Chapter 4

Search-Based Planning and Replanning in Robotics and Autonomous Systems

An T. Le and Than D. Le

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.71663

Abstract

In this chapter, we present one of the most crucial branches in motion planning: search-based planning and replanning algorithms. This research branch involves two key points: first, representing traverse environment information as discrete graph form, in particular, occupancy grid cost map at arbitrary resolution, and, second, path planning algorithms calculate paths on these graphs from start to goal by propagating cost associated with each vertex in graph. The chapter will guide researcher through the foundation of motion planning concept, the history of search-based path planning and then focus on the evolution of state-of-the-art incremental, heuristic, anytime algorithm families that are currently applied on practical robot rover. The comparison experiment between algorithm families is demonstrated in terms of performance and optimality. The future of search-based path planning and motion planning in general is also discussed.

Keywords: A*, RRT, holonomic path planning, trajectory planning, occupancy map, D* Lite, incremental planning, heuristics planning, ARA*, anytime dynamic A*

1. Introduction

Nowadays, as the rapid advances of computational power together with development of state-of-the-art motion planning (MP) algorithms, autonomous robots can now robustly plan optimal path in narrow configuration space or wide dynamic complex environment with high accuracy and low latency. These recent MP developments have a large impact in medical surgery, animation, expedition and many other disciplines. For instance, RRT [1] algorithm was applied for multi-arm surgical robot in [2]. Expedition robot GDRS XUV was implemented field D* any-angle path planner [3] that enables the robot to optimally move in harsh environment. D* [4] is implemented for Mars Rover prototypes and tactical mobile robots in [5]. Bug algorithms were implemented in multi-robot cooperation scenarios [6].

IntechOpen

© 2018 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. cc BY

In general, the problem statement of MP can be generalised as follows: Given the initial defined world space and the robot's configuration space, the MP algorithm must generate a series of consecutive collision-free configurations of the robot that connects start configuration and goal configuration. This series configuration must satisfy any inherent motion or non-motion constraints of the robot.

To cope with a wide range of environment characteristics, MP can be divided into two categories: gross MP and fine MP [7]. The gross MP concerns with the scenarios when world space is much wider than obstacles' size and positional error of the robot, whereas the fine MP solves the planning problems in narrow space that requires high accuracy.

This manuscript presents the development of gross MP algorithm family, in particular search-based planning and replanning paradigm. The foundation concepts of MP, configuration space representations, and the position of mentioned paradigm in MP big picture is presented in Section 2. Section 3 describes historical basis of search-based algorithm family. Section 4 demonstrates the properties and pitfall of D* Lite, which is one of the most crucial algorithms to plan path in dynamic environment. After that, the variants of D* Lite, which improve D* Lite's optimality and performance, are presented. To confirm the improvements, we provide experimental results of recent path planning algorithms and their comparisons in terms of performance and optimality in Section 5. Section 6 will discuss about the future development of MP and provide conclusion.

2. Motion planning concepts

This section will provide an overview of the basic elements that every MP problem must involve. These elements are configuration space of robot and obstacles, environment representation, MP method and search method. The mentioned factors must be analysed consecutively in order to apply suitable MP algorithm family for each scenario.

2.1. Classification of motion planning problems

There still does not exist unified MP algorithm that can robustly solve MP problems in any scenarios such as time optimality, path optimality, moving target, non-holonomic motion, etc. However, with the active recent development of MP, a variety of MP algorithm families are invented to deal with the mentioned scenarios. We will provide detail MP algorithm family classifications based on problem type and therefore demonstrate the location of search-based paradigm in MP.

Figure 1 describes the family tree of MP algorithms based on problem-type classification.

As can be seen, MP with non-holonomic (velocity and kinodynamic) constraints, which is handled by sampling-based paradigm, is still an open research area due to the hardness of transforming high DOF robot and surroundings into configuration space. This configuration space problem has been proved to be NP hard, and computing configuration space operation has

http://dx.doi.org/10.5772/intechopen.71663

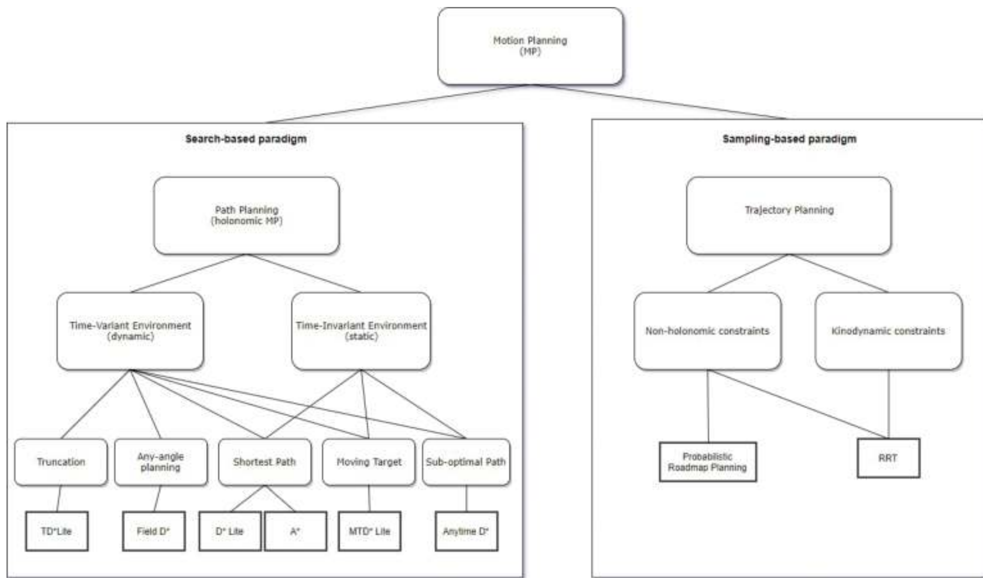

Figure 1. Classification of MP algorithm families based on problem type; the deepest leaves of algorithm tree are representatives for their families.

exponential lower bound [7]. Until recently, the mainstream of non-holonomic MP research is developed based on random rapidly exploring random tree planner (RRT). For example, heuristics property of A* [8] has been applied to RRT for faster trajectory convergence [9]. Fast Marching Square method was developed for non-holonomic car-like robot based on RRT that produces smoother trajectory than RRT [10].

Unlike sampling-based paradigm, search-based paradigm, which represents for path planning algorithms, has a long history of evolution, from basic graph searching to dynamic motion planner with constraints. In this paradigm, robot is treated as point or scalar robot that is able to move in any direction at any time interval. Hence, the configuration obstacle space has the same dimension with the environment, and the generated trajectory is just a path in operating environment. Search-based paradigm is divided into time-invariant and time-variant environment categories. A* is the representative for time-invariant algorithm family; its cost function is incorporated with heuristic property for faster optimal path planning. When dealing with time-variant problem, although we can ensure the optimality and correctness of path solution, we cannot just rerun A* from the point that the robot detects changes in environment due to high latency. To efficiently path replanning in dynamic environment, incremental property is combined with heuristic property to develop D* Lite algorithm; this algorithm is the basis for future development of search-based replanning. Many variants of D* Lite for different MP problems are presented in **Table 1**.

The development of search-based algorithm family is described detail in Section 3 and Section 4.

Problem scenarios	Algorithms
Moving target	MTD* Lite [11]
Fast/suboptimal	Anytime D* [12], truncated D* Lite [13], anytime Truncated D* [14]
Any-angle movement	Field D* [3], incremental Phi* [15]
Performance improvement	D* Lite with Reset [16]

Table 1. Different families of D* Lite variants.

2.2. Problem statement formulation

The general MP problem can be formulated as the following six terms:

1. **State space**: the configuration space of the robot transformed from physical space, W.
2. **Boundary values**: $x_{init} \in W$ and $X_{goal} \subset W$.
3. **Collision detector:** $D: W \rightarrow \{true, \ false\}$ the function to detect whether the global constraints are satisfied from robot state x; it can output binary or real values.
4. **Input space:** a set U of input, which specifies a complete set of robot operation that affects the state x.
5. **Incremental rules:** a set of rules to transition state $x(t)$ to state $x(t + \Delta t)$ when an operation is input over time interval, $\{u(t_c) \mid t < t_c < t + \Delta t\}$.
6. **Metric:** a real-valued function $\rho: W \times W \rightarrow [0, +\infty)$ that defines the distance between two points in state space W.

