An Efficient On-line Path Planner for Outdoor Mobile Robots

Alex Yahja, Sanjiv Singh and Anthony Stentz

Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

Mobile robots operating in outdoor unstructured environments often have only incomplete maps and must deal with new objects found during traversal. Path planning in these environments must be incremental to accommodate new information and must use efficient representations. This article reports recent results in path planning using an efficient data structure (framed-quadtrees) and an optimal algorithm (D*) to incrementally replan optimal paths. We show how the use of framed-quadtrees leads to paths that are shorter and more direct than when other representations are used. We also show the difference in performance when the robot starts with no information about the world versus when it starts with partial information about the world. Our results indicate that, as would be expected, starting with partial information is better than starting with no information. However, in many cases, partial information results in performance that is almost as good as starting out with complete information about the world, while the computational cost incurred is significantly lower. Our system has been tested in simulation as well on an autonomous jeep equipped with local obstacle avoidance capabilities. Results from both simulation and real experimentation are discussed.

Keywords: optimal path planning; framed quadtrees; outdoor mobile robots; unstructured environments

1. Introduction

Path planning for a mobile robot is typically stated as getting from one place to another. The robot must successfully navigate around obstacles, reach its goal, and do so efficiently. Outdoor environments pose special challenges over the structured world that is often found indoors. Not only must a robot avoid colliding with an obstacle such as a rock, it must also avoid falling into a pit or ravine and avoid travel on terrain that would cause it to tip over.

Vast areas have their own associated issues. Such areas typically have large open spaces where a robot might travel freely and are sparsely populated with obstacles. However, the range of obstacles that can interfere with the robot's passage is large- the robot must still avoid a rock as well as go around a mountain. Large areas are unlikely to be mapped at high resolution *a priori* and hence the robot must explore as it goes, incorporating newly discovered information into its database. Hence, the solution must be incremental by necessity.

Another challenge is dealing with a large amount of information and a complex model of the vehicle. Taken as a single problem, so much information must be processed to determine the next action that it is not possible for the robot to perform at any reasonable rate. We deal with this issue by using a layered approach to navigation. That is, we decompose navigation into two levels-

local and global. The job of local planning is to safeguard the robot, avoiding obstacles and reacting to sensory data as quickly as possible [4][6]. A more deliberative process, operating at a coarser resolution of information, is used to decide how best to direct the robot toward the goal. This approach has been used successfully in the past in several systems at Carnegie Mellon [2][4][16].

Approaches to path planning for mobile robots can be broadly classified into two categories: those that use exact representations of the world (e.g. [7][8]), and those that use a discretized representation (e.g. [1][7][9]). The main advantage of discretization is that the computational complexity of path planning can be controlled by adjusting cell size. In contrast, the computational complexity of exact methods is a function of the number of obstacles and/or the number of obstacle facets, which we cannot normally control. Even with discretized worlds, path planning for outdoor environments can be computationally expensive, and on-line performance is typically achieved by use of specialized computing hardware [9]. By comparison the proposed method requires general-purpose computing only. This is made possible by precomputing an optimal path off-line given whatever *a priori* map is available, and then efficiently modifying the path as new map information becomes available, on-line.

Methods that use uniform grid representations must allocate large amounts of memory for regions that may never be traversed or that may not contain any obstacles. Efficiency in map representation can be obtained by the use of quadtrees, but at a cost of optimality. Kambhampati and Davis [5] introduced hierarchical path-searching using quadtrees, but the method produces suboptimal paths and suffers from the fundamental problem of misrepresentation of terrain data in quadtree-nodes. Zelinsky [19] proposed strategies to smooth paths found using quadtrees, which include the use of a quadtree-based visibility graph, but the resulting paths are not optimal to the resolution of the smallest cell. Both methods require complete replanning whenever new information becomes available.

Recently, a new data structure called a framed-quadtree has been suggested as a means to overcome some of the issues related to the use of quadtrees [3]. We have used this data structure to extend an existing path planner that has in the past used uniform (regular) grid cells to represent terrain. This path planner, D* [14][15], has been shown to produce optimal paths in changing environments by incorporating knowledge of the environment as it is incrementally discovered. Coupling the two provides a method that is correct, resolution-complete and resolution-optimal (D* search on framed-quadtrees is performed on the highest-resolution cells of framed-quadtrees). It also does this efficiently. Its paths are always shorter, and in all but the most cluttered environments it executes faster and uses less memory than when regular-grids are used [18]. In general, the sparser or the more unknown the world, the greater the advantage of using framed-quadtrees.

2. Optimal and Incremental Path Planning

Unstructured outdoor environments are often not only sparse, but also have been mapped at a coarse resolution at best. If complete and accurate maps were available, it would be sufficient to use a standard search method such as A* [10] to produce a path. Imperfections in control, inertial sensing, and perception often introduce erroneous and changing information. Thus, a mobile robot must gather new information about the environment and efficiently replan new paths based on this new information. This approach, known as Best Information Planning, produces a path

based on all available information and replans from the current position to the goal when new information becomes available. It is intuitively satisfying and has been shown to produce lower-cost traverses on average than other selected algorithms for unknown and partially-known environments [17]. Also, Best Information Planning is able to make use of prior information to reduce the traversal cost.

It is possible to use A* to replan a new path, but this approach is computationally expensive because it must replan the entire path to the goal every time new information is added [7]. Our approach is to use D*, an algorithm suited for Best Information Planning, that allows replanning to occur *incrementally* and *optimally* in real-time. In other words, in contrast to A*, D* does not require complete replanning of path every time new information comes in. Incremental replanning makes it possible to greatly reduce computational cost, as it only updates the path locally, when possible, to obtain the globally optimal path. D* produces the same results as planning with A* for each new piece of information, but is much faster. The reason for this is that D* adjusts optimal path costs by increasing and lowering the costs only locally and incrementally as needed. D* is the dynamic version of A*, which maintains optimality, unlike other distance/path transform planners [20]. Hence the name D*, or *Dynamic* A*. A detailed description of D* is given in [14][15].

Like A*, D* operates on a cost graph. D* maintains a list of states (or graph nodes) queued for cost expansion (that is, cost recalculation and propagation), initially with the goal state put on the list with a cost of zero. This list is called the open list, as it contains states “open” for expansion. The state with the minimum path cost on the open list is repeatedly expanded, propagating path cost calculations to its neighbors. As the robot moves, it may detect new obstacles or the absence of expected obstacles. When it detects an obstacle, the arc of the path passing through this obstacle is marked with a prohibitively large value and the adjoining states are put on the open list for cost correction. Encountering unexpected obstacles will set off a “raise” wave, a wave of increasing cost, through neighboring states. When this wave meets with states that are able to lower path costs, these “lower” states are put on the open list to recalculate new optimal paths. When it detects the absence of an expected obstacle, the arc of the path passing through this “missing” obstacle is assigned a small cost, denoting an empty space, and the adjoining state is put on the open list as a lower state, setting off a “lower” wave, a wave of decreasing cost. D* is able to determine how far the raise and lower waves need to propagate while providing the optimal path for robot traverse continuously.

