MULTI-QUERY MOTION PLANNERS BASED ON SPARSE ROADMAPS FOR CHANGING AND DIFFICULT ENVIRONMENTS

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea Politehnica Timişoara
în domeniul INGINERIE ELECTRONICĂ
ŞI TELECOMUNICAŢII
de către

ing. Mihai Pomarlan

Conducător ştiinţific:	prof.univ.dr.ing. Virgil Tiponuţ
Referenţi ştiinţifici:	prof.univ.dr.ing. Alin Albu-Schaeffer
	prof.univ.dr.ing. Alexandru Gacsadi
	prof.univ.dr.ing. Arjana Davidescu

Ziua susţinerii tezei: 28 Februarie 2014

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor
12. Ingineria sistemelor
13. Inginerie energetică
14. Calculatoare şi tehnologia informaţiei
15. Ingineria materialelor
16. Inginerie şi Management

Universitatea Politehnica Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
Tel./fax 0256 403823
e-mail: editura@edipol.upt.ro

Foreword

Teza de doctorat a fost elaborată pe parcursul activităţii mele în cadrul Departamentului de Electronică-Telecomunicaţii al Universităţii „Politehnica” din Timişoara în cadrul proiectului POSDRU 107/1.5/S/77265.

Aş dori să mulţumesc conducătorului de doctorat prof.dr.ing. Virgil Tiponuţ pentru sprijinul oferit, membrilor comisiei de îndrumare prof. dr. ing. Ivan Bogdanov, conf. dr. ing. Cătălin Căleanu şi as.dr.ing. Radu Mîrşu pentru sfaturile oferite în procesul de redactare al tezei, şi membrilor comisiei de doctorat pentru atenţia acordată şi pentru comentariile constructive.

Mulţumesc şi lui Cristian Ancuţa pentru că mi-a adus la cunoştiintă de posibilitatea unei colaborări cu Willow Garage, şi mulţumesc coordonatorilor mei pe perioada deplasării la Willow Garage: Ioan Alexandru Şucan şi Sachin Chitta. Ei au făcut posibilă şi foarte fructuoasă perioada mea de internship, pe parcursul căreia am putut face cercetare în robotică în cadrul unui colectiv puternic.

Şi bineinţeles, doresc să mulţumesc familiei pentru că mi-au fost alături cu răbdare pe toată durata perioadei de doctorat.

Timişoara, februarie 2014 Mihai Pomarlan

To my parents, my brother, and my sister.

Pomarlan, Mihai

Multi-query motion planners based on sparse roadmaps for changing and difficult environments

Teze de doctorat ale UPT, Seria 7, Nr. 70, Editura Politehnica, 2014, 124 pagini, 50 figuri, 16 tabele.

Cuvinte cheie: motion planning, sparse roadmaps, multi-query planners, manipulation planning, temporal logic.

Rezumat,

We investigate multi-query motion planners that use sparse roadmaps, and extend their domain of applicability to non-reversible systems and path-existence LTL trajectory specifications; we prove probabilistic completeness for the extensions. We offer simulation and experimental evidence that such planners are competitive with single-query planners for changing environments. We present a planner system for intricate manipulation tasks that would be too difficult for a single-query planner to handle in a practical amount of time. We propose a data structure to guide planning in highly constrained environments and provide an object classification criterion that captures kinematic interactions between objects and is suitable for motion planning.

Table of Contents

List of figures

List of tables

1. Motion planning: history and current trends

1.1 Motion planning and its applications

In a general sense, to plan is to find a sequence of simple actions that will accomplish some given goal. A chess-player might plan several moves ahead, taking into account the possible responses of their opponent, with the ultimate goal of winning; a taxi-driver might plan the route to take in order to reach their destination. Plans can therefore be sequences of discrete actions (like moves in a chess-game), or continuous ones. This work will focus on the latter, as motion planning is most often concerned with finding trajectories through continuous spaces.

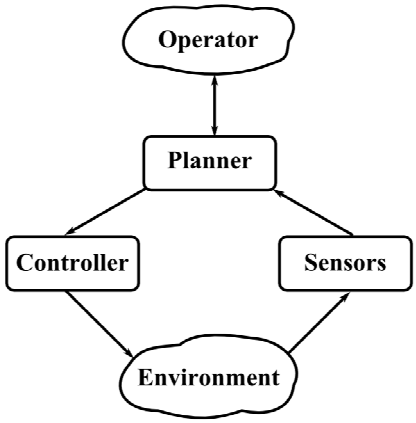

Figure 1-1. Block diagram for a robotic system.

Motion planning means finding a sequence of movements that will take a given system from an initial state to a goal state, or accomplish some given task, while respecting certain constraints imposed either by the environment in which the system moves, by the task, or by the system itself. The kinds of systems that this work will most be interested in are mechanical systems, robots specifically. As will be shown, motion planning has found applications beyond the field of robotics.

A robot includes, and sometimes is considered to be synonymous with, a mechanical system with several actuators that allow it to move. At this level of description, it is not very "smart"- it doesn't even know how to move the actuators. A control system is an information processing system tasked with issuing commands to these actuators, telling each of them how much force or torque to exert at any given moment in order to follow a given, desired, trajectory.

The robot with a control system still isn't very smart (even though designing a good controller is itself a challenging task!). It might know how to follow a trajectory, but it doesn't know what trajectory to follow unless one is given. This might be enough for robots which will only ever do predetermined movements, in rigurously controlled environments, as is the case for industrial robots. However, for

a robot to have any autonomy, it must be able to convert a task supplied by the human operator into a sequence of trajectories, preferably without needing the user's assistence in doing so. This is where the planner comes in.

The simplest kind of planning problem for a robot is to navigate itself inside an environment in which obstacles restrict the kinds of available motions (we refer to these as **geometric constraints**). We gain some further insight here about how a planner differs from a controller: a controller will just tell the robot to move toward a given goal, without knowing or caring whether something is in the way. It's the job of the planner to give such a trajectory to the controller, so as to avoid any obstruction.

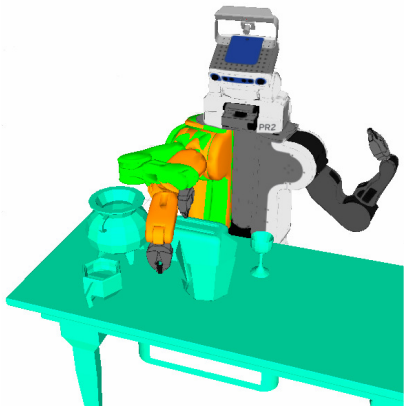

Figure 1-2. Geometric constraints. The robot arm must move from the start (green) to the goal configuration (orange) while avoiding the objects and table.

There are other kinds of constraints that a planner must take into account. A plan is useless unless it is **feasible**, that is, the system can follow it. For example, a car, can only move along the direction in which the wheels are aligned, this direction cannot change instantly, and there's a limit to its possible values. As consequences, a car has a certain minimum turning radius, and cannot move sideways; such constraints that restrict the velocities and paths a system may take, but do not restrict where it can go to, are referred to as **nonholonomic constraints**. Their relevance becomes clear for a task like parallel parking; the planner can't just tell the car to move sideways, and instead, a sequence of maneuvers must be performed.

The robot's mechanical system itself brings some constraints to the planner: the mechanism parts have mass and friction. Further, the actuators can only supply at most a certain finite force (or torque). The deformability of moving parts, in particular their elastic behaviour, may also impact the way that a system behaves, in ways that are significant to the given task. Sometimes, these restrictions can be ignored; maybe the mass is small, the movements are slow, and the actuators are "powerful enough". In real applications however, they become increasingly important to account for, particularly if performance at accomplishing the task is sought. Such constraints are refered to as **differential**, or sometimes **dynamic**, constraints.

To be able to plan, the robot needs to know its surroundings. Sometimes, the environment it moves in is static, or rigurously controlled, but for most

applications where planning is needed, it is also the case that the environment may be, at least at first, unknown, or might change with time (**dynamic environment**) in ways that are not always predictable. The robot needs to have a sensor system, and the planner and sensor system will work together to keep the robot's plans updated.

Figure 1-3. An example of non-holonomic constraint. A car cannot go sideways but parallel parking is possible.

Uncertainty is also a factor while planning. As previously mentioned, the robot might not know its environment and needs to explore and map it; or the environment might change under the influence of unpredictable agents. Further, sensors themselves have measurement errors and actuators have limited precision. A planner would need to account for all this and manage this uncertainty: keep some safety margins in the plan, use an estimator of the current state of the system, some kind of filtering on measurements etc.

Tasks might also specify some kind of **cost function** that must be minimized: fuel consumption, or time to accomplish, for example. An **optimal plan** is one that minimizes this cost. Sometimes, a **reward function** that an optimal plan would maximize is defined instead of a cost.

Ideally, a planner should work in real-time, or close to it. That is, it can grab new information about the environment, and elaborate a plan quickly enough so that the plan remains useful after the time spent planning. A planner that can manage this mode of operation is called on-line. For example, a robot moving in an unknown and uncertain environment needs an **on-line** planner to be able to intelligently respond to changes. Some problems can be handled by off-line planners however. An **off-line** planner would get to study a problem without the constraint to plan fast; the plan will be useful later, and possibly many times. Grasping is a problem of this kind. The robot knows what its arm is like, and might know the geometry of the objects it will have to grab. Given this information, it can spend quite some time looking for efficient grasps around those objects. When it actually comes to grasping, the robot can quickly search through a library of pre-planned movements.

One other important factor necessary to describe a planning problem is the number of **degrees of freedom** of the system that the planner commands, and, if the environment is dynamic, its degrees of freedom of the environment, if it can change. Simply speaking, degrees of freedom are the independent motions that a

system can perform. A rigid body in free ordinary space has six: it can be displaced and/or rotated along three mutually perpendicular axes. A rigid body constrained to move on a plane has just three: it can be displaced along two perpendicular directions, and it also has a heading. The degrees of freedom in a planning problem however are often different from those of a free rigid body. For example, a car has two actuated degrees of freedom (turning and forward velocity), even if it can reach any point and heading combination and therefore its possible destinations need three degrees of freedom to specify. A robot arm with ten revolute joints has ten degrees of freedom, one for each joint. In general, if a robot has n joints, each with one degree of freedom (also known as class 5 joints) then the robot itself has n degrees of freedom.

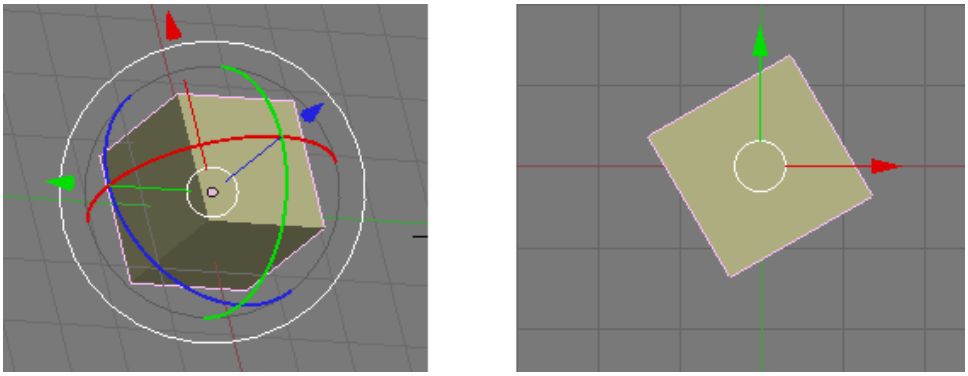

Figure 1-4. Degrees of freedom. A rigid object in three dimensional space has six; planar motion has three.

As mentioned in the beginning, the focus of this work will be continuous systems, which mechanical systems are; their states vary continuously in continuous time. Of course, a planner, operating on a numerical computer, must discretize the system in both state and time. Further, it is sometimes useful to model the system dynamics as capable of discrete transitions between state domains. For example, a walking robot has several sets of possible states: one when no legs contact the ground, and a set for each possibility of legs-to-ground contact. A similar situation occurs when grasping, and finger to object contacts. Maintaining a certain arrangement of contacts to the ground or to an object requires enforcing certain constraints that are not present when no contact is required, and this changes the equations of the system. The planner must take these changes into account as it instructs a walking robot on how to step on uneven terrain, or a manipulator on how to grasp an object of complicated geometry.

From the above it can be seen that planners have tackled a variety of problems, and indeed motion planning has found applications in several fields: vehicle navigation, obstacle avoidance for manipulators, grasp planning, walking robots[Goo02], service robots moving in public environments[Liu10], but also character animation for virtual reality[Kal01], unit movement for computer games[Li08]; crowd simulation; chemists might use motion planning methods to study how proteins fold, or how enzymes interact[Apa04]; motion planning can also be used for system verification[Kim05], by attempting to find a legal trajectory that would result in system failure. The table below summarizes various applications of

motion planning, as well as the capabilities they require of a planner. Green cells are soft requirements, yellow cells indicate important ones.

Table 1-1. Planning problems and capabilities required from a planner.

	Best paths	Many DoF	On-line	Dynamics	Noise	Changing work space	Resolution complete
Manipu-lators							
Vehicle navigation							
Grasping							
Walking							
Service/ Rescue							
Protein folding							
Animation							
Computer games							
Crowd simulation							
System checking							

1.2 Brief history of the field

Research into motion planning for robots began in the 1970s, when the problem of moving a robot, considered as a chain of linked polyhedrals, among an environment populated by static polyhedral obstacles, was first given attention. The algorithms developped in this period used to decompose the environment into regions based on certain geometric properties. They could find a solution, if one existed, but were too computationally demanding to be practical. Nevertheless, hope remained that more efficient algorithms could be found.

Such hopes were dashed when the complexity of the planning problem was proven, in works by Reif[Rei87] and Canny[Can87], to be such that algorithms capable of always finding a feasible plan (if one exists; these are called complete or exact planners) must be too computationally expensive for practical applications. Though Canny's PhD thesis[Can87] described what was at the time the most efficient exact planner for the generalized piano mover problem (also known as

geometric planning), his work contributed to shifting the focus of research away from exact planners. Even Canny's planner was too slow for problems of practical interest. Today, if they are researched at all, exact planners are more of a theoretical curiosity [Var05, Che07].

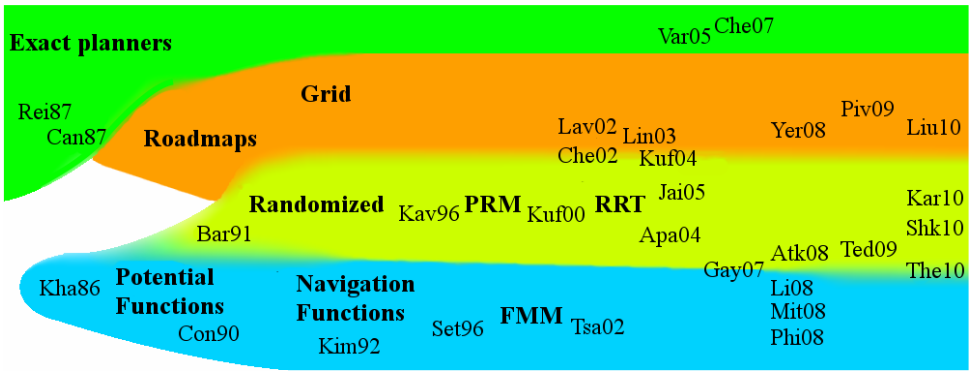

Figure 1-5. Evolution of various motion planning approaches, with some landmark papers.

Instead of exact planners, what is now sought are planners that are, in some sense, "good enough". They find the solution often enough to be useful, without using an impractically large amount of computational resources like time and memory.

Two basic approaches to planning have been developped in the 80s, from which subsequent planners can be said to have branched off. The first is the idea of a **roadmap**[Bar91]. Rather than being aware of the whole environment and everything in it, to the level of precision supported by the number representation on the computer, the planner instead attempts to construct a structure that is as simple as possible but still captures the connectivity of the environment- what points are directly accessible from where. A graph of places and simple paths between them is the typical roadmap. Another kind of roadmap is a grid of cells that covers the environment, where each cell simply records whether an obstacle intersects its volume; algorithms using such a data structure were usually refered to as **grid-based**, rather than roadmap however.

The second approach is that of **potential functions**[Kha86]. The goal that must be reached generates an "attractive field" (a ficticious field, it has no physical existence; the planner just pretends it's there). Meanwhile, the obstacles generate repulsive fields, ficticious as well. The planner uses these fields to compute "forces" on the robot, and instructs the robot to move as if those forces were real.

It quickly became clear that potentials defines as above (attractive goal, repulsive obstacles) may suffer from a problem of local minima, that is, points where the ficticious field doesn't generate any force on the robot, but without being a goal point [Kod87]. Potential functions that avoid this problem for spherical obstacle shapes have been presented, called "harmonic potential functions" [Con90, Kim92]. Another approach has been to utilize the then-newly-developped level set methods used to numerically simulate wave propagation [Set96] in order to generate "navigation functions"- functions whose extremal points are all located at goal points. Navigation function based methods have since grown quite sophisticated, and are the standard for applications to vehicle navigation[Phi08]. They are capable even to provide plans that are optimal under many kinds of

criteria. Note however, that vehicle navigation problems have few degrees of freedom, typically three (for planar movement). Computing a navigation function becomes impractical as the number of degrees of freedom increases, because they rely on the existence of a grid of points to calculate the navigation function at. The number of points needed in such a grid, while imposing a maximum distance (or some other resolution) criterion on it, grows exponentially in the number of degrees of freedom. Triangulating polihedra in spaces of dimension more than 2 is also a difficult problem[Rup92], which further complicates grid construction.

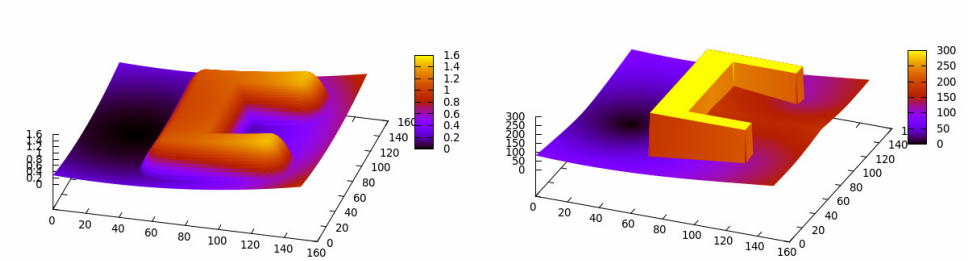

Figure 1-6. A potential function approach (left) has local minima that can trap a robot away from the goal. Navigation functions constructed by a marching method (right) avoid this problem, but are more computationally expensive.

Therefore, applications of navigation function-based methods are limited to situations where there are few degrees of freedom, or the high-dimensional problem can be split into several lower dimensional ones that are either independent or "almost" so. Such is the case of several vehicles moving in formation is one such problem, and so is the case of planning simultaneously for several otherwise independent vehicles: the planner can check for any conflicts in the plan and use a few relatively simple strategies to resolve them.

Another approach to evade local minima of potential functions has been to construct a roadmap between local minima of a potential function, effectively merging the roadmap and potential function approaches[Bar91]. Though itself not later used, the method proved instrumental in developping randomized roadmaps, which have been the dominant approach for problems with many degrees of freedom, from the 1990s onwards.

The idea behind a randomized roadmap[Ama96, Kav96] is to take random "samples" from the environment: these are random positions in space. Advances in collision checking algorithms have made it possible to quickly check whether a sample is inside an obstacle (and therefore invalid) or not[Que09]; methods to compute the level sets of the point-to-3D obstacle distance function also exist[Tsa02] and may be used to speed up collision checking of 3D objects. The planner would keep only valid samples, and check to see which sample can be connected, by way of some simple paths, to nearby samples. This produces a graph that describes the connectivity of the environment.

The Probabilistic Roadmap Method[Kav96] was the first such method to be developped. It constructs a roadmap (a process which is allowed to take a long time), which once constructed allows quick searches for plans. Since the roadmap is a graph, looking for paths through it is easy, and several efficient algorithms exist for this purpose. Since the initial and goal state might not be in the roadmap, they must be added to it before searching for a plan; nodes from the graph, close to the

starting state (and goal, respectively) are chosen, and simple connecting paths are attempted.

The 2000s saw planners attacking problems with dynamics, and the development of exploring-tree methods, also known as diffusion or single-query methods, to distinguish them from the multi-query methods inspired by PRM. At their core, single-query methods are also based on probabilistic roadmaps; they however are constructed so as to rapidly construct the roadmap, and do not reuse it later.

The most representative single-query motion planning algorithm is the Rapidly exploring Random Trees method (RRT)[Kuf00]; many other planning methods are variations on its ideas. The RRT method proceeds by growing trees from the start, and towards the goal. Tree growth happens by selecting a random vertex in the tree and a random direction to take a "small" step towards. When the start and goal trees get sufficiently close, some kind of gap reduction algorithm starts to look for connections between them.

There has been some recent interest however in algorithms that use deterministic sampling strategies, to obtain somewhat better guarantees of finding a solution (if one exists) and to do this, if possible, with fewer samples taken from the environment[Lin03]. As a consequence of this as well as the need of randomized roadmaps to estimate how well they have covered a region with samples, interest in grid-based methods has increased somewhat in recent years.

Improvement in planner capabilities has resulted in more complicated problems being tackled. One direction of research focuses on how to split a problem with many degrees of freedom into a hierarchy of simpler (fewer degrees of freedom) problems, as this can dramatically improve a planner's performance[Shk10]. There is also some literature on replanning and uncertainty management for systems with many degrees of freedom[Toi10].

1.3 Current trends: single-query planners dominate

Most recent papers in motion planning concern themselves with topics about single-query planners (also known as tree-based, or diffusion planners). Further, while support for roadmap planners exists in libraries like OMPL[Şuc12], it is minimal in robotics software packages like MoveIt![Mov12] or OpenRAVE. For MoveIt!, the planning architecture strongly favors a single-query planner that does not keep data between calls. For OpenRAVE, PRM-like planners do not exist in the core installation; a few users have written PRM-like planners as plugins. In general, it appears that single-query planners are added in robotics suites by default, with roadmap based planners considered, at best, as an afterthought. At first, this migration away from roadmap, multi-query planners appears counterintuitive. However, there are a few good reasons for why it has happened.

First, the assumption behind multi-query planners is that a lot of the data they accumulated about the freespace stays relevant; passages that the planner believes are free stay so. In other words, multi-query planners work best if the environment doesn't change. If however, as has been the focus of recent research, we are interested in dynamic environments, then we might as well run a (hopefully fast) single-query planner, and explore the environment fresh each time[Şuc10].

The second assumption multi-query planners tend to make is that trajectories are bidirectional, reversible, and therefore roadmaps are undirected graphs. This is likely because, when sampling to construct a roadmap for future use, there are no special start or goal samples to diffuse away from, or diffuse to, respectively. As a result, most implementations of PRM variants do not use directed graphs for roadmaps.

The situation is completely different for single-query planners. The fact that, when such a planner is run, start and goal configurations are known, imposes a directionality on the trees it constructs. They diffuse away from the start, towards the goal, and trajectories are not checked for reversibility.

A lot of real systems have non-reversible maneuvers. For example, inertia and actuator limits may make it possible for a robot to accelerate in one direction with minimal cost, but reversing its present trajectory in time to avoid colliding with a wall may well be impossible for its actuators. If one is interested in the costs of various maneuvers, these are also not identical for maneuvers that are reverses of each other: it may take no fuel expenditure to go from A to B for example, if that's what the system does because of inertia, but it will take some actuator effort to overcome that inertia and go from B to A.

One sees then that single-query planners make fewer assumptions and are therefore more naturally amenable to a wider class of systems, including systems that are interesting in current research.

1.4 Thesis summary: multi-query planners deserve to be brought back

Recent trends notwithstanding, it seems that the computational effort invested in exploring and building a roadmap should not be discarded from one planning query to the next. The rest of the thesis will investigate ways in which multi-query planners could compete with single-query ones, as well as suggest areas where multi-query planners would be not just well suited but practically the only useful choice.

Chapter 2 presents some theoretical background. Chapters 3 and 6 investigate how sparse planners using the visibility heuristic can be extended so as to apply to a larger class of problems than previously considered. Chapter 4 deals with using multi-query planners in a changing environment. Chapter 5 presents how multi-query planners are suitable for intricate manipulation problems in spaces with many degrees of freedom and narrow passages, as well as suggest an object classification criterion specifically tailored for motion planning applications.

2. Theoretical background

In this chapter we give some definitions and technical background that will be useful in the later chapters: basic notions of graph theory, control theory, temporal logic and motion planning, with an emphasis on sample based planning and its variants.

2.1 Graph theory

A **graph** is a tuple G={V, E} where V is a set, referred to as the **vertex set**, and E is a set of pairs of elements from V referred to as the **edge set**. Elements of V are called vertices and elements of E are called edges. Depending on whether the edges are ordered pairs or not, one speaks of **directed graphs** (or **digraphs**) and **undirected graphs** (or simply, graphs). One may think of undirected graphs as being directed graphs where for any edge going from one vertex to another, there also exists an edge going the opposite way.

The general definition of graphs places no restriction on the number of edges that may exist between the same two vertices. In this work however, we will only work with simple graphs in which, given a pair of vertices and a direction from one to the other, at most one edge exists inside the graph between the given vertices, in the specified direction.

Undirected edges which contain a given vertex are said to be **incident** at that vertex. Vertices connected by an undirected edge are said to be **adjacent** to each other. The case of directed edges needs more care when speaking about incidence and adjacency. A directed edge is interpreted as allowing one to "go" from one of its ends to the other but not in the opposite direction. One can then define two incidence relations (edges leaving, versus edges entering the vertex) and two adjacency relations (vertices reachable from a given vertex via a single edge, versus vertices that can reach a given vertex via a single edge). We'll take adjacency to mean vertices reachable from a given vertex via a single edge.

A **path** in a graph is a sequence of vertices, such that each vertex is adjacent to its predecessor in the path (except for the first vertex, which has no predecessor). A **cycle** is a sequence of vertices that begins and ends at the same vertex, and each vertex is adjacent to its predecessor.

It is easy to see that in undirected graph, any given path can be converted into a cycle by simply going along the path in the usual way, then reversing the sequence of vertices to return to the beginning. This is not necessarily the case in digraphs.

A **subgraph** of a graph G is a graph G'={V', E'} where V' is a subset of the vertex set of G, and E' is a subset of the edge set of G, such that, if an edge appears in E', then both of its ends are from V'. It is not necessarily the case that all edges from G, that have vertices from V' as their ends, are in E'.

A **connected component** of a graph is a subgraph with the property that through any two vertices from the connected component, there exists a cycle that passes through them and is completely included in (uses only vertices and edges from) the connected component. In digraphs, one speaks of **strongly connected components**, but the definition is identical.

Trivially, a vertex is a connected component. However we will require a definition that is more restrictive, so we will also impose a maximality condition on connected components. A connected component is **maximal** if, given any vertex in

the component and any vertex in the graph but not in the component, there is no cycle in the graph that includes the two vertices. In the rest of this work, when we say connected component we usually mean maximal connected component.

Various properties can be associated with vertices and edges beyond the relations of incidence and adjacency. For example, vertices may represent points in some space. Often, edges are labeled with some flow capacity or cost of travel. It is the latter property which makes possible to speak of "best paths" inside a graph.

Graphs are fairly abstract, and one can argue that most data structures in computer science are special cases of graphs. As such, graphs and algorithms for solving various problems involving them are central to many applications. In this work we will concentrate on two of these- roadmaps for planning and (strongly) connected component maintenance- and will now review some specific definitions and algorithms.

As will be shown later, a **roadmap** is a simplified representation of a configuration space of a robot, where vertices are points in that space and edges are "simple" paths between them. Supposing that such a roadmap is provided, in order to solve a planning query one would need to find a path in the roadmap that goes from the start to the goal configuration.

Since typically edges have costs attached to them (for example, distance between the edge's endpoints, time of travel or energy expenditure) it makes sense to ask for the least cost path inside the roadmap. Further, since edge costs in planning applications are typically above 0, one may use the **Dijkstra shortest path algorithm**, which, when implemented with Fibonacci heaps, has $O(|E| + |V|\log(|V|))$ asymptotic complexity, where $|A|$ is the number of elements of the set A.

Dijkstra's algorithm needs to be provided with a graph to work on and an initial vertex to grow the path from. For all other vertices in the graph it constructs a minimum cost and/or predecessor function; the **minimum cost function** returns, for a specified vertex, the cost of the least cost path starting from the initial vertex and ending at the specified vertex. The **predecessor function** returns, for a specified vertex, its predecessor in the least cost path from the initial vertex.

It follows that the Dijkstra shortest path algorithm can be used to find the best paths to all vertices in a graph, starting from a given intitial vertex. Extracting a path can be done, for example, by iterating the predecessor function starting from the goal vertex. When one has several goals to choose from, knowing the minimum cost to reach each of them offers a good way to rank them in some order of efficiency of reachability. The simplicity of the algorithm, both conceptual as well as computational, make it a tried and true component of most roadmap planners.

The developments in this work on the roadmap planners require that a certain heuristic- visibility- be employed when constructing the roadmap. Details on the visibility heuristic will be provided in later sections, however it requires the maintenance of connected components, and so we review that problem, and ways to tackle it, here. Since roadmaps are constructed incrementally, by adding new vertices and edges, we are primarily interested in dynamic component maintenance that is efficient to readjust as the graph changes.

One can treat the connected components of a graph as defining another graph, which we will refer to as **SCC(G)**- the graph of (strongly) connected components of the graph G. We have that the vertex set of SCC(G) is comprised of the connected components of G. An edge in SCC(G) between two components A and B means that there exists at least one edge in G which goes from a vertex in the A component to a vertex in the B component.

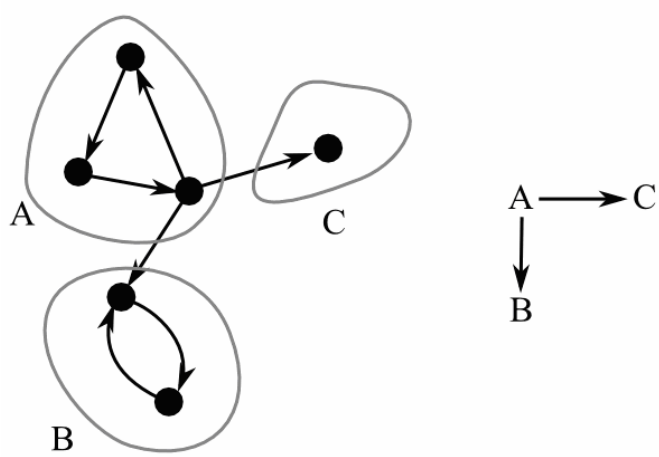

Figure 2-1. A directed graph (left) and its graph of strongly connected components (right).

Naturally, the case of undirected graphs is simpler. If we require that connected components be maximal, then it follows that SCC(G) is a collection of vertices without edges between them. Should an edge appear between two components, then the maximality requirement implies that they should be merged into a single component. It is then easily seen that maintaining connected components in undirected graphs, as long as only vertex/edge additions take place, is the same as maintaining sets when union operations are performed. Disjoint-set data structures solve this problem efficiently; an algorithm by R. E. Tarjan[Tar75] has $O(\alpha(|V|)$ complexity, where α is the inverse of the Ackermann function and as such very slow growing. For practical roadmaps, Tarjan's disjoint-set union algorithm is effectively constant time per query and very fast.

Maintaining connected components in case of vertex/edge deletions is more involved, but an algorithm exists due to Y. Shiloach and S. Even with $O(|V|)$ amortized complexity per edge deletion.

The case of directed graphs is more complex, as SCC(G) is no longer a collection of disconnected vertices. However, if we require connected components to be maximal, SCC(G) has the property of being **acyclic**. For, assuming that SCC(G) does contain a cycle, it follows that a cycle exists in G linking nodes from different components.

Therefore if the addition of an edge to G produces a cycle in SCC(G), it follows that several connected components can be merged into one. This observation allows incremental algorithms for topological sorting/cycle detection to be adapted to strong connected component maintenance[Hae12].

The case of edge deletion doesn't have specialized algorithms. Instead, one could simply use Tarjan's strongly connected components algorithm, which has $O(|V|+|E|)$ complexity.

2.2 Control theory

While aspects of control are beyond the scope of this thesis, control and motion planning are related topics; these two processes also need to work together

in a real system, so it pays to have some overview of what motion control is supposed to do.

Roughly speaking, control is also tasked with taking a system from some start configuration to a goal configuration or trajectory. The difference between control and planning is that an approach that considers only the dynamical equations of the system would be called "control". One that takes into account various constraints in the environment like obstacles would be called "planning".

One could then say that control is motion planning, if the environment were free of obstacles. And indeed, the role of control methods inside a sampling motion planner (to be described in section 2.4.5) is to provide some simple trajectories to link configurations into a graph called the roadmap. "Simple" here just means the control algorithm is not tasked with obstacle avoidance. Depending on the dynamical system, control can be quite a challenging task.

2.2.1 Optimal control

With the application of motion planning to dynamic systems, concepts and methods from optimal control theory are becoming increasingly relevant to motion planning research[Wes04, Tas07, Atk08, Chi08, Mit08, Ted09, The10]. This chapter will briefly review a few of those concepts.

The system dynamics is characterized by a **state update function** which takes as parameters the current system state and current control input, and outputs the next state (for discrete time systems) or the rate of change of the state (for continuous time systems). The state update function may not be linear, may vary with time, and may also be affected by some kind of noise or disturbance (process noise)[The10]. Often, one uses a linearized approximation of the state update function, by retaining terms from its Taylor expansion only up to first order derivatives[Tas07, Atk08]. Second-order approximations are sometimes used as well[Wes04].

One also defines, for any system trajectory, a **cost functional** that must be minimized (or conversely, a **reward functional** that must be maximized). The optimal (best possible) cost (or reward) for getting from one state to another is the **value function** for trajectories that link those states. Several control algorithms attempt to either compute, or approximate, this value function, usually by an iterative process referred to as a value iteration[Tas07], and in so doing, determine the sequence of controls that will optimally guide the system between two states. A function which will, for any state of the system, provide the control to apply in response, is called a **policy**. The policy does not have to be a globally linear function, and can implement more complicated behaviours than linear controllers.

Several planners have used libraries of motion primitives as simple connections between samples in a roadmap. These are sequences of controls for which the cost can be precalculated. They are often selected to be in some sense an optimal link between their endpoints, or at least obey a necessary condition for optimality[Chi08].

Stability concerns the behaviour of a system when subject to disturbances. Intuitively, a stable system will respond with small variations in output to small variations in input. There are more mathematically rigurous criteria for stability. **Lyapunov stability** (of a point x) means that if the system starts close to the point x, it will stay close to x as time passes. If it actually converges to x, then it is **asymptotically stable**. If it converges at least as fast as a certain given exponential rate, then it is also **exponentially stable**. Stability is relevant for

planning because the controller must be able to follow the trajectory that the planner puts forth as a solution and stabilize to the goal[Ted09].

2.2.2 Kinematically reducible systems

In general, real physical systems have inertia. A consequence of this is that velocity cannot jump discontinuously, and needs to be included in the state variables that describe the system.

On the other hand, the fewer state variables a system has, the easier it is to construct plans for it. In particular, if one can remove velocity considerations from planning, one can construct plans much easier. While such an approach does not make any attempt at optimality, it can be quite efficient in practice.

Let then a **driftless system** be a system with no inertia, whose state variables are only positions and whose inputs are velocities. As previously mentioned, physical systems tend not to be driftless.

