Optimal and Efficient Path Planning for Partially-Known Environments

Anthony Stentz

The Robotics Institute; Carnegie Mellon University; Pittsburgh, PA 15213

Abstract

The task of planning trajectories for a mobile robot has received considerable attention in the research literature. Most of the work assumes the robot has a complete and accurate model of its environment before it begins to move; less attention has been paid to the problem of partially known environments. This situation occurs for an exploratory robot or one that must move to a goal location without the benefit of a floorplan or terrain map. Existing approaches plan an initial path based on known information and then modify the plan locally or replan the entire path as the robot discovers obstacles with its sensors, sacrificing optimality or computational efficiency respectively. This paper introduces a new algorithm, D, capable of planning paths in unknown, partially known, and changing environments in an efficient, optimal, and complete manner.*

1.0 Introduction

The research literature has addressed extensively the motion planning problem for one or more robots moving through a field of obstacles to a goal. Most of this work assumes that the environment is completely known before the robot begins its traverse (see Latombe [4] for a good survey). The optimal algorithms in this literature search a state space (e.g., visibility graph, grid cells) using the distance transform [2] or heuristics [8] to find the lowest cost path from the robot's start state to the goal state. Cost can be defined to be distance travelled, energy expended, time exposed to danger, etc.

Unfortunately, the robot may have partial or no information about the environment before it begins its traverse but is equipped with a sensor that is capable of measuring the environment as it moves. One approach to path planning in this scenario is to generate a "global" path using the known information and then attempt to "locally" circumvent obstacles on the route detected by the sensors [1]. If the route is completely obstructed, a new global path is planned. Lumelsky [7] initially assumes the environment to be devoid of obstacles and moves the robot directly toward the goal. If an obstacle obstructs the path, the robot moves around the perimeter until the point on the obstacle nearest the goal is found. The robot then proceeds to move directly toward the goal again. Pirzadeh [9] adopts a strategy whereby the robot wanders about the environment until it discovers the goal. The robot repeatedly moves to the adjacent location with lowest cost and increments the cost of a location each time it visits it to penalize later traverses of the same space. Korf [3] uses initial map information to estimate the cost to the goal for each state and efficiently updates it with backtracking costs as the robot moves through the environment.

While these approaches are complete, they are also suboptimal in the sense that they do not generate the lowest cost path given the sensor information as it is acquired and assuming all known, a priori information is correct. It is possible to generate optimal behavior by computing an optimal path from the known map information, moving the robot along the path until either it reaches the goal or its sensors detect a discrepancy between the map and the environment, updating the map, and then replanning a new optimal path from the robot's current location to the goal. Although this brute-force, replanning approach is optimal, it can be grossly inefficient, particularly in expansive environments where the goal is far away and little map information exists. Zelinsky [15] increases efficiency by using a quad-tree [13] to represent free and obstacle space, thus reducing the number of states to search in the planning space. For natural terrain, however, the map can encode robot traversability at each location ranging over a continuum, thus rendering quad-trees inappropriate or suboptimal.

This paper presents a new algorithm for generating optimal paths for a robot operating with a sensor and a map of the environment. The map can be complete, empty, or contain partial information about the environment. For

1050-4729/94 $03.00

regions of the environment that are unknown, the map may contain approximate information, stochastic models for occupancy, or even a heuristic estimates. The algorithm is functionally equivalent to the brute-force, optimal replanner, but it is far more efficient.

The algorithm is formulated in terms of an optimal find-path problem within a directed graph, where the arcs are labelled with cost values that can range over a continuum. The robot's sensor is able to measure arc costs in the vicinity of the robot, and the known and estimated arc values comprise the map. Thus, the algorithm can be used for any planning representation, including visibility graphs [5] and grid cell structures. The paper describes the algorithm, illustrates its operation, presents informal proofs of its soundness, optimality, and completeness, and then concludes with an empirical comparison of the algorithm to the optimal replanner.

2.0 The D* Algorithm

The name of the algorithm, D*, was chosen because it resembles A* [8], except that it is *dynamic* in the sense that arc cost parameters can change during the problem-solving process. Provided that robot motion is properly coupled to the algorithm, D* generates optimal trajectories. This section begins with the definitions and notation used in the algorithm, presents the D* algorithm, and closes with an illustration of its operation.

