Grid-based path planning for a differential drive robot

Course: Autonomous and Mobile Robotics
Professor: Giuseppe Oriolo
Tutor: Giulio Turrisi

Authors:
Denise Landini – 1938388
Alessandro Lambertini – 1938390
Gianluca Lofrumento – 1956579

La Sapienza University – Roma
Academic Year: 2020-2021

SAPIENZA
UNIVERSITÀ DI ROMA

Summary

ASSIGNMENT

In the literature, different approaches exist for computing a path between two points in a known map, such as via sampling, nonlinear optimization, grid-based search, etc. The latter is generally considered to be a viable solution when the dynamic constraint is not crucial for the obtained path, as in the case of a differential drive robot, and exhibits completeness, fast convergence, and good replanning capability in the case of non-static environments. The aim of this project is to test and compare two different grid-based methods, namely D* (Stentz, 1994) and D* Lite (Koenig et al., 2005), implementing all the simulations in Matlab and CoppeliaSim.

INTRODUCTION

In this work, we study and compare different algorithms that use a grid-based methods with a differential drive robot: these are D*, D* Lite (2 versions) and Field D*.

We consider the fact that mobile robots can navigate in environments that are *partially known*. So, they plan a trajectory from the starting position to the goal position, but they do not know all the terrain in which they drive, and they do not know all the obstacles that they will meet. For this reason, it is important that our mobile robots can quickly replan locally the trajectory according to the environment changes.

The graph used to represent the partially unknow (or totally unknown) terrain that we consider is an eight-connected grid, meaning that we consider that one cell has eight adjacency cells: one up, one down, one right, one left and four diagonals (**Figure 1**).

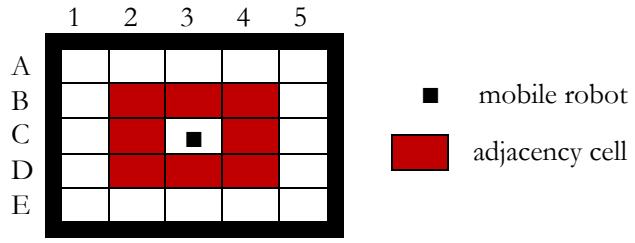


Figure 1: *Adjacency cells*

In this graph, edge costs are one but, when the mobile robots find the non-traversability of an edge, it becomes infinity. The non-traversability of a cell often is due to the presence of obstacles in the environment – like a wall, a person, etc… – and it necessarily implies a change in the scheduled path.

D*

D* is a path planning algorithm, variation of A*, which can generate optimal trajectories. Arc cost parameters can change during the process, making it dynamic with respect to A*.

The problem space can be defined as a set of states, defining a position, connected with each other by directional arcs, each of which associated with a cost. Like A*, D* maintains a list of open states, used to spread the information about the arc cost changes and to compute the path costs. A state is composed of:

- Two coordinates, marking the position on the grid.
- A backpointer to another state, except for the goal, thus from the start state it is formed a chain representing the actual path.
- A tag, which can assume the values NEW, $OPEN$ or $CLOSED$, respectively if the state has never been on, currently is on or has left the list of open states.
- A value h, computed by the heuristic function $h(Goal, State)$. The heuristic function is chosen as the straight-line cost of the path from the current position of the robot to the goal assuming the cells in the path $EMPTY$.
- A key value k, computed by the key function $k(Goal, State)$.

Initially, all the states have tag to NEW and h and k to zero, then the goal state is added to the open list and the process-state function is executed until the goal is found – or there are no states to expand anymore, in this case a failure will be returned – then the robot starts to execute the path, by traversing the backpointer of each state, and checks, at each step, if the following crossing cell is an obstacle: in case it is, it uses the modify-cost function to update the cost and compute an alternative path to the goal like described before.

So, there are two main functions: **process-state (Figure 2)**, which computes optimal path costs, and **modify-cost (Figure 3)**, which updates the arc cost function.

Process-state takes no parameters and consists of:

1. Finding the lowest k value state X (so the optimal one).
2. If X is a $RAISE$ – meaning if $k(X) < h(X)$ – its path cost may not be optimal so, before propagating the cost changes to its neighbors, it is examined if h can be reduced.
3. Cost changes are propagated to its neighbors:
   a. If X is a $LOWER$ – meaning if $k(X) = h(X)$, its path cost is optimal, and its neighbors are analyzed to find the correct successors.
   b. Otherwise cost changes are propagated to NEW states and in the same way as for $LOWER$ states. However, if X can lower the path cost of a state that is not its immediate successor, it is deeply analyzed – so put in the open list – else if the path cost of X can be reduced by a suboptimal neighbor, the latter is further analyzed.

```
Function: PROCESS-STATE ()
L1   X = MIN-STATE( )
L2   if X = NULL then return -1
L3   k_old = GET-KMIN( ); DELETE(X)
L4   if k_old < h(X) then
L5     for each neighbor Y of X:
L6       if h(Y) ≤ k_old and h(X) > h(Y) + c(Y, X) then
L7         b(X) = Y; h(X) = h(Y) + c(Y, X)
L8   if k_old = h(X) then
L9     for each neighbor Y of X:
L10      if t(Y) = NEW or
L11        (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) or
L12        (b(Y) ≠ X and h(Y) > h(X) + c(X, Y)) then
L13        b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L14  else
L15    for each neighbor Y of X:
L16      if t(Y) = NEW or
L17        (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) then
L18        b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L19      else
L20        if b(Y) ≠ X and h(Y) > h(X) + c(X, Y) then
L21          INSERT(X, h(X))
L22        else
L23          if b(Y) ≠ X and h(X) > h(Y) + c(Y, X) and
L24            t(Y) = CLOSED and h(Y) > k_old then
L25            INSERT(Y, h(Y))
L26  return GET-KMIN( )
```

Figure 2: *Process-state function*

Modify-cost function takes two states and the new cost as input, then:

```
Function: MODIFY-COST (X, Y, cval)
L1   c(X, Y) = cval
L2   if t(X) = CLOSED then INSERT(X, h(X))
L3   return GET-KMIN( )
```

Figure 3: *Modify-cost function*

1. Update the arc cost between the two states as the new cost passed.
2. If the first state is $CLOSED$, it is inserted in the open list to further analysis.

The role of $RAISE$ and $LOWER$ is fundamental since they respectively propagate cost increases and cost reductions. For example, when the cost of an arc is increased, the neighbor affected state is placed on the open list and the cost increase is propagated through $RAISE$ states; these states meet neighbors of lower cost, so these $LOWER$ states are placed on the open list and they eventually decrease the cost of the sequences, if possible.

D* properties

- Soundness: once a state has been visited – so its tag is not NEW – a finite sequence to the goal has been constructed.
- Optimality: if the value returned by process-state equals or exceeds $h(Goal, State)$, then $h(Goal, State) = minimumCost(Goal, State)$ – so the optimal one.
- Completeness: if a path from start to goal exists, and the search space has a finite number of states, the path will be constructed in a finite amount of time. If it does not exist, it will be reported in a finite amount of time too.

