2011 IEEE International Conference on Robotics and Automation
Shanghai International Conference Center
May 9-13, 2011, Shanghai, China

SIPP: Safe Interval Path Planning for Dynamic Environments

Mike Phillips* and Maxim Likhachev*
* Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract— **Robotic path planning in static environments is a thoroughly studied problem that can typically be solved very efficiently. However, planning in the presence of dynamic obstacles is still computationally challenging because it requires adding time as an additional dimension to the search-space explored by the planner. In order to avoid the increase in the dimensionality of the planning problem, most real-time approaches to path planning treat dynamic obstacles as static and constantly re-plan as dynamic obstacles move. Although gaining efficiency, these approaches sacrifice optimality and even completeness. In this paper, we develop a planner that builds on the observation that while the number of safe timesteps in any configuration may be unbounded, the number of safe time intervals in a configuration is finite and generally very small. A safe interval is a time period for a configuration with no collisions and if it were extended one timestep in either direction, it would then be in collision. The planner exploits this observation and constructs a search-space with states defined by their configuration and safe interval, resulting in a graph that generally only has a few states per configuration. On the theoretical side, we show that our planner can provide the same optimality and completeness guarantees as planning with time as an additional dimension. On the experimental side, in simulation tests with up to 200 dynamic obstacles, we show that our planner is significantly faster, making it feasible to use in real-time on robots operating in large dynamic environments. We also ran several real robot trials on the PR2, a mobile manipulation platform.**

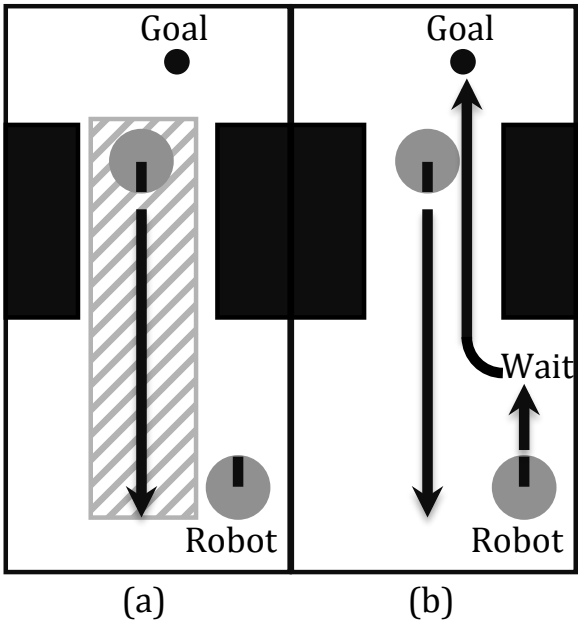

Fig. 1. (a) Treating the dynamic obstacle as a static one results in no solution, (b) Planning with time finds a solution by waiting for the obstacle to pass and then proceeding

I. INTRODUCTION

Whether it be autonomously driving a vehicle or performing household jobs, such as cleaning a room, almost all tasks that robots perform assume the ability to safely navigate from place to place in the presence of moving objects such as people, pets, cars, etc. In order to accomplish this, robots need to be able to predict where these dynamic obstacles will be moving in the near future. They need to plan short paths to their goals that do not endanger or inconvenience people. Robots also need to be able to plan these paths very quickly, in the likely situation that a dynamic obstacle's trajectory is not what the robot predicted, and a new plan must be created to prevent collision. Being able to plan paths efficiently will allow robots to be more responsive and robust to a constantly changing environment.

When planning in dynamic environments, adding a time dimension to the state-space is needed in order to properly handle moving obstacles, shown in Figure 1(b). However, the large increase in the number of states to be searched causes planning times to be much longer. Since the environment is constantly changing, plans need to be generated quickly or they will be out of date before they can even be used. Therefore, for practical purposes, a common approach to producing faster plans is to treat the environment as though it were static [5], [9]. This is done by turning each dynamic obstacle (and sometimes its predicted trajectory for the near future) into a static obstacle. This ensures that the plan generated will not be allowed to collide with the dynamic obstacle in the near future. However, this approach suffers from suboptimality in cases where the robot could have crossed a trajectory without being hit, or just waited until the obstacle passed and then crossed. Instead it takes a long path around the trajectory of the obstacle. There are even cases when this approach will fail to find a solution at all, such as when a dynamic obstacle's trajectory goes through or crosses in front of a doorway that the robot must also use in order to reach its goal, shown in Figure 1(a).

In this paper we propose a method that exploits the observation that the number of contiguous safe intervals is generally significantly smaller than the number of timesteps that compose those intervals. For example, a configuration that no dynamic obstacles pass through only has one safe interval, which spans from the start time to infinity. With

978-1-61284-385-8/11/$26.00 ©2011 IEEE

this in mind, we develop a planner that uses states defined by configuration (x,y,θ,etc) and the safe interval as independent variables, while only storing the actual timestep as a dependent variable. A configuration is the set of non-time variables that describe the robot's state, such as position, heading, joint angles, etc. An independent variable is one that is used to actually identify a state, while a dependent variable is stored with the state but is not used to identify it. Specifically, two states with the same values for their independent variables, but different dependent ones, are still considered the same state, by the planner. We provide theoretical guarantees that the states we eliminate from consideration cannot be part of the optimal solution with respect to the time it takes to traverse the path. As a result, our planner can guarantee optimality, while avoiding the increase in dimensionality of the problem. We demonstrate the efficiency in simulation with experiments on randomly generated indoor and outdoor environments, showing a significant speed increase over a planner that represents states with configuration and timestep as independent variables. Using the PR2 robot, we also demonstrate that the planner is suitable for real-time use.