General MP is viewed as a search for path (a series of configuration) in state space W that connects start configuration x_{init} to goal configuration region X_{goal}. The robot is incorporated with a set of global constraints (small discrete headings, velocity, balancing, etc.). We denote W_{free} as a set of configuration that satisfies global constraints, and the generated path must be in W_{free}. The incremental rules can be considered as discrete-time response system, and together with input space, it defines possible robot state transitions. Metric can affect heavily to the algorithm's optimality and performance; it indicates the distance between pair of points in topological space. One can construct MP algorithm to deal with specific constraints in certain environment by following these basic terms.

2.3. Environment representation

This section will describe the transformation of world space to state space. This is the first step to formulate a MP algorithm; it creates an operation environment for MP algorithm and a way to represent physical world information as data structure in computer.

2.3.1. Configuration space (C-space) transformation

The world space (physical space) is where the robot and obstacles exist; it is a map of the practical world. However, we cannot apply directly MP algorithm to this space due to the

hardness of representing orientation dimension and other parameters such as motion constraints on computer. Therefore, a C-space is needed, which incorporates all independent parameters that completely define the position of all points on the robot and specifies global constraints of the robot as Cartesian space. **Figure 2** [17] shows a mapping between an effector of 2DOF robot arm and a set of possible two angle parameters that constitutes C-space of the effector.

After computing the C-space, all MP problems are basically reduced to finding a series of configuration that connects start configuration and goal configuration. In other word, the problem is reduced to finding a path for a point robot from start to goal. The number of parameters that defines robot position is the dimension of C-space. The method to compute C-space is mentioned in [7].

For simplicity, to follow the scope of this chapter, we will treat C-space of point robot the same as world space; the reason is that search-based paradigm deals with holonomic MP problem in which the size of robot is neglected compared to operating environment.

2.3.2. Continuous to discrete approximation

After transforming world space to C-space, we still cannot apply search-based algorithms to C-space. The problem is that search-based algorithms like A* or D* Lite work on graph-like structures; hence, applying search-based algorithms on continuous C-space is intractable. However, other MP algorithm families such as sampling based can apply directly to C-space. Unfortunately, the path optimality and performance of sampling-based algorithms are currently worse than state-of-the-art search-based algorithms.

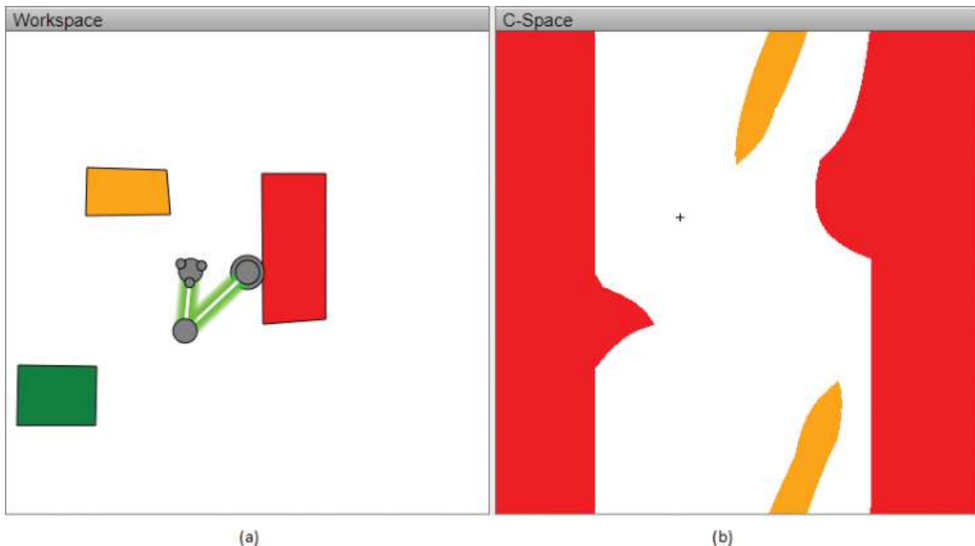

Figure 2. Configuration space of 2DOF robot arm that represents a set of collision-free angles in white and specific object collided in colours (a) Workspace, (b) C-space.

There are two main approaches to discretize C-space into graph-like structure:

- Cell decomposition
- Roadmap

In cell decomposition approach, we divide C-space into eight-connected square grid environment with arbitrary resolution. Then we colour all cells that intersect with obstacle configuration with black, and other free cells are white. **Figure 3** illustrates this approach.

This approximation has limited assumptions on obstacle configuration. Therefore, the approach is used widely in practice. However, there is no concept of path optimality, because we can infinitely divide C-space into smaller squares. It is a trade-off between optimality and computation. Cell decomposition in high dimensions is also expensive; it has exponential growth in PSPACE.

In roadmaps approach, the idea is avoiding scanning the entire C-space by computing an undirected graph with "road" edges that are guaranteed to be collision-free. The main methods of this approach are visibility graph [17] and Voronoi diagrams. The examples of the two methods are demonstrated in **Figure 4**.

As can be seen, this approach generates fewer vertices than cell decomposition approach. Visibility graph method tends to generate with vertices that are the vertices of obstacles; this property leads to finding shortest path. However, the visibility graph's roadmaps are close to obstacles; collision is inevitable due to some movement error. Voronoi diagram solves the problem by generating roadmaps that keep robot as far away as possible from obstacles.

Despite this approach constructs efficiently graph representation for search-based algorithm; it is difficult to compute in higher dimension or non-polygonal environment. The approach

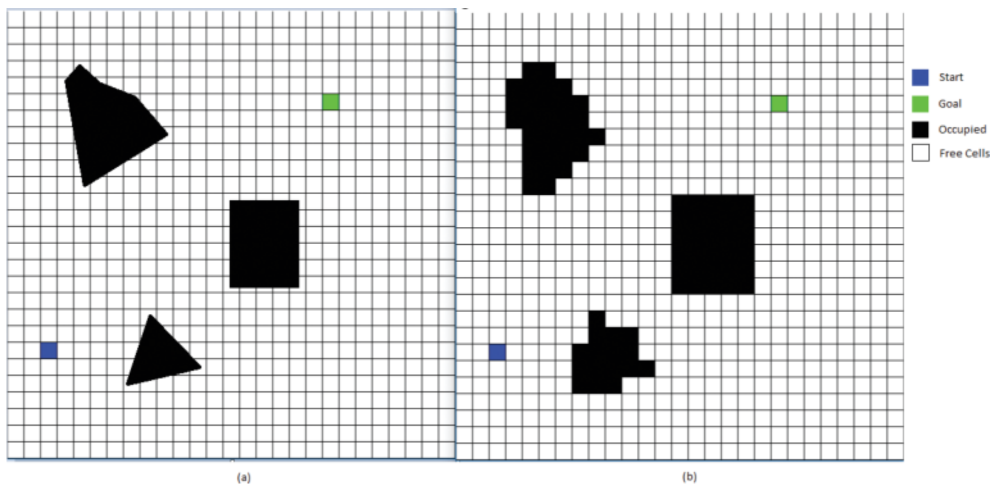

Figure 3. Cell decomposition approach (a) Original Objects, (b) Encoded Objects into cells.

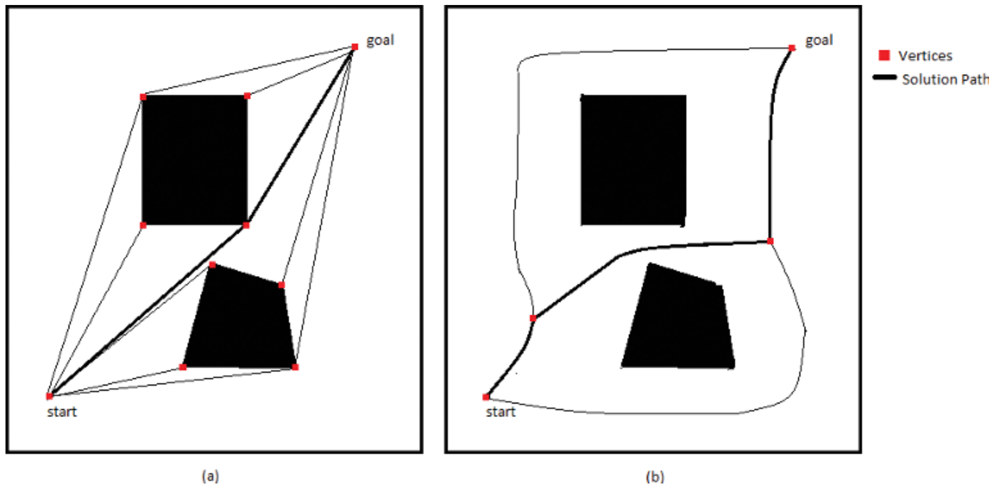

Figure 4. Path topologies of visibility graph and Voronoi diagram methods in roadmap approach (a) Visibility Graph method, (b) Voronoi method.

also can be unstable in dynamic scenarios; small changes in obstacles can lead to large changes in graph.

