into the dead-end state. In other words, the decision problem is as hard as nondeterministic planning, i.e., EXP-hard.

So, given our assumption that a planning task is dead-end free, it is possible to guarantee that a replanning approach will eventually reach a goal. As uncertainty is reduced monotonically, the number of nondeterministic choice points due to sensing actions reduces when these are executed. For this reason, eventually, we will have a completely informed state in which a classical plan is sufficient to reach the goal, or we have reached a goal state before.

### 5.4.4 Discussion

Our formulation for real-world planning tasks is generic and thus solving these can be unachievable in practice or even in theory. Realistic demands to a system are often lower, especially when balancing solution guarantees with efficiency. We will now discuss the implications that the simplifications and assumptions stated above cause for practical applications. In particular we are interested in how realistic these are, what the expected robot behavior is when the assumptions are violated—obviously then without guarantees, and in how far this behavior is acceptable for robot domains without hard requirements such as service robotics. We also consider situations that cannot be guaranteed to be solved at all, where nevertheless a reasonable strategy to tackle such tasks exists. As a measure for acceptable behavior we compare our robot to a human performing these tasks—of course given that the human only possess the robot's skill set or if a human instead of the planner decides which action is executed next.

The first simplification is the use of *expanding instead of open universes.* We mentioned a solution for creating necessary objects like tools during planning by grounding modules. However, it is questionable if one wants the robot to start looking for imaginary objects that could help to solve the task. Our formalism does not exclude this possibility, but usually a domain writer has a general idea if there is a hammer, a sponge, duct tape or whatever else is needed and thus these will be included in the planning task. In any practical experiments we always provided necessary tools in the input. One thing that is definitely not possible is the invention of a new kind of tool with a yet unknown behavior. We consider this kind of improvisation to be beyond the scope of a planning-based system.

The other assumption is that our knowledge about which objects exist increases monotonously. If we drop this assumption, termination is impossible to guarantee. Consider a butler robot that has to "Bring all dirty dishes into the dish washer". If someone places dirty dishes on a table after the robot checked if there is something to do, the robot needs to return. Otherwise there is no guarantee that these will be cleaned. Unless we assume that we know all objects at some point in time this will go on indefinitely. There exist scenarios, where termination is not necessary and in fact undesirable. For example, a robot serving drinks in a cocktail party should not stop offering drinks if everyone has gotten one single drink. Depending on the formulation of our planning task we can enable either behavior.

Our model of *limited uncertainty* prevents the robot from doing any diagnostic reasoning in the planner. We consider this sufficient for modeling robotics tasks. This simplification already appears during the formulation of a task and thus does not influence the execution. We therefore focus on the assumption that knowledge about fluents increases monotonously. An increase in uncertainty is caused by nondeterministic effects that we cannot observe after an action execution, e.g., exogenous effects. If we drop this assumption, again we might have tasks that cannot terminate. A robot that is tasked to tidy up two kids' rooms can clean either room, but when the kids are still in their room as soon as the robot turns its back to tidy the other room the one that was just cleaned might get messy again. Unless both rooms are observable at the same time there is not way to ensure that both are clean. If we want the robot to act at all, we must therefore decide what behavior we want: Continuously make sure that each room is clean or assume that tidying a room once is sufficient. In either case the continual planning loop will insert sensing actions whenever needed.

The assumption that our planning tasks are *dead end free* is violated in reality. As a straight-forward example consider a robot picking up a plate. If the plate is dropped and breaks, there is no way to place the unbroken plate at its goal location. However, this is inevitable. When handling breakable objects we accept that even humans might break something. Formally speaking these are situations, where no strong (cyclic) plan exists. In practice if there is no guaranteed safe plan, we would still like the robot to try to reach the goal. In this case nondeterministic planning techniques do not help and therefore following a weak plan is the only option.

However, there do exist real-world scenarios that leave the robot trapped in a dead end, but are preventable. Consider a mobile manipulation robot with two arms that is supposed to bring multiple objects from one room into another, where both rooms are on opposite sides of a hallway. If the robot picks up an object in each hand and then enters the hallway, a human might close the door behind the robot. Unless the door to the other room is open or the robot can find a spot to place an object intermediately the robot is now trapped. This could have been easily prevented by only taking one object at a time, so that the other hand is free to open doors again.

This is an example, where a nondeterministic planner guarantees that the robot does not get trapped. As weak plans are usually more goal-directed they have lower costs (e.g., here taking both objects at the same time), but they might leave the robot trapped. Nevertheless we still argue that even when this is possible a classical planning approach might be preferable. The reason for this is that strong plans are overly cautious and thus overly long. In our example the robot would have to traverse between both rooms twice as often when following a strong in comparison to a weak plan. The decision between using a classical or nondeterministic planner here is a decision between efficient execution and the guarantee to reach the goal.

We have now seen that there are realistic situations, where our assumptions are violated. Here we need to consider what the best solution for these is. Some assumptions are actually necessary as there is just no possible solution method. Planning tasks with open domains are undecidable and without a monotonous decrease in uncertainty either in the known objects or in the value of fluents we cannot guarantee termination. Dead ends can also not always be avoided. In these cases we consider it better if the robot tries to reach the goal than to give up without trying. Finally there are some situations, where a nondeterministic planner can provide guarantees over a classical planner. Still, a classical planner here also has advantages in computation time and execution efficiency, which we consider more important especially when the environment in general is benevolent to the robot, i.e., no one actively tries to make the robot fail. For a service robot, we assume that the contrary is the case. For example, nowadays simple vacuum robots like a Roomba are available outside of science labs and users gladly accept that they might need to move chairs out of the way to save time overall instead of vacuuming themselves.

## 5.5 Lazy Module Evaluation and Caching

In this section we are concerned with computation speed for integrated task and motion planning. We introduce two novel techniques. Lazy module evaluation is suited for search-based planning with semantic attachments as it modifies the search itself. Most of the presented caching techniques are implemented on the module side, i.e., before the geometric computations and could therefore also be used in other integrated reasoners. An important aspect of all techniques within this section is that they do not impact soundness.

### 5.5.1 Lazy Module Evaluation

Lazy module evaluation tackles the same point in the search as a technique from classical planning named *deferred heuristic evaluation* (Richter and Helmert, 2009). Many successor states are generated and pushed in the open queue, but might never be visited. Deferred heuristic evaluation identifies heuristic calculations as the expensive operation. By taking the heuristic measure of the successor's parent instead of the successor itself, only a single heuristic computation is needed for all successors. In our case, generating successors is the costly part as this means computing *applicable*$(s, o)$, which might call modules. Therefore, we defer these calculations to a later time by using a relaxed version of an operator to perform the applicability-check.

**Definition 34.** *The operator* $o^+$ *is a relaxation of an operator* $o$ *if for all states* $s \in S$*: applicable*$(s, o) \Rightarrow$ *applicable*$(s, o^+)$.

This will never discard applicable states, but might include inapplicable ones. Thus, we cannot always compute a successor state and store the pair $s, o$ implicitly defining the successor (see Section 2.3). The additional problem that we cannot calculate a heuristic value without the successor is easily solved by applying deferred heuristic evaluation. The issue of possibly inapplicable open queue entries is addressed once a state is taken out of the queue to be expanded. Now we perform the full, i.e., non-relaxed, applicability check. If *applicable*$(s, o)$ fails at this point, we discard this entry and continue by taking the next state out of the open queue. What we have done is to defer the costly applicability check from the point, where a successor is inserted into the open queue to the point, where it is removed again. In comparison to only replacing applicability checks by faster, approximate versions when the insertion is performed this method is sound.

We can derive relaxed operators as operator conditions are represented as conjunctions (see Section 3.4). Herein module predicates are positive literals. The simplest option is to skip all module calls. A better way is offered by our module interface that optionally calls a fast, relaxed version. Often there is an obvious way to come up with such a relaxation. Consider for example the `checkTransfer` condition checker that computes if an object can be placed. Instead of computing a motion plan, the relaxation only tests whether the target placement is collision-free.

### 5.5.2 Caching Techniques

In relation to classical planning applying an operator or just testing for applicability can take orders of magnitude longer when external modules are called. The main goal of the caching techniques is to avoid as many module calls as possible while retaining soundness. One way is to cache results to module calls, i.e., store and reuse results from previous calls. Different types of modules (condition checker, effect applicator or cost module) might require the same underlying geometric computation. Therefore, as a first step we store computation results instead of just the return values for a specific module type and the same cache is exchanged between compatible modules.

For example, our `checkTransfer` *condition checker* stores the placement pose of the object and the resulting manipulator configuration. Subsequent calls to the `applyTransfer` *effect applicator* can now access the pose and manipulator configuration without recomputing the motion plan. We name a call to any type of module a *request*. A request is issued for an operator $o$ in a state $s$. The caching techniques presented in this section store computation results for such requests in a mapping data structure. Here the central question is how a cache key is computed that matches as many requests as possible without loosing soundness.

### Full State Caching

The full state $s$ paired with the operator $o$ that the request was issued from is used as a key. The cache hits these cases, where the same request might be issued multiple times. This does happen during search, e.g., when revisiting a state that a shorter path was found to. An advantage of this method is that cache keys can be provided by the planner.

### Partial State Caching

A partial state $s^p \sqsubseteq s$ is computed by the module and used as the cache key together with the operator $o$.

**Definition 35.** *A partial state $s^p \sqsubseteq s$ is a partial variable assignment, i.e., a function $s^p : \mathcal{F_L}^p \rightarrow \{true, false\}$ and $s^p : \mathcal{F_N}^p \rightarrow \mathbb{R}$, where $\mathcal{F_L}^p \subseteq \mathcal{F_L}$ and $\mathcal{F_N}^p \subseteq \mathcal{F_N}$. For all variables $v_L \in \mathcal{F_L}^p$ or $v_N \in \mathcal{F_N}^p$ the following holds: $s^p(v_L) = s(v_L)$ and $s^p(v_N) = s(v_N)$.*

This partial state must contain all relevant information for the computation. A minimal partial state is straight-forward to determine as it consists of all variables used in the module computation. For example, a putdown motion only considers poses of objects that are within the reach of the robot. A partial state implicitly defines the set of states $S^p = \{s' \in S | s^p \sqsubseteq s'\}$. Any request for a state in $S^p$ will thus hit the same cache entry. $|S^p|$ determines the effectiveness of partial state over full state caching.

### Subsumption Caching

We take the idea of a partial state one step further and include requests for different states. The idea is to build a subsumption hierarchy[1] of more or less constrained states.

Take the example of a putdown request for an object that succeeded when two objects were already placed on a table. Now if only one object was present, a request will surely succeed as it is less constrained (always given the same object positions and dimensions). This is illustrated in Figure 5.3. On the other hand, when a putdown request failed with three objects present, requests with four objects must also fail as it is more constrained. This notion is formalized as follows.

[1]This term is not to be confused with the well known subsumption architecture. Here we mean logical subsumption.

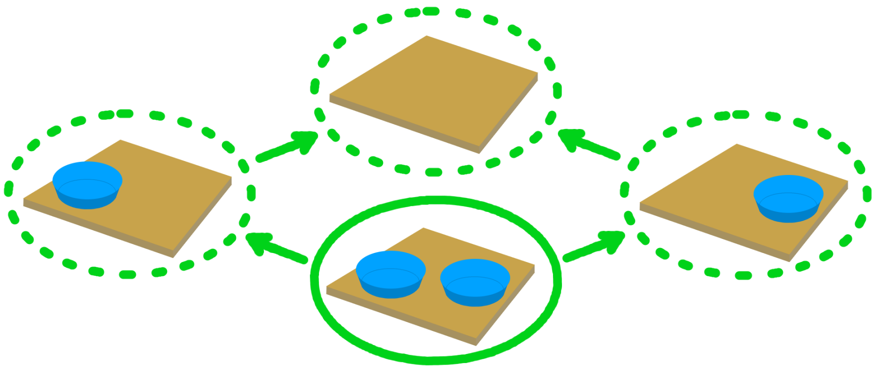


Figure 5.3: This figure shows the subsumption hierarchy for successful requests to put an additional object on the table. When a putdown request succeeded on the table with two objects already placed, we can infer that less constrained cases (dashed lines) must also succeed without the need to do any calculations.

**Definition 36.** *For a state $s$ and operator $o$ a state $s^-$ is less constrained, i.e., $s^- \leq_o s$ iff $applicable(s, o) \Rightarrow applicable(s^-, o)$. Likewise, a state $s^+$ is more constrained iff $s \leq_o s^+$.*

Given such a *constrained* relation between states, the procedure for answering a request by subsumption caching is illustrated in Algorithm 7. First, we test if the request can be answered by the cache. For a (partial) state $s$ and operator $o$, we test if $s$ is less constrained than some state $s'$, where the request succeeded (ll. 4–7). Analogously we check whether $s$ is more constrained than some state $s'$, where the request failed (ll. 8–11). Should neither test match, the request is computed and the result added to the cache (ll. 13, 21, 28). We build a subsumption hierarchy of cached requests by only retaining the most constrained states for succeeded requests (ll. 15–20) and the least constrained states for failed requests in the cache (ll. 23–27). Note that in comparison to the first two caching methods the *constrained* relation is domain dependent. Without a non-trivial *constrained* relation subsumption caching becomes partial state caching. Thus, if such a relation can be found, subsumption caching dominates partial state caching.

### Global caching

As we use continual planning, the planner might be called multiple times during a task. It is therefore likely that the same geometric computation is requested in different planner calls, e.g., if there is a path to drive from location B to location C. Global caching reuses cache entries from previous planner calls. Therefore the caches are not only stored in memory during planning time (local), but also optionally in a persistent storage (global). To ensure soundness, we require that the cached

**Algorithm 7** Subsumption Caching request for $s, o$

```
1:  **Cache Mapping** $Successes : S \times O \to Result$
2:  **Cache Mapping** $Failures : S \times O \to Result$
3:  // Test for cache hit
4:  for all $(s', o' \mapsto r') \in Successes$ do
5:      if $s \leq_o s'$ and $o = o'$ then return $r'$
6:      end if
7:  end for
8:  for all $(s', o' \mapsto r') \in Failures$ do
9:      if $s \geq_o s'$ and $o = o'$ then return $r'$
10:     end if
11: end for
12: // Cache miss, compute and insert into cache
13: $r, success \leftarrow computeRequest(s, o)$
14: if $success$ then
15:     for all $(s', o' \mapsto r') \in Successes$ do
16:         if $s \geq_o s'$ and $o = o'$ then
17:             // Subsumed by $s, o$
18:             Remove $(s', o' \mapsto r')$ from $Successes$
19:         end if
20:     end for
21:     Insert $(s, o \mapsto r)$ into $Successes$
22: else
23:     for all $(s', o' \mapsto r') \in Failures$ do
24:         if $s \leq_o s'$ and $o = o'$ then
25:             Remove $(s', o' \mapsto r')$ from $Failures$
26:         end if
27:     end for
28:     Insert $(s, o \mapsto r)$ into $Failures$
29: end if
30: return $r$
```

information fully defines the module computation. This means that a request can be answered independent of the current robot state, sensor data, or similar. For example, when all object poses are contained in the request key the fact that the robot can put a cup next to two bowls on a table is fully determined. This technique is complementary to the other techniques mentioned above and thus can be combined with either.

## 5.6 Applications and Evaluation

The goal of this section is to show that our approach to solving real-world planning tasks by continual planning works in realistic robotics scenarios. We illustrate three aspects. First, on the *TidyUp-Robot* example we demonstrate in detail how a complex robotics task is modeled using the techniques from Section 5.4. In a similar setting we perform a quantitative evaluation of the computation times for such a system. Here, we also address in how far the methods introduced in Section 5.5 are able to reduce planning times. Finally, to show the versatility of the approach we give a short overview of various other robotic applications that used our system as a high-level control method.

### 5.6.1 TidyUp-Robot

There are multiple reasons why evaluating high-level robot control architectures is difficult. Given that planning tasks with open domains are undecidable a proper solution cannot exist. The same holds for situations, where no strong plan exists. There is no clear best solution to compare against. Furthermore, even when such a guaranteed solution strategy exists, we make assumptions about the structure of real-world tasks. Here we answer two central questions in detail: How exactly is a real-world planning task modeled for a fully implemented robot system? Do our simplifications lead to a working system that performs well in realistic tasks?

Thus we implemented a proof of concept system (see Figure 5.4) that integrates multiple skills of the PR2 mobile manipulation robot into a single complex system. The robot's capabilities in terms of its sensors and actuators as well as the algorithms for its skills directly influence how well the full system performs. For example, if a grasping algorithm would fail 50% of the time it might be necessary to consider contingencies during planning. We use state of the art algorithms to represent what one can expect from a modern robot system. This means that to be able to judge how well a continual planning approach works, it is necessary to state what skills we use and also what information about the world is given to the robot as part of the input.

The goal of *TidyUp-Robot* is to tidy objects in a household scenario. The task of tidying up serves as a generic example to demonstrate the problems that a robot faces in a real-world setting. This consists of finding objects such as cups or bowls and bringing them to a predefined target location—their *tidy-location*—that might be

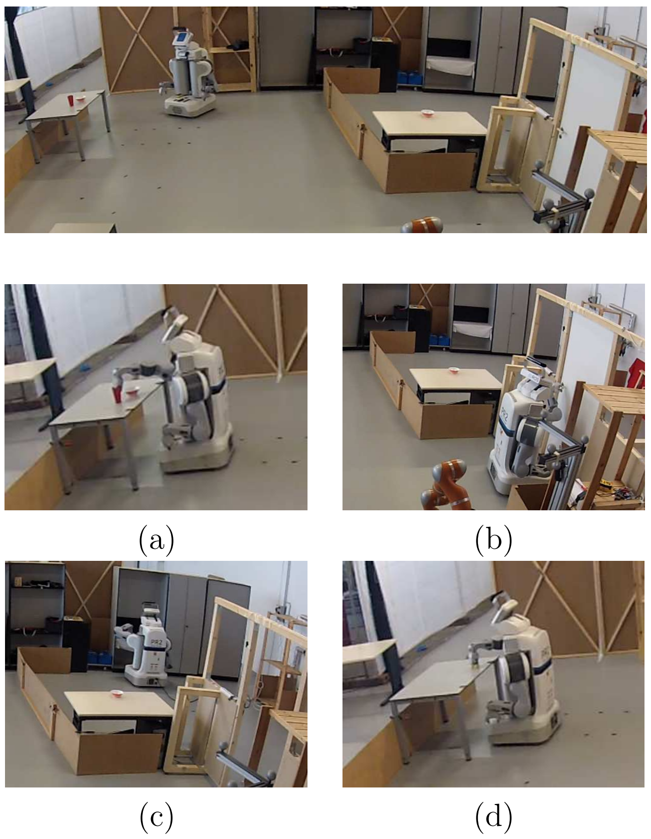


Figure 5.4: Example scenes from the PR2 operating in the TidyUp-Robot domain. The robot collects any cups and bowls from the two tables (a). Next, it opens the door to the other room (b) to put the objects into the shelf (c). Finally, the robot wipes the tables where the objects were initially located (d). A video is available at: `http://www.youtube.com/watch?v=pTSz2RBZ2wA`

in another room. In addition, we require that all spots, where objects were found, should be wiped clean with a sponge. The robot is given a three dimensional map of the environment that is annotated with locations of static objects such as tables, shelves or a door. No prior information about any of the movable objects is known, not even how many there are.

The PR2 robot shown in Figure 5.4 was originally developed by Willow Garage and provides basic implementations for mobile manipulation skills based on the Robot Operating System (ROS) (Quigley et al., 2009). From these we use the table top segmentation and object detection, as well as the "arm navigation" components for motion planning and pick and place execution for both arms. For navigation of the base we use a search-based motion planner with full 3d collision checking that uses the ARA* algorithm (Likhachev et al., 2004). To open doors the robot can detect if they are closed and push these open (Endres et al., 2013). Wiping tables is based on a coverage approach to plan a trajectory for a sponge (Hess et al., 2012).