2.1 Definitions

The objective of a path planner is to move the robot from some location in the world to a goal location, such that it avoids all obstacles and minimizes a positive cost metric (e.g., length of the traverse). The problem space can be formulated as a set of *states* denoting robot locations connected by *directional arcs*, each of which has an associated cost. The robot starts at a particular state and moves across arcs (incurring the cost of traversal) to other states until it reaches the *goal* state, denoted by G. Every state X except G has a *backpointer* to a next state Y denoted by $b(X) = Y$. D* uses backpointers to represent paths to the goal. The cost of traversing an arc from state Y to state X is a positive number given by the *arc cost* function $c(X, Y)$. If Y does not have an arc to X, then $c(X, Y)$ is undefined. Two states X and Y are *neighbors* in the space if $c(X, Y)$ or $c(Y, X)$ is defined.

Like A*, D* maintains an *OPEN* list of states. The *OPEN* list is used to propagate information about changes to the arc cost function and to calculate path costs to states in the space. Every state X has an associated *tag* $t(X)$, such that $t(X) = NEW$ if X has never been on the *OPEN* list, $t(X) = OPEN$ if X is currently on the *OPEN* list, and $t(X) = CLOSED$ if X is no longer on the *OPEN* list. For each state X, D* maintains an estimate of the sum of the arc costs from X to G given by the *path cost* function $h(G, X)$. Given the proper conditions, this estimate is equivalent to the optimal (minimal) cost from state X to G, given by the implicit function $o(G, X)$. For each state X on the *OPEN* list (i.e., $t(X) = OPEN$), the *key* function, $k(G, X)$, is defined to be equal to the minimum of $h(G, X)$ before modification and all values assumed by $h(G, X)$ since X was placed on the *OPEN* list. The key function classifies a state X on the *OPEN* list into one of two types: a *RAISE* state if $k(G, X) < h(G, X)$, and a *LOWER* state if $k(G, X) = h(G, X)$. D* uses *RAISE* states on the *OPEN* list to propagate information about path cost increases (e.g., due to an increased arc cost) and *LOWER* states to propagate information about path cost reductions (e.g., due to a reduced arc cost or new path to the goal). The propagation takes place through the repeated removal of states from the *OPEN* list. Each time a state is removed from the list, it is *expanded* to pass cost changes to its neighbors. These neighbors are in turn placed on the *OPEN* list to continue the process.

States on the *OPEN* list are sorted by their key function value. The parameter k_{min} is defined to be $min(k(X))$ for all X such that $t(X) = OPEN$. The parameter k_{min} represents an important threshold in D*: path costs less than or equal to k_{min} are optimal, and those greater than k_{min} may not be optimal. The parameter k_{old} is defined to be equal to k_{min} prior to most recent removal of a state from the *OPEN* list. If no states have been removed, k_{old} is undefined.

An ordering of states denoted by $\{X_1, X_N\}$ is defined to be a *sequence* if $b(X_{i+1}) = X_i$ for all i such that $1 \le i < N$ and $X_i \ne X_j$ for all (i,j) such that $1 \le i < j \le N$. Thus, a sequence defines a path of backpointers from X_N to X_1. A sequence $\{X_1, X_N\}$ is defined to be *monotonic* if $(t(X_i) = CLOSED$ and $h(G, X_i) < h(G, X_{i+1}))$ or $(t(X_i) = OPEN$ and $k(G, X_i) < h(G, X_{i+1}))$ for all i such that $1 \le i < N$. D* constructs and maintains a monotonic sequence $\{G, X\}$, representing decreasing current or lower-bounded path costs, for each state X that is or was on the *OPEN* list. Given a sequence of states $\{X_1, X_N\}$, state X_i is an *ancestor* of state X_j if $1 \le i < j \le N$ and a *descendant* of X_j if $1 \le j < i \le N$.

For all two-state functions involving the goal state, the following shorthand notation is used: $f(X) \equiv f(G, X)$. Likewise, for sequences the notation $\{X\} \equiv \{G, X\}$ is used. The notation $f(\cdot)$ is used to refer to a function independent of its domain.

