Efficient Cost Computation in Cost Map Planning for Non-Circular Robots

Jennifer King
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
jeking@seas.upenn.edu

Maxim Likhachev
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
maximl@seas.upenn.edu

Abstract—For a robot with a circular footprint, obstacles in a map can be inflated by the radius of the footprint, and planning can treat the robot as a point robot. Many robotic vehicles however have non-circular footprints. When operating in cluttered spaces it therefore becomes important to evaluate the footprint of these robots against a cost map. This evaluation is one of the major computational burdens in planning for robots whose footprints can not be assumed to be circular.

In this paper, we propose an efficient method for evaluating a footprint of the robot against a cost map. Our method involves a transformation of the set of points covered by the footprint of the robot into two sets of points: points that should be evaluated against the cost map with inflated obstacles, and points that should be evaluated against the original cost map. The cumulative number of these points is much smaller than the number of points in the original footprint of the robot. Moreover, the method automatically reduces the robot to a single point when its footprint is circular. On the theoretical side, the paper proves the correctness of our method. On the experimental side, the paper shows that the method results in a significant speedup.

I. INTRODUCTION

Fast path planning is an important part of many robotic applications. The ability to quickly compute and recompute paths is imperative for mobile robots moving amongst static and dynamic obstacles. Perhaps, one of the most common approaches to planning is planning with grid-based cost maps [7][8][13]. Grid-based cost maps are simple to construct, are able to represent arbitrary shape obstacles and can incorporate complex cost functions.

When planning for a robot with a circular footprint, all obstacles in the cost map are typically inflated by the radius of the footprint and the representation of the robot is reduced to a point robot [3]. In such cases, the evaluation of a cost of an action reduces to a simple look-up of the costs of the cells traveled by the robot while executing this action. While very effective for circular robots, this optimization is not applicable to the robots with non-circular footprints.

For non-circular robots operating in cluttered environments with narrow passages (e.g., doorways, parking spots), it is important to plan by taking the orientation of the robot into account and using the actual footprint of the robot [10]. Such planning becomes expensive for several reasons. First, it needs to be done in at least 3-dimensions (x, y and orientation). Second, the evaluation of a cost of an action becomes highly expensive. In fact, usually, this cost evaluation procedure becomes the most time-consuming part of planning: it requires the convolution of the set of all the cells visited by the robot during execution of an action with a cost map. For example, a 1m by 1m square robot operating in a 2.5cm grid covers almost 500 cells for a simple turn-in-place action.

In this paper, we propose an efficient method for calculating action costs. The method is applicable to any ground robots, regardless of their footprints or holonomic constraints. Our method first reduces the number of cells to consider in the cost calculation by removing sets of points from the footprint which fall inside circles of a fixed radius. Then, it computes a second cost map which essentially inflates the original cost map by the radius of the removed circles. When calculating the action cost, the points which do not fall inside the circle are convolved against the original cost map and the circle centers are convolved against the new cost map. The results of these two separate convolutions are combined to create a single action cost. This transformation reduces significantly the number of points involved in the convolution and thus allows for a faster cost calculation.

In case of a circular robot, our method automatically reduces to representing the robot with a single point and inflating the map by the radius of the robot. For non-circular robots, our method provides a provably-correct optimization whose benefit depends on how close the footprint of the robot resembles a circular footprint. In our theoretical analysis, we show the correctness of the proposed method. In our experimental analysis, we show that the method can result in a significant speedup (about three-fold) over the current approaches.

II. RELATED WORK

In some approaches to planning, the cost map utilized for computing action cost is binary [2][5]. In such cases, a cell cost is infinite when the cell is occupied by an obstacle and zero otherwise. In these environments, action cost calculation essentially requires the detection of obstacle-collisions during execution of the action. Rather than perform the footprint convolution, collision detection algorithms such as those described in [1][11][12] can be utilized. These typically provide methods which are computationally less expensive. However, they do not generalize well to action...
cost computation in maps with non-binary costs. Additionally, they are often limited to a particular shape or class of obstacles.