In the following sections, we use cell decomposition approach for search-based algorithms due to its clarity to describe the operation of search-based algorithms and its feasibility to apply in practice.

3. Search-based planning on time-invariant environment

This section demonstrates one of the most well-known algorithms in graph search family: A*. The A* algorithm's properties are also examined and utilised to use in different cases.

3.1. A* algorithm

There are three main properties of A* [8] that are inherited from historical graph search algorithms:

- **Search tree:** a search tree T, which root is the starting cell, stores expanded cells as branches. This tree is capable to extract path to starting cell from any expanded cell s in the map. A* inherits this tree from breadth-first search algorithm.
- **Uniform cost search:** This property includes a data structure $g(s)$ that stores the cost to travel from starting cell to any cell s in the map, which is formulated as

$$f(s) = g(s), \tag{1}$$

where f(s) is the priority of cell in open list O; the smaller the f(s), the higher the priority. The open list O handles processing expanding cells, and therefore this property prioritises expanding cells with less cost to travel. A* inherits this property from Dijkstra's algorithm.

- **Heuristic:** a rule to guide expanding search towards goal cell. This rule is formulated:

$$f(s) = h(s), \tag{2}$$

where $h(s)$ is the heuristics function for each cell s that indicates the closeness from cell s to goal. $h(s)$ can be Euclidean distance or Manhattan distance function in this case. In addition, $h(s)$ must satisfy admissible property:

$$h(s) \leq cost(s, s') + h(s'), \tag{3}$$

for any successor s' of s to ensure path optimality. A* inherits this property from greedy best-first search.

Figure 5 illustrates each property of A* when they are applied to search for goal:

The total expanded cells in each algorithm constitutes for their performance (e.g. how many cells are processed before path is found). As can be seen, Dijkstra's algorithm has the worst performance due to lack of guidance to expand search; it just expands uniformly to all directions. Greedy best-first search has the best computation; however, it does not guarantee the shortest path like Dijkstra's algorithm, because its search is trap in local minima shown in the picture. A* has both computation and optimality advantages over these old algorithms by combining uniform cost search rule to guarantee path optimality and heuristic rule of greedy best-first search to guide search process towards goal. Both rules can be combined and formulated as priority function:

$$f(s) = g(s) + h(s). \tag{4}$$

Intuitively, one could think f(s) is an estimated cost to travel from start cell to goal through concerning cell s. Hence, A* expands towards cells that have least cost travel (**Figure 6**, line 11).

The pseudo code for A* is shown in **Figure 6**.

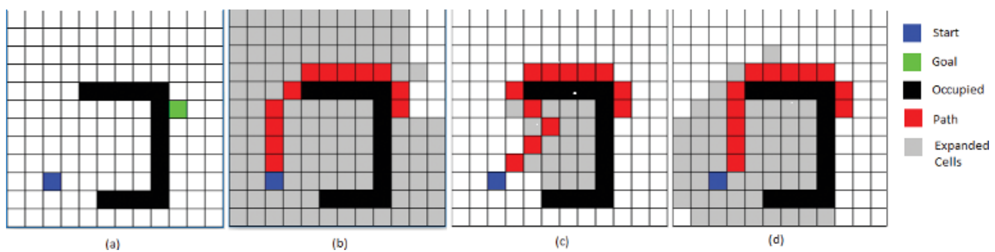

Figure 5. Operation demonstration of properties of A* and A* itself (a) Map, (b) Uniform cost search, (c) Greedy Best-First Search and (d) A*.

Algorithm 1: A*

Input : Graph G, s_{start}, s_{goal}
Output: Search Tree TREE

```
1  OPEN = TREE = ∅
2  Set all nodes g(s) = ∞
3  s_start = 0
4  OPEN.insert(s_start, g(s_start) + h(s_start, s_goal))
5  TREE(s_start) = None
6  while OPEN is not empty OR s is not s_goal do
7  |   s = OPEN.Pop()
8  |   forall s' ∈ Successor(s) do
9  |   |   if g(s) + cost(s, s') < g(s') then
10 |   |   |   g(s') = g(s) + cost(s, s')
11 |   |   |   OPEN.insert(s', g(s') + h(s', s_goal))
12 |   |   |   TREE(s') = s
13 |   |   end
14 |   end
15 end
```

Figure 6. Pseudo code of A* algorithm.

3.2. Anytime A*: path suboptimal bound (ARA*) algorithm

In practice, the performance issue is more critical; time for robot to "think" before making decision is limited. Therefore, a path planner, which has these properties, is essential:

- Quickly producing a suboptimal solution and then gradually improving its solution as time allowed by reusing its previous search effort as much as possible
- Having control over the suboptimal bound and hence indicating a bound of processing time of each search iteration

We introduce the algorithm that is well-suited for this scenario: ARA* [18].

Basically, ARA* is developed from A*; it inherits all intrinsic properties of A*. The idea to quickly plan suboptimal path is derived from inflated heuristics function [18] by a factor ε. The search is greedier to provide solution faster, and the solution is proven to be bounded:

$$g^*(s) \leq g(s) \leq \varepsilon * g^*(s), \tag{5}$$

where $g^*(s)$ is the optimal path cost from start to s.

The pseudo code for ARA* is shown in **Figure 7**.

To understand the behaviours of ARA*, we must keep in mind that ARA* violates admissible property—$h(s) \leq cost(s, s') + h(s')$—for any successor s' of s. ARA* modifies A* $f(s)$ function by inflating heuristics function $h(s)$:

```
Algorithm 2: ARA*
1  Function Key(s):
2  |  return g(s) + ε * h(s)
3  Function ImprovePath():
4  |  while Key(s_goal) > min_{s∈OPEN}(Key(s)) do
5  |  |  s = OPEN.Pop()
6  |  |  CLOSED.append(s)
7  |  |  forall s' ∈ Successor(s) do
8  |  |  |  if g(s) + cost(s, s') < g(s') then
9  |  |  |  |  g(s') = g(s) + cost(s, s')
10 |  |  |  |  if s' ∉ CLOSED then
11 |  |  |  |  |  OPEN.insert(s', Key(s'))
12 |  |  |  |  else
13 |  |  |  |  |  INCONS.insert(s')
14 |  |  |  |  end
15 |  |  |  end
16 |  |  end
17 |  end
18 Function Main():
19 |  Set all nodes g(s) = ∞
20 |  OPEN = CLOSED = INCONS = ∅
21 |  g(s_start) = 0
22 |  OPEN.insert(s_start, Key(s_start))
23 |  ImprovePath()
24 |  Choose initial sub-optimal bound ε
25 |  Publish ε
26 |  while ε > 1 do
27 |  |  Decrease ε
28 |  |  Move nodes in INCONS to OPEN
29 |  |  Update all priority values in OPEN according to Key(s)
30 |  |  CLOSED = ∅
31 |  |  ImprovePath()
32 |  |  Publish ε
33 |  end
```

Figure 7. Pseudo code of ARA* algorithm.

$$f(s) = g(s) + \varepsilon * h(s). \tag{6}$$

Hence, the computed path is no longer optimal. Moreover, each search iteration is no longer guaranteed to expand searching each cell at most once like A* due to decreasing ε. However, to maintain efficiency and ensure suboptimal bound, ARA* introduces INCONS list to store local inconsistent cells as specified function:

$$g(s') > \min_{s'' \in pred(s')}(cost(s', s'') + g(s'')), \tag{7}$$

http://dx.doi.org/10.5772/intechopen.71663

(**Figure 7**, line 13) that already are expanded once and processes these cells in the next search iteration.

In general, ARA* executes consecutive search iterations with decreasing suboptimal bound; each search does not recalculate consistent cells from previous search. Therefore, the path improvement process is efficient. Theoretical properties of ARA* is described in [18].

4. Search-based replanning on time-varying environment

In real-world application, there is often a scenario that the robot initially does not know a priori information about its surroundings. We cannot encode the world space information each time the robot runs, because it is expensive, tedious, and infeasible due to rapid changes in practice. To maintain collision-free path, one can naively rerun A* to replan the shortest path from the point that the robot detects changes. However, this naïve approach will waste computation by reprocessing cells that are irrelevant to compute a new path and hence increase idle time between each search. This section will demonstrate search-based algorithms to solve mentioned problem in time-variant environment.

