Real-Time Randomized Path Planning for Robot Navigation*

James Bruce (jbruce@cs.cmu.edu)
Manuela Veloso (mmv@cs.cmu.edu)

Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15213, USA

Abstract

Mobile robots often find themselves in a situation where they must find a trajectory to another position in their environment, subject to constraints posed by obstacles and the robot's capabilities. This poses the problem of planning a path through a continuous domain. Several approaches have been used to address this problem each with some limitations, including state discretizations, planning efficiency, and lack of interleaved execution. Rapidly-exploring random trees (RRTs) are a recently developed algorithm on which fast continuous domain path planners can be based. In this work, we build a path planning system based on RRTs that interleaves planning and execution, first evaluating it in simulation and then applying it to physical robots. Our algorithm, ERRT (execution extended RRT), introduces two novel extensions of previous RRT work, the waypoint cache and adaptive cost penalty search, which improve replanning efficiency and the quality of generated paths. ERRT is successfully applied to a multi-robot system. Results demonstrate that ERRT is significantly efficient and performs competitively with existing heuristic and reactive real-time path planning approaches. ERRT has shown to offer a major step with great potential for path planning in challenging continuous, highly dynamic domains.

Introduction

The path-planning problem is as old as mobile robots, but is not one that has found a universal solution. Specifically, in complicated, fast evolving environments such as RoboCup [4], currently popular approaches have their strengths, but still leave something to be desired. In particular, most require a state discretization and are best suited for domains with relaxed time constraints for planning. One of the relatively recently developed tools that may help tackle the problem of real-time path planning are Rapidly-exploring random trees (RRTs) [8]. RRTs employ randomization to explore large state spaces efficiently, and can form the basis for a probabilistically complete though non-optimal kinodynamic path planner [9]. 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. In this paper, we report on our work applying and extending this basic RRT algorithm to complex domains that require fast path planning interleaved with execution.

Most current robot systems that have been developed to date are controlled by heuristic or potential field methods at the lowest level, and many extend this upward to the level of path navigation [6]. Since the time to respond must be bounded, *reactive* methods are used to build constant or bounded time heuristics for making progress toward the goal. One reactive method that has proved quite popular is motor schemas [1]. Although they meet the need for action under time constraints, most of these

*This research was sponsored by Grants Nos. DABT63-99-1-0013, F30602-98-2-0135 and F30602-97-2-0250. The information in this publication does not necessarily reflect the position of the funding agencies and no official endorsement should be inferred.

methods suffer from the lack of lookahead, which can lead to highly non-optimal paths and problems with oscillation. This is commonly accepted, and dealt with at a higher layer of the system that detects failure or a local minimum and tries to break out of it. RRTs, as used in our work and presented in this paper, should provide a good compliment for very simple control heuristics, and take much of the complexity out of composing them to form a navigation system. Specifically, local minima can be reduced substantially through lookahead, and rare cases need not be enumerated since the planner has a nonzero probability of finding a solution on its own. Furthermore, an RRT system can be fast enough to satisfy the tight timing requirements needed for fast navigation.

While not as popular as heuristic methods, non-reactive planning methods for interleaved planning and execution have been developed, with promising results. Among these are agent-centered A* search methods [5] and the D* variant of A* search [10]. However, using these planners requires discretization or tiling of the world in order to operate in continuous domains. This leads to a tradeoff between a higher resolution, with is higher memory and time requirements, and a low resolution with non-optimality due to discretization. Most of the features of agent-centered search methods do not rely on A* as a basis, however, so we can achieve many of their benefits using an RRT based planner which fits more naturally into domains with continuous state spaces. The RRT planner we developed is roughly competitive with there other methods in that both can meet tight timing requirements and can reuse information from previous plans, but at this point it does not perform significantly better either. Its strength instead lies mainly as a proof of concept, since the base RRT system is relatively easy to extend to environments with moving obstacles, higher dimensional state spaces, and kinematic constraints. The primary goal of our work was thus to demonstrate the feasibility of an RRT-based algorithm on a real robot. This work appears to be the first successful application of an RRT planner to a real mobile robot [7].

In order to make online planning efficient enough to be practical, two novel additions to the planning algorithm are introduced, specifically the waypoint cache for replanning and adaptive cost penalty search. The second section of this paper defines the basic RRT algorithm. The next section introduces our ERRT contribution. We then describe the implementations for simulated and real robot domains. Specifically, ERRT is applied to a multi-robot domain with small fast moving, remote computer-controlled robots with a single overhead camera providing global sensing. Plans are reconstructed 30 times a second, because in this continuous domain, failure is constant, deviation from a plan occurs, and therefore replanning is needed. By introducing the waypoint cache for replanning, our ERRT approach remembers previous solutions so that a new, similar plan is constructed more effectively. Finally, the last section offers concluding remarks and some lessons learned about working with RRT planners.