These skills directly map to PDDL/M operators. Pick and place is represented by the `pick-up` and `put-down` actions from Chapter 3 and the navigation component is called by the `drive-base` action. `detect-objects` calls the table top segmentation and object detection and `detect-door-state` checks if the door the robot is located at is open or closed. There are also an `open-door` and a `wipe` operator that are implemented straight-forward in PDDL/M semantics, e.g., before wiping a table the spot to be wiped must be free and the robot must a have sponge in its hand. Navigation is split into `drive-through-door` that drives from one room into another and `drive-base` that navigates only within the same room. Both call the same navigation skill. The reason for this split is to enable efficient partial state caching. If there was only a single navigation action, it would be relevant for each door if it is open or not. Thus for $N$ doors there would be $2^N$ different path planning queries. Additional helper actions move the arm to the side of the robot for navigation or to provide a clear field of view for object detection and two operators that pickup or putdown the sponge before and after wiping.

We can already see that this system is more complex than the simple `pick-up`, `put-down`, and `drive-base` examples from before. This is caused by the additional skills, by adding the helper actions and by the detection actions. This shows one practical advantage of task planning in comparison to a scripted approach. A planner handles the dependencies between different operators gracefully, while any extra feature must be considered everywhere in a manually defined behavior. A detailed discussion of the complete planning domain would be repetitive as most parts are standard PDDL and thus not specific to robotics. Therefore we focus on two aspects: the semantic attachments used for navigation and manipulation and the modeling of an expanding universe and incomplete knowledge.

Navigation costs are integrated as a *cost module* in the `drive-base` operator. The module calls the aforementioned path planner with 3d collision checking to compute actual path costs for this operator. In addition if infinite cost is returned the operator is not applicable. For manipulation tasks, we employ two modules for put-down actions: A *condition checker* that determines if an object can be put down on a table and an *effect applicator* that provides the numerical state update writing the resulting pose to the state. The implementation is based on a discretization of possible positions on the table. Given the size of the object to place and the poses of objects already on the table, first a set of free spots is determined. If this is empty, the condition checker returns false. The free spot closest to a point 80 centimeters in front of the chosen manipulator arm's origin is chosen as the unique placement pose to be updated by the effect applicator. The motivation here is to prefer poses with the least amount of kinematic constraints. A similar free space check is performed as a *condition checker* for the `wipe` operator although in that case the pose is not chosen, but determined by the spot to wipe.

Expanding universes are addressed by planning for an explicit sensing action `detect-`

`objects` that, when *executed*, calls the robot's object detection method. Upon each observation recognized objects are matched on type and distance to previously known objects that are updated with the new pose. Newly detected objects are added to the state. In addition a new wipe spot at the object's pose is added. In PDDL/M the operator is specified as follows.

```
(:action detect-objects
    :parameters (?l - manipulation_location)
    :duration (= ?duration 1.0)
    :precondition
    (and
        (at-base ?l)
        (arms-drive-pose)
        (or
            (not (searched ?l))
            (not (recent-detected-objects ?l))
        )
    )
    :effect
    (and
        (searched ?l)
        (recent-detected-objects ?l)
    )
)
```

We can `detect-objects` at a location `?l` if the robot is at this location and both arms are out of the field of view (`arms-drive-pose` is true). Here `searched` and `recent-detected-objects` are knowledge predicates. Note that *within a plan* this operator does only change these predicates, but no other state variables. The planner assumes that the state does not change. Of course this can happen during execution, in which case replanning might become necessary.

The planner must know when object detection should be performed. This is the function of the knowledge predicates. There are two such predicates that have different semantics. `searched ?l` states if we ever executed this action at `?l`. It is necessary that all locations are searched at least once. This is represented in a generic goal formula.

```
(:goal (and
    (forall (?o - arm) (hand-free ?o))
    (forall (?o - movable_object) (tidy ?o))
    (forall (?o - wipe_point) (wiped ?o))
    (forall (?o - manipulation_location) (searched ?o))
))
```

For the task to be solved all known objects have to be cleaned, i.e., they need to be at their `tidy` location and the spot, where this object was found has to be `wiped`. In addition all `manipulation_locations` must be `searched`. With the assumption that knowledge about known objects monotonously increases searching each location once is sufficient to know all objects.

The other predicate `recent-detected-objects` states if fluents about *all* objects at a location (e.g., their poses) are known. This models uncertainty in the initial state or about nondeterministic action outcomes, i.e., the effect `(not (recent-detected-objects ?l))` is found in every manipulation action. Likewise `(recent-detected-objects ?l)` is a precondition for each manipulation action. Thus between consecutive manipulation actions the planner is forced to insert `detect-objects` actions, and planning continues assuming the actual effect took place as desired. The reason for declaring *all* objects' fluents unknown after a manipulation action for a specific object is that we cannot know what might happen to other objects. For example placing an object near others might not only place that object slightly different than planned, but could also knock another object over. Under the assumption that we will actually observe any such nondeterministic outcome uncertainty will thus still monotonously decrease.

A simpler representation is the state of doors, i.e., if a door is open or not. In this case the doors in the environment are given with the input. Two predicates describe the state of a door. `door-state-known ?d` is true iff we know if a door is open. If this is true, `door-open` holds that value. Therefore the single effect of the `detect-door-state` action is `(door-state-known ?d)`. These predicates are used by the `open-door` and `drive-through-door` actions. Both actions require `door-state-known` to be true. For `drive-through-door` the value of `door-open` must be true, while for `open-door` the value of `door-open` must be false. The following excerpts from the operators `detect-door-state`, `drive-through-door`, and `open-door` illustrate this.

```
(:action detect-door-state
    :parameters (?l - door_in_location ?d - door)
    :precondition (and
        (at-base ?l)
        (not (door-state-known ?d))
        ...
    )
    :effect
        (door-state-known ?d)
)
```

```
(:action drive-through-door
    :parameters (?d - door ?s - door_in_location ?g - door_out_location)
    :precondition
    (and
        (at-base ?s)
        (door-state-known ?d)
        (door-open ?d)
        ...
    )
    ...
)

(:action open-door
    :parameters (?l - door_in_location ?d - door ?a - arm)
    :precondition
    (and
        (at-base ?l)
        (door-state-known ?d)
        (not (door-open ?d))
        ...
    )
    ...
)
```

The full *TidyUp-Robot* system consists of the domain formulation in PDDL/M including semantic attachments, the implementations of all operators within the domain to execute on the robot, as well as the underlying robotics algorithms that are being called. We tested the complete system in the environment shown in Figure 5.4. During this experiment the robot detected all objects, brought them to the tidy location in the shelf (see Figure 5.4 c), and wiped all tables. Most actions succeeded as planned. Replanning was mostly triggered when new objects were detected, which is expected, or actual placement poses were too far off the planned pose. The first execution of `open-door` failed, which was initially not detected. When the robot came back to the door with two objects in hand it had to replan and bring one object back freeing a hand to open the door again. While such unexpected situations are not an issue for an automated reasoning approach like planning, we reckon that finding a proper solution for any conceivable situation with manual scripting becomes impractical quickly.

### 5.6.2 Lazy Module Evaluation and Caching

While the previous section illustrated how the techniques introduced in this chapter provide the high-level control architecture for a real-world robot system, in this section we are interested in a quantitative evaluation in particular with regard to the planning techniques that we developed. Besides comparing the efficiency techniques with each other we are also interested in absolute planning and execution times as these serve to judge the viability as a practical system.

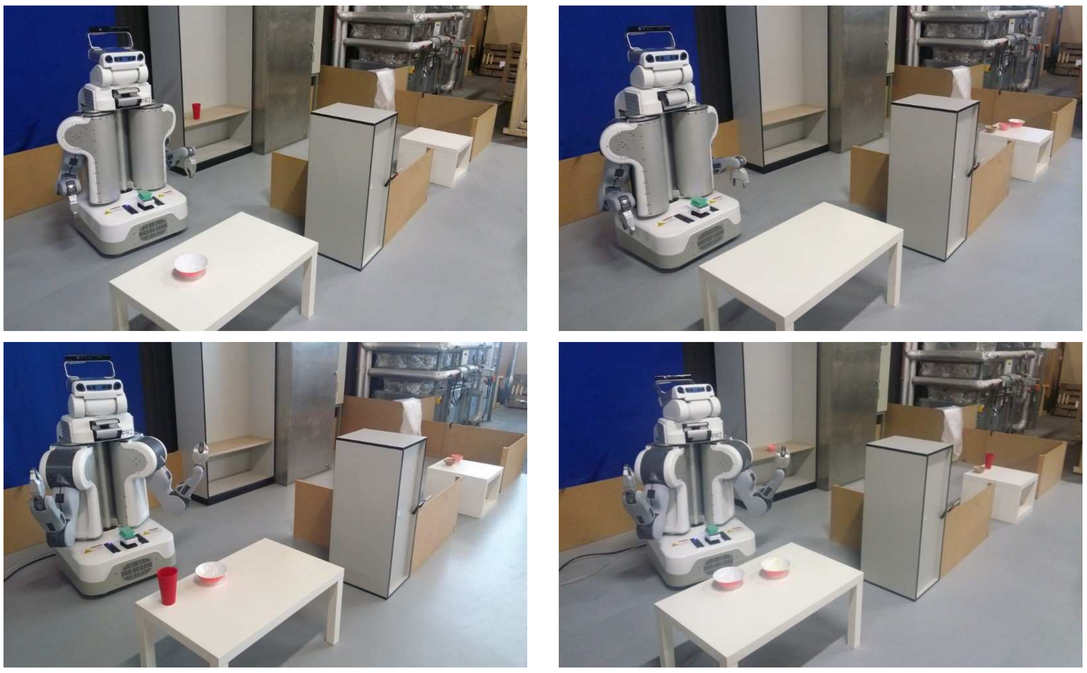


Figure 5.5: This figure shows four of the evaluation settings. The task was to bring all objects to the front table and wipe the spot under each object.

We evaluated our system on the PR2 robot running the *TidyUp-Robot* description in the scenario shown in Figure 5.5. Although this scene appears to be significantly smaller than the one used in the previous experiment from a planning perspective the only notable difference is that there is no door. The number of objects to consider is similar. Only navigation times become less relevant, when everything is closer together. We positioned one to five objects in two different configurations for each number of objects. The goal was to find all objects, bring them to a specific table and wipe the initial location of each object, so each object had to be interacted with at least once.

Each setting was executed until the system reported the task to be completed. For each planner call we ran the planner with lazy and eager (i.e., non-lazy) module evaluation. After the first plan was found we continued the anytime search for another 25% of the time it took to find the first plan. This way we automatically adapt to the problem complexity. The plan from the lazy solution was executed. We used partial state caching in combination with global caching.

| Task | Objects | Total Planning Time [s] | | Max Single Plan Time [s] | | Planner Calls | Execution Time [s] | Actions |
|---|---|---|---|---|---|---|---|---|
| | | Eager | Lazy | Eager | Lazy | | | |
| 1 | 1 | 76.7 | 41.8 | 72.0 | 37.3 | 2 | 497.2 | 24 |
| 2 | 1 | 66.0 | 43.0 | 61.2 | 39.0 | 2 | 303.4 | 21 |
| 3 | 2 | 57.4 | 42.3 | 47.9 | 32.2 | 3 | 807.8 | 38 |
| 4 | 2 | 106.2 | 71.2 | 73.2 | 42.2 | 4 | 631.8 | 40 |
| 5 | 3 | 221.0 | 158.1 | 112.6 | 76.6 | 4 | 823.8 | 46 |
| 6 | 3 | 124.0 | 99.8 | 42.4 | 27.7 | 18 | 1630.9 | 94 |
| 7 | 4 | 289.6 | 220.8 | 203.1 | 153.5 | 10 | 1226.1 | 63 |
| 8 | 4 | 120.9 | 99.8 | 57.8 | 56.4 | 4 | 1019.3 | 55 |
| 9 | 5 | 686.2 | 505.3 | 263.7 | 220.1 | 6 | 1651.7 | 82 |
| 10 | 5 | 350.2 | 255.5 | 281.3 | 211.0 | 3 | 1195.0 | 56 |

Table 5.1: This table gives an overview of the system performance under eager and lazy evaluation.

#### System Performance

We investigate the overall system performance in Table 5.1. For each task we give the total accumulated planning time for lazy and eager module evaluation, as well as the maximum time for a single plan. Single plan times show how hard the task itself was once all objects had been seen for a first time. Subsequent planner calls might have cached computations or face a problem that is already partially solved by the robot's previous actions. During the experiments two tasks were not fully completed. In task 6 one wipe action could not be executed and in task 9 the robot placed an object too far on the edge dropping it irrecoverably. Nevertheless, the system continued to solve all remaining goals.

As expected we observe increasing planning and execution times when more objects are present. The relation between execution and total planning time indicates that the system is usable in practice. Besides the number of objects additional factors are the initial object placements and the order in which the robot finds and interacts with objects. If the robot solves the task partially for some objects before finding others, later planning calls might be easier. Such eagerness might be advantageous, but can have adversary effects if those solved objects block others later on. For smaller problems planning time was considerably lower than for larger ones. We see that in relation to the execution time planning scales slightly worse, which is not a surprise given the combinatorial nature of the problem. Comparing lazy and eager module evaluation we observe that lazy evaluation is able to find plans faster than eager evaluation.

## Caching Techniques

We perform a detailed comparison of the different caching techniques. For each module call, we determined if each caching strategy produced a hit or miss. We also recorded the time it would have taken to compute the query for a miss. Recorded times for all queries are accumulated over each run. Module computations were only performed if partial state caching had a miss. Not using caching or full state caching misses would repeat identical computations and in these cases we use the stored time from partial state caching. If subsumption caching misses, we use the time from partial state caching. Note that we must perform all partial state computations, even if subsumption caching hits as this might have been subsumed from a different partial state. We omit the time it takes to answer a cache hit, which in our case is much lower than the computation time for a miss.

| Task | No Caching | | Full State | | Partial State | | Subsumption | |
|---|---|---|---|---|---|---|---|---|
| | Requests | Time [s] | Misses | Time [s] | Misses | Time [s] | Misses | Time [s] |
| 1 | 98 | 60.5 | 60 | 36.7 | 8 | 4.5 | 8 | 4.5 |
| 2 | 140 | 101.0 | 70 | 49.7 | 5 | 3.0 | 5 | 3.0 |
| 3 | 347 | 170.4 | 224 | 110.5 | 22 | 11.3 | 20 | 10.5 |
| 4 | 1124 | 898.7 | 402 | 318.9 | 37 | 27.0 | 37 | 27.0 |
| 5 | 7488 | 4701.7 | 1954 | 1184.1 | 70 | 46.4 | 68 | 45.3 |
| 6 | 2559 | 1422.1 | 1568 | 862.4 | 200 | 121.7 | 178 | 111.7 |
| 7 | 5234 | 3411.2 | 1880 | 1285.5 | 175 | 140.2 | 159 | 126.6 |
| 8 | 1167 | 790.9 | 540 | 358.3 | 66 | 40.3 | 66 | 40.3 |
| 9 | 11823 | 15907.6 | 3980 | 5301.0 | 247 | 472.1 | 203 | 393.7 |
| 10 | 6900 | 4230.3 | 1844 | 1112.8 | 63 | 38.8 | 62 | 37.6 |

Table 5.2: This table compares the different caching methods for the putdown module, when no global caching is used.

Table 5.2 shows cache misses and times for putdown module calls without global caching. Table 5.3 shows the comparison results when global caching is used. As expected, no caching at all or full state caching are inefficient and not feasible in practice. Also there is no difference for full state caching with or without global caching, mainly due to the fact that the full state contains the current robot state, which is unlikely to be exactly the same between planner calls. Partial state caching performs considerably better than those techniques as it never repeats the same calculation. Subsumption caching is able to reduce cache misses even further. We presume that the impact would be greater for even more complex problems.

Global caching is also able to reduce cache misses, mainly in tasks 6 and 7. Table 5.1 shows that these tasks required multiple replanning steps, where global caching was able to utilize the stored computations. Although subsumption caching dominates partial state caching, in task 6 we see more misses for subsumption caching than

| Task | No Caching | | Full State | | Partial State | | Subsumption | |
|---|---|---|---|---|---|---|---|---|
| | Requests | Time [s] | Misses | Time [s] | Misses | Time [s] | Misses | Time [s] |
| 1 | 98 | 60.5 | 60 | 36.7 | 8 | 4.5 | 8 | 4.5 |
| 2 | 140 | 101.0 | 70 | 49.7 | 5 | 3.0 | 5 | 3.0 |
| 3 | 347 | 170.4 | 224 | 110.5 | 17 | 8.9 | 15 | 8.0 |
| 4 | 1124 | 898.7 | 402 | 318.9 | 32 | 23.0 | 32 | 23.0 |
| 5 | 7488 | 4701.7 | 1954 | 1184.1 | 62 | 42.2 | 62 | 42.2 |
| 6 | 2559 | 1422.1 | 1568 | 862.4 | 68 | 46.8 | 95 | 71.1 |
| 7 | 5234 | 3411.2 | 1848 | 1259.7 | 90 | 69.1 | 74 | 55.5 |
| 8 | 1167 | 790.9 | 540 | 358.3 | 66 | 40.3 | 66 | 40.3 |
| 9 | 11823 | 15907.6 | 3980 | 5301.0 | 228 | 413.1 | 190 | 340.9 |
| 10 | 6900 | 4230.3 | 1844 | 1112.8 | 63 | 38.8 | 62 | 37.6 |

Table 5.3: This table compares the different caching methods for the putdown module with global caching.

partial caching in conjunction with global caching. These were due to numerical inaccuracies when converting global cache keys back to poses, which is only required when matching states in subsumption caching. We also investigated the navigation module and partial caching proved similarly effective. As subsumption caching is not applicable in this case, we omit these results.

## 5.6.3 Robot Applications

A main feature of many modern symbolic planners is that they are domain-independent, i.e., they can solve arbitrary tasks for different domains without any changes to the planner itself. Keeping this property when adding semantic attachments in Chapter 3, infinite branching in Chapter 4, and the continual planning loop in Section 5.3 was an important aspect during all these steps. Therefore, to apply our current system to any robot one only needs to define robot-specific interfaces for action and perception as well as a PDDL/M domain that describes the robot's skills. This principle allowed us to apply our planning system to various robots in addition to the PR2 shown above. Here we demonstrate the versatility of this approach by illustrating three other applications that in some form use our planner TFD/M. We focus the descriptions on the planning part and shortly address the capabilities of each system.

### Nao-TidyUp

Our planning system was used as the high-level control architecture in a tidy up scenario with a humanoid Nao robot (see Figure 5.6). The PDDL/M planning domain was derived from the PR2 domain described above. The goal here was similar:

Bring all objects to their target locations shown by a colored marker. Instead of a generic pick-up action object-specific pick-up actions were introduced as, e.g., picking up a larger cube requires to use both arms. These actions map to full-body motions of the Nao robots. Paths for the robots might be blocked by objects that need to be removed. Therefore, first, the Nao's path planner is used as a *cost module* and also to check the applicability of a navigation action. The robot has two options to clear a blocked path: Either kick an object like a small ball out of the way or pick the object up and move it somewhere else. As it might not be possible to directly bring an object to its target location intermediate placement poses are chosen with a *grounding module*. More details about the robot implementation are given in the paper by Hornung et al. (2014).

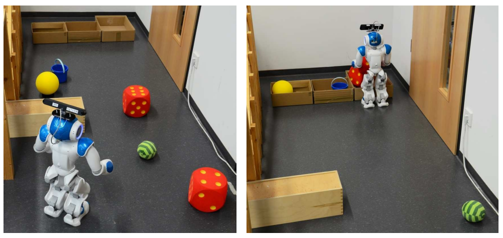


Figure 5.6: A Nao robot tidying various objects.

### Sokobot

Sokobot brings the puzzle game Sokoban to robotics. Sokoban models a warehouse keeper that has to organize boxes by pushing them to their target locations in a grid world. A Turtlebot 2 was used that could localize itself in the grid world and was able to recognize and push boxes (see Figure 5.7). The planning domain expresses Sokoban as a purely symbolic problem on the grid world with full observability and no semantic attachments. The main difficulty in Sokoban is that push actions cannot be reversed. Thus it is important to plan a sequence of actions to reach the goal without getting into a dead end. Although Sokoban is a challenging planning problem (Botea et al., 2003) for the task sizes that we could actually build in reality computation time never was an issue. In this setup the planner takes care of the combinatorial part of the problem, i.e., choosing and applying the different skills of the robot when they are needed. The robotics implementation only provides the individual actions for the robot.