3. Efficient Representation of Space

While discretization of space allows for control over the complexity of path planning, it also provides a flexible representation for obstacles and cost maps and eases implementation of search methods. One method of cell decomposition is to tessellate space into equally sized cells, each of which is connected to its eight neighbors. This method, however, has two drawbacks: resulting paths can be suboptimal and memory requirements are high. Quadtrees address the latter problem, while framed-quadtrees address both problems, especially in sparse environments. Below we compare the representations using a simple example.

3.1 Regular-Grids

Regular-grids represent space inefficiently. Natural terrain is usually sparsely populated and is

often not completely known in advance. Many equally sized cells are needed to encode empty areas, making search expensive since a very large number of cells must be searched. Moreover, regular-grids allow only eight angles of direction between cells, resulting in abrupt changes in path direction and an inability, in some cases, to generate straight paths through empty areas (Fig. 1a). It is possible to smooth such jagged paths, but there is no guarantee that the smoothed path will converge to the optimal path.

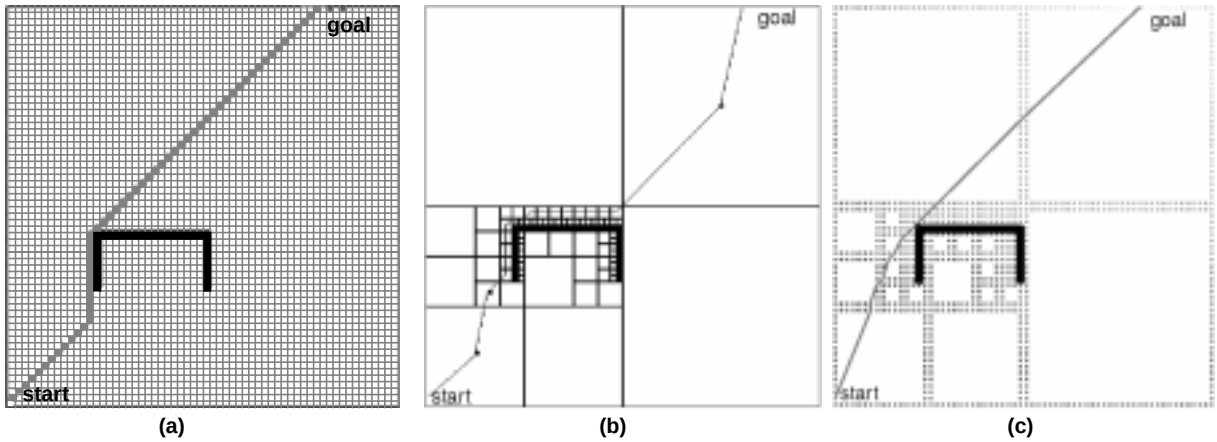


Fig. 1. An example of a path generated using (a) regular-grid representation, (b) quadtree, (c) framed-quadtree.

3.2. Quadtrees

One way to reduce memory requirements is to use a quadtree instead of a regular-grid. Quadtrees [12] are created by recursively subdividing each map square with non-uniform attributes into four equal-sized sub-squares. The process is repeated until a square is uniform or the highest resolution of the map is reached. In the latter case, a highest-resolution cell is marked as an obstacle cell if it is non-uniform. Fig. 2 shows the quadtree subdivision of the map area and the corresponding quadtree data structure. The leaves (i.e., quadtree-nodes without children) are called terminal quadtree-nodes. A quadtree with a single top-level node is called a single-root quadtree. A collection of connected single-root quadtrees forms a multiple-root quadtree or a quadforest.

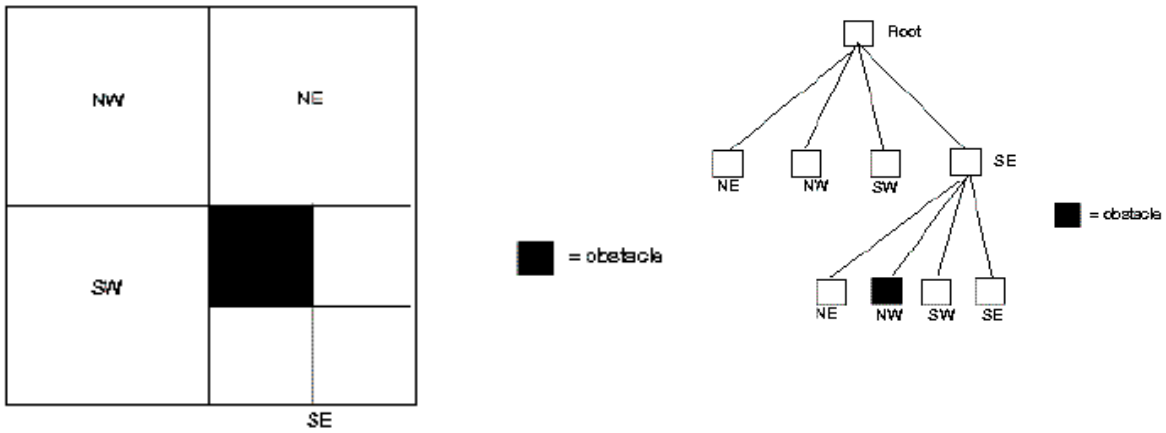


Fig. 2. Quadtree tessellation of a region due to a single obstacle and the corresponding quadtree data structure. The quadrants are labelled NE (Northeast), NW (Northwest), SW (SouthWest), and SE (Southeast), respectively.

Quadtrees allow efficient partitioning of the environment since single cells can be used to encode large empty regions. However, paths generated using quadtrees are suboptimal because they pass through the center of each quadtree-node. There are other strategies to construct and smooth paths for quadtrees [19] so that the paths do not necessarily pass through the center of each quadtree-node, but there is no guarantee that they are optimal to the highest cell-resolution (e.g., see Fig. 12 in Zelinsky's paper [19]). Fig. 1b shows an example path generated using a quadtree.

3.3 Framed-Quadtrees

To remedy the above problem, we have used a modified data structure in which cells of the highest resolution are added around the perimeter of each quadtree region. This augmented representation is called a framed-quadtree [3]. A path generated using this representation is shown in Fig. 1c. The small grey rectangles around each quadtree region are the "border" cells of each framed-quadtree node. The shade of gray of the border-cells signifies the cost to the goal: the lighter the shade, the lower the cost to the goal from that border-cell.

As paths can move through any border-cell via straight lines between border-cells and there are many border-cells inside large framed-quadtree nodes, this representation permits many more discrete angles of direction between cells, instead of just eight angles, as in the case with regular-grids. Most importantly, the paths generated more closely approximate optimal paths. The drawback of framed-quadtrees is that they can require more memory than regular-grids in highly cluttered environments because of the overhead involved in the book-keeping.

