Chapter 4

Motion Planning

4.1 Approach

In our domains (see 2), each environment is a mix of static and dynamic obstacles:

- The **RoboCup small-size** domain contains up to 10 agents moving at a given time, with five agents not controlled by our system. The teams in the environment are operating adversarial with conflicting goals.
- The **fixed-wing UAV** domain contains a static terrain paired with a variable number of geometric obstacles reflecting constraints that can change per-query.
- The **humanoid** domain contains an occupancy grid model which is being continuously updated, and thus it nearly always changes between queries. Much of the grid is highly correlated across queries however.

Thus, in each case unmodelled or unpredictable elements are modifying the environment. Dynamic elements, such as the trajectory of opponent robots, are not accurately predictable beyond a short time in the future. The result is that we cannot apply system which assumes a known future trajectory for moving obstacles, but instead one that allows new decisions to be made each control cycle. In addition, we assume no notification of which objects have moved or how their geometry has been altered. While in some domains, this information may be available, other domains may not be able to recover this information efficiently. The factors of the dynamic and unpredictable environment led to the choice of an iterative approach to replanning, were replanning is unconditional as new state information arises each cycle.

In addition, the geometry and dimensionality of the obstacles varies between domains. If we desire a single planning framework that works for all of the domains, this limits our choices of applicable planners. Of the available options (see 6), we have chosen randomized approaches to motion planning, which require no explicit model of their environment. Instead, the configuration space is discovered experimentally through queries to a collision detection system.

The two methods we have chosen to extend are RRT and PRM. We have developed a single planning framework which supports a derivative of RRT, called ERRT, as well as an incremental variant of PRM. As a result of the combination as a single parameterized method, we have removed some of the unnecessary distinctions between the algorithms. The resulting parameterized method can be tuned to act as the method most suitable for a given problem domain.

4.1.1 Contributions of this Chapter

- The waypoint cache, and its associated point distribution, are novel. It is introduced as part of the author's Execution Extended RRT (ERRT) algorithm, which extends the RRT algorithm of LaValle and Kuffner [65]
- The ERRT algorithm is tested on a physical robot platforms for a the RoboCup small-size team. As far as the author is aware, this is the first application of an RRT-derived planner to a physical robot platform (fall 2001), and the longest-running use at 5 years (It has been used in all subsequent RoboCup experiments and applications.)
- Bidirectional Multi-Bridge ERRT introduces novel parameters to the Kuffner's RRT-Connect algorithm [51]. These new parameters allow a tradeoff between planner efficiency and plan length optimality.
- The novel Dynamic PRM algorithm is introduced, which extends Kavraki et al.'s Probabilistic Roadmap (PRM) algorithm [54]. It is tested on the QRIO humanoid robot.
- This chapter identifies the requirements for robust motion planning for robotics applications, and in particular robustness to location error. The author is not aware of these requirements or their solutions being identified in existing work on randomized or graph-based motion planners.
- The efficient heuristic of an *active goal* is introduced to solve kinematically constrained planning problems where the goal is defined as a fixed-radius orbit around a point. It is demonstrated as part of a planner for fixed-wing unmanned

aerial vehicles (UAVs).

4.2 Existing Planning Methods

4.2.1 RRT Algorithm in Detail

In essence, an RRT planner searches for a path from an initial state to a goal state by expanding a search tree. For its search, it requires the following three domain-specific function primitives:

function *Extend* (env:Environment, current:State, target:State):State
function *Distance* (current:State, target:State):Real
function *RandomState* ():State

First, the *Extend* function calculates a new state that can be reached from the target state by some incremental distance (usually a constant distance or time), which in general makes progress toward the goal. If a collision with an obstacle in the environment would occur by moving to that new state, then a default value, EmptyState, of type state is returned to capture the fact that there is no "successor" state due to the obstacle. In general, any heuristic methods suitable for control of the robot can be used here, provided there is a reasonably accurate model of the results of performing its actions. The heuristic does not need to be very complicated, and does not even need to avoid obstacles (just detect when a state would hit them). However, the better the heuristic, the fewer nodes the planner will need to expand on average, since it will not need to rely as much on random exploration. Next, the function *Distance* needs to provide an estimate of the time or distance (or any other objective that the algorithm is trying to minimize) that estimates how long repeated application of *Extend* would take to reach the goal. Finally, *RandomState* returns a state drawn uniformly from the state space of the environment.

For a simple example, a holonomic point robot with no acceleration constraints can implement *Extend* simply as a step along the line from the current state to the target, and *Distance* as the Euclidean distance between the two states. Table 4.1 shows the complete basic RRT planner with its stochastic decision between the search options:

- with probability p, it expands towards the goal minimizing the objective function *Distance*,

- with probability $1 - p$, it does random exploration by generating a *RandomState*.

The function *Nearest* uses the distance function implemented for the domain to find the nearest point in the tree to some target point outside of it. *ChooseTarget* chooses the goal part of the time as a directed search, and otherwise chooses a target taken uniformly from the domain as an exploration step. Finally, the main planning procedure uses these functions to iteratively pick a stochastic target and grow the nearest part of the tree towards that target. The algorithm terminates when a threshold distance to the goal has been reached, though it is also common to limit the total number of nodes that can be expanded to bound execution time.

4.2.2 PRM Algorithm in Detail

Probabilistic Roadmap (PRM) planners take a related but alternative approach to planning compared to RRT. The main motivation for using PRM is that it allows planing to be split into two phases, the first of which can be reused in a static environment. The two stages are the following:

Learning Phase where a finite graph model, or roadmap, is constructed up to approximate the geometry of the free configuration space.

Query Phase where the roadmap is extended with a particular problem instance and searched for a free path.

An implementation of this variant of PRM can be found in 4.2. The learning phase proceeds by adding points randomly distributes within the environment, and then attempting to find pairs of nodes which can be reached with a local planner. If the local planner succeeds, an edge is added to the roadmap graph. This is implemented in the function *PRMLearn*, which first calls the function *AddStates* to sample random states from the environment. *AddStates* only adds states which are in C_{free}. Next, *PRMLearn* calls *ConnectStates*, which attempts to connect each state with its neighbors using the function *Connect*.

The function *Connect* is in a sense the core of the PRM planner, much like *Extend* for RRT. This implementation find finds all other states within a certain maximum distance (line 1), and attempts a straight-line connection with those neighboring states (line 3-4). Many variants of *Connect* exist in PRM literature, in particular by varying the neighbors

```
function RRTPlan(env:Environment, initial:State, goal:State) : RRTTree
    var nearest,extended,target : State
    var tree : RRTTree
    nearest ← initial
    tree ← initial
    while Distance(nearest,goal) < threshold do
        target ← ChooseTarget(goal)
        nearest ← Nearest(tree,target)
        extended ← Extend(env,nearest,target)
        if extended ≠ EmptyState
            then AddNode(tree,extended)
    return tree

function ChooseTarget(goal:State) : State
    var p : $\mathbb{R}$
    p ← UniformRandom(0,1)
    if p $\in [0, \text{GoalProb}]$
        then return goal
    else if p $\in [\text{GoalProb}, 1]$
        then return RandomState()

function Nearest(tree:RRTTree,target:State) : State
    var nearest : State;
    nearest ← EmptyState;
    foreach State s ∈ tree do
        if Distance(s,target) < Distance(nearest,target)
            then nearest ← s;
    return nearest;
```

Table 4.1: The basic RRT planner stochastically expands its search tree to the goal or to a random state.

chosen or the local planner which is used [6, 49]. The implementation is the earliest variant as described in Kavraki et al. [53, 54].

The query phase for PRM is shown in the function *PRMQuery*. It begins by adding the initial and goal states to the roadmap, and then proceeds with a graph search to find a path on this augmented roadmap. The function *GraphSearch* can be implemented using A* search in order to find the shortest length path of those that exist in the roadmap graph.

One aspect of note for the two-phase PRM is that its completeness depends on a parameter n reflecting the number of samples with which to model the environment. This means the planner may fail in the query phase, and does not attempt any further search. Early work in PRM recognized this issue, and for cases where one wishes to continue planning until a solution is found, a one-shot version of PRM was developed. In Table 4.3, a one-shot PRM variant which continues expanding the search is shown. It simple alternates a learning phase with a search phase, increasing the number of samples added at each stage by a factor *ExpandFactor* which is greater than one. The sub-functions are all reused from the two-phase PRM planner.

4.3 Execution Extended RRT (ERRT)

Some optimizations over the basic described in existing work are bidirectional search to speed planning, and encoding the tree's points in an efficient spatial data structure [4]. In this work, a KD-tree was used to speed nearest neighbor lookup, but bidirectional search was not used because it decreases the generality of the goal state specification (it must then be a specific state, and not a region of states). Additional possible optimizations include a more general biased distribution, which was explored in this work in the form of a waypoint cache. If a plan was found in a previous iteration, it is likely to yield insights into how a plan might be found at a later time when planning again; The world has changed but usually not by much, so the history from previous plans can be a guide. The waypoint cache was implemented by keeping a constant size array of states, and whenever a plan was found, all the states in the plan were placed into the cache with random replacement. This stores the knowledge of where a plan might again be found in the near future. To take advantage of this for planning, Table 4.4 shows the modifications to the function *ChooseTarget*.

With ERRT, there are now three probabilities in the distribution of target states. With probability $P[goal]$, the goal is chosen as the target; With probability $P[waypoint]$, a random waypoint is chosen, and with the remaining probability a uniform state is chosen as before. Typical values used in this work were $P[goal] = 0.1$ and $P[waypoint] = 0.6$. Another

```
type StateSet = set of State
type EdgeSet = set of Edge
tuple RoadMap = (StateSet * EdgeSet)

function AddStates(env:Environment, V:StateSet, n:$\mathbb{Z}$) : StateSet
    for $i = 1$ to $n$
        $q \leftarrow$ RandomState();
        if CheckObs(env,$q$)
            then $V \leftarrow V + s$
    return $V$

function Connect(env:Environment, $(V, E)$:RoadMap, q:State) : EdgeSet
    $L$ = NearbyStates($V$,$q$,MaxDist)
    foreach $s \in L$ do
        if CheckObsLine(env,$q$,$s$)
            then $E \leftarrow E + (q, s)$
    return $E$

function ConnectStates(env:Environment, G:RoadMap) : EdgeSet
    foreach $q \in V$ do
        $E \leftarrow$ Connect(env,$G$,$q$)
    return $E$

function PRMLearn(env:Environment, num:$\mathbb{Z}$) : RoadMap
    var $(V, E)$ : RoadMap
    $V \leftarrow$ AddStates(env,$\emptyset$,num)
    $E \leftarrow$ ConnectStates(env,(V,$\emptyset$))
    return $(V, E)$

function PRMQuery(env:Environment, initial:State, goal:State) : Plan
    $V \leftarrow V$ + {initial,goal}
    $E \leftarrow$ Connect(env,$(V, E)$,initial)
    $E \leftarrow$ Connect(env,$(V, E)$,goal)
    $P \leftarrow$ GraphSearch((V,E),initial,goal)
```

Table 4.2: A basic implementation of a PRM planner

```
function PRMOneShot(env:Environment, initial:State, goal:State) : Plan
    var $(V, E)$ : RoadMap
    var $P$ : Path
    $V \leftarrow$ {initial,goal}
    $E \leftarrow \emptyset$
    repeat
        let $n = Max(Size(V) \cdot ExpandFactor, MinNumAddStates)$
        $V \leftarrow AddStates$(env,$V$,$n$)
        $E \leftarrow ConnectStates$(env,$(V, E)$)
        $P \leftarrow GraphSearch$($(V, E)$,initial,goal)
    until $P \neq \emptyset$
    return $P$
```