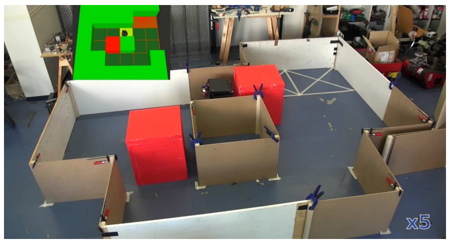


Figure 5.7: An example of our Turtlebot 2 solving a Sokoban task. The top left shows the internal world representation.

### Multi-Robot Exploration

We tackled the problem of multi-robot exploration with marsupial robots. Marsupial robots are robots that can carry other robots. In our case we considered two types of robots: Legged high-mobility robots that can traverse rough terrain, but slowly, and larger wheeled robots that carry the legged robots and can drive faster. Figure 5.8 illustrates this. Here the orange areas are rough terrain, white areas are flat terrain, red dots indicate an exploration frontier, and green dots are meeting points that lie between rough and flat terrain to allow legged robots to be loaded and unloaded on carrier robots. The task is to generate parallel minimal time plans that explore all frontiers. Therefore it is necessary to combine the different robot types according to their functionality while considering the dependencies between both. For example, a carrier robot could choose to explore a frontier in the white area or instead drive to a meeting point to pick up a legged robot that is waiting. These different action types and interactions between robots make it well suited as a planning problem. Although our experiments were mainly executed in simulation and thus the planning domain assumes deterministic action outcomes we also face issues present in real-world planning tasks. A basic property of any meaningful exploration task is that the world is not fully observable. That means when new areas are uncovered exploration frontiers shift, new frontiers might be discovered, or explored frontiers disappear. These partially observable tasks with open domains were addressed by embedding the planner TFD/M in a continual planning loop. As we are dealing with a multi-robot problem, in this setting we enabled the temporal planning feature of TFD/M that allows to plan with concurrent actions. Our results indicate that a planning-based solution outperforms a hand-scripted approach (Wurm et al., 2010, 2013). In the next chapter we also address the related problem of coverage search in simulation and with real robots.

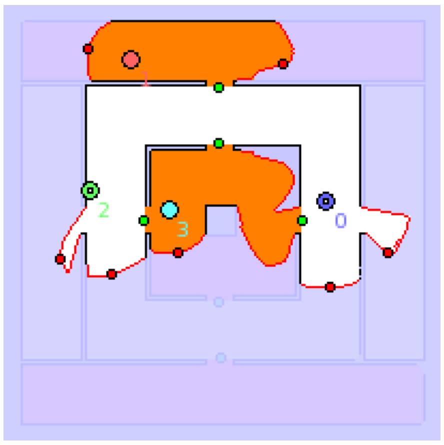


Figure 5.8: An example task of multi-robot exploration with marsupial robots.

## 5.7 Conclusion

In this chapter we considered planning techniques for real-world applications. As a first step we formulated planning tasks as open, partially-observable, and nondeterministic. Not surprisingly, it is impossible to guarantee that an arbitrary real-world task is solved. As we are aiming for efficient solutions to this problem, we simplified the generic problem to a classical planning problem running within a continual planning loop. We stated what soundness and completeness means in this case and have given explicit conditions under which we retain these properties. We also addressed efficiency in terms of computation time by introducing novel techniques for lazy module evaluation and caching of geometric computations.

We applied our planning techniques in the *TidyUp-Robot* system to demonstrate how a complex mobile manipulation system is created and to evaluate our approach. The robot was able to successfully handle unexpected events and adapt to new situations by replanning. With this implementation we have shown the feasibility of our simplifications in a real-world scenario. Thus, at least for this kind of setting our assumptions are reasonable. We also evaluated this system quantitatively. Observed runtimes were in the tens of seconds up to some minutes. The presented efficiency techniques were able to reduce planning times to a good balance between planning and execution time. When judging how fast or slow the system is keep in mind that an automated vacuum robot like a Roomba is also a lot slower than a human. Still, the work a human has to do to keep the room clean is less when using a robot.

Lazy module evaluation notably reduces planning time. The largest improvement in caching methods was observed when using partial state caching instead of full state caching. Subsumption caching works, but its efficiency is dependent on the ability to create a subsumption hierarchy and thus shows mostly in complex tasks. Global caching is complementary to other caching strategies and has proven useful for longer tasks with numerous replanning steps. In this work, we looked at avoiding module calls. State of the art search-based planning relies on well informed guidance heuristics. Making these aware of external computations in a generic manner, for example by relaxed operators, could lead to further improvements on the planning side.

We focused our evaluation on mobile manipulation tasks with the PR2 robot, but have also applied our system to other robots and settings. Using a domain-independent planner proved to be advantageous whenever heterogeneous skills of robots needed to be combined to reach a goal, especially when it is not always clear what the next action must be, in contrast to, e.g., a cooking recipe. Integrating different robot skills with the planner is not always straight-forward and requires a user to match the action execution, the declarative part of the PDDL/M domain, semantic attachments associated with an operator, and state creators. On the side of the planner interface we already introduced the object-oriented planning language OPL (Hertle et al., 2012). Beyond OPL, we believe it will be beneficial for consistency to integrate action execution and state creation similar to the module interface into a common description language.

Given that there cannot be a system that guarantees to always reach the goal for any possible task, what could be a best-case approach? From the planning perspective the answer would be: Use a planner for partially observable nondeterministic planning tasks (POND). Besides the increased computational demands, unfortunately such an approach cannot do anything reasonable when there is no guaranteed solution. A way out might be to use a probabilistic formulation like partially observable Markov decision processes (POMDPs) to get an "as safe as possible" solution without guarantees. We compared both approaches with classical planning and in terms of computation time our initial results indicate that either approach might even be feasible in practice (Hertle et al., 2014).

However, another question is: What is our desired robot behavior? Here, we come back to our initial reference, which is how a human would behave. Unless there are hard requirements like “I need to catch this train to reach my flight”, humans tend to prefer an efficient execution over a cautious one. For example, how many objects should a robot take from a dish washer to bring to a cupboard in another room if it believes that the door is open? For a human this is easy: As many as you can carry. If the door is closed, a human can put the objects down somewhere nearby or call for help or even use elbows or knees. In other words: We can improvise. Humans only consider multiple contingencies when the most-likely outcome is not clear. This *does* work in reality and we believe being able to improvise would be a promising approach for reliable robots in real-world scenarios. Task planning is actually very well suited for this as improvisation is just another plan with different actions. The main problem here is that the robot has to have the skills to do so. For example, when trying to pick an object up topples it instead, the robot must still be able to pick the object up possibly regrasping it. What this kind of improvisation skills would provide is making sure that our assumption that we can always somehow reach the goal holds.

# Chapter 6

# Multi-Robot Coverage in 3D

Coverage search is a robotics problem relevant for many real-world scenarios and applications. These range from household robots searching for objects, area inspection, such as searching for leaking pipelines or cracks in walls, up to searching for survivors in debris after a disaster in Urban Search And Rescue (USAR). In challenging areas like USAR this is often a truly three dimensional problem, i.e., survivors can be enclosed within complex and heavily confined 3d structures and sensors are mounted on a manipulator arm that reaches arbitrary poses within its workspace. Benchmarks for autonomous rescue robots, such as those proposed by NIST (Jacoff et al., 2003), simulate such situations in so called "rescue arenas" using artificially generated rough terrain, where victims are hidden in crates only accessible through confined openings. The real-world test environment used in the experiments for this chapter also is a three dimensional environment, where robots traverse to elevation levels that enable different view angles. An algorithm must take the environment and the degrees of freedom of the robots—or more precisely of the sensors—into account. Figure 6.1(a) shows our test area and Figure 6.1(b-f) shows a sequence of sensor locations from which a robot could observe part of the environment.

The primary goal of *3d coverage search* is to compute a sequence of sensor locations that can be visited by mobile robots on shortest paths and from which their sensors will have seen all areas of interest in a known 3d environment. We assume that the sensor has a specific field of view with an opening angle and a detection distance that resembles that of most IR and regular cameras. This is a similar problem to *coverage planning*, where a robot is required to pass over all points in a given environment (LaValle, 2006). However, in these applications the footprint of the robot does not change, while in 3d coverage search the sensor footprint, i.e., what is visible by a sensor, can vary dramatically with a small change in its pose. The goal of coverage planning is to compute an optimal shortest motion strategy in order to cover a planar environment with mobile robots. Solutions to this problem often employ cellular decompositions of the free space in 2d that are then input into a simple graph search (Choset, 2001).

In contrast, for our 3d problem we rely on sampling since decompositions of 3d spaces are very costly to compute. Graph search is also not the best option any more

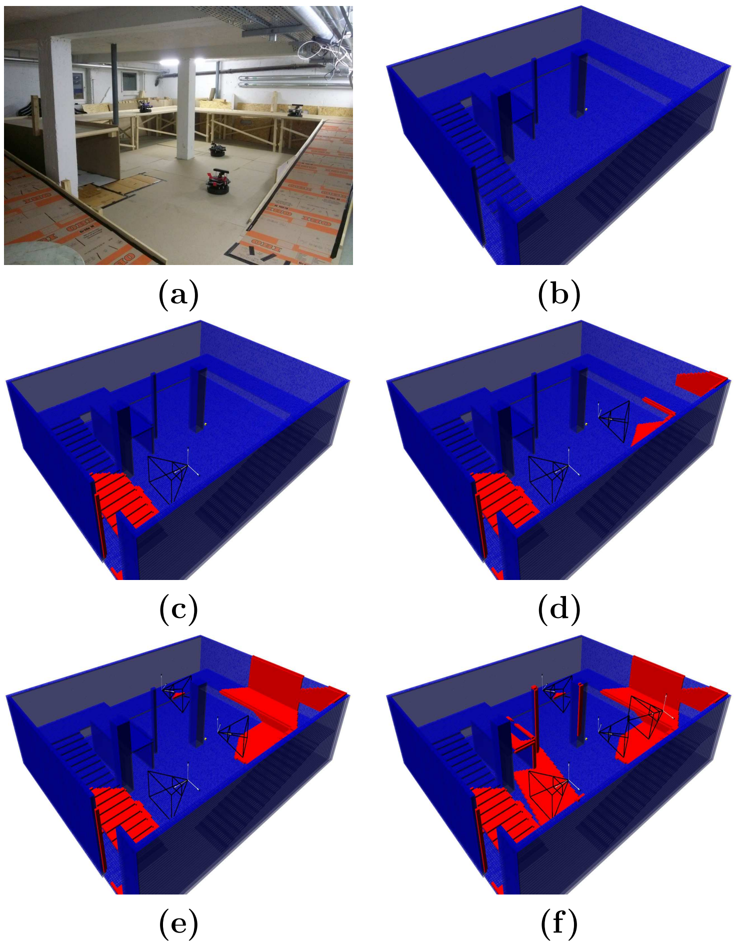


Figure 6.1: Motivating example: A progressive search of our test environment by visiting sensor locations in sequence. Unexplored area (blue) is stepwise covered (red) by sensors of the mobile robots.

as we have to consider that different views might overlap each other and thus might be redundant. As we also aim for multi-robot solutions, even after a decomposition into single views we are still facing a hard combinatorial problem. Note that our 3d coverage problem is also different from coverage problems that deploy multiple sensors to continuously monitor the same area for extended periods of time (Golovin and Krause, 2011). In our case every part of the environment has to be seen at most once and does not have to be observed continuously. Other coverage-related problems often deal with unknown or partially known environments in 2d (Kollar and Roy, 2008; Bourgault et al., 2002) and attempt to improve the map or explore unknown parts of the map. In our case the 3d environment is known and we are foremost interested in computing and visiting a set of useful sensor locations in the shortest time possible with paths that are feasible to execute for real systems. We will review the literature relating to coverage search in more detail in Section 6.1.

The purpose of the method presented in this chapter is to provide a practical solution to 3d coverage search that can be feasibly deployed. Our solution copes with realistic sensor models and the complexity of visibility in 3d, as well as time constraints of robots navigating on rough terrain. Given that the configuration space for most sensors has at least six dimensions (the pose), the problem is not suited for the kinds of decomposition approaches that work well in 2d. Therefore we use a sampling-based approach that generates a large number of sensor poses so that the views from these poses cover the 3d environment of interest. The sampling space is computed as part of the algorithm and generates poses that are likely feasible and that observe volumes to be covered. The sampled set of sensor poses might result in many different set covers of the environment when selecting sensor views from these poses. This leaves us with a combinatorial problem that can be formulated as a planning problem that selects a good set cover and sequences of poses that are visited in a short time. We use multiple different algorithms to solve this. These include greedy algorithms and a decomposition of the complete planning problem into a set cover and traveling salesman problem. One of the key advantages of our approach is that the input size to the planning problem is significantly reduced by the sampling of sensor poses. In addition, we use a *minimal partition* introduced in detail in Section 6.4 that further reduces the input size.

The sampling-based candidate view generation is based on our earlier work on computing a next best view pose for exploration in 3d (Dornhege and Kleiner, 2011, 2013). We describe this procedure to give an illustration of the complete algorithm for coverage search in 3d, but then focus on the solution of the combinatorial multi-robot problem given these views. In our prior work for this problem we presented results for the case of a single robot (Dornhege et al., 2013b). Here we determined that already with a single robot the overall problem is intractable, which is not surprising since our formulation contains subproblems that require solutions to the *set cover problem* (Karp, 1972) and the *traveling salesman problem* (Applegate et al., 2007), two well-known NP-hard problems. These results, however, indicated that some of the algorithmic variants of our approach already perform reasonably well, both in terms of the time to compute solutions and their cost, i.e., the estimated travel time of real robots executing the solution. In this chapter we present the generalized and extended approach (Dornhege et al., 2015) that tackles multiple robots with the goal to reduce the total concurrent execution time. This includes a generically applicable procedure that we use to convert single-robot solutions into multi-robot solutions, as well as extensions to our greedy algorithms that consider the multi-robot case directly. In addition to extensive experiments on 3d maps collected from real environments, we also present real robot experiments with four robots and demonstrate the applicability and feasibility of our approach under realistic conditions.

The remainder of this chapter is summarized as follows. In Section 6.2 we define the 3d multi-robot coverage search problem. Section 6.3 presents the generation of candidate views by sampling a set of high utility sensor poses. These views are then used to generate a minimal partition, which is described in Section 6.4. Section 6.5 shows how we efficiently plan sequences of sensor poses with short travel times using multiple algorithms. Greedy algorithms for single robots are presented in Section 6.5.1 and Section 6.5.2 describes a formulation of this as a planning problem, where we also show how this can be decomposed into a set cover and traveling salesman problem. Section 6.5.3 extends the greedy approach to multiple robots, while in Section 6.5.4 we adapt the single-robot algorithms to the multi-robot case by splitting single-robot solutions into multi-robot solutions. We demonstrate the feasibility of our approach by evaluating it in an extensive series of experiments in Section 6.6. These include experiments on real 3d maps in Section 6.6.2 comparing our various algorithms and the results of real-world experiments with up to four robots in a two-story scenario shown in Section 6.6.4. The latter verify the results of the simulation experiments and provide further insight about problems that are relevant to address for efficient execution in realistic scenarios. A detailed discussion is presented in Section 6.7 and we conclude in Section 6.8.

## 6.1 Related Work

There are a large number of variations of coverage problems and a vast literature on the topic. We review the work that is most closely related to our 3d coverage search problem. Much of the literature on coverage in robotics is concerned with approaches for distributing a team of robots to cover an environment continuously, as one would do for environmental monitoring and surveillance applications. This is known as the *area coverage problem* (Howard et al., 2002), but also often referred to simply as the coverage problem (Cortes et al., 2002). Our problem of coverage search is more closely related to *coverage planning*, which is motivated by applications such as vacuum cleaning, lawn mowing, farming, or demining. In coverage planning the goal is to compute an optimal motion strategy that visits every location in the environment at least once (LaValle, 2006; Choset, 2001). Coverage planning for finding optimal shortest paths is NP-hard due to the similarity to the traveling salesman problem, which also appears as a special case of our 3d coverage search problem. Solutions to coverage planning generally rely on exact or approximate cellular decompositions of the environment (Choset, 2001), which then enables to plan in the resulting graph structure. The kinds of decompositions used for this approach vary from spanning trees to boustrophedon decompositions with different properties regarding practicality and optimality.

Most of the work on coverage planning was concerned with 2d environments. The work by Renzaglia et al. (2011) considers 2.5d environments and applies optimization techniques to compute 3d paths for unmanned aerial vehicles. The emphasis, however, lies on the application of an optimization technique in order to maximize area coverage in an unknown environment for the current situation. To the best of our knowledge most of the prior work on coverage planning is restricted to 2d environments. It is a considerable challenge to adapt solutions that are based on decompositions of an environment to 3d environments. Perhaps the best illustration of this appears in the work of Lazebnik (2001). Therein the 2d visibility-based pursuit-evasion problem (Sachs et al., 2004) is generalized to 3d and the resulting complications are staggering. Needless to say not much progress has been made on this problem since then.

Other than the extension to 3d a key distinction between coverage search and coverage planning is that the footprint of the robot in coverage planning is static. For example, a vacuum robot does not change its shape when moving, while the visible volume of a camera varies wildly. Hence the distinction between *coverage planning* and *coverage search*. As a consequence of this sensor model we have to consider 3d visibility. Note that 2.5d visibility as in the work of Renzaglia et al. (2011) is still far simpler than 3d visibility and we are not aware of any prior work that considers 3d visibility for coverage planning. Here we partly rely on our previous work in which we extended the well known problem of frontier-based exploration on 2d grid maps (Yamauchi, 1997) to 3d environments (Dornhege and Kleiner, 2011, 2013) by computing so called frontier-voids.

Most closely related to our 3d coverage search problem is the work by Englot and Hover (2013). They address coverage planning of a ship hull, while also taking into account visibility by ray casting. Candidate views are generated by local sampling around geometric primitives to observe that are given as an input. These are collected into a roadmap. The only algorithm they consider to compute a coverage plan is the subsequent application of set cover and traveling salesman. They apply an iterative traveling salesman variant that enables the lazy evaluation of paths between view poses as the time to compute feasible paths is comparably large in their setting. This results in a less generic formulation than our work, but allows to derive theoretical completeness results and optimized trajectories for the specific scenario.

Coverage search also relates to research that is concerned with the computation of views and visibility, such as next best view approaches, the art gallery problem, and pursuit-evasion problems. Traditional next best view (NBV) algorithms compute a sequence of viewpoints until an entire scene or the surface of an object has been observed by a sensor (Banta et al., 1995; Gonzalez-Banos et al., 2000). These methods are, however, not suitable for coverage search on mobile robots since they ignore the costs for changing between different sensor poses (Gonzalez-Banos et al., 2000). Our previous work dealt with frontier-void based exploration in 3d (Dornhege and Kleiner, 2011). We focused on finding cells at the exploration frontier with good views into unknown parts of the environment, so called voids. Computing visibility is also addressed by the art gallery problem (Shermer, 1992). There the problem is to find an optimal placement of guards on a polygonal representation of 2d environments so that the entire environment is continuously observed by the guards. The emphasis is on the placement of guards with relation to the complex geometric features of the environment and the problem is known to be NP-hard. Pursuit-evasion problems relax this requirement since the goal is not to keep a static coverage of the environment, but to search for moving targets (Chung et al., 2011). The environment can be observed in a dynamic fashion by placing and moving guards over time. This also requires the computation of the agent's field of view, which often induces a decomposition of the environment (Sachs et al., 2004). Again, much of this work focuses on 2d environments, although there are approaches for 2.5d environments (Kleiner et al., 2013; Kolling et al., 2010).

While area coverage, pursuit-evasion, and the art gallery problem are naturally multi-robot problems, our coverage search problem can be applied to a single robot or multiple robots, just as coverage planning. Multi-robot coverage in 2d has also been considered (Agmon et al., 2008; Kong et al., 2006). Rekleitis et al. (2004) extend a single robot approach for coverage planning to multiple robots. They use a boustrophedon decomposition developed for the single-robot case. The environment is unknown and robots have line of sight communication, which precludes a multiple traveling salesman approach (Bektas, 2006). Other multi-robot approaches for coverage planning in 2d are also discussed by Choset (2001).

