November/December 2020
ISSN: 0193-4120 Page No. 36- 45

Study And Development Of Various Techniques For Path Planning And Optimization Of Differential Drive Wheel Robot

Shailesh Verma[1], D. D. Mishra[2], Rahul Mishra[3] , V. C. Jha[4], Mukesh Kumar Verma[5]

[1] Research Scholar, Mechanical Engineering Department, Kalinga University, Raipur
[2] Research Scholar, Mechanical Engineering Department, Kalinga University, Raipur
[3] Mechanical Engineering Department, Kalinga University, Raipur
[4] Mechanical Engineering Department, Kalinga University, Raipur
[5] Mechanical Engineering Department, Professional Institute of Engineering & Technology, Raipur

Article Info
Volume 83
Page Number: 36-45
Publication Issue:
November/December 2020

Article History
Article Received: 25 October 2020
Revised: 22 November 2020
Accepted: 10 December 2020
Publication: 31 December 2020

Abstract
Navigation consisting of two essential components known as localization and planning is the art of steering a course through a medium. Localization matches an actual position in the real-world to a location inside a map; in other words, each location in the map refers to an actual position in the environment. Planning is finding a short, collision-free path from the starting position towards the predefined ending location. This study is a survey which focuses on introducing classic and heuristic-based path planning approaches and investigates their achievements in search optimization problems. The methods are categorized, their strengths and drawbacks are discussed, and the applications in which they have been utilized are explained.

Keywords: *Navigation, Path Planning, Cell Decomposition, Potential Field, Road Map, Subgoal Network, Neural Network, Genetic Algorithm, Particle Swarm Optimization, Ant Colony Optimization.*

INTRODUCTION:

In this paper, we describe a family of heuristic-based planning algorithms that has been developed to address various challenges associated with planning in the real world. Each of the algorithms presented have been verified on real systems operating in real domains. However, a prerequisite for the successful general use of such algorithms is an analysis of the common fundamental elements of such algorithms, a discussion of their strengths and weaknesses, and guidelines for when to choose a particular algorithm over others. Although these algorithms have been documented and described individually, a comparative analysis of these algorithms is lacking in the literature. With this paper we hope to fill this gap. We begin by providing background on path planning in static, known environments and classical algorithms used to generate plans in this domain. We go on to look at how these algorithms can be extended to efficiently cope with partially-known or dynamic environments. We then intro-duce variants of these algorithms that can produce suboptimal solutions very quickly when time is limited and improve these solutions while time permits. Finally, we discuss an algorithm that combines principles from all of the algorithms previously discussed; this algorithm can plan in dynamic environments and with limited deliberation time. For all the algorithms discussed in this paper, we provide example problem scenarios in which they are very effective and situations in which they are less effective. Although our primary focus is on path planning, ~~several of the~~se algorithms are applicable in more general planning scenarios. Our aim is to share intuition and lessons learned over the course of several system implementations and guide readers in choosing algorithms for their own planning domains.

PATH PLANNING

Planning consists of finding a sequence of actions that trans-forms some initial state into some desired goal state. In path planning, the states are agent locations and transitions between states represent actions the agent can take, each of which has an associated cost. A path is optimal if the sum of its transition costs (edge costs) is minimal across all possible paths leading from the initial position (start state) to the goal position (goal state). A planning algorithm is complete if it will always find a path in finite time when one exists, and will let us know in finite time if no path exists. Similarly, a planning algorithm is optimal if it will always find an optimal path.

Several approaches exist for computing paths given some representation of the environment. In general, the two most popular techniques are deterministic, heuristic-based algorithms.

When the dimensionality of the planning problem is low, for example when the agent has only a few degrees of freedom, deterministic algorithms are usually favored because they provide bounds on the quality of the solution path re-turned. In this paper, we concentrate on deterministic algorithms. For more details on probabilistic techniques.

A common technique for robotic path planning consists of representing the environment (or configuration space) of the robot as a graph G = (S, E), where S is the set of possible robot locations and E is a set of edges that represent transitions between these locations. The cost of each edge represents the cost of transitioning between the two endpoint locations.

Planning a path for navigation can then be cast as a search problem on this graph. A number of classical graph search algorithms have been developed for calculating least-cost paths on a weighted graph algorithms return an optimal path and can be considered as special forms of :-

```
ComputeShortestPath()
  01. while ($\arg\min_{s \in OPEN}(g(s) + h(s, s_{goal}))$ 6$\neq s_{goal}$)
  2.     remove state s from the front of OPEN;
  3.     for all $s' \in Succ(s)$
  4.       if ($g(s') > g(s) + c(s, s')$)
  5.         $g(s') = g(s) + c(s, s')$;
  6.         insert $s'$ into OPEN with value ($g(s') + h(s', s_{goal})$);
Main()
  07. for all $s \in S$
  8.     $g(s) = \infty$;
  9. $g(s_{start}) = 0$;
  10.    OPEN = $\varnothing$;
  11.    insert $s_{start}$ into OPEN with value ($g(s_{start}) + h(s_{start}, s_{goal})$);
  12.    ComputeShortestPath();
```