4.1. Incremental heuristic algorithm: D* Lite algorithm

4.1.1. D Lite algorithm*

In goal-directed navigation task, with cell decomposition approximation, the robot always observes a limited range of eight connected grids. The robot is able to move in eight directions with cost one, and it assumes that unknown cells are traversable. The robot follows the initial calculated path to goal and encounters blockage cells; it must be able to process only cells that are relevant to compute the new path. The challenge is to find these relevant cells. **Figure 8** illustrates this idea.

Note that grey cells (in **Figure 8**) are expanded cells to compute initial path or new path when robot detects blockage cell in purple at position yellow cell. Darker grey cells are processed multiple times. As can be seen, total expanded cells in replanning process of D* Lite is 61, whereas expanded cells of rerunning A* are 75.

D* Lite [19] is developed directly from Lifelong Planning A* (LPA*) [20] for applying on mobile robot, which is a combination of Dynamic SWSF-FP [21] and A* [8]. Therefore, D* Lite possesses these properties:

- Reverse search: Unlike A*, D* Lite expands its search from goal; $h(s)$ now indicates the closeness from cell s to start cell. $g(s)$ now also stores estimated distance from goal. After searching is finished, the path from start to goal is generated by iteratively moving from cell s towards neighbour cells s' that have the lowest sum $g(s) + cost(s, s')$ in greedy style.
- Heuristics: D* Lite inherits this property from A* with admissible rule. Thus, D* Lite maintains path optimality by expanding heuristically towards start cell.

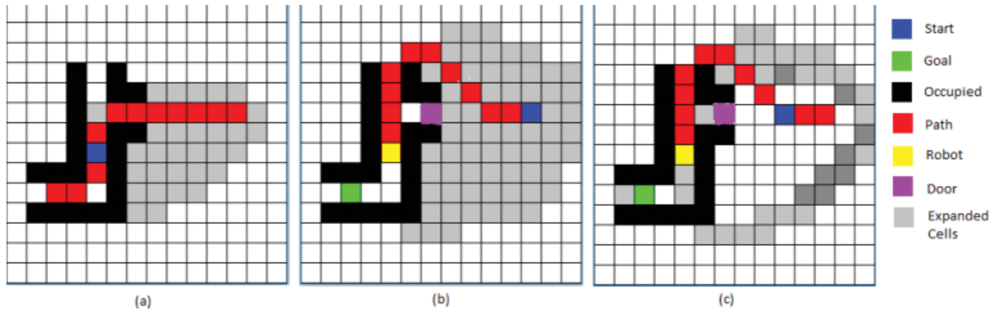

Figure 8. MP simulation on grid environment (a) Initial path, (b) Reset A* and (c) D* Lite.

- Incremental: D* Lite inherits incremental search property from Dynamic SWSF-FP; it re-uses information from previous search to repair path in a series of similar searches, which is much efficient than calculating path from scratch.

The pseudo code for D* Lite is shown in **Figure 9**.

In general, the pseudo code of D* Lite maintains three invariants:

- Invariant 1: $rhs(s) = \begin{cases} 0 & if\ s = s_{start} \\ min_{s' \in Pred(s)}(g(s') + cost(s, s')) & otherwise \end{cases}$.
- Invariant 2: OPEN list contains exactly only local inconsistent cells $g(s) \neq rhs(s)$.
- Invariant 3: Priority value of cells in OPEN list is equal to its $Key(s)$.

At the first run, D* Lite is exactly like A*. It guarantees to expand cells at most twice in each search routine due to the concept of one-step look-ahead estimated goal distance r $hs(s)$ that is inherited from LPA*. r $hs(s)$ leads to the terms of over-consistent cell $g(s) > rhs(s)$ and under-consistent cell $g(s) < rhs(s)$. Intuitively, these concepts help propagating the inconsistency of cells to their neighbours. To maintain Invariants 1 and 2, ComputePath() function updates rhs-values of changed cells, checks their consistency and decides their membership of OPEN list accordingly. Invariant 3 is maintained by updating the OPEN list keys while expanding (**Figure 9**, lines 17–18). ComputePath() stops when the smallest key of OPEN list is less than $Key(s_{start})$ or s_{start} is consistent; this criteria indicates that cell expansion has reached target s_{start}. Theorems of D* Lite are described detail in [19].

4.1.2. Pitfall of D Lite*

Despite being an effective replanner for dynamic environment, D* Lite does have a big pitfall for certain circumstances. In fact, D* Lite is designed to be implemented in mobile robot with range sensors, in which the environment changes are perceived near the robot (the starting cell). In other word, the changes occurred at the perimeter of expansion. Therefore, D* Lite just propagates inconsistencies in a small area near the search front; the replanning process is efficient. However, the problem arises when we combine other sensors (e.g. UAV, satellite,

http://dx.doi.org/10.5772/intechopen.71663

Algorithm 3: D* Lite

```
1  Function Key(s):
2  |  return
   |  [min(g(s), rhs(s)) + h(s_start, s) +
   |  k_m; min(g(s), rhs(s))]
3  Function UpdateVertex(s):
4  |  if s ≠ s_goal then
5  |  |  rhs(s) =
   |  |  min_{s'∈Succ(s)}(cost(s, s') +
   |  |  g(s'))
6  |  end
7  |  if s ∈ OPEN then
8  |  |  OPEN.remove(s)
9  |  end
10 |  if g(s) ≠ rhs(s) then
11 |  |  OPEN.insert(s, Key(s))
12 |  end
13 Function ComputePath():
14 |  while
   |  OPEN.TopKey() < Key(s_start)
   |  OR rhs(s_start) ≠ g(s_start) do
15 |  |  k_old = OPEN.TopKey()
16 |  |  s = OPEN.Pop()
17 |  |  if k_old < Key(s) then
18 |  |  |  OPEN.insert(s, Key(s))
19 |  |  else if g(s) > rhs(s) then
20 |  |  |  g(s) = rhs(s)
21 |  |  |  forall s' ∈ Pred(s) do
22 |  |  |  |  UpdateVertex(s')
23 |  |  |  end
24 |  |  else
25 |  |  g(s) = ∞
26 |  |  forall s' ∈ Pred(s) ∪ {s} do
27 |  |  |  UpdateVertex(s')
28 |  |  end
29 |  end
30 Function Main():
31 |  forall s ∈ S do
32 |  |  rhs(s) = g(s) = ∞
33 |  end
34 |  s_last = s_start
35 |  OPEN = ∅
36 |  rhs(s_goal) = 0; k_m = 0
37 |  OPEN.insert(s_goal, Key(s_goal))
38 |  ComputePath()
39 |  while s_start ≠ s_goal do
40 |  |  s_start =
   |  |  argmin_{s'∈Succ(s_start)}(cost(s_start, s')+
   |  |  g(s'))
41 |  |  Move to s_start
42 |  |  Scan for cell changes in
   |  |  environment (e.g. sensor
   |  |  ranges)
43 |  |  if Cell changes detected then
44 |  |  |  k_m = k_m + h(s_last, s_start)
45 |  |  |  s_last = s_start
46 |  |  |  forall s ∈ CHANGES do
47 |  |  |  |  Update cell s state
48 |  |  |  |  forall
   |  |  |  |  s' ∈ Pred(s) ∪ {s} do
49 |  |  |  |  |  UpdateVertex(s')
50 |  |  |  |  end
51 |  |  |  end
52 |  |  |  ComputePath()
53 |  |  end
54 |  end
```

Figure 9. Pseudo code of D* Lite algorithm.

etc.) to detect environment changes in further area near the goal. Intuitively, we can imagine a valley where $g(s)$ of each cell substitutes for its height; the goal cell has the lowest height (the bottom of the valley), and robot position (start cell) is always at valley's edge. Suddenly, there is a change in height near the goal; D* Lite has to give enormous effort to correct the continuity of the valley slope from the bottom to the surface. Because of the overhead of storing

g-value information, the correction effort now is more expensive than starting the search from scratch. This is a big limitation for multi-sensor-based robot system; the problem also makes D* Lite unreliable in high-dimensional state space.

This behaviour leads us to a problem statement: The location of environment changes with respect to goal position makes an enormous difference to efficiency of D* Lite. This problem is also addressed by the author of D* Lite in [12] as open question. In other papers, mathematical approach is used to study this pitfall in [16]. Unfortunately, the problem is still not solved thoroughly; however, there are approaches to partly overcome this pitfall in certain situation that will be presented in the following section.

4.2. Performance improvements

This section describes variants of D* Lite that partially solves the mentioned pitfall of D* Lite. Hence, these state-of-the-art algorithms improve the computation factor of D* Lite.

