Chapter 6

Related Work

6.1 Motion Planning

Motion planning is one of the most studied problems in mobile robotics. Latombe [62] gives a thorough overview of early approaches, while Reif [76] establishes the exponential complexity of the general path planning problem. This complexity has inspired many approximation methods, such as local minima free grid-based potentials [56], and common application of the A* algorithm [72] on cost grid representations of the robot's state space. Stentz's D* algorithm [80] builds on A* to create a variant which only recalculates portions of the problem where costs change, achieving significant speedup for domains where the environment changes slowly over time. Much recent work has centered around the idea of randomized sampling for approximation, such as LaValle and Kuffner's RRT algorithm [64,65], and planners based on Kavraki et al.'s Probabilistic Roadmap (PRM) framework [53, 54]. RRT grows random trees in configuration space to solve single-query problems efficiently, and was extended by Bruce [23] to work efficiently in unpredictably changing domains by using continuous replanning with a bias from past plans. PRM separates planning into a learning and query phase. In the learning phase, a random subgraph of the configuration space is build by sampling points and connections between points to find free locations and paths, respectively. In the query phase, this graph can be used with ordinary graph search methods such as A* to solve the path planning problem. It relies on largely static domains to achieve efficiency, since an in an unchanging workspace the learning phase need only be computed once. Boor et al. [9] changed the sampling method of PRM to focus on the boundaries of free space in their Gaussian PRM approach. Amato and Wu [1, 2] create another modified sampling approach called Obstacle PRM, and Hsu [47] created a bridge test to bias sampling to difficult narrow passages. Isto [49] looked at applying complex local planners to PRM, replacing the commonly used "straight-line" method for local planning to connect two points. It was found to significantly improve graph connectivity for difficult problems resulting in more reliable planning.

Our particular domain contains multiple moving objects, which must be treated as obstacles for safe navigation. Edrman and Lozano-Perez [32] look at the effects of multiple moving objects on the planning problem and offer some early solutions by adding time to the configuration space. Latombe [62] provides background and investigates the effects of moving obstacles on the planning problem. Fiorini and Shiller [36] focus on using relative velocity to simplify the planning problem from each agent's point of view, leading to a more tractable problem for mobile robots. Their work was extended to construct explicit velocity obstacles, first for linear paths and then for arbitrary nonlinear paths [37, 61]. The approach assumes that the complete obstacle paths are known in advance. More recently, Hsu [46] applied randomized planning to domains with moving obstacles, and tested the system on physical robots. This approach assumes constant velocity obstacles, but recovers via replanning when a change in velocity is detected.

6.1.1 Scope and Categorization

One could not hope to cover every path planning method that has been developed, so our attention will be instead restricted to several representative approaches. Some are from recent research while others are classical approaches that have proved popular in applications. They can be compared against the desirable attributes useful for control of a mobile robot. We'd like for an approach to be complete, meaning that if a feasible path exists it is always found. However, path planning has been shown to be PSPACE-hard [76], indicating that any complete planner will likely be exponential in the the number of degrees of freedom. This has led to research into approximate algorithms with relaxed notions of completeness. One relaxation is *resolution completeness*, where a planning algorithm with a resolution is complete for a particular input given a sufficient finite resolution. An example of this is a planner using a grid approximation of the environment; with a sufficiently fine grid a solution can be found if one exists. However, for problems with no solution, such planners are typically not capable of indicating that no path exists with a finite resolution. Another form of relaxation is *probabilistic completeness*, where an algorithm has a nonzero probability of finding a path if one exists. Many randomized sampling based planners have this property. Another important property of planners is what notions of optimality a they can capture. Since all any search can cover a finite number of cases, one way of viewing path planners is in how they reduce the continuous domain problem down to a finite graph for searching.

This aids in analyzing what kind of optimality the algorithm can provide. The reduction to a graph can (and usually does) impose limits on most metrics for optimality (such as length optimality). In addition, given a graph, an algorithm may or may not return an optimal result given its graph representation.