4. Implementation of Framed-Quadtree D*

We have developed a system, called Framed-Quadtree D*, implemented by creating a framed-quadtree data structure that uses the D* search algorithm. The structure is created by building a quadtree data structure, connecting each neighboring terminal quadtree-node, adding the border-cells, and connecting each border-cell to its border-cell neighbors.

4.1 Building Framed-Quadtrees

After building the quadtree structure recursively, neighbor pointers to each terminal quadtree-node are assigned. Samet's algorithm finds the neighboring terminal quadtree-nodes by going up and down the quadtree structure guided by the nodes' size, quadrants, and directions [11]. Specifically, two procedures are utilized. One procedure finds quadtree neighbors whose quadtree areas intersect at a common side (known as the side-adjacent neighbors). The other procedure finds quadtree neighbors whose areas intersect at a common corner (known as the corner neighbors).

After allocating a numbered list of border-cells around the perimeter of every terminal quadtree-node, border-cell neighborhood pointers are assigned. If both border-cells are in the same terminal quadtree-node, they are simply implicitly connected. On the other hand, if the border-cells belong to different terminal quadtree-nodes, it is necessary to determine if these quadtree-nodes are adjacent. If they are, we find neighboring border-cells adjacent to every border-cell. Border-cell neighborhood pointers of these border-cells are then allocated to point to each of their neighboring border-cells. In addition to these structural pointers, each border-cell has one specialized pointer, called a backpointer, for use by the D* algorithm.

Fig. 3. illustrates the typical connection patterns of a framed-quadtree (not all connections are shown for an obvious reason):

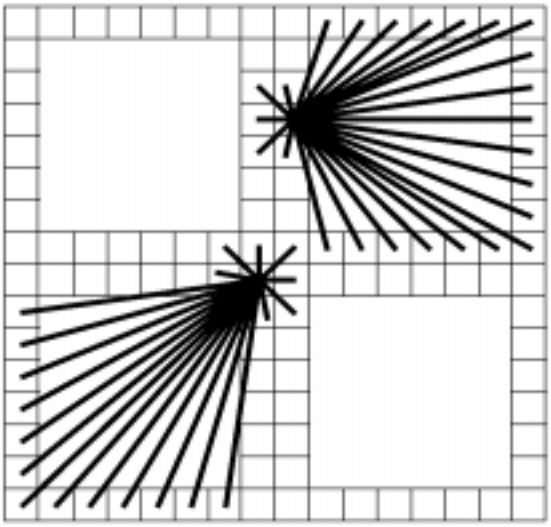


Fig. 3. Typical connection patterns of a framed-quadtree from a *corner* border-cell (left) and a *side* border-cell (right). The number of connections from a border-cell constitutes the branching factor.

4.2 D* with Framed-Quadtrees

D* operates on a cost graph. Framed-quadtree border-cells represent the nodes in the cost graph. As D* has been shown to be optimal on a cost graph [14], it follows that it is also optimal on framed-quadtrees to the resolution of the border-cells.

Each border-cell has a static cost (an obstacle cell has a prohibitive static cost while a free cell has a small static cost of traversing half the cell, i.e., a traverse from the center to the side of the cell) and a path cost corresponding to the cost of traveling to the goal from that border-cell. The link between two border-cells corresponds to an edge in the cost graph.

The cost of traveling through this edge is given by:

edge cost ***ab*** *= (static cost of border-cell* ***a*** *+ static cost of border-cell* ***b****)*

** straight line distance between the centers of* ***a*** *and* ***b****.*

where static cost is defined as the cost of traversing half of the cell size. Fig. 4 gives examples of this edge cost calculation.

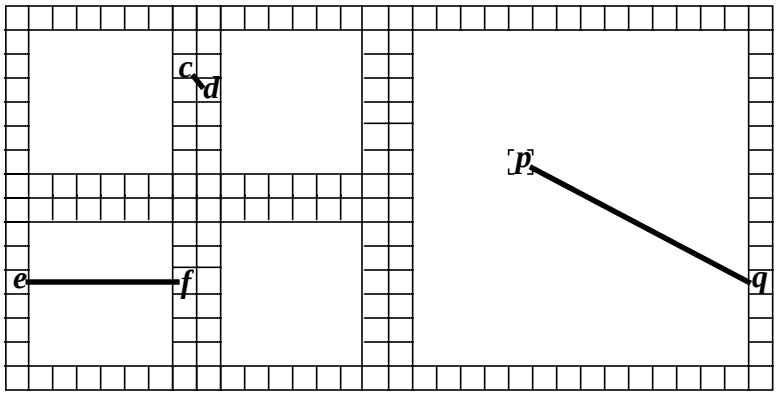


Fig. 4. The edge cost of ***cd*** is *(static cost* ***c*** *+ static cost* ***d****) x squared root of 2*, the edge cost of ef is *(static cost* ***e*** *+ static cost* ***f****) x 7*. Point ***p*** is inside an empty framed-quadtree node, surrounded by border-cells ***q***.

To determine the cost of traveling from point ***p*** inside an empty framed-quadtree area to the goal, the following is employed with the static cost associated with point ***p*** set equal to an empty cell cost and point ***p*** is assumed to fit into an imaginary highest-resolution cell. We have

path cost ***p*** *=* ***min*** *{edge cost* ***pq*** *+ path cost* ***q****} for all border-cells* ***q***

where ***q*** is a border-cell of the framed-quadtree node containing point ***p***.

We assign a cost of 0 to the goal cell and start propagating D* values from the goal cell initially. After this first propagation, D* only needs to visit cells locally as needed, that is, not all border-cells have to be visited in order for D* to recompute the optimal path. As the robot senses its environment, it puts new cells on the open list of D*. D* then calculates the optimal path [14][15]. The results of optimal path recalculation by D* are adjusted path costs for the cells (thus, an adjusted cost graph) and adjusted backpointers. The backpointer of each border-cell points to the neighboring border-cell having the least cost to the goal. Following backpointers from any cell, the optimal path from the cell is recovered.

4.3 Updating Border-Cells based on New Information

As discussed above, if a new obstacle is detected inside an empty quadtree-node, we split that

node into 4 subnodes as illustrated in Fig. 5. However, we need to split the node in a principled way, so that the consistency of D* values is maintained.

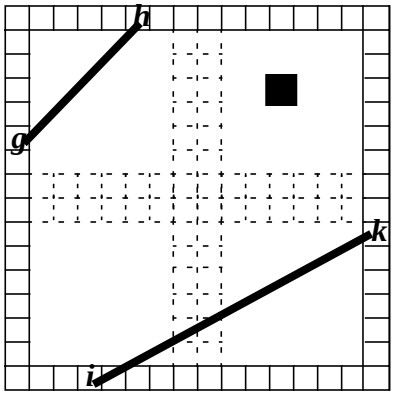


Fig. 5. As the splitting of a framed-quadtree node occurs (the dotted rectangles show the first step of splitting), some links like edge ***gh*** are not affected, while others like edge ***ik*** are affected. Border-cells at the ends of the affected links may need their path costs recomputed. The black rectangle indicates the obstacle causing the split.

