Real-Time Robot Path Planning via a Distance-Propagating Dynamic System with Obstacle Clearance

Allan R. Willms, Simon X. Yang *Member, IEEE*

***Abstract*—An efficient grid-based distance-propagating dynamic system is proposed for real-time robot path planning in dynamic environments which incorporates safety margins around obstacles using local penalty functions. The path through which the robot travels minimizes the sum of the current known distance to a target and the cumulative local penalty functions along the path. The algorithm is similar to D* but does not maintain a sorted queue of points to update. The resulting gain in computational speed is offset by the need to update all points in turn. Consequently, in situations where many obstacles and targets are moving at substantial distances from the current robot location, this algorithm is more efficient than D*. The properties of the algorithm are demonstrated through a number of simulations. A sufficient condition for capture of a target is provided.**

Keywords:
dynamic system, path planning, safety margins, obstacle clearance, mobile robot, real-time navigation, dynamic environment, dynamic programming.

I. Introduction

In a previous paper [1], we presented a simple yet efficient distance-propagating dynamic system for real-time robot path planning in dynamic environments. The algorithm is essentially a distance-transform method [2]–[5] applied to a fully dynamic environment. Distance-transform methods solve the shortest path problem by using a dynamic programming (DP) algorithm on a cyclic network [6]. Here we extend the algorithm to incorporate safety margins around obstacles; robots not only avoid obstacles, but travel a "safe" distance around them. This is achieved by propagating not just the distance to a target, but also the distance to an obstacle. The "distance" to a target is then modified by a penalty function based on the distance to an obstacle along the path.

Similar to many robot path-planning approaches, the environment is discretized and represented by a topologically organized map. For the distance-propagating dynamic system, each grid point has only local connections to its neighboring grid points. Neighbors need not be all at the same distance. At each time step, each grid point i queries its neighbors to determine their information about distances to targets and obstacles. Distance information thus propagates outward from target/obstacle locations through neighboring grid points. If a target or obstacle moves, a new "wave" of information spreads out from the new location. The algorithm prevents target distance information from traveling through obstacles.

The safety margins around obstacles are computed in a way conceptually equivalent to distance transformation methods [3], [7], [8], with our local updating of neighbors acting similar to sequential erosions with small structuring elements [8]. Our algorithm however allows for dynamic environments where obstacles and targets are permitted to move, and there is no limitation on the size of the obstacles or size of the free space.

True wave front path planners [2], [9], [10] spread information from a source outward in waves to all other points on the grid by updating a grid point's neighbors in the direction of wave propagation and in the order in which the wave arrives at the sites. This information may be simply the distance to a target, or a more complicated function such as a penalized distance for safety considerations, or any other quantity that is intended to be minimized. The order of arrival of waves at different points also depends on the information being propagated. For expositional simplicity we consider here the information as simply the distance to a target. The D* algorithm [10], [11] (which is a modification of A* [9]) and its variants (Focused D* [12], D*-Lite [13], and E* [14]) are true wave front planners. D* determines the correct order of updating points by sorting its open list according to the current distance to the target; updates for points close to the target occur before those further away. By sorting its open list, D* ensures that all points that have moved off its open list have recorded the optimal distance to the target (up to the current information available in the map). The primary feature of D* is that it is capable of re-computing new optimal trajectories when alterations to the map are made (the movement of an obstacle or target is detected) without having to necessarily re-compute the entire solution. Only the solution "down stream" from the alteration is re-computed. Thus, for example, if an obstacle half way between the target and the robot shifts its location, only the solution from this distance out to the robot needs to be re-computed, however, if the target itself moves D* essentially needs to re-compute the entire solution from scratch. D* is most efficient when the alterations to the map occur at points close to the robot [10], and this makes it well-suited for a robot path-planning problem where the robot is equipped with an on-board sensor of limited range, and where information is not being incorporated from other distant

A. R. Willms is with the Department of Mathematics and Statistics, University of Guelph, Guelph, Ontario N1G 2W1, Canada. Email: AWillms@uoguelph.ca

S. X. Yang is with the Advanced Robotics and Intelligent Systems (ARIS) Laboratory, School of Engineering, University of Guelph, Guelph, Ontario, N1G 2W1, Canada. E-mail: syang@uoguelph.ca.

sources.

Our algorithm is similar to D* in that solutions are changed down stream from where the map alteration occurred. However, unlike D*, our algorithm does not maintain a sorted open list but rather simply updates each point in turn. Speed is gained by not having to maintain a queue but is lost because points far from the optimal solution are also being updated. Consequently, in comparison with D*, our algorithm is best suited for highly dynamic environments where alterations to the map are being incorporated from all points, not just those in proximity to the robot. Our algorithm is not a true wave front planner since information does not spread to neighboring points in the order of the distance from the target. This means that occasionally the solution at a particular point may not be the *minimal* distance to the target even assuming the map is completely up to date. However, given a constant map, our algorithm does quickly converge to the optimal solution [1]. We accept this suboptimality in order to achieve simplicity and efficiency by dispensing with the necessity to determine which point needs to be updated next. Computing each point in turn rather than working from the target outwards as done by D* may make it seem like our algorithm needs to do considerably more work. This is certainly the case for constant environments or ones where only map alterations associated with obstacle movements close to the robot are made. However, if we are in highly dynamic environments where many obstacles are moving, where the target itself is moving, and where this information is available locally to the points where the movements are occurring, and not just in a limited proximity to the robot, then our algorithm is actually more efficient.

We emphasize that in our algorithm, the computations *at each point* use only local information. This means that our algorithm is exceedingly easy to parallelize in a shared memory architecture: the grid points are divided into subsets and assigned to different processors for updating. In contrast, most other path-planning algorithms use global information in some way and hence are difficult to parallelize. Algorithms based on a dynamic programming approach make use of global information by stepping through the nodes in an order determined by the current recorded distances at each node or the order in which node values most recently changed. For example, the A* algorithm [9] estimates the distance from a node n to the goal as the sum of a locally propagated distance $g(n)$ from the starting point to n, plus an estimate $h(n)$ of the distance from n to the goal (a piece of global information). Even if A* is implemented with a non-informed heuristic, $h(n) \equiv 0$, the algorithm still sorts the nodes in its open list in ascending distance order, which implies the algorithm has global knowledge of the current distance for each node. The D* family of algorithms also sort based on global knowledge of the current distance at each point in the grid, and focused D* also uses global heuristics like A*.

Neural network approaches to robot path planning [15]–[19] are similar to our approach in that information about the location of the target and obstacles is propagated through local neighbors, however our algorithm has several advantages over these. Typically, in a neural network approach, target locations in the grid input a positive activity to the network and obstacles are either sinks or held at a minimum activity level. Activity is then propagated through the network by local connections according to some ordinary differential equation (ODE) model, and robots follow the path of steepest ascent to the target location. Unlike our algorithm where the penalized distance is propagated through the network, the activity values of most neural networks do not have a direct physical meaning. In addition, although correlated with distance, activity levels often suffer from saturation effects [1] where the gradient in the activity is very small and/or shows considerable sensitivity to the arbitrary parameters of the model. Finally, the computational effort for neural networks is generally considerably more, since numerically solving the ODE requires more work than the simple computations in our discrete algorithm.

Our original algorithm is similar to the "dynamic wave expansion neural network" model proposed by Lebedev et al. [20]. Their model does not record the *physical* distance to the target but rather the sum of twice the *grid* distance and the number of time steps since the target last moved. Also, their algorithm uses only integer arithmetic which decreases computation time but consequently only gives physically minimal distance paths if all neighbors are an equal physical distance from each other. Thus, even for regular square grids, neighbors are restricted to the four horizontal and vertical neighbors; diagonal connections are not allowed. In addition, their algorithm as specified reduces computation time by choosing the *first* neighbor which is passing updated information. They do this to avoid checking all neighbors. However, as a result, their algorithm will generally fail to find the optimal path in situations where moving obstacles suddenly open up shorter paths to a target. To rectify this situation, their algorithm would need to check all neighbors to determine the best information. In contrast, our algorithm records the actual physical (penalized) distance to the nearest target. This information may be useful in a real situation where the behavior of the robot may wish to be altered depending on the proximity of a target. In addition, our algorithm works for any grid, the only requirement is that each grid point has a predefined set of neighbors at known distances through which the robots and targets may travel.