Figure 1: The A* Algorithm (forwards version)

dynamic programming. A* operates essentially the same as This algorithm except that it guides its search towards the most promising states, potentially saving a significant amount of computation. A* plans a path from an initial state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is the set of states in some finite state space. To do this, it stores an estimate g(s) of the path cost from the initial state to each state s. Initially, g(s) = for all states $s \in S$. The algorithm begins by updating the path cost of the start state to be zero, then places this state onto a priority queue known as the OPEN list. Each element s in this queue is ordered according to the sum of its current path cost from the start, g(s), and a heuristic estimate of its path cost to the goal, $h(s, s_{goal})$. The state ∞ with the minimum such sum is at the front of the priority queue. The heuristic $h(s, s_{goal})$ typically underestimates the cost of the optimal path from s to s_{goal} and is used to focus the search.

The algorithm then pops the state s at the front of the queue and updates the cost of all states reachable from this state through a direct edge: if the cost of state s, g(s), plus the cost of the edge between s and a neighboring state s', c(s, s'), is less than the current cost of state s', then the cost of s' is set to this new, lower value. If the cost of a neighboring state s' changes, it is placed on the OPEN list. The algorithm continues popping states off the queue until it pops off the goal state. At this stage, if the heuristic is admissible, i.e. guaranteed to not overestimate the path cost from any state to the goal, then the path cost of s_{goal} is guaranteed to be optimal. The complete algorithm is given in Figure 1.

It is also possible to switch the direction of the search in A*, so that planning is performed from the goal state to-wards the start state. This is referred to as 'backwards' A*, and will be relevant for some of the algorithms discussed in the following sections.

RELATED WORKS

Basic Mathematical Model. The path planning problem of UCAV can be modeled as a constrained optimization problem. Before searching track, flight condition and elements (like terrain, threats, climate, etc.) of relevant path planning are represented as symbol information.

Let (X_L, Y_L, Z_L) be longitude, latitude, and height of a certain point in state space. T he path planning space can be represented as a set: $\{(X_L, Y_L, Z_L) \mid 0 \leq X_L \leq \max X_L,\ 0 \leq Y_L \leq \max Y_L,\ 0 \leq Z_L \leq \max Z_L\}$, which represents a space district. In practical planning, the planning space is divided into two-dimensional grids or three-dimensional grids; a series of nodes are acquired and built into a network graph. The path planning problem can be simply attributed to a combinational optimization problem for getting the shortest path of the network graph. That is to say, when UCAV is flying along the path formed by some nodes of the network graph, a certain kind of path takes minimum cost.

Supposing the nodes of network graph form a set

$$S = \{ s_1,\ s_2,\ s_3 \ldots s_m\} \qquad (1)$$

Define a set that includes all paths from the start point to the end point as E:

$$E = (e_1, e_2, e_3, \ldots, e_n) \qquad (2)$$

Let s_i and s_j be two adjacent nodes on the path e_k, the connecting line between two nodes can be expressed by V (s_i, s_j), the cost value of the connecting line between two nodes can be expressed by u_{ij} and the path planning problems of UCAV are defined as follows:

$$\text{Min} \quad f(e_k) \quad = \quad \sum_{(s_i,s_j)ce_k} u_{ij}$$

As can be seen from the above content, the performance constraint of UCAV is not reflected in the planning. If the nodes in the network graph are feasible points which take performance constraint of UCAV into account, the path with the performance constraint of UCAV can be reflected from solving the above optimization problems. This is a new mathematical model. Compared with, our new mathematical model takes more constraints into account. And simulation result shows that it is highly useful for the approximate optimal solution. Besides, it is also good at processing path planning in complicated conditions.

Basic Constraint Conditions of Path Planning. There are many factors that influence the result of path planning. These factors, which include terrain features, threat locations, and mission requirements, are basic constraints in mathematical modeling. Path planning should meet basic constraints, and they mainly include the following constraints.
Minimum Route Length. Aircraft generally does not want to weave and turn constantly, because this adds to fuel cost and increases navigational errors.
Maximum Turning Angle. The turning angle of the aircraft does not exceed maximum turning angle. For instance, the aircraft cannot make severe turns without a greater risk of collision in formation fight.
Route Distance Constraint. The length of the route does not exceed maximum distance because of fuel restriction. Specific Approaching Angle to Target Point. This constrains UCAV to approach the hostile aircraft from a predetermined angle to ensure UCAV defend the weak part.
Path Planning Cost. On the premise that some constraints' are met, the UCAV path planning aims to generate Trajectory with the highest survival rate. Therefore, threat locations in battle field should be fully taken into account. Threat factors and fuel restriction are mainly taken into account when calculating trajectory cost.

Models of Threats

Threat Model of Radar. The factors that influence the probability of radar detection mainly includes earth curvature, atmospheric refraction and absorption, ground clutter interference, distance between aircraft and radar, radar cross section, radar performance, and ground multipath effect. For the sake of simplification, here we mainly take the distance from aircraft to radar and radar performance into account. Supposing the flying height is $\hbar$, the horizontal distance from aircraft to radar is, radar maximal horizontal range is max, radar performance coefficient is, and the probabilistic model of radar detection can be presented.