RRT Planning

Basic RRT Algorithm

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

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 are 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.

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 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*.

```
function RRTPlan (env:environment,initial:state,
                  goal:state):rrt-tree
    var nearest,extended,target:state;
    var tree:rrt-tree;
    nearest := initial;
    rrt-tree := initial;
    while(Distance (nearest,goal) < threshold)
        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:real;
    p = UniformRandom in [0.0 .. 1.0];
    if 0 < p < GoalProb then
        return goal;
    else if GoalProb < p < 1 then
        return RandomState();

function Nearest (tree:rrt-tree,target:state):state
    var nearest:state;
    nearest := EmptyState;
    foreach state s in current-tree
        if Distance (s,target) <
        Distance (nearest,target) then
            nearest := s;
    return nearest;
```

Table 1: *The basic RRT planner stochastically expands its search tree to the goal or to a random state.*

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.

Extended RRT Algorithm - 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 [2]. 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 2 shows the modifications to the function *ChooseTarget*.

```
function ChooseTarget'(goal:state):state
    var p:real;
    var i:integer;
    p = UniformRandom in [0.0 .. 1.0];
    i = UniformRandom in [0 .. NumWayPoints-1];
    if 0 < p < GoalProb then
        return goal;
    else if GoalProb < p < GoalProb+WayPointProb
    then
        return WayPointCache[i];
    else if GoalProb+WayPointProb < p < 1 then
        return RandomState();
```

Table 2: *The extended RRT planner stochastically expands its search tree to the goal; to a random state; or to a waypoint cache.*

Now, there are 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. The way the extended algorithm progresses is illustrated in Figure 1. Typical values used in this work were $P[goal] = 0.1$ and $P[waypoint] = 0.6$. Another 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)

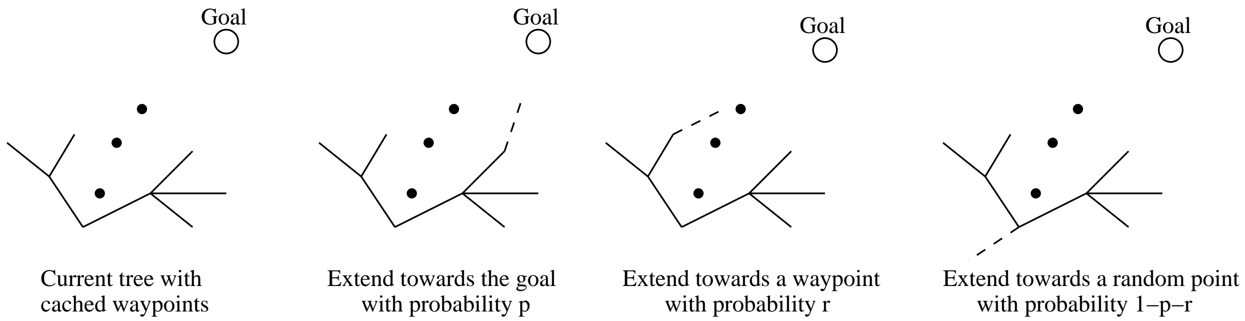

Figure 1: *Extended RRT with a waypoint cache for efficient replanning.*

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.

Domain Implementations and Results

Simulation

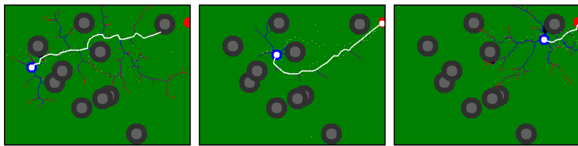

Figure 2: *An example from the simulation-based RRT planner, shown at three times during a run. The initial state (agent) is in blue, while the fixed target state is shown in red. The search tree varies from blue at the beginning of the search to red at the end. The best plan (the one that gets closest to the goal) is shown in white, as are the cached waypoints from previous plans (small white dots).*

We developed an RRT demonstration program some time ago. For this project, to test strategies for interleaved execution and learning it was extended to not only plan but also move the initial state a step along the plan at each iteration. An example run can be seen in figure 2. 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. 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

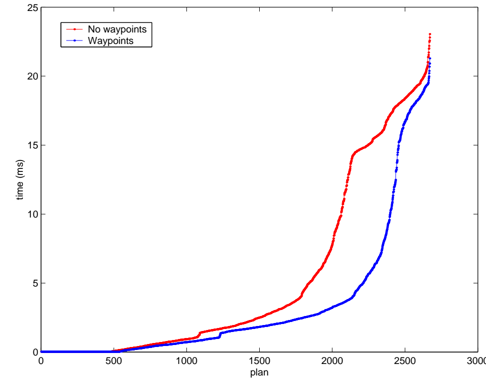

Figure 3: *The effect of waypoints. The lines shows the sorted planning times for 2670 plans with and without waypoints, sorted by time to show the cumulative performance distribution. The red line shows performance without waypoints, while the blue line shows performance with waypoints (Waypoints=50, p[Waypoint]=0.4).*

on performance was examined in an experiment, and the results are shown in in Figure 3. 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 recode the system for a real robot, and to employ a KD-tree to speed up the nearest neighbor lookup step to further improve efficiency.