2.2 Algorithm Description

The D* algorithm consists primarily of two functions: *PROCESS – STATE* and *MODIFY – COST*. *PROCESS – STATE* is used to compute optimal path costs to the goal, and *MODIFY – COST* is used to change the arc cost function $c(\cdot)$ and enter affected states on the *OPEN* list. Initially, $t(\cdot)$ is set to *NEW* for all states, $h(G)$ is set to zero, and G is placed on the *OPEN* list. The first function, *PROCESS – STATE*, is repeatedly called until the robot's state, X, is removed from the *OPEN* list (i.e., $t(X) = CLOSED$) or a value of -1 is returned, at which point either the sequence $\{X\}$ has been computed or does not exist respectively. The robot then proceeds to follow the backpointers in the sequence $\{X\}$ until it either reaches the goal or discovers an error in the arc cost function $c(\cdot)$ (e.g., due to a detected obstacle). The second function, *MODIFY – COST*, is immediately called to correct $c(\cdot)$ and place affected states on the *OPEN* list. Let Y be the robot's state at which it discovers an error in $c(\cdot)$. By calling *PROCESS – STATE* until it returns $k_{min} \geq h(Y)$, the cost changes are propagated to state Y such that $h(Y) = o(Y)$. At this point, a possibly new sequence $\{Y\}$ has been constructed, and the robot continues to follow the backpointers in the sequence toward the goal.

The algorithms for *PROCESS – STATE* and *MODIFY – COST* are presented below. The embedded routines are *MIN – STATE*, which returns the state on the *OPEN* list with minimum $k(\cdot)$ value (*NULL* if the list is empty); *GET – KMIN*, which returns k_{min} for the *OPEN* list (-1 if the list is empty); *DELETE(X)*, which deletes state X from the *OPEN* list and sets $t(X) = CLOSED$; and *INSERT*(X, h_{new}), which computes $k(X) = h_{new}$ if $t(X) = NEW$, $k(X) = min(k(X), h_{new})$ if $t(X) = OPEN$, and $k(X) = min(h(X), h_{new})$ if $t(X) = CLOSED$, sets $h(X) = h_{new}$ and $t(X) = OPEN$, and places or re-positions state X on the *OPEN* list sorted by $k(\cdot)$.

In function *PROCESS – STATE* at lines L1 through L3, the state X with the lowest $k(\cdot)$ value is removed from the *OPEN* list. If X is a *LOWER* state (i.e., $k(X) = h(X)$), its path cost is optimal since $h(X)$ is equal to the old k_{min}. At lines L8 through L13, each neighbor Y of X is examined to see if its path cost can be lowered. Additionally, neighbor states that are *NEW* receive an initial path cost value, and cost changes are propagated to each neighbor Y that has a backpointer to X, regardless of whether the new cost is greater than or less than the old. Since these states are descendants of X, any change to the path cost of X affects their path costs as well. The backpointer of Y is redirected (if needed) so that the monotonic sequence $\{Y\}$ is constructed. All neighbors that receive a new path cost are placed on the *OPEN* list, so that they will propagate the cost changes to their neighbors.

If X is a *RAISE* state, its path cost may not be optimal. Before X propagates cost changes to its neighbors, its optimal neighbors are examined at lines L4 through L7 to see if $h(X)$ can be reduced. At lines L15 through L18, cost changes are propagated to *NEW* states and immediate descendants in the same way as for *LOWER* states. If X is able to lower the path cost of a state that is not an immediate descendant (lines L20 and L21), X is placed back on the *OPEN* list for future expansion. It is shown in the next section that this action is required to avoid creating a closed loop in the backpointers. If the path cost of X is able to be reduced by a suboptimal neighbor (lines L23 through L25), the neighbor is placed back on the *OPEN* list. Thus, the update is "postponed" until the neighbor has an optimal path cost.

Function: PROCESS-STATE ()

```
L1   X = MIN – STATE( )
L2   if X = NULL then return –1
L3   k_old = GET – KMIN( ); DELETE(X)
L4   if k_old < h(X) then
L5     for each neighbor Y of X:
L6       if h(Y) ≤ k_old and h(X) > h(Y) + c(Y, X) then
L7         b(X) = Y; h(X) = h(Y) + c(Y, X)
L8   if k_old = h(X) then
L9     for each neighbor Y of X:
L10      if t(Y) = NEW or
L11        (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) or
L12        (b(Y) ≠ X and h(Y) > h(X) + c(X, Y)) then
L13        b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L14  else
L15    for each neighbor Y of X:
L16      if t(Y) = NEW or
L17        (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) then
L18        b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L19      else
L20        if b(Y) ≠ X and h(Y) > h(X) + c(X, Y) then
L21          INSERT(X, h(X))
L22        else
L23          if b(Y) ≠ X and h(X) > h(Y) + c(Y, X) and
L24            t(Y) = CLOSED and h(Y) > k_old then
L25            INSERT(Y, h(Y))
L26  return GET – KMIN( )
```

In function *MODIFY – COST*, the arc cost function is updated with the changed value. Since the path cost for state Y will change, X is placed on the *OPEN* list. When X is expanded via *PROCESS – STATE*, it computes a new $h(Y) = h(X) + c(X, Y)$ and places Y on the *OPEN* list. Additional state expansions propagate the cost to the descendants of Y.