The following is the procedure:

1. For each border-cell pair ***i***, ***j*** in a quadtree-node that is going to be split, do:
   - •Check if the neighborhood pointers between ***i*** and ***j*** are affected by the split. If they are not, then do nothing, otherwise proceed to the next step.
   - •Label the link ***ij*** "to be deleted".
   - •Place both ***i*** and ***j*** on the OPEN list with their current path costs.
2. *D* will pop up cells from the OPEN list for recomputation of path costs (also known as expansion). When expanding a cell **x**, if a cell **y** has a backpointer to **x** through a "to be deleted" link, place **y** back on the OPEN list with its path cost set equal to the maximum cost MAXCOST and set **y**'s backpointer to NULL.*
3. *Delete all links connected to cell **x** that are to be deleted.*

The splitting of a quadtree-node also causes the modification of framed-quadtree structures by re-executing the framed-quadtree build-up procedures incrementally and locally. In this way, the framed-quadtree operation mimics the local and incremental nature of D*, allowing efficient computation. The proposed approach is locally adaptable, that is, it changes the framed-quadtree structures and applies the D* algorithm locally in response to some local environment change, while maintaining global path optimality.

5. Graph Complexity

As stated, border-cells form the nodes in the D* cost graph. Let *k* be the maximum number of border-cells along one dimension. For an empty world of *k x k* cells, the number of border-cells is *4 x (k-1)* cells. The number of links between border-cells is given by (the number of *side* border-cells x the number of links from a *side* border-cell) + (the number of *corner* border-cells x the

number of links from a *corner* border-cell) divided by 2. Fig. 3 earlier displays the typical pattern of links from a *corner* border-cell and a *side* border-cell.

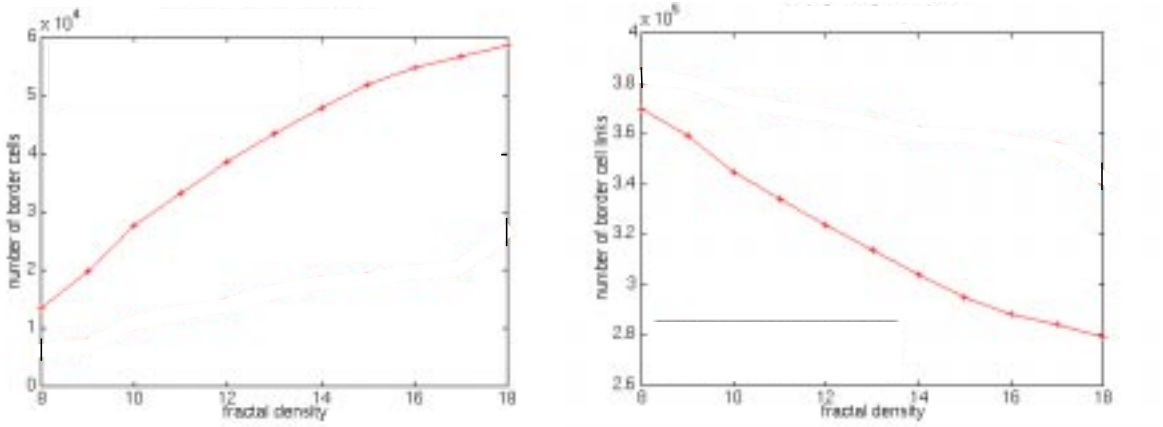
The number of side border-cells is 4(k-2), while the number of corner border-cells is 4. The number of links from a side border-cell is given by the links to (the number of side border-cells on three opposing sides + 2 opposing corner border-cells)+(2 adjacent border-cells)+(3 border-cells on the adjacent quadtree-node), which is $3(k-2)+2+2+3$. The number of links from a corner border-cell is given by the links to (the number of side border-cells on two opposing sides + 1 opposing side corner border-cell) + (2 adjacent border-cells) + (5 border-cells on the adjacent three quadtree-nodes), which is $2(k-2)+1+2+5$. Thus the number of links is given by the formula $(4(k-2)(3(k-2)+2+2+3) + 4(2(k-2)+1+2+5))/2 = 2(3k^2 - 3k +2)$.

If this empty world is subdivided into 4 equally sized quadtree areas, the number of border-cells becomes $4 \times (4 \times (k/2 - 1))$, which is an increase. The number of links, however, is $4 \times 2 \times (3 \times (k/2)^2 - 3 \times (k/2) + 2) = 2(3k^2 - 6k +8)$, which is a decrease if $k>2$ and is the same if $k=2$. For a world of $k \times k$ cells with fully-instantiated framed-quadtrees, the number of border-cells is k^2, and the number of links is $8k^2/2$. Thus, along the subdivision process, the number of border-cells will increase from $4k-4$ to k^2 cells, while the number of links will decrease from $(6k^2 - 6k +4)$ to $4k^2$. For k=256, the number of border-cells for an empty 256 x 256 cell world is 1,020, while the number of links is 391,684. For a 256 x 256 cell world with fully-instantiated framed-quadtrees, the number of border-cells is 65,536, while the number of links is 262,144.

Let n be the number of nodes in the graph (that is, $n = k \times k$) and b be the branching factor (which is upper-bounded by $4k-4$ for framed-quadtrees). Note that the worst case complexity of D* path cost evaluation is $O(b\ n\ log_2(n))$.

For a world with a certain cell-size and obstacle-distribution, we could estimate the optimal size of the root node of framed-quadtrees and decide if we should use multiple roots.

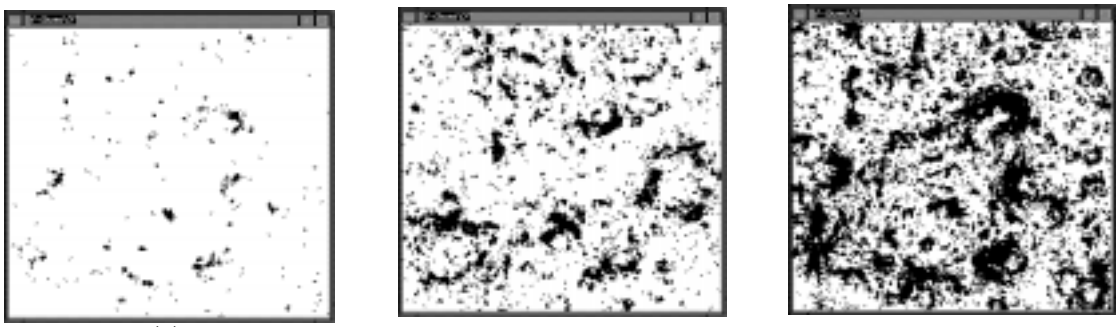
Fig. 6 shows the trends for border-cell and link count in between an empty and a fully-instantiated framed-quadtrees, respectively, for a 256 x 256 cell fractal binary-world. The number of obstacles, thus the degree of framed-quadtree instantiation, grows with fractal density. Each data point represents an average of 20 randomly generated terrains.