However, a physical system can be **kinematically reducible** if there exists a driftless system such that any trajectory that the driftless system is capable of, the dynamic system can also follow in a controlled fashion[Bul02, Bul10]; the driftless system is called a **kinematic reduction** of the dynamic system. Without entering into the details of the geometrical formalism, the condition for a driftless system to be a kinematic reduction is that any trajectory it can follow requires only accelerations that the actuators/controls on the dynamic system can produce. Note that a kinematic reduction might not be able to follow all trajectories that the original dynamic system can.

A special case of kinematic reduction is the **kinematic decoupling field**, which is a kinematic reduction whose space of possible velocites is one dimensional. Intuitively, a kinematic decoupling field describes trajectories that the dynamic system can follow, starting from a zero velocity configuration, using only one of its actuators/controls.

Another case is that of a **maximal kinematic reduction** (dynamic systems for which such a reduction exists are called **maximally reducible**), where the space of possible velocities has the same number of dimensions as the space of controls for the dynamic system. Intuitively, the trajectories of a maximal kinematic reduction are the trajectories that the dynamic system can have, when starting from a zero velocity configuration.

Kinematic reductions are useful for local planning (taking a system from one state to another, assuming obstacles are absent) because simple sequences of maneuvers along decoupling fields, or subspaces of a maximal reduction, can take the system from any position to any other (again, assuming no obstacles present). To concatenate trajectory segments along different decoupling fields/subspaces, the velocity at the start and end of the segments need to be zero. Planning then is done by selecting a sequence of segments, seeing how long each segment should be, and finally producing a velocity profile to make the concatenations possible.

Selecting the sequence is usually trivial- often, a single sequence is capable to reach anywhere, depending on the length of the various segments.

Selecting the lengths of the segment is analogous to the inverse kinematics problem in robotics, and is solved by similar algorithms. One has a specified destination in the system's workspace, and needs to find parameters in a "joint space" of the robot (or in this case, lengths of segments along kinematic reductions).

The most important constraint on the velocity profile along the segments is that velocity at the start and end needs to be zero for each segment. One would then start at zero velocity, accelerate to some maximum velocity that doesn't require the actuators to overexert themselves, then slow down to zero by the time the end of the segment is reached.

2.3 Temporal logic

Temporal logic is a catch-all term for various formal systems meant to capture and reason about the time evolution of transition systems. Subsequent sections will describe what a formula in a temporal logic looks like, and how it relates to an abstract description of a system capable to transition, under some rules, from one state to another. Such abstract descriptions of transition systems are known as Kripke structures.

One then defines a **verification problem** (checking that the transition system can behave as indicated by the temporal logic formula) and a **control synthesis problem** (finding a sequence of actions that will satisfy the formula). Temporal logic emerged as a method to verify the execution flow of programs, but it, or subsets of it, have found use in other domains like planning.

Several versions of temporal logic exist, of various levels of expressive power: CTL, LTL[Pnu77], their superset CTL*[Eme86], and the even more powerful μ-calculus [Koz82]. In this work we will use a subset of LTL that is concerned with path existence formulas.

2.3.1 Kripke structures

Kripke structures are abstract representations of transition systems-systems for which various states, each with a set of properties, are defined along with transition possibilities between these states. Mathematically, a **Kripke structure** is a tuple K={V, v_0, E, Π, L} where V is a set of states, E is a set of transitions between these states (and one can think of V and E as forming a directed graph), is v_0 an initial state, Π is a **set of atomic propositions** and L:V->2^{Π} is the **labelling function**.

Kripke structures can be used to model program flow (and indeed, program verification was one of their first uses), but in the context of planning they can be employed to describe the behaviour of a dynamic system, or rather, of a discrete model of it. Vertices are then states of the system and edges give what transitions are possible with the allowed controls. The atomic propositions are some statements that can be checked knowing simply the coordinates of a state; for example, belonging to a certain region of the configuration space can be atomic proposition.

2.3.2 Temporal logic formulas

In order to define the subset of LTL, one first needs to define the syntax of the formulas in the language. The syntax can be given in Backus-Naur form will now be given: a formula Φ can be one of:

$$\phi ::= p | \neg p | \phi \wedge \phi | \phi \vee \phi | \phi U \phi$$

where p is an atomic proposition. In other words, a formula may be an atomic proposition or the negation of an atomic proposition, a conjuction or a disjunction of formulas, or the product of two formulas by the U operator. We also allow, as "syntactic sugar", the names **true** and **false** as atomic propositions and hence formulas (true holds everywhere, false holds nowhere).

The recursive way in which the syntax is defined can naturally be interpreted as describing a way to construct a syntactic tree of subformulas for a given formula of LTL, whose leaves would be the atomic propositions. For a given operator (U, conjuction, disjunction), a subformula appears on either side.

Formulas that do not contain the U operator are called **locally checkable**, because the validity of such a formula can be verified at a state simply by knowing the state in question, without any information about the possible transitions. Locally checkable formulas therefore describe what atomic propositions hold at a given state. In this paper, we will use syntactic trees where the leaves are the locally checkable subformulas.

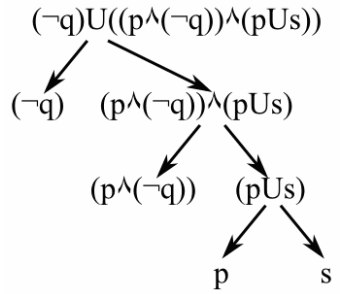

Figure 2-2. A syntax tree for an LTL formula.

The **U (or, "until") operator** in a formula like $\Phi_1 U \Phi_2$ has the following meaning: until a state where Φ_2 holds is reached, Φ_1 must hold all along the path that gets there (it is not required to hold once Φ_2 is reached however). Formulas containing the U operator therefore describe paths. One can however say that a formula containing U operators holds (or not) at a state if there exists a path, starting at that state, which obeys the formula.

We denote the set of states of a Kripke structure K where a formula Φ holds as $[\![\Phi]\!]_K$.

Notice that we do not allow negations on general formulas, which is what restricts this fragment of LTL to path existence formulas. Without the restriction, we could write formulas like the example below

$$\neg(true\, U\, (\neg\phi))$$

which means that it is impossible to reach a state where Φ does not hold (or equivalently, Φ holds on all paths). In the case of this work, where we are concerned with planning for a single system, or possibly a collection of cooperative systems, we only need path existence formulas however. If there's a good choice for the path, then the planner can choose it.

Intuitively, one expects that as a planner discovers new possible transitions of a system, new ways for it to move among the obstacles, then the repertoire of plans it can find increases. This is true for path existence plans, and will be formalized below in the "more isn't less" lemma. It is not true however for all-paths

specifications, as finding previously unknown behaviors of the system may invalidate an all-path specification that was previously thought to hold.

All-paths specifications become important in adversarial scenarios, where one wants to make sure certain properties hold, whatever the adversary may do. Even then, the full power of all-path specifications is overkill, as one only needs to account for all possible "reasonable" choices of the adversary. This is beyond the scope of this work however.

We now state the **"more isn't less" lemma** (an analogous result can be found in [Kar09] for μ-calculus): let there be two Kripke structures K={V, E, v_0, Π, L} and K'={V', E', v_0, Π, L'} which are such that: {V, E} is a subgraph of of {V', E'} and for any vertex v from V, L(v) = L'(v). Then, for any formula Φ in the subset of temporal logic defined above we have that $[\![\Phi]\!]_K$ is a subset of $[\![\Phi]\!]_{K'}$.

The proof proceeds by structural induction on formulas. It trivially follows that locally checkable formulas can not provide counterexamples to the lemma. Suppose now that Φ_1 and Φ_2 are formulas that do not provide counter-examples to the lemma. Then, $\Phi_1 \wedge \Phi_2$ is also not a counter example to the lemma because if it were, then it means there exists a vertex v at which one of the component formulas ceased to hold in the larger Kripke structure, whereas it held in the smaller one. But since neither formula is a counterexample to the lemma, this cannot happen. A similar reasoning shows that a disjunction of formulas that do not contradict the lemma will itself not contradict the lemma.

Suppose now that $\Phi_1 U \Phi_2$ is a counter-example to the lemma. Therefore one can find a vertex v at which it holds in the smaller Kripke structure, but not in the larger one. Let v, v_1, v_2, ..., v_N be a path that verified the formula in the smaller Kripke structure, such that Φ_2 holds at vN, and Φ_1 holds at all previous points. Since by assumption neither Φ_1 nor Φ_2 are counterexamples to the lemma, they continue to hold at those vertices in the larger Kripke structure as well. This then implies that v, ..., v_N is no longer a valid path in the larger Kripke structure, which further implies that at least one edge has disappeared. But this contradicts the manner in which the larger Kripke structure was constructed. Therefore $\Phi_1 U \Phi_2$ isn't a counter-example to the lemma.

Finally, one sees that all formulas that can be constructed in the considered subset of LTL can be constructed from formulas that are not counter-examples to the lemma, via operations that do not produce counter-examples to the lemma. Therefore the subset considered does not contain counter-examples, and the lemma holds. QED.

The "more isn't less" lemma guarantees that as the number of vertices and edges in a roadmap increases, the ability of a planner to find paths to satisfy plans does not decrease.

2.3.3 Specifying robotic tasks in temporal logic

A typical robot task is to go from some location A to another location B; we can consider these as regions of the workspace. A real robot with real actuators and sensors will always have some positioning error, so one should allow for some tolerances when defining regions[Lah10, Ciz12]. Also, suppose we want our robot to avoid a region C (which can be some kind of obstacle). Nonetheless, for computational purposes one would also model the continuous workspace of the robot as a finite collection of points with some links between them, in a graph that

then becomes the Kripke structure we have to validate the formula over. A temporal logic formula to describe this task is:

$$(\neg C)U(B \wedge (\neg C))$$

Notice we need to explicitly require that the robot reaches a region that is inside B but outside C. Notice also that the formula does not include the region A. The formula defines a subset of points from the workspace that satisfy it.

The condition for a plan's existence then is that there are points in A that are also inside the subset of points that satisfies the formula. To find a plan, if one exists, one will run a graph search on points that are inside the region of points that satisfy the formula.

Other specifications are possible. For example, to visit regions A, B, and C, in this order, one would use:

$$(true)U(A \wedge ((true)U(B \wedge ((true)U(C)))))$$

Searching whether a plan exists, and identifying that plan, would proceed similar to the process described before. One would see which points in the Kripke structure modelling the robot's possible configurations satisfy the formula. General tools for finding sequences of states in a Kripke structure that obeys an LTL specification exists[Cim02, Hol04]. In a later chapter, we'll present a more efficient procedure adapted to our subset of LTL.

2.4 Motion planning

2.4.1 Spaces of planning

The **workspace** is simply the three dimensional space that the system exists in. A fourth degree of freedom, time, may be considered in some replanning approaches to handle changing environments.

The **configuration space** is the space defined by the degrees of freedom of the system. A rigid body moving freely in a three dimensional workspace has six degrees of freedom (three for translations, three for rotations), while a robot arm has as many degrees of freedom as it has joints. Note that the configuration space usually has more dimensions than the workspace. Typical robot arms have six or seven joints, not including any degrees of freedom introduced by the end effector; a robot with several arms and/or legs has even more. Also, the configuration space is often not just some higher dimensional Euclidean space ($\mathbf{R}^n$). For example, the configuration space of a rigid body is $\mathbf{R}^3 \times \mathbf{SO}(3)$, where $\mathbf{SO}(3)$ is the space of Special ($\det(\mathbf{A}) = 1$) Orthogonal ($\mathbf{A}^T = \mathbf{A}^{-1}$) 3×3 matrices, and corresponds to the rotation part. Several kinds of coordinates may be used on $\mathbf{SO}(3)$ (Euler angles, Axis-Angle, Quaternions etc.), and one must choose carefully so as to fit the application.

The **task space** is some task-defined space, of lower dimension than the configuration space. For example, the task space of a painter robot might simply be the surface it paints on, which can be specified by a mere two degrees of freedom. Another example of task space would be the orientation of the end effector (three degrees of freedom). A task space is useful precisely because it is of lower dimension than the configuration space. As will be shown later, the number of dimensions a space has drastically affects the complexity of a planning problem, and

thus the time and memory resources needed to solve it. It is a lot preferable to plan in spaces of fewer dimensions, if possible.

For systems with dynamics (where notions like inertia and momentum are significant to the behaviour of the system), one can also define the **state space**, the degrees of freedom of which are the state variables. These are all the variables that one would need to know, in order to be able to predict the system's reaction to inputs. Most of the time, the state space is simply the product of the configuration space of the system, together with the space of velocities of the system; it therefore has twice as many degrees of freedom as the configuration space. For example, for a rigid body, the state space is $\mathbf{R}^3 \times \mathbf{SO}(3) \times \mathbf{R}^3 \times \mathbf{Sk}(3)$ where $\mathbf{Sk}(3)$ is the space of skew-symmetric ($\mathbf{A}^T = -\mathbf{A}$) 3×3 matrices.

Finally, one can define the **control space** as the space of possible control inputs to the system. For a rigid body, it might be the space of all possible forces and torques that can be applied to the body. For a car-like robot, it might be the combination of the possible values for the throttle and steering wheel angle. For a robot arm where each joint has one degree of freedom (which is typical), this space would have as many degrees of freedom as the robot has actuated joints (some of the joints may be passive, in which case one speaks of an **underactuated system**).

Obstacles are regions of a space through which movement is forbidden. An obstacle exists because some object (a wall or another robot, for example) is located somewhere in the workspace. An obstacle in the workspace also defines a forbidden region in configuration space (a robot's position may not be inside the obstacle; or, if the robot is close to the obstacle, it cannot rotate freely any more). All the points in configuration space through which movement is possible make up the **free space**.

2.4.2 Topological issues involved in planning

A **topology** on a set is a collection of subsets, referred to as the "open sets", which has the following properties: the set itself is open; the empty set is open; any union of open sets is open; any finite intersection of open sets is open. A set which is the complement of an open set is **closed**. An open set which contains a point is a **neighbourhood** of that point.

The usual definitions of open and closed intervals (don't contain, and do contain their endpoints, respectively) on the real line obey the above definition. It is worthy of note however that one can define a topology by stipulating that intervals which contain their endpoints are open sets as well. If one defines such a topology, it quickly follows that for every point there is a neighbourhood that contains only itself. This is the discrete topology in which all points are isolated from each other; such topologies are unlikely to appear in a planning context, because they describe totally disconnected spaces.

It is therefore not too misleading to think of open sets as sets which are the same as their interior. Every point in an open set is surrounded "on all sides" by other points from the open set. In contrast, closed sets have points that sit on the boundary.

A collection of open sets such that their union is a superset of some set B is said to be an **open covering** of B. A set is called **compact** if, given any open covering of it, one can select a finite number of sets from that covering that will still form an open covering of it.

If one can define a real-valued function on pairs of element of a set, such that the function is non-negative, zero valued if the pair is of the form (x, x), symmetric, and obeying the triangle inequality, then that set is called a **metric space** and the function is a **distance function**. Actual distance in geometric space is the prototypical example, but one can also use cost to reach to define a (pseudo)metric; the "pseudo" is because cost to reach is not necessarily symmetric.

Given a set, one can define a **σ-algebra** on it as a collection of subsets obeying these properties: the set itself is in the algebra; if a set is in the algebra, then so is its complement; any countable union of sets from the algebra is a set in the algebra. Given a σ-algebra on a set, one can define a **measure function** on the algebra as a real-valued function which is non-negative, returns 0 for the empty set, and the value returned for the union of pairwise disjoint sets is the sum of the values of the function for the disjoint sets.

Measure functions are used to describe two related mathematical concepts, volume and probability. Volume is an intuitive enough concept, but care must be taken to remember that a set of volume 0 is not necessarily empty. A plane in three dimensional space has volume 0 (is a measure 0 set) but does contain points.

The measure as probability interpretation arises by saying that the measure of the original, entire set be called 1. The sets of the σ-algebra then represent events and their volume represents the probability of occurrence. Points in a set become possible outcomes of some random experiment.

Same observation applies to the measure as probability interpretation. Just because an event has measure 0 doesn't mean that there are no outcomes consistent with it. It does mean however that the event almost surely will not happen.

An example can be supplied by considering a cube, and some process which selects a random point inside this cube. Assuming the process isn't biased towards any cube region, then the probability of the selected point to be a specified point inside the cube is 0 (there are 'an infinity of points' that could be selected instead of the given point). The probability of the selected point to lie on a given two dimensional slice is also 0 (there are 'an infinity of slices', all disjoint, that the selected point may be in, rather than the specified slice). The probability of the point lying inside some three-dimensional region (assuming the region is well formed, ie. part of the σ-algebra) is equal to the ratio of its volume to that of the cube.

2.4.3 Complexity of motion planning

It is known, due to work by Reif[Rei87] and Canny[Can87], that even a simple version of the planning problem (moving a chain of rigid bodies from one configuration to another while avoiding obstacles; only geometric constraints, no dynamics or uncertainty, no optimization requirements) is **PSPACE-complete**.

PSPACE means is that any algorithm that can solve any instance of this problem, or correctly report that there is no solution (such an algorithm is called a complete algorithm, do not confuse with PSPACE-complete explained below) will require memory space that is proportional to a polynomial in the number of dimensions of the configuration space.

A problem class X is **PSPACE-complete** if any other problem that is in PSPACE may be efficiently reformulated as an instance of the problem class X. Efficient reformulation means that the reformulation takes few computational

resources- polynomial time in the instance complexity. For example, the fact that geometric motion planning is PSPACE-complete means that any other problem in PSPACE may be efficiently reformulated as a planning problem. It is not yet proven, but after decades of research it is currently believed that PSPACE-complete algorithms require computation time that is exponential in the problem instance complexity; in this case, degrees of freedom of the configuration space. Since typical systems in robotics have six or more degrees of freedom (indeed, humanoid robots may exceed fifty!), complete algorithms are impractical even for simple versions of the planning problem.

Some of the more complex versions of the planning problem (where uncertainties and/or system dynamics are considered) are not even proven to be decideable. This means, it is not known whether there even exists an algorithm that will correctly find a solution (or lack thereof) for any problem instance in some finite time, however large. There is the recent work by P. Cheng, G. Pappas and V. Kumar which shows that planning under differential constraints, and some assumptions on the system's trajectories, is decideable[Che07], but further research on this and related topics has been pursued less. Reif and Canny's complexity results from the 80s have shifted the focus of research into other kinds of algorithms for planning. If completeness of an algorithm is too expensive to ensure, then a compromise becomes acceptable instead. Two such compromises have been pursued in the literature.

Resolution completeness means that the algorithm maintains enough information about the environment, so as to be able to tell apart features that are not too small. If solving a planning problem instance does not depend on features that are below the resolution of the data that the algorithm uses, then the algorithm will find a solution[Che02]. The typical resolution-complete algorithm is grid-based: a grid of cells is constructed, and each cell knows whether there is an obstacle there or not. The smaller the cells, the better the resolution, but the greater the number of cells required to cover the environment. In fact, the number of cells needed to ensure a certain resolution is itself growing exponentially in the number of dimensions of the configuration space, apparently an even worse situation than the PSPACE algorithm designed by Canny[Can87]. The strength of resolution-complete approaches however lies in the ability to prioritize degrees of freedom- some may be undersampled, and thus the algorithm would have poor resolution over them, but may still efficiently find solutions. Careful selection of which degrees of freedom to undersample, and conversely which to sample at high resolution (identifying a task space), is key [Zha07].

Probabilistic completeness means that the algorithm has a chance of finding a solution, if one exists, and that, as the algorithm runs for a longer time, this chance improves. Current probabilistic algorithms have good rates of convergence to certainty (almost surely, the chance of solving converges to 1 exponentially in the time spent on searching), but of course they do not guarantee that a solution will in fact be found. If a solution does not exist, then the algorithm may run forever if allowed to do so. In practice, a probabilistically complete planner is allowed to run for a given interval of time, and whatever solution it found in that time (if any) is used. Planning may fail to find a solution in the allocated time, even if one exists.

Resolution-completeness is, in a sense, a better guarantee than probabilistic completeness. If a resolution-complete algorithm reports no solution, then indeed no solution exists that does not depend on too fine features of the environment. Because of sensor or actuator errors, such too fine features may also be too small

for the system to manipulate or maneuver through anyway, and any plan that depends on them unfeasible. This ability to guarantee that a solution exists (or not) at a given level of resolution is critical in applications like system verification.

On the other hand, probabilistically-complete algorithms seem to not require a grid-like structure the way resolution-complete algorithms do, and as such seem free from the curse of dimensionality. This however is, if at all, only partially true. The chance of a probabilistically-complete algorithm to find a solution grows as it gains more information about the environment, and in effect what it does is construct a grid-like structure of its own, only irregular because of random sampling. However, it often has a better chance of finding a solution "early", before constructing the equivalent of an extensive "grid", unlike resolution-complete algorithms. The two approaches have since influenced each other, as methods from one kind of algorithm have been applied to the other. There is some research for example that suggests using deterministic sampling sequences, whose long term output resembles a regular grid, will bring better results for sampling planners[Lin03].

2.4.4 Planning approaches

The first planning algorithms to be developped were complete planners based on computational geometry algorithms. Such approaches, while workable for simple problems with few degrees of freedom, have fallen out of favour in light of Reif and Canny's complexity results.

Most of the time, higher dimensional problems are handled by constructing road maps. These are graphs, the vertices of which are points in (usually) the configuration space; an edge between two vertices is then a trajectory between those two configurations. The purpose of the roadmap then is to capture the connectivity of the space that the problem is formulated in. A planner would contstruct the roadmap (or use an already available one), connect the initial and final configurations to vertices in the roadmap, then use graph search algorithms to find a path from the initial to the final configuration.

The advantage of this approach is that once a roadmap is built, searching for a path along it is an easy problem; graph search algorithms are already efficient (for example, Dijsktra's algorithm requires time polynomial in the number of vertices), and several heuristic methods may find paths even faster (A*, D*)[Ste94].

Building the roadmap itself can also be done fairly efficiently, as long as resolution-completeness or probabilistic-completeness is acceptable. This means modern planners do not explicitly use knowledge of obstacle shape and distribution to identify passages, cul-de-sacs or any other feature. Indeed, identifying narrow passages among obstacles is at least as hard a problem as planning! It turns out however that collision checking (testing that a configuration is not inside an obstacle) can be solved fairly efficiently, regardless of the number of dimensions of the space. In fact, it is most often enough to perform collision checking in the 3D work-space, or 4D space-time, and not the configuration space itself. This is what suggested a very commonly used method of roadmap construction, which the next section describes.

2.4.5 Sample-based planners

A sampling planner is a method of planning that proceeds by first constructing a roadmap of the (free) configuration space of a system via an iterated sample and connect process, and then using that roadmap to answer planning queries.

A **roadmap** is a graph in which vertices are states in the free space, and edges are simple trajectories between them that do not collide with obstacles. A roadmap therefore attempts to capture the connectivity of the free space.

Several ways to construct roadmaps exist, but one that has proven especially useful in practice is the sample-and-connect approach, which is now described. A sequence of points from the free space is generated, often by using a random process. When a point is generated, a list of points stored previously in the roadmap and that are "close" to the new point is produced, and connection attempts between points on the list and the new one is performed. The connections are trajectories generated by some local planning procedure, which is some control/steering method for the system, required to find a path between given points. The local planner does not take obstacles into account however, and the generated trajectory needs to be checked to be free of obstacle collisions before it can be accepted to the roadmap.

Even if the local planner would be unable to navigate around obstacles, the sampling planner nonetheless may have this capability. By placing "enough" samples in the free space, and connecting close ones, paths around obstacles will form if the planner is **probabilistically complete**: as the number of samples increases, the probability that a path between two points is found (if one exists) tends to one. Many sampling planners are probabilistically complete, at least for certain versions of the planning problem in static and/or well-known environments.

The central notion behind sampling theory (as used in motion planning) is **coverage**. A planner needs to "know" what the environment looks like, while at the same time not require too many samples, which would make computation inefficient. Coverage of an area then means that the planner has taken samples from it. Ideally, all the configuration space should be covered by samples so as to capture its connectivity (and possible paths).

It follows that a natural requirement for samples is **uniformity**. The planner should not oversample some areas and ignore others; this might result in it functioning slowly or even missing a path! In random planners, uniformity is ensured by having samples selected according to a random, but uniformly distributed, procedure.

Recent research has focused on uniformity criteria for both probabilistic- and resolution-complete algorithms. Two such criteria are mentioned in the literature: discrepancy and dispersion.

Dispersion is, roughly speaking, the radius of the largest open ball in configuration space that does not contain any sample. More rigurously, the dispersion of a collection of points **P** in a configuration space **C**, under some distance function ρ is defined as such:

$$\delta(\boldsymbol{P}, \rho) = \max_{q \in C} \min_{p \in P} \rho(q, p)$$

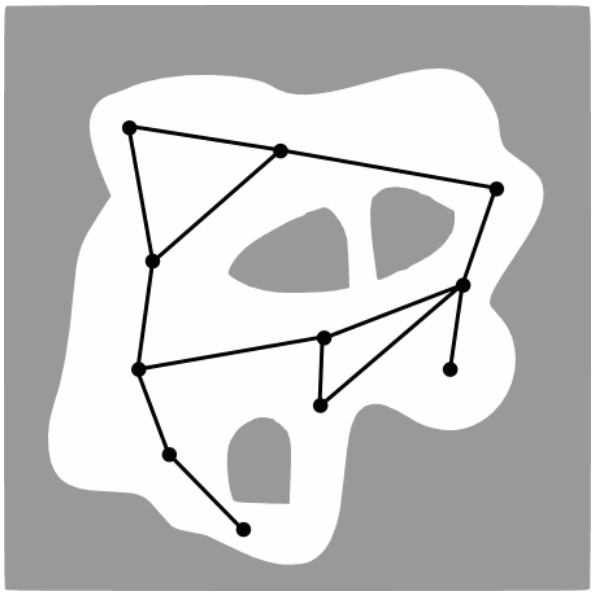

Figure 2-3. A roadmap for an environment. Only connections in the roadmap are known to the planner.

Discrepancy is, intuitively, an indicator of how good a set of points is for estimating the volume of a region in configuration space, when knowing only the fraction of the points that lie inside the region. More rigurously, the discrepancy of a collection of points **P** in a configuration space **C**, under some measure function μ (measure as in mathematical measure theory, corresponding to the volume and not the distance function!), and for a collection R of subsets of **C** is:

$$D(\boldsymbol{P},\boldsymbol{R}) = \sup_{A\in R}\left|\frac{|\boldsymbol{P}\cap\boldsymbol{A}|}{|\boldsymbol{P}|} - \frac{\mu(\boldsymbol{A})}{\mu(\boldsymbol{C})}\right|$$

where | | applied to a finite set (like **P**) is the number of its elements. Low discrepancy means low dispersion, but the converse is not necessarily true.

It turns out that dispersion and discrepancy indicate how good a collection of points is for solving a problem[Lin03]. In particular, dispersion indicates the resolution of a resolution-complete planner. If the samples used by the planner have a certain dispersion, then the planner will find paths as long as those paths do not require corridors thinner than the dispersion.

There are two kinds of point (sample) collections: sets and sequences. A **point set** is simply a finite set of points, generated "at once" by some method. It must be known in advance how many points the set should contain. A grid of n-by-n points, evenly spaced, is a point set. A **point sequence**, unlike a set, generates the points one by one according to some method, and can be continued indefinitely. Picking a new point at random according to a uniform probability distribution is an example of a point sequence. Any point sequence, when stopped at a given number

of points n, is usually slightly worse in terms of dispersion or discrepancy than a dispersion (or discrepancy) optimizing point set with the same number of points. Nevertheless, sequences are preferred in practice because it is hard to tell how many points will be needed to solve a problem. The ability of sequences to continue generating points until a solution is found (or time runs out) trumps their slight disadvantages compared to point sets.

It turns out that deterministic, dispersion-optimizing sequences, obtain better dispersion than random uniform sampling[LaV02, Lin03]. Random sampling cannot be too uniform (or else it is not actually random). Some of the current research in sampling has therefore focused on obtaining dispersion-optimizing sample sequences on various spaces common to motion planning in robotics[Yer08].

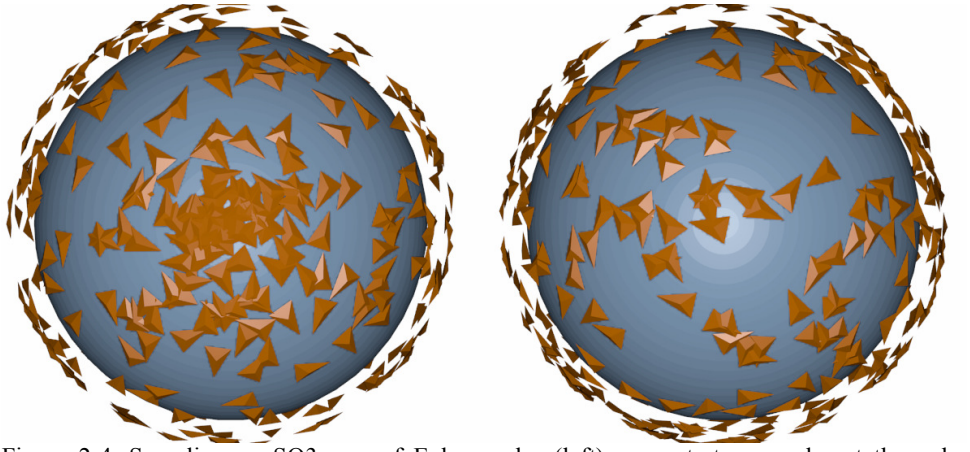

Figure 2-4. Sampling on SO3: use of Euler angles (left) concentrates samples at the polar regions. Sampling quaternions results in a more uniform distribution.

Sequences for $\mathbf{R}^n$ have been fairly well known for some time. Recent work has developped such sequences for $\mathbf{S}^n$ (sphere of dimension n), **SO**(3) and **SE**(3) (the space of rigid body motion, which includes **SO**(3))[Kuf04]. Uniform sampling on **SO**(3) is particularly tricky, as one has to carefully choose measure functions on it. It turns out that there is such a function, the Haar measure, that is a natural choice because it doesn't change after rotation. Therefore, the Haar measure of a subset of **SO**(3) does not change as that subset is "moved around" on **SO**(3). However, some commonly used coordinates on **SO**(3) (like Euler angles) do not preserve Haar measure.

Random sampling has also been researched. The concept of Voronoi bias was introduced to explain why the original RRT is capable to explore a space quickly[Lin04]. **Voronoi bias** means that the probability of a node to be chosen for expansion (connection to a new sample) is proportional to the volume of its Voronoi cell (the collection of points in the configuration space which are closer to this node than to any other node in the tree or roadmap maintained by the planner). Therefore, large unexplored areas will tend to be broken up quicker as more samples are selected there and nodes are expanded towards them. Note that the planner never explicitly constructs a Voronoi diagram, which would be a computationally expensive operation.

Voronoi bias can also explain a problem with random sampling called the **bug trap**. The planner does not know the shape of the obstacles, nor does it know the shape of Voronoi cells. Therefore, it can happen that large Voronoi cells cannot be expanded into, because any expansion attempts are blocked by obstacles. Nonetheless, random sampling would select more samples in those regions, resulting in many failed expansions and/or superfluous nodes and poor planner performance.

To combat this, adaptive sampling domain methods have been developped[Jai05, Yer05]. These start by having the sampling domain start as the entire configuration space, but failed expansion attempts from a node results in sampling around that node being restricted to some sphere. The radius of this is either fixed by tuning, or dependent on the number of failed or successful expansion attempts. Leaf or obstacle nodes (nodes from which expansion is not allowed, and if a sample is closest to one then it is ignored) have also been investigated for possible use in planning. Another approach, similar in spirit, is to use principal component analysis and similar methods on the set of vertices that have failed expansion to try and estimate which directions in the configuration space are blocked by obstacles, and thereafter favor sampling in a submanifold that appears free of obstacles[Dal09]. Other methods proposed to help planners handle narrow passages include obstacle retraction[Sah05] and biasing the sampling process close to obstacle edges[Hsu03].

Sampling in lower dimensional spaces has been considered. Task-space RRT (TS-RRT)[Shk09, Shk10] samples most of the time in some task space selected for the planner. The planner also uses some method (like counting successful expansions) to determine whether it "got stuck" and if this happened, revert to doing a few sample/expansion iterations in configuration space before returning to sampling in task space.

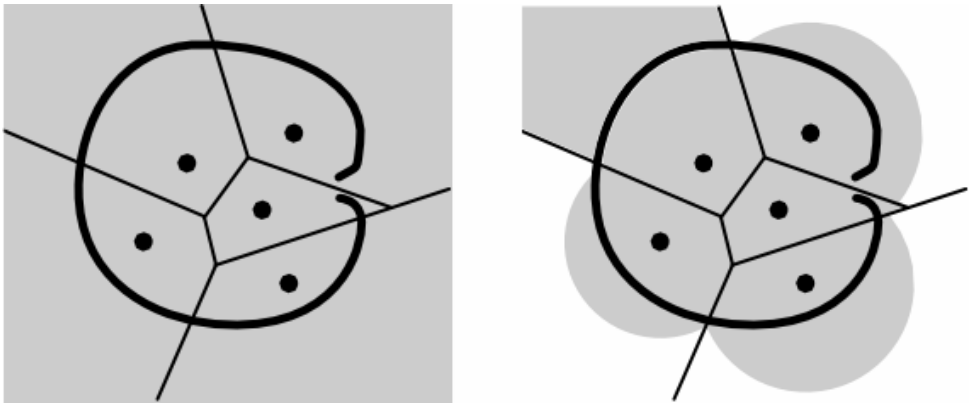

Figure 2-5. Samples in a roadmap determine an implicit Voronoi diagram (left). Adaptive sampling (right) restricts where new samples are generated, to increase the probability of successful roadmap expansions.

Some planning problems explicitly require movements that are constrained to some submanifolds of the space[Ber09, Gui09, Suh11, Şuc12b]. In such cases, constrained sampling algorithms are used, and the relevant measure of sampling

quality is dispersion on the constraint submanifold, not on the configuration space as a whole.

Some planners also use a simpler notion of coverage than dispersion or discrepancy. They use a projection of the configuration space onto a lower dimensional space (like a task-space) and assume that if "enough" samples are located in an area in the projection, then the volume in configuration space that corresponds to that area is considered to be well-covered by samples[Şuc08].

Finally, sampling for systems with dynamics is done in either the state space, the product of the state space with the control space, or some lower dimensional task-space (with occasional forays into state space in case the planner finds itself stuck). More on planning for systems with dynamics in a section below.

Various reconnection strategies also exist. The typical RRT connection strategy is to seek the nearest neighbour to a sample, and select that neighbour for expansion. Therefore, there has been research interest in developping efficient algorithms for the nearest neighbour (or neighbour nearness ranking) problem[Yer08]. Good algorithms and data structures for Euclidean spaces and distance metrics exist, as well as for spaces like **SO**(3) or $\mathbf{S}^n$.

One version, called RRT*[Kar10], expands from the node in a neighbourhood around the sample (not necessarily the nearest node to the sample) that offers the lowest-cost path from the root to the sample. Changing edges in the tree is also done, if the newly added node makes cheaper paths to nodes already in the tree possible.

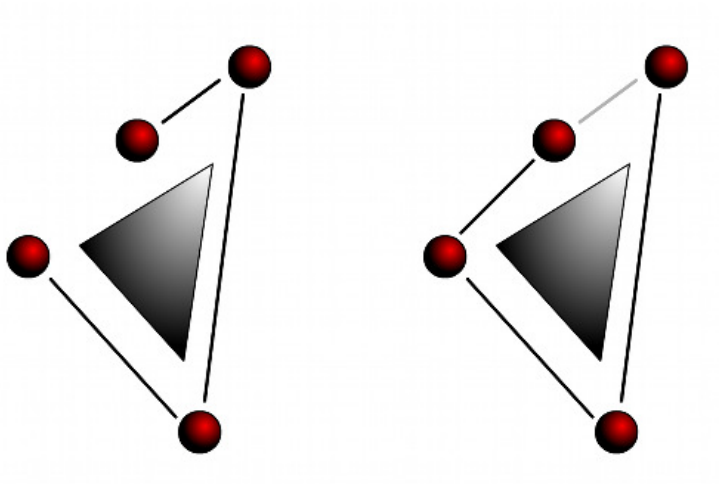

Figure 2-6. Reconnection strategy to shorten paths in a roadmap.