Since we are concerned with changing environments and moving targets, it is not possible to give a measure of the computational effort required for a robot to reach a target on a grid of certain size without reference to the particular environment and how it is changing. (Indeed, it is easy to construct changing environments where the robot can never reach a target no matter how fast the robot moves [1]). However, the computational effort for each node in the grid to be updated once is clearly proportional to the total number of points, M, since each node must simply query the values of its finite set of neighbors. See [1] for a comparison of our algorithm with a number of others.

Although our exposition and most of our simulations are in terms of a mobile robot because this scenario is easy to visualize, the algorithm has more application for robots such as manipulator arms (see Section III-D) where a complete grid of the configuration space is more likely attainable.

II. The Robot Path Algorithm

In this section, we present the distance-propagating dynamic system with obstacle clearance, and the algorithm for the robot, target and obstacle movement. For a description of the original algorithm without obstacle clearance, see [1]. Note that [1] uses different notation than we use here.

A. The Penalized Distance-Propagating Dynamic System

Suppose the robot environment is discretized into a grid of M points, labeled by an index, i, each point being either a free space or an obstacle location. The targets and the robot may occupy any free space. For each point i, B_i is the set of its neighbors, and d_{ij} is the physical distance from i to neighbor j. B_i could be, for example, the eight nearest points to i in a regular square grid, but in general, could be any set of points which you wish to define as the neighbors of i. However, it is assumed that the targets and robots may only move from points to neighboring points. We define $d_{\min}$ and $d_{\max}$ to be the minimum and maximum distances between any two neighbors in the grid.

Each grid point has two associated real-valued variables: $x_i(n)$ which records the "distance" to the nearest obstacle at time step n, and $y_i(n)$ which records the "penalized distance" to the nearest target. In addition, each point maintains two integer-valued variables, $p_i^x(n)$ and $p_i^y(n)$ which record the "parents" for x and y, that is, the neighbors of i through whom the values of $x_i(n)$ and $y_i(n)$ were calculated respectively. Define a local obstacle penalty function, $q(x)$, $0 \le q(x) < q_{\max}$, where $q_{\max}$ is some finite constant. The function $q(x)$ is the penalty per unit distance traveled. The cost of travel, that is, the penalized distance, along a path that is a distance x from an obstacle is equivalent to an unpenalized path that is $1 + q(x)$ times longer.

The system is initialized as

$$x_i(0) = \begin{cases} 0, & \text{if an obstacle is at } i,, \\ D, & \text{otherwise,} \end{cases}$$
$$p_i^x(0) = i,$$
$$y_i(0) = \begin{cases} 0, & \text{if a target is at } i, \\ D, & \text{otherwise,} \end{cases}$$
$$p_i^y(0) = i.$$

where D as some large maximal penalized distance. The precise value of D is not too important. If D is chosen too small, then only grid points which are less than a penalized distance D from a target will participate in the algorithm. A sufficiently large maximal D value which ensures all grid points participate is given by $D > (M-1)\left(d_{\max}\left[1 + q_{\max}\right]\right)$.

The dynamic system evolves as follows:

$$x_i(n) = \begin{cases} 0, & \text{if an obstacle is at } i, \\ \min\limits_{j \in B_i}\left(x_j(n-1) + d_{ij}\right), & \text{otherwise.} \end{cases} \quad (1)$$

$$p_i^x(n) = \begin{cases} i, & \text{if an obstacle is at } i, \\ \operatorname{argmin}\limits_{j \in B_i}\left(x_j(n-1) + d_{ij}\right), & \text{otherwise.} \end{cases} \quad (2)$$

$$y_i(n) = \begin{cases} D, & \text{if an obstacle is at } i, \\ d_{\min} q\left(x_i(n)\right), & \text{if a target is at } i, \\ \min\limits_{j \in B_i}\Big(y_j(n-1) + d_{ij}\left[1 + q(x_i(n))\right]\Big), & \\ & \text{otherwise.} \end{cases} \quad (3)$$

$$p_i^y(n) = \begin{cases} i, & \text{if an obstacle or target is at } i, \\ \operatorname{argmin}\limits_{j \in B_i}\Big(y_j(n-1) + d_{ij}\left[1 + q(x_i(n))\right]\Big), & \\ & \text{otherwise.} \end{cases} \quad (4)$$

$$\left.\begin{array}{l} \text{if } y_i(n) \ge D \\ \quad y_i(n) = D \\ \quad \text{if an obstacle is not at } i \ \&\ q(x_i(n)) > 0 \\ \quad\quad p_i^y(n) = \operatorname{argmax}\limits_{j \in B_i} x_j(n), \\ \quad \text{else} \\ \quad\quad p_i^y(n) = i. \end{array}\right\} \quad (5)$$

where the function $\operatorname{argmin}_{j \in B_i} f(j)$ returns the *first* element j in B_i at which $f(j)$ is a minimum. The argmax function is defined analogously. In Section II-C we discuss how we sort B_i so that argmin and argmax preferentially select an appropriate parent when more than one possibility exists. The first step of the evolution, (1) and (2), simply computes the distance to the nearest obstacle, keeping track of the x-parent, that is, the neighbor through whom the minimal distance is measured. Note that the argmin in (2) is computed at the same time as the minimum in (1). If an obstacle is at i, we define the x-parent of i to be itself. Equations (3) and (4) compute the penalized distance to the nearest target, keeping track of the y-parent (again, the argmin in (4) is computed simultaneously with the minimum in (3)). In this case, if either an obstacle or a target is at i, we define the y-parent of i to be itself. The significance of setting y_i to be nonzero when i is a target is discussed below in Section II-B. Finally, (5) modifies the parent of y_i if the distance to the target is as big as D; this is for robot movement and is discussed in Section II-C. Note that the dynamical system, (1)–(5), uses values of $x(n-1)$ to update $x(n)$ and $p^x(n)$, and values of $x(n)$ and $y(n-1)$ to update $y(n)$ and $p^y(n)$. Thus $x(n)$ must be computed before $y(n)$. It is not necessary to store values of x, and y for all n, the current and previous values are sufficient. An illustration of how Equations (1)–(4) work is shown in Figure 1.

The variable $x_i(n)$ records the distance from i to the nearest obstacle at time step n. Of course obstacles can move and it takes some time for the entire system to reflect this alteration, so the value of $x_i(n)$ may not be up to date for all i. More precisely, $x_i(n)$ records the minimum over $k \in \{0, 1, \ldots, n\}$ of the physical distance from i to the nearest obstacle which was k grid steps away from i at k time steps in the past. The physical distance, x, and the number of grid steps, k, are related by

$$k d_{\min} \le x \le k d_{\max}.$$

Define f_u to be the system update frequency: the number of time steps per unit of real time. Thus if $x_i(n)$ is less than D, it records information from between $\left\lceil \frac{x_i(n)}{d_{\max}} \right\rceil / f_u$ and $\min\left(n, \left\lfloor \frac{x_i(n)}{d_{\min}} \right\rfloor\right) / f_u$ time units in the past, where $\lceil w \rceil$ and