II. RELATED WORK

Most of the approaches to dealing with dynamic obstacles, model them as static obstacles with a short window of high cost around the beginning of their projected trajectories [5], [9]. While efficient, these approaches suffer from potential high suboptimality and even incompleteness, as described in our introduction. Another common approach is to only consider dynamic obstacles (still treating them as static obstacles) while executing the path, using local obstacle avoidance methods [2]. This method can get stuck in local minima and is not globally optimal. Another alternative is for the local planner to use velocity obstacles, which determine the controls that lead to collision with moving obstacles [13]. While this is more accurate, it still can lead to local minima as it greedily minimizes the difference between the desired control without dynamic obstacles, and the set of feasible controls that are not in velocity obstacles.

Some approaches plan in the full space-time search space [10]. Silver's HCA* algorithm is designed for planning for multiple robots, but in the paper, he points out that it can be applied to planning in dynamic environments. In dynamic environments, HCA* provides the same guarantees on optimality and makes the same assumptions that we do. In our experimental results, we show a comparison against the HCA* algorithm and show a significant speed up. The reason our approach is faster is because our search space is much smaller. HCA* and other planners that use a time dimension have a state for every (configuration, timestep) pair and since the number of timesteps is usually large the search space is also large. Our algorithm groups contiguous, collision-free timesteps into safe intervals, and then represents each state by a (configuration, safe interval) pair. The maximum number of safe intervals for any given configuration is at most the number of dynamic obstacles whose trajectories intersect in that configuration. Therefore, the number of safe intervals is significantly smaller than the number of timesteps for the same configuration. As a result, the search space our planner constructs is much smaller, leading to faster planning and smaller memory requirements.

The approach [12] is similar to ours in that it recognizes that a state is only needed for the earliest arrival time in a safe interval. The major difference is how we evaluate an edge from one interval to another. In [12], an "expand" is a partial move of a "probe" along an edge via a single timestep which then must be replaced back into the queue. This means that for each edge a "probe" goes back into the queue many times (and queue operations are the expensive part of the A* search) instead of just once as it is for our planner. Essentially, the approach in [12], has taken each cost-n edge and converted it into n cost-1 edges, which raises the asymptotic runtime to be a function of the edge weights as well as the number of vertices and edges. Nevertheless, [12] is very closely related to our approach.

Planning with time, required for dealing with dynamic obstacles, is hard to perform on-line, since constant demand for re-planning enforces tight constraints on execution cycle. To address the real-time constraints, a number of approaches have been proposed that sacrifice near-optimality guarantees for the sake of efficiency [3], [11]. Our proposed approach differs in that we aim for computing paths that are optimal with respect to shortest time. Some other approaches use RRT-variants to plan quickly in higher-dimensional search spaces that can handle the kinodynamic constraints of more complex robots [7], [1]. However, these sampling-based approaches cannot provide the guarantees on optimality that we strive for. There have also been approaches that plan with the added time dimension only until the end of an obstacle's trajectory (or when the uncertainty about the obstacle is too large) and then finish the plan in the simpler static state-space [4]. In this Time-Bounded Lattice approach, the dynamic obstacles and the time dimension are dropped in the search space, after a certain point in the time dimension. This sacrifices optimality. Our algorithm does not prune dynamic obstacle trajectories at all, making it optimal with respect to the entire given trajectories.

III. ALGORITHM

We define a safe interval as a contiguous period of time for a configuration, during which there is no collision and it is in collision one timestep prior and one timestep after the period. The obvious exception to this is that the last safe interval for a configuration may go until infinity, if a dynamic obstacle never again is predicted to pass through this configuration. A collision interval is the opposite of a safe interval. A collision interval is a contiguous period of time for a configuration, where each timestep in this period is in collision with a dynamic obstacle and it is safe one timestep prior and one timestep after the period. Each spatial configuration (such as the one shown in Figure 2) has a timeline (Figure 3), which is just an ordered list of intervals, alternating between safe and collision. Clearly, there cannot be two safe intervals in a row in the timeline because that