It is important to note that the concept of a **metric** (a distance function) is at the heart of a planner. Node nearness only makes sense when having a good metric, and so do sample uniformity criteria and adaptive sampling domains. Often, the metric used is the Euclidean distance metric. This however is not always appropriate, in particular for systems where dynamics are significant.

In general, sampling planners work by "saturation"; they pick many samples from free space and thus guarantee good properties like probabilistic completeness and even convergence to optimal paths.

A variation on the sampling planner concept, of interest to this work, are the **visibility-based planners**. They are planning algorithms which attempt to reduce the number of vertices needed in the roadmap, by using the information already present in the roadmap as a way to decide if a new candidate sample appears useful.

The visibility heuristic as originally defined works thusly: a candidate sample is useful if it cannot be connected via simple trajectories to samples in the roadmap (so it improves coverage by exploring as yet unreachable areas) or if it can be connected to samples in the roadmap that were previously in different connected components (so it improves connectivity). If a sample is not useful by the criteria defined before, it is rejected and not stored.

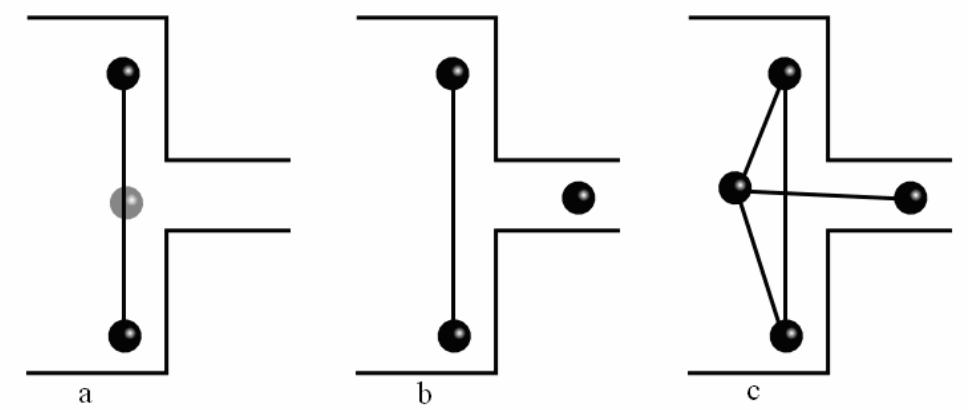

Figure 2-7. Example run on a visibility-based planner. (a) the new, gray sample is rejected as it's visible from both samples in the roadmap. (b) a sample is placed in the corridor. (c) a new sample connects the roadmap.

It appears at first glance that it is dangerous to reject a sample that is, nonetheless, valid (outside of obstacles). Consider the situation in Fig. 2-7 (a): while the new sample can look inside the corridor to the right, the planner is not aware of the corridor and rejects the sample as it appears to bring no new information. However, if the roadmap construction process were to continue, since the volume of the corridor is non-zero and samples are generated by a uniformly distributed process, at some point a sample will be generated in the corridor (see Fig. 2-7 (b)). Afterwards, since the volume of the region containing points visible from the corridor and the original pair of samples is non-zero, a sample will be placed there and this time it will be kept, as it provides a connection between two previously distinct connected components (Fig. 2-7 (c)).

The number of samples needed to cover a configuration space depends on the shape of that space as well as the local planner. It is possible that one local planner would require a roadmap with an infinite amount of samples to completely cover a space, while with another local planner the same space could be adequately covered with a finite number of samples [Sim00]. One would prefer to cover the configuration space in such a way that one could select a subset of vertices from the roadmap, the guards, whose visibility domains covered the entire allowed configuration space, in such a way that no guard vertex is visible from another

guard vertex. Whether such coverage is possible again depends both on configuration space shape and local planner behaviour. In any case, a random algorithm, like sample-based ones usually are, in general will not produce a minimal set of guards even if one exists. One can expect however that a visbility-based planner will produce a more economical roadmap than other kinds of planners.

Visibility based planners are probabilistically complete for the application in which they were proposed (point system moving on straight line trajectories in an euclidean space with purely holonomic constraints and no dynamics), and they produce very sparse roadmaps. The rest of this work considers their applicability and efficiency for a wider class of problems.

2.4.6 Planning in changing or unknown environments

Many environments of interest are changing- they contain moving obstacles or other agents, the behaviour of which is only sometimes predictable. The most naive approach to this problem is to periodically replan. If the planner is fast enough, or the environment simple enough, periodic replanning is workable, but wastes the previous work put in by the planner.

It is often the case that a plan needs only slight adjustments to respond to changes in the environment. Several graph search algorithms (A*, D*)[Ste94] are constructed in such a way so as to use an old path as a starting point when searching for a new path after the graph changed slightly. D* has also been applied to the problem of vehicle navigation in an unknown environment- the vehicle finds out the structure of the obstacles as it moves around.

Figure 2-8. Deformable roadmaps. Connections are pushed away by moving obstacles.

Other planners use adaptive roadmaps. The basic concept here is that the roadmaps are "deformable" as if the links between the nodes were elastic bands[Gay07, Sud07, Gay09]. Some of the nodes in a roadmap ("milestones") behave as weights that are "pushed" around by forces exerted by the moving obstacles; some other nodes ("link nodes") form the elastic chains between the milestones. If a chain of link nodes becomes too stretched, it will break and its nodes removed from the roadmap. It may happen that the roadmap becomes split into several disjoint components (subgraphs that have no edge going from one to the other). The planner will then periodically try to find connections between milestones in the roadmap that belong to disjoint components, so as to repair it and detect any new passages that may have appeared.

Checking for necessary changes to the roadmap may be an expensive operation, and one wants to avoid doing it more often than necessary. Grids on projections into lower dimensional work- or task space are used to detect

whether/where changes to the environment have occured, and where the roadmap needs adjustments.

For dynamical systems with uncertainties (process noise, imperfect state information, measurement noise), the planning problem gains a new aspect, that of predicting what the state of the dynamical system will be after a plan is performed. The prediction uncertainty grows the further in the future one tries to predict, so many planners adopt a receding horizon approach: plan a short segment of the path, allow the system to perform it, then measure and try to estimate what the state of the system is before repeating the procedure. More sophisticated prediction algorithms, which make use of expected future information, may manage the growth in uncertainty to some extent and allow the planning horizon to be extended.

2.4.7 Planning for systems with dynamics

Systems with significant dynamics pose new kinds of problems for planners, not just obstacle avoidance. If a vehicle has momentum, for example, it cannot stop or turn instantly; any plan, to be feasible, must take such limitations into account. In particular, a region of the configuration space becomes indirectly invalid. While the robot is not in collision at those configurations, its momentum is too great for its actuators and collision would be imminent. Mapping this region of inevitable collision is of research interest, but has turned out to be difficult and is done only approximately.

Also, a planned trajectory is useless if the controller cannot stabilize to it, and stabilization is not trivial once the system's dynamics cannot be ignored. Therefore, for dynamic systems, the relevant spaces for planning are the state and control spaces.

It's also the case that the simple Euclidean distance between two states is not necessarily a reflection of how difficult it is to get from one to the other, and offers little information on how to control the system so as to move it toward the desired state (or if that control is possible without destabilizing the system). Consider a vehicle moving with constant velocity along a straight road, a point A at a short distance behind the vehicle, and a point B at a distance in front that is slightly larger than the distance between the vehicle and A. However, B is in the direction the vehicle's momentum is, whereas, to get to A, the vehicle must stop and reverse.

Observe that it may be easier to get from A to B, than it is getting from B to A, if there is momentum from A to B. One therefore speaks of **pseudometric** functions (since unlike a true distance function, they are not symmetric). Finding a good (pseudo)metric function for a planner, or some workaround for not having one, has been the topic of much research in the area of planning under dynamic constraints.

The best pseudometric function would be the cost-to-go function, that is, the smallest possible cost (time, fuel etc) that must be expended to get from one state to another. Computing this function however would require too many computational resources to be practical, especially for high dimensional state spaces.

A few approaches have been put forward to handle system dynamics. One category of approaches uses a library of motion primitives, precalculated "basic" trajectories (sequences of controls) that a plan is to be built out of [Fra01]. It is

possible to compute the cost of basic trajectories, and one can approximate the cost-to-go function from one state to another by ignoring the existence of any obstacles between them, finding basic trajectories that would link those two states assuming no obstacles are present, calculating the cost of those trajectories and selecting the one with the smallest cost.

Another approach based on motion primitives constructs a lattice in state space, in which any two neighboring nodes are connected by all feasible motion primitives that can get from one to the other[Piv09]. The reason for this is that not all sequences of basic trajectories are valid (a car cannot change its turning angle instantly, for example). Edges and nodes from the lattice are then removed if they intersect with an obstacle. Planning is then achieved by searching the remaining graph for a path from the starting to the desired configuration.

Careful selection of motion primitives for a given system is crucial for the good functioning of such planners, and is itself a research problem. The selected motion primitives should cover the system dynamics, without being too many (and thus result in more complicated lattices and longer processing times, for example), while also, ideally, ensuring some kind of optimality conditions.

A more recent approach also uses the approximate cost-to-go (computed assuming no obstacles), but not based on motion primitives. Rather, it attempts to approximate the cost-to-go with a Linear-Quadratic-Regulator (LQR) method[Tas07, Atk08, Ted09]. This is a method for optimally controlling a linear system while optimizing some quadratic cost function; in practice, most systems of interest are not linear, and linearizations of them are used. A linearization is an approximation of the system dynamics, "good enough" for a region that is "small enough". Several linearized models may be used as the system goes along a trajectory.

The LQR based approach has also been combined with computation of Lyapunov functions to estimate coverage of the state space[Ted09, Gla10]. Lyapunov functions are used to estimate, for each node in the planner's tree, an area of states that can be stabilized to the state of the node by way of an LQR. Then, to each node in the tree, a "basin of attraction" is found. The goal state would be the root of this tree, and its child nodes would be located in its basin of attraction. The algorithm would then proceed by trying to bring as many states in the basins of attraction of nodes in the tree. If the initial state is brought in such a basin, then a plan (and control sequence) can be found to take it to the goal.

Yet another recent approach attempts to estimate a "reachable" zone for each node in the tree. This may also use some collection of motion primitives to estimate the reachable zone, but appears less dependent on the quality of the motion primitives[Shk10]. Then, Euclidean distance is used, but not distance between the new sample and the nodes of the tree, rather the distance between the sample and the reachable zones. The reachable zone of a node may also be coarsely estimated by recording failed attempts at node expansion (keeping track of which controls, when applied to the state of the node, fail to produce a valid expansion).

2.4.8 Brief intro to task planning

A robot's task is often more complicated than simply moving from one position to another. For example, a robot might assemble some piece of equipment; it would then need to prepare the relevant pieces, and bring each one, in a certain order, to its required position inside the assembly. **Task planning** is then a field of

research concerned with making a robot capable to reason about, and plan for, the various actions that are required to complete a task.

While motion planning concerns itself with the geometry of the robot's environment, and possibly the dynamics of the robot as well, task planning tackles more abstract concerns. For example, a robot might need to GRAB a piece A and PLACE it in a container B. The detailed geometry of the grab and subsequent place are not too important at the task planning level. More important would be a logical precondition for the action GRAB(A): piece A needs to be at the top of its container.

Then, more generally, one would specify actions (GRAB, PLACE etc) and states of the environment (IS_INSIDE, LOCKED etc) in a symbolic fashion. Actions would have effects that change the symbolic state of the environment, while also having preconditions: unless certain combinations of state variables occur before an action is attempted, then the action cannot succeed. One such logic formalism used for task planning is STRIPS[LaV06].

Of course, checking which symbolic state descriptions apply at any given moment requires there to be some procedure to map configurations of objects in the workspace to those symbolic descriptions. Also, each symbolic action should have a straightforward way of conversion to a motion planning problem [Pla07, Dor09].

Using STRIPS or a similar formalism, task planning would search for a sequence of actions such that each action in the sequence has its preconditions satisfied at the moment the action is attempted, and the state of the environment at the end of the sequence is the desired state. Usually such search employs some hierarchy of subtasks, where one subtask would have as a goal to create the precondition for some subsequent subtask. Various search algorithms like branch-and-bound, or iterative deepening depth-first search are applied to look for such a sequence of actions and subtasks. Actually carrying out the sequence requires not just generating the list of actions, but also turning them into a series of motion planning queries for the robot. The queries will then be treated at the geometric level where motion planning occurs. This is the level that is aware of the geometry of the work environment, and the level that can decide if a motion planning query is feasible or not.

Various ways to ensure collaboration between task and motion planning exist. The previous paragraph describes the simplest one: generate a discrete specification for a sequence of actions or subtasks, and convert it to a sequence of planning problems. Sometimes, it may be that the sequence of actions required by the task planner has no feasible solution that the motion planner can find. For example, a task planner may produce a sequence of subtasks to be achieved, and then each subtask would be analyzed recursively until a sequence of motion planning queries is obtained. The task planner may also generate several candidate solution sequences, in the hope that at least one of them will have feasible plans for all required actions.

Another way that is fairly popular has been to make motion planning methods that are themselves capable of some degree of higher level, symbolic planning. One of the more studied approaches of this kind is that of **manipulation graphs**. Several subspaces of the configuration space of the robot are defined[Gra03, Cam04, Kae11], where one such subspace corresponds to the configurations of the robot if keeps a certain grasp on an object. For example, the robot grabbing a coffee mug with the left arm would be one such subspace; grasping with the right arm another; grasping with both would be yet another subspace. There would also be a subspace corresponding to the robot grasping

nothing in its arms. Subspaces in which the robot grasps something are called **CG** (**continuous grasp**), while the subspace with no grasp is called **CP** (**continuous planning**). A manipulation graph will then represent ways for the robot to go from one such subspace to another (chage grasps, in other words), and allows reasoning about task preconditions and effects.

The planner would create roadmaps for the grasping and no-grasp subspaces of the configuration space. Two kinds of maneuvers are defined. **Transit maneuvers** are maneuvers through CP (the object(s) to be manipulated are not affected by these maneuvers). **Transfer maneuvers** meanwhile are maneuvers through one of the CGs, and affect the configuration of at least one object to be manipulated.

A task plan is then a concatenation of transit and transfer maneuvers; concatenation is possible at places where CP and the various CGs intersect. Such places are configurations where the robot is "about to grasp" one object, and passes from CP into one of the CGs. A manipulation graph would then have (various subsets of) the CGs and CP as vertices, and edges would usually go from the CG subsets to subsets of CP.

The approach neatly encapsulates both logical and geometric aspects of task planning. The manipulation graph is a representation of what actions are compatible to follow one another in a logical sense, while the roadmaps inside the various CGs and CP are a representation of what the robot is physically capable of.

Some refinements on the manipulation graph idea that have appeared in the literature include: using constraint satisfaction algorithms to find destination specifications for the motion planning subtasks[Pla10]; running several PRM searches to avoid having the task planner get stuck if one of the motion planning queries fails to return a solution[Hau09]; modeling a robot's redundant actuators as a Task Motion Multigraph[Şuc11] (a graph with several edges between pairs of vertices), and thus explicitly model the various options a robot has at achieving a subtask.

3. Sparse planners

In this chapter we will describe and analyze a new visibility planner that uses a variable radius for connection attempts. The radius decreases as the number of samples in the roadmap increases. Then we adapt this method to nonreversible systems; as far as we know, this is the first time visibility planners have been applied to non-reversible systems. We show that the proposed algorithms are probabilistically complete by offering original proofs for both the reversible and non-reversible cases. We analyze the computational complexity of the planning algorithms. Finally, we show some simulation results. This chapter contains parts from our paper "Visibility based planners for kinematically constrained vehicles" [Pom13a].

It must be noted that the visibility based planners discussed in this thesis will not in general produce optimal paths. Once a path is generated however, one can apply some local trajectory optimization to improve it. Or, in the roadmap construction, some samples can be kept even when a strict visibility heuristic would dictate they be rejected; one can obtain planners that, while not optimal, produce "near-optimal" paths that are within some guaranteed factor away from an optimal path. We analyze such optimization possibilities in the last section of the chapter.

3.1 The variable radius visibility method

In this section we consider two versions of a new visibility planner for reversible systems. In an attempt to reduce the number of visibility tests performed for each new sample, visibility tests will be restricted to those nodes in the roadmap that are within a certain radius of the new sample. The radius will decrease as the number of samples in the roadmap increases, in a similar fashion to the connection test range of the RRT* and RRG algorithms, so that the expected number of roadmap nodes that will be tested remains constant. If no node within that radius is visible from the new sample, then a fallback radius, which is kept constant, is used for visibility tests. This section has content from the author's "Compact roadmaps from variable radius visibility planners and local trajectory refinement" paper.

3.1.1 Algorithm description

Define *Sample()* as a function returning a random point sample of C_{free}, with uniform distribution. Define $r_{RRT^*}(n)$ as a function returning a real number depending on the sample count of the roadmap. *GenerateTrajectory(**x**, **y**)* is a function which returns true if the local planner procedure can generate a valid trajectory between the two points, ***x*** and ***y***, of C_{free}. *Near(**x**, A, r)* is a procedure that, given a point ***x*** and a set of points A, returns the subset of points from A that are at a distance of at most r from ***x***. For a set A, let $|A|$ be the number of elements of A. *ImprovesMap(A)* is a function of a set of points in the roadmap that decides whether a new sample should be accepted to the roadmap or not.

Algorithm 1 Variable radius visibility algorithm

```
1: V ← {x_init}; E ← ∅; k ← 0;
2: while k < N do
3:     X_vis ← ∅;
4:     x_rand ← Sample();
5:     X_near ← Near(x_rand, V, r_RRT*(|V|));
6:     for all x ∈ X_near do
7:         if GenerateTrajectory(x, x_rand) succeeds then
8:             X_vis ← X_vis ∪ x;
9:         end if
10:    end for
11:    if ∅ == X_vis then
12:        X_near ← Near(x_rand, V, r_f);
13:        for all x ∈ X_near do
14:            if GenerateTrajectory(x, x_rand) succeeds
   then
15:                X_vis ← X_vis ∪ x;
16:            end if
17:        end for
18:    end if
19:    if ImprovesMap(X_vis) then
20:        V ← V ∪ {x_rand};
21:        for all x ∈ X_vis do
22:            E ← E ∪ {(x, x_rand), (x_rand, x)};
23:        end for
24:    end if
25: end while
```

Figure 3-1. Pseudocode for the variable radius visibility planner.

The first version of the visibility planner considered here (algorithms VRV and its map improvement condition) is simply the visibility planner with adaptive radius. The improvement condition for a sample to be accepted to the roadmap is: the sample should either be invisible to samples currently in the roadmap, or link two previously unlinked connected components (see Fig. 3-1 and 3-2).

The second version (algorithms IVY and its map improvement condition) changes the condition for sample acceptance. If the new sample sees only one node in the roadmap, then it is accepted, unlike the case of the classical visibility planner. The reason for the change is that allowing such samples may allow the roadmap to explore faster inside regions at the boundaries of the roadmap.

One can expect visibility planners, including the proposed versions, to explore narrow sections of the map faster than an RRT-like algorithm, because samples generated in narrow passages will only be rejected if they are redundant (the passage has already been mapped).

Algorithm 2 map improvement condition for VIS

```
1: function ImprovesMap(A)
2:     if ∅ == A then
3:         return true
4:     else if A containts nodes in different components
   then
5:         return true
6:     else
7:         return false
8:     end if
9: end function
```

Algorithm 3 map improvement condition for IVY

```
1: function ImprovesMap(A)
2:     if 1 ≥ |A| then
3:         return true
4:     else if A containts nodes in different components
   then
5:         return true
6:     else
7:         return false
8:     end if
9: end function
```

Figure 3-2. Map improvement conditions for versions of the variable radius visibility planner.

3.1.2 Probabilistic completeness

Define $S(\boldsymbol{x}, r)$ as the set of all points from C_{free} visible from $\boldsymbol{x}$ and that can be reached by local-planner generated edges of cost r or less; this set will be called the radius r snapzone of $\boldsymbol{x}$. Let $r_{RRT*}(n)$ be defined as in [Kar10]:

$$r_{RRT*}(n) = \left(\lambda * \frac{\log(n)}{n}\right)^{1/dim(C_{free})}$$

where n is the number of samples in the roadmap, and λ is some constant.

Let r_f be a fallback radius, which is kept constant. Define $S(\boldsymbol{x}, r_f)$ to be the snap zone of $\boldsymbol{x}$, and the union of snap zones of all vertices in the roadmap is then snap zone of the roadmap. These are the regions of the configuration space where, if one places a point, it can be connected to at least one vertex of the roadmap

The interaction between configuration space shape and local planner behaviour affects how visibility domains behave. Therefore, in this work we make a few assumptions about that interaction.

Then, let C_{free} be a **compact subset** of the configuration space. We also assume it is a metric space which also possesses a measure function, and that for all $\boldsymbol{x}$ in C_{free} and any positive real r, $S(\boldsymbol{x}, r)$ is a neighbourhood of $\boldsymbol{x}$ in the topology of C_{free}. Further, for any $\boldsymbol{x}$ in C_{free} and any $r_1 > r_2 > 0$, $Vol(S(\boldsymbol{x}, r_1)) \geq Vol(S(\boldsymbol{x}, r_2)) >$

0 and $S(\mathbf{x}, r_2) \subseteq S(\mathbf{x}, r_1)$; if the inclusion is strict, then $Vol(S(\mathbf{x}, r_1)) - Vol(S(\mathbf{x}, r_2)) \neq 0$.

Further, we assume in this section a symmetric visibility relationship (because the system is reversible and any trajectory can be taken in either direction), and that the shape of C_{free} remains constant in time.

(A technical point should be brought up: often, the shape of C_{free} is actually an open set, whereas the compactness assumption requires C_{free} to be closed. The issue is minor however, as one typically considers a C_{obs} region that was extended by some arbitrarily small amount, and one can use that to define a closed C_{free}.)

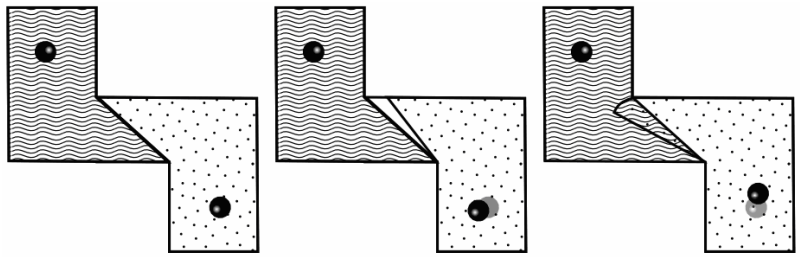

Figure 3-3. Wobbly free boundary condition. Moving a sample also moves the free boundary of its reachable set.

Finally, we make an assumption referred to as **wobbly free boundary**: let $\mathbf{x}$ be a point in C_{free}, with $S(\mathbf{x}, r)$ its snap zone of some arbitrary radius r, such that its boundary has points in C_{free}, and $\mathbf{x}'$ is another point in C_{free}, then almost surely $Vol(S(\mathbf{x}, r) \Delta S(\mathbf{x}', r)) \neq 0$ and $C_{free} \cap \partial S(\mathbf{x}, r) \cap \partial(S(\mathbf{x}, r) - S(\mathbf{x}', r))$ is open in the subspace topology of $\partial S(\mathbf{x}, r)$ (see Fig. 3-3).

An immediate consequence of the wobbly free boundary is that two points almost surely have snap zones that either have disjoint closures or intersect in a non-zero volume set, since the third possibility (only their boundaries intersect) is vanishingly unlikely. This is what allows overlap regions to be sampled; if the overlaps had measure 0, then they would almost surely never be sampled, and no sample-based planner, whether using visibility or not, would be able to connect a roadmap.

Consider first the situation that there exists a subset of C_{free} that has non-zero volume and such that, if a sample were generated there, it would be accepted by VRV (or IVY). Since samples are randomly uniformly distributed, a sample will eventually be generated in that subset. In particular, if the subset of C_{free} that is outside the r_f snap-zone of the roadmap has non-zero volume, a sample will eventually be generated there and added to the roadmap.

A consequence of the compactness of C_{free} is that a finite number of samples is sufficient to produce a roadmap whose r_f snap-zone is the whole of C_{free}; in fact, any (countably) infinite collection of samples whose r_f snap zone covers C_{free} contains a finite subset of samples which will cover C_{free} with their r_f snap-zones.

Suppose then that, having taken an infinity of sample-and-connect steps using the VRV (or IVY) algorithm, there still exists a subset of C_{free} outside the r_f snap-zone of the roadmap. That set may have volume zero, but that case is vanishingly unlikely by the wobbly free boundary assumption. Or, that set may have non-zero volume, however that implies that it was never sampled, which is

vanishingly unlikely because the samples are uniformly distributed. Therefore the supposition is false and hence, after having tried an infinity of sample-and-connect steps, the roadmap's r_f snap-zone covers C_{free} completely. However, by the compactness of C_{free}, out of that infinity of samples tried, there is some finite collection of them, the first N for some sufficiently large N which depends on the realization of the sampling process, which completely cover C_{free} and are included in the roadmap. The number of distinct connected components in the roadmap will therefore also be bounded on any run of the VRV (or IVY) algorithm.

As the number of samples in the roadmap increases, C_{free} is covered by expanding connected components in the roadmap. Suppose then that there are several connected components in the roadmap that are included in the same path connected component of C_{free}. As the volume of C_{free} outside the snap zone of the roadmap shrinks to zero, these connected components of the roadmap will "meet", meaning that it will eventually be the case that the snap zones of two different connected components will overlap. By the wobbly free boundary assumption, the overlap has non-zero volume. Further, the overlap region will never decrease, as samples, once inside the roadmap, are kept in place and neither the roadmap nor C_{free} change. Let $r_{RRT^*,m} = r_{RRT^*}(m)$, where m is the number of samples in the roadmap at the time step when the two components first had overlapping r_f snap-zones. Without loss of generality, assume that $r_f > r_{RRT^*,m}$ and that the $r_{RRT^*,m}$ snap-zones of the two components do not overlap. (These other cases can be handled easily as it's immediately clear that the components have non-zero volume overlaps of snap-zones.)

Note that the number n of vertices in the roadmap never decreases, both r_f and $r_{RRT^*,m}$ are constants, and that for all $n > m$, $r_{RRT^*,m} > r_{RRT^*}(n)$.

Consider then the event that two connected components with overlapping snap zones will never be connected, no matter how many subsequent sample-and-connect steps are undertaken. Since the overlap of the snap zones is non-zero volume, it will be sampled, infinitely often if the roadmap construction is allowed to proceed forever. Suppose that the two connected components will never be connected; that supposition implies that whenever their overlap is sampled, the sample is placed either in the $r_{RRT^*}(n)$ snap-zone of one component or the other. However, the $r_{RRT^*}(n)$ snap-zone of a component is included in that component's $r_{RRT^*,m}$ snap-zone, itself included in the component's r_f snap-zone. The latter inclusion is strict (the r_f snap-zones of the components overlap, but the $r_{RRT^*,m}$ ones do not), so there is a set of volume non-zero in the r_f snap-zone that is outside the $r_{RRT^*,m}$ snap zone. Then, there exists some non-zero volume overlap of the r_f snap-zones that is not included in the $r_{RRT^*,m}$ snap zone of either component (or else, the $r_{RRT^*,m}$ snap-zones would be disjoint but have intersecting closures- vanishingly unlikely by the wobbly free boundary assumption). The supposition that the connected components are never connected requires that this non-zero volume set never be sampled, a contradiction with the uniformly distributed nature of the sampling process. Therefore the supposition is false, and if two connected components grow to have overlapping snap-zones, they will eventually be merged into one component.

Finally, consider the case in which a path-connected component of C_{free}, let it be called D, has been covered by the snap-zone of the roamap, in such a way that there are several connected components in the roadmap that cover it. By the wobbly free boundary, almost surely none of the connected components that cover D is such that it is disjoint from all others, while having closures that are not

disjoint. Therefore, connected components in the roadmap that cover the same path connected component of C_{free} will eventually be merged.

Since there will be a finite number of connected components, which grow and/or multiply to fill the entirety of C_{free}, and since producing a sample that connects two of them is almost sure to happen eventually if the two connected components have overlapping snap-zones, all connected components of the roadmap that grow to have overlapping snap-zones will almost surely eventually be connected. By the wobbly free boundary assumption, any two connected components in the roadmap either have non-zero volume overlap of r_f snap-zones (and will almost surely be merged eventually), or they have disjoint closures. Also, as shown above, connected components in the roadmap that cover the same path connected component of C_{free} will almost surely be connected eventually.

If that happens, as it almost surely will, then consider any pair of initial and target points in C_{free}. If they are in the same path connected component of C_{free}, then they are in the snap-zone of one (and only one) connected component in the roadmap, and a path will be generated between them. If however they are not in the same path connected component of C_{free}, then there is no connected component of the roadmap that has both in its snap-zone, and no path is generated. Therefore VRV and IVY are probabilistically complete.

We observe also that at some point the VRV and IVY algorithms will no longer accept samples to the roadmap. This sample deadlock happens when the configuration space connectivity has been completely covered, by the roadmap.

3.1.3 Computational complexity

In this section, we will further assume that C_{free} is an Euclidean space, and further, if $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ in C_{free} and $r>0$ are such that $S(\boldsymbol{x}_1, r) \subset C_{free}$ and $S(\boldsymbol{x}_2, r) \subset C_{free}$, then $Vol(S(\boldsymbol{x}_1, r)) = Vol(S(\boldsymbol{x}_2, r))$. The Euclidean space assumption is present in other computational complexity analyses [KarARX].

Consider then that n points have been placed in C_{free} and let $r>0$ be such that for each $\boldsymbol{x}_k$ from among those points, $S(\boldsymbol{x}_k, r) \subset C_{free}$. No other assumption is made on the distribution of the $\boldsymbol{x}_{1..n}$ points. Let $v_r = Vol(S(\boldsymbol{x}_k, r))$. Let $\boldsymbol{x}_{n+1}$ be a new point, that is generated inside C_{free} by a random, uniformly distributed, process. Then it is easy to check that the expected value for the number of points from among $\boldsymbol{x}_{1..n}$ that are visible from $\boldsymbol{x}_{n+1}$ is $n*v_r/Vol(C_{free})$.

If we remove the assumption that the $\boldsymbol{x}_{1..n}$ points and value of r be such that for all $\boldsymbol{x}_k$, $S(\boldsymbol{x}_k, r) \subset C_{free}$, then we have that $Vol(S(\boldsymbol{x}_k, r)) <= v_r$. One obtains that the expected value for the number of points from among $\boldsymbol{x}_{1..n}$ that are visible from $\boldsymbol{x}_{n+1}$ is less than or equal to $n*v_r/Vol(C_{free})$.

In particular, we see that the expected value of nodes from the roadmap that are inside the $r_{RRT^*}(n)$ snap-zone of a new sample generated by a uniformly distributed process is bounded above by a number proportional to

$$n * \frac{Vol\left(B\left(r_{RRT^*}\right)\right)}{Vol\left(C_{free}\right)} \sim n * \frac{r_{RRT^*}^{dim\left(C_{free}\right)}}{Vol\left(C_{free}\right)} \sim \log\left(n\right)$$

where n is the number of vertices currently present in the roadmap.

A similar argument shows that the expected value of the number of vertices in the r_f snap-zone of a new uniformly distributed sample is bounded above by a number proportional to n.

The probability that a new sample fails to see any sample in the roadmap inside a $r_{RRT^*}(n)$ snap-zone is the ratio of the volume of the $r_{RRT^*}(n)$ snap-zone of the roadmap to the volume of C_{free}, and is difficult to estimate.

We will compare a sample-and-connect step of VRV and IVY, regardless of whether it results in the new sample being accepted to the roadmap or not, with a sample-and-connect step of the RRG [Kar10] algorithm. At step n, it will usually be the case that the number of samples in a VRV (or IVY) roadmap is less than or equal to the number of samples in the RRG roadmap, which is n. Nonetheless, we will consider that at step n, VRV and IVY have n samples in their roadmaps.

All of the considered algorithms have exactly one call to the sample procedure and all of them contain a number of calls to the Near procedure that is bounded above by a constant. To ascertain the differences in complexity, we will now look at the differences between the algorithms.

First, observe that unlike RRG, both VRV and IVY include connected component checks and possibly merges. There are methods for which the asymptotic complexity of these operations is proportional to $\alpha(n)$, the inverse of the Ackerman function $A(n, n)$ [Tar75], which is an extremely slow growing function. For example, $\alpha(61)$ is 3, and $\alpha(POW(2; POW(2; 2^{65536})) - 3)$ is 4 ($POW(a; b) = a^b$). In other words, for roadmaps of practical size, the asymptotic computational cost of the merging and component find operations can be considered proportional to a constant. Further, since the operations in the find/merge algorithms are simple table lookups and index assignments, one can safely neglect the impact of them in the complexity analysis of the sample-and-connect step as a whole.

The other difference results from the range of the Near query, and the number of calls to GenerateAndCheckTrajectory. In a "typical" step of VRV (or IVY), a Near search is performed with the same radius as in the case of RRG, yielding the same number of expected near vertices in the roadmap, and the same number of calls to GenerateAndCheckTrajectory (a number that is proportional to $\log(n)$). If however none of the attempted trajectories succeeds, VRV (and IVY) perform a second Near query and another set of calls to GenerateAndCheckTrajectory, a number that is expected to be proportional to n, the number of vertices in the roadmap. The probability of a second call to the Near procedure (and further GenerateAndCheckTrajectory attempts) is proportional to the ratio of the volume of the $r_{RRT^*}(n)$ snap-zone of the roadmap to the volume of C_{free}.

3.1.4 Simulation verification

We compare the performance of the two visibility planners presented with that of PRM by posing five maze solving problems that require finding a path between two specified points. "Wide" is a very simple maze, "Basic" is a regular perfect maze (a maze with no loops), "Intricate" is a more complex perfect maze, "Bug" contains a bug trap (a very narrow passages the robot must pass through from one great hall to another) and "Narrow" contains several narrow passages. The local planner generates straight-line trajectories.

The number of samples (both accepted and rejected samples in the case of the visibility planners) is tallied for each, and mean and standard deviation statistics calculated over one thousand runs of each planning method for each problem. The results are shown in Tables 3-1 and 3-2.

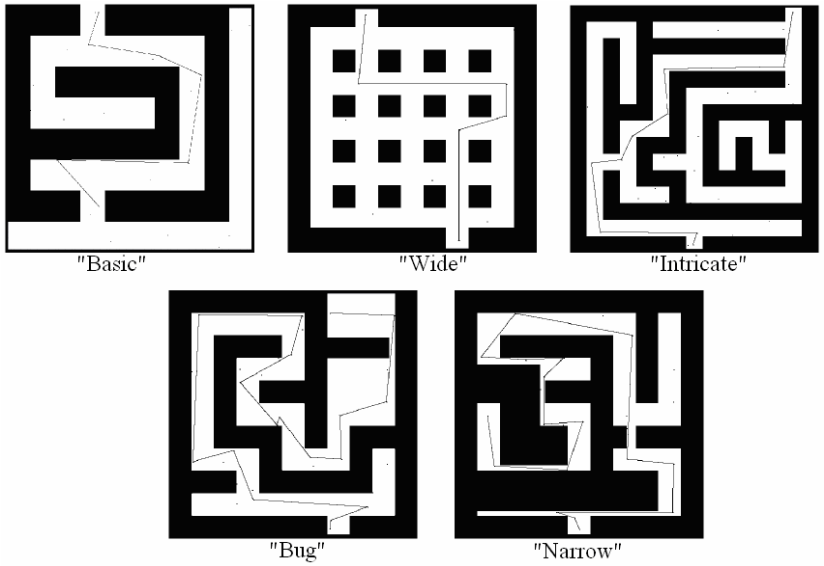

Figure 3-4. Mazes used for testing the planners, along with sample solutions.

