University of Pennsylvania
ScholarlyCommons

Publicly Accessible Penn Dissertations

1-1-2016

Planning With Adaptive Dimensionality

Kalin Vasilev Gochev
University of Pennsylvania, kgochev@seas.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/edissertations

Part of the Artificial Intelligence and Robotics Commons, and the Robotics Commons

Recommended Citation

Gochev, Kalin Vasilev, "Planning With Adaptive Dimensionality" (2016). *Publicly Accessible Penn Dissertations*. 1739.
http://repository.upenn.edu/edissertations/1739

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/edissertations/1739
For more information, please contact libraryrepository@pobox.upenn.edu.

Planning With Adaptive Dimensionality

Abstract

Modern systems, such as robots or virtual agents, need to be able to plan their actions in increasingly more complex and larger state-spaces, incorporating many degrees of freedom. However, these high-dimensional planning problems often have low-dimensional representations that describe the problem well throughout most of the state-space. For example, planning for manipulation can be represented by planning a trajectory for the end-effector combined with an inverse kinematics solver through obstacle-free areas of the environment, while planning in the full joint space of the arm is only necessary in cluttered areas. Based on this observation, we have developed the framework for Planning with Adaptive Dimensionality, which makes effective use of state abstraction and dimensionality reduction in order to reduce the size and complexity of the state-space. It iteratively constructs and searches a hybrid state-space consisting of both abstract and non-abstract states. Initially the state-space consists only of abstract states, and regions of non-abstract states are selectively introduced into the state-space in order to maintain the feasibility of the resulting path and the strong theoretical guarantees of the algorithm---completeness and bounds on solution cost sub-optimality. The framework is able to make use of hierarchies of abstractions, as different abstractions can be more effective than others in different parts of the state-space. We have extended the framework to be able to utilize anytime and incremental graph search algorithms. Moreover, we have developed a novel general incremental graph search algorithm---tree-restoring weighted A*, which is able to minimize redundant computation between iterations while efficiently handling changes in the search graph. We have applied our framework to several different domains---navigation for unmanned aerial and ground vehicles, multi-robot collaborative navigation, manipulation and mobile manipulation, and navigation for humanoid robots.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Maxim Likhachev

Second Advisor

Alla Safonova

Keywords

Planning Algorithms, Planning for Humanoid Mobility, Planning for Mobile Manipulation, Planning for Navigation, Planning for Robotics, Search-Based Planning

This dissertation is available at ScholarlyCommons: http://repository.upenn.edu/edissertations/1739

Subject Categories
Artificial Intelligence and Robotics | Computer Sciences | Robotics

This dissertation is available at ScholarlyCommons: http://repository.upenn.edu/edissertations/1739

PLANNING WITH ADAPTIVE DIMENSIONALITY

Kalin Vasilev Gochev

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2016

Supervisor of Dissertation

Maxim Likhachev
Adjunct Assistant Professor
Computer and Information Science

Co-Supervisor of Dissertation

Alla Safonova
Assistant Professor
Computer and Information Science

Graduate Group Chairperson

Lyle Ungar
Professor
Computer and Information Science

Dissertation Committee

Christopher Atkeson, Professor of Computer Science

Norman Badler, Professor of Computer and Information Science

Kostas Daniilidis, Professor of Computer and Information Science

Daniel Lee, Professor of Electrical and Systems Engineering

PLANNING WITH ADAPTIVE DIMENSIONALITY

© COPYRIGHT

2016

Kalin Vasilev Gochev

This work is licensed under the

Creative Commons Attribution

NonCommercial-ShareAlike 3.0

License

To view a copy of this license, visit

http://creativecommons.org/licenses/by-nc-sa/3.0/

ABSTRACT

PLANNING WITH ADAPTIVE DIMENSIONALITY

Kalin Vasilev Gochev

Maxim Likhachev

Alla Safonova

Modern systems, such as robots or virtual agents, need to be able to plan their actions in increasingly more complex and larger state-spaces, incorporating many degrees of freedom. However, these high-dimensional planning problems often have low-dimensional representations that describe the problem well throughout most of the state-space. For example, planning for manipulation can be represented by planning a trajectory for the end-effector combined with an inverse kinematics solver through obstacle-free areas of the environment, while planning in the full joint space of the arm is only necessary in cluttered areas. Based on this observation, we have developed the framework for Planning with Adaptive Dimensionality, which makes effective use of state abstraction and dimensionality reduction in order to reduce the size and complexity of the state-space. It iteratively constructs and searches a hybrid state-space consisting of both abstract and non-abstract states. Initially the state-space consists only of abstract states, and regions of non-abstract states are selectively introduced into the state-space in order to maintain the feasibility of the resulting path and the strong theoretical guarantees of the algorithm—completeness and bounds on solution cost sub-optimality. The framework is able to make use of hierarchies of abstractions, as different abstractions can be more effective than others in different parts of the state-space. We have extended the framework to be able to utilize anytime and incremental graph search algorithms. Moreover, we have developed a novel general incremental graph search algorithm—tree-restoring weighted A*, which is able to minimize redundant computation between iterations while efficiently handling changes in the search graph. We have

applied our framework to several different domains—navigation for unmanned aerial and ground vehicles, multi-robot collaborative navigation, manipulation and mobile manipulation, and navigation for humanoid robots.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF TABLES viii

LIST OF ILLUSTRATIONS x

CHAPTER 1 : Introduction 1

CHAPTER 2 : Motivating Observation 5

CHAPTER 3 : Related Work 7

3.1 State Abstraction Techniques 7

3.2 Two-Layer Planners 9

3.3 Sampling-Based Planners 10

3.4 Optimization Methods 13

3.5 Incremental Search Algorithms 13

CHAPTER 4 : Planning with Adaptive Dimensionality 16

4.1 Definitions and Notations 16

4.2 Overview 17

4.3 Hybrid State-Space Construction 20

4.4 Algorithm 23

4.5 Identifying Areas that Require High-Dimensional Planning 26

4.6 Theoretical Properties 27

4.7 Algorithm Parameters 29

CHAPTER 5 : Hierarchical Planning with Adaptive Dimensionality 31

5.1 Motivation 31

5.2 Related Work 31
5.3 Combining Multiple Abstractions 33
5.4 Theoretical Properties 37
5.5 Identifying Useful Abstractions 39

CHAPTER 6 : Incremental Graph Search for *PAD* 46
6.1 Motivation 46
6.2 Definitions and Notations 48
6.3 Tree-Restoring Weighted A* Search 48
6.4 Anytime Tree-Restoring Weighted A* Search 55
6.5 Efficiently Detecting Changes in the Graph 57
6.6 Experimental Evaluation 58
6.7 Analysis of Results 60

CHAPTER 7 : Application: *PAD* for Navigation 65
7.1 Non-Incremental 3D Path Planning for a Non-Holonomic Vehicle 65
7.2 Incremental 3D Path Planning for a Non-Holonomic Vehicle 69
7.3 Interleaving Planning and Execution 74

CHAPTER 8 : Application: *PAD* for Multi-Robot Collaborative Navigation 85
8.1 Related Work 86
8.2 State Lattice with Controller-based Motion Primitives 88
8.3 Implementation Details 91
8.4 Experimental Setup 97
8.5 Analysis of Results 99

CHAPTER 9 : Application: *PAD* for Manipulation 101
9.1 Using 3D Low-Dimensional Representation 101
9.2 Using 4D Low-Dimensional Representation for Manipulators with Independent Wrist Joints 106

CHAPTER 10 : Application: *PAD* for Mobile Manipulation 120
10.1 Using a Single Abstraction 120
10.2 Using Multiple Abstractions 125

CHAPTER 11 : Application: *PAD* for Humanoid Robot Mobility 132
11.1 Domain Background and Related Work 132
11.2 Algorithm Extension 135
11.3 Implementation Details 140
11.4 Experimental Evaluation 150
11.5 Analysis of Results 151

CHAPTER 12 : Conclusion 154

APPENDIX 156

BIBLIOGRAPHY 165

LIST OF TABLES

TABLE 1 : Anytime $TRA^{\star}$ simulation results 60

TABLE 2 : Fixed sub-optimality $TRA^{\star}$ simulation results 61

TABLE 3 : Non-holonomic vehicle navigation planning 68

TABLE 4 : Non-holonomic vehicle navigation planning on PR2 robot 71

TABLE 5 : Incremental vs. non-incremental PAD for navigation 72

TABLE 6 : Interleaving vs. non-interleaving PAD for navigation 81

TABLE 7 : Results for multi-robot collaborative navigation planning 99

TABLE 8 : Results for multi-robot collaborative navigation planning 100

TABLE 9 : 7D/3D manipulation planning results 105

TABLE 10 : Consistency comparison of search-based vs. sampling-based planners 106

TABLE 11 : 7D/4D manipulation planning results 117

TABLE 12 : Path quality comparison results 118

TABLE 13 : Performance of 7D/4D manipulation planning for independent wrist joints 118

TABLE 14 : 11D/3D mobile manipulation results 124

TABLE 15 : Manipulating a stick trough a window: 11D/7D adaptive planner vs. RRT planner results 124

TABLE 16 : Results for mobile manipulation planning with multiple abstractions 131

LIST OF ILLUSTRATIONS

FIGURE 1 : Weighted A^* search example 3

FIGURE 2 : Motivation example: navigation planning 6

FIGURE 3 : Motivation example: motion planning 6

FIGURE 4 : Example of an asymmetric cost function 11

FIGURE 5 : Hybrid graph transitions 18

FIGURE 6 : Planning with Adaptive Dimensionality in 3D/2D 23

FIGURE 7 : Planning with Adaptive Dimensionality in 7D/3D 24

FIGURE 8 : Transitions between abstract sub-spaces 35

FIGURE 9 : Estimating cost gradient for graphs 44

FIGURE 10 : Tree-restoring $A^\star$ example 49

FIGURE 11 : Computing affected graph edges from changed map cells 57

FIGURE 12 : $TRA^\star$ example environment 58

FIGURE 13 : Example maps for non-holonomic vehicle navigation 67

FIGURE 14 : Incremental Planning with Adaptive Dimensionality example . . . 70

FIGURE 15 : 3D/2D Planning with Adaptive Dimensionality for PR2 robot . . 71

FIGURE 16 : Number of iterations vs. speed-up factor 72

FIGURE 17 : PR2 in a cluttered indoor environment. 80

FIGURE 18 : Example graph using state-lattice controllers 90

FIGURE 19 : Multi-robot adaptive navigation planning example 94

FIGURE 20 : Robots used for multi-robot collaborative navigation 96

FIGURE 21 : Maps used in multi-robot collaborative navigation 99

FIGURE 22 : Degrees of freedom of the right arm of a PR2 robot 102

FIGURE 23 : Manipulation planning example environments 103
FIGURE 24 : Trajectory being executed by actual PR2 robot 105
FIGURE 25 : 7D/4D manipulation planning example 109
FIGURE 26 : Heuristic local minimum example 113
FIGURE 27 : PR2 retrieving an object from a fridge 116
FIGURE 28 : Degrees of freedom for mobile manipulation on PR2 robot 121
FIGURE 29 : PR2 manipulating a stick trough a window 122
FIGURE 30 : PR2 reaching from a high shelf to a low shelf of a bookcase 122
FIGURE 31 : Example kitchen environment 128
FIGURE 32 : Example indoor environment from real sensor data 128
FIGURE 33 : Planning phase of Planning with Adaptive Dimensionality with multiple abstractions 129
FIGURE 34 : Tracking phase of Planning with Adaptive Dimensionality with multiple abstractions 130
FIGURE 35 : Initial distribution of sub-spaces in environment 130
FIGURE 36 : Humanoid Robots 133
FIGURE 37 : Humanoid Mobility Example 134
FIGURE 38 : Yamabiko Humanoid Robot 140
FIGURE 39 : Humanoid Robot Abstraction Hierarchy 141
FIGURE 40 : Bipedal Abstraction for Flat Terrain 143
FIGURE 41 : Bipedal Abstraction for Stairs 144
FIGURE 42 : Maintaining Static Stability 147
FIGURE 43 : Planning with Adaptive Dimensionality for Humanoid Mobility . 153

CHAPTER 1 : Introduction

Planning is an important component of any intelligent system. It allows the system to adapt to changing environment conditions and sensory inputs. In recent years robotics research has moved from the controlled predictable industrial environments towards the cluttered, uncontrolled and unpredictable domestic environments, where robots need to be able to safely perform a variety of tasks necessitating careful, but efficient, planning. Search-based planning algorithms are often used in many areas of robotics and artificial intelligence. They represent the planning problem as a graph search in a graph consisting of a set of nodes, denoting valid system configurations, and a set of edges, denoting valid transitions from one system configuration to another. The planning problem then becomes finding a path (a sequence of edges) in the graph from a given start node to a given goal node. Henceforth, we will refer to nodes in a graph as system states, or simply states, and to the set of nodes in a graph as the state-space.

The most common application of search-based planning is navigation planning or path-finding (Likhachev and Ferguson, 2008; Dolgov et al., 2010). It is also commonly used to solve discrete combinatorial problems, such as various puzzles and games (Holte et al., 1996b). There are several important reasons for the popularity of search-based planners. Firstly, they typically provide strong theoretical guarantees on completeness with respect to the graph representing the search problem, and bounds on solution cost sub-optimality. Usually, in search-based planners one can easily trade-off solution optimality for faster planning time, for example, by varying the heuristic weighting factor ϵ in Weighted A* search, and still have strong guarantees that the cost of the solution is within a desired bound of the optimal solution cost. Second, a number of anytime search algorithms have been developed, that find the best solution they can within a given time limit and continue to improve the solution quality as the planning time allows (ARA* (Likhachev et al., 2003), Anytime A* (Zhou and Hansen, 2002), Beam-Stack Search (Zhou and Hansen, 2005b)). Third, a number of search algorithms can re-use previous search efforts to find new solu-

tions faster (Focussed D* (Stentz, 1995a), Incremental A* (Koenig and Likhachev, 2002b), D* Lite (Koenig and Likhachev, 2002a)). Such algorithms are well-suited for planning in dynamic environments, where fast re-planning is necessary. Finally, formulating the search problem as a cost-minimization problem allows one to define and incorporate complex cost functions and constraints into the planning process. These properties of search-based planning algorithms address common considerations when designing intelligent systems, such as efficiency, response time, and consistency.

Modern intelligent systems, such as robots or virtual agents, need to be able to plan their actions in increasingly more complex state-spaces with many degrees of freedom. These degrees of freedom are often introduced to capture the full capabilities of the system, or to account for its various kinodynamic constraints. In the context of search-based planning, the increasing number of degrees of freedom of the system introduces an exponential increase in the size of the search space, also known as the "curse of dimensionality". Thus, the high dimensionality of the states-space often leads to a dramatic increase in the time and memory required by the search algorithm to find a solution. Dijkstra's graph search algorithm (Dijkstra, 1959) is probably the most well-known graph search algorithm with running time of about $O(|E|\log(|V|))$, depending on the specific implementation, where E is the set of edges and V is the set of nodes in the graph. However, this running time becomes impractical for systems with large number of degrees of freedom, as $|V|$ and $|E|$ scale exponentially with the number of degrees of freedom.

Search-based algorithms try to alleviate the problem by focusing the search efforts in promising directions by using heuristic functions (or simply heuristics) (Hart et al., 1968)—functions that estimate the cost of reaching the goal from every state in the search space. A heuristic is said to be admissible if it never overestimates the cost of reaching the goal. Usually, admissible heuristics are required in order for the search-based planning algorithms to provide guarantees about the cost of the solution. However, some algorithms impose an even stronger requirement on the heuristic functions they are able to use—consistency. A

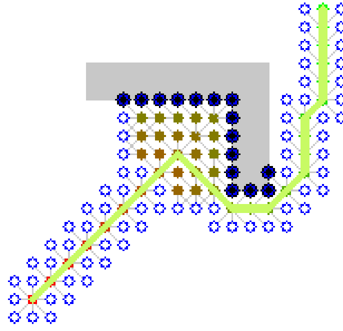


Figure 1: Weighted A^* search ($\epsilon = 5$) on an 8-connected 2D grid using Euclidean distance cost between nodes from start (bottom left) to goal (top right). The heuristic used is Euclidean distance to the goal. The gray shape represents an obstacle. Blue circles represent nodes on the OPEN list, solid colored nodes represent expanded nodes, with color varying from red to green based on increasing g value. Solid blue nodes are invalid nodes that are in collision with the obstacle. The green path represents the solution to the problem found by the weighted A^* search. The shape of the obstacle introduces a local minimum of the heuristic function. All nodes in the local minimum are expanded by the search. Image taken from (Wikipedia, 2015).

heuristic h is said to be consistent (or monotone) if and only if it is admissible and obeys the triangle inequality for any state A in the state-space S and any state $B \in successors(A)$:

$$h(A) \leq cost(A, B) + h(B) \ \forall A \in S, \forall B \in successors(A).$$

These limitations on the heuristic functions, combined with the increasing complexity of the search problems being studied, make it very challenging for researchers to find good heuristics that perform well over a wide range of problems. It is often challenging to find heuristics that perform consistently over a wide variety of problem instances. Pronounced local minima in the heuristic can lead to a significant performance decrease, as the search needs to expand all states in the local minimum in order to overcome it and proceed towards the goal (Fig. 1). In Fig. 1, the heuristic does not take into account obstacles in the search space, and thus, certain obstacle shapes and configurations can produce very large local minima, which significantly degrade the performance of the search.

We have developed a framework for search-based planning, Planning with Adaptive Dimensionality (PAD), that tries to address the size of the state-space and the "curse of dimensionality" for high-dimensional planning problems based on the observation described in the next chapter. We demonstrate that the framework provides the important theoretical properties of search-based planning algorithms—completeness with respect to the graph representing the problem and strong guarantees on solution cost sub-optimality bounds. We have also experimentally validated our framework in a number of planning domains—navigation, manipulation, mobile manipulation, and motion planning for a bipedal humanoid robot.

CHAPTER 2 : Motivating Observation

While planning in a high-dimensional state-space is often necessary to capture the full capabilities of the system and its kinodynamic constraints, large portions of the computed solutions exhibit low-dimensional structures. For example, a 3-DoF (x,y,heading) path for a non-holonomic vehicle typically contains large portions that are straight-line segments and do not therefore require three-dimensional planning. On the other hand, sections of the path that include turning do require planning in all three degrees of freedom in order to capture the minimum turning radius constraints of the system (Fig. 2). Similarly, planning for manipulation can often be reduced to 3-DoF (x, y, z) planning for the manipulator's end-effector position and using an inverse kinematics solver to find a full-dimensional manipulator path that corresponds to the computed end-effector path (Fig. 3 (a)). At the same time, there are situations when the planner does need to consider the full configuration of the arm when trying to ensure the feasibility of the end-effector path—in highly cluttered areas of the environment or certain obstacle configurations (Fig. 3 (b)), for example. Such low-dimensional representations can be found for many robotic systems, as they are inherently embedded into the 2D planar or 3D spatial geometric environments in which they operate. Moreover, a number of informative heuristics exist for such geometric environments, such as various distance metrics, which can even account for the obstacles in the environment. Thus, search in these low-dimensional spaces can be performed quickly and efficiently.

In this work, we present an algorithm framework that exploits this observation. It iteratively constructs a hybrid state-space that utilizes a low-dimensional state representation (abstraction) throughout most of the search space (e.g. end-effector position), except for the areas where low-dimensional planning fails and full-dimensional planning is necessary (e.g. full manipulator configuration) to ensure that the solution is feasible and satisfies a desired cost sub-optimality bound. At each iteration the algorithm identifies areas of the state-space that require high-dimensional planning and introduces them into the hybrid

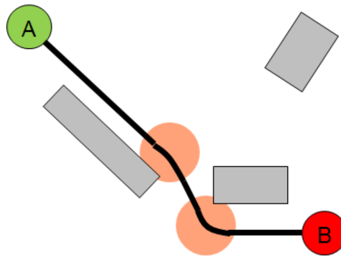


Figure 2: Example trajectory for a non-holonomic vehicle with minimum turning radius constraints. Planning for the heading of the vehicle is needed in areas that require turning in order to ensure constraints are satisfied (light red circles). Planning for the heading of the vehicle is unnecessary for areas that can be traversed in a straight line. A: start location; B: goal location; gray boxes: obstacles.

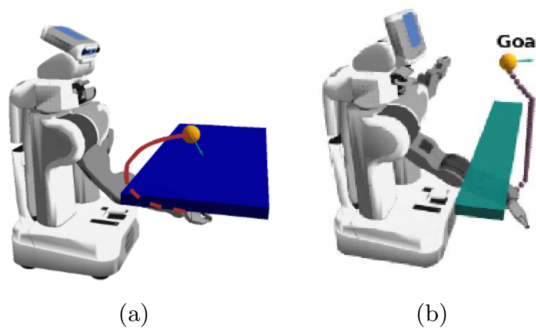


Figure 3: Motion planning for a manipulator: (a) Simple example: planning an end-effector trajectory and using an inverse kinematics solver to compute a corresponding manipulator trajectory. (b) Example: planning an end-effector trajectory combined with an inverse kinematics solver fails to produce a valid trajectory.

graph, until the necessary high-dimensional areas have been identified and the planner is able to compute a feasible solution. Using such low-dimensional abstractions results in substantial reduction in the size of the state-space and considerable speedups in planning time and lower memory requirements of the planner. On the theoretical side, we have shown that the method is complete with respect to the state-space representing the search problem and can provably guarantee to find a solution, if one exists, within a desired cost sub-optimality bound. Additionally, we present a number of extensions of the framework that can further improve its performance.

CHAPTER 3 : Related Work

In order to improve planning times and memory requirements, researchers have used a variety of techniques to avoid performing global planning in large high-dimensional state-spaces.

3.1. State Abstraction Techniques

State abstraction is a general technique for simplifying search problems by reducing the size and complexity of the search space. The general idea is to combine states in the original state-space into abstract states based on pre-defined set of criteria, thus creating a much smaller abstract state-space. A search is then performed on the abstract state-space and the results of the search are used to guide a subsequent search of the original state-space. In other words, abstraction provides a means of automatically creating admissible heuristics for graph search algorithms and has been studied by researchers since the 70's (Guida and Somalvico, 1978; Gaschnig, 1979; Pearl, 1984; Prieditis, 1993). In order for the abstraction to generate an admissible heuristic, the distance between every pair of states A and B in the original state-space S must be no less than the distance between their corresponding abstract states A' and B' in the abstract state-space S' (i.e. the abstraction underestimates distances or costs between states).

$$cost(A, B) \geq cost(A', B') \forall A, B \in S$$

$$A' = image(A) \in S', B' = image(B) \in S'$$

Different hierarchical planners use different methods of abstraction to make better-informed heuristics to guide the search, such as the clique abstraction in (Bulitko et al., 2007) and the max-degree star abstraction in (Holte et al., 1996a,b). The most important difference between such hierarchical planners and our approach is that, rather than computing more

informative heuristic functions, our approach focuses on removing irrelevant dimensions from the planning process itself. These dimensions are only considered in regions of the state-space that require them in order to ensure the feasibility of the resulting solution and its cost sub-optimality bound. Moreover, the abstractions considered by hierarchical planners (Holte et al., 1996b,a; Botea et al., 2004; Bulitko et al., 2007) usually combine states that are adjacent or within a certain distance in terms of number of edges within the state-space. Usually, such abstractions require the full graph representing the state-space to be constructed and stored in memory, which may be infeasible in many high-dimensional systems. Then the abstract graph is constructed and used to compute a heuristic for the problem instance in a pre-processing step that can be quite computationally expensive. Thus, these approaches are not well-suited for dynamic environments. Both the clique (Bulitko et al., 2007) and max-degree star (Holte et al., 1996a,b) abstractions require significant pre-processing. In addition, computing the clique abstraction in a general graph is an NP-complete problem. Bulitko et al. (Bulitko et al., 2007) are able to compute the abstraction efficiently only in 8-connected 2D grids. In contrast, our method uses projection functions to project states to and from the low-dimensional state-space. This allows us to dynamically construct both the low-dimensional and high-dimensional regions of the graph, and thus, we do not need to pre-allocate memory for the entire graph.

Our approach is also somewhat relevant to planners that use very accurate pre-computed heuristic values (Knepper and Kelly, 2006). Similarly to the hierarchical planners using state abstraction, the heuristics are often derived by solving a simplified lower-dimensional problem. As a result, these methods can be viewed as full-dimensional planning that uses the results of lower dimensional planning. Unlike our approach however, these methods do not explicitly decrease the dimensionality and, as a result, can run into severe computational problems when the heuristic is inaccurate. As mentioned above, our approach does not focus on computing accurate heuristics, but rather decreases the dimensionality of the problem in order to explicitly reduce the size of the state-space. In addition, our framework for Planning with Adaptive Dimensionality can use and benefits from accurate heuristics. However, due

to the reduced size of the state-space, our approach is more robust to handling possible heuristic local minima than approaches that perform full-dimensional planning. Thus, the performance of our approach does not rely solely on the quality of the heuristic.

Kapadia et al. (Kapadia et al., 2013) use an approach very similar to ours. They use the same concept of a “tunnel” around a low-dimensional path, which we introduce in the following chapter, to focus and constrain a subsequent search of a high-dimensional space. They also incorporate multiple low-dimensional representations in their planning framework to form an abstraction hierarchy, which we discuss in Chapter 5, and use an incremental graph search algorithm to speed-up subsequent search queries, which we discuss in Chapter 6. Their approach, however, is significantly different than ours in that they do not use hybrid graphs containing both low- and high-dimensional states, which we use to ensure the completeness of our algorithm and that the desired cost sub-optimality bound is met. In contrast, their approach relies on increasing the width of the “tunnel” until a valid high-dimensional path is found through it. Moreover, their approach does not provide bounds on the solution cost sub-optimality.

3.2. Two-Layer Planners

Many path planners implement a two layer planning scheme, where a low-dimensional global planner provides input to a high-dimensional local planner. Since these local planners operate on a small subset of the entire environment, usually in the immediate vicinity of the robot, they can afford to incorporate more dimensions, while still meeting planning time constraints. The local planners have been implemented using reactive obstacle avoidance planners (Thrun et al., 1998) and dynamic windows (Philippsen and Siegwart, 2003; Brock and Khatib, 1999) to produce feasible paths from an underlying low-dimensional global planner. However, these techniques can result in highly sub-optimal paths and even paths that are infeasible for execution by the system due to mismatches in the assumptions made by the global and the local planners. In contrast, our approach does not split the planning process into two fixed layers, but rather mixes the different dimensionality of the planning

problem within a single planning process. By combining the abstract and non-abstract state representations in a single hybrid graph, our framework is able to identify areas exhibiting inconsistencies between the low-dimensional and high-dimensional state representations and remedy these inconsistencies by requiring high-dimensional planning to be performed in those areas. Moreover, our approach is complete with respect to the full-dimensional state-space and guarantees to compute a path that is feasible in the full-dimensional state-space if one exists.

3.3. Sampling-Based Planners

Sampling-based motion planners, such as probabilistic roadmaps (PRM) (Kavraki et al., 1996; Bohlin and Kavraki, 2000), and rapidly-exploring random trees (RRT) (LaValle and Kuffner, 2001a) and its variants (Kuffner and LaValle, 2000; Berenson et al., 2009, 2011; Karaman et al., 2011) have become extremely popular in recent years for solving high-dimensional planning problems. They have been shown to solve impressive high-dimensional motion planning problems, while being simple, fast, and easy to implement. These methods have also been extended to support motion constraints through rejection sampling (Sucan and Kavraki, 2009).

Our search-based approach to planning differs from the sampling-based methods in several important aspects. First, sampling-based motion planners are mainly concerned with finding any feasible path, rather than minimizing the cost of a solution. The notable exception is the RRT* algorithm (Karaman et al., 2011), which asymptotically converges to an optimal solution and is one of the algorithms that we compare our approach against experimentally. In general, sampling-based approaches sacrifice cost minimization in order to gain very fast planning speeds. As such, they may often produce solutions of unpredictable length involving highly sub-optimal or jerky motions that may be hard for the system to execute. To compensate for the lack of solution cost minimization, sampling-based methods rely on various smoothing techniques to improve the quality of the computed trajectory. While often helpful, smoothing may fail in cluttered environments. Sampling-based meth-

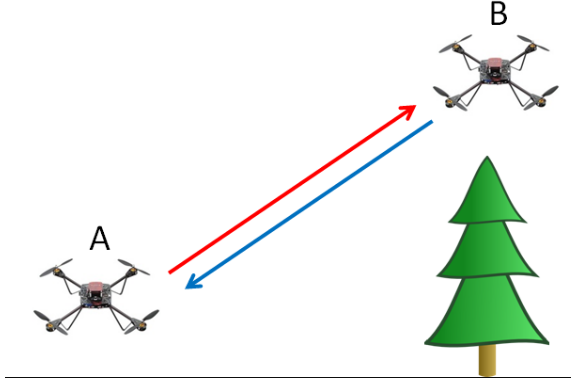


Figure 4: Example of an asymmetric cost function: energy consumption effect of changing altitude for a UAV. Transitioning from state A to state B (red arrow) is more costly than transitioning from state B to state A (blue arrow). Thus, in terms of cost, moving from B to A is closer than moving from A to B, which makes the cost function not a proper distance metric. Such cost functions present a challenge for sampling-based algorithms.

ods which aim to provide solution cost minimization, such as RRT*, usually have to use a distance metric for their cost function and do not support arbitrary cost functions, due to the requirement for solving k-nearest neighbors queries in the cost space. However, in robotics cost functions are often non-symmetric, which violates the distance metric requirements. In many systems the cost of transitioning from state A to state B is not necessarily equal to the cost of transitioning from B to A in the state-space S.

$$\exists A, B \in S \text{ s.t. } cost(A, B) \neq cost(B, A)$$

For example, the energy consumption for a unmanned aerial vehicle (UAV) is higher for increasing altitude than it is for decreasing altitude (Fig. 4). Thus, if the cost function is based on energy consumption, the costs of actions that involve changes in altitude are highly asymmetric. Moreover, it is generally much easier and safer for a robot to move forward than backward, as sensors are usually located at the front of the robot. Thus, moving backward is often considered undesirable and such actions are associated with much higher costs than equivalent actions for moving forward. Such asymmetry in the cost function is

easily handled by search-based planners by using a directed graph to represent the problem.

Another difference between search- and sampling-based planning methods is that search-based planners produce more consistent solutions between planning episodes with similar start and goal configurations. Due to their randomized nature, sampling-based methods may often produce solutions of unpredictable length that can be inconsistent from one planning episode to another. It is often preferable for planners to produce similar solutions for planning queries with similar start/goal configurations. Consistency of planners is an important consideration when the system needs to operate in environments with proximity to humans. Humans need to be able to predict and anticipate the behavior of the system in order to feel safe and comfortable around it. In our experimental evaluation we compute a consistency measure of the trajectories produced by our approach and sampling-based alternatives and compare the results.

Finally, the performance of sampling-based methods can suffer significantly in very cluttered environments with narrow solution spaces. Researchers have tried to address this problem by developing non-uniform sampling techniques that try to identify narrow passages and bias sampling in those areas (Lee et al., 2012). However, one must be very careful how one biases the sampling, as biased sampling can often break the algorithm's probabilistic completeness, and in the case of RRT*, even its guarantee to converge to optimality.

There are certain similarities between search- and sampling-based methods. For example, methods based on PRM (Kavraki et al., 1996) only differ from search-based methods in the way the underlying graph is constructed. PRM-based methods use sampling to generate nodes in the graph, rather than regular discretization of the continuous space. However, both PRM methods and search-based methods rely on efficient graph search algorithms in order to compute a solution from the underlying graph.

3.4. Optimization Methods

Several motion planning algorithms have been developed that also try to minimize the cost of solutions through optimization techniques (Ratliff et al., 2009; Kalakrishnan et al., 2011). The Covariant Hamiltonian Optimization and Motion Planning (CHOMP) algorithm (Ratliff et al., 2009) works by creating a naive initial trajectory from start to goal, and then uses a method similar to gradient descent to try to minimize the cost function. The use of gradient descent, however, makes the approach vulnerable to local minima in the cost function. The final solution of the algorithm usually lies in the same homotopic class as the initial estimate. Thus, the approach needs an initial solution estimate that is fairly close to the optimal in order to converge to global optimality. The Stochastic Trajectory Optimization for Motion Planning (STOMP) algorithm (Kalakrishnan et al., 2011), on the other hand, relies on generating noisy trajectories to explore the space around a naive initial trajectory. It then iteratively combines these trajectories to produce an updated trajectory with lower cost. A cost function based on a combination of obstacle avoidance and path smoothness is optimized in each iteration. The stochastic nature of the approach makes it less vulnerable to local minima in the cost function, but does not guarantee convergence to global optimality.

3.5. Incremental Search Algorithms

Researchers have developed various methods for performing incremental heuristic searches, based on the observation that information computed during previous search queries can be used to perform the current search faster. Due to its iterative nature, our framework for Planning with Adaptive Dimensionality can benefit greatly from incremental graph search algorithms in order to minimize redundant computation between iterations. Generally, incremental heuristic search algorithms fall into three categories.

The first class of algorithms, such as Lifelong Planning $A^\star$ (Koenig et al., 2004), $D^\star$ (Stentz, 1995b), $D^\star$-Lite (Koenig and Likhachev, 2002a), Anytime $D^\star$ (Likhachev et al., 2005), and

Anytime Truncated $D^\star$ (Aine and Likhachev, 2013), aim to identify and repair inconsistencies in a previously-generated search tree. These approaches are very general and don't make limiting assumptions about the structure or behavior of the underlying graph. They also demonstrate excellent performance by repairing search tree inconsistencies that are relevant to the current search task. The main drawback of these algorithms is the bookkeeping overhead required, which sometimes may significantly offset the benefits of avoiding redundant computation.

The second class of algorithms, such as Fringe-Saving $A^\star$ (Sun et al., 2009) and Differential $A^\star$ (Trovato and Dorst, 2002), also try to re-use a previously-generated search tree, but rather than attempting to repair it, these approaches aim to identify the largest portion of the search tree that is unaffected by the changes and still valid, and resume searching from there. These approaches tend to be less general and to make limiting assumptions about the graph on which they operate. The Fringe-Saving $A^\star$, for example, only works on 2D grids with unit cost transitions between neighboring cells. It uses geometric techniques to reconstruct the search frontier based on the 2D grid structure of the graph. The assumptions made by these algorithms allow them to perform very well in scenarios that meet these limiting assumptions.

The third class of incremental heuristic search algorithms, such as Generalized Adaptive $A^\star$ (Sun et al., 2008), aim to compute more accurate heuristic values by using information from previous searches. As the heuristic becomes more informative, search tasks are performed faster. The main challenge for such algorithms is maintaining the admissibility or consistency of the heuristic when edge costs are allowed to decrease. Path- and Tree-Adaptive $A^\star$ (Hernández et al., 2011) algorithms, for example, rely on the fact that optimal search is performed on the graph and edge costs are only allowed to increase. However, often in robotics incremental search algorithms need to be able to support both increasing and decreasing edge costs to capture obstacles appearing and disappearing from the environment. A typical example in navigation planning is opening and closing doors or passageways in

the environment. Moreover, performing optimal search is often impractical for systems with a large number of degrees of freedom, making approaches that allow for trade-off between solution quality and planning time more appealing for such systems.

We have developed a novel general anytime incremental graph search algorithm that works well with our framework for Planning with Adaptive Dimensionality. We call the algorithm Tree-Restoring Weighted $A^{\star}$ and it falls into the second class of incremental search algorithms described above, which identify and reuse valid portions of the search tree generated by the previous search iteration. However, our approach differs from similar techniques in that we do not make any limiting assumptions about the structure of the graph or the behavior of the cost function. Our algorithm is able to support both increasing and decreasing edge costs on arbitrary graphs. The algorithm is an extension to the Anytime Repairing $A^{\star}$ ($ARA^{\star}$) algorithm (Likhachev et al., 2003) and has the same strong theoretical properties, such as completeness and provable bounds on solution cost sub-optimality. Similarly to $ARA^{\star}$, our algorithm allows for trading-off between solution quality and planning time. Moreover, the algorithm has much lower book-keeping overhead when compared to alternative approaches, such as $D^{\star}$ (Stentz, 1995b), $D^{\star}$-Lite (Koenig and Likhachev, 2002a), Anytime $D^{\star}$ (Likhachev et al., 2005), and Anytime Truncated $D^{\star}$ (Aine and Likhachev, 2013).

CHAPTER 4 : Planning with Adaptive Dimensionality

In this chapter we provide a detailed description of our algorithm for Planning with Adaptive Dimensionality (PAD). We begin by introducing the important assumptions, definitions, and notations used.

4.1. Definitions and Notations

We assume that the planning problem is represented by a discretized finite state-space S of dimensionality d, consisting of states represented by state-vectors $X = (x_1, ..., x_d)$, and a set of transitions $T = \{(X_i, X_j) | X_i, X_j \in S\}$. Each transition (X_i, X_j) corresponds to a feasible transition between the corresponding state vector values and is associated with a cost $c(X_i, X_j)$ which is bounded from below by some positive δ, that is, $c(X_i, X_j) > \delta > 0$. Thus, we have an edge-weighted directed graph G with a vertex set S and edge set T. We will use the notation $\pi_G(X_i, X_j)$ to denote a path in graph G from state X_i to state X_j. The cost of any path $\pi_G(X_i, X_j)$ is the cumulative costs of the transitions along it and will be denoted by $c(\pi_G(X_i, X_j))$. We will use $\pi^*_G(X_i, X_j)$ to denote a least-cost path and $\pi^\epsilon_G(X_i, X_j), \epsilon \geq 1$ to denote a path of bounded cost sub-optimality $c(\pi^\epsilon_G(X_i, X_j)) \leq \epsilon \cdot c(\pi^*_G(X_i, X_j))$. The goal of the planner is to find a least-cost path in G from the start state X_S to the goal state X_G. Alternatively, given a desired sub-optimality bound $\epsilon \geq 1$, the goal of the planner is to find a path $\pi^\epsilon_G(X_S, X_G)$.

Definition 4.1 *A heuristic function h is said to be **admissible** for a graph search problem on an edge-weighted graph G and a goal state $g \in G$ if*

$$h(s) \leq c(\pi^*_G(s, g)) \forall s \in G$$

*i.e. the heuristic never overestimates the optimal cost of reaching the goal. Such heuristic functions are sometimes called **optimistic**.*

Definition 4.2 *A heuristic function h is said to be **consistent** or **monotone** for a graph*

search problem on an edge-weighted graph G and a goal state $g \in G$ if

$$h(s) \leq c(s,t) + h(t) \forall s \in G, \forall t \in successors(s)$$
$$h(g) = 0$$

4.2. Overview

Let us consider two state-spaces—a high-dimensional S^{HD} with dimensionality h, and a low-dimensional S^{LD} with dimensionality l, which is a projection of S^{HD} onto a lower dimensional manifold ($h > l, |S^{HD}| > |S^{LD}|$). We define a many-to-one mapping

$$\lambda : S^{HD} \rightarrow S^{LD}$$

from the high-dimensional state-space S^{HD} to the low-dimensional state-space S^{LD}. For example, in the case of navigation planning for a non-holonomic vehicle in 3 dimensions (x, y, heading) described in Chapter 7 we used a 2-dimensional state representation (x, y) and the simple mapping $\lambda((x, y, \theta)) = (x, y)$, just dropping the heading information θ from the state-vector for low-dimensional states.

We also define the mapping $\lambda^{-1} : S^{LD} \rightarrow (S^{HD})^*$ from the low-dimensional state-space S^{LD} to subsets of the high-dimensional state-space S^{HD}, defined by

$$\lambda^{-1}(X^{LD}) = \{X \in S^{HD} | \lambda(X) = X^{LD}\}$$

Notice that λ^{-1} is a one-to-many mapping and produces a set of high-dimensional states corresponding to a given low-dimensional state X^{LD}—the set of pre-images of X^{LD}.

Each of the two state-spaces may have its own transition set. For example, in the 3D/2D navigation planning scenario described in Chapter 7 we used 8-connected 2D grid transitions for the 2D state-space, and a set of precomputed feasible atomic actions that capture the

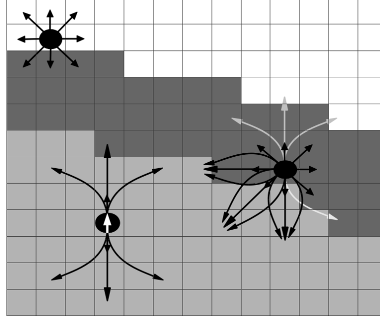


Figure 5: Example state transitions for a 3D/2D state-space–white cells are 2D states (x, y), dark gray cells are 2D states with feasible 3D transitions to 3D states $(x, y, \text{heading})$, and the light gray cells are 3D states. On the upper left is shown a 2D state with all of its feasible transitions (only 2D transitions). The state in the middle right is in the boundary area, so its feasible transitions include all 2D transitions that end in a 2D state and all 3D transitions (from all possible heading values) that end in a 3D state. In light gray are shown some of the disallowed 3D transitions, since they lead to 2D states. In the lower left is a 3D state with all of its 3D transitions (heading indicated by the white arrow).

kinodynamic constraints of the vehicle, called motion primitives (Likhachev and Ferguson, 2008), as transitions for the 3D state-space (Fig. 5).

Let G^{HD} and G^{LD} denote the corresponding graphs defined by S^{HD} and S^{LD} and their respective transition sets T^{HD} and T^{LD}.

The idea of our algorithm is to iteratively construct and search a hybrid graph G^{AD} consisting of both low- and high-dimensional states and transitions. Initially G^{AD} is identical to G^{LD}. The iterative nature of the algorithm stems from the fact that each iteration identifies new areas of G^{AD} where high-dimensional regions need to be introduced until a valid solution is found. Upon addition of new high-dimensional regions into G^{AD}, another search iteration is performed on the new instance of G^{AD} taking into account the new high-dimensional regions. The process is repeated until a search iteration is able to successfully compute a solution that is feasible in the high-dimensional state-space and satisfies the specified cost sub-optimality bound. We discuss the structure and the construction of G^{AD} below.

In order to provide guarantees on bounded solution cost sub-optimality, we require that the

costs of the transitions in T^{HD} and T^{LD} be such that for every pair of states X_i and X_j in S^{HD},

$$c\left(\pi^*\left(X_i, X_j\right)\right) \geq c\left(\pi^*\left(\lambda(X_i), \lambda(X_j)\right)\right) \tag{4.1}$$

That is, we require that the cost of a least-cost path between any two states in the high-dimensional state-space to be at least the cost of a least-cost path between their images in the low-dimensional state-space. The intuition behind this requirement is that path segments through the low-dimensional areas of the state-space provide optimistic estimates of the true cost of their high-dimensional images. These optimistic estimates are used to establish a lower bound on the overall optimal solution cost. The algorithm then uses this lower bound to ensure that the final solution cost is within the desired sub-optimality factor of the optimal solution cost.

Then let us formally define a state-abstraction in the context of Planning with Adaptive Dimensionality as follows:

Definition 4.3 *A state-abstraction of a state-space* S^{HD} *is a tuple* $\mathcal{A} = (\lambda, \lambda^{-1}, G^{LD} = (S^{LD}, T^{LD}), c)$*, where:*

- S^{LD} *is a projection of* S^{HD} *to a lower-dimensional sub-space of* S^{HD} *through a projection function* $\lambda : S^{HD} \to S^{LD}$
- $\lambda^{-1} : S^{LD} \to (S^{HD})^*$ *is defined as* $\lambda^{-1}(X^{LD}) = \{X \in S^{HD} | \lambda(X) = X^{LD}\}$
- $G^{LD} = (S^{LD}, T^{LD})$ *is an edge-weighted (directed) graph with vertex set* S^{LD} *and transition set* T^{LD}
- $c : T^{LD} \to \mathbb{R}^+$ *is a cost function satisfying 4.1*

When referring to the full-dimensional abstraction, we mean the identity abstraction of the full-dimensional state space: $\mathcal{H} = (\lambda_{HD}, \lambda_{HD}^{-1}, G^{HD} = (S^{HD}, T^{HD}), c_{HD})$, where λ_{HD} and

λ_{HD}^{-1} are both equal to the identity mapping over S^{HD} ($\forall X \in S^{HD}$ $\lambda_{HD}(X) = X$ and $\lambda_{HD}^{-1}(X) = \{X\}$).

4.3. Hybrid State-Space Construction

4.3.1. Structure of the Hybrid Graph

Recall that the goal of our algorithm was to use the faster low-dimensional planning, except for areas of the environment where high-dimensional planning is necessary to ensure the feasibility of the resulting path and the desired cost sub-optimality bound. We want our hybrid state-space to capture this property—namely, we want G^{AD} to consist largely of low-dimensional states, except for the areas where high-dimensional planning needs to be performed, represented by areas of high-dimensional states in G^{AD}. To ensure path feasibility in the high-dimensional regions of G^{AD}, we have to use high-dimensional transitions. In the low-dimensional areas we can use simpler low-dimensional transitions. However, recall that the transitions we have in T^{HD} and T^{LD} connect two states of the same dimensionality, which do not allow us to transition from the low-dimensional to the high-dimensional regions. Therefore, we have to construct a transition set T^{AD} that allows for transitions between states of different dimensionality.

4.3.2. Construction of the Hybrid Graph

Our algorithm iteratively constructs G^{AD}, beginning with the low-dimensional state-space S^{LD} and introducing a set of high-dimensional regions R in it. We first explain how the high-dimensional regions are being introduced into G^{AD} and connected with the low-dimensional regions. The algorithm that decides when and where to introduce these regions will be explained later.