Tornero, Hamlin and Keyy suggest that any three-dimensional polygon can be represented by an infinitely sized set of spheres, or alternatively, any two-dimensional polygon can be represented by an infinitely sized set of circles. They note that the accuracy of the representation is directly related to the complexity for obtaining these circles[15]. Limiting the set of spheres used in the representation to a single radius can allow the methods applicable to circular robots to generalize.

Previous works have also improved performance of action cost computation for any class of robots by pre-computing the cells covered when executing an action[10]. Additional performance improvements have been obtained by keeping a second cost space map with all obstacles inflated by the radius of the smallest circle completely enclosing the robot. All of the cells through which the center of the robot passes during the action are first checked against this second map. If all of the centers are obstacle free in the second map, the action is guaranteed to be obstacle free. Only those actions that do not produce conclusive results need to be convolved with the cost map. Each of these methods can be used in conjunction with our transformation to further increase efficiency.

III. ALGORITHM

Usually, in cost map planning, the cost of each cell ranges from 0 to \( \infty \), where 0 corresponds to a fully traversable cell, \( \infty \) corresponds to an obstacle (untraversable), and any finite value in between corresponds to some degree of traversability (or risk or some other cost function)[16]. The cost map planners then compute the cost of an action in the following manner. First, they iterate over all of the cells (points) covered by the footprint of the robot during the execution of the action (several examples are shown in Figure 4, top image in each sub-figure). During this iteration, they compute either the average or the maximum of the costs of the cells they iterate over. If any of these cells contain obstacles, then both the average and the maximum will be equal to \( \infty \). Second, the cost of the action is set to be the length of the action (or time it takes to execute it) times the computed average or computed maximum plus one. Thus, the cost of an action that involves the footprint of the robot going over an obstacle becomes infinite, and therefore the whole action becomes invalid. On the other hand, the cost of an action for which the footprint of the robot goes over only fully traversable cells becomes simply the length of the action (or the time it takes to execute it). The first of these steps is highly expensive, and our method is directed towards reducing its expense. In the following, we will assume that in this step we are interested to execute it. The first of these steps is highly expensive, and our method is directed towards reducing its expense. In the following, we will assume that in this step we are interested.

The basic idea behind our algorithm is to represent the footprint of the robot traversing each action (footprint of the action) with a set of circles of fixed radius \( r \), which fit completely inside the footprint (see Figure 4, bottom image in each sub-figure). This modification greatly reduces the number of cells used in the representation of the footprint of the action. Using only circles, however, leaves some cells out. To address this, we introduce the concept of a remainder set. The remainder set is defined as all of the points in the footprint of the action which cannot be enclosed by any of the circles used in our representation. Our new representation of the footprint of an action is therefore defined as the union of two sets of points: the centers of the circles, each of radius \( r \) and the remainder set.

In addition to modifying the representation of the footprint of each action, we also require the use of two cost maps. The first map is the original cost map. This map is used to look up the value of the remainder points when performing the action cost computation. The second map is a modified cost map with all the obstacles inflated by the radius \( r \) of the circles used in the representation. To achieve this inflation, every cell in the new cost map is defined to be the maximum of all the cells within radius \( r \) in the original map. This map is used to calculate the value of each circle center.

Picking an optimal circle radius \( r \) and finding the number of circles that minimize the number of the circle centers plus the size of the remainder set is an optimization problem. One approximation to this however can simply be picking \( r \) to be the radius of the largest circle that can be inscribed into the footprint of the robot at the center of the robot and computing the centers of the circles for an action to be all the cells visited by the center of the robot while executing the action.

Formal definition of the algorithm In the following, we define the algorithm formally. Let \( \{m_1...m_N\} \) be a set of cells. Let us also define a mapping, \( M_0(\cdot) \), which takes a cell and returns a value in between 0 and \( \infty \) that represents the cost of that cell in the cost map.