D* LITE: v1

D* Lite is an extension of Lifelong Planning A* (LPA*). One of the main features of D* lite is that it can replan in short time the new path when a cost changes.

Since when the mobile robot moves, heuristics change because it is computed with respect to the current vertex of the mobile robot. Moreover, keys of the vertices need to be recomputed in the priority queue. This is one of the important differences that we have between D* Lite and the other classical algorithms. So, this algorithm works as we can see in **Figure 4**.

We set the start and the goal position the mobile robot must reach, then we initialize the ***g_value*** and the ***rhs_value*** of all the vertices and we insert the goal in the priority queue.

g_value and ***rhs_value*** are estimate of the start distance of each vertex. In particular, ***rhs_value*** (righ-hand value) is the minimum among the predecessor and its value is based on the ***g_value*** and the cost between two vertices.

Now, we are ready to compute the shortest path from the current position to the goal. We do one step toward the goal, following the shortest path, and we update the current configuration of the mobile robot. Since we have done one step, we rescan the cells to know if they are traversable or not and we update the edge costs.

The last step is to update the keys of all vertices in the priority queue and recompute the shortest path.
This is an iterative algorithm because we iterate this phase until the mobile robots reach the goal position.

procedure CalcKey(s)
{01'} return $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))]$;
procedure Initialize()
{02'} $U = \emptyset$;
{03'} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04'} $rhs(s_{goal}) = 0$;
{05'} U.Insert$(s_{goal}$, CalcKey$(s_{goal}))$;
procedure UpdateVertex(u)
{06'} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s'))$;
{07'} if $(u \in U)$ U.Remove(u);
{08'} if $(g(u) \neq rhs(u))$ U.Insert$(u$, CalcKey$(u))$;
procedure ComputeShortestPath()
{09'} while (U.TopKey()<CalcKey(s_{start}) OR $rhs(s_{start}) \neq g(s_{start}))$
{10'} u = U.Pop();
{11'} if $(g(u) > rhs(u))$
{12'} $g(u) = rhs(u)$;
{13'} for all $s \in$ Pred(u) UpdateVertex(s);
{14'} else
{15'} $g(u) = \infty$;
{16'} for all $s \in$ Pred$(u) \cup \{u\}$ UpdateVertex(s);
procedure Main()
{17'} Initialize();
{18'} ComputeShortestPath();
{19'} while $(s_{start} \neq s_{goal})$
{20'} /* if $(g(s_{start}) = \infty)$ then there is no known path */
{21'} $s_{start} = \arg\min_{s' \in \text{Succ}(s_{start})}(c(s_{start}, s') + g(s'))$;
{22'} Move to s_{start};
{23'} Scan graph for changed edge costs;
{24'} if any edge costs changed
{25'} for all directed edges (u, v) with changed edge costs
{26'} Update the edge cost $c(u, v)$;
{27'} UpdateVertex(u);
{28'} for all $s \in U$
{29'} U.Update$(s$, CalcKey$(s))$;
{30'} ComputeShortestPath();

***Figure 4**: D* Lite v1 algorithm*

Advantages and disadvantages of D* Lite v1

We have some ***advantages*** with respect to D* because D* Lite is simpler to implement and to understand., indeed it is based on Lifelong Planning A* (LPA*) method. Furthermore, D* Lite is shorter than D* because it uses only tie-braking criterion when comparing priorities, and this simplify the code because we do not need a lot of nested conditional instruction, so it simplifies the analysis of the program flow.
Despite these simplifications, D* Lite has at least the same efficiency of D*, if not better.
The ***disadvantage*** is that it has always to reorder the priority queue.

D* LITE: v2

The main features of the second version of D* Lite is that it uses the search method of D* and this is better with respect to those used in D* Lite v1 because it is not necessary to reorder the priority queue when the mobile robot moves: vertices remain in order in the priority queue. The algorithm works as we can see in **Figure 5**.

Differences with the first version are:

- when new keys are computed, in the first component we add K_m that at start it is initialized to zero.
- if some edge costs changed while the robot is moving, then the K_m constant need to be updated.
  It is updated by adding to the current value of K_m the result of the heuristic computation considering the starting point and S_{last}. S_{last} is used to update K_m and it is the position in which there was the last change in the environment.

```
procedure CalcKey(s)
{01"} return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];
procedure Initialize()
{02"} U = ∅;
{03"} k_m = 0;
{04"} for all s ∈ S rhs(s) = g(s) = ∞;
{05"} rhs(s_goal) = 0;
{06"} U.Insert(s_goal, CalcKey(s_goal));
procedure UpdateVertex(u)
{07"} if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
{08"} if (u ∈ U) U.Remove(u);
{09"} if (g(u) ≠ rhs(u)) U.Insert(u, CalcKey(u));
procedure ComputeShortestPath()
{10"} while (U.TopKey()<CalcKey(s_start) OR rhs(s_start) ≠ g(s_start))
{11"}   k_old = U.TopKey();
{12"}   u = U.Pop();
{13"}   if (k_old<CalcKey(u))
{14"}     U.Insert(u, CalcKey(u));
{15"}   else if (g(u) > rhs(u))
{16"}     g(u) = rhs(u);
{17"}     for all s ∈ Pred(u) UpdateVertex(s);
{18"}   else
{19"}     g(u) = ∞;
{20"}     for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{21"} s_last = s_start;
{22"} Initialize();
{23"} ComputeShortestPath();
{24"} while (s_start ≠ s_goal)
{25"}   /* if (g(s_start) = ∞) then there is no known path */
{26"}   s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
{27"}   Move to s_start;
{28"}   Scan graph for changed edge costs;
{29"}   if any edge costs changed
{30"}     k_m = k_m + h(s_last, s_start);
{31"}     s_last = s_start;
{32"}     for all directed edges (u, v) with changed edge costs
{33"}       Update the edge cost c(u, v);
{34"}       UpdateVertex(u);
{35"}     ComputeShortestPath();
```

Figure 5: *D* Lite v2 algorithm*

- Compute shortest path function modified it by since the keys are always the lower bounds with respect of the keys that we have computed using D* Lite v1.
  1. Remove from the queue the vertex that has the minimum value and we called it K_{old}.
  2. Use $CalcKey()$ function to compute the key value that it should have.
     - If $K_{old} < CalcKey(u)$ then we insert the removed vertex with the key computed by using the $CalcKey()$ function in the priority queue.
     - Otherwise, we do the same operation that we do in the D* Lite v1 for the vertex u.

D* LITE: v2 (optimized version)

The **optimization** is done to avoid expansion of the vertices that have equal keys of the goal state. In fact, this number of vertices can be very large.

In particular, we do the following **changes**:

- In the old $UpdateVertex()$, it can happen that a vertex is removed and then reinserted (line {08", 09"}); instead, in this optimized version, we have a new $UpdateVertex()$ in which we update the key of the vertex and we change only the position, without removing it, from the priority queue. {07'"}.