Table 3-1. Average sample counts for the various planners. For visibility planners, averages are given as accepted + rejected samples.

Problem	Planner		
	PRM	VRV	IVY
"Basic"	51.86	14.59+56.85	18.24+40.23
"Wide"	17	11.43 + 14.76	12.83 + 5.59
"Intricate"	147	36.09 + 243.11	45.13 + 155.77
"Bug"	123.8	26.55 + 369.47	31.5 + 237.6
"Narrow"	249.98	29.92 + 387.99	38.04 + 305.99

Table 3-2. Standard deviation sample counts for the various planners. For visibility based planners, these are listed as accepted + rejected samples.

Problem	Planner		
	PRM	VRV	IVY
"Basic"	27.19	1.96+36.39	2.63+32.09
"Wide"	6.65	2.51 + 9.55	3.15 + 5.56
"Intricate"	58.39	4.39 + 194.28	5.02 + 110.57
"Bug"	57.05	2.89 + 305.59	3.58 + 237.73
"Narrow"	106.98	3.29 + 200.62	3.73 + 163.36

Both visbility based algorithms produce roadmaps that are very compact compared to those produced by a planner not using a visibility heuristic, and the number of samples kept in the roadmap is fairly constant throughout the planner runs. VRV tends to make roadmaps that use fewer samples than IVY, while IVY solves the planning problems with fewer sample and connect steps as the total (accepted plus rejected) is less for IVY.

Since both proposed variants limit the snap zones of nodes to be of at least a minimal fallback radius, which suggests each sample is guaranteed to cover at least some constant volume element of the configuration space, it follows that the number of samples needed to completely capture the connectivity of a space is still exponential in the number of dimensions of that space. Nonetheless, because of the constraint that a node should either improve coverage or connectivity when added, the number of samples that are kept in a visibility based roadmap will be lower than that of not employing this heuristic planners.

3.2 Applying sparse planners to nonreversible systems

In this section we apply the visibility based planner to problems in which not all movements are reversible; specifically, motion planning problems for a few simple vehicles. The local planners for these vehicles work by concatenating sequences of maneuvers, not all of which will be reversible in a given environment. We modify the visibility heuristic to handle the new situation, and present some simulation results. This section has content from the author's paper "Visibility based planners for kinematically constrained vehicles".

3.2.1 The new visibility heuristic

Since the existence of a trajectory from one state to another does not imply the existence of an easy return, the roadmap becomes a directed graph (also called a digraph). Therefore, a difference appears in the ImprovesMap heuristic. It is expected that the roadmap will contain several strongly connected components (indeed, a lone vertex may be a strongly connected component). As such, the new notion of "improves connectivity" means a new sample is linked by an inbound edge to a strongly connected component, and by an outbound edge to another, so that it allows a path to go from the first component to the second. It will not necessarily be the case that the sample will allow travel in the other direction; but if it did, then it would result in the two strongly connected components merging, if we impose a maximality condition on connected components.

Then the heuristic of deciding when a new sample is "useless" and should be discarded is this: either all the edges of that sample are in the same direction (it is a sink with all edges inbound, or a source with all edges outbound), or it is visible only from several vertices in the roadmap that are all located in the same strongly connected component. The first rejection criterion helps eliminate samples that the robot cannot leave from or get at; in particular, regions of inevitable collisions (because momentum is too great to be changed by the actuators in time to prevent a collision) are configurations of the "sink" type. The second rejection criterion

prunes unnecessary connections, similar to the visibility heuristic for reversible systems and roadmaps on unoriented graphs.

Algorithm 4 map improvement condition for the visibility based planner on digraph roadmaps

```
1: function IMPROVESMAP(X_vis, X_rev)
2:     if 0 == |X_vis| AND 0 == |X_rev| then
3:         return true
4:     else if 0 == |X_vis| OR 0 == |X_rev| then
5:         return false
6:     else if 1 < |X_vis ∪ X_rev| AND X_vis ∪ X_rev in one component then
7:         return false
8:     else
9:         return true
10:    end if
11: end function
```

Figure 3-5. Pseudocode for map improvement condition on directed graphs.

3.2.2 Non-zero dimensional sample subspaces

We also consider a modificaiton to planner implementations presented in the literature: whereas samples are usually points in C_{free}, we allow them to be non-zero dimensional subspaces of C_{free}. Distance functions must be changed accordingly, and the sampling procedure must be able to generate such subspaces and make sure they stay inside C_{free}. The details of the sampling procedure are given in the subsections describing the system models for each of the studied vehicles. The reason for this change is to ease the validation of maneuver sequences generated by the local planner. Storing some maneuvers which are known to be obstacle free allows their quick reuse later for validating other maneuvers.

The necessary condition imposed on a sample subspace is that it be a strongly connected subspace of C_{free}, meaning, for any two configurations in the subspace, the local planner can generate paths between them both ways, such that the paths do not exit C_{free}. If true, the condition implies that if one point in the subspace can be reached from a given configuration, then all are reachable.

3.2.3 Probabilistic completeness

The proof of probabilistic completeness follows similar steps to that of the reversible systems case. The assumptions are the same as in the reversible case: the free space should be compact, the visibility sets should be open, the sampling is a uniformly distributed process, and the local controller should have the wobbly free boundary property.

A difference to the reversible case is that we need to consider, for each vertex in the roadmap, two visibility sets: the set of points that can be reached from

that vertex, using trajectories generated by the local planner, and the set of points that can reach that vertex through trajectories generated by the local planner.

As opposed to the reversible case, where sampling deadlock (a condition where the planner will no longer accept samples to the roadmap) will occur, in the non-reversible case it may happen that two or more points in free space exist, which cannot be linked in a loop, and thus cannot be in the same strongly connected component. So unlike the reversible case, where the fact that sampling deadlock will happen if enough samples are taken, and corresponds to fully capturing the connectivity, we need to proceed slightly differently here.

Since the free space is compact, it follows, just as in the reversible case, that after some finite number of samples the free space will be covered by the reunion of the visibility sets. Also, because of the wobbly free boundary, there will be measure non-zero overlaps between the visibility sets.

Suppose then that two visibility sets overlap: the set of points reachable from some vertex **x**, with the set of points that can reach vertex **y**. Since the overlap is not measure 0, eventually a sample will be placed there. If **x** and **y** are already in the same strongly connected component, then the new sample isn't necessary because there are already connections from **x** to **y** and back. However, if **x** and **y** are not in the same strongly connected component, the new sample adds a link between them.

Suppose then that during roadmap construction, after the entire free space is covered by reachability sets of vertices, there is some region of free space **A** that is reachable from other parts of the free space, but there are no links from vertices outside of **A** towards vertices in **A**. Because of wobbly free boundary, the overlaps will eventually contain samples linking at least some vertices of **A** to other vertices in the roadmap, so the volume of free space that is reachable, but not yet connected to the rest of the roadmap, will shrink as sampling continues.

Therefore, by taking more sample and connect steps, the free space will eventually be completely covered, and all connections between different regions of it will be discovered, and therefore the chance of finding a plan, if one exists, increases with the number of sample and connect steps. Therefore, the planner is probabilistically complete.

As in the reversible case, we observe that a sample deadlock (planner accepts no new samples to the roadmap) happens when, and only when, the roadmap completely captures the connectivity of the configuration space. The rate at which samples are accepted to the roadmap can be used to provide an estimation of how much of the configuration space is covered.

3.2.4 Simulation verification

The vehicles simulated in this section all move in a two dimensional work space and will be maneuvered in such a way that kinematic models (position variables for state and continuous velocity controls) are enough to describe and plan their motion. The following subsections describe, for each vehicle, its model and steering procedure.

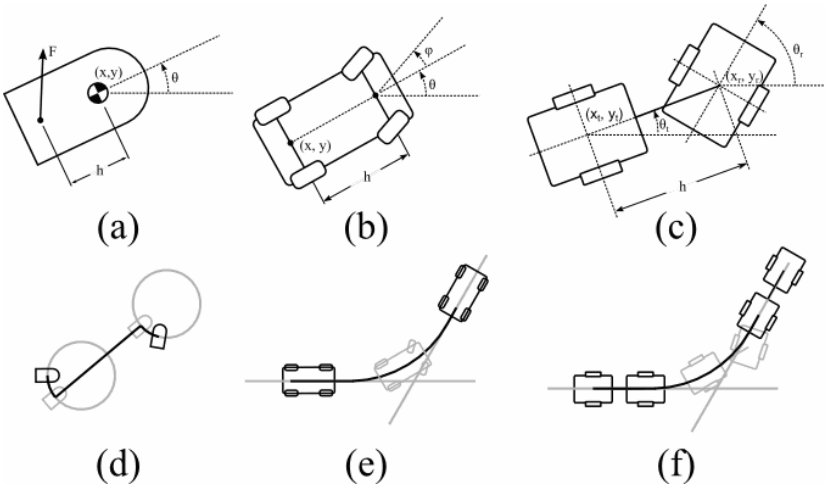

Figure 3-6. Vehicle models and basic maneuvers. (a), (d): planar object with variable direction thruster; (b), (e): car; (c), (f): robot with trailer.

Planar object with variable direction thruster

A two dimensional rigid body with a thruster located at some offset away from the center of mass (see Fig. 3-6 (a)). Its model is

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix}=\begin{bmatrix}\cos(\theta)\\ \sin(\theta)\\ 0\end{bmatrix}*u_1+\begin{bmatrix}-\sin(\theta)\\ \cos(\theta)\\ \frac{m*h}{J}\end{bmatrix}*u_2$$

where x and y give the position of the object's center of mass in the plane, while θ is the heading angle. J is the moment of inertia and m is the mass of the object, while h is the distance between the center of mass and the thruster. The thruster is assumed to be able to swivel from pushing the object to the left, to pushing it forward, to pushing it to the right. The thruster is assumed to not be able to push the object backwards.

Steering is done by noticing that the object model is kinematically reducible [Bul02] and that two decoupling vector fields exist:

$$\boldsymbol{X}_1=\cos(\theta)*\frac{\partial}{\partial x}+\sin(\theta)*\frac{\partial}{\partial y}$$

$$\boldsymbol{X}_2=-\frac{m*h}{J}*\frac{\partial}{\partial\theta}-\sin(\theta)*\frac{\partial}{\partial x}+\cos(\theta)*\frac{\partial}{\partial y}$$

Each field corresponds to a basic maneuver. $\boldsymbol{X}_1$ is simply the object moving in the direction given by the heading angle. $\boldsymbol{X}_2$ is the object rotating around a point located at a fixed distance in front of the thruster, in the direction given by the heading. The trajectory traced by an $\boldsymbol{X}_2$ maneuver is referred to as an $\boldsymbol{X}_2$ circle. The $\boldsymbol{X}_2$ maneuver is reversible (the object can spin in either direction in a given circle) but the $\boldsymbol{X}_1$ maneuver is not (the object cannot push itself backwards).

The procedure to steer between two given configurations starts by identifying the $\boldsymbol{X}_2$ circles associated to the two configurations, and computing the line between their centers. Then, an $\boldsymbol{X}_2$ maneuver on the source configuration is done to bring the object on that line such that it faces the destination's $\boldsymbol{X}_2$ circle. An $\boldsymbol{X}_1$ maneuver is then done to bring the object to the destination's $\boldsymbol{X}_2$ circle. Finally, another $\boldsymbol{X}_2$ maneuver brings the object to the destination (see Fig. 3-6 (d)). Each maneuver must start and end with the planar object at 0 velocity, so that they can be concatenated; the velocity of a real vehicle cannot vary discontinuously.

The motion planner uses arcs from $\boldsymbol{X}_2$ circles as samples. An arc is a valid sample if it does not intersect obstacles, and is generated thusly: a random configuration is selected in the workspace. If it is valid, meaning that it doesn't intersect obstacles, then the planner identifies its $\boldsymbol{X}_2$ circle and uses the largest arc of that circle that includes that randomly selected configuration and does not include configurations that collide with obstacles.

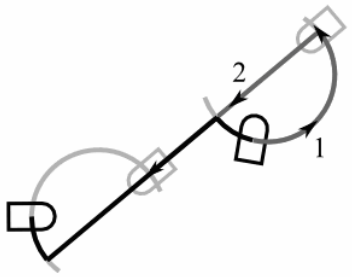

Figure 3-7. Reversing the planar object with variable direction thruster.

Notice that it is not necessarily the case that if one can get from a sample A to a sample B, one could also move backwards. Two way travel is possible only if the line that joins the centers of the samples' $\boldsymbol{X}_2$ circles intersects each sample's arc twice (see Fig. 3-7).

Planar car

A two dimensional car with steerable front wheels (see Fig. 3-6 (b)). Its model is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \dfrac{\tan(\varphi)}{h} \\ 0 \end{bmatrix} * u_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} * u_2$$

where x and y give the position of the car in the plane, θ is the heading angle and φ is the steering angle. h is the distance between the front and back axles. The car is assumed able to also go in reverse.

The car model is fully kinematically reducible [Bul10], but for it a different approach is illustrated, that of the maneuver automaton [Fra01]. A basic "trim"

maneuver is for the car to simply go in a straight line with direction given by the heading, with steering angle φ equal to 0. This trim maneuver will be referred to as a driving segment, and the states that the planner uses are such driving segments.

Generating a driving segment goes as follows: a random configuration is selected for the car in the workspace. If it is valid (it does not intersect with obstacles) then the longest drive segment that passes through that configuration and does not collide with obstacles is used.

Steering between two drive segments is done by a sequence of three maneuvers (see Fig. 3-6 (e)). First, the car moves toward the destination segment while quickly increasing its steering angle. Second, the car moves in a circular arc with constant steering angle, and the length of the arc is varied such that it will meet the destination drive segment properly after the third maneuver. The third maneuver is the car still moving forward and quickly decreasing its steering angle to 0. The forward velocity of the car is kept constant throughout all maneuvers.

The three maneuvers can be thought of as transformations that move an initial configuration $\boldsymbol{x}_{ini}$ to a final one $\boldsymbol{x}_{fin}$:

$$\boldsymbol{x}_{fin} = \boldsymbol{R}_{\varphi\downarrow 0} \cdot \boldsymbol{R}_{\varphi=0}(\theta_d) \cdot \boldsymbol{R}_{\varphi\uparrow max} \cdot \boldsymbol{x}_{ini}$$

The first and last maneuvers ($\boldsymbol{R}_{\varphi\uparrow max}$ and $\boldsymbol{R}_{\varphi\downarrow 0}$) result in fixed displacements in the configuration variables. It is the second ($\boldsymbol{R}_{\varphi=max}(\theta)$) that can be tuned, by making the arc longer or shorter, so as to change the heading of the car from the heading of the initial drive segment to that of the destination drive segment. Once the necessary θ_d angle for the arc is known, the displacement in position can be computed, and therefore one can determine the points on the two segments between which the steering maneuvers occur.

Since the car's maneuvers are fully reversible, if one can go from a given drive segment A to a given drive segment B, then one can always make the journey backwards. The direction the car is facing on a drive segment is unimportant, as it can do the same maneuvers independent of which way it is facing. Drive segments are then undirected.

Robot with trailer

A robot with a trailer attached to it via a rigid rod articulated at the robot (see Fig. 3-6 (c)). The configuration variables are x_r, y_r, θ_r for the robot position and heading, and likewise x_t, y_t and θ_t for the trailer. h is the distance between the trailer's and the robot's centers. Because the system is differentially flat, there are certain relations between the robot and trailer variables [Lam00]:

$$\begin{bmatrix} x_r \\ y_r \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} + \begin{bmatrix} \cos(\theta_t) \\ \sin(\theta_t) \end{bmatrix} * h$$

$$\tan(\theta_r - \theta_t) = h * \frac{d\,\theta_t}{ds}$$

where s is the natural parametrization of the curve traced by the trailer, or in other words the time variable if the speed of the trailer is equal to *1*.

Therefore, if one is given a trajectory for the trailer, one can deduce the trajectory that the robot needs to follow, and this fact is used for the steering procedure.

Just like in the case of the car, the planner uses drive segments as states, where drive segment here means a maneuver in which the robot and trailer are aligned and moving with constant heading. It is assumed that a drive segment is

reversible (the robot can either push or pull the trailer in a straight line). However, unlike the case of the car, it is important which way the robot with trailer is facing in a drive segment. Therefore the drive segment is directed, because maneuvers that make the robot leave the drive segment are not necessarily reversible.

Steering from one drive segment to another is done by imposing a circular arc trajectory for the trailer that joins points on the two drive segments (see Fig. 3-6 (f)). The circle is generated with a fixed turning radius. From the trailer trajectory the robot's trajectory can then be deduced using the formulae above.

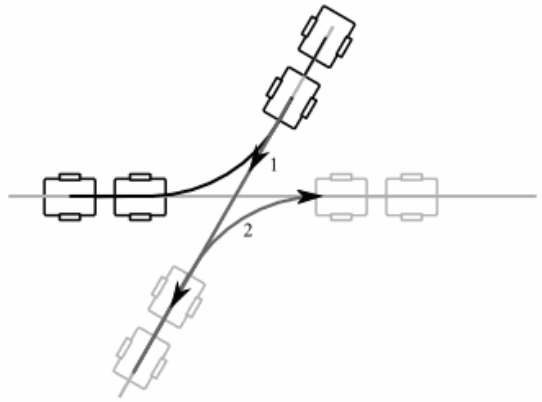

Figure 3-8. Reversing the robot with trailer.

Even if the robot with trailer can be steered from a drive segment A to a drive segment B, it doesn't necessarily follow that it can be steered from B back to A using the procedure above. In order for getting back to be possible, the robot must have enough room to reverse on the drive segment B, and enough room on segment A, to fit a circle arc maneuver between them (see Fig. 3-8).

Simulation results

The planar object with variable direction thruster is placed in a simple maze and required to go from a given start configuration to a target one (see Fig. 3-9 (a)). Ten runs are given to the algorithm, and it needs on average 4 vertices in the roadmap in order to be able to connect the given start and target configurations.

All vehicles are then placed in a maze containing "bug traps" (areas with narrow entrances, see Fig. 3-9 (b), (c), (d)), and given one hundred runs each. Average and standard deviation values for the number of samples stored in the roadmap before connecting start to target was possible is given in Tables 3-3 for the classical planner (no visibility heuristic) and in 3-4 for the visibility based planner.

The fully reversible vehicle, the planar car, needs the fewest samples in order to solve the planning problem. It's also the least sensitive to unlucky sampling. The other two vehicles tend to need more samples on average, but it also is the case that the number of samples used varies widely from run to run.

Of the two non-reversible vehicles, the robot and trailer needs fewer samples. This expected to be because its sample subspaces tend to be larger than those of the planar object: drive segments can be longer than $\boldsymbol{X}_2$ circle arcs, and thus offer more connection opportunities to other sample subspaces.

Table 3-3. Classical planner sample count statistics

	Problem		
	(b)	**(c)**	**(d)**
Avg	1770.6	14.71	239.25
StdDev	1697.28	12.86	196.82
Max	53	3	67
Min	7769	73	859

Table 3-4. Visibility based planner sample count statistics

	Problem		
	(b)	**(c)**	**(d)**
Avg	156.45	8.62	45.45
StdDev	141.11	3.79	29.61
Max	9	3	6
Min	630	21	128

Similar tests were run with the planners using point samples, which is the typical approach. While sample count statistics are similar, average execution times for the planners using point-samples are larger by a factor of two or more than the times for non-zero dimensional sample planners.

It can be seen that the variable radius visibility based algorithm resulted in compact roadmaps that could solve the given planning problems, for a set of vehicles and local planning strategies that account for the vehicles' kinematics, using fewer samples than a planner without the visibility heuristic. Besides using local planners adapted to models of real vehicles, our planner was also modified to use non-zero dimensional sample subspaces, a different approach than the classical sampling approach, so as to ease verification of maneuver sequences generated by local planning.

The effect of non-reversibility manifests in the algorithm needing more samples than it would for a reversible system, because the visibility-based sample selection heuristic is most efficient when the roadmap tends towards large connected components; ideally, only one connected component will cover the entirety of C_{free} when the roadmap is complete. With non-reversible maneuvers, this is no longer necessarily the case. The configuration space may contain irreversible passages between regions, and thus several strongly connected components. Also, maintaining maximal strongly connected components is more difficult than maintaining maximal connected components for undirected graphs; the algorithms are still polynomial time, but more expensive than the inverse Ackermann function complexity of Tarjan's set union that suffices for undirected graphs. If a heuristic to maintain strong connected components is used (as was the case here), then the visibility heuristic will not always be aware of components merging, and thus will be more permissive with accepting samples to the roadmap.

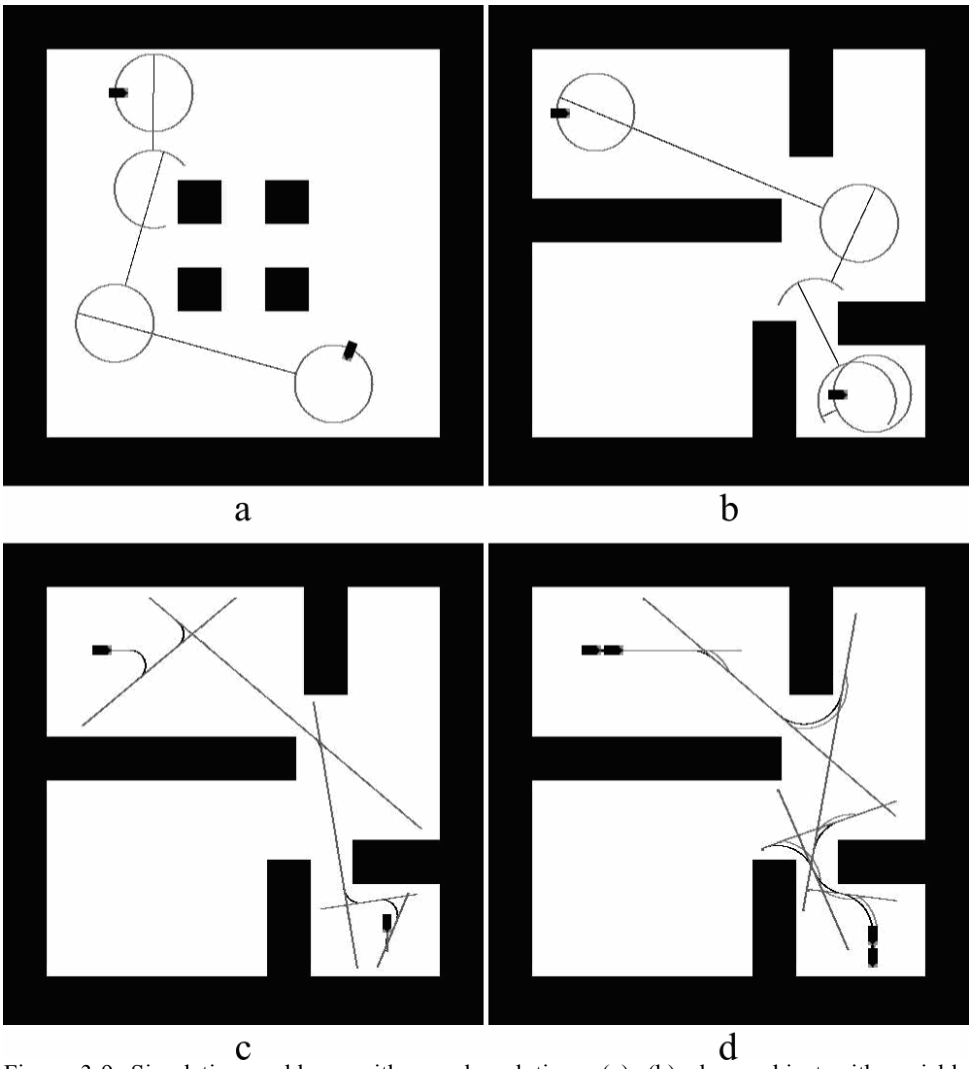

Figure 3-9. Simulation problems with sample solutions: (a), (b) planar object with variable direction thruster; (c) car; (d) robot with trailer.

It was also observed that "luck" while sampling plays an important part in determining how many samples are needed in the roadmap, since there is quite some variance in the number of samples needed to solve the given problems. It is suspected that the size of a sample subspace is a possible indicator of its "quality", and that larger sample subspaces will allow roadmaps with fewer samples to cover the configuration space. An investigation of such a possible quality metric is left for future work.

3.3 Path near-optimality of proposed solutions

The main reason to use a sparse or visibility planner is to have a small roadmap that nonetheless manages to capture the connectivity of the free space. The paths produced by such planners will be suboptimal, however good quality paths can be guaranteed, for various notions of near-optimality.

Sampling based planners that asymptotically converge to a globally optimal path exist. However, in order to achieve good quality paths, the planner needs to take many samples, which would be counterproductive when planning times should be kept small.

More recently, the notion of **asymptotic near-optimality** has been proposed, as a compromise between optimality and practicality. A planner is asymptotically near optimal if it can be shown that it will asymptotically converge to a path that is longer (or more costly) than the optimal one by a given factor k. For example, for k=2, the planner will asymptotically converge to a path that is at most twice the length of the shortest one.

It turns out that visibility heuristics can be softened, either by the inclusion of a graph diameter criterion or a useful loops criterion, or both, and these will guarantee asymptotic near-optimality. Of course, for the planner to keep converging, it needs to keep adding samples to the roadmap, whereas we'd like to keep as few samples as possible, so that the roadmap is quick to reuse. A planner seeking to converge to a global optimal, or near optimal, solution needs to sample everywhere, and as a result, convergence may in general be slow.

In contrast, local optimization of a path, while not an easy problem, is easier than trying to optimize globally. Assuming the planner produces some solution path candidate, of whatever quality, it will be possible to apply some deformation procedures to the path and reduce its cost or length. Indeed, planning software typically has a path simplification step to post-process the results from a sampling planner. Single-query planners like RRT or RRTConnect tend to produce very long and windy paths otherwise, which makes path simplification very important for them. Some multi-query algorithms like PRM, which keep large roadmaps, can produce good quality paths without simplification, but the planners we discuss in this work aim to keep few samples in the roadmap. Local path optimization is then a natural choice for our visibility-based planners too, because it only needs to analyze the candidate path neighborhood, and does not require one to continue to grow a roadmap.

The price paid for local optimization is a restriction on what paths can be found. Specifically, there is no guarantee that a local optimization procedure can produce a path outside the homotopy class of the initial path candidate. A path's homotopy class is the set of all paths that can be continuously deformed into it (and therefore into one another).

The planner should then have some way of exploring the various homotopy classes existent in the free space, or at least the homotopy classes that do not wind around the obstacles, as these are the classes where optimal solutions will likely be found (winding around an obstacle usually means the path is longer than it should be, and hence it will not be optimal). Since homotopy groups tend to be difficult to compute, guaranteeing that even the non-winding classes all have at least one representative in the roadmap turns out to be a difficult problem. However, a good heuristic exists, that of 'useful loops' [Nie04], which we will employ in chapters 4 and 5 when using sparse planners.

To decide whether to keep a new sample in cases where the visibility heuristic would decide to reject it, the useful loop heuristic selects two vertices already in the roadmap that can both be connected to this new sample; the selection can either be random, or simply the closest two vertices, from the visibility set of the sample. If there is some point on the shortest path between the vertices that is not in the visibility set of the new sample, then it is kept.

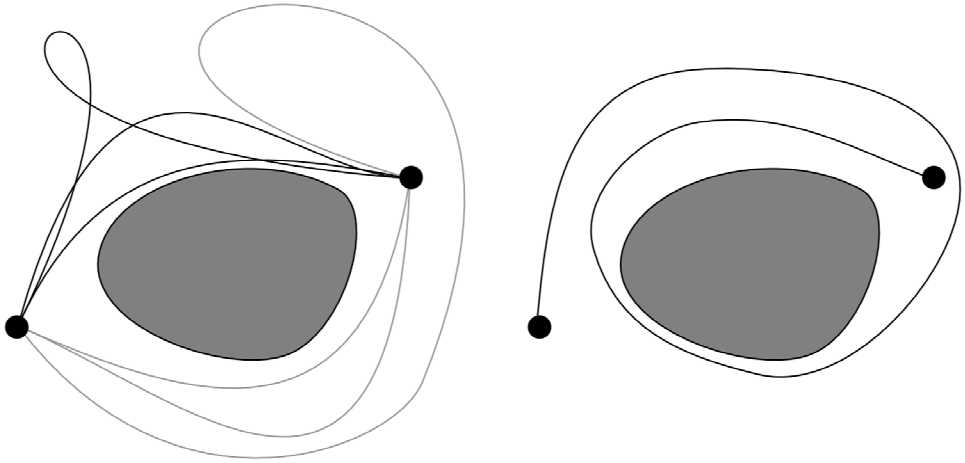

Figure 3-10. Homotopy classes are sets of curves that can be deformed continuously into one another. Left: black paths are in the same homotopy class, gray paths are in another. Right: the winding path is in a different homotopy class from those on the left.

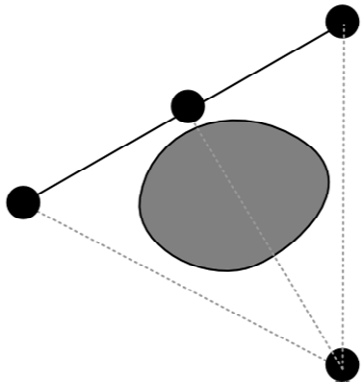

Figure 3-11. "Useful loops" heuristic: path between neighbors of new sample contains unreachable points from the new sample. Therefore, a loop around an obstacle is created by the new sample.

An often used approach for improving path quality is to locally refine a trajectory once it has been obtained from the planner. We describe a simple procedure below.

Let there be a current best path between a start and goal configuration, produced by the planner. Then at each step the algorithm will either produce a new

vertex halfway on an edge in the path or select one of the vertices already present between but not including the start and goal and perturb it randomly. If $\boldsymbol{x}_k$ is the original vertex and $\boldsymbol{x}'_k$ is the perturbed one, let $\boldsymbol{x}_{k-1}$ and $\boldsymbol{x}_{k+1}$ be the vertices immediately before and after $\boldsymbol{x}_k$ in the path. If both $\boldsymbol{x}_{k-1}$ and $\boldsymbol{x}_{k+1}$ are visible from $\boldsymbol{x}'_k$ and the sum of the costs of the edges ($\boldsymbol{x}_{k-1}$, $\boldsymbol{x}'_k$) and ($\boldsymbol{x}'_k$, $\boldsymbol{x}_{k+1}$) is smaller than the sum of the costs of the edges ($\boldsymbol{x}_{k-1}$, $\boldsymbol{x}_k$) and ($\boldsymbol{x}_k$, $\boldsymbol{x}_{k+1}$), then $\boldsymbol{x}'_k$ replaces $\boldsymbol{x}_k$ in the path.

To justify why this will eventually converge to an optimal path, consider the requirement that the path between start and goal be of minimal cost. This corresponds to a problem of minimizing the functional:

$$L = \sum_{k=1}^{n} c(\boldsymbol{x}_{k-1}, \boldsymbol{x}_k)$$

where c($\boldsymbol{x}_{k-1}$, $\boldsymbol{x}_k$) is the cost of the corresponding edge and *n* is the number of vertices on the path.

Suppose $\boldsymbol{x}^*$ is a function that maps the positive integers to points in the configuration space such that $\boldsymbol{x}^*_0$ is the start, $\boldsymbol{x}^*_n$ is the goal, and the value of *L* is smaller on this path than on any path with the same number of vertices. Consider then a perturbed function, $\boldsymbol{x}^{\sim} = \boldsymbol{x}^* + \varepsilon * \chi$ where ε is a real scalar, and χ is any perturbation. Since $\boldsymbol{x}^*$ is the path of n vertices which minimizes *L*, one expects the following condition to hold:

$$\frac{d}{d\,\epsilon} L(\boldsymbol{x}^* + \epsilon * \boldsymbol{\chi})|_{\epsilon=0} = 0$$

Performing the derivation, and grouping together terms multiplied by the same index of χ and remembering that the sum must be null for any χ, results in the following condition on $\boldsymbol{x}^*$ which is known as the discrete Euler-Lagrange equation:

$$\frac{\partial}{\partial \boldsymbol{x}_k} c(\boldsymbol{x}^*_{k-1}, \boldsymbol{x}^*_k) + \frac{\partial}{\partial \boldsymbol{x}_k} c(\boldsymbol{x}^*_k, \boldsymbol{x}^*_{k+1}) = 0$$

Suppose then that for a given path from start to goal with n vertices, selecting and perturbing any node inside the path will produce a worse cost sum to/from its neighbours. In that case, the path is already the best possible path with n nodes in its neighbourhood. (To be more rigurous, the edge cost function should be such that edges crossing obstacles be evaluated as having infinite cost, and that there be an arbitrarily small radius such that the cost of an edge increases rapidly as it approaches an obstacle at distances smaller than this radius.)

One can usually make the following assumption on paths and their costs: if some perturbations of one vertex from the current path will yield a better cost to/from its neighbours, the perturbations also form a non-zero volume set, therefore they will eventually be sampled and the path tends to become the locally optimal path with its number of nodes.

The step where vertices are produced along the edges in the currently known path is justified by the fact that paths with more vertices can approximate an optimal path in continuous space better. The method described above then will tend to increase n (the count of vertices in a path) from time to time, as well adjust vertices to converge to the best paths of length n.

Typically, samples produced by post-processing/local path optimization are not retained in the roadmap. When post-processing is done for a single-query planner, there is no persistent roadmap to store the vertices to. In the case of our visibility based planners, though we do use path optimization, we will also decide in later chapters to not store samples obtained from post-processing. The reason for

the decision is that the roadmaps will typically contain some useful configurations and trajectory segments which should better not be perturbed in unpredictable ways by local path optimization, or function as landmarks for other heuristics about the configuration space and as such roadmap vertex positions should stay constant. We will not handle changes in the environment by roadmap deformation, but rather in a way to be described in chapter 4.

3.4 Conclusions

In this chapter we provided a new proof of probabilistic completeness for visibility planners applied to reversible systems. We clarify the necessary conditions for probabilistic completeness on the interaction between the local trajectory planner and the shape of the free space. The proof we give is more general than the one given in [Nis99], which applies only to local planners that generate linear trajectories for point robots.

We then extend our proof to systems that have non-reversible maneuvers and/or non-holonimc constraints. We show that a suitably modified visibility heuristic remains probabilistically complete in this case, and we propose such a modification to the visibility heuristic.

We propose the use of non-zero dimensional samples in a roadmap as a way to speed up construction. We verify this approach in simulation, where it is shown that roadmap construction is sped up by a factor of two or more.

We verify our planners in simulations for both the reversible and non-reversible cases. They prove capable to generate more compact roadmaps than classical sample based planners (PRM) when capturing the configuration space's connectivity.