While the previous two properties apply to most planning domains, mobile robots benefit from additional properties since they execute the plan. During execution, new information about the environment is obtained, which can force updates the robots model of reachable free space, as well as changing the ultimate goal the robot is trying to reach. Thus for efficiency, we would like planners to have an efficient method for updating the environment or the query. Of course efficiency is a relative measure and not very meaningful without clarification of its definition within this context. One useful measure is to look at the speed relative to the size in a change in the environment or requested goal. A large change in the problem would be expected to take the same time as a new problem, while a small change in the problem should ideally only require a short update. Thus if a planning algorithm has a large speed component that is proportional to the amount of change in the environment since the last plan was calculated, then we can say it allows for efficient environment updates. We can define efficient goal updates similarly; running time should mostly depend on how far the goal has been moved.

The final property we will consider is if the planner is capable of planning for non-holonomic robots or robots with dynamics constraints. A robot is said to be holonomic if the actuated degrees of freedom match the robot's total degrees of freedom; or alternatively phrased, that the robot can begin accelerating along any degree of freedom at any time. Normally a robotic arm is considered holonomic, as well as some mobile robots with special omnidirectional drive systems. Robots with motion constraints are said to be non-holonomic, such a car-like robot, or a robot with two independently driven wheels on a common axis (called a differential-drive robot). Planning for a non-holonomic robot is in general substantially more difficult. Another similar difficulty is dealing with dynamics constraints such as bounded accelerations and velocities. A planner that can deal with both kinematic and dynamics constraints is referred to as a kinodynamic planner [65].

6.1.2 Graph and Grid Methods

One of the most classical approaches for low dimensional planning are grid methods, where the workspace and configuration space are represented with uniform rectangular grids. Common approaches involve using Dijkstra's algorithm or A^* to plan discrete actions on the grid [72]. Using Dijkstra's algorithm filling C_{free} outward from the goal generates a minima-free distance on the grid, called a Navigation Function [56]. Such a function has only one global minimum at the goal point, so the robot can follow the gradient downward to reach the goal. Although A^* can deal with arbitrary edge costs, while Dijkstra's is limited to unit length costs, the latter has a significant speed advantage when implemented on regular grids. Specifically, a queue can be used in place of a priority queue, along with several other minor optimizations. Thus for mobile robot applications with frequent replanning, navigation functions have proved popular. However, A^* can be extended to efficiently propagate edge cost changes, as shown by the D^* algorithm [80] and the related Incremental A^* or "D^* Lite" [57]. These algorithms maintain dependency information so that the solution can be updated when edge costs can change. Starting from vertices bordering edges with updated costs, changes are propagated to only the affected nodes, thus saving the cost of a total replan as would be required with A^*. Both algorithms are guaranteed to return a path with the same cost as A^*, so cost optimality is maintained. Thus D^* and its variants support efficient updates to the environment model without affecting optimality. If backwards planning is used, the initial position can be changed efficiently (which is important for mobile robots), while if forward planning is used, the goal position can be moved efficiently. To a limited extent, the other endpoint can be moved in each case by shifting every obstacle and the opposite endpoint (in effect, changing a goal move into a start location move, and vice-versa). This may or may not be more efficient than replanning from scratch for a given environment and representation. In terms of optimality, A^* and D^* are optimal with respect to edge weight, while a navigation function is optimal with respect to the number of edges in a path only (or alternatively viewing all edges as equal weight). The primary problems of grid methods is that they not scale well with the degrees of freedom, and cannot directly plan non-holonomic actions.

Another problem with grid methods is that they are traditionally limited to motion along the directions of the grids or diagonals (four connected or eight connected grids), resulting in paths which are not as straight as they could be, and thus non-optimal in a continuous sense. In the case of an eight-connected grid, paths can be up to 8% longer than optimal [34]. To address this issue, a variant of D^* called Field D^* has been developed [34, 35]. Field D^* modifies traditional grid based planners by allowing edges to traverse at intermediate angles. First, it modifies the search graph by moving the search vertices to the corners of the cost grid cells. This allows edges connecting from an adjacent cell to a neighboring cell to pass through only one cost region (instead of two when the vertices are at the center of cost cells). While the D^* algorithm will generate path lengths for each vertex, Field D^* also estimates the cost of traversing to any point along the edge of a cell, using interpolation between the two neighboring vertices. This heuristic can fail in certain situations, since the cost variation may not be linear between nodes. As a result Field D^* is not guaranteed to find a path with a cost at least as low as D^*. In most problem instances the heuristic works well however, and results in significantly straighter paths which are approximately 4% shorter than D^* or A^*, while less than doubling the planning time [34]. Thus it works quite well in practice however, and has been applied to many robotic domains to replace classical grid-based planners [35].