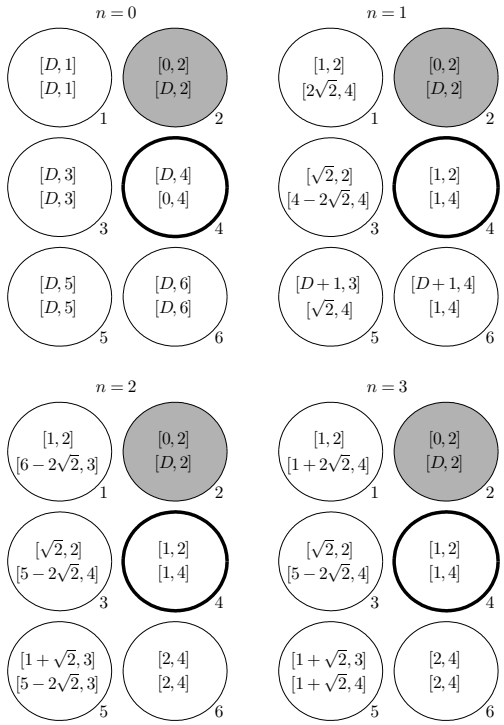

Fig. 1. Illustration of the dynamic system evolution, equations (1)–(4). The figure shows four time steps for a unit grid with six points, labeled 1 through 6. Point 2 is an obstacle (shaded) and point 4 is a target (thick circle). The two sets of numbers inside each circle are $[x, p^x]$ (upper set), and $[y, p^y]$ (lower set). The initial situation, $n = 0$, is displayed in the upper left. The penalty function is $q(x) = \max(3-2x, 0)$. At $n = 1$, points 1 through 4 have correct x values and x-parents while points 5 and 6 have not yet received information about the obstacle. By step $n = 2$ all points have correct $[x, p^x]$ pairs. The situation is more complicated for the y values. For example, at $n = 1$, point 1 has $x(1) = 1$ hence $q(x(1)) = 1$ and so $y(1)$ is $2\sqrt{2}$, while point 3 has $x(1) = \sqrt{2}$ hence $q(x(1)) = 3 - 2\sqrt{2}$ and so $y(1)$ is $4 - 2\sqrt{2}$. Also at $n = 1$, the y value for point 4 increases to 1 due to the second line of (3). This increase in y for the target at $n = 1$ causes alterations in y for its neighbors at $n = 2$. For example, the computation of $y(2)$ for point 1 yields $6-2\sqrt{2}$ with the parent being point 3 since the penalized distance to point 4 from point 1 is $1 + \sqrt{2}\,(1 + q(1)) = 1 + 2\sqrt{2} > 6 - 2\sqrt{2} = 4 - 2\sqrt{2} + (1 + q(1))$. By $n = 3$ though, y values for all points have converged to their appropriate values.

$\lfloor w \rfloor$ are the smallest integer greater than or equal to w and largest integer less than or equal to w respectively. Similarly, $y_i(n)$ records the minimal cost (penalized distance to a target) for point i at time step n, which may be out of date by at most $\left\lfloor \frac{y_i(n)}{d_{\min}} \right\rfloor / f_u$ real time units.

Since the minimization is performed by searching over the neighbors of each point and each point has a finite number of neighbors, the computational burden for this penalized distance-propagating dynamic system (the time required to update every grid point once) is proportional to the total number of points, M.

B. Penalty Function

The relative degree of "safety" is specified by the user in the penalty function formation. The safety margin around an obstacle is defined as the region in which the penalty function is nonzero. The safety margins around the obstacles are "soft" in the sense that if a route that passes into a safety margin is lower cost (distance plus penalty) than all routes which stay out of the safety margin, then the former will be chosen. If hard safety margins were desired then one could simply define the points in the safety margin to be obstacle points (that is, grow the obstacle) and then use the original algorithm without obstacle clearance. In particular, the soft margins allow the robot to pass adjacent to obstacles if there is no other safer way to reach a target.

The precise manner in which the penalty information has been applied warrants some discussion. Along the path between any two neighbors, the distance to the nearest obstacle is in general changing. In a continuous setting, the penalty should be specified as

$$Q = \int_0^{d_{ij}} q\left(x(s)\right)\, ds, \tag{6}$$

where s is the distance along the path. Since $x(s)$ is only known at the two end points of this integral, an obvious approximation is

$$Q \approx \frac{d_{ij}}{2}\left(q(x_i) + q(x_j)\right).$$

However, in (3) the single end-point approximation $Q \approx d_{ij} q(x_i)$ has been used. The reason for this is that the two-point approximation yields a dynamical system with approximately 5/3 the number of arithmetic operations as the one-point approximation. If extra storage is used to hold the values $d_{ij}\left(1 + \left(q(x_j) + q(x_i)\right)/2\right)$, then this factor can be reduced to about 4/3. In any event, we did not feel that the improvement in the approximation warranted the slow down in the algorithm. Also to minimize computations, the penalty at x_i is used in (3) rather than at x_j; thus the one-point approximation for the penalty is applied at the "uphill" ends of the path segments, that is, at the ends closest to the robot. Consider the situation where a target at point i lies within the safety margin of some obstacle but the rest of the points along the optimal path to the robot lie outside all safety margins. If y_i were zero, then, the distance along this path would incur no penalty. In order to avoid ignoring a penalty contribution at the target end of the path, the value for y_i when a target is at i is set to $d_{\min} q(x_i)$ rather than zero. However, this only has any effect on the algorithm if there is more than one target.

The form of the penalty function is important. It is reasonable to assume that $q(x)$ is a decreasing function of x, and that beyond some certain value, $x = B$, q is zero (although such local limitation on the size of penalty margins is not essential). The simplest such form for $q(x)$, is a piecewise linear function

$$q(x) = \begin{cases} A(B - x), & x < B, \\ 0, & x \geq B, \end{cases} \tag{7}$$

where $A > 0$ is a strength factor and $B > 0$ represents the width of the safety margin. However, other forms are possible. The simulations of Section III illustrate that the algorithm can be very sensitive to the precise penalty function being used. For (7), increasing A has the effect of "hardening" the safety margin so that the robot is less likely to get close to an obstacle, and increasing B widens the margin.

C. Robot Movement

We assume that the robot can move from any grid point to any neighboring free space, that is, point robot dynamics. The robot location, $r(t)$, is specified as an index of one of the points on the grid, and is a (discontinuous) function of real time $t \geq t_0$. Initially, $r(t_0) = i_0$. We assume that the robot's travel path is updated at a set of real time values $t_1 < t_2 < t_3 < \ldots$, and that the robot's actual location for $t \in (t_k, t_{k+1})$ is somewhere between the grid points $r(t_k)$ and $r(t_{k+1})$. At time t_k, the next update time, t_{k+1}, and next update location, $r(t_{k+1})$, are determined. The latter is defined as

$$r\left(t_{k+1}\right) = p^y_{r(t_k)}\big(n(r(t_k), t_k)\big), \tag{8}$$

where $n(i, t_k)$ means the highest time step n for which y_i has been computed up to time t_k. Thus, when a robot arrives at location i it proceeds to the current y-parent of i. For example, consider the situation shown in Figure 1 and suppose the points are updated in their label order and that updating each point requires 0.1 time units. To update all six points once would require 0.6 time units. Suppose the robot is at point 5 at time $t_k = 2.0$ ($r(2.0) = 5$). For this value of t_k, $n(i, t_k)$ is 3 for points $i = 1$ and $i = 2$, but is only 2 for points $i = 3, \ldots, 6$. Consequently, the next position of the robot will be $r(t_{k+1}) = p^y_{r(2.0)}(2) = 3$. However, if t_k was 2.3 or higher then $n(5, t_k) = 3$ and the next robot position would have been $r(t_{k+1}) = 4$.

When computing $y_i(n)$ in (3), it is possible that more than one index $j \in B_i$ attains the minimum of $f = y_j(n-1) + d_{ij}\big(1 + q(x_i(n))\big)$. Which index to select as the parent of $y_i(n)$ is not unique. The method we used was to select the neighbor $j \in B_i$ such that f was minimized and the angle from i to $p^y_i(n)$ was as close as possible to the angle from i to $p^y_i(n-1)$. In other words, we minimized changes over time in the direction of the optimal path through i. To achieve this, we had each grid point, i, store not just a set B_i of its m neighbor indices, but m differently ordered versions of B_i, each in increasing angle magnitude starting from one of the m neighbors. These orderings are determined by the geometry of the grid and do not change, hence need only be computed once at the beginning. If $p^y_i(n-1)$ is not i (that is, if the parent at the previous time step is one of the neighbors) then the neighbor $p^y_i(n-1)$ is always queried first, followed by the other neighbors in increasing angle magnitude order. The value of $p^y_i(n)$ only changes if f as measured through the next neighbor strictly decreases.