4. Handling changes in the environment

The environment in which a robot works changes over time, often due to actions not controlled by the robot, so the robot must detect and cope with these changes and adjust its behavior. Planning, in particular, is strongly affected by changes to the workspace. If the robot had a roadmap to describe the connectivity of the workspace a few minutes ago, that roadmap may now not be valid. Places that the robot thinks are free have become occupied by obstacles, or places once thought disconnected can now reach each other because some obstacles moved or disappeared. Nonetheless, most often the changes to the robot's environment are gradual. If a roadmap was useful and correct a minute ago, at least parts of it may still be useful now. In this chapter, we study ways to eficiently reuse roadmaps in changing environments. We present, for comparison, a naive approach as a baseline, then a refinement existent in the literature called "lazy PRM"[Boh00], and our own cost bump method. We show our cost bump method to be able to improve on planning efficiency compared to the other approaches. It also performs better than single-query methods, which are the ones most often used for changing environments. This chapter contains material from the author's paper "Motion planning for manipulators in dynamically changing environments using real-time mapping of free space" [Pom13b].

4.1 Previous approaches for handling changes

The problem with multi-query planners when used in changing environments is the roadmap does not necessarily describe connectivity after a change. Some vertices or edges in the roadmap might now pass through obstacles and should not be used. Or, some disconnected components should not be disconnected because now a passage exists between them that the roadmap does not contain.

Assuming one nonetheless wishes to reuse the roadmap (as is our case), then some roadmap update procedure is necessary. The simplest, "naive" approach is to revalidate the entire roadmap: check each vertex, then check each edge, for obstacle collisions. If a roadmap element collides with an obstacle, it is marked as unusable, perhaps by way of some flag bit.

The naive approach is however very inefficient. The number of vertices in a roadmap often exceeds the hundreds. The number of edges meanwhile is often of the same order of magnitude as the square of the number of vertices. Not only are edges very numerous, they are also difficult to check, as they require several collision checks along the way, or some kind of continuous collision checking on solids of revolution.

So a naive approach would require several tens of thousands of collision checks. However, a single-query planner like RRTConnect[Kuf00] would often be able to solve a planning query with only a few hundred. The workspace of the robot is actually often "simple", in that there's a lot of free space for the robot to move in, so placing new random samples will likely result in quick connections from start to goal. One can see then that RRTConnect (or some single-query method like it) would be more efficient than the naive complete roadmap revalidation. Further,

checking the entire roadmap before planning may in fact be self-defeating: since the checks take so long, by the time they complete, the environment might have changed again.

Since the naive approach is unworkable as-is, it needs improvement. One such improvement is to delay collision checking of a roadmap segment until it is actually necessary; this approach is known as "lazy PRM"[Boh00].

The method is still fairly simple. Assume there is a roadmap for the environment. All vertices and edges are assumed valid at first. When given a planning query, the planner links the start and goal states to the closest k (for some value of the parameter k) vertices in the roadmap, and runs a shortest path algorithm like A* to find a solution path candidate.

Only now is collision checking performed, and only on the vertices and edges in the path candidate. If everything is valid, the path candidate is returned as the solution. If some vertex or edge is invalid, then a flag bit is flipped and the offending vertex or edge is marked unusable. The shortest path algorithm is run again (with the understanding that it will treat invalid vertices or edges as if they did not exist, and will therefore not use them), and a new path candidate is produced. The process is repeated until either a solution is found, or some planner termination condition happens (typically, this is a time condition: the planner is only allowed a few seconds to find a solution).

Lazy PRM greatly reduces the number of necessary collision checks. However, in practice, it is still less efficient than RRTConnect. We tested this by giving a PR2 (Willow Garage's Personal Robot 2, a two armed manipulator with mobile base) five planning problems for the right arm, and running each problem for one hundred times. We collected planning times for each problem and each run, and computed averages and standard deviations which can be seen in the next table. The roadmap for lazy PRM was constructed by a sparse planner implemented in OMPL SPARS2, for an environment containing nothing except the robot; the problems we run have other obstacles as well. The resulting roadmap is fairly small (357 vertices), but it still manages to describe the connectivity of the arm's configuration space as well as describe good quality paths.

Table 4-1. Planning time statistics for RRTConnect and lazy PRM

Problem	RRTConnect		lazy PRM	
	Avg. time (s)	Std. dev. (s)	Avg. time (s)	Std. dev. (s)
0	0.205	0.095	0.311	0.001
1	0.170	0.091	0.152	0.001
2	0.202	0.079	1.157	0.005
3	0.059	0.034	0.233	0.001
4	0.095	0.060	0.113	0.001

The increased planning times for lazy PRM are a direct effect of it needing more collision checks than RRTConnect (more than 95% of the time for both algorithms is covered by collision checking, as measured with the Profiler class of OMPL). While more deterministic in terms of running times, and often capable to produce better quality paths, lazy PRM still appears the less practical choice if compared to RRTConnect.

What lazy PRM attempts to do is to get the best path (that the roadmap contains) which is still valid between start and goal. However, if the first solution path candidate fails, meaning, the shortest path is not actually feasible, then lazy PRM will attempt to find the second best path. Often, the second best path is not

that different from the first, because it passes through more or less the same regions. It might therefore pass through the same obstacles that invalidated the best path, and be unfeasible too.

So while RRTConnect can try any shape of random paths, and find one quickly because the environment is often simple and placing a random path is likely to result in success, lazy PRM slowly moves away from an invalid optimal path. It loses efficiency by being biased towards path optimality.

4.2 The cost bump method

To counter this bias, we propose a "cost bump": when a vertex or edge is found invalid, then the costs of other vertices in the roadmap are increased by some value depending on the distance to the invalid vertex (or edge). We choose the following formula for the cost bump:

$$c_b(\mathbf{x},\mathbf{p}) = \frac{q}{1+\left(\frac{\|\mathbf{x}-\mathbf{p}\|}{r}\right)^2}$$

where c_b is the cost bump to be applied to a vertex **x**, **p** is the point (vertex or position along an edge) found invalid, and q and r are parameters that control the shape of the cost bump function: its height and fall-off, respectively. These parameters should be tuned so that the bump is high close to the center, with quick fall-off as distance increases. An intuitive justification for the cost bump is, if a vertex is invalid (so, inside an obstacle) then vertices near it are probably invalid too.

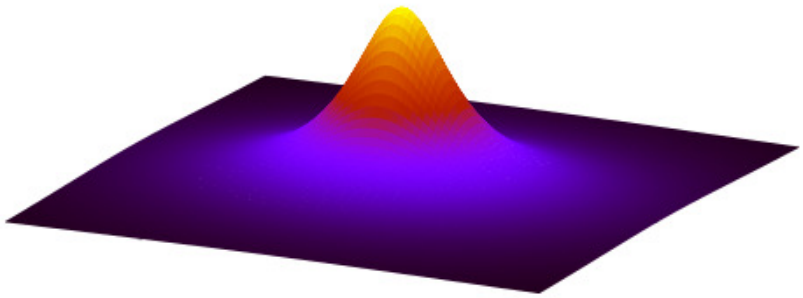

Figure 4-1. Shape of the cost bump function.

The pseudocode for the cost bump method is given in the next figure. Note that in our version, we apply the cost bump to all vertices in the roadmap. The roadmaps we use are constructed by visibility or sparse methods, and are therefore small. Computing the cost bump is then a cheap operation, even if applied to all vertices. Alternatively, one could instead do a nearness query first, to restrict cost bump computation to only the vertices inside a sphere of a given radius around the invalid point.

The above pseudocode starts with a preconstructed roadmap G, which is assumed fully valid at the start of a planner run. All vertex costs are zero, all vertices are usable by the computeShortestPath algorithm. Obstacle collisions

remove pieces of the roadmap if necessary, by flagging them unusable. Vertices receive cost bumps when collisions are detected.

Algorithm 1 LazySolve (G, q_{start}, q_{goal})

1: $startNear \leftarrow \text{NearestNeighbors}(G, q_{start}, r)$
2: $goalNear \leftarrow \text{NearestNeighbors}(G, q_{goal}, r)$
3: **for** $q \in startNear$ **do**
4: addEdge$(G, q_{start}, q, |q_{start} - q|)$
5: **for** $q \in goalNear$ **do**
6: addEdge$(G, q, q_{goal}, |q_{goal} - q|)$
7: $G' \leftarrow G$
8: **while** time remaining **do**
9: $shortestPath \leftarrow \text{computeShortestPath}(q_{start}, q_{goal}, G')$
10: **if** pathIsValid$(shortestPath, collisionState)$ **then**
11: **return** $shortestPath$
12: **else**
13: $G' \leftarrow G' - \{collisionState\}$
14: **for** each $q \in G'$ **do**
15: $q.cost \leftarrow q.cost + c_b(q, collisionState)$
16: **return** no solution

Figure 4-2. Pseudocode for planning with cost bumps.

Note that vertex and edge removal are temporary; future planning runs, using the algorithm presented above, will all start with the same fully valid roadmap. All cost or validity information discovered in one planning run is discarded before the next planning run begins.

4.2.1 Roadmap construction

In this work we use roadmaps constructed offline; the construction process need not be time-efficient therefore, as long as it produces a good quality roadmap. Good quality here means that the roadmap is small (has few vertices and few edges) while at the same time being rich enough to capture a wide variety of behaviors of the robot. Since we want the roadmap to be useful in changing environments, some redundancy in the trajectories between points is recommended. Rather than a simple visibility heuristic, which tries to minimize the number of connections in a roadmap, we use a visibility heuristic augmented with some considerations of path near-optimality as described in the previous chapter, section 3.3. The robot will then know of several ways to move between points in the roadmap, so that if one will fail in an actual environment, there will still be other options to consider.

During roadmap construction we are interested in what the robot can do, and how well it can do it: what points it can reach, and how long are the paths between those points. Putting obstacles inside the planning environment while we construct the roadmap is counterproductive; they will simply prevent us from getting a good picture of the robot's maneuvering capabilities.

Therefore, we construct the roadmap in an almost empty environment. The only obstacle is the one guaranteed to be forever present: the robot itself. Our roadmap filters out self-colliding maneuvers and configurations.

In actual use, the environment will contain other obstacles as well, not just the robot. Depending on how these obstacles are placed, various vertices or edges in the roadmap will be unusable or avoided.

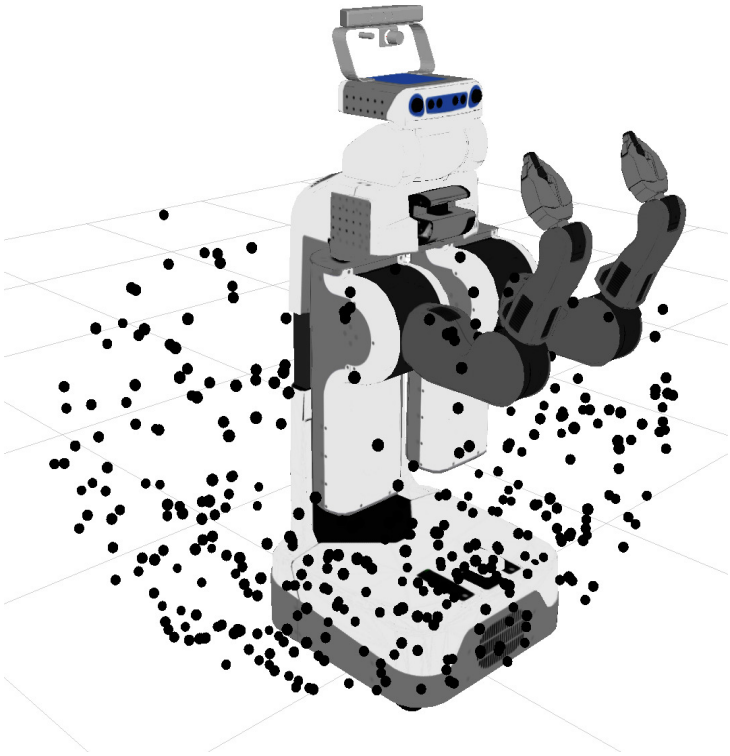

Figure 4-3. The sparse roadmap for the PR2 right arm. Each end effector position is represented by a black dot.

For this study, we constructed a roadmap for PR2's right arm using OMPL SPARS2. The roadmap contains 357 vertices, and the end effector positions corresponding to the vertices' configurations are shown in Fig. 4-3.

4.2.2 Cost as a way to learn the environment

In the previous subsections, we described a method to adjust a roadmap during a planner run so as to avoid regions that might be obstacles; however, we did not keep the cost or flag information from one planning run to the next. However, in the same spirit of reusing as much work as we can (which is why in this

thesis we argue for multi-query planners), we should try to reuse the vertex costs too.

Since environment changes can either increase or reduce the number of vertices passing through invalid regions, we also need to reverse cost bumps. We define a "cost unbump" function using the same formula as for the cost bump, but apply it as a reduction to vertex costs. Also, cost unbumps cannot take a vertex's cost below 0 (or an edge's cost below its natural cost).

We apply cost unbumps when a vertex is found to be valid. The intuitive justification is similar to the one for the cost bump: if a vertex is valid, then vertices close to it are likely valid too.

Vertex costs will then track the distribution of obstacles in the environment. We obtain a representation that is tailored for the planner and maintained by it, updated whenever necessary during planner queries. Note that the representation we maintain is of the configuration space of the robot, which has as many degrees of freedom as the robot has joints (seven, for the PR2 arm). Other data structures, for example octrees as implemented in the software package Octomap, keep representations of the three dimensional task space of the robot: which voxels/cubes in the task space are occupied by obstacles. A single occupied voxel in the task space however may have complex effects on the shape of the free space for the robot, which is why we need our representation, created by the planner.

We can also reuse vertex and edge validity flags from one planner run to another, as long as there is a way for the flags to be reset. Since the planner will ignore vertices and edges if they are marked invalid, then some other, background thread should run periodic collision checks on flagged vertices and edges in between planner runs. In this work however we simply reset validity flags once a planner run is over.

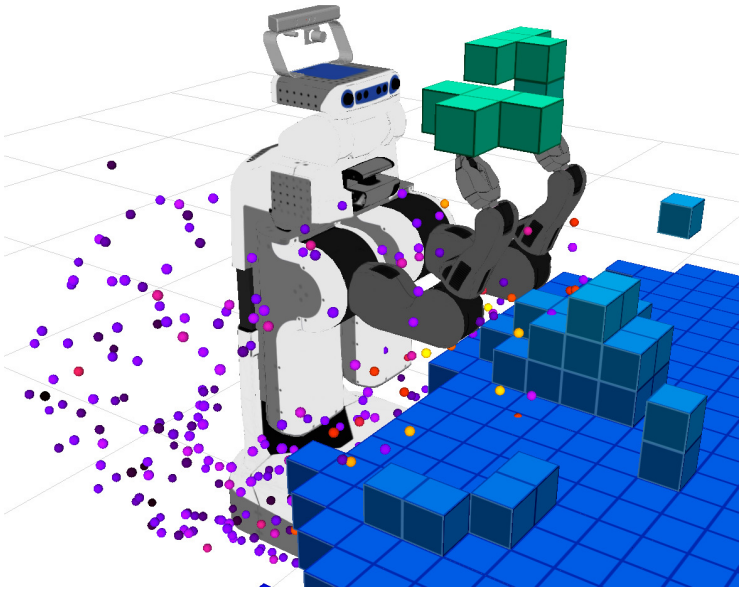

Figure 4-4. Vertex costs (black and purple indicate low values; oranges and yellows indicate high ones) in a planning environment after a query.

4.3 Fallbacks

It is possible that, during a planning run, enough vertices and edges are marked invalid that there is no longer any path in the roadmap from start to goal. Such events can be detected, because computeShortestPath also creates a list of vertices reachable from the start vertex. If the list does not contain the goal, the roadmap has become too disconnected to contain a solution and the planner can terminate early. While our cost bump method is therefore faster at finding solutions than single-query planners, and often produces better quality paths (as will be shown in the next section), it does so with a penalty to reliability: by relying on a roadmap, it becomes sensitive to the roadmap being so different to the current environment so as to be unusable. In practice, this does not happen often, but one would still want some kind of fallback for the times when it does.

One can increase reliability by running a single-query planner in parallel with our roadmap based method, and return the first valid solution (found either by the single-query or the roadmap based planner). This way, one obtains the reliability and performance of a single-query planner as a baseline, with the bonus that the roadmap based planner is also available to speed up planning computations, especially once roadmap costs capture a good representation of the current state of the obstacles.

4.4 Simulation verification

We used The Open Motion Planning Library (OMPL)[Şuc12] and MoveIt![Mov12] for the implementation.

Figure 4-5. The PR2 and the environment used for planning.

For a first set of tests, we ran our planner and RRTConnect on a set of planning queries (different from the problems in subsection 4.1) and recorded execution times and path lengths. The queries are planning problems requiring the manipulator to move around a table (taking the end effector from beneath the table to above it, for example) as well as around objects on that table. Our planner was run for 30 times for each problem. Every planner run started from zero vertex costs, and the planner had to rediscover the environment. Because its results show more variation, RRTConnect was run 100 times for each problem. Averages and standard deviations of the results from these runs are available in Tables 4-2 and 4-3 for execution times and path lengths respectively. Box plots are given in Fig. 4-6 and 4-7.

Table 4-2. Planning time statistics for RRTConnect and our method

Problem	RRTConnect		our method	
	Avg. time (s)	Std. dev. (s)	Avg. time (s)	Std. dev. (s)
0	0.102	0.057	0.065	0.003
1	0.075	0.053	0.024	0.002
2	0.292	0.309	0.054	0.003
3	0.331	0.270	0.128	0.003
4	0.201	0.143	0.136	0.002

Table 4-3. Path length statistics for RRTConnect and our method

Problem	RRTConnect		our method	
	Avg. path len.	Std. dev. path len.	Avg. path len.	Std. dev. path len.
0	7.314	2.895	4.553	0.000
1	5.205	1.900	3.606	0.000
2	7.597	3.353	3.826	0.000
3	8.626	3.238	5.224	0.000
4	9.976	3.068	9.420	0.000

As the table and plots reveal, the proposed planner is capable of finding better quality paths faster than RRTConnect, sometimes twice as fast or better. The performance of the planner is dependent of course on the sparse roadmap used, which should be small enough to enable fast queries, but large enough to capture manipulator movements that would allow it to move gracefully in a cluttered environment.

For a second set of tests, we keep the vertex cost values from one planning problem to another. We run our planner 30 times for each problem, and each planner run starts from the same initial vertex costs. However, the vertex costs at the end of the last run for a problem will become the initial vertex costs for all runs of the next planner problem. To compare, we use RRTConnect, which we run for 100 times for each problem. We again collect averages and standard deviations of planning time and path length. Boxplots for planning time are shown in Fig. 4-8; statistics for planning time and path length are also shown in Tables 4-4 and 4-5 respectively.

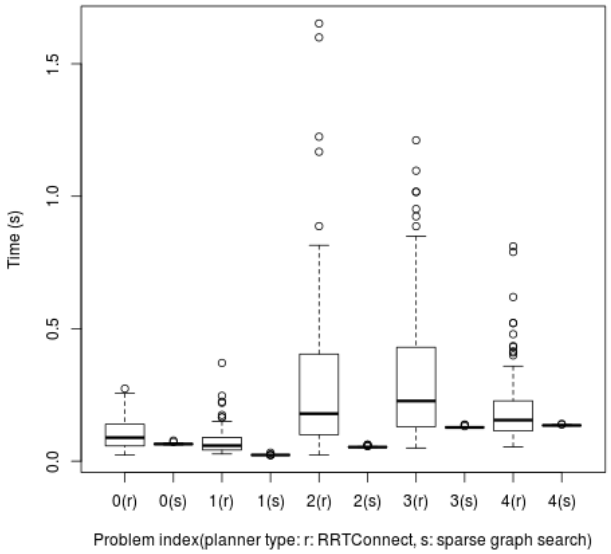

Figure 4-6. Planning time boxplots for RRTConnect and our method.

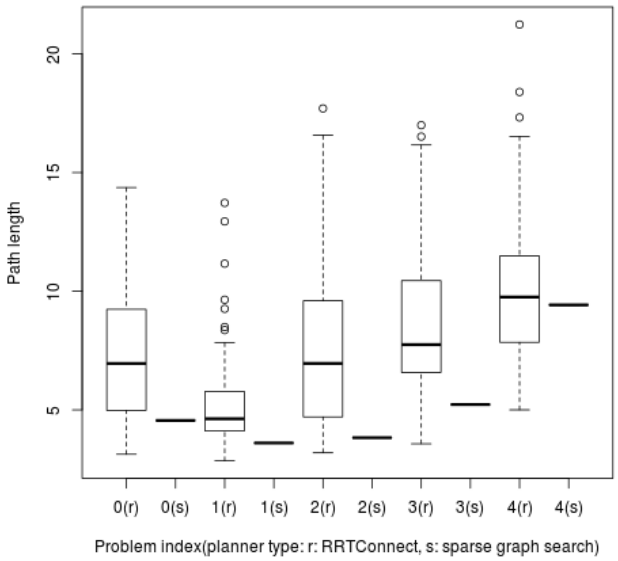

Figure 4-7. Path length boxplots for RRTConnect and our method.

Table 4-4. Planning time statistics for RRTConnect and our method. Vertex costs in one problem run are initialized with the costs from the previous problem

Problem	RRTConnect		our method	
	Avg. time (s)	Std. dev. (s)	Avg. time (s)	Std. dev. (s)
0	0.174	0.072	0.082	0.001
1	0.242	0.200	0.097	0.002
2	0.157	0.075	0.127	0.001
3	0.165	0.079	0.252	0.012
4	0.173	0.080	0.142	0.003
5	0.209	0.198	0.054	0.002
6	0.197	0.183	0.136	0.002
7	0.181	0.078	0.103	0.002
8	0.163	0.069	0.083	0.002
9	0.097	0.092	0.048	0.002
10	0.207	0.169	0.158	0.001
11	0.284	0.180	0.158	0.002

Table 4-5. Path length statistics for RRTConnect and our method. Vertex costs in one problem run are initialized with the costs from the previous problem

Problem	RRTConnect		our method	
	Avg. path len.	Std. dev. path len.	Avg. path len.	Std. dev. path len.
0	10.644	2.563	6.861	0.000
1	7.910	3.321	6.552	0.000
2	10.253	2.524	7.959	0.000
3	10.461	2.244	10.008	0.000
4	10.984	2.808	10.008	0.000
5	6.361	2.887	2.938	0.000
6	7.250	3.731	3.525	0.000
7	11.350	2.564	9.796	0.000
8	10.876	2.505	5.725	0.000
9	5.599	2.375	4.212	0.000
10	6.977	2.534	9.977	0.000
11	10.406	3.342	7.777	0.000

Again we can see our planner is usually faster than the RRTConnect average and median. One exception is problem 3. Problem 4 is the same start/goal state pair, and our planner is now faster, because vertex costs help steer the planner away from some dead ends. Also the environment changes between problems 5 and 6, but our planner re-adapts costs quickly and maintains its efficiency.

The proposed heuristic of adjusting vertex costs while planning proved a promising way to obtain good quality plans and fast planning times. Having a good precomputed roadmap however is key; the roadmap needs to be small enough to be quick to query, yet rich enough to capture enough variety of behavior for the robot. It may be useful, as future work, to investigate other procedures for roadmap generation, not just sparse planners; for example, some other methods that explicitly take into account the geometry of the configuration space.

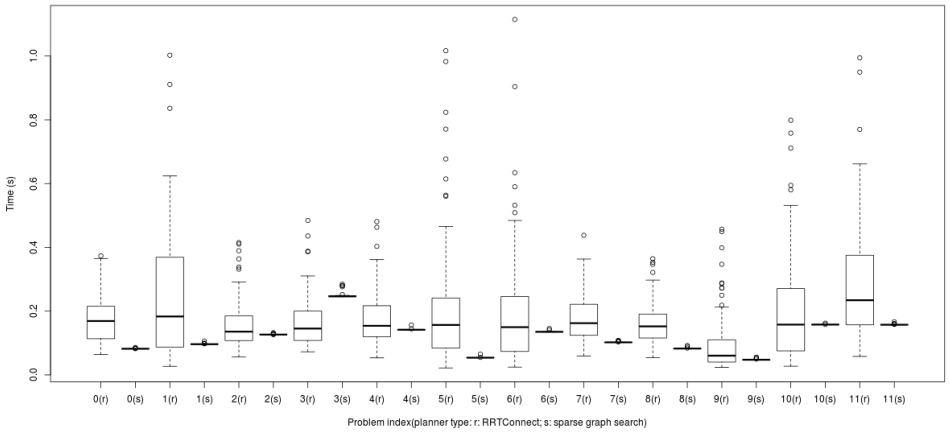

Figure 4-8. Boxplots for planning times for RRTConnect and our method when vertex costs in one problem run are initialized to the vertex costs from the previous problem.

4.5 Conclusions

In this chapter we study ways to use multi-query planners in changing environments. We first show that a classical solution (lazy PRM) is slower than a single query planner (RRTConenct), and we observe that this is a result of lazy PRM seeking an optimal path inside its known roadmap. We then propose a cost bump heuristic to steer graph search algorithms away from regions that might be obstacles. The cost bump heuristic can also be used as a way for the planner to learn an approximate shape of the free space of the robot, and we show how this may be done. It should be noted that the dimensionality of a robot's configuration space is typically much larger than 3, the dimension of the workspace for which it has a 3D model reconstruction, and that simple objects in the 3D workspace may result in very complex shapes in the configuration space.

We use a PR2 from Willow Garage to validate our cost bump approach in simulation and experiment. A PRM using cost bump can outperform single query planners, both in terms of planning time (reduced by a factor of two or better), as well as shorter, more efficient paths. It is important for the performance gain that the roadmap be small in number of samples, and therefore generated by some kind of sparse planner like a visibility based one. The roadmap should be small to allow fast queries, but large enough to capture a variety of possible behaviors.

5. Planning for manipulation tasks

Manipulation is an important component of a robot's tasks. It also can quickly become a very challenging problem for the robot if it is to plan its motion autonomously. Manipulation tasks often require precision, and the manipulated objects may be such that only a few of the maneuvers a robot can do are actually useful, and it will be difficult to find a sequence of movements just by a usual search for a plan. Further, manipulation tasks often make use of all of a robot's arms as well as end effector degrees of freedom for grasping and changing of grasps. As a result, manipulation planning is difficult because of the high-dimensional configuration space. In this chapter we investigate a multi-level approach to manipulation planning and provide an intuitive justification for it, as well as a simulation verification of the concept. This chapter contains parts from the author's papers "A two-level approach for intricate manipulation planning" [Pom13c], "Improving reliability and efficiency of intricate manipulation planning through mapping of grasp feasibility zones" [Pom14b], and "Mapping kinematic interactions between objects for robot motion planning" [Pom13d].

5.1 Justification for a two level approach for planning

Sample-based planners do not explicitly analyze the geometry of a configuration space; they do not identify boundary regions or narrow passages. Since the task space of the robot is often occupied with only a few obstacles, the result is a configuration space with plenty of free space to place samples in, and easy connections between them. A geometrical analysis on a configuration space will always be hard, and must complete before a plan can be searched for. Sample based planners meanwhile may quickly stumble into a solution by chance, especially if there is lots of free space, and this is one reason they are the standard today.

However, a robot must also be able to work in cluttered environments, on tasks that involve precise manipulations of objects that might be tangled with one another. The result is the presence of narrow passages in the free space of the robot, regions of small volume and poor visibility from other parts of the free space. They are difficult to handle by sample based planners; the small volume of a passage means the chance of samples being placed there is low. Plans that traverse narrow passages are therefore hard to find.

Problems of manipulation may present such narrow passages. Objects often need to be manipulated in close proximity to another, and may need to pass through or around one another. Consider for example placing a key in, or removing it from, a keychain, or hanging a coat in a wardrobe. Day to day life presents several such problems of (dis)entangling objects. The circumstances in which these appear are also varied, so simple static approaches (e.g., as done for industrial robots programming[Cho05, Pir07]) may not always be adequate. The quintessential disentangling problem is the puzzle, and by construction these are made to be hard for humans too. While rather artificial, they easily showcase, in an exaggerated form, the kinds of manipulation problems described here.

Apart from the narrow passages, manipulation problems can also be difficult because of the number of degrees of freedom involved. Often, the manipulation will have to be performed with two arms, either because there are two or more objects that must be manipulated, or because manipulating one requires changing grasps. This greatly increases the configuration space to explore for a solution, especially if the grasps themselves add even more degrees of freedom (the placement of gripper elements around the object) to consider.

Figure 5-1. Entangling or disentangling motion planning problems. Some are common in daily life, others are difficult enough to be challenging for humans.

Since having high-dimensional configuration spaces increases the complexity of the search for plans, one should try to work in as few dimensions as possible. Human intuition appears to take this approach when planning manipulation tasks. We seem to first think of how the manipulated object should move, or at least establish a few waypoints, and then think of how the arms should move and grasp, so that the object follows the planned trajectory.

In this chapter, we propose a two-level approach for complex manipulation planning: plan for the object considered as if it were capable to move on its own, then for the arms to manipulate it into following the solution trajectory. The benefit of our approach is planning now happens in the configuration space of the object (six dimensional), rather than the configuration space of robot with the object (for the PR2: 16 DoF for the arms and grippers). The plan obtained at this stage will be called solution path, or solution trajectory, for the manipulated object.

The plan we obtain for the object will constrain the space we need to search for arm and grasp planning. Rather than searching in the full space of possible motions of the manipulator, we search the subspace capable to manipulate the object towards following the previously planned trajectory. This greatly reduces dimensionality and improves efficiency. We will say the robot follows the solution path for the object when the robot grips the object, and moves its arms such that the object follows the solution path we obtained at the first level of our approach.

Our approach allows representing, in a compact fashion, a set of useful motions in the object configuration space as a roadmap; for example, trajectories passing through narrow passages. We can then concatenate trajectories from the roadmap into more complex plans as needed by the task. The construction of the roadmap for the object is done with human assisted motion planning, to ease

identification of the narrow passages and interesting regions in configuration space. The capability to use human in the loop planning is also important in some contexts like industrial robotics, as it allows the operators to have a greater degree of control over the robot's repertoire of manipulation maneuvers. However, our approach is not the same as teaching a robot by demonstration, as seen in for example [Arg09, Nie13]. The roadmap stored by the planner allows solving queries not formulated during training.

To validate our approach, we consider here a particular problem class: manipulating a ring piece around a card with two holes. A particular planning problem in this class requires the ring to be taken from some start to some goal state. A small arc of the ring is missing, which allows it to slide along the card and through the holes. The card is assumed to be a fixed obstacle in space and the the ring is assumed to be stable at the start and goal configurations. In our experiment, the robot doing the manipulation is a PR2. The PR2 has two 7-DOF arms, and a typical problem will require the use of both arms. Planning is done using a modified OMPL plugin for the MoveIt! package, and the simulation is viewed in RViz (the default visualization tool for ROS).

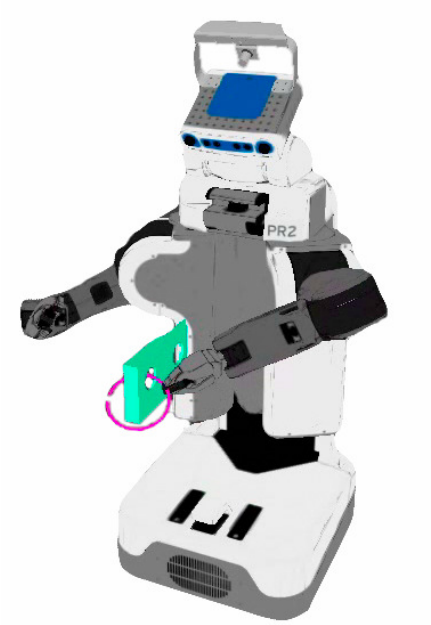

Figure 5-2. PR2 and an example disentangling problem.

For this particular problem class, one needs to make use of both the robot's arms and grippers, which results in a fairly high dimensional configuration space. Furthermore, by design, there are several narrow passages to make the problem difficult. However, it is natural to consider subspaces of fewer dimensions when solving this problem, and we pursue such an approach here.

One level of planning concerns the object to be manipulated, treated as a rigid body capable to move under its own power, in the environment with obstacles (but without the robot being present, or not considered as an obstacle in any case).

The purpose of this level of planning is to provide a solution path for the object to follow from start to goal.

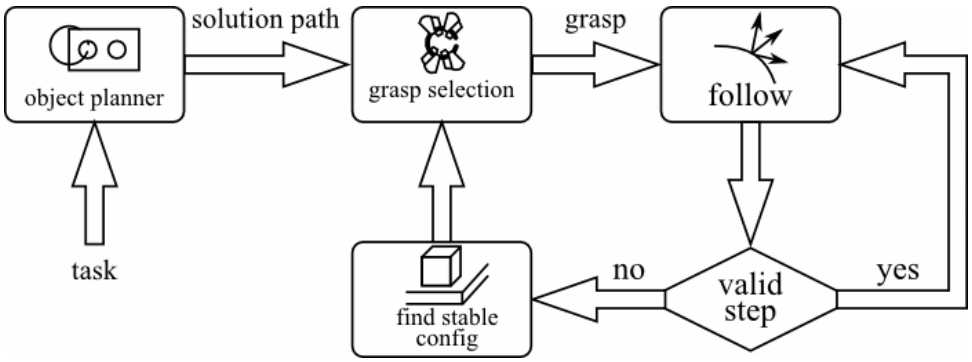

Figure 5-3. Two-level planner fora disentanglement problem.

Of course, in reality the object is not capable to move by itself. The second level of planning is then tasked with getting the robot arms to move in such a way, so they will drag the object along the solution path provided by the first level. The movements of the arms must themselves be feasible, and the feasibility constraint may require grasp changes on the object, for example when continuing to move the object using the current grasp results in a collision, or is kinematically impossible.

We present a planning architecture that fits our approach, and implement it atop the MoveIt! software package. We describe the method, and a test class of problems, in sections 5.2 to 5.4. We give some experimental results and analysis in section 5.5.

5.2 Motion planning for the manipulated object

The first step of our approach is to obtain a solution to the simplified motion planning, which considers the ring as a free-flying object. Note that the small missing arc, the card and the holes create several narrow passages. As a result, the single query sampling-based planners implemented in MoveIt! will have a difficult time solving it. Even with a couple of minutes allowed, we found such planners fail to find a solution for problems manipulating the ring through the card. These planning problems are for a relatively small-dimensional space (the possible configurations of the ring, which has six degrees of freedom), not the space of the two manipulator arms (which has fourteen, with two more DoFs for the grippers). Trying to use a single-query planner on the larger space would require even longer planning times.

On the other hand, a human can easily identify a few interesting configurations for the ring, which will guide the search for a solution. For this class of problems, humans can quickly see the narrow passages and place samples inside them. Thus, a roadmap is constructed with human assistance, to speed up planning. The roadmap construction proceeded as illustrated in the algorithm in Fig. 5-5: the user would set a start and goal configuration, then request a plan to link them. The customized OMPL planner then attempts to link the start and goal to the constructed roadmap. If they can be linked to the same connected component, then they are not interesting samples and discarded, to keep the roadmap small. However, if they

cannot be linked to the same component (because the roadmap is fragmented, or empty, or simply cannot see the start or goal), then they are added to the roadmap, and the planner spends some time generating random samples in an attempt to connect start and goal.

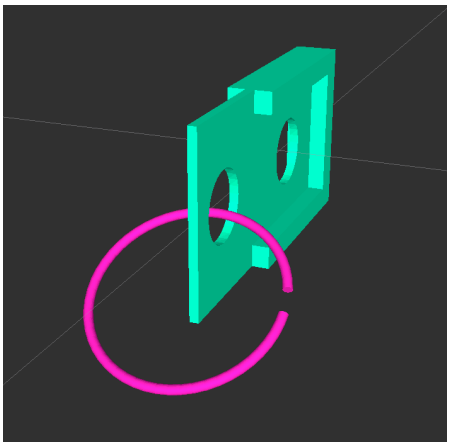

Figure 5-4. Example disentangling problem: the card and ring piece.

