A&QT-R 2002 (THETA 13)
2002 IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics
May 23 – 25, 2002, Cluj-Napoca, Romania

PATH PLANNING IN AN UNKNOWN ENVIRONMENT

Radu Robotin and Gheorghe Lazea

Department of Automation, Technical University of Cluj-Napoca,
C. Daicoviciu str. 15, 3400 Cluj-Napoca, ROMANIA
E-mail: {Radu.Robotin,Gheorghe.Lazea}@aut.utcluj.ro

Abstract: 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. The optimal algorithms in this literature search a state space (e.g., visibility graph, grid cells) using the distance transform or heuristics to find the lowest cost path from the robot's start state to the goal state. Cost can be defined to be distance traveled, energy expended, time exposed to danger, etc. This paper presents the D* algorithm for generating optimal paths for a robot operating with a sensor and a map of the environment. 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 and grid cell structures. The paper describes the algorithm, illustrates its operation, presents our approach for implementation, and then concludes with an proof of operation.

Keywords: mobile robot, replanning, dynamic environment, planning algorithm

1. INTRODUCTION

The problem of path planning can be stated as finding a sequence of state transitions through a graph from some initial state to a goal state, or determining that no such sequence exists. The path is optimal if the sum of the transition costs, also called arc costs, is minimal across all possible sequences through the graph. If during the "traverse" of the path, one or more arc costs in the graph is discovered to be incorrect, the remaining portion of the path may need to be replanned to preserve optimality. The states in the graph are robot locations, and the arc values are the costs of moving between locations, based on some metric such as distance, time, energy expended, risk, etc. The robot begins with an initial estimate of arc costs comprising its "map", but since the environment is only partially-known or changing, some of the arc costs are likely to be incorrect. As the robot acquires sensor data, it can update its map and replan the optimal path from its current state to the goal. It is important that this replanning be fast, since during this time the robot must either stop or continue to move along a suboptimal path. A number of algorithms exist for producing optimal traverses given changing arc costs. One algorithm plans an initial path with A* [3] or the distance transform [Jarvis, 1985] using the prior map information, moves the robot along the path until either it reaches the goal or its sensor discovers a discrepancy between the map and the environment, updates the map, and then replans a new path from the robot's current state to the goal [5]. The limitation of these algorithms is that the entire affected portion of the map must be repaired before the robot can resume moving and subsequently make additional corrections. Thus, the algorithms are inefficient when the robot is near the goal and the affected portions of the map have long "shadows". Stentz [4] overcomes these limitations with D*, an incremental algorithm which maintains a partial, optimal cost map limited to those locations likely to be of use to the robot. Likewise, repair of the cost map is generally partial and re-entrant, thus reducing computational costs and enabling real-time performance. Other algorithms exist for addressing the problem of path planning in unknown or dynamic environments but these algorithms emphasize fast operation and/or low memory usage at the expense of optimality.

This paper describes an extension to D* which focuses the cost updates to minimize state expansions and further reduce computational costs. The algorithm uses a heuristic function similar to A* to both propagate cost increases and focus cost reductions. A biasing function is used to compensate for robot motion between replanning operations.

2. DEFINITIONS AND FORMULATIONS

We begin with the notation and definitions used in Stentz [4]. The problem space can be formulated as a set of *states* denoting robot locations connected by *directional arc*s, 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 visited state except G has a *backpointer* to a next state denoted by *b(X)=Y*. D* uses backpointers to represent paths to the goal. The cost of traversing an arc from state to state 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.

D* uses an OPEN list 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 if X has never been on the list *t(X)=NEW*, if X is currently on the list *t(X)=OPEN*, and if X is no longer on the list *t(X)=CLOSED*. For each visited state X, D* maintains an estimate of the sum of the arc costs from X to G given by the path cost function *h(X)*. Given the proper conditions, this estimate is equivalent to the minimal cost from state X to G. For each state X on the OPEN list (i.e., *t(X)=OPEN*), the key function, *k(X)*, is defined to be equal to the minimum of *h(X)* before modification and all values assumed by *h(X)* since X was placed on the OPEN list. The key function classifies a state X on the list into one of two types: a RAISE state if *k(X)<h(X)*, and a LOWER state if *k(X)=h(X)*. D* uses RAISE states on the OPEN list to propagate information about path cost increases and LOWER states to propagate information about path cost reductions. The propagation takes place through the repeated removal of states from the list. Each time a state is removed from the OPEN 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 are sorted on the OPEN list by a *biased f(◦)* value, given by $f_B(X,R_i)$, where X is the state on the OPEN list and R_i is the robot's state at the time X was inserted or adjusted on the OPEN list. Let $\{R_0, R_1, \ldots, R_N\}$ be the sequence of states occupied by the robot when states were added to the OPEN list. The value of $f_B(\circ)$is given by $f_B(X,R_i) = f(X,R_i) + d(R_i,R_0)$ where *f(◦)* is the estimated robot path cost given by