6.1.3 Visibility Graph

Due to the high space requirements for grids with more than a couple degrees of freedom, much of the modern work in path planning has tried to use randomization to create "summary graphs" of a workspace using randomized sampling techniques. These summaries are referred to as *roadmaps* [62]. One well known 2D roadmap method is the visibility graph method [69], which noted that a point robot following an optimal trajectory in a field of convex obstacles is always either: (1) following a boundary of an object or (2) following a tangent between two objects (the initial and goal configurations are treated as objects with radius zero). Thus a finite graph could be created that still contained the optimal (minimum length) path in from continuous space. Unfortunately, this method does not generalize to higher dimensions, nor does it scale well with the number of obstacles. It also cannot optimize other metrics, such as those that encourage safety margins around obstacles. In fact, without postprocessing, the plan will skim every obstacle along the path. Thus this simple, optimal method works very well, but only for a very limited environment. It cannot be extended for additional capabilities and thus is the algorithmic equivalent of a local maximum.

6.1.4 Randomized Path Planner (RPP)

Due to the limitations of the classical approaches such as grids and exact roadmap methods, alternative approaches were explored. Randomization has proved a powerful tool in this pursuit. One of the first randomized planners was RPP (Randomized Path Planner) [62]. It constructed a navigation function in the workspace, which may however contain local minima in the configuration space. The planner proceeds by following the navigation function until it reaches a local minimum, and then executes random motions in an attempt to escape the attraction well of the minimum. It records each minimum so it can determine if a random motion escaped the attraction well. A list of minima along the current path is maintained so that the search can backtrack to a random configuration if the last minima cannot be escaped after several iterations. RPP is capable of solving difficult problems, but by relying on both grids and randomization it is only probabilistically resolution complete. It does not guarantee any form of optimality if a local minima is reached during the search, and does not offer any efficient environment or query updates. It scales well with the degrees of freedom of the robot, but not with workspace dimensions. In its plain form it does not deal with holonomic or dynamic constraints.