Function: MODIFY-COST (X, Y, cval)

L1 $c(X, Y) = cval$

L2 if $t(X) = CLOSED$ then $INSERT(X, h(X))$

L3 return $GET – KMIN( )$

2.3 Illustration of Operation

The role of *RAISE* and *LOWER* states is central to the operation of the algorithm. The *RAISE* states (i.e., $k(X) < h(X)$) propagate cost increases, and the *LOWER* states (i.e., $k(X) = h(X)$) propagate cost reductions. When the cost of traversing an arc is increased, an affected neighbor state is placed on the *OPEN* list, and the cost increase is propagated via *RAISE* states through all state sequences containing the arc. As the *RAISE* states come in contact with neighboring states of lower cost, these *LOWER* states are placed on the *OPEN* list, and they subsequently decrease the cost of previously raised states wherever possible. If the cost of traversing an arc is decreased, the reduction is propagated via *LOWER* states through all state sequences containing the arc, as well as neighboring states whose cost can also be lowered.

Figure 1: Backpointers Based on Initial Propagation

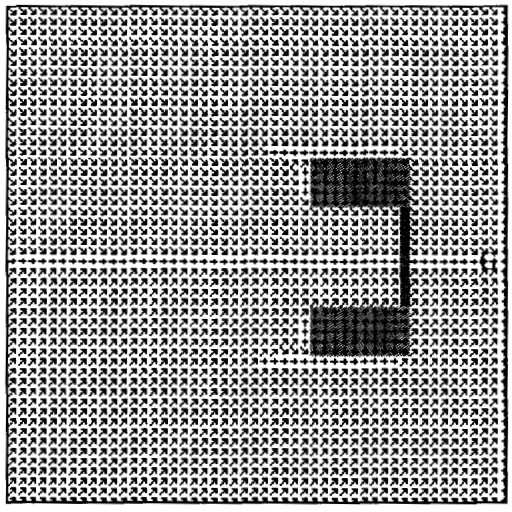
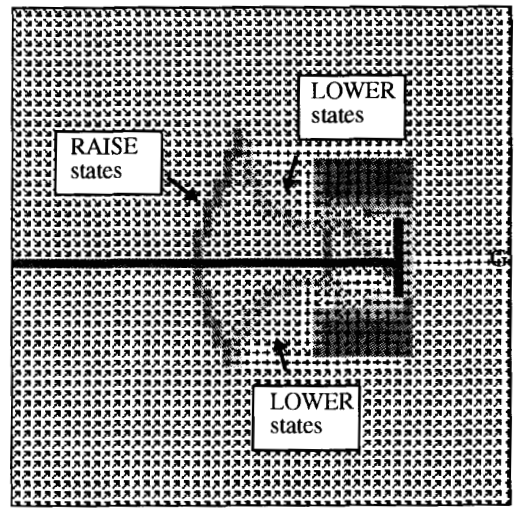


Figure 1 through Figure 3 illustrate the operation of the algorithm for a "potential well" path planning problem. The planning space consists of a 50 x 50 grid of *cells*. Each cell represents a state and is connected to its eight neighbors via bidirectional arcs. The arc cost values are small for the *EMPTY* cells and prohibitively large for the *OBSTACLE* cells.[1] The robot is point-sized and is equipped with a contact sensor. Figure 1 shows the results of an optimal path calculation from the goal to all states in the planning space. The two grey obstacles are stored in the map, but the black obstacle is not. The arrows depict the backpointer function; thus, an optimal path to the goal for any state can be obtained by tracing the arrows from the state to the goal. Note that the arrows deflect around the grey, known obstacles but pass through the black, unknown obstacle.