Algorithm 1 GrowRoadmap (G, q_{start}, q_{goal})

$startNear \leftarrow \text{NearestNeighbors}(G, q_{start}, r)$
$goalNear \leftarrow \text{NearestNeighbors}(G, q_{goal}, r)$
$nearVerts \leftarrow startNear \cup goalNear$
$connComps \leftarrow GetConnectedComponents(nearverts)$
if $(\emptyset = connComps)$ **or** $(1 < |connComps|)$ **then**
  $G \leftarrow G \cup \{q_{start}, q_{goal}\}$
  for $q \in startNear$ **do**
    $\text{addEdge}(G, q_{start}, q, |q_{start} - q|)$
  for $q \in goalNear$ **do**
    $\text{addEdge}(G, q, q_{goal}, |q_{goal} - q|)$
  $\text{RunPlanner}(G, q_{start}, q_{goal})$

Figure 5-5. Growing a roadmap for the rigid object planner.

At this stage, with the roadmap empty or small, and the user would place start and goal at interesting locations. In effect, the user grows the roadmap, not the planner. At each step, a list of maximally connected components of the roadmap is maintained. The goal is to have a roadmap with several interesting configurations of the ring around the card, which also contains only one maximal connected component. We used 167 vertices in the roadmap. Thus prepared, the planner proved capable to solve planning queries for the ring quickly and reliably.

With the roadmap prepared, we can submit planning queries for the ring object. The result is a solution path, in other words a series of poses for the ring to

move through. We feed this path to the next level, where we define a sequence of planning problems for the arms.

One thing to notice is, the strategy presented here constructs a solution path, then imposes on the robot the task to make the ring follow the path. The path may become infeasible if changes in the environment occur. However, that is not a disadvantage of our strategy; if the solution becomes infeasible, then replanning is possible, with the currently reached configuration as a starting point. Indeed, because of the presence of narrow passages, simple path deformation may not be adequate, leaving replanning as the only choice. Further, having a roadmap can greatly speed replanning for a problem with narrow passages, assuming the roadmap is rich enough to capture a wide enough variety of behaviors, while also being small enough to be fast to query.

5.3 Planning for the robotic arms

The second level of our approach obtains a sequence of arm motions to take the manipulated object along the solution trajectory found by the first level. In our case, the arms must manipulate the ring around and through the card obstacle.

One thing to note is that hooking/unhooking the ring to/from the card will often require changing the arm with which the robot does the manipulation; a single arm has limited reach.

There are therefore two subtasks the second level of planning needs to implement. One is grasp selection and switching, and it provides the start and goal configurations for the second subtask, actual planning of arm motion. It should be noted that grasp switching is itself a task on which some intuitive constraints are imposed. Simply put, the robot should not drop the object. Or, if the robot does release the object at some point, it should leave it in a stable or otherwise predictable configuration.

In our problem class, we allow the robot to "drop" the object only at the start and goal states. At all other times, at least one arm must grip the ring. A more general approach would also include a physics engine, or at least some procedure to reason about and generate configurations for a manipulated object that are stable.

5.3.1 Grasp selection

For our study here we defined twentyfour grasps around the ring for each arm. A grasp is indexed by a so called incident angle (six choices) of the gripper towards the ring. For each incidence angle, we have "aligned" (gripper forward direction is parallel to normal of ring plane) and "side" (gripper forward direction perpendicular to ring normal) configurations. Each aligned and side configuration also has a flipped version of itself, depending on which of the gripper fingers is inside/on top of the ring while gripping.

Grasp selection is guided by a few criteria. The most obvious is feasibility: if an arm cannot reach the grasping position (either because of kinematic constraints of the arm itself or because of obstacles), then that grasp cannot be an option.

Some kind of optimality measure may also help refine the search. In particular, one can try to select grasps in such a way so as to minimize the number of necessary grasp switches during the solution. What we do here is more a greedy approach than a true minimum of grasp switches: we try "promising" grasps first

and backtrack if the robot encounters dead ends. A promising grasp is one which allows the arm to take the manipulated object through many waypoints on the solution trajectory before the grasp needs to be changed. While not guaranteed to be optimal, this heuristic gives good results in practice without having to search through all possible grasp sequences.

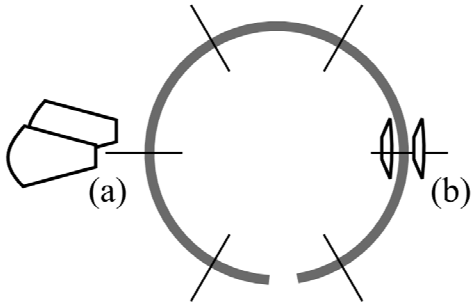

Figure 5-6. Grasp positions and orientations on the ring piece. Grasps can be side (a) or aligned (b).

5.3.2 Grasp switching

Grasp switching becomes necessary when the robot cannot continue, using the current grasp, to manipulate the object along the solution trajectory found by our first level of planning. This produces a sequence of planning problems for the two arms as they need to grip/ungrip the object. The sequence is straightforward, since the robot must always keep at least one gripper on the ring. Also, the arm currently not gripping is moved away, so as to not interfere with the solution path.

It may happen however that the other arm cannot reach the ring, and therefore a grasp switch is not possible. Since continuing with the current grasp is also not possible (or else, there would not have been a need for a grasp switch), such a situation is called a "dead end". To escape it, the robot should move back along the solution trajectory of the manipulated object, and try grasp switches at past points.

For our implementation here, we try to avoid dead ends at the planning stage by using a backtrack procedure, to be described in section 5.4.2. In effect, our robot "moves back" while thinking about a solution. As a result, our robot has a complete plan when it starts moving, and doesn't need to retrace its steps during actual execution.

In an actual environment, where changes might appear, the capability to go back along the solution path while executing the manipulation is however useful. We do not implement it here, but it is a straightforward extension to our approach.

5.3.3 Trajectory checking

While planning for the arms, we need to check the validity of manipulation motions while the robot arm grasps the manipulated object (the ring, in our case). A particular grasp, which in our case entails a choice of incidence angle, alignment,

and flip status, induces a transformation from the pose of the object to that of the end effector, and from that an arm configuration may be deduced by inverse kinematics (IK).

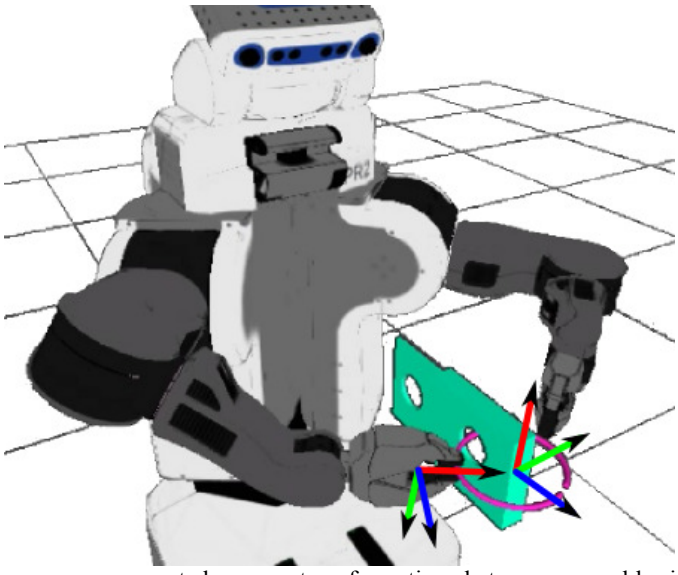

Figure 5-7. Grasps are represented as pose transformations between moveable rigid object and gripper.

It may happen however that IK fails to find an arm configuration that brings the gripper into the desired pose. Or, while the IK can find a solution, it's a "jump" from the previous one, which indicates that the arm cannot follow, using a continuous trajectory, the motion described by the ring solution path.

Finally, another source of possible failure to follow the solution path are the obstacles- including the robot and ring themselves. If the IK computation yields a solution without jumps, we also perform collision checking as well on each step the arm makes while following the solution trajectory.

5.4 Improving the two level approach

The core idea of our approach is to split a complicated manipulation problem into several simpler ones, where "complicated" and "simple" have a clear meaning in terms of degrees of freedom we need to consider at any time. Rather than plan for the whole robot at once, or at least arms with grippers (16DoF), we plan for the manipulated object once (6DoF) and for the arms (7DoF; gripper action is handled outside of planning) as needed because of grasp switches while following the solution trajectory.

We now present two further additions to this basic outline to improve efficiency and reliability.

5.4.1 Roadmaps for the arms

We saw in the previous chapter that roadmap planners can improve search speed, if helped with a heuristic like our cost bump. Further, the cost bump can help the planner learn the free space of the robot.

This is especially helpful for our approach since we need several planning queries solved as we try various grasp switches, all in quick succession. Our cost bump method is then given plenty of opportunity to learn about change in the environment. Further, the planning problems are about moving the arms between a given known configuration (the "arm away" configuration) and some configuration around the ring, which will not move too far away as we try several grasps. As a result, the movements of the arms will be similar for the various grasp/ungrasp actions. The cost bump method will be particularly apt to try such exercised trajectories first, because having a valid region to move the arms through will reduce costs in that region and reinforce its prioritization in future searches, in a similar manner to how a human being would do manipulation.

Cost bumps can also provide assistance with another problem, that of dead ends, by tracking which regions of space contain configurations that are awkward for the robot to reach and grasp. Planning for the manipulated object will then avoid such regions, which should improve the chance that a solution candidate produced by the first level is actually feasible for the robot.

5.4.2 Backtracking

In section 5.3.2 we described the problem of dead ends- situations when the robot cannot follow forward along the solution trajectory for the manipulated object. We observed that, at least for the manipulation problems we posed to the robot, our greedy grasp selection heuristic was often good enough to avoid dead ends.

Nonetheless, dead ends can happen, hence the need to backtrack. In our implementation here, at every moment where a grasp switch is necessary, we make a list of feasible grasps with the other arm and the length of the segment of the object solution trajectory that they can follow. When we need to backtrack, we return to the moment of the grasp switch and select the next best. If all the grasps in the list have been tried, we go back a further step and so on. This type of backtrack is done in the planning and validation stage, before execution; it is the type of backtracking we implement.

Backtracking may also be done while the robot performs the manipulation. The idea is the same: have the robot move backwards along the object's trajectory, possibly testing for grasp changes along the way as well. We do not implement this here, but it is a straightforward extension of our approach.

We limited the moments at which we can backtrack to, to the moments when grasp switches must happen. In principle, one could test grasps at every step (every waypoint) along the solution trajectory for the manipulated object. However, these waypoints are often very close to each other so that testing grasps everywhere is a waste of effort; feasible grasp lists will not change much between close waypoints. In practice, we found that switching grasps only when we have to (because continuing to follow using the current grasp is unfeasible) works reliably.

Increasing the moments when we consider grasp switches only increases complexity and reduces the efficiency of the method.

5.4.3 Grasp feasibility zones

Our heuristic (select a grasp based on the length of object solution path that it can be kept on) produces good results in terms of number of grasps needed, however it is expensive to evaluate. To speed grasp selection, we first rank them according to a grasp suggestion strength metric, which is easy to compute and will define in this section. The grasp suggestion strength provides a quick, provisional ranking, which we can use to see which grasps are promising to test first. Further, we will limit tests to the first k grasps in order of suggestion strength; only when all these grasps fail will we test the others.

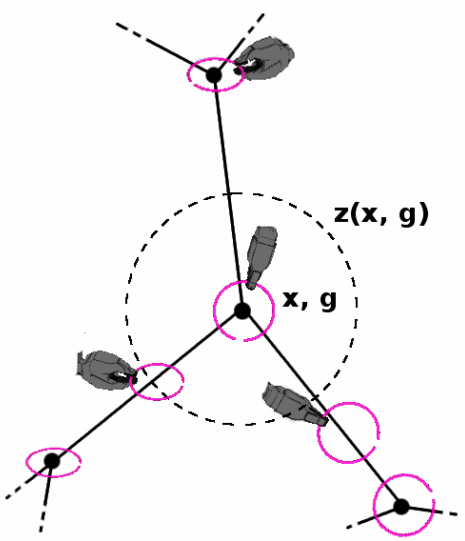

Figure 5-8. Grasp suggestion strength.

In order to define suggestion strength, we first define a **grasp feasibility zone** z(**x**, **g**) as a real number representing an estimate of the radius of a ball around vertex **x** (representing a configuration of the manipulated object) where grasp **g** can be maintained. This is estimated off-line after roadmap construction, for each vertex and each grasp. In order to compute z(**x**, **g**), we attempt to follow, using grasp **g**, every edge in the manipulated object roadmap which has **x** as an endpoint. The smallest distance before the grasp becomes infeasible is z(**x**, **g**).

In general, grasp switches will not happen at vertices in the roadmap, so let the configuration of the manipulated object at the point of grasp switch be **y**. We find the set $\mathbf{N_k}$ of closest k neighbors to **y** and let **x** be some vertex in the set. For each known grasp **g**, we can get the suggestion strength from **x** for grasp **g** at **y**:

$$s(\mathbf{x},\mathbf{g},\mathbf{y}) = \frac{z(\mathbf{x},\mathbf{g})}{1+dist(\mathbf{x},\mathbf{y})}$$

For each grasp **g**, we collect suggestions strengths from all vertices in $\mathbf{N_k}$. The maximum is the grasp suggestion strength for **g** at **y**:

$$s(\mathbf{g},\mathbf{y}) = \max_{\mathbf{x}\in\mathbf{N}k} s(\mathbf{x},\mathbf{g},\mathbf{y})$$

Intuitively, grasp feasibility zones give the robot some knowledge of what grasps to try in various regions of the manipulated object configuration space. Once computed offline, the grasp feasibility zones allow a quick estimation of how promising some particular grasp is for use when solving a given problem.

5.4.4 Recalculating object solution paths

Splitting the manipulation planning into two stages as we do here creates a potential problem: what if the solution path found for the object is infeasible for the robot? For example, the solution path might pass through some kind of tunnel which would prevent the robot to reach and manipulate the object. Such a solution path cannot be used, and a new one should be tried.

To help steer the planner from problem areas, we reuse the concept of the cost bump we introduced in chapter 4. If a configuration along a solution path is a dead end, meaning we must backtrack, it is an indication the solution path candidate passes through a region that is awkward for the robot to reach, and we apply a cost bump to vertices in the object roadmap. Only a small number of backtracks are allowed before the planner attempts to find a new solution path candidate and reset the grasp selection process. Conversely, if a solution path candidate results in a complete, feasible sequence of grasps, we apply cost unbumps to the vertices from the roadmap, to mark that regions the path passed through allow the robot to reach and manipulate.

In this way, the manipulated object roadmap also contains information about the robot's ability to reach the object from various configurations. The regions may change as the environment changes and obstacles move, but the planner will track them as needed, during planning queries, by updating vertex costs when invalid regions are detected as intersecting roadmap vertices.

5.4.5 Separation of grasp selection from arm motion planning

Arm planning for grasping/ungrasping is a computationally costly operation, which nonetheless is necessary whenever the robot switches grasps during manipulation. However, it is not necessary to plan for the arms at every grasp test. Since not all grasp tests will result in grasps used for the final solution to the manipulation problem, we further split the second level of our planning approach into two stages: "grasp selection" and "plan extraction".

During grasp selection the planner attempts to find a sequence of grasps that is feasible and will allow the robot to take the manipulated object all along the trajectory obtained from the first level, from the start to the goal configuration of the manipulated object.

Only after we have one such complete sequence of grasps do we pass to the plan extraction phase, which generates arm planning queries for each grasp switch. Should arm planning fail for any of the switches, we go back to grasp selection, and resume backtracking from the last grasp switch for which we had an arm plan, to

avoid maneuvers for which arm planning doesn't find answers. In general, this will result in a small number of motion planning queries since we only plan when we have a likely good and complete grasp sequence.

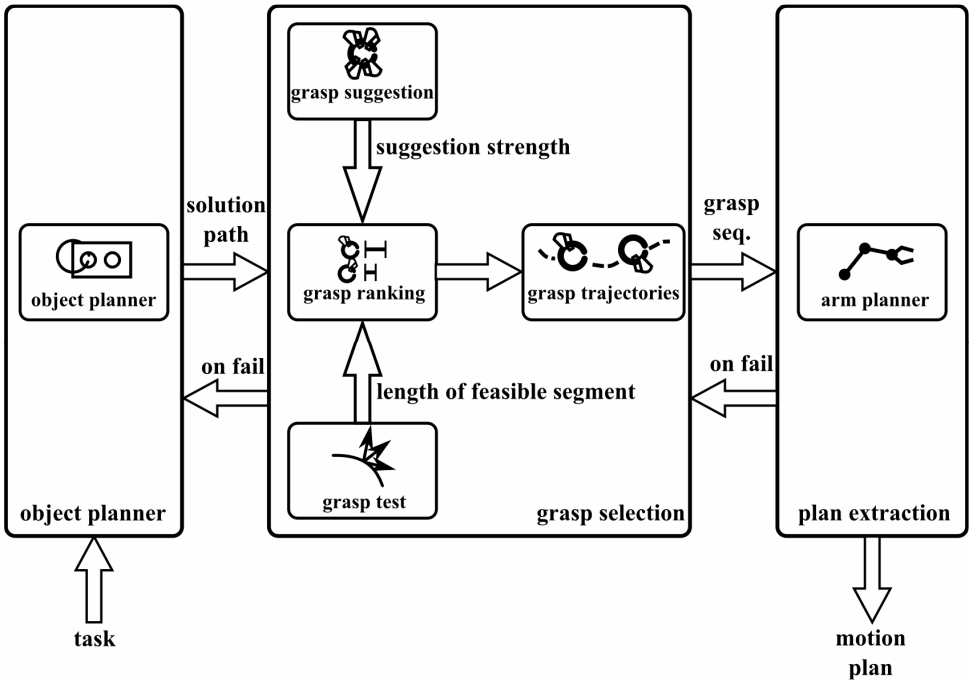

Figure 5-9. Three-level planner block diagram. Arm planning queries are delayed until absolutely needed, to improve planning time.

5.5 Simulation verification

We implement our planner as an addition to the MoveIt! package and test it on a few manipulation problems that require moving a ring piece around a card with two holes. The card is a fixed obstacle. The ring has a small arc missing, to allow it to hook or slide around the card. We allow the robot to leave the ring ungrasped only at the start and goal configurations; everywhere else, the robot must keep at least one gripper on the ring. The simulation is visualized in RViz, the default ROS visualization tool. We run the simulations on an Intel® Core™ i5-3210M CPU running at 2.5GHz.

The problems are: change the hole that the ring is hooked to (query "change"); take the ring out, flip it, then rehook it to the same hole (query "flip"); starting from the ring hooked on both holes, take it out of the card, flip it, rehook it to both holes in the card (query "dhook").

For a first test, we test our planner without grasp suggestion (which is the same as setting k to infinity). All grasps are tested for feasible length along the solution path and the one allowing the longest following of the candidate trajectory is chosen. Each problem is run five times. Average and standard deviations of

planning times, split in the three stages of planning, as well as number of grasp switches (including the first grasp) are given in Table 5-1.

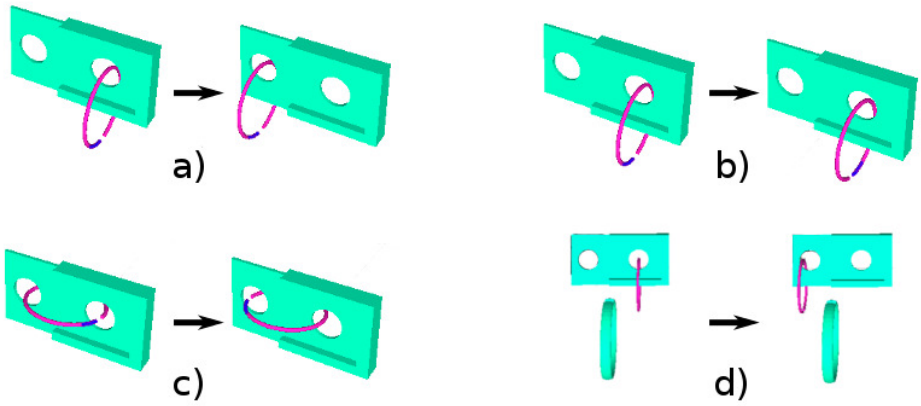

Figure 5-10. Test problem queries (start and goal configurations): "change" (a), "flip" (b), "dhook" (c), "change" with a new obstacle in the environment (d).

Table 5-1. Planning time statistics without grasp suggestion

Query	puzzle plan		grasp selection		plan extraction		grasp switches
	avg (s)	sdev (s)	avg (s)	sdev (s)	avg (s)	sdev (s)	
change	0.4	0.04	6.5	0.16	5.24	0.59	2
flip	0.52	0.02	15.63	0.26	16.22	0.97	7
dhook	0.51	0.06	25.91	1.46	26.48	1.17	13

As a second test, we now run our planner with k=8: we use grasp suggestion strength to find at most 8 grasps to test first, and only test more if these first 8 fail. Again, each of the three problems is run five times, and we show the collected statistics in Table 5-2.

Table 5-2. Planning time statistics with grasp suggestion (k=8)

Query	puzzle plan		grasp selection		plan extraction		grasp switches
	avg (s)	sdev (s)	avg (s)	sdev (s)	avg (s)	sdev (s)	
change	0.37	0.02	3.21	0.07	5.5	0.42	2
flip	0.49	0.01	6.71	0.12	15.89	1.0	7
dhook	0.48	0.01	11.62	0.35	24.45	1.35	13

For a third test, we run the problem "change" several times, however we insert an obstacle in some of the runs. Planning times are shown in Table 5-3; where several solution path candidates were generated, we list planning times for each. The first two runs are in an environment where the only obstacle is the card. For run 3, we insert an obstacle and as a result, the plan times for this situation are large because several solution path candidates are attempted. Eventually a seven grasp solution is found. However, once the planner learns what regions to avoid, it quickly finds good solution paths; in three subsequent runs (4, 5 and 6) the planner

is much quicker to find a solution, using even fewer grasps (five). For the next runs, we remove the obstacle. Fewer grasp switches are needed (four), but the planner will revert to the original solution only after it learns that the regions it passed through are available again.

Table 5-3. Planning times on runs in a changed environment

Problem run	puzzle plan (s)	grasp selection (s)	plan extraction (s)	grasp switches
1	0.33	2.62	5.04	2
2	0.33	2.51	5.03	2
3	0.37; 0.35; 0.37; 0.35; 0.35	79.51	13.46	7
4	0.39	4.29	9.31	5
5	0.39	4.31	8.94	5
6	0.43	4.32	8.4	5
7	0.4	3.83	7.86	4
8	0.42	4.4	7.75	4
9	0.4	4.13	7.49	4

Finally, using a precomputed roadmap is not equivalent to storing one precomputed behavior for a given task. The roadmap is capable to answer more planning queries, and it needs not be constrained to a particular environment. In our case, only the ring and the card are relevant when the roadmap is constructed. The environment can be different and include more obstacles when the roadmap is actually used. As long as the roadmap is rich enough to contain various movements of the ring relative to the card in an otherwise empty space, it will work well in an environment with other obstacles and/or the card in a different pose. To handle such situations, one can use roadmap planners adapted for changing environments, for example lazy versions of PRM.

5.6 Reusing roadmaps for manipulation

The previous sections have presented a manipulation planner which uses a roadmap for the object manipulation task. The approach splits the planning problem into tasks in lower dimensional configuration spaces, and allows handling complex situations that contain narrow passages, but leaves open the question of how a roadmap is specified in such a way so as to be useful for a broader category of objects. Not all keychains are the same, not all door latches are strictly identical geometrically, not all containers are the same, etc. Nonetheless, in some sense all door latches are similar to one another, a keychain has the same function and operates in almost the same way as another keychain, and so on. While operating such objects requires maneuvers that pass through narrow passages and tightly fitting parts, once a human being learns to operate one object in the class, they will quickly generalize that knowledge to other objects in the class. It is worth therefore to find ways in which roadmaps used for planning are similarly reusable.

Typically, roadmaps are specified as graphs with vertices being points in the configuration space of a given object. As such, a roadmap specification depends on the exact geometry of an object- its size and shape. Consider the test problem of

the ring and card from the previous section. The relative pose of the ring to the card has a translational component that depends on the size of the two bodies. Scaling them twice means the translational components in the vertex specifications must also be scaled.

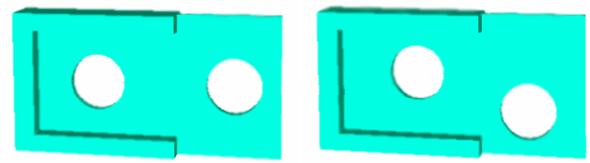

Figure 5-11. A roadmap constructed for the ring piece around the card on the left would not work for the card on the right, even if for a human being the two cards are similar.

However, that is not the only possible transformation we could apply to the ring and card objects, and still consider it the same problem. Imagine, for example, that the positions of the holes in the card changes (see Fig. 5-11). This would invalidate the roadmap constructed with the original card and ring pair; nonetheless, the modified card still seems sufficiently similar to the original, so that a slightly, and hopefully cheaply, modified roadmap based on the roadmap for the original could be useful for planning.

One obvious way to address this problem is to specify the vertices in the roadmap as poses relative to "important" features on the two objects. For example, one could, rather than give the exact coordinates of the ring relative to the card piece, specify its position as being inside a plane defined by the centers of the two card holes. In general, translation components of a pose would be specified in terms of dimensions of the pieces like diameters and thicknesses, while orientation components would be specified so that vectors between object features align in certain ways, to obtain a scale- and coordinate-free specification of the roadmap vertices.

This approach would then make use of some object detection and recognition methods to see where the important features of the objects the robot works with are, and automatically adjust the roadmap based on the information about those features.

One problem remains unadressed however- just what is an "important" feature? A human operator could define the card piece as having two holes and a thicker segment on the ridge, then proceed to define the roadmap vertices according to these features. However, if we want some autonomy in the robot, it should be able to identify at least some features that are important enough to define an object class.

Previous work in reasoning about object parts has specifically focused on identifying affordances for grasping: just where an object can be grasped by the robot's hand [Kra01, Mil03, Dia09, Cio10, Xue09, Krm10, Nie12]. More recently, there has been some work about identifying the geometric primitives that approximately make up an object and, based on what primitives an object contains and how they are arranged, recognize it [Ten13]. Other work[Hoo12] attempts to find plans that will maximize a robot's knowledge of how objects are situated in an environment. There exists a gap between such research and motion planning however, in that even when knowing the shape primitives that make up an object, it is not trivial to describe how an object would interact with another. One can

describe a ring piece as a torus, and a hook as a bent cylinder, but the robot still has to infer- or more often, be told- that the hook can stop the ring from falling down.

It is however these object interactions that are important for planning. Some of these interactions are of a physical nature: will the two objects stick to each other, what is the friction between them, how would they behave in a gravity field? Some interactions, and these will be the focus of the next subsections, are kinematic: how do the objects restrict each other's movement in various configurations?

We will study the problem of describing the kinematic interaction of two complicated rigid objects in such a way that would be both compact and reusable to planning queries involving a larger class of objects of similar, but not identical, geometry. The classic kinematic pairs are included, but do not exhaust, the categories of objects we consider here. As an example, there is no lower kinematic pair with two rotational degrees of freedom; however, the gear stick of a car is such a system, as rotations around its length axis are restricted; further, rotations along axis perpendicular to its length are restricted depending on where the kinematic stick is in its configuration space.

We will introduce a datastructure, a degree of freedom map (or DoF map), which stores information about how the degrees of freedom of one object change as it moves around the other. Important configurations are those where the degrees of freedom in the motion change.

Unlike most previous approaches, ours is compatible with a robot exploring the environment through touch. The sense of touch for robots has been used to allow safe interaction with humans[Had08] and exploring a one-dimensional object[Had11]; we extend it to explore more complicated geometries here. While the sense of touch cannot replace vision, it offers a good complement because in many cases a robot's vision systems cannot see all the relevant parts in a mechanism: occlusions prevent a robot from getting a complete picture of an object. Even if some features are detected, there will be errors in reconstructing a 3D model from them. If, as is often the case, several objects fit very snuggly together, even very tiny errors would result in the robot believing there are no solutions because all possible corridors have been blocked by noise. Humans however can easily solve these problems: we don't need to know what's inside a lock in order to use it; we can do that almost entirely by touch.

The next sections will describe procedures to construct and reuse DoF maps for planning queries. We will end with a discussion of what would be an object classification criterion that would be useful for motion planning, and argue that DoF maps are such a criterion.

5.6.1 Degree of freedom (DoF) maps

A **degree of freedom map** (or **DoF map**) is defined as a graph where the vertices are regions in the configuration space of a pair of rigid objects, one of which is considered a fixed obstacle. Vertices can be **corridors** (long regions where only one degree of freedom is important at any point, and the free directions at nearby points must be close to parallel to each other; there must be no sudden change in allowed direction of movement). Vertices can also be **junctures**, small regions where several degrees of freedom are available, and for each degree of freedom there is at most one corridor passing through (or stopping at) the juncture. A corridor may pass through several junctures. Also, a corridor can be circular; an

example of such a corridor is spinning in place. An, L-shaped bend is not considered a corridor (the allowed direction changes too fast), but a juncture where two corridors meet and stop.

During the construction phase, the DoF map includes geometric information such as relative object poses expressed in some coordinate system and the exact components of the vectors describing the degrees of freedom. Each juncture has exactly one representative pose, whereas a corridor will often have several representatives, closely spaced along its length. Links between a juncture and a corridor are treated as short paths between representative poses.

Also, for every degree of freedom at a vertex in a DoF map, we store step lengths along both positive and negative directions to keep information about how far one can go before either colliding or reaching a maximum step threshold. This data is used to represent whether junctures provide abrupt stops for corridors (see Fig. 5-12, left), and characterize 'how free' a degree of freedom is.

During reuse, the geometric information about representative poses is discarded. There is no need to keep exact coordinate representatives if the objects the DoF map is used for are different from those used for its construction. Step length information for junctures is kept however.

5.6.2 Constructing a DoF map

To avoid singularities, during construction the relative poses of the moveable object to the fixed one are stored as elements of the special Euclidean group **SE**(3). Such an element is a 4-by-4 homogenous matrix, which contains a linear translation part and a rotation submatrix.

In order to discover the DoFs at some pose, we will perform small relative motions relative to that pose. These small movements are represented as elements in the Lie algebra se(3) of the special Euclidean group. Let **v** be a direction in se(3) and t a small time interval to follow this direction, then new pose obtained after such a motion, assuming it is successful and does not encounter a collision, is given by

$$\mathbf{g}_{next} = \mathbf{g} \cdot \exp(t \cdot \mathbf{v})$$

where **g** is the current pose and the vector **v** is of unit length and of the form (ω_x, ω_y, ω_z, d_x, d_y, d_z), in which the first three components are an angular velocity axis, while the last three are a translation one. The product t**v** gives a vector representation of the displacement.

DoFs at a pose will be identified by collecting several (t_k, $\mathbf{v}_k$) pairs (or equivalently, displacement vectors $t_k\mathbf{v}_k$). In previous literature, principal component analysis was used to identify directions of roadmap expansion from such collections of vectors[Dal09].

However, principal component analysis assumes all components in the collected vectors are similar quantities with similar scales and units of measurement. In our case however, some components are linear displacements, while others are angles.

To address this problem, to identify the DoFs at a pose we first perform only purely translational movements around that pose. From the resulting collection of vectors we obtain the maximum distance travelled from the starting pose; let it be

called r_s, and it will serve as a rotation scaling parameter for the pose. Other data vectors representing displacements around the pose which have rotation components will be modified to have the form $t_k \cdot (r_s \cdot \omega_x, r_s \cdot \omega_y, r_s \cdot \omega_z, d_x, d_y, d_z)$. The principal component analysis is then run on these modified vectors, in which all components are linear displacements.

The result from the analysis is a collection of eigenvalues and eigenvectors which characterize the directions in which there is the most variance in the collected displacement vector data. The eigenvectors are by construction orthogonal, and the larger the eigenvalue associated to an eigenvector, the "freer" the movement is along that vector. Therefore a cut-off threshold on eigenvalues can be used to decide whether a direction is free, or is sufficiently constrained that it can be called blocked.

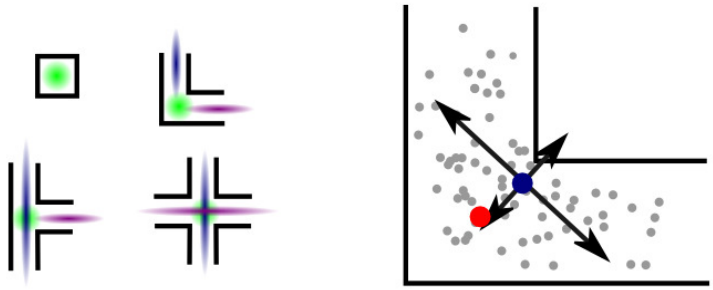

Figure 5-12. Left: types of junctures for two degrees of freedom. Right: PCA skew at a juncture where corridors end.

If after DoF identification only one direction is found to matter, the current region is a corridor. Corridors are explored by alternating steps along the free direction with DoF identification procedures. If following the previously estimated corridor direction results in a collision, we take a step back and do another DoF identification procedure. If there is no close to parallel direction of continuation, one end of the corridor has been reached. The angle between two vectors can be estimated by their dot product; for orthogonal vectors, it is zero, for parallel vectors that go in the same direction it is positive and equal to the product of their lengths, while if the vectors are parallel but opposite, it is the negative of their length product.

A corridor is explored until both of its ends are encountered, or the corridor is found to be circular, which happens when a neighborhood of the starting point is found without changing the direction of exploration.

If while exploring a corridor a pose is found with several degrees of freedom, the new pose becomes a candidate to represent a juncture. It is first compared with previously encountered junctures, and if it is not close to any of them, a new juncture region is added to the roadmap and for each of its free directions, except that of the current corridor, pairs of the form (juncture index, direction vector) are added to a "to explore" list. If however the encountered pose is close to a previously recorded juncture, we remove the corresponding (juncture, direction) pair from the "to explore" list.

If we find more representatives for a juncture, we keep the most central one. The centrality of a pose inside a juncture is estimated by the balance of the

eigenvalues corresponding to degrees of freedom at that pose. For example, if a juncture has three degrees of freedom, we store the representative which has three large eigenvalues that are closest to equal.

The eigenvectors which describe the directions at a juncture however are taken from the eigenvectors of the incident corridors. This is because if corridors end at a juncture, the principal component analysis produces skewed results inside the juncture (see Fig. 5-12, right). Principal component analysis first substracts the average from the data vectors. This should be the equal to the pose that we collected the data vectors around, however if a corridor ends at the juncture, displacements towards the end will be shorter, and as a result the average moves away from the pose, which also changes the detected free directions.

The map DoF construction process is assumed to operate with tactile feedback only (the robot knows where the moveable piece is because it can know its end effector position, and can detect when collisions are encountered by force disturbances at the end effector). First, identify whether the starting location is inside a corridor or juncture. If a corridor, explore it and add junctures and (juncture, direction) pairs as encountered; if the start is a juncture, add (juncture, direction) pairs to the 'to explore' list. Keep exploring corridors from the 'to explore' list until it is empty.

5.6.3 Reusing DoF maps

Exploring the configuration space of a rigid object is a costly operation, regardless of the method used, so it pays to reuse information by generalizing to other object pairs than those previously explored for DoF map construction. In particular, it may be the case that two pairs of rigid bodies may look different, but have DoF maps that are isomorphic graphs. As will be discussed later, daily life offers several examples of classes of objects defined by DoF map isomorphism (see Fig. 5-13 for an example, or the simulation test case for another). We now describe a DoF map reusal procedure for such cases.