$f(X,R_i)=h(X)+g(R_i,R_{i-1})$ and *d(◦)*is the *accrued bias* function given by $d(R_i,R_0)=g(R_1,R_0)+g(R_2,R_1)+\ldots+g(R_i,R_{i-1})+\varepsilon$ if $i>0$ and $d(R_0,R_0)=0$ if $i=0$.

The function *g(X,Y)* is the focussing heuristic, representing the estimated path cost from Y to X. The list states are sorted by increasing $f_B(\circ)$ value, with ties in $f_B(\circ)$ ordered by increasing *f(◦)*, and ties in *f(◦)* ordered by increasing *k(◦)*. Ties in *k(◦)* are ordered arbitrarily. Thus, a vector of values $\langle f_B(\circ), f(\circ), k(\circ)\rangle$ is stored with each state on the list. Whenever a state is removed from the OPEN list, its *f(◦)* value is examined to see if it was computed using the most recent focal point. If not, its *f(◦)* and $f_B(\circ)$ values are recalculated using the new focal point and accrued bias, respectively, and the state is placed back on the list. Processing the $f_B(\circ)$ values in ascending order ensures that the first encountered *f(◦)* value using the current focal point is the minimum such value, denoted by f_{min}. Let k_{val} be its corresponding *k(◦)* value. These parameters comprise an important threshold for D*. By processing properly-focussed *f(◦)* values in ascending order (and *k(◦)* values in ascending order for a constant *f(◦)* value), the algorithm ensures that for all states X, if $f(X)<f_{\min}$ or ($f(X)=f_{\min}$ and $h(X)<k_{val}$), then *h(X)* is optimal. The parameter *val* is used to store the vector $\langle f_{\min}, k_{val}\rangle$ for the purpose of this test.

3. ALGORITHM DESCRIPTION

The D* algorithm consists primarily of three functions: PROCESS_STATE, MODIFY_COST, and MOVE_ROBOT. PROCESS_STATE computes optimal path costs to the goal, MODIFY_COST changes the arc cost function and enters affected states on the OPEN list, and MOVE_ROBOT uses the two functions to move the robot. The algorithms for PROCESS_STATE, and MOVE_ROBOT are presented below. The embedded routines are: *MIN(a,b)* returns the minimum of the two scalar values *a* and b; *LESS(a,b)* takes a vector of values $\langle a_1, a_2\rangle$ for *a* and a vector $\langle b_1, b_2\rangle$ for *b* and returns *TRUE* if $a_1<b_1$ or ($a_1=b_1$ and $a_2<b_2$); *LESSEQ* takes two vectors *a* and *b* and returns *TRUE* if $a_1<b_1$ or ($a_1=b_1$ and $a_2<=b_2$); *COST(X)* computes $f(X,R_{curr})=h(x)+GVAL(X,R_{curr})$ and returns the vector of $\langle f(X,R_{curr}), h(X)\rangle$ values for a state X; *DELETE(X)* deletes state X from the OPEN list and sets *t(X)=CLOSED*; *PUT_STATE(X)* sets *t(X)=OPEN* and inserts X on the OPEN list according to the vector $\langle f_B(X), f(X), k(X)\rangle$; *GET_STATE* returns the state on the OPEN list with minimum vector value (*NULL* if the list is empty).

In function PROCESS_STATE cost changes are propagated and new paths are computed. At lines L1 through L3, the state X with the lowest *f(◦)* value is removed from the OPEN list. If X is a LOWER state (i.e., *k(X)=h(X)*), its path cost is optimal. 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. 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.

```
Function PROCESS_STATE ()
L1 X=MIN_STATE()
L2 if X=NULL then return NO_VAL
L3 val=<f(X),k(X)>; $k_{val}$=k(X); DELETE(X)
L4 if $k_{val}$<h(X) then
L5 for each neighbor Y of X
L6      if t(Y)!=NEW and
        LESSEQ(COST(Y),val) and
        h(X)>h(Y)+c(Y,X)
        then
L7       b(X)=Y; h(X)=h(Y)+c(Y,X)
L8       if $k_{val}$=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) and
L21      t(X)=CLOSED then
L22               INSERT(X,h(X))
L23      else
L24        if b(Y)!=X and h(X)>h(Y)+c(Y,X)
L25          and t(Y)=CLOSED and
L26          LESS(val,COST(Y)) then
L27               INSERT (Y,h(Y))
L28 return MIN_VAL()
```

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 through L22), is placed back on the OPEN list for future expansion. This action is required to avoid creating a closed loop in the backpointers [4]. If the path cost of X is able to be reduced by a suboptimal neighbor Y (lines L24 through L27), the neighbor is placed back on the OPEN list. Thus, the update is "postponed" until the neighbor has an optimal path cost.