Coverage search and exploration are also closely related, with the obvious difference that the environment to be explored is unknown. The literature for robot exploration is also vast, especially in 2d. Of particular interest in relation to our work are 3d exploration methods. For example, Nüchter et al. (2003) propose a method for determining the next scan pose of a robot for digitalizing 3d environments. They compute a polygon representation from 2d planes in 3d range scans by connecting detected lines and free space between detected lines. From this polygon potential next-best-view locations are sampled and weighted according to the information gain computed from the number of polygon intersections with a virtual laser scan simulated by ray tracing. An extended approach to this problem uses 2.5d elevation maps and 3d ray tracing (Joho et al., 2007).

## 6.2 Problem Definition

In this section multi-robot coverage search is formally described. We first introduce the robot model, their sensors, and the environment followed by the definition of the coverage search problem. We consider homogeneous mobile robot platforms, the searchers, each equipped with a 3d sensor in a bounded 3d environment $\mathcal{E} \subset \mathbb{R}^3$. The 3d sensor generates a view consisting of a set of $n$ 3d points $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$ with $\mathbf{p}_i = (x_i, y_i, z_i)^T$. We associate a view with the sensor state that generates it. A sensor state $\mathbf{x} \in X \cong \mathbb{R}^3 \times \mathbb{RP}^3$ (see (LaValle, 2006) for details) can also be written as a 6d pose $(x, y, z, \phi, \theta, \psi)^T$. Here $(x, y, z)^T$ denotes the translational part (the position) and $(\phi, \theta, \psi)^T$ the rotational part in Euler angles (the orientation). The possible sensor states and therefore the resulting views that can be obtained depend on the collision-free configurations $q \in \mathcal{C}_{\mathit{free}} \subset \mathcal{C}$ of a robot's configuration space $\mathcal{C}$. We make no assumptions regarding $\mathcal{C}$ and $\mathcal{C}_{\mathit{free}}$ other than that we have a function $IK : X \rightarrow \{0, 1\}$ with $IK(\mathbf{x}) = 1$ if there is a valid path in $\mathcal{C}_{\mathit{free}}$ for the robot that puts the sensor into state $\mathbf{x}$ and 0 otherwise[1]. This allows us to define the set of all reachable sensor states $X_{\mathit{reach}} := \{\mathbf{x} \in X \mid IK(\mathbf{x}) = 1\}$.[2] Note that we do not consider collisions between robots when determining $IK$ and $X_{\mathit{reach}}$, so that these are static. In our experiments we avoid collisions with a simple scheme that requires robots to wait when a collision with another robot is predicted. Depending on the robot type and their knowledge about the position of other robots, one may choose to use another deconfliction scheme or a multi-robot path planning approach.

We furthermore assume the existence of a function $\mathit{cost} : \mathcal{C}_{\mathit{free}} \times X_{\mathit{reach}} \rightarrow \mathbb{R}^+$ that returns the time to move a robot from one configuration to another, place the sensor in a desired state and record a view. We will refer to this also as travel time. A cost function could also incorporate additional criteria, such as risk of failure, detection by hostiles or other undesired consequences that would lead to increased costs. For our purposes we equate cost with travel time. Note that for most applications we can conflate $\mathcal{C}_{\mathit{free}}$ and $X_{\mathit{reach}}$ by mapping exactly one configuration onto each sensor state in $X_{\mathit{reach}}$. This effectively ignores additional degrees of freedom of a robot and with slight abuse of notation we can then write $\mathit{cost}(\mathbf{x_i}, \mathbf{x_j})$. This is now simply the time of moving from one sensor state to another. We apply this simplification to the remaining sections as it simplifies the presentation without directly impacting the applicability of our solutions. For applications in which the additional degrees of freedom can be exploited, e.g., highly redundant humanoid robots with cameras, one would have to consider these separately.

Finally, the goal of coverage search is to cover every point in a given search set $\mathcal{S} \subseteq \mathcal{E}$. This search set might contain only a small part or all of $\mathcal{E}$, depending on what we are interested in covering. For every sensor state let the detection set $D(\mathbf{x}) \subset \mathcal{S}$ be the set of points in $\mathcal{S}$ visible from $\mathbf{x} \in X_{\mathit{reach}}$. Note that there is a subtle formal difference between $D(\mathbf{x})$ and a view at $\mathbf{x}$. $D(\mathbf{x})$ is a 3d volume and a view is a discrete set of points in a 3d volume. In addition, $D(\mathbf{x})$ is restricted to the search set $\mathcal{S}$ while a view contains points from all of $\mathcal{E}$. Yet, in colloquial terms, we can think of a detection set as a view.

The multi-robot coverage problem for $N$ robots is to find and visit a sequence of sensor states for each robot $n \in \{1, \dots, N\}$ with length $m(n)$, written as $\mathbf{x}_1^n, \mathbf{x}_2^n, \dots, \mathbf{x}_{m(n)}^n$, so that the entire search space $\mathcal{S}$ has been seen and covered, i.e., $\bigcup_{n=1}^{N} \bigcup_{i=1}^{m(n)} D(\mathbf{x}_i^n) =$

[1]To make this definition complete we further make the usual assumption that a starting configuration $q_0$ is given or that $C_{\mathit{free}}$ is connected.

[2]Depending on the configuration space reachable states can be precomputed for efficient access during the search, e.g., using capability maps (Zacharias et al., 2007).

$\mathcal{S}$. In addition, the overall execution time needed to visit all sensor states given by

$$cost_{max} = \max_{n \in \{1,\dots,N\}} \sum_{i=1}^{m(n)-1} cost(\mathbf{x}_i^n, \mathbf{x}_{i+1}^n)$$

has to be minimized. We refer to travel time when we consider the time a single robot takes to travel between locations and *execution time* when referring to the maximum combined travel time across all robots, i.e., the time to execute the entire sequence for all robots in parallel.

## 6.3 Sampling High Utility Views

We now describe how to find sensor states from $X_{reach}$ that have large views volumes. First we compute a utility function $util : \mathcal{E} \to \mathbb{R}^+$ that identifies good $3d$ positions in $\mathcal{E}$, ignoring the orientation for now. An efficient sampling-based method can significantly decrease the number of states that have to be considered. In this spirit, we will compute *util* via sampling and then later use it to identify 3d poses from which a large part of $\mathcal{S}$ can be seen. These high-utility poses in $\mathcal{E}$ are then turned into sensor states, which will serve as a basis for the coverage search methods described in Section 6.5.

The representation of $\mathcal{E}$ is given in form of an efficient hierarchical 3d grid structure, known as *OctoMap* (Hornung et al., 2013). Therein our 3d search region $\mathcal{S}$ is tessellated into equally sized cubes. The minimum size of the cubes is typically chosen relative to the size of the target that one searches for, i.e., the size of the cubes should generally be smaller than the target. The implementation for OctoMap is based on an octree that represents occupied areas in a hierarchical manner. Free space, as well as unknown areas are encoded in the map.

We construct *util* in two steps, shown in detail in Algorithm 8 and briefly described below. First, for every $s \in \mathcal{S}$ we sample $k_{max}$ vectors that start at $s$ and go towards a random position in $pos(X)$, sampled using $getRandom(.)$. Here $pos(.)$ returns the position of a state, simply ignoring its orientation, or the set of positions for a set of states, respectively. These vectors are collected in $\mathcal{V}$. Second, for each vector $\langle s, dir \rangle \in \mathcal{V}$ we compute by using the ray tracing function $getGridCells(s, dir, s_r)$ the set of grid cells $\mathcal{GC}$ that are visible from $s$ in direction $dir$ up to the sensor range limit $s_r$. Ray tracing is performed efficiently on the OctoMap. Then, for each cell in $\mathcal{GC}$ that corresponds to a reachable sensor state the utility value is incremented by one.

We now obtain our set of sampled sensor states $\tilde{X}$, from which large parts of $\mathcal{S}$ are visible, as follows. First, we sample grid cells that correspond to points $(x, y, z)^T \in \mathcal{E}$ with a positive and large *util* value. Note that by construction these points are such that $(x, y, z)^T \in pos(X_{reach})$, i.e., they correspond to the positions of reachable sensor states. For each of these points we sample an orientation $(\phi, \theta, \psi)^T$, so that we obtain a full sensor state $\mathbf{x} = (x, y, z, \phi, \theta, \psi) \in \mathbf{X}$. If $IK(\mathbf{x}) = 0$, we discard this state.

**Algorithm 8** Construct *util*

```
1: procedure FindGoodViews($\mathcal{S}$)
2:     $\mathcal{V} \leftarrow \emptyset$
3:     // Sample random vectors from $\mathcal{S}$ into $pos(X)$
4:     for all $s \in \mathcal{S}$ do
5:         $k \leftarrow k_{max}$
6:         while $k \neq 0$ do
7:             $\mathbf{x} \leftarrow getRandom(X)$
8:             $dir = normalize(pos(\mathbf{x}) - s)$
9:             $\mathcal{V} \leftarrow \mathcal{V} \cup \langle s, dir \rangle$
10:            $k \leftarrow k - 1$
11:        end while
12:    end for
13:    // Accumulate utilities in $\mathcal{E}$
14:    for all $v = \langle s, dir \rangle \in \mathcal{V}$ do
15:        $\mathcal{GC} \leftarrow getGridCells(s, dir, s_r)$
16:        for all $gc \in \mathcal{GC} \cap pos(X_{reach})$ do
17:            $util(gc) \leftarrow util(gc) + 1$
18:        end for
19:    end for
20: end procedure
```

Note that for some robots, such as ground robots, the set of reachable sensor poses $pos(X_{reach})$ can be much smaller than $\mathcal{E}$ and our method samples only from this much smaller space. Now that we have sampled a sensor state $\mathbf{x}$ we compute its actual utility $U(\mathbf{x}) := |D(\mathbf{x})|$ by ray tracing the sensor's field of view and counting all visible octree voxels in $\mathcal{S}$. If $U(\mathbf{x}) \geq \epsilon$, for some given $\epsilon$, we add $\mathbf{x}$ to $\tilde{X}$. Once $\tilde{X}$ reaches a certain size, i.e., $N_{sensor} = |\tilde{X}|$ for a given value of $N_{sensor}$, we stop adding to $\tilde{X}$ and terminate the sampling.

The method described above for sampling sensor states with high utility is rather generic and can easily be modified in order to achieve additional objectives or bias the sampling. For example, to formally guarantee complete coverage of $\mathcal{S}$ with the sensor states from $\tilde{X}$ one could continue to sample poses with non-zero *util* values and incrementally add more views until $\mathcal{S}$ is covered, as done in the work by Kleiner et al. (2013). However, this requires that every part of $\mathcal{S}$ can be seen by some $\mathbf{x} \in X_{reach}$—a property required for the problem to be solvable that unfortunately can be violated in many practical applications. We make no assumptions that the environment data and maps were collected with the robot that is used for the search and thus the environment $\mathcal{E}$ and search set $\mathcal{S}$ can cover arbitrary non-reachable space. Requiring that a user ensures that $\mathcal{S}$ can be covered would in turn require the user to solve this problem, which is not a reasonable assumption. Therefore, from a practical perspective it is better to ignore completeness and implement a best effort that is robust and allows the user to increase $N_{sensor}$ to increase coverage and determine whether it is sufficient for the application.

The primary feature of our sampling approach is that we provide an efficient initial estimate of the utility of sensor states with the *util* function that can be used in a number of ways leading to small sets of useful sensor states $\tilde{X}$. In this chapter we only presented the most straightforward sampling that considers sampling the highest *util* poses with an actual utility of at least $\epsilon$ until we reach $N_{sensor}$ sensor states for $\tilde{X}$. Investigating the wide range of possible variations to this sampling method could be a fruitful area for further work. We now proceed to the next section that shows how to exploit the set $\tilde{X}$ for planning purposes by first constructing a partition of the detection sets corresponding to the sensor states.

## 6.4 Partition Induced by Views

Based on the sampled $\tilde{X} \subset X_{reach}$, which represents a number of high-utility sensor states, we will later seek to determine a smaller set that gives us sequences of sensor states $\{\mathbf{x}_1^n, \ldots, \mathbf{x}_{m(n)}^n\} \subset \tilde{X}$ whose detection sets cover all of $\mathcal{S}$, i.e., $\bigcup_{n=1}^{N} \bigcup_{i=1}^{m(n)} D(\mathbf{x}_i^n) = \mathcal{S}$. Rather than computing $\bigcup_{i=1}^{m(n)} D(\mathbf{x}_i^n)$ for different sequences, which is computationally expensive, we first compute a *minimal partition* of the search set $\mathcal{S}$ that is induced by the detection sets for a given set of sensor states $Q$ (in our case $Q = \tilde{X}$).

Figure 6.2 illustrates the reduction achieved by such a partition. Here the search set,

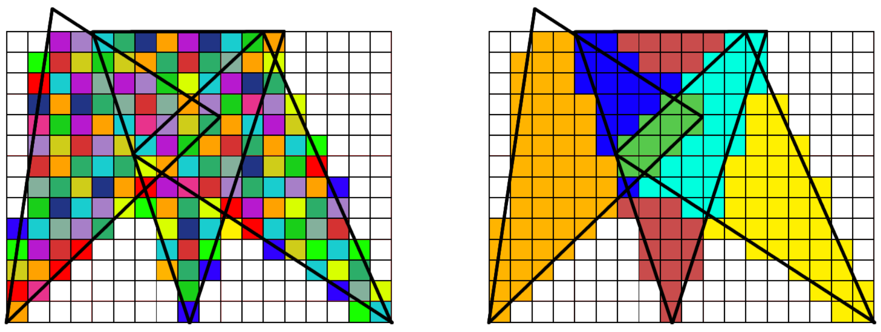

Detection Sets $D(x), \forall x \in Q$
Parts of $P(\mathcal{S}|Q)$.

Figure 6.2: This figure illustrates the effect of a minimal partition. The left shows the detection sets for three states, i.e., $|Q| = 3$, as black triangles. The right shows the minimal partition $P(\mathcal{S}|Q)$ with several parts, each represented with a different color.

$\mathcal{S}$, is the entire grid. The individual voxels that are part of the detection sets are shown in random colors on the left side, while white voxels are not part of any detection set. Let $\mathbf{x}$ be the state for the leftmost view. Then in the partition $P(\mathcal{S}|Q)$ shown on the right $P(\mathbf{x})$ contains three parts, the sets of blue, green, and orange voxels. Without partitioning one would have to merge detection sets using individual voxels like seen on the left in the example. A *minimal partition* collects all voxels that are contained in exactly the same detection sets into one partition part. We define the *reduction factor* achieved by a partition as the ratio of the number of individual voxels to the number of partition parts, i.e., for the example in Figure 6.2 it is $\frac{164}{6}\frac{\text{voxels}}{\text{partition parts}} \approx 27.3$. A minimal partition of $\mathcal{S}$ given $Q$ is defined as follows.

**Definition 37.** ***Minimal partition given a set of sensor states***
*Given any $Q \subseteq X_{reach}$ let $P(\mathcal{S}|Q)$ be a partition of $\mathcal{S}$ minimal for $Q$ defined as follows:*

1. $\emptyset \notin P(\mathcal{S}|Q)$
2. $\bigcup_{A \in P(\mathcal{S}|Q)} A = \mathcal{S}$
3. $\forall A, B \in P(\mathcal{S}|Q) : A \neq B \Rightarrow A \cap B = \emptyset$
4. $\forall \mathbf{x} \in Q, \forall A \in P(\mathcal{S}|Q) : A \cap D(\mathbf{x}) = A \vee A \cap D(\mathbf{x}) = \emptyset$
5. $\forall A, B \in P(\mathcal{S}|Q) : A \neq B \Rightarrow \exists \mathbf{x} \in Q : A \cap D(\mathbf{x}) \neq \emptyset \wedge B \cap D(\mathbf{x}) = \emptyset$

Conditions 1) to 3) state that $P(\mathcal{S}|Q)$ is a partition. Condition 4) states that every part of $P(\mathcal{S}|Q)$ is either entirely in a detection set or it does not intersect the detection set. Condition 5) states that $P(\mathcal{S}|Q)$ is minimal. With slight abuse of notation we

**Algorithm 9** Minimal Partition for $Q$

```
 1: $P(Q) \leftarrow \{\mathcal{S}\}$
 2: $\mathit{Views}(\mathcal{S}) \leftarrow Q$
 3: for all $\mathbf{x} \in Q$ do
 4:     $P(\mathbf{x}) \leftarrow P(Q)$
 5: end for
 6: for all $\mathbf{x} \in Q$ do
 7:     for all $A \in P(\mathbf{x})$ do
 8:         $A_{in} \leftarrow A \cap D(\mathbf{x})$
 9:         $A_{out} \leftarrow A \cap (\mathcal{S} \setminus D(\mathbf{x}))$
10:         if $A_{in} = \emptyset \vee A_{out} = \emptyset$ then
11:             continue // Condition 4. holds.
12:         end if
13:         $P(Q) \leftarrow P(Q) \setminus \{A\} \cup \{A_{in}, A_{out}\}$
14:         $P(\mathbf{x}) \leftarrow P(\mathbf{x}) \setminus \{A\} \cup \{A_{in}\}$
15:         for all $\mathbf{x}' \in \mathit{Views}(A) \setminus \{\mathbf{x}\}$ do
16:             $P(\mathbf{x}') \leftarrow P(\mathbf{x}') \setminus \{A\} \cup \{A_{in}, A_{out}\}$
17:         end for
18:         $\mathit{Views}(A_{in}) \leftarrow \mathit{Views}(A)$
19:         $\mathit{Views}(A_{out}) \leftarrow \mathit{Views}(A) \setminus \{\mathbf{x}\}$
20:     end for
21: end for
22: return $P(Q)$
```

will write $P(\mathbf{x}) \subset P(\mathcal{S}|Q)$ for all parts of $P(\mathcal{S}|Q)$ with $P(\mathbf{x}) \subset D(\mathbf{x})$. [3] Minimal partitions can be computed iteratively, as shown in Algorithm 9, when $Q$ is finite. In colloquial terms one can think of $P(\mathcal{S}|Q)$ as the Venn diagram of the search set and all detection sets for states from $Q$, i.e., of $D(\mathbf{x}_1), \ldots, D(\mathbf{x}_{|Q|})$, and $\mathcal{S}$. As a shorthand we also write $P(Q) = P(\mathcal{S}|Q)$ since $\mathcal{S}$ is given and fixed.

Algorithm 9 works as follows. First $P(Q)$ is initialized to the trivial partition $\{\mathcal{S}\}$. For each $\mathbf{x} \in Q$, we update $P(Q)$ by splitting all parts in $P(Q)$ that violate condition 4. This ensures that at termination condition 4 holds. As we only perform necessary splits, condition 5 also holds. Note that we only test against $P(\mathbf{x})$ instead of all parts in $P(Q)$ as required by condition 4. $P(\mathbf{x}) \subseteq P(Q)$ is maintained in addition to $P(Q)$ and only contains those parts that intersect with the detection set $D(\mathbf{x})$. Thus often $P(\mathbf{x}) \subset P(Q)$. We also maintain and update the inverse mapping *Views*$(A)$ for each part $A$ in $P(Q)$ to efficiently update $P(\mathbf{x})$.

[3] Notice that $\left(\mathcal{S} \setminus \bigcup_{\mathbf{x} \in Q} D(\mathbf{x})\right) \in P(\mathcal{S}|Q)$ is a part of the partition for all $Q$ that does not contain enough configurations to cover $\mathcal{S}$. From the sensors perspective this part is undetectable from the states in $Q$.

## 6.5 Multi-Robot Coverage Search

In this section we are concerned with the combinatorial problem of multi-robot coverage search, i.e., selecting a set of sensor states from $\tilde{X}$, assigning these to individual robots, and computing the shortest paths for each robot, with the goal of covering the entire environment in the minimal amount of time. Note that all the above steps have dependencies that complicate the problem, e.g., the quality of the shortest paths obviously depends on the assignment or shortest paths may lead to collisions in addition to obstructions of views by other robots. To be able to find feasible solutions in a reasonable time we treat individual robot paths as if they were independent from each other. This means that we will not consider multi-robot collision avoidance, view obstructions, or synchronizing parallel actions for multiple robots. These problems can be dealt with for each specific application and have varying degrees of impact, depending on the scenario. Hence, we believe these issues should be addressed in a particular implementation and its execution. Unless a particularly hard scenario is constructed, the interactions between the coverage search and the above issues should be minimal, especially considering that the robots should naturally spread to different parts of the environment.