6.1.5 Rapidly Exploring Random Trees (RRT)

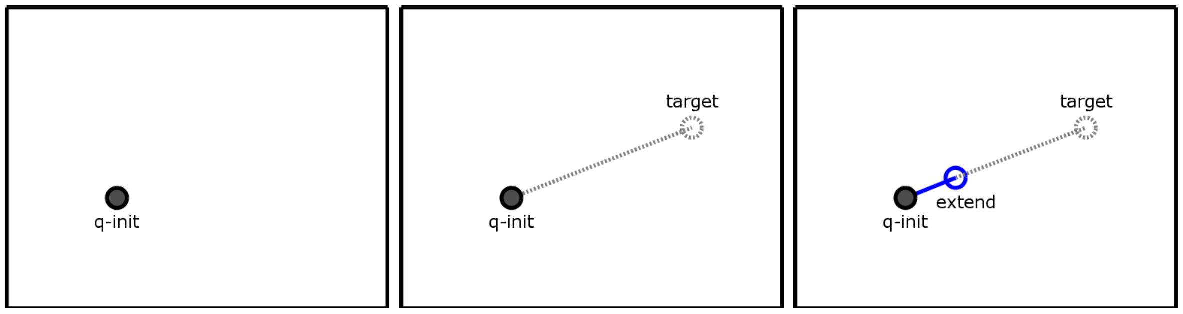

Figure 6.1: Algorithm steps in RRT

One of the relatively recently developed randomized planning approaches are those based on Rapidly-exploring random trees (RRTs) [64]. RRTs employ randomization to explore large state spaces efficiently, and can form the basis for a probabilistically complete though non-optimal kinodynamic path planner [65]. Their strengths are that they can efficiently find plans in high dimensional spaces because they avoid the state explosion that discretization faces. Furthermore, due to their incremental nature, they can maintain complicated kinematic constraints if necessary. A basic planning algorithm using RRTs is as follows: Start with a trivial tree consisting only of the initial configuration. Then iterate: With probability p, find the nearest point in the current tree and extend it toward the goal g. Extending means adding a new point to the tree that extends from a point in the tree toward g while maintaining whatever kinematic constraints exist. In the other branch, with probability $1 - p$, pick a point x uniformly from the configuration space, find the nearest point in the current tree, and extend it toward x. Thus the tree is built up with a combination of random exploration and biased motion towards the goal configuration. Search efficiency can be improved by using bidirectional search growing a tree both from the initial and goal configurations [51]. RRT planners do not support efficient environment or query updates. However they are fast enough that realtime rates can be achieved for relatively simple problems, and with additional extensions quite reasonable performance can be achieved.

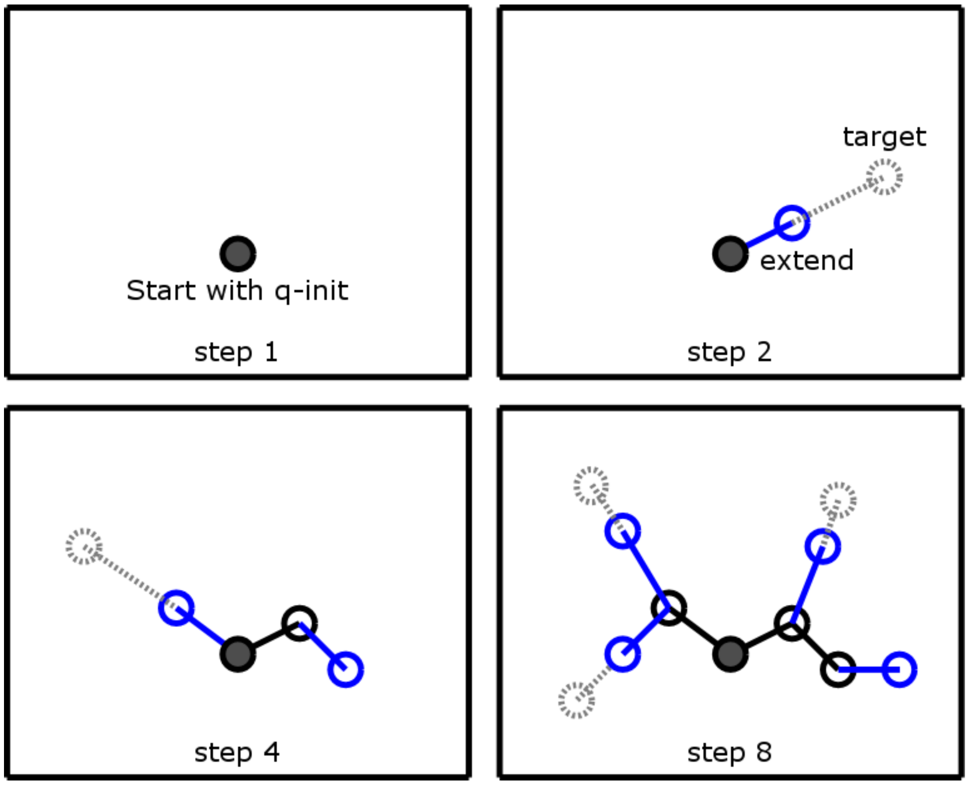

Figure 6.2: Example Growth of RRT

6.1.6 RRT Variants

Two variants of RRT exist which attempt to tackle the replanning problem. The first, called Reconfigurable Random Forest (RRF) [66], extends RRT-Connect by allowing a forest of random trees instead of just the two rooted at the initial and goal configurations. For an initial plan, RRT-Connect is executed as normal, but for replans, search starts with the existing trees. First, all the vertices and edges which may be affected by a change in the environment are checked for collisions, and those that are no longer in C_{free} are removed. Removing edges can "orphan" parts of the search tree, and those subsets are considered as new trees and added to a list of trees to use during search. RRT-Connect proceeds as normal, extending the initial and goal trees, but the Connect operation is replaces with a Merge-Tree operation. The Merge-Tree operation attempts to connect a newly added node to all other trees on the search list, and if the connection succeeds those two trees are re-parented into a single tree. Search continues until some maximum number of iterations, or the initial and goal trees have been connected. Because RRF continues adding nodes with each new query, eventually the search tree may become too large. Thus RRF introduces a pruning