It is possible that while a robot is waiting at a grid point (with $y = D$) for distance information to be propagated to it from a target, an obstacle moves toward the robot and would collide with it if the robot did not move. Since the dynamical system calculates the distance to obstacles, this information can be used to make the robot move away from obstacles when it does not know which way to go toward a target. This is the purpose of (5). If i is a free space with $y_i(n) = D$, and the penalty value $q(x_i(n))$ is positive, then a further computation is performed to set the parent of y_i to be the neighbor that is furthest from an obstacle. Again, this maximum may be attained by several neighbors so we query the neighbors in the order described above so that again the changes in the angle of the potential robot paths are minimized over time.

Note that the update interval, $\Delta t_k = t_{k+1} - t_k$, need not be constant nor predetermined. For example, with a regular unit square grid, if the robot moves at constant speed v_r, then Δt_k will be either $1/v_r$ or $\sqrt{2}/v_r$ time units, depending on whether the distance from $r(t_k)$ to $r(t_{k+1})$ is 1 or $\sqrt{2}$.

In the case of a static environment, once the location at which the robot resides has settled to its final y value (which occurs when the number of time steps exceeds the number of grid steps between the target and the robot via the shortest distance path) evolution of the dynamic system can cease and the robot can simply follow the minimum distance path to the target using (8).

D. Target and Obstacle Movement

Alterations to the environment map regarding the locations of targets and obstacles can happen at any time. An alteration at point i will begin to be reflected in the dynamical system as soon as point i or any of its neighbors is next updated. When a point i that was previously a free space becomes an obstacle, all paths from i outward (increasing distance) are no longer valid. To increase the speed of computing new solutions for these descendant points, j, the values of y_j and p^y_j are reset to D and j. This is done before the system resumes updating points in turn. More precisely, if an obstacle newly appears at point i during time step n, then the function $\mathrm{ErasePath}(i)$ is called where

```
ErasePath(i)
    for each neighbor j of i
        if p_j^y(n-1) = i
            y_j(n-1) = D
            p_j^y(n-1) = j
            ErasePath(j)
        end if
    end for
end
```

Since obstacles can move, collisions with robots or targets are possible. The simulations of the next section were designed so that obstacles never collided with the target. (Except in the fourth simulation where points can be simultaneously obstacles and targets.) The penalty function is designed to keep the robots away from the obstacles but collisions with obstacles are still possible, for example when the only path to a target passes adjacent to a moving obstacle. If such a collision occurs then the penalty function needs to be increased, and/or the robot speed needs to be increased to prevent collisions.

III. Simulation Studies

In this section we demonstrate the effectiveness and efficiency of the proposed algorithm with various simulations. In all of these simulations the system clock is used to create a real-time environment. For example, if the robot begins (system clock at $t = 0$) traveling at two units per second toward a point i that is one unit away, the robot will arrive at $t = 0.5$ and the decision as to where the robot will next move

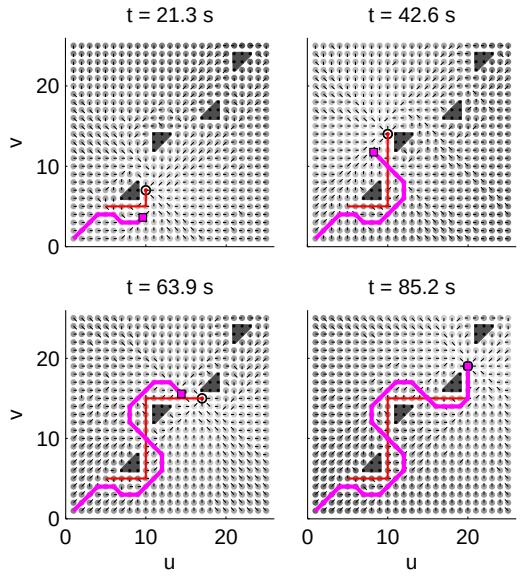

Fig. 2. Target Chasing. Four time snap shots of the environment are shown. Obstacles are black points with dark grey shading around them. The target is a bull's eye pattern and its path is a thin solid line; it starts at (5,5) and zig-zags around the obstacles. The robot is a square and its path, starting at (1,1), is shown as a thick solid line. Free spaces are shaded from light to dark, light being smallest values for the penalized distance, y, to a target. The short lines emanating from the free spaces point to the neighbor through whom the information for updating y was obtained, that is, they point in the direction of greatest decrease in y. The safety margin is $1+\sqrt{2}$ and the robot remains outside these margins until $t = 71$, after which it remains within the safety margin of the third obstacle until catching the target at (20,19).

is made based on the value of p_i^y at time $t = 0.5$. Similarly, if an obstacle is moving at a speed of one grid unit per second, then its presence at each subsequent point on the grid is not signaled until the system clock advances another second.

A. Target Chasing

In this simulation, the target moves at a speed of 0.35 grid units per second from the location $(u, v) = (5, 5)$ in a zig-zag pattern around some non-moving obstacles toward (25,25), see Figure 2. The robot starts at (1,1) and travels at a speed of 0.5 grid units per second. The penalty function is given by (7) with $A = 2$ and a safety margin of $B = 1 + \sqrt{2}$. Notice that the robot remains outside the safety margins around the obstacles until $t = 71$ when it occupies the point $(u, v) = (17, 14)$. It enters the safety margin at that point since the target is very close by at (19,15), and all other safer paths have larger distance to the target. Thereafter, the robot remains close to the obstacle as it chases down the target, finally reaching it at (20,19).

The precise form of the penalty function can substantially alter the robot's path. For example, if the penalty function (7) is altered so that $A = 1.95$ rather than 2, the resulting robot path is shown in Figure 3. Note that the robot enters the safety margins several times because the penalty for doing so is not as great as remaining outside the margin and traveling a longer path. The robot ends up catching the target earlier at (18,15). Similar results can be achieved by decreasing B.

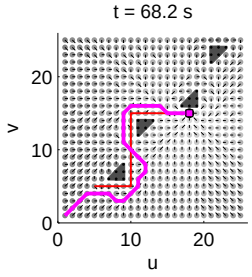

Fig. 3. The same simulation as in Fig. 2, but with $A = 1.95$ in the penalty function rather then 2.0. With the lower penalty for being close to obstacles, the robot ventures inside the safety margins (again of width $1+\sqrt{2}$) a number of times while chasing the target, resulting in a shorter overall path and an earlier capture of the target.

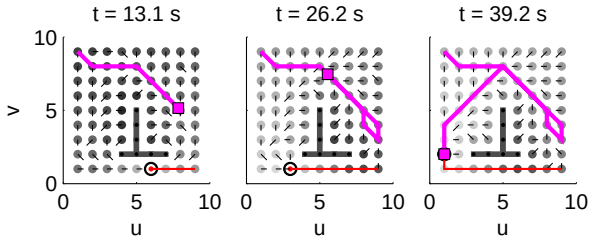

Fig. 4. Path Abandonment. Three time snap shots of the environment are shown. Symbols as in Figure 2. The target moves clockwise around the edge starting in the south-east corner. The robot, starting in the north-west corner, initially moves east of the central barrier toward the target. However, as the target moves west and the barrier at $v = 2$ slides east, this path to the target becomes increasingly unsafe. Eventually, the robot turns around and pursues the target by going north around the barrier at a safe distance.