We now introduce the two principal ideas we use for solving the multi-robot planning problem. The first is to create a high-quality single robot coverage plan and then divide this plan into smaller segments that are assigned to individual robots. For the single robot case we present variants of greedy algorithms as well as approaches based on task planning. The second idea is to adapt the single robot approaches directly to the multi-robot problem, which we do for the greedy procedures. In the following we thus first describe our algorithms for single robot coverage plans. We will then adapt these algorithms to the multi-robot case.

### 6.5.1 Single-Robot Greedy Solutions

From hereon, whenever we reason about detection sets, we will always use partition parts instead of individual voxels in our implementation. We utilize $P(\tilde{X})$, as well as the corresponding mappings $P(\mathbf{x})$, to construct a sequence $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \subseteq \tilde{X}$ that covers $\mathcal{S}$ and has a short execution time. Since we now consider the sequence of sensor states, each with an associated detection set and view, we define a new sequential utility function $U_i$ that reduces the original utility $U(\mathbf{x}) = |D(\mathbf{x})|$ by the volume in $\mathcal{S}$ that has been seen previously by any robot in the sequence:

$$U_i(\mathbf{x}) = \left| D(\mathbf{x}) \setminus \bigcup_{j<i} D(\mathbf{x}_j) \right|$$

All greedy algorithms compute the coverage sequence by repeatedly selecting a state $\mathbf{x} \in \tilde{X}$ according to some strategy until all parts in $P(\tilde{X})$ are covered. The greedy algorithms differ by how the next $\mathbf{x}$ is selected. For all algorithms let $UC$ be the uncovered parts initialized as $UC \leftarrow P(\tilde{X})$. For every $i = 1, \ldots, m$ we select a new $\mathbf{x}_i$ and update $UC \leftarrow UC \setminus P(\mathbf{x}_i)$. The algorithms terminate when $UC = \emptyset$. We now describe two greedy algorithms.

### Simple Greedy

The simplest greedy strategy selects a $\mathbf{x}_i$ that minimizes the travel time from the last view, i.e., $cost(\mathbf{x}_{i-1}, \mathbf{x}_i)$. The first view is chosen to be the one with maximum utility $U$. We denote this variant as *Simple-Greedy.*

### Greedy Next Best View

Our second greedy variant also considers the utility of views in addition to the travel time, i.e., it chooses a $\mathbf{x}_i$ that maximizes $U_i(\mathbf{x}_i)/cost(\mathbf{x}_{i-1}, \mathbf{x}_i)$. The idea is to prefer high-utility views that are also easily reachable. For this we choose the ratio of utility and travel time to quantify the tradeoff between the two. Again, the first view is chosen to be the one with maximum utility $U$. We denote this variant as *Greedy Next Best View (Greedy-NBV).*

## 6.5.2 Single-Robot Planning Solutions

Greedy algorithms provide simple and reasonably fast solutions that are also easily modified to consider additional criteria for specific application scenarios. These solutions, however, are usually not optimal. Therefore we also formulate the problem of finding a coverage sequence for a set of sensor states from $\tilde{X}$ as a classical planning problem that can be solved optimally. As finding optimal solutions to this problem is often infeasible, as discussed in further detail in Section 6.6.3, we also present a sub-optimal decomposition of the problem by solving a set cover and subsequent traveling salesman problem. We now give the problem formulation as a planning problem and afterwards show how this formulation is easily adapted to solve the decomposition.

### Optimal Planning Formulation

We model our problem as a planning task in PDDL/M introduced in Chapter 3. In our domain, there are two types of objects:

```
(:types view view_part)
```

An object of type `view` is added for each sensor state in $\tilde{X}$ and a `view_part` is defined for each part in the partition. Next, the planning state is given by the following logical predicates:

`(searched ?x`$_i$` - view)`

describes that a view $\mathbf{x}_i$ has already been visited and

`(covered ?`$p_j$` - view_part)`

states that part $p_j$ has already been covered in a state. `searched` and `covered` are initially set to `false` for all views and parts, respectively. The current view location of the robot is given by

`(at-view ?v - view)`

For each view $\mathbf{x}_i$ and each part $p_j$ the predicate

`(view-covers ?`$\mathbf{x}_i$` - view ?`$p_j$` - view_part)`

is set to `true` in the initial state, iff $p_j$ is in $P(\mathbf{x}_i)$. Only a single action is needed that searches the next view:

```
(:action search
  :parameters (?cur - view ?v - view)
  :duration (= ?duration [costSearch ?cur ?v])
  :precondition
    (and
      (not (searched ?v))
      (at-view ?cur)
    )
  :effect
    (and
      (searched ?v)
      (not (at-view ?cur))
      (at-view ?v)
      (forall (?_vp - view_part)
        (when (view-covers ?v ?_vp)
          (covered ?_vp))
      )
    )
)
```

The precondition prohibits the planner from choosing the same view twice. Accordingly `searched` is set in the effect. We also assign `at-view` to the view reached by the search action. The term `[costSearch ?cur ?v]` specifies a cost module that calls the *cost* function defined in Section 6.2. The `forall` statement defines a conditional effect that sets `covered` to true for a view part `when` it is contained in $P(\text{?v})$, i.e., when it is observed by this view.

Finally, we specify as the goal formula:

`(forall (?vp - view_part) (covered ?vp))`

This requires each part to be covered and thus guarantees that any plan found by a planner actually provides a coverage plan. As we use the actual *cost* function to define action costs, a shortest plan found by the planner also constitutes a minimum execution time coverage sequence, i.e., an optimal solution to our coverage search problem. We use our planner TFD/M to solve all planning tasks in this chapter. We denote this variant as *Complete Plan*.

### Decomposition into Set Cover and Traveling Salesman Problem

While the previous planning formulation can yield optimal solutions, the search space is still rather large and the optimal solutions may not be found in a reasonable amount of time for large problem instances. In this section we simplify the problem by decomposing it into a set cover and a traveling salesman problem. More precisely, we first find a minimal set of views that covers all parts of the partition, which is the classical *set cover* problem. We are looking for a minimum cardinality subset $Q_C \subseteq \tilde{X}$, so that all parts of $P(\tilde{X})$ are covered, i.e., that fulfills the following condition.

$$\bigcup_{\mathbf{x}_j \in Q_C} P(\mathbf{x}_j) = \bigcup_{\mathbf{x}_i \in \tilde{X}} P(\mathbf{x}_i)$$

We use the minimal partition to reduce the input size to the set cover problem by ignoring views covering unique parts of the search set that are not covered by any other view. We call these views *necessary*, since they have to be part of any cover, and then only determine the minimum set cover for the remaining views. The set cover problem is solved by a simple reformulation of the complete planning problem using the same planner. More precisely, action costs from the above definition are set to 1, so that the cost of a plan is identical with the number of views. These problems are solved quite fast (within seconds in all our experiments) as they contain a considerably smaller set of views and permutations do not need to be considered.

Given the minimum cardinality subset of views that covers the search space it only remains to find an optimal execution time sequence visiting all views. This is a *Traveling Salesman Problem* (TSP) without the requirement to return to the first location. We already have a PDDL formulation for the complete problem and can easily apply this formulation to the Traveling Salesman Problem by changing the goal formula to:

```
(forall (?v - view) (searched ?v))
```

This requires all views in the planning task, which are now only the views that are part of the minimal cardinality cover, to be visited and thus an optimal cost plan to this problem results in a minimum execution time path through all views. The coverage information can be safely ignored as that is already guaranteed by the set cover.

There exist efficient solvers specifically designed for the Traveling Salesman Problem and thus we investigate the application of the LKH solver (Helsgaun, 2000), an efficient implementation of the Lin-Kernighan heuristic to solve the TSP. When we use the TFD/M planner for solving the TSP in the decomposed formulation, we denote this variant as *Set Cover/TSP (TFD)*. When using the LKH solver we call the variant *Set Cover/TSP (LKH)*. TFD/M is always used to solve the set cover problem.

To summarize, the decomposition of the complete planning formulation into a set cover and a TSP problem is clearly suboptimal. Yet, it still requires solutions for two NP-hard problems. From a practical perspective, this decomposition allows the application of advanced solvers that have been developed specifically for these problems. In the experimental section we will investigate the tradeoff between the time to compute solutions and the quality of these solutions comparing the optimal and the decomposition approach.

### 6.5.3 Multi-Robot Greedy Solutions

In this section we adapt the single-robot greedy algorithms directly to the multi-robot case. As before we utilize $P(\tilde{X})$, as well as the corresponding mappings $P(\mathbf{x})$, to construct a sequence $\{\mathbf{x}_1^n, \ldots, \mathbf{x}_{m(n)}^n\} \subseteq \tilde{X}$ for each robot $n \in \{1, \ldots, N\}$ with length $m(n)$. The multi-robot greedy algorithms compute the coverage sequences by repeatedly selecting a specific robot $n$ and a view $\mathbf{x}^n \in \tilde{X}$ until all parts in $P(\tilde{X})$ are covered. Similar to the single robot case we define a sequential utility function $U_k$ that reduces the original utility $U$ by the volume in $\mathcal{S}$ that has been seen previously in any sequence for any robot. Here $k$ is the $k$-th step in the greedy procedure, when the $k$-th view is assigned. Let $m_k(n)$ be the sequence length for robot $n$ in the $k$-th step. Then the utility to choose the view from sensor state $\mathbf{x}$ in the $k$-th step is:

$$U_k(\mathbf{x}) := \left| D(\mathbf{x}) \setminus \left( \bigcup_{n=1}^{N} \bigcup_{j \leq m_{k-1}(n)} D(\mathbf{x}_j^n) \right) \right|$$

Note that this measure is independent of the robot. First, using this expected utility and the cost to reach the sensor state for a view, every robot is choosing its next preferred view at step $k$ exactly as in the single robot case. The preferred view is selected either using the *Greedy-NBV* or the *Simple-Greedy* equations from Section 6.5.1 leading to the *Multi-Greedy-NBV* and *Multi-Simple-Greedy* variants.

Let $\hat{\mathbf{x}}_k^n$ be the sensor state chosen by robot $n$ for step $k$. We now select the sensor state from the robot that leads to the minimum increase in overall execution time, i.e., we greedily select the robot with the shortest overall path to its preferred state. More precisely, let this robot $n^*$ be

$$n^* := \operatorname*{arg\,min}_{n \in \{1,\ldots,N\}} cost(\mathbf{x}_{m_{k-1}(n)}^n, \hat{\mathbf{x}}_k^n) + \sum_{i=1}^{m_{k-1}(n)-1} cost(\mathbf{x}_i^n, \mathbf{x}_{i+1}^n).$$

The view $\hat{\mathbf{x}}_k^{n^*}$ is then appended to the sequence of views for robot $n^*$ and we continue with the next step by incrementing $k$. As for the single robot case the procedure continues until all parts in $P(\tilde{X})$ are covered. Let $UC$ again be the uncovered parts initialized as $UC \leftarrow P(\tilde{X})$. For every $k$ we update $UC \leftarrow UC \setminus P(\hat{\mathbf{x}}_k^{n^*})$. The algorithm terminates when $UC = \emptyset$.

### 6.5.4 Multi-Robot Solutions From Single-Robot Solutions

The greedy spirit of the single-robot greedy algorithms was readily extendable to the multi-robot case by simply greedily selecting the best robot. The single-robot planning algorithms, however, are not extendable to the multi-robot case in such a straight forward manner. Therefore, our multi-robot planning approach starts with a single robot plan and splits this into $N$ parts to produce $N$ paths that will be executed by $N$ robots in parallel. Although this is clearly not optimal we gain the practical advantage of being able to use improved planners that work well for the single robot case without any additional effort, i.e., we can simply substitute a single robot planner with an improved version and simultaneously improve our multi-robot plans. This advantage should not be underestimated, especially for the development of practical and fielded systems.

This approach takes as the input a single robot coverage sequence $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \subseteq \tilde{X}$ that covers $\mathcal{S}$. Our goal is to split this into $N$ sequences $\{\mathbf{x}_1^n, \ldots, \mathbf{x}_{m(n)}^n\} \subseteq \tilde{X}$ for each robot $n \in \{1, \ldots, N\}$ with length $m(n)$. There are $N-1$ splitting points $s_n$ for $N$ robots. A splitting point $s_n \in \{1, \ldots, m\}, i < j \Rightarrow s_i < s_j$ defines the end index of the $n$-th robot's subsequence, so that $\mathbf{x}_{m(n)}^n = \mathbf{x}_{s_n}$. Starting points are one past the end of the previous robot's sequence, so that the subsequences connect, i.e., $\mathbf{x}_1^n = \mathbf{x}_{s_{n-1}+1}$. The first robot's subsequence must start with the first sensor state of the single robot plan, i.e., $\mathbf{x}_1^1 = \mathbf{x}_1$ and likewise the last robot's subsequence must complete the single robot sequence, so that all views are covered, i.e., $\mathbf{x}_{m(N)}^N = \mathbf{x}_m$. There are in the order of $m^{N-1}$ ways to perform such splits. As long as the number of robots $N$ is not too large it is feasible to enumerate all solutions, despite the exponential complexity in the number of robots. We do so and select the split that minimizes the overall execution time as our solution. If computation time is an issue, the method described above can be approximated, e.g., by k-means clustering of sensor states. In the experimental section, however, we will see that the complexity of this step plays a minor role in the overall computation time.

Applying the above splitting method to the single robot algorithms described in Sections 6.5.1 and 6.5.2 leads to new variants. To describe these we append a '-S' to the single robot variant name leading to the new variants *Simple-Greedy-S*, *Greedy-NBV-S*, *Set-Cover/TSP (TFD)-S*, *Set-Cover/TSP (LKH)-S*, and *Complete Plan-S.*

## 6.6 Experiments and Evaluation

We evaluated our approach and algorithms on four real-world data sets, i.e., OctoMaps obtained from sensor data collected by robots in real environments. In addition, we carried out real-world experiments in which robots execute the computed solutions for 3d coverage search. The results of the experiments on the data sets are presented in Section 6.6.2 and results of the real-world experiments are found in Section 6.6.4.

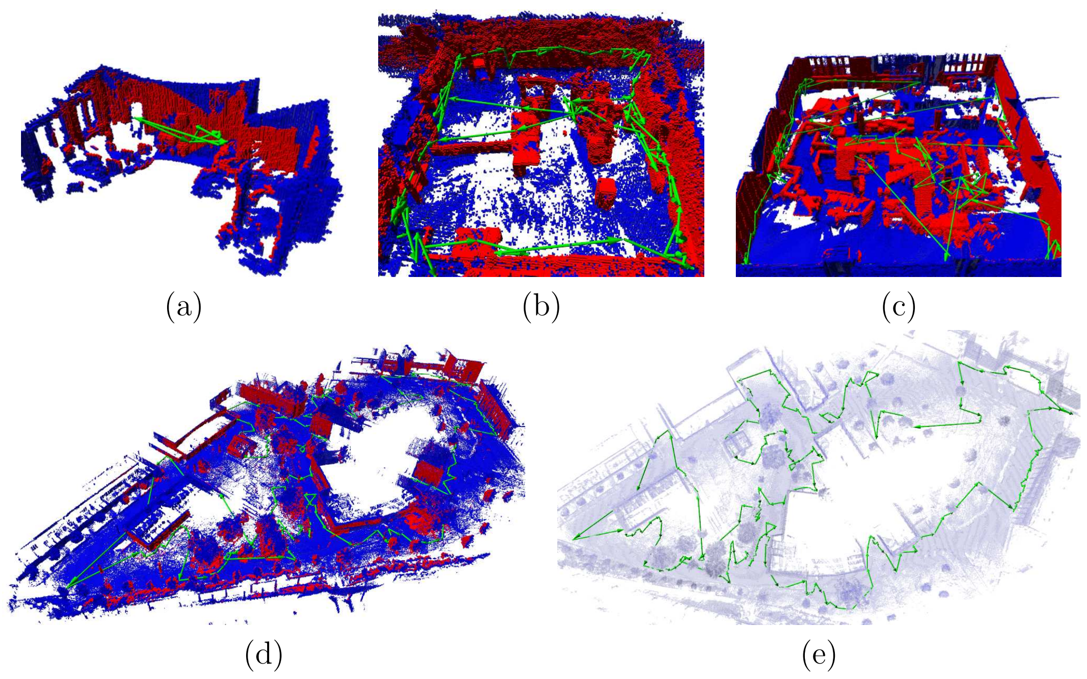


Figure 6.3: This figure shows the data sets used for evaluation: The 3d scan in our lab (a), Building 78 (b), the rescue arena (c) and the Computer Science Campus at the University of Freiburg (d). For a better visualization of the plan in the campus a top-down view is given (e). The shown plans (green) are generated with the Set Cover/TSP (LKH) method for one robot. Occupied cells are displayed blue, covered cells red.

Section 6.6.1 discusses how we compute travel times efficiently, while Section 6.6.3 briefly discusses optimality for the planning problem.

The four data sets represent one small indoor, two large indoor and one large outdoor environment. The small indoor data set consists of a 3d scan taken in the robotics lab at the University of Freiburg. The first large indoor data set was recorded in Building 78 at the University of Freiburg, which consists of two rooms separated by a door. The second one was recorded in the rescue arena at Jacobs University in Bremen. The large outdoor data set was recorded on the Computer Science Campus at the University of Freiburg. OctoMaps for the indoor data sets are generated with 5 cm resolution while the outdoor data set uses a resolution of 20 cm. Visualizations of the maps obtained from the data sets are shown in Figure 6.3.

In all simulation experiments the search set $\mathcal{S}$ to be examined consists of all vertical structures of the map, thus aiming for a complete coverage of everything that is not floor or ceiling. This choice models an inspection task for inspecting walls. For our purposes it serves to have a large search set $\mathcal{S}$ in a larger environment $\mathcal{E}$, but within a similar order of magnitude. The robot model used is a mobile ground robot with

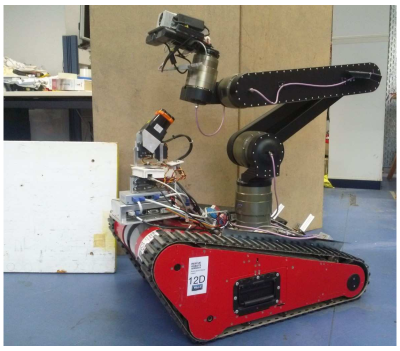


Figure 6.4: An example for a robot platform used for coverage search. Sensors are mounted on a versatile manipulator arm.

the sensor mounted on a 6-DOF manipulator with a reach of one meter. The sensor model is a camera with 60 degrees horizontal and 40 degrees vertical field of view. For the indoor data sets a maximum range of five meters was used, the outdoor data set was searched with a 35 meter maximum range. This searcher model is motivated by a common setup for USAR robots depicted in Figure 6.4: A tracked robot with a manipulator arm as a sensor platform. The small indoor map, denoted as *Lab*, was used for two scenarios. The first scenario, denoted by *Lab 1*, only allowed manipulator movements and no motion of the ground platform. The map is sufficiently small, so that the manipulator has a reasonably large $X_{reach}$. The second scenario, denoted by *Lab 2*, allowed manipulator movements and navigation for the ground platform. In all other maps we considered manipulator movements and navigation for the ground platform.

The variants of our algorithm, described in detail in Section 6.5, that we used for the experiments are Multi-Greedy-NBV, Greedy-NBV-S, Multi-Simple-Greedy, Simple-Greedy-S, Set-Cover/TSP (TFD)-S, Set-Cover/TSP (LKH)-S, and Complete Plan-S. For the TFD/M planner we used anytime search that stopped after twice the time it took to find the first valid plan. The LKH solver was executed with the default parameters supplied by the software. All greedy variants of the algorithm were run until a solution was found.