INCREMENTAL REPLANNING ALGORITHMS

The above approaches work well for planning an initial path through a known graph or planning space. However, when operating in real world scenarios, agents typically do not have perfect information. Rather, they may be equipped with incomplete or inaccurate planning graphs. In such cases, any path generated using the agent's initial graph may turn out to be invalid or suboptimal as it receives updated information. For example, in robotics the agent may be equipped with an onboard sensor that provides updated environment information as the agent moves. It is thus important that the agent is able to update its graph and replan new paths when new information arrives.

One approach for performing this replanning is simply to replan from scratch: given the updated graph, a new optimal path can be planned from the robot position to the goal using A*, exactly as described above. However, replanning from scratch every time the graph changes can be very computationally expensive. For instance, imagine that a change occurs in the graph that does not affect the optimality of the current solution path. Or, suppose some change takes place that does affect the current solution, but in a minor way that can be quickly fixed. Replanning from scratch in either of these situations seems like a waste of computation. Instead, it may be far more efficient to take the previous solution and repair it to account for the changes to the graph.
D* and D* Lite are extensions of A* able to cope with changes to the graph used for planning. The two algorithms are fundamentally very similar; we restrict our attention here to D* Lite because it is simpler and has been found to be slightly more efficient for some navigation tasks. D* Lite initially constructs an optimal solution path from the initial state to the goal state in exactly the same manner as backwards A*. When changes to the planning graph are made (i.e., the cost of some edge is altered), the states whose paths to the goal are immediately affected by these changes have their path costs updated and are placed on the planning queue (OPEN list) to propagate the effects of these changes to the rest of the state space. In this way, only the affected portion of the state space is pro-cessed when changes occur. Furthermore, D* Lite uses a heuristic to further limit the states processed to only those states

whose change in path cost could have a bearing on the path cost of the initial state. As a result, it can be up to two orders of magnitude more efficient than planning from scratch using A* (Koenig & Likhachev 2002).
In more detail, D* Lite maintains a least-cost path from a

start state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is again the set of states in some finite state space. To do this, it stores an estimate g(s) of the cost from each state s to the goal. It also stores a one-step lookahead cost rhs(s) which satisfies:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in Succ(s)}(c(s, s') + g(s')) & \text{otherwise,} \end{cases}$$

here Succ(s) $\in$ S denotes the set of successors of s and c(s, s') denotes the cost of moving from s to s' (the edge cost). A state is called consistent iff its g-value equals its rhs-value, otherwise it is either over consistent (if g(s) > rhs(s)) or under consistent (if g(s) < rhs(s)).

Like A*, D* Lite uses a heuristic and a priority queue to focus its search and to order its cost updates efficiently. The heuristic h(s, s') estimates the cost of moving from state s to s', and needs to be admissible and (backward) consistent: $h(s, s') \leq c^*(s, s')$ and $h(s, s'^0) \leq h(s, s') + c^*(s', s'^0)$ for all states s, s', $s'^0 \in S$, where $c^*(s, s')$ is the cost associated with a least-cost path from s to s'. The priority queue OPEN always holds exactly the inconsistent states; these are the states that need to be updated and made consistent.
The priority, or key value, of a state s in the queue is:

$$key(s) = [k_1(s), k_2(s)] = [\min(g(s), rhs(s)) + h(s_{start}, s), \min(g(s), rhs(s))].$$

A lexicographic ordering is used on the priorities, so that pri0ority key(s) is less than or equal to priority key(s'), denoted $key(s) \dot{\leq} key(s')$, iff $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) \leq k_2(s')$. D* Lite expands states from the queue in increasing priority, updating their g-values and their pre-decessors' rhs-values, until there is no state in the queue with a priority less than that of the start state. Thus, during its generation of an initial solution path, it performs in exactly the same manner as a backwards A* search.

To allow for the possibility that the start state may change over time D* Lite searches backwards and consequently fo-cusses its search towards the start state rather than the goal state. If the g-value of each state s was based on a least-cost path from s_{start} to s (as in forward search) rather than from s to s_{goal}, then when the robot moved every state would have to have its cost updated. Instead, with D* Lite only the heuristic value associated with each inconsistent state needs to be updated when the robot moves. Further, even this step can be avoided by adding a bias to newly inconsistent states being added to the queue (see (Stentz 1995) for details).

```
key(s)
 1. return [$\min(g(s), rhs(s)) + h(s_{start}, s)$; $\min(g(s), rhs(s))$];

UpdateState(s)
 02. if s was not visited before
 3.    $g(s) = \infty$;
 4. if (s 6= $s_{goal}$) $rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
 5. if (s $\in$ OPEN) remove s from OPEN;
 6. if (g(s) 6= rhs(s)) insert s into OPEN with key(s);

ComputeShortestPath()
 07. while ($\min_{s \in OPEN}(key(s)) \dot{<} key(s_{start})$ OR $rhs(s_{start})$ 6= $g(s_{start})$)
 8.    remove state s with the minimum key from OPEN;
 9.    if (g(s) > rhs(s))
 10.      g(s) = rhs(s);
 11.      for all s' $\in$ P red(s) UpdateState(s');
 12.   else
 13.      $g(s) = \infty$;
 14.      for all s' $\in$ P red(s) $\cup$ {s} UpdateState(s');

Main()
 15.      $g(s_{start}) = rhs(s_{start}) = \infty$; $g(s_{goal}) = \infty$;
 16.      $rhs(s_{goal}) = 0$; OPEN = $\emptyset$;
 17.      insert $s_{goal}$ into OPEN with key($s_{goal}$);
 18.      forever
 19.   ComputeShortestPath();
 20.   Wait for changes in edge costs;
 21.   for all directed edges (u, v) with changed edge costs
 22.      Update the edge cost c(u, v);
 23.      UpdateState(u);
```