- In row {17"} of the $UpdateVertex()$, when we compute the rhs_value, we can only compute the rhs_value as the minimum of its old_rhs_value and the sum of the cost of moving, from the locally overconsistent vertex, to the successor and the new g_value of the locally overconsistent vertex {20'"}. It is possible to do this because, since the g_value has decreased, the consequence is that only the rhs_value of the successors can decrease.

- In row {20"} of $UpdateVertex()$ we compute, for all the predecessors, the rhs_value. But the rhs_value, in this case, can be only affected by the old_rhs_value that is computed by using the old_g_value. With this reasoning we must recompute the rhs_value only in case the rhs_value is equal to the sum of the cost of moving from the locally overconsistent vertex to the successor and the old_g_value {26'", 27'"}.

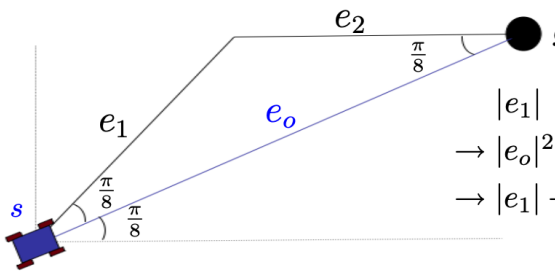
procedure CalcKey(s)
{01'"} return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$;
procedure Initialize()
{02'"} $U = \emptyset$;
{03'"} $k_m = 0$;
{04'"} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05'"} $rhs(s_{goal}) = 0$;
{06'"} U.Insert$(s_{goal}, [h(s_{start}, s_{goal}); 0])$;
procedure UpdateVertex(u)
{07'"} if $(g(u) \neq rhs(u)$ AND $u \in U)$ U.Update$(u, CalcKey(u))$;
{08'"} else if $(g(u) \neq rhs(u)$ AND $u \notin U)$ U.Insert$(u, CalcKey(u))$;
{09'"} else if $(g(u) = rhs(u)$ AND $u \in U)$ U.Remove(u);
procedure ComputeShortestPath()
{10'"} while (U.TopKey() $\dot{<}$ CalcKey(s_{start}) OR $rhs(s_{start}) > g(s_{start})$)
{11'"} u = U.Top();
{12'"} k_{old} = U.TopKey();
{13'"} k_{new} = CalcKey(u);
{14'"} if$(k_{old} \dot{<} k_{new})$
{15'"} U.Update(u, k_{new});
{16'"} else if $(g(u) > rhs(u))$
{17'"} $g(u) = rhs(u)$;
{18'"} U.Remove(u);
{19'"} for all $s \in$ Pred(u)
{20'"} $rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
{21'"} UpdateVertex(s);
{22'"} else
{23'"} $g_{old} = g(u)$;
{24'"} $g(u) = \infty$;
{25'"} for all $s \in$ Pred$(u) \cup \{u\}$
{26'"} if $(rhs(s) = c(s, u) + g_{old})$
{27'"} if $(s \neq s_{goal})$ $rhs(s) = \min_{s' \in \text{Succ}(s)}(c(s, s') + g(s'))$;
{28'"} UpdateVertex(s);
procedure Main()
{29'"} $s_{last} = s_{start}$;
{30'"} Initialize();
{31'"} ComputeShortestPath();
{32'"} while $(s_{start} \neq s_{goal})$
{33'"} /* if $(rhs(s_{start}) = \infty)$ then there is no known path */
{34'"} $s_{start} = \arg\min_{s' \in \text{Succ}(s_{start})}(c(s_{start}, s') + g(s'))$;
{35'"} Move to s_{start};
{36'"} Scan graph for changed edge costs;
{37'"} if any edge costs changed
{38'"} $k_m = k_m + h(s_{last}, s_{start})$;
{39'"} $s_{last} = s_{start}$;
{40'"} for all directed edges (u, v) with changed edge costs
{41'"} $c_{old} = c(u, v)$;
{42'"} Update the edge cost $c(u, v)$;
{43'"} if $(c_{old} > c(u, v))$
{44'"} $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
{45'"} else if $(rhs(u) = c_{old} + g(v))$
{46'"} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s'))$;
{47'"} UpdateVertex(u);
{48'"} ComputeShortestPath();

***Figure 6**: D* Lite v2 algorithm (optimized version)*

- When there are some edge costs that change, for all these directed edges, we use a new computation of the cost. This is because the rhs_value must be computed in different way based on some conditions related to the old cost {40'", 46'"}.

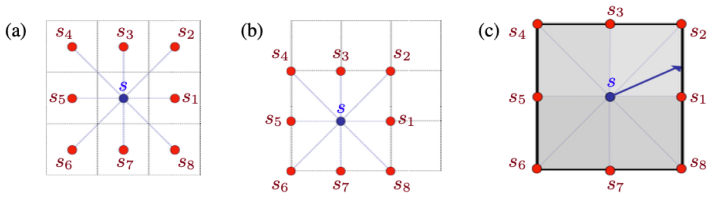
FIELD D*

Most of the grid-based path planners, like D* and D* lite, use a discrete set of transitions, constraining the robot heading to be a multiple of 45°. As result, these paths could be optimal in discrete environments, but suboptimal in continuous ones, especially when the goal is not at a multiple of 45° with respect to the agent.



Field D* is an interpolation-based planning and replanning algorithm with the aim of alleviating this problem. It is an extension of D* and D* lite and uses linear interpolation to produce globally smooth paths. Generally speaking, linear interpolation is not the best, but it produces good approximations with good performances.

With respect to classical planning techniques, environment representation (a) is changed moving the nodes from the center of the cell to the bottom left corner of it (b). Moreover, not only 45° movements are considered, but also intermediate ones. Optimal paths will likely intersect one of the eight segments defined by a couple of vertices (c).



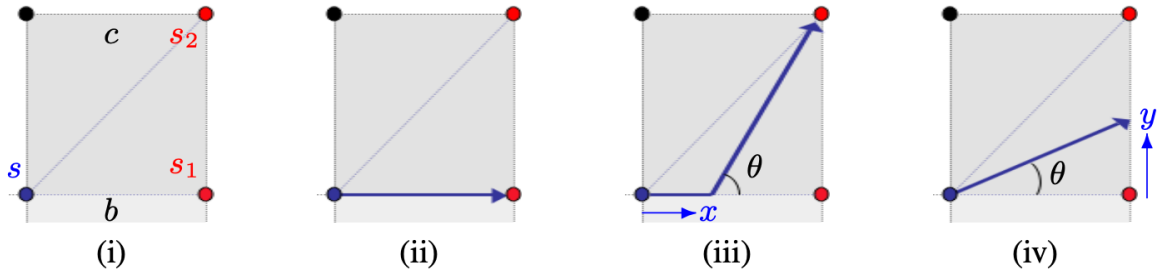
The key of this algorithm is how the path cost is computed. In classical approach, this value is:

$$g(s) = min_{s' \in nbrs(s)}(c(s, s') + g(s'))$$

So, the minimum sum between the cost to go from s to one of its neighbors and the cost of this neighbor. This calculation assumes the only transitions possible from node s are straight-line trajectories to one of its neighbors. However, relaxing these assumptions, from node s the possible trajectories are anywhere towards its boundaries. Hence, the cost of s given two of its neighbors s1 and s2 is:

$$min_{x,y}[bx + c\sqrt{(1-x)^2 + y^2} + g(s_2)y + g(s_1)(1-y)]$$

Where: b and c are respectively the cost from s to s1 and the cost from s to s2; x is the distance traveled along the bottom edge of s; y is the distance traveled on the edge between s1 and s2. If both x and y are zero, then the path taken is along the bottom edge but its cost is computed from the traversal cost (c) of the center cell.