## 6.6.1 Efficient travel time computation

As noted in the problem definition in Section 6.2 we require the computation of time estimates for moving between different sensor states on the map, i.e., the function $cost(\mathbf{x}_i, \mathbf{x}_j)$. A major part of the computation time is spent when computing the time to navigate between poses. We use value iteration, a popular dynamic programming algorithm frequently used for robot planning (Burgard et al., 1998). As shown in Figure 6.5 the planner takes as input a segmented elevation map in which important structural elements such as stairs and ramps are discriminated and indicated by a different color. Value iteration computes for each grid cell on the map a time estimate for reaching a goal cell. These time estimates are composed of travel distance as well as costs for overcoming different types of terrain, such as flat ground, ramps, or stairs. The resulting value function is then used by an $A*$ planner as the heuristic for finding

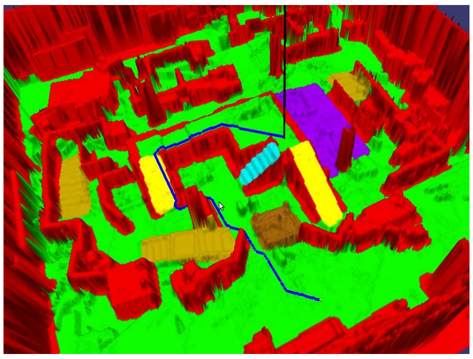


Figure 6.5: The computation of travel time on segmented elevation maps considers different structural elements such as flat ground (green), walls (red), stairs (light blue), or ramps (yellow).

shortest paths on the map.

Besides these travel times between robot base poses, we are also considering the time for moving the manipulator from one view configuration to the next based on the maximum angular displacement of any joint and the maximum angular velocity for that joint. The expected combined time defines the cost of moving between two sensor states and hence we obtain $cost(\mathbf{x}_i, \mathbf{x}_j)$.

## 6.6.2 Evaluation of Coverage Search Algorithms

The first series of experiments applies all variants of our multi-robot coverage search algorithm to the scenarios generated from the real-word data sets, using one to four robots. For these experiments we are reporting computation times and the planned execution times, i.e., the cost of the best plans found by the algorithms on each of the maps. Since the generation of views for $\tilde{X}$ involves randomization, we ran the algorithm ten times for each scenario and report mean values with standard deviation. The minimum utility $\epsilon$ to accept a view as well as the number of views, $N_{sensor}$, that need to be generated for $\tilde{X}$ was chosen with respect to the average expected utility which depends on the sensor model and the environment. Therefore, for the outdoor environment with a 35 meter sensor, this minimum needs to be significantly higher than the one chosen for an indoor environment with limited field of view. The choices of parameters are shown in Table 6.1. Table 6.1 also shows computation times for the first part of our algorithm, i.e., the generation of views for $\tilde{X}$ and the minimal partition. In addition, we provide the *reduction factor* achieved by the partition. This reduction factor introduced in Section 6.4 is defined as the ratio of cells in $\mathcal{S}$ to parts in the partition. It provides a measure for the reduction of the search space achieved by the sampling of $\tilde{X}$ and its minimal partition. With a large reduction factor the number of parts in the partition is significantly smaller than the number of cells that are to be covered and the input size to the planning algorithms can be thought of as being reduced by this factor.

| Scenario | Lab 1 | Lab 2 | Bldg. 78 | Arena | Campus |
|---|---|---|---|---|---|
| Computing $\tilde{X}$ [s] | $2.1 \pm 0.2$ | $3.0 \pm 0.5$ | $30.8 \pm 2.5$ | $79.0 \pm 4.9$ | $201.9 \pm 10.1$ |
| Minimal Partition [s] | $0.1 \pm 0.0$ | $0.0 \pm 0.0$ | $2.2 \pm 0.3$ | $5.5 \pm 0.7$ | $31.5 \pm 2.8$ |
| Reduction Factor | $127.2 \pm 17.1$ | $49.3 \pm 12.5$ | $16.5 \pm 1.9$ | $37.0 \pm 4.6$ | $71.6 \pm 4.2$ |
| $\epsilon$ [dm$^3$] | 150 | 150 | 125 | 250 | 8000 |
| $N_{sensor}$ | 15 | 15 | 100 | 142 | 280 |

Table 6.1: This table shows computation times for the generation of $\tilde{X}$ and its minimal partition. In addition the reduction factor achieved by the partition is given, as well as the parameters $\epsilon$ and $N_{sensor}$ that were used.

Each of the ten $\tilde{X}$ and minimal partitions obtained for every scenario were used as an input to the variants for the coverage search presented in Section 6.5. This was done with one, two, three and four robots, i.e., $N = 1, \ldots, 4$. Table 6.2 shows the average plan cost for a solution for the smaller Lab scenarios, where all algorithms performed similarly well. Figures 6.6 and 6.7 plot the resulting cost for each variant and number of robots for the Bldg. 78, Arena, and Campus scenarios. Table 6.3 shows the average measured computation time required to compute the solutions for each variant, number of robots, and scenario.

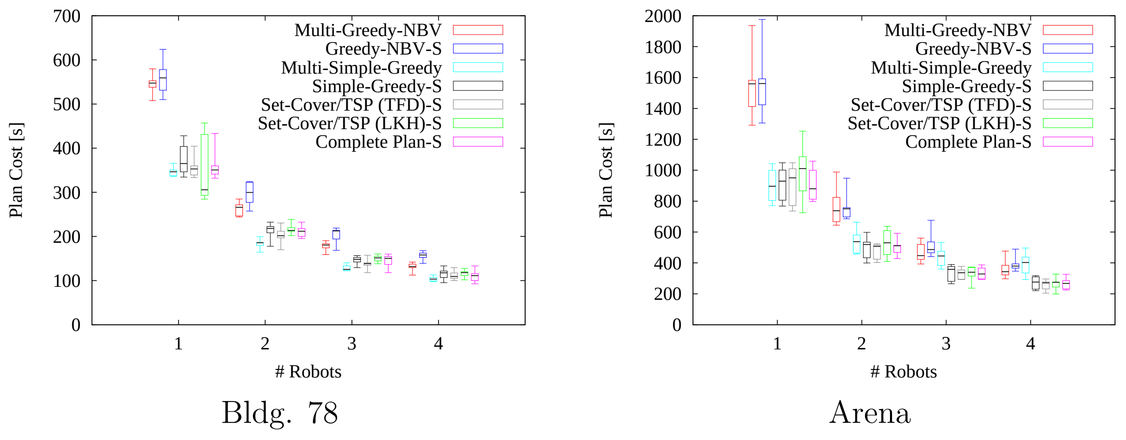


Figure 6.6: Box plots of the plan cost for $N = 1, \ldots, 4$ of multiple algorithm variants for the Bldg. 78 and Arena scenarios.

| Scenario | Lab 1 | | | |
|---|---|---|---|---|
| Num Robots | 1 | 2 | 3 | 4 |
| Multi-Greedy-NBV | 25.7 ± 3.2 | 12.6 ± 2.3 | 9.1 ± 1.5 | 7.5 ± 0.8 |
| Greedy-NBV-S | 28.8 ± 3.5 | 17.3 ± 3.9 | 10.1 ± 1.6 | 9.8 ± 2.1 |
| Multi-Simple-Greedy | 25.9 ± 3.3 | 13.9 ± 2.2 | 10.0 ± 1.3 | 8.0 ± 1.2 |
| Simple-Greedy-S | 30.3 ± 7.3 | 15.8 ± 4.2 | 11.9 ± 3.1 | 9.8 ± 2.1 |
| Set-Cover/TSP (TFD)-S | 23.4 ± 7.6 | 11.9 ± 3.4 | 9.9 ± 2.6 | 8.0 ± 2.1 |
| Set-Cover/TSP (LKH)-S | 21.6 ± 5.4 | 14.9 ± 3.5 | 9.3 ± 2.5 | 8.1 ± 1.9 |
| Complete Plan-S | 31.7 ± 6.0 | 17.0 ± 4.0 | 12.0 ± 3.3 | 9.9 ± 2.8 |
| Scenario | Lab 2 | | | |
| Num Robots | 1 | 2 | 3 | 4 |
| Multi-Greedy-NBV | 60.4 ± 6.9 | 31.4 ± 3.8 | 22.0 ± 2.1 | 17.3 ± 1.8 |
| Greedy-NBV-S | 66.6 ± 8.4 | 38.0 ± 2.6 | 27.8 ± 3.2 | 21.1 ± 2.1 |
| Multi-Simple-Greedy | 50.5 ± 6.9 | 29.6 ± 2.9 | 22.8 ± 2.3 | 19.3 ± 1.9 |
| Simple-Greedy-S | 57.1 ± 5.9 | 33.9 ± 5.1 | 26.4 ± 4.1 | 20.9 ± 2.5 |
| Set-Cover/TSP (TFD)-S | 54.3 ± 11.6 | 35.7 ± 5.5 | 25.1 ± 4.0 | 21.4 ± 3.9 |
| Set-Cover/TSP (LKH)-S | 60.7 ± 10.0 | 35.0 ± 5.1 | 26.5 ± 4.7 | 22.3 ± 3.2 |
| Complete Plan-S | 56.1 ± 12.1 | 33.8 ± 6.9 | 25.0 ± 3.2 | 20.8 ± 3.8 |

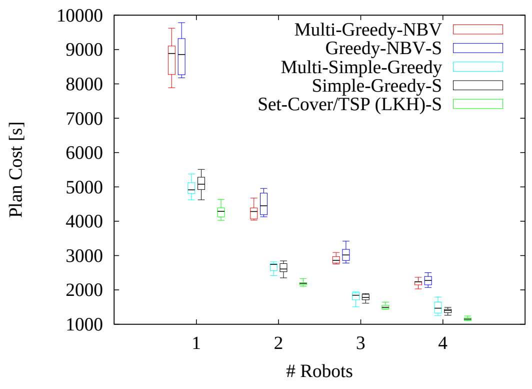
Table 6.2: This table shows plan cost in seconds for the Lab 1 and Lab 2 scenarios.

For the greedy variants, we observe that the Simple-Greedy variants usually perform better than Greedy-NBV. The balancing of travel time and utility of views in Greedy-NBV does not pay off in these experimental scenarios. One explanation for this effect is that the selection of views for $\tilde{X}$ is already biased towards high utility views and the additional consideration of utility penalizes travel times too much. This suggests that when considering to use utilities in a greedy approach, a different tradeoff equation than the simple ratio could be more beneficial. It is unclear however, which tradeoff can lead to a good greedy heuristic, and the simple greedy approach seems already to perform rather well.

Another interesting observation is found in the comparison of the '-S' variants of the greedy algorithms (Greedy-NBV-S and Simple-Greedy-S) against the multi-robot greedy variants (Multi-Simple-Greedy and Multi-Greedy-NBV). For the medium sized scenario, i.e., Bldg. 78, the multi-robot variants perform slightly better (see Figure 6.6). This advantage, however, vanishes for the larger maps (Arena and Campus). The computation times for the multi-robot variants scale linearly with the number of robots (see Table 6.3). This is due to the fact that for each step the single robot variants compute the travel times to all other views, while the multi-robot implementation computes travels times to all other views *for every robot.* This linear growth in the number of robots has a much larger impact on computation time than the exponential growth in the number of robots that is due to the splitting for the '-S' variants. As it happens in real applications the constants hidden in the complexity classes can

| Num Robots | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Scenario | Lab 1 | | | |
| Multi-Greedy-NBV | 0.0 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 |
| Greedy-NBV-S | 0.0 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 |
| Multi-Simple-Greedy | 0.0 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 |
| Simple-Greedy-S | 0.0 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 | 0.1 ± 0.0 |
| Set-Cover/TSP (TFD)-S | 2.6 ± 0.5 | 2.5 ± 0.5 | 2.6 ± 0.4 | 2.6 ± 0.5 |
| Set-Cover/TSP (LKH)-S | 0.7 ± 0.2 | 0.6 ± 0.2 | 0.7 ± 0.2 | 0.6 ± 0.2 |
| Complete Plan-S | 3.2 ± 0.6 | 2.9 ± 0.3 | 2.9 ± 0.3 | 3.1 ± 0.7 |
| Scenario | Lab 2 | | | |
| Multi-Greedy-NBV | 0.2 ± 0.0 | 0.3 ± 0.0 | 0.4 ± 0.0 | 0.6 ± 0.0 |
| Greedy-NBV-S | 0.2 ± 0.0 | 0.2 ± 0.0 | 0.2 ± 0.0 | 0.3 ± 0.0 |
| Multi-Simple-Greedy | 0.1 ± 0.0 | 0.3 ± 0.0 | 0.4 ± 0.0 | 0.6 ± 0.1 |
| Simple-Greedy-S | 0.2 ± 0.0 | 0.2 ± 0.0 | 0.2 ± 0.0 | 0.2 ± 0.0 |
| Set-Cover/TSP (TFD)-S | 2.8 ± 0.8 | 2.8 ± 0.8 | 2.9 ± 0.5 | 2.6 ± 0.6 |
| Set-Cover/TSP (LKH)-S | 0.6 ± 0.0 | 0.6 ± 0.0 | 0.6 ± 0.0 | 0.6 ± 0.0 |
| Complete Plan-S | 3.8 ± 0.9 | 4.3 ± 1.2 | 4.3 ± 0.8 | 4.3 ± 0.9 |
| Scenario | Bldg. 78 | | | |
| Multi-Greedy-NBV | 19.0 ± 1.2 | 40.7 ± 1.3 | 59.7 ± 1.3 | 79.8 ± 4.7 |
| Greedy-NBV-S | 21.9 ± 1.2 | 22.0 ± 1.0 | 22.5 ± 1.0 | 34.4 ± 1.2 |
| Multi-Simple-Greedy | 17.9 ± 1.1 | 40.5 ± 1.5 | 58.9 ± 3.0 | 79.6 ± 4.3 |
| Simple-Greedy-S | 20.1 ± 1.3 | 21.2 ± 1.0 | 20.9 ± 0.6 | 33.3 ± 1.6 |
| Set-Cover/TSP (TFD)-S | 158.4 ± 3.5 | 159.3 ± 4.6 | 158.8 ± 4.4 | 174.1 ± 4.6 |
| Set-Cover/TSP (LKH)-S | 43.3 ± 2.0 | 43.9 ± 2.8 | 43.8 ± 2.2 | 56.6 ± 2.5 |
| Complete Plan-S | 1363.0 ± 255.8 | 1337.6 ± 245.9 | 1363.1 ± 240.0 | 1356.0 ± 202.2 |
| Scenario | Arena | | | |
| Multi-Greedy-NBV | 64.4 ± 3.7 | 133.0 ± 9.0 | 201.0 ± 10.6 | 272.5 ± 12.8 |
| Greedy-NBV-S | 69.9 ± 4.3 | 71.6 ± 4.0 | 73.9 ± 3.3 | 103.9 ± 3.9 |
| Multi-Simple-Greedy | 61.5 ± 3.9 | 132.9 ± 9.3 | 210.3 ± 5.9 | 279.6 ± 15.3 |
| Simple-Greedy-S | 65.7 ± 3.7 | 68.3 ± 3.9 | 70.9 ± 3.4 | 101.8 ± 3.1 |
| Set-Cover/TSP (TFD)-S | 487.4 ± 33.0 | 488.7 ± 24.4 | 504.0 ± 27.1 | 523.7 ± 25.6 |
| Set-Cover/TSP (LKH)-S | 131.7 ± 11.7 | 131.9 ± 7.9 | 134.7 ± 8.9 | 165.1 ± 12.2 |
| Complete Plan-S | 2421.9 ± 483.3 | 2387.2 ± 326.4 | 2445.4 ± 305.4 | 2501.4 ± 372.7 |
| Scenario | Campus | | | |
| Multi-Greedy-NBV | 690.0 ± 54.6 | 1471.7 ± 83.3 | 2164.0 ± 94.4 | 2847.0 ± 140.3 |
| Greedy-NBV-S | 781.8 ± 60.0 | 803.5 ± 51.8 | 800.6 ± 33.9 | 976.0 ± 52.8 |
| Multi-Simple-Greedy | 717.9 ± 50.4 | 1488.8 ± 106.0 | 2229.5 ± 129.4 | 2947.6 ± 178.6 |
| Simple-Greedy-S | 815.4 ± 44.8 | 844.6 ± 49.8 | 845.6 ± 30.6 | 1071.4 ± 50.4 |
| Set-Cover/TSP (LKH)-S | 1249.2 ± 97.8 | 1285.0 ± 105.2 | 1275.2 ± 98.0 | 1432.7 ± 110.9 |

Table 6.3: This table shows computation times in seconds for the different scenarios. Set-Cover/TSP (TFD)-S and Complete Plan-S were not applicable in the Campus scenario due to limited memory.



Campus

Figure 6.7: A box plot of the plan cost for $N = 1, \ldots, 4$ of multiple variants of our algorithm for the Campus scenario. The variants Set-Cover/TSP (TFD)-S and Complete Plan-S were not applicable due to limited memory.

matter more than the complexity classes themselves, especially when considering a limited range of input sizes (here $N = 1, \ldots, 4$). The same effect also explains why the Set-Cover/TSP algorithms in some cases are faster to compute than the multi-robot greedy variants as the number of robots increases. Again, the computation of the single robot solution dominates the computation time and the splitting into segments, despite the exponential complexity is comparatively fast and its overhead does not become relevant until four robots are used. The increase in computation time for four robots is particularly noticeable on the larger maps. This indicates that computing optimal scheduling by enumeration is unlikely to scale for larger number of robots, where more sophisticated scheduling algorithms must be applied.

For the planning based algorithms, we generally see better results at the cost of increased computation time. The results for the smaller scenarios are similar. However, the Simple-Greedy-S algorithm is quite competitive overall. Although costs computed for the rescue arena do not significantly differ, here the computation times are shorter for Simple-Greedy-S. With increasing problem size the decomposed variant becomes superior on the Campus map, when using a specialized TSP solver. As the planner operates on a grounded representation the very large problems become infeasible for the planning based variants. In these cases the system runs out of memory not in the search, but already during the grounding phase of the planner. Although plan cost for Complete Plan-S is similar to the other variants, the computation times are not competitive. A brief discussion of optimal planning is found in Section 6.6.3.

When we compare the cost with an increasing size of robots, we see that all algorithms were able to utilize more robots efficiently. Overall planned costs scale down almost linearly with an increasing number of robots, which is an important aspect especially for the larger scenarios. The question if this scaling behavior carries on to real-world scenarios, especially in smaller settings where robots can obstruct each other will be addressed in Section 6.6.4.

| Algorithm | | first solution | best solution | optimality |
|---|---|---|---|---|
| Simple-Greedy | cost [s] | 22.22 | 22.22 | |
| | computation time [s] | 0.03 | 0.03 | |
| Set Cover/TSP (TFD) | cost [s] | 19.34 | 16.83 | |
| | computation time [s] | 0.15 | 7.76 | 100.09 |
| Complete Plan | cost [s] | 20.79 | 15.78 | |
| | computation time [s] | 0.16 | 12722.88 | 33281.29 |

Table 6.4: Comparison of best results that the algorithms are capable of. Computation time and planned cost until the first and best plan are found are listed together with the time of the full run needed by the planners to prove the best plan to be optimal.

## 6.6.3 Optimal Solutions and Anytime Planning

The Set Cover/TSP approaches are closely related to the formulation for the Complete Plan approach, but decompose the problem. As briefly discussed in Section 6.5.2 this decomposition can lead to suboptimal solutions. Clearly, it is of interest to experimentally determine the loss of quality of the solutions that is due to the decomposition into a set cover and TSP problem. One problem for such an experiment is that the Complete Plan variant only runs within a reasonable amount of time on small maps and computing optimal solutions for the larger maps was not feasible. Thus we used the $\tilde{X}$ and minimal partition obtained for the smallest map (Lab 1) and ran *Complete Plan* against *Set Cover/TSP (TFD)* for one robot until the state space was completely explored. We used TFD/M in both instances which allowed us to find optimal solutions and prove optimality.

Table 6.4 shows the cost of the first plan found by the respective variant, also including Simple-Greedy as a reference, the best plan as well as the time it took to determine the plan and prove optimality. All variants quickly found a reasonable first solution with the Set Cover/TSP (TFD) variant finding the best first solution, although taking an order of magnitude longer than the greedy variant. As one would expect the final optimal solution returned by the Complete Plan variant is better (15.78 s) than the best solution for the decomposition approach Set Cover/TSP (TFD) (16.83 s) as that is only optimal for the set cover and TSP problem independently. The Set Cover/TSP (TFD) solution is 6.6% longer than optimal, yet the reduction in computation time from 12722.88 s for the optimal solution to 7.76 s is substantial. In addition, the Complete Plan approach only found a slightly better plan than 16.83 s after 4996 s. Hence, with an anytime approach the Complete Plan formulation is not likely to yield any improvements over the decomposition approach. A complete theoretical or experimental analysis of related questions, e.g., the derivation of approximation factors is beyond the scope of this work. This brief experimental investigation, however, indicates that our choice to decompose the problem has a reasonable tradeoff between quality and computation time. This observation encourages the use of our algorithms for real-world scenarios with the expectation of finding solutions with good quality. The next section demonstrates the application of our approach with a team of real robots.

| Computing $\tilde{X}$ [s] | Minimal Partition [s] | Reduction Factor | $\epsilon$ [dm$^3$] | $N_{sensor}$ |
|---|---|---|---|---|
| 4.8 | 0.02 | 1.92 | 1500 | 45 |

Table 6.5: This table shows computation times for the generation of $\tilde{X}$ and its minimal partition for the real-world scenario shown in Figure 6.8.