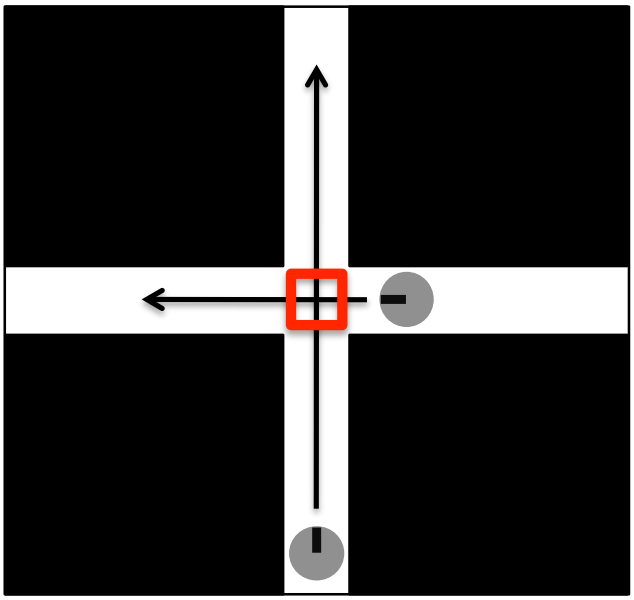

Fig. 2. An environment with dynamic obstacles and a highlighted configuration

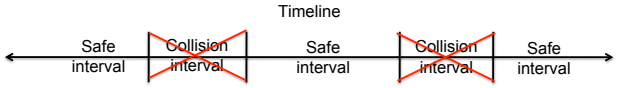

Fig. 3. A timeline for the highlighted configuration in Figure 2

would violate the definition of a safe interval, since a safe interval should be bounded by collision intervals or infinity. The same argument applies for collision intervals. Using safe intervals to represent the time dimension of the search space is what makes our algorithm efficient, and is the key idea to our approach. A single state, represented by a (configuration, interval) pair, replaces what used to be many states, one for each timestep in the safe interval. This makes the state space significantly smaller, allowing us to search for a solution in far less time.

Dynamic Obstacle Representation Our algorithm assumes there is another system that tracks dynamic obstacles in the environment, predicts their future trajectories, and formats them into a general representation we define. We are given a list of dynamic obstacles, where each obstacle has a radius and a trajectory. A trajectory is just a list of points, where each point has state variables, specifying its configuration, time, and some measure of the point's uncertainty. The points in the trajectory list are ordered from earliest time to latest time, so by reading the points in order, it can be seen how the obstacle is predicted to move in the near future. Our representation also allows for dynamic obstacles to have more than one possible future trajectory. However, while the algorithm described in this paper applies to multiple hypothesis trajectories, for simplicity, we will assume each obstacle only has one trajectory.

Notations and Assumptions We will now introduce some notation used to explain the algorithm. A state s has a variable, $g(s)$, which is the cost of best known path from

```
1  g(s_start) = 0; OPEN = ∅;
2  insert s_start into OPEN with f(s_start) = h(s_start);
3  while(s_goal is not expanded)
4    remove s with the smallest f-value from OPEN;
5    successors = getSuccessors(s);
6    for each s' in successors
7      if s' was not visited before then
8        f(s') = g(s') = ∞;
9      if g(s') > g(s) + c(s, s')
10       g(s') = g(s) + c(s, s');
11       updateTime(s');
12       f(s') = g(s') + h(s');
13       insert s' into OPEN with f(s');
```

Fig. 4. A* with safe intervals

the start state to s. The heuristic function $h(s)$ is an estimate of the cost from s to the goal state. We will be assuming the heuristic function is consistent, meaning that it never overestimates the cost to the goal and it satisfies the triangle inequality (without this our approach our approach loses completeness). The cost of a transition or edge from s to one of its successors s' is defined by $c(s, s')$.

The main assumptions that our algorithm makes are:

- $c(s, s')$ = time to execute the action from s to s'. In other words, the goal of the planner is to find a time-minimal trajectory.
- The robot is capable of waiting in place (this assumption would not be true of a motorcycle).
- Inertial constraints (acceleration/deceleration) are negligible. The planner assumes the robot can stop and accelerate instantaneously.

A. Planning with Safe Intervals

Graph Construction When the planner is initialized, we create a timeline for each spatial configuration, using the predicted dynamic obstacle trajectories[1]. This is done by iterating through each point along the trajectory of each dynamic obstacle and updating the timelines for all the configurations within collision distance of the point. This collision distance is the sum of the obstacle's radius and the radius of the robot. (The radius may also be inflated appropriately with respect to the uncertainty associated with the point.)

Graph Search After the initialization, we run the A* search, shown in Figure 4, which runs as usual except for how it gets the successors of a state (Figure 5) and how it updates the time variable for states (line 11, Figure 4).

In Figure 5, the function $M(s)$ returns the motions that can be performed from state s. These motions indicate how they change the spatial variables of a state. The motions also have an amount of time that it takes to execute them, which we will use to help determine what safe intervals we can get to in the resulting configuration. The $startTime(i)$ returns the start time of safe interval i and $endTime(i)$, the end time of safe interval i.

When a state s is expanded, we generate successors for it, shown in Figure 5. For each of the motions that our robot can perform from s, we compute the resultant configuration

[1]On the implementation side, we only generate and store timelines for configurations that the planner examines. This avoids memory requirements to be on the order of the size of the configuration space.

```
1  getSuccessors(s)
2    successors = ∅;
3    for each m in M(s)
4      cfg =configuration of m applied to s
5      m_time =time to execute m
6      start_t = time(s) + m_time
7      end_t = endTime(interval(s)) + m_time
8      for each safe interval i in cfg
9        if startTime(i) > end_t or endTime(i) < start_t
10           continue
11       t =earliest arrival time at cfg during interval i with no collisions
12       if t does not exist
13         continue
14       s' = state of configuration cfg with interval i and time t
15       insert s' into successors
16   return successors;
```