(a) (b)

Fig. 6. As the world becomes denser: (a) Border-cell count increases, the count is in between the empty framed-quadtree (1,020 cells) and the fully-instantiated framed-quadtree (65,536 cells) cases. (b) Border-cell's link count decreases, the count is in between the empty framed-quadtree (391,684 links) and the fully-instantiated framed-quadtree (262,144 links) cases.

As illustrated, more border-cells are created while fewer links are needed as the world becomes denser. The fractal densities are illustrated in Fig. 7.



(a) (b) (c)

Fig. 7. Examples of simulated fractal terrain of various fractal densities: (a) 8, (b) 13, (c) 18.

6. Simulation Results

We have conducted path planning experiments using simulated fractal terrains of varying complexity. The simulation environment is a 256 x 256 cell world with obstacles (each a 1 x 1

cell) distributed by a fractal terrain generator. The amount of clutter in the world is parameterized by a fractal gain.

We use two representations for obstacles. The first is a simplified representation in which the terrain has a binary cost distributed by a fractal generator. Either the terrain is passable, in which case the cost to move from one cell to another is the Euclidean distance, or, the terrain is impassable and the cost to move to a cell containing an obstacle is infinite. The second is a more realistic method that encodes the cost of moving from one cell to another as a function of a fractal, resulting in a continuous-cost map. In this case, the fractal generator directly produces the cost map; that is, it directly produces the cost of traversing from one cell to another. This cost map can be thought of as a map of terrain slopes. Large slopes are difficult to traverse so they have high costs, while smaller ones are easier so they have lower costs. We assume a point robot with a perfect 8-cell circular sensor.

Below we show simulation results that show the benefit of using framed-quadtrees over regular-grids and the benefit of using partial map information.

6.1 Binary-Cost Worlds

An extensive set of simulations were conducted using for binary-cost worlds [18], comparing the use of regular-grids and framed-quadtrees for D*, in which framed-quadtrees were shown to have memory and path length advantages over regular-grids. As an illustration of the results, Fig. 8 shows the difference in the traverse length when framed-quadtrees are used as opposed to regular-grids. The path created using framed-quadtrees (right) is shorter and smoother than the one using regular-grids (left). For sparse environments like this one, Framed-Quadtree D* is also faster and uses less memory.

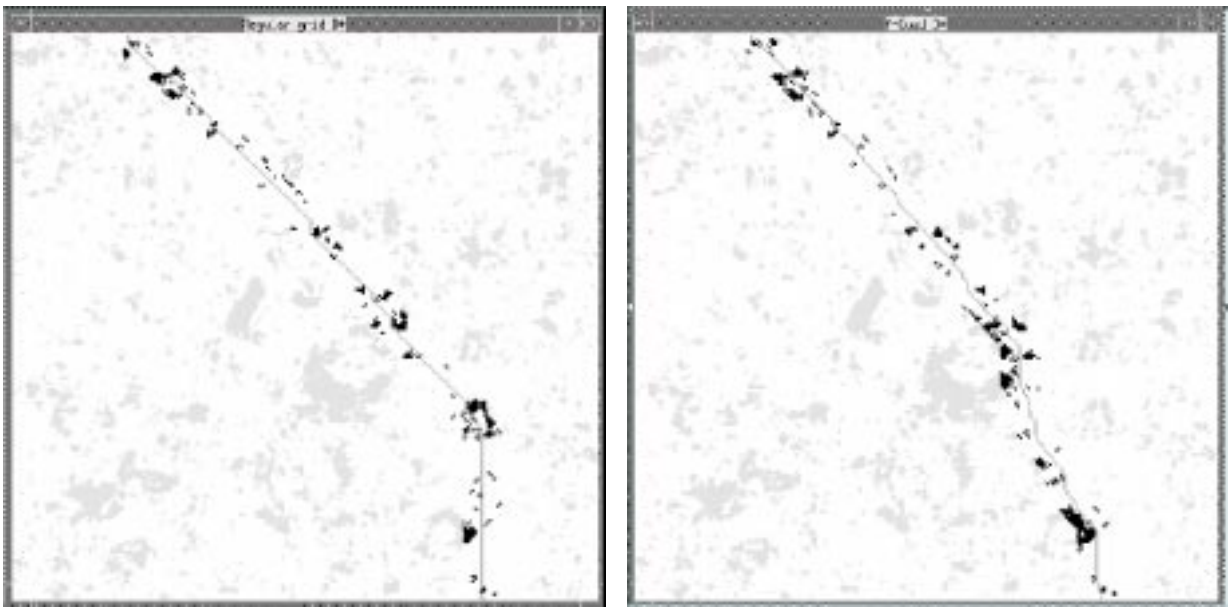


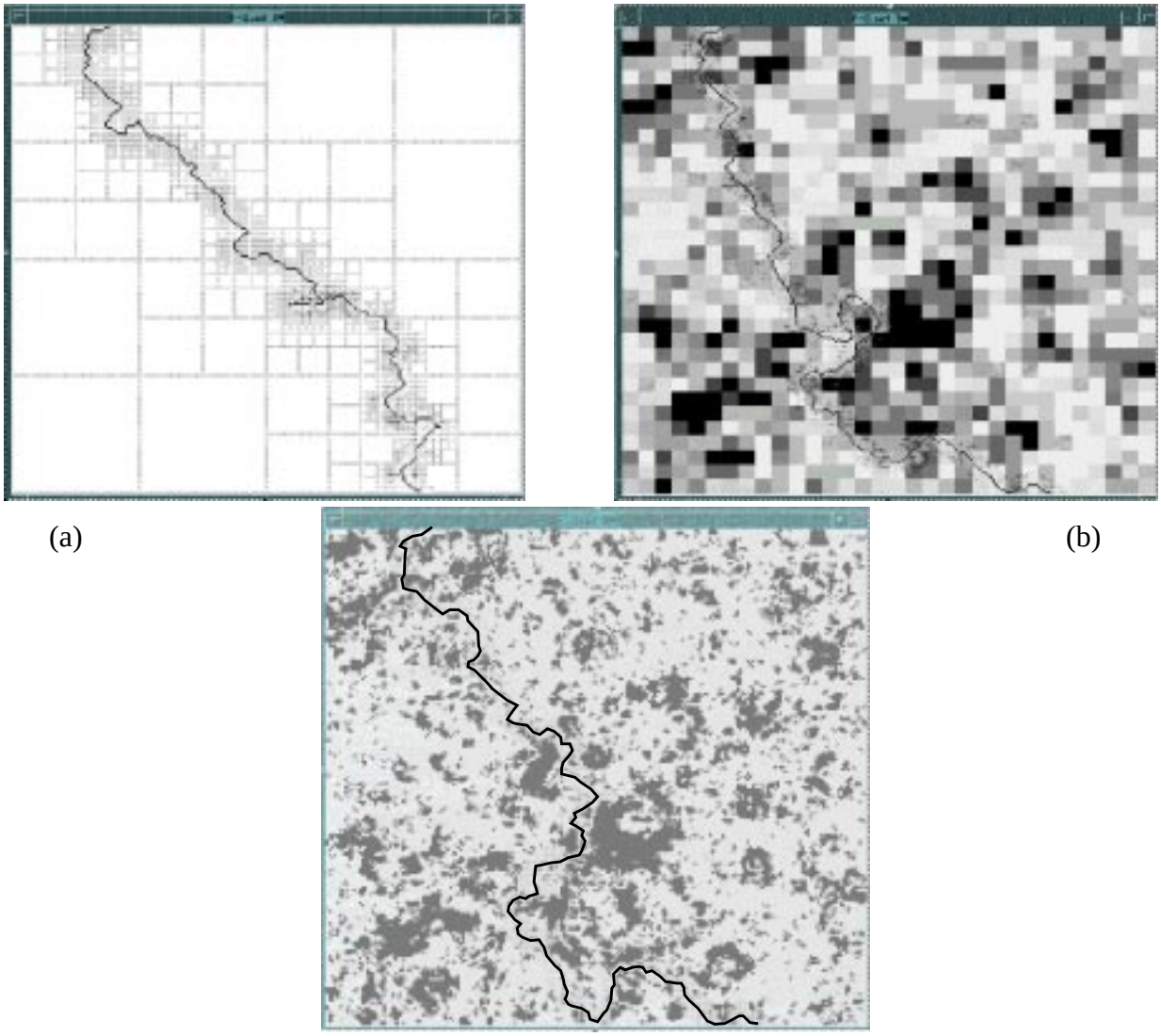
Fig. 8. Traverses generated in a binary fractal world using regular-grids (left) and framed-quadtrees (right). The lighter cells represent occupied areas that are unknown in advance. The dark cells represent the obstacles that are discovered by the robot's sensors. The traverse