Let us now define the footprint \( F \) of an action as a set of cells \( \{f_1...f_n\} \) where each \( f_i \) is a cell in \( \{m_1...m_N\} \). We define the cost of the footprint as follows:

\[
\text{Definition 1: } C_F = \max_{f_i \in \{f_1...f_n\}} M_0(f_i)
\]

Thus, evaluating the cost of an action involves computing \( C_F \). Our transformation allows for the computation of \( C_F \) to be faster than iterating over all the points in \( \{f_1...f_n\} \). In particular, our method transforms the set \( \{f_1...f_n\} \) into two sets: set \( \{c_1...c_q\} \) of \( q \) centers of circles, each of radius \( r \), and set \( \{r_1...r_m\} \) of the remaining \( m \) cells. In other words, our method transforms the footprint \( F \) into a new footprint \( F' = \{c_1...c_q\} \cup \{r_1...r_m\} \). The following three definitions formally define this transformation.

\[
\text{Definition 2: Define a new footprint } F' = \{c_1...c_q\} \cup \{r_1...r_m\} \text{ such that } \forall f_i \in F \text{ one and only one of the following statements hold: }
\]

1) \( D(f_i, c_j) \leq r \) for some \( j \) (where \( D(f_i, c_j) \) defines the distance between \( f_i \) and \( c_j \))
2) \( f_i = r_j \) for some \( j \)
procedure ComputeM2(w, h, M1, M2)
M2 = M1
for each x ≥ 0 and x < w
  for each y ≥ 0 and y < h
    if M1(x, y) ≠ 0
      ... 4
shows that M1(rk) = M0(rk). Definition 2 states that rk must be a member of the original footprint. If rk is a
total contained within the footprint. Formally:
∀i, j s.t. D(mi, cj) ≤ r ⇒ ∃k s.t. mi = fk
Definition 4: Define two new mappings, M1(·), M2(·) as follows:
1) Mapping M1 is an exact copy of M0. Formally, ∀i M1(mi) = M0(mi).
2) Mapping M2 returns the value which is the maximum of all cells in the original map within radius r. Formally, ∀i M2(mi) = maxm,j|D(mi, mj)≤r M0(mj).
The above definition 4 corresponds to defining two maps. Our method maintains both of these maps, and uses both of
them in evaluating the cost of the transformed footprint F'.

The following two definitions define exactly how the cost of the
footprint F' of an action is computed. First, let us break
F' into two subsets F'_1 and F'_2, where F'_1 = {r1...rm} and
F'_2 = {c1...ck}.
Definition 5: The costs of the footprints F'_1 and F'_2 are defined as:
C_{F'_1} = max_{rj ∈ {r1...rm}} M1(rj)
C_{F'_2} = max_{cj ∈ {c1...ck}} M2(cj)
Definition 6: We define the cost of F' as the maximum of C_{F'_1} and C_{F'_2}.

A high-level pseudocode of our transformation is shown
in Figure 1. The ComputeM2 function shows the compu-
tation of the map M2 according to the definition 4. The

procedure ComputeM2(w, h, M1, M2)
M2 = M1
for each x ≥ 0 and x < w
  for each y ≥ 0 and y < h
    if M1(x, y) ≠ 0
      ... 4
procedure PrecomputeActions()
for each action a in ActionList[θ] for each orientation θ of the robot
compute footprint F as the set of cells covered by the footprint of the robot traversing action a starting at 0, 0, θ
each f in F
  a.Footprint.add(f)
compute at most q circles of radius r that are fully enclosed in F
for each such circle with center c
  a.FootprintCircles.add(c)
for each f in F
  d = euclideanDistance(f, c)
  if d <= r remove f from a.Footprint
procedure Main()
Given the original cost map M1 of width w and height h
ComputeM2(w, h, M1, M2)
PrecomputeActions()
Plant()
member of the original footprint then $M_0(f_p) \geq M_1(r_k)$ because $f_p$ is the largest value in the footprint by definition. But we have already shown that $M_2(c_j) = M_0(f_p)$ and $M_2(c_j) < M_1(r_k)$. So we have a contradiction. Therefore, $C_F = C_{F_2} = M_2(c_j) = M_0(f_p) = C_F$.