Fig. 5. getSuccessors

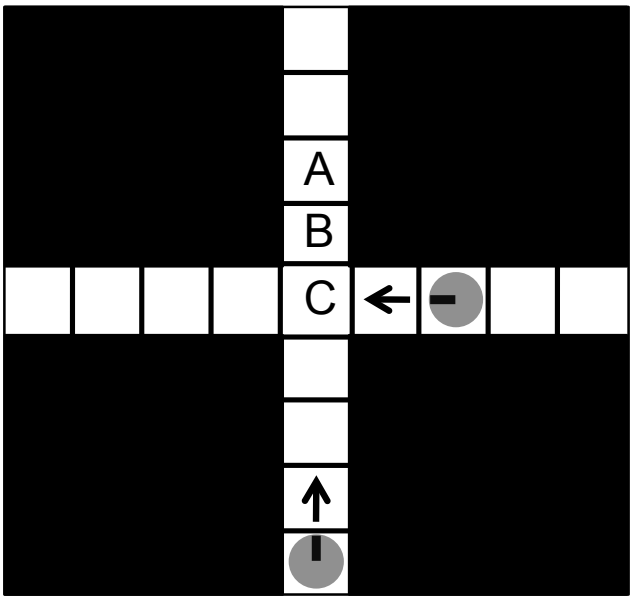

Fig. 6. An example environment with two dynamic obstacles moving at a speed of one cell per timestep. There are three highlighted configurations (A,B,C) and the robot is located at configuration B.

and the time of execution (line 3-5, Figure 5). Then for each of the time intervals in this new configuration, we generate a successor with the earliest possible arrival time that does not have any collisions along the motion. When generating successors, we say that s uses "wait and move" actions. This means that for each safe interval in the new configuration, we wait the minimal amount of time possible, so that when the move to the new configuration is used, we arrive in the new safe interval as early as possible. The arrival time is stored with the state, but the state will not be identified by its time value, only by its other dimensions.

During A*, whenever a shorter path to s' is found, the cost is replaced with the smaller one. Similarly, when this happens it corresponds to arriving at s' at an earlier time (since cost is equal to time) and so we also replace the time value stored in state s' with this new shorter time (line 10-11, Figure 4). We are guaranteed that when s' is expanded we have found the earliest time that we can arrive at s'. This allows us to safely generate and set the time for the successors of s'.

Figures 6 and 7 show an example of an expansion under the SIPP algorithm. In Figure 6, we have two gray dynamic

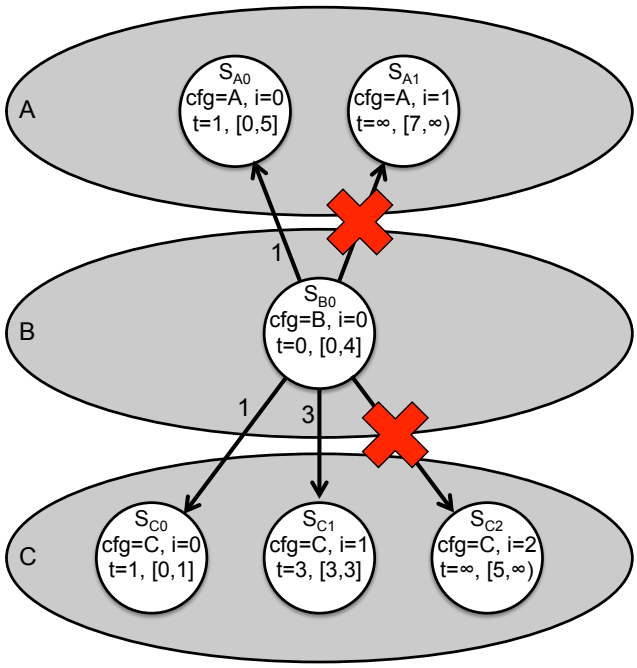

Fig. 7. An illustration of an expansion of state S_{B0}. Each white circle in this figure represents a state. The first line in each state is its name. The second line defines the state by indicating its configuration (cfg) from Figure 6 and its safe interval (i). The third line shows the earliest known time we can reach this state (t) and the safe interval [a,b]. The gray ovals group states by their configuration. The arrows indicate transitions from one state to another, labeled with their cost. The red X indicate invalid transitions due to collision with a dynamic obstacle.

obstacles moving in the direction indicated by their arrows at a speed of one cell per timestep. We have three labeled configurations (A,B,C) that we will examine. The robot is located at configuration B and can move one cell per timestep in a 4-connected fashion. In Figure 7, we show the graph during the expansion of the robot's current state, S_{B0}. The other state in configuration B (S_{B1}) is not pictured because you cannot transition between safe intervals within the same configuration because the robot would have to wait through the separating collision interval (making it irrelevant). In configuration B, there are two valid motions the robot can perform on a 4-connected grid (up to configuration A and down to configuration C).

There are two possible successors in A. The robot could move immediately to A and arrive in safe interval 0, one timestep later (S_{A0}). The other successor in A is at safe interval 1 (S_{A1}), however this interval starts at time 7 and we can only safely wait in S_{B0} until time 4 (after which we would get hit by the upward moving dynamic obstacle), making this transition invalid. This case is handled on lines 9-10 in Figure 5. So for the upward motion we would add only S_{A0} to our set of successors. The time (t) for S_{A0} is 1 because it takes one timestep to move one cell, while the t for S_{A1} is ∞ because we haven't found any valid paths to that state yet.

For the downward motion to configuration C we have three possible successors. The robot could move down immediately to C and arrive in safe interval 0 (S_{C0}) one timestep

later (before both dynamic obstacles reach C). Another successor is S_{C1}, which is available only during timestep 3 (after the first dynamic obstacle has passed, but before the second one arrives). To arrive at time 3, the robot would have to wait 2 timesteps before moving to C (which takes one timestep). This is an example of our "wait and move" actions. This transition has a cost of 3, since it takes that much time to execute. The final possible successor is S_{C2}, which is the state below the robot (C) after the upward moving dynamic obstacle has passed. The safe intervals align such that the robot could wait until time 4 and then move to C, arriving at 5, within safe interval 2. However, the collision check on lines 11-13 in Figure 5 would return no valid transition in this case. This collision check interpolates between the two states the robot moves between and in this case would detect that the robot and the dynamic obstacle are switching places by passing through each other. Therefore, S_{C2} is not a valid successor.