Figure 2: The D* Lite Algorithm (basic version).

When edge costs change, D*Lite updates the rhs values of each state immediately affected by the changed edge costs and places those states that have been made inconsistent onto the queue. As before, it then expands the states on the queue in order of increasing priority until there is no state in the queue with a priority less than that of the start state. By incorporating the value $k_2(s)$ into the priority for state s, D*Lite ensures that states that are along the current path and on the queue are processed in the right order. Combined with the termination condition, this ordering also ensures that a least-cost path will have been found from the start state to the goal state when processing is finished. The basic version of the algorithm (for a fixed start state) is given in Figure 2.

D*Lite is efficient because it uses a heuristic to restrict attention to only those states that could possibly be relevant to repairing the current solution path from a given start state to the goal state. When edge costs decrease, the incorporation of the heuristic in the key value (k_1) ensures that only those newly-over consistent states that could potentially decrease the cost of the start state are processed. When edge costs increase, it ensures that only those newly-under consistent states that could potentially invalidate the current cost of the start state are processed.

In some situations the process of invalidating old costs Because the optimizations of D*Lite can significantly speed up the algorithm, for an efficient implementation of D*Lite please refer to that paper. may be unnecessary for repairing a least-cost path. For ex-ample, such is the case when there are no edge cost de-creases and all edge cost increases happen outside of the current least-cost path. To guarantee optimality in the future, D* Lite would still invalidate portions of the old search tree that are affected by the observed edge cost changes even though it is clear that the old solution remains optimal. To overcome this a modified version of D* Lite has recently been proposed that delays the propagation of cost increases as long as possible while still guaranteeing optimality. Delayed D* is an algorithm that initially ignores under consistent states when changes to edge costs occur. Then, after the new values of the over consistent states have been adequately propagated through the state space, the resulting solution path is checked for any under-consistent states. All under consistent states on the path are added to the OPEN list and their updated values are propagated through the state space. Because the current propagation phase may alter the solution path, the new solution path needs to be checked for under consistent states. The en-tire process repeats until a solution path that contains only consistent states is returned.

APPLICABILITY: REPLANNING ALGORITHMS

Delayed D* has been shown to be significantly more efficient than D*Lite in certain domains, Typically, it is most appropriate when there is a relatively large distance between the start state and the goal state, and changes are being observed in arbitrary locations in the graph (for example, map updates are received from a satellite). This is because it is able to ignore the edge cost increases that do not involve its current solution path, which in these situations can lead to a dramatic decrease in over-all computation. When a robot is moving towards a goal in a completely unknown environment, Delayed D* will not pro-vide much benefit over D*Lite, as in this scenario typically the costs of only few states outside of the current least-cost path have been computed and therefore most edge cost in-creases will be ignored by both algorithms. There are also scenarios in which Delayed D* will do more processing than D*Lite: imagine a case where the processing of under consistent states changes the solution path several times, each time producing a new path containing under consistent states. This results in a number of replanning phases, each potentially updating roughly the same area of the state space, and will be far less efficient than dealing with all the under-consistent states in a single replanning episode. However, in realistic navigation scenarios, such situations are very rare.

In practice, both D*Lite and Delayed D* are very effective for replanning in the context of mobile robot navigation. Typically, in such scenarios the changes to the graph are happening close to the robot (through its observations), which means their effects are usually limited. When this is the case, using an incremental replanner such as D* Lite will be far more efficient than planning from scratch. However, this is not universally true. If the areas of the graph being changed are not necessarily close to the position of the robot, it is possible for D*Lite to be less efficient than A*. This is because it is possible for D*Lite to process every state in the environment twice: once as an under consistent state and once as an over consistent state. A*, on the other hand, will only ever process each state once. The worst-case scenario for D*Lite, and one that illustrates this possibility, is when changes are being made to the graph in the vicinity of the goal. It is thus common for systems using D* Lite to abort the replanning process and plan from scratch whenever either major edge cost changes are detected or some predefined threshold of replanning processing is reached.