So, we use the same function of D* Lite v1 for the keys, update of the state and computation of the shortest path; the only thing that change is the computation of the path cost. This latter thing slight modify the computation of the update state when our state does not coincide with the goal state.

Interpolation-based Cost Calculation

In details, by knowing the path costs of the neighbouring nodes, for each grid node we can compute the cost that we have from the node itself and the goal node.

The cost of an edge that is on the boundary of two grid cells is computed as the minimum of the costs between the traversal costs of each of the two cells.

During the computation we must consider also that the less expensive path depends on the size of the traversal costs, so we compute their difference and manage the different situation that we can have.

The *advantage* of Field D* with respect to the other analysed algorithms is that, by using **interpolation-based planning**, we have a smoother and low-cost path in real applications.

ComputeCost(s, s_a, s_b)
01. if (s_a is a diagonal neighbor of s)
02. $s_1 = s_b$; $s_2 = s_a$;
03. else
04. $s_1 = s_a$; $s_2 = s_b$;
05. c is traversal cost of cell with corners s, s_1, s_2;
06. b is traversal cost of cell with corners s, s_1 but not s_2;
07. if $(\min(c, b) = \infty)$
08. $v_s = \infty$;
09. else if $(g(s_1) \leq g(s_2))$
10. $v_s = \min(c, b) + g(s_1)$;
11. else
12. $f = g(s_1) - g(s_2)$;
13. if $(f \leq b)$
14. if $(c \leq f)$
15. $v_s = c\sqrt{2} + g(s_2)$;
16. else
17. $y = \min(\frac{f}{\sqrt{c^2 - f^2}}, 1)$;
18. $v_s = c\sqrt{1 + y^2} + f(1 - y) + g(s_2)$;
19. else
20. if $(c \leq b)$
21. $v_s = c\sqrt{2} + g(s_2)$;
22. else
23. $x = 1 - \min(\frac{b}{\sqrt{c^2 - b^2}}, 1)$;
24. $v_s = c\sqrt{1 + (1 - x)^2} + bx + g(s_2)$;
25. return v_s;

***Figure 7**: ComputeCost function in Field D* algorithm*

Details for implementation

The main difference between Field D* and the optimized version of Field D* is in the $ComputeShortestPath()$.

To improve the ***efficiency*** of Field D* we have done some changes when we update the neighbours of a node by ***reducing the amount of computation.*** To do this, we introduce:

- $Bptr(s)$: for each node s, allow us to specify from which nodes arrives its path cost. In detail, it represents the most clockwise of the two endpoint nodes related to the node s.
- $Cknbr(s, s_1)$: given the node s and its neighbour s_1, it computes the next neighbour in the clockwise direction.
- $Ccknbr(s, s_1)$: given the node s and its neighbour s_1, it computes the next neighbour in the counter clockwise direction.

This is a first optimization that we can do, however there are other kind of possible optimizations to avoid the repetition of some lines of code like {12-13, 15-16, 23-24}.

Summarizing, to optimize these lines of code we have done these changes:

- Instead of computing the rhs_value many times, we update this only by giving the rhs_value of the neighbour to infinity.
- The computation of the rhs_value is done only in the case in which the node s is already in the $OPEN$ list with the g_value grater or equal than the rhs_value. This is useful for the update of the traversal costs and cells when the mobile robot is in the replanning phase.

RESULTS OBTAINED

In this section, we present the results that we obtained by running these different algorithms. These results refer to the optimized version of the algorithms. We explain the parameters and metrics that we have used.

The **parameters** we consider are:

- **Radius**: is the range in which the robot can explore the cells. For all the experiments, we have set $Radius = 3$ units (cells).
- **Cost**: is the cost assigned to each action done. In these experiments, we choose 4 sample costs:
  - $Cost = 0.3$
  - $Cost = 0.6$
  - $Cost = 1.0$
  - $Cost = 5.0$

The metrics considered to study the results and to do the comparisons among the different algorithms are:

- **Initialization time**: time needed to create the first path.
- **Replanning time**: average time needed for replanning.
- **Numbers of explored cells**: number of the cells that the mobile robot explores during the path from the start to the goal position.
- **Total algorithm steps**: number of times the process function is invoked.
- **Path length**: number of cells that the mobile robot has traversed to reach the goal.