This simulation also illustrates how penalty functions around obstacles can prevent the robot from reaching a target even if the robot travels considerably faster than the target. If the target continues to weave close in around obstacles, and the penalty function is sufficiently high, the optimal safe path for the robot will be substantially longer than the target's path. In the above simulation, the robot speed was sufficiently large that eventually the robot caught the target. However, it is not difficult to imagine a scenario where the speed advantage of the robot is exactly countered by its longer path. Indeed when we re-ran the above simulation, either increasing A to 3.5 or B to 3, the robot failed to catch the target until after it had halted at (25,25). If the obstacles were arranged in a closed loop and the target wove tightly through them, it would be possible to have the distance between the target and robot settle into a stable periodic cycle, bounded above and below.

B. Path Abandonment

In this simulation, Fig. 4, the robot's path to the target becomes increasingly unsafe as the target and obstacles move. Eventually, the robot abandons this path to the target and chooses a safer path. The penalty function is again given by (7) with a safety margin of $B = 1 + \sqrt{2}$, and $A = 2.5$. The target starts at $(u, v) = (9, 1)$ and moves clockwise around the exterior at a speed of 0.25 grid units per second. There is a stationary north-south barrier up the center of the grid grid,

and an east-west barrier that starts by blocking $2 \leq u \leq 5$, $v = 2$. This barrier moves eastward at a rate of 0.18 grid units per second until it blocks $5 \leq u \leq 8$ at which point it moves back west to its initial placement and so on. The robot starts at (1,9) and moves at a speed of 0.65 grid units per second. Initially, the robot moves to the east of the central barrier toward the moving target. However, when the robot reaches $(9, 3)$ shortly before $t = 19$, the path it has been following toward the target now runs adjacent to the east-west barrier along its entire length. The penalty for being this close to the barrier for this many points outweighs the extra distance needed to reach the target by going back north around the barrier. The robot therefore turns around and pursues the target by going north safely around the barrier.

C. Sliding Grates

This example illustrates how our algorithm is more efficient than D* in situations where the target is moving and there are many obstacles continually moving. Simulations were performed on an Intel Pentium 4, 3.06 GHz machine. Qualitatively similar results were obtained on a faster SGI Altix machine using larger grids.

The D* algorithm can determine when a point is recording the up-to-date optimal path to a target. In contrast, our algorithm as we have specified it here, does not keep track of whether the current recorded information at a point is either optimal or up-to-date. As indicated earlier, we accept this suboptimality to maintain simplicity. In most situations we also expect that the information will not be significantly suboptimal since our system converges rapidly to the optimal solution [1], and will not be significantly out of date provided the system update frequency, f_u, is sufficiently large. Our algorithm could also be modified to implement a convergence test for optimality as described in [1], or global information could be utilized to check if the information is up-to-date, if it was felt necessary. In the following simulations, we compare our algorithm with two implementations of D*. The first implementation requires the robot to wait at a point until the information there is optimal, and the second allows the robot to move based on the current information available. We name these two implementations D*-patient and D*-eager respectively. The second implementation allows a fairer comparison with our algorithm. Further, we have incorporated safety margins in both implementations of D* by modifying the cost functions whenever the distance to an obstacle changes for a point [11]. This "distance to an obstacle" information is itself computed with another D* algorithm which halts when correct distances from obstacles out to the edge, B, of the safety margin are computed.

In the first simulation, a grid of size 89 by 86 is used. There are two stationary obstacles along the north ($v = 86$) and south ($v = 0$) boundaries from $u = 5$ to $u = 85$. There are also 11 sets of north-south "grates", at $u = 5, 13, 21, \ldots, 85$. Each set has 12 grates spanning three points and separated by four points. All the grates in a set slide in unison northward until the north-most grate touches the north stationary obstacle, then southward until the south-most grate touches the south

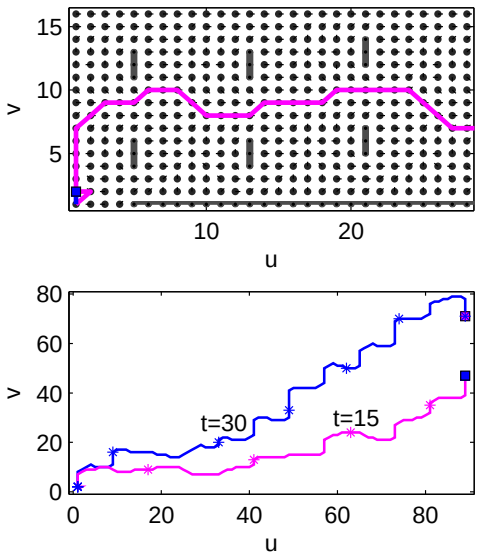

Fig. 5. Sliding Grates. Symbols as in Figure 2. The top panel shows a portion of the grid near the robot's initial location at $t = 10$ seconds. At this point, the obstacles at $u = 5$ are moving northward, while those at $u = 13$, and 21 are moving southward. The target is off the graph to the east. The grey line shows a portion of the path of the robot under our algorithm. Under both D* implementations, the robot is still waiting at $(1, 1)$. The bottom panel shows the robot paths under the two algorithms when the target is captured at the eastern edge. Under D*-eager (dark line), the target is captured at $t = 50$ seconds while under our algorithm (grey line) it is captured sooner at $t = 23$ seconds. Under D*-patient the robot never leaves the starting point. The stars on the two robot paths are at five second intervals, to allow comparison of the two paths in time. Two of these times are labeled.

stationary obstacle, etc., at a speed of two units per second. The initial north-south location of each grate set is randomly selected. See Figure 5. The target starts in the south-east corner and moves northward with a speed of two units per second; if it reaches the north-east corner it begins moving south at the same speed. The robot starts in the south-west corner, at $(u, v) = (1, 1)$, and moves with a speed of six units per second. The penalty function is given by (7) with $A = 2$ and $B = 4$. Under our algorithm, the robot safely reaches the target at time $t = 23$. Using D*-eager, where the robot is permitted to move based on suboptimal information, the robot captures the target at $t = 50$. If the robot is required to wait until optimal information has been computed (D*-patient) then the robot never actually leaves the starting point. The many map alterations due to distant moving obstacles and targets cause D* to continually insert points onto its sorted open list and to make changes to its path cost functions. The time required to do these alterations and the sorting slows the propagation of information outward to the robot. Consequently, under both D* implementations the robot ends up waiting at the starting location for a long time (21.3 seconds) before any information arrives there. Under D*-eager, once the robot starts to move the algorithm performs comparably to ours, with the robot capturing the target about 29 seconds later. Under D*-patient, even though some information arrives at $(1, 1)$, the continual updating of target and obstacle locations over the whole grid does not allow D* to ever complete computation of optimal

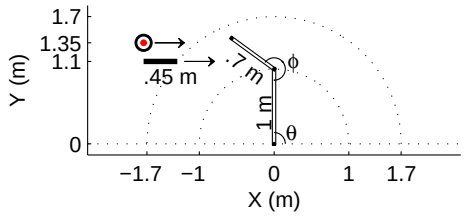

Fig. 6. Manipulator arm specifications. The barrier and target move left to right at constant height and speed.

information at $(1, 1)$, and hence the robot never moves. In comparison, information under our algorithm first arrives at the robot at time $t = 0.6$ and the robot immediately begins to move.

We also ran similar simulations with different height grids. Removing one grate from each set (so the grid is 89×79) was enough to make all three algorithms comparable. Capture times were: $t = 23.16$ for ours, $t = 28.11$ for D*-patient, and $t = 23.01$ for D*-eager. If one grate was added to each set (so the grid is 89×93), then under our algorithm the robot captured the target at $t = 23.2$ while under both implementations of D* the robot never moved. This is beyond D*'s limit of size for this example on our machine; the continual updates to obstacle and target movements completely occupy the computations preventing information from ever reaching all the way across the grid to the robot's initial location. Our algorithm was successful for much larger grids also, for example it captured the target at $t = 46$ seconds under a grid of size 161×142 (20 sets of 20 grates). We also performed similar simulations on a faster computer (an Altix 350 with an Itanium 2 processor) and found qualitatively similar results with larger grid sizes.