Table 4.3: The code for a one-shot PRM planner

```
function ChooseTarget(goal:State) : State
    let p = UniformRandom(0,1)
    let i = RandomInt(0,NumWaypoints-1)
    if p $\in$ [0, GoalProb] then
        return goal
    else if p $\in$ [GoalProb, GoalProb + WaypointProb] then
        return WaypointCache[i]
    else if p $\in$ [GoalProb + WaypointProb, 1] then
        return RandomState()
```

Table 4.4: The extended RRT planner chooses stochastically between expanding its search tree to the goal, to a random state, or to a waypoint cache.

extension was adaptive beta search, where the planner adaptively modified a parameter to help it find shorted paths. A simple RRT planner is building a greedy approximation to a minimum spanning tree, and does not care about the path lengths from the initial state (the root node in the tree). The distance metric can be modified to include not only the distance from the tree to a target state, but also the distance from the root of the tree, multiplied by some gain value. A higher value of this gain value (beta) results in shorter paths from the root to the leaves, but also decreases the amount of exploration of the state space, biasing it to near the initial state in a "bushy" tree. A value of 1 for beta will always extend from the root node for any Euclidean metric in a continuous domain, while a value of 0 is equivalent to the original algorithm. The best value seems to vary with domain and even problem instance, and appears to be a steep tradeoff between finding an shorter plan and not finding one at all. However, with biased replanning, an adaptive mechanism can be used instead that seems to work quite well. When the planner starts, beta is set to 0. Then on successive replans, if the previous run found a plan, beta is incremented, and decremented otherwise. In addition the value is clipped to between 0 and 0.65. This adaptive bias schedule reflects the idea that a bad plan is better than no plan initially, and once a plan is in the cache and search is biased toward the waypoints, nudges the system to try to shorten the plan helping to improve it over successive runs.

4.3.1 Testing ERRT for RoboCup

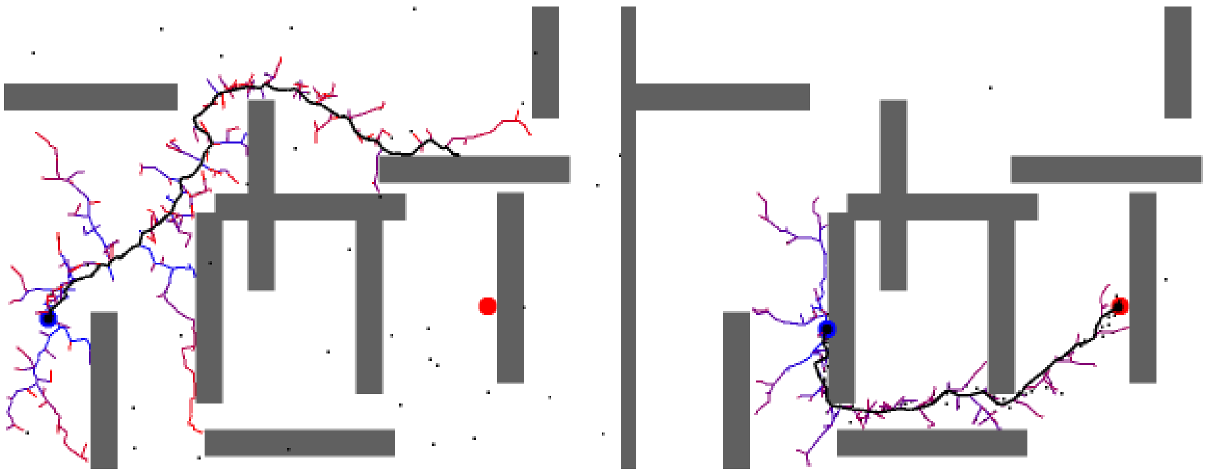

Figure 4.1: An example from the simulation-based RRT planner, shown at two times during a run. The tree grows from the initial state. The best plan is shown in bold, and cached waypoints are shown as small black dots.

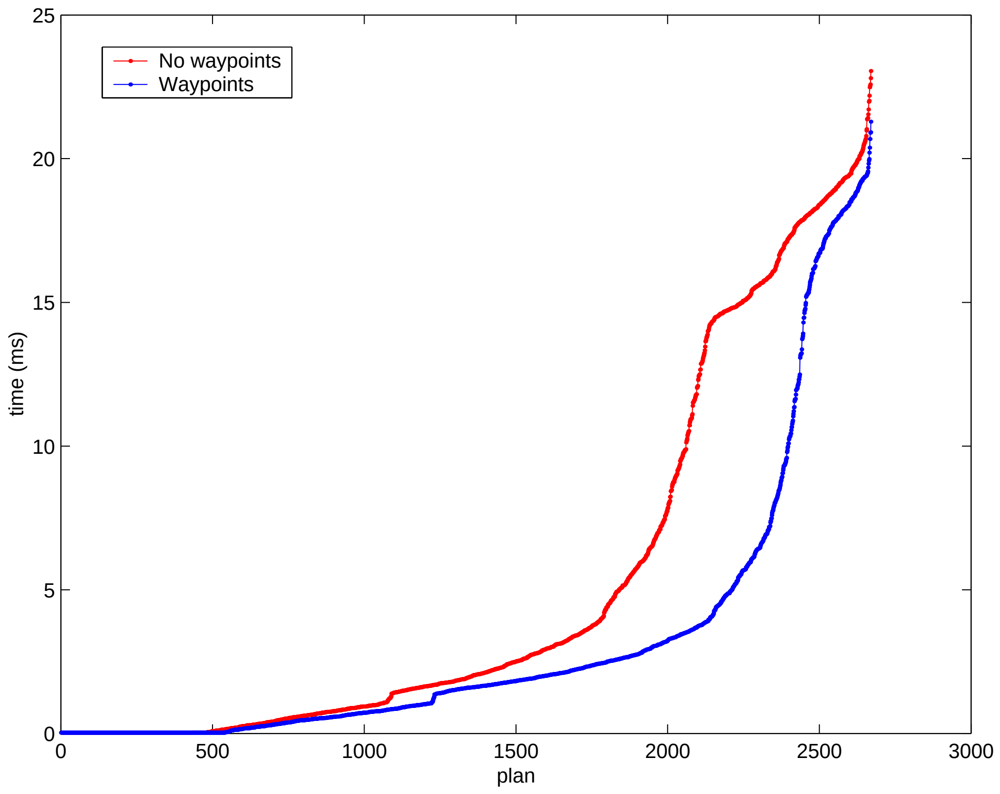

Figure 4.2: The effect of waypoints. The lines shows the sorted planning times for 2670 plans with and without waypoints, sorted by time to show the cumulative performance distribution. The red line shows performance without waypoints, while the blue line shows performance with waypoints (Waypoints=50, p[Waypoint]=0.4).

For a RoboCup like domain with 10 other obstacles, we found that expanding a constant number of nodes (N=500) and 50 waypoints worked fairly well, and would normally run in 20ms or less. The step size of the extensions was 8cm (the world is 320cm by 240cm). The goal was used as a target with $P[goal] = 0.1$ and the waypoints were used with $P[waypoint] = 0.4$. The adaptive beta parameter was turned off in the final version, since waypoints appear to achieve most of the benefits of beta search without increasing the search time, although in future work we'd like to explore the relationship between the two. The waypoints qualitatively seemed to help get the agent out of difficult situations, since once it found a valid plan to take it out of some local minima or other oscillation point, it would be highly likely to find a similar plan again since the waypoints had such a high probability of being chosen as targets. This effect of waypoints on performance was examined in an experiment, and the results are shown in in Figure 4.2. Since the curves diverge at moderate difficulty, it appears that waypoints help speed planning by offering "hints" from previous solutions. When the problem becomes impossible or nearly impossible, neither performs well. This is what one would expect, because waypoints are only available when a solution was found in a previous iteration. Overall, he simulation appeared successful enough that we set out to adapt the system for a real robot, and to employ a KD-tree to speed up the nearest neighbor lookup step to further improve efficiency.

For RoboCup F180 robot control, the system for must take the input from a vision system, reporting the position of all field objects detected from a fixed overhead camera, and send the output to a radio server which sends velocity commands to the robots that are being controlled. The path planning problem here is mainly to navigate quickly among other robots, while they also more around executing their own behaviors. As a simplification for data gathering, we first examined the more simple problem of running from one end of the field to the other, with static robots acting as obstacles in the field. In filling in the domain dependent metrics, we first tried metrics that maintained continuous positional and angular velocity, continuous positional velocity only, and fixed curvature only. None of these worked well, substantially increasing planning times to unacceptable levels or failing to find it at all within a reasonable number of node expansions (N=2000). All metrics were written in terms of time and timesteps so the planner would tend to optimize the time length of plans. Although this was a substantial setback, we could still fall back on the obviously physically incorrect model of no kinematic constraints whatsoever and fixed time step sizes, which had been shown to work in simulation. The extension metric then became a model of a simple heuristic "goto-point" that had already been implemented for the robot.

The motivation for this heuristic approach, and perhaps an important lesson is the following: (1) in real-time domain, executing a bad plan immediately is often better than sitting still looking for a better one, and (2) no matter how bad the plan, the robot could follow it at some speed as long as there are no positional discontinuities. This worked reasonably, but the plans were hard to follow and often contained unnecessary motion. It did work however, and the robot was able to move at around $0.8m/s$, less that the $1.2m/s$ that could safely be achieved by our pre-existing reactive obstacle avoidance system. The final optimization we made was the post-process the plan, which helped greatly. After a path had been determined, the post processing iteratively tested replacing the head of the plan with a single straight path, and would keep trying more of the head of the plan until it would hit an obstacle. Not only did this smooth out the resulting plan, but the robot tended to go straight at the first "restricted" point, always trying to aim at the free space. This helped tremendously, allowing the robot to navigate at up to $1.7m/s$, performing better than any previous system we have used on our robots. Videos of the this system are available in the supplemental materials [21].

The best combination of parameters that we were able to find, trading off physical performance and success with execution time was the following: 500 nodes, 200 waypoints, $P[goal] = 0.1$, $P[waypoint] = 0.7$, and a step size of 1/15sec. To examine the efficiency gain from using a KD-tree, we ran the system with and without a KD-tree. The results are shown in Figure 4.3. Not only does the tree have better scalability to higher numbers of nodes due to its algorithmic advantage, but it provides an absolute performance advantage even with as few as 100 nodes. Using the tree, and the more efficient second implementation for the robot rather than the initial prototype simulator, planning was able to perform on average in 2.1ms, with the time rarely going above 3ms. This makes the system fast enough to use in our production RoboCup team, as it will allow 5 robots to be controlled from a reasonably powerful machine while leaving some time left over for higher level action selection and strategy.

4.3.2 Conclusions in Applying ERRT to Soccer Robot Navigation

In this work a robot motion planning system was developed that used an RRT path planner to turn a simple reactive scheme into a high performance path planning system. The novel mechanisms of the waypoint cache and adaptive beta search were introduced, with the waypoint cache providing much improved performance on difficult but possible path planning problems. The real robot was able to perform better than previous fully reactive schemes, traveling 40% faster while avoiding obstacles, and drastically reducing oscillation and local minima problems that the reactive scheme had. This was also the first application of which we are aware using an RRT-based path planner on a real mobile robot.