### 6.6.4 Real-World Experiments

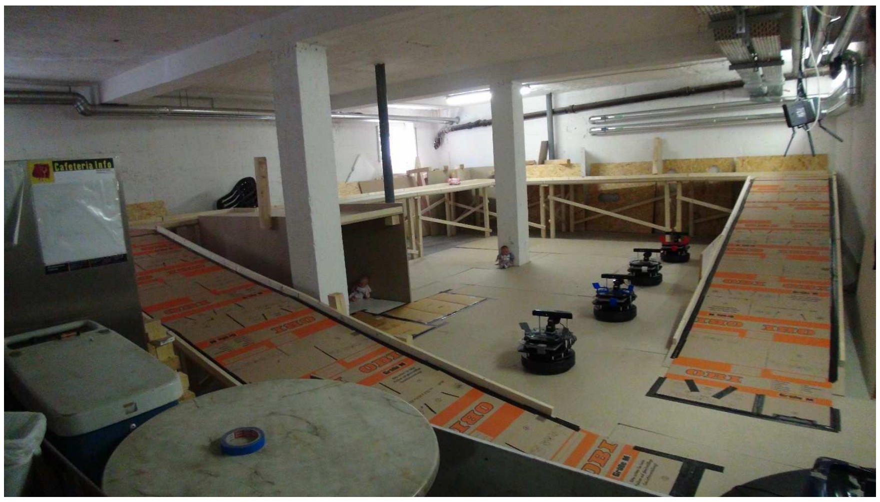


Figure 6.8: This figure shows our two story test environment for multi-robot coverage search experiments.

We performed real-robot experiments with up to four robots in a two story test environment[4] shown in Figure 6.8. The goals for these experiments are manifold. First, we show that the presented algorithms can be applied in practice. From the actual execution we learn in how far the simulation results can predict the real-world performance relative between algorithms, i.e., which will lead to shorter execution times, and absolute, i.e., do the observed execution times lie within reasonable margins

[4]A video of the experiments is available at: `http://www.youtube.com/watch?v=jEFZMoxNGMI`

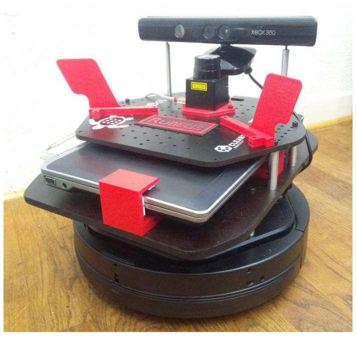


Figure 6.9: One of the Turtlebot 2 robots used in the experiments with a Kinect sensor and Hokuyo laser.

of the planned cost. In addition, we also determine how well the scaling properties of the multi-robot solutions transfer to the real world and where the limitations of an offline approach lie. As noted previously, our approach does not directly consider multi-robot collisions, view obstructions, and other issues arising when using multiple robots. These are dealt with on the implementation level for our specific system and may also have an impact on the real execution of the coverage search solutions.

The experiments are performed in a two story test environment that allows to observe the lower level from the upper level and has a cave-like section to create a three dimensional problem. Figure 6.8 shows the test environment. The test environment was built according to a manually created three dimensional blueprint. In contrast to the experiments presented in the previous sections, we did not build an OctoMap from sensor data, but used the 3d blueprint to generate an OctoMap with a resolution of 0.05 $m$. This OctoMap was used as an input for our algorithms. We ran the sampling for $\tilde{X}$ to search for volumes of 0.064 $m^3$—a volume too small for a human not to be found in. Computation times for this are shown in Table 6.5. To determine a realistic *cost*-function we ran preliminary experiments and matched the *cost*-function to the observed execution times. Then we ran the greedy and decomposition variants of the multi-robot coverage search algorithms from Section 6.5 for one to four robots and executed the resulting plans on the robots. As a robotic platform we use four modified Turtlebot 2, shown in Figure 6.9, that are equipped with a laser range finder for localization and a Kinect RGBD camera that is used as the observation sensor. Note that the robot is different from the robot used for the experiments presented in the previous section, most importantly it does not have its camera mounted on a manipulator. Each robot gets its specific path preloaded for an experiment and all robots visit their sequence of sensor locations in parallel and record a view at each of these in form of a 3d point cloud. For safety reasons ramp transitions between levels have been tele-operated. All other actions, especially navigation to sensor locations were autonomous. Thus in total there were 24 multi-robot coverage search runs containing 60 individual robot runs. Figure 6.11 gives the measured overall execu-

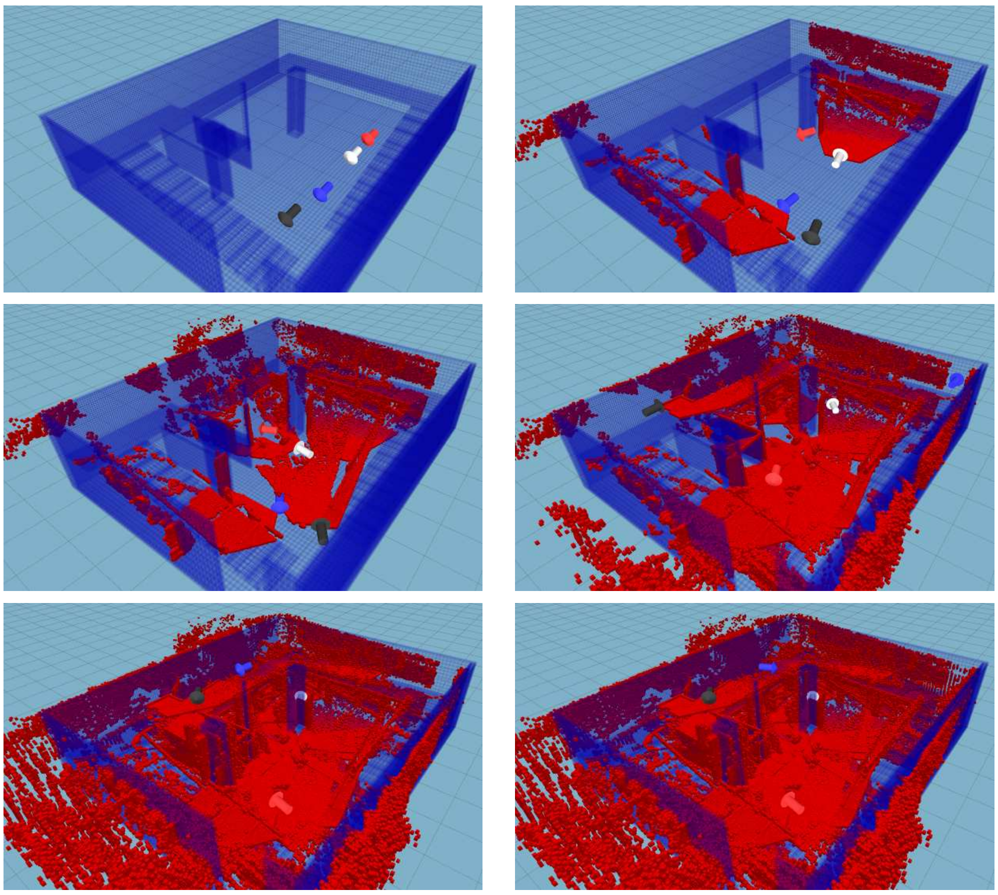
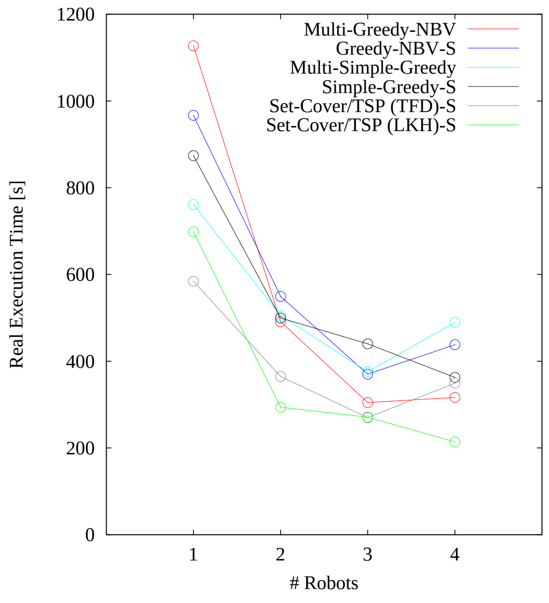
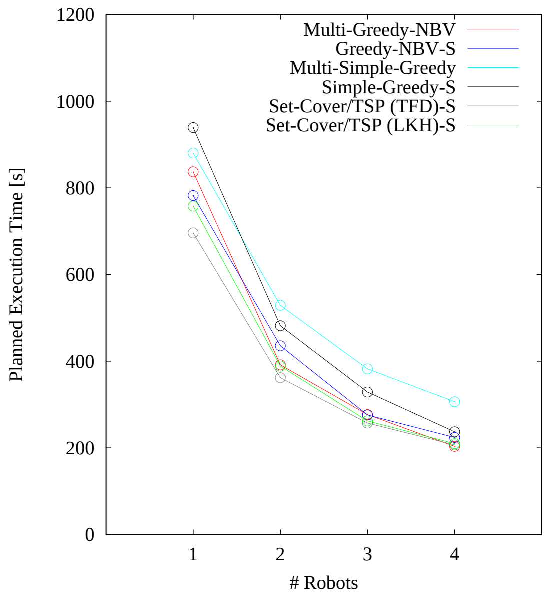


Figure 6.10: Illustration of multi-robot coverage search in the environment shown in Figure 6.8. The four robot solution for Set-Cover/TSP (LKH) is displayed. Endpoints of view rays are shown in red.

tion time for each algorithm and number of robots in comparison to the overall costs computed by the algorithms. Table 6.6 gives the computation times for running the multi-robot coverage search algorithms and shows the total path length of all robots for each algorithm and number of robots.



Real Execution Time [s]



Planned Execution Time [s]

Figure 6.11: This figure shows the execution times of the real robots (left) in comparison to the planned cost by the coverage search algorithms (right).

| | Computation Time [s] | | | | Path Length [m] | | | |
|---|---|---|---|---|---|---|---|---|
| Num Robots | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| Multi-Greedy-NBV | 6.7 | 14.4 | 20.9 | 27.8 | 87.0 | 59.5 | 50.6 | 67.6 |
| Greedy-NBV-S | 6.7 | 7.2 | 7.5 | 7.5 | 70.1 | 80.0 | 70.6 | 76.0 |
| Multi-Simple-Greedy | 12.4 | 28.0 | 40.3 | 53.0 | 39.4 | 44.5 | 58.7 | 93.2 |
| Simple-Greedy-S | 13.0 | 13.8 | 14.1 | 14.4 | 44.6 | 48.6 | 52.7 | 53.4 |
| Set-Cover/TSP (TFD)-S | 49.7 | 47.0 | 47.6 | 49.2 | 30.5 | 33.3 | 34.7 | 52.1 |
| Set-Cover/TSP (LKH)-S | 37.5 | 31.5 | 29.2 | 29.5 | 48.7 | 32.7 | 48.1 | 40.0 |

Table 6.6: This table shows the computation times for the different algorithms (left) and the total path length driven by all robots in a run.

## 6.7 Discussion

We now discuss in detail the results from the previous experiments, especially the applicability of our algorithms to real-world systems and with respect to the observations from simulation experiments. First, we report that the observed coverage in the real sensor data was never lower than 98.4% from what the algorithms predicted for any run. This indicates that the presented algorithms provide a viable solution to the coverage search problem for realistic environments. As an example for an execution see Figure 6.10. Comparing the algorithms we see that the Greedy-NBV variants perform well in comparison to the other variants, especially against Simple-Greedy. This is due to the fact that Greedy-NBV considering higher utility views usually uses fewer views in total, which comes into play in this particular environment. In relation to the distance to be driven between views, the time to approach and record a view matters for this smaller environment in comparison to the larger simulation maps where the driving distance dominates the execution time given by the *cost* function. We can also see that in the driven path length. Greedy-NBV usually drives longer distances, but is still faster than Simple-Greedy with real robots. Nevertheless, the Set-Cover/TSP variants still have better performance than all greedy algorithms in almost all cases in their planned costs and, more importantly, also in the observed execution times. This means that the longer computation times for these algorithms do pay off in faster execution times. A fact that is relevant for real-world applications as offline computation time usually is cheaper than robot operation time. Only for online approaches or when running the algorithms on the robot itself upon deployment one might prefer a lower computation time, such as with *Greedy-NBV-S*.

If we compare the planned cost with the real execution time, we observe that computed times are not a very accurate prediction of real execution times, although the overall ranking between algorithms is still fairly similar whether one considers computed or real execution times. This is not really surprising as autonomous robot navigation is influenced by many factors, including but not limited to sensor noise and inaccurate motion execution. These affect our autonomous navigation and are contained in measured travel times and thus execution times. More importantly, the real execution time is determined by the slowest robot and when running multiple robots it is more likely that at least one will have some delays in its path, increasing the likelihood of a delay with each new robot. Here our assumption that robots operate independently also comes into play. A robot blocking the path of another robot temporarily might have bad effects on the blocked robot's travel time. This is what happened in the four robot case for Set-Cover/TSP (TFD)-S. See Figure 6.12(a) for an illustration. Another example can be seen in Figure 6.12(b). Final poses of two robot paths lie next to each other. Although it is the best solution, in terms of cost, i.e., computed execution time, it is very likely that one robot will arrive at that location earlier than the other. In such a case it would be beneficial to transfer the assignment of sensor locations to the robot that is already there. Conflicting situa-

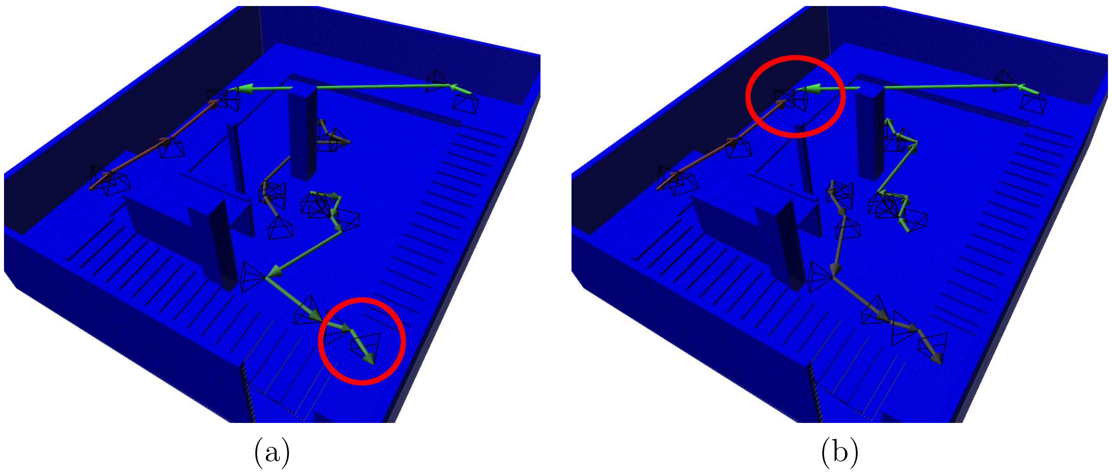


Figure 6.12: Illustration of sub-optimal real-world behavior. Four robot solutions from *Set-Cover/TSP (TFD)-S* (a) and *Set-Cover/TSP (LKH)-S* (b) are shown. The path to both ramps is in a confined space marked on the left image. If a robot assigned to the lower level temporarily blocks access to the ramps, robots aiming for the upper level must wait. The final poses of two robots' paths might lie next to each other (marked on the right image) predicting both robots arriving at the same time. It is likely that one robot arrives earlier and thus should target both views.

tions between robots are not always that extreme, but they do explain most of the cases in which solutions for three or four robots have computed execution times that underestimate the real execution times. These situations are most effectively solved online during execution, but it is still beneficial to have plans that distribute robots preventing possible conflicts in the first place. An example is the four robot execution for the Set-Cover/TSP (LKH)-S solution. Nevertheless, we are approaching the limits of gaining performance by scaling up the number of robots for this environment. While keeping this in mind, using multiple robots to solve the coverage search problem clearly reduces the time to solve the coverage search and the algorithms utilize additional robots fairly effectively.

## 6.8 Conclusion

We considered the problem of identifying and planning efficient sequences of sensor locations for covering complex environments represented in 3d with multiple robots. For that purpose we introduced a sampling based method that reduces the size of the search space by selecting a large number of high utility sensor locations, which can then be used to efficiently plan sequences of sensor locations. We introduced several variants for the planning problem for single robots and multi-robot teams. We evaluated these empirically in order to determine the trade-off between computation time and execution time of the solutions. Our results in simulation and real-world experiments indicate that despite the intractability of the problem, efficient multi-robot coverage in 3d is feasible.

Small size problems, such as incremental vicinity exploration by a single robot in the Lab scenario, were solved close to real-time and thus can directly be deployed in real-world applications. For larger problems, our *Simple-Greedy* variant provided solutions that were competitive with the ones based on more elaborate planning approaches that solve the set cover and traveling salesman subproblems. If the time for visiting a single sensor location is noticeable, the *Greedy-NBV* algorithm is a viable alternative. Although the computation times for the *Simple-Greedy* variants were smaller than for the advanced solutions, the latter are still the overall better choice since they result in lower execution times. This is especially the case when the map is known prior to deployment and not transmitted to robots after deployment, giving more time for offline computations.

As one would expect, performance increases when adding more robots. All algorithms were able to reduce the execution time for the coverage search problem when given more robots. Yet, the dedicated multi-robot greedy solutions only showed advantages for small to medium sized problems. The decomposition approach that solves the set cover and traveling salesman subproblems turned out to be superior, i.e., producing high-quality solutions in a short amount of time, especially in combination with the TSP solver. Even the splitting of the resulting single robot solution into a solution for multiple robots did not lead to inferior performance compared to the multi-robot greedy solutions, which do not require the splitting. Overall, producing multi-robot plans from single-robot plans showed to be a good approach for extending single-robot algorithms.

Our real-world experiments have shown that the simulation results transfer well to robots acting in real environments. The same algorithms that have been shown to be superior in simulation also performed better in reality, especially if we value execution time more than offline computation time. An increasing number of robots also lead to shorter execution times when the robots were distributed well. Although one would expect that advanced dedicated multi-robot algorithms might produce even better solutions with lower execution times, our experiences from the real world experiments suggest that the strongest potential for improvement is to consider online adaptation of solutions. This is mainly due to the fact that robot execution times are hard to predict precisely in reality and the location of an error or unanticipated delay is crucial for finding a high quality solution that mitigates the delay. Overall, despite the large amount of future work one may carry out to improve multi-robot 3d coverage search, we were able to show that our current approach is already well suited for real-world applications.

# Chapter 7

# Conclusion

In this thesis we investigated task planning as a method to control the high-level decision making of mobile robots. The aim was to enable robots to solve complex tasks by combining their skills in an intelligent and goal-directed manner. We chose to use automated planning to tackle this problem as planners only need a description of a robot's skills and the goal to reach. Based on that they produce plans that solve arbitrary situations, which is important for real-world tasks, where one cannot rely on known or deterministic scenarios. Nevertheless, classical planning does have some limitations, mainly that tasks are described symbolically or with limited numerics, and that at least for classical planners an observable and deterministic world model is assumed. Therefore a central topic was to develop techniques to make automated planners suitable for planning in real-world scenarios and integrate these into a real robot system.