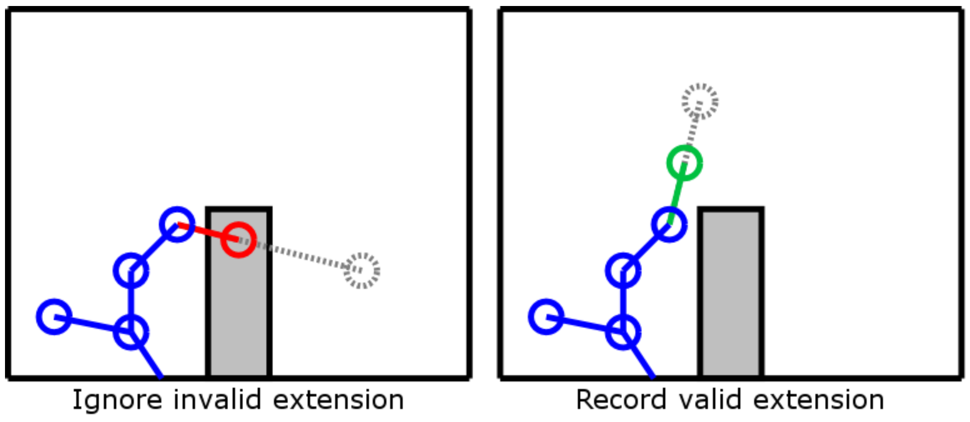

Figure 6.3: RRT extensions with obstacles

operation to decrease the number of nodes to a more reasonable number while maintaining good coverage. While RRF offers good coverage of domains, it suffers from two drawbacks. The first problem is that its runtime tends to by cyclic, increasing as nodes are added, rising dramatically when pruning occurs, and the dropping to a lower level. This could be addressed with continuous online pruning, but at the expense of even more algorithm complexity. The second problem is that despite good coverage, the tree representation means there is always a unique path between any two vertices in the tree. In configuration spaces with loops, this can result in highly non-optimal paths being returned. RRT, by growing a new tree outward from the initial and goal points each query, tends to avoid this problem.

The second extension of RRT for replanning is Dynamic Rapidly-Exploring Random Trees (DRRT) [33]. This work can be seen as a more conservative version of RRF which aims to be a continuous planner analogous to D^* on grids. For an initial query, DRRT grows a tree backward from the goal configuration using the standard RRT algorithm until the initial configuration is reached. To handle an environment update, all possibly affected vertices and edges are checked for collisions and removed if they are no longer in C_{free}. Unlike RRF however, DRRT removes the subtree of any such node, thus preventing orphaned trees from being created. Replanning queries proceed from the existing tree, and thus can be quite efficient when updates occur near the initial configuration. DRRT works best when the sample distribution for search is biased toward areas where obstacles have been updated. The drawbacks of DRRT are that it does not support moving the goal configuration, that obstacle changes near the goal can invalidate large parts of the tree, and that for efficiency it needs to be notified explicitly of which areas of the environment have changed and which have remained static.