Figure 2: LOWER States Sweep into Well

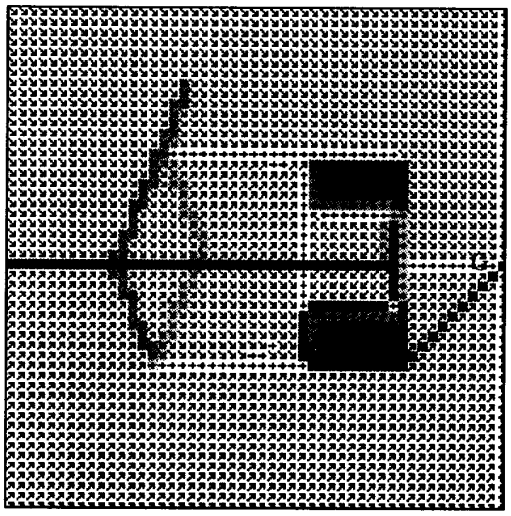


The robot starts at the center of the left wall and follows the backpointers toward the goal. When it reaches the unknown obstacle, it detects a discrepancy between the map and world, updates the map, colors the cell light grey, and enters the obstacle cell on the *OPEN* list. Backpointers are redirected to pull the robot up along the unknown obstacle and then back down. Figure 2 illustrates the information propagation after the robot has discovered the well is sealed. The robot's path is shown in black and the states on the *OPEN* list in grey. *RAISE* states move out of the well transmitting path cost increases. These states activate *LOWER* states around the

1. The arc cost value of OBSTACLE must be chosen to be greater than the longest possible sequence of EMPTY cells so that a simple threshold can be used on the path cost to determine if the optimal path to the goal must pass through an obstacle.

"lip" of the well which sweep around the upper and lower obstacles and redirect the backpointers out of the well.

Figure 3: Final Backpointer Configuration



This process is complete when the *LOWER* states reach the robot's cell, at which point the robot moves around the lower obstacle to the goal (Figure 3). Note that after the traverse, the backpointers are only partially updated. Backpointers within the well point outward, but those in the left half of the planning space still point into the well. All states have a path to the goal, but optimal paths are computed to a limited number of states. This effect illustrates the efficiency of D*. The backpointer updates needed to guarantee an optimal path for the robot are limited to the vicinity of the obstacle.

Figure 4 illustrates path planning in fractally generated terrain. The environment is 450 x 450 cells. Grey regions are fives times more difficult to traverse than white regions, and the black regions are untraversible. The black curve shows the robot's path from the lower left corner to the upper right given a complete, a priori map of the environment. This path is referred to as *omniscient optimal*. Figure 5 shows path planning in the same terrain with an optimistic map (all white). The robot is equipped with a circular field of view with a 20-cell radius. The map is updated with sensor information as the robot moves and the discrepancies are entered on the *OPEN* list for processing by D*. Due to the lack of a priori map information, the robot drives below the large obstruction in the center and wanders into a deadend before backtracking around the last obstacle to the goal. The resultant path is roughly twice the cost of omniscient optimal. This path is optimal, however, given the information the robot had when it acquired it.

Figure 4: Path Planning with a Complete Map

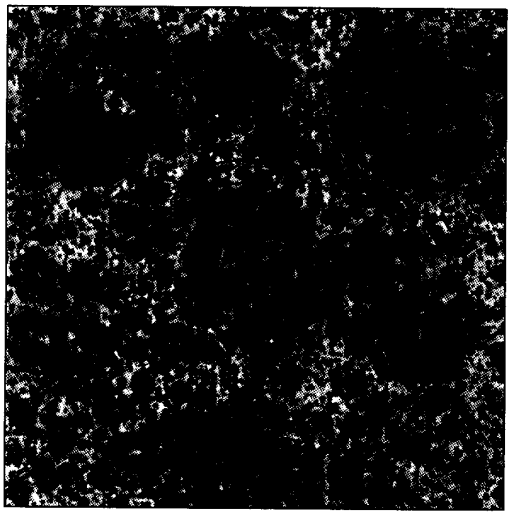


Figure 5: Path Planning with an Optimistic Map

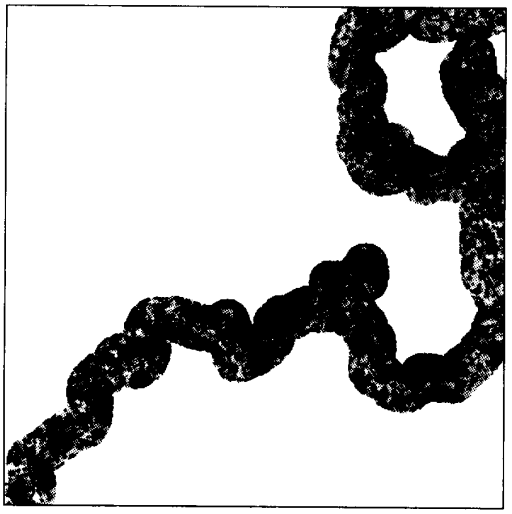
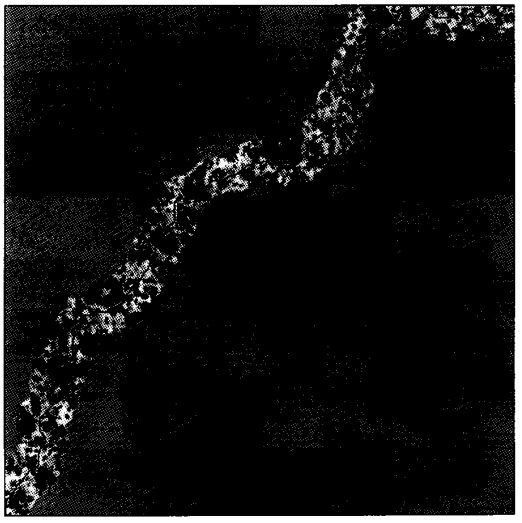