So from the expansion of S_{B0}, the set of successors is (S_{A0}, S_{C0}, S_{C1}).

B. Theoretical Analysis

Here we sketch the proofs for completeness and optimality of our algorithm.

Theorem 1: Arriving at a state at the earliest possible time guarantees the maximum set of possible successors.

A state s is defined by a configuration and a safe interval. Let t_0 and t_1 be two times within the safe interval of s, such that $t_0 < t_1$. Let A_1 be the set of successors generated, if we expanded s from time t_1 and let A_0 be the set of successors generated, if we expanded from time t_0. Since s exists in a safe interval, the robot can wait from any time in the interval to any later time in the interval, before moving. Therefore, A_0 is a superset of A_1, since it can wait until t_1 and later to get the successors in A_1, but it may also have more successors from earlier than t_1. This is illustrated in Figure 8. Therefore, if a state is expanded at the earliest possible time, we are guaranteed to have the largest set of successors. This is why the algorithm is still complete, even though it only has one state per safe interval. By having that state be the earliest time possible in the interval and using "wait and move" actions, we can still generate all the successors that would be generated by an algorithm that plans with timesteps. This theorem leads to the proof of completeness.

Theorem 2: When the safe interval planner expands a state in the goal configuration, it has found a time-minimal, collision-free path to the goal.

In optimal A*, when a state is expanded, its g-value is minimal. In this planner, the cost on an edge is equal to the time it takes to execute that edge and whenever a g-value is updated (from finding a shorter path), the time value is also updated to the earlier time. Therefore, when a state (defined by configuration and safe interval) is expanded it is also the earliest possible time to arrive in this safe interval. So when a state in the goal configuration is expanded, it is also the earliest time we can arrive at the goal configuration. This is optimal with respect to our time-minimal cost function. We also know that all states exist within safe intervals, which makes the entire path collision-free, from the start state to the goal state.

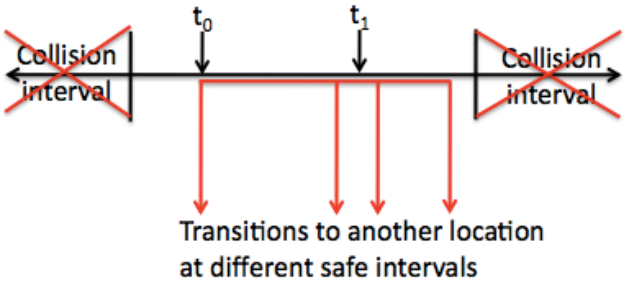

Fig. 8. The set of successors generated from t_0 is a superset of the set of successors generated from t_1

Theorem 3: If the configuration with the most dynamic obstacles passing through it has n such occurrences, then each configuration can have at most $n+1$ safe intervals.

Since every collision interval is followed by a safe interval and the configuration has n collision intervals then the configuration already has n safe intervals. If the configuration also starts out safe then there is one more safe interval before the first collision interval, making $n+1$ safe intervals. Since this is the configuration with the most collision intervals we can say for all configurations in the environment, the number of safe intervals is no more than $n+1$.

This shows why usually the maximum number of safe intervals is very small, since it isn't common for large numbers of dynamic obstacles to pass through the same configurations.

IV. EXAMPLE

Here we will go over a small example showing how the algorithm works. Figure 9 shows the initial environment and the first 3 actions in the optimal plan. Figure 9(a) shows the initial environment we will be planning on. The robot starts in the cell marked R and it has a goal in the cell marked G. The dynamic obstacle on the right is moving to the left and the dynamic obstacle at the bottom is moving upward. The dynamic obstacles both move at one cell per timestep. The robot moves on a 4-connected grid at a speed of one cell per timestep and can also wait in place. When the planner expands the starting state, it will generate successors in the cells above and below the current state. The cell below the robot's initial position, c has 3 safe intervals and therefore, 3 states. From the start state, we cannot generate a state in c during the first interval, because by the time we move there, we would be in collision with the first dynamic obstacle. We can generate a state in the second safe interval, after the first dynamic obstacle passes. The earliest time we can do so is by waiting one time step and then moving down. This is a single "wait and move" action. We cannot generate a state in the third safe interval, because we would be hit by the bottom dynamic obstacle before we could arrive there.