RESULTS: D*

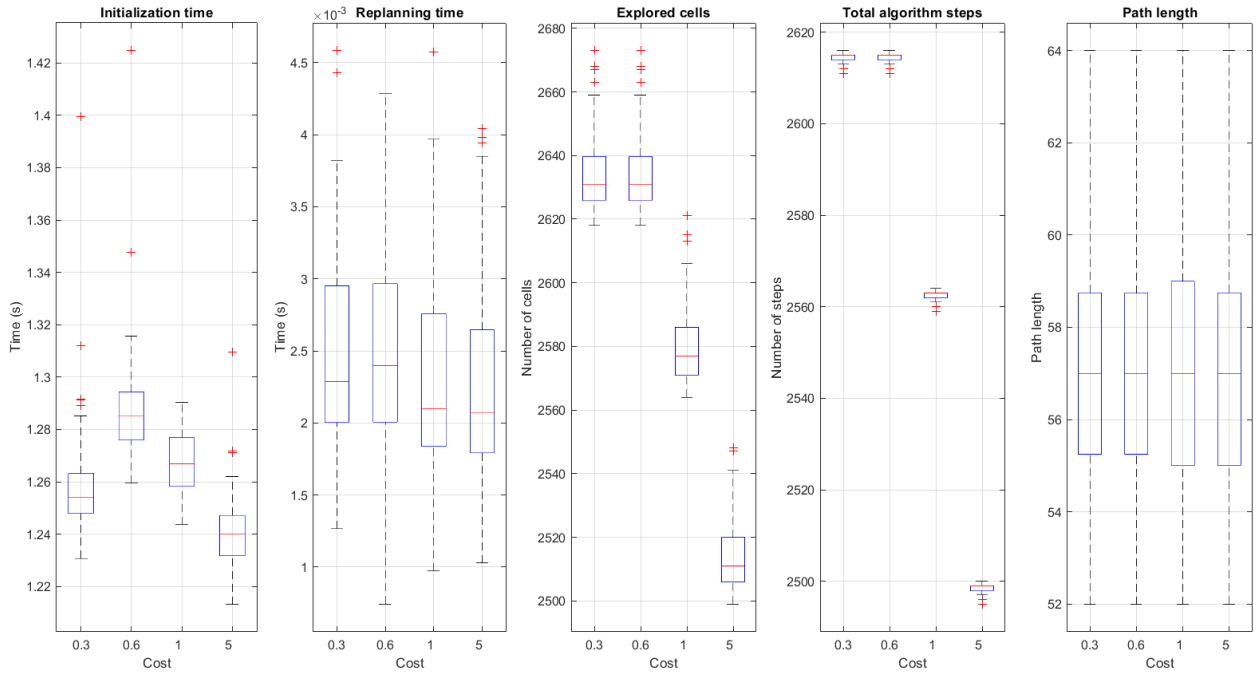


Figure 8: *Results of D**

- The **initialization time** linearly decreases when the $cost > 0.6$ starting from $1.29s$.
- The **replanning time** on average is always the same amount $0.0022s$ with a big variance.
- The explored cells decrease a lot. In detail when $cost < 1$ it is around 2630 and then when $cost \geq 1$ it starts to decrease reaching for $cost = 5$ 2510 explored cells.
- The **total algorithm steps** when $cost < 1$ it is around 2615 and then when $cost \geq 1$ it starts to decrease a lot reaching for $cost = 5$ less than 2500 steps.

RESULTS: D* Lite v1

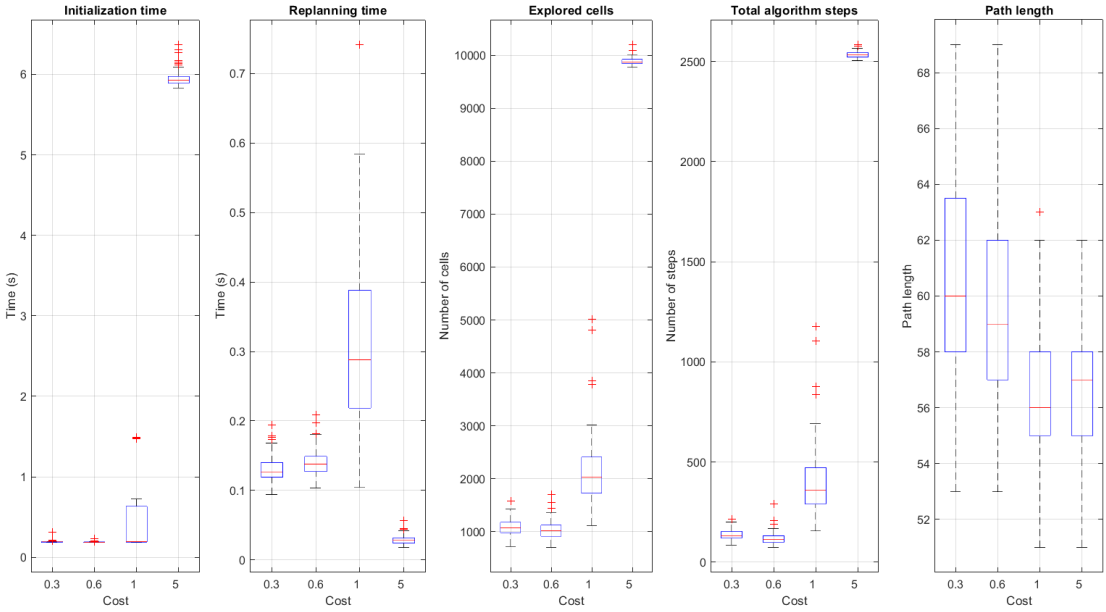


***Figure 9**: Results of D* Lite v1*

- The **initialization time** is very low, around $0.02s$, but when the $cost \geq 1$ it increases a lot reaching for $cost = 5$ $5.80s$.
- The **replanning time** has an incremental behaviour until $cost = 1$ where the value is $0.3s$. The trend decreases when $cost \geq 1$, in which we obtain for $cost = 5$ a very low value (like 0).
- The **number of explored cells** have the same behaviour of initialization time. For $cost \leq 1$ is around 1000, for $cost \geq 1$ it reaches very high value i.e., for $cost = 5$ it is 10000 explored cells.
- The **total algorithm steps** have the same behaviour of initialization time. For $cost \leq 1$ is around 200, for $cost \geq 1$ it reaches very high value i.e., for $cost = 5$ it is 2500 total steps.
- The **path length** decreases.

RESULTS: D* Lite v2

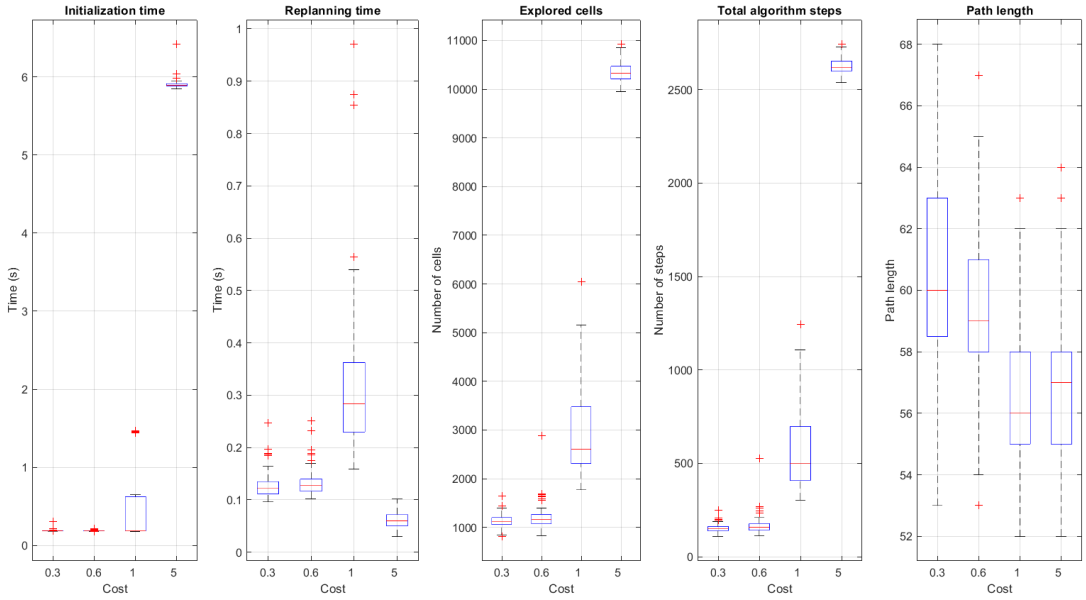


***Figure 10:** Results of D* Lite v2*

Same behaviour of D* Lite v1, but: the path length has a lower variance.