Once a high-dimensional region r is introduced, the following changes are made to G^{AD}. If a low-dimensional state X_i^{LD} falls inside a new high-dimensional region $r \in R$, we replace it with its high-dimensional projection states in $\lambda^{-1}(X_i^{LD})$. Thus, G^{AD} contains both low-

dimensional and high-dimensional states. Notice that if a high-dimensional state X^{HD} is in S^{AD}, then its low-dimensional projection $\lambda(X^{HD})$ is not in S^{AD}, and also if $X^{HD} \notin S^{AD}$, then $\lambda(X^{HD}) \in S^{AD}$. Thus, for every state X^{HD} in the original high-dimensional state-space, either $X^{HD} \in S^{AD}$ or $\lambda(X^{HD}) \in S^{AD}$ (but not both). Adding new high-dimensional regions or increasing the sizes of existing regions requires the reconstruction of S^{AD} and T^{AD}, and thus, will produce a new instance of $G^{AD} = (S^{AD}, T^{AD})$.

Next we define the transition set T^{AD} for the hybrid graph G^{AD} as follows.

Definition 4.4 ***Transitions in*** G^{AD}***:*** *For any state* $X_i \in S^{AD}$*:*

- *If* X_i *is high-dimensional (*$X_i \in S^{HD}$*), then for all high-dimensional transitions* $(X_i, X_j^{HD}) \in T^{HD}$*, if* $X_j^{HD} \in S^{AD}$ *then* $(X_i, X_j^{HD}) \in T^{AD}$*. If* $X_j^{HD} \notin S^{AD}$*, then* $(X_i, \lambda(X_j^{HD})) \in T^{AD}$*. That is, for high-dimensional states we allow only high-dimensional transitions to other high-dimensional states if they fall inside* S^{AD}*, or their low-dimensional projections (Fig. 5 lower left).*

- *If* X_i *is low-dimensional (*$X_i \in S^{LD}$*), then for all low-dimensional transitions* $(X_i, X_j^{LD}) \in T^{LD}$*, if* $X_j^{LD} \in S^{AD}$ *then* $(X_i, X_j^{LD}) \in T^{AD}$ *and for all high-dimensional transitions* $(X, X_j^{HD}) \in T^{HD}$*, where* $X \in \lambda^{-1}(X_i)$*, if* $X_j^{HD} \in S^{AD}$ *then* $(X_i, X_j^{HD}) \in T^{AD}$*. That is, for low-dimensional states we allow low-dimensional transitions if they lead to another low-dimensional state in* S^{AD} *(Fig. 5 upper left), and high-dimensional transitions from their high-dimensional projections if they lead to a high-dimensional state in* S^{AD} *(Fig. 5 right).*

Notice, that the above definition of T^{AD} allows for transitions between states of different dimensionality. Figure 5 illustrates the set of transitions in the adaptive graph in the case of 3D (x, y, θ) path planning.

4.3.3. Mapping Hybrid Solutions to the High-Dimensional State-Space

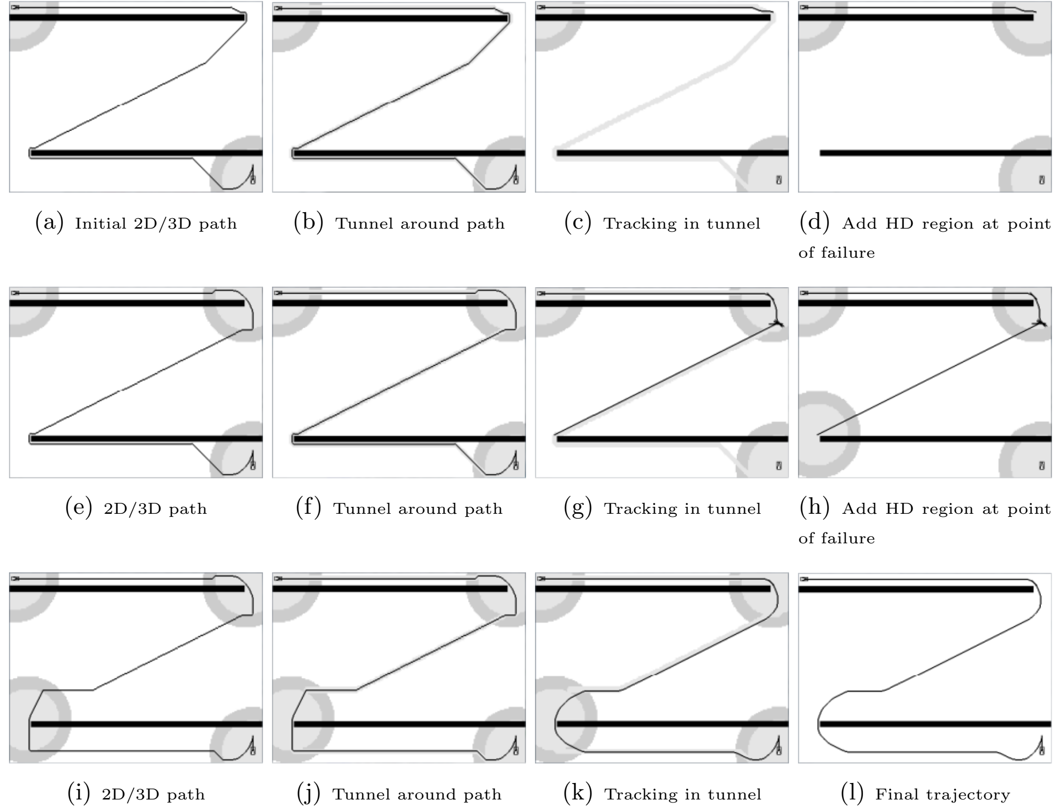
Once we have computed a path through our hybrid graph G^{AD}, which can contain low-dimensional states and transitions, we need to be able to project it to the high-dimensional state-space in order to ensure that it is feasible and satisfies the desired solution cost sub-optimality bound. Therefore, we define a tunnel τ of radius w around a hybrid path π_{AD} as follows:

Definition 4.5 *A tunnel τ of width w around a hybrid path π_{AD} is a sub-graph $\tau = (S^\tau, T^\tau)$ $\tau \subseteq G^{HD}$ such that*

$$S^\tau \subseteq S^{HD}$$
$$T^\tau \subseteq T^{HD}$$
$$\forall X^{HD} \in S^{HD}, X^{HD} \in S^\tau \text{ iff } \exists X_i \in \pi_{AD} \text{ s.t.}$$
$$dist(\lambda(X^{HD}), X_i) \leq w \text{ if } X_i \in S^{LD} \text{ or}$$
$$dist(\lambda(X^{HD}), \lambda(X_i)) \leq w \text{ if } X_i \in S^{HD}$$
$$\forall E^{HD} = (X_i, X_j) \in T^{HD}, E^{HD} \in T^\tau \text{ iff } X_i \in \tau \text{ and } X_j \in \tau$$

where dist is some pre-defined distance metric in S^{LD}.

In other words, τ is a sub-graph of G^{HD}, and thus consists only of high-dimensional states and transitions. Moreover, τ contains all high-dimensional states X^{HD} if they fall within distance w of some state $X_i \in \pi_{AD}$. We include in τ all transitions (X_j, X_k) from T^{HD} such that both X_j and X_k are in τ. It is important to note that the above definition of τ for tunnel width $w = 0$ becomes equivalent to the sub-graph produced by projecting the hybrid path π_{AD} to the high-dimensional state-space S^{HD} through the projection function λ^{-1}. This λ^{-1} projection method can be used when no distance metric is available in the low-dimensional state-space. The above definition, however, allows for more flexibility when mapping hybrid paths into the high-dimensional state-space S^{HD}. To produce a high-



(a) Initial 2D/3D path (b) Tunnel around path (c) Tracking in tunnel (d) Add HD region at point of failure

(e) 2D/3D path (f) Tunnel around path (g) Tracking in tunnel (h) Add HD region at point of failure

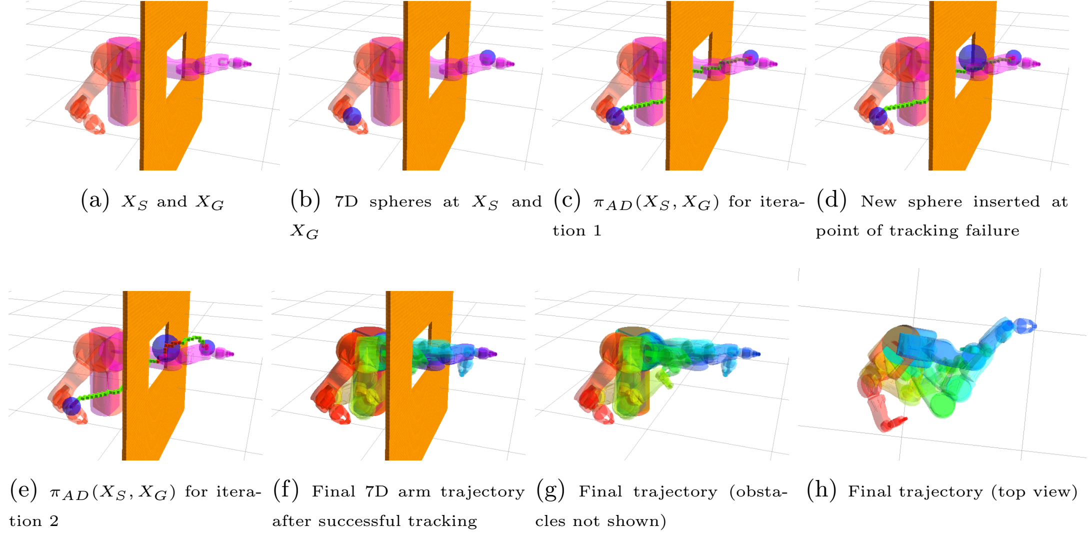
(i) 2D/3D path (j) Tunnel around path (k) Tracking in tunnel (l) Final trajectory

Figure 6: Example of the iterative process of Planning with Adaptive Dimensionality on simple map in the context of 3D (x,y,heading) path planning for a non-holonomic vehicle. Start: top left; goal: bottom right; light gray circles: 3D regions; darker gray outer circles: borders between 2D and 3D regions consisting of 2D states which have valid 3D transitions going into the 3D areas; white: 2D regions; black bars: obstacles.

dimensional path from a hybrid path π_{AD}, we construct a tunnel τ around π_{AD}; then we perform a graph search from start to goal in τ, which is a small sub-graph of the original high-dimensional state-space. The search, if successful, produces a fully high-dimensional path π_{HD} corresponding to our hybrid path π_{AD}.

4.4. Algorithm

We begin this section with an intuitive description of our algorithm for Planning with Adaptive Dimensionality. Figure 6 provides an illustration of a run of the algorithm for 3D (x, y, θ) path planning, that completed in 3 iterations. Figure 7 provides an illustration of a run of the algorithm for 7-DoF motion planning for a robotic manipulator, that completed



(a) X_S and X_G (b) 7D spheres at X_S and X_G (c) $\pi_{AD}(X_S, X_G)$ for iteration 1 (d) New sphere inserted at point of tracking failure

(e) $\pi_{AD}(X_S, X_G)$ for iteration 2 (f) Final 7D arm trajectory after successful tracking (g) Final trajectory (obstacles not shown) (h) Final trajectory (top view)

Figure 7: Example of the iterative process of Planning with Adaptive Dimensionality on simple environment (a wall with an opening) in the context of 7D motion planning for a robotic manipulator using 3D end-effector (x, y, z) position low-dimensional representation. 3D states are represented by squares. Dark gray spheres represent the regions in which 7D planning is performed; 3D planning is performed in all other regions.

in 2 iterations. Algorithm 1 gives the pseudo code for our algorithm.

Each iteration of the algorithm consists of two phases—an adaptive planning phase (Fig. 4.6(a), Alg. 1 line 5) and a path tracking phase (Fig. 4.6(b) - 4.6(d), Alg. 1 line 10). In the adaptive planning phase, the current instance of the hybrid graph G^{AD} is searched for a least-cost path from start to goal. The tracking phase, then attempts to construct a feasible high-dimensional path to match (or track) the hybrid path computed in the adaptive planning phase.

Initially, G^{AD} is the same as G^{LD}, with two high-dimensional regions added around the start and goal states (Algorithm 1, lines 1-3), which are necessary since the start and goal states provided to the planner are high-dimensional. At each iteration, a new instance of G^{AD} is constructed based on the set of high-dimensional regions, and is searched for a least-cost path π^*_{AD} from X_S to X_G. Notice that π^*_{AD} consists of both low-dimensional and high-dimensional states, so it is not a feasible path. If no path is found in the adaptive planning

Algorithm 1 Planning with Adaptive Dimensionality

```
1: G^{AD} = G^{LD}
2: Add-HD-Region(G^{AD}, λ(X_S))
3: Add-HD-Region(G^{AD}, λ(X_G))
4: loop
5:                                                    ▷ Adaptive Planning Phase
6:     search G^{AD} for least-cost path π*_{AD}(X_S, X_G)
7:     if π*_{AD}(X_S, X_G) is not found then
8:         return no path from X_S to X_G exists
9:     end if
10:                                                   ▷ Tracking Phase
11:    construct a tunnel τ around π*_{AD}(X_S, X_G)
12:    search τ for least-cost path π*_τ(X_S, X_G)
13:    if π*_τ(X_S, X_G) is not found then
14:        find state(s) X_r where to insert new HD region(s)
15:        Add-or-Grow-HD-Region(G^{AD}, X_r)
16:    else if c(π*_τ(X_S, X_G)) > ε_track · c(π*_{AD}(X_S, X_G)) then
17:        find state(s) X_r where to insert new HD region(s)
18:        Add-or-Grow-HD-Region(G^{AD}, X_r)
19:    else
20:        return π*_τ(X_S, X_G)
21:    end if
22: end loop
```

phase, then no feasible path exists from start to goal and the algorithm terminates. If an adaptive path π^*_{AD} is found, then the path tracking phase constructs a tunnel τ of radius w around the adaptive path π^*_{AD} (Fig. 4.6(b)). Then τ is searched for a least-cost path π^*_τ from start to goal (Fig. 4.6(c)). Note that τ always contains the start and goal states X_S and X_G, but does not guarantee that X_G is reachable from X_S, so π^*_τ may not exist. In addition, note that since τ consists of only high-dimensional states and transitions, π^*_τ (if it exists) is a fully high-dimensional path, and thus, it is feasible. If no path is found in τ, then a new high-dimensional region is introduced in G^{AD} or the sizes of the existing regions are increased, and the algorithm proceeds to the next iteration (Algorithm 1, line 14). If a path is found in τ, but its cost $c(\pi^*_\tau) > \epsilon_{\text{track}} \cdot c(\pi^*_{AD})$, then a new high-dimensional region is introduced or the sizes of existing high-dimensional regions are increased, and another iteration is started (Algorithm 1, line 17). If $c(\pi^*_\tau) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{AD})$, then the algorithm returns π^*_τ as a feasible path from start to goal that satisfies the desired sub-optimality bound and terminates (Algorithm 1, line 20). The returned path is guaranteed to have cost

that is no more than ϵ_{track} times the cost of an optimal path in G^{HD}.

$$c(\pi_\tau^*) \leq \epsilon_{\text{track}} \cdot c(\pi_{HD}^*).$$

4.5. Identifying Areas that Require High-Dimensional Planning

Identifying the places where high-dimensional regions need to be introduced is a non-trivial problem in itself. In our experiments, the search within the tunnel during the path tracking phase keeps a record of how far along the tunnel states have been expanded. Thus, if the search in τ fails, we are able to reconstruct a path to the point where the search had failed, and we introduce a new high-dimensional region there, as seen in Fig. 4.6(c),4.6(d),4.6(g), and 4.6(h).

The way we keep track of how far along a tunnel τ around a hybrid path π_{AD} the search has reached is the following. By Definition 4.5, for every state $X_i \in \tau$ there exists a nearest state $X_j \in \pi_{AD}$ according to our distance metric $dist$. More specifically, $\forall X \in \pi_{AD}$ $dist(\lambda(X_i), X) \geq dist(\lambda(X_i), X_j)$. Thus, when the search through τ expands a state X_i it can compute the corresponding nearest state $X_j \in \pi_{AD}$ and its sequence number n in the hybrid path π_{AD}. Therefore, if the search through τ keeps track of the highest sequence number N that has been encountered during the search, upon search failure we can say that the search was able to reach near to the N-th state along π_{AD} before getting "stuck". In our experiments, we have found that this is an effective strategy that can be used on line 14 of Alg. 1. It works well in identifying areas that cause the tunnel τ to be disconnected indicating a mismatch between the low- and high-dimensional state-spaces, and thus, the region requires high-dimensional planning.

Line 17 of Alg. 1 is obscure about how exactly the state X_r, where a new high-dimensional region needs to be introduced, is being computed when a path through τ exists, but it is too costly. There are a number of approaches that can be taken in identifying such a state. Perhaps the simplest one is to pick a random location along the path π_{AD}^* where to

introduce a new region. However, such an approach can lead to the introduction of many unnecessary high-dimensional regions, which we would like to avoid. A more sophisticated technique, which we use in our implementation, is to approximate the location, where the largest cost discrepancy between π^*_{AD} and π^*_{τ} is observed. We do this similarly to the way we keep track of progress along the tunnel described above. We find correlating states between the two paths $X_i \in \pi^*_{\tau}$ and $X_j \in \pi^*_{AD}$ such that X_j is the state in π^*_{AD} nearest to $\lambda(X_i)$; then we compare the cumulative costs along both paths for reaching X_i and X_j, respectively. If the cumulative path cost along π^*_{τ} exceeds the cumulative path cost along π^*_{τ} by more than a factor of ϵ_{track}, we introduce a new high-dimensional region at the location of X_j. Introducing a new high-dimensional region at that location tends to remedy the cost discrepancy, and generally works well in identifying the regions that require high-dimensional planning. The exact approach taken in computing X_r on line 17 of Algorithm 1 does not affect the theoretical properties of the algorithm, such as algorithm termination and sub-optimality guarantees. However, it can have a significant effect on the performance of the algorithm as its underlying idea is to efficiently identify the regions that require high-dimensional planning and refrain from introducing unnecessary high-dimensional regions into the state-space.

4.6. Theoretical Properties

In this section we present a number of theorems relating to the algorithm for Planning with Adaptive Dimensionality and provide sketches of their proofs. For detailed proofs we refer the reader to Appendix A.

The algorithm for Planning with Adaptive Dimensionality presented in Alg. 1 is complete with respect to G^{AD} and provides guarantees on the sub-optimality related to the ϵ_{track} constant.

Theorem 4.1 *The cost of a least-cost path from X_S to X_G, $\pi^*_{AD}(X_S, X_G)$, in G^{AD} is a*

lower bound on the cost of a least-cost path from X_S to X_G, $\pi^*_{HD}(X_S, X_G)$, in G^{HD}.

$$c(\pi^*_{AD}(X_S, X_G)) \leq c(\pi^*_{HD}(X_S, X_G))$$

Proof Consider the projection of the path $\pi^*_{HD}(X_S, X_G)$ onto the hybrid state-space S^{AD}. In this projection, every state X in $\pi^*_{HD}(X_S, X_G)$ is mapped onto itself if $X \in S^{AD}$ and onto $\lambda(X)$ otherwise. Then according to equation 4.1, every transition T_i in the projected version of the path $\pi^*_{HD}(X_S, X_G)$ will either be bounded from above by the cost of the corresponding transition in $\pi^*_{HD}(X_S, X_G)$ if T_i is a low-dimensional transition, or will be exactly equal to the cost of the corresponding transition if T_i is a high-dimensional transition. Consequently, the cost of the projected version of $\pi^*_{HD}(X_S, X_G)$ will be no larger than $c(\pi^*_{HD}(X_S, X_G))$. Furthermore, since $\pi^*_{AD}(X_S, X_G)$ is a least-cost path from X_S to X_G in S^{AD}, its cost is no larger than the cost of any other path including the cost of the projected version of $\pi^*_{HD}(X_S, X_G)$. As a result, $c(\pi^*_{AD}(X_S, X_G)) \leq c(\pi^*_{HD}(X_S, X_G))$. $\square$

Theorem 4.2 *If we have a finite state-space, algorithm 1 terminates and upon successful termination, the cost of the returned path $\pi(X_S, X_G)$ is no more than ϵ_{track} times the cost of an optimal path from state X_S to state X_G in G^{HD}.*

Proof The termination of the algorithm is ensured by the fact that after each iteration we are introducing new high-dimensional states to G^{AD}. Since we have a finite state-space, after finitely many iterations, G^{AD} will become identical to G^{HD}, containing only high-dimensional states. G^{AD} will then be searched for a least-cost path in a finite time. If a path is successfully computed by the adaptive planning phase, it will be fully high-dimensional and the tracking phase will be able to track the computed path exactly, causing the algorithm to terminate. If no path is found in G^{AD}, the algorithm again terminates stating that no feasible path exists from start to goal.

The second statement of Theorem 4.2 follows from Theorem 4.1. By Theorem 4.1, the

adaptive planning phase produces an underestimate of the real cost from start to goal.

$$c(\pi^*_{AD}(X_S, X_G)) \leq c(\pi^*_{HD}(X_S, X_G))$$

Upon algorithm termination, the tracking phase succeeds in finding a path of cost no more than ϵ_{track} times the cost of the computed adaptive path. Thus, we have $c(\pi_\tau(X_S, X_G)) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{AD}(X_S, X_G)) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{HD}(X_S, X_G))$. Hence, the cost of the tracked path is no larger than ϵ_{track} times the cost of an optimal path from start to goal in G^{HD}. □

ϵ-suboptimal graph searches such as weighted-A* are often used by researchers (Likhachev and Ferguson, 2008), since they provide the flexibility of quickly finding paths of cost no more than ϵ times the cost of an optimal path. The following result can be proven if we modify algorithm 1 to use such ϵ-suboptimal graph searches:

Theorem 4.3 *If ϵ_{plan}-suboptimal searches are used in lines 6 and 12 of Algorithm 1, the cost of the path returned by our algorithm is no larger than $\epsilon_{plan} \cdot \epsilon_{track} \cdot \pi^*_{HD}(X_S, X_G)$.*

Proof If we use an ϵ-suboptimal search in the adaptive planning phase, we know that that the cost of the produced path $c(\pi_{AD})$ is no larger than $\epsilon \cdot c(\pi^*_{AD})$. Then we have $c(\pi_{AD}) \leq \epsilon \cdot c(\pi^*_{AD}) \leq \epsilon \cdot c(\pi^*_{HD})$. Then we know that the tracking phase produced a path π_τ of cost no larger than $\epsilon_{\text{track}} \cdot c(\pi_{AD})$. Hence, we have $c(\pi_\tau) \leq \epsilon_{\text{track}} \cdot c(\pi_{AD}) \leq \epsilon_{\text{track}} \cdot \epsilon \cdot c(\pi^*_{HD})$.
□

4.7. Algorithm Parameters

The algorithm for Planning with Adaptive Dimensionality has several parameters that can be used to tune its performance depending on the particular domain of application.

The ϵ_{plan} and ϵ_{track} parameters allow the user to specify the desired sub-optimality bound of the produced solutions. It allows for easy trade-off between solution quality and faster planning times.

The parameters controlling the sizes and shapes of the newly introduced high-dimensional regions are highly domain specific. Generally, introducing large regions into the hybrid graph increases its size and may slow down subsequent search iterations. On the other hand, if the introduced regions are too small, the algorithm may need to perform additional iterations to introduce more regions or grow the sizes of existing ones. The sizes of the new high-dimensional regions generally trade-off between time per iteration and the number of iterations.

The parameter w controlling the width of the tunnel constructed around hybrid paths is also very domain specific. Large tunnel width increases the chances of successfully finding a solution through the tunnel at the expense of larger search space and higher planning time to find a path through the tunnel. If the tunnel width is too narrow, then there is a higher chance that the tunnel is disconnected and no path exists from start to goal. This, in turn, will require additional iterations of the algorithm. Generally, the width of the tunnel allows for trade-off between the time each tracking phase takes and the number of iteration performed by the algorithm.

We discuss the specific choice of parameter values for each of the application domains described in the following chapters.

CHAPTER 5 : Hierarchical Planning with Adaptive Dimensionality

5.1. Motivation

So far, we have discussed how to use a single abstraction of a state-space and construct a hybrid graph. However, many high-dimensional planning problems might have multiple abstract representations that may be more or less relevant in different parts of the state-space. For instance, mobile manipulation planning for grasping or putting down an object can often be split into two very different planning problems—navigation planning for moving the base to a suitable location where the goal is within reach of the manipulator, and manipulation planning for computing a manipulator trajectory to the goal location. Each of these sub-problems can have a different abstract representation that considers the relevant dimensions for the task at hand. Thus, a single abstraction might not be suitable for all areas of the state-space. Moreover, finding a single abstraction that performs well over all areas of the state-space might be difficult, or even impossible, for complex high-dimensional planning problems.

In this chapter, we discuss a method for extending the framework for Planning with Adaptive Dimensionality to be able to utilize multiple state-space abstractions and hierarchies of abstractions. We begin with an overview of important known results about abstractions.

5.2. Related Work

5.2.1. State Space Abstractions

The earliest abstractions studied are the so called "embeddings", which rather than grouping states into abstract states, introduce additional edges into the original state space. For example, adding "macro-operators" or relaxing preconditions for operators in the state space generate embeddings. The other common type of abstractions are the "homomorphisms", which group states together into abstract states.

In general, there exist an exponential number of abstractions that can be generated over a given state-space. Li et al. (Li et al., 2006) discuss the structure of the space of all abstractions over a given state-space. This space of abstractions is partially ordered and the partial ordering allows us to say A is “more abstract” or “coarser” than B for some pairs of abstractions A and B. The partial ordering of the space also means that it forms a directed acyclic graph (DAG). At one end of the abstraction space is “the finest” identity abstraction, which maps every state to itself. At the other end is “the coarsest” abstraction (called the null abstraction), which maps all states into a single abstract state. The homomorphic abstractions are equivalent to set partitions and equivalence relations of the state-space.

5.2.2. Valtorta’s Theorem

One goal of using abstractions is to create heuristics to guide and speed up a search algorithm, such as $A^\star$. Without a heuristic, the algorithm will blindly search the large original state-space. Focusing the search with a heuristic will reduce the search effort by a certain amount, called the “saving” in (Holte et al., 1996b). The challenge is to create abstractions for which the additional effort of computing the heuristic using the abstraction does not outweigh the benefits of using it. In other words, the “saving” from utilizing the heuristic is significantly more than the effort required to compute it.

In (Valtorta, 1984), Valtorta presents a cost-benefit analysis of automatically generated heuristics by using embedding transformations. The work proves that if a state space S is embedded into an abstracted state space S' and a heuristic h is computed by blindly searching S', then an $A^\star$ search of S using h will expand every state that is expanded by a blind $A^\star$ search of S. This result, known as Valtorta’s theorem, states that using embedding transformations to compute heuristics could not possibly speed up search, as the search efforts for computing the heuristic combined with the search efforts of using the heuristic must always equal or exceed the search efforts of a blind search in the original state-space. In (Holte et al., 1996b), the authors define “Valtorta’s Barrier” as the number of states expanded when blindly searching in a state-space, and by Valtorta’s theorem, this

barrier cannot be broken using any embedding transformation.

In (Holte et al., 1996b), a generalized version of Valtorta's theorem is presented, which states the following:

Theorem 5.1 ***Valtorta's Theorem – Generalized***
Let E be any state necessarily expanded when the given problem to find $\pi^*_S(Start, Goal)$ is solved by blind search directly in state space S, let ϕ be any abstraction mapping from S to S' and let $h_\phi(E)$ be computed by blindly searching in S' from $\phi(E)$ to $\phi(Goal)$. If the problem is solved in S by an $\mathrm{A}^\star$ search using $h_\phi(-)$, then either:
(1) E itself will be expanded, or
(2) $\phi(E)$ will be expanded.

According to this generalized result, when ϕ is an embedding $\phi(E) = E$ and we observe the result of the original Valtorta's theorem and that speedup using embeddings is not possible. However, this result also shows that if ϕ is a homomorphism, derived by grouping states together into abstract states, speedup becomes possible, as many expansions in the original space S can be replaced by a single expansion of the corresponding abstract state in S'.

Another important result derived in (Valtorta, 1984) and generalized in (Holte et al., 1996b) is the following:

Theorem 5.2 *Let ϕ be any abstraction mapping from S to S' and let $h_\phi(s)$ be computed by blindly searching in S' from $\phi(s)$ to $\phi(Goal)$. Then h_ϕ is a consistent (monotone) heuristic.*

5.3. Combining Multiple Abstractions

Let us assume that we are given a set of low-dimensional sub-spaces $S^{LD}_1, ..., S^{LD}_k$ and respective transition sets for each sub-space $T^{LD}_1, ..., T^{LD}_k$. Let us also assume that we know the respective projection functions $\lambda_1, ..., \lambda_k$ and their inverses, which allow us to project high-dimensional states to each of our low-dimensional sub-spaces and back. Let each of the low-dimensional sub-spaces, their transition sets, and projection functions adhere

to the restriction given in 4.1. Thus, we can define a set of abstractions $\{\mathcal{A}_1, ..., \mathcal{A}_k\}$, $\mathcal{A}_i = (\lambda_i, \lambda_i^{-1}, G_i^{LD} = (S_i^{LD}, T_i^{LD}), c_i)$. Let $\mathcal{H}$ denote the full-dimensional abstraction (i.e. the identity abstraction over S^{HD}). We also assume that the environment is partitioned into a finite number of regions $R = \{\rho_1, ..., \rho_n\}$. Let us assume, for now, that each region is associated with a particular abstraction based on some oracle function that determines which abstraction is "most suitable" for each region.

Thus, every region is associated with an abstraction $\mathcal{A}_i$ and a corresponding sub-space S_i^{LD}, or in the case when the region is high-dimensional—$\mathcal{H}$ and S^{HD}. What we need to do is "stitch" the sub-spaces associated with each region together into a hybrid graph G^{AD}. The following defines how to transition between high-dimensional states and a low-dimensional sub-space S_k^{LD} based on the projection functions λ_k and λ_k^{-1}.

- If X_i is high-dimensional then for all high-dimensional transitions $(X_i, X_j^{HD}) \in T^{HD}$, if $X_j^{HD} \in G^{AD}$ then $(X_i, X_j^{HD}) \in T^{AD}$. If $X_j^{HD} \notin G^{AD}$, then there exists k such that $\lambda_k(X_j^{HD}) \in G^{AD}$ and $(X_i, \lambda_k(X_j^{HD})) \in T^{AD}$. That is, for high-dimensional states we allow only high-dimensional transitions to other high-dimensional states if they fall inside S^{AD}, or their low-dimensional projections to the corresponding sub-space in G^{AD}.

- If X_i is low-dimensional in sub-space S_k^{LD} then for all low-dimensional transitions $(X_i, X_j^{LD}) \in T_k^{LD}$, if $X_j^{LD} \in S^{AD}$ then $(X_i, X_j^{LD}) \in T^{AD}$ and for all high-dimensional transitions $(X, X_j^{HD}) \in T^{HD}$, where $X \in \lambda_k^{-1}(X_i)$, if $X_j^{HD} \in S^{AD}$ then $(X_i, X_j^{HD}) \in T^{AD}$. That is, for low-dimensional states we allow low-dimensional transitions if they lead to another low-dimensional state in S^{AD} within the same sub-space, and high-dimensional transitions from their high-dimensional projections if they lead to a high-dimensional state in S^{AD}.

We also need to define the transitions between two low-dimensional sub-spaces S_i^{LD} and S_j^{LD} in regions $\rho_{i'}$ and $\rho_{j'}$ respectively. One can define specific projection functions between

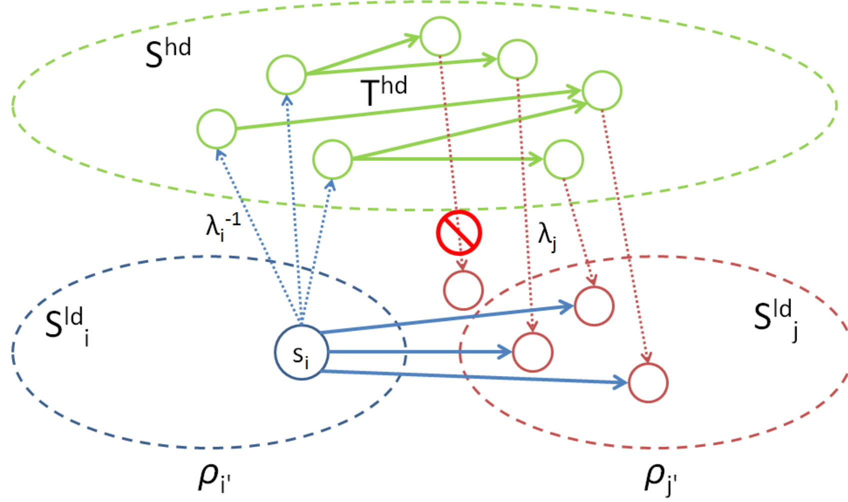


Figure 8: Illustration of the process of computing transitions between different low-dimensional sub-spaces S_i^{LD} and S_j^{LD} via the high-dimensional space S^{HD}. We use the projection functions λ_i^{-1} and λ_j (dotted arrows), and high-dimensional transitions from T^{HD} (solid green arrows). The resulting valid successors (blue arrows) allow us to transition from S_i^{LD} to S_j^{LD}.

each pair of low-dimensional sub-spaces and utilize them in the same fashion as when computing transitions between high- and low-dimensional states. However, that might become cumbersome from implementation standpoint if one wants to consider a large number of sub-spaces. A more computationally expensive, but more general approach is the following.

Let $s_i \in S_i^{LD}$. To compute the valid successor states of s_i that fall in S_j^{LD}, we can compute the high-dimensional projections of s_i, $H = \lambda_i^{-1}(s_i)$. Then, we compute the set of successor states U_{HD} for all states in H using transitions from T^{HD}. Once we have computed the set of high-dimensional successors U_{HD}, we can project them to sub-space S_j^{LD} using λ_j, while also discarding all projections that fall outside region $\rho_{j'}$. Thus, the successors of state $s_i \in S_i^{LD}$ are $Successors(s_i, S_j^{LD}) = \{s_j \in \lambda_j(U_{HD}) | s_j \in \rho_{j'}\}$. This approach effectively introduces a high-dimensional boundary between different low-dimensional sub-spaces, which we transition through in order to get from one sub-space to the other. Figure 8 illustrates the process of computing transitions between low-dimensional sub-spaces.

Algorithm 2 Planning with Adaptive Dimensionality Using Multiple Abstractions

```
1:  G^AD = Initialize-Regions((G_1^LD, ρ_1) ... (G_n^LD, ρ_n))
2:  Add-HD-Region(G^AD, X_S)
3:  Add-HD-Region(G^AD, X_G)
4:  loop
5:                                                      ▷ Adaptive Planning Phase
6:      search G^AD for least-cost path π*_AD(X_S, X_G)
7:      if π*_AD(X_S, X_G) is not found then
8:          return no path from X_S to X_G exists
9:      end if
10:                                                     ▷ Tracking Phase
11:     construct a tunnel τ around π*_AD(X_S, X_G)
12:     search τ for least-cost path π*_τ(X_S, X_G)
13:     if π*_τ(X_S, X_G) is not found then
14:         find state(s) X_r where to introduce next-best abstraction
15:         Introduce-Next-Best-Abstraction(G^AD, X_r)
16:     else if c(π*_τ(X_S, X_G)) > ε_track · c(π*_AD(X_S, X_G)) then
17:         find state(s) X_r where to introduce next-best abstraction
18:         Introduce-Next-Best-Abstraction(G^AD, X_r)
19:     else
20:         return π*_τ(X_S, X_G)
21:     end if
22: end loop
1:  function Introduce-Next-Best-Abstraction(G^AD, X_r)
2:      ρ = Get-Region-For-State(X_r)
3:      α = Get-Abstraction-For-Region(ρ)
4:      β = Get-Next-Abstraction-For-Region(ρ, α)
5:      if ∃β then
6:          Set-Abstraction-For-Region(ρ, β)
7:          Update-Hybrid-Graph-Region(G^AD, ρ, β)
8:      else
9:          Add-or-Grow-HD-Region(X_r)
10:     end if
11: end function
```

Now, let us relax the assumption that we have an oracle function that tells us which "the best" abstraction is for each region. Instead, for each region ρ_i we have a score for each of the abstractions. We will assume that "the best" sub-space to use for the region ρ_i is the abstraction with the highest score. Initial scores can be left to the user to specify, or they can be estimated automatically based on features of the environment, such as distance to obstacles, inclination of the ground plane, or distance to the goal, for example. They can also be computed from example trajectories through the environment. In the context of mobile manipulation, for instance, trajectory segments that move the base could increase

the score for the base abstraction in the region, whereas segments moving the arm could increase the score of the arm abstraction. Trajectory segments with complex movements of many joints can increase the score for the full-dimensional abstraction $\mathcal{H}$. Moreover, after each planning query, information from the search tree constructed during the tracking phase can be used to update the scores for each region, based on the high-dimensional paths available in the search tree. Thus, when initializing our hybrid graph (Alg. 2, line 1), rather than using a single abstract representation for all regions, we "stitch" together the representations with highest scores for each region to form the initial instance of the hybrid graph.

The tracking phase of the algorithm can also be modified to accommodate the fact that multiple low-dimensional sub-spaces are available to the planner. Instead of directly introducing or growing high-dimensional regions into G^{AD}, we can attempt to use the next-best low-dimensional sub-space for the region where tracking failed (Alg. 2, lines 15,18). Once we have tried all promising abstractions for a region, we can revert to introducing high-dimensional regions.

5.4. Theoretical Properties

The proposed extension to the algorithm for planning with adaptive dimensionality does not break any of the theoretical guarantees provided by the original algorithm, provided that all low-dimensional sub-spaces used conform to the assumptions stated above and satisfy constraint 4.1. Thus, the algorithm is complete with respect to the original high-dimensional graph and provides strong theoretical bounds on solution cost sub-optimality.

Theorem 5.3 *The cost of a least-cost path from X_S to X_G, $\pi^*_{AD}(X_S, X_G)$, in G^{AD} is a lower bound on the cost of a least-cost path from X_S to X_G, $\pi^*_{HD}(X_S, X_G)$, in G^{HD}.*

$$c(\pi^*_{AD}(X_S, X_G)) \leq c(\pi^*_{HD}(X_S, X_G))$$

Sketch Consider the projection π'_{AD} of the path $\pi^*_{HD}(X_S, X_G)$ onto the hybrid state-space S^{AD}. In this projection, every state X in $\pi^*_{HD}(X_S, X_G)$ is mapped onto itself, if $X \in S^{AD}$, or onto $\lambda_k(X)$ otherwise, where $\mathcal{A}_k = (\lambda_k, \lambda_k^{-1}, G_k^{LD} = (S_k^{LD}, T_k^{LD}), c_k)$ is the low-dimensional abstraction associated with the region in which X falls into. Then according to equation 4.1, every transition T_i in the projected π'_{AD} will either be bounded from above by the cost of the corresponding transition in $\pi^*_{HD}(X_S, X_G)$ if T_i is a low-dimensional transition, or will be exactly equal to the cost of the corresponding transition if T_i is a high-dimensional transition. Consequently, the cost of the projected version of $\pi^*_{HD}(X_S, X_G)$ will be no larger than $c(\pi^*_{HD}(X_S, X_G))$. Furthermore, since $\pi^*_{AD}(X_S, X_G)$ is a least-cost path from X_S to X_G in S^{AD}, its cost is no larger than the cost of any other path including the cost of the projected version of $\pi^*_{HD}(X_S, X_G)$. As a result, $c(\pi^*_{AD}(X_S, X_G)) \leq c(\pi^*_{HD}(X_S, X_G))$. $\square$

Theorem 5.4 *If S^{HD} is finite and we have a finite number of regions $\{\rho_1, ..., \rho_n\}$ and a finite number of low-dimensional abstractions $\{\mathcal{A}_1, ..., \mathcal{A}_k\}$, algorithm 2 terminates and upon successful termination, the cost of the returned path $\pi(X_S, X_G)$ is no more than ϵ_{track} times the cost of an optimal path from state X_S to state X_G in G^{HD}.*

Proof The termination of the algorithm is ensured by the fact that after each iteration one of the following occurs:

- We are exhausting an assignment of a low-dimensional abstraction to a region (i.e. assigning the next-best abstraction to a region until no more abstraction are available). Since the number of regions and abstractions are finite, this can occur only a finite number of times.

- We are introducing new high-dimensional states to G^{AD} by adding or growing a high-dimensional region. Since we have a finite number of high-dimensional states in S^{HD}, this can also occur only a finite number of times.

Thus, our algorithm will perform a finite number of iterations. In the worst case, after

finitely many iterations, all regions will be assigned the high-dimensional abstraction and all states in S^{HD} will be added to G^{AD} ($G^{AD} = G^{HD}$). Thus, the adaptive planning phase will produce a fully high-dimensional path π_{AD}, which the tracking phase will be able to match exactly ($\pi_\tau = \pi_{AD}$, $c(\pi_\tau) = c(\pi_{AD})$). π_τ will satisfy $c(\pi_\tau) \leq \epsilon_{\text{track}} \cdot c(\pi_{AD})$ for any $\epsilon_{\text{track}} \geq 1$ and the algorithm will terminate.

The second statement of Theorem 5.4 follows from Theorem 5.3. By Theorem 5.3, the adaptive planning phase produces an underestimate of the real cost from start to goal.

$$c(\pi^*_{AD}(X_S, X_G)) \leq c(\pi^*_{HD}(X_S, X_G))$$

Upon algorithm termination, the tracking phase succeeds in finding a path of cost no more than ϵ_{track} times the cost of the computed adaptive path. Thus, we have $c(\pi_\tau(X_S, X_G)) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{AD}(X_S, X_G)) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{HD}(X_S, X_G))$. Hence, the cost of the tracked path is no larger than ϵ_{track} times the cost of an optimal path from start to goal in G^{HD}. $\square$

As we have shown previously, we can allow ϵ-suboptimal graph searches, such as weighted-A*, to be used in the *PAD* framework.

Theorem 5.5 *If ϵ_{plan}-suboptimal searches are used in lines 6 and 12 of Algorithm 2, the cost of the path returned by our algorithm is no larger than $\epsilon_{plan} \cdot \epsilon_{track} \cdot \pi^*_{HD}(X_S, X_G)$.*

Proof If we use an ϵ-suboptimal search in the adaptive planning phase, we know that that the cost of the produced path $c(\pi_{AD})$ is no larger than $\epsilon \cdot c(\pi^*_{AD})$. Then we have $c(\pi_{AD}) \leq \epsilon \cdot c(\pi^*_{AD}) \leq \epsilon \cdot c(\pi^*_{HD})$. Then we know that the tracking phase produced a path π_τ of cost no larger than $\epsilon_{\text{track}} \cdot c(\pi_{AD})$. Hence, we have $c(\pi_\tau) \leq \epsilon_{\text{track}} \cdot c(\pi_{AD}) \leq \epsilon_{\text{track}} \cdot \epsilon \cdot c(\pi^*_{HD})$.
$\square$

5.5. Identifying Useful Abstractions

In all of our applications of Planning with Adaptive Dimensionality we have relied on human intelligence and intuition to provide the abstractions that are being used by the planner.

We acknowledge that in many domains it might be difficult or impractical to rely on human input to define suitable abstractions that work well for the particular domain. One of our research goals was to explore methods for computing useful abstractions automatically. As we mentioned previously, the set of all abstraction over a given state-space has a well-defined structure—a partially ordered set, or a directed acyclic graph. This observation poses an interesting question: can "good" abstractions be generated or learned automatically? In other words, can finding "good" abstractions be formulated as a search problem in itself?

As discussed in (Li et al., 2006), the space of all abstractions is partially-ordered and forms a directed acyclic graph (DAG), which defines a well-structured state-space. In addition, the work by Holte et al. (Holte et al., 1996b) provides a method of evaluating the maximum theoretical speedup that an abstraction can achieve on a given problem when compared to blind search. This can theoretically be used as a measure of how "good" and abstraction is for a given problem (i.e. a cost function). Unfortunately, the space of abstractions over a given state-space is exponentially larger than the state-space itself, so performing graph search on it might be impractical. However, such search need only be performed once during the design phase of the planner development in order to identify the promising abstractions for a particular domain. Moreover, the space of all abstractions can be pruned significantly by only considering abstractions that form regular partitions of the state-space and that can be encoded using projection functions. Recall our definitions of the projection functions $\lambda(\cdot)$ and $\lambda^{-1}(\cdot)$. λ operates by combining states into a single abstract state in a regular fashion (e.g. $\lambda((x, y, \theta)) = (x, y)$ for ground vehicle navigation). Thus, for certain domains, it might be feasible to use such meta-search of the space of abstractions in order to identify promising ones.

Vernaza and Lee (Vernaza and Lee, 2012) explore the problem of learning low-dimensional structures of cost functions in the context of holonomic motion planning in continuous spaces ($\mathbb{R}^N$). They have shown that the minimum-cost path between two points in a Euclidean space is always contained entirely within the smallest affine sub-space containing

both points and all directions in which the cost varies. Thus, if the cost function variations can be captured by a low-dimensional sub-space containing the start and goal locations, then planning in this low-dimensional sub-space is sufficient for finding the optimal path. This result is formalized as follows.

Theorem 5.6 *Consider a holonomic motion planning problem over* $\mathbb{R}^N$ *from* x_a *to* x_b *with a cost function* $C : \mathbb{R}^N \rightarrow \mathbb{R}$*, and suppose that there exists an* $N \times d$ *matrix* W *such that* $d \leq N$ *and*

$$C(y) = C(WW^T y), \forall y \in \mathbb{R}^N$$

Let $I = [0, 1]$ *denote the unit interval. Then there exists an optimal path* $x^* : I \rightarrow \mathbb{R}^N$ *of this planning problem and functions* $a : I \rightarrow \mathbb{R}^d$*,* $s : I \rightarrow \mathbb{R}$*, such that*

$$x^*(t) = Wa(t) + x_a + (x_b - x_a)s(t), \forall t \in I$$

Algorithm 3 Estimating d-dimensional basis of a cost function C over a continuous space $\mathbb{R}^N$

1: **for** $i = 1 \rightarrow$ numberOfGradientSamples **do**
2: $\quad x \leftarrow$ randomConfigurationSample()
3: $\quad G(:, i) \leftarrow \nabla C(x)$
4: **end for**
5: $W \leftarrow$ TopNEigenvectors(d, GG^T)
6: **return** W

Informally, this results shows that optimal paths exist that deviate from the linear interpolation between the start and goal only in the directions upon which the cost depends. They provide a method for computing a d-dimensional basis for compressing the cost function through sampling of the cost gradient (Alg. 3). Thus, they compute the d-dimensional sub-space that best captures the cost variation.