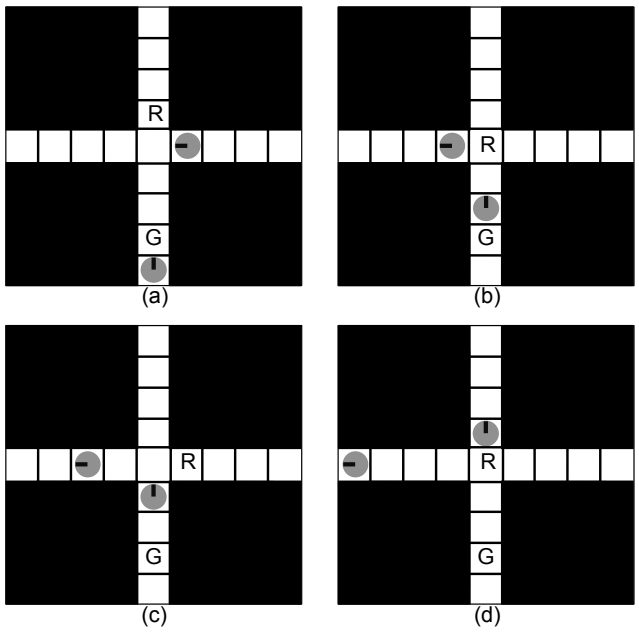

Fig. 9. (a) The initial environment with the robot R trying to get to goal G. The dynamic obstacle on the right moves to the left and the one at the bottom moves up. The dynamic obstacles and the robot can move one cell each timestep. (b) The optimal plan's first step is to wait one timestep and move down. (c) Then the plan moves to the right with no waits. (d) The third action of the plan waits a timestep and then moves to the left.

Figure 9(b) shows the environment after the optimal plan's first action. The cell to the right of the robot, c, has only one safe interval. Moving right immediately, and waiting one timestep and then moving right are both safe moves. Our algorithm only stores one state per (configuration, interval) pair and that is the one with the earliest time reachable. Therefore, moving right immediately is the only action considered for the cell to the right.

Figure 9(c) shows the environment after the optimal plan's second action. Here the planner's next action will be to move to the left into that cell's third safe interval (after the second dynamic obstacle has passed). That means this action will wait one timestep in place and then move left.

Figure 9(d) shows the environment after the optimal plan's third action. The plan's remaining 3 actions are all downward with no waits.

V. EXPERIMENTAL RESULTS

We ran experiments in simulation and on a real robot to show the benefits of using safe intervals, while planning in dynamic environments.

A. Planner Implementation

Our test domain is in 4D (x,y,θ,time), with the first 3 dimensions (x,y,θ) being used for non-holonomic robots to generate smooth paths that satisfy constraints on the minimum turning radius. The actions used to get successors for states are a set of "motion primitives," which are short kinematically feasible motions sequences [5] used in a lattice-type planner shown in Figure 10. For our A* heuristic, we initially run a 16-connected 2D Dijkstra search from the goal to all the (x,y) cells in the environment, assuming the robot is a circular with a radius equal to the actual robot's inscribed circle. Since our lattice planner also has its orientation dimension discretized into 16 directions, the heuristic value in each cell underestimates the cost to the goal. This stems from the fact that the 2D search essentially assumes the robot can turn in place at no cost and its collision model is its narrowest diameter. This heuristic is computed quickly, relative to our search, because it has much lower dimensionality. However, it is also much more informative than the common Euclidean distance heuristic, since it takes static obstacle information into account.

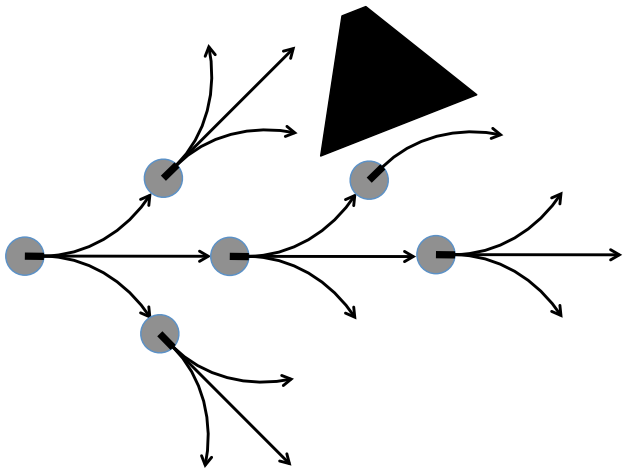

Fig. 10. An example of a lattice type graph

B. Simulation

1) Experiment Design: We compare SIPP against HCA*. This implementation of Silver's HCA* [10] is in a 4D search space (x,y,θ,time). For the heuristic, this algorithm uses a lower dimensional search from the goal backward that ignores the time dimension and the dynamic obstacles. In our case, this was a 2D search, just over the (x,y) dimensions[2]. Under the guidance of this informative heuristic, it then plans a path forward in the full state space with the time dimension and taking dynamic obstacles into account.

In order to test the efficiency of the algorithm we made two sets of 50 randomly generated experiments to simulate indoor and outdoor type environments. All environments are 500 by 500 cells with θ being discretized into 16 directions. The time dimension had a resolution of 0.1 seconds. Optimal solutions had an average duration of 12.7 seconds (127 timesteps). The robot's footprint occupies a single cell on the map and it has a random start and goal for each map. For each environment 200 dynamic obstacles were generated. Each dynamic obstacle could come in a large or small size (chosen randomly) and started at a random configuration in the environment. To generate a trajectory for a dynamic

[2]In Silver's work he used a 2D heuristic for a 3D problem. Although our problem is in 4D, we chose to keep the heuristic as a 2D search. We believe the additional dimension θ would slow down the heuristic part of the search significantly.