The function MOVE_ROBOT illustrates how to use PROCESS_STATE and MODIFY_COST to move the robot from state S through the environment to G. At lines L1 through L4, *t(*)* is set to *NEW* for all states, the accrued bias and focal point are initialized, *h(G)* is set to zero, and G is placed on the OPEN list. PROCESS_STATE is called repeatedly at lines L6 and L7 until either an initial path is computed to the robot's state (i.e., *t(S)=CLOSED*) or it is determined that no path exists (i.e., *val=NO_VAL* and *t(S)=NEW*). The robot then proceeds to follow the backpointers until it either reaches the goal or discovers a discrepancy (line L11) between the *sensor measurement* of an arc cost and the stored arc cost (e.g., due to a detected obstacle). Note that these discrepancies may occur anywhere, not just on the path to the goal. If the robot moved since the last time discrepancies were discovered, then its state R is saved as the new focal point, and the accrued bias, d_{curr}, is updated (lines L12 and L13). MODIFY_COST is called to correct c(*) and place affected states on the OPEN list at line L15. PROCESS_STATE is then called repeatedly at line L17 to propagate costs and compute a new path to the goal. The robot continues to follow the backpointers toward the goal.

```
Function MOVE_ROBOT (S,G)
L1      for each state X in the graph:
L2               t(X)=NEW
L3      $d_{curr}$=0; $R_{curr}$=S
L4      INSERT(G,0)
L5      val=<0,0>
L6      while        t(S)!=CLOSED        and
        val!=NO_VAL
L7               val=PROCESS_STATE()
L8      if t(S)=NEW then return NO_PATH
L9               R=S
L10     while R!=G
L11     if s(X,Y)!=c(X,Y) for some (X,Y) then
L12         if $R_{curr}$!=R then
L13     $d_{curr}$=$d_{curr}$+GVAL(R,$R_{curr}$)+e;
        $R_{curr}$=R
L14     for each (X,Y) such that
        s(X,Y)!=c(X,Y):
L15        val=MODIFY_COST(X,Y,s)
L16     while LESS(val,COST(R))
        and val!=NO_VAL
L17               val=PROCESS_STATE()
L18               R=b(R)
L19 return GOAL_REACHED.
```

It should be noted that line L8 in MOVE_ROBOT only detects the condition that no path exists from the robot's state to the goal if, for example, the graph is disconnected. It does not detect the condition that all paths to the goal are obstructed by obstacles. In order to provide for this capability, obstructed arcs can be assigned a large positive value of *OBSTACLE* and unobstructed arcs can be assigned a small positive value of *EMPTY*. *OBSTACLE* should be chosen such that it exceeds the longest possible path of *EMPTY* arcs in the graph. No unobstructed path exists to the goal from S if $h(S) \geq OBSTACLE$ after exiting the loop at line L6. Likewise, no unobstructed path exists to the goal from a state R during the traverse if $h(R) \geq OBSTACLE$ after exiting the loop at line L16. Since $R = R_{curr}$ for a robot state R undergoing path recalculations, then $g(R, R) = 0$ and $f(R, R) = h(R)$.

4. IMPLEMENTATION

The above-presented algorithm was implemented on a Pioneer 2 mobile robot, equipped with 8 range finding sensors. We have used client-server architecture, with P2OS operating system on robot's microcontroller, and Saphira routines for the client on a PC workstation. The software was implemented using Microsoft Visual C++, and Saphira OS functions.

In order to perform obstacle detection, Saphira provides occupancy functions. These functions look at the raw sonar readings to determine if an obstacle is near the robot. Other Saphira interpretation micro-tasks use the sonar readings to extract line segments representing walls and corridors. Saphira has several functions for testing whether sonar readings exist in areas around the robot. The different functions are useful in different types of obstacle-detection routines; for example, when avoiding obstacles in front of the robot, it's often useful to disregard readings taken from the side sonars. The detection functions come in two basic types: *box* functions and *plane* functions. Box functions returns obstacles in a rectangular region in the vicinity of the robot, while plane functions look at a portion of a half-plane.