Also, when navigating through completely unknown environments, it can be much more efficient to search forwards from the agent position to the goal, rather than backwards from the goal. This is because we typically assign optimistic costs to edges whose costs we don’t know. As a result, areas of the graph that have been observed have more expensive edge costs than the unexplored areas. This means that, when searching forwards, as soon as the search exits the observed area it can rapidly progress through the unexplored area directly to the goal. However, when searching backwards, the search initially rapidly progresses to the observed area, then once it encounters the more costly edges in the observed area, it begins expanding large portions of the unexplored area trying to find a cheaper path. As a result, it can be significantly more efficient to use A* rather than backwards A* when replanning from scratch. Because the agent is moving, it is not possible to use a forwards-searching incremental re-planner, which means that the computational advantage of using a replanning algorithm over planning from scratch is reduced.

As mentioned earlier, these algorithms can also be applied to symbolic planning problems. However, in these cases it is important to consider whether there is an available predecessor function in the particular planning domain. If not, it is necessary to maintain for each state s the set of all states s' that have used s as a successor state during the search, and treat this set as the set of predessors of s. This is also useful when such a predecessor function exists but contains a very large number of states; maintaining a list of just the states that have actually used s as a successor can be far more efficient than generating all the possible predecessors.

In the symbolic planning community it is also common to use inconsistent heuristics since problems are often infeasible to solve optimally. The extensions to D*Lite presented in enable D*Lite to handle in-consistent heuristics. These extensions also allow one to vary the tie-breaking criteria when selecting states from the OPEN list for processing. This might be important when a problem has many solutions of equal costs and the OPEN list contains a large number of states with the same priori-ties.

Apart from the static approaches (A*), all of the algorithms that we discuss in this paper attempt to reuse previous results to make subsequent planning tasks easier. However, if the planning problem has changed sufficiently since the previous result was generated, this result may be a burden rather than a useful starting point.

For instance, it is possible in symbolic domains that altering the cost of a single operator may affect the path cost of a huge number of states. As an example, modifying the cost of the load operator in the rocket domain may completely change the nature of the solution. This can also be a problem when path planning for robots with several degrees of freedom: even if a small change occurs in the environment, it can cause a huge number of changes in the complex con-figuration space. As a result, replanning in such scenarios can often be of little or no benefit.

ANYTIME ALGORITHMS

When an agent must react quickly and the planning problem is complex, computing optimal paths as described in the previous sections can be infeasible, due to the sheer number of states required to be processed in order to obtain such paths. In such situations, we must be satisfied with the best solution that can be generated in the time available.

A useful class of deterministic algorithms for addressing this problem are commonly referred to as anytime algorithms. Anytime algorithms typically construct an initial, possibly highly suboptimal, solution very quickly, then improve the quality of this solution while time permits. Heuristic-based anytime algorithms often make use of the fact that in many domains inflating the heuristic values used by A* (resulting in the weighted A* search) often pro-vides substantial speed-ups at the cost of solution optimality. Further, if the heuristic used is consistent[2], then multiplying it by an inflation factor > 1 will produce a solution guaranteed to cost no more than times the cost of an optimal solution. this property to develop an anytime algorithm that performs a succession of weighted A* searches, each with a decreasing inflation factor, Their approach provides suboptimality bounds for each successive search and has been shown to be much more efficient than competing approaches .

This algorithm, Anytime Repairing A* (ARA*), limits the processing performed during each search by only considering those states whose costs at the previous search may not be valid given the new value. It begins by performing an A* search with an inflation factor $_0$, but during this search it only expands each state at most once[3]. Once a state s has been expanded during a particular search, if it becomes inconsistent (i.e., g(s) 6= rhs(s)) due to a cost change associated with a neighboring state, then it is not reinserted into the queue of states to be expanded. Instead, it is placed into the INCONS list, which contains all inconsistent states already expanded. Then, when the current search terminates, the states in the INCONS list are inserted into a

A (forwards) heuristic h is consistent if, for all $s \in S$, $h(s, s_{goal}) \leq c(s, s') + h(s', s_{goal})$ for any successor s' of s, and $h(s_{goal}, s_{goal}) = 0$.

[3]It is proved in (Likhachev, Gordon, & Thrun 2003) that this still guarantees an $_0$ suboptimality bound.

```
key(s)
  01. return $g(s) + \cdot h(s_{start}, s)$;

ImprovePath()
  02. while ($\min_{s \in OPEN}(key(s)) < key(s_{start})$)
  3.    remove s with the smallest key(s) from OPEN;
  4.    CLOSED = CLOSED $\cup$ {s};
  5.    for all $s' \in$ P red(s)
  6.      if s' was not visited before
  7.        $g(s') = \infty$;
```

```
8.      if g(s’) > c(s’, s) + g(s)
9.        g(s’) = c(s’, s) + g(s);

10.       if s’ 6 CLOSED
11.         insert s’ into OPEN with key(s’);
12.       else

13.         insert s’ into INCONS;

Main()

14.     g(s_start) = ∞; g(s_goal) = 0;

15.     =  0;

16.     OPEN = CLOSED = INCONS = ∅;

17.     insert s_goal into OPEN with key(s_goal);

18.     ImprovePath();

19.     publish current  -suboptimal solution;

20.     while  > 1

21.   decrease  ;

22.   Move states from INCONS into OPEN;

23.   Update the priorities for all s ∈ OPEN according
      to key(s);

24.   CLOSED = ∅;

25.   ImprovePath();

26.   publish current  -suboptimal solution;
```