Figure 6 illustrates the same problem using coarse map information, created by averaging the arc costs in each square region. This map information is accurate enough to steer the robot to the correct side of the central obstruction, and the resultant path is only 6% greater in cost than omniscient optimal.

Figure 6: Path Planning with a Coarse-Resolution Map



3.0 Soundness, Optimality, and Completeness

After all states X have been initialized to $t(X) = NEW$ and G has been entered onto the *OPEN* list, the function *PROCESS – STATE* is repeatedly invoked to construct state sequences. The function *MODIFY – COST* is invoked to make changes to $c(°)$ and to seed these changes on the *OPEN* list. D* exhibits the following properties:

Property 1: If $t(X) \neq NEW$, then the sequence $\{X\}$ is constructed and is monotonic.

Property 2: When the value k_{min} returned by *PROCESS – STATE* equals or exceeds $h(X)$, then $h(X) = o(X)$.

Property 3: If a path from X to G exists, and the search space contains a finite number of states, $\{X\}$ will be constructed after a finite number of calls to *PROCESS – STATE*. If a path does not exist, *PROCESS – STATE* will return -1 with $t(X) = NEW$.

Property 1 is a soundness property: once a state has been visited, a finite sequence of backpointers to the goal has been constructed. Property 2 is an optimality property. It defines the conditions under which the chain of backpointers to the goal is optimal. Property 3 is a completeness property: if a path from X to G exists, it will be constructed. If no path exists, it will be reported in a finite amount of time. All three properties hold regardless of the pattern of access for functions *MODIFY – COST* and *PROCESS – STATE*.

For brevity, the proofs for the above three properties are informal. See Stentz [14] for the detailed, formal proofs. Consider Property 1 first. Whenever *PROCESS – STATE* visits a *NEW* state, it assigns $b(°)$ to point to an existing state sequence and sets $h(°)$ to preserve monotonicity. Monotonic sequences are subsequently manipulated by modifying the functions $t(°)$, $h(°)$, $k(°)$, and $b(°)$. When a state X is placed on the *OPEN* list (i.e., $t(X) = OPEN$), $k(X)$ is set to $h(X)$ preserve monotonicity for states with backpointers to X. Likewise, when a state X is removed from the list, the $h(°)$ values of its neighbors are increased if needed to preserve monotonicity. The backpointer of a state X, $b(X)$, can only be reassigned to Y if $h(Y) < h(X)$ and if the sequence $\{Y\}$ contains no *RAISE* states. Since $\{Y\}$ contains no *RAISE* states, the $h(°)$ value of every state in the sequence must be less than $h(Y)$. Thus, X cannot be an ancestor of Y, and a closed loop in the backpointers cannot be created. Therefore, once a state X has been visited, the sequence $\{X\}$ has been constructed. Subsequent modifications ensure that a sequence $\{X\}$ still exists.

Consider Property 2. Each time a state is inserted on or removed from the *OPEN* list, D* modifies $h(°)$ values so that $k(X) \leq h(Y) + c(Y, X)$ for each pair of states (X, Y) such that X is *OPEN* and Y is *CLOSED*. Thus, when X is chosen for expansion (i.e., $k_{min} = k(X)$), the *CLOSED* neighbors of X cannot reduce $h(X)$ below k_{min}, nor can the *OPEN* neighbors, since their $h(°)$ values must be greater than k_{min}. States placed on the *OPEN* list during the expansion of X must have $k(°)$ values greater than $k(X)$; thus, k_{min} increases or remains the same with each invocation of *PROCESS – STATE*. If states with $h(°)$ values less than or equal to k_{old} are optimal, then states with $h(°)$ values between (inclusively) k_{old} and k_{min} are optimal, since no states on the *OPEN* list can reduce their path costs. Thus, states with $h(°)$ values less than or equal to k_{min} are optimal. By induction, *PROCESS – STATE* constructs optimal sequences to all reachable states. If the arc cost $c(X, Y)$ is modified, the function *MODIFY – COST* places X on the *OPEN* list, after which k_{min} is less than or equal to $h(X)$. Since no state Y with $h(Y) \leq h(X)$ can be affected by the modified arc cost, the property still holds.