First, we addressed the discrepancy between a symbolic description and geometric robotics skills, like grasping an object. We determined that state of the art numerical extensions are not expressive enough for this kind of task and thus developed the concept of *semantic attachments*. An important feature for these is that they use a generic interface to include arbitrary external reasoners, so that the resulting planner is applicable to any application domain. As semantic attachments are transparent to the planner, we could base our planner on a state of the art classical planner, in this case Temporal Fast Downward. We introduced three kinds of semantic attachments that allow us to provide external semantics for different aspects of planning tasks: condition checkers, effect applicators, and cost modules. These were motivated by our requirements for modeling robotics tasks that we encountered in practice. The result of this work is the planner Temporal Fast Downward with Modules (TFD/M) that generates sound plans for mixed symbolic-geometric robotic planning tasks. We demonstrated this with manipulation planning tasks that use a robot and world model representing real robots. The resulting plans contain, for example, motion plans that are directly executable on a robot.

Another problem related to the expressiveness of planning formalisms was action parametrization. Commonly these only allow a finite number of operator instances, which prohibit to model geometric choices like where to place an object. State of the art solutions mitigated this by providing a limited number of options to the planner with the input. Our solution removed this limitation. We provided a mechanism to generate action parameters on the fly during planning. In the spirit of semantic attachments an external interface allows to use arbitrary algorithms to do so. One reason for this limitation of classical planning algorithms is that the generic problem is undecidable. Therefore we also developed a new search algorithm that finds solutions for planning tasks with an infinite branching factor. Thus domain designers do not need to commit to giving any constant in the input that in reality is problem dependent. This is now solved in the planner as part of the planning process. We believe this principle is important in general, i.e., a user should ideally need to specify only what is needed to define the problem, while the system determines everything else. Our evaluation shows that in comparison to an adaption of classical planning algorithms our new algorithm also performs better. The main reason for this is that the planner is now free to choose a suitable number of operator instantiations that might be lower than a given fixed limit.

The second part of the thesis dealt with applications of our planner in real-world systems. As a basis for this we identified challenges of real-world tasks such as an incomplete or uncertain world knowledge and unexpected action outcomes. To deal with these issues we embedded our planner in a continual planning loop. Here we focused not only on the implementation of this system, but also specifically addressed what simplifications we made and under what assumptions we still can guarantee to reach the goal. It turns out that for many situations one has to make assumptions, i.e., there is no best way to solve these problems in general. This is important to recognize as thus the actual impact of our simplifications in comparison to other possible approaches like nondeterministic planning is not that strong. Most noticable is that we require the robot to never get into a dead end state. We implemented a complex mobile manipulation system on the PR2 robot and thereby showed that our system works in practice or in other words that our assumptions hold in the scenarios that we investigated.

We also addressed the problem of multi-robot coverage search in 3d. The interesting aspect here was that this is a challenging problem from the geometric side of robotics tasks, but also contained a combinatorial problem that can be formulated as a planning problem. We compared various approaches using greedy and planning based formulations. Our evaluation showed that the general approach that we proposed works well to solve this problem in simulation and real-world settings. Besides that we also gained another insight. Although the decomposition approach using the specific traveling salesman solver was slightly better than the approach that only used our planner, the planning based variant was competitive. This means that if one needs to built an ad-hoc system and a generic domain-independent planner as ours is already integrated in the robot, it might be unnecessary to look for specialized solvers when the main goal is to get a working system. Task planning can already be well suited for any new problem.

Within this thesis we have investigated how task planning can be used as the high-level control method for robotics. We identified the major problems that such a system faces and introduced new planning techniques for integrated task and motion planning to solve these. One major contribution is the resulting planning system Temporal Fast Downward with Modules (TFD/M) that implements these techniques. We also addressed how a planner is integrated into a robotic system that must act in the real world. Here we have not only shown experimentally that these techniques work in practice, but also stated the assumptions under which real-world planning tasks are solved. Our domain formulation is thus more than a single example as it demonstrates general concepts for dealing with unknown objects and observation actions in a replanning context. An imporant aspect here was that our system is generic and not specific to a robot platform or scenario. This was shown by applying our planner to various different robots and scenarios.

The general applicability to many different tasks and scenarios makes task planning an interesting method to pursue in the future. While this thesis mainly focused on establishing planning techniques for robotics there are some topics that we only touched upon. Better computation speed is always relevant. Heuristic guidance had a lot of success for classical planning. Developing specific heuristics for robotics planning tasks is thus promising. The challenge here is to keep these generally applicable without being specific to a certain domain or even problem instance. Another interesting point is to work towards even more generic skill descriptions. Currently one has to give these as a planning task with semantic attachments. Although this is a generic method, this requires users to have at least a basic understanding of planning itself. To solve this problem providing world models and skill descriptions that are easily adaptable to any robot system as well as the necessary tools to allow experts in robotics, but not necessarily planning, to use our system is necessary. This will enable anyone to use the strength and generality of task planning as the high-level control method in their robotic system.

# Appendix

The planning and execution system developed in this thesis is released as open source software and integrated in the Robot Operating System (ROS) (Quigley et al., 2009).

The most recent version of this software is available at
`https://github.com/dornhege/tfd_modules`.
The current code revision at the time of writing this thesis is
`907080467ffb191a14bc948b75efe42e77362008`.
A direct download of this revision is available at
`http://www.informatik.uni-freiburg.de/~dornhege/tfd_modules.tar.gz`.

The following software and documentation is contained therein.

- The directory `tfd_modules` contains the planner TFD/M.
- The directory `continual_planning_executive` contains our continual planning framework.
- The full PDDL/M domain for the TidyUp scenario is in `planner_benchmarks/modular/tidyup/domain_modules.pddl`. Note that the TidyUp domain is specified in temporal PDDL, i.e., using `durative-action` instead of `action` and specifying temporal conditions and effects. As already stated in the thesis, these were not in effect during the experiments.
- PDDL/M domains and problems from Chapter 3 are in
  `planner_benchmarks/modular/crewplanning-modules`
  and
  `planner_benchmarks/modular/transport-modules`

# Bibliography

N. Agmon, N. Hazon, and G. A. Kaminka. The giving tree: constructing trees for efficient offline and online multi-robot coverage. *Annals of Mathematics and Artificial Intelligence*, 52(2-4):143–168, 2008.

R. Alami, J. P. Laumond, and T. Simeon. Two manipulation planning algorithms. In *Workshop on algorithmic foundations of robotics (WAFR)*, 1995.

V. Alczar, M. Veloso, and D. Borrajo. Adapting a rapidly-exploring random tree for automated planning. In *International Symposium on Combinatorial Search (SoCS)*, 2011.

J. A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1988.

D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The traveling salesman problem: a computational study.* Princeton University Press, 2007.

F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

C. Bäckström and B. Nebel. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

J. Banta, Y. Zhieng, X. Wang, G. Zhang, M. Smith, and M. Abidi. A "best-next-view“ algorithm for three-dimensional scene reconstruction using range images. In *SPIE Symposium on Intelligent Robots and Computer Vision: Algorithms*, 1995.

J. Barry, L. P. Kaelbling, and T. Lozano-Pérez. A hierarchical approach to diverse action manipulation. In *International Conference on Robotics and Automation (ICRA)*, 2013.

T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, 2006.

P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

P. Bertoli, A. Cimatti, U. Dal Lago, and M. Pistore. Extending pddl to nondeterminism, limited sensing and iterative conditional plans. In *ICAPS Workshop on PDDL*, 2003.

P. Bertoli, A. Cimatti, and P. Traverso. Interleaving execution and planning in nondeterministic partially observable domains. In *European Conference on Artificial Intelligence (ECAI)*, 2004.

J. Bohren, R. Rusu, E. Gil Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, , and S. Holzer. Towards autonomous robotic butlers: Lessons learned with the PR2. In *International Conference on Robotics and Automation (ICRA)*, 2011.

A. Botea, M. Müller, and J. Schaeffer. Using abstraction for planning in sokoban. In *Computers and Games*, 2003.

F. Bourgault, A. Makarenko, S. Williams, B. Grocholsky, and H. Durrant-Whyte. Information based adaptive robotic exploration. In *International Conference on Intelligent Robots and Systems (IROS)*, 2002.

M. Brenner and B. Nebel. Continual planning and acting in dynamic multiagent environments. *Journal of Autonomous Agents and Multi-Agent Systems*, 19(3):297–331, 2009.

D. Bryce. POND: The partially-observable and non-deterministic planner. In *International Planning Competition (IPC)*, 2006.

D. Bryce, S. Kambhampati, and D. E. Smith. Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*, 26:35–99, 2006.

C. Burbridge and R. Dearden. An approach for efficient planning of robotic manipulation tasks. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1998.

S. Cambon, R. Alami, and F. Gravot. A robot task planer that merges symbolic and geometric reasoning. In *European Conference on Artificial Intelligence (ECAI)*, 2004.

S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *International Journal of Robotics Research*, 28(1): 104–126, 2009.

M. Cashmore, M. Fox, and E. Giunchiglia. Partially grounded planning as quantified boolean formula. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

H. Choset. Coverage for robotics–a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31(1):113–126, 2001.

T. Chung, G. Hollinger, and V. Isler. Search and pursuit-evasion in mobile robotics. *Autonomous Robots*, 31(4):299–316, 2011.

J. Cortes, S. Martinez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. In *International Conference on Robotics and Automation (ICRA)*, 2002.

M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. OMiGA : An open minded grounding on-the-fly answer set solver. In *European Conference on Logics in Artificial Intelligence*, 2012.

L. de Silva, A. Pandey, and R. Alami. An interface for interleaved symbolic-geometric planning and backtracking. In *International Conference on Intelligent Robots and Systems (IROS)*, 2013.

C. Dornhege and A. Hertle. Integrated symbolic planning in the tidyup-robot project. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI II*, 2013.

C. Dornhege and A. Kleiner. A frontier-void-based approach for autonomous exploration in 3d. In *International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2011.

C. Dornhege and A. Kleiner. A frontier-void-based approach for autonomous exploration in 3d. *Advanced Robotics*, 27(6), 2013.

C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel. Semantic attachments for domain-independent planning systems. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009a.

C. Dornhege, M. Gissler, M. Teschner, and B. Nebel. Integrating symbolic and geometric planning for mobile manipulation. In *International Workshop on Safety, Security and Rescue Robotics (SSRR)*, 2009b.

C. Dornhege, A. Hertle, and B. Nebel. Lazy evaluation and subsumption caching for search-based integrated task and motion planning. In *IROS Workshop on AI-based robotics*, 2013a.

C. Dornhege, A. Kleiner, and A. Kolling. Coverage search in 3d. In *International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2013b. (Best Paper Award Finalist).

C. Dornhege, A. Kleiner, A. Hertle, and A. Kolling. Multi-robot coverage search in 3d. *Journal of Field Robotics*, 2015. To appear.

F. Endres, J. Trinkle, and W. Burgard. Learning the dynamics of doors for robotic manipulation. In *International Conference on Intelligent Robots and Systems (IROS)*, 2013.

B. Englot and F. S. Hover. Three-dimensional coverage planning for an underwater inspection robot. *International Journal of Robotics Research*, 32(9-10):1048–1073, 2013.

K. Erol, D. S. Nau, and V. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

P. Eyerich, M. Brenner, and B. Nebel. On the complexity of planning operator subsumption. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2008.

P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

M. Fox and D. Long. Identifying and managing combinatorial optimisation subproblems in planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

M. Fox and D. Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.

A. Gaschler, R. P. A. Petrick, T. Kröger, A. Knoll, and O. Khatib. Robot task planning with contingencies for run-time sensing. In *ICRA Workshop on Combining Task and Motion Planning*, 2013.

M. Gissler, C. Dornhege, B. Nebel, and M. Teschner. Deformable proximity queries and their application in mobile manipulation planning. In *Symposium on Visual Computing (ISVC)*, 2009.

D. Golovin and A. Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *Journal of Artificial Intelligence Research*, 42: 427–486, 2011.

H. Gonzalez-Banos, E. Mao, J. Latombe, T. Murali, and A. Efrat. Planning robot motion strategies for efficient model construction. In *International Symposium on Robotics Research (ISRR)*, 2000.

P. Gregory, D. Long, M. Fox, and J. C. Beck. Planning modulo theories: Extending the planning paradigm. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1–2):35–62, 2001.

M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:505–535, 2009.

M. Helmert and H. Geffner. Unifying the causal graph and additive heuristics. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

A. Hertle, C. Dornhege, T. Keller, and B. Nebel. Planning with semantic attachments: An object-oriented view. In *European Conference on Artificial Intelligence (ECAI)*, 2012.

A. Hertle, C. Dornhege, T. Keller, R. Mattmüller, M. Ortlieb, and B. Nebel. An experimental comparison of classical, FOND and probabilistic planning. In *German Conference on Artificial Intelligence (KI)*, 2014.

J. Hess, D. Tipaldi, and W. Burgard. Null space optimization for effective coverage of 3d surfaces using redundant manipulators. In *International Conference on Intelligent Robots and Systems (IROS)*, 2012.

J. Hoffmann and R. Brafman. Contingent planning via heuristic forward search with implicit belief states. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

A. Hornung, S. Böttcher, J. Schlagenhauf, C. Dornhege, A. Hertle, and M. Bennewitz. Mobile manipulation in cluttered environments with humanoids: Integrated perception, task planning, and action execution. In *International Conference on Humanoid Robots (HUMANOIDS)*, 2014.

A. Howard, M. J. Matarić, and G. S. Sukhatme. Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem. In *International Symposium on Distributed Autonomous Robotic Systems*, 2002.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

A. Jacoff and E. Messina. Urban search and rescue robot performance standards: progress update. In *SPIE Defense and Security Conference*, 2007.

A. Jacoff, B. Weiss, and E. Messina. Evolution of a performance metric for urban search and rescue robots. In *Performance Metrics for Intelligent Systems (PERMIS)*, 2003.

D. Joho, C. Stachniss, P. Pfaff, and W. Burgard. Autonomous exploration for 3d map learning. *Autonome Mobile Systeme*, pages 22–28, 2007.

L. Kaelbling and T. Lozano-Pérez. Hierarchical task and motion planning in the now. In *International Conference on Robotics and Automation (ICRA)*, 2011.

L. Kaelbling and T. Lozano-Pérez. Integrated task and motion planning in belief space. *International Journal of Robotics Research*, 32(9-10):1194–1227, 2013.

R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 1972.

L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, NJ, 1950.

A. Kleiner, A. Kolling, M. Lewis, and K. Sycara. Hierarchical visibility for guaranteed search in large-scale outdoor terrain. *Journal of Autonomous Agents and Multi-Agent Systems*, 26(1):1–36, 2013.

C. A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

T. Kollar and N. Roy. Efficient optimization of information-theoretic exploration in slam. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2008.

A. Kolling, A. Kleiner, M. Lewis, and K. Sycara. Pursuit-evasion in 2.5d based on team-visibility. In *International Conference on Intelligent Robots and Systems (IROS)*, 2010.

S. Konecny, S. Stock, F. Pecora, and A. Saffiotti. Planning domain + execution semantics: a way towards robust execution? In *AAAI Spring Symposium on Qualitative Representations for Robots*, 2014.

C. S. Kong, N. A. Peng, and I. Rekleitis. Distributed coverage with multi-robot system. In *International Conference on Robotics and Automation (ICRA)*, 2006.

K. Konolige and N. J. Nilsson. Multiple-agent planning systems. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1980.

I. Kresse and M. Beetz. Movement-aware action control – integrating symbolic and control-theoretic action execution. In *International Conference on Robotics and Automation (ICRA)*, 2012.

U. Kuter, D. S. Nau, E. Reisner, and R. P. Goldman. Using classical planners to solve nondeterministic planning problems. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):119–169, 2000.

T. Lang and M. Toussaint. Relevance grounding for planning in relational domains. *Machine Learning and Knowledge Discovery in Databases*, 5781:736–751, 2009.

J. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

S. Lazebnik. Visibility-based pursuit-evasion in three-dimensional environments. Technical report, University of Illinois at Urbana-Champaign, 2001.

D. Leidner and C. Borst. Hybrid reasoning for mobile manipulation based on object knowledge. In *IROS Workshop on AI-based robotics*, 2013.

D. Leidner, C. Borst, and G. Hirzinger. Things are made for what they are: Solving manipulation tasks by using functional object classes. In *International Conference on Humanoid Robots (HUMANOIDS)*, 2012.

D. Leidner, A. Dietrich, F. Schmidt, C. Borst, and A. Albu-Schäffer. Object-centered hybrid reasoning for whole-body mobile manipulation. In *International Conference on Robotics and Automation (ICRA)*, 2014.

M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS)*. 2004.

M. L. Littman. Probabilistic propositional planning: Representations and complexity. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1997.

S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.

R. Mattmüller, M. Ortlieb, M. Helmert, and P. Bercher. Pattern database heuristics for fully observable nondeterministic planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2010.

D. McDermott. Robot planning. *AI Magazine*, 13:55–79, 1992.

D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wau, and F. Yaman. Shop2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20: 379–404, 2003.

B. Nebel, C. Dornhege, and A. Hertle. How much does a household robot need to know in order to tidy up your home? In *AAAI Workshop on Intelligent Robotic Systems*, 2013.

A. Nüchter, H. Surmann, and J. Hertzberg. Planning robot motion for 3d digitalization of indoor environments. In *International Conference on Advanced Robotics (ICAR)*, pages 222–227, 2003.

J. Orkin. Three states and a plan: The A.I. of F.E.A.R. In *Game Developers Conference (GDC)*, 2006.

R. P. A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

I. Rekleitis, V. Lee-Shue, A. P. New, and H. Choset. Limited communication, multi-robot team based coverage. In *International Conference on Robotics and Automation (ICRA)*, 2004.

A. Renzaglia, L. Doitsidis, A. Martinelli, and E. B. Kosmatopoulos. Multi-robot 3d coverage of unknown terrains. In *Conference on Decision and Control and European Control Conference (CDC-ECC)*, 2011.

S. Richter and M. Helmert. Preferred operators and deferred evaluation in satisficing planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

J. Rintanen. Complexity of planning with partial observability. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2004.

S. Sachs, S. Rajko, and S. M. LaValle. Visibility-based pursuit-evasion in an unknown planar environment. *International Journal of Robotics Research*, 23(1):3–26, 2004.

T. Shermer. Recent results in art galleries. *Proceedings of the IEEE*, 80(9):1384–1399, 1992.

T. Simeon, J. Cortes, J. Laumond, and A. Sahbani. Manipulation planning with probabilistic roadmaps. *International Journal of Robotics Research*, 23(7-8):729–746, 2004.

B. Srivastava and S. Kambhampati. Scaling up planning by teasing out resource scheduling. In *European Conference on Planning*, 1999.

S. Srivastava, L. Riano, S. Russell, and P. Abbeel. Using classical planners for tasks with continuous operators in robotics. In *ICAPS Workshop on Planning and Robotics (PlanRob)*, 2013.

F. Stulp, A. Fedrizzi, L. Mösenlechner, and M. Beetz. Learning and reasoning with action-related places for robust mobile manipulation. *Journal of Artificial Intelligence Research*, 43:1–42, 2012.

M. Westphal, C. Dornhege, S. Wölfl, M. Gissler, and B. Nebel. Guiding the generation of manipulation plans by qualitative spatial reasoning. *Spatial Cognition and Computation: An Interdisciplinary Journal*, 11(1):75–102, 2011.

R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1-2):133–170, 1980.

J. Wolfe, B. Marthi, and S. J. Russell. Combined task and motion planning for mobile manipulation. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2010.

K. M. Wurm, C. Dornhege, P. Eyerich, C. Stachniss, B. Nebel, and W. Burgard. Coordinated exploration with marsupial teams of robots using temporal symbolic planning. In *International Conference on Intelligent Robots and Systems (IROS)*, 2010.

K. M. Wurm, C. Dornhege, C. Stachniss, B. Nebel, and W. Burgard. Coordinating heterogeneous teams of robots using temporal symbolic planning. *Autonomous Robots*, 34(4):277–294, 2013.

B. Yamauchi. A frontier-based approach for autonomous exploration. In *International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, 1997.

S. W. Yoon, A. Fern, and R. Givan. FF-replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2007.

F. Zacharias, C. Borst, and G. Hirzinger. Capturing robot workspace structure: representing robot capabilities. In *International Conference on Intelligent Robots and Systems (IROS)*, 2007.