Several conclusions could be drawn from this work. First, a heuristic approach (incorrectly) assuming a purely holonomic robot for the extend operator may perform better than a more

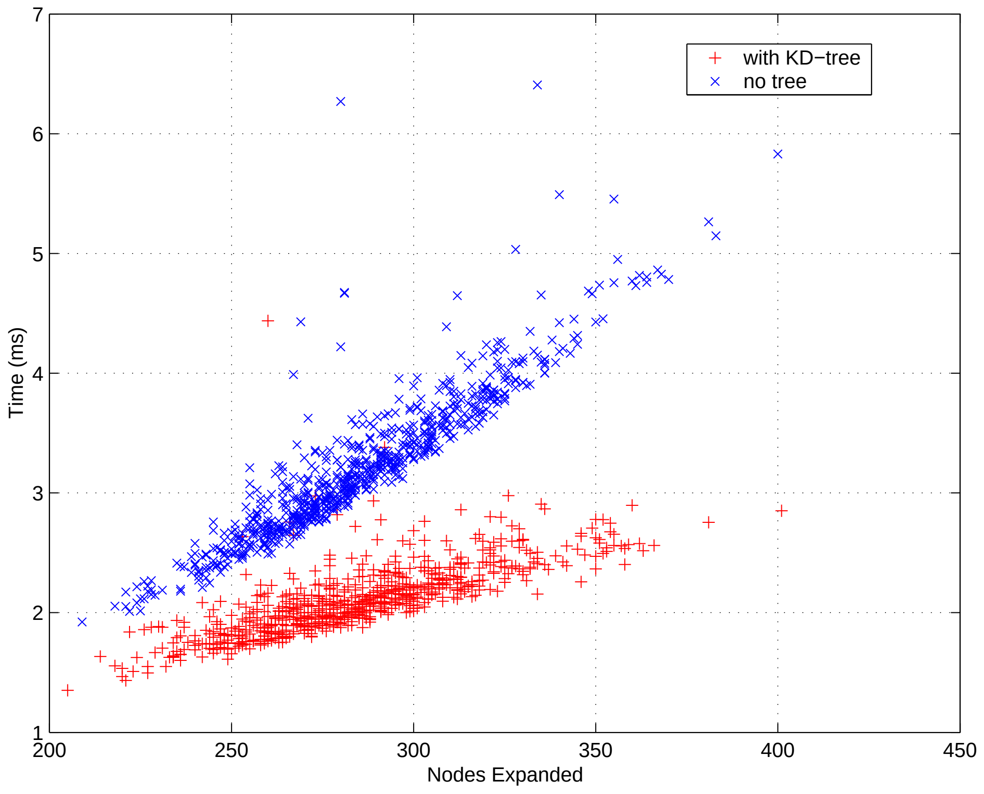

Figure 4.3: Planning times vs. number of nodes expanded with and without a KD-tree for nearest neighbor lookup. The KD-tree improves performance, with the gap increasing with the number of nodes due to its expected complexity advantage ($E[O(\log(n))]$ vs. $E[O(n)]$ for the naive version).

correct version with a dynamics model when planning time is critical. In other words, a better model may not improve the entire system even if it makes the generated plans better, because the more correct model severely limits the number of points or iterations that can be run. Second, it was found that plan simplification was necessary when incorrect motion models are used. Our real robot system worked without any plan post processing, but not nearly as well as when a local post-processing metric could apply its more accurate model over the head of the plan and thus remove most of its excess motion for the period about to be executed. Finally, the highly flexible extend operator of RRT demonstrates how existing reactive control methods can easily be added to improve the planner by adding domain specific control rules. ERRT can build on these reactive methods to help eliminate oscillation and local minima through its lookahead mechanism. Together they can form a fast yet robust navigation system.

4.3.3 Exploring waypoint Cache Replacement Strategies

In the base ERRT planner, the waypoint cache of previous plans is a fixed size bin with random replacement. However many other possible replacement strategies are possible, so we thought it to be an interesting area to explore. One approach that is even simpler than random replacement is to simply use the last valid plan in its entirety. By not splitting the plan, we still maintain a sequence ordering which allows an optimization: drawing waypoints from the cache which the search tree has already reached only wastes computation, since the tree already can reach that far up the plan. Thus the truncation heuristic was devised: Any time the planner reached a waypoint w that it was extending toward, successive draws from the waypoint cache are limited to nodes that came after w in the plan stored in the cache. This effectively limits and focuses the waypoints drawn to those areas not yet explored by the search tree. We called this heuristic "truncation" as can inactivate the leading segment of the waypoint cache during planning.

To test the effect, we created a sample set of domains inspired by the RoboCup small environment, with additional obstacles added to make the planning problems more complex. The planner was run for 2000 iterations on the first six example environments shown in Figure 4.7 (i.e. all those with 34 or fewer obstacles). Each environment is approximately 5.5m by 4.1m, and the agent's avoidance radius is 90mm. The initial and goal positions goal positions were varied in a vertical sinusoid to represent the changing robot position and newly calculated target points for navigation.

Since the waypoint cache's effect on planning is dependent on how frequently we select from the cache, the waypoint cache drawing probability was also varied. In all the problems, the probability of picking the goal configuration was fixed at $p = 0.05$. The timing results from running on a 2.4GHz AMD Athlon-based computer are shown in Figure 4.4. First, we can see that all methods performed well, planning in less than 1ms on average. Next, we can see that there is little difference between the random replacement bin and the last single plan approaches. When we apply the truncation heuristic to the last single plan approach however, we achieve a noticeable speedup. On average, planning times improved by 25% in the range of "reasonable" waypoint cache probabilities from 0.5 to 0.8. Thus in further ERRT derived planners, the truncation heuristic as an alternative to random replacement bins of the original ERRT approach.

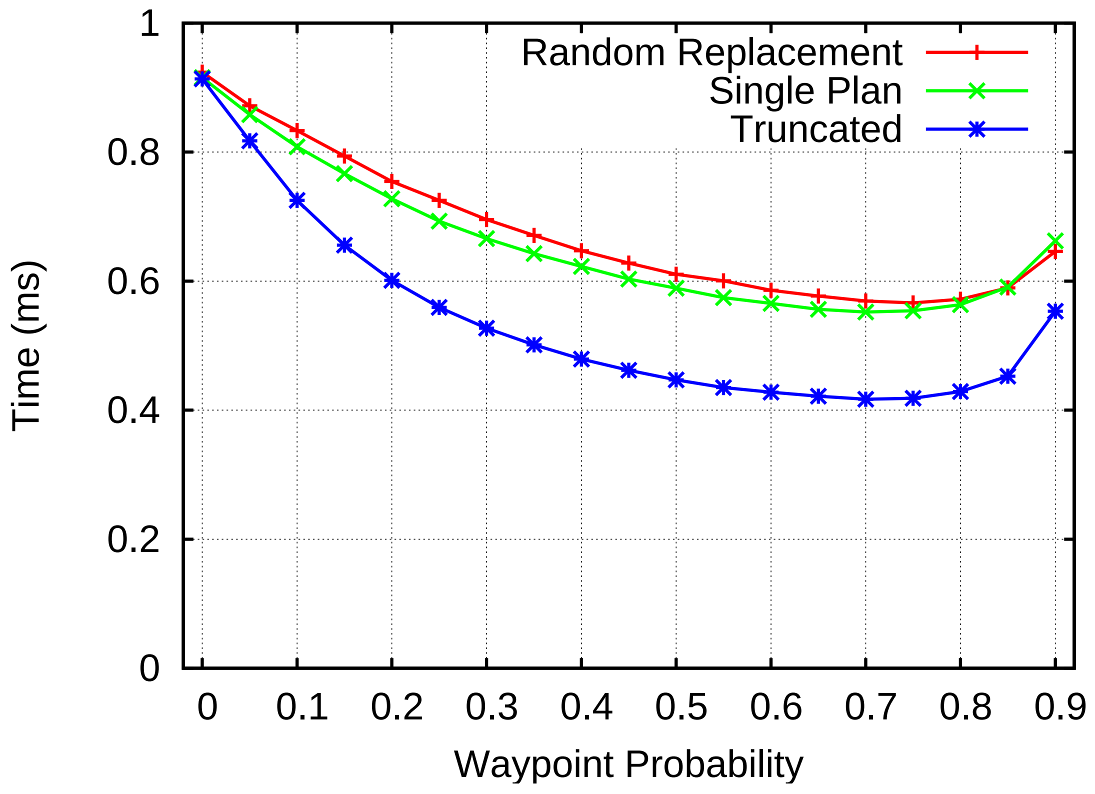

Figure 4.4: Comparison of several waypoint cache replacements strategies while varying the waypoint cache selection probability.

4.3.4 Bidirectional Multi-Bridge ERRT

One variant of the RRT planner which has demonstrated high performance for one-shot queries is RRT-Connect [51]. It combines bidirectional search with an iterated extension step. In RRT-Connect, a random target is chosen just as with the base RRT planner. However, instead of calling the extend operator once, the RRT-Connect repeats the extension until either the target point is reached, or the extension fails (such as when it would hit an obstacle). The search is performed bidirectionally, with a tree growing from both the initial and goal configurations. In each step, after a random target point is chosen, one tree repeatedly extends toward that target, reaching some final configuration q'. The second tree then repeatedly extends toward q' (as opposed to the random target), in an operation referred to as *Connect*. After each iteration, the tree swaps roles for extending and connecting operations, so that both the initial and goal trees grow using both operations.

While these improvements can markedly improve RRT's one-shot planning time, they do have an associated cost. While RRT, with its single extensions, tends to grow in a fixed rate due to the step size, RRT-Connect has a much higher variance in its growth due to the repeated extensions. As a result, when planning is iterated, RRT-Connect tends to find more widely varying homotopic paths. This is not an issue for one-show planning, but can become a problem for iterated planning with interleaved execution. Thus ERRT adopts the improvements of RRT-Connect, but modified somewhat. First, ERRT supports repeated extensions, but only up to some maximum constant, which can be tuned for the domain. Figure 4.5 shows the effect of this parameter on the average plan length for a domain. Each datapoint is includes the mean and confidence interval for 300 test runs, where each run represents 240 iterated plans on the domain *RandRect* (see Figure 4.7), under sinusoidal motion of the initial and goal positions. As the number of extensions increases, the plan average length grows. While applications can apply smoothing to remove some of the inefficiency of less optimal plans, a shorter plan is more likely to be in the same homotopy class as the optimal plan. Thus plan length is at least one indicator of the reachable length even after smoothing, and stability in the homotopic path is important for interleaved execution of the plan. In the figure, it is unclear why the average plan length drops after 10 repeated extensions, although this change is not large in comparison to the confidence interval. Thus repeated extensions, while they may speed planning, may come at the expense of average plan length. ERRT, by using a tunable constant to set the maximum number of extensions, allows the system designer to trade off between the two. In most applications, we have used a value of 4, and found it to represent a reasonable compromise.