Case 2 - $f_p = r_j$. First we argue that $C_{F_1} = M_1(r_j)$ and $M_1(r_j) = M_0(f_p)$. We show this by contradiction. Assume $C_{F_1} \neq M_1(r_j)$. By definition 5 this means $\exists k$ such that $M_1(r_k) > M_1(r_j)$. Definition 2 states that both $r_j$ and $r_k$ were in the original footprint $F$. But then $r_j$ would not be the maximum value in the footprint. This is a contradiction because $r_j = f_p$ and $f_p$ is the maximim value in the footprint by definition. So $C_{F_1} = M_1(r_j) = M_0(f_p)$.

Next we need to show $C_{F'} = C_{F_1}$. This equality will follow from definition 6 if we show that $C_{F_1} \geq C_{F_2}$. Again we show this by contradiction. Assume $C_{F_1} < C_{F_2}$. This means $\exists k$ such that $M_2(c_k) > M_1(r_j)$. By definition 4, this means that $\exists k$ such that $D(m_1, c_k) \leq r$ and $M_0(m_1) > M_0(r_j) = M_1(r_j)$. Definitions 2 and 3 state that $m_2$ and $r_j$ were inside the original footprint. This means that $r_j$ cannot be the maximum value in the footprint. This poses a contradiction because $r_j = f_p$ and $f_p$ is the maximum in the footprint by definition. Therefore $C_{F_2} = C_{F'}$ and $C_{F'} = C_{F_1}$ by definition 6. Thus $C_{F'} = M_1(r_k) = M_0(f_p) = C_F$.

V. INTEGRATION WITH PLANNER

We have integrated our method into a lattice-based planner [10][14]. Lattice-based planning allows for planning complex dynamically-feasible maneuvers. A state lattice [14] is a discretization of the configuration space into a set of states, representing configurations, and connections between these states, where every connection represents a feasible path. As such, lattices provide a method for motion planning problems to be formulated as graph searches. However, in contrast to many graph-based representations (such as 4-connected or 8-connected grids), the feasibility requirement of lattice connections guarantees that any solutions found using a lattice will also be feasible. This makes them very well suited to planning for non-holonomic and highly-constrained robotic systems, such as passenger vehicles.

In our scenario, we define each state to be a 3-dimensional state: $x, y, \theta$. To demonstrate the benefits of our proposed optimization, we consider a square robot capable of turning in place as well as moving forward while rotating. (An example of actions suitable specifically to large non-holonomic vehicles can be found in [10]). Fourteen actions are used in our lattice, each defined by a unique combination of forward velocity, rotational velocity and time. We define two turn-in-place actions, one for turning left forty-five degrees and another for turning right by the same amount. We define five forward actions with a duration of one second and a forward velocity of 1 m/s. The rotational velocity varies for each action but remains constant throughout the execution of the action. The range of rotational velocities covers $-\pi/2$ to $\pi/2$. Additionally, we define five identical backward actions. For these, the forward velocity is defined to be -1 m/s.

Finally, we define an extra forward and backward action with a shorter duration than the others to allow for the robot to move forward or backward by a single cell. These actions have no rotational velocity. Figure 3 shows the trajectories of all of the actions. The two turn in place actions coupled with the short forward and backward action guarantee that the planner can generate a path when one is available. The ten longer actions allow for the planner to select actions which make larger progress toward the goal, thereby reducing the overall length (in terms of number of actions) of the planned path. Note that our method could be used with a non-holonomic vehicle by adjusting the action space to include only actions feasible for the robot given its constraints.

Fig. 3. Action Space: Pictured are ten of the fourteen predefined actions in the action space. Each action is uniquely defined by a translational velocity, rotational velocity and duration. In addition to each of these actions, the action space contains four more predefined actions which cannot be seen in the figure: turn in place left, turn in place right, one short forward action and one short reverse action. The short actions allow for movement by a single grid space.

During the pre-computation step, we determine the set of states covered by the robot as it executes each action. We then perform the transformation defined above on the state set. To do this, we fit a set of circles inside the footprint of the robot as it stands still. We then translate these circles along the path of the robot, rotating them as the body of the robot rotates through the action. This sweeps a set of points from the original footprint and leaves only the set of points that did not fall inside a circle at any point during the action. These become the remainder set. We record the locations of the circle centers as they translate and rotate through the action. These points become the center set. Figure 4 shows the footprints of the actions before and after the transformation. Note that only five actions are pictured in the figure. Each of the fourteen predefined actions is a rotation or reflection (or both) of one of these five actions.