4.2.1. Anytime dynamic A (AD*) algorithm*

As one of the prominent properties of D* Lite, it maintains the optimality of solution paths. However, in real-world application, optimal paths are difficult to calculate due to the complexity and uncertainty of environment within available time; the paths are also quickly to become out of date because of dynamic surroundings. Moreover, the state space, which encodes global motion constraints of the robot, tends to be high dimensions. With these difficulties, D* Lite becomes unreliable when implementing in real robot.

Anytime Dynamic A* (AD*) [12], which is a combination of D* Lite and ARA* algorithm, is a trade-off between computation and path optimality. It sacrifices the shortest path to quickly generate suboptimal solution to cope with imperfect information and dynamic environment. AD* inherits these properties from its parent algorithms:

- **Anytime:** AD* uses inflated heuristic function to increase the greedy factor that expands aggressively towards goal while still maintaining path suboptimal bounds ε. In addition, AD* also is capable to reuse information from previous search to improve its path. This property is inherited from ARA* to cope with complex planning scenario.
- **Incremental heuristic:** AD* has the ability to efficiently identify relevant cells that contribute to replan a new path when environment changes are detected. However, the heuristic function in this property is not inflated to guarantee the suboptimal bound in replanning process. This property is inherited from D* Lite to cope with dynamic environment.

In general, AD* executes a series of similar searches with decreasing suboptimal bounds to generate a series of paths with improved bounds. As environment changes are detected, the locally inconsistent cells are placed in OPEN list with uninflated heuristic keys, and AD* processes these cells to correct the outdated path. The pseudo code for AD* is shown in **Figure 10**.

Algorithm 4: AD*

```
1  Function Key(s):
2      if g(s) > rhs(s) then
3          return [rhs(s)+ε*h(s_start, s); rhs(s)]
4      else
5          return [g(s) + h(s_start, s); g(s)]
6  Function UpdateVertex(s):
7      if s ≠ s_goal then
8          rhs(s) = min_{s'∈Succ(s)}(cost(s, s') + g(s'))
9      end
10     if s ∈ OPEN then
11         OPEN.remove(s)
12     end
13     if g(s) ≠ rhs(s) then
14         if s ∉ CLOSED then
15             OPEN.insert(s, Key(s))
16         else
17             INCONS.insert(s)
18     end
19 Function ComputeOrImprovePath():
20     while OPEN.TopKey() < Key(s_start) OR rhs(s_start) ≠ g(s_start) do
21         s = OPEN.Pop()
22         if g(s) > rhs(s) then
23             g(s) = rhs(s)
24             CLOSED.insert(s)
25             forall s' ∈ Pred(s) do
26                 UpdateVertex(s')
27             end
28         else
29         g(s) = ∞
30         forall s' ∈ Pred(s) ∪ {s} do
31             UpdateVertex(s')
32         end
33     end
34 Function Main():
35     forall s ∈ S do
36         rhs(s) = g(s) = ∞
37     end
38     OPEN = CLOSED = INCONS = ∅
39     rhs(s_goal) = 0
40     OPEN.insert(s_goal, Key(s_goal))
41     ComputeOrImprovePath()
42     Choose initial sub-optimal bound ε and publish ε
43     while True do
44         Scan for cell changes in environment (e.g. sensor ranges)
45         if Cell changes detected then
46             forall s ∈ CHANGES do
47                 Update cell s state
48                 forall s' ∈ Pred(s) ∪ {s} do
49                     UpdateVertex(s')
50                 end
51             end
52         end
53         if Significant cell changes detected then
54             Increase ε or replan from scratch
55         else if ε > 1 then
56             Descrease ε
57         Move nodes in INCONS to OPEN and set CLOSED = ∅
58         Update all priority values in OPEN according to Key(s)
59         ComputeOrImprovePath()
60         Publish ε
61     end
```

Figure 10. Pseudo code of AD* algorithm.

At first, we set the suboptimal bound ε to be large enough in order to generate solution quickly (**Figure 10**, line 42). Unless environment changes are detected, AD* iteratively decrease suboptimal bound ε to improve the solution as time allowed (**Figure 10**, lines 55–60); this phase is exactly the same with ARA*.

When changes are perceived, the suboptimal bound of solution is no longer guaranteed, especially the under-consistent cells, due to the fact that there could have shorter paths exist. Therefore, these under-consistent cells are placed in OPEN list with uninflated heuristic to ensure these cells propagate their inconsistences to neighbours first when ComputeOrImprovePath() function is called (**Figure 10**, lines 5 and 45–50). However, because of this effect, many under-consistent cells quickly rise to the top of OPEN list; they usually do not contribute to calculate new path in practice (e.g. objects cause under-consistent changes like human movement, etc.). Hence, AD* tends to slow down when the path is near optimal. Theorems of AD* are described in detail in [12].

When "significant cell changes are detected" (**Figure 10**, line 53), there is a high chance that the problem of D* Lite occurs and the search tree may be corrupted heavily; the replanning process now is expensive; we can increase suboptimal bound ε to speed up the correction effort or start a new search from scratch. The problem is how to estimate "significant cell changes". This algorithm does not solve the mentioned problem of D* Lite completely; it is just a trade-off between performance and path optimality. However, AD* performs quickly in large-scaled map compared to other algorithms, in which the robot has significant time to iteratively repair its path, and thus overall path is near optimal.

4.2.2. D Lite with reset algorithm*

Unlike AD*, D* Lite with Reset (D*LR) [16] partially solves the problem D* Lite while still maintaining path optimality. The idea of D*LR is simple; it decides flushing previous search data and starts searching from scratch when the replanning process is expensive.

D*LR is a variant of D* Lite; it inherits all the properties of D* Lite. The main contribution of D*LR is that it proposes two criteria to decide whether to incrementally replan path or calculate a fresh path using A* at the position the robot detects changes. Let total traversed cell is N_T; total cell of path that exists between consecutive detection incidents is N_P, and the remaining path count is $N_R = N_P - N_T$; the criteria are:

- **Ratio of traversed length:** The criteria measure how many percentages of the path the robot has moved between two consecutive positions that the robot detects environment changes. The ratio $\frac{N_T}{N_P}$ is then compared with a threshold:

$$\frac{N_T}{N_P} < \propto . \tag{8}$$

If the ratio is greater than **threshold** $\propto$ when the robot perceives changes, it triggers the reset routine and starts searching from scratch. Intuitively, the position that robot detects changes is nearer the goal, the more likely replanning process is expensive.

- **Linear heuristic distance:** The criteria measure the complexity of the remaining path count N_R between consecutive detection incidents. The method to measure the complexity is to use inflated heuristic function:

$$N_R > \varepsilon * h(s, \ s_{goal}), \tag{9}$$

where s is the current position of the robot. If $N_R \leq \varepsilon * h(s, \ s_{goal})$, then the remaining path is simple enough; there is a chance that the new path is much more complex. Hence, the algorithm must plan over from scratch.

As can be seen, these criteria use only path information between consecutive detection incidents in order to estimate the amount of computation of replanning process comparing with planning over from scratch. The reason is that it is hard to predict propagation behaviour of OPEN list, because the state space is only partially known. Moreover, these criteria only work in high cluttered and complex environment, where environment changes usually block initial path and the new path is likely to be much longer than initial path. The pseudo code of D*LR is presented in **Figure 11**.

The proposed criteria of D*LR are not robust due to extensively replying on environmental assumptions. However, the algorithm can be improved if criteria that can robustly estimate computation of replanning process in any kind of environment are applied. If criteria are robust, D*LR performance of each iteration is bounded by the complexity of A*:

$$O(|V|) = O(|E|) = O(b^d) \tag{10}$$

4.3. Optimality improvements: Field D* algorithm

Although cell decomposition approximation is widely used to discretize C-space for search-based algorithms due to its robustness (no prior environmental assumptions), this approximation intrinsically prevents search-based algorithm to produce optimal path. The search-based algorithms just allow to transition between cell centres, thus restricting robot traverse directions to increment of $\frac{\pi}{4}$. Moreover, the produced path involves many sharp turns and jerky segments in large map that makes robot difficult to move.

There were many approaches to cope with this problem. For instance, post-processing method that finds the furthest point P along the solution path for which a straight line path from P to robot position is collision-free and replaces the original path to P with this straight line. However, this method sometimes does not work and increases the path cost. Another approach is fast marching method [22]; this method incorporates interpolation step in planning step to produce low-cost interpolated path. Nonetheless, this method assumes that transition cost between grid cells is constant and does not have heuristic property like A*; hence it is not applicable to outdoor environment, which requires fast path generating and non-uniform cost grid.