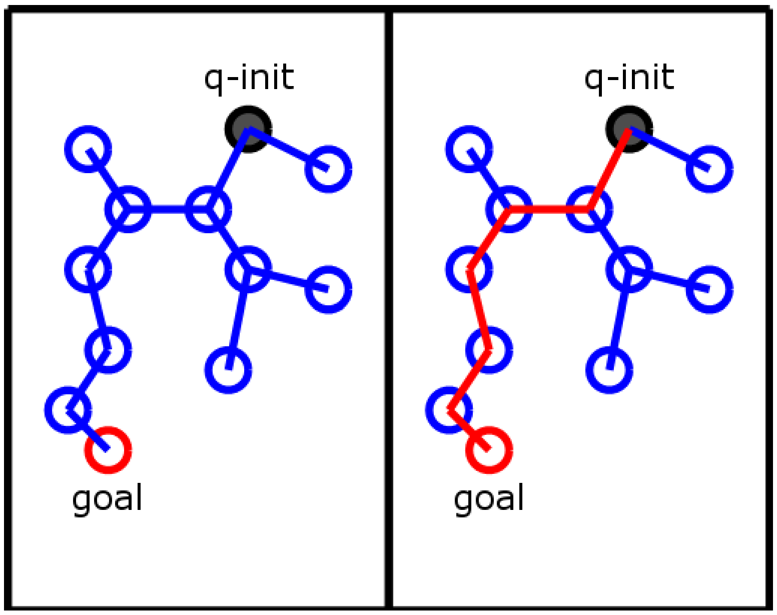

Figure 6.4: RRT as a motion planner

6.1.7 Probabalistic Roadmaps (PRM)

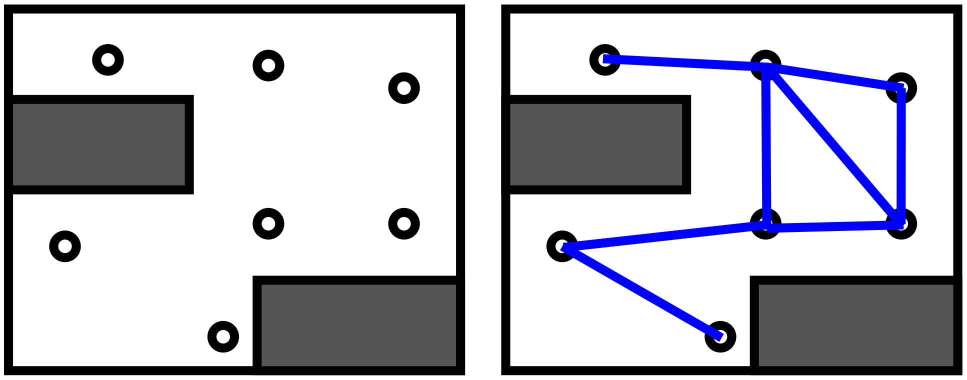

Figure 6.5: Uniformly Sampled PRM Example

Finally, a widely researched planner for static workspaces with higher dimensionality are those based on the concept of Probabilistic Roadmaps, or PRMs [53, 54]. PRMs have two distinct planning phases; The learning phase where a roadmap is built from the configuration space, and the query phase where it is used to solve a particular problem. The learning phase needs to be run whenever the environment changes, to create the summary that the planning stages use. First, free configurations are found through some sampling distribution (such as uniform). Then, connections are made among these configurations using a "local planner'. A simple straight line planner is often used, which checks if one configuration could be linearly interpolated to another configuration without hitting an obstacle. In figure 6.5 a workspace is shown first after sampling for free configurations using uniform sampling, and then after connections have been made to create the roadmap. The connections in this example are attempted with a simple straight line planner. At the end of the learning phase, we have a graph where the free configurations are the vertices, and the successful local plans are the edges. If the local plan weights are recorded as weights in the graph, we can apply A^* or any other graph searching method to get from one configuration in the graph to any other configuration in the same connected graph component. In the query phase, the initial and goal position are added to the graph, by trying to connect them to nearby vertices using the local planner, then the graph search algorithm is used to find the actual path. The planner fails if it cannot connect the initial or goal configurations to the graph, or if the graph sections they connect to are not connected by the roadmap. Taken together, these two phases comprise the basic PRM planning algorithm. It is probabilistically complete, and when using A^* can offer optimal plans (with respect to the roadmap, which is an finite approximation of the paths in free space). It does not allow efficient environment updates, because the learning stage and query stage must both be recomputed (i.e. it starts from scratch). Query updates however tend to be very efficient, because only the graph search is required and PRM graphs can be relatively small when compared to grids. In addition, if the graph is large enough that speed becomes a problem, D^* could be used since it applies to general weighted graphs [80]. PRM has good scalability with degrees of freedom thanks to its random sampling techniques. The plain version of PRM does not handle non-holonomic constraints. It can be extended to solve some non-holonomic problems using specialized local planners that satisfy the constraints [81].