ERRT also adopts the notion of the connect operation from RRT-Connect, however this can also cause plan length to suffer. ERRT again offers a way to mitigate this with a tunable parameter, which is the number of connections between the initial and goal trees before planning is terminated. Thus, ERRT allows multiple connection points, and can choose the shortest path of those available when planning terminates. Implementing such a feature represents an important departure from RRT however; with multiple connections the connected planning "tree" is actually now a graph. This does not pose any significant

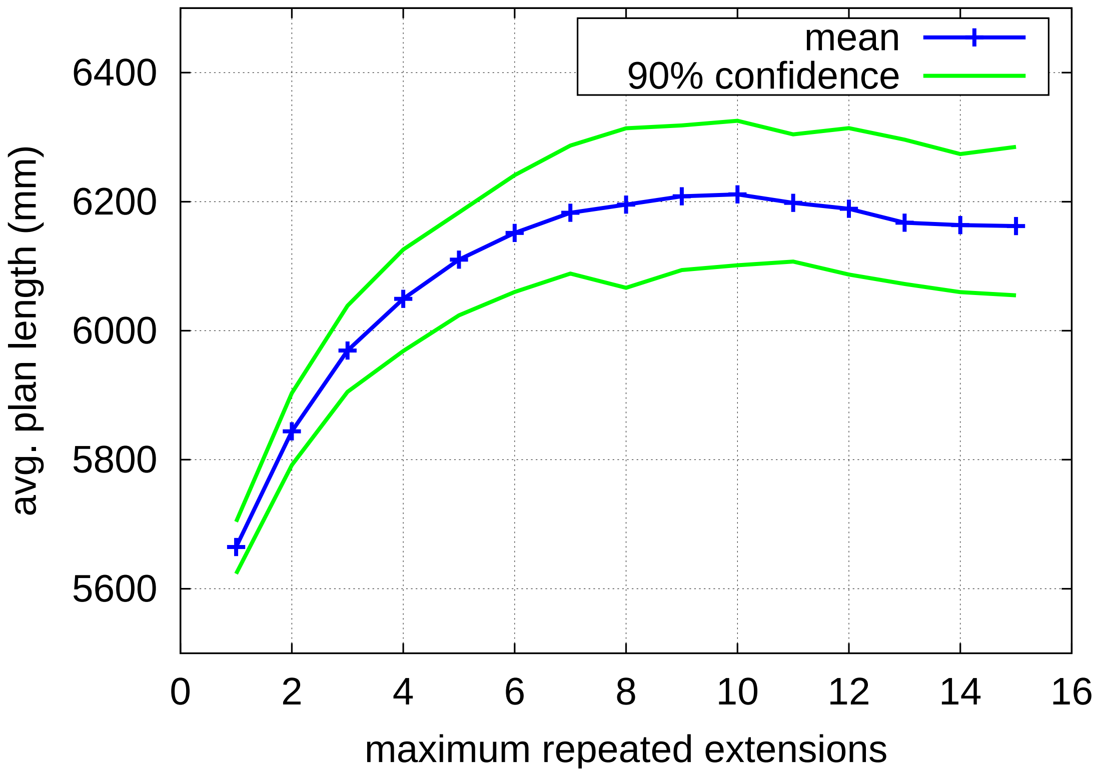

Figure 4.5: The effect of repeated extensions in ERRT on plan length.

theoretical problems to RRT, but requires the implementation to support graph operations efficiently instead of the normally faster tree operations. When planning terminates, A* search is used over the resulting graph. The effect of varying the number of connection points is shown in Figure 4.6. The methodology is the same as used for the maximum repeated extensions, but using the domain *RandCircle*. As can be seen, increasing the number of connections improves the plan length, although the effect decreases after 6-8 connection points. Multiple connection points by definition increase planning execution time, since ERRT continues to search even after a path has been found. However, after the first connection, ERRT can operate in a purely any-time fashion, using up idle time in the control cycle to improve the solution, but able to terminate and return a valid solution whenever it is needed. Again, by offering the number of connections as a parameter, the tradeoff can be set by the system designer.

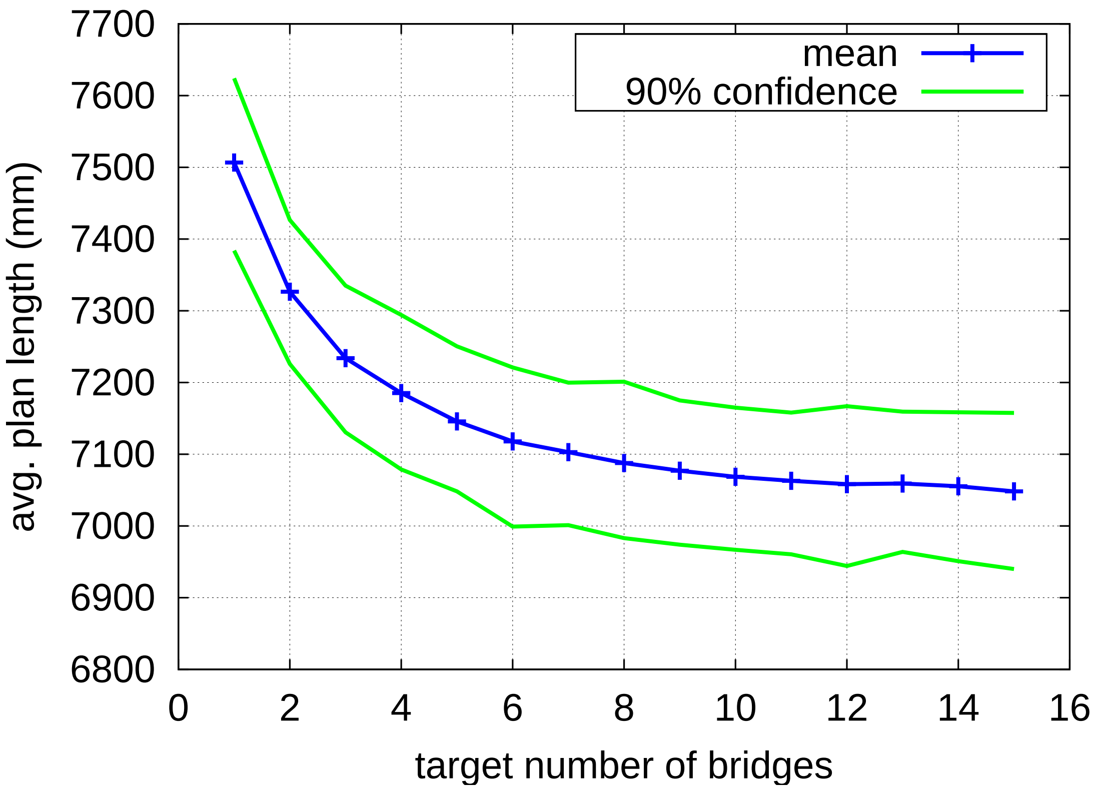

Figure 4.6: The effect of multiple connection points in ERRT on plan length.

Supporting multiple extensions and multiple connection points give ERRT the benefits of RRT-Connect, as well as the ability to tune the system as needed. By supporting a graph representation instead of search trees, as well as planning over the resulting roadmap graph, ERRT takes a step toward unification with the PRM family of planners. The only differences that remain are in the sampling strategies for building and maintaining the search graph or roadmap structure.

4.3.5 Comparing ERRT with the Visibility Graph in 2D

Since ERRT operates in 2D for our test domains, it is a reasonable question to ask whether it offers any advantage over traditional methods. Because ERRT is a continuous domain planner, it is most directly comparable to the Visibility Graph method [69] (see 6.1.3). Although ERRT is not limited to 2D, it can be compared with the Visibility Graph within this context. In addition, as the Visibility Graph is a minimum-length-optimal planner, we can compare the plan lengths produced by ERRT to the optimal lengths. The environments used can be seen in Figure4.7, and range from four to 128 obstacles, and cover a broad range of free volume ratios and C-space connectivity. The domain is roughly modelled on RoboCup small-size, where each environment is 5.5m by 4.1m, with the agent's avoidance radius set at 90mm. The agents moved in a vertical sinusoid with a period of 120 iterations, and the data points are the mean values of 2000 planning iterations. The ERRT parameter values are shown in Table 4.3.5.

ERRT Parameter	Value
Maximum number of nodes	512
Goal target point probability	0.05
Initial target point probability (for bidirectional search)	0.05
Waypoint cache selection probability	0.80
Waypoint cache size	100
Extension step size	120
Max repeated extensions	4
Target number of connections	4

Table 4.5: The parameter values for ERRT used in the benchmark experiment.

The results of the benchmark testing can seen graphically in Figure 4.8, and a tabular summary in Table 4.3.5. First, it is apparent that ERRT has an advantage in execution time for more complex domains, particularly those with many small obstacles. Because the Visibility Graph must model all possible tangents in its constructed roadmap, it scales poorly with obstacles. It uses up to $O(n^2)$ edges for n obstacles, even for domains that conceptually that difficult, such as *CircleGrid* and *BoxGrid*. ERRT fares worst in a comparative sense

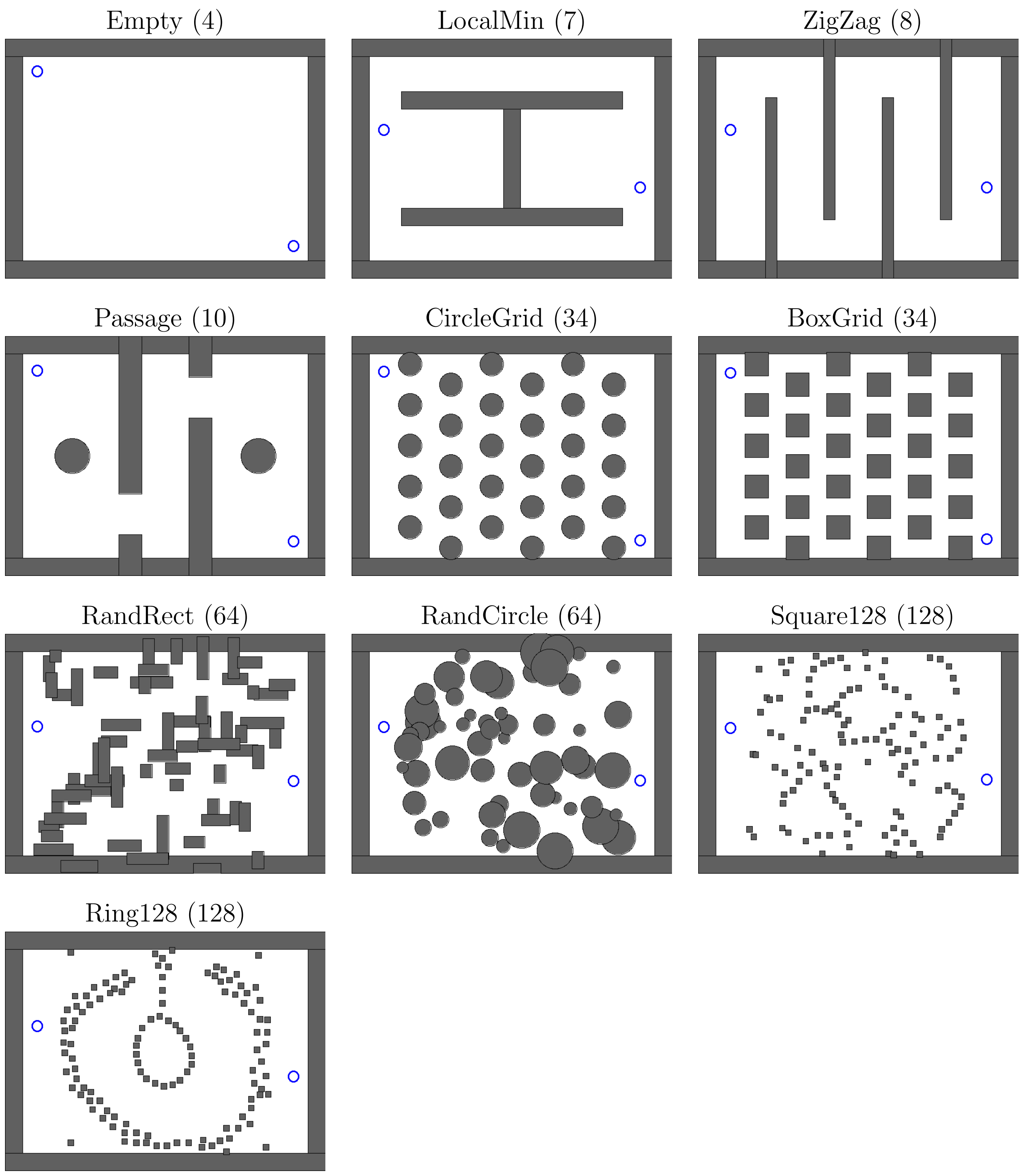

Figure 4.7: Environments used for benchmarking planners.

on *ZigZag*, which has few obstacles but a long solution path. ERRT takes almost 13 times the execution time of Visibility Graph, although ERRT still only requires 1.4ms of planning time on average for the domain. ERRT fares best on *Square128*, running 28 times faster than Visibility Graph, which takes over 12ms of planning time. ERRT's slowest execution time across all domains is only 2.2ms.