Such cost-space analysis can be directly used in discrete planning problems represented as graphs, provided that the graphs are embedded in a continuous space and the edge costs are computed based on a continuous cost function C. A problem arises when the continuous cost function C is not known and has to be estimated from edge costs. Let's assume that

edge costs c_e are computed by integrating an unknown continuous cost function over the path taken by an edge π_e. Let $e = (x_0, x_1)$ be an edge between two vertices $x_0, x_1 \in \mathbb{R}^N$. Let π_e be the time parametrized path taken by the edge from x_0 to x_1 on the interval $t = [0, 1]$, where $\pi_e(0) = x_0$ and $\pi_e(1) = x_1$. Then, the cost of an edge is given by the path integral

$$c_e = \int_0^1 C(\pi_e(t))|\pi_e'(t)|dt$$

where C is an unknown continuous cost function. In order to apply the cost compression method from (Vernaza and Lee, 2012), we need a way of computing an estimate of ∇C. Expressing c_e as a function of t gives us

$$c_e(t) = \int_0^t C(\pi_e(\tau))|\pi_e'(\tau)|d\tau$$

$$\frac{d}{dt}c_e(t) = \frac{d}{dt}\int_0^t C(\pi_e(\tau))|\pi_e'(\tau)|d\tau = C(\pi_e(t))|\pi_e'(t)|$$

$$C(\pi_e(t)) = \frac{\frac{d}{dt}c_e(t)}{|\pi_e'(t)|}$$

Thus, we can estimate C along any edge by estimating the derivative of the edge cost and the "speed" at which we travel along the edge. Assuming the edge is traversed in unit time, then $|\pi_e'(t)| = ||\pi_e|| \forall t \in [0, 1]$. Moreover, $c_e(t)$ is only specified at $t = 0$ and $t = 1$. Thus, we can only estimate its derivative crudely by

$$\frac{d}{dt}c_e(t) \approx c_e(1) - c_e(0) = c(e) \forall t \in [0, 1]$$

$$C(\pi_e(t)) \approx \frac{c(e)}{||\pi_e||} \forall t \in [0, 1]$$

In other words, we estimate the continuous instantaneous cost along an edge as the cost of the edge over the distance traveled by the edge. This estimate remains constant throughout the edge.

The fact that we can compute estimates of C for each edge, and respectively, for each vertex,

allows us to compute estimates of ∇C by sampling C at several locations. Algorithm 4 shows a proposed algorithm for adapting the continuous cost-compression algorithm (Alg. 3) for graphs, embedded into continuous spaces.

Algorithm 4 Estimating d-dimensional basis of a cost function C from a graph $G = (V, E)$ embedded in a continuous space $\mathbb{R}^N$

1: **for** $i = 1 \rightarrow$ numberOfGradientSamples **do**
2: $x \leftarrow$ randomVertex()
3: Let $\mathcal{S} = \emptyset$
4: **for** $j = 1 \rightarrow$ numberSuccessors(x) **do**
5: Let x_j be the j-th successor of x
6: Let $e = (x, x_j) \in E$ be the edge connecting x and x_j
7: $\hat{C}_j(x_j) \leftarrow \text{cost}(e)/\text{length}(e)$
8: Insert $(x_j, \hat{C}_j(x_j))$ in $\mathcal{S}$
9: **end for**
10: $\nabla\hat{C}(x) = \text{approxGradientFromSamples}(\mathcal{S})$
11: $G(:, i) \leftarrow \nabla\hat{C}(x)$
12: **end for**
13: $W \leftarrow \text{TopNEigenvectors}(d, GG^T)$
14: **return** W

This algorithm can be used on a graph generated for a free space environment as a tool to give information about which dimensions have most significant effect on the cost function. This can provide insight to possible low-dimensional representations that can be implemented for the system, which operate in those dominant dimensions. Additionally, the same analysis can be performed on localized regions of the graph during planning in order to determine which the dominant dimensions are in each region, and thus decide which of the low-dimensional representations available to the planner is most suitable for the region.

As a simple example, we applied the method to a graph generated for $(x, y, \text{heading})$ navigation for a non-holonomic vehicle. The graph was generated by applying the same 6 motion primitives at any vertex $X = (x, y, \theta)$ from its initial heading. The cost of each edge c_e was computed as the distance traveled along the edge, multiplied by a penalty factor (the heading value was not used for cost computations). A factor of 1 was used for the 2 edges going forward with the same heading, a factor of 2 was used for the 2 edges that turned, and a factor of 5 was used for the two edges that went in reverse. As we used $\hat{C}(x) = \text{cost(e)/length(e)}$ to estimate $C(x)$, in this case we had $\hat{C}(x) = \text{cost}(e)/\text{length(e)} =$

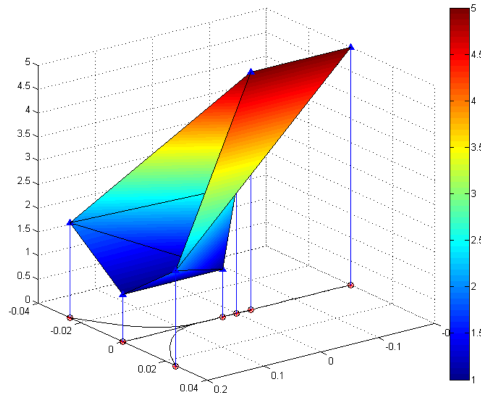


Figure 9: Estimating the continuous scalar field $C(x)$ from a graph. The figure shows the outgoing motion primitives (edges) obeying minimum turning radius constraints in the context of $(x, y, \text{heading})$ navigation for a non-holonomic vehicle. Edge cost was calculated as the distance traveled along the edge multiplied by a penalty factor. A penalty factor of 1 was used for the two edges going straight, a factor of 2 was used for the two turning primitives, and a factor of 5 for the two primitives moving backwards. The estimated continuous scalar field $\hat{C}(x) = \text{cost}(e)/\text{length(e)}$ is shown.

$\text{penalty} \cdot \text{length(e)}/\text{length(e)} = \text{penalty}$. Figure 9 shows the estimated scalar field for $\hat{C}(x)$ for a vertex centered at the origin and a heading of 0. Estimates for C were computed at each of the 6 successor vertices and Delaunay triangulation was used to compute a surface estimate from the samples. The surface gradient was estimated at the locations of all 7 vertices shown and used as samples in the G matrix of algorithm 4. Since the same motion primitives are applied to every vertex depending on its orientation, the estimated scalar field $\hat{C}$ looks identical for all vertices with the same heading (assuming obstacle-free environment), and thus, the gradient $\nabla\hat{C}$ estimates are also identical. The heading was discretized uniformly into 16 values on the interval $[0, 2\pi)$. Thus, we had 16 different scalar field estimates (one for each heading) each with 7 estimates of $\nabla\hat{C}$ (112 samples total). The three eigenvectors of GG^T found were exactly $\langle 1, 0, 0\rangle^T$, $\langle 0, 1, 0\rangle^T$, and $\langle 0, 0, 1\rangle^T$, with corresponding eigenvalues $5.5594 \cdot 10^5$, $5.5594 \cdot 10^5$, and 0. Perhaps unsurprisingly, this tells us that the dominant dimensions of the planning problem are the x and y coordinates, with the heading being secondary. Thus, using a low-dimensional representation that considers only (x, y) would be a promising low-dimensional representation to use in the framework

for planning with adaptive dimensionality. The heading is not completely irrelevant, even though in our example it had no effect on the cost function, since it is necessary to ensure that the minimum turning radius constraints of the vehicle are satisfied. In Chapter 7 we discuss in detail the application of the *PAD* framework to planning for navigation.

It is also important to note that changing the representation encoding the degrees of freedom of the system, such as the one discussed in chapters 9 and 10, might also provide insight into identifying dominant dimensions. In chapters 9 and 10 we describe an alternative representation of a 7-DoF anthropomorphic robotic arm, not in terms of its 7 joint angle values $\langle j_1, ..., j_7 \rangle$, but rather in terms of the 6-DoF Cartesian pose of the end-effector and the arm's free/swivel angle $\langle \text{end-effector pose}_{6D}, \text{swivel} \rangle$ (Fig. 28).

In general, computing useful abstractions of a state-space is a challenging problem. The general approach of performing a meta-search over the space of abstractions discussed above may quickly become impractical for systems with large states-spaces. Thus, one may have to develop approaches exploiting domain-specific information to simplify the problem, such as the cost compression approach by (Vernaza and Lee, 2012).

CHAPTER 6 : Incremental Graph Search for *PAD*

6.1. Motivation

Incremental search is a technique for continual planning that reuses information from previous searches to find solutions to a series of similar search problems potentially faster than it is possible by solving each search problem from scratch. In many situations, a system has to continuously adapt its plan to changes in its environment or in its knowledge of the environment. In such cases, the original plan might no longer be valid, and thus, the system needs to re-plan for the new situation. In these situations, solving the new search problem independently of previous search efforts (planning from scratch) can be very inefficient. This is especially true for situations when the changes of the search problem are small or very localized. For example, a robot might have to re-plan when it detects a previously unknown obstacle, which generally affects the graph structure and edge costs in a very localized fashion. Incremental graph search is certainly useful in the context of Planning with Adaptive Dimensionality, as the algorithm performs multiple search iterations and the changes to the graph structure and edge costs between iterations is very localized—inside the newly inserted high-dimensional regions. The motivation for developing a new incremental graph search algorithm came from our goal to use incremental search in the framework for Planning with Adaptive Dimensionality. However, the popular approaches $D^{\star}$-Lite and Anytime $D^{\star}$ demonstrated prohibitive book-keeping overhead and often exhibited worse performance than starting the planning from scratch each iteration. We wanted our incremental search algorithm to take full advantage of the properties of the hybrid graphs used by Planning with Adaptive Dimensionality. In addition, we wanted the algorithm to efficiently handle localized changes in the graph structure with as little overhead as possible.

In this chapter, we present a simple, but very effective, technique for performing incremental weighted $A^{\star}$ graph search in an anytime fashion, we call Tree-Restoring Weighted $A^{\star}$ ($TRA^{\star}$). The algorithm employs a heuristic to focus the search and allows for trading off

bounded path cost sub-optimality for faster search, just like weighted $A^\star$. In addition, the algorithm re-uses information from previous search queries in order to improve planning times. Moreover, the algorithm can be used for anytime search, similarly to the Anytime Repairing $A^\star$ ($ARA^\star$) algorithm (Likhachev et al., 2003) starting the search with a large heuristic inflation factor ϵ to produce an initial solution faster, and continuously decreasing ϵ to 1 as time permits to find paths of lower sub-optimality bound. On the theoretical side, we show that our anytime incremental algorithm preserves the strong theoretical guarantees provided by the weighted $A^\star$ and $ARA^\star$ algorithms, such as completeness and bounds on solution cost sub-optimality. The algorithm is able to handle a variety of changes to the underlying graph, such as both increasing and decreasing edge costs, and changes in the heuristic. On the experimental side, we demonstrated the effectiveness of our Tree-Restoring Weighted $A^\star$ algorithm in the context of (x,y,z,yaw) navigation planning for an unmanned aerial vehicle (UAV) in unknown and partially-known environments and compared our algorithm to popular incremental and anytime graph search algorithms.

We also applied our incremental search algorithm in the framework for Planning with Adaptive Dimensionality and observed significant performance improvements. We applied the incremental algorithm in the context of 3-DoF (x,y,heading) path planning for Willow Garage's PR2 robot, performing full-body collision checking. Our results suggest that using $TRA^\star$ rather than performing planning from scratch at each iteration improves planning times by up to a factor of 5 in the context of Planning with Adaptive Dimensionality. Moreover, we observed that the Tree-Restoring Weighted $A^\star$ algorithm tends to work better in the context of Planning with Adaptive Dimensionality than alternative incremental graph search techniques, such as $D^\star$. The experimental setup and results for this experimental evaluation are discussed in Chapter 7.

In this chapter we describe the Tree-Restoring Weighted $A^\star$ algorithm and its theoretical properties as a stand-alone general anytime incremental graph search algorithm.

6.2. Definitions and Notations

This chapter focuses heavily on the weighted $A^\star$ search algorithm, so here we provide some useful definitions relating to the algorithm.

Definition 6.1 *Each state s in a graph G has an associated value called g-value during a weighted* A* *graph search. The g-value of a state s, denoted $g(s)$, represents the currently best known cost for reaching S from the start state.*

Definition 6.2 *Each state s in a graph G has an associated value called f-value during a weighted* A* *graph search. The f-value of a state s, denoted $f(s)$, is computed as*

$$f(s) = g(s) + \epsilon \cdot h(s)$$

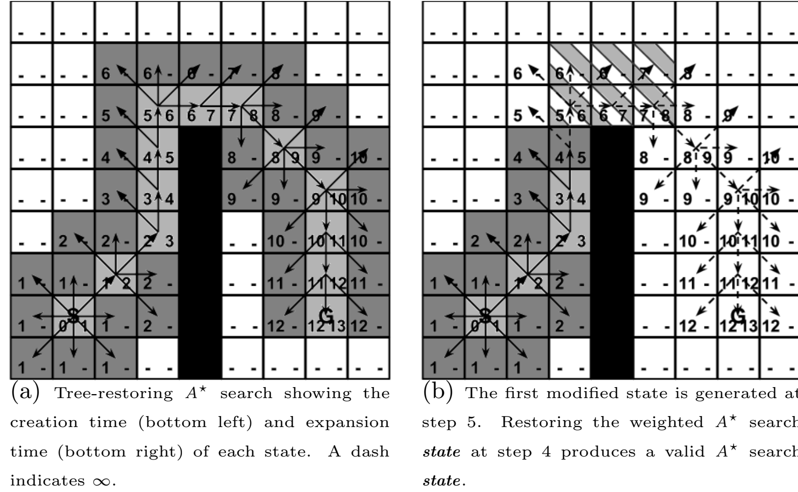
where $g(s)$ is the state's g-value, $h(s)$ is the state's heuristic function value, and ϵ is the heuristic inflation factor used in the weighted A* *graph search. The f-value represents an estimate of the expected cost of a path from start to goal passing through s. The f-values are used as keys for the priority queue of the algorithm.*

Notation 6.1 *We use $OPEN$ to denote the set containing all states in the priority queue of a weighted* A* *search. We use $CLOSED$ to denote the set containing all expanded states that are not in $OPEN$. We use $UNSEEN$ is the set of states that have not been encountered by the search (all states that are not in $OPEN$ and not in $CLOSED$).*

6.3. Tree-Restoring Weighted A* Search

6.3.1. Algorithm

The ***state*** of a weighted $A^\star$ search can be defined by the $OPEN$ list, the $CLOSED$ list, the g-values of all states, and the back-pointer tree. Note the distinction between a ***state*** of a search and a state in the graph being searched; we will use "***state***" when referring to a state of a search. The idea of our approach to incremental weighted $A^\star$ planning is to



(a) Tree-restoring $A^\star$ search showing the creation time (bottom left) and expansion time (bottom right) of each state. A dash indicates ∞.

(b) The first modified state is generated at step 5. Restoring the weighted $A^\star$ search ***state*** at step 4 produces a valid $A^\star$ search ***state***.

Figure 10: Simple 8-connected grid tree-restoring weighted $A^\star$ example (assuming a perfect heuristic for simplicity). Light gray: $CLOSED$ list (expanded states), dark gray: $OPEN$ list, striped: modified states, black: obstacles/invalid states, solid arrows: valid back-pointer tree, dashed arrows: invalid back-pointer tree.

keep track of the ***state*** of the search, so that when the graph structure is modified, we can restore a valid previous search ***state*** and resume searching from there.

We call a ***state*** of a weighted $A^\star$ search valid with respect to a set of modified states, if the $OPEN$ and $CLOSED$ lists, and the back-pointer tree do not contain any of the modified states and the g-values of all states are correct with respect to the back-pointer tree.

At any one time during a weighted $A^\star$ search, each state falls in exactly one of the following categories:

- *unseen* - the state has not yet been encountered during the search; its g-value is infinite; the state is not in the back-pointer tree, not in $OPEN$, and not in $CLOSED$.

- *inOPEN* - the state is currently in the $OPEN$ list; the state has been encountered (generated), but has not yet been expanded; its g-value is finite (assuming that when states with infinite g-values are encountered, they are not put in the $OPEN$ list); the state is in the back-pointer tree.

- *inCLOSED* - the state is currently in the *CLOSED* list; the state has been generated and expanded; its g-value is finite; the state is in the back-pointer tree.

We assume that the weighted $A^\star$ search expands each state at most once, which preserves the sub-optimality guarantees of the algorithm as proven in (Likhachev et al., 2003) when using a consistent heuristic. The Tree-Restoring Weighted $A^\star$ algorithm ($TRA^\star$) keeps a discrete time variable *step* that is initialized at 1 and incremented by 1 after every state expansion. Thus, if we record the step $C(X)$ in which a state X is generated (first placed in the $OPEN$ list, $C(X) = \infty$ if state has not yet been generated) and the step $E(X)$ in which a state is expanded (placed in the $CLOSED$ list, $E(X) = \infty$ if the state has not yet been expanded), we can reconstruct the $OPEN$ and $CLOSED$ lists at the end of any step s (Fig. 10).

$$CLOSED_s = \{X|E(X) \leq s\}$$

$$OPEN_s = \{X|C(X) \leq s \text{ and } E(X) > s\}$$

Note that $C(X) < E(X) \forall X$ (i.e. a state's creation time is before the state's expansion time), and if $E(X) = E(X')$ then $X \equiv X'$ (i.e. no two states could have been expanded during the same step).

In order to be able to reconstruct the back-pointer tree and g-values for all states at the end of a previous step s, each state must store a history of its parents and g-values. Every time a better g-value g and parent X_p are found for a state X (when X_p is being expanded), a pair (X_p, g) is stored for the state X. Note that the pair stores the g-value of the state X itself, not the g-value of its parent X_p. Thus, we can compute the parent $P_s(X)$ and g-value $g_s(X)$ of a state X at the end of a previous step s by going through X's list L_X of stored (parent, g-value) pairs.

$$(P_s(X), g_s(X)) =$$

$$(X_p, g)_{\in L_X} | \forall (X', g')_{\in L_X} : E(X') \leq E(X_p) \leq s$$

In other words, the valid (parent, g-value) pair of X at step s is the pair containing the parent that was expanded last (most recently), but before or during step s. Storing the history in a list or array and searching it backwards seems to be very effective in quickly identifying the most recent valid parent and g-value.

When a set of states M get modified between search episodes by changes in the costs of some of their transitions, we identify the earliest step c_{min} in which a modified state was created: $c_{min} = min(C(X)|X \in M)$. If we then restore the search ***state*** at the end of step $c_{min} - 1$, we will end up with a valid search ***state*** with respect to the modified states, and thus, we can resume searching from there, provided the heuristic has not changed or does not need to be recomputed.

An important consideration is allowing the algorithm to handle decreasing edge costs, which in turn, require the heuristic to be recomputed so that it remains admissible. In such cases, we might have to restore the search ***state*** to an even earlier step than $c_{min} - 1$ in order to ensure that correct expansion order is maintained with respect to the new heuristic values. We maintain correct expansion order by identifying all possible states that might have been expanded out-of-order relative to the current search ***state*** and the new heuristic values. An expanded state X might have been expanded out-of-order relative to the current best candidate for expansion X' from $OPEN$, if X's f-value at the time of its expansion was lower than the current f-value of X', and also, at the step when X was selected for expansion X' had been created and was in $OPEN$ (i.e. $C(X') < E(X)$). In other words, at time $E(X) - 1$ both X and X' were in $OPEN$ and X' had potentially better f-value than X, and therefore X might have been expanded incorrectly before X'. If we don't find any such states, then the current search ***state*** is valid with respect to the new heuristic and does

not violate the proper expansion order. On the other hand, if we find a set of states I, that were potentially expanded out-of-order, we identify the state $X_f = \arg\min_{X \in I}(E(X))$ with the earliest expansion time and restore the search ***state*** at step $E(X_f) - 1$, right before the potentially incorrectly expanded state X_f was selected for expansion. We repeat this process of restoring previous search ***states*** until the current search ***state*** does not have any states that might have been expanded out-of-order.

We note that the $TRA^\star$ algorithm can be extended to allow for re-expansion of states by keeping multiple records of C and E values for each state for every time a state is placed on $OPEN$ and every time a state is expanded, respectively. However, such an extension will additionally increase the memory overhead of the algorithm. If re-expansions are allowed, however, maintaining correct expansion order is no longer necessary, as re-expansions of states will correctly propagate any inconsistencies in the search tree within the current search iteration.

Algorithms 5 and 6 give the pseudo code for all the important functions in the $TRA^\star$ algorithm.

6.3.2. Theoretical Properties

In this section we provide sketches of proofs for each of the theorems stated. Complete rigorous proofs can be found in Appendix B.

Theorem 6.1 *All states X with $C(X) > c$ will become unseen after $restoreSearch(c)$ is called.*

Proof Follows trivially from definition. □

Theorem 6.2 *The contents of the $OPEN$ and $CLOSED$ lists after $restoreSearch(c)$ is called are identical to what they were at the end of step c of the algorithm.*

Proof Let $OPEN_c$ and $CLOSED_c$ be the $OPEN$ and $CLOSED$ lists at the end of step c of the algorithm. Let $OPEN'$ and $CLOSED'$ be the $OPEN$ and $CLOSED$

lists after the function $restoreSearch(c)$ is called. In can be easily shown that $X \in OPEN_c$ iff $X \in OPEN'$ and $X \in CLOSED_c$ iff $X \in CLOSED'$. Thus, $OPEN_c \equiv OPEN'$ and $CLOSED_c \equiv CLOSED'$. □

Theorem 6.3 *All states X with $C(X) \leq c$ will have correct parent pointers and corresponding g-values after $restoreSearch(c)$ is called.*

Proof We construct a proof by contradiction. Suppose a state X has an incorrect parent pointer, i.e there exists a state $P' \in CLOSED$ such that $g(P') + cost(P', X) < g(P) + cost(P, X)$ (a better parent P' for X exists in the $CLOSED$ list). We argue that P' must have been expanded before P, and since P' provides better g-value than P, then P cannot have been recorded as a parent for X—contradiction. □

Theorem 6.4 *Let M be the set of all modified states after a successful incremental* A* *search episode. Let $c_{min} = min(C(X)|X \in M)$. $restoreSearch(c)$ for any $c < c_{min}$ results in a search **state** that is valid with respect to the modified states M.*

Proof The result follows directly from the above theorems. □

If edge costs cannot decrease, the heuristic remains admissible between search episodes and does not need to be re-computed. However, the heuristic does need to be re-computed when edge costs decrease, in order to ensure that the current search is performed with admissible heuristic values. Changes in the heuristic values, however, affect the ordering of states in the $OPEN$ list and the order of state expansions during the search. As we only allow states to be expanded once, it is necessary to maintain correct expansion order.

Thus, although by Theorem 6.4 $restoreSearch(c_{min} - 1)$ produces a search ***state*** that is valid with respect to the modified states, that search ***state*** is not necessarily valid with respect to the new heuristic values, as the order of expansions might be no longer correct. $heuristicChanged()$ is the function that maintains the correct expansion order when the heuristic changes. As described above, the idea of this function is to keep restoring the

search to earlier search ***state*** until there are no states that could have been expanded in incorrect order. In the worst case, the change in the heuristic is such that expansion order changes from the very beginning, in which case *heuristicChanged*() will restore the search ***state*** to the end of step 0–right after the start state was expanded, which would be equivalent to starting the search from scratch.

It is important to note that, in the context of Planning with Adaptive Dimensionality, introducing new high-dimensional regions can only increase edge costs, and thus, the original heuristic remains admissible for the new instance of the hybrid graph. Consequently, the Tree-Restoring Weighted $A^\star$ algorithm needs to perform only a single restoring step in order to produce a valid search ***state***. Thus, the $TRA^\star$ algorithm is highly efficient when applied in the framework for Planning with Adaptive Dimensionality.

Theorem 6.5 *The function heuristicChanged() terminates and at the time of its termination the search is restored to a search* **state** *that is valid with respect to the new heuristic values. That is, no state has been expanded out-of-order with respect to the new f-values.*

Proof Let X_0 be the state with lowest f-value in $OPEN$ in the current search ***state***. X_0 was first put in $OPEN$ at step $C(X_0)$.

Consider the set I computed in *heuristicChanged*(). As in (Likhachev et al., 2003), $v(X)$ stores the value of $g(X)$ at the time X was expanded. Therefore $v(X) + \epsilon \cdot h(X)$ represents the f-value of X at the time of its expansion $E(X)$, but also accounting for the new heuristic values. $I = \{X_i \in CLOSED | v(X_i) + \epsilon \cdot h(X_i) > f(X_0) \wedge C(X_0) < E(X_i)\}$. In other words, I contains all expanded states that had higher f-values at the time of their expansion than the current candidate for expansion X_0 and that were expanded while X_0 was in $OPEN$. As such, I contains all possible states that might have been expanded incorrectly before X_0 according to the new f-values. Note that it is possible that the current $f(X_0)$ is lower than the value of $f(X_0)$ at step $E(X_i)$, as $g(X_0)$ might have decreased as the search progressed after step $E(X_i)$. Therefore, it is possible that $f(X_i) \leq f(X_0)$ was true at step $E(X_i)$

and that $f(X_i)$ was correctly selected for expansion before X_0. Thus, states in I are not necessarily expanded incorrectly, but they are the only possible states that might have been expanded incorrectly. Let $s' = \min(E(X')|X' \in I) - 1$ as computed in $heuristicChanged()$. Restoring the search ***state*** to step s' ensures that no states have been expanded incorrectly before X_0. At the end of the while loop $I = \emptyset$, thus no states in $CLOSED$ could have been expanded incorrectly with respect to the current expansion candidate X_0.

To prove that $heuristicChanged()$ terminates, we argue that the integer s' strictly decreases through the execution of the while loop. If s' becomes 0, then $CLOSED = \emptyset$ making $I = \emptyset$. □

By Theorem 6.5, $TRA^\star$ algorithm maintains the same expansion order (up to tie-breaking) as non-incremental weighted $A^\star$ and thus, both algorithms have the same theoretical guarantees for completeness, termination, and upper bounds on path cost sub-optimality, assuming that an admissible heuristic is used.

Theorem 6.6 TRA* *expands each state at most once per search query and never expands more states than Weighted* A* *from scratch (up to tie-breaking).*

Proof It is easy to verify that each state can be expanded at most once per search query, as once a state has been expanded and put in $CLOSED$ it can never be placed in $OPEN$. The fact that $TRA^\star$ does not expand more states than performing Weighted $A^\star$ from scratch follows almost trivially from the fact that the two algorithms produce the same order of state expansions (up to tie-breaking), but $TRA^\star$ is able to resume searching from a step $s \geq 0$, thus not performing the first s expansions that Weighted $A^\star$ from scratch would have to perform. □

6.4. Anytime Tree-Restoring Weighted A* Search

In many situations, producing a lower-quality initial solution very quickly, and then improving the solution as time permits, is a desirable property of a planning algorithm.

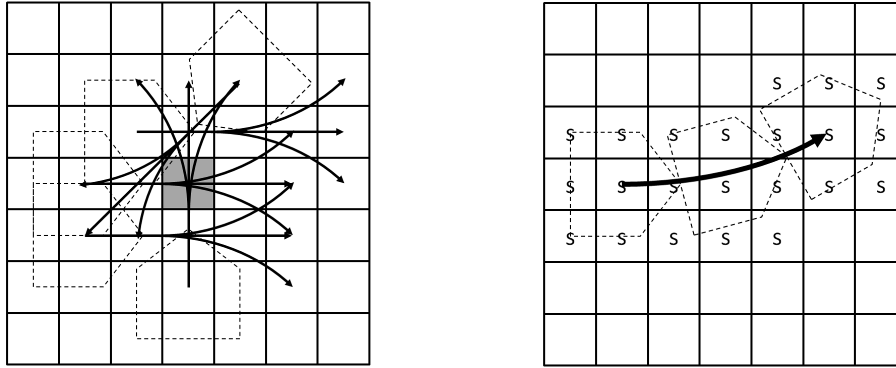
By following the concept of the $ARA^{\star}$ search algorithm, we can extend the $TRA^{\star}$ algorithm to perform in an anytime fashion. $ARA^{\star}$ runs a series of searches with decreasing heuristic weighting factor ϵ until the allocated time runs out or an optimal solution is found for $\epsilon = 1$. It keeps track of an INCONSISTENT list of all states that have been expanded already during the current search iteration (in CLOSED), yet a better parent and lower g-value for them was found after their expansion. The states in INCONS. are moved to OPEN at the beginning of every search iteration, OPEN is re-ordered based on the new ϵ value, and the search proceeds.

To make $TRA^{\star}$ an anytime algorithm similar to $ARA^{\star}$, we need to be able to reconstruct the INCONS. list at a particular time step. Thus, we have to record the step at which a state X is inserted into INCONS., $I(X)$. Also, since $ARA^{\star}$ allows re-expansions of states between search iterations (the ones from INCONS. list), we also need to maintain separate creation $C_{\epsilon}(X)$, expansion $E_{\epsilon}(X)$, and inconsistent $I_{\epsilon}(X)$ records for each ϵ value for which a search episode is performed. Thus, the memory overhead introduced by the algorithm for each state increases proportionally to the number of times it is expanded.

We can reconstruct INCONS. at a desired step s by noting that a state X is in INCONS. from the step $I_{\epsilon_1}(X)$ when the state was inserted into INCONS. for a particular ϵ_1, until it was inserted in OPEN at the beginning of the next planning iteration (for ϵ_2). Thus, $X \in$ INCONS. iff $I_{\epsilon_1}(X) \leq s < C_{\epsilon_2}(X)$.

Then, given a desired target restore step s, we can reconstruct the contents of OPEN, CLOSED, and INCONS. lists, the back-pointer tree, g-values, and the ϵ_s value of the search ***state*** at step s. For every state we drop the creation $C_{\epsilon}(X)$, expansion $E_{\epsilon}(X)$, and inconsistent $I_{\epsilon}(X)$ records for $\epsilon < \epsilon_s$, only maintaining the records up to the current heuristic inflation value ϵ_s.

The proposed Anytime Tree-Restoring Weighted $A^{\star}$ ($ATRA^{\star}$) search algorithm preserves the theoretical properties of the $ARA^{\star}$ algorithm, such as completeness with respect to the



(a) Computing all graph edges affected by a change in a map cell. The figure shows a sub-set of the edges (arrows) affected by a change in the shaded cell. Dashed polygons represent the robot's perimeter.

(b) $ATRA^{\star}$ algorithm storing the expansion step s for which each cell is encountered first, done during the expansion and collision checking of each edge (arrow).

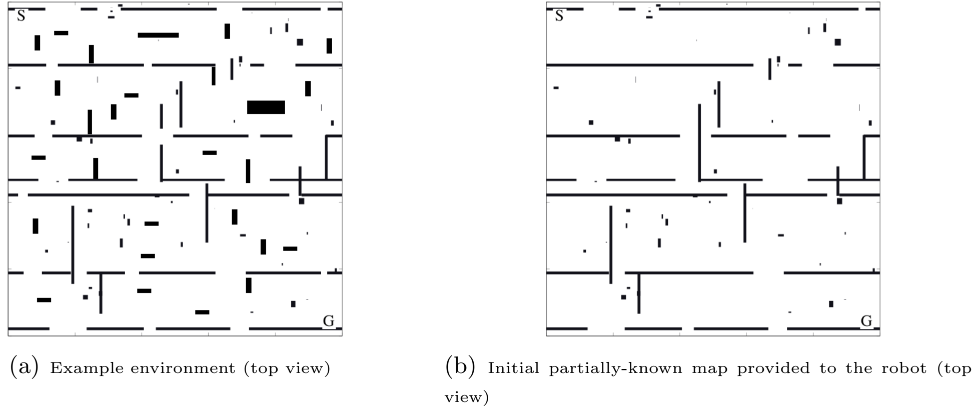
Figure 11: Computing affected graph edges from changed map cells.

graph encoding the problem and bounds on solution cost sub-optimality.

6.5. Efficiently Detecting Changes in the Graph

In the context of navigation planning, lattice-based graphs are often used to encode the search problem by discretizing the configuration space of the robot, and using pre-computed kinodynamically feasible transitions between states (Likhachev and Ferguson, 2008). On the other hand, the map data and obstacle information is usually stored on a grid. Thus, most incremental search algorithms, such as $D^{\star}$, $D^{\star}$-Lite, and Anytime $D^{\star}$, need to be able to translate changes in the map grid to the actual graph edges that are affected by the changes. In other words, the algorithm needs to consider all edges in the graph that cause the robot's perimeter to pass through the changed cell (Fig. 6.11(a)). This procedure can be prohibitively expensive for graphs with high edge density and for large robot perimeters, often significantly diminishing or completely eliminating the benefit of using incremental graph search.

Our approach, however, does not rely on knowing all affected edges, but rather just the expansion step at which the first affected edge was encountered during the search. Thus, for each cell on the map grid, we can record the earliest expansion step for which the



(a) Example environment (top view)

(b) Initial partially-known map provided to the robot (top view)

Figure 12: Example environment and corresponding initial map. The start and goal locations are marked by S and G, respectively.

search encountered an edge that passes through this cell (Fig. 6.11(b)). This introduces a small memory overhead to the size of the map grid data stored (additional integer per cell). However, the performance overhead is negligible, as the collision-checking procedure already enumerates all map cells that an edge passes through to make sure they are obstacle-free. With this extension, when a map cell changes, we can very quickly look up the earliest expansion step for which this cell affected an edge in the graph. Taking the minimum expansion step s across all changed cells in the map and restoring the search ***state*** to step $s-1$ produces a valid search tree with respect to the modified map cells, and thus, their respective modified graph edges.

As shown in our experiments, this approach significantly reduces the time needed for $TRA^\star$ and $ATRA^\star$ to compute the changes to the graph, and subsequently, the overhead of performing incremental search.

6.6. Experimental Evaluation

To validate the $ATRA^\star$ algorithm we implemented it for 4-DoF (x,y,z,yaw) path planning for an unmanned aerial vehicle. The graph representing the problem was constructed as a lattice-based graph, similar to the approach taken in (Likhachev and Ferguson, 2008), except we used constant resolution for all lattices. In lattice-based planning, each state

consists of a vertex encoding a state vector and edges corresponding to feasible transitions to other states. The set of edges incident to a state are computed based on a set of pre-computed motion primitives, which are executable by the robot. The state-space was obtained by uniformly discretizing the environment into $5cm \times 5cm \times 5cm$ cells and the heading values were uniformly discretized into 16 on the interval $[0, 2\pi)$. The robot was tasked to navigate to a fixed goal location. Search was performed backwards from the goal state and the start state changed as the vehicle navigated through the environment. Whenever a path to the goal was computed, the robot advanced by one edge along the path to a new start state; sensed any previously unknown obstacles or gaps through obstacles in its vicinity and updated its environment map; then re-planned for a new path to the goal accounting for the changes in the environment. The appearing and disappearing of obstacles in the map caused, respectively, increasing and decreasing of edge costs in the graph. This, in turn, required a set of modified states to be computed and the heuristic to be re-computed. Moreover, it was necessary re-compute the heuristic at the beginning of every re-planning iteration, as the robot moved through the environment and the start state changed. The heuristic was computed using 3D BFS search from the (x,y,z) position of the start state on an 26-connected 3D grid accounting for obstacles. The heuristic was not perfect as did not account for the orientation of the robot or its perimeter shape. Thus, some scenarios exhibited pronounced heuristic local minima. We ran the planner on 50 maps of size $25m \times 25m \times 2m$ ($500 \times 500 \times 40$ cells) (example shown in Fig. 12). For each of the environments, the planner was run on both an unknown initial map, and a partially-known initial map. An example of a partially-known initial map is shown in Fig. 6.12(b). The maps were generated semi-randomly to resemble floor plans. The partially-known initial maps were generated by randomly adding and removing obstacles from the true map. The start and goal states for each environment were in diagonally opposite corners of the map. We used a set of pre-computed transitions obeying minimal turning radius constraints. The vehicle was also allowed to turn in-place, but the cost of such transitions was penalized by a factor of 5. The non-holonomic transitions and the high penalty factor for turning in-place

Algorithm	Avg. Sub-optimality Bound Achieved	Compute Changes Avg. Time (s)	Repair/Restore Avg. Time (s)	% Iters finished within 1s	# Expansions per Re-plan avg	std dev	Avg. Path Cost Ratio
ATRA*	2.2720	0.0000	0.1615	95.95%	52029	29826	1.0 (baseline)
Anytime $D^\star$	2.2324	0.6214	0.3240	91.01%	50052	47874	0.98
ARA*	2.4211	0 (n/a)	0 (n/a)	93.70%	77377	16786	1.21
Anytime Truncated $D^\star$	2.1124	0.6271	0.3952	91.67%	48853	46904	0.95
Beam-Stack Search	(n/a)	0 (n/a)	0 (n/a)	98.07%	65149	21479	1.32
ATRA*	1.8185	0.0000	0.1501	99.63%	47230	20194	1.0 (baseline)
Anytime $D^\star$	1.8176	0.4874	0.4067	96.40%	53976	39737	1.03
ARA*	2.1600	0 (n/a)	0 (n/a)	99.20%	82041	17681	1.17
Anytime Truncated $D^\star$	1.7802	0.4753	0.4247	97.73%	50974	38225	0.98
Beam-Stack Search	(n/a)	0 (n/a)	0 (n/a)	99.54%	69203	22405	1.24

Table 1: Simulation results on a set of 50 unknown maps (top) and 50 partially-known maps (bottom) for 4-DoF (x,y,z,yaw) path planning for an unmanned aerial vehicle performing anytime planning with time limit of 1 second.

made the path planning problem very challenging. For sensing obstacles, we simulated a forward-facing tilting laser range finder with 180° horizontal and 90° vertical field of view, and a maximum sensing range of $2.0m$.

We ran our planner in anytime mode with an initial sub-optimality bound of $\epsilon = 5.0$ with 1 second allowed for planning. In cases when no plan was found within the time limit, the planner was allowed to continue planning for an additional 1 second for up to 10 times until a solution is found. We also ran our planner in fixed-ϵ mode, planning until the first solution satisfying the specified sub-optimality bound is found.

6.7. Analysis of Results

We compared the $ATRA^\star$ algorithm to other incremental and anytime graph search algorithms—Anytime $D^\star$ (Koenig and Likhachev, 2002a), $ARA^\star$ (Likhachev et al., 2003), Anytime Truncated $D^\star$ (Aine and Likhachev, 2013), and Beam-Stack Search (Zhou and Hansen, 2005a). The non-incremental algorithms $ARA^\star$ and Beam-Stack Search performed planning from scratch at each iteration. In order to replicate the planning conditions across all planners for fair performance comparison, the vehicle followed a predefined path through the environment regardless of the paths produced by the planners. Thus, each of the planners performed the same number of re-planning iterations with identical map information. All planners used the same heuristic, recomputed for every re-planning iteration. The time reported as "Compute Changes" is the time each incremental algorithm required to trans-

Algorithm	Sub-optimality Bound	Compute Changes Avg. Time (s)	Repair/Restore Avg. Time (s)	Re-planning Time (s) avg	std dev	# Expansions per Re-plan avg	std dev
$ATRA^\star$	5.0	0.0000	0.0327	0.2105	0.4643	11065	21499
Anytime $D^\star$	5.0	0.4063	0.0194	0.5973	3.7712	12799	44616
$ARA^\star$	5.0	0 (n/a)	0 (n/a)	0.2770	0.3163	22666	21602
Anytime Truncated $D^\star$	5.0	0.4177	0.0231	0.5031	1.6433	11533	36551
$ATRA^\star$	2.0	0.0000	0.0895	0.3583	0.6435	19994	30477
Anytime $D^\star$	2.0	0.6111	0.2109	0.3397	0.8574	21580	54726
$ARA^\star$	2.0	0 (n/a)	0 (n/a)	0.5004	0.4017	44352	26316
Anytime Truncated $D^\star$	2.0	0.6093	0.2242	0.3088	0.6881	18306	28487
$ATRA^\star$	1.25	0.0000	0.1593	1.6718	5.9480	70116	225425
Anytime $D^\star$	1.25	1.5722	1.5150	4.0458	11.565	213777	592017
$ARA^\star$	1.25	0 (n/a)	0 (n/a)	5.6696	16.038	384546	904201
Anytime Truncated $D^\star$	1.25	1.5983	1.5311	2.3184	8.1722	107634	472733
$ATRA^\star$	5.0	0.0000	0.0258	0.0322	0.1260	2338	8248
Anytime $D^\star$	5.0	0.1986	0.0207	0.1326	0.6359	5427	25114
$ARA^\star$	5.0	0 (n/a)	0 (n/a)	0.3118	0.2125	30705	19612
Anytime Truncated $D^\star$	5.0	0.2043	0.0313	0.1196	0.5363	5214	22756
$ATRA^\star$	2.0	0.0000	0.0698	0.2635	1.0427	14178	50706
Anytime $D^\star$	2.0	0.3367	0.1338	0.2531	1.2339	16042	85201
$ARA^\star$	2.0	0 (n/a)	0 (n/a)	0.7860	0.8860	70229	64742
Anytime Truncated $D^\star$	2.0	0.3274	0.1459	0.2364	1.1491	14954	76638
$ATRA^\star$	1.25	0.0000	0.4295	1.5719	10.448	66014	395437
Anytime $D^\star$	1.25	0.6864	0.9521	1.9882	8.1290	120754	467172
$ARA^\star$	1.25	0 (n/a)	0 (n/a)	4.1886	9.5945	271330	548481
Anytime Truncated $D^\star$	1.25	0.6593	0.9361	1.7318	6.9560	97811	422109

Table 2: Simulation results on a set of 50 unknown maps (top) and 50 partially-known maps (bottom) for 4-DoF (x,y,z,yaw) path planning for an unmanned aerial vehicle performing fixed-ϵ planning until first solution for various sub-optimality bounds.

late changes in the map grid into relevant changes to the graph (computing modified edges for Anytime $D^\star$ and Anytime Truncated $D^\star$, and computing the target restore step for $ATRA^\star$). The time reported as "Repair/Restore" is the time each incremental algorithm took to update its search ***state*** with the new edge costs and heuristic values, so that a new search iteration can be started.

The results we observed for anytime planning for each of the planners for unknown and partially-known maps are summarized in Table 1. As seen from the results, $ATRA^\star$ is able to detect graph changes and restore the search tree significantly faster than Anytime $D^\star$ and Anytime Truncated $D^\star$, while achieving nearly identical sub-optimality bounds and path costs, on average, on both unknown and partially-known maps. Beam Stack Search was able to meet the 1-second deadline in the highest number of iterations at the expense of about 20-30% higher solution cost and no theoretical sub-optimality bound guarantees.

The results we observed for planning until the first solution satisfying a fixed sub-optimality

bound for each of the planners for unknown and partially-known maps are summarized in Table 2. Beam Stack Search was not included in these experiments as it does not provide theoretical sub-optimality bounds on intermediate solutions. The results illustrate the benefit of using incremental graph search as the desired sub-optimality bound decreases. Overall, $ATRA^\star$ performed significantly better than the rest of the planners by both reducing the overhead of performing incremental search and reducing the number of expansions (and thus re-planning time) required for each iteration.

We observed that $ATRA^\star$ is able to outperform planning from scratch most significantly on the difficult planning scenarios (ones exhibiting heuristic local minima, or ones with low sub-optimality bound), as it is able to avoid re-expanding a large number of states between iterations in such cases. The most significant performance gain of $ATRA^\star$ over the two $D^\star$-based algorithms, apart from the reduced overhead in computing changes in the graph, were in scenarios when increasing edge costs cause a large number of expansions of under-consistent states (significantly more expensive than regular over-consistent expansions) in the $D^\star$-based algorithms. On the other hand, our approach suffers most in situations where the search ***state*** needs to be restored to a very early step , in which cases the overhead of performing repeated tree restoring eliminates the benefits of avoiding relatively few re-expansions.

Even though $TRA^\star$ was designed with Planning with Adaptive Dimensionality in mind, we found that $TRA^\star$ and anytime $TRA^\star$ as stand-alone planning algorithms are able to outperform popular alternative incremental and anytime approaches by efficiently avoiding redundant computation and significantly reducing the overhead of performing incremental search. Moreover, $TRA^\star$ and $ATRA^\star$ are general incremental graph search algorithms that can handle arbitrary graphs and arbitrary edge cost changes.