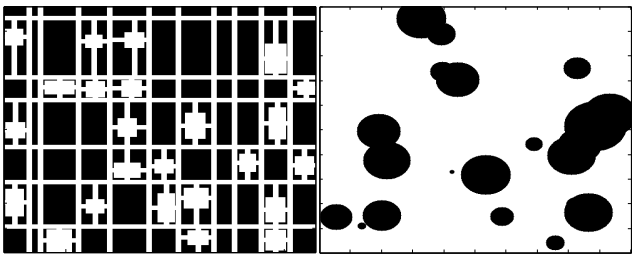

Fig. 11. (a) An example indoor map used for our experiments, (b) An example outdoor map used for our experiments

obstacle, random goals were chosen and 2D A* was used to find a paths to follow between the goal points. The indoor environments (Figure 11(a)) are composed of a series of randomly placed narrow hallways and rooms on a grid. The large dynamic obstacles fill the entire width of the hallways, so there is no way to pass them while the narrow dynamic obstacles only fill half a hallway, so they can be passed. The outdoor environments (Figure 11(b)) are very open, with randomly placed circular obstacles (representing trees, rocks, etc.) that occupy roughly 20% of the map.

2) Results: Table I shows our results from our indoor experiments. The times shown are average planning times. Our safe interval planner found solutions for all 50 trials but HCA* was only able to find solutions to 28 of the trials, since we have a 5 minute cap on the planning time. The results in this table are therefore computed only across the 28 tests that both planners found solutions for. For Table II, the safe interval planner found solutions for all 50 outdoor trials and HCA* found solutions for 47 of them within the 5 minute limit. Our results show SIPP outperforms HCA* in both types of tests, however the speed up is significantly greater for indoor environments. HCA* also seems to have a harder time getting solutions at all, in 5 minutes on the indoor cases. The major reason is that in the indoor cases, the places the robot can move are highly constrained, due to the tight hallways. This results in the robot often having to wait in place for an obstacle to move out of a hall or doorway, in order to get to the goal. In HCA*, waiting for an obstacle to go by would require many expands in the time dimension depending on how long the wait is, while SIPP would only expand one state to accomplish the same thing, regardless of the length of the wait. In the outdoor environments, this fact is less important, since there is so much open area, that the optimal path rarely involves waiting and the robot usually can just go around the dynamic obstacles. Across the experiments, we averaged 0.4s to precompute the safe intervals for 200 dynamic obstacles (this time drops to almost nothing as we approach a more reasonable number of dynamic obstacles, shown in our real robot results).

C. Tests on the PR2

In order to show SIPP works in real world environments, we implemented the planner in ROS and tested it on Willow Garage's PR2 robot. The PR2 is a mobile manipulation platform with two 7-DOF, mechanically counterbalanced

TABLE I

RESULTS FOR INDOOR ENVIRONMENTS (AVERAGES COMPUTED ONLY ACROSS TESTS BOTH ALGORITHMS COMPLETED)

Planner	Average Expands	Average Time(s)	% Completed in 5 minute limit
Full 4D (HCA*)	2,396,378.64	29.61	56%
SIPP	172,815.61	0.97	100%

TABLE II

RESULTS FOR OUTDOOR ENVIRONMENTS (AVERAGES COMPUTED ONLY ACROSS TESTS BOTH ALGORITHMS COMPLETED)

Planner	Average Expands	Average Time(s)	% Completed in 5 minute limit
Full 4D (HCA*)	2,497,147.15	47.04	94%
SIPP	334,878.79	2.88	100%

arms, an omni-directional base, and an array of sensors such as two lidars (one base scanner and one tilt scanner for making point clouds), several cameras, and IMU.

1) Implementation Detail: We wrote a global planner node which plugs in to ROS's navigation stack. The navigation stack provides the planner with the robot's pose and a map of the environment and expects a path in return. The only additional input we had to add were predicted dynamic obstacle trajectories. To do this, we wrote a node, which uses one of the PR2's lidar sensors to track people. We do a basic clustering of lidar points from each scan and then match the clusters from this scan to clusters from the last iteration. If a cluster appears to be moving from frame to frame, we mark it as a dynamic obstacle and on each iteration, we update its pose using a constant velocity motion model, and the cluster matching. By saving a few hundred previous locations (2-4 seconds) for a cluster, we use a linear regression fit to predict the velocity vector of the dynamic obstacle. We then use a basic linear extrapolation to predict where the dynamic obstacle will be over the next 20 seconds.

2) Results: In one of our experiments, the PR2 had to get past a person in a narrow hallway to get to its goal. The person was walking toward the robot, so the planner had the robot duck into a doorway (Figure 12), wait for the person to pass and then proceed to its goal when the hall was empty. Another one of our cases has a very slow moving person starting in front of the PR2 and walking in the direction of its goal (Figure 13). The hall is initially narrow and the PR2 must wait for the person to get to where the hall