Consider Property 3. Each time a state is expanded via *PROCESS – STATE*, it places its *NEW* neighbors on the *OPEN* list. Thus, if the sequence $\{X\}$ exists, it will be constructed unless a state in the sequence, Y, is never selected for expansion. But once a state has been placed

on the *OPEN* list, its $k(\circ)$ value cannot be increased. Thus, due to the monotonicity of k_{min}, the state Y will eventually be selected for expansion.

4.0 Experimental Results

D* was compared to the optimal replanner to verify its optimality and to determine its performance improvement. The optimal replanner initially plans a single path from the goal to the start state. The robot proceeds to follow the path until its sensor detects an error in the map. The robot updates the map, plans a new path from the goal to its current location, and repeats until the goal is reached. An optimistic heuristic function $\hat{g}(X)$ is used to focus the search, such that $\hat{g}(X)$ equals the "straight-line" cost of the path from X to the robot's location assuming all cells in the path are *EMPTY*. The replanner repeatedly expands states on the *OPEN* list with the minimum $\hat{g}(X) + h(X)$ value. Since $\hat{g}(X)$ is a lower bound on the actual cost from X to the robot for all X, the replanner is optimal [8].

The two algorithms were compared on planning problems of varying size. Each environment was square, consisting of a start state in the center of the left wall and a goal state in center of the right wall. Each environment consisted of a mix of map obstacles (i.e., available to robot before traverse) and unknown obstacles measurable by the robot's sensor. The sensor used was omnidirectional with a 10-cell radial field of view. Figure 7 shows an environment model with 100,000 states. The map obstacles are shown in grey and the unknown obstacles in black.

Table 1 shows the results of the comparison for environments of size 1000 through 1,000,000 cells. The runtimes in CPU time for a Sun Microsystems SPARC-10 processor are listed along with the speed-up factor of D* over the optimal replanner. For both algorithms, the reported runtime is the total CPU time for all replanning needed to move the robot from the start state to the goal state, after the initial path has been planned. For each environment size, the two algorithms were compared on five randomly-generated environments, and the runtimes were averaged. The speed-up factors for each environment size were computed by averaging the speed-up factors for the five trials.

The runtime for each algorithm is highly dependent on the complexity of the environment, including the number, size, and placement of the obstacles, and the ratio of map to unknown obstacles. The results indicate that as the environment increases in size, the performance of D* over the optimal replanner increases rapidly. The intuition for this result is that D* replans locally when it detects an unknown obstacle, but the optimal replanner generates a new global trajectory. As the environment increases in size, the local trajectories remain constant in complexity, but the global trajectories increase in complexity.

Figure 7: Typical Environment for Algorithm Comparison

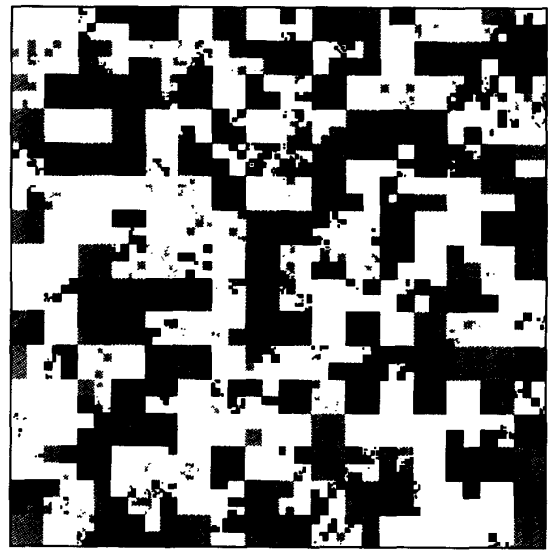


Table 1: Comparison of D* to Optimal Replanner

Algorithm	1,000	10,000	100,000	1,000,000
Replanner	427 msec	14.45 sec	10.86 min	50.82 min
D*	261 msec	1.69 sec	10.93 sec	16.83 sec
Speed-Up	1.67	10.14	56.30	229.30

5.0 Conclusions

5.1 Summary

This paper presents D*, a provably optimal and efficient path planning algorithm for sensor-equipped robots. The algorithm can handle the full spectrum of a priori map information, ranging from complete and accurate map information to the absence of map information. D* is a very general algorithm and can be applied to problems in artificial intelligence other than robot motion planning. In its most general form, D* can handle any path cost optimization problem where the cost parameters change during the search for the solution. D* is most efficient when these changes are detected near the current starting point in the search space, which is the case with a robot equipped with an on-board sensor.