Algorithm 5 Tree-Restoring Weighted $A^\star$

```
CLOSED : Set
OPEN : MinHeap
CREATED : Array
step : Integer
function initializeSearch(X_S X_G)
    CLOSED ← ∅
    OPEN ← {X_S}
    g(X_S) ← 0
    f(X_S) ← g(X_S) + ε · h(X_S)
    step ← 1
    C(X_S) ← 0
    insert(CREATED, X_S)
    E(X_S) ← ∞
end function
function resumeSearch( )
    if needed to recompute heuristic then
        recompute admissible heuristic
        heuristicChanged()
    end if
    while OPEN ≠ ∅ do
        X ← extractMin(OPEN)
        if f(X_G) > f(X) then
            return reconstructPath()
        end if
        Expand(X)
    end while
    return no path exists
end function
function heuristicChanged
    update f-values for created states and re-order OPEN
    while not done do
        Let X_0 be the state with lowest f-value in OPEN
        I ← {X ∈ CLOSED | v(X) + ε · h(X) > f(X_0) ∧ C(X_0) < E(X)}
        if I = ∅ then
            done
        else
            s' ← min(E(X') | X' ∈ I) − 1
            restoreSearch(s')
        end if
    end while
end function
function updateParents(X, s)
    latestG ← 0
    latestParent ← ∅
    latestParentStep ← 0
    for all (X_p, g_p) in stored parent/g-value pairs of X do
        if E(X_p) ≤ s then                          ▷ X_p is a valid parent for step s
            if E(X_p) > latestParentStep then
                                                    ▷ Found more recent parent
                latestParentStep ← E(X_p)
                latestParent ← X_p
                latestG ← g_p
            end if
        else                                        ▷ X_p is not a valid parent for step s
            Remove (X_p, g_p) from stored parent/g-value pairs
        end if
    end for
    return (latestParent, latestG)
end function
```

Algorithm 6 Tree-Restoring Weighted $A^\star$

```
function restoreSearch(s)
                                ▷ restores the search state to just after the expansion at step s
    OPEN ← ∅
    CLOSED ← ∅
    CREATED' ← ∅
    if s ≤ 0 then
        initializeSearch(X_S, X_G)
        return
    end if
    for all X ∈ CREATED do
        if E(X) ≤ s then                                        ▷ state created and expanded
            (X_p, g) ← updateParents(X, s)
            g(X) ← g
            parent(X) ← X_p
            insert(CLOSED, X)
            insert(CREATED', X)
        else if C(X) ≤ s then                                   ▷ state created, not expanded
            (X_p, g) ← updateParents(X, s)
            g(X) ← g
            v(X) ← ∞
            parent(X) ← X_p
            f(X) ← g + ϵ · h(X')
            insertOpen(X, f(X))
            E(X) ← ∞
            insert(CREATED', X)
        else                                                    ▷ state not created
            clearParents(X)
            g(X) ← ∞
            v(X) ← ∞
            parent(X) ← ∅
            C(X) ← ∞
            E(X) ← ∞
        end if
    end for
    CREATED ← CREATED'
    step ← s + 1
end function
function Expand(X)
    v(X) ← g(X)
    for all X' ∈ successors of X do
        if X' was not visited before then
            g(X') = ∞
        end if
        g' ← g(X) + cost(X, X')
        if g' ≤ g(X') then
            g(X') ← g'
            storeParent(X',(X, g'),step)
            f(X') ← g' + ϵ · h(X')
            if X' ∉ CLOSED then
                if X' ∉ OPEN then
                    insertOPEN(X', f(X'))
                    C(X') ← step                                ▷ record state put in OPEN
                    insert(CREATED, X')
                else
                    updateOPEN(X', f(X'))
                end if
            end if
        end if
    end for
    E(X) ← step                                                 ▷ record state expanded
    insert(CLOSED, X)
    step ← step + 1
end function
```

CHAPTER 7 : Application: *PAD* for Navigation

In this section we discuss our results from applying our framework for Planning with Adaptive Dimensionality to the domain of path planning for a non-holonomic vehicle done in three dimensions—(x,y,heading θ). The low-dimensional representation which we used in all the navigation planning experiments described below was the 2-dimensional (x,y) position of the vehicle, disregarding the heading information. We used a very simple projection function λ to transform 3D states to 2D states:

$$\lambda_{3D/2D}(x, y, \theta) = (x, y).$$

We used a 16-discretized value for the heading angle, thus, our λ^{-1} mapping was:

$$\lambda^{-1}_{3D/2D}(x, y) = \{(x, y, 0), ..., (x, y, 15)\}.$$

Our algorithm implementation kept track of the high-dimensional regions of the environment as circles. This allowed us to quickly check if a state falls inside a region or not, and also quickly add new regions and grow the sizes of existing ones.

The graph G representing the problem was constructed as a lattice-based graph, similar to the approach taken in (Likhachev and Ferguson, 2008), except we used constant resolution for all lattices. In lattice-based planning, each state consists of a vertex encoding a state vector and edges corresponding to feasible transitions to other states. The set of edges incident to a state are computed based on a set of pre-computed motion primitives, which are executable by the robot.

7.1. Non-Incremental 3D Path Planning for a Non-Holonomic Vehicle

The results reported in this section were originally published in our work (Gochev et al., 2011) presented at the Symposium on Combinatorial Search (SoCS 2011).

7.1.1. Implementation Details

We modeled our environment as a planar world and a polygonal robot. For 3D states we performed accurate collision-checking of the robot's footprint against the obstacles in the environment. We used a relaxed collision model for 2D states as heading information was unavailable, treating the robot as a circle of radius equal to the radius of the inscribed circle of the robot's perimeter.

The set of motion primitives used for 3D states consisted of long straight, short straight, left and right turn elements for both forward and reverse motion, as can be seen in the lower left corner of Fig. 5. The motion primitives used for 2D states were the eight neighboring states (8-connected 2D grid), as seen in the upper left of Fig. 5. It should be noted that the motion primitives for 2D states do not produce feasible paths.

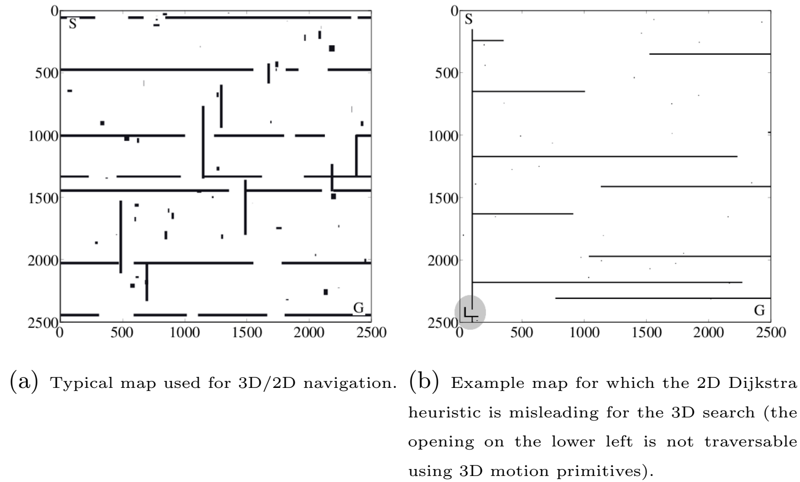
The relaxed collision model and the lower costs of edges in the 2D regions ensured that the cost functions for our 3D and 2D regions satisfied Equation 4.1.

7.1.2. Experimental Evaluation

We compared our algorithm to a weighted A* planner performing full 3D search on several different map sizes. Small maps with few hundred cells in each dimension were quickly solved by the full 3D planner, so little benefit was seen of our algorithm. On maps with 5000 or more cells in both x and y dimensions, the full 3D planner was unable to find a solution due to memory constraints, while our algorithm, having to expand a lot fewer states, was still able to plan successfully.

As a middle ground and to prevent the results from being skewed by the 3D planner having to use the significantly slower hard drive swap space, we randomly generated 50 2500x2500 cell maps for our test runs (typical example can be seen in Fig 7.13(a)).

In all instances we used a Dijkstra's search on the 2D map grid to compute a heuristic accounting for obstacles to help guide the planners towards the goal state. We inflated



(a) Typical map used for 3D/2D navigation. (b) Example map for which the 2D Dijkstra heuristic is misleading for the 3D search (the opening on the lower left is not traversable using 3D motion primitives).

Figure 13: Maps of size 2500x2500 cells.

the obstacles on the map by the inscribed circle radius to preclude the generation of paths through areas too narrow for the robot to physically traverse.

For these experiments, the underlying search algorithm used in both the adaptive planning phase and the tracking phase of our algorithm for Planning with Adaptive Dimensionality was weighted A* and planning was started from scratch at each iteration of the algorithm. In addition, the tunnel width we used for the tracking phase was six cells, and the radii of newly added spheres were 20 cells. Since the longest motion primitive was 10 cells long, these parameter values seemed sufficient to allow reasonable range of maneuvering to occur within a sphere and within the tracking tunnel τ.

For each map three values of the sub-optimality parameter ϵ were tried: 1.1, 1.5 and 3.0 with the adaptive planner using the square root of ϵ for both ϵ_{plan} and ϵ_{track}, giving an overall sub-optimality bound of the adaptive algorithm of ϵ. For both planners a maximum planning time was enforced based on the value of ϵ: $\epsilon = 1.1$: 5 minutes, $\epsilon = 1.5$: 4 minutes, $\epsilon = 3.0$: 3 minutes. If planning time exceeded the time limit the run was reported as a failure.

Algorithm	Sub-optimality Bound	Time (secs)		# 3D Expands (in thousands)		# 2D Expands (in thousands)		Total Expands (in thousands)		Path Cost	
		mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
3D	1.1	142.57	60.24	5218	2177	n/a		5218	2177	58763	9610
adaptive	1.1	184.99	112.93	4448	2884	2434	1793	6957	3946	59202	9856
3D	1.5	83.74	104.94	2813	3533	n/a		2813	3533	68360	11946
adaptive	1.5	25.78	48.96	648	1665	826	1332	1476	2541	66630	13400
3D	3.0	59.99	79.16	2252	3064	n/a		2252	3064	79707	13463
adaptive	3.0	15.21	35.80	396	1319	656	1145	1053	1903	71358	13372

Table 3: Testing results on 50 randomly generated maps for 3D path planning on non-holonomic robot. Non-incremental 3D/2D Adaptive Planner vs. 3D weighted A* planner.

7.1.3. Analysis of Results

We compared the total number of states expanded, number of high-dimensional states expanded, final path cost, and execution time of the adaptive planner compared to the high-dimensional planner, for each of the maps tested. Our results are summarized in table 3.

While the average time for the adaptive planner was significantly shorter than the average time for the 3D weighted A* planner, it is interesting to note that the 3D planner was actually faster on 54 out of 100 runs. When the map was benign, the 2D Djikstra heuristic allowed the 3D planner to expand very few states, particularly at higher ϵ_{plan} values. However, two particular cases led to very long plan times for the 3D weighted A* planner: the case of a map with no solution and the case of a map where the solution required a route very different from the one computed by the heuristic. Of the 18 runs where neither algorithm was able to find a solution in the allowed time the adaptive planner recognized no solution was available in an average of 12 seconds with a maximum of 25 seconds. On the other hand, the 3D planner in all but two cases ran out of allowable execution times (the two cases completed after 177 and 175 seconds for $\epsilon = 1.5$ and $\epsilon = 3.0$ respectively).

The second case where the adaptive planner performed significantly better than the 3D planner is the set of maps where the heuristic for the 3D planner is misleading or has pronounced local minima. An example of this type of map is shown in figure 7.13(b). A significantly shorter path exists from start to goal going through the narrow opening depicted in the lower left. Even after inflating the obstacles, the 2D planner is capable of

finding a route through the narrow passage. However, this path is not executable using the 3D motion primitives. The 3D planner cannot make use of this information and update its heuristic due to its non-iterative nature. The adaptive algorithm initially plans a 2D path through the short-cut, but after attempting to track this path, finds that it cannot negotiate the tight turn and places a sphere at that location. During the next iteration while expanding the 3D states in the sphere the adaptive planner determines that no path through the sphere exists and reverts back to the faster 2D planner to explore other alternative routes. By using the lower-dimensional search to find the alternate route, this search can be performed significantly quicker than the full 3D search.

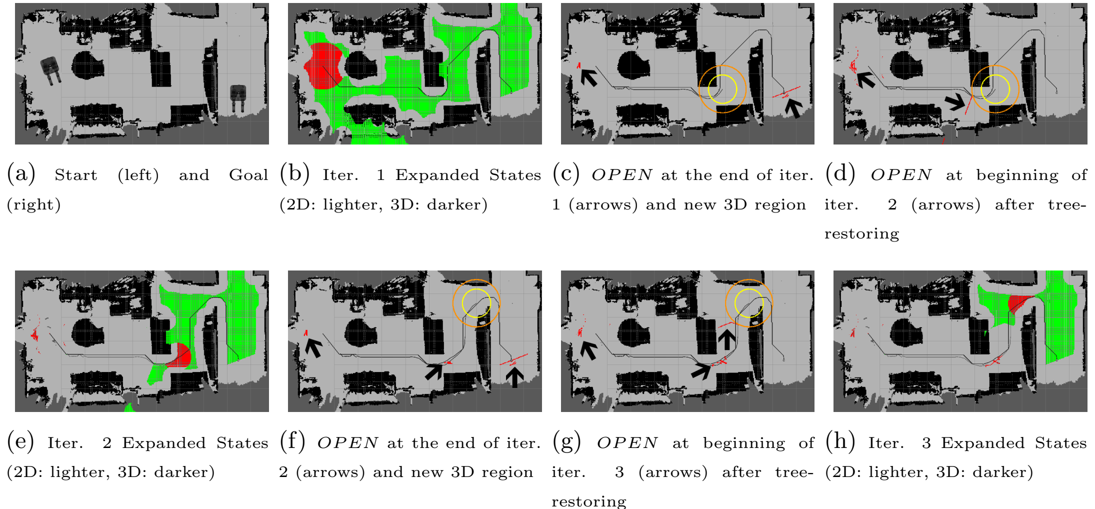
The type of maps used in this experimental evaluation required quite a lot of turning and careful maneuvering, which in turn caused the algorithm to introduce many high-dimensional regions and perform many iterations of planning. In this evaluation, each iteration performed planning from scratch, which caused a lot of redundant computation (state expansions) between iterations. We realized this was a potential area for improvement of the framework for Planning with Adaptive Dimensionality, which motivated the development of the Tree-Restoring Weighted A* graph search algorithm described in Section 6.

7.2. Incremental 3D Path Planning for a Non-Holonomic Vehicle

The results reported in this section were originally published in our work (Gochev et al., 2013) presented at the International Conference on Automated Planning and Scheduling (ICAPS 2013).

7.2.1. Implementation Details

The domain we chose to experimentally validate our incremental version of the Planning with Adaptive Dimensionality algorithm was path planning for non-holonomic vehicles in three dimensions (x,y,heading) with full-body collision checking. We used Willow Garage's PR2 robot as our experimental platform. We used the same approach to 3-DoF planning as



(a) Start (left) and Goal (right)
(b) Iter. 1 Expanded States (2D: lighter, 3D: darker)
(c) $OPEN$ at the end of iter. 1 (arrows) and new 3D region
(d) $OPEN$ at beginning of iter. 2 (arrows) after tree-restoring
(e) Iter. 2 Expanded States (2D: lighter, 3D: darker)
(f) $OPEN$ at the end of iter. 2 (arrows) and new 3D region
(g) $OPEN$ at beginning of iter. 3 (arrows) after tree-restoring
(h) Iter. 3 Expanded States (2D: lighter, 3D: darker)

Figure 14: Example of Planning with Adaptive Dimensionality using tree-restoring weighted A^* search (with no heuristic for illustration purposes). New high-dim. regions introduced in the graph are represented by the inner circles. The outer circles represent states that are affected by the introduction of the new region (modified states). Dark cells indicated by arrows represent the $OPEN$ list (search frontier). Note the reduction of the number of expanded states as iterations progress.

in (Gochev et al., 2011) discussed in Section 7.1—we used lattice-based graphs of uniform resolution (2.5cm×2.5cm) and heading angle values were uniformly discretized into 16 on the interval $(-\pi, \pi]$. We used a set of pre-computed transitions for a non-holonomic robot for 3D states and simple 8-connected 2D grid transitions for the 2D states. The costs of 2D transitions were representative of the distance traveled and the costs of 3D transitions were computed based on the distance traveled, inflated by a pre-computed penalty factor: 3D transitions that required the robot to move backwards had higher penalty factors than transitions moving forward. We used a 2D 8-connected grid-based distance-to-goal heuristic, accounting for obstacles. The heuristic values were computed by a single backward Dijkstra search on the 2D grid. In the incremental versions of the algorithm, every time a new high-dimensional region was introduced, all states falling inside the region and all states on the boundary of the region (states that have valid high-dimensional transitions into the region) were tagged as modified. At the beginning of each iteration the Tree-Restoring Weighted $A^\star$ algorithm ($TRA^\star$) restored a valid search ***state*** with respect to the modified states and

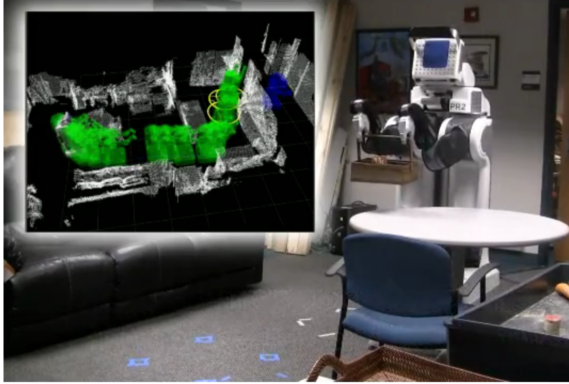


Figure 15: Example run of an Adaptive-Dimensionality planner on an indoor environment. The high-dimensional regions introduced by the algorithm, represented by circles, and the computed path are shown in the embedded figure. 3D planning is performed inside the circles and 2D planning is performed everywhere else in the environment.

Algorithm	Sub-opt. Bound	Time (s)				# Iterations		# 3D Expands		# 2D Expands		Total Expands		Successful Searches
		mean	std dev	min	max	mean	std dev	mean	std dev	mean	std dev	mean	std dev	
3D Weighted A^*	5.0	39.41	34.45	2.42	118.57	n/a		37.29K	32.53K	n/a		37.29K	32.53K	23 of 30
Non-incremental Adaptive	5.0	14.43	15.92	0.89	48.69	2.07	1.10	13.92K	15.52K	1.41K	0.99K	15.31K	16.39K	30 of 30
Tree-restoring A^* Adaptive	5.0	6.86	2.75	0.89	21.75	2.07	1.10	6.83K	6.34K	0.69K	0.29K	7.51K	6.59K	30 of 30
Incremental D^* Adaptive	5.0	10.40	10.80	0.89	34.97	2.07	1.10	7.35K	8.96K	2.22K	1.76K	9.55K	9.96K	30 of 30
Bi-directional RRT	n/a	22.56	20.48	0.03	87.87	n/a		n/a		n/a		n/a		286 of 300

Table 4: Experimental results on 30 scenarios for 3-DoF (x,y,$heading$) path planning (weighted A^* planner vs. non-incremental adaptive-dimensionality planner vs. adaptive-dimensionality planner using tree-restoring weighted A^* vs. adaptive-dimensionality planner using D^*-Lite vs. sampling-based bi-directional RRT planner). The deterministic search-based planners were run only once on each scenario. RRT results are averaged over 10 runs on each scenario. The reported times for RRT do not include trajectory post-smoothing. A search was reported as successful if it took less than 60 seconds to compute a path to the goal.

resumed the search from there, avoiding a lot of redundant computation.

7.2.2. Experimental Evaluation

We compared three implementations of the algorithm for Planning with Adaptive Dimensionality: the non-incremental version of the algorithm, an incremental version using Tree-Restoring Weighted A^* discussed in Chapter 6, and an incremental version using D^*-Lite planner (Koenig and Likhachev, 2002a). We also compared the three adaptive algorithms with a 3D weighted A^* lattice-based planner and a 3D sampling-based bi-directional RRT planner based on the approach taken in (LaValle and Kuffner, 2001b). The RRT planner

Algorithm	Sub-opt. Bound	Time (s) mean	Time (s) std dev	Time (s) min	Time (s) max	# Iterations mean	# Iterations std dev	Time spent in planning phase mean	Time spent in planning phase std dev	Time spent in planning phase min	Time spent in planning phase max
Non-incremental Adaptive	5.0	22.91	15.50	6.41	48.69	2.78	0.83	15.35	13.28	2.11	40.21
Incremental A^* Adaptive	5.0	12.03	5.67	4.96	21.75	2.78	0.83	5.16	3.30	1.42	9.75
Inremental D^* Adaptive	5.0	17.08	10.87	5.29	34.97	2.78	0.83	10.94	8.94	1.69	27.27

Table 5: Statistical data for the 18 scenarios that required more than one iteration of planning demonstrating the benefits of using incremental graph searches in the context of Planning with Adaptive Dimensionality. Using tree-restoring weighted A^* reduced the time spent in the planning phase of the algorithm by a factor of 3.

used controllers for a non-holonomic robot with the same parameters (minimum turning radius and nominal velocity) as the 3D transitions used by the search-based planners. We ran all algorithms on 30 indoor environments of varying degree of difficulty (example can be seen in Fig. 15). Most scenarios exhibited challenging features such as pronounced heuristic local minima and narrow passages. All algorithms performed full-body collision checking (base, torso, arms and head) to ensure that the computed paths were completely collision-free. This is much more computationally expensive (orders of magnitude) than collision-checking just the footprint of the robot, but is much more precise. The adaptive planners used simpler collision checking for 2D states, treating the robot as a point and inflating the obstacles by the robot's inscribed circle radius.

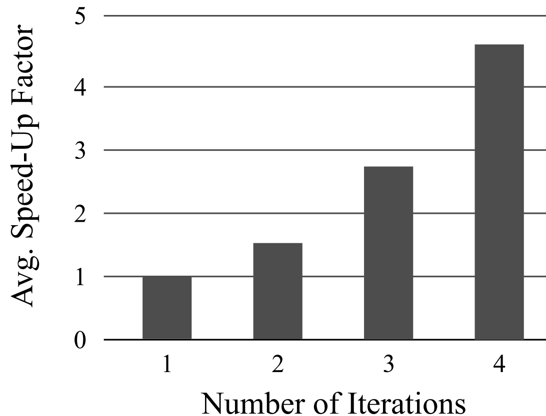


Figure 16: Relationship between the number of iterations performed and the average speed-up factor between non-incremental Adaptive-Dimensionality planner and incremental Adaptive-Dimensionality planner using tree-restoring weighted A^* observed in our 30 experimental scenarios. The incremental algorithm demonstrates better speed-up as the difficulty of the problem increases.

7.2.3. Analysis of Results

As seen in Table 4, both the 3D weighted A^* lattice planner and the bi-directional RRT planner were outperformed by the adaptive algorithms. The poor performance of the RRT algorithm can be attributed to the many narrow passages (such as doorways and narrow gaps between furniture) present in our test environments. A significant drawback of the bi-directional RRT algorithm was the fact that it frequently produced highly sub-optimal paths and paths that required the robot to drive backwards for long periods, which we consider undesirable. The performance of the 3D weighted A^* lattice planner was reasonable only in a few scenarios that did not exhibit local minima of the heuristic function. We observed that the adaptive algorithm using tree-restoring weighted A^* performed best on average, improving performance over the non-incremental version by a factor of 2 on average.

Table 5 compares the performance of the incremental and non-incremental versions of the adaptive algorithm on 18 of the 30 scenarios, which required multiple search iterations to produce a path. On scenarios that required only a single iteration of planning, all three versions of the algorithm behaved identically, since no re-planning was needed. The adaptive algorithm using D^*-Lite performed significantly better than the non-incremental version of the algorithm, improving the overall planning time by a factor of 1.35. However, using D^*-Lite seems to introduce significantly more overhead than using the simple tree-restoring technique for incremental weighted A^* planning. This can be explained by the fact that D^*-Lite needs to generate both successor and predecessor states for all modified states in the graph in order to propagate the inconsistencies in its search tree. This involves expensive collision-checking and some book-keeping overhead. Also, in the context of Planning with Adaptive Dimensionality, edge costs only increase when a new high-dimensional region is added, which results in underconsistent states ($g(X) < rhs(X)$, defined in (Koenig and Likhachev, 2002a)) in the D^* search. D^*-Lite propagates the consistency by expanding these underconsistent states to make them overconsistent ($g(X) > rhs(X)$, defined in (Koenig and Likhachev, 2002a)), after which it may have to expand these states again to make

them consistent ($g(X) = rhs(X)$). Thus, while attempting to correct its search tree, D^*-Lite might have to expand many states twice, which introduces significant overhead. On the other hand, the tree-restoring weighted A^* does not attempt to correct its search tree, but rather quickly identifies a re-usable portion of the search tree and resumes searching from there. In our experiments we observed that the tree-restoring weighted A^* algorithm needed an average of 5 milliseconds to restore itself to a valid previous search ***state*** and resume searching. As a result, tree-restoring weighted A^* improves the performance of the planning phase of the algorithm for Planning with Adaptive dimensionality by a factor of 3 on average and seems to be a better incremental search alternative than D^*-Lite in this context. As shown in Fig. 16, the performance benefit of using tree-restoring weighted A^* increases as the difficulty of the search problem increases and more iterations are needed to solve it, since a lot of redundant computation is avoided by using incremental search.

7.3. Interleaving Planning and Execution

7.3.1. Motivation

The iterative nature of our framework for Planning with Adaptive Dimensionality and the fact that the tracking phase of each iteration is able to produce at least a partial trajectory towards the goal, suggested that the algorithm can be extended to support the interleaving of planning and execution, thus reducing the robot's idle time while waiting for the planner to produce a complete trajectory to the goal. A high-dimensional trajectory piece from the robot's current state towards the goal is available after every tracking phase. This trajectory piece, or at least the first few actions of it, can be sent to the controller for execution, while the planner continues to search for a complete solution. Therefore, the robot's idle time is reduced to the time it takes for a single iteration of the algorithm to complete and makes the system more responsive. This motivated the following extension to our framework in order to support interleaving planning and execution, while preserving the theoretical guarantees of the original algorithm. The extension is general and it is not restricted to navigation planning, but we chose this domain for our experimental evaluation.

7.3.2. Algorithm Extension and Implementation Details

Structurally, our algorithm interleaving Planning with Adaptive Dimensionality and execution (Algorithm 7) is very similar to our original algorithm for Planning with Adaptive Dimensionality (Algorithm 1). It iteratively constructs and searches a graph G^{ad} consisting of largely low-dimensional states, and high-dimensional states are only introduced in areas of the state-space that require high-dimensional planning to ensure the feasibility and the cost sub-optimality bound of the resulting path.

As before, each iteration of the algorithm consists of two phases—planning and tracking. In the planning phase G^{ad} is searched for a path π_{plan} of bounded sub-optimality from start to goal (Alg. 1 Line 6, Alg. 7 Line 9). Then, in the tracking phase, a high-dimensional tunnel τ is constructed around the path π_{plan} produced in the planning phase. This tunnel is then searched for a path π_{track} from start to goal, which consists entirely of high-dimensional states and transitions, and thus is feasible for execution by the robot (Alg. 1 Line 12, Alg. 7 Line 21). The cost of π_{track} is checked to satisfy a sub-optimality constraint condition ($c(\pi_{track}) \leq \epsilon \cdot c(\pi_{plan})$). If the sub-optimality constraint on π_{track} is not met or the tracking phase fails to produce a path trough the tunnel τ, more high-dimensional regions are added to G^{ad} or existing regions are grown.

Lines 12-16 of Algorithm 7 are taken from the algorithm presented in (Zhang et al., 2012), which are important in order to guarantee that the algorithm is complete and terminates, provided that actions have inverses and can be undone.

At the end of each iteration of Algorithm 7, a high-dimensional trajectory piece is extracted from the complete or partial trajectory produced by the tracking phase and is sent to the controller for execution. The planner then advances the start state to the last state of the produced trajectory piece and continues with the next iteration of planning. To ensure that high-dimensional planning is done near the start state, a moving high-dimensional region is always associated with the current start state S_{curr} in the current instance of G^{ad}.

Next, we provide a brief overview of the important functions used by our algorithm extension.

ComputeTunnelPath(τ**,** S_{start}**,** S_{goal}**,** $[t_{limit}]$**)** is a forward graph search that computes a least-cost path from S_{start} to S_{goal} in a tunnel τ. τ is a high-dimensional graph, subgraph of G^{hd}. If the graph search is able to successfully compute a path π from start to goal, it returns the ordered pair $[\pi, true]$ indicating that a path to the goal was computed successfully. If the search fails, it returns the ordered pair $[\pi_{partial}, false]$, where $\pi_{partial}$ is a path from S_{start} to the state $S_{farthest}$—the state farthest along the tunnel τ which was expanded during the search. $false$ indicates that the search was unable to reach the goal and that a partial path is returned. If a time limit t_{limit} is specified, anytime graph search is used to compute a path within the allowed time.

ComputeADPath(G^{ad}**,** S_{start}**,** S_{goal}**,** $[t_{limit}]$**)** is a backward A^* graph search that computes a least-cost path from S_{start} to S_{goal} in a hybrid graph instance G^{ad}. A high-dimensional region is always inserted in the current instance of G^{ad} at the location of the given start state S_{start}. The function returns a pair $[\pi, g]$, where π is a path from start to goal (if one exists), and g is an array containing the corresponding A^* g-values of the states on the path π. Notice, that the g-values along the path π are representing cost-to-goal, and thus are strictly decreasing along π. If a time limit t_{limit} is specified, anytime backward A^* search is used to compute a path within the allowed time limit.

ExtractTrajectoryPiece(π**,** $[t_{limit}]$**)** is a function that, given a path $\pi = \{S_1, S_2, ..., S_n\}$, returns the largest possible high-dimensional trajectory piece $\pi_{hd} = \{S_1, ..., S_i\}$, such that $S_k \in G^{hd} \forall k = 1..i$. If time limit t_{limit} is specified, the returned trajectory π_{hd} is truncated, so that its estimated execution time does not exceed t_{limit}.

7.3.3. Theoretical Properties

In this section, we prove the following theoretical properties of Algorithm 7.

Theorem 7.1 *At the conclusion of every iteration i of Algorithm 7, lines 9-31 for $i > 1$, one of the following conditions holds prior to the execution of line 33:*

- ***Case 1:*** $g^{i-1}(S_{prev}) > g^i(S_{curr})$
- ***Case 2:*** $\exists S \in G^{hd} : P^{i-1}(S) < P^i(S)$
- ***Case 3:*** $\exists j < i$ *such that* $(\|Q_c^j\| < \|Q_c^i\|$ *or* $G_{ad}^j \neq G_{ad}^i)$ *and* $S_{curr}^j = S_{curr}^i$ *and* $\forall k = j+1, ..., i-1$, $S_{curr}^k \neq S_{curr}$, *where* S_{curr}^j *denotes the value of* S_{curr} *right before line 33 is executed during the* j*-th* *iteration.*
- ***Case 4:*** $G_{ad}^{i-1} \neq G_{ad}^i$.

Where G_{ad}^k *denotes the instance of* G^{ad} *that was searched during iteration* k*.*

Proof Assume **Case 1** does not hold. Then $g^{i-1}(S_{prev}) \leq g^i(S_{curr})$. We have the following cases:

- If $P^{i-1}(S_{curr}) = 0$, then by line 16 S_{curr} will be moved to Q_1 and $P^i(S_{curr}) = 1 > P^{i-1}(S_{curr})$. Thus, **Case 2** will hold.
- Alternatively, if $P^{i-1}(S_{curr}) = 1$, therefore S_{curr} has been the start state at least once before since only start states are inserted into Q_1. Let j be the most recent iteration that started from S_{curr} and let $g^j(S_{curr})$ be the corresponding g-value. By line 17, $g_{low}(S_{curr}) \geq g^j(S_{curr})$. Then, either $\|Q_c^i\| = \|Q_c^j\|$ or $\|Q_c^i\| > \|Q_c^j\|$. If $\|Q_c^i\| > \|Q_c^j\|$, then **Case 3** holds. Alternatively, if $\|Q_c^i\| = \|Q_c^j\|$, then we can assert that either that the graphs searched in iterations j and i were identical $G_{ad}^j = G_{ad}^i$, or not $G_{ad}^j \neq G_{ad}^i$. If $G_{ad}^j \neq G_{ad}^i$, then **Case 3** holds. Else, we have $g^i(S_{curr}) = g^j(S_{curr})$, since the graphs that were searched in iteration i and iteration j were identical and had the same start and goal states. Since $g^i(S_{curr}) \geq g^{i-1}(S_{prev}) > g^{i-1}(S_{curr})$, then $g_{low}(S_{curr}) > g^{i-1}(S_{curr})$. In this case the condition on Line 12 will hold, and a new state will be introduced in Q_c. Therefore $\|Q_c^i\| > \|Q_c^j\|$, which is a contradiction.

- Finally, if $P^{i-1}(S_{curr}) = 2$, it indicates that S_{curr} is already the center of a high-dimensional region. We also have $g^{i-1}(S_{prev}) > g^{i-1}(S_{curr})$. If a new state was added to Q_1 or Q_c during iteration i, then **Case 2** will hold. Let's assume no new states were added to Q_1 and Q_c, and also that **Case 4** does not hold (i.e. $G_{ad}^{i-1} = G_{ad}^{i}$). Since the graphs searched during iterations $i-1$ and i were identical, the goal state is the same, and the current start state S_{curr} was closer to goal than the previous start S_{prev} according to the search in iteration $i-1$, then it must be the case that $g^{i-1}(S_{prev}) > g^{i-1}(S_{curr}) \geq g^{i}(S_{curr})$, which contradicts the assumption that **Case 1** does not hold.

□

Theorem 7.2 *On a finite graph, Algorithm 7 is guaranteed to reach the goal state if any state reachable from the start has a feasible path to the goal.*

Proof By theorem 7.1, through every iteration of our algorithm, at least one of four cases hold. We will argue that each of the four cases can occur only a finite number of times on a finite graph. This implies that our algorithm must run for a finite number of iterations and terminate.

Case 1 states that the cost-to-goal of the current start state S_{curr} is strictly decreasing over time, but it is also bounded from below by 0 and only takes integer values. Thus, **Case 1** can only be true a finite number of times, before $g^{i}(S_{curr})$ reaching 0.

Case 2 states that the priority of some high-dimensional state increased during the iteration. The priority of each state can increase at most twice. As we have a finite number of high-dimensional states N, we can increase their priority at most $2N$ times.

Case 3 states that if we use a state S for a second time as the current start state, then we must have added a new high-dimensional state to Q_c or the current instance of G^{ad} has changed since the last iteration that we used S as start state. Since we have a finite

number of states, we can add new states to Q_c only a finite number of times. Moreover, the instance of G^{ad} changes either when the environment changes, or when a new high-dimensional region is introduced. We assume that the environment changes a finite number of times. Also, we can only introduce a finite number of high-dimensional regions before G^{ad} becomes identical to G^{hd} and stops changing. Thus, the instance of G^{ad} can only change a finite number of times. Therefore, **Case 3** can only hold a finite number of times.

Case 4 states that the instances of G^{ad} used in the previous iteration is different from the current instance of G^{ad}. As we argued above, the instance of G^{ad} can only change a finite number of times. Thus, **Case 4** can only hold a finite number of times.

We have shown that our algorithm always terminates. Now we will argue that if any state reachable from the start has a feasible path to the goal, then the algorithm will terminate by reaching the goal. For the algorithm to terminate in any other way other than reaching the goal state, it must fail to find a path from S_{curr} to S_{goal} in the current instance of G^{ad} during some iteration. Assume our graph search from S_{curr} to S_{goal} failed. Thus, S_{goal} is not reachable from S_{curr}. However, S_{curr} is reachable from S_{start}, since during the previous iterations our algorithm has produced a path from S_{start} to S_{curr}. This contradicts our assumption that any state reachable from the start has a feasible path to the goal. □

Observation 7.1 *On a finite graph, Algorithm 7 will start producing trajectories of bounded sub-optimality after finitely many iterations.*

We leave this claim without an official proof, but we give the following argument to support it. In order to guarantee the sub-optimality of the produced trajectory, the tracking phase of our algorithm must complete successfully and find a path to the goal that satisfies the sub-optimality constraint on Line 26. The two other possible results of the tracking phase serve to identify locations along the proposed adaptive path π_{ad}, where high-dimensional planning is needed to ensure feasibility and sub-optimality bound. At each iteration, the algorithm either identifies a new location where a high-dimensional region will be introduced,

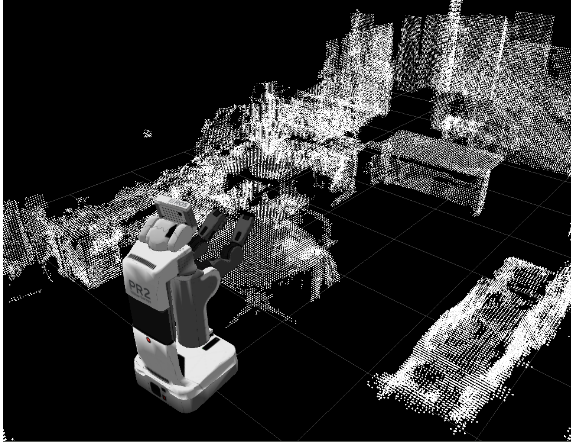


Figure 17: PR2 in a cluttered indoor environment.

or produces a trajectory that satisfies the sub-optimality constraint. After finitely many iterations, sufficiently many high-dimensional regions will be introduced that the tracking phase will find a path satisfying the sub-optimality constraint. In the worst case, π_{ad} would go through only high-dimensional regions (no low-dimensional segments on the path), in which case the tracking phase will be able to match π_{ad} perfectly ($\pi_{\tau} = \pi_{ad}$), and thus, satisfy the sub-optimality constraint for any $\epsilon_{track} \geq 1$.

We also developed a time-constrained version of our algorithm for Interleaving Planning with Adaptive Dimensionality and Execution (Algorithm 8), where the user can specify the maximum amount of time t_{lim} that each iteration can take, and thus, the maximum amount of time before the next trajectory piece is computed and sent for execution. Algorithm 8 must use anytime graph searches in order to ensure timely completion of each iteration. The same theoretical properties proven for Algorithm 7 hold for Algorithm 8, provided that the specified time limit is sufficiently large for **ComputeADPath** to complete in each iteration. If t_{lim} is not sufficient to perform the tracking phase of an iteration (Algorithm 8, Line 23), the algorithm behaves like the algorithm presented in (Zhang et al., 2012).

Algorithm	Sub-optimality	Planning Time (secs)				Idle Time (secs)				Execution Time (sec.)				# Iterations				Successful
	Bound	mean	std dev	min	max	mean	std dev	min	max	mean	std dev	min	max	mean	std dev	min	max	Plans
3D ARA*	5.0	14.0	16.3	4.8	60.0	14.0	16.3	4.8	60.0	161.8	30.1	87.5	214.1	n/a				42 of 50
Original Adaptive	5.0	9.6	10.3	1.6	39.4	9.6	10.3	1.6	39.4	128.8	32.6	83.2	257.4	2.9	2.3	1	10	50 of 50
Global/Local$_1$	n/a	9.7	3.9	5.6	19.9	0.75	0.3	0.4	1.5	182.7	28.4	95.4	336.2	14.5	3.2	9	19	38 of 50
Global/Local$_2$	n/a	10.2	3.2	5.8	22.4	0.82	0.3	0.4	1.7	171.3	34.2	93.7	321.5	13.8	4.1	9	18	50 of 50
Interleaving Adaptive	5.0	9.8	10.3	1.6	40.1	2.4	0.6	1.4	3.2	128.8	32.6	83.2	257.4	2.9	2.3	1	10	50 of 50

Table 6: Experimental results on 50 indoor scenarios comparing interleaving and non-interleaving planners.

7.3.4. Experimental Evaluation

To experimentally validate our algorithm, we chose the problem of non-holonomic navigation planning with full-body collision-checking for Willow Garage's PR2 robot, as in Section 7.2. The task in our experiments was to navigate the robot through a cluttered indoor environment (Fig. 17).

We ran a number of simulation experiments to compare the performance of our planner to other similar algorithms. Each algorithm was run on 50 scenarios with varying degree of difficulty. For each algorithm we used a heuristic computed by running a 2D Dijkstra search. Some of the more challenging scenarios contained passages that seemed traversable when planning in 2D, but in reality were not ("*false passages*"), thus exhibiting pronounced heuristic local minima. Table 6 summarizes the simulation results we observed for each of the planners.

7.3.5. Analysis of Results

When comparing our algorithm with a 3-DOF Anytime Repairing A* planner (3D ARA*), we observed that our algorithm was able to achieve faster planning times and greater planning success rate. The ARA* planner failed to produce a plan within 60 seconds on 8 of the scenarios. Since the ARA* planner did not interleave planning and execution, the robot was idle for the entire time it took for the planner to produce a path.

We also compared our algorithm with a planner (Global/Local$_1$) combining a local 3D planner (near the robot's current position) and a global 2D planner. This algorithm exhibited very low robot idle times since planning episodes completed very quickly. However, one dis-

advantage of this planner is that it does not provide bounds on solution cost sub-optimality and often the planner produced highly sub-optimal trajectories that required much more time to execute than the trajectories produced by our algorithm. Another disadvantage is that "*false passages*" caused the planner to oscillate infinitely, thus failing to reach the goal on 12 of the 50 scenarios. Moreover, since 3D planning is done only in the vicinity of the start state, the planner can only produce a very small piece of executable trajectory at each planning episode, thus requiring many iterations of re-planning until the goal is reached and exhibiting high overall planning times.

The algorithm presented in (Zhang et al., 2012) (Global/Local$_2$) exhibited performance most similar to our planner. It was able to successfully navigate the robot to the goal in all 50 scenarios and had low robot idle times. However, it produced very sub-optimal trajectories in scenarios with "*false passages*". It navigated the robot to the "*false passage*" before realizing that the passage was not traversable and finding an alternative route. In contrast, the tracking phase of our planner was able to "look ahead" and identify such "*false passages*" much earlier than the Global/Local$_2$ algorithm. Similarly to Global/Local$_1$, Global/Local$_2$ could only produce a small piece of executable trajectory at each planning episode and required many iterations of re-planning before reaching the goal, hence the high overall planning time.

The proposed extension to the framework for Planning with Adaptive Dimensionality allows for faster response times of the system at the cost of possibly taking sub-optimal actions in the early stages of the planning process. However, we have demonstrated that the extension provides important theoretical guarantees, such as completeness and termination. Moreover, we argue that after a finite number of planning iteration the planner will start producing solutions of bounded cost sub-optimality.

Algorithm 7 Interleaving Planning with Adaptive Dimensionality and Execution

1: $G^{ad} = G^{ld}$
2: $G^{ad} =$ Add-HD-Region($G^{ad}, \lambda(S_{goal})$)
3: $i = 0$, $Q_c = \emptyset$, $Q_1 = \emptyset$
4: $g_{low}(S) = 0$, $g^0(S) = \infty$, $\forall S$
5: $S_{prev} = S_{curr} = S_{start}$
6: **loop**
7: Update map with sensor data
8: $i = i + 1$
9: $[\pi^i_{ad}, g^i] =$ ComputeADPath(G^{ad}, S_{curr}, S_{goal})
10: **if** π^i_{ad} is not found **then return** No path from S_{curr} to S_{goal} exists
11: **end if**
12: **if** $\exists S \in \pi^i_{ad} : (S \in Q_1) \wedge (g^i(S) < g_{low}(S))$ **then**
13: move S from Q_1 to Q_c
14: $G^{ad} =$ Add-or-Grow-HD-Region(G^{ad}, $\lambda(S)$)
15: **end if**
16: **if** $g^i(S_{curr}) \geq g^{i-1}(S_{prev})$ **then**
17: $g_{low}(S_{curr}) = max(g^i(S_{curr}), g_{low}(S_{curr}))$
18: insert/update S_{curr} in Q_1 with $g_{low}(S_{curr})$
19: **end if**
20: Construct a tunnel τ around π^i_{ad}
21: $[\pi_\tau, ReachedGoal] =$ ComputeTunnelPath(S_{curr}, S_{goal}, τ)
22: **if** $ReachedGoal = false$ **then**
23: Let S_{end} be the last state on the returned partial path π_τ
24: $G^{ad} =$ Add-or-Grow-HD-Region(G^{ad}, $\lambda(S_{end})$)
25: $\pi_{partial} =$ ExtractTrajectoryPiece(π_τ)
26: **else if** $c(\pi_\tau) > \epsilon_{track} \cdot c(\pi^i_{ad})$ **then**
27: Identify state(s) S_r with large cost discrepancy between π^i_{ad} and π_τ where to insert new HD region(s)
28: $G^{ad} =$ Add-or-Grow-HD-Region(G^{ad}, S_r)
29: $\pi_{partial} =$ ExtractTrajectoryPiece(π_τ)
30: **else**
31: $\pi_{partial} =$ ExtractTrajectoryPiece(π_τ)
32: **end if**
33: $S_{prev} = S_{curr}$
34: Advance S_{curr} to last state on $\pi_{partial}$
35: Send $\pi_{partial}$ to controller for execution
36: **if** $S_{curr} = S_{goal}$ **then return** reached the goal
37: **end if**
38: **end loop**

Algorithm 8 Interleaving Time-Constrained Planning with Adaptive Dimensionality and Execution

1: $G^{ad} = G^{ld}$
2: G^{ad} = AddFullDimRegion($G^{ad}, \lambda(S_{goal})$)
3: $i = 0$, $Q_c = \emptyset$, $Q_1 = \emptyset$
4: $g_{low}(s) = 0$, $g^0(s) = \infty$, $\forall S$
5: $S_{prev} = S_{curr} = S_{start}$
6: Set iteration time limit t_{lim} as defined by user
7: **loop**
8: Update map with sensor data
9: $i = i + 1$
10: $[\pi^i_{ad}, g^i]$ = ComputeADPath(G^{ad}, S_{curr}, S_{goal}, t_{lim})
11: **if** π^i_{ad} is not found **then return** No path from S_{curr} to S_{goal} found within t_{lim}
12: **end if**
13: **if** $\exists S \in \pi^i_{ad} : (S \in Q_1) \wedge (g^i(S) < g_{low}(S))$ **then**
14: move S from Q_1 to Q_c
15: G^{ad} = AddOrGrowFullDimRegion(G^{ad}, $\lambda(S)$)
16: **end if**
17: **if** $g^i(S_{curr}) \geq g^{i-1}(S_{prev})$ **then**
18: $g_{low}(S_{curr}) = max(g^i(S_{curr}), g_{low}(S_{curr}))$
19: insert/update S_{curr} in Q_1 with $g_{low}(S_{curr})$
20: **end if**
21: Let t_{plan} be the time elapsed since the beginning of the iteration
22: Set $t_{available} = t_{lim} - t_{plan}$
23: **if** $t_{available} \leq 0$ **then**
24: $\pi_{part.}$ = ExtractTrajectoryPiece(π^i_{ad}, t_{lim})
25: **else**
26: Construct a tunnel τ around π^i_{ad}
27: $[\pi_\tau, ReachedGoal]$ = ComputeTunnelPath(S_{curr}, S_{goal}, τ, $t_{available}$)
28: **if** $ReachedGoal = false$ **then**
29: Let S_{end} be the last state on the returned partial path π_τ
30: G^{ad} = AddOrGrowFullDimRegion(G^{ad}, $\lambda(S_{end})$)
31: $\pi_{partial}$ = ExtractTrajectoryPiece(π_τ, t_{lim})
32: **else if** $c(\pi_\tau) > \epsilon_{track} \cdot c(\pi^i_{ad})$ **then**
33: Identify a state S_r where a new FullDimRegion needs to be introduced
34: G^{ad} = AddOrGrowFullDimRegion(G^{ad}, S_r)
35: $\pi_{partial}$ = ExtractTrajectoryPiece(π_τ, t_{lim})
36: **else**
37: $\pi_{partial}$ = ExtractTrajectoryPiece(π_τ, t_{lim})
38: **end if**
39: $S_{prev} = S_{curr}$
40: Advance S_{curr} to last state on $\pi_{partial}$
41: Send $\pi_{partial}$ to controller for execution
42: **if** $S_{curr} = S_{goal}$ **then return** reached the goal
43: **end if**
44: **end if**
45: **end loop**

CHAPTER 8 : Application: *PAD* for Multi-Robot Collaborative Navigation

Robots are being utilized in an increasing number and variety of situations, which provides many opportunities to collaborate between robots in different ways. Different robots can provide computational or sensing resources for each other, they can act as transports, provide communication relays, or provide other kinds of support. For these types of tasks, the robots need to be able to generate plans that take into account the differences in movement, sensing, and localization abilities of the team members in order to take full advantage of the team's capabilities.