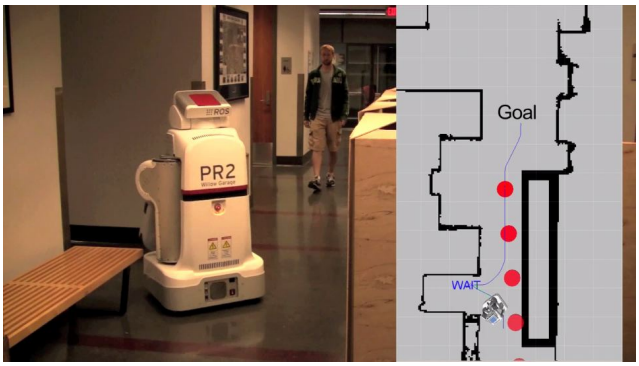

Fig. 12. The PR2 ducking to a doorway to avoid a person

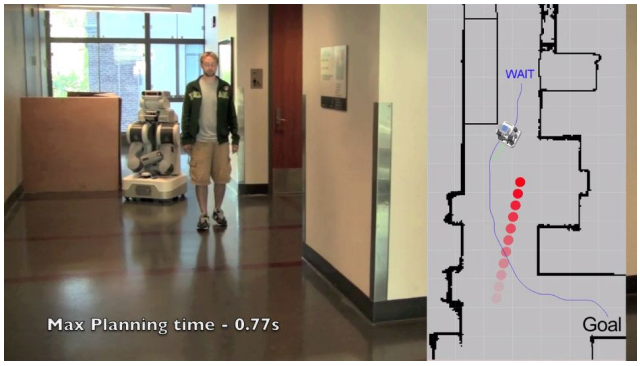

Fig. 13. The PR2 starting to pass a slow moving person

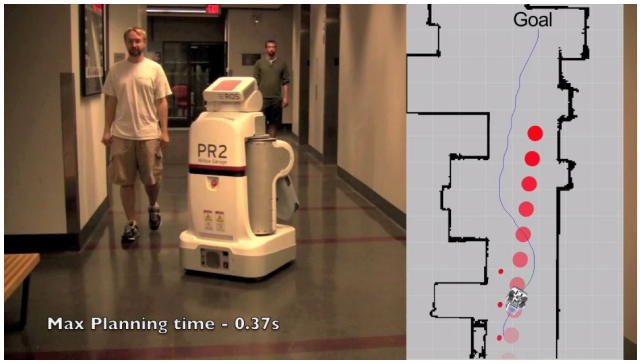

Fig. 14. The PR2 weaving between two people

widens. Then, it drives up along side the person and passes him to get to its goal quickest. The final case involved two staggered people coming toward the PR2. The robot had to weave between them in order to reach its goal (Figure 14). In the three experiments we ran, we had an average planning time of 0.49 seconds. Additionally, it took less than 0.01s to precompute the safe intervals. The images of our real robot trials are taken from our accompanying video.

VI. CONCLUSIONS

In this paper, we have developed the concept of safe intervals for planning in dynamic environments. Safe intervals represent time using the indices of contiguous periods, instead of using timesteps. This idea greatly decreases the number of states that need to be searched, without sacrificing the theoretical guarantees on optimality. As shown in our experimental results, this reduction in the state-space results in a planner that finds solutions significantly faster than the standard approach of planning with a timestep variable.

While SIPP does have an order of magnitude improvement over standard approaches and often produces plans in under a second, there are still instances of larger environments in which it can take a few seconds to plan. As any provably optimal planning, it doesn't scale as well as suboptimal approaches to planning. To make it scalable to larger dynamic environments, in our future work, we plan to consider extending it to use a weighted A* search [8], which has proven to be significantly faster than optimal search. Weighted A* uses an inflated heuristic to trade-off optimality for gains in efficiency, while maintaining bounds on sub-optimality. The challenge is that SIPP with inflated heuristics would become incomplete. But if we do succeed in extending SIPP to use inflated heuristics, then anytime searches, such as ARA* [6], would be possible, making SIPP suitable for the use on large environments.

Another concern is fast replanning since dynamic obstacle trajectories are difficult to predict and often change. While having anytime searches would go a long way in fixing this, an incremental extension would be ideal. The challenge with this is that doing an efficient backward search would require the planner to know in advance what the final g-value is so that it could initialize the time of the goal state in order to plan properly through the dynamic obstacles. We are also interested in relaxing some of our assumptions, so that we can handle arbitrary cost functions and more accurately handle the acceleration/deceleration limits of the robot.

VII. ACKNOWLEDGMENTS

This research was partially sponsored by the Army Research Laboratory Cooperative Agreement Number W911NF-10-2-0016, ONR grant N00014-09-1-1052, DARPA grant N10AP20011 and DARPA contract W31P4Q10C0202. We also thank Willow Garage for their partial support of this work.

REFERENCES

[1] K. Bekris and L. Kavraki. Greedy but safe replanning under kinodynamic constraints. *In IEEE International Conference on Robotics and Automation*, 2007.

[2] Dieter Fox, W. Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation*, 4(1), 1997.

[3] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21:233–255, 2002.

[4] A. Kushleyev and M. Likhachev. Time-bounded lattice for efficient planning in dynamic environments. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

[5] M. Likhachev and D. Ferguson. Planning long dynamically-feasible maneuvers for autonomous vehicles. In *Proceedings of Robotics: Science and Systems (RSS)*, 2008.

[6] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems (NIPS) 16*. Cambridge, MA: MIT Press, 2003.

[7] S. Petty and T. Fraichard. Safe motion planning in dynamic environments. In *Proceedings of IEEE Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 3726–3731, 2005.

[8] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:219–236, 1970.

[9] M Rufli, D Ferguson, and R Siegwart. Smooth path planning in constrained environments. In *Proc. of The IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

[10] D. Silver. Collaborative pathfinding. *In Proceedings of AIIDE*, 2005.

[11] J. van den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2366–2371, 2006.

[12] Jur P. van den Berg and Mark H. Overmars. Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics*, 21(5):885–897, 2005.

[13] David Wilkie, Jur P. van den Berg, and Dinesh Manocha. Generalized velocity obstacles. In *IROS*, pages 5573–5578, 2009.