In terms of success probability, as shown in part (b) of Figure 4.8, Visibility Graph demonstrates its completeness, never failing to generate a solution, while ERRT is not far behind. ERRT is worst on *Ring128*, where it finds a solution 97.5% of the time, second worst on *ZigZag* at 99.6%, and 100% on the eight other domains. The success probability is adequate for all these domains when using iterated planner, as failures in one iteration only result in a short stall of execution. Finally, in terms of plan length ERRT is at worst 28% longer than the optimal path, even without whole-path optimization. This is aided by tuning the extension step size, maximum repeated extensions, and the number of connections for bidirectional search.

Domain	VisGraph exec. (ms)	ERRT exec. (ms)	speedup (VisGraph/ERRT)	relative plan length (ERRT/VisGraph)
Empty	0.057	0.091	0.626	1.046
LocalMin	0.102	0.558	0.182	1.154
ZigZag	0.110	1.401	**0.078**	**1.283**
Passage	0.154	0.300	0.513	1.225
CricleGrid	1.654	0.245	6.746	1.077
BoxGrid	1.623	0.522	3.107	1.226
RandRect	1.986	0.432	4.596	1.132
RandCircle	1.311	0.652	2.011	1.124
Square128	**12.194**	0.428	**28.471**	1.163
Ring128	7.329	**2.199**	3.333	1.246

Table 4.6: A comparison of ERRT with the visibility graph method across several 2D domains.

In summary ERRT compares favorably even against the classic optimal 2D Visibility Graph planner. Even in relatively simple domains, if numerous obstacles need to be supported with low execution times, ERRT makes a good choice in place of Visibility Graph. With some robotic applications modelling the environment with large numbers of obstacles retrieved from sensors, this can be an important property. If however, the solution paths are long, with only a few obstacles, ERRT does not appear to be the best choice for 2D planning. ERRT's advantage with large numbers of obstacles, along with its ability to operate in higher dimensions inherited from RRT, appears to make it a reasonable choice for general mobile robot planning applications.

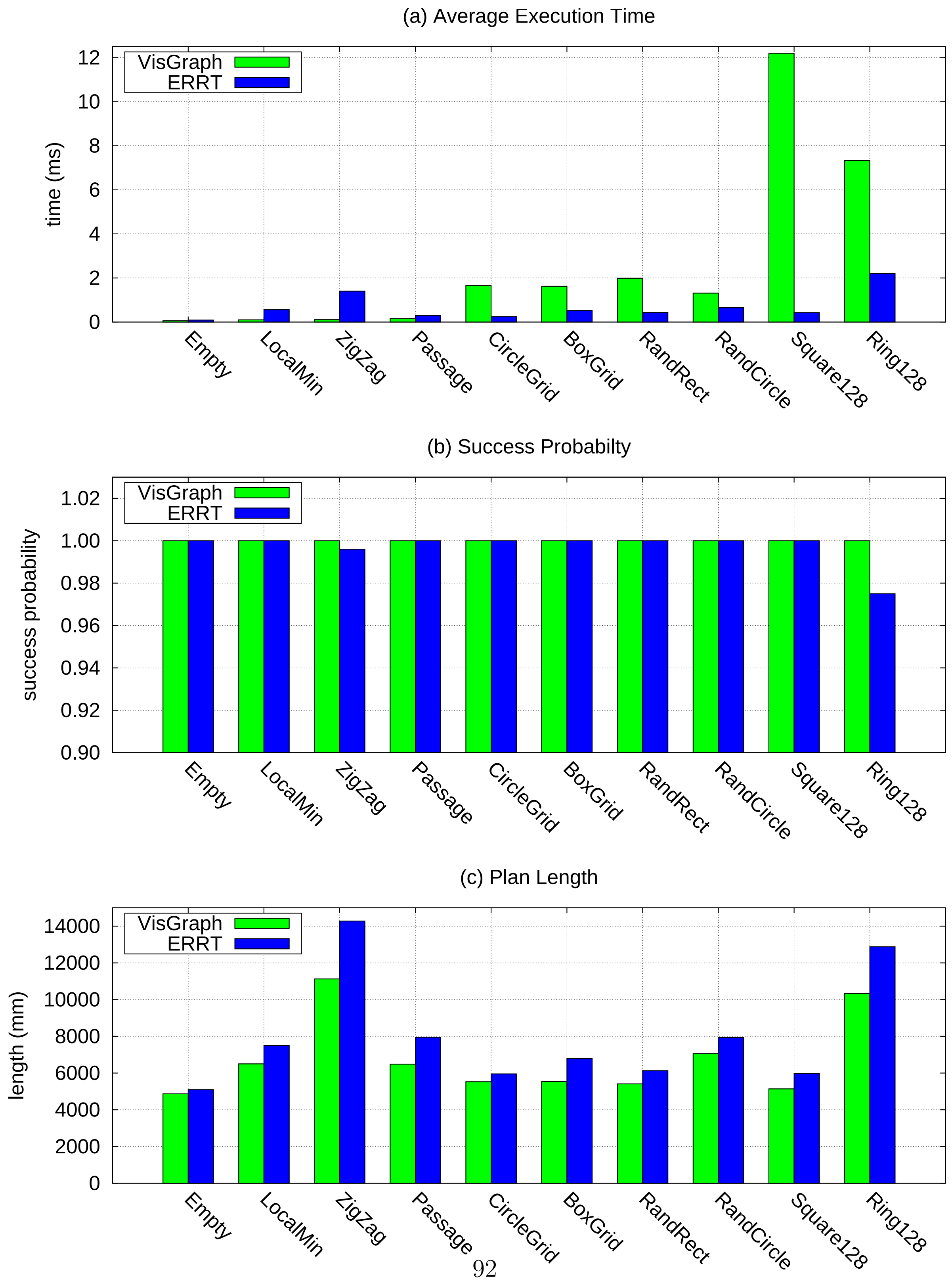

Figure 4.8: A comparison of ERRT with the visibility graph method across several 2D domains.

4.4 Dynamic PRM

PRM planners have proved to be the most popular randomized planner for static environments. Since they split planning into an environment learning stage and a very fast query stage, they work very efficiently for environments that do not change often. In addition, they support explicit minimum cost searches of the roadmap, which is useful when there are non-binary obstacles with different traversal costs. Unfortunately, for mobile robotics problems the environment is often non-static, which means the learning stage needs to be rerun frequently. However, a large class of environments can be considered *nearly static* however; For example, in a typical cluttered room only a few objects move at any time. One particularly motivating domain for this approach is the QRIO humanoid domain, where path stability is important due to high action latency, and the environmental obstacles evolve slowly over time.

The only change that needs to be made to support dynamic environments is make the learning stage operate incrementally. First, each node in the roadmap graph is augmented with a replacement probability. Every sensor update cycle, all nodes are checked for validity. They are marked as invalid if they no longer represent a free configuration. Additionally, they can be randomly invalidated based on their replacement probability. All edges are then checked to make sure they represent free paths, and marked invalid if they would result in a collision, or if either of their two endpoint nodes has been marked as invalid. After this step, the invalid nodes and edges are removed from the graph. The next step is to add nodes and edges to the graph so that roadmap size and connectivity are maintained. Unlike advanced static PRM planners, Dynamic PRM uses uniform sampling to generate configurations. In order to achieve non-uniform graph density however, the initial replacement probability of a node is based on its proximity to obstacles, decreasing as it gets further from obstacles. After many iterations, this converges to a distribution similar to Gaussian PRM [9], which samples more configurations near the boundary of free space. However Dynamic PRM can achieve that density without the necessity for running more costly rejection sampling techniques for free configurations, instead spreading the work over time.

Adding edges to a roadmap is an extremely costly operation if performed naively. The simplest algorithm to implement would be to test all pairs of nodes, which of course scales quadratically, and is not practical for large roadmap graphs. In [54], a variation is selected where the nearest neighbors from different connected components are chosen, up to some distance threshold or degree limit. This results in a forest, which can poorly describe workspaces with cycles; The query phase will only return one homotopic path, while in the case of cycles we would like it to return the shortest path (or any other cost metric minimized during the graph search). If we remove the tree constraint however, then local cliques may tend to form where all the nodes are closer to each other than neighboring groups, and thus fail to connect to the graph outside of that group, even though a free path exists. Furthermore, querying the closest set of neighbors is a more complex operation than returning only a single nearest neighbor for a query.

In order to explore this effect, Dynamic PRM implements three possible sampling strategies for adding edges. The first technique, called *nearest-k* is the traditional PRM approach of attempting to connect to all nodes up to some limit k within some maximum connection distance d. The second approach, called *gauss*, randomly samples a point x near a particular node q based on the a multivariate Gaussian centered on that node with standard deviation $d/2$. The random point is then used to query the nearest node q' in the roadmap, and a connection is attempted between q and q'. In other words, the method takes a random step, and then looks for a neighboring node in that area with which to connect. This is repeated several times to form multiple connections. The third method is a refinement of *gauss*, called *ring*, which independently samples a direction and a distance. The direction is sampled uniformly, while the distance is a Gaussian centered at some non-zero distance. The resulting distribution for the random step in 2D is approximately[1] a Gaussian revolved around the origin, and thus resembles a “fuzzy ring”.

To visualize how the sampling method of edge connection works, Figure 4.9 shows a simplified sampling method which samples at fixed angles in 2D. On the left, the sampled steps are shown extending radially from a node we wish to connect to other nodes in the roadmap. After taking the nearest neighbor to the end of each of the steps, and connecting those two nodes if a straight-line path exists, the resulting roadmap is shown on the right of the figure. The *guass* and *ring* methods operated similarly, although the randomized directions allow them to generalize to higher configuration space dimensionality.

Several enhancements to the basic algorithm can be applied to improve performance. First, in order to fill in gaps left behind moving objects, it is necessary to recycle nodes, freeing them up to be used in a new location. Although all nodes have a random replacement probability, the recycling rate can be more directly controlled by introducing a parameter called node aging. It is an increment applied to node replacement probabilities so that older nodes are more likely to be replaced than new ones. A lower increment is more efficient since less time is spent adding new nodes and edges to replace the replaced ones, while a higher rate helps fill in gaps faster for dynamic environments. A second optimization is that during the query stage, all nodes used in the final path have their replacement probabilities set to zero. This ensures they won’t be replaced in the next sensor update cycle, and less likely immediately afterward than other nodes. It also effectively increases the sampling density

[1]It is approximate because the true density increases as one nears the axis. The larger the mean offset compared to the standard deviation, the more the distribution will approach a revolved Gaussian.