6.1.8 PRM Variants

The PRM approach is a flexible one, allowing modification of various components while maintaining its basic capabilities. This has led to a large number of variants which try to address limitations or improve efficiency. The three major variables are the sampling strategy, the roadmap construction method, and the local planner used. For sampling, it has been found that a uniform distribution generally does not perform as well as modified distributions that are a more complicated function of free space. Obstacle PRM [1, 2] is a variant which generates samples on the boundary of free space. It first generates samples on the surface of one obstacle, and uses rejection sampling to remove those samples that are inside other obstacles. The bridge test [47] sampling method is a rejection sampling method that accepts free configurations that lie on the midpoint of two non-free configurations. The length of the bridge is varied with a Gaussian distribution. Finding these bridges is very time consuming due to the higher fraction of candidate samples being rejected. However when combined with a small uniform distribution to fill free space, relatively few nodes are needed to describe fairly complicated environments with narrow passages. Thus it is very efficient for connecting the roadmap configurations and in the query phase.

The next kind of modification of PRM is to change the roadmap construction approach, such as which sampled configurations we try to connect, and when we check them. Connected components analysis is used in some PRM variants, for example, to speed roadmap construction by only trying to connect configurations from different components. While a significant speedup, this also means no redundant paths are found, thus the resulting roadmap is a tree. These are most useful for problems where path length optimality is not an issue. Another optimization trades slower queries to gain a faster learning phase. In Lazy PRM [7,8], collision checks are not done while constructing the roadmap; instead of testing if an edge can be added it is added as a "potentially free" local path. During the query, the graph is searched to find the shortest potentially free path. Then the path is checked and any edges that are not free are removed. If no edges were removed, the path is a valid solution, while if edges are removed the query is repeated with the pruned graph. Eventually a free path will be found if it exists in the graph, and only a minimal number of edges will have to be checked for collisions.

The third kind of modification of PRM is to replace the local planner, which tries to connect two configurations when building the roadmap or connecting the query configurations to the roadmap. For non-holonomic problems for which a local planner exists (which can fail in some circumstances, but must work when no obstacles are present), PRM can use that local planner to become a non-holonomic global planner. Such a local planner has been derived for car-like robots in [81]. Isto showed in [49] that increasing the power of the local planner improves the overall performance of PRM on most benchmarks. Specifically, allowing the local planner to slide along obstacles after contact, or using discretized heuristic planners improved roadmap compactness and connectedness. Finally, some planners combine powerful local planners with advanced construction methods. Probabilistic roadmaps of trees (PRT) [6] is a PRM planner that uses RRT as the local planner to connect a relatively small number of configurations. It has been shown to work for high dimensional problems of at least 18 degrees of freedom. A related approach is the previously mentioned Reconfigurable RRT Forest [67], where a tree is maintained by using RRT-Connect to connect a forest and merging the trees together by re-parenting. It can be thought of as a PRM variant instead of an RRT derivative in many respects. Unfortunately the tree representation of the roadmap means that only one path can ever be found in the graph search, and it will generally be far from optimal.

6.1.9 Randomized Forward Planners

In contrast to the RRT and PRM approach of "pulling" a search graph by choosing random samples and then trying to connect a path to those points, some planners "push" samples by first choosing some vertex to expand, and then extending it using a random action. These algorithms can be thought of as modern descendents of the RPP algorithm. One of the first descendents was Hsu's Expansive Configuration Space Planner (ECSP) [48]. ECSP initializes its search with the initial and goal configuration as roots of two search trees. It then executes and *Expand* operation on each tree, where a vertex v is chosen at random with probability $1/w(v)$. The weight $w(v)$ is derived from a local density estimate of the search tree. The chosen configuration v has a number of random points v' sampled around it, which are kept with probability $1/w(v')$ if a free direct path between v and v' exists. After each tree has been expanded, a *Connect* operation is employed which tries to link all pairs of vertices in each tree if they are below some distance threshold. If the link is a free path, the trees have been connected and search can terminate. Otherwise, search continues until some time limit has been reached. ECSP has not found broad application due to efficiency that is generally less than PRM or RRT, but it inspired much additional research.