In some scenarios, these differences within the team can be significant. Teams composed of a ground vehicle and an aerial vehicle differ in many important ways. They have drastically different on-station endurance times, different payload capacities (which impact the number, types, and precision of sensors), and can traverse different types of terrain. However, these differences can be used to make the team more capable than they are individually. For many tasks, such as search and rescue, both the high endurance of the unmanned ground vehicle (UGV) and the capability to traverse difficult environments typical of unmanned aerial vehicles (UAV) are important. Thus, the robustness and effectiveness of a team of robots can be improved significantly by leveraging the strengths of the individual members. In this scenario, the UAV's limited payload places limits on the sensors it can carry, while the environment places external limitations on the availability of common localization methods such as GPS. It is important that the planner is capable of generating trajectories that use all of the capability of both vehicles, including the ability to gain information from each other.

Our approach incorporates the recently developed planning framework State Lattice with Controller-based Motion Primitives (*SLC*) (Butzke et al., 2014) into the framework for Planning with Adaptive Dimensionality. *SLC* allows plans to incorporate multiple different modes of localization that a robot has available along with the associated collaboration

constraints into a unified planning framework. Moreover, when planning for teams of robots, the degrees of freedom of the system increase dramatically, which makes the framework for Planning with Adaptive Dimensionality a suitable approach for such planning problems.

Our goal was to implement a planner that allows a UAV with very limited computation, sensing, and localization capabilities to navigate to a desired location in an indoor environment, while using assistance from a ground robot that is able to accurately localize itself within the environment. The UAV had several controllers, which allow it to autonomously navigate certain parts of the environment without accurate localization, such as following a wall and going around 90° corners. Metric motions were only available to the UAV when it was well-localized—it had clear line of sight to the ground robot or a known landmark in the environment. Thus, the planner needed to produce synchronized trajectories for both robots that allowed the UAV to navigate through the environment and reach the desired goal location.

8.1. Related Work

Collaborative localization has been a goal of robotics research for many years (Fox et al., 1999). A lot of this work has been directed at making the detection of the other robots of a team more reliable and accurate (De Silva et al., 2012), even for chains of robots, where the farthest ones have no direct knowledge or sensor measurements regarding any known landmarks, and instead, must rely entirely on their neighboring robots (Wanasinghe et al., 2014). Other approaches have focused on the sensor data integration (data fusion), ensuring that the data is used more effectively (Song et al., 2008). Our approach keeps the localization scheme simple, we used fiducial markers and a simple camera to determine the estimated relative pose of the UGV from the UAV. Then, using the strong localization capabilities of the UGV and the relative position of the UAV, we can compute an accurate estimate of the position of the UAV in the environment. While we did not use these advanced data fusion techniques in this work, our algorithm is capable of incorporating this improved data into its planning framework.

With the recent increase in the availability of small, low cost UAV's, in particular easy to use quadcopters, more research effort has been directed at teams of air-ground robots (Lacroix and Le Besnerais, 2011) including work on exploration (Burgard et al., 2005), and collaborative localization between the team members (Rekleitis et al., 2001). Communication in a variety of forms has been the focus of several works in this area (Viguria et al., 2010; Vaughan et al., 2000), although frequently these include high-quality localization of all robots, including the use of GPS on both the UGV and UAV's, even with vision augmentation (Grocholsky et al., 2006). However, some approaches rely purely on well-localized UGV's (Li et al., 2011), forcing the UAV to update its position estimate only by visually extracting the pose of the UGV. Our work differs from these approaches by incorporating the collaborative localization element into a larger planning framework.

An in-depth look at multi-robot localization and planning for air-ground teams of robots is found in (Peasgood, 2007). In this work, the planner uses a simplified topological approach to planning and is not sufficient to be used directly by the UGV and UAV for navigation through a complex 3-dimensional environment.

The state lattice with controller-based motion primitive planner allows the execution of controllers similar to the sequential composition of controller approaches (Burridge et al., 1999; Conner et al., 2011; Kallem et al., 2011). The *SLC* planner also includes the functionality of switching between controllers based on external perceptual triggers similar to the Linear Temporal Logics (Kress-Gazit et al., 2007).

The key feature that distinguishes our work from the prior work in this domain is that we include the collaborative localization element directly in our planning process. This allows the robots to go on separate trajectories and only meet up when required rather than travel in a fixed formation or conversely, operate completely independently.

In this section, we provide a very brief overview of the State Lattice with Controller-based Motion Primitives algorithm. We refer the reader to (Butzke et al., 2014) for a detailed algorithm analysis.

A state lattice-based planner uses a regular lattice constructed from motion primitives to form the search graph, $\mathcal{G} = (\mathcal{S}, \mathcal{E})$. In a typical planner, the edges, $\mathcal{E}$, are formed by applying fixed motion primitives at each state, $s \in \mathcal{S}$. These motion primitives are usually short pre-computed trajectories, which carry the implicit assumption that the robot has sufficient localization capabilities to be able to execute the motion and to determine the stopping point. However, in cases that this does not hold true, we can instead turn to controller-based motion primitives. By adding additional directed edges to the search graph based on forward simulating different types of controllers, the planner is capable of finding trajectories through areas that are impassable using only metric-based motion primitives. These controller-based motions rely solely on the capabilities of the controller, independent of the robots ability to localize. For example, a wall following controller may not, at any point during its trajectory, know where in the environment it is with any degree of precision, however, by executing this controller to its natural stopping point—the end of the wall—the robot ends up in a known (and repeatable) position. Thus, the *SLC* planner (Butzke et al., 2014) adds additional directed edges to the graph, which correspond to executing a controller c from a given set of controllers $\mathcal{C}$, at a given starting state. These new edges are formed by forward simulating the desired controller from a given state, s_i, in order to determine the end state, s_j, and thus forming a new edge, $e(s_i, s_j)$, which is added to the set of states in the search graph.

Formally, *SLC* requires three functions to be defined to generate the graph $\mathcal{G}$, $\mathrm{C}(s)$, $\mathrm{T}(c)$, and $\Phi(s, c, \tau)$. The first function, $\mathrm{C}(s)$ defines the available controllers at a given state, $s \in \mathcal{S}$:

$$\mathrm{C}(s) : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{C})$$

The result is a set of available controllers, $\mathscr{C}$, from the powerset of all controllers, $\mathcal{P}(\mathcal{C})$, i.e. $\mathrm{C}(s)$ provides all of the controllers which can be executed at state s. These controllers can be simple, such as a wall following controller using two range measurements, or complex, such as a full visual odometry system to navigate to a particular key-frame.

The *SLC* algorithm also allows controllers to be stopped in the middle of execution through the use of perceptual triggers. A trigger can be setup to halt a controller based on any perceptual signal, such as sighting a new landmark. In addition, each controller has an intrinsic trigger that is the default stopping point for that controller. For example, a wall following controller has an intrinsic trigger that stops execution when the robot reaches the end of the wall. An example of various controllers and triggers is shown in Fig. 18.

The second required function maps the controllers onto available triggers:

$$\mathrm{T}(c) : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{T})$$

returning a set of available triggers, $\mathscr{T}$, based on the given controller, $c \in \mathcal{C}$. $\mathcal{T}$ is the set of all triggers available to the robot.

The last required function is the actual controller logic defined as:

$$\Phi(s, c, \tau) : \mathcal{S} \times \mathrm{C}(s) \times \mathrm{T}(c) \rightarrow \mathcal{S}$$

For a given state s, an allowable controller c for that state, and an allowable trigger τ for that controller, function Φ simulates the execution of the controller c starting at state s until either trigger τ or an intrinsic trigger is detected (whichever comes first). The resulting state s' is returned by the function.

The problem is thus formally a 6-tuple,

$$\mathcal{G} = \{\mathcal{S}, \mathcal{C}, \mathcal{T}, \mathrm{C}(\cdot), \mathrm{T}(\cdot), \Phi(\cdot, \cdot, \cdot)\}$$

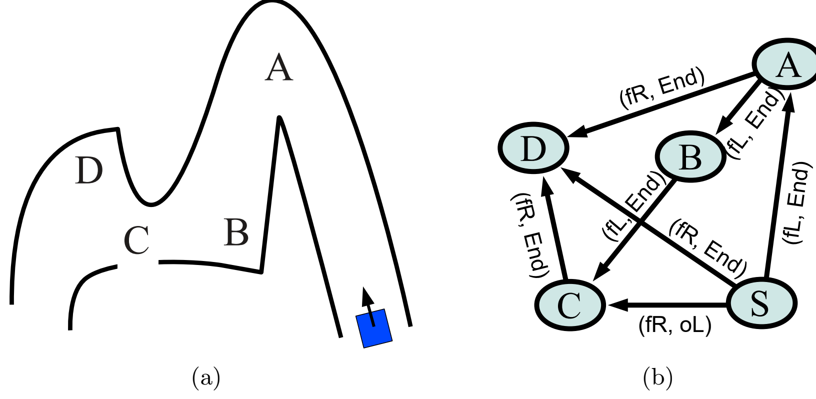


Figure 18: (a) Environment and (b) segment of graph $\mathcal{G}$ based on controllers $\mathcal{C} = \{\text{FollowLeftWall}(fL), \text{FollowRightWall}(fR)\}$ and triggers $\mathcal{T} = \{\text{Completion}(End), \text{OpeningLeft}(oL), \text{OpeningRight}(oR)\}$.

used to produce the graph, $\mathcal{G}$. Further details on the algorithm can be found in (Butzke et al., 2014).

As an example, suppose we are given a set of controllers

$$\mathcal{C} = \{\text{FollowLeftWall}, \text{FollowRightWall}\}$$

with an intrinsic trigger of Completion corresponding to the end of the wall, and a set of extrinsic triggers $\mathcal{T} = \{\text{OpeningLeft}, \text{OpeningRight}\}$. Given an example environment as shown in Fig. 8.18(a) we can see how the graph, $\mathcal{G}$, is constructed in Fig. 8.18(b). Consider a state S, indicated by the square in the lower right corner and suppose both controllers are available at S. From state S there is an edge to A corresponding to the controller FollowLeftWall, fL, and trigger Completion, *End*, as shown in the portion of $\mathcal{G}$. Likewise, with controller FollowRightWall, fR, and trigger *End*, the edge goes from S to D. However, if the trigger were OpeningLeft, oL, then the edge would have been from S to C. Note, it is possible for multiple controller/trigger combinations to connect two nodes. For example, $B \to C$ is formed by the (fL, End) pair in the graph, however $B \to C$ is also connected by the pair (fR, oL) (which is not depicted).

8.3. Implementation Details

A free flying aerial robot has six degrees of freedom—$\langle x, y, z, \text{heading}, \text{roll}, \text{pitch}\rangle$. However, planning is typically done in a 4-dimensional space—$\langle x, y, z, \text{heading}\rangle$—allowing the underlying controller full control over the roll and pitch of the UAV in order to maintain safe flight. For this application, the state-space for the UAV was defined by $\langle x, y, z, \text{controller type}\rangle$, where controller type denoted one of the available UAV controllers for the particular state. The heading of the UAV was intrinsically computed based on the controller type. For example, for WallFollowing controllers, the heading was kept parallel to the direction of the wall, while for GoToLandmark controllers, the heading was kept facing the landmark. The ground vehicle, on the other hand, has 3 positional degrees of freedom—$\langle x, y, \text{heading}\rangle$. Thus, the overall state-space has a total of 7 degrees of freedom—4 for the UAV and 3 for the UGV.

The heuristic for the UAV was computed using 3D BFS search from the $\langle x, y, z\rangle$ position of the goal state on an 26-connected 3D grid accounting for obstacles. The heuristic was not perfect as did not account for the orientation of the robot or its perimeter shape. Thus, some scenarios exhibited pronounced heuristic local minima.

8.3.1. Controllers and Triggers Implemented for Ground-Air Teams

In order to use the *SLC* planner, a set of available controllers and triggers were constructed. For the UAV, we implemented WallFollowing, GoToLandmark, MetricTurn, MetricStep, and GoAroundCorner controllers. The GoAroundCorner controller was specified in terms of the two metric controllers, MetricTurn and MetricStep: GoAroundCorner = MetricStep(0.5m) $\rightarrow$ MetricTurn($\pm 90°$) $\rightarrow$ MetricStep(1.0m). The UGV had high-quality localization data from its two scanning laser range finders and was only given a MetricMotion controller.

Since the UAV does have an IMU and optical flow system there are locations within the environment that it is capable of generating short range metric motions. We used two such

motions, an ability to yaw to a desired heading (MetricTurn), and the ability to move a set distance forward (MetricStep). The accuracy of the IMU and optical flow system did not allow for continuous metric motion without receiving some external sensor information. These metric motions were only available to the UAV when it was well-localized within the environment—within line-of-sight of the UGV or a known landmark. Thus, the drift in the IMU and visual odometry could be corrected by using the accurate localization estimate provided by the landmarks or the UGV. The only exception was the GoAroundCorner controller, which exercised those metric motions for a very limited time.

We used two different instantiations of the GoToLandmark controller for our implementation. The first, ($\text{GoToLandmark}_{\text{Stat}}$), used static landmarks in the environment that the UAV could detect with its on-board camera system and, knowing the position and orientation of the landmark, could determine its own position within the environment. The second controller ($\text{GoToLandmark}_{\text{Dyn}}$) used fiducial markers on the ground robot for the same purpose. These were the only controllers that could compute an accurate localization estimate for the UAV and allowed it to execute precise metric motions. Thus, these controllers allowed the UAV to execute actions, such as move to a target $\langle x, y, z \rangle$ location. However they did require the corresponding landmark or the ground vehicle to remain within line-of-sight throughout the metric motion being used.

The WallFollowing controller on the UAV used two IR range sensors mounted on each side of the UAV in order to maintain a flight path parallel to, and a specified distance from, any given wall in the environment. It was given the ability to trigger when the wall ended (Completion) and when an obstacle was within a certain distance of the front or back of the UAV (Obstacle).

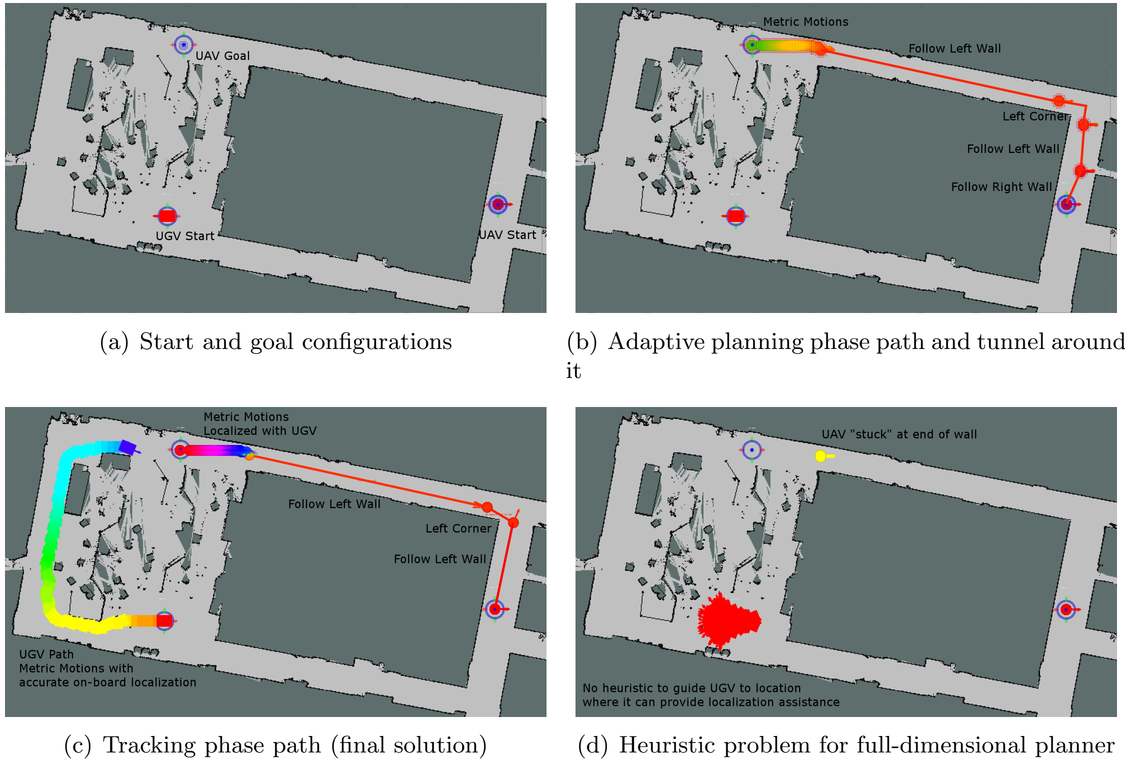
8.3.2. Planning with Adaptive Dimensionality for Ground-Air Teams

The full-dimensional system state was represented by 7-dimensional state-vectors:

$$\langle (x, y, z, \text{controller type})_{\text{uav}}, (x, y, \text{heading})_{\text{ugv}} \rangle$$

The transitions available for each state consisted of pre-computed motion primitives (metric motions) for both vehicles, and state-lattice controller actions for the aerial vehicle. The cost of each transition was proportional to the cumulative distance traveled by each vehicle during the transition. The roll and pitch of the UAV were derived variables and were calculated by the controllers in order to maintain a desired nominal velocity and follow the desired trajectory points. As mentioned previously, the heading of the aerial vehicle was also not a free variable and was determined by the specific controller used in a transition. For example, when executing a WallFollowing transition, the heading is kept parallel to the direction of the wall, and for transitions using the ground vehicle for localization ($\text{GoToLandmark}_{\text{DYN}}$), the heading is kept facing the ground vehicle. We assumed that in many areas of the environment the aerial vehicle is capable of autonomous navigation by using the state-lattice controllers (following walls or going around corners, for example), and the localization assistance of the ground vehicle is needed only in rare occasions, when no state-lattice controllers are available to the UAV and metric motions need to be performed. Thus, the low-dimensional representation of the system used for Planning with Adaptive Dimensionality was a 4-dimensional state-vector $\langle x, y, z, \text{controller type}\rangle_{\text{uav}}$, only considering the position of the aerial vehicle and its available controllers. For low-dimensional states, we allowed metric motions to be executed at any point, since such states did not have information about the location of the UGV. In other words, for low-dimensional regions, we assumed that the UGV is in a location that allows the UAV to localize using $\text{GoToLandmark}_{\text{DYN}}$ controller. The costs of transitions in the low-dimensional space satisfies 4.1 as only the cost of moving the aerial vehicle is considered and the restriction of using metric motions was relaxed. High-dimensional regions are introduced in the hybrid graph only in areas of the environment where ground vehicle localization assistance is needed and the planner needs to consider how to navigate the ground vehicle to an appropriate location to provide localization assistance.

The goal was specified only in terms of desired position for the UAV and a tolerance radius. The ground vehicle had no desired goal position and moved only when it had to provide



(a) Start and goal configurations

(b) Adaptive planning phase path and tunnel around it

(c) Tracking phase path (final solution)

(d) Heuristic problem for full-dimensional planner

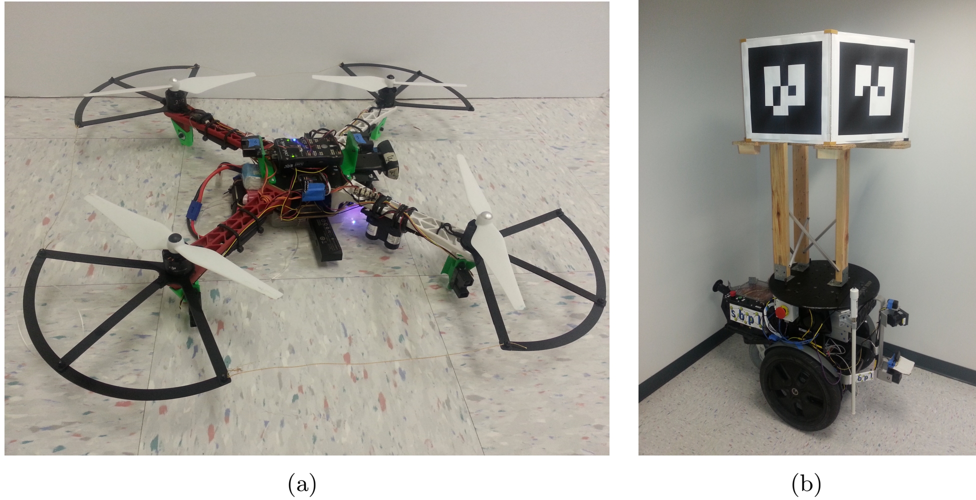
Figure 19: Simple example of SLC Planning with Adaptive Dimensionality for navigation and collaborative localization for a UAV and UGV (a)-(c). Heuristic problems when using full-dimensional planner—no guidance for the UGV (d)

assistance to the UAV. In scenarios when the UAV could navigate completely autonomously to the goal, the UGV did not move at all, thus incurring no movement cost. The challenge of performing full-dimensional search for this planning problem is that no heuristic is available to guide the ground robot, since the locations that the UAV might require localization assistance is not known a priori, and the lower bound on the cost of moving the UGV in this case is 0.

An example of a planning scenario is shown in Fig. 19. Figure 8.19(a) shows the start configurations for both the UAV and UGV in red, and the target goal location for the UAV. It also shows the high-dimensional regions around the UAV start and goal locations. During the adaptive planning phase (Fig. 8.19(b)) the planner only considers the UGV location for states within the high-dimensional regions and computes a path for the UAV

following the walls of the first hallway, making a left turn around the corner, then following the walls of the second hallway until the end. Then, since metric motions are allowed for low-dimensional states without enforcing localization constraints, the planner uses metric motions to move from the end of the second hallway to the goal location. Figure 8.19(b) also shows the high-dimensional tunnel (red shaded areas) constructed around the path found in the adaptive planning phase. Figure 8.19(c) shows the trajectory for both robots computed by a successful completion of the tracking phase. The trajectory for the UAV is almost identical to the one computed in the adaptive planning phase, navigating the UAV to the end of the second hallway using the WallFollowing and GoAroundCorner controllers. However, since localization constraints are enforced during the full-dimensional tracking phase, the UAV is unable to proceed on its own and requires the UGV to move to a location within line-of-sight in order to be able to execute the metric motion required to get to the goal location. Thus, the planner computes a corresponding trajectory for the UGV that allows the UAV to localize and get to the goal. This scenario was fairly easy for the adaptive planner, requiring less than 10 seconds to solve with a final sub-optimality bound of $\epsilon = 1.16$. A full-dimensional planner, on the other hand, is unable to solve the scenario within 180 seconds. The problem that a full-dimensional planner runs into is illustrated in Fig. 8.19(d). The full-dimensional planner is able to quickly find a path to the end of the second hallway by following the heuristic for the UAV guiding it to the goal location. However, once the UAV is no longer able to proceed autonomously and requires the UGV's assistance, the planner gets "stuck" trying to figure out a way of moving the UGV without a heuristic to guide it. As we mentioned previously, it is difficult to find an admissible (optimistic) heuristic that is able to guide the UGV, since in the best case the UAV can get to the goal completely autonomously and the cost of moving the UGV is 0.

In the case of Planning with Adaptive Dimensionality, we can actually leverage information from the path computed during the adaptive planning phase to compute an admissible heuristic for the UGV for the tracking phase. The path of the adaptive planning phase tells us exactly where the UAV begins to use metric motions under the assumption that



(a) (b)

Figure 20: (a) Segway-based unmanned ground vehicle (2 scanning laser range finders, high gain antenna, webcam, high computational capabilities, high-capacity battery). (b) Pixhawk-based quadcopter unmanned aerial vehicle (low-grade IMU, barometric altimeter, laser altimeter, 6 IR range sensors, standard webcam, low-capacity battery, low computational capabilities).

the UGV will be there to help it localize. Thus, we know all the intermediate waypoints that the UGV will have to navigate to in order to help with localization. In other words, we know if and where the UGV will be needed for assistance before we begin our full-dimensional tracking search. For example, in Fig. 19, we know that the final part of the UAV's path (from the end of the second hallway to the goal location) is comprised of metric motions that require the UGV to be within line-of-sight. We compute all UGV locations (x, y) in discrete coordinates that are within line-of-sight of the UAV's location at the end of the second hallway (right before metric motions are needed) and we use those locations as goals for computing a multi-goal heuristic using 2D Dijkstra's grid search accounting for obstacles. Thus, during the tracking phase we are able to use this heuristic to guide the search on how to navigate the UGV to the location where localization assistance is needed.

8.4. Experimental Setup

8.4.1. Robots

For our testing we used two robots: a Pixhawk/DJI-based aerial robot (Fig. 8.20(a)), and a Segway-based ground robot (Fig. 8.20(b)).

The UGV is a relatively large indoor robot with a significant payload capacity and high endurance. With a normal operating load, Melvin is capable of operating for 3+ hours running two independent computer systems, and carrying all required communications infrastructure. The first computer system is used as the low level controller and consists of an Intel Core i3 3.4GHz processor with 8GB of RAM. This system is used for all navigation, sensing, and interfacing with the Segway base. The second system is a general purpose computer equipped with an Intel Xeon processor with 8 physical cores and 16GB of RAM. The planner and plan execution agent both ran on this computer. Due to the limited onboard processing of the UAV, video from the webcam was streamed to the UGV, which ran all the necessary image processing to perform landmark detection. In order to increase the battery life of the UAV, all of the mid-level controllers (the wall following controller, the metric motion controllers, and the landmark controllers) also ran on the UGV computers, and commands were sent to the UAV on-board low-level controller over the network. The UGV is also equipped with two Hokuyo scanning laser sensors mounted on tilt mounts for a full 3-dimensional scanning capability, and a web camera for visual sensing. To assist the UAV with collaborative localization, the UGV has a bundle of six AR markers arranged in a horizontally aligned hexagon, so that the UAV can detect and accurately determine the position and orientation of the UGV from any direction, even in the presence of some low obstacles.

Unlike the UGV, the UAV is very limited in terms of sensing and computing power. The airframe itself is a DJI Flamewheel 450 with a Pixhawk flight control computer and an ODROID XU3 supplemental computer. A standard web camera is used for landmark

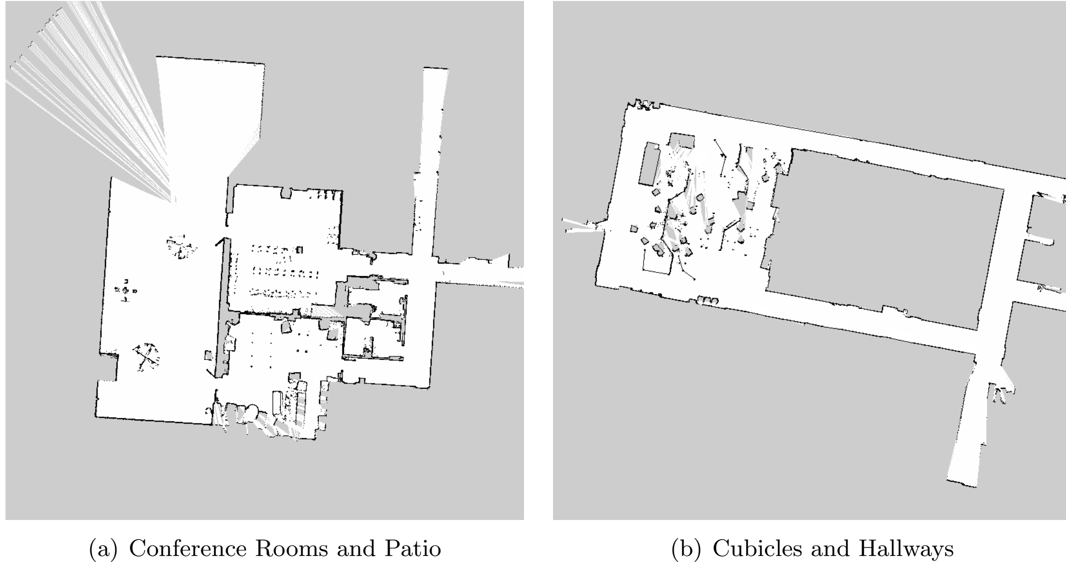
detection, while 6 Sharp IR sensors with 1.5m range are arranged around the perimeter to provide obstacle detection and wall following capabilities. The ODROID captures the images and transmits them to the UGV for processing, then receives the output from the mid-level controllers and translates them into the required format for the Pixhawk to execute.

8.4.2. Environment

Our test environments are meant to replicate a standard indoor office environment (see Fig. 21). We used one area that consisted of two large conference rooms, an outdoor patio area, and a few hallways with small offices. The other test area was comprised of a cluster of cubicles, boxes, equipment, and office furniture in half the area, while the other half is a set of featureless hallways. For our experiments, we restricted the UGV to operate only in the room portions of the environments by placing obstacles at each hallway entrance. The UAV was free to operate throughout the map with different areas performing better with different controllers. For example, since the hallways had no features and the UGV was unable to enter them, the GoToLandmark controllers were not usable (for both static and dynamic landmarks). On the other hand, the crowded, erratically configured cubicle area did not feature many navigable straight walls.

To test our planners performance in real-world scenarios, we randomly selected start and goal points throughout the environment for the UAV and start points only for the UGV. This allowed us to construct plans where the two robots started near each other but allowed the UAV to operate independently if required. The planner would allow the UGV to move as necessary to support the UAV motion to get to the goal.

The cost function used for these experiments was proportional to the time and distance traversed for each motion.



(a) Conference Rooms and Patio

(b) Cubicles and Hallways

Figure 21: Maps of two testing environments.

Algorithm	Planning Time (s) Avg.	Std. Dev.	Min	Max	Num. Iter. Avg.	Num. Expansions Avg.	Path Cost Avg.	Final Eps. Avg.	Success Rate (%) @3min
Adaptive MR *SLC*	20.05	22.00	1.19	91.58	1.59	7348	24401	1.36	100
Full-D $ARA^{\star}$ MR *SLC*	9.37	20.32	0.08	128.04	n/a	1385	23960	1.30	82

Table 7: Experimental results comparing the adaptive multi-robot *SLC* planner and a full-dimensional $ARA^{\star}$ multi-robot *SLC* planner on 50 randomly generated start/goal configuration on map 1 (Fig. 8.21(a)). The results shown are for the 41 trials completed by both planners.

8.5. Analysis of Results

Overall the system was able to generate plans that would not be solvable without using the controller-based motion primitives due to the lack of an adequate localization capability of the UAV operating alone. In addition, the adaptive planner played a key role in making these plans computationally feasible given the high dimensionality of the combined state space. Moreover, using Planning with Adaptive Dimensionality significantly increased the number and difficulty of the scenarios that could be solved when compared to full-dimensional planning. Planning times for 50 randomly generated start-goal pairs on each of the two indoor environments are shown in Table 8.5 and Table 8.5 respectively. The performance of our collaborative localization algorithm (labeled Adaptive MR *SLC*)

Algorithm	Planning Time (s) Avg.	Std. Dev.	Min	Max	Num. Iter. Avg.	Num. Expansions Avg.	Path Cost Avg.	Final Eps. Avg.	Success Rate (%) @3min
Adaptive MR *SLC*	12.74	9.95	1.19	40.00	1.26	2736	36112	1.31	100
Full-D $ARA^\star$ MR *SLC*	12.33	22.23	0.01	88.65	n/a	960	38083	1.38	38

Table 8: Experimental results comparing the adaptive multi-robot *SLC* planner and a full-dimensional $ARA^\star$ multi-robot *SLC* planner on 50 randomly generated start/goal configuration on map 2 (Fig. 8.21(b)). The results shown are for the 19 trials completed by both planners.

is compared against a full-dimensional $ARA^\star$ algorithm using $SL\dot{C}$The results shown in the tables are averaged over the scenarios that both planners were able to solve successfully. The full-dimensional $ARA^\star$ planner was unable to solve the most difficult scenarios within 180s, which was considered a planning failure, whereas the maximum time that our approach took to solve a scenario was 125.32s.

CHAPTER 9 : Application: *PAD* for Manipulation

In this section we discuss extensions to the framework for Planning with Adaptive Dimensionality and experimental results in the domain of planning for a robotic manipulator.

9.1. Using 3D Low-Dimensional Representation

The results reported in this section were originally published in our work (Gochev et al., 2011) presented at the Symposium on Combinatorial Search (SoCS 2011).

9.1.1. Implementation Details

In our initial application of Planning with Adaptive Dimensionality to planning for a robotic manipulator, we chose to use a low-dimensional representation with 3 dimensions—(x,y,z) position of the manipulator's end-effector. Our testing platform was Willow Garage's PR2 robot. We used a 7D/3D adaptive planning, where 3D states represented the arm's end-effector position in 3D, and 7D states represented the full arm configuration. Generally, the full arm configuration on the PR2 robot is given by its seven joint angles (shoulder pan, shoulder lift, shoulder roll, elbow flex, forearm roll, wrist flex, wrist roll) (Fig. 22). Constructing a λ mapping reducing full joint angle configuration to end effector position presented several challenges—namely discretization of the joint angle space could not be easily matched to a discretization of the end-effector position space, and λ and λ^{-1} would have needed to involve expensive FK and IK computations. Instead, we decided to transform the standard 7D robot arm configuration representation to one described in (Tolani et al., 2000), which converts joint angles representations of a 7 DOF arm to 7 DOF representations consisting of the following values: (end-effector x position, end-effector y position, end-effector z position, end-effector roll, end-effector pitch, end-effector yaw, swivel angle). We are going to adopt the following short-hand notation for describing such states: ($ee_{position}$, $ee_{orientation}$, $swivel$), where $ee_{position}$ and $ee_{orientation}$ consist of 3 values each. For more details on the representation, consult (Tolani et al., 2000). This alternative representation

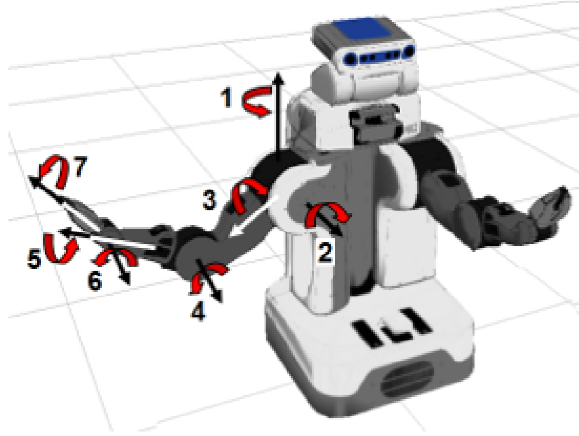


Figure 22: The degrees of freedom of the right arm of a PR2 robot: (1) shoulder pan, (2) shoulder lift, (3) shoulder roll, (4) elbow flex, (5) forearm roll, (6) wrist flex, (7) wrist roll.

of the full arm configuration did not change the dimensionality of the high-dimensional state-space, but provided clean and easy λ and λ^{-1} mappings without any dicretization inconsistencies.

$$\lambda_{7D/3D}(ee_{position}, ee_{orientation}, swivel) = (ee_{position})$$

$$\lambda^{-1}_{7D/3D}(ee_{position}) = \{(ee_{position}, ee_{orientation}, swivel)|$$

$$\text{for all feasible values of } swivel \text{ and } ee_{orientation}\}$$

We used very simple motion primitives for the 7D arm motion planning—namely we allow ± 1 change in each of the seven discretized state-vector values. This produces 14 possible transitions for 7D states and 6 possible transitions for 3D states. Due to the simplicity of the motion primitives, the resulting arm trajectories were not very smooth, but applying simple short-cutting and interpolation produced satisfactory results.

We chose a 2cm 3D grid resolution for the end-effector position, and 16-discretized values for the four angles. This produced a 3D grid of 75x75x75, or roughly 420,000 low-dimensional states, centered at the shoulder joint. In each cell of the grid we have $16^4 \sim 65,000$ possible high-dimensional states, giving us a total of about 28 billion states in the high-dimensional state-space.

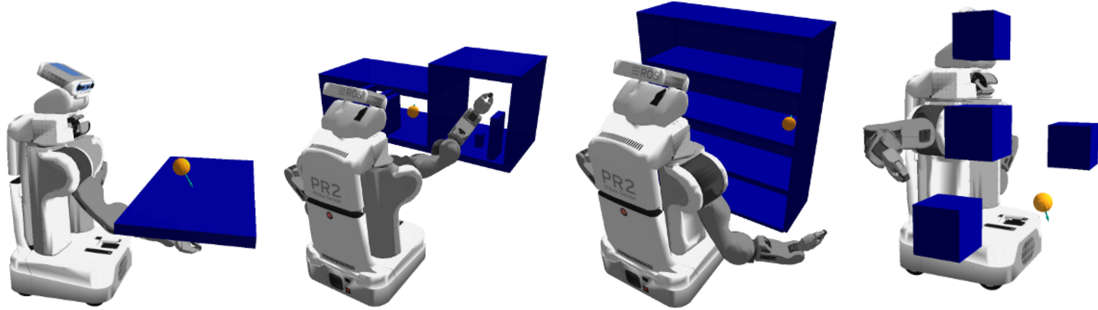


Figure 23: Examples of environments used in simulation: table-top, shelf, bookcase, cuboid obstacles

In this evaluation, the adaptive planner used the non-incremental weighted A* search for both its adaptive planning and tracking phases, planning from scratch at each iteration, as these results precede the development of the Tree-Restoring Weighted A* algorithm.

9.1.2. Experimental Evaluation

We compared the adaptive planning algorithm against a full 7D weighted A* planning algorithm on 35 environments. Environments ranged in degree of difficulty—some required very simple motions to navigate from start to goal, while others were more cluttered and required a set of complex maneuvers to navigate around the obstacles. Some of the types of environments we used included various table tops, bookshelves, and random cuboid obstacles (Fig. 23). Both the adaptive and the 7D algorithm utilized a 3D Dijkstra heuristic to guide the planners to the goal position constraint of the end-effector. We treated the end-effector as a point robot of radius equal to the radius of the largest link of the arm. More sophisticated collision checking and enforcing of joint limits were done on high-dimensional states.

We observed that new sphere radius parameter value of about 10cm allows sufficient arm maneuvering. Also tunnel radius of 10-20cm provides a good balance between the success rate of the tracking phase and the time needed for tracking a path. Since we have a large number of high-dimensional states, we imposed time limits on both the adaptive planning phase and the tracking phase. The time limit we used for the adaptive planning phase

was 120 seconds. If the limit was reached the adaptive planning failed and the algorithm terminated, reporting that no path from start to goal could be found in the given time limit. Due to the number of states inside the tunnel τ even with a small radius, the tracking search might take a long time to find a path through the tunnel or fail. It becomes impractical to wait long for tracking to fail before starting a new iteration, thus, we limited the time for the tracking phase to 20 seconds, allowing us to proceed to the next iteration more quickly.

We also compared our adaptive planner with a sampling-based planner—RRT (LaValle and Kuffner, 2001a)—in the 7DOF robot-arm setting. Although our algorithm could not match the speed of RRT, the consistency of our planner was significantly better—it produced very similar trajectories for similar start/goal configurations within an environment.

We used the following experimental setup for measuring the consistency of the planners. We picked a random table-top environment in which the goal is to maneuver the robotic arm from under to over a table-top. We created 10 scenarios with similar (but not the same) start and goal configurations in that environment. We ran both the adaptive planner and the RRT planner on these scenarios. To measure the consistency between a pair of arm trajectories produced by a planner, we measured the average and maximum distances between end-effector positions along the trajectories and also the average and maximum distances between elbow positions along the trajectories. We calculated the consistency between all (45) pairs of the 10 trajectories produced by our planner and compared it with the consistency between all (45) pairs of the 10 RRT trajectories (we compared with both RRT with post-smoothing and RRT without smoothing; smoothing operations included short-cutting and quintic spline smoothing).

9.1.3. Analysis of Results

We compared the total number of states expanded, number of high-dimensional states expanded, final path cost, and execution time of the adaptive planner compared to the high-dimensional planner, for each of the environments tested. Our results are summarized

Figure 24: Trajectory from Fig. 7 being executed by an actual PR2 robot

Algorithm	Sub-optimality Bound	Time (secs)		# Iterations		# 7D Expands		# 3D Expands		Total Expands		Path Cost		Successful Plans
		mean	std dev	mean	max	mean	std dev	mean	std dev	mean	std dev	mean	std dev	
7D	2.0	147.88	59.93	n/a		769743	1103939	n/a		769743	1103939	63417	18088	12 of 35
adaptive	2.0	14.42	41.95	1.31	6.0	47419	151391	33219	189870	79689	244112	72656	17000	33 of 35
7D	5.0	10.63	15.66	n/a		46529	65586	n/a		46529	65586	73344	19092	31 of 35
adaptive	5.0	5.23	10.45	1.06	2.0	23877	45427	113	40	23986	45439	75400	18839	34 of 35

Table 9: Testing results on 35 environments for 7D motion planning on robotic arm. Adaptive 7D/3D planner vs. 7D weighted A* planner.

in Table 9.

In the case of 7D motion planning on a robotic arm, we noticed results similar to those obtained in the 3D path planning experiments discussed in Section 7.1. For simple environments where the 3D Dijkstra heuristic provides good guidance to the goal and for high ϵ_{plan} values, 7D planning is able to quickly identify a path from start to goal satisfying the sub-optimality constraint, without having to expand many states. However, in cases of complex environments, where the heuristic fails to provide good guidance to the goal, or for lower sub-optimality bounds the adaptive planner performs significantly faster. As seen in Table 9, adaptive planning is able to achieve about two times speedup on the average over seven-dimensional planning for sub-optimality bound of 5.0, and about ten times speedup for sub-optimality bound of 2.0. We ran our algorithm with several sets of parameter values. It is interesting to note that increasing the tracking tunnel radius by a factor of 2 results in about 4 times increase in the average number of 7D states expanded during tracking, and thus, about 4 times increase in the average planning time (19.59s). On the other hand, decreasing the tracking tunnel radius by a factor of 2 results in increased number of algorithm iterations on some of the more cluttered environments, slightly increasing the average planning time (7.66s).

Algorithm	End-effector distance between a pair of trajectories		Elbow distance between a pair of trajectories	
	Avg.	Max.	Avg.	Max.
RRT (smoothed)	8.2 cm	27.5 cm	6.6 cm	18.0 cm
RRT (not smoothed)	9.7 cm	28.8 cm	6.5 cm	17.9 cm
Adaptive	2.5 cm	7.7 cm	2.2 cm	7.9 cm

Table 10: Trajectory consistency comparison between an adaptive planner and an RRT planner for 7-DOF robotic arm motion planning.

Table 10 shows the maximum and average end-effector and elbow distances averaged over the 45 pairwise comparisons of the 10 trajectories for the adaptive and RRT planners. We observed that the key points of the arm we measured (end-effector and elbow) followed much more consistent and predictable trajectories for the paths produced by the adaptive planner than those produced by the RRT planner, even when short-cutting and smoothing were applied.

9.2. Using 4D Low-Dimensional Representation for Manipulators with Independent Wrist Joints

The results reported in this section were originally published in our work (Gochev et al., 2014) presented at the IEEE International Conference on Robotics and Automation (ICRA 2014).

We considered the motion planning problem for robotic manipulators whose joints can be controlled independently from the configuration of the rest of the arm. We developed an extension to the framework for Planning with Adaptive Dimensionality that considerably improves the performance of the algorithm in the context of planning for manipulators with independent wrist joints. Our approach subdivides the original high-dimensional planning problem into two lower-dimensional problems—planning for the main arm joints, and planning for the wrist joints.

Our high-dimensional state-space S^{hd} is defined by the full arm configuration. We consider a lower-dimensional state-space S^{ld}, which only considers the main arm joints, disregarding the degrees of freedom of the wrist. As before, each iteration of our algorithm consists of

two phases: adaptive planning and tracking.