generated when framed-quadtrees are used is shorter and more natural.

6.2 Continuous-Cost Worlds

For continuous cost worlds, we examine three cases: completely unknown, partially known, and fully known continuous cost worlds. In completely unknown worlds, we do not have any map information before the robot starts moving. In contrast, in fully known worlds, we have all map information *a priori*. In some cases, however, the terrain that the robot must traverse is known at a low resolution such as would be produced by an aerial flyover. A low resolution map (coarse map) contains partial information about the world.

For example, Fig. 9 shows traverses in a world of which the robot (a) has no knowledge before it starts, (b) has a coarse map, which each cell's cost corresponding to the average of an 8 x 8 area, before it starts, and finally (c) has a complete map *a priori*. Fig. 9a also shows the structure of framed-quadtrees generated by the end of the traverse. As shown, the path found when a coarse map is available resembles the best possible traverse that can be followed for the fully known world. As we show below, the cost of generating such traverses is less than the cost incurred when a full map is available.



(a) (b)

(c)

Fig. 9. A traverse in a continuous-cost fractal world: (a) completely unknown, (b) partially known (coarse information), and (c) fully known world.

We present results of 3000 trials below, comparing traversal cost, memory usage, and execution time. Terrain density is parameterized in ten steps of fractal gain. For each step in terrain density, we conducted 100 runs for each of the three cases (completely unknown, partially known and totally known).

6.2.1 Traversal Cost

Note that traversal cost is not exactly the same as the traversal length. Hence a meandering path might be picked even if a direct path is available, because the direct path happens to have a high cost path. Fig. 10 shows traverses in the fully known world have the lowest cost. Conversely, traverses in a completely unknown world have the highest cost because the robot often enters high cost areas and in some cases goes down dead ends before it backtracks. Traverses in the partially

known world are in between the two curves, but it is interesting to note that even with 1/64th as much *a priori* information as in the fully known case, the traversal cost is not significantly higher.

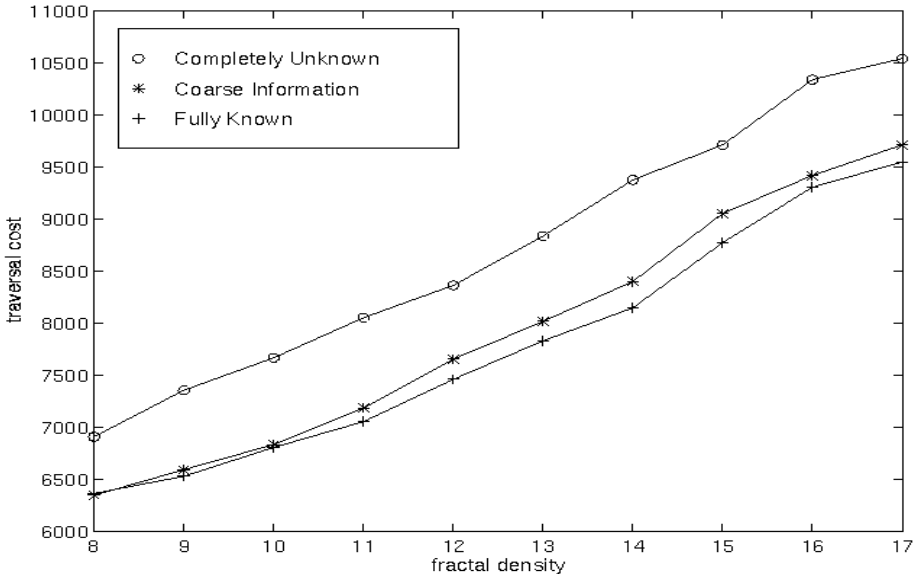


Fig. 10. Traversal cost comparison.

6.2.2 Memory Usage

Fig. 11 shows that encoding the fully known world uses the most memory, while the completely unknown world uses the least. The coarsely known world is in between. As expected, the more information, the more memory is required by a framed-quadtree to represent that information.

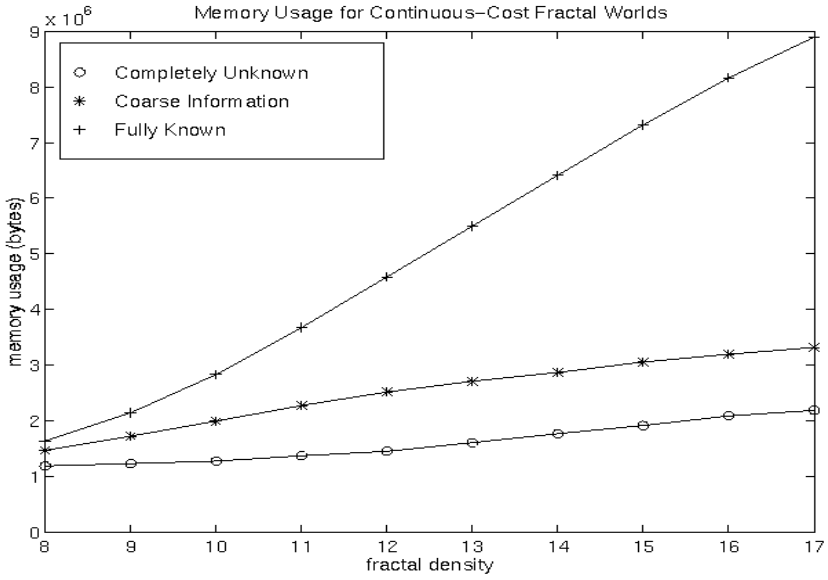


Fig. 11. Memory usage comparison.