Figure 3: The ARA* Algorithm (backwards version).

fresh priority queue (with new priorities based on the new inflation factor) which is used by the next search. This improves the efficiency of each search in two ways. Firstly, by only expanding each state at most once a solution is reached much more quickly. Secondly, by only reconsidering states from the previous search that were inconsistent, much of the previous search effort can be reused. Thus, when the inflation factor is reduced between successive searches, a relatively minor amount of computation is required to generate a new solution. A simplified, backwards-searching version of the algorithm is given in Figure 3. Here, the priority of each state s in the OPEN queue is computed as the sum of its cost g(s) and its inflated heuristic value ·h(s_{start}, s). CLOSED contains all states already expanded once in the current search and INCONS contains all states that have already been expanded and are inconsistent.

APPLICABILITY: ANYTIME ALGORITHMS

ARA* has been shown to be much more efficient than com-peting approaches and has been applied successfully to path planning in high-dimensional state spaces, such as kinematic robot arms with 20 links. It has thus effectively extended the applicability of the backwards-searching version is shown because it will be useful when discussing the algorithm’s similarity to D* Lite. deterministic planning algorithms into much higher dimensions than previously possible. It has also been used to plan smooth trajectories for outdoor mobile robots in known environments. Figure 4 shows an outdoor robotic system that has used ARA* for this purpose. Here, the search space involved four dimensions: the (x, y) position of the robot, the robot’s orientation, and the robot’s velocity. ARA* is able to plan suboptimal paths for the robot very quickly, then improve the quality of these paths as the robot begins its traverse (as the robot moves the start state changes and therefore in between search iterations the heuristics are re-computed for all states in the OPEN list right before their priorities are updated).
ARA* is well suited to domains in which the state space is very large and suboptimal solutions can be generated efficiently. Although using an inflation factor usually expedites the planning process, this is not guaranteed. In fact, it is possible to construct pathological examples where the best-first nature of searching with a large can result in much longer processing times. The larger is, the more greedy the search through the space is, leaving it more prone to get-ting temporarily stuck in local minima. In general, the key to obtaining anytime behavior with ARA* is finding a heuristic function with shallow local minima. For example, in the case of robot navigation a local minimum can be a U-shaped obstacle placed on the straight line connecting a robot to its goal (assuming the heuristic function is Euclidean distance) and the size of the obstacle determines how many states weighted A*, and consequently ARA*, will have to process before getting out of the minimum.
Depending on the domain one can also augment ARA* with a few optimizations. For example, in graphs with considerable branching factors the OPEN list can grow prohibitively large. In such cases, one can borrow an interesting idea from the OPEN list whose priorities based on un-inflated heuristic are already larger than the cost of the current solution (e.g., g(s_{goal}) in the forwards-searching version).
However, because ARA* is an anytime algorithm, it is only useful when an anytime solution is desired. If a solution with a particular suboptimality bound of $_d$ is

desired, and no intermediate solution matters, then it is far more efficient to perform a weighted A* search with an inflation factor of $_d$ than to use ARA*.
Further, ARA* is only applicable in static planning domains. If changes are being made to the planning graph, ARA* is unable to reuse its previous search results and must replan from scratch. As a result, it is not appropriate for dynamic planning problems. It is this limitation that motivated research into the final set of algorithms we discuss here: any-time replanners.