D. Manipulator Arm

In this simulation we consider a two-hinged manipulator arm as shown in Figure 6. A barrier of length 45 cm moves from left to right across the arm's field at a height of 110 cm and a speed of 10 cm per second. At its trailing edge and 25 cm higher, a target moves with the same velocity. The first link of the arm has length 1 m and is initially pointing straight upward ($\theta = 90$ degrees). The second link is 0.7 m and initially the angle between the two arms, ϕ, is 243 degrees. The task is to make the tip of the arm reach the target. The arm's speed is set at 30 degrees per second, that is, $\sqrt{\dot{\theta}^2 + \dot{\phi}^2} = 30$. Rather than physical space, the optimal path is determined in the arm's configuration space, (θ, ϕ). In this space, the bar shaped barrier becomes an irregularly shaped obstacle since no part of the arm is allowed to contact the barrier. Also, the mapping from configuration space to physical space is two to one (except when ϕ is a multiple of π), hence the single target has two representations in configuration space. Also, since there are manipulator arm configurations which reach the target but which also intersect the barrier, a point in configuration space can be both an obstacle and a target. In this case, to avoid moving toward such a point, the obstacle takes precedence in (3) so that y will have a value D there.

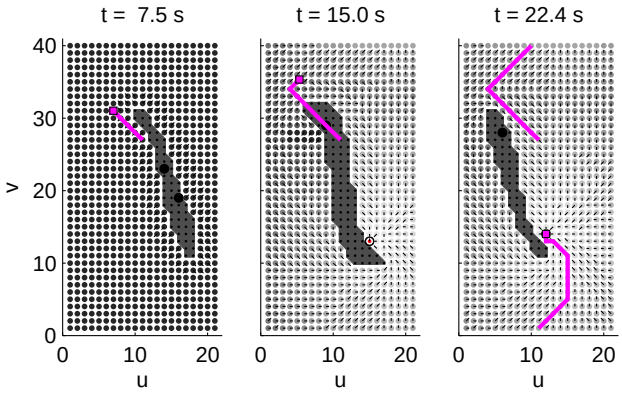

Fig. 7. Manipulator arm configuration space at three times. Target points which are simultaneously obstacles are indicated as large solid black circles. Initially the target is not in the arm's field but the barrier moving toward the arm causes, through the penalty function, the arm to move out of the way. By $t = 7.5$, the target is in the arm's field but is unreachable due to the barrier's location. The target first becomes reachable at about $t = 15$ and the second link of the arm swings completely around ($\phi = 2\pi$) in order to reach it, which occurs at $t = 22.5$.

Figure 7 shows four time snap shots of the environment in configuration space. Here we have discretized the angles so that there are nine degrees between successive grid points. That is, $u = 1 + 20\theta/\pi$, $0 \le \theta \le \pi$, and $v = 1 + 40\,(\phi \mod 2\pi)\,/(2\pi)$. The penalty function is again given by (7) with $A = 0.2$, and $B = 0.4$. The clock starts when the leading edge of the barrier first enters the arm's field. The target does not enter the arm's field until about $t = 7.1$, and when it does so, it is unreachable since the barrier location precludes arm configurations that would otherwise reach the target. This remains the case until about $t = 14$ seconds when the target becomes reachable from the left. However, well before the target is ever visible to the robot, the barrier has moved sufficiently far into the arm's field to have collided with the arm had the arm not moved. In the first panel of Figure 7 we see that at $t = 5.7$ the robot arm has already moved up and to the left a short distance. (Movement began at about $t = 2.6$.) This movement is in response to the obstacle's penalty function becoming nonzero at the robot's location even though the value of y is everywhere still D on the grid. The robot movement algorithm has the robot moving in a direction the greatest distance from an obstacle if the value of y is D and the penalty function is nonzero. Once the robot's location receives the information about the reachability of the target, it moves, increasing ϕ through 2π so that the manipulator's second link passes over the first. The movement of the arm in physical space is shown in Figure 8.

E. Hall and Rooms

In this final simulation there is a hallway connecting four rooms, each of size 6 m by 11 m. The grid is rectangular but not uniform. In the center of the rooms the grid spacing is 1 m but in the hallway, around the edges of each room, and near the doorways the grid spacing is 0.5 m, as shown in Figure 9. Each room contains one target moving to random points in the room

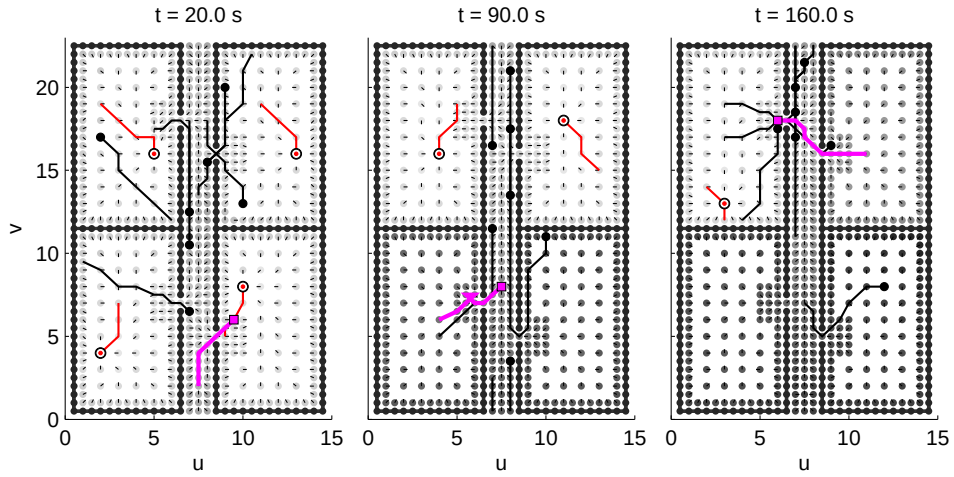

Fig. 9. Hall and Rooms. Three time snap shots of the environment are shown. Symbols as in Figure 2. Moving obstacles are shown as solid black circles and move up and down the hall randomly visiting two locations in a room. Initially each room contains one target and the robot starts at the south end of the hall. The previous 15 seconds of target, robot and obstacle paths are shown in each plot as solid lines. At $t = 20$ the robot has entered the south-east room and is chasing down the target there. By $t = 90$ the robot has captured the target in the south-west room and has returned to the hall heading to the north-east room. By $t = 160$ the target in the north-east room has been captured and the robot is entering the north-west room to capture the final target.

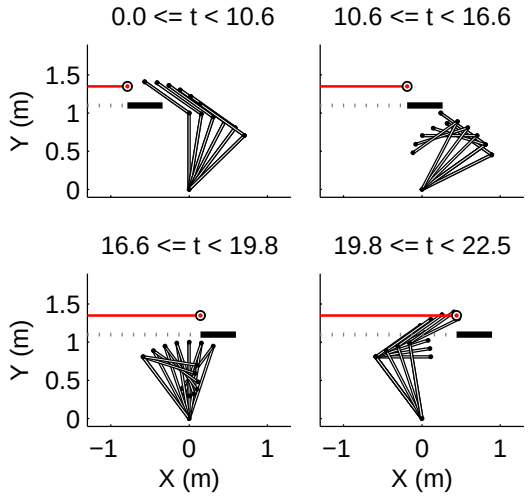

Fig. 8. Manipulator arm physical space showing the arm, barrier and target movement through time.

at a speed of 0.25 m/s. In addition to the permanent walls, there are moving obstacles ("people") which primarily move up and down the hallway randomly visiting several points in a room before returning to the hall and moving on. These obstacles move at a speed of 0.5 m/s. The robot starts at the south end of the hall and moves at a speed of 0.375 m/s. The robot's goal is to capture all four targets while avoiding all obstacles. The penalty function is given by (7) with $A = 10$ and $B = 2\sqrt{2}$.

Figure 9 shows three snapshots of this simulation. The locations of the robot, targets, and moving obstacles in the 15 seconds prior to the indicated time are shown as solid lines. At $t = 20$ the robot has entered the southeast room and is chasing down the first target. By $t = 90$ it has captured the target in the southwest room and is returning to the hall. The robot moves down the center of the hall since the moving obstacles tend to move along the sides of the hall. At $t = 160$ the robot has captured the target in the north-east room and successfully crossed a very crowded hallway and entered the north-west room. The final target is captured at $t = 170.7$. A video (avi file) of this simulation is attached.