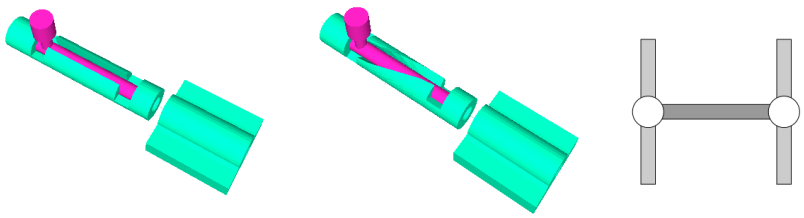

Figure 5-13. Two objects (left and middle) with isomorphic DoF map (right).

For reuse, we modify the DoF map representation slightly. Previously, in the construction phase, it was a graph of corridors and junctures. For reuse, we split each corridor into segments, and a segment is such that junctures can appear only at its ends. Junctures and corridor segments will thereafter be referred to as **regions**, and let R be the set of all regions in the DoF map. Geometric information about representative poses is discarded, as it is no longer relevant. Step length information for the regions is still useful however to characterize them as junctures where corridors pass through or stop.

Suppose then that a DoF map is known, which was constructed for some previously encountered pair of rigid objects, and we wish to reuse this DoF map to solve planning queries involving a new pair of rigid objects that are such that their DoF map, if it were constructed, would be isomorphic to the known one. We assume that the DoF map region which contains the goal is known and that, while the exact goal configuration may be unknown, either the exact goal is unimportant (all that matters is to bring the object in a certain region of its configuration space) or, if the goal is near, the robot will know this. In these conditions, we can run a simple shortest path search from the goal region to every other region in the DoF map, so that for every possible starting region, we have a direction to go to next so as to approach the goal region.

First we must estimate the starting region. Prior estimations may be available from a vision-processing method, or the priors may be completely uninformative (all regions are equally likely to be the starting region). In either case, there is an initial, quantified belief about the start, which is adjusted based on a degree of freedom identification step at the starting pose. For every discovered degree of freedom, a minimum and maximum step value is observed, which represent the distance one can go by using the degree of freedom in the negative and positive direction until either a collision or a maximum step threshold is encountered. Let **b** be the data of such step limits along the degrees of freedom at the start. Then, the posterior probability that a region j of the DoF map is in fact the start is given by a Bayes formula:

$$p_{j,0|0} = \frac{s(\mathbf{b} \mid j)\, p_{j,0|-1}}{\sum_{m \in R} s(\mathbf{b} \mid m)\, p_{m,0|-1}}$$

where $p_{j,0|-1}$ is the prior probability that region j is the starting region, $p_{j,0|0}$ is the posterior probability, and s(**b**|j) is a function quantifying the similarity between the observed data and the data we would expect if j were indeed the starting region.

We also need to estimate the "previous" region, and use a (current, previous) region pair to describe the direction through the DoF map; even if it is not strictly speaking physical, we still need an estimation of this previos region at the start. For a region j, let L(j) be the set of regions that are linked to it. Then, the probability that j was the previous region is given by

$$p''_{j,0} = \sum_{m \in L(j)} \frac{p_{m,0|0}}{|\, L(j)\,|}$$

where |L(j)| is the number of elements in L(j).

The region with the maximum posterior probability is considered to be the starting region, and from its neighbors, we select the one with the greatest probability of being the previous region. This information is then used to determine a direction to move in so as to reach the goal.

As the object is moved, steps along the current direction are alternated with DoF identification procedures. When the number of DoFs changes, the current region has changed, and the estimations need to be updated. Estimating the current region is done with a similar formula as before:

$$p_{j,k|k} = \frac{s(\mathbf{b} \mid j) p_{j,k|k-1}}{\sum_{m \in R} s(\mathbf{b} \mid m) p_{m,k|k-1}}$$

where $p_{j,k|k-1}$ is the prior probability that region j is the starting region after k region changes, and $p_{j,k|k}$ is the posterior probability. Unlike the previous equation however, the prior probabilities need estimating themselves, based on the structure of the DoF map and our beliefs of what the current region and direction of movement are. Consider then the region j, with L(j) its set of linked regions; let m be a region in L(j). Then, let L(m,j) be the set of regions that are linked to m, and are such that a (current, previous) pair of the form (m, n $\in$ L(m, j)) implies a direction towards j. Then the prior probability of j is given by:

$$p_{j,k|k-1} = \sum_{m \in L(j)} \left(p_{m,k-1|k-1} \cdot \frac{\sum_{n \in L(m,j)} p''_{n,k-1}}{\sum_{n \in L(m)} p''_{n,k-1}} \right)$$

To update the estimations for the previous region, first we define the auxiliary events $p^j_{m,k}$ as "after k region changes, current region is j and previous is m". The probability of such an event is given by

$$p^j_{m,k} = \frac{p_{j,k|k} \cdot p''_{m,k}}{\sum_{n \in L(j)} p''_{n,k}}$$

Then, to get the new posterior probability that j was the previous region we use

$$p''_{j,k} = \sum_{m \in L(j)} \left(\frac{p_{m,k|k} \cdot \sum_{n \in L(m,j)} p^j_{n,k-1}}{\sum_{n \in L(m)} \sum_{i \in L(n,m)} p^n_{i,k-1}} \right)$$

Throughout the previous formulas, we have used sets of the form L(j) and L(m, j). Of these, L(j), the set of neighbors of j, is trivial to obtain from the DoF map structure. For the L(m, j) sets there is a slight complication because, at a juncture, we may choose to stay on the same direction (and remain in the same corridor as before), or switch to an orthogonal direction. If the direction stays the same, then L(m,j) is the set that contains the other neighbor of m that is on the same corridor as j; otherwise, the set contains the neighbors not on the same corridor as j.

5.6.4 Simulation verification

To test the DoF map strucutre, we implemented code for DoF map creation and reusal that works with the MoveIt! robotics software package and used RViz (the default visualization tool for ROS) as a simulation front end. The computer hardware was a laptop with 3.8GB RAM and an Intel® Core™ i5-3210M CPU quadcore processor running at 2.5GHz. Tactile feedback was approximated by moving the object in very small increments, and performing collision checking after each one. A step is undone if it generates a collision. A real robot equipped with force-torque sensors may gain more information than just collision from the force disturbance; it may obtain some knowledge about the shape at contact and speed up constrained direction estimation. The primary bottleneck in our simulations is the collision detection steps. On a real robot, the limit would be given by how fast the robot could move, and still obtain good data from the force disturbance observer.

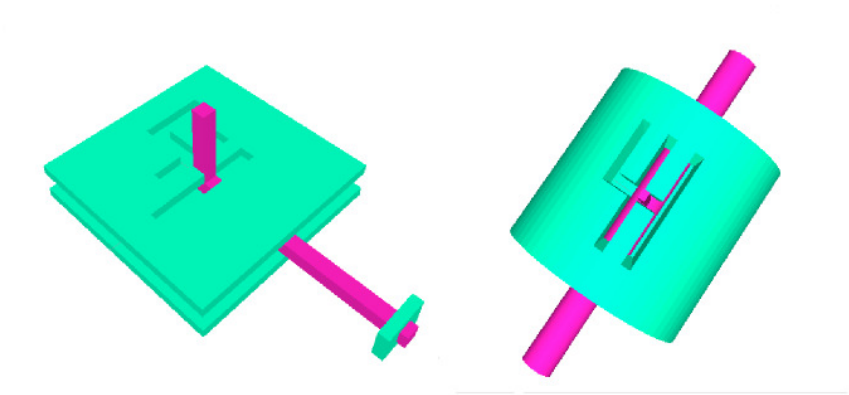

Figure 5-14. Two objects with isomorphic DoF maps: a planar maze (left) and a gear stick (right). Teal is fixed, purple is moveable.

For the simulation we used two pairs of objects, referred to as the "planar maze" and the "gear stick" (see Fig. 5-14). Both are similar to the pattern on a car's gearstick, but the two shapes are different. Also, the "planar maze" allows two translational degrees of freedom, while the "gear stick" allows one rotation and one translation. However, they have isomorphic DoF maps.

For both object pairs, motion planning queries done with a general motion planner like RRTConnect (a well established planner for robotic manipulators) will fail even if allowed five minutes of computation time as the presence of narrow passages and movement constraints makes expansion of roadmaps very difficult.

As a first test, we run five DoF map construction procedures for each of the rigid object pairs, and keep statistics of construction times. The averages are 17.4s for the planar maze and 97.1s for the gear stick; the difference is a result of collision checking being more time consuming for the gear stick because of the more complex geometry. The variance in run times is small (on the order of teths of a second), and is explainable by background thread activity in the laptop. The resulting DoF maps from the construction steps are isomorphic graphs, as expected. Once constructed for a rigid object, a DoF map allows very fast planning queries

(twenty milliseconds) for the same object pair, since we also can reuse pose coordinate information.

As a next test, we ran a motion planning query for the gear stick (see Fig. 5-15), by reusing the DoF map constructed for the planar maze; no pose coordinate information is reused. We assume goal region known, but the priors for the starting region are uninformative (all regions equally likely). Even so, the method quickly identifies the current location (it does so after the transition from the start to the next region), and reaches the goal in an average of 50.1s, about half of the time needed to construct the DoF map, which is consistent with the path necessary to reach the goal being about half the total length of the corridors that needed to be explored for map construction. The results are summarized in Table 5-4.

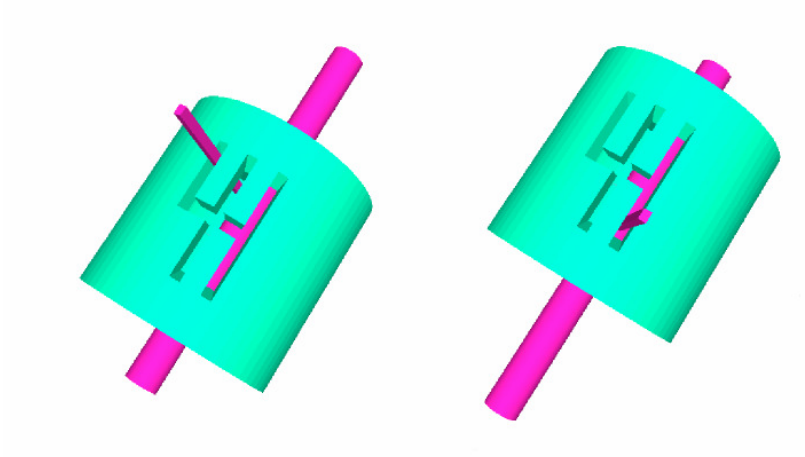

Figure 5-15. A motion planning query on the gear stick: start (left), goal (right).

Table 5-4. Average planning times for the test problems with different planning methods

Test case	Planning method	Average time (s)
Planar maze	RRTConnect	(fails)
	DoF map construction	17.4
Gear stick	RRTConnect	(fails)
	DoF map construction	97.1
	DoF map (planar maze) reuse	50.1

5.6.5 DoF maps as object classification criterion for motion planning

Object classification allows reasoning about classes, rather than particular objects, and is a useful tool to generalize knowledge gained through specific examples[Pan12]. However, this raises the problem of what a useful classification criterion is; and, for this thesis in particular, what a criterion useful for motion planning would be.

The exact shape and size of an object carries too much information and does not generalize well. An object's topology is at the opposite extreme; it carries too little information. For example, an open cupboard with three shelves is topologically identical to a sphere, because there exists a continuous deformation from it to a sphere that does not use tearing nor gluing (inflating the back panel until the cupboard is filled, then rounding the shape). However, one cannot put plates inside a sphere, whereas with a cupboard, one can.

Previous work has focused on fitting simple geometric primitives to an object[Ten13], and while this is a good way to classify what an object "looks like", it's not trivial to extract from this what an object can "do". The DoF map approach presented here focuses precisely on how shapes can restrict each other's movement, and attempts to identify interesting configurations by noticing where and how degrees of freedom change, and what paths exist between these interesting configurations. As such, it is a classification criterion specifically meant for motion planning.

The DoF map is meant as a complementary approach to the vision-based methods that involve shape primitives. Image processing, together with machine learning, can suggest regions of configuration space to explore, which is especially useful if the objects aren't too tightly constrained; for example, the existence of features like holes or toroidal handles can suggest constructing DoF maps near them to see whether two objects can become interlocked. It is the DoF map representation that will function as a bridge between visual object recognition and recognizing motion planning opportunities.

The DoF map is also meant to complement a vision based approach in that object recognition can suggest what DoF map (or maps) may be appropriate for an encountered object pair. The suggested DoF maps will then guide planning, and update confidence in their suggestions, using the procedures outlined above.

5.7 Conclusions

In this chapter we describe a multi-level planning architecture for intricate manipulation tasks. The problem class we consider here is that of entangling and disentangling two rigid objects with/from one another. Sources of difficulty in such problems are the existence of narrow passages, the high dimensionality of the configuration space of the robot, the necessity of using both of a robot's arms, and finding a sequence of grasp changes on the movable object as the robot performs the manipulation. A single-query planner was unable to solve the planning queries even in the lower dimensional configuration space of a rigid object, despite having several minutes of allowed computational time.

Our approach handles the high dimensionality problem by first planning in a smaller dimensional space, that of a rigid object. It uses a roadmap that was previously constructed with human assistance to navigate through narrow passages and between configurations of interest. Once a solution candidate path for the rigid body is found, we use it to guide planning for the arms and grasp selection.

We then improve the efficiency of our multi-level architecture for complex manipulation planning by providing a grasp suggestion heuristic, which ranks grasps by an easily computed expected measure of how good they appear. More expensive testing of grasps is then done in this order, from most to least promising. This ranking allows sequences with few grasp changes to be found.

We then improve the robustness of our multi-level architecture for complex manipulation planning. There is no guarantee that any sequence of arm movements

exists that is capable to follow a given solution path candidate, because while searching for the solution path candidate, we do not consider the higher dimensional configuration space of the robot. To prevent such infeasible solution candidates, we reuse the cost bump concept previously introduced in chapter four, which allows the robot to learn which regions of the rigid object's configuration space are awkward to grasp and should be avoided. Replanning for the rigid object is done if the current solution candidate appears too difficult to follow.

We validate our multi-level planning architecture for complex manipulation. in simulation Unlike single-query planners that fail even with minutes of computation time allowed, our proposed planner architecture is capable to handle queries in reasonable time (a few tens of seconds, depending on the number of grasp changes needed).

We then investigate automatic construction of roadmaps in very constrained environments, so that the robot would not need to rely on a human operator, but instead might identify the narrow passages and interesting configurations on its own. We propose a data structure called a degree of freedom map (or DoF map) to model configuration spaces that can be described as a collection of narrow corridors linking small regions where more degrees of freedom are available, then provide procedures to construct and reuse DoF maps for pairs of rigid objects. We intend for our approach to allow a robot to construct a DoF map using tactile feedback only. This is not meant as a replacement for visual feedback, but integration with visual perception systems remains for future work.

We then argue that the DoF map is a good object classification criterion for motion planning, in that it captures just enough information about the interaction of objects, rather than too much and thus fails to generalize easily, or too little and thus fails to be useful.

We validate the DoF map concept in simulation. We show that classical sample-based planners fail in the highly constrained cases we studied, but that our DoF map construction procedure allows planning queries to be solved. Further, if it is known that two pairs of rigid objects have isomorphic DoF maps, then we show it is necessary to construct only the DoF map for one of the pairs. Then, the same DoF map can be reused to efficiently solve planning queries for the other pair.

6. Going beyond point to point planning

The typical motion planning problem requires a robot to find a way to reach some goal configuration, starting from some other given state. However, real robotic tasks are often more complicated than simply reaching a goal. Sequences of goals may be required[Fai05, Fai09], or additional conditions on the solution imposed, beyond its mere feasibility[Bha10]. An overlap should appear therefore between task planning at some symbolic level, which produces sequences of subtasks and motion plan queries, and motion planning, which must find some feasible maneuvers to perform the queries. Some constraints on task sequences are then at the logical level, and concern what prerequisites tasks have before they can be attempted, and what side-effects they produce. Some other constraints on tasks are at a geometric level, and concern whether there are any trajectories to get the task done. Various ways have been proposed in the literature to allow an integration between the logic level and the geometric level of task planning; among them, the use of temporal logic for planning specifications, to make planners more aware of the sequencing or branching requirements of a task (therefore, more aware of the logical constraints). In this chapter, we look at some planning problems with specifications more complicated than simple reachability, and investigate how to augment roadmap based planners so as to handle some aspects of task planning efficiently. This chapter contains content from the author's paper "Visibility based planners for path existence queries in temporal logic" [Pom14a].

6.1 Planner algorithm for LTL specifications

In this section, we modify and apply the variable radius visibility based planner to problems specified in a subset of temporal logic that can encode specifications on path existence. Goals like visiting some regions in a certain sequence, while staying inside a region from which some other, 'safe haven' region can be reached, are an example of planning problem that can be posed using this fragment of temporal logic. Though the kinds of paths that can be described in the logic are finite, one can also obtain infinite loops by closing the obtained path in on itself with some gap reduction algorithm. We analyze the probabilistic completeness of the visibility based planner, including for systems with non-reversible maneuvers, and provide some simulation results.

6.1.1 A new sparsity heuristic

Visibility based planners attempt to use information already present in the roadmap to decide whether a newly generated sample is worth keeping: either it improves coverage, or the connectivity of the roadmap [Nis99, Lam00b]. Both requirements are formulated in terms of changes to the graph of (strongly) connected components of the roadmap. A change to this graph means that either a new SCC has appeared (the new sample explores a previously uncovered region of the free space), or a connection between two previously unconnected SCCs, or two

or several SCCs have merged into one (the new sample improves roadmap connectivity).

The heuristic described above is probabilistically complete for simple reachability queries of the type that "free space Until free space and goal" formulas can express. We now present a modification to the heuristics, meant to tackle some general formula in the subset of temporal logic we consider here.

Suppose we have a formula Φ for which we want to design a planner. We first construct the syntactic tree of subformulas of Φ; in order to minimize the tree size, we allow locally checkable formulas to be leaves. Based on the syntactic tree, we then create a list of all distinct subformulas of Φ. This list includes Φ itself.

For each subformula φ in the list, we define a subgraph of the roadmap thusly: a vertex belongs to the φ subgraph if the φ formula holds at that vertex; an edge belongs to the φ subgraph if φ holds at all points on the trajectory represented by the edge. An edge may be such that a subformula does not hold all along it however. We'll call such an edge a bridge, because it connects vertices from different subgraphs.

We need to define how to check which subformulas hold at vertices and edges. In the case of vertices and locally checkable formulas, this is obvious as by definition locally checkable formulas are made of conjunctions and disjunctions of atomic propositions, which can be verified knowing only a position in configuration space. Verification of locally checkable subformulas along an edge is analogous to collision checks along the edge in usual planners.

If all along an edge, the same locally checkable formulas hold, then that edge is also a member of whatever locally checkable subgraphs its endpoints are members of. If instead one finds that along an edge there are regions where different locally checkable formulas hold, then one can cut it by either generating new vertices at the points of contact between regions, or inside each region. The resulting edges will then be bridge edges between locally checkable subgraphs.

Suppose then that a subformula φ_1 holds at a vertex, or respectively along an edge. Then that vertex, or respectively edge, is added to the subgraph for a $\varphi_1 \vee \varphi_2$ subformula (if one exists).

Supposing that both φ_1 and φ_2 hold at a vertex, or along an edge, then that vertex (or edge) is added to the subgraph for a $\varphi_1 \wedge \varphi_2$ subformula (if one exists).

If φ_2 holds at a vertex or all along an edge, then that vertex (or edge) is added to the subgraph for a φ_1 U φ_2 subformula, if one exists. Then, one can then recursively add vertices from the φ_1 subgraph to the φ_1 U φ_2 subgraph, if they connect via an edge where φ_1 holds everywhere (except maybe in a region around the destination where φ_2 holds) to a vertex already in the φ_1 U φ_2 subgraph; the edges used for connection are also added to the subgraph.

For each of these subgraphs, one can define a graph of strongly connected components. Let SCC(φ) be that graph for the subgraph defined by the subformula φ. Further, one can define, by way of the briding edges, connections between components in different subgraphs.

Then, we can define the visibility heuristic for a formula Φ, in which we denote subformulas by φ. A new sample is useful if adding it to the roadmap causes a change in at least one of the SCC(φ), or it adds a new connection between components in some SCC(φ_1) and SCC(φ_2). A change in an SCC(φ) means either a change in its number of vertices, where each vertex represents an SCC of the φ subgraph, or the appearance of a new edge in SCC(φ), which means a new connection between SCCs of the φ subgraph. If no such change nor new connection between subgraphs occurs, then the sample, along with any auxiliary samples

produced by edge splits, is rejected. An exception to the heuristic described before, the sample representing the start configuration is always added to the roadmap.

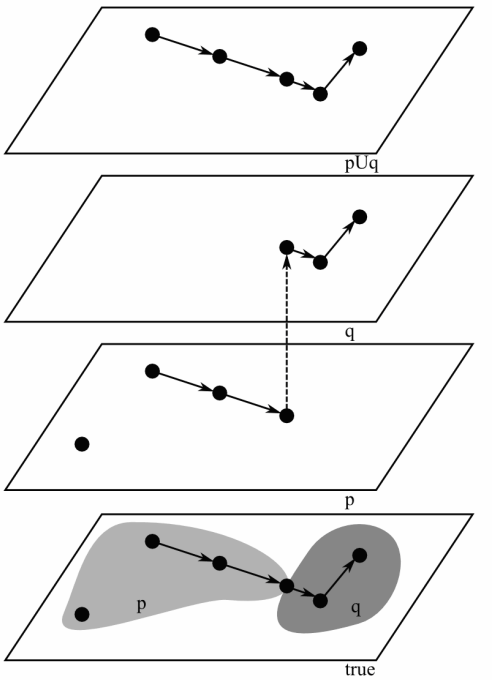

Figure 6-1. An example environment and roadmap (below) and subformula subgraphs (above).

Usually, visibility based algorithms check possible connections between a new sample and all vertices already present in the roadmap. In the interests of computational efficiency, we check the new sample against those vertices that are inside some distance threshold from it, where this distance threshold is decreased by a *pow((log(N)/N);dim(C_{free}))* factor, where *N* is the number of vertices in the roadmap and *dim(C_{free})* is the dimensionality of the free space, until a minimal, "fallback" value is reached, which is used thereafter.

6.1.2 Probabilistic completeness

We will assume that the free space in which planning occurs is a metric space which is also equipped with a measure. For a subformula φ, we define the φ subspace as the subset of points of the configuration space at which φ holds. The roadmap will always contain only an approximation of such a subspace, but we use this theoretical construct here to check the algorithm.

It is often considered natural to use some approximation of the cost to go as a metric for C_{free}. For dynamic systems, it is usually a pseudometric as it may be the case that the cost to go from some state ***s*** to another ***s***′ differs from the cost to go from ***s***′ to ***s***. One can define a proper metric by selecting the minimum cost from

between the two, and thus one can, at least in theory, define a metric on C_{free} that is related to the system dynamics.

Let the reunion of all φ subspaces, where φ here represents a locally checkable formula, be called the subspace of interest, C_{free}. To be relevant for the specification, a state must satisfy at least one of the locally checkable subformulas, therefore it must be a member of the subspace of interest, and one can safely restrict the sampling procedure to this subspace.

For some φ subspace, one may find that it contains several path-connected components. Here, path connected refers simply to the existence of some path in the subspace topology induced by C_{free}, not necessarily that said path is feasible for the robot.

We require that the sampling procedure be "fair": given any subset of any path connected component of some φ subspace, such that the subset has the same dimensionality as the component it is included in, then the chance to produce a sample inside that subset is non-zero.

Typically, the fair sampling requirement is simplified to uniform sampling, if we also require that all φ subspaces be of measure non-zero. In general however, it may happen that some atomic proposition holds only on some surface inside C_{free}. This is not necessarily an obstacle to probabilistic completeness as long as the sampling procedure is guaranteed to generate samples on that surface as well. Note that a uniform sampling procedure over C_{free} will almost surely not generate a sample inside a surface in C_{free}, because a surface has 0 volume according to the measure defined over C_{free}.

For a sample $\boldsymbol{s}$, let $R(\boldsymbol{s}, r)$ be the set of states reachable from $\boldsymbol{s}$ with a cost, or distance, of at most r, where $r \geq 0$. We require that $R(\boldsymbol{s}, r)$ be a closed set, and in keeping with the distance/cost semantics, if $r_1 < r_2$ then it must follow that $R(\boldsymbol{s}, r_1) \subset R(\boldsymbol{s}, r_2)$. Finally, it must be the case that $\cup_{\{r < r0\}} R(\boldsymbol{s}, r)$ is open. We similarly define $S(\boldsymbol{s}, r)$ as the set of points that may reach $\boldsymbol{s}$ with a cost, or distance, of at most r, and place similar conditions.

We require that the local planner have the "wobbly free boundary" property: let $\boldsymbol{s}$, $\boldsymbol{s}'$ be any states inside C_{free} generated by the sampling procedure. The it must almost surely be the case that either $R(\boldsymbol{s}, r) \cap S(\boldsymbol{s}', r) = \varnothing$ or $R(\boldsymbol{s}, r) \cap S(\boldsymbol{s}', r)$ is a subset in which the sampling procedure has a non-zero chance of generating a sample. Often, the latter condition translates into the intersection being a measure non-zero set, however if we allow certain surfaces inside C_{free} to have non-zero probability to be sampled, then we also allow $R(\boldsymbol{s}, r) \cap S(\boldsymbol{s}', r)$ to contain a subset of such a surface, of the same dimension as the surface.

Intuitively, "wobbly free boundary" ensures that either there exists some space between two samples that neither can reach or be reached from if given a distance limit, or that their reachability zones overlap in such a way that the overlap will eventually be sampled.

Finally, we require that for any subformula φ, the φ subspace be compact.

Note that coverage of any subformula subspace can only improve as the planner adds new samples to its corresponding subgraph because of the "more isn't less" lemma. We will now show that a finite number of attempted samples will be sufficient for coverage.

Define "finite total coverability" to be a property of a φ subspace that means that, for any infinite sequence of samples taken from C_{free}, there exists a finite sequence of samples, starting with the first, such that the subspace is covered by (completely included in) the union of R and S sets for all the samples in the sequence that are also themselves inside that subspace. In other words, a subspace

is finitely totally coverable if it will eventually be completely covered by the *R* and *S* sets of samples generated inside it by any realization of the fair sampling process.

Finite total coverability follows easily for any compact set if the sampling is fair, since if the negation were true it would follow there is some open subset in which no sample is generated, in violation of the fair sampling assumption. We also assumed that subspaces for all subformulas are compact, therefore they are all finitely totally coverable.

The planner constructs a covering of φ subspace by placing samples in the φ subgraph. It must now be shown that, in a fair sampling process run forever with a wobbly free boundary local planner, the algorithm will attempt to place an infinite number of samples in all subgraphs in such a way that, if all those samples were kept, they would generate a complete covering for all the subspaces. Since all subgraphs correspond to finitely totally coverable subspaces, this implies that eventually, after some finite number of samples taken and kept, the planner will have constructed a covering for all of the subspaces corresponding to subformulas of the problem specification. We call this property "finite total constructible cover" or FTCC, and will now prove that all subspaces corresponding to subformulas in the specification have it. We do this in a manner similar to structural induction on the subformulas.

From the compactness of locally checkable subspaces and the fairness of sampling, it follows immediately that the locally checkable subspaces are FTCC. For if the negation were true, it would imply that there is some subset in the space which is not ever covered, even by an infinity of samples; but that means that no samples are ever generated there, in contradiction with the fair sampling assumption.

Suppose then that φ_1 and φ_2 are (not necessarily locally checkable) subformulas that define subspaces that are FTCC. Then it follows easily that $\varphi_1 \vee \varphi_2$ is FTCC.

It also follows that $\varphi_1 \wedge \varphi_2$ is FTCC. Since a fair sampling procedure will generate, over any realization of the sampling process if extended forever, an infinity of samples inside the subspace corresponding to $\varphi_1 \wedge \varphi_2$, and since this subspace is by definition compact and thus finitely totally coverable, one finds that a finite sequence of samples will eventually cover it completely with the union of its *R* and *S* sets.

Finally, consider the subspace corresponding to a subformula $\varphi_1 \; U \; \varphi_2$, where the subspaces for φ_1 and φ_2 are FTCC. The subspace for $\varphi_1 \; U \; \varphi_2$ can be thought of as containing two components, both of them compact and thus finitely totally coverable: the subspace corresponding to φ_2 and a subset of the φ_1 subspace, of points that may reach φ_2 points; let the latter component be referred to as *X*.

Since the φ_1 and φ_2 subspaces are FTCC, it will be the case that after some finite number of samples they will be totally covered. Because of the wobbly free boundary assumption, we have that there will exist an overlap between *R* sets of φ_1 vertices and *S* sets of φ_2 vertices, in which some samples will be eventually placed by the planner. Therefore we have that after a finite number of samples, some φ_1 samples have been placed inside *X* and thus inside $\varphi_1 \; U \; \varphi_2$. In fact, all samples in the roadmap that are in φ_1 and are then known to reach φ_2 samples will be placed in $\varphi_1 \; U \; \varphi_2$ and thus in *X*.

Thereafter, the planner, using a fair sampling procedure and a wobbly free boundary local planner, will improve the coverage of *X* either by discovering samples that connect previously known vertices from the φ_1 subgraph to vertices

already in X, or by finding new samples from φ_1 that can reach samples already in X. Either event will happen infinitely often in a fair sampling process continued forever, and since X is compact and therefore finitely totally coverable, it follows that it is also FTCC.

Therefore, one finds that the subspace corresponding to the specification formula is FTCC. Therefore, if a plan exists to satisfy the specification, the vertex corresponding to the starting configuration will eventually be placed inside the subgraph corresponding to the specification formula, and it will also be the case that a path will exist in the roadmap that will satisfy the specification formula.

Therefore, the algorithm is probabilistically complete; the chance that it finds a plan that meets the specification (if one exists) tends to certainty as the number of sampling attempts goes to infinity.

6.1.3 Changing the start configuration

In the previous sections, we considered that the start configuration is known from the start and added to the roadmap. Should we desire to change it however, one simply adds the new start configuration to the roadmap, and thereafter proceeds with sample-and-connect steps, using the algorithm described before. If the new start is inside the subspace corresponding to the specification, it will eventually be added to the subgraph corresponding to that subspace of C_{free}.

6.1.4 Extracting a plan from the roadmap

Checking that a plan exists is made very easy by the process of roadmap construction. If the starting vertex is inside the subgraph corresponding to the specificaiton, then a plan exists; otherwise, it does not.

Once a roadmap is constructed, the issue remains to extract a plan from it, if one exists, to meet a given specification. Tools for general LTL formulas exist, like SPIN [Hol04] and NuSMV [Cim02], which have also been used in a planning context. They check a formula by providing a counter-example to its negation, if such a counter-example can be found, and if it can be, the counter-example is the sought after plan.

SPIN and NuSMV are capable of handling general LTL formulas, outside the subset of interest to this paper. Restricting to the subset of "existence of paths" formulas allows one to work with a formula directly, instead of requiring its negation, so we present a plan finding procedure specialized to this subset of LTL and which makes use of the auxiliary structures maintained by the planning algorithm.

The basic planning query, "pUq" where p and q are some locally checkable statements, is typically done with a Dijkstra's Shortest Paths algorithm, which provides a distance map over the vertices in the roadmap. Each vertex has associated with it the smallest cost required to reach it from some given starting vertex. Based on the distance map, the shortest path between the given starting vertex, and any other vertex in the graph, can be obtained.

Further, one can use the Dijkstra algorithm to find all the q nodes which are such that there is a shortest path to them, from the initial configuration, that does not pass through any other q node. We'll refer to the set of such nodes as a front.

While finding a plan, we'll need to concatenate path segments. We say that two paths can be concatenated if the end of the first is the same vertex as the start of the second. The cost of the concatenation of paths is the sum of their cost. We say that an infinite cost path is infeasible. In practice, some other flag variable will be used to signal unfeasibility, but thereafter we'll use infinite cost for this purpose because of its intuitive semantics; if an infinite cost segment is added to a path, it renders it unfeasible.

We can now define a function, FindFront, which takes as input a start configuration, and an LTL formula representing a specification on paths. It will return a list of front vertices and the paths to them from the starting vertex. We will now specify the behavior of FindFront in more detail.

If the starting vertex is not in the subgraph corresponding to the LTL formula, then the function returns one path, containing just the starting vertex, which is said to have infinite cost. We therefore have a quick test to check whether searching for a plan is fruitless because none exists.

If the LTL formula is locally checkable, and is obeyed at the starting vertex, then FindFront returns a path containing just the starting vertex, of cost *0*.

If the LTL formula is of the $\varphi_1 \, U \, \varphi_2$ type, then FindFront will restrict itself to the subgraph corresponding to the formula. Assuming the starting vertex can be found in this subgraph (or else, an infinite cost path containing just the start vertex would have been returned), FindFront will run a Dijkstra algorithm and locate the φ_2 front vertices and the paths toward them. After that, for each front vertex *v*, a new instance of FindFront is called, with *v* as start vertex and φ_2 as the formula. The return value of the upper level FindFront is then the set of paths obtained by concatenating, to the paths to each front vertex *v*, the paths obtained for that vertex by the lower level FindFront.

If the LTL formula is of the $\varphi_1 \wedge \varphi_2$ type, where one of the formulas, say φ_1, is locally checkable, then FindFront first checks that the starting vertex is inside the $\varphi_1 \wedge \varphi_2$ subgraph. If it is (which implies that φ_1 also holds at it), then another instance of FindFront is called with the same starting vertex and φ_2 as the formula. The return value for the upper level FindFront is then the return value from the lower level one.

If the LTL formula is of the $\varphi_1 \wedge \varphi_2$ type, where neither formula is locally checkable, then one should first use rewrite rules to bring it to a form in which the $\wedge$ operator always has at least one locally checkable formula as operand. Some useful rewriting rules are summarized in table 6-1; notice that the rewrite rules tend to shorten the formulas appearing as operands to the $\wedge$ operator, and therefore eventually we will only apply it to operands out of which at least one is locally checkable.

Table 6-1. Some formula rewrite rules for the path existence LTL fragment

$(\varphi_1 \vee \varphi_2) \wedge \varphi_3$ $\varphi_1 \wedge (\varphi_2 \, U \, \varphi_1)$	$(\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ φ_1
$\varphi_1 \vee (\varphi_2 \, U \, \varphi_1)$	$\varphi_2 \, U \, \varphi_1$
$\varphi_1 \, U \, (\varphi_1 \, U \, \varphi_2)$	$\varphi_1 \, U \, \varphi_2$
$(\varphi_1 \, U \, \varphi_2) \wedge (\varphi_3 \, U \, \varphi_4)$	$((\varphi_1 \wedge \varphi_3) \, U \, (\varphi_2 \wedge (\varphi_3 \, U \, \varphi_4))) \vee$ $((\varphi_1 \wedge \varphi_3) \, U \, (\varphi_4 \wedge (\varphi_1 \, U \, \varphi_2)))$

Finally, if the LTL formula is of the $\varphi_1 \vee \varphi_2$ type, then two instances of FindFront are called, both with the same start vertex, but one with the φ_1 and the other with the φ_2 formula. The return value of the upper level FindFront is the union of the return values of the lower level FindFronts.

Looking for a plan then requires that a FindFront be called, with the starting vertex and plan specification. From the resulting set of paths, one can pick the lowest cost one as the plan to follow.