ANYTIME REPLANNING ALGORITHMS

Although each is well developed on its own, there has been relatively little interaction between the above two areas of research. Replanning algorithms have concentrated on finding a single, usually optimal, solution, and anytime algorithms have concentrated on static environments. But some of the most interesting real world problems are those that are both dynamic (requiring replanning) and complex (requiring anytime approaches).
As a motivating example, consider motion planning for a kinematic arm in a populated office area. A planner for such a task would ideally be able to replan efficiently when new information is received indicating that the environment has changed. It would also need to generate suboptimal solutions, as optimality may not be possible given limited deliberation time.
This developed Anytime Dynamic A* (AD*), an algorithm that combines the replanning capability of D*Lite with the anytime performance of ARA*. AD* performs a series of searches using decreasing inflation factors to generate a series of solutions with improved bounds, as with ARA*. When there are changes in the environment affecting the cost of edges in the graph, locally affected states are placed on the OPEN queue to propagate these changes through the rest of the graph, as with D*Lite. States on the queue are then processed until the solution is guaranteed to be -suboptimal.
The algorithm is presented in Figures 5 and 6. AD* begins by setting the inflation factor to a sufficiently high value $_0$, so that an initial, suboptimal plan can be generated quickly. Then, unless changes in edge costs are detected, is gradually decreased and the solution is improved until it is guaranteed to be optimal, that is, = 1. This phase is exactly the same as for ARA*: each time is decreased, all inconsistent states are moved from INCONS to OPEN and CLOSED is made empty. When changes in edge costs are detected, there is a chance that the current solution will no longer be -suboptimal. If the changes are substantial, then it may be computation-ally expensive to repair the current solution to regain -suboptimality. In such a case, the algorithm increases so As with D*Lite the optimizations can be used to substantially speed up AD* and are recommended for an efficient implementation of the algorithm.

```
key(s)
  1.  if (g(s) > rhs(s))
  2.      return [min(g(s), rhs(s)) + · h(s_start, s); min(g(s), rhs(s)))];
  3.  else
  4.      return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s)))];
UpdateState(s)
  05. if s was not visited before
  6.      g(s) = ∞;
  7.  if (s 6= s_goal) rhs(s) = min_{s'∈Succ(s)}(c(s, s') + g(s'));
  8.  if (s ∈ OPEN) remove s from OPEN;
  9.  if (g(s) 6= rhs(s))
  10.     if s6∈CLOSED
  11.         insert s into OPEN with key(s);
  12.     else
  13.         insert s into INCONS;
ComputeorImprovePath()
  14. while (min_{s∈OPEN}(key(s)) <˙ key(s_start) OR rhs(s_start) 6= g(s_start))
  15.     remove state s with the minimum key from OPEN;
  16.     if (g(s) > rhs(s))
  17.         g(s) = rhs(s);
  18.         CLOSED = CLOSED ∪ {s};
  19.         for all s' ∈ P red(s) UpdateState(s');
  20.     else
  21.         g(s) = ∞;
  22.         for all s' ∈ P red(s) ∪ {s} UpdateState(s');
```

Figure 5: Anytime Dynamic A*: ComputeorImprovePath function.

```
Main()
  1.  g(s_start) = rhs(s_start) = ∞; g(s_goal) = ∞;
  2.  rhs(s_goal) = 0;  = _0;
  3.  OPEN = CLOSED = INCONS = ∅;
  4.  insert s_goal into OPEN with key(s_goal);
  5.  ComputeorImprovePath();
  6.  publish current  -suboptimal solution;
  7.  forever
```

```
8.  if changes in edge costs are detected
9.      for all directed edges (u, v) with changed edge costs
10.         Update the edge cost c(u, v);
11.         UpdateState(u);
12. if significant edge cost changes were observed
13.     increase  or replan from scratch;
14. else if  > 1
15.     decrease  ;
16. Move states from INCONS into OPEN;
17. Update the priorities for all s ∈ OPEN according to key(s);
18. CLOSED = ∅;
19. ComputeorImprovePath();
20. publish current  -suboptimal solution;
21. if  = 1
22.     wait for changes in edge costs;
```

Figure 6: Anytime Dynamic A*: Main function.

that a less optimal solution can be produced quickly. Because edge cost increases may cause some states to become under consistent, a possibility not present in ARA*, states need to be inserted into the OPEN queue with a key value reflecting the minimum of their old cost and their new cost. Further, in order to guarantee that under consistent states propagate their new costs to their affected neighbors, their key values must use admissible heuristic values. This means that different key values must be computed for under consistent states than for over consistent states.

By incorporating these considerations, AD* is able to handle both changes in edge costs and changes to the inflation factor. Like the replanning and anytime algorithms we've looked at, it can also be slightly modified to handle the situation where the start state s_{start} is changing, as is the case when the path is being traversed by an agent. This allows the agent to improve and update its solution path while it is being traversed.

Applicability: Anytime Replanning Algorithms

AD* has been shown to be useful for planning in dynamic, complex state spaces, such as 3 DOF robotic arms operating in dynamic environments . It has also been used for path-planning for outdoor mobile robots. In particular, those operating in dynamic or partially-known outdoor environments, where velocity considerations are important for generating smooth, timely trajectories. As discussed earlier, this can be framed as a path planning problem over a 4D state space, and an initial suboptimal solution can be generated using AD* in exactly the same manner as ARA*.

Once the robot starts moving along this path, it is likely that it will discover inaccuracies in its map of the environment. As a result, the robot needs to be able to quickly re-pair previous, suboptimal solutions when new information is gathered, then improve these solutions as much as possible given its processing constraints.

AD* has been used to provide this capability for two robotic platforms currently used for outdoor navigation: an ATRV and a Segway Robotic Mobility Platform (Segway RMP), To direct the 4D search in each case, a fast 2D (x, y) planner was used to provide the heuristic values.

Acknowledgments

I **shailesh verma** would like to express my deep and sincere gratitude to my research supervisor, for giving me the opportunity to do research and providing invaluable guidance throughout the research.

REFERENCES

1. Li, X. Ma, Y. Feng, X. (2013). Self-adaptive autowave pulse-coupled neural network for shortest-path problem, Neuro computing 115, pp. 63−71.
2. Zhang, Y. Wu, L. Wei, G. Wang, S. (2011). A novel algorithm for all pairs shortest path problem based on matrix multiplication and pulse coupled neural network, Digital Signal Processing 21, pp. 517−521.