To search the constructed lattice-based graph, we use ARA* search [9]. ARA* exploits a property of A* that can result in much faster generation of solutions, namely that if consistent heuristics are used and multiplied by an inflation
Fig. 4. Action Footprints before and after transformation: This figure shows the footprint of the robot as it executes each action. The number of cells in the footprint is indicated for each action. This number directly corresponds to the number of points involved in the convolution with the cost map. Each of the fourteen actions is a rotation or reflection of one of the five actions pictured above.

factor $\epsilon > 1$, then $A^*$ can often generate a solution much faster than if no inflation factor is used [6], and the cost of the solution generated by $A^*$ will be at most $\epsilon$ times the cost of an optimal solution [4]. $ARA^*$ operates by performing a series of these inflated $A^*$ searches with decreasing inflation factors, where each search reuses information from previous searches. By doing so, it is able to provide suboptimality bounds on all solutions generated and allows for control of these bounds, since the user can decide how much the inflation factor is decreased between searches.

VI. EXPERIMENTAL ANALYSIS

We utilize two separate test scenarios. In the first scenario, we allow the planner enough time to compute the optimal path and compare the amount of time to run with and without the transformation detailed above. In the second test, we limit the amount of time the planner is given to compute a path and compare the cost of the paths achieved by the planner when run with and without our optimization. This test more accurately represents the use of a system in the real world, where planning time is often limited.

We define two environments for our testing. The first is a 25m by 25m space discretized into a 500 by 500 grid with each cell representing a 5cm by 5cm area. We define the robot to be 1m by 1m. The second environment is a 48m by 54m space discretized into cells of size 2.5cm by 2.5cm. In this environment, the robot is defined as a 0.5m by 0.5m square. Figure 5 shows the environments.

All times presented in the following result sets were recorded from runs of the software executed on a MacBook Pro with a Pentium Dual Core processor.

A. Optimal Planning

For the first set of tests we utilize the planner with and without our proposed cost computation optimization. Our main goal for these tests is to study the effectiveness of the algorithm. We select three different start and goal locations for each environment. The planner is run for each different start/goal pair and we average the results across runs. Table I shows the results. We study two measurements of time, the time to precompute and the time to plan. The time to precompute includes the time it takes to generate the action footprints pictured in figure 4, bottom images. It also includes the time it takes to generate the map $M_2$ using the rules defined in the Algorithm section of this paper.
### Fig. 5. Test Environments

The figures above show two test environments.

<table>
<thead>
<tr>
<th>World</th>
<th>Without Optimization</th>
<th>With Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time To Precompute(s)</td>
<td>Time To Plan(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-1</td>
<td>.460</td>
<td>8.410</td>
</tr>
<tr>
<td>1-2</td>
<td>.450</td>
<td>23.380</td>
</tr>
<tr>
<td>1-3</td>
<td>.430</td>
<td>23.450</td>
</tr>
<tr>
<td>2-1</td>
<td>.690</td>
<td>696.480</td>
</tr>
<tr>
<td>2-2</td>
<td>.740</td>
<td>156.270</td>
</tr>
<tr>
<td>2-3</td>
<td>.740</td>
<td>1001.000</td>
</tr>
<tr>
<td>1.090</td>
<td>2.880</td>
<td></td>
</tr>
<tr>
<td>.660</td>
<td>9.790</td>
<td></td>
</tr>
<tr>
<td>.660</td>
<td>9.810</td>
<td></td>
</tr>
<tr>
<td>1.150</td>
<td>24.970</td>
<td></td>
</tr>
<tr>
<td>1.170</td>
<td>81.690</td>
<td></td>
</tr>
<tr>
<td>World</td>
<td>Time To Precompute(s)</td>
<td>Time To Plan(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.090</td>
<td>2.880</td>
<td></td>
</tr>
<tr>
<td>.660</td>
<td>9.790</td>
<td></td>
</tr>
<tr>
<td>.660</td>
<td>9.810</td>
<td></td>
</tr>
<tr>
<td>1.150</td>
<td>24.970</td>
<td></td>
</tr>
<tr>
<td>1.170</td>
<td>81.690</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I**

**Optimal Planning Results:** The first three rows represent results from tests against environment 1. Each of these tests have a unique start and goal state. The second three rows represent results from test against environment 2.