In the planning phase, the current instance of G^{ad} is searched for a path $\pi_{G^{ad}}^{\epsilon_{plan}}(X_S, X_G)$, which is of cost no greater than ϵ_{plan} times the optimal path cost from start to goal in G^{ad}. The planning phase, in effect, solves the first sub-problem of our high-dimensional planning problem, by providing a trajectory for the main arm angles only. The trajectory contains wrist information only for segments of the path that go through high-dimensional regions of G^{ad}.

The tracking phase then needs to solve the second sub-problem—planning for the wrist—and provide a feasible, collision-free fully high-dimensional trajectory for the manipulator from start to goal. In (Gochev et al., 2014) we proposed two approaches for extending and speeding up the tracking phase of the algorithm, which are based on the fact that the wrist degrees of freedom can be controlled independently of the rest of the arm. Thus, the tracking phase now consists of 3 steps (Alg. 9).

9.2.1. Algorithm Extension: Interpolation

Consider an adaptive path $\pi_{ad} = S_1, S_2, ...S_n$ computed by the adaptive planning phase of the current iteration of the algorithm. As G^{ad} is a hybrid graph consisting of both low- and high-dimensional states, π_{ad} is a path that also consists of both low- and high-dimensional states. Also S_1 and S_n are the start X_S and goal X_G states, respectively, and are always high-dimensional states. Without loss of generality, let $\pi_{ld} = S_i, S_{i+1}, ..., S_{i+k}$ be a segment of π_{ad} containing only low-dimensional states, such that the state S_{i-1} preceding the segment, and the state S_{i+k+1} following the segment are high-dimensional states. Thus, we know the desired wrist joint coordinates at the beginning and at the end of π_{ld}, but we do not have information about the wrist joint coordinates throughout π_{ld}. Then we can interpolate between the two desired wrist joint coordinates to compute wrist joint coordinates for each of the low-D states on the segment π_{ld}. If we use such interpolation for every low-D segment along the adaptive path π_{ad}, we can convert the adaptive path to

Algorithm 9 Manipulation Planning with Adaptive Dimensionality for Independent Wrist Joints

1: $G^{ad} = G^{ld}$
2: Add-HD-Region($G^{ad}, \lambda(X_S)$)
3: Add-HD-Region($G^{ad}, \lambda(X_G)$)
4: **loop**
5: ▷ Adaptive Planning Phase
6: search G^{ad} for least-cost path $\pi^*_{ad}(X_S, X_G)$
7: **if** $\pi^*_{ad}(X_S, X_G)$ is not found **then**
8: **return** no path from X_S to X_G exists
9: **end if**
10: ▷ Tracking Phase
11: ▷ 1. Interpolation
12: π_{interp} = ComputeInterpolatedPath(π^*_{ad})
13: **if** interp. success and $c(\pi_{interp}) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{ad})$ **then**
14: **return** π_{interp}
15: **end if**
16: ▷ 2. Planning for the wrist joints
17: π_{HD} = PlanForWristJoints(π^*_{ad})
18: **if** wrist plan success and $c(\pi_{HD}) \leq \epsilon_{\text{track}} \cdot c(\pi^*_{ad})$ **then**
19: **return** π_{HD}
20: **end if**
21: ▷ 3. High-dimensional tracking
22: construct a tunnel τ around $\pi^*_{ad}(X_S, X_G)$
23: search τ for least-cost path $\pi^*_{\tau}(X_S, X_G)$
24: **if** $\pi^*_{\tau}(X_S, X_G)$ is not found **then**
25: find state(s) X_r where to insert new HD region(s)
26: Add-or-Grow-HD-Region(G^{ad}, X_r)
27: **else if** $c(\pi^*_{\tau}(X_S, X_G)) > \epsilon_{\text{track}} \cdot c(\pi^*_{ad}(X_S, X_G))$ **then**
28: find state(s) X_r where to insert new HD region(s)
29: Add-or-Grow-HD-Region(G^{ad}, X_r)
30: **else**
31: **return** $\pi^*_{\tau}(X_S, X_G)$
32: **end if**
33: **end loop**

a fully high-dimensional path π_{interp} (Alg. 9, Line 12). The use of interpolation is feasible only if the degrees of freedom of the wrist can be controlled independently from the other joint angles in the arm, otherwise the wrist trajectory generated by interpolation might not be feasible for execution by the manipulator. If π_{interp} is collision-free and satisfies the joint limit constraints, and moreover, its cost satisfies the sub-optimality bound criteria $c(\pi_{interp}) \leq \epsilon_{track} \cdot c(\pi_{ad})$, then π_{interp} is a valid high-dimensional path that satisfies the desired sub-optimality bound $c(\pi_{interp}) \leq \epsilon_{plan} \cdot \epsilon_{track} \cdot \pi^*_{G^{hd}}$. Thus, we can stop planning and return π_{interp} as a valid plan. If the interpolation step fails to produce a feasible collision-

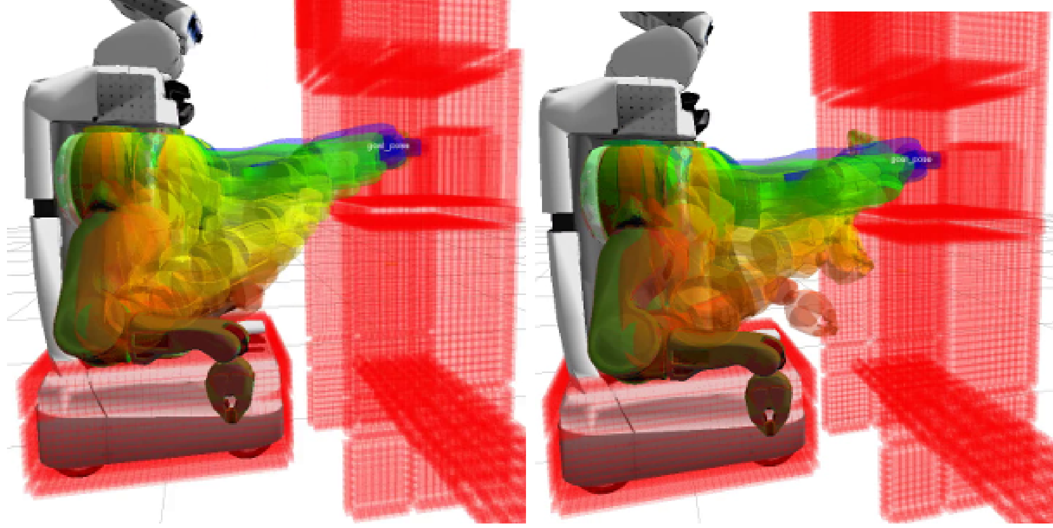


Figure 25: Trajectory computed for the 4 main arm angles during the adaptive planning phase (left) and the resulting 7-DoF trajectory after successful planning for the wrist in the tracking phase (right).

free path, the tracking phase proceeds to step 2. If π_{interp} is invalid, the locations where it violates system constraints, such as collisions with the environment or joint limits, are used as potential locations for introducing new high-dimensional regions into G^{ad}. Since interpolation is very fast, this additional step does not add any significant computational burden per iteration. However, in open environments with few obstacles, this approach is very effective in quickly producing a valid high-dimensional path.

9.2.2. Algorithm Extension: Planning for the Wrist Joints

The fact that the degrees of freedom controlling the wrist are independent from the configuration of the rest of the arm allows us to treat the wrist separately. Thus, the second step of the tracking phase is effectively a search through the wrist configurations over the adaptive path $\pi_{ad} = S_1, S_2, ...S_n$ computed by the planning phase of the current iteration (Fig. 25). Each state in this state-space S^w is defined by a state vector $(wrist, i)$, where $wrist$ is a vector of the wrist joint coordinates, and $i = 1...n$ is a path index. In addition, each such state $X = (wrist, k)$ corresponds to a high-dimensional state $X' = (S_k^{arm}, wrist)$, where S_k^{arm} is the state vector of the main arm joint coordinates of the k-th state S_k in π_{ad},

and $wrist$ is the vector of wrist joint coordinates of the state X. Thus, the state X augments the adaptive state S_k with information about the wrist joint coordinates to produce a fully-defined high-dimensional state X' for the entire arm. Let us denote this mapping by $\Omega_{\pi_{ad}} : S^w \to S^{hd}$. We will omit the subscript π_{ad} if it is understood.

The start state of this search is then $W_S = (S_1^{wrist}, 1)$, where S_1^{wrist} is the vector of wrist joint coordinates of S_1, and the goal state is $W_G = (S_n^{wrist}, n)$, where S_n^{wrist} is the vector of wrist joint coordinates of S_n. Note that S_1 and S_n are the start state X_S and the goal state X_G, respectively, and thus are always high-dimensional, so S_1^{wrist} and S_n^{wrist} are defined. Also, $\Omega(W_S) = X_S$ and $\Omega(W_G) = X_G$.

We allow the following transitions T^w within this state-space:

- We allow the path index to increase by 1, while the wrist joint coordinates remain the same $(wrist, i) \Rightarrow (wrist, i + 1)$ (if $i + 1 \leq n$). This corresponds to moving the arm along the computed path without changing the wrist angles.

- We allow the path index to remain the same, but the wrist joint coordinates to change by using a set of feasible transitions for the wrist $(wrist_1, i) \Rightarrow (wrist_2, i)$. This corresponds to changing the wrist angles only, without changing the configuration of the main arm joints.

- We allow the path index to increase by 1 and also the wrist joint coordinates to change by using a set of feasible transitions for the wrist $(wrist_1, i) \Rightarrow (wrist_2, i + 1)$ (if $(i + 1 \leq n)$. This corresponds to changing the wrist angles while moving along the computed path.

Such transitions are only feasible if the degrees of freedom of the wrist can be controlled independently from the other joint angles in the arm. The cost of each transition

$$X = (wrist_x, i) \Rightarrow Y = (wrist_y, j)$$

is equal to the cost between the two corresponding high-dimensional states $X' = \Omega(X) = (S_i^{arm}, wrist_x)$ and $Y' = \Omega(Y) = (S_j^{arm}, wrist_y)$. We also perform high-dimensional collision-checking on the transition $X' \Rightarrow Y'$ and invalid transitions are discarded by the search.

If we find a path π_{S^w} from start to goal through this graph $G^w = (S^w, T^w)$, we can convert it to a complete high-dimensional path π_{HD} by using the mapping Ω (Alg. 9, Line 17). Then π_{HD} is a valid path from the start arm configuration X_S to the goal arm configuration X_G. If its cost satisfies the sub-optimality bound criteria $c(\pi_{HD}) \leq \epsilon_{track} \cdot c(\pi_{ad})$, then π_{HD} is a valid high-dimensional path that satisfies the desired sub-optimality bound $c(\pi_{HD}) \leq \epsilon_{plan} \cdot \epsilon_{track} \cdot \pi^*_{G^{hd}}$. Thus, we can stop planning and return π_{HD} as a valid path. If π_{S^w} does not exist or $c(\pi_{HD}) > \epsilon_{track} \cdot c(\pi_{ad})$, we proceed to step 3 of the tracking phase. If the search fails, the location of the state with the highest path index value expanded during the search is used as a potential location for introducing a new high-dimensional region into G^{ad}, as it indicates the location farthest along π_{ad} the search was able to reach before it failed, and that location might require high-dimensional planning.

The search through S^w is very constrained and low dimensional. As such, it usually completes very quickly and incurs only a minor computational burden on the tracking phase. Moreover, our results suggest that it is extremely effective in computing feasible high-dimensional paths even in cluttered environments.

Tracking steps 1 and 2 solve the second sub-problem of our original high-dimensional planning problem, augmenting the solution of the first sub-problem with valid coordinates for the wrist joint angles to produce a valid feasible trajectory for the full arm. If steps 1 and 2 fail to produce a valid high-dimensional path from start to goal, we revert to the default method for tracking used by Planning with Adaptive Dimensionality—constructing a tunnel τ around the hybrid path produced by the adaptive planning phase and searching it for a path from start to goal. We have already discussed this method in detail above.

The two extensions of the tracking phase described above preserve all the theoretical prop-

erties of the framework for Planning with Adaptive Dimensionality we have already established, such as completeness and solution cost sub-optimality bounds.

9.2.3. Implementation Details

To validate our extension to Planning with Adaptive Dimensionality, we revisited the problem of motion planning for a 7-DoF robotic arm on the Willow Garage's PR2 platform. The full arm configuration on the PR2 is defined by its seven joint angles: shoulder pan, shoulder lift, upper arm roll, elbow flex, forearm roll, wrist flex, and wrist roll (Fig. 22). Thus, we have a 7-DoF high-dimensional state-space S^{hd}. Low-dimensional states, on the other hand, are defined by only 4 angles: shoulder pan, shoulder lift, upper arm roll, and elbow flex, disregarding the 3 degrees of freedom controlling the wrist. In our implementation, all angles are uniformly discretized with $3°$ resolution within their respective joint limit intervals. Full-arm (and grasped object, if any) collision checking against the environment is performed on 7D states. A more relaxed collision checking is performed on 4D states–only the upper arm and the forearm links are collision-checked against the environment. Since 4D states do not contain information about the end-effector orientation, it is not possible to perform gripper and grasped object collision checking.

Planning for the wrist joint over an adaptive path (step 2 of the tracking phase) is done in a 4-DoF state-space S^{W} defined by 4D state vectors (forearm roll, wrist flex, wrist roll, path index).

In Section 9.1 we discussed our initial implementation for performing 7D/3D planning with adaptive dimensionality for the arm of a PR2 robot, where 3D states represented the end-effector position in (x, y, z). In contrast, in this algorithm extension we choose to do the planning with adaptive dimensionality in 7D/4D. Firstly, this allows us to speed up the tracking phase, as described in sections 9.2.1 and 9.2.2. Second, the four dimensions we select for the low-dimensional states determine the positions and orientations of the two largest links of the arm–the upper arm and the forearm. This allows for much more

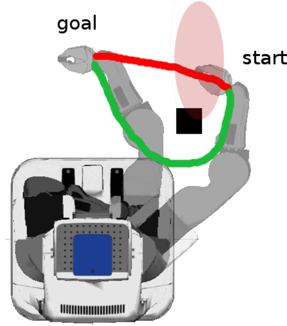


Figure 26: PR2 planning an arm motion around a thin tall obstacle. Black box: tall obstacle, red: heuristic shortest path, green: feasible end-effector path, shaded region: heuristic function local minimum.

accurate collision checking for low-D states, as the positions of the two largest links of the arm are known for low-D states. Thus, the adaptive planning phases produces trajectories that are more likely to be tracked successfully. For example, when using 3D end-effector (x, y, z) low-D states, the planning phase will produce a low-D end-effector path similar to the one shown in red in Fig. 26, which will be impossible for the tracking phase to follow and a new high-D region will be introduced behind the obstacle. By using 4D main arm joint angles as low-D states, on the other hand, the planning phase search will produce a low-D path similar to the one shown in green in Fig. 26, which will be more likely to be tracked successfully without additional iterations being necessary.

A desired 6-DoF cartesian pose was used to define the end-effector goal. Note that, due to the redundancy in the manipulator, a 6-DoF cartesian pose corresponds to multiple goal states in the 7-DoF state-space of the arm.

We use a lattice-based approach (Pivtoraiko and Kelly, 2005) to construct the transitions we use in our graph. Thus, each transition represents a feasible path from one state to another. The graph is constructed dynamically as the graph search progresses, as the size of the state-space is prohibitively large to pre-compute the entire graph. Similar to (Cohen et al., 2010), each of our high-dimensional transitions is a 7-DoF vector of joint velocities for each of the joints in the arm, and a 4-DoF vector for low-dimensional states. We use

a very simple set of fixed transitions, allowing only one joint angle to change with each transition. For each joint, we have 2 short transitions allowing ± 1 unit of discretization change in the joint angle value, and 2 long transitions allowing ± 2 units of discretization change in the joint angle value. Thus, we have a total of 28 possible transitions for each high-dimensional state, and 16 possible transitions for low-dimensional states. Similarly, when planning for the wrist joint over an adaptive path (step 2 of the tracking phase) we use transitions allowing ± 1 unit of discretization change in the joint angle value for each of the three wrist joints. These transitions were selected for the sake of simplicity. However, more complex transitions that operate on several joints simultaneously can be used by the planner.

We take the same approach as in (Cohen et al., 2011) for computing dynamic transitions. For any high-dimensional state S whose end-effector position is within a fixed distance threshold of the goal position, we try to compute a dynamic transition using inverse kinematics. The inverse kinematics solver is seeded with the joint angles $angles_S$ of the state S and is asked to compute joint angles $angles_{IK}$ that satisfy the goal position and orientation of the end-effector (i.e. the 6-DoF cartesian goal pose). If the kinematics solver is able to compute joint angles $angles_{IK}$ satisfying the goal constraints, and the interpolated trajectory from $angles_S$ to $angles_{IK}$ is collision-free and obeys joint limit constraints, then this trajectory (from $angles_S$ to $angles_{IK}$) is used as a transition from S to the goal state defined by $angles_{IK}$.

For any high-dimensional state S whose end-effector position matches exactly the goal position, we use an analytical solver to compute the values for the forearm roll, wrist flex, and wrist roll angles, that would satisfy the goal orientation constraints, while maintaining the same values for the other 4 joint angles (Cohen et al., 2011). If the transition from S to the desired values for forearm roll, wrist flex, and wrist roll is collision-free and obeys joint limit constraints, it is used as a transition from S to the goal state.

To compute the heuristic function, we discretize the environment into 3D voxels and we

use a 3D Dijkstra's search accounting for obstacles to find the least cost paths for the end-effector from every voxel to the goal voxel (corresponding to the (x, y, z) position of the cartesian goal pose). We use a highly optimized implementation of 3D Dijkstra's search, which is able to very quickly compute the heuristic. This heuristic is very helpful in guiding the search around the obstacles in the environment and towards the cartesian goal position. Figure 26 shows an example where the 3D Dijkstra's search heuristic has a pronounced local minimum (shaded area behind the black obstacle). Our approach is quite robust with respect to such local minima as these local minima are overcome by expanding states in the much smaller 4D state-space.

9.2.4. Experimental Evaluation

To measure the performance of the algorithm, we used 524 planning scenarios through various environments. The difficulty level of the environments varied from obstacle-free to highly cluttered. Some examples of environments used in our simulations are shown in Fig. 23—various tables, shelves, bookcases, and cuboid obstacles of random sizes and locations. The difficulty level of the 524 planning scenarios varied based on the environment used in the scenario, and the particular start and goal configurations. In some scenarios the path from start to goal was fairly trivial, where in others, highly complex maneuvering was necessary to reach the goal. We compared our 7D/4D adaptive planner to a number of popular planners available from the Open Motion Planning Library (OMPL) (Şucan et al., 2012)—PRM planner, RRT-Connect planner, and RRT* planner. We also compared against a 4D ARA* planner that only considers the wrist orientation near the goal position, and a 7D ARA* planner that plans in all 7-DoF. Each planner was given a 10-second planning limit to produce a path for each of the 524 environments. If a planner failed to produce a path within the allowed time limit, the scenario was reported as failure. Due to their randomized nature, the sampling-based OMPL planners were given 10 planning trials on each of the scenarios and the observed results were averaged.

We also developed a framework for the use of the 7D/4D adaptive planner on a real PR2.

Figure 27: PR2 robot retrieving an object from a fridge using 7D/4D adaptive manipulation planner.

The framework accepts and serves planning requests to a desired 6-DoF cartesian pose for the end-effector either programmatically or through the use of a GUI to allow for teleoperation of the arm. We observed quick responsiveness from the planner when asked to produce paths through typical household environments. An example scenario of the robot reaching into a refrigerator, grasping an object, and safely retrieving the object from the fridge is shown in Fig. 27. Since grasp selection is outside the scope of this work, a teleoperator selected a suitable grasp pose.

9.2.5. Analysis of Results

As seen in Table 11, the adaptive planner was not able to match the planning times of the sampling-based OMPL planners. However, the achieved average planning time is still quite satisfactory. The 7D ARA* planner demonstrated the worst performance with highest average planning time and only solving just over half of the scenarios. The 4D ARA* planner was able to achieve planning times similar to the OMPL planners, however as it considers the end-effector orientation only in a small region around the goal, it is unable to

Algorithm	Sub-optimality Bound	Planning Time (s)				Successful Plans
		mean	std dev	min	max	
7D/4D Adaptive	100	0.93	0.70	0.03	8.47	87.36%
4D ARA*	100	0.12	0.16	0.01	1.27	71.51%
7D ARA*	100	2.96	2.00	0.01	9.95	52.96%
OMPL PRM	n/a	0.33	2.13	0.01	9.43	83.80%
OMPL RRT-Connect	n/a	0.03	0.03	0.01	0.39	86.62%
OMPL RRT*	n/a	0.36	1.32	0.01	9.73	86.42%

Table 11: Planning time and success rate comparison between arm planners on 524 planning scenarios in simulation. Results for all sampling-based (OMPL) planners are averaged over 10 planning trials on each scenario.

solve planning problems that require the end-effector orientation to change far from the goal position, which explains the relatively low success rate, especially in cluttered environments. The 7D/4D adaptive planner had the highest success rate and was able to solve some of the toughest scenarios within the allowed time limit.

More specifically, the sampling-based methods performed best on the more open scenarios with fewer obstacles, where a feasible path was easy to identify with only a few samples. Our approach was also able to solve such problems quickly, however, the planning times were 2-4 times slower (but still within 1.5 seconds). The benefits of our algorithm were most obvious on the more cluttered scenarios, some of which exhibited narrow solution spaces, which were challenging for the sampling-based methods. The performance of our approach does not suffer in such scenarios and it was able to solve those scenarios quickly. The scenarios that our approach exhibited its worst performance were situations for which the 3D Dijkstra heuristic for the end-effector was leading the search in an unfeasible direction or it exhibited pronounced local minima. This occurred most often on the environments with random cuboid obstacles. For some scenarios the heuristic was "pulling" the end-effector to the far side of a cuboid obstacle, similar to the example shown in Fig. 26. Significantly larger number of state expansions than average were necessary to overcome the heuristic local minimum leading to higher planning times. However, despite the pronounced local minima in such scenarios, the adaptive planner was still able to find a solution within the allowed time, while the 7D ARA* planner failed to do so.

Algorithm	Sub-optimality Bound	Distance Traveled (m)					
		Wrist		Gripper Tip		Elbow	
		mean	std dev	mean	std dev	mean	std dev
7D/4D Adaptive	100	1.30	0.70	1.84	0.72	0.64	0.36
4D ARA*	100	1.64	0.91	1.98	1.22	0.79	0.41
7D ARA*	100	1.44	0.80	1.86	1.20	0.71	0.39
OMPL PRM	n/a	1.75	0.91	2.23	1.12	1.10	0.58
OMPL RRT-Connect	n/a	1.56	0.79	1.93	0.93	1.01	0.52
OMPL RRT*	n/a	1.53	0.77	1.91	0.92	0.98	0.50

Table 12: Path quality comparison between various arm planners: the average distance traveled by the wrist, gripper tip, and elbow for the trajectories computed by each planner on 524 planning scenarios in simulation. The same smoothing was perfomed on the trajectories from all planners. Results for all sampling-based (OMPL) planners are averaged over 10 planning trials on each scenario.

Tracking Step	% of successful tracking phases	Avg. Time (s)
1. Interpolation	49.55%	0.001
2. 4D Orientation Planning	44.59%	0.244
3. HD Tracking	5.86%	1.431

Table 13: Performance and success rate of each of the three steps of the tracking phase of the 7D/4D adaptive planner. Over 94% of successful tracking phases are completed by the much faster interpolation or 4D orientation planning steps. The more computationally expensive HD tracking is performed in less than 6% of the tracking phases.

Table 12 illustrates the average quality of the paths generated by each of the 6 planners. For each computed trajectory, we kept track of the distance traveled by three key points on the arm—the wrist, the elbow, and the gripper tip. As seen in the table, the 7D/4D planner produced the shortest trajectories on average. The OMPL planners had significantly higher distances traveled by the elbow, even after trajectory smoothing. This suggests that many of the trajectories computed by the OMPL planners had unnecessary elbow motions.

Table 13 is the most relevant to the main contribution of this extension to the framework for Planning with Adaptive Dimensionality. It illustrates the number of successful tracking phases completed by each of the three tracking steps as a percentage of all tracking phases performed, and the average time of each tracking step. Nearly half of all tracking phases were solved by simple interpolation, which took a negligible amount of time. The 4-DoF end-effector orientation tracking was successful in nearly 45% of the tracking phases performed, and was much quicker than performing high-dimensional tracking. The more computationally expensive HD tracking had to be performed only in less than 6% of all

tracking phases. Thus, the two algorithm extensions for performing tracking in the context of Planning with Adaptive Dimensionality for robotic arms with independent wrist joints prove to be very effective and significantly improve the performance of the algorithm.

CHAPTER 10 : Application: *PAD* for Mobile Manipulation

10.1. Using a Single Abstraction

The results reported in this section were originally published in our work (Gochev et al., 2012) presented at the IEEE International Conference on Robotics and Automation (ICRA 2012). We present our results of applying the framework for Planning with Adaptive Dimensionality to mobile manipulation planning for Willow Garage's PR2 platform.

10.1.1. Implementation Details

Similarly to our approach to 7D manipulation planning, we used a 11D/3D adaptive planning, where 3D states represented the arm's end-effector (x,y,z) position, and 11D states represented the full arm, torso and base configurations. As discussed in 9.1, we transformed the standard 7-DOF robot arm configuration representation to one described in (Tolani et al., 2000), which converts joint angles representations of a 7-DOF arm to 7-DOF representation consisting of the following values: (end-effector x position, end-effector y position, end-effector z position, end-effector roll, end-effector pitch, end-effector yaw, arm swivel angle) (Fig. 28). This alternative representation of the full arm configuration does not change the dimensionality of the high-dimensional state-space, but provides clean and easy λ and λ^{-1} mappings without any dicretization inconsistencies and not involving expensive forward and inverse kinematics computations.

Using this alternative representation, our 11-dimensional states were represented by the following state vector:

$$(ee_{position}, ee_{orientation}, swivel, base, torso_{height}),$$

where ee_{pos}, ee_{ori} and $base$ consist of 3 values each—end-effector (x,y,z), end-effector (roll,

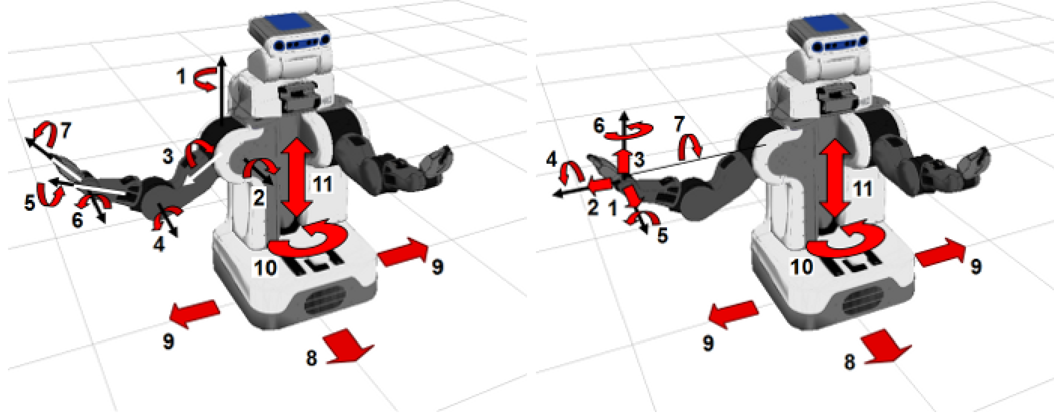


Figure 28: The 11-DOF of the PR2 robot (left) and the alternative 7-DOF arm representation (right) used by our planner. Left: (1: shoulder pan; 2: shoulder lift; 3: upper-arm roll; 4: elbow flex; 5: forearm roll; 6: wrist flex; 7: wrist roll; 8,9,10: base XY position and heading; 11: torso height) Right:(1,2,3: end-effector XYZ position; 4,5,6: end-effector RPY orienation; 7: arm swivel angle; 8,9,10: base XY position and heading; 11: torso height)

pitch, yaw), and base (x, y, heading), respectively. We used the following mapping functions:

$$\lambda(ee_{pos}, ee_{ori}, swivel, base, torso_{ht}) = (ee_{pos})$$

$$\lambda^{-1}(ee_{pos}) = \{(ee_{pos}, ee_{ori}, swivel, base, torso_{ht})|$$

$$\text{for all feasible values of } ee_{ori}, swivel, base \text{ and } torso_{ht}\}$$

The end-effector was allowed to move in a 3m×3m×2m 3D uniform grid with resolution of 2cm, centered around the robot. We used 6cm resolution for the base position and torso height. We uniformly discretized the values for the end-effector roll, pitch and yaw angles, the arm swivel angle, and the base heading angle into 16 on the interval $(-\pi, \pi]$. This discretization produced a 3D grid for the end-effector of size 150×150×100, or roughly 2.25×10^6 low-dimensional states. Our high-dimensional state-space consisted of about 1.8×10^{15} states.

We used very simple motion primitives for graph transitions for the motion planning—namely we allow ± 1 change in each of the eleven discretized state-vector values. This

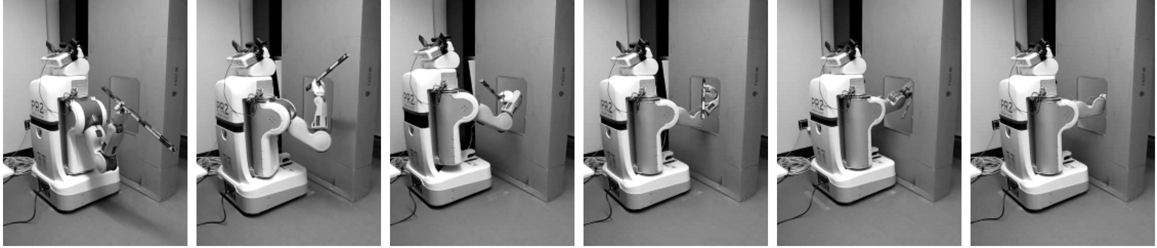


Figure 29: PR2 manipulating 80cm stick trough a 40cm×50cm window.

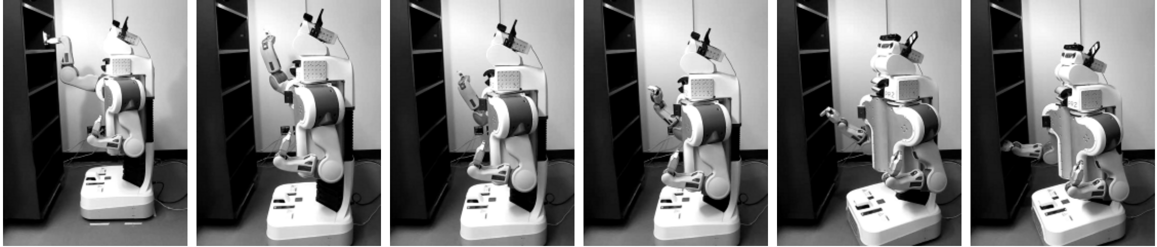


Figure 30: PR2 reaching from a high shelf to a low shelf of a bookcase

produces 22 possible transitions for 11D states and 6 possible transitions for 3D states. The cost of each low-dimensional motion primitive was representative of the distance traveled by the end-effector when executing that primitive. The costs of high-dimensional motion primitives included the distance traveled by the base and penalties for changes in any of the angular values of the state, as well as the distance traveled by the end-effector.

Obstacles in the environment are obtained through a collision map produced by the tilting laser scanner of the PR2. Very basic collision-checking is performed on low-dimensional states, treating them as point-robots and checking them against the obstacle map. Full collision checking is performed on high-dimensional states, checking the full robot configuration (arm, torso, and base) against the obstacle map, while also enforcing joint-limits on the arm configuration. States that are found to be in collision during the search are discarded from G^{ad}. Recall that the path returned by our algorithm consists of only high-dimensional states, on which full collision-checking has been performed, and thus are collision-free.

The graph search algorithm we used for both the adaptive planning and the path tracking phases was Anytime Repairing $A^\star$ ($ARA^\star$) (Likhachev et al., 2003).

10.1.2. Experimental Evaluation

We compared the 11D/3D adaptive planning algorithm, a full 11D weighted $A^{\star}$ planning algorithm, and an 11D bi-directional RRT algorithm (Kuffner and LaValle, 2000) on 30 environments in simulation. Environments ranged in degree of difficulty—some required very simple motions to navigate from start to goal, while others were more cluttered and required a set of complex maneuvers to navigate around the obstacles. Some of the types of environments we used included various table tops, bookshelves, and random cuboid obstacles (Fig. 23). Both the adaptive and the 11D planners utilized a 3D Dijkstra heuristic for the end-effector to guide the planners to the position constraint. We treated the end-effector as a point robot of radius equal to the radius of the smallest link of the arm. Full collision checking and enforcing of joint limits were performed on high-dimensional states.

As with the 7D/3D adaptive manipulation planner (Sec. 9.1), we observed that inserting new spheres of radius of about 10cm allowed sufficient arm maneuvering without introducing too many unnecessary high-dimensional states. Also a tunnel radius of 10-20cm provides a good balance between the success rate of the tracking phase and the time needed for tracking a path at each iteration. Since we have a large number of high-dimensional states, we imposed time limits on both the adaptive planning phase and the tracking phase. The time limit we used for the adaptive planning phase was 180 seconds per iteration. If the limit was reached the adaptive planning failed and the algorithm terminated, reporting that no path from start to goal could be found in the given time limit. We also limited the time for the tracking phase to 20 seconds. All planners were limited to 600 seconds to produce a path.

10.1.3. Analysis of Results

The results we observed are summarized in Table 14. As seen in the table, our adaptive planner was able to achieve much faster planning times than the full 11D planner and was able to successfully produce a solution in all 30 instances. The 11D planner, on the other

Algorithm	Sub-optimality Bound	Time (secs)				# Iterations		# 11D Expands		# 3D Expands		Total Expands		Successful Plans
		mean	std dev	min	max	mean	max	mean	std dev	mean	std dev	mean	std dev	
11D	5.0	340.80	243.57	6.81	600.00	n/a		214K	159K	n/a		214K	159K	17 of 30
adaptive	5.0	19.07	16.65	5.35	55.44	1.30	3	10.2K	12.3K	67.1	30.79	10.2K	12.3K	30 of 30
RRT	n/a	4.15	6.25	0.02	25.69	n/a		n/a		n/a		n/a		600 of 600

Table 14: Experimental results on 30 environments for 11D mobile manipulation planning (full 11D planner vs. adaptive planner vs. bi-directional RRT planner). The deterministic 11D and adaptive planners were run only once on each environment. RRT results are averaged over 20 runs on each of the 30 environments (600 runs total).

Algorithm	20cm stick					50cm stick					80cm stick				
	Time (sec.)				Success	Time (sec.)				Success	Time (sec.)				Success
	mean	std dev	min	max	rate	mean	std dev	min	max	rate	mean	std dev	min	max	rate
RRT (20 runs)	0.981	0.640	0.080	1.990	100%	33.885	36.474	0.320	130.270	100%	751.458	405.371	351.150	1176.66	20%
adaptive ($\epsilon = 5.0$) (1 run)	1.520	0.00	1.520	1.520	100%	3.540	0.00	3.540	3.540	100%	9.890	0.00	9.890	9.890	100%

Table 15: Bi-directional RRT planner (Kuffner and LaValle, 2000) vs. adaptive planner. The task was to manupulate a stick of varying length through a 40cm×50cm window similar to Fig. 29. RRT results are averaged over 20 runs with the same start and goal configurations. A time limit of 20min. was imposed on each run.

hand, was much slower and was unable to find a solution within the allowed limit in 13 of the 30 instances. We observed an average speedup of x17.87. The minimal observed speedup was x1.12 on a very simple scenario that required only about 6 seconds to solve by both planners. In several cases, however, the adaptive planner was able to produce a solution within 5-10 seconds, while the 11D planner ran out of the allowed 10 minutes to produce a plan, giving us very high speedup values of over two orders of magnitude. On average, the sampling-based bi-directional RRT planner significantly outperformed both search-based planners. However, on the more cluttered environments, we observed that the adaptive planner was only marginally outperformed by the RRT planner, and in a few situations the RRT planner was actually slower than the adaptive planner.

We chose the task of manipulating sticks of varying length trough a 40cm×50cm window as a basis for further comparison between our adaptive planner and the RRT planner. This task is challenging for sampling-based planners as it has a narrow solution space. The RRT planner needs to produce sufficently many valid samples within a narrow "tunnel", defined by the window, in order to successfully compute a feasible trajectory. From the results shown Table 15 we observe that increasing the length of the stick being manipulated causes a significant increase in the time required for RRT to produce a solution. On the other

hand, our adaptive planner does not suffer such a significant performance decrease and it is able to significantly outperform the RRT planner on this scenario for large stick length values.

We ran several real-world experiments on an actual PR2 robot using our adaptive planner. The experiments included tasks such as manipulating an 80cm stick through a window of size 40cm×50cm (Fig. 29), and reaching to and from shelves of various heights (Fig. 30). All of the tasks required torso or base movement in order to complete successfully. The planner was able to successfully navigate from start to goal in all instances, and the planning times ranged from 4 to 20 seconds.

10.2. Using Multiple Abstractions

There is a significant drawback to using only a single abstraction in the context of mobile manipulation. The end-effector abstraction we described previously works well in scenarios that do not require extensive base movement in order to solve, but rather, the base movements are used to expand the workspace of the manipulator for a manipulation-focused task. On the other hand, scenarios that do require significant relocation of the base can make use of a different abstraction that focuses on moving the base, rather than the end-effector. Using the single end-effector abstraction in scenarios that require significant base movement tends to produce awkward-looking trajectories that seem like the robot is being pulled by the wrist towards the goal.

Thus, after we developed the extension of the framework for Planning with Adaptive Dimensionality that allows using multiple abstractions, we decided to re-visit the problem of mobile manipulation planning. Again, we used the PR2 robot as our experimental platform and included only the right arm in the planning process. The base and torso provided additional 4 degrees of freedom: base (x, y, heading), and torso height. Similarly to the experimental setup described in Section 10.1, we used the arm representation described in (Tolani et al., 2000), which converts joint angles representation of a 7-DoF arm to 7-DoF representation

consisting of the following values: (end-effector x, y, z position, end-effector roll, pitch, yaw, swivel angle). We used an under-defined goal state, described by a 6-DoF Cartesian pose for the end-effector. Our idea was to use two low-dimensional abstractions—the 3D end-effector abstraction described in Section 10.1, and an abstraction that only considers the base configuration. Thus, the planner could take advantage of the base abstraction in areas of the state-space that are far from the goal and require base movement. The end-effector abstraction could be used in areas that required manipulator movement or reconfiguration, such as near the goal, and in narrow and cluttered areas.

10.2.1. Implementation Details

Our planner used two heuristic functions—a heuristic for the base, and a heuristic for the end-effector. To compute the heuristic function for the end-effector, we discretized the environment into 3D voxels and we used a 3D Dijkstra's search accounting for obstacles to find the least cost paths for the end-effector from every voxel to the goal voxel (corresponding to the (x, y, z) position of the Cartesian goal pose). To compute the base heuristic, we used a discretized 2D grid and a 2D Dijkstra's search accounting for obstacles inflated by the radius of the inscribed circle of the base.

High-dimensional states were represented by 11-DoF vectors of discretized coordinates. All Cartesian coordinates were discretized uniformly with 2.5cm resolution. The base was allowed to move on a 10m$\times$10m range in the X-Y plane, and the end-effector was allowed to move on a 10m$\times$10m$\times$2m range in 3D space. All angular coordinates were discretized uniformly into 16 values on the interval $(-\pi, \pi]$, except the base heading angle, which was discretized uniformly into 32 values on $(-\pi, \pi]$. Simple transitions allowing ± 1 change in each discretized coordinate were used for the high-dimensional state-space. The heuristic function for high-dimensional states was computed as the maximum between the base and the end-effector heuristic functions. Full-body collision checking was performed for high-dimensional states.

The low-dimensional sub-spaces we used were as follows:

- 4-DoF base sub-space, consisting of (x, y, heading) of the base, and the torso height. No information about the right arm is available in those states. Collision checking was performed for the base, torso, head, and the left arm in its fixed configuration. We used very simple transitions allowing for ± 1 change in each of the discretized coordinates. We use the base heuristic for this sub-space, which is very informative and makes searching through the sub-space very efficient.
- 3-DoF end-effector sub-space, consisting of the (x, y, z) coordinates of the end-effector. Very simplified collision checking was performed ensuring that there are no obstacles within 5cm of the end-effector coordinates. The transitions in this sub-space were again ± 1 change in each of the discretized coordinates. We use the end-effector heuristic for this sub-space. Again, the heuristic is very accurate and search in this sub-space is very fast.

The discretization of coordinates in the two low-dimensional sub-spaces was identical to the discretization used in the high-dimensional space.

10.2.2. Experimental Evaluation

We tested the performance of the planner on several typical indoor environments, giving multiple start/goal configurations for each environment. Two of the environments we used are shown in Fig. 31 and Fig. 32. The one in Fig. 31 was manually constructed to represent a kitchen, while the one in Fig. 32 was constructed from real-world sensor data. Fig. 33 and Fig. 34 show the results of an adaptive planning phase and a tracking phase of the algorithm for an example start/goal configuration in the environment shown in Fig. 31.

To compute initial estimates for the scores of each low-dimensional sub-space in the environment, we used a very simple approach, computing the scores based on distance to the nearest obstacle for each cell on a 10m$\times$10m$\times$2m 3D grid of 2.5cm uniform resolution. Cells

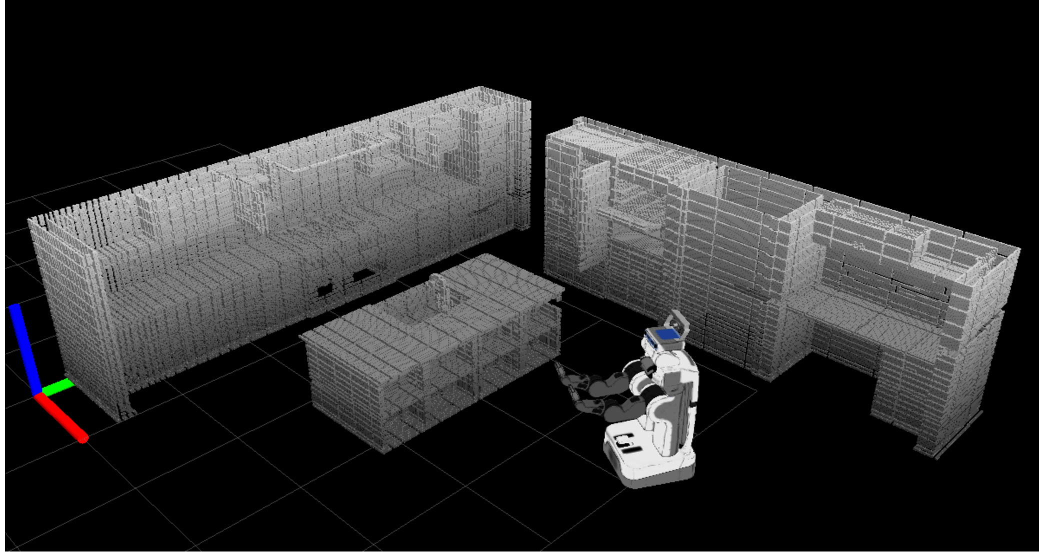


Figure 31: Example indoor environment (kitchen) of size 7m×7m×2m. It was one of the environments we used in our simulations.

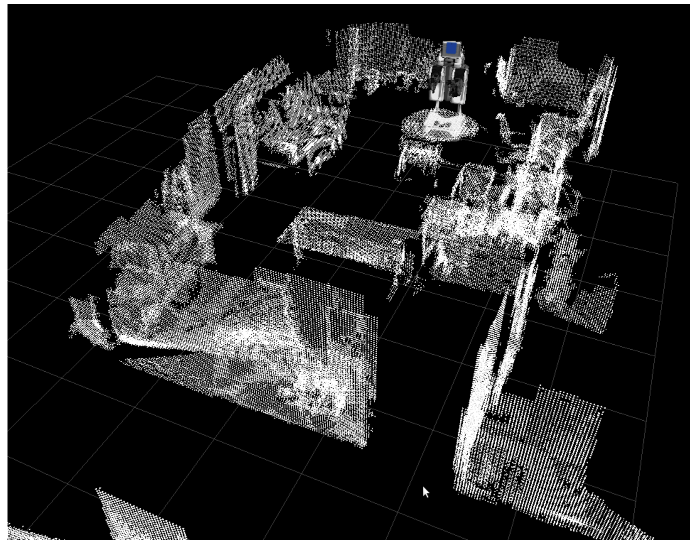


Figure 32: Example indoor environment built from real sensor data.

close to obstacles received higher scores for the 3-DoF end-effector sub-space, whereas cells far from obstacles received higher scores for the 4-DoF base sub-space. Fig. 35 illustrates the allocation of the low-dimensional sub-spaces throughout the environment based on the initial score estimates.