6.2 Simulation verification

We apply the planner to the problem used for simulation verification of RRG in [Kar09], which asks for a discrete time linear dynamic system to be steered towards a looping trajectory that passes through two specified regions while avoiding a third. The system is characterized by the equations of state:

$$\begin{bmatrix} x[k+1] \\ y[k+1] \end{bmatrix} = \begin{bmatrix} 1.019 & -0.029 \\ 0.049 & 0.95 \end{bmatrix} . \begin{bmatrix} x[k] \\ y[k] \end{bmatrix} + \begin{bmatrix} 0.101 & -0.0015 \\ 0.0025 & 0.098 \end{bmatrix} . \begin{bmatrix} u_1[k] \\ u_2[k] \end{bmatrix}$$

from which it is straightforward to define a local steering procedure between arbitrary positions, assuming obstacles are not in the way. Note further that the system is fully reversible.

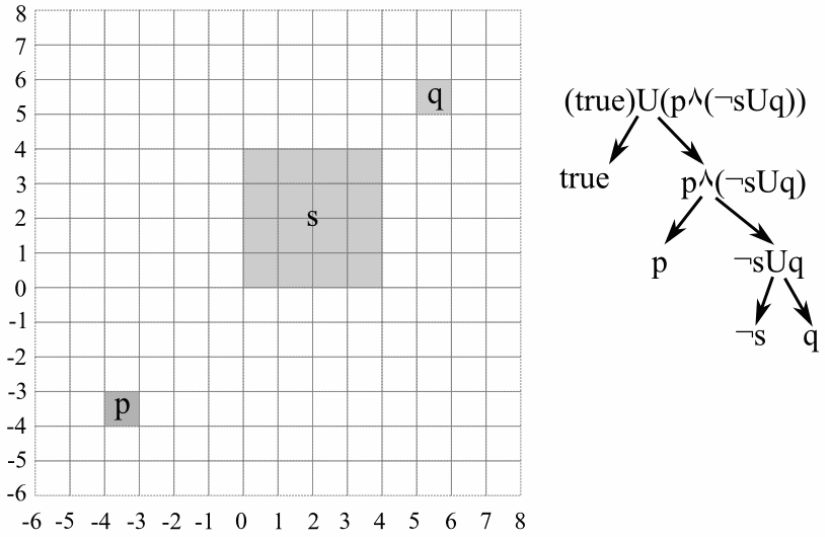

Figure 6-2. Problem environment and syntactic tree for the specification.

The environment is shown in Fig. 6-2. The system starts at *(0, 0)*, on the edge of the *s* region. We require that it reach the *p* region, then the *q* region while avoiding *s*, then the *p* region again while avoiding *s*. We then formulate a specification:

$$(true)\,U\,(p \wedge ((\neg s)\,U\,(q)))$$

Note that, while the specification above produces a finite path, the problem in [Kar09] requires a loop to be formed between *p* and *q*, which avoids *s*. To close

the loop, notice that the system's reversibility allows the *s*-free path from *p* to *q* to be used in reverse.

6.2.1 Customizing the visibility heuristic

We will first make an inventory of the distinct subformulas that we need to track: *true* (which has the roadmap in its entirety as representative subgraph), *p*, *q*, $\neg s$, $(\neg s)U(q)$, $p \wedge ((\neg s)U(q))$, and finally the specification itself, $(true)U(p \wedge ((\neg s)U(q)))$. Each subformula will have a subgraph in the roadmap to represent it; vertices and edges inside a subformula's subgraph satisfy that subformula, meaning a path exists which starts at the vertex, or the point on the edge, and satisfies the subformula.

We maintain connected components for each subgraph using Tarjan's set union algorithm[Tar75], and keep track of edges between the subgraphs. If a new sample changes the graphs of strongly connected components or the connections between formula subgraphs, it is kept in the roadmap. Sampling for this problem will be uniform on the problem area.

6.2.2 Results

One thousand runs of the algorithm are performed, and statistics on final roadmap size and rejected sample counts are shown in table 6-2. As can be seen, the size of the roadmap is reliably small, as an average of nine samples is sufficient to find a suitable path. In comparison, RRG used more than one thousand samples for the same problem [Kar09].

One notices that the average number of rejected samples is around the same order of magnitude as the number of samples used by the RRG, suggesting that a uniformly sampling planner needs a few hundred attempts in order to pick some samples useful for a solution. The visibility based planner however can determine that most of those samples do not improve the ability of the roadmap to solve the problem.

In terms of time spent, whereas RRG requires several seconds [Kar09], the visibility based planner is nigh-instant. Although roughly the same number of sample and connect steps are made, the fact that the roadmap is kept dramatically smaller makes each of these steps much less expensive.

Table 6-2. Sample count statistics for the visibility planner

Samples	Avg	StdDev	Max	Min
Kept	8.84	1.63	12	5
Rejected	368.6	257.43	1445	5

We have presented an extension of the visibility heuristic which is applicable to planning problems written as specifications in a path-existence subset of temporal logic, defined an algorithm to handle such specifications, and showed that it is probabilistically complete.

It should be noted that while the kinds of specifications the algorithm can natively handle are about existence of paths, it may be useful in some cases where

the existence of an infinite loop is sought. To form a loop, one would need to get a path from some starting point, to a destination, then back inside the region of the starting point, then use some gap closing procedure to close the loop.

The formulation of the planning problem used here assumed perfect actuation. Probabilistic temporal logics exist which account for errors in motion [Lah10, Ciz12], and it may be possible to extend the strategies presented here for planning specifications written in such logics. Other methods for obtaining sparse roadmaps besides visibility exist, including methods which aim for some guarantee of partial optimiality. It may be fruitful to apply the subgraph constructions presented here to such methods, so that partially optimal, sparse roadmaps are made possible for general path existence temporal logics specifications, which would be useful in contexts like grasping and manipulation. Both of the previous topics are left for future work.

6.3 Conclusions

In this chapter we give a proof of probabilistic completeness for visibility based planners applied to more complex planning queries specified in a path-existence subset of LTL. These queries allow specification of not only go-to location queries, but also sequencing, region avoidance based on step along a sequence, staying inside a reachability region in case falling back to another region is necessary, robot synchronization and coordination, task planning etc. We show the planner needs to maintain subformula subgraphs, and we use the outline provided by the proof to describe a probabilistically complete visibility based planner for path existence LTL, which is novel.

We then show how the subformula subgraphs can be used to quickly test whether a roadmap contains a path capable to satisfy a plan specification, then give a method to search for such a path.

Path-existence LTL describes only open paths, however we show how our planner can be enhanced into handling a specification requiring a possibly infinite looping trajectory by first planning to obtain a path equivalent to the opened loop, then using a gap reduction step to close the loop ends.

We simulate and check our visibility based planner against another planner capable to handle LTL specifications (RRG), on a problem which appeared in the paper that proposed the RRG algorithm [Kar09]. Simulation shows our planner can solve the problem faster and with fewer samples tested than what was reported for RRG [Kar09].

7. Contributions

7.1 Summary of results

In this thesis we have studied how multi-query roadmap algorithms can be adapted and efficiently applied to more challenging motion planning problems in the field of robotics. The theoretical contributions are:

1) a new proof of probabilistic completeness for visibility planners applied to reversible systems that makes clear the assumptions needed on the interaction between the local trajectory planner used to generate connections between vertices in a roadmap and the shape of the free space. Our proof is more general than that of [Nis99], which applies only to local planners that generate linear trajectories for point robots.

2) extend our proof to systems that have non-reversible maneuvers and non-holonimc constraints, and provide a guarantee for probabilistic completeness for a suitably modified visibility based planner. We propose such a modification to the visibility heuristic.

3) further extend the proof of probabilistic completeness of visibility based planners for more complex planning queries specified in a path-existence subset of LTL. Such queries allow specification of not only go-to location queries, but also sequencing, region avoidance based on step along a sequence, staying inside a reachability region in case falling back to another region is necessary, robot synchronization and coordination etc, and are relevant for task planning. Our proof shows the need for a planner to maintain subformula subgraphs, and we use the outline provided by the proof to describe a probabilistically complete visibility based planner for path existence LTL, which is novel.

4) we describe a multi-level planning architecture for intricate manipulation tasks. We focus on the problem of entangling and disentangling two rigid objects with one another, a problem made difficult by the existence of narrow passages. Further, the manipulation problem requires the use of both of a robot's arms, therefore a solution must contain a sequence of grasp changes on the movable object as the robot performs the manipulation; this adds a further complication, since the number of dimensions of the space to explore is increased. A single-query planner proved unable to solve the planning queries in several minutes. On the other hand, our approach tackles the dimensionality problem by first planning in a smaller dimensional space, that of a rigid object. To help with narrow passages, it uses a roadmap that was previously constructed with human assistance for identifying narrow passages and configurations of interest. Once a plan for the rigid body is found, we use it to guide planning for the arms and grasp selection.

5) we investigate automatic construction of roadmaps in very constrained environments, so that the robot might identify the narrow passages and interesting configurations on its own, rather than rely on a human operator. We propose a data structure called a degree of freedom map (or DoF map) and present procedures to

construct and reuse such maps for pairs of rigid objects, when the free configuration space can be described as one-dimensional corridors linking small juncture regions where several degrees of freedom are available. Our approach is intended to allow a robot to construct a DoF map using tactile feedback only, and extends previous work in the literature which has explored single-degree-of-freedom configuration spaces by sense of touch.

6) we argue for the DoF map as an object classification criterion that would be useful to motion planning.

7) we describe a variable radius visibility planner for reversible systems (VRV) and analyze its probabilistic completeness and computational complexity per sample and connect iteration.

8) we propose the usage of non-zero-dimensional sample subspaces, in contrast with the usual sampling based approach where each roadmap vertex is a point in configuration space. We show that using non-zero-dimensional subspaces as samples greatly speeds up collision checking during the roadmap construction and expansion phases.

9) We show how the subformula subgraphs can be used as quick tests of whether a roadmap contains a path capable to satisfy a plan specification, and provide a method to extract such a path if it exists.

10) though path-existence LTL describes only open paths, we show how our planner can be enhanced into handling a specification that requires a looping trajectory by first planning to obtain a path equivalent to the opened loop, then using a gap reduction step to close the path ends.

11) we propose a cost bump method as a way to steer graph search algorithms on the planner's roadmap away from regions that may be invalid or inside obstacles. We further show how the cost bump method can be used as a way for the planner to learn an approximate shape of the free space of the robot. We stress that the three dimensional representation of the environment that a robot may get through its sensors is not the same as its free space, which has as many dimensions as the robot has degrees of freedom.

12) we improve the efficiency of our multi-level architecture for complex manipulation planning by proposing a grasp suggestion heuristic, which orders expensive grasp testing according to an easily computed expected measure of how good a grasp appears.

13) we improve the robustness of our multi- level architecture for complex manipulation planning. Planning for the rigid object first doesn't guarantee there is any sequence of arm movements capable to make it follow that plan, so to prevent such infeasible solution candidates, we reuse the cost bump concept previously introduced in chapter four, to allow the robot to learn which regions of the rigid object's configuration space are awkward to grasp and should be avoided.

Applicative/experimental contributions are:

14) we verify our variable radius visibility algorithm in simulation and show it is capable to generate more compact roadmaps than classical sample based planners (PRM) to capture the connectivity of a configuration space.

15) we implement and verify our visibility planner for non-reversible systems in simulation on a variety of vehicle models. Application of visibility based planners to such systems is new in the literature, as so far such problems would have been handled by single-query planenrs. Our approach is capable to construct a

compact and reusable roadmap, which would make future queries faster than resorting to single-query planners.

16) we validate our LTL planner in simulation by checking it against another planner capable to handle LTL specifications (RRG). Our planner is capable to solve the problem faster and with fewer samples tested than RRG.

17) we validate our cost bump approach in simulation and experimentally on a PR2 from Willow Garage. We show that it can outperform single query planners, both in that it reduces the planning time by a factor of two or better, but it also tends to produce shorter, more efficient paths. Key to this performance is the fact that the roadmap used by the planner is compact, which allows fast queries, and that the roadmap is also capable to capture connections through the configuration space, the way visibility based planners can. We also show that a simple, classic solution based on Lazy PRM would not be able to outperform single-query planners for a robotic manipulator, and hence an approach such as our cost bump is necessary.

18) we validate our multi-level planning architecture for complex manipulation. Unlike single-query planners that fail even with minutes of computation, our proposed planner architecture is capable to handle queries in reasonable time (a few tens of seconds, depending on the number of grasp changes needed).

19) we show that classical sample-based planners fail in the highly constrained cases we studied, but that our DoF map construction procedure allows planning queries to be solved.

20) we show that if it is known that two pairs of rigid objects have isomorphic DoF maps, it's necessary only to construct the DoF map for one of the pairs. Then, that DoF map can be reused to efficiently solve planning queries for the other pair. Objects of very different geometry may have isomorphic DoF maps, and by design the DoF map structure captures the kinematic interaction between rigid objects.

7.2 Contributed papers

"Visibility based planners for kinematically constrained vehicles", in proceedings of the 8th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), 2013.

(with Ioan A. Şucan) "Motion planning for manipulators in dynamically changing environments using real-time mapping of free space", in proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), 2013.

(with Ioan A. Şucan) "A two-level approach for intricate manipulation planning", in proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), 2013.

"Mapping kinematic interactions between objects for robot motion planning", in proceedings of the 12th IEEE International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2014.

"Visibility based planners for path existence queries in temporal logic", accepted for the Advances in Electrical and Computer Engineering Journal.

(with Ioan A. Şucan) "Improving reliability and efficiency of intricate manipulation planning through mapping of grasp feasibility zones", accepted for ICRA 2014.

7.3 Future work

Open source robotics software packages like MoveIt! offer, at present, very limited support for either multi-query planners or task planning, and integration of our algorithms, in particular the planner for path-existence LTL and the multi-level planning approach for intricate manipulation, would be a useful addition to such software that would also allow more extensive testing on real robots.

Research-wise, one direction that stands out is extending the DoF map concept to allow description for spaces with large regions with multiple degrees of freedom. In particular, such an extension would require interaction with an object recognition/image processing pipeline as well as force feedback from the robot's end effector. The image processing would have the task to recognize whether a rigid object pair 'looks like' it has a DoF map that's isomorphic to an already known one. Conversely, the DoF map approach would augment the image processing, in that it is able to get some information about the objects' shape even when occlusions prevent 3D models of them from being reconstructed through vision. The DoF map would also help train image processing to look for regions of space that may be 'interesting', because they are junctures of narrow corridors. Identifying narrow passages through geometric analysis is a computationally costly operation; instead, the approach we propose here is a machine learning one, which makes a hypothesis about certain regions being narrow passages, tests the hypothesis, and remembers visual features of the interesting regions found.

The DoF map concept should further be generalized to deformable objects like strings, ropes, or sheets of cloth, since several tasks a robot might be required to perform in a human environment may involve disentangling, knotting/unknotting, or folding such objects.

The long term goal is to experimentally validate the DoF map concept as a way to classify objects for motion planning, which allows the robot to abstract most of the objects' geometry and reason about them at the logical, task planning level; it is at this level that the robot realizes how the objects affect each other, and also at this level that the robot needs to discover what preconditions an action has, as well as its results on the world state. So far, robotic ability to reason about objects involved in a task plan is limited, as the actions, pre-conditions and effects must be defined by human operators, and apply to fairly limited test cases; the robot cannot easily generalize from a specific object or action to a concept of the object or action class. It is our conjecture that the DoF map can improve generalization and abstract reasoning about tasks. Note that the tasks humans encounter every day (and solve easily with experience and habit) often involve more than two objects interacting with each other. Nevertheless, we conjecture that a good portion of those multi-object interactions can be described by reducing them to several pairs of interacting objects, and applying a suitably extended DoF map concept.

References

1: Ama96: Nancy M. Amato, Yan Wu, "A randomized roadmap method for path and manipulation planning", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 1996.

2: Arg09: B. Argall, S. Chernova, M. Veloso, B. Browning, "A survey of robot learning from demonstration", Robotics and Autonomous Systems, vol. 57, no. 5, 2009.

3: Apa04: Mehmet S. Apaydin, "Stochastic roadmap simulation: an efficient representation and algorithm for analyzing molecular motion", PhD thesis, Stanford University, 2004.

4: Atk08: Christopher G. Atkeson, Benjamin Stephens, "Random sampling of states in dynamic programming", IEEE Transactions on Systems, Man and Cybernetics part B: Cybernetics, vol. 38, no. 4, 2008.

5: Bar91: Jérôme Barraquand, Jean-Claude Latombe, "Robot motion planning: a distributed representation approach", International Journal of Robotics Research, vol. 10, no. 6, 1991.

6: Ber09: D. Berenson, S. Srinivasa, D. Ferguson, J. J. Kuffner, "Manipulation planning on constraint manifolds", in proceedings of the IEEE Intl. Conference on Robotics and Automation (ICRA), 2009.

7: Bha10: A. Bhatia, L. E. Kavraki, M. Y. Vardi, "Sampling based motion planning with temporal goals", in IEEE Intl. Conference on Robotics and Automation (ICRA), 2010.

8: Boh00: R. Bohlin, L. E. Kavraki, "Path planning using Lazy-PRM", in proceedings of the IEEE Intl. Conference on Robotics and Automation, 2000.

9: Bul02: Francesco Bullo, Kevin M. Lynch, and Andrew D. Lewis, "Controllable kinematic reductions for mechanical systems: concepts, computational tools, and examples", in proceedings of the MTNS02, 2002.

10: Bul10: Francesco Bullo and Andrew D. Lewis, "Geometric control of mechanical systems", Springer-Verlag, 2010.

11: Cam04: T. S. Cambon, J. P. Laumond, J. Corts, A. Sahbani, "Manipulation planning with probabilistic roadmaps", International Journal of Robotics Research, vol. 23, no. 7, 2004.

12: Can87: John F. Canny, "The complexity of robot motion planning", PhD Thesis, Massachussetts Institute of Technology, 1987.

13: Che02: Peng Cheng, Steven M. LaValle, "Resolution complete rapidly-exploring random trees", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2002.

14: Che07: Peng Cheng, George Pappas, Vijay Kumar, "Decidability of motion planning with differential constraints", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2007.

15: Chi08: Hamid R. Chitsaz, "Geodesic problems for mobile robots", PhD Thesis, University of Illinois, 2008.

16: Cho05: H. Choset, K. M. Lynch, S. hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, S. Thrun, "Principles of robot motion: theory, algorithms, and implementations", MIT Press, 2005.

17: Cio10: M. Ciocarlie, K. Hsiao, E. G. Jones, S. Chitta, R. B. Rusu, I. A. Sucan, "Towards reliable grasping and manipulation in household environments", in proceedings of the Intl. Symposium on Experimental Robotics (ISER), 2010.

18: Cim02: A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: an open-source tool for symbolic model checking", in proceedings of International Conference on Computer-Aided Verification (CAV), 2002.

19: Ciz12: Igor Cizelj and Calin Belta, "Control of noisy differential-drive vehicles from time-bounded temporal logic specifications", CoRR, abs/1209.1139, 2012.

20: Con90: C. I. Connolly, J.B. Burns, R. Weiss, "Path planning using Laplace's equation", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 1990.

21: Dal09: S. Dalibard, J. P. Laumond, "Control of probabilistic diffusion in motion planning", Algorithmic Foundations of Robotics VIII, Springer 2009.

22: Dia09: Rosen Diankov, Takeo Kanade, James Kuffner, "Integrating grasp planning and visual feedback for reliable manipulation", in proceedings of the IEEE-RAS International Conference on Humanoid Robots, 2009.

23: Dor09: Christian Dornhege, Marc Gissler, Matthias Teschner, Bernhard Nebel, "Integrating symbolic and geometric planning for mobile manipulation", IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR), 2009.

24: Eme86: E. A. Emerson and J. Y. Halpern, ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic", J. ACM, 33(1):151–178, 1986.

25: Fai05: Georgios E. Fainekos, Hadas Kress-gazit, and George J. Pappas, "Temporal logic motion planning for mobile robots", in Proceedings of the 2005 IEEE International Conference on Robotics and Automation, pages 2020–2025, 2005.

26: Fai09: G. E. Fainekos, A. Girard, H. Kress-Gazit, G. J. Pappas, "Temporal logic motion planning for dynamic robots", Automatica, vol. 45, 2009.

27: Fra01: Emilio Frazzoli, "Robust hybrid control for autonomous vehicle motion planning", PhD Thesis, Massachussetts Institute of Technology, 2001.

28: Gay07: Russel Gayle, Avneesh Sud, Ming C. Lin, Dinesh Manocha, "Reactive deformation roadmaps: motion planning of multiple robots in dynamic environments", in proceedings of IEEE/RS International Conference on Intelligent Robots and Systems (IROS), 2007.

29: Gay09: Russel Gayle, Avneesh Sud, Erik Andersen, Stephen J. Guy, Ming C. Lin, Dinesh Manocha, "Interactive navigation of heterogenous agents using adaptive roadmaps", IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 1, 2009.

30: Gla10: Elena Glassman, Russ Tedrake, "LQR-based heuristics for rapidly exploring state space", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2010.

31: Goo02: "Motion planning for kinematic stratified systems with application to quasi-static legged locomotion and finger gaiting", IEEE Transactions on Robotics and Automation, 2002.

32: Gra03: F. Gravot, S. Cambon, R. Alami, "ASyMov: a planner that deals with intricate symbolic and geometric problems", in proceedings of the Intl. Symposium on Robotics Research, 2003.

33: Gui09: J. Guitton, J. L. Farges, "Taking into account geometric constraints for task-oriented motion planning", in proceedings of the ICAPS Workshop on Bridging the gap between Task and Motion Planning (BTAMP), 2009.

34: Had08: S. Haddadin, A. Albu-Schaeffer, A. De-Luca, G. Hirzinger, "Collision detection and reaction: a contribution to safe physical human-robot interaction", in proceedings of the IEEE Intl. Conference on Intelligent Robots and Systems (IROS), 2008.

35: Had11: S. Haddadin, R. Belder, A. Albu-Schaeffer, "Dynamic motion planning for robots in partially unknown environments", in proceedings of the IFAC World Congress (IFAC), 2011.

36: Hae12: B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan, "Incremental cycle detection, topological ordering, and strong component maintenance", ACM Transactions on Algorithms, 8(1), 2012.

37: Hau09: K. Hauser, J. C. Latombe, "Integrating task and PRM motion planning: dealing with many infeasible motion planning queries", in proceedings of the ICAPS Workshop on Bridging the gap between Task and Motion Planning (BTAMP), 2009.

38: Hol04: G.J. Holzmann, "The Spin model checker primer and reference manual", Addison-Wesley, 2004.

39: Hoo12: H. van Hoof, O. Kroemer, H. Ben Amor, J. Peters, "Maximally informative interaction learning for scene exploration", in proceedings of the IEEE Intl. Conference on Intelligent Robots and Systems (IROS), 2012.

40: Hsu03: D. Hsu, T. Jiang, J. Reif, Z. Sun, "The bridge test for sampling narrow passages with probabilistic roadmap planners", in proceedings of the IEEE Intl. Conference on Robotics and Automation, 2003.

41: Jai05: Leonard Jaillet, Anna Yershova, Steven M. LaValle, Thierry Simeon, "Adaptive tuning of the sampling domain for dynamic-domain RRTs", in proceedings of IEEE/RS International Conference on Intelligent Robots and Systems (IROS), 2005.

42: Kae11: L. Kaelbling, T. Lozano-Perez, "Hierarchical task and motion planning in the now", in proceedings of the IEEE Intl. Conference on Robotics and Automation, 2011.

43: Kal01: Maciej Kalisiak, Michiel van de Panne, "A grasp-based motion planning algorithm for character animation", Journal of Visualization and Computer Animation vol. 12, no. 3, 2001.

44: Kar09: Sertac Karaman, Emilio Frazzoli, "Sampling-based motion planning with deterministic μ-calculus specifications", in proceedings of the IEEE Conference on Decision and Control (CDC), 2009.

45: Kar10: Sertac Karaman, Emilio Frazzoli, "Incremental sampling-based algorithms for optimal motion planning", in proceedings of Robotics: Science and Systems (RSS), 2010.

46: Kav96: Lydia E. Kavraki, Petr Švestka, Jean-Claude Latombe, Mark H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces", IEEE Transactions on Robotics and Automation, vol. 12, no. 4, 1996.

47: Kha86: Oussama Khatib, "Real-time obstacle avoidance for manipulators and mobile robots", The International Journal of Robotics Research, vol. 5, no. 1, 1986.

48: Kim92: Jin-Oh Kim, Pradeep K. Khosla, "Real-time obstacle avoidance using harmonic potential functions", IEEE Transactions on Robotics and Automation, 1992.

49: Kim05: Jongwoo Kim, Joel M. Esposito, Vijay Kumar, "An RRT-based algorithm for testing and validating multi-robot controllers", in proceedings of Robotics: Science and Systems (RSS), 2005.

50: Kod87: Daniel E. Koditscek, "Exact robot navigation by means of potential functions: some topological considerations", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), vol. 4, 1987.

51: Koz82: Dexter Kozen, "Results on the propositional μ-calculus", in Proceedings of the 9th Colloquium on Automata, Languages and Programming, pages 348–359, 1982.

52: Kra01: Danica Kragić, Andrew T. Miller, Peter K. Allen, "Real-time tracking meets online grasp planning", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2001.

53: Krm10: O. Krmer, R. Detry, J. Piater, J. Peters, "Adapting preshaped grasping movements using vision descriptors", From Animals to Animats 11, ser. Lecture Notes in Computer Science, Springer Berlin-Heidelberg, 2010.

54: Kuf00: James J. Kuffner, Steven M. LaValle, "RRT-connect: an efficient approach to single-query path planning", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2000.

55: Kuf04: James J. Kuffner, "Effective sampling and distance metrics for 3D rigid body path planning", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2004.

56: Lah10: M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta, "Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2010.

57: Lam00: F. Lamiraux and Jean Paul Laumond, "Flatness and small-time controllability of multibody mobile robots: application to motion planning", IEEE Transactions on Automatic Control, 45(10):1878–1881, 2000.

58: LaV02: Steven M. LaValle, Michael S. Branicky, Stephen R. Lindemann, "On the relationship between classical grid search and probabilistic roadmaps", in proceedings of the Workshop on the Algorithmic Foundations of Robotics, 2002.

59: LaV06: Steven M. LaValle, "Planning algorithms", Cambridge University Press, 2006.

60: Li08: Yi Li, "Real-time motion planning of multiple agents and formations in virtual environments", PhD Thesis, Simon Fraser University, 2008.

61: Lin03: Stephen R. Lindemann, Steven M. LaValle, "Steps towards derandomizing RRTs", in proceedings of IEEE/RS International Conference on Intelligent Robots and Systems (IROS), 2003.

62: Lin04: Stephen R. Lindemann, Steven M. LaValle, "Incrementally reducing dispersion by increasing Voronoi bias in RRTs", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2004.

63: Liu10: Hong Liu, Weiwei Wan, "Adaptive replanning in hard changing environments", in proceedings of IEEE/RS International Conference on Intelligent Robots and Systems (IROS), 2010.

64: Mil03: Andrew T. Miller, Steffen Knoop, Henrik I. Christensen, Peter K. Allen, "Automatic grasp planning using shape primitives", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2003.

65: Mit08: Ian Mitchell, "Dynamic programming algorithms for planning and robotics in continuous domains and the Hamilton-Jacobi equation", tutorial presented at IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2008.

66: Mov12: MoveIt! [Online], available at: http://moveit.ros.org

67: Nie04: D. Nieuwenhuisen, M. H. Overmars, "Useful cycles in probabilistic roadmap graphs", in proceedings of the IEEE Intl. Conference on Robotics and Automation, 2004.

68: Nie12: M. Nieuwenhuisen, J. Stueckler, A. Berner, R. Klein, S. Behnke, "Shape-primitive based object recognition and grasping", in proceedings of the 7th German Conference on Robotics (ROBOTIK), 2012.

69: Nie13: S. Niekum, S. Chitta, A. Barto, B. Marthi, S. Osentoski, "Incremental semantically grounded learning from demonstration", in proceedings of Robotics: Science and Systems (RSS), 2013.

70: Nis99: Carole Nissoux, "Visibilite et methodes probabilistes pour la planification de mouvement en robotique" (PhD thesis), University Paul Sabatier, Toulouse, 1999.

71: Pan12: D. Pangercic, B. Pitzer, M. Tenorth, M. Beetz, "Semantic object maps for robotic housework representation, acquisition, and use", in proceedings of the IEEE Intl. Conference on Intelligent Robots and Systems (IROS), 2012.

72: Phi08: Roland Philippsen, "Dependency tracking for Fast Marching - dynamic replanning for ground vehicles", tutorial presented at IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2008.

73: Pir07: J. N. Pires, "Industrial robots programming: building applications for the factories of the future", Springer, 2007.

74: Piv09: Mikhail Pivtoraiko, Ross A. Knepper, Alonzo Kelly, "Differentially constrained mobile robot motion planning in state lattices", Journal of Field Robotics, vol. 26, no. 3, 2009.

75: Pla07: E. Plaku, M. Y. Vardi, L. E. Kavraki, "Discrete search leading continuous exploration for kinodynamic motion planning", in proceedings of Robotics: Science and Systems (RSS), 2007.

76: Pla10: E. Plaku, G. D. Hager, "Sampling based motion and symbolic action planning with geometric and differential constraints", in proceedings of the IEEE Intl. Conference on Robotics and Automation, 2010.

77: Pom13a: **M. Pomarlan**, "Visibility based planners for kinematically constrained vehicles", in proceedings of the 8th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), 2013.

78: Pom13b: **M. Pomarlan**, Ioan A. Şucan, "Motion planning for manipulators in dynamically changing environments using real-time mapping of free space", in proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), 2013.

79: Pom13c: **M. Pomarlan**, Ioan A. Şucan, "A two-level approach for intricate manipulation planning", in proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), 2013.

80: Pom13d: **M. Pomarlan**, "Mapping kinematic interactions between objects for robot motion planning", in proceedings of the 12th IEEE International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2014.

81: Pom14a: **M. Pomarlan**, "Visibility based planners for path existence queries in temporal logic", accepted for the Advances in Electrical and Computer Engineering Journal (AECE).

82: Pom14b: **M. Pomarlan**, Ioan A. Şucan, "Improving reliability and efficiency of intricate manipulation planning through mapping of grasp feasibility zones", accepted for ICRA 2014.

83: Pnu77: A. Pnueli, "The temporal logic of programs", in proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), 1977.

84: Que09: Avril Quentin, Gouranton Valérie, Bruno Arnaldi, "New trends in collision detection performance", in proceedings of Virtual Reality International Conference (VRIC), 2009.

85: Rei87: John H. Reif, "Complexity of the generalized mover's problem", appearing in ch. 11 of "Planning, geometry and complexity of robot motion", Jacob Scwartz (ed.), Ablex Pub., 1987.

86: Rup92: Jim Rupert, Raimund Seidel, "On the difficulty of triangulating three-dimensional nonconvex polyhedra", Discrete Computational Geometry 7, 1992.

87: Set96: James A. Sethian, "A Fast Marching Level Set method for monotonically advancing fronts", in proceedings of the National Academy of Sciences of the USA, vol. 93, no. 4, 1996.

88: Sah05: M. Saha, J. Latombe, "Finding narrow passages with probabilistic roadmaps: the small step retraction method", in proceedings of the IEEE Intl. Conference on Intelligent Robots and Systems (IROS), 2005.

89: Shk09: Alexander Shkolnik, Russ Tedrake, "Path planning in 1000+ dimensions using a task-space Voronoi bias", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2009.

90: Shk10: Alexander Shkolnik, "Sample-based motion planning in high-dimensional and differentially constrained systems", PhD Thesis, Massachussetts Institute of Technology, 2010.

91: Sim00: Thierry Simeon, Jean-Paul Laumond, and Carole Nissoux, "Visibility based probabilistic roadmaps for motion planning". Journal of Advanced Robotics, 14(6), 2000.

92: Ste94: Stentz, Anthony, "Optimal and efficient path planning for partially-known environments", in proceedings of IEEE International Conference on Robotics and Automation (ICRA), 1994.

93: Sud07: Avneesh Sud, Russell Gayle, Erik Andersen, Stephen Guy, Ming Lin, Dinesh Manocha, "Real-time navigation of independent agents using adaptive roadmaps", ACM Symposium on Virtual Reality Software and Technology, 2007.

94: Suh11: C. Suh, T. T. Um, B. Kim, H. Noh, M. Kim, F. C. Park, "Tangent space RRT: a randomized planning algorithm on constraint manifolds", in proceedings of the IEEE Intl. Conference on Robotics and Automation (ICRA), 2011.

95: Şuc08: Ioan A. Şucan, "Kinodynamic motion planning for high-dimensional physical systems", MSc. Thesis, Rice University, 2008.

96: Şuc10: I. A. Şucan, M. Kalakrishnan, S. Chitta, "Combining planning techniques for manipulation using real time perception", in proceedings of the IEEE Intl. Conference on Robotics and Automation (ICRA), 2010.

97: Şuc11: I. A. Şucan, L. E. Kavraki, "Mobile manipulation: encoding motion planning options using task motion multigraphs", in proceedings of the IEEE Intl. Conference on Robotics and Automation, 2011.

98: Şuc12: I. A. Şucan, M. Moll, L. E. Kavraki, "The Open Motion Planning Library", IEEE Robotics and Automation Magazine, 2012; [online] available at: http://ompl.kavrakilab.org

99: Şuc12b: I. A. Sucan and S. Chitta, "Motion planning with constraints using configuration space approximations", in proceedings of the IEEE International Conference on Robotics Systems (IROS), 2012.

100: Tar75: Robert E. Tarjan, "Efficiency of a good but not linear set union algorithm", ACM, 22(2), 1975.

101: Tas07: Yuval Tassa, Tom Erez, Bill Smart, "Receding horizon differential dynamic programming", in proceedings of Neural Information Processing Systems (NIPS), 2007.

102: Ted09: Russ Tedrake, "LQR-trees: feedback motion planning on sparse randomized trees", in proceedings of Robotics: Science and Systems (RSS), 2009.

103: Ten13: M. Tenorth, S. Profanter, F. Balint-Benczedi, M. Beetz, "Decomposing CAD models of objects of daily use and reasoning about their functional parts", in proceedings of the IEEE/RSJ Intl. Conference on Intelligent Robots and Systems (IROS), 2013.

104: The10: Evangelos Theodorou, Yuval Tassa, Emo Todorov, "Stochastic differential dynamic programming", in proceedings of American Control Conference, 2010.

105: Toi10: Noel E. Du Toit, "Robot motion planning in dynamic, cluttered, and uncertain environments: the partially closed-loop, receding horizon approach", PhD Thesis, California Institute of Technology, 2010.

106: Tsa02: Yen-hsi Richard Tsai, "Rapid and accurate computation of the distance function using grids", Journal of Computational Physics, 2002.

107: Var05: Gokul Varadhan, Dinesh Manocha, "Star-shaped roadmaps - a deterministic sampling approach for complete motion planning", in proceedings of Robotics: Science and Systems (RSS), 2005.

108: Wes04: Matthew West, "Variational integrators", PhD Thesis, California Institute of Technology, 2004.

109: Xue09: Zhixing Xue, Ulrich Stadie, J. Marius Zoellner, Ruediger Dillmann, "An efficient grasp planning system using impulse-based dynamic simulation", in proceedings of ECCOMAS Thematic Conference on Multibody Dynamics, 2009.

110: Yer05: Anna Yershova, Leonard Jaillet, Thierry Simeon, Steven M. LaValle, "Dynamic-domain RRTs: efficient exploration by controlling the sampling domain", in proceedings of IEEE International Conference on Robotics and Automation, 2005.

111: Yer08: Anna Yershova, "Sampling and searching methods for practical motion planning algorithms", PhD Thesis, University of Illinois, 2008.

112: Zha07: L. Zhang, Y. Kim, D. Manocha, "A hybrid approach for complete motion planning", in proceedings of the Intl. Conference on Intelligent Robots and Systems (IROS), 2007.