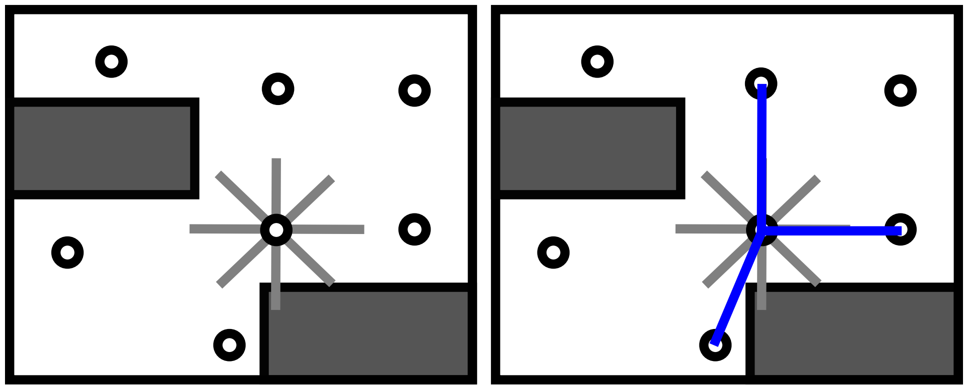

Figure 4.9: Dynamic PRM connection technique, using radial direction sampling. The sampled directions (left) and the resulting roadmap (right) are shown.

along the path since nodes are still added to those areas at the same rate. Higher sampling density helps to improve the local quality of the path.

To test the algorithm and the multiple sampling strategies, a test domain was constructed which is pictured in Figure 4.10. An user interface was created for testing where the a user can drag the goals and obstacles interactively as the system plans and navigates in real-time. For benchmarking the algorithm, the domain is altered systematically. First, the robot starts in one corner of the maze-like environment, and has to reach a goal in the next corner in a counter-clockwise fashion. Each time the robot reaches a goal, the goal position is moved to the next corner, resulting in the robot doing "laps" around the environment. The planner was allowed up to 500 nodes, and up to eight edges per node in constructing the roadmap. Although the obstacles do not move in the benchmark, the algorithm assumes arbitrary changes in the environment may have taken place, as is possible in the interactive testing mode. Thus, the Dynamic PRM method used aging and removing edges and nodes in the roadmap, and sampling continuously to extend the roadmap into existing or possibly new free configuration space.

The results of benchmarking can be seen in Table 4.4. The simulated robot was run for 32 laps around the environment for each sample, and the test was repeated 8 times to generate a mean and standard deviation for each measurement. The simulated time represents the total time for the agent to complete 32 laps, and is mainly depending on successful planning and the quality of the roadmap paths. A failing planner or poor quality paths will increase the total simulated time. Execution time represents the average execution time of the planner each control cycle. Success probability is the chance that the planner returns a full path to the goal each control cycle. Finally, roadmap entropy measures the connectedness of the roadmap by measuring the total entropy based on the sizes of the connected components. A lower

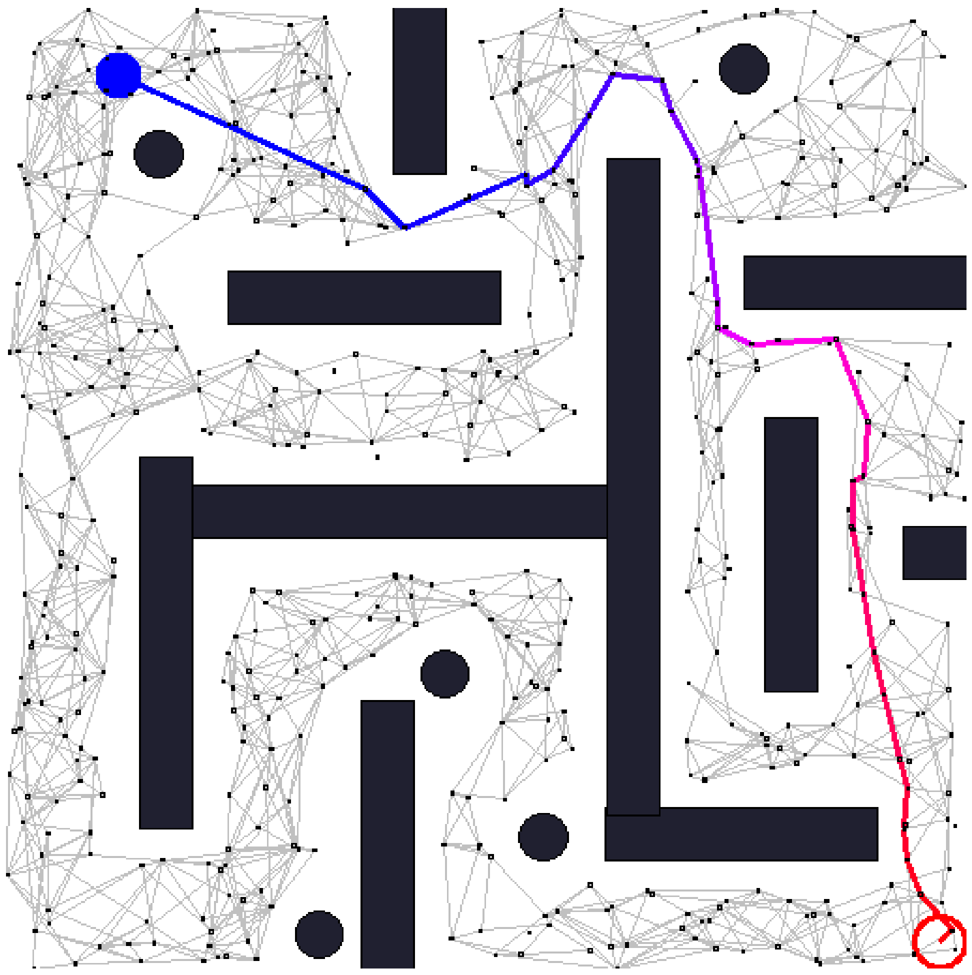

Figure 4.10: The test domain for interactive and benchmark testing of Dynamic PRM.

entropy indicates a more connected roadmap. As can be seen in the table, few differences exist between the methods. The *ring* sampling method leads *nearest-k* in all categories, and leads *gaussian* in all but one category, although the results are not statistically significant. The high success probability is due in part to the optimization of never removing edges from the roadmap used in the plan of a previous cycle, however even then the sampling methods do not have a particularly large impact. This can be seen as a positive result however, because the sampling methods are easier to implement than *nearest-k*. This is due to the fact that the sampling methods do not require a multiple-nearest-neighbor query to be implemented by the spatial data structure storing the roadmap. Larger differences have been found between the methods for roadmaps with an insufficient number of nodes, and thus a low planning success probability, however this is not particular advantage. Such badly-tuned parameters would be avoided in most implementations anyway, as tuning the number of nodes is just as easy as changing the sampling method.

Sampling Method	Simulated Time [sec (std)]	Avg. Execution Time [msec (std)]	Success Prob. [p (std)]	Roadmap Entropy [e (std)]
nearest-k	327.8 (6.49)	1.059 (0.00237)	0.9810 (0.00314)	1.030 (0.0173)
gaussian	324.6 (5.42)	1.051 (0.00212)	0.9808 (0.00315)	1.030 (0.0090)
ring	323.1 (4.00)	1.052 (0.00203)	0.9815 (0.00320)	1.029 (0.0122)

Table 4.7: A comparison of performance of Dynamic PRM with various approaches for roadmap connection.

Although not explored to the extent that RRT derivatives have been investigated, Dynamic PRM demonstrated high success probability and good coherence between control cycles, making it a good choice for slowly evolving domains such as the planner for the QRIO humanoid.

4.5 The Abstract Domain Interface

The domain interface resulted from an attempt to unify platform interfaces so that common planning code could be developed. In traditional planning work, there are typically three primary modules: Planner, collision detection, and a platform model. While this approach works well for robots of relatively similar type, the communication required between the platform model and collision detection are through the planner, complicating the interface so that the planner needs to know much more about the domain than is really necessary. Thus the current approach was devised, in which platform model and collision detection are wrapped into a single “domain” module that the planner interacts with. Internally, collision detection and the platform model are implemented separately, but importantly the planner doesn't depend on anything involved in the communication between the two. This allows states in the configuration space to be a wholly opaque type to the planner (denoted by S), which needs only to be copied and operated on by the domain's functions. Additionally, to speed up nearest neighbor lookups, the state must provide bounds and accessors to its individual component dimensions. This allows the planner to build a K-D tree of states so that linear scans of the tree are not necessary for finding the nearest state to a randomly drawn target. While the traditional architecture can be made nearly as flexible, it typically does so at a cost in efficiency. The domain interface approach leaves open the opportunities for improved collision detection speed that can come with constraints or symmetry present in the agent model. The domain operations are as follows:

- *RandomWorldTarget():S* - Returns a state uniformly distributed in C
- *RandomGoalTarget():S* - Returns a random target from the set of goal states
- *Extend(s_0:S,s_1:S):S* - Returns a new state incrementally extending from s_0 toward s_1
- *Check(s:S):bool* - Returns true iff $s \in C_f$
- *Check(s_0:S,s_1:S):bool* - Returns true if swept-sphere from s_0 to s_1 is contained in C_f
- *Dist(s_0:S,s_1:S):real* - Returns distance between states s_0 and s_1
- *GoalDist(s:S):real* - Returns distance from s to the goal state set

Using these primitives, the ERRT planner can be built which operates across multiple platforms, and does so without sacrificing runtime efficiency. In addition, in the case of a purely holonomic platform without dynamics constraints, a PRM variant can be implemented using only these primitives (the local planner is formed by calling *Extend* repeatedly). One could argue that the abstract domain approach moves almost all of the "important" code into the domain itself, making the planner itself simplistic. In other words, all of the "planning intelligence" has been shifted to the domain itself. However, the crucial difference is that the code in the domain is relatively straightforward and self contained, while the intricate interactions and practicality driven fall-back cases of planning reside in the core planning code. Thus one could implement a domain with little or no knowledge of path planning, reaching a core goal of general modular programming.

4.6 Practical Issues in Planning

4.6.1 Simple General Path Smoothing

Due to their random nature, the randomized planning approaches generate paths with additionally unnecessary motion present. Although technically safe, such motion is undesirable for direct execution by the agent. ERRT as originally posed used a "straight line" heuristic to smooth the beginning of the path. This approach would check the straight line from the initial position to each successive vertex along the path to see if they were free. The longest free line found was used to replace the head of the plan. For iterative execution this works well in smoothing out the motion of the agent; Later areas of the plan need not be smoothed until the agent reaches that segment. Thus our 2D holonomic planner adopted this simplification method and it has been found to work well in practice.

The straight-line method is however quite limited, since it does not respect kinodynamic constraints. Thus for the UAV planner, a method was needed that respected kinodynamic constraints. At the same time, it is a desirable feature that the smoothing method not be tied to the exact constraints, so that the smoothing would not have to be reimplemented if the constraint model was changed. Thus a more general method was desired, as well as one which operated on entire paths if possible. The method we subsequently developed follows a "leader-follower" approach. The returned plan from ERRT is treated as a "leader", and a new path is grown as a follower. The lead point on the plan maintains a minimum distance from the follower path, and the follower path is grown using the domain's extend operator with the leader point as its target. This method is depicted in Figure 4.11.