6.2.3 Execution Time

Traverses in the fully known world execute faster for sparse worlds, but execute slower for denser worlds than the completely unknown world, due to the time needed to build up framed-quadtree structures and to propagate initial D* values. However, traverses in the coarsely known world execute faster than both, as fewer data structures are necessary than in the case of the fully known world, and its traverse is less likely to stumble into high cost areas than in the case of the completely unknown world. This is depicted in Fig. 12.

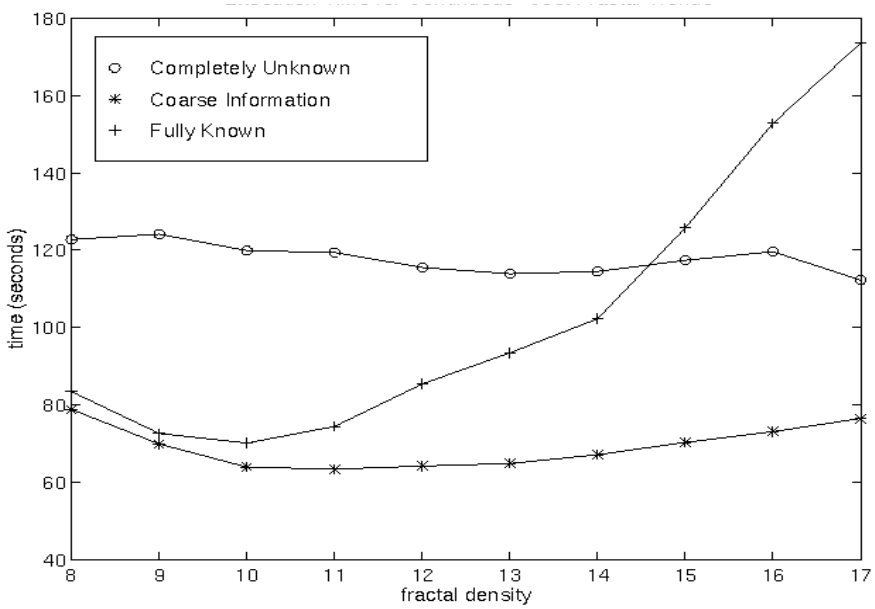


Fig. 12. Execution time comparison.

7.Test Results on an Autonomous Vehicle

Since our system is targeted for implementation on a car-like vehicle (with nonholonomic and kinematic constraints), it is necessary to determine realistic commands that can be issued to the vehicle. At each control step, we hypothesize a small set (approximately 20) of arcs that the vehicle can execute in the next interval. The approach fans out steering arc curves over cells, querying the cells' path costs, adding them up, and selecting the arc that has the lowest total cost. This sum corresponds to the best arc that reduces the traveling cost to the goal. Fig. 13 shows an autonomous vehicle navigating in the presence of obstacles. The arc that avoids the obstacles and gets the vehicles closest to the goal is chosen.

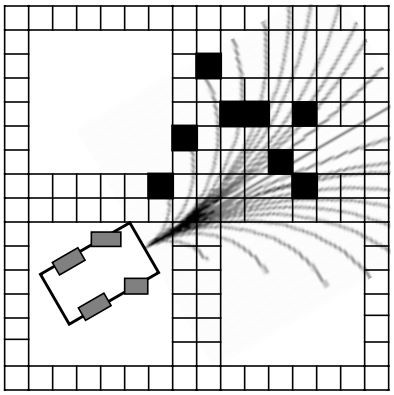


Fig. 13. The arc command to be issued to the vehicle is determined by querying path costs of cells lying along each arc. Black rectangles denote obstacle cells.

We have performed several tests on an automated HMMWV (Fig. 14). Our vehicle uses a vertical-baseline stereo system to generate range images [13]. The resulting images are processed by the SMARTY local navigator [4], which handles local obstacle detection and avoidance. This obstacle map is fed to a global navigator running a path planning algorithm, such as Framed-Quadtree D*. Both the local and global navigators submit steering advice to an arbiter, which selects a steering command each time interval and passes it to the controller [16]. Fig. 15 shows the system modules and data flow.

Fig. 14. The autonomous vehicle (HMMWV) used for our experiments. The vehicle is equipped with stereo vision, inertial guidance and GPS positioning.

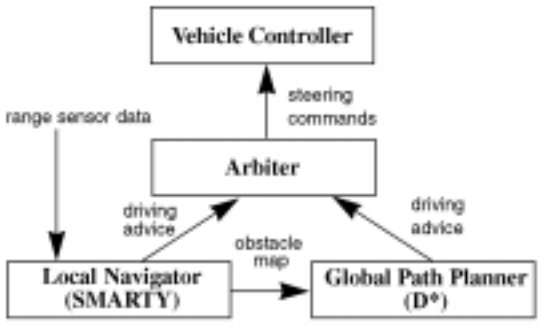


Fig. 15. Data flow in the implemented system.

Fig. 16 shows a successful traverse of the vehicle that covered 200 meters in 6 minutes. During this traverse, the vehicle detected and avoided 80 obstacles.

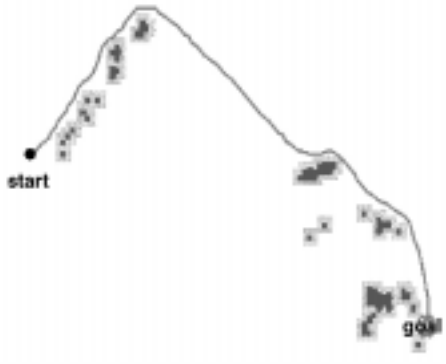


Fig. 16. Successful long traverse of the vehicle using Framed-Quadtree D* through a terrain with obstacles to the goal. The dark rectangles are obstacles detected and avoided during the traverse. The shaded areas surrounding the dark obstacles are potentially dangerous zones.

Fig. 17 shows a close-up of the data structure produced after the above run. As expected, a large part of the environment not explored is represented by a small number of cells.