RESULTS: Field D*

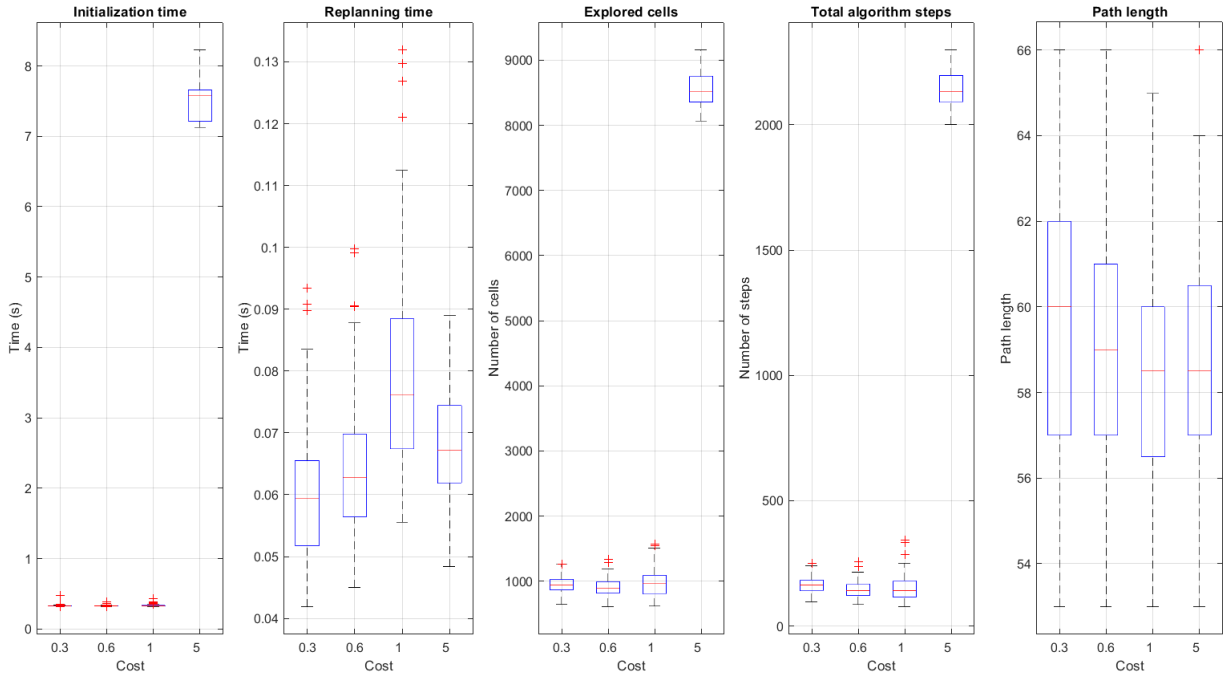


Figure 11: *Results of Field D**

- The **initialization time** is very low when $cost \leq 1$ and it is around $0.2s$ but it increases a lot when $cost = 5$ reaching $7.8s$.
- The **replanning time** has an incremental behaviour that has its maximum when we choose $cost = 1$ around **0.075s**, but the behaviour changes when we set $cost > 1$, in which we obtain lower values.
- The **number of explored cells** has the same value around 1000 for $cost \leq 1$ but then it starts to increase and i.e., for $cost = 5$ it reaches 8500 explored cells.
- The total algorithm steps has the same value around 180 for $cost \leq 1$ but then it starts to increase and i.e., for $cost = 5$ it reaches more than 2000 total steps.
- The **path length** decreases a little.

Moreover, analysing Field D*, we considered another parameter: the **continuous path length**. This is the length of the path not in terms of traversed cells, but rather the length of the path in a continuous – real – environment. From **Figure 12** we can clearly see how Field D* produces a shorter path than the other two thanks to the interpolation planning technique. This is also the reason why, in real applications, it is preferred over D* and D* Lite.

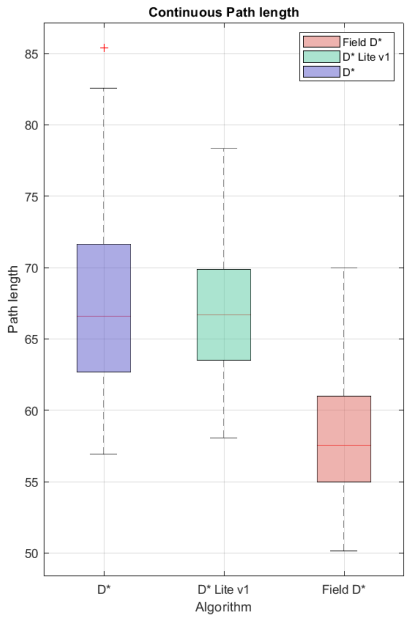
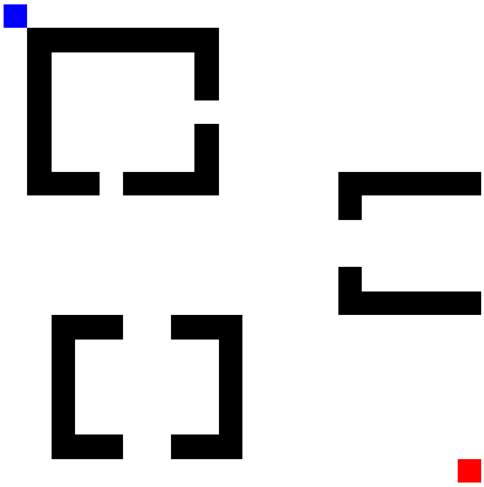


Figure 12: *Continuous path length comparison*

COMPARISONS

To compare better the algorithms, we have built a global map that allows us to highlight the differences.



In the global map we have:

■ : start position

■ : goal position

■ : obstacles

***Figure 13**: Global Map*

The **comparisons** done by using this global map are:

1. D* vs D* Lite v1
2. D* Lite v1 vs Field D*

D* and D* Lite v1

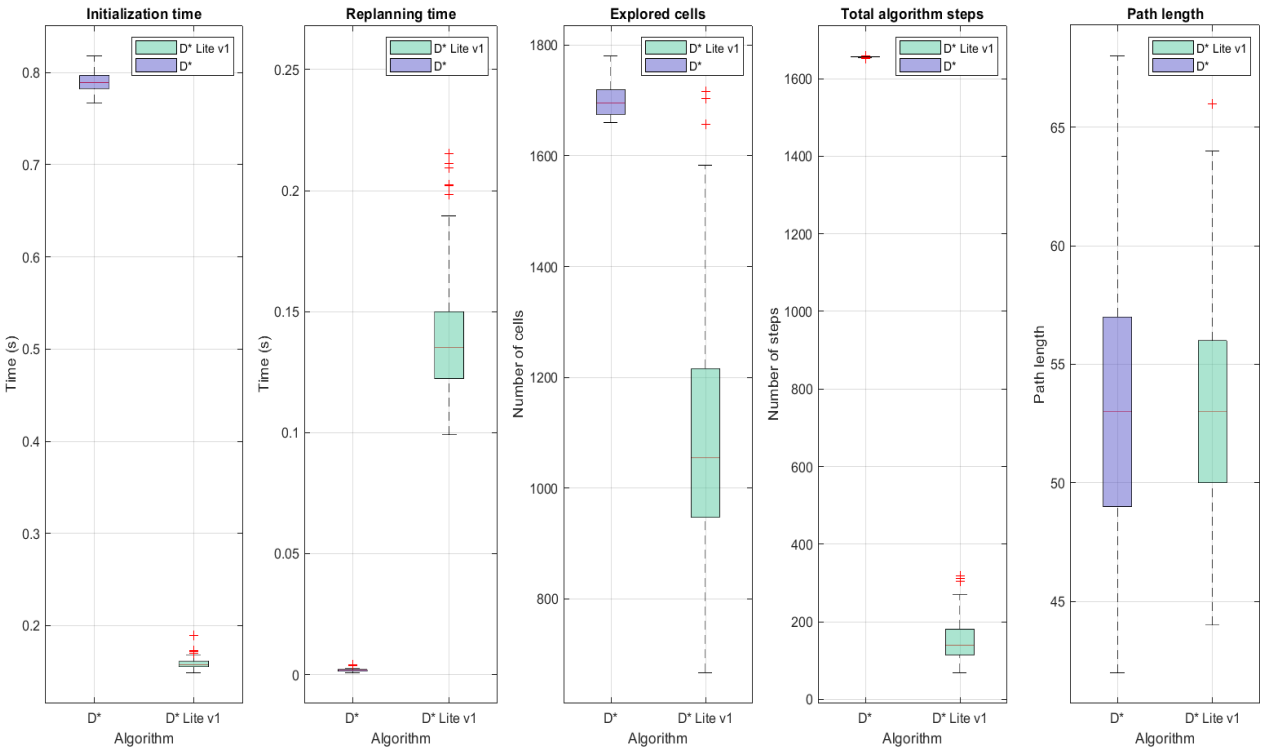


***Figure 14:** Comparison between D* and D* Lite v1*

From this comparison, we expect D* Lite v1 to outperform D*.

In fact, as we can see in **Figure 14**:

- The **initialization time** of D* is higher than those of D* Lite v1 because, during the initialization phase, D* must explore much more open states than D* Lite v1. For states we intend the explored cells and the open cells that we can see in the image.
  In particular, the D* initialization time is around $0.8s$ instead for D* Lite v1 is around $0.1s$;
- For the **replanning time**, we can notice that D* has a very low value that is around $0.01s$ with no variation during the epochs. Instead, in D* Lite v1 the replanning time is higher, around $0.13s$, than D* and it has higher variation during epochs. So, D* Lite v1 has replanning time about ten times higher than those of D* in this example.
- During all the path, the **number of the explored cells** is greater in D* algorithm for how the algorithm is made. In fact, for D* we have 1700 explored cells compared to around **1000** of D* Lite v1.
- The **total algorithms steps** is greater in D* where we have more than 1600 steps to reach the solution, compared to less than 200 in D* Lite v1.

We have captured from our video the final trajectory followed by the mobile robot in the two algorithms, in which we notice the differences:

***Figure 15**: Final trajectories performed by D* and D* Lite v1*

COMPARISON: D* Lite v1 and Field D*

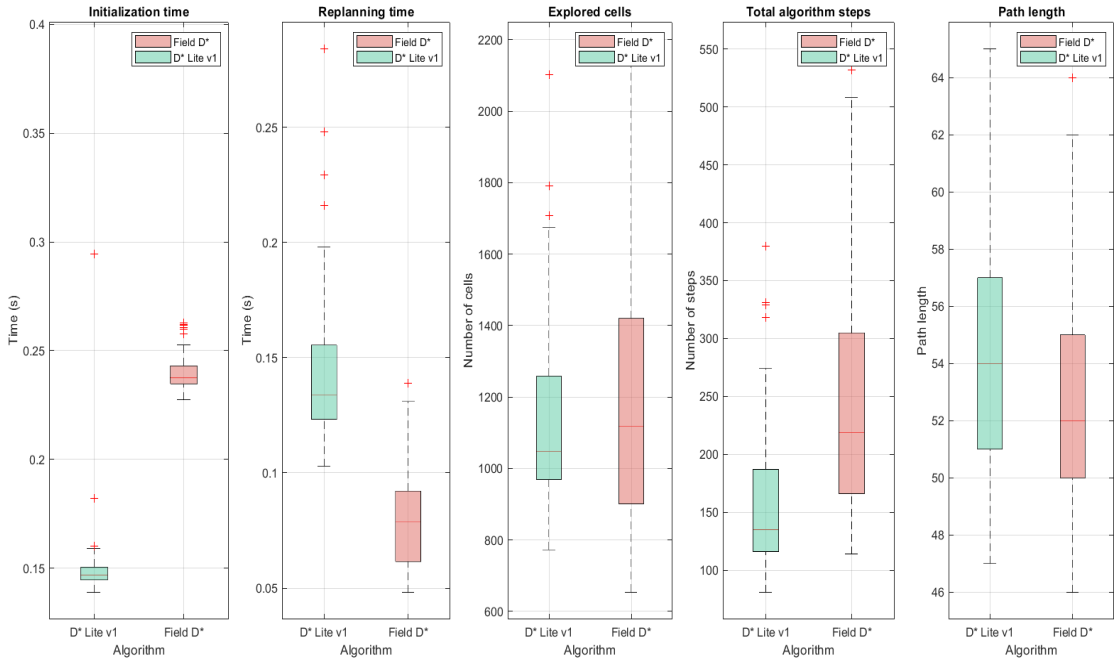


Figure 16: *Comparison between D* and D* Lite v1*

From this comparison, we expect Field D* to outperform D* Lite v1. In fact, as we can see in **Figure 16**:

- The **initialization time** in Field D* is higher about $0.1s$ than D* Lite v1.
- The **replanning time** in Field D* is better because it is lower, so Field D* allows to reach the goal in a faster way by using on average $0.08s$. The difference between the two algorithms is about $0.06s$;
- The **number of explored cells** during the path computation is a little greater on average in Field D*. The variance is very high in Filed D* and it can reach lower values than D* Live v1;
- The **total algorithms steps** is greater in Field* in which there are a little more than 200 total steps compared to less than 150 total steps for D* Lite v1;
- The **path length** is grater in D* Lite v1, around 54 than Field D* around 52.

We have captured from our video the final trajectory followed by the mobile robot in the two algorithms, in which we notice the differences:

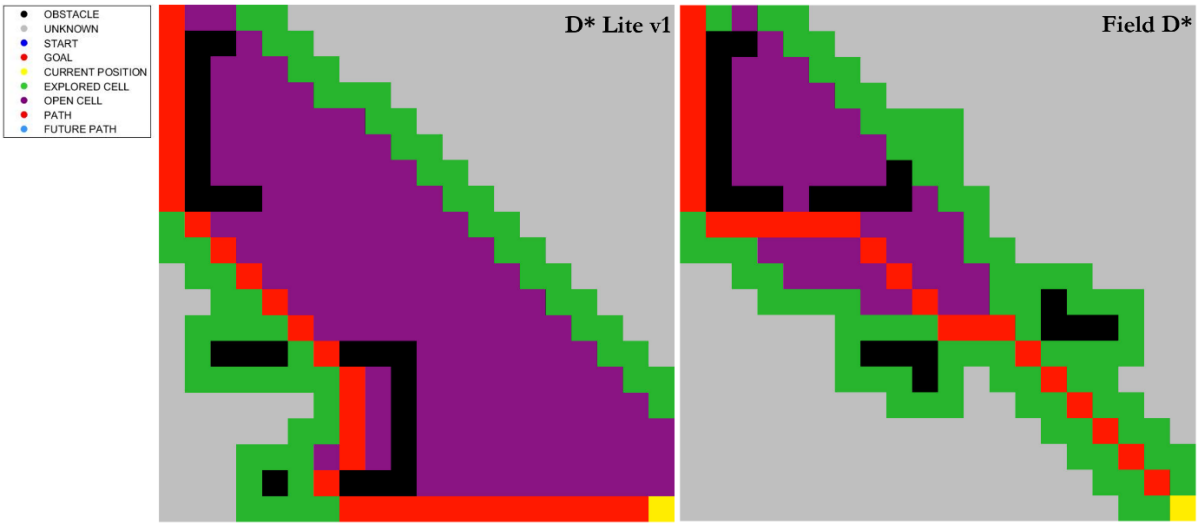


Figure 17: *Final trajectories performed by D* Lite v1 and Field D**

COPPELIASIM SIMULATION

The final part of this job has been making a 3D simulation of a differential drive robot which is placed on a grid and is driven to the goal by the algorithms we have implemented.

First, to make a 3D simulation, we have assumed that obstacles (like walls) are surrounded by "fake" obstacles to avoid collisions and have safer path. We have chosen just one layer of fake obstacles around a real one because we managed to build the environment to have obstacles as cubes with the side greater than the robot length.

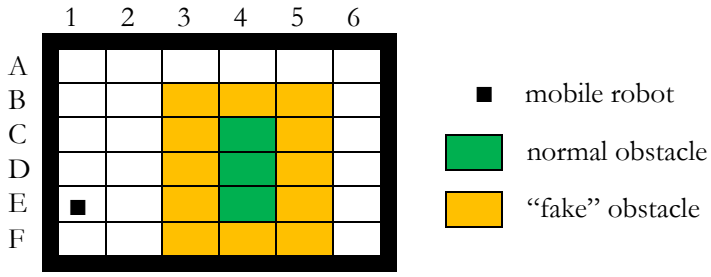
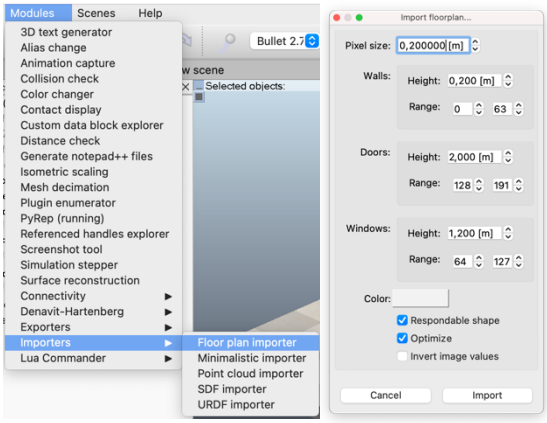


Figure 18: *Obstacles assumption*

We have built the environment in an automated manner by the "Floor plan importer" tool in CoppeliaSim. This tool accepts as inputs a PGM file, which is basically a text file in which are specified details about the grid, which allows to classify a cell grid into four: empty, wall, door, and window.

With walls, doors, and windows it is possible to create a house-like structure, however for our task we needed just walls to represent obstacles. Eventually, we set up the pixel size and the wall's height to $0.2\ m$ to have cube-shaped obstacles. We have chosen $0.2\ m$, because the provided robot has a length of about $0.16\ m$, so, as specified above, only one layer of fake obstacles was necessary to have a safe path.



The final environment obtained is this:

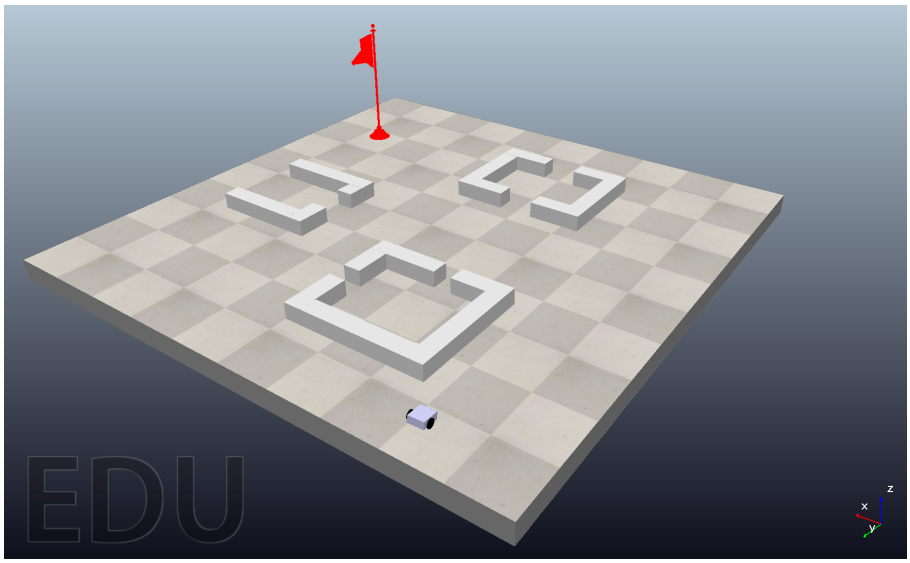


Figure 19*: 3D environment in CoppeliaSim*

Last step was commanding the robot's joints. This was achieved through the RemoteAPI available for CoppeliaSim which expose a web server, available on a port of our choice, with which is possible to give command to any object in the environment. We implemented this in MATLAB and allow the Input-Output Linearization controller to send the inputs directly to the robot.

All the simulation work can be resumed in the following lines:

1. A "modified" algorithm was run to find the path (collision free).
2. The map was converted into PGM and imported into CoppeliaSim.
3. An Input-Output Linearization controller converted the path into control inputs and, at each step, sent them to the robot through the RemoteAPI.

FINAL CONSIDERATIONS

In this work four grid-based path planning algorithms have been studied, implemented, and compared. Pros and cons of every of them have been analysed along this paper, however they can be summarized as the following:

- D* produces good solutions but requires a higher initialization time (and a lower replanning time). On scale, it could be quite slow.
- D* Lite reaches about the same solutions of D* but with considerably less time. On scale, it could be a valid solution.
- Field D* is a bit worse in time than D* lite, but it keeps the promise of making shorter path in real applications.

So, considering real world applications, without any doubt Field D* is the one to choose among the others. D* and D* Lite still remains valid alternatives but nowadays they seem overcome.

REFERENCES

- A. Stentz, "Optimal and efficient path planning for partially-known environments", ICRA 1994;
- S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain", IEEE Transactions on Robotics 2005;
- D. Ferguson and A. Stents, "Field D*: An interpolation-based Path Planner and Replanner", Carnegie Mellon University.
- S. Koenig *et al.*, "Lifelong planning A*," *Artificial Intell. J.,* vol. 155, 2004.