RoboCup F180 Robot Control

Figure 4: *A robot (lower left) navigating through a field of static obstacles, using an RRT path planner. Color pictures and short movies are available at* ftp://sponge.coral.cs.cmu.edu/pub/movies/F180-RRT/.

For F180 robot control, the system for must take the input from a vision server, 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 is to navigate quickly among other robots, while they also more around executing their own behaviors. As a simplification, we 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. We first implemented the KD-tree, rewrote the path planner (though it ended up a mess due to experimental modifications), and then proceeded to fill in the distance and extension metrics for this domain and robot. We 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. We were disheartened by this, of course, but since it had been shown to work in simulation, we thus went back to the obviously physically incorrect model of no kinematics whatsoever and fixed time step sizes. 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) executing a bad immediately plan is often better than sitting still looking for a better one (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 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 current system are available at the following address: *ftp://sponge.coral.cs.cmu.edu/pub/movies/F180-RRT*

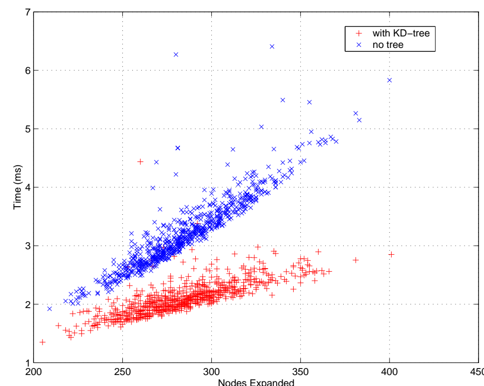

Figure 5: *Planning times vs. number of nodes expanded with and without a KD-tree for nearest neighbor lookup. The KD-tree improves performance, with the gap increasing with the number of nodes due to its expected complexity advantage ($E[O(\log(n))]$ vs. $E[O(n)]$ for the naive version).*

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 5. 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.

Conclusion

In this work a robot control 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 is also the first application of which we are aware using an RRT-based path planner on a real mobile robot.

Several important lessons can be drawn from this work in the context of real-time path planning:

- A heuristic may perform better than a correct 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. Using a heuristic distance metric is not unlike symbolic planners that generate relaxed plans as a heuristic to guide search. Here a heuristic metric was used to guide a local motion control algorithm that actually took steps whose model was an accurate model the actual system.

- A plan generated from an incorrect model requires post-processing for optimal performance. The system worked without post processing, but not nearly as well as when the local metric could apply its accurate model over a longer range of the plan and thus remove most of its inaccuracies over that period of the plan. Future work might explore a system that maintains a kinematic constraint early in the plan, but relaxes it later to make the goal easier to achieve, and reflecting the idea that the specific details of the motion constraints can usually be deferred.

- Pre-existing reactive control methods can easily be adapted to be RRT extension and distance metrics. It can build on these to help eliminate oscillation and local minima through its lookahead mechanism. This system ended up working the best, and was by far the simplest. Since most existing robots already have reactive navigation systems, and the RRT core code is highly generic, We expect this to be a common adaptation in the future.

This work could not have been conducted without the many people in our group who support our RoboCup F180 small robot team. We would specifically like to thank Brett Browning and Mike Bowling, without whom we wouldn't have a team or robots with which interesting research projects could be done.

References

[1] R. C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research, August 1989*, 8(4):92–112, 1989.

[2] A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *submitted to 2002 IEEE International Conference on Robotics and Automation*, 2002.

[3] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/ALife*, 1995.

[4] S. Koenig. Agent-centered search. *Artificial Intelligence*, 2002, in print.

[5] J.-C. Latombe. *Robot Motion Planning*. Kluwer, 1991.

[6] S. M. LaValle. Rapidly-exploring random trees (http://msl.cs.uiuc.edu/rrt/).

[7] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report No. 98-11*, October 1998.

[8] S. M. LaValle and J. James J. Kuffner. Randomized kinodynamic planning. In *International Journal of Robotics Research, Vol. 20, No. 5*, pages 378–400, May 2001.

[9] A. Stentz. Optimal and efficient path planning for unknown and dynamic environments. In *International Journal of Robotics and Automation, Vol. 10, No. 3*, 1995.