In order to smooth the path maximally, the algorithm can be iterated, with successively longer minimum distances maintained by the leader in each pass. If the simplification fails (due to the follower's extension hitting an obstacle), the last successful simplification is returned. If the first few levels of simplification fail, and excess random motion is unacceptable for that platform, it can be treated the same as a plan failure. By running the planner several times, we can obtain a statistically very high probability of a successful smoothed plan being returned. Given the speed of the basic path planner, this can be a viable approach.

4.6.2 Robust Planning

An additional issue important for planning on real robot platforms is that of robustness. While iterative replanning helps deal with positional error within free space, it still does

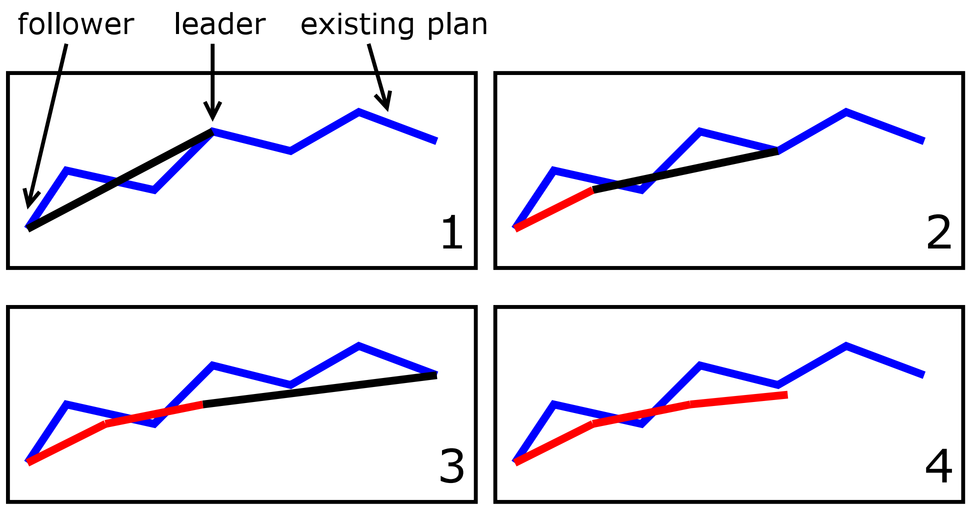

Figure 4.11: Path smoothing using the leader-follower approach.

not gracefully handle problems when the robot steps outside of free space. In practice this can happen quite easily due to sensing error in position, or action errors driving us off the expected course. Implementation on the ground robots found numerous instances where the robot would drive until an unfortunate series of sensor or action error resulted with the robot having 1% of its radius lying with an obstacle. Intuitively we'd like the robot to fail gracefully and plot a smooth path that leads out of the obstacle, but a typical planner only has a binary view of safety, thus it simply fails to generate a path at all. This is because all obstacle checks from the initial position fail at the initial position itself, preventing any extensions from growing the tree. In previous planning work we explored relatively ad-hoc approaches to returning to free space, but none was general enough to apply across domains. Ultimately, we decided to split the general situation of "in an obstacle" into two categories. The more common category is that the agent is only partially inside an obstacle while it's center coordinate is not. This is expressed as a obstacle distance from the center of less than the robot's radius r but greater than 0. The more serious case is when the robot's center is within an obstacle, resulting in an obstacle distance of 0 and meaning that deep penetration has occurred. While the latter case can only really be handled in a domain dependent way (some sort of safe failure or domain specific failure recovery), the partial case can be handled in a general way. Our planner does so in a graceful way to avoid abrupt changes in trajectory which might exacerbate the problem instead of correcting it. When planning is initiated, if the the initial position q_i is found be partially within an obstacle ($0 < D(q_i) < r$), then collision checks from q_i are treated specially. When checking a swept-line from q_i to any point b, we first find a point along the line f where $D(f) \geq r$. Then we check the swept line from q_i to f with a radius of $D(q_i)$, and a second swept line from f to b with a radius of r. This is shown in Figure 4.12. The first condition ensures the trajectory does not penetrate the obstacle any worse than the initial position, while the segment from f to b ensures that the trajectory stays out of obstacles once it enters C_f. To keep the system independent of the exact geometry of obstacles, f is found by recursive bisection, with a domain specified maximum distance from q_i. Changing the maximum allows varying how aggressively the robot attempts to exit obstacles. The system has been found to work well in practice, handling issues where minor penetration into the obstacles occurs. In doing so, it helped decrease the occurrence of major penetration so that the domain specific "escape" method is only called in cases of serious error or radically changing environments.

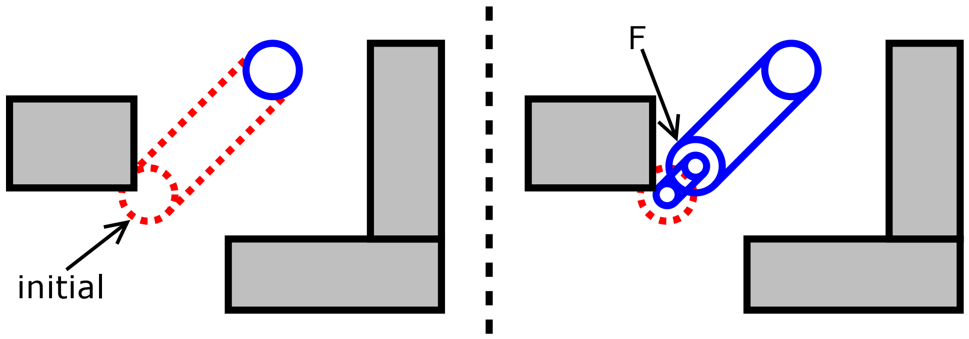

Figure 4.12: An example of a specially handled collision check when the initial position is partially penetrating an obstacle. The collision check is handled in two segments.

4.7 Applications

4.7.1 Fixed Wing UAV

In order to explore ERRT's capability as a planner in 3D environments, and in support of a project supporting FixedWing UAVs, a new abstract domain for ERRT to model UAVs (see 2.2. The UAV operated in 3D with limited climb and descent, as well as a severely limited turning radius. Kinematic constraints were supported by mapping extensions to the nearest reachable action, as in the example in Figure 4.14. The environment consisted of terrain segments, and a total of 1.5 million triangles (see 3.2.3). For speed reasons, a weighted Euclidean distance metric was used. Better metrics exist that depend on the kinematics, but these are difficult to support efficient nearest-neighbor queries for with a hierarchical spatial data structure. As a compromise, the distance function weights components of the position. x and y coordinates along the ground were defined with unity weights. z (altitude), was

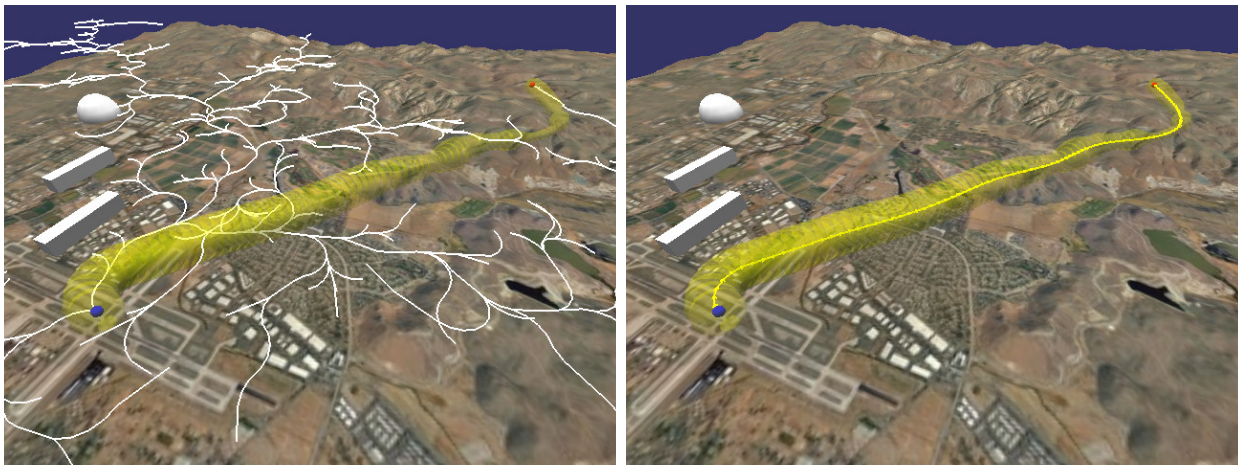
Figure 4.13: A kinodynamicly-limited search tree (left), and the corresponding simplified plan (right) for the UAV. The plan length is approximately 11km with waypoints every 100m. Obstacle clearance is shown as a translucent tunnel.

given a weight of 10, which was the minimum required to generate a “spiraling” behavior when repeated extensions are performed with a relative target that differs only in the z dimension. Distance for x,y orientation was approximated using a heading vector scaled by the current aircraft velocity. The orientation was given a weight of 2, although it did not appear to have a large effect on the plans. The weighted Euclidean distance metric worked, although the planner would sometimes get stuck in a local minima, requiring a target point to be selected in a very small area for additional progress to be made. In addition, application usage constraints enforced one-shot planning by the navigation system, with no context associated with a problem. We solved both problems by iterating the planner on an identical problem instance - local minima were avoided by “sampling” of the planner itself, and multiple solutions can be ranked by quality criteria to help ensure that the returned plan tended toward the shorter and smoother plans of those available. Because no context was associated with a problem, and the replanning samples were ideally independent, the waypoint cache was disabled for this domain. Multiple repeated extensions were still used, although bidirectional search was disabled because the abstract domain only supports forward kinematic constraints. In the final version, the planner replanned up to 48 times or 12 valid solutions. It simplifies any valid solution with the leader-follower path smoothing approach, and picks the a plan based on the minimum cost weighting of length of curvature changes. Execution time was highly dependent on the configuration, but averages between 0.5 to 4.0 seconds on a 2GHz processor.

One optimization that contributed to the performance on difficult problems was adaptive iteration thresholds. With repeated replanning, once we have found a solution, that can serve

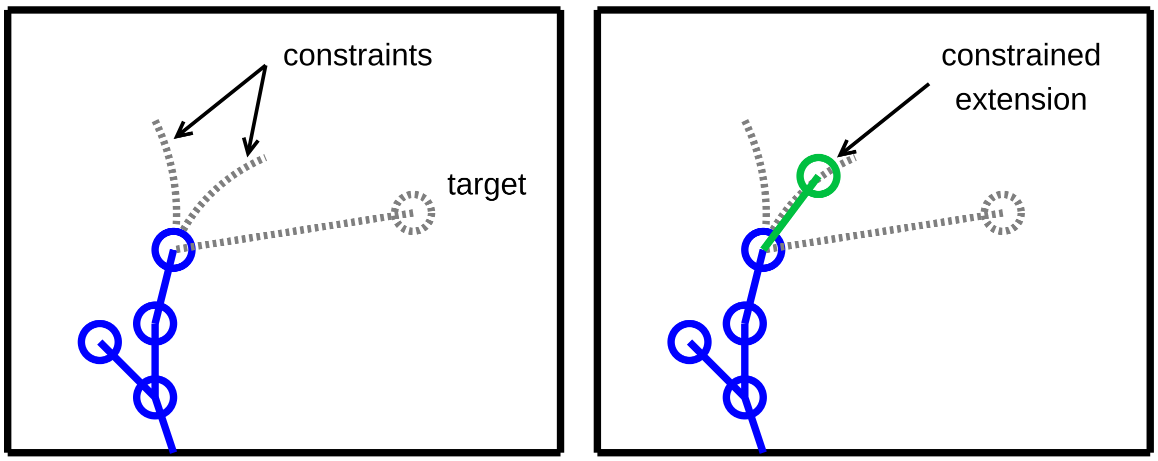

Figure 4.14: An example of limiting Kinematics for the RRT extend operator

as an existence proof that a solution can be found within a certain number of iterations. In subsequent iterations of the planner, we can use that number as a threshold speeding up failure detection if those searches are stuck in a local minima and will not find a solution. In order to help ensure multiple plan iterations are successful, the limit is set at a multiple of the number for the successful plans. Thus, we maintain a iteration threshold m, and update it whenever the planner is successful after i extensions using the rule in Equation 4.1, where k is some constant larger than one (we used $k = 2$ for our implementation).