To incorporate incremental heuristic property of D* Lite, the authors of Field D* [3] embed linear interpolation method to the replanning process to generate "any-angle" optimal path

Algorithm 5: D* Lite with Reset

```
1  Function Initialize():
2      forall s ∈ S do
3          rhs(s) = g(s) = ∞
4      end
5      OPEN = ∅
6      rhs(s_goal) = 0; k_m = 0
7      OPEN.insert(s_goal, Key(s_goal))
8  Function Main():
9      s_last = s_start
10     Initialize()
11     ComputePath()
12     Count length of path N_P and N_T = 0
13     while s_start ≠ s_goal do
14         s_start = argmin_{s'∈Succ(s_start)}(cost(s_start, s') + g(s'))
15         Move to s_start and N_T = N_T + 1
16         Scan for cell changes in environment (e.g. sensor ranges)
17         if Cell changes detected then
18             if Reset Criteria is satisfied then
19                 Initialize()
20             else
21                 k_m = k_m + h(s_last, s_start)
22                 forall s ∈ CHANGES do
23                     Update cell s state
24                     forall s' ∈ Pred(s) ∪ {s} do
25                         UpdateVertex(s')
26                     end
27                 end
28             s_last = s_start
29             ComputePath()
30             Count length of path N_P and N_T = 0
31         end
32     end
```

Figure 11. Pseudo code of D*LR. Other functions such as Key(), UpdateVertex() and ComputePath() are the same with D* Lite and thus are not presented.

that overcomes grid limitation in dynamic environment. The root cause of restriction of path optimality is the rule to transition between cell centres; the idea of Field D* to solve this problem is to remap state space graph vertices to the corner of each cell (see **Figure 12**). The nodes s can be considered as sample points of continuous cost field, where the optimal path must pass one of the edges $\{\overrightarrow{s_1 s_2}, \overrightarrow{s_2 s_3}, \overrightarrow{s_3 s_4}, \overrightarrow{s_4 s_5}, \overrightarrow{s_5 s_6}, \overrightarrow{s_6 s_7}, \overrightarrow{s_7 s_8}, \overrightarrow{s_8 s_1}\}$ that connects consecutive neighbours of s; the edge is $\overrightarrow{s_1 s_2}$ in the picture's case.

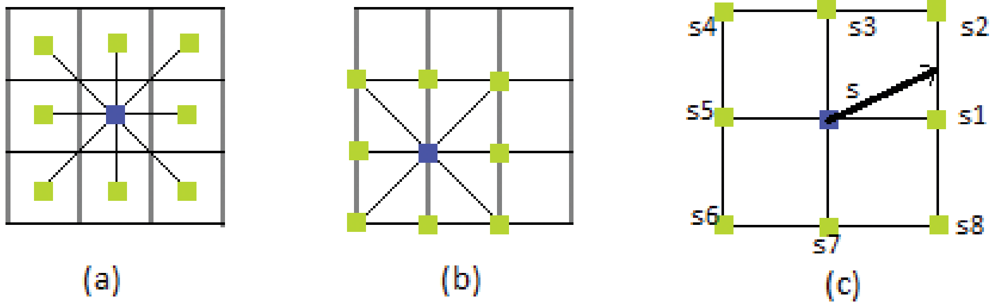

Figure 12. Remapping state space graph vertices from cell centres to cell corners (a) Center Vertices, (b) Corner Vertices and (c) Optimal Path intersected $\longrightarrow s_1\, s_2$.

In this case, the edge, which resides on the boundary of two cells, has the edge cost equal to the minimum of cost of the two cells. Field D* use linear interpolation to compute approximately the cost of any point s_y on the edge $\overrightarrow{s_1 s_2}$ by using the path cost (cost from the node to goal) $g(s_1)$ and $g(s_2)$:

$$g(s_y) = yg(s_2) + (1 - y)g(s_1), \tag{11}$$

where y is the distance from s_1 to s_y (**Figure 13**). Given the centre cell cost c and bottom cell cost b, we can compute the path cost of s using edge $\overrightarrow{s_1 s_2}$ as

$$g(s) = \min_{x,y} \left(bx + c\sqrt{(1-x)^2 + y^2} + g(s_2)y + g(s_1)(1 - y)\right) \tag{12}$$

where x is the distance travel along the bottom edge from s before cutting through the centre cell to reach the right edge at the point s_y a distance y from s_1. (see **Figure 13**).

The interpretation from formulas (4) into ComputeCost() function is described in detail in [3]. This optimization approach can be plugged in any dynamic planner by replacing standard cost function between cell centres by function ComputeCost(). In addition, due to remapping

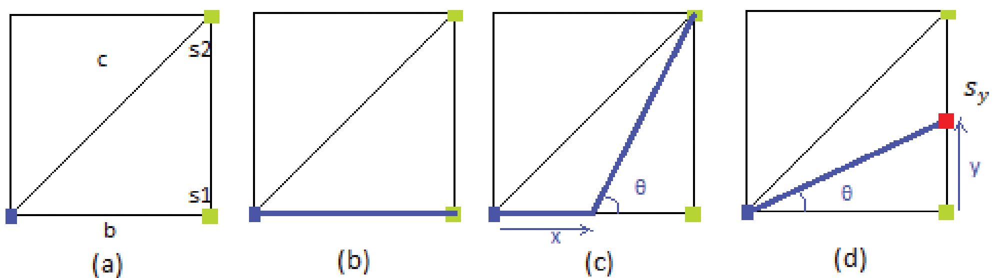

Figure 13. Linear interpolation process to compute path cost of s using edge $\longrightarrow s_1\, s_2$. The subfigures illustrate possible optimal path cost (a) Calculate g(s) based on s_1, s_2 (b) Case g(s_1) < g(s_2) (c) Case g(s_1) > g(s_2) and (d) Case path costs that pass arbitrary point.

graph vertices into cell corners, we also need to change finding cell centre neighbours to a pair of corner nodes as illustrated in **Figure 13**. Once the path costs of necessary nodes are computed, the path is generated by starting from the initial node and iteratively finds, using linear interpolation, the optimal node on the neighbor cell boundary to move next. The pseudo code of Field D* and modifications in red colour are shown in **Figure 14**. Note that the differences

Algorithm 6: Field D*

1 **Function** *ComputeCost()*:
2   **if** *s_a is diagonal neighbor of s* **then**
3     $s_1 = s_b$; $s_2 = s_a$
4   **else**
5     $s_1 = s_a$; $s_2 = s_b$
6   Set c is traversal cost of cell with corners s, s_1, s_2
7   Set b is traversal cost of cell with corners s, s_1 but not s_2
8   **if** $min(c, b) = \infty$ **then**
9     $v_s = \infty$
10   **else if** $g(s_1) \leq g(s_2)$ **then**
11     $v_s = min(c, b) + g(s_1)$
12   **else**
13     $f = g(s_1) - g(s_2)$
14   **if** $f \leq b$ **then**
15   **if** $c \leq f$ **then**
16         $v_s = c\sqrt{2} + g(s_2)$
17   **else**
18         $y = min(\frac{f}{\sqrt{c^2 - f^2}}, 1)$
19         $v_s = c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$
20   **else**
21   **if** $c \leq b$ **then**
22         $v_s = c\sqrt{2} + g(s_2)$
23   **else**
24         $x = 1 - min(\frac{b}{\sqrt{c^2 - b^2}}, 1)$
25         $v_s = c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$
26   **return** v_s
27 **Function** *UpdateVertex(s)*:
28   **if** $s \neq s_{goal}$ **then**
29     $rhs(s) = min_{(s', s'') \in connbrs(s)}(ComputeCost(s, s', s''))$
30   **end**
31   **if** $s \in OPEN$ **then**
32     $OPEN.remove(s)$
33   **end**
34   **if** $g(s) \neq rhs(s)$ **then**
35     $OPEN.insert(s, Key(s))$
36   **end**
37 **Function** *Main()*:
38   **forall** $s \in S$ **do**
39     $rhs(s) = g(s) = \infty$
40   **end**
41   $s_{last} = s_{start}$
42   $OPEN = \emptyset$
43   $rhs(s_{goal}) = 0$; $k_m = 0$
44   $OPEN.insert(s_{goal}, Key(s_{goal}))$
45   ComputePath()
46   **while** $s_{start} \neq s_{goal}$ **do**
47   Move s_{start} along interpolated path.
48   Scan for cell changes in environment (e.g. sensor ranges)
49   **if** *Cell changes x detected* **then**
50       $k_m = k_m + h(s_{last}, s_{start})$
51       $s_{last} = s_{start}$
52   **forall** $x \in CHANGES$ **do**
53   Update cell x state
54   **forall** *Node s' on a corner of x* **do**
55   UpdateVertex(s')
56   **end**
57   **end**
58   ComputePath()
59   **end**
60   **end**

Figure 14. Pseudo code of Field D*.

between D* Lite and Field D* are highlighted in red. The function Key(), ComputePath() are the same as D* Lite and thus is not presented. This pseudo code is a basic version of Field D*; optimised versions are presented in [3].