The Guided Expansive Space Tree planner [73] extended ECSP through the addition of a goal distance heuristic to bias search, much in the way A^* uses a heuristic to limit search on finite graphs. It also made use of the forward planning possible with ECSP to sample actions, thus allowing highly non-holonomic problems to be solved. The PDST-Explore [59] planner takes this further, planning exclusively in action space using forward simulation. Local density estimates are calculated in a coarse way using KD-trees, and nodes are chosen for expansion deterministically, based on the volume of the KD-tree cell containing the node, and the number of times the node has been chosen. However, instead of expanding that specific node, PDST-Explore samples a point randomly along the continuous path from the initial configuration to that node's configuration, which it then extends using a random action. As a result, PDST-Explore can converge to uniform coverage of control space, and thus is probabilistically complete even for non-holonomic problems. It has been applied successfully to non-holonomic problems with drift and under-actuation [60]. However, it does not currently have a variant tailored for replanning.

6.2 Safety Methods

6.2.1 Dynamic Window

Though not a path planner in the same sense as the other algorithms, the dynamic window approach [38] is a search method for controlling mobile robots in light of both kinematic and dynamics constraints. It is a local method, in that only the next velocity command is determined, however it can incorporate non-holonomic constraints, limited accelerations, and the presence of obstacles into that determination, guaranteeing safe motion. The search space is the velocities of the robot's actuated degrees of freedom. The two developed cases are for synchro-drive robots with a linear velocity and an angular velocity, and for holonomic robots with two linear velocities [12, 38]. A grid is created for this velocity space, reflecting an evaluation of velocities falling in each cell. First, the obstacles of the environment are considered, by assuming the robot travels at a cell's velocity for one control cycle and then attempts to brake at maximum deceleration while following that same trajectory. If the robot cannot come to a stop before hitting an obstacle along that trajectory, the cell is given an evaluation of zero. Next, due to limited accelerations, velocities are limited to a small window that can be reached within the acceleration limits over the next control cycle (for a holonomic robot this is a rectangle around the current velocities). Finally, the remaining velocities are scored using a heuristic distance to the goal. Like all local methods, the dynamic window approach is incomplete, but it demonstrated practical applicability on real robots moving at relatively high speeds. In addition, when combined with a navigation function for mid-level planning, a non-optimal but resolution complete planner was developed and tested at speeds up to 1m/s in cluttered office environments with dynamically placed obstacles [12].

6.3 Algorithm Summary

Taken together, these algorithms and approaches solve many variants of the path planning problem. Many have limitations when viewed A summary of all related algorithms mentioned in this section is shown in Table 6.1.

Table 6.1: Comparison of related planning algorithms

Approach	Complete	Optimal	Efficient Environ. Updates	Efficient Query Updates	Good dof Scalability	Non-Holonomic
Grid A^*	res	grid	no	no	no	no
Grid D^*	res	grid	yes	yes	no	no
Field D^*	res	no	local	part	no	no
Nav Func	res	grid	no	no	no	no
RPP	prob+res	no	no	no	no	yes
RRT	prob	no	no	no	yes	yes
RRF	prob	no	yes	yes	yes	no
DRRT	prob	no	local	part	yes	no
PRM	prob,res	graph	no	yes	yes	some
Lazy PRM	prob	graph	yes	yes	yes	no
PRT	prob	no	no	yes	yes	no
ECSP	unkn	no	no	no	maybe	no
Guided ECSP	unkn	no	no	no	yes	yes
PDST-Explore	prob	no	no	no	yes	yes
Dynamic Win	no	res	no	no	no	yes

Key	Term	Meaning
res	resolution complete	path is found if the resolution is sufficiently small
prob	prob. complete	path is found with nonzero probability
unkn	unknown completeness	depends on unproven conjecture
grid	grid optimal	shortest path on grid is always found
graph	graph optimal	shortest path in roadmap is always found
local	local environ. updates	efficient updates supported near the initial configuration
part	partial query updates	initial configuration can change, but not goal
some	some variants	specialized variants support non-holonomic planning