$$m \leftarrow \min(m, ki) \tag{4.1}$$

The UAV domain motivated a unique new property for ERRT that mapped well onto the randomized sampling planning approach, and would be difficult to implement with a more model-driven approaches to planning. For fixed-wing UAVs, it is often desirable to fly between stable orbits, allowing an aircraft to maintain an average position while it maintains its required forward velocity to prevent stalling. Adding the ability to plan to an orbit is thus a useful extension, but one that could be expensive due to less defined nature of the goal along with the existing kinematic constraints. Because the planner is already rerun on the same instance multiple times however, we can get away with a heuristic that has a non-zero probability of achieving the objective (in fact all randomized planners are already this way due to probabilistic completeness). We thus developed a method we call *active goal*, where the nominal goal configuration is updated dynamically during search. For the fly-to-orbit feature, we want to fly to the tangent of some defined orbit. Using the active goal approach, we set the nominal goal to always be the tangent point relative to the closest point in the current search tree to the goal. Every time a new extension is added to the search tree, if it is closer to the goal configuration than the previously nearest point, the tangent is updated for the new configuration. An example is shown in Figure 4.15. This method works much of the time, as the point nearest to a goal in an RRT planner is by far the most likely to be extended toward a goal. Due to random exploration, its possible for another branch of the search tree to wander in and hit the goal non-tangentially, but in those cases, that search can be aborted as a failure, leaving a subsequent rerun of the planner to find the solution. Two results from the planner can be seen in Figure 4.16. In practice, the active goal approach allowed the UAV navigation system to support orbits as goals with negligible additional overhead compared to a fixed goal configuration.

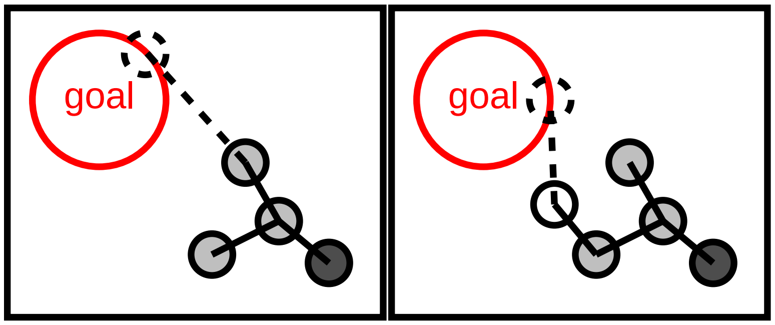

Figure 4.15: An example of a goal updating dynamically during search to try to maintain a tangent constraint. The original tangent, shown dotted (left), is updated as a the ERRT search tree expands (right).

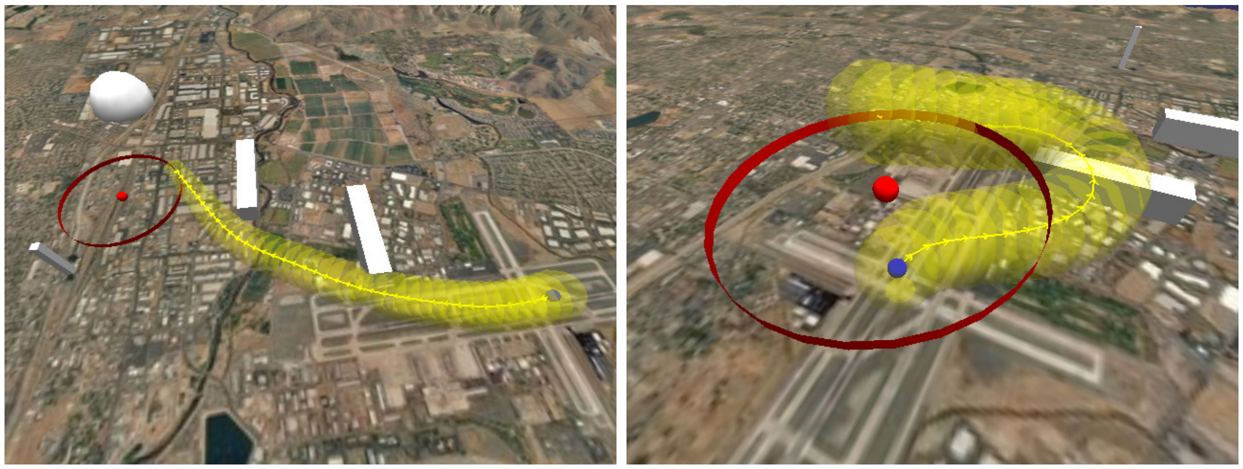

Figure 4.16: Two examples of the fly-to-orbit capability of the UAV planner.

Development of a unified ERRT planner for multiple mobile robot platforms provided many insights that would be difficult to determine if only one platform or similar platforms were considered for an implementation. However there are still many opportunities for further work. First and foremost, the relationship between distance metric, kinodynamic constraints, and accelerated nearest-neighbor search could be explored. Development of good distance metrics for kinematically constrained platforms is possible, but tedious, and more seriously it prevents most forms of accelerating nearest-neighbor search to fail because the triangle inequality is no longer satisfied. Using Euclidean distance worked, but generated local minima that could only be avoided by rerunning the planner several times, which is an inelegant solution. Better approximations which still allow the use of fast geometric data structures most likely exist.

4.7.2 QRIO Humanoid Robot Planner

Path planning for a small humanoid robot was also explored, using the Sony QRIO [42] (see 2.3). To implement the navigation module two different path planning approaches were attempted. QRIO builds a local occupancy grid of its environment, which it updates using depth maps calculated from stereo vision. Thus the environment is static except for the portion where the robot can currently see. New information is incorporated into the occupancy grid using Bayesian updates. ERRT was successfully adapted for purely 2D planning, but could not easily represent varying traversal costs which would be useful planning which incorporates 3D actions, i.e. climbing and descending.

First, a 2D path planner was implemented using the RRT-based method described above, which compared favorably to an existing grid-based A* implementation. For collision detection, a variant of the distance transform [62] was used on a thresholded occupancy grid, with each free cell storing a offset to the nearest non-free cell. By using the constraint that the nearest non-free cell is either the cell itself or one of the eight neighbors, the transform can be created in two passes, a forward pass where the neighbors to the left and above a cell are checked for their closest non-free cell, and a reverse pass where the neighbors to the right and below are checked. This approach avoids the overhead of the queue based implementation normally used, and results in an approximation which is perfect for convex obstacles, and has at most half of a cell of error in concave corners. The result is shown in Figure 4.17.

To extend the planner to 3D stepping actions, the occupancy grid world model was extended to include height values for each cell. This creates a heightmap, similar to that used in Chestnutt et al. [25]. The point obstacle check used for nodes in the roadmap was modified to check the flatness of the heightmap, using the standard deviation of the cells on which the robot would stand as a metric. The check for edges was separated into two cases, one for flat surfaces and one for stair transitions. The flat surface check used the point query to see if

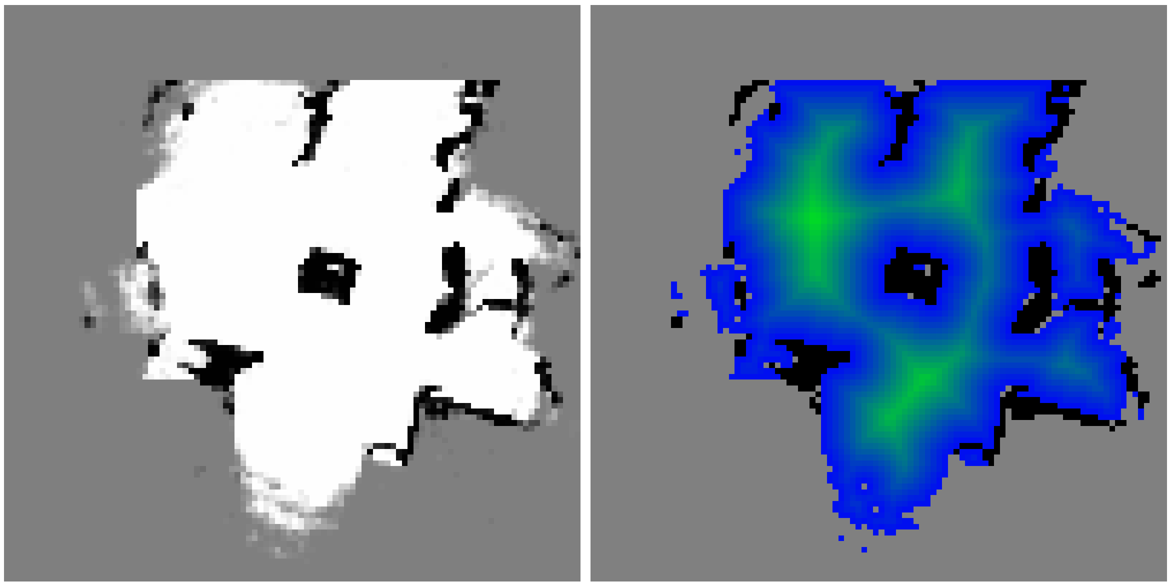

Figure 4.17: An occupancy grid and its distance transform.

robot could stand at intermediate positions after each step. A 2.5D planner using Dynamic PRM was subsequently developed using this collision detection model. The planner used heightmaps to check that no obstacle impinged on the legs or torso for a given location, in addition to the flatness constraint on the ground. The Dynamic PRM representation or free space was updated incrementally as new sensor data was gathered, with cycles occurring several times a second. In addition to planning paths on a single plane, the robot considered actions to step between two planes if it detected such a transition. The stair transition check determined that the change in height was less than the maximum climbable stair size, and that the transition was short enough to be stepped over. Addition metrics for foot placement, and step transition costs as described in [25] were not used.

During testing it was only occasionally successful. However, the system demonstrated that it could autonomously identify, plan, and execute paths between different planes, without being told that the height discontinuities exist before planning, nor the position, or height of those transitions (see Figure 4.18). It was also noteworthy in that most numerical information, including the difference between plane heights, was determined through sensor data rather than supplied as a prior model. Videos of the system are available in the supplementary materials [21].

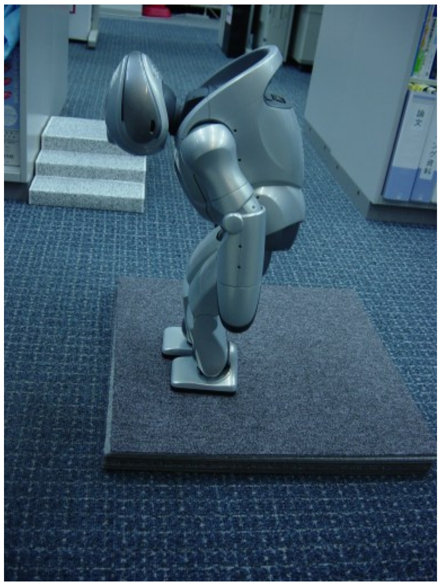

Figure 4.18: QRIO on top of an autonomously reached step obstacle.