Lilly, J. H. (2007). Evolution of a negative-rule fuzzy obstacle avoidance controller for an autonomous

3. vehicle, IEEE Trans. Fuzzy Systems, Vol.15, pp. 718−728.
4. A. Zhu, S. X. Yang, A Fuzzy Logic Approach to Reactive Navigation of Behavior-based Mobile Robots, IEEE International Conference on Robotics and Automation (ICRA), New Orleans, LA, pp. 5045−5050, 2004.
5. Ahmed, F. Deb, K. (2013) Multi-objective optimal path planning using elitist non-dominated sorting genetic algorithms, Soft Computing, Vol. 17, pp.1283−1299.
6. Sleumer Nora, Tschichold-Gurmann, Nadine. "Exact cell decomposition ofvarrangements used for path planning in robotics", technical report/ETH zürich. Department of Computer Science; 1999. https://doi.org/10.3929/ ethz-a-006653440.
7. Cai Chenghui, Ferrari Silvia. Information-driven sensor path planning by approximate cell decomposition. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) June 2009;39(3). https://doi.org/10.1109/ TSMCB.2008.2008561.
8. Kubota, N., Moriokab, T., Kojimac, F., & Fukudad, T. (2001). Learning of mobile robots using perceptionbased genetic algorithm. Measurement, 29, 237–248.

9. Wang, J., Zhang, Y., & Xia, L. (2010, March 13–14). Adaptive genetic algorithm enhancements for path planning of mobile robots. In International conference on measuring technology and mechatronics automation (ICMTMA),Changsha,China (Vol. 3, pp. 416–419). IEEE. doi:10.1109/ICMTMA.2010.44
10. Yun, S. C., Ganapathy, V., & Chong, L. O. (2010, December 7–10). Improved genetic algorithms based optimum path planning for mobile robot. In 11th international conference on control, automation, robotics and vision, Singapore (pp. 1565–1570). IEEE. doi:10.1109/ICARCV.2010.5707781.
11. Alfaro, H. M., & Garcia, S. G. (1998). Mobile robot path planning and tracking using simulated annealing and fuzzy logic control. Expert Systems with Applications, 15, 421–429.
12. Chen, X., Kong, Y., Fang, X., & Wu, Q. (2013). A fast two-stage ACO algorithm for robotic path planning. Neural Computing and Applications, 22, 313–319.
13. Parhi, D. R., & Pothal, J. K. (2010). Intelligent navigation of multiple mobile robots using an ant colony optimization technique in a highly cluttered environment. Proceedings of IMechE Part C: Journal Mechanical Engineering Science, 225, 225–232.
14. Liu, C., Gao, Z., & Zhao, W. (2012, June 23–26). A new path planning method based on firefly algorithm. In Fifth international joint conference on computational sciences and optimization (CSO), Harbin, China (pp. 775–778). IEEE. doi:10.1109/CSO.2012.174
15. Juang, C. F., & Chang, Y. C. (2011). Evolutionary-group-based particle-swarm-optimized fuzzy controller with application to mobile-robot navigation in unknown environments. IEEE Transactions on Fuzzy Systems, 19, 379–391.
16. Lu, L., & Gong, D. (2008, October 18–20). Robot path planning in unknown environments using particle swarm optimization. In Fourth international conference on natural computation, Jinan (pp. 422–426). IEEE. doi:10.1109/ICNC.2008.923.
17. Meng Wang and James N. K. Liu (2008), "Fuzzy Logic Navigation in Unknown Environment with Dead Ends", Robotics and Atonomous Systems, 56: 625-643.
18. Kundu, S., & Parhi, D. R. (2016). Navigation of underwater robot based on dynamically adaptive harmony search algorithm. Memetic Computing, 8(2), 125-146.
19. Mohanty, P. K., & Parhi, D. R. (2013). Controlling the motion of an autonmous mobile robot using various techniques: a review. Journal of Advance Mechanical Engineering, 1(1), 24-39.
20. Singh, M. K., & Parhi, D. R. (2009, January). Intelligent neuro controller for navigation of mobile robot.In Proceedings of the International conference on advances in computing, communication and control (pp.13-128).ACM.
21. Pradhan, S. K., Parhi, D. R., & Panda, A. K. (2006). Navigation of multiple mobile robots using rule-based neuro-fuzzy technique. International Journal of Computational Intelligence, 3(2), 142-152.
22. Toda M., Kitani O., Okamoto T. and Torii T. (1999), "Navigation Method for a Mobile Robot via Sonar Based Crop Row Mapping and Fuzzy Logic Control", Journal of Agricultural Engineering Research, 72 (4): 299–309.
23. Jena, P. K., & Parhi, D. R. (2015). A modified particle swarm optimization technique for crack detection in Cantilever Beams. Arabian Journal for Science and Engineering, 40(11), 3263-3272.
24. Parhi, D. R., & Mohanty, P. K. (2016). IWO-based adaptive neuro-fuzzy controller for mobile robot navigation in cluttered environments. The International Journal of Advanced Manufacturing Technology, 83(9-12), 1607-1625.