Field D* inherits all properties of D* Lite; it combines linear interpolation method to compute path from any point inside cell, not just corners or cell edges. This feature is crucial for robot to get back on track if the actuator execution is faulty. Moreover, Field D* is not subjected to direction restriction; hence, it produces much shorter and smoother path.

5. Experimentation

In this section, using our path planning framework, we demonstrate the evaluation comparison between algorithms in search-based family in terms of performance and path optimality.

5.1. Evaluation method

To visualise the evolution in computation of search-based algorithms, we compare the replanning computation of D* Lite, Anytime Dynamic A* and D* Lite with Reset. The purpose of the comparison is to demonstrate the performance improvements of D* Lite variants in order to apply on robot that operates in complex and dynamic environment. However, since the planning time depends on the implementation and machine configuration, we therefore choose the amount of cell expansion in each replanning iteration of search-based algorithm to be standard performance measurement of the mentioned algorithms. This method is independent on machine specifics and actual implementation and therefore firmly accurately shows the enhancement of this evaluation. The path solution ratio between AD* with different ε suboptimal bound and optimal path of other algorithms is also measured to visualise the trade-off between optimality and computation.

The experiments are conducted on our 2D simulation engine. The state space is a 2D grid cell with uniform resolution [23]. The conceptual robot in this simulation has two-cell-unit range and its own known grid map to detect environment changes (unblocked cell to blocked cell and vice versa) as it moves along the initial path (see **Figure 15**).

5.2. Evaluation results

We evaluate the performance and path solution of search-based algorithms in two scenarios: partially known and unknown 2D grid environment with uniform resolution. The total expanded cells are averaged based on total replanning processes on each simulation instance, with 95% confident. The path solution of each algorithm is counted as the total cells that the robot has traversed from corner to corner of the map. We decrease the suboptimal bound of AD* for 0.1 per step the robot travels until the suboptimal bound reaches 1.0 (optimal path).

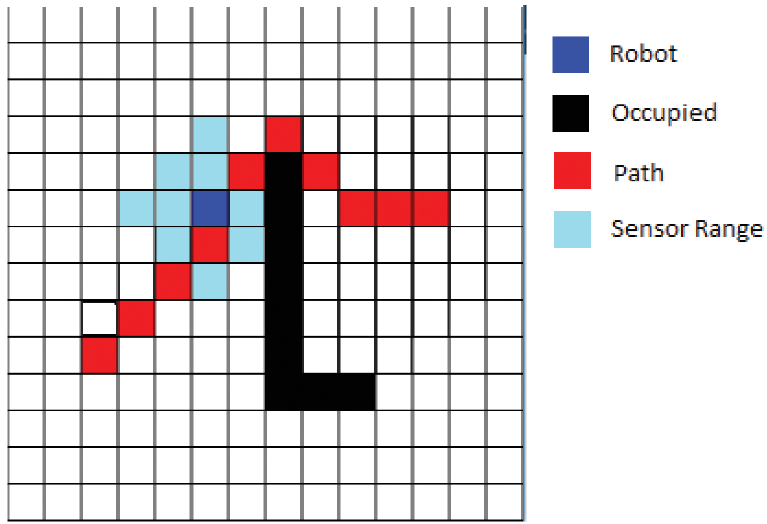

Figure 15. Simulated environment on our framework.

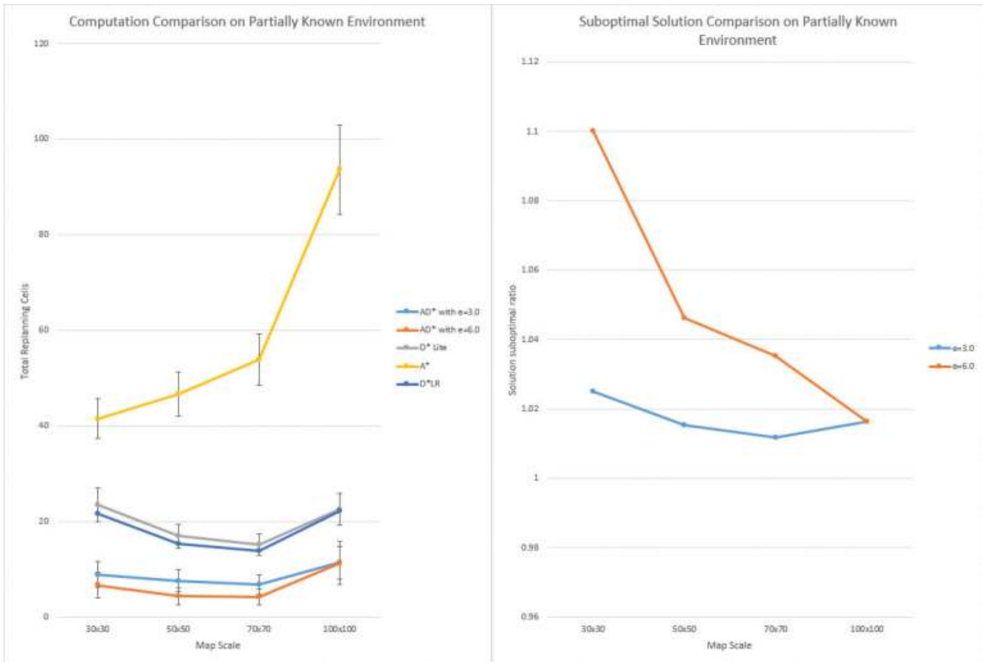

Figure 16. Comparison between search-based algorithms on partially known environment with increasing map scale in terms of computation and path solution.

Figure 16 shows the speedup result throughout the evolution from Replanning A* to AD* with different ε suboptimal bounds as well as the trade-off of AD*. The environment is initially generated randomly obstacles that occupy 25% of the map. The initial map is then input to the robot. While the robot is moving, we randomly change the cell states that are 15% of the map, thus forcing the robot to replan its path whenever it detects environmental changes.

As can be seen, AD* has the highest performance that has least total expanded cells in replanning process; the higher the suboptimal bound, the better the performance. The reason is AD* is inflated its heuristic function to make it greedier in expanding cells towards goal. It is interesting that path solution of AD* is not much longer than optimal path. As the scale of map is increasing, the path between map corner is longer to travel, and thus, the robot is given enough time to improve its solution (path ratio with $\varepsilon = 6.0$ is gradually converged to the one $\varepsilon = 3.0$ at 1.016).

D*LR slightly improves the performance of D* Lite; it is because D*LR relies on computation differences between Replanning A* and D* Lite. In fact, the pitfall of D* Lite rarely happens in scenarios that the robot detects changes near its position. Replanning A* does not have incremental property and thus uses the highest computation.

The data confirms the fact that AD*, in average throughout the increasing map scale, improves 125% and 194% performance compared to D* Lite with $\varepsilon = 3.0$ and $\varepsilon = 6.0$, respectively. The path produces by AD* only 1% longer than optimal path in average.

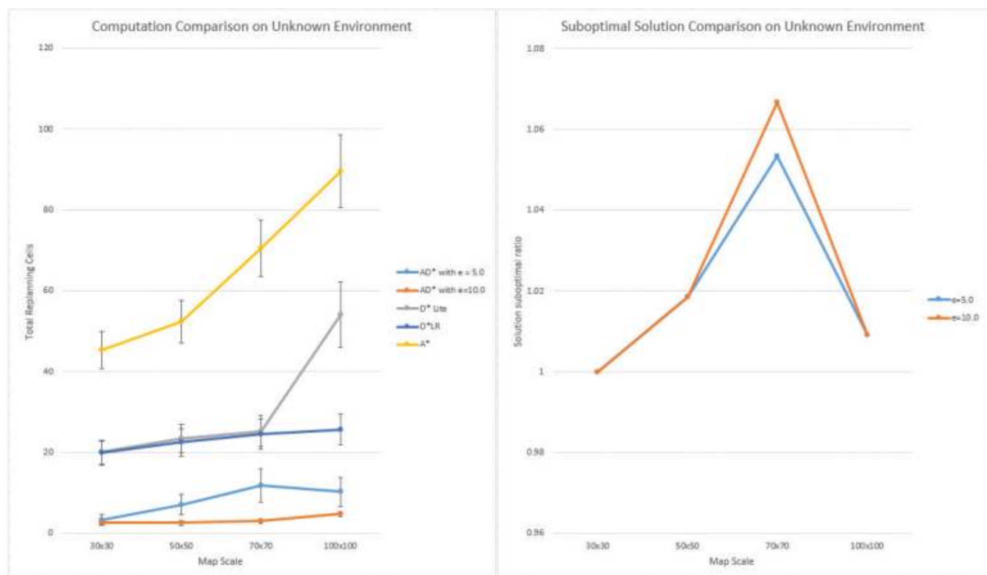

Figure 17. Comparison between search-based algorithms on unknown environment with increasing map scale in terms of computation and path solution.