IV. Sufficient Conditions for Capture

In [1] it was proved in the case where obstacles are static, that the algorithm always results in the robot catching a target provided the system update frequency, f_u, (the number of times the entire grid system is updated in each unit of real time) satisfies

$$f_u > \frac{3}{d_{\min}\left(\frac{1}{v_t} - \frac{1}{v_r}\right)} \tag{9}$$

where $d_{\min}$ is the minimum distance between any two neighboring grid points, and v_t and v_r are the target and robot speeds respectively. The sufficient condition (9) can be extended to the dynamic obstacle situation with some restrictions on the obstacle movement. In an unrestricted moving obstacle situation, no matter how fast the robot speed and the system update frequency, and no matter how slow the obstacles move, it is always possible to construct an environment where the target cannot be captured [1]. As an obstacle moves (or extends) from one grid point to a neighbor which is on the current optimal path from the robot to the target, the length of the path that the robot is taking toward the target may increase (either because the optimal path is displaced one grid point or the optimal path is completely occluded and a new optimal path to the target which goes around the obstacle is found). If the number of times that obstacle movements cause increases in the length of the optimal path is finite, then (9) will also be a sufficient condition for capture of the target in the fully dynamic environment.

As noted in Section III-A, the new algorithm with obstacle clearance via the local penalty function, can result in situations

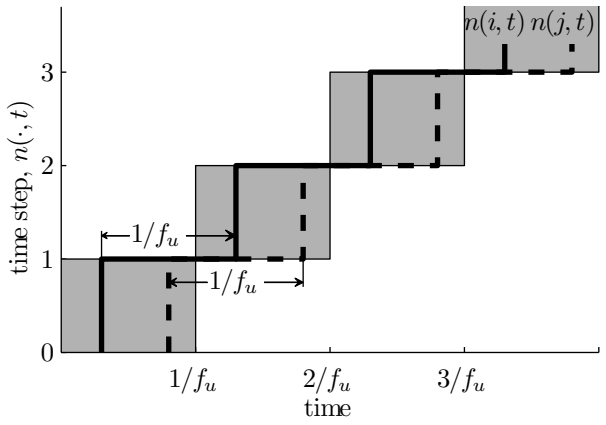

Fig. 10. The time step for all grid points as a function of time. The shaded areas are where $n(i,t)$ lies for all i. Two particular curves (thick solid and dashed lines) are also plotted.

where, despite moving significantly faster than the target, the penalty function may force the robot to take a longer "safe" path and thus never catch the target. Here we re-derive (9) to show how the penalty function is incorporated into the sufficient condition for capture.

First consider $n(i,t_k)$ as defined in Section II-C, the highest time step for which y_i has been computed by time t_k. If f_u is the system update frequency, then $n(i,t)$ is as shown in Figure 10. As can be seen from this figure, for all t, $n(i,t)$ satisfies

$$n(i,t+k/f_u) = n(i,t)+k, \quad \forall\, k\in\mathbb{Z} \tag{10}$$

$$n(i,t+g/f_u) \le n(i,t)+1, \quad 0<g<1, \tag{11}$$

$$n(j,t) \le n(i,t)+1, \quad \forall\, i,j. \tag{12}$$

Consider now a robot moving along an optimal path, leaving from point i at time t_k and arriving at a neighbor j at time $t_{k+1} = t_k + d_{ij}/v_r$. We have at departure, since $j = p_i^y\big(n(i,t_k)\big)$,

$$\begin{aligned} y_i(n(i,t_k)) &= y_j(n(i,t_k)-1) + d_{ij}\Big(1+q\Big(x_i\big(n\,(i,t_k)\big)\Big)\Big) \\ &\ge y_j(n(i,t_k)-1) + d_{ij} \end{aligned} \tag{13}$$

and at arrival, by (10)–(12),

$$\begin{aligned} y_j\left(n(j,t_{k+1})\right) &= y_j\big(n\left(j,t_k+d_{ij}/v_r\right)\big) \\ &= y_j\big(n\left(j,t_k+\mathrm{rem}[d_{ij}f_u/v_r]\,/f_u\right. \\ &\qquad \left.+\,\lfloor d_{ij}f_u/v_r\rfloor\,/f_u\right)\big) \\ &= y_j\big(n\left(j,t_k+\mathrm{rem}[d_{ij}f_u/v_r]\,/f_u\right) \\ &\qquad +\lfloor d_{ij}f_u/v_r\rfloor\big) \\ &\le y_j\big(n\left(j,t_k\right)+1+\lfloor d_{ij}f_u/v_r\rfloor\big) \\ &\le y_j\big(n\left(i,t_k\right)+2+\lfloor d_{ij}f_u/v_r\rfloor\big) \end{aligned} \tag{14}$$

where $\mathrm{rem}[w] = w - \lfloor w\rfloor$. Comparing (13) and (14) we see that we must consider at most $1+2+\lfloor d_{ij}f_u/v_r\rfloor$ updates of y_j. This corresponds to a real time of

$$\Delta t = \big(3+\lfloor d_{ij}f_u/v_r\rfloor\big)/f_u,$$

and in this time, the target can have moved at most a distance $v_t\Delta t$ further away from the robot. In the worst case, all of this extra distance is alongside obstacles incurring maximum penalty, so that y_j cannot have increased in value more than

$$v_t\Delta t\,(1+q_{\max})\,.$$

Here we have ignored the time taken for the propagation of the signal back from the new target location, which, if included would reduce this upper bound. Thus a sufficient condition for $y_j(n(j,t_{k+1}))$ to be smaller than $y_i(n(i,t_k))$ is

$$v_t\big(3+\lfloor d_{ij}f_u/v_r\rfloor\big)\,(1+q_{\max})\,/f_u < d_{ij},$$

which, since $\lfloor d_{ij}f_u/v_r\rfloor \le d_{ij}f_u/v_r$, will be satisfied if

$$\frac{3}{f_u} < d_{ij}\left(\frac{1}{v_t\,(1+q_{\max})} - \frac{1}{v_r}\right).$$

If the target speed is faster than the robot speed or if $q_{\max}$ is too large then the term in parantheses above will be negative and consequently the inequality cannot be met. This does not mean the robot will fail to reach the target, only that this sufficient condition is not applicable to the situation. If the term in parantheses is positive, a sufficient condition for y to be strictly decreasing along the robot's path is

$$f_u > \frac{3}{d_{\min}\left(\frac{1}{v_t(1+q_{\max})} - \frac{1}{v_r}\right)}. \tag{15}$$

Since the grid is finite, y can only take on a finite number of values, hence strict decreasing along the path implies y eventually reaches zero or the value $q(x)d_{\min}$, and thus reaches the target.

Comparing (15) with (9) we see that the addition of penalties for being close to obstacles effectively causes a increase in the target speed by a factor of $(1+q_{\max})$. This represents the worst case scenario where the target travels through a maximum penalty area, directly away from the robot, and the robot is traveling outside the obstacle safety margins. We emphasize that this is just a sufficient condition, and we would expect the robot to capture the target in many situations where this condition is not met, including many situations where y is not strictly decreasing along the robot's path. Indeed, for all of simulations in Section III, this condition is not met since the $q_{\max}$ values are quite large, yet in all cases the robots capture the targets.

V. Conclusion

The distance-propagating dynamic system algorithm for robot path planning in dynamic environments has been extended to the situation where the optimal paths are not simply the shortest path from a robot to a target, but paths which minimize a cost function based on both distance to a target and proximity to obstacles. The algorithm works in real time and requires no prior knowledge of target or obstacle movements. Updating the distance values at each grid point is done without any global knowledge, neither of distances at non-neighboring points nor the update history of any point. Thus each point could be implemented as an independent processor with only local connections to its neighbors. This feature makes it easy to

implement the algorithm on a parallel architecture. The robot path is determined in real-time completely from information obtained from the robot's current grid point location. The computational effort to update each point is minimal allowing for rapid propagation of the distance information outward along the grid from target locations.