After each tracking phase, we used paths from the search tree generated by the planner

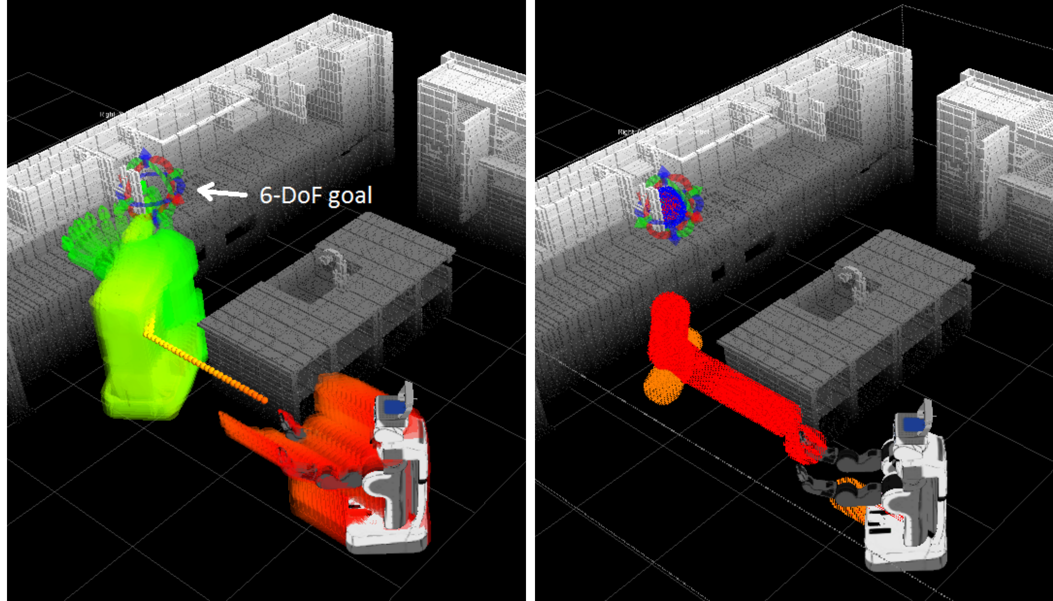


Figure 33: Left: path computed by the adaptive planning phase for the given start/goal configuration. The goal is to reach into an overhead cabinet. The colored states are low-dimensional base states, representing the robot's base, torso, and the fixed left arm configuration. The dots represent low-dimensional end-effector states, representing the end-effector position in (x, y, z). Right: the corresponding tunnel τ constructed for the tracking phase, based on the adaptive path. Orange: tunnel through base sub-spaces. Red: tunnel through end-effector sub-spaces.

during tracking to update the scores for each sub-space in the relevant regions. For all transitions t in the search tree, we update the scores for the base and end-effector sub-spaces in the region where the transition is located, based on how much the base and the end-effector move as a result of the transition. In the future, we would like to further explore alternative approaches for updating the scores for each sub-space and learn from experience or examples what "the best" sub-space is for each region of the environment.

In order to measure the performance of our planner, we ran it in simulation on 4 example indoor environments. On each environment, we had 15 start/goal scenarios with varying degree of difficulty, giving us a total of 60 planning scenarios. We compared the performance of our planner against 3 sampling-based planners available in the Open Motion Planning Library (OMPL)—*PRM*, *RRT*, and *RRT**. All planners used the same collision-checking library. We gave a time limit of 120 seconds to each planner to solve each scenario. If a

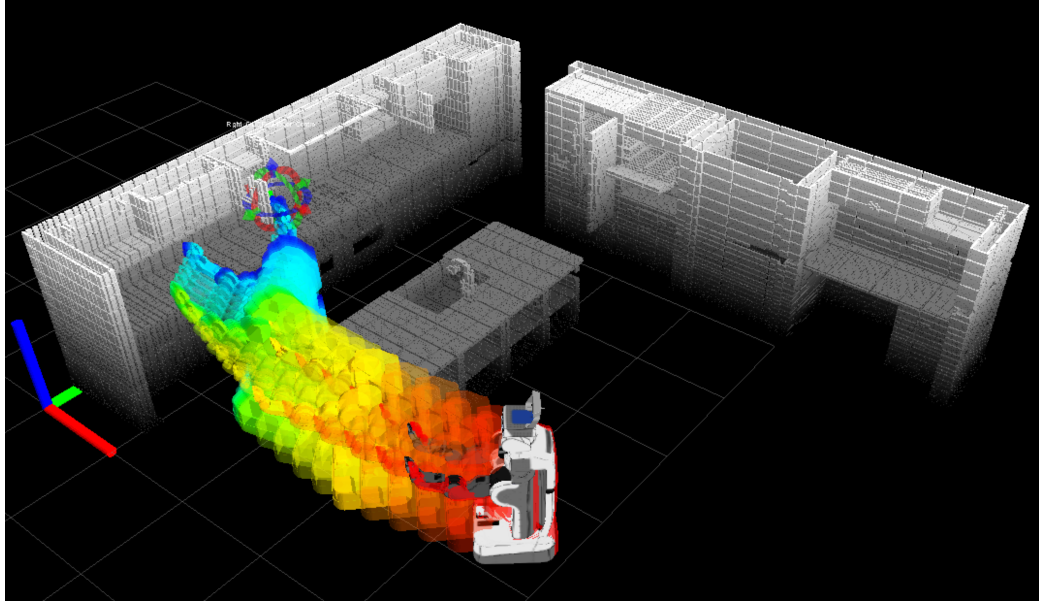


Figure 34: Fully high-dimensional path computed by a successful tracking phase, based on the tunnel constructed around the adaptive path shown in Fig.33.

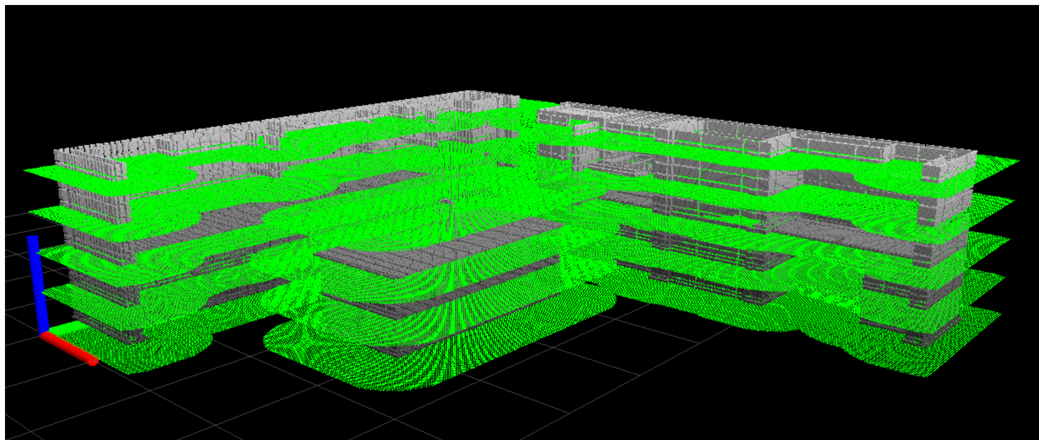


Figure 35: The initial distribution of sub-spaces in the environment shown in Fig. 31. The green regions near obstacles are associated with the end-effector low-dimensional sub-space. All other regions are associated with the base low-dimensional sub-space. Note that only several horizontal slices of the full 3D grid are visualized for clarity.

planner failed to produce a path within the allowed time limit, the scenario was reported as failure. Since our planner is deterministic, we ran it only once on each scenario, whereas the sampling-based planners were run 10 times on each scenario. In some scenarios the path from start to goal was fairly trivial, while in others, highly complex maneuvering was

Algorithm	Sub-optimality Bound	Planning Time (s)				Successful Plans (@120s)
		mean	std dev	min	max	
11-DoF Adaptive	10	32.93	41.40	6.47	108.37	93.33% (56/60)
OMPL *PRM*	n/a	24.33	38.13	0.01	112.52	82.00% (492/600)
OMPL *RRT*	n/a	20.05	29.03	0.01	103.40	86.33% (518/600)
OMPL $RRT^{\star}$	n/a	29.36	46.23	0.04	118.73	84.50% (507/600)

Table 16: Planning time and success rate comparison between mobile manipulation planners on 60 planning scenarios in simulation. Results for all sampling-based (OMPL) planners are averaged over 10 planning trials on each scenario.

necessary to reach the goal. Some scenarios required maneuvering through doorways and narrow gaps between furniture, which were challenging for the sampling-based planners to solve. The results we observed are summarized in Table 16.

10.2.3. Analysis of Results

Our results demonstrate the excellent performance of the sampling-based planners on the more trivial scenarios, solving them almost instantly. On the other hand, the average results suggest that our search-based approach had very comparable performance, while also achieving the best success rate within the allotted time. The sampling-based planners had difficulty solving the more challenging scenarios, which required navigating through narrow gaps, such as doorways. Moreover, they often produced very chaotic trajectories, even after smoothing was applied, which exhibited unnecessary and erratic arm motions while far from the goal state. Our search-based planner, on the other hand, produced very predictable and consistent solutions.

It is also worth noting that our planner was the only one able to solve all 60 scenarios within 180s.

CHAPTER 11 : Application: *PAD* for Humanoid Robot Mobility

In recent years significant research efforts have been directed towards development of humanoid robots. Such robots provide the opportunity to operate in environments that are designed for humans, such as buildings and vehicles, and to be capable of performing the tasks that a human might. The ultimate goal of this research is to produce robots that have human-like agility and versatility. These robots can be used in environments that are dangerous for humans, such as disaster areas and contaminated areas, in order to perform support tasks, such as search-and-rescue, cleanup, repair, and maintenance. There are a large number of challenges that need to be overcome before robots can reach the agility and versatility of humans. One such challenge is planning for such complex robotic systems that have large number of degrees of freedom.

In this chapter, we present our work on applying the framework for Planning with Adaptive Dimensionality (*PAD*) to the domain of planning for humanoid robot mobility. Our goal was to develop a planner that allows the robot to navigate through typical household or industrial environments, which might exhibit challenging features, such as uneven terrain, stairs, and ladders.

11.1. Domain Background and Related Work

Motion planning for legged and humanoid robots is a challenging domain due to the large number of degrees of freedom of the system and the complex balancing and collision avoidance constraints needed to ensure that system stability can be maintained throughout the planned motions. These constraints severely restrict the set of allowable configurations. Efficient methods for maintaining dynamic balance for biped robots have been developed (Raibert, 1986; Vukobratovc, 1990; Pratt and Pratt, 1999; Kagami et al., 2001; Yin et al., 2007a,b). These methods, however, do not consider obstacle avoidance. Kuffner et al. (Kuffner et al., 2001) developed a sampling-based *RRT* planning framework that combines statically-stable motion planning with ***AutoBalancer*** (Kagami et al., 2001) to transform

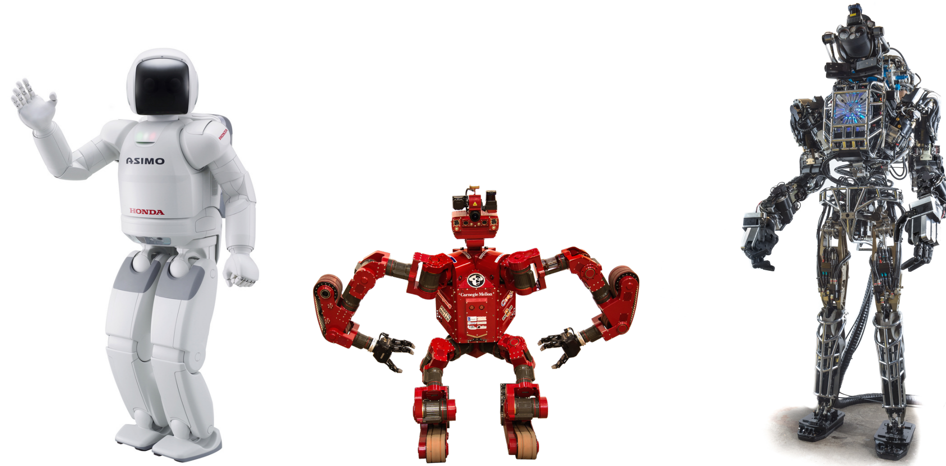


Figure 36: Some of the humanoid robots that are currently under development: Honda's Asimo, NREC's Chimp, Boston Dynamics' Atlas.

statically-stable trajectories into dynamically-stable ones. The approach works well for computing short motions, but does not allow for the location of the supporting foot (or feet in the case of dual-leg support) to change during the planned motion.

Robotics systems which need to reason about contacts with the environment, such as legged robots, need to be able to distinguish between different contact *modes*. For each *mode* the set of contacts is fixed. When an existing contact is broken or a new contact is made, the *mode* of the system changes. To plan for such systems, one must find both a discrete sequence of *mode* switches and single-*mode* motions to achieve them. This process is usually referred to as multi-modal planning (Hauser, 2008). In his dissertation (Hauser, 2008), Hauser explores the problem of legged robot mobility and proposes an algorithm for multi-modal probabilistic roadmap planning (*MM-PRM*). The approach decouples the planning process into planning for the sequence of *mode* switches and single-*mode* motion planning to compute trajectories that transition from one *mode* to another. For example, bipedal locomotion is decomposed into footstep planning, which computes a sequence of contacts (*modes*), and full-body motion planning to compute a trajectory from one footstep configuration (*mode*) to another. *MM-PRM* is able to solve planning queries for a wide range of legged robots—the 6-legged ATHLETE robot, the 4-legged Capuchin robot, and the

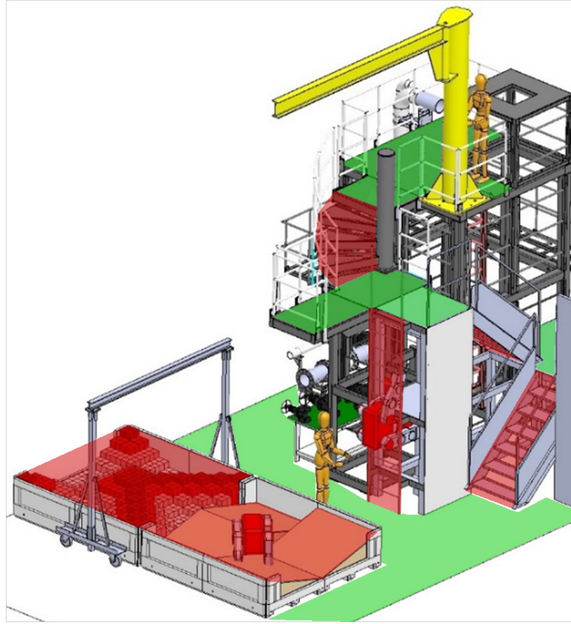


Figure 37: Using a high-level footstep following controller for bipedal navigation. Green: areas of the environment where the high-level controller is safe to use (flat ground); red: areas where the high-level controller is not available (rough terrain, stairs, ladders, etc.).

humanoid HRP-2.

When developing complex robotic systems, such as legged robots, researchers often develop higher-level controllers that allow the system to execute simple tasks or behaviors based on simple inputs. For instance, controllers that maintain balance while minimizing the distance to a desired robot configuration, or even controllers that achieve basic locomotion based on a desired direction of movement and speed (Yin et al., 2007a,b). These controllers are usually carefully tuned operate with precision on the specific robotic system. The motion planning framework can leverage such built-in system capabilities in order to improve its performance. Rather than having to always produce full-body trajectories, the planner can produce the simplified inputs required by a given high-level controller to achieve a desired action or task, provided that the high-level controller is safe to use in the particular part of the state-space. For parts of the state-space that do not allow for the safe utilization of high-level controllers, the planner can revert to full-body planning.

Consider, for example, the problem of navigating a bipedal robot through a complex environment, such as the one in Fig. 37. Let us assume that the robot has the built-in capability to robustly follow a sequence of footstep locations defined by

$$\langle (x, y, \text{heading})_{\text{left}}, (x, y, \text{heading})_{\text{right}} \rangle ,$$

which conform to a set of pre-defined constraints $\mathcal{Q}$ (e.g. consecutive footsteps are not too far from each other, the change in heading between consecutive footsteps does not exceed a threshold, the sequence maintains a minimum safe distance from obstacles, etc.). Thus, if the planner is made aware of this built-in capability, by specifying the state-space on which the high-level controller operates (expected input to the controller), a set of transitions available in this state-space, the capability constraints $\mathcal{Q}$, and the parts of the environment that the capability is available, then the planner can make use of this simplified state-space and perform footstep planning for large areas of the environment (Fig. 37), thus limiting the use of full-body planning to challenging areas of the environment.

The fact that such high-level controllers are available for many robotics systems prompted us to develop an extension to the framework for Planning with Adaptive Dimensionality that leverages them.

11.2. Algorithm Extension

In this section we formalize our definition of high-level controllers and their application to the *PAD* framework.

We assume that every high-level controller $\mathcal{C}$ expects a sequence of state-vectors $X = \langle x_1, ..., x_i \rangle$ as input, where the components of the state-vector represent a sub-space $\mathcal{S}$ of the full-dimensional robot configuration space. We also assume that an input sequence $X_1, ..., X_k$ needs to satisfy a set of constraints $\mathcal{Q} = (\mathcal{Q}_s(\cdot), \mathcal{Q}_t(\cdot))$, where $\mathcal{Q}_s : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ is a function that checks whether a state $X \in \mathcal{S}$ satisfies the controller constraints, and $\mathcal{Q}_t : \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ is a function that checks if a transition between two states,

$X_i, X_j \in \mathcal{S}$ satisfies the controller constraints. Thus, we define a high-level controller as

Definition 11.1 *A high-level controller $\mathcal{C}$ is a tuple $\mathcal{C} = (\mathcal{S}, \mathcal{Q}, \Phi)$, where $\mathcal{S}$ is a sub-space of the full-dimensional state-space $\mathcal{S}^{hd}$ ($\mathcal{S} \subseteq \mathcal{S}^{hd}$), and $\mathcal{Q} = (\mathcal{Q}_s(\cdot), \mathcal{Q}_t(\cdot))$ is a pair of functions*

$$\mathcal{Q}_s : \mathcal{S} \rightarrow \{true, false\}$$

$$\mathcal{Q}_t : \mathcal{S} \times \mathcal{S} \rightarrow \{true, false\},$$

which define constraints for states $X \in \mathcal{S}$ and transitions $T = (X_i, X_j)$, ($X_i, X_j \in \mathcal{S}$).

Given sequence $\pi = (X_1, \cdots, X_k), X_{1\ldots k} \in \mathcal{S}$ s.t.

$$\mathcal{Q}_s(X_i) = true \; \forall X_i \in \pi$$

$$\mathcal{Q}_t(X_i, X_{i+1}) = true \; \forall X_i \in \pi, i = 1 \ldots k-1,$$

$\Phi(\pi) = \pi_{hd} = (X'_1, \cdots, X'_n)$, $X'_i \in \mathcal{S}^{hd}$, is a valid path in $\mathcal{S}^{hd}$. Moreover, image$(X'_1, \mathcal{S}) = X_1$ and image$(X'_n, \mathcal{S}) = X_k$.

In other words, a high-level controller is able to generate a high-dimensional path $\pi_{hd} \in \mathcal{S}^{hd}$ from a lower-dimensional path $\pi_{ld} \in \mathcal{S}^{ld}$, provided that π_{ld} satisfies the controller constraints $\mathcal{Q}$. For the purposes of planning, we treat the function Φ encoding the controller logic as a black box. The only assumption that we make is that constraints $\mathcal{Q}$ are defined so that any state transition that satisfies them is safe for the control logic Φ to execute.

For every high-level controller $\mathcal{C} = (\mathcal{S}, \mathcal{Q}, \Phi)$, we can construct a state-abstraction $\mathcal{A} = (\lambda, \lambda^{-1}, G = (\mathcal{S}, \mathcal{T}), c)$ that operates in the sub-space $\mathcal{S}$, by providing a set of transitions $\mathcal{T}$, which satisfy $\mathcal{Q}_t$, and a cost function $c : \mathcal{T} \rightarrow \mathbb{R}^+$, which needs to satisfy 4.1. The projection function λ and λ^{-1} are implicitly defined by the choice of $\mathcal{S}$. For example, if the high-level controller operates on footstep locations, then λ has to compute the footstep locations for a given full-dimensional robot state. Thus, we can construct state-abstractions to be used

by the *PAD* framework based on the available high-level controllers for the system.

So far we have assumed that path segments through low-dimensional state-abstractions in the hybrid graph of the *PAD* framework are not directly executable by the robot, and thus, we required the *PAD* planner to compute corresponding full-dimensional paths in the tracking phase of the algorithm. However, path segments through state-abstractions constructed from high-level controllers can be executed by the robot provided that these path segments satisfy the corresponding high-level controller requirements. Thus, we can simplify and expedite the search performed during the tracking phase of the algorithm by not requiring full-dimensional tracking. This can be extremely beneficial for very high-dimensional planning problems, such as motion planning for humanoid robots, as even the tunnel-constrained full-dimensional search during the tracking phase can be prohibitively expensive. We illustrated the benefits of using alternative methods for performing the tracking phase in the context of planning for manipulation in Chapter 9, Section 9.2.

The main purpose of the tracking phase of the *PAD* framework is to find executable path π_{hd} that corresponds to given path π_{ad} through our hybrid graph G^{AD}. Leveraging the fact that high-level controllers provide us with abstractions that produce executable paths, we can modify the tracking phase of the algorithm for Planning with Adaptive Dimensionality. Rather than constructing a high-dimensional tunnel τ_{hd} around π_{ad}, we can construct a hybrid tunnel τ_e that consists of low-dimensional states from executable abstractions only and high-dimensional states for the areas where executable low-dimensional abstractions are not available. τ_e is a hybrid graph corresponding to the high-dimensional graph τ_{hd}, as G^{AD} is a hybrid graph corresponding to G^{HD}, and we can perform a search on it. If a path π_e through τ_e is found, then it consists fully of executable transitions (some low-dimensional and some high-dimensional). Each high-dimensional transition in π_e is assumed to be inherently executable. Each low-dimensional transition in π_e is associated with a corresponding high-level controller capable of executing it.

Definition 11.2 *We construct a hybrid tunnel* $\tau_e = (S^{\tau_e}, T^{\tau_e})$ *of width* w *consisting of*

executable states and transitions around a hybrid path π_{AD} as follows:

$$\forall X \in \pi_{AD}$$

1. We find the abstraction $\mathcal{A}$ to which X belongs.

$$\exists \text{ abstraction } \mathcal{A} = \left(\lambda_{\mathcal{A}}, \lambda_{\mathcal{A}}^{-1}, G^{\mathcal{A}} = \left(S^{\mathcal{A}}, T^{\mathcal{A}}\right), c_{\mathcal{A}}\right) \text{ s.t.} X \in S^{\mathcal{A}}$$

2. We find the next finest executable abstraction $\mathcal{B} \succeq \mathcal{A}$ in the abstraction hierarchy, which is a finer abstraction than $\mathcal{A}$. Note that $\mathcal{B}$ might be the full-dimensional abstraction, which is the finest available abstraction and is executable. If $\mathcal{A}$ is executable, then $\mathcal{B} = \mathcal{A}$. Let $\mathcal{B} = \left(\lambda_{\mathcal{B}}, \lambda_{\mathcal{B}}^{-1}, G^{\mathcal{B}} = \left(S^{\mathcal{B}}, T^{\mathcal{B}}\right), c_{\mathcal{B}}\right)$

3. We include all states from $S^{\mathcal{B}}$ in S^{τ_e} whose projections to $S^{\mathcal{A}}$ are within distance w of $X \in S^{\mathcal{A}}$

$$\forall X' \in S^{\mathcal{B}}, X' \in S^{\tau_e} \text{ iff } dist(\phi(X'), X) \leq w,$$

where ϕ is a function projecting from the finer abstraction $\mathcal{B}$ to the coarser abstraction $\mathcal{A}$, and thus is a many-to-one mapping, similar to the λ projection functions. When $\mathcal{A}$ is executable, and thus $\mathcal{B} = \mathcal{A}$, ϕ is the identity mapping.

4. We construct T^{τ_e} in the usual manner for constructing edges for hybrid graphs, including all transitions that connect states from the same abstraction, and using the corresponding λ and λ^{-1} projection functions to construct transitions that connect states from different abstractions. (Fig. 8).

Note that when no executable state representations are available, other than the full-dimensional one, the above definition becomes identical to constructing a full-dimensional tunnel around the hybrid path π_{AD}, since in step 2 of the definition $\mathcal{B}$ will always be the full-dimensional abstraction. Then step 3 becomes equivalent to

$$\forall X' \in S^{HD}, X' \in S^{\tau_e} \text{ iff } dist(\lambda_{\mathcal{A}}(X'), X) \leq w.$$

Thus S^{τ_e} will contain only full-dimensional states whose projections are within w of some state $X \in \pi_{AD}$, and consequently, T^{τ_e} will only contain full-dimensional transitions, which coincides with our original definition of high-dimensional tunnel (Def. 4.5). Also note that when the hybrid path π_{AD} consists only of full-dimensional states, then Def. 11.2 again coincides with Def. 4.5 and τ_e will be a full-dimensional tunnel. Then the tracking phase will still be able to track the full-dimensional path π_{AD} exactly ($\pi_{\tau_e} = \pi_{AD}$, $c(\pi_{\tau_e}) = c(\pi_{AD})$), and thus the algorithm will terminate returning π_{τ_e} as a valid solution which satisfies the desired cost sub-optimality bound.

Algorithm 10 Planning with Adaptive Dimensionality Using Executable Abstractions

```
 1: G^AD = Initialize-Regions((G_1^LD, ρ_1) ... (G_n^LD, ρ_n))
 2: Add-HD-Region(G^AD, X_S)
 3: Add-HD-Region(G^AD, X_G)
 4: loop
 5:                                                   ▷ Adaptive Planning Phase
 6:     search G^AD for least-cost path π*_AD(X_S, X_G)
 7:     if π*_AD(X_S, X_G) is not found then
 8:         return no path from X_S to X_G exists
 9:     end if
10:                                 ▷ Tracking Phase using executable hybrid tunnel
11:     construct an executable hybrid tunnel τ_e around π*_AD(X_S, X_G) by Def. 11.2
12:     search τ_e for least-cost path π*_τe(X_S, X_G)
13:     if π*_τe(X_S, X_G) is not found then
14:         find state(s) X_r where to introduce next-best abstraction
15:         Introduce-Next-Best-Abstraction(G^AD, X_r)
16:     else if c(π*_τe(X_S, X_G)) > ε_track · c(π*_AD(X_S, X_G)) then
17:         find state(s) X_r where to introduce next-best abstraction
18:         Introduce-Next-Best-Abstraction(G^AD, X_r)
19:     else
20:         return π*_τe(X_S, X_G)
21:     end if
22: end loop
 1: function INTRODUCE-NEXT-BEST-ABSTRACTION(G^AD, X_r)
 2:     ρ = Get-Region-For-State(X_r)
 3:     α = Get-Abstraction-For-Region(ρ)
 4:     β = Get-Next-Abstraction-For-Region(ρ, α)
 5:     if ∃β then
 6:         Set-Abstraction-For-Region(ρ, β)
 7:         Update-Hybrid-Graph-Region(G^AD, ρ, β)
 8:     else
 9:         Add-or-Grow-HD-Region(X_r)
10:     end if
11: end function
```

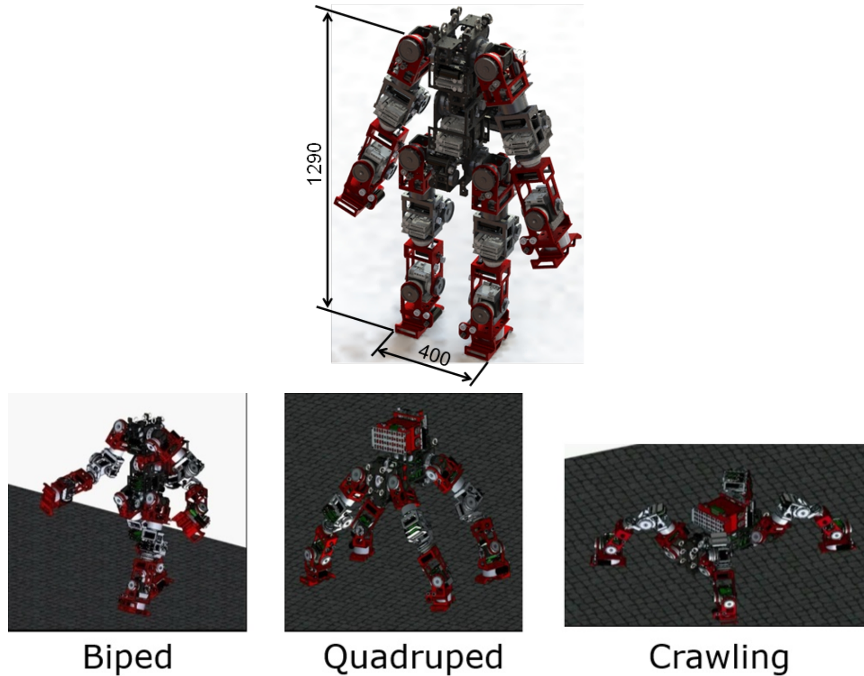


Figure 38: The Yamabiko humanoid robot, which is being developed by Waseda University and Mitsubishi Heavy Industries.

Algorithm 10 gives the pseudo-code for the algorithm extension allowing the PAD framework to perform faster tracking when using executable low-dimensional state-abstractions constructed from corresponding high-level controllers.

11.3. Implementation Details

The Yamabiko humanoid robot (Fig. 38) was selected as the development platform for our algorithm in a joint project with Mitsubishi Heavy Industries and Waseda University. Our goal in this project was to develop a single planning framework that is able to produce plans to navigate the robot to a desired goal location in a complex environment (Fig. 37), while being able to reason about the various locomotion modes and capabilities of the system. The framework for Planning with Adaptive Dimensionality using multiple abstractions provided the backbone of the planning framework.

Figure 39 illustrates the abstraction hierarchy we have developed so far for the Yamabiko

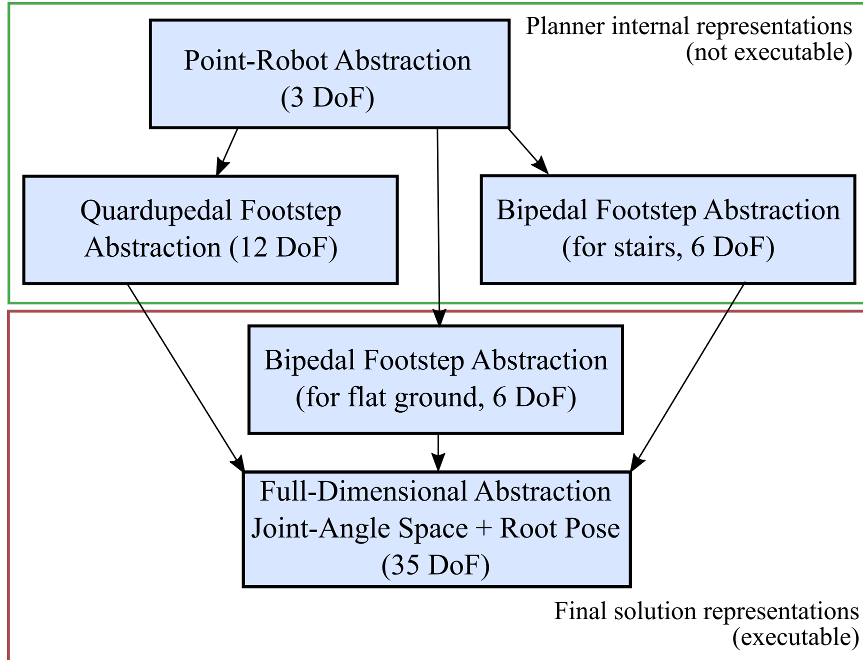


Figure 39: The abstraction hierarchy developed so far for the Yamabiko robot.

robot. The abstractions are separated into two general categories—planner internal representations, and representations that can be executed by a high-level controller on the system (and thus, can be used in the final solution of the planner). Two high-level controllers are currently available for the system: a full-body joint-angle controller, and a bipedal footstep following controller that can follow sequences of footsteps, provided that the sequence of footsteps conforms to the constraints specified by the controller (e.g. distance between consecutive footsteps, distance to the nearest obstacle, etc.). The planner ensures that the constraints specified by a high-level controller are satisfied for all segments of the final solution that are intended to be executed by that high-level controller. There is currently no high-level controller capable of following footstep sequences when navigating a staircase, so the planner uses the bipedal footstep abstraction for stairs as an internal non-executable low-dimensional state abstraction during the adaptive planning phase. Then, during the tracking phase, any states from the bipedal footstep abstraction for stairs will be tracked using the full-dimensional abstraction (the next finest executable abstraction in the hierarchy) to produce a sequence of full-body joint-angle actions to be executed by the full-body

controller. A quadrupedal footstep following controller is currently being developed, which will allow the quadrupedal footstep abstraction to be treated as executable in the future. However, currently it is considered non-executable and the tracking phase of the PAD framework uses the full-dimensional abstraction to track states from the quadrupedal footstep abstraction and produce corresponding full-body motions.

11.3.1. Point-Robot Abstraction

The point-robot abstraction is the most abstract representation we use and represents the $\langle x, y, z \rangle$ position of the root of the robot. It is used to establish the general traversability of the environment and mainly serves as a heuristic to the less abstract representations. The representation resembles traditional 3D Dijkstra's search of the free space of the environment, however it enforces support constraints, making sure that there is a suitable support surface within reach of the root position (ground, platform, ladder, or staircase step) and that the robot does not float through the environment. The state-space is a 3D grid with 26-connected grid transitions. A simplified cylindrical collision model representing the the inscribed circle of the robot is used for collision checking. The height of the cylinder is determined from the z coordinate of the root position and the z coordinate of the nearest support point.

11.3.2. Bipedal Footstep Abstraction for Flat Terrain

The bipedal footstep abstraction for flat terrain represents only the position and heading of each foot and an indicator variable that keeps track of which foot can be moved next: $\langle \text{limbID}, (x, y, \theta)_{\text{left}}, (x, y, \theta)_{\text{right}} \rangle$, where $\text{limbID} \in \{\text{left}, \text{right}, \text{either}\}$. The z coordinate for each foot is also stored as part of the state, but it is not a free variable—it is calculated from the z coordinate of the current support surface. The transitions in this state space are calculated from a set of motion primitives, which satisfy the constraints of the high-level footstep following controller. The collision model used for this representation ensures that the foot locations are in contact with a support surface and are not colliding with obstacles.

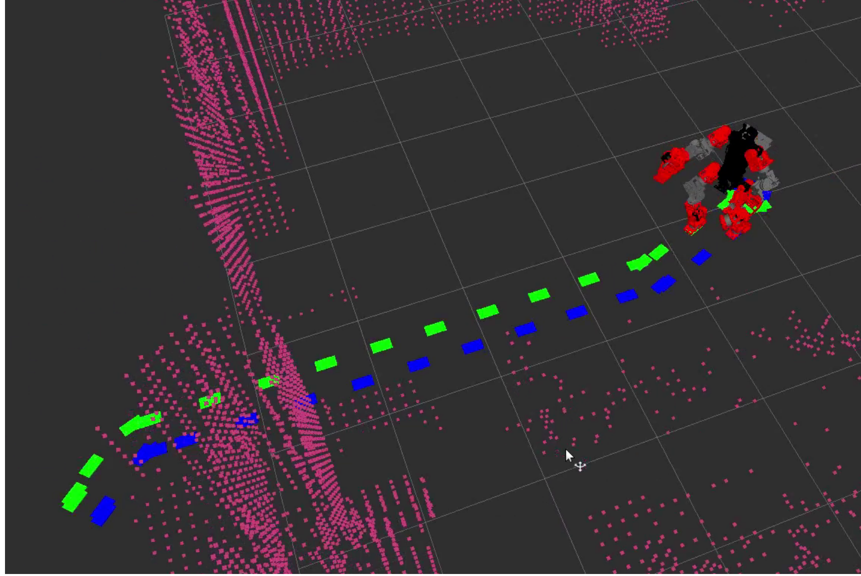


Figure 40: Example plan showing the output of the bipedal footstep abstraction for flat terrain. Green cuboids: right foot locations, blue cuboids: left foot locations.

The feet are modeled as cuboids. Figure 40 shows an example plan consisting entirely of states from the bipedal footstep abstraction for flat terrain.

11.3.3. Bipedal Footstep Abstraction for Stairs

The bipedal footstep abstraction for stairs is very similar to the bipedal footstep abstraction for flat terrain. Currently the two abstractions use the same state representation: $\langle \text{limbID}, (x, y, \theta)_{\text{left}}, (x, y, \theta)_{\text{right}} \rangle$, where limbID $\in \{\text{left}, \text{right}, \text{either}\}$. The difference is that the bipedal footstep abstraction for stairs has a different transition set constructed from a different set of motion primitives. Additionally, footstep transitions that fall outside the current support surface (the current step of the staircase), are checked against the previous and next steps of the staircase (with adjusted z coordinates). Thus, transitions allow for changes in the support surface, and thus, footsteps to go up and down a staircase. Figure 41 shows an example of a plan consisting of states from the bipedal footstep abstraction for stairs. Ultimately, we would like to extend this representation to allow reasoning for possible hand contacts as well, thus allowing additional support when climbing the stairs (by holding onto a railing, for example).

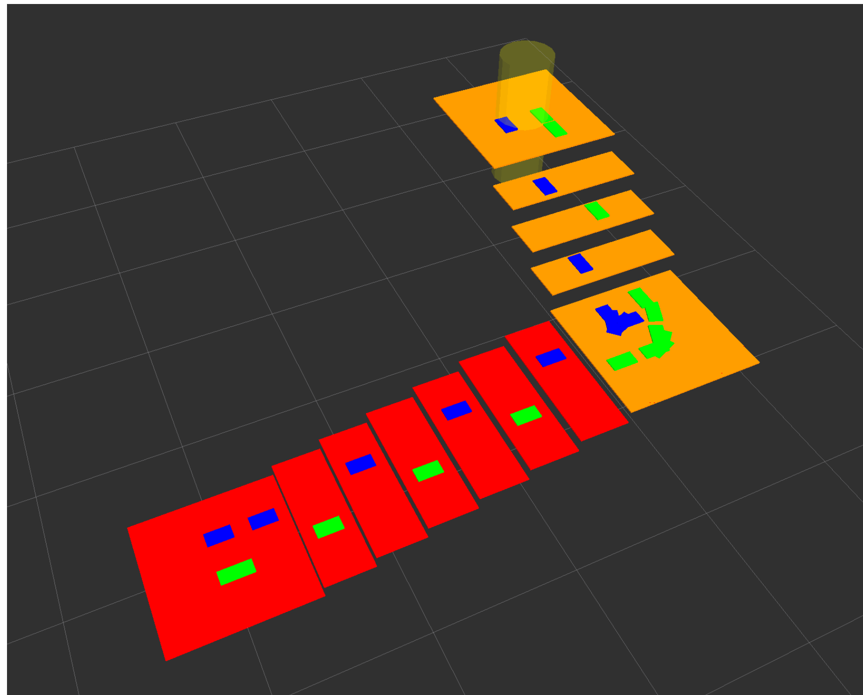


Figure 41: Example plan showing the output of the bipedal footstep abstraction for stairs. Green cuboids: right foot locations, blue cuboids: left foot locations. The robot starts at the bottom of the stairs and navigates to the top.

11.3.4. Quadrupedal Footstep Abstraction

The quadrupedal footstep abstraction extends the bipedal footstep abstraction by additionally keeping track of the position and heading of the hand contacts (treating them as footprints):

$$\langle \text{limbID}, (x, y, \theta)_{\text{left hand}}, (x, y, \theta)_{\text{right hand}}, (x, y, \theta)_{\text{left foot}}, (x, y, \theta)_{\text{right foot}} \rangle$$

$$\text{limbID} \in \{\text{left hand}, \text{right hand}, \text{left foot}, \text{right foot}, \text{either hand}, \text{either foot}, \\ \text{either left}, \text{either right}, \text{either}\}$$

The transitions used in this abstraction move one limb at a time, thus maintaining three limbs in contact with support surfaces. This significantly increases the size of the support polygon of the robot and therefore its stability when compared to the bipedal locomotion mode. The design requirements of the project specify that quadrupedal locomotion mode is the preferred method for locomotion through rough terrain, where maintaining stability can be a challenge. In addition, the transitions for this abstraction allow the user to define a preferred quadrupedal gait—the repeating sequence in which limbs should be moved (left hand $\Rightarrow$ right foot $\Rightarrow$ right hand $\Rightarrow$ left foot, for example). Note that specifying a preferred gait does not impose strict constraints on the planner to always move the limbs in that sequence, but rather penalizes transitions that deviate from the preferred sequence. Thus, the planner is encouraged to use transitions which conform to the preferred gait, but it is still allowed to deviate from it when necessary. We noticed that when the preferred gait is not specified, the planner produced trajectories that “seemed erratic” because they did not exhibit an obvious regular repeating pattern, which we innately expect from legged locomotion.

11.3.5. The Full-Dimensional Representation

As mentioned previously, the full-dimensional representation of the robot consists of 35 degrees of freedom—6D pose of the root, 4 limbs with 7 joints each, and 1 joint for the torso. However, planning in this space is very challenging as the state-space does not allow explicit reasoning about selecting suitable contacts for the limbs in order to achieve locomotion. Thus, we augmented the state space with additional 7 degrees of freedom: $\langle \text{limbID}, \text{pose6D}_{\text{contact}} \rangle$ encoding the next contact target for the current state. This is similar to the approach taken in *MM-PRM* (Hauser, 2008), where the planner is explicitly allowed to search for and select suitable contacts for the limbs (i.e. *mode* changes) in order to achieve locomotion. Selecting a target contact allows for a more focused search over the remaining 35 degrees of freedom, to drive the system to the selected target contact. Once the current target contact is satisfied, the planner selects the next target contact. The difference in our approach is that we do not separate the search for contacts and the search for trajectories to achieve those contacts into two separate searches. Rather, the contact search and selection and the search for motions to achieve the selected contact is combined into a single state-space representation that reasons for both concurrently. The next target contact is selected similarly to how next footstep locations are generated in the bipedal and quadrupedal representations. Thus, when a state satisfies the current target contact, we generate successor states for each of the possible next target contacts based on the available bipedal and quadrupedal motion primitives. Thus, the total number of degrees of freedom in the full-dimensional representation is 42.

Another important challenge in planning for locomotion for humanoid robots is ensuring the stability of the system, especially when breaking or making contacts. For the scope of this project, dynamic stability was not required and the planner needed to produce statically stable trajectories. To ensure static stability, the planner enforced that for every state, the center of mass of the robot is fully supported by the support polygon defined by the current set of contacts. Moreover, before a contact can be broken, the planner ensured that

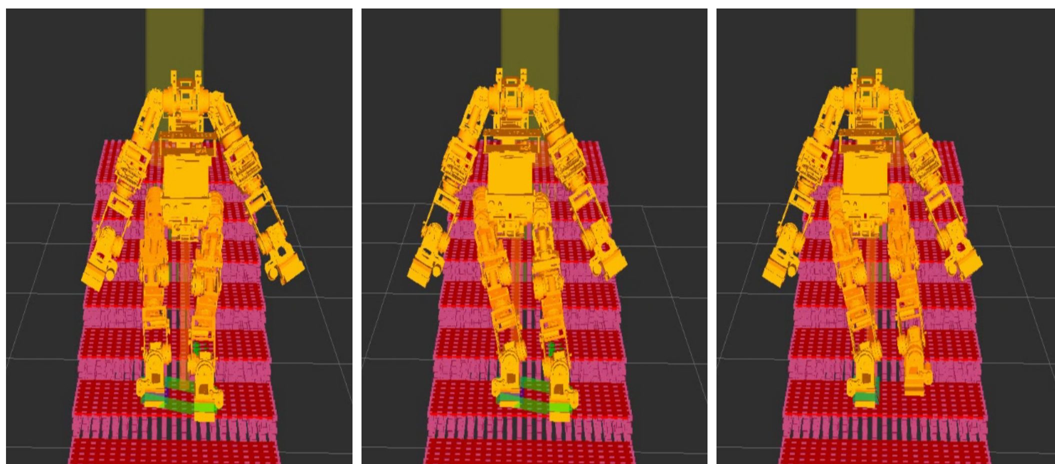


Figure 42: Maintaining static stability: the planner ensures that the projection of the center of mass (translucent vertical orange line) falls inside the support polygon (green polygon). The robot needs to shift its center of mass over the stance foot before it can break the contact of the stepping foot.

the center of mass of the robot is fully supported by the support polygon defined by the remaining contacts. For example, in the bipedal case, before the robot can make a step it needs to shift its center of mass over the support polygon of the stance leg. Once the center of mass is fully supported by the stance leg, then the robot is allowed to proceed with breaking the contact of the stepping leg and move it towards the selected target contact location (Fig. 42). This approach for checking stability only works for resting contacts on horizontal surfaces, assuming that the force of gravity is counteracted by vertical normal forces at the contact locations. We are currently developing a more complex system for balance checking, which can handle arbitrary contacts by reasoning about the necessary forces at the contact locations in order to maintain balance and if those forces can be achieved within the torque limits.

When more than one limb is in contact, every pair of contacts forms a closed kinematic chain, which constrains the available motions of the joints in the chain. Consider, for example, the case when the robot is standing with both feet in contact with the ground. In order for the robot to shift its center of mass over one of the feet to be able to take a step, it needs to perform a synchronized motion with both legs, such that the body moves

in the desired direction, while the feet maintain their current contacts (Fig. 42 left and middle). To achieve such motions, we dynamically generate transitions with the use of inverse kinematics. For example, let's consider trying to shift the root of the robot from it's current pose R to a new pose R'. The current root pose R and the current joint configurations J make a set of contacts $\mathcal{C}$ with the environment (represented as 6D poses of the links that make the contacts). We would like to compute new set of joint configurations J' such that the new root pose R' and the new joint configurations J' maintain the same set of contact poses $\mathcal{C}$. We use inverse kinematics from the new root pose R' to the set of contacts $\mathcal{C}$ to look for feasible joint configurations J' of the contacting limbs that satisfy those contacts (i.e. the poses of the contacting links remain the same). If we find such joint configurations J', then we check if the transition from (R, J) to (R', J') is collision-free and that it satisfies joint limit and balance constraints. If so, it is used as a valid transition in the state-space, allowing us to shift the root position, while maintaining the contacts with the environment.