Figure 17 describes the evaluation case on unknown environment. The environment is initially generated with random obstacles that occupy 15% of the map. The robot does not know the initial conditions; it will replan its path whenever it detects obstacles that do not exist in its map.

For unknown environment scenario, D*LR performs significantly better than D* Lite as increasing map scale. The reason is that if the replanned path is much longer than the initial path, which is the common case in unknown environment, the replanning process of D* Lite is also expensive. AD* still has the least computation compared to old search-based algorithm; it reduces drastically the computation of D* Lite with 845% better performance, in the case $\varepsilon = 10.0$, while still maintains good path solution.

6. Conclusion

In practice, motion planning algorithms can be implemented on top of navigation layer such as simultaneous localization and mapping (SLAM) for autonomous robot. While navigation layer enables the robot to perceive surrounding information and its position relative to the surroundings, motion planning layer gives the robot abilities to plan a path in surrounding environment and make decision to avoid obstacles. Because of that fact, navigation and motion planning are always paired up to enable autonomous robot to operate in dynamic and complex environment.

This chapter is a guide to comprehend the foundation of motion planning, in particular, search-based path planning algorithms. In this chapter, we present the steps to develop and formulate a motion planning problem. We also describe the evolution branches of motion planning and then focus on the development of search-based algorithm family. Each algorithm in search-based family is invented to cope with increasing demands in performance or solution quality, for the robot to operate in more complex scenarios. To reinforce the revolution statement of state-of-the-art search-based algorithms, we provide a computation and optimality comparison between search-based algorithms on partially known and unknown environment. Based on the data, we conclude that Anytime Dynamic A* is the most suitable algorithm that enables the robot to operate in cluttered and fast changing scenario.

Until recently, the mainstream of motion planning development is to enhance the performance of search-based algorithm and their solution optimality by modifying cell decomposition method. There are signals that the trajectory planning paradigm is starting to be active research field after being frozen for a decade. We expect that the future development of trajectory planning will robustly incorporate motion constraints with higher optimality and better computation. The ultimate goal of motion planning field is giving robot spatial decision planning converging to human ability.

Author details

An T. Le[1]* and Than D. Le[2]

*Address all correspondence to: eeit2015_an.lt@student.vgu.edu.vn

1 Department of Electrical Engineering and Information Technology, Vietnamese-German University, Hồ Chí Minh, Vietnam

2 Faculty of Engineering, Bristol Robotics Laboratory, Bristol University, Bristol, United Kingdom

References

[1] Lavalle SM, Kuffner JJ Jr. Rapidly-exploring random trees: Progress and prospects. In: Algorithmic and Computational Robotics: New Directions. 2000. pp. 293-308. DOI: 10.1.1.38.1387

[2] N. Preda, A. Manurung, O. Lambercy, R. Gassert and M. Bonfè. Motion planning for a multi-arm surgical robot using both sampling-based algorithms and motion primitives. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS); Hamburg. 2015. pp. 1422-1427. DOI: 10.1109/IROS.2015.7353554

[3] Ferguson D, Stentz A. Field D*: An interpolation-based path planner and replanner. Journal of Robotics Research. 2007:239-253. DOI: 10.1007/978-3-540-48113-3_22

[4] Stentz A. Optimal and efficient path planning for partially-known environments. In: Proceedings of the 1994 IEEE International Conference on Robotics and Automation; San Diego, CA. 1994. pp. 3310-3317. DOI: 10.1109/ROBOT.1994.351061

[5] Matthies L, Xiong Y, Hogg R, Zhu D, Rankin A, Kennedy B, Hebert M, Maclaehlan R, Won C, Frost T, Sukhatme G, Mchenry M, Goldberg S. A portable, autonomous, urban reconnaissance robot. In: Proceedings of the International Conference on Intelligent Autonomous Systems; 2000. DOI: 10.1.1.98.9353

[6] Doan KN, Le AT, Le TD, Peter N. Swarm robots' communication and cooperation in motion planning. In: Zhang D, Wei B, editors. Mechatronics and Robotics Engineering for Advanced and Intelligent Manufacturing. Cham: Springer; 2016. pp. 191-205. DOI: 10.1007/978-3-319-33581-0_15

[7] Hwang YK, Ahuja N. Gross motion planning—A survey. ACM Computing Survey. 1992;**24**(3):219-291. DOI: 10.1145/136035.136037

[8] Dechter R, Pearl J. Generalized best-first search strategies and the optimality of a*. Journal of the ACM (JACM). 1985;**32**(3):505-536. DOI: 10.1145/3828.3830

[9] Li J, Liu S, Zhang B, Zhao X. RRT-A* motion planning algorithm for non-holonomic mobile robot. In: 2014 Proceedings of the SICE Annual Conference (SICE); Sapporo. 2014. pp. 1833-1838. DOI: 10.1109/SICE.2014.6935304

[10] Arismendi C, Álvarez D, Garrido S, Moreno L. Nonholonomic motion planning using the fast marching square method. International Journal of Advanced Robotic Systems. 2015;**12**:5. DOI: 10.5772/60129

[11] Sun X, Yeoh W, Koenig S. Moving target D* Lite. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '10); Toronto, Canada. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems; p. 67-74. ISBN: 978-0-9826571-1-9

[12] Likhachev M, Ferguson D, Gordon G, Stentz A, Thrun S. Anytime dynamic A*: An anytime, replanning algorithm. In: Biundo S, Myers KL, Rajan K, editors. Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling (ICAPS'05); Monterey, California, USA. AAAI Press; 2005. p. 262-271. ISBN:1-57735-220-3

[13] Aine S, Likhachev M. Truncated incremental search. Journal of Artificial Intelligence. 2016;**234**(C):49-77. DOI: 10.1016/j.artint.2016.01.009

[14] Aine S, Likhachev M. Anytime truncated D*: Anytime replanning with truncation. In: Sixth Annual Symposium on Combinatorial Search; AAAI Publications. 2013

[15] Nash A, Koenig S, Likhachev M. Incremental Phi*: Incremental any-angle path planning on grids. In: Kitano H, editor. Proceedings of the 21st International Joint Conference on Artificial intelligence (IJCAI'09); 2009; San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2009. p. 1824-1830

[16] Le AT, Bui MQ, Le TD, Peter N. D* Lite with reset: Improved version of D* Lite for complex environment. In: First IEEE International Conference on Robotic Computing (IRC); Taichung, Taiwan. IEEE; 2017. pp. 160-163. DOI: 10.1109/IRC.2017.52

[17] Miao H, Tian YC. Robot path planning in dynamic environments using a simulated annealing based approach. In: 2008 10th International Conference on Control, Automation, Robotics and Vision; Hanoi. 2008. pp. 1253-1258. DOI: 10.1109/ICARCV.2008.4795701

[18] Likhachev M, Gordon G, Thrun S. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In: Proceedings of the 2003 Conference Advances in Neural Information Processing Systems 16 (NIPS-03); MIT Press; 2004. DOI: 10.1.1.3.9449

[19] Sven Koenig and Maxim Likhachev. D*lite. In: Rina Dechter, Michael Kearns, and Rich Sutton, editors. In Eighteenth National Conference on Artificial Intelligence; Edmonton, Alberta, Canada. Menlo Park, CA, USA: American Association for Artificial Intelligence; 2002. p. 476-483. ISBN:0-262-51129-0

[20] Sven Koenig, Maxim Likhachev, David Furcy. Lifelong Planning A∗. Artificial Intelligence. 2004;**155**(1):93-146. DOI: http://dx.doi.org/10.1016/j.artint.2003.12.001

[21] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms. 2000;**34**(2):251-281. DOI: http://dx.doi.org/10.1006/jagm.1999.1048

[22] Sethian JA. A fast marching level set method for monotonically advancing fronts. Proceedings of the National Academy of Sciences. 1996;**93**(4):1591-1595

[23] Prof. Ron Alterovitz. Configuration Space Visualization of 2-D Robotic Manipulator [Internet]. Available from: https://www.cs.unc.edu/~jeffi/c-space/robot.xhtml [Accessed: 8/18/2017]