The algorithm can cope well with situations where there are many distant obstacles moving and this information is to be incorporated into the path-determination algorithm. This feature distinguishes it from the D* algorithm which is most efficient when alterations to the environment are primarily local to the robot's current location. Indeed, the simulation in Section III-C showed that in a very highly cluttered environment, the D* algorithm may fail to propagate any information to the robot at all, whereas our algorithm will always bring information and direct the robot toward the target.

The Hall and Rooms simulation (Section III-E) suggests a modification of the penalty function based on directionality of the moving obstacles. Clearly, when people navigate a crowded hallway, they anticipate where others are going and avoid stepping in front of others but do not hesitate to pass behind them, often at very close quarters. Such behavior requires an estimation of the moving obstacle's velocity, and not just its position. Work along these lines has been done by Shiller and others [21], [22]. Fraichard and others [23], [24] have used obstacle velocity information to define "inevitable collision states" which must be avoided by the robot. Such a behavior could possibly be mimicked with our algorithm by modifying the penalty function to take into account both the distance to the nearest obstacle and whether this distance is increasing or decreasing. In the decreasing case, the penalty should be larger. This would effectively make paths cutting in front of moving obstacles more costly than those going behind them. However, such a simple modification would not be effective for points near fixed obstacles, since their distance to an obstacle is unaffected by other moving obstacles. Also, there is the question as to how long one should consider back in time to determine whether the distance to an obstacle has increased or decreased. It would not make sense to base this simply on the last update, since the system is updating at a rapid rate and hence this information would very rapidly be lost. In order to base this on the speed of the obstacles, each grid point would also have to have access to a clock. More work in this direction is required.

Kinematic constraints of the robot have not been considered here. Combining this path planning algorithm with measures which address the kinematic constraints is a requirement before appropriate implementation on a real robot can succeed. The same statement is true for the D* algorithm. In addition, as with all grid-based approaches, the resulting path is not smooth. Recent work by Philippsen [14] has extended the D* algorithm to E* using interpolation schemes to generate smooth paths. As our algorithm is computing the same information as D* but in a different way, interpolation schemes could also be applied to this algorithm to generate smooth trajectories.

A sufficient condition for the robot to capture the target was derived and was shown to be the same as for the algorithm without safety considerations but with the target speed effectively increased by a factor of $(1 + q_{\max})$, where $q_{\max}$ is the maximum value for the penalty function, that is, the value of the penalty function at points that are a distance $d_{\min}$ away from obstacles. Although for high penalty functions the sufficient condition is not likely to be met, we still expect the target to be captured in many of these situations.

The computationally local nature of our algorithm distinguishes it from most algorithms. For example, A* and D* sort their open lists using knowledge of the function values at all grid points, and focused D* uses global information to estimate the distance to the robot. Thus global information is used when updating each point, or determining the order of updating. In contrast, our algorithm uses only local information when computing an update at each point, and the order of updating is pre-determined. This makes our algorithm exceedingly easy to parallelize by simply assigning a subset of points to each processor. Neural network approaches are also local in this sense, however, our algorithm is faster than most of these approaches which generally require numerical integration of a nonlinear differential equation.

Many robot path-planning algorithms do not easily extend to the dynamic situation and become computationally expensive when the environment is complex. Our algorithm in contrast is designed for the dynamic situation and is not at all affected by the complexity of the environment as far as computational speed is concerned.

REFERENCES

[1] A. R. Willms and S. X. Yang, "An efficient dynamic system for real-time robot path planning," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 36, no. 4, pp. 755–766, 2006.

[2] A. Zelinsky, "Using path transforms to guide the search for findpath in 2D," *Intl. J. of Robotics Research*, vol. 13, no. 4, pp. 315–325, 1994.

[3] G. Borgefors, "Distance transformations in digital images," *Comput. Vision, Graphics, Image Proc.*, vol. 34, pp. 344–371, 1986.

[4] F. Y.-C. Shih and O. R. Mitchell, "A mathematical morphology approach to Euclidean distance transformation," *IEEE Trans. on Image Processing*, vol. 1, no. 2, pp. 197–204, 1992.

[5] S.-C. Pei and J.-H. Horng, "Finding the optimal driving path of a car using the modified constrained distance transformation," *IEEE Trans. Robot. Automat.*, vol. 14, no. 5, pp. 663–670, Oct. 1998.

[6] E. V. Denardo, *Dynamic Programming: Models and Applications.* Englewood Cliffs, NJ: Prentice Hall Inc., 1982.

[7] R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image analysis using mathematical morphology," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-9, pp. 532–550, July, 1987.

[8] C. R. Giardina and E. R. Dougherty, *Morphological Methods in Image and Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. on Systems, Science and Cybernetics*, vol. SSC-4, pp. 100–107, 1968.

[10] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proc. of the IEEE Internat. Conf. on Robotics and Automation*, 1994, pp. 3310–3317.

[11] ——, "Optimal and efficient path planning for unknown and dynamic environments," Carnegie Mellon University Robotics Institute, Tech. Rep. CMU-RI-TR-93-20, Aug. 1993.

[12] ——, "The focused D* algorithm for real-time replanning," in *Proc. of the 1995 Internat. Joint Conf. on Artificial Intelligence*, 1995, pp. 1652–1659.

[13] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," in *IEEE Transactions on Robotics*, vol. 21, no. 3, 2005, pp. 354–363.

[14] R. Philippsen, "Motion planning and obstacle avoidance for mobile robots in highly cluttered dynamic environments," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, 2004, http://library.epfl.ch/en/theses/?nr=3146.

[15] R. Glasius, A. Komoda, and S. C. A. M. Gielen, "Neural network dynamics for path planning and obstacle avoidance," *Neural Networks*, vol. 8, no. 1, pp. 125–133, 1995.

[16] E. Zalama, P. Gaudiano, and J. L. Coronado, "A real-time, unsupervised neural network for the low-level control of a mobile robot in a nonstationary environment," *Neural Networks*, vol. 8, pp. 103–123, 1995.

[17] S. X. Yang and M. Meng, "An efficient neural network method for real-time motion planning with safety consideration," *Robotics and Autonomous Systems*, vol. 32, no. 2–3, pp. 115–128, 2000.

[18] ——, "Neural network approaches to dynamic collision-free robot trajectory generation," *IEEE Trans. on Systems, Man, and Cybernetics, Part B*, vol. 31, no. 3, pp. 302–318, June, 2001.

[19] A. Adamatzky, P. Arena, A. Basile, R. Carmona-Galán, B. D. L. Costello, L. Fortuna, M. Frasca, and A. Rodríguez-Vázquez, "Reaction-diffusion navigation robot control: From chemical to VLSI analogic processors," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 51, no. 5, pp. 926–938, 2004.

[20] D. V. Lebedev, J. J. Steil, and H. J. Ritter, "The dynamic wave expansion neural network model for robot motion planning in time-varying environments," *Neural Networks*, vol. 18, pp. 267–285, 2005.

[21] Z. Shiller, F. Large, and S. Sekhavat, "Motion planning in dynamic environments: Obstacles moving along arbitrary trajectories," in *Proc. of the IEEE Int. Conf. on Robotics and Automation, Vol. 4*, 2001, pp. 3716–3721.

[22] F. Large, S. Sekhavat, Z. Shiller, and C. Laugier, "Towards real-time global motion planning in a dynamic environment using the nlvo concept," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, Vol. 1*, 2002, pp. 607–612.

[23] T. Fraichard and H. Asama, "Inevitable collision states. a step towards safer robots?" in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, Vol. 1*, 2003, pp. 388–393.

[24] S. Petti and T. Fraichard, "Safe motion planning in dynamic environments," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2005, pp. 2210–2215.