See Stentz [14] for an extensive description of related applications for D*, including planning with robot shape, field of view considerations, dead-reckoning error, changing environments, occupancy maps, potential fields, natural terrain, multiple goals, and multiple robots.

5.2 Future Work

For unknown or partially-known terrains, recent research has addressed the exploration and map building problems [6][9][10][11][15] in addition to the path finding problem. Using a strategy of raising costs for previously visited states, D* can be extended to support exploration tasks.

Quad trees have limited use in environments with cost values ranging over a continuum, unless the environment includes large regions with constant traversability costs. Future work will incorporate the quad tree representation for these environments as well as those with binary cost values (e.g., *OBSTACLE* and *EMPTY*) in order to reduce memory requirements [15].

Work is underway to integrate D* with an off-road obstacle avoidance system [12] on an outdoor mobile robot. To date, the combined system has demonstrated the ability to find the goal after driving several hundred meters in a cluttered environment with no initial map.

Acknowledgments

This research was sponsored by ARPA, under contracts "Perception for Outdoor Navigation" (contract number DACA76-89-C-0014, monitored by the US Army TEC) and "Unmanned Ground Vehicle System" (contract number DAAE07-90-C-R059, monitored by TACOM).

The author thanks Alonzo Kelly and Paul Keller for graphics software and ideas for generating fractal terrain.

References

[1] Goto, Y., Stentz, A., "Mobile Robot Navigation: The CMU System," IEEE Expert, Vol. 2, No. 4, Winter, 1987.

[2] Jarvis, R. A., "Collision-Free Trajectory Planning Using the Distance Transforms," Mechanical Engineering Trans. of the Institution of Engineers, Australia, Vol. ME10, No. 3, September, 1985.

[3] Korf, R. E., "Real-Time Heuristic Search: First Results," Proc. Sixth National Conference on Artificial Intelligence, July, 1987.

[4] Latombe, J.-C., "Robot Motion Planning", Kluwer Academic Publishers, 1991.

[5] Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach", IEEE Transactions on Computers, Vol. C-32, No. 2, February, 1983.

[6] Lumelsky, V. J., Mukhopadhyay, S., Sun, K., "Dynamic Path Planning in Sensor-Based Terrain Acquisition", IEEE Transactions on Robotics and Automation, Vol. 6, No. 4, August, 1990.

[7] Lumelsky, V. J., Stepanov, A. A., "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment", IEEE Transactions on Automatic Control, Vol. AC-31, No. 11, November, 1986.

[8] Nilsson, N. J., "Principles of Artificial Intelligence", Tioga Publishing Company, 1980.

[9] Pirzadeh, A., Snyder, W., "A Unified Solution to Coverage and Search in Explored and Unexplored Terrains Using Indirect Control", Proc. of the IEEE International Conference on Robotics and Automation, May, 1990.

[10] Rao, N. S. V., "An Algorithmic Framework for Navigation in Unknown Terrains", IEEE Computer, June, 1989.

[11] Rao, N.S.V., Stoltzfus, N., Iyengar, S. S., "A 'Retraction' Method for Learned Navigation in Unknown Terrains for a Circular Robot," IEEE Transactions on Robotics and Automation, Vol. 7, No. 5, October, 1991.

[12] Rosenblatt, J. K., Langer, D., Hebert, M., "An Integrated System for Autonomous Off-Road Navigation," Proc. of the IEEE International Conference on Robotics and Automation, May, 1994.

[13] Samet, H., "An Overview of Quadtrees, Octrees and Related Hierarchical Data Structures," in NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics, Berlin: Springer-Verlag, 1988.

[14] Stentz, A., "Optimal and Efficient Path Planning for Unknown and Dynamic Environments," Carnegie Mellon Robotics Institute Technical Report CMU-RI-TR-93-20, August, 1993.

[15] Zelinsky, A., "A Mobile Robot Exploration Algorithm", IEEE Transactions on Robotics and Automation, Vol. 8, No. 6, December, 1992.