### B. Real-time Planning

For the second set of tests, we examine the performance of the planner in real-time environments where suboptimal planning is often necessary to meet real-time requirements. We consider a single environment with a single start/goal location pair. Specifically, we select the configuration which took the largest amount of time to achieve an optimal path in the first set of tests. We vary the time limit on the planner and compare the optimality of the planned paths. Table II shows the results of this set of tests. Figure 6 compares the paths achieved when planning time is limited to four seconds for each planner.

To show the statistical soundness of our results we run a final set of tests. Here we run the planner multiple times, randomly selecting a start and goal for each run. We examine the percentage increase in precomputation time and the percentage decrease in planning time when using our
<table>
<thead>
<tr>
<th>Time(s)</th>
<th>Without Optimization Cost</th>
<th>Epsilon</th>
<th>With Optimization Cost</th>
<th>Epsilon</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>-</td>
<td>-</td>
<td>51000</td>
<td>5.0</td>
</tr>
<tr>
<td>1.0</td>
<td>-</td>
<td>-</td>
<td>51000</td>
<td>4.4</td>
</tr>
<tr>
<td>2.0</td>
<td>51000</td>
<td>5</td>
<td>51000</td>
<td>3.8</td>
</tr>
<tr>
<td>4.0</td>
<td>51000</td>
<td>5</td>
<td>48000</td>
<td>2.2</td>
</tr>
<tr>
<td>8.0</td>
<td>48000</td>
<td>2.2</td>
<td>47000</td>
<td>1.4</td>
</tr>
<tr>
<td>16.0</td>
<td>44000</td>
<td>1.6</td>
<td>40000</td>
<td>1.2</td>
</tr>
</tbody>
</table>

TABLE II
REAL-TIME PLANNING RESULTS: THIS TABLE SHOWS RESULTS PRODUCED WHEN PLANNING TIME IS LIMITED. THE NON-OPTIMIZED VERSION OF THE PLANNER FAILS TO FIND A SOLUTION UNTIL GIVEN AT LEAST 5.0 SECONDS TO PLAN. ADDITIONALLY, THE OPTIMIZED VERSION OF THE PLANNER CONSISTENTLY OUTPERFORMS THE NON-OPTIMIZED VERSION IN BOTH COST OF THE RESULTANT PATH AND ACHIEVED EPSILON VALUE.

<table>
<thead>
<tr>
<th>World</th>
<th>Precomputation Time Increase (%)</th>
<th>Plan Time Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58.60</td>
<td>63.00</td>
</tr>
</tbody>
</table>

TABLE III
COMPUTATION TIME SAVINGS: THE TABLE SHOWS THE AVERAGE % INCREASE IN PRECOMPUTATION TIME AND THE % DECREASE IN PLANNING TIME WHEN PLANNING BETWEEN RANDOM START AND GOAL LOCATIONS IN EACH ENVIRONMENT.

transformation. Results are averaged across runs. Table III shows the results when we give the planner enough time to calculate the optimal path.

VII. CONCLUSIONS
Our experimental results show around a sixty percent decrease in planning time when using the transformation we propose. This savings is a savings that will be experienced each time the planner is run. We note that the results also indicate a sixty percent increase in precomputation time. While initially this number may appear to offset any gains made by the planner, it is important to remember that most of the precomputations (precomputing actions) must only be run once at startup. They do not need to be run every time the planner is called. Additionally, the run time of the precomputation step is dependent upon the size and complexity of the robot while the runtime of the planner is dependent upon the size and complexity of the configuration space. Thus, as environments become larger and more densely packed with obstacles the increase in precomputation time becomes small compared to the savings in planning time.

REFERENCES