On the other hand, joints that are not part of closed kinematic chains (i.e. joints of limbs that are not currently used for support) can be moved freely within their respective joint limit ranges. For such joints, we allow transitions based on motion primitives, which rotate the joint by a specified angle. Currently we use only 2 motion primitives for each joint, which rotate the joint by $\pm 5°$ respectively.

11.3.6. Planning Framework Design

The planning framework is designed to allow easy incorporation of new abstract representations regardless of their internal state-spaces and transition sets. Every abstraction is registered with the planning framework and gets assigned a unique ID. Each abstraction also specifies where it fits in the abstraction hierarchy by registering with their corresponding parent abstractions. The planning framework is completely agnostic to the internal state representations that the abstractions use. The planner represents every state in the graph through the following data structure:

Abstract State:

- *stateID* - an integer uniquely identifying every state.
- *abstractionID* - an integer identifying the abstraction that state belongs to.
- *stateData* - the state data specific to the abstraction that state belongs to. Our particular implementation uses a void pointer to the memory location where the data is stored. The planner has no knowledge of how to interpret the state data. Only the abstraction that generated the state knows exactly what is stored in the state data and how to interpret it.

Each abstraction needs to define the following interface functions that the planner uses to communicate with it

- **GetSuccesors**(*stateID*, out array *successorIDs*, out array *transitionCosts*) - the function that generates the successor states of a given state, defined by its *stateID*. It returns a set of successor states (defined by *stateID*'s) and the corresponding transition costs (integers). This function defines the transition set and cost function of the abstraction.
- **GetPredecessors**(*stateID*, out array *predecessorIDs*, out array *transitionCosts*) - the function that generates the predecessor states of a given state, defined by its *stateID*. It returns a set of predecessor states (defined by *stateID*'s) and the corresponding transition costs (integers). This function defines the transition set and cost function of the abstraction.
- **GetGoalHeuristic**(*stateID*) - the function returns the heuristic value (estimated cost to goal) of the given state. It is used for forward graph searches.
- **GetStartHeuristic**(*stateID*) - the function returns the heuristic value (estimated cost to start) of the given state. It is used for backward graph searches.

- **ProjectFromFullD**(*fullDstateID*, out array *lowDProjectionIDs*) - the function defines the λ function for the particular abstraction and produces a set of projected low-dimensional states from a given full-dimensional state.

- **ProjectToFullD**(*lowDstateID*, out array *fullDProjectionIDs*) - the function defines the λ^{-1} function for the particular abstraction and produces a set of projected full-dimensional states from a given low-dimensional state.

Note that the projection functions assume that each abstraction has knowledge of what the full-dimensional state-space is and how to interpret the state data of full-dimensional states. Thus, each abstraction is intended to work with a specific full-dimensional representation. If the full-dimensional representation for the system is changed, the projection functions of each abstraction have to be modified accordingly to work with the new full-dimensional representation. Additionally, note that the projection functions can take advantage of domain-specific knowledge and assumptions in order to optimize the projection process, which can be quite expensive, especially when projecting to the full-dimensional space. For example, when projecting from bipedal footstep abstraction to the full-dimensional state-space, we ***do not*** generate all possible full-dimensional configurations that have the desired footstep locations, since the vast majority of those configuration will be undesirable. Instead, we generate relatively few full-dimensional configurations that are close to a nominal standing pose which satisfy the desired footstep locations. This domain-specific strategy significantly improves the performance of the projection functions.

11.4. Experimental Evaluation

We have begun initial evaluation of the planning framework on simplified test environments designed to test specific components of the framework (examples in Fig. 40-43). Figure 43 shows an example plan which combines the two bipedal abstractions—bipedal footstep abstraction for flat terrain and bipedal footstep abstraction for stairs. The adaptive planning phase in the example required about 60 seconds of planning to identify the sequence of

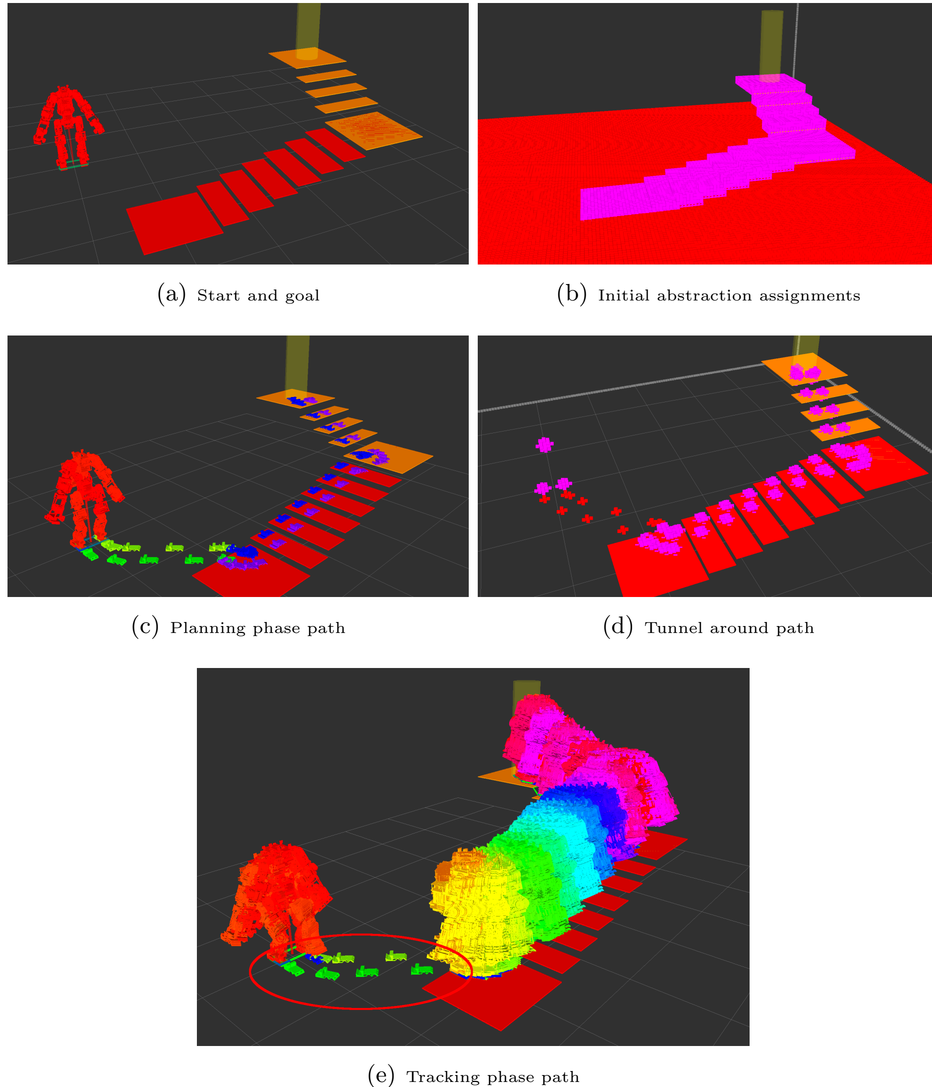
footsteps required to reach the goal and expanded about 1 million low-dimensional states in the process. The tracking phase performed planning using the two corresponding executable abstractions—bipedal footstep abstraction for flat terrain and the full-dimensional abstraction (for climbing up the stairs)—and required about 290 seconds. Thus, the planner required about 6 minutes to produce the final trajectory. These results should be considered preliminary and treated as a proof-of-concept, as there is room for further improvement and optimization of the code. For example, expansion of full-dimensional states is quite expensive—planner currently achieves about 10-15 full-dimensional expansions per second. The computations requiring the most significant amount of time during full-dimensional expansions are the calls to the inverse kinematics solver ($\approx$ 50-100 ms per expansion), which is over an order of magnitude more time consuming than the next most significant computation—transition validation and collision-checking ($\approx$ 5 ms per expansion). We are currently using a generic inverse kinematics solver, but we believe that the performance of the inverse kinematics computations can be improved significantly by using a solver specifically designed to work for the limbs of the robot.

The ultimate goal of the project is to also incorporate interleaving of planning and execution, which we discussed in Chapter 7.3, in order to achieve low robot idle times while waiting for the planner to produce a complete trajectory to the goal. Additionally, as the robot navigates through the environment, the planner will also need to incorporate new sensor data between planning iterations in order to be able to react to any changes in the environment and ensure that planning is done with respect to the most current environment data.

11.5. Analysis of Results

Despite the fact that the application of Planning with Adaptive Dimensionality for humanoid mobility is in its early stages of development, we believe that our initial results provide a compelling argument for the performance benefits that can be achieved by using the PAD framework combining multiple low-dimensional representations when planning for

robotic systems with a large number of degrees of freedom. We have illustrated that the framework can solve difficult planning problems in reasonable time, while providing the strong theoretical guarantees associated with search-based planners—completeness with respect to the graph representing the planning problem and bounds on solution cost sub-optimality. We have outlined several ways that can further improve the performance of the planner, which will be incorporated into the planning framework as the project moves forward. Our goal is to perform rigorous performance evaluation of the planning framework and compare it with alternative approaches, such as *MM-PRM* (Hauser, 2008), in a future publication.



(a) Start and goal

(b) Initial abstraction assignments

(c) Planning phase path

(d) Tunnel around path

(e) Tracking phase path

Figure 43: Example of Planning with Adaptive Dimensionality for humanoid mobility using multiple abstractions for different areas of the state-space. (a) The starting robot configuration and the goal location at the top of the stairs. (b) Initial abstraction assignments in the environment: red—bipedal footprint abstraction for flat ground, pink—bipedal footprint abstraction for stairs. (c) The path produced by the adaptive planning phase showing the two different bipedal abstractions used in different colors. (d) Tunnel constraining the footstep locations to near the ones selected by the adaptive planning phase: red—tracking using executable bipedal footprint abstraction, pink—tracking using executable full-D abstraction. (e) Final solution after successful tracking phase highlighting the use the the executable bipedal footstep following abstraction.

CHAPTER 12 : Conclusion

While many planning problems are seemingly high-dimensional, they often exhibit low-dimensional structures that describe the problem well throughout most of the state-space. Based on this observation, we have developed the framework for Planning with Adaptive Dimensionality, which makes effective use of such low-dimensional representations in order to reduce the size and complexity of the state-space, resulting in faster planning times and lower memory requirements, while providing strong guarantees about the feasibility of the resulting path, completeness with respect to the high-dimensional graph representing the problem, and bounds on solution cost sub-optimality. The *PAD* framework provides a general principled way of combining planning for a hierarchy of multiple different state abstractions in a single planning process, which avoids expensive high-dimensional planning through areas of the state-space which do not require it. Moreover, the *PAD* framework effectively identifies the areas which do require high-dimensional planning in order to ensure the feasibility of the final solution and its cost sub-optimality bound. The use abstraction hierarchies allows the framework to capture the different capabilities of the system and use suitable state abstractions for different areas of the state-space.

The *PAD* framework also allows for domain-specific extensions aimed to improve its performance. We have developed a number of such extensions, such as the tree-restoring weighted A* incremental graph search algorithm, which is able to minimize redundant computation between iterations while efficiently handling changes in the search graph. We have also developed a method that allows the algorithm to interleave planning and execution, thus reducing the system's idle time while waiting for the planner to produce a complete solution. We have presented several extensions aimed at improving the tracking phase of the algorithm by introducing more efficient ways to construct a high-dimensional path from the hybrid path produced by the planning phase.

We have demonstrated the applicability of our framework in several different domains—

planning for single robot navigation, multi-robot collaborative navigation, manipulation and mobile manipulation, and planning for humanoid mobility. We have experimentally validated the advantages of our framework over popular alternative approaches in these domains. Our experimental results illustrate that the *PAD* framework outperforms high-dimensional planners, especially on difficult planning problems, while also providing more consistent solutions for similar planning problems than sampling-based alternatives.

We have identified a number of topics relevant to Planning with Adaptive Dimensionality that present interesting areas for future research—how can suitable abstractions be computed automatically, or how to automatically decide which of the available abstractions is best suited for a certain area of the state-space, for instance. We have presented our insight into trying to address those questions, however, we consider them as open questions which require further exploration.

In conclusion, in this work we have presented a general principled approach for using state abstractions to deal with the curse of dimensionality for high-dimensional planning problems—Planning with Adaptive Dimensionality.

APPENDIX A : Planning with Adaptive Dimensionality Proofs

Theorem A.I *The cost of a least-cost path from X_S to X_G, $\pi^*_{ad}(X_S, X_G)$, in G^{ad} is a lower bound on the cost of a least-cost path from X_S to X_G, $\pi^*_{hd}(X_S, X_G)$, in G^{hd}.*

$$c(\pi^*_{ad}(X_S, X_G)) \leq c(\pi^*_{hd}(X_S, X_G))$$

Proof We will construct a proof by contradiction. Let's assume to the contrary

$$c(\pi^*_{ad}(X_S, X_G)) > c(\pi^*_{hd}(X_S, X_G))$$

Let π_{hd} be any path in the high-dimensional graph $G_{hd} = (S_{hd}, T_{hd})$. We can represent π_{hd} in terms of the sequence of states that it visits $\pi_{hd} = (s_1, s_2, ..., s_n)$, where $s_i \in S_{hd}$ and $(s_i, s_{i+1}) \in T_{hd}$. Consider the following projection function λ_{ad} that projects high-dimensional paths $\pi_{hd} = (s_1, s_2, ..., s_n)$ onto the hybrid graph $G_{ad} = (S_{ad}, T_{ad})$:

$$\forall s_i \in \pi_{hd} : \lambda_{ad}(s_i) = \begin{cases} s_i & \text{if } s_i \in S_{ad} \\ \lambda(s_i) & \text{if } s_i \notin S_{ad} \end{cases} \tag{A.1}$$

Thus, λ_{ad} projects high-dimensional states onto themselves if they fall inside a high-dimensional region of G_{ad}, or to their low-dimensional projections otherwise.

Without loss of generality, let $\pi^*_{hd}(X_S, X_G) = (s_1, s_2, ..., s_n)$. Consider the projection $\pi' = \lambda_{ad}(\pi^*_{hd}(X_S, X_G))$ onto the hybrid graph G^{ad}. Recall that $X_S = s_1$ and $X_G = s_n$ are always in high-dimensional regions of G_{ad}. Then

$$\pi' = \lambda_{ad}(\pi^*_{hd}(X_S, X_G)) = \lambda_{ad}((s_1, s_2, ..., s_n)) = (s_1, ...s_k, \lambda(s_{k+1}), ...\lambda(s_m), s_{m+1}, ...s_n)$$

Thus, π' consists of high-dimensional path segments, which are separated by sequences of low-dimensional states. If π' contains no low-dimensional states, then $\pi^*_{hd}(X_S, X_G) = (s_1, s_2, ..., s_n)$ is a valid path in G_{ad} and it goes through high-dimensional regions only. Then, its cost is cannot be smaller than the cost of a least-cost path in G_{ad}, which violates our initial assumption. Thus, π' contains at least one low-dimensional state.

Note that the transitions between high- and low-dimensional states, such as $(s_k, \lambda(s_{k+1}))$ and $(\lambda(s_m), s_{m+1})$ are high-dimensional transitions by our definition of how to construct T_{ad} and their cost is equal to the corresponding transitions in the high-dimensional graph G_{hd}, where c_{hd} denotes the cost of a transition in G_{hd} and c_{ad} denotes the cost of a transition in G_{ad}.

$$c_{ad}(s_k, \lambda(s_{k+1})) = c_{hd}(s_k, s_{k+1}) \tag{A.2}$$

$$c_{ad}(\lambda(s_m), s_{m+1}) = c_{hd}(s_m, s_{m+1}) \tag{A.3}$$

Let's use $\pi_i = (s_j, ..., s_k)$ to represent a maximum-length high-dimensional path segment (a maximum-length sequence of high-dimensional states) in π'. Also, let $t_i = (s_j, \lambda(s_{j+1}))$ represent a transition from a high-dimensional to a low-dimensional state in π', $t'_i = (\lambda(s_j), s_{j+1})$ represent a transition from a low-dimensional to a high-dimensional state in π', and $L_i = (\lambda(s_j), ..., \lambda(s_k))$ represent a maximum-length sequence of low-dimensional states in π'. Then, π' can be represented as

$$\pi' = (\pi_1, t_1, L_1, t'_1, \pi_2, t_2, L_2, t'_2, ...\pi_{r-1}, t_{r-1}, L_{r-1}, t'_{r-1}, \pi_r)$$

In other words, π' is a combination of high-dimensional path segments and sequences of low-dimensional states connected by appropriate transitions.

Now let us make the following definitions. For any $L_i = (\lambda(s_j), ..., \lambda(s_k))$, let $H_i = (s_j, ..., s_k)$ be the corresponding sequence of high-dim. states in $\pi^*_{hd}(X_S, X_G)$. For any

$t_i = (s_j, \lambda(s_{j+1}))$, let $\tau_i = (s_j, s_{j+1})$. For any $t'_i = (\lambda(s_j), s_{j+1})$, let $\tau'_i = (s_j, s_{j+1})$. Notice that $c_{ad}(t_i) = c_{hd}(\tau_i)$ and $c_{ad}(t'_i) = c_{hd}(\tau'_i)$, as mentioned above (Eq. A.2 and Eq. A.3). Then, we can write $\pi^*_{hd}(X_S, X_G)$ in terms of our new definitions as

$$\pi^*_{hd}(X_S, X_G) = (\pi_1, \tau_1, H_1, \tau'_1, \pi_2, \tau_2, H_2, \tau'_2, ...\pi_{r-1}, \tau_{r-1}, H_{r-1}, \tau'_{r-1}, \pi_r)$$

We basically break down $\pi^*_{hd}(X_S, X_G)$ into path segments that map into high-dimensional regions in G_{ad} (π_i's), path segments that map into low-dimensional regions of G_{ad} (H_i's), and transitions to connect them (τ_i's and τ'_i's).

Then, the cost of $\pi^*_{hd}(X_S, X_G)$ can be written as the sum of its individual segments and transitions:

$$c(\pi^*_{hd}(X_S, X_G)) = \sum_{i=1...r} c(\pi_i) + \sum_{i=1...(r-1)} c_{hd}(\tau_i) + \sum_{i=1...(r-1)} c_{hd}(\tau'_i) + \sum_{i=1...(r-1)} c(H_i)$$

Then, using Eq. A.2 and Eq. A.3 we get:

$$c(\pi^*_{hd}(X_S, X_G)) = \sum_{i=1...r} c(\pi_i) + \sum_{i=1...(r-1)} c_{ad}(t_i) + \sum_{i=1...(r-1)} c_{ad}(t'_i) + \sum_{i=1...(r-1)} c(H_i) \quad \text{(A.4)}$$

Now let's consider an arbitrary $H_i = (s_j, ..., s_k)$. By our assumption stated in 4.1:

$$c_(H_i) = c(\pi^*_{hd}(s_j, s_k)) \geq c(\pi^*_{ld}(\lambda(s_j), \lambda(s_k)))$$

Let's denote $\pi^*_{ld}(\lambda(s_j), \lambda(s_k))$ for an arbitrary $H_i = (s_j, ..., s_k)$ by $\pi^*_{ld}[H_i]$. Then, from Eq.

A.4, we have:

$$c(\pi^*_{hd}(X_S, X_G)) \geq \sum_{i=1...r} c(\pi_i) + \sum_{i=1...(r-1)} c_{ad}(t_i) + \sum_{i=1...(r-1)} c_{ad}(t'_i) + \sum_{i=1...(r-1)} c(\pi^*_{ld}[H_i])$$

Rewriting the sumations in a more intuitive form leads to:

$$c(\pi^*_{hd}(X_S, X_G)) \geq c(\pi_1) + c_{ad}(t_1) + c(\pi^*_{ld}[H_1]) + c_{ad}(t'_1) + c(\pi_2) + c_{ad}(t_2) + c(\pi^*_{ld}[H_2]) +$$

$$+ c_{ad}(t'_2) + ... + c(\pi_{r-1}) + c_{ad}(t_{r-1}) + c(\pi^*_{ld}[H_{r-1}]) + c_{ad}(t'_{r-1}) + c(\pi_r) \quad \text{(A.5)}$$

Notice that for all $\pi_i = (s_j, ..., s_k)$, π_i is a valid path in G_{ad}, as $s_j, ..., s_k \in S_{ad}$. Then, for any two $\pi_i = (s_j, ..., s_k)$ and $\pi_{i+1} = (s_p, ..., s_q)$ we have:

- $t_i = (s_k, \lambda(s_{k+1}))$—a transition from the end of π_i to $\lambda(s_{k+1})$;
- $\pi^*_{ld}[H_i] = \pi^*_{ld}(\lambda(s_{k+1}), \lambda(s_{p-1}))$—a low-dimensional path from $\lambda(s_{k+1})$ to $\lambda(s_{p-1})$;
- $t'_i = (\lambda(s_{p-1}), s_p)$—a transition from $\lambda(s_{p-1})$ to the beginning of π_{i+1}.

Therefore, the sequence

$$\pi''_{ad} = (\pi_1, t_1, \pi^*_{ld}[H_1], t'_1, \pi_2, t_2, \pi^*_{ld}[H_2], t'_2, ...\pi_{r-1}, t_{r-1}, \pi^*_{ld}[H_{r-1}], t'_{r-1}, \pi_r)$$

is a valid path in G_{ad} from $X_S = s_1$ to $X_G = s_n$, where all π_i are high-dimensional path segments lying in high-dimensional regions of G_{ad}, all t_i and t'_i are valid transitions between high- and low-dimensional regions, and all $\pi^*_{ld}[H_i]$ are valid paths through the low-dimensional regions of G_{ad}. The cost of π''_{ad} is:

$$c(\pi''_{ad}) = c(\pi_1) + c_{ad}(t_1) + c(\pi^*_{ld}[H_1]) + c_{ad}(t'_1) + c(\pi_2) + c_{ad}(t_2) + c(\pi^*_{ld}[H_2]) +$$

$$+ c_{ad}(t'_2) + ... + c(\pi_{r-1}) + c_{ad}(t_{r-1}) + c(\pi^*_{ld}[H_{r-1}]) + c_{ad}(t'_{r-1}) + c(\pi_r) \quad \text{(A.6)}$$

Thus, from our assumption, Eq. A.5, and Eq. A we have the following:

$$c(\pi^*_{ad}(X_S, X_G)) > c(\pi^*_{hd}(X_S, X_G)) \geq c(\pi''_{ad})$$

We have shown that we were able to construct a valid path π''_{ad} in G_{ad}, which has strictly lower cost than the least-cost path $\pi^*_{ad}(X_S, X_G)$ in G_{ad}—a contradiction. Thus, our assumption that

$$c(\pi^*_{ad}(X_S, X_G)) > c(\pi^*_{hd}(X_S, X_G))$$

must be incorrect and we have

$$c(\pi^*_{ad}(X_S, X_G)) \leq c(\pi^*_{hd}(X_S, X_G))$$

Theorem A.II *If we have a finite state-space, algorithm 1 terminates and upon successful termination, the cost of the returned path $\pi(X_S, X_G)$ is no more than ϵ_{track} times the cost of an optimal path from state X_S to state X_G in G^{hd}.*

Proof The termination of the algorithm is ensured by the fact that after each iteration we are introducing new high-dimensional states into G^{ad} and removing the corresponding low-dimensional states. Since we have a finite state-space, after finitely many iterations, G^{ad} will become identical to G^{hd}, containing only high-dimensional states. G^{ad} will then be searched for a least-cost path in a finite time.

If a path π^*_{ad} is successfully computed by the adaptive planning phase, it will be fully

high-dimensional. Then, by definition, the tunnel sub-graph τ constructed around π^*_{ad} will contain π^*_{ad} as a valid fully high-dimensional path from start to goal. Thus, the tracking phase graph search will be able to compute a valid optimal path from start to goal π^*_τ and its cost will be equal to the cost of the optimal high-dimensional path π^*_{ad}, i.e. $c(\pi^*_\tau) = c(\pi^*_{ad})$. Therefore, $c(\pi^*_\tau) \leq \epsilon_{\text{ track}} \cdot c(\pi^*_{ad}) \forall \epsilon_{\text{ track}} \geq 1$ and the sub-optimality check on line 16 of Alg. 1 will be satisfied and the algorithm will return π^*_τ as an optimal solution and terminate.

On the other hand, if no path is found in G^{ad}, the algorithm again terminates stating that no feasible path exists from start to goal.

The second statement of the theorem follows from Theorem A.I. By Theorem A.I, the adaptive planning phase produces an underestimate of the real cost from start to goal.

$$c(\pi^*_{ad}(X_S, X_G)) \leq c(\pi^*_{hd}(X_S, X_G))$$

Upon successful algorithm termination, the tracking phase succeeds in finding a path of cost no more than $\epsilon_{\text{ track}}$ times the cost of the computed adaptive path. Thus, we have

$$c(\pi_\tau(X_S, X_G)) \leq \epsilon_{\text{ track}} \cdot c(\pi^*_{ad}(X_S, X_G)) \leq \epsilon_{\text{ track}} \cdot c(\pi^*_{hd}(X_S, X_G)).$$

Hence, the cost of the tracked path is no larger than $\epsilon_{\text{ track}}$ times the cost of an optimal path from start to goal in G^{hd}. □

Theorem A.III *If $\epsilon_{\text{ plan}}$-suboptimal searches are used in lines 6 and 12 of algorithm 1, the cost of the path returned by our algorithm is no larger than $\epsilon_{\text{ plan}} \cdot \epsilon_{\text{ track}} \cdot \pi^*_{hd}(X_S, X_G)$.*

Proof If we use an ϵ-suboptimal search in the adaptive planning phase, we know that that the cost of the produced path $c(\pi_{ad})$ is no larger than $\epsilon \cdot c(\pi^*_{ad})$. Then we have $c(\pi_{ad}) \leq \epsilon \cdot c(\pi^*_{ad}) \leq \epsilon \cdot c(\pi^*_{hd})$. Then we know that the tracking phase produced a path π_τ of cost no larger than $\epsilon_{\text{ track}} \cdot c(\pi_{ad})$. Hence, we have $c(\pi_\tau) \leq \epsilon_{\text{ track}} \cdot c(\pi_{ad}) \leq \epsilon_{\text{ track}} \cdot \epsilon \cdot c(\pi^*_{hd})$. □

APPENDIX B : Tree-Restoring Weighted A* Proofs

Theorem B.I *All states X with $C(X) > c$ will become unseen after $restoreSearch(c)$ is called.*

Proof The function will not insert X into the $OPEN$ or $CLOSED$ lists since $C(X) > c$. $g(X)$ will be set to ∞ and the parent pointer of X will be cleared, making X *unseen*. Also, any descendant X_d of X in the back-pointer tree must have been created after X ($C(X_d) > C(X) > c$). Thus, the call to $restoreSearch(c)$ will make X_d *unseen* as well. □

Theorem B.II *The contents of the $OPEN$ and $CLOSED$ lists after $restoreSearch(c)$ is called are identical to what they were at the end of step c of the algorithm.*

Proof Let $OPEN_c$ and $CLOSED_c$ be the $OPEN$ and $CLOSED$ lists at the end of step c of the algorithm. Let $OPEN'$ and $CLOSED'$ be the $OPEN$ and $CLOSED$ lists after the function $restoreSearch(c)$ is called. In can be easily shown that $X \in OPEN_c$ iff $X \in OPEN'$ and $X \in CLOSED_c$ iff $X \in CLOSED'$. Let $X \in CLOSED_c$, then X has been created and expanded before or during step c. Thus, $C(X) < c$ and $E(X) \leq c$. Then X will be placed in $CLOSED'$ by $restoreSearch(c)$. Let $X \in CLOSED'$, then $C(X) < c$ and $E(X) \leq c$. Thus, X has been created and expanded before or during step c of the algorithm and $X \in CLOSED_c$. Let $X \in OPEN_c$, then X has been created, but not yet expanded at the end of step c. Thus, $C(X) \leq c$ and $E(X) = \infty$. Then X will be placed in $OPEN'$ by $restoreSearch(c)$. Let $X \in OPEN'$, then $C(X) \leq c$ and $E(X) > c$. Thus X has been created, but not yet expanded at the end of step c. Then $X \in OPEN_c$. Thus, $OPEN_c \equiv OPEN'$ and $CLOSED_c \equiv CLOSED'$. □

Theorem B.III *All states X with $C(X) \leq c$ will have correct parent pointers and corresponding g-values after $restoreSearch(c)$ is called.*

Proof We construct a proof by contradiction. Suppose a state X has an incorrect parent pointer, i.e there exists a state $P' \in CLOSED$ such that $g(P') + cost(P', X) < g(P) +$

$cost(P, X)$ (a better parent P' for X exists in the $CLOSED$ list). We argue that P' must have been expanded before P, and since P' provides better g-value than P, then P cannot have been recorded as a parent for X. Suppose P was expanded before P'. Then $E(P) < E(P') \leq c$. The call to $updateParents(c)$, then should have found P' as the parent of X as P' has been expanded more recently than P, but still before or during step c—a contradiction. Then, P must have been expanded after P' and $E(P') < E(P) \leq c$. However, since the g-value obtained through P is larger than the g-value obtained through P', P would not have been recorded as a parent of X when P was expanded because a better parent had been found already. Thus, P could not be a parent of X—contradiction. Thus, the parent pointers and their corresponding g-values are computed correctly by $restoreSearch(c)$. □

Theorem B.IV *Let M be the set of all modified states after a successful incremental A^* search episode. Let $c_{min} = min(C(X)|X \in M)$. $restoreSearch(c)$ for any $c < c_{min}$ results in a search* ***state*** *that is valid with respect to the modified states M.*

Proof The result follows directly from the above theorems. □

Theorem B.V *The function $heuristicChanged()$ terminates and at the time of its termination the search is restored to a search* ***state*** *that is valid with respect to the new heuristic values. That is, no state has been expanded out-of-order with respect to the new f-values.*

Proof Let X_0 be the state with lowest f-value in $OPEN$ in the current search ***state***. X_0 was first put in $OPEN$ at step $C(X_0)$.

Consider the set I computed in $heuristicChanged()$. As in (Likhachev et al., 2003), $v(X)$ stores the value of $g(X)$ at the time X was expanded. Therefore $v(X) + \epsilon \cdot h(X)$ represents the f-value of X at the time of its expansion $E(X)$, but also accounting for the new heuristic values. $I = \{X_i \in CLOSED | v(X_i) + \epsilon \cdot h(X_i) > f(X_0) \wedge C(X_0) < E(X_i)\}$. In other words, I contains all expanded states that had higher f-values at the time of their expansion than the current candidate for expansion X_0 and that were expanded while X_0 was in $OPEN$. As such, I contains all possible states that might have been expanded incorrectly before X_0

according to the new f-values. Note that it is possible that the current $f(X_0)$ is lower than the value of $f(X_0)$ at step $E(X_i)$, as $g(X_0)$ might have decreased as the search progressed after step $E(X_i)$. Therefore, it is possible that $f(X_i) \leq f(X_0)$ was true at step $E(X_i)$ and that $f(X_i)$ was correctly selected for expansion before X_0. Thus, states in I are not necessarily expanded incorrectly, but they are the only possible states that might have been expanded incorrectly. Let $s' = \min(E(X')|X' \in I) - 1$ as computed in $heuristicChanged()$. Restoring the search ***state*** to step s' ensures that no states have been expanded incorrectly before X_0. At the end of the while loop $I = \emptyset$, thus no states in $CLOSED$ could have been expanded incorrectly with respect to the current expansion candidate X_0.

To prove that $heuristicChanged()$ terminates, we argue that the integer s' strictly decreases through the execution of the while loop. If s' becomes 0, then $CLOSED = \emptyset$ making $I = \emptyset$. $\square$

Theorem B.VI *TRA^* expands each state at most once per search query and never expands more states than Weighted A* from scratch (up to tie-breaking).*

Proof It is easy to verify that each state can be expanded at most once per search query, as once a state has been expanded and put in $CLOSED$ it can never be placed in $OPEN$. The fact that TRA^* does not expand more states than performing Weighted A^* from scratch follows almost trivially from the fact that the two algorithms produce the same order of state expansions (up to tie-breaking), but TRA^* is able to resume searching from a step $s \geq 0$, thus not performing the first s expansions that Weighted A^* from scratch would have to perform. $\square$

BIBLIOGRAPHY

S. Aine and M. Likhachev. Anytime truncated D* : Anytime replanning with truncation. In M. Helmert and G. Rger, editors, *SOCS*. AAAI Press, 2013. ISBN 978-1-57735-584-7. URL http://dblp.uni-trier.de/db/conf/socs/socs2013.html#AineL13.

D. Berenson, S. Srinivasa, D. Ferguson, A. Collet, and J. Kuffner. Manipulation planning with workspace goal regions. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 618–624, 2009.

D. Berenson, S. Srinivasa, and J. Kuffner. Task space regions: A framework for pose-constrained manipulation planning. *International Journal of Robotics Research (IJRR)*, March 2011.

R. Bohlin and L. E. Kavraki. Path planning using lazy prm. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 521–528. IEEE Press, IEEE Press, April 2000. doi: 10.1109/ROBOT.2000.844107.

A. Botea, M. Müller, and J. Schaeffer. Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, 1(1):7–28, 2004.

O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 341–346, 1999.

V. Bulitko, N. Sturtevant, J. Lu, and T. Yau. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 30:51–100, 2007.

W. Burgard, M. Moors, C. Stachniss, and F. Schneider. Coordinated multi-robot exploration. *Robotics, IEEE Transactions on*, 21(3):376–386, June 2005. ISSN 1552-3098. doi: 10.1109/TRO.2004.839232.

R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *IJRR*, 18(6):534–555, 1999. doi: 10.1177/02783649922066385. URL http://ijr.sagepub.com/content/18/6/534.abstract.

J. Butzke, K. Sapkota, K. Prasad, B. MacAllister, and M. Likhachev. State lattice with controllers: Augmenting lattice-based path planning with controller-based motion primitives. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 258–265, Sept 2014. doi: 10.1109/IROS.2014.6942570.

B. Cohen, S. Chitta, and M. Likhachev. Search-based planning for manipulation with motion primitives. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2902–2908, 2010.

B. J. Cohen, G. Subramania, S. Chitta, and M. Likhachev. Planning for manipulation with adaptive motion primitives. In *ICRA*, pages 5478–5485. IEEE, 2011.

D. C. Conner, H. Choset, and A. A. Rizzi. Integrating planning and control for single-bodied wheeled mobile robots. *Autonomous Robots*, 30(3):243–264, 2011.

O. De Silva, G. Mann, and R. Gosine. Development of a relative localization scheme for ground-aerial multi-robot systems. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 870–875, Oct 2012. doi: 10.1109/IROS.2012.6386015.

E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel. Path planning for autonomous vehicles in unknown semi-structured environments. *International Journal of Robotics Research*, 29:485–501, April 2010. ISSN 0278-3649.

D. Fox, W. Burgard, H. Kruppa, and S. Thrun. Collaborative multi-robot localization. In W. Frstner, J. Buhmann, A. Faber, and P. Faber, editors, *Mustererkennung 1999*, Informatik aktuell, pages 15–26. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66381-2. doi: 10.1007/978-3-642-60243-6_2. URL http://dx.doi.org/10.1007/978-3-642-60243-6_2.

J. Gaschnig. A problem similarity approach to devising heuristics: First results. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, pages 301–307, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc. ISBN 0-934613-47-8. URL http://dl.acm.org/citation.cfm?id=1624861.1624931.

K. Gochev, B. Cohen, J. Butzke, A. Safonova, and M. Likhachev. Path planning with adaptive dimensionality. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 2011.

K. Gochev, A. Safonova, and M. Likhachev. Planning with adaptive dimensionality for mobile manipulation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2012.

K. Gochev, A. Safonova, and M. Likhachev. Incremental planning with adaptive dimensionality. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

K. Gochev, V. Narayanan, B. Cohen, A. Safonova, and M. Likhachev. Motion planning for robotic manipulators with independent wrist joints. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

B. Grocholsky, J. Keller, V. Kumar, and G. Pappas. Cooperative air and ground surveillance. *Robotics Automation Magazine, IEEE*, 13(3):16–25, Sept 2006. ISSN 1070-9932. doi: 10.1109/MRA.2006.1678135.

G. Guida and M. Somalvico. A method for computing heuristics in problem-solving. In *AISB/GI (ECAI)'78*, pages 115–121, 1978.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2): 100 –107, july 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

K. Hauser. *Motion Planning for Legged and Humanoid Robots*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3332836.

C. Hernández, X. Sun, S. Koenig, and P. Meseguer. Tree adaptive A*. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 123–130, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9826571-5-3, 978-0-9826571-5-7. URL http://dl.acm.org/citation.cfm?id=2030470.2030488.

R. Holte, T. Mkadmi, R. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *ARTIFICIAL INTELLIGENCE*, 85:321–361, 1996a.

R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. Macdonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *In Proceedings of the National Conference on Artificial Intelligence*, pages 530–535, 1996b.

S. Kagami, F. Kanehiro, Y. Tamiya, M. Inaba, and H. Inoue. Autobalancer: An online dynamic balance compensation scheme for humanoid robots. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions : the Fourth Workshop on the Algorithmic Foundations of Robotics*, pages 329–339. A K Peters, Ltd., Natick, MA, USA, 2001. ISBN 1-56881-125-X. URL http://books.google.com/books?id=zr9j_CL0OigC&pg=PA329#v=onepage&q=&f=false.

M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *International Conference on Robotics and Automation*, 2011.

V. Kallem, A. Komoroski, and V. Kumar. Sequential composition for navigating a nonholonomic cart in the presence of obstacles. *Robotics, IEEE Transactions on*, 27(6): 1152–1159, Dec 2011. ISSN 1552-3098. doi: 10.1109/TRO.2011.2161159.

M. Kapadia, A. Beacco, F. Garcia, V. Reddy, N. Pelechano, and N. I. Badler. Multidomain real-time planning in dynamic environments. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13, pages 115–124, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2132-7. doi: 10.1145/2485895.2485909. URL http://doi.acm.org/10.1145/2485895.2485909.

S. Karaman, M. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning

using the RRT*. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2011.

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

R. Knepper and A. Kelly. High performance state lattice planning using heuristic look-up tables. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3375–3380, Oct. 2006. doi: 10.1109/IROS.2006.282515.

S. Koenig and M. Likhachev. D*-lite. In *Eighteenth national conference on Artificial intelligence*, pages 476–483, Menlo Park, CA, USA, 2002a. American Association for Artificial Intelligence. ISBN 0-262-51129-0. URL http://dl.acm.org/citation.cfm?id=777092.777167.

S. Koenig and M. Likhachev. Incremental A*. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems (NIPS) 14*. Cambridge, MA: MIT Press, 2002b.

S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artif. Intell.*, 155(1-2): 93–146, May 2004. ISSN 0004-3702. doi: 10.1016/j.artint.2003.12.001. URL http://dx.doi.org/10.1016/j.artint.2003.12.001.

H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 3116 –3121, april 2007. doi: 10.1109/ROBOT.2007.363946.

J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 995–1001, 2000.

J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue. Motion planning for humanoid robots under obstacle and dynamic balance constraints. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 1, pages 692–698 vol.1, 2001. doi: 10.1109/ROBOT.2001.932631.

S. Lacroix and G. Le Besnerais. Issues in cooperative air/ground robotic systems. In M. Kaneko and Y. Nakamura, editors, *Robotics Research*, volume 66 of *Springer Tracts in Advanced Robotics*, pages 421–432. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-14742-5. doi: 10.1007/978-3-642-14743-2_35. URL http://dx.doi.org/10.1007/978-3-642-14743-2_35.

S. LaValle and J. Kuffner. Rapidly-exploring random trees progress and prospects. *Algorithmic and Computational Robotics New Directions*, pages 293–308, 2001a.

S. LaValle and J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20:378–400, May 2001b.

J. Lee, O. Kwon, L. Zhang, and S. eui Yoon. Sr-rrt: Selective retraction-based rrt planner. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2543–2550, 2012.

L. Li, T. J. Walsh, and M. L. Littman. Towards a unified theory of state abstraction for MDPs. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (ISAIM-06)*, 2006. URL http://research.microsoft.com/apps/pubs/default.aspx?id=178885.

W. Li, T. Zhang, and K. Kuhnlenz. A vision-guided autonomous quadrotor in an air-ground multi-robot system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2980–2985, May 2011. doi: 10.1109/ICRA.2011.5979579.

M. Likhachev and D. Ferguson. Planning long dynamically-feasible maneuvers for autonomous vehicles. In *Proceedings of Robotics: Science and Systems (RSS)*, 2008.

M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS)*. Cambridge, MA: MIT Press, 2003.

M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN 0-201-05594-5.

M. Peasgood. *Cooperative Navigation for Teams of Mobile Robots*. PhD thesis, University of Waterloo, 2007.

R. Philippsen and R. Siegwart. Smooth and efficient obstacle avoidance for a tour guide robot. In *ICRA*, pages 446–451, 2003.

M. Pivtoraiko and A. Kelly. Efficient constrained path planning via search in state lattices. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*, Aug. 2005.

J. E. Pratt and G. A. Pratt. Exploiting natural dynamics in the control of a 3d bipedal walking simulation. In *In In Proc. of Int. Conf. on Climbing and Walking Robots (CLAWAR99*, 1999.

A. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993. URL http://dblp.uni-trier.de/db/journals/ml/ml12.html#Prieditis93.

M. H. Raibert. *Legged Robots That Balance*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1986. ISBN 0-262-18117-7.

N. Ratliff, M. Zucker, J. A. D. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.

I. Rekleitis, R. Sim, G. Dudek, and E. Milios. Collaborative exploration for the construction of visual maps. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 3, pages 1269–1274 vol.3, 2001. doi: 10.1109/IROS.2001.977157.

K.-T. Song, C.-Y. Tsai, and C.-H. C. Huang. Multi-robot cooperative sensing and localization. In *Automation and Logistics, 2008. ICAL 2008. IEEE International Conference on*, pages 431–436, Sept 2008. doi: 10.1109/ICAL.2008.4636190.

A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1652–1659, 1995a.

A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, IJCAI'95, pages 1652–1659, San Francisco, CA, USA, 1995b. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8. URL http://dl.acm.org/citation.cfm?id=1643031.1643113.

I. A. Sucan and L. E. Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII (Proceedings of Workshop on the Algorithmic Foundations of Robotics)*, volume 57, pages 449–464, 2009. doi: 10.1007/978-3-642-00312-7_28. URL http://www.springerlink.com/content/gm47pt40p0740125/.

I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi: 10.1109/MRA.2012.2205651. http://ompl.kavrakilab.org.

X. Sun, S. Koenig, and W. Yeoh. Generalized adaptive A*. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 1*, AAMAS '08, pages 469–476, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-0-9. URL http://dl.acm.org/citation.cfm?id=1402383.1402451.

X. Sun, W. Yeoh, and S. Koenig. Dynamic fringe-saving A*. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 891–898, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-7-8. URL http://dl.acm.org/citation.cfm?id=1558109.1558136.

S. Thrun et al. Map learning and high-speed navigation in RHINO. In D. Kortenkamp,

R. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case Studies of Successful Robot Systems*. Cambridge, MA: MIT Press, 1998.

D. Tolani, A. Goswami, and N. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62:353–388, Sep. 2000.

K. I. Trovato and L. Dorst. Differential A*. *IEEE Trans. on Knowl. and Data Eng.*, 14 (6):1218–1229, Nov. 2002. ISSN 1041-4347. doi: 10.1109/TKDE.2002.1047763. URL http://dx.doi.org/10.1109/TKDE.2002.1047763.

M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, 34:777–779, 1984.

R. T. Vaughan, G. S. Sukhatme, F. J. Mesa-Martinez, and J. F. Montgomery. Fly spy: Lightweight localization and target tracking for cooperating air and ground robots. In *Distributed autonomous robotic systems 4*, pages 315–324. Springer, 2000.

P. Vernaza and D. D. Lee. Learning and exploiting low-dimensional structure for efficient holonomic motion planning in high-dimensional spaces. *I. J. Robotic Res.*, 31(14): 1739–1760, 2012. URL http://dblp.uni-trier.de/db/journals/ijrr/ijrr31.html#VernazaL12.

A. Viguria, I. Maza, and A. Ollero. Distributed service-based cooperation in aerial/ground robot teams applied to fire detection and extinguishing missions. *Advanced Robotics*, 24 (1-2):1–23, 2010. doi: 10.1163/016918609X12585524300339. URL http://dx.doi.org/10.1163/016918609X12585524300339.

M. Vukobratovc. *Biped locomotion : dynamics, stability, control, and application*. Scientific fundamental of robotics. Springer-Verlag, Berlin, New York, 1990. ISBN 0-387-17456-7. URL http://opac.inria.fr/record=b1087909. Translated from the Serbo-Croatian (Cyrillic).

T. Wanasinghe, G. Mann, and R. Gosine. Distributed collaborative localization for a heterogeneous multi-robot system. In *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*, pages 1–6, May 2014. doi: 10.1109/CCECE.2014.6900998.

Wikipedia. A* search algorithm — wikipedia, the free encyclopedia, 2015. URL http://en.wikipedia.org/wiki/A*_search_algorithm. [Online; accessed 01-March-2015].

K. Yin, K. Loken, and M. van de Panne. Simbicon: Simple biped locomotion control. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007a. ACM. doi: 10.1145/1275808.1276509. URL http://doi.acm.org/10.1145/1275808.1276509.

K. Yin, K. Loken, and M. van de Panne. Simbicon: Simple biped locomotion control. *ACM Trans. Graph.*, 26(3), July 2007b. ISSN 0730-0301. doi: 10.1145/1276377.1276509. URL http://doi.acm.org/10.1145/1276377.1276509.

H. Zhang, J. Butzke, and M. Likhachev. Combining global and local planning with guarantees on completeness. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2012.

R. Zhou and E. A. Hansen. Multiple sequence alignment using A*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2002. Student abstract.

R. Zhou and E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005a.

R. Zhou and E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 90–98, 2005b.