5. RESULTS, CONCLUSION AND FUTURE WORK

The goal of this project was to provide a navigation system that will allow Pioneer mobile robot to explore an unknown terrain without the benefit of an a priori map. This exploration is collision free, as long as the robot's ultrasonic sensors detect the obstacles. There are some cases when the software + hardware (ultrasonic sensors) cannot cope with the environment (e.g., the robot cannot detect chairs legs or other obstacles thinner than 4-5 cm). When a collision occurs, the bump behavior will drive the robot back followed by a detour.

The figures below illustrate the path planning in a clutter environment. An optimistic map of the environment in figure 1 was used. The robot moves from the start point S in the left towards the goal point G and discovers the obstacles in the environment as depicted in figure 2. No a priori information was available.

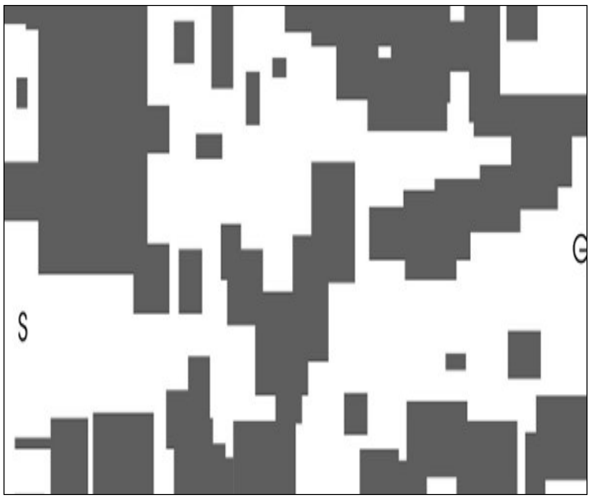

Figure 1. Environment used for simulation .

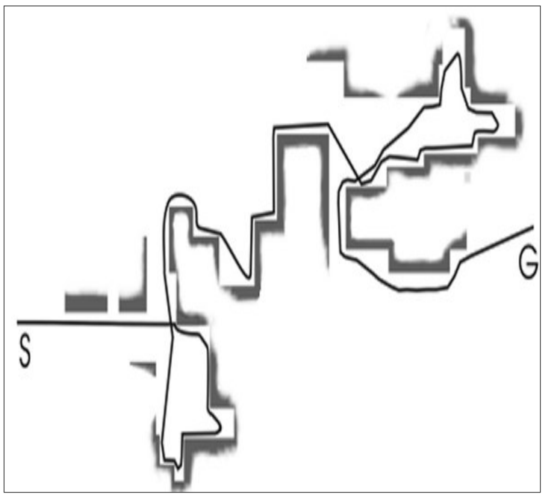

Figure 2. The resulting robot path.

The algorithm can handle the complete spectrum of a priori maps, from accurate maps to absence of map information. The future work plans to compensate and eliminate some of the disadvantages of the current approach: the current implementation of D* algorithm can be improved, especially by using arc costs ranging over a continuum. Also it can be provided a focussing heuristic which will improve the speed performance by reducing the number of searched states. The actual implementation can be inefficient especially when there are series of detected obstacles in the near vicinity of the robot. This drawback can be compensated by using an arbiter which will decide what motor commands are more important in some given conditions: the output of the path-planning routine, or the output of the obstacle avoidance routine. At this point we are looking at the possibility of representing the detected environment in the Saphira Navigation Display, since this feature in not supported

6. REFERENCES

[1] Latombe, J.-C.,(1991). *Robot Motion Planning*, Kluwer Academic Publishers, ISBN 0-7923-9206-X, Boston MA.

[2] Lumelsky, V. J.; Mukhopadhyay, S. & Sun, K., (1990).", IEEE Transactions on Robotics and Automation, Vol. 6, No. 4.

[3] Nilsson, N. J., (1980). *Principles of Artificial Intelligence*, Tioga Publishing Company, ISBN 3-540-11340-1, Palo Alto CA.

[4] Stentz, A.,(1994). *Optimal path planning for partially known environments*, *Available from:* http://www.frc.ri.cmu.edu/~axs/ *Accessed:* 2000-12-10.

[5] Zelinsky, A.,(1992). *A Mobile Robot Exploration Algorithm*, IEEE Transactions on Robotics and Automation, Vol. 8, No. 6, pp 703 – 717.

[6] *** , *Saphira User Manual*, ActivMedia Robotics, 1998.