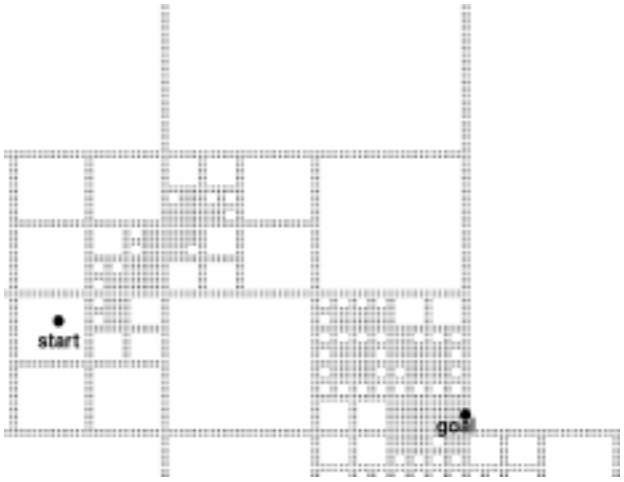


Fig. 17. A close up of the data structure produced during the traverse of Fig. 16.

8. Conclusion

Appropriate map representation allows us to plan paths more efficiently in large unstructured environments than a regular-grid representation. Combining an optimal path planning algorithm with framed-quadtree map representation has the benefit of optimal path planning while minimizing the memory requirements. The power of this combination comes from two aspects: incrementally discovered information is used to modify paths such that optimality is preserved, and an efficient data structure is used to represent terrain topology. Our experiments verify that efficiency is highest in sparsely populated environments.

We have extended our method for global path planning suited to autonomous robots operating in large unstructured environments. Our previous method combined the D* algorithm with a binary framed-quadtree data structure. While a binary world is acceptable for robots operating indoors, it is desirable to use a representation that encodes the three dimensional nature of outdoor terrain. Our extended method can handle continuous-cost maps that are more representative of natural terrains. Current rover path planners do not take into account any *a priori* and/or coarse map information. Our method would allow this be done efficiently and optimally.

The results from extensive simulation in continuous-cost worlds shows that coarse terrain information, while not only practical, can also result in reduced execution times while incurring only a small cost in optimality over the perfectly known world. In terms of memory usage, there is also a persuasive argument to use coarse information about the world. As a comparison, note that the memory requirements remain constant, irrespective of the number of the objects in the world when a regular-grid is used. Hence this method lends itself to applications such as planetary rovers [21], which will have severe limitations in memory, but nevertheless require relatively high performance.

Several extensions can be made to the system to further enhance its capability and

performance:

- Adapting cell sizes. Large framed-quadtree nodes result in a large number of border cells, and thus increase the branching factor of the graph. This effect can be mitigated by assigning smaller root framed-quadtree nodes and larger border cells, adapting their sizes to changes in distance and in the environment. An optimal root size can be determined as a function of map extent, expected clutter and cell size.
- Summarizing cells that have been traversed and are far away from the robot by collapsing several framed-quadtree nodes will allow further reduction in memory requirements.
- Throwing away nodes and summarizing only obstacle and/or cost-cell information for nodes long traversed and far away from the robot would allow Framed-Quadtree D* to have an extremely small memory requirement. If the need ever arises, the framed-quadtrees can be built from the summarized obstacle and/or cost-cell information.
- Smart allocation and sizing of quadtree-nodes and border-cells by taking into consideration factors such as the distance from the areas covered by quadtree-nodes to the robot, the importance of the areas, and the reliability of the map information about the areas.
- Extension to handle moving objects. Moving objects, given that their speed is within the computational speed bound, can be handled by Framed-Quadtree D* by dividing and summarizing framed-quadtree nodes as the obstacle moves along.
- Extensions to 3D in the form of Framed-Octree D* would allow its use in 3D motion planning applications.

References

[1] J. Barraquand and J.-C. Latombe, Robot motion planning: a distributed representation approach, in: International Journal of Robotics Research 10(6), 628-649.

[2] B. Brumitt and A. Stentz, Dynamic mission planning for multiple mobile robots, in: Proc. IEEE International Conference on Robotics and Automation (May 1996).

[3] D. Z. Chen, et. al., Planning conditional shortest paths through an unknown environment: a framed-quadtree approach, in: Proc. 1995 IEEE/RJS International Conference on Intelligent Robots and Systems, IROS 95, Vol. 3 (August 1995) 33-38.

[4] M. Hebert, C. Thorpe, and A. Stentz, eds., Intelligent Unmanned Ground Vehicles: Autonomous Navigation Research at Carnegie Mellon (Kluwer Academic Publishers, Dordrecht, 1997).

[5] S. Kambhampati and L.S. Davis, Multiresolution path planning for mobile robot, in: IEEE Journal of Robotics and Automation, Vol. RA-2, No. 3 (September 1985) 135-145.

[6] A. Kelly, An intelligent predictive control approach to the high speed cross country autonomous navigation problem, Ph.D. Thesis, Robotics Institute, Carnegie Mellon University, 1995.

[7] J.-C. Latombe, Robot Motion Planning (Kluwer Academic Publishers, Boston, MA.,1991).

[8] T. Lozano-Perez and M.A. Wesley, An algorithm for planning collision-free paths among polyhedral obstacles, in: Communications of the ACM, Vol. 22, No. 10 (October 1979), 560-570.

[9] J. Lengyel, et. al., Real time robot motion planning using rasterizing computer graphics hardware, in: Proc. SIGGRAPH (1990).

[10] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach (Prentice Hall, Singapore, 1995).

[11] H. Samet, Neighbor finding techniques for images represented by quadtrees, in: Computer Graphics and Image Processing 18 (1982), 37-57.

[12] H. Samet, An overview of quadtrees, octrees, and related hierarchical data structures, in: NATO ASI Series, Vol. F40 (1988), 51-68.

[13] S. Singh and B. Digney, Performance improvements for autonomous cross country navigation using stereo vision, Technical Report, Robotics Institute, Carnegie Mellon University, CMU-RI-TR-99-03 (January 1999).

[14] A. Stentz, Optimal and efficient path planning for partially-known environments, in: Proc. IEEE International Conference on Robotics and Automation (May 1994).

[15] A. Stentz, The Focussed D* algorithm for real-time replanning, in: Proc. the International Joint Conference on Artificial Intelligence (August 1995).

[16] A. Stentz and M. Hebert, A complete navigation system for goal acquisition in unknown environments, in: Autonomous Robots 2(2) (1995).

[17] A. Stentz, Best Information Planning for unknown, uncertain, and changing domains, in: AAAI-97 Workshop on On-line-Search (1997).

[18] A. Yahja, A. Stentz, S. Singh and B. Brumitt, Framed-quadtree path planning for mobile robots operating in sparse environments, in: Proc. IEEE Conference on Robotics and Automation, Leuven, Belgium (May 1998).

[19] A. Zelinsky, A mobile robot exploration algorithm, in: IEEE Trans. on Robotics and Automation, Vol. 8, No.6 (December 1992) 707-717.

[20] A. Zelinsky, Using path transforms to guide the search for findpath in 2D, in: International Journal of Robotics Research, Vol. 13, No. 4 (August 1994) 315-325.

[21] ***http://www.frc.ri.cmu.edu/projects/mars*** or ***http://www.ri.cmu.edu/projects/project_319.html***