(DRAFT) Assignment 1 (DRAFT)

Path Planning for Disk Holonomic Robots

Deadline: September 30, 11:59pm
Perfect score: 100 points

Assignment Instructions:

Teams: Assignments should be completed by teams of students. Up to three members are allowed for CS460 students and up to two members for CS560 students. Since the assignment has different requirements for the undergraduate and graduate version of the course, we will not allow mixed teams in terms of CS460 and CS560. No additional credit will be given for students working in smaller than the allowed teams. You are strongly encouraged to form a team of the maximum size. Please inform the TAs (email: comprobfall2018/AT/gmail.com) as soon as possible about the members of your team so they can update the scoring spreadsheet. Failure to inform the TAs on time will result in your assignment not being graded.

Submission Rules: Submit your reports electronically as a PDF document through Sakai (`sakai.rutgers.edu`). For your reports, do not submit Word documents, raw text, or hardcopies etc. Make sure to generate and submit a PDF instead. You also need to submit a compressed file via Sakai, which contains your results and/or code as requested by the assignment. *Each team of students should submit only a single copy of their solutions and indicate all team members on their submission*. Failure to follow these rules will result in a lower grade in the assignment.

Program Demonstrations: You will need to demonstrate your program to the TAs and grader on a date after the deadline. The schedule of the demonstrations will be coordinated by the TAs. During the demonstration you have to use the files submitted on Sakai and execute it either on your laptop or an available machine, where the TAs and grader will be located (you probably need to coordinate this ahead of time). You will be asked to describe the architecture of your implementation and algorithmic aspects of the project. You need to make sure that you are able to complete the demonstration and answer the TAs' questions within the allotted 12 minutes of time for each team. If your program is not directly running on the computer you are using and you have to spend time to configure your computer, this counts against your allotted time.

Late Submissions: No late submissions are allowed. You will be awarded 0 points for late assignments.

Extra Credit for LaTeX: You will receive 6% extra credit points if you submit your answers as a typeset PDF (i.e., using LaTeX). Resources on how to use LaTeX are available on the course's website. There will be a 3% bonus for electronically prepared answers (e.g., on MS Word, etc.) that are not typeset. Have in mind that these bonuses are computed as percentage of your original grade, i.e., if you were to receive 50 points and you have typesetted your report using LaTeX then you get 3 points bonus. If you want to submit a handwritten report, scan it and submit a PDF via Sakai. We will not accept hardcopies. If you choose to submit scanned handwritten answers and we are not able to read them, you will not be awarded any points for the part of the solution that is unreadable.

Precision: Try to be precise in your description and thorough in your evaluation. Have in mind that you are trying to convince skeptical evaluators (i.e., computer scientists...) that your answers are correct.

Collusion, Plagiarism, etc.: Each team must prepare its solutions independently from other teams, i.e., without using common code, notes or worksheets with other students or trying to solve problems in collaboration with other teams. You must indicate any external sources you have used in the preparation of your solution. Unless explicitly allowed by the assignment, do not plagiarize online sources and in general make sure you do not violate any of the academic standards of the course, the department or the university (the standards are available through the course's website). Failure to follow these rules may result in failure in the course.

Course's website:
`https://robotics.cs.rutgers.edu/pracsys/courses/intro-to-computational-robotics/`

Assignment Description

1 Setup

The objective of this project is to apply path planning algorithms for defining the sequence of positions a disk, holonomic robot needs to follow in order to reach a desired goal configuration in a known, static polygonal environment.

The robot is a model of a Turtlebot in the Gazebo simulation environment. The environment contains obstacles, whose 2D projections correspond to the polygons, which are provided for path planning purposes. The modeled robot can receive paths as a sequence of intermediate 2D coordinates that need to be reached. Given such a sequence, the simulated robot in Gazebo first rotates in place towards the next intermediate position and then moves along a straight-line path until it reaches it. Then, it proceeds to the next intermediate position on the path in the same manner.

The primary objective of this assignment is to develop algorithms that compute such paths of minimum length. If the path planning process is erroneous, then the robot may collide with obstacles and an error is reported. The format of the ROS messages that the provided Gazebo simulation is able to receive is indicated in the Software Infrastructure subsection below.

The assignment involves the implementation of alternative methods for solving the problem, including both grid-based search algorithms as well as a combinatorial planner.

1.1 Software Infrastructure and Available Input

2 Description of the Grid-based Representation and Search

Given the polygonal representation of the obstacles, you will need to define a 2D grid of square cells that are either blocked or unblocked. The grid should be surrounded by blocked cells.

In order to extract a graph-based representation from the grid so as to perform search, define vertices to be the corner points of the cells, rather than their centers. You will have to pick the length of the cells appropriately as well as their values so as to guarantee that collision-free paths on the grid remain collision-free for the Turtlebot. You will have to explain your choices in your report. You should compute a path on the grid that does not pass through blocked cells or moves between two adjacent blocked cells. A solution path, however, can pass along the border of blocked and unblocked cells. It is up to you to decide whether it makes sense for a solution path to pass through vertices where two blocked cells diagonally touch one another.

The objective of the path planning component is to find the shortest path from a given unblocked start vertex to a given goal vertex. For this project, the algorithms search unidirectionally from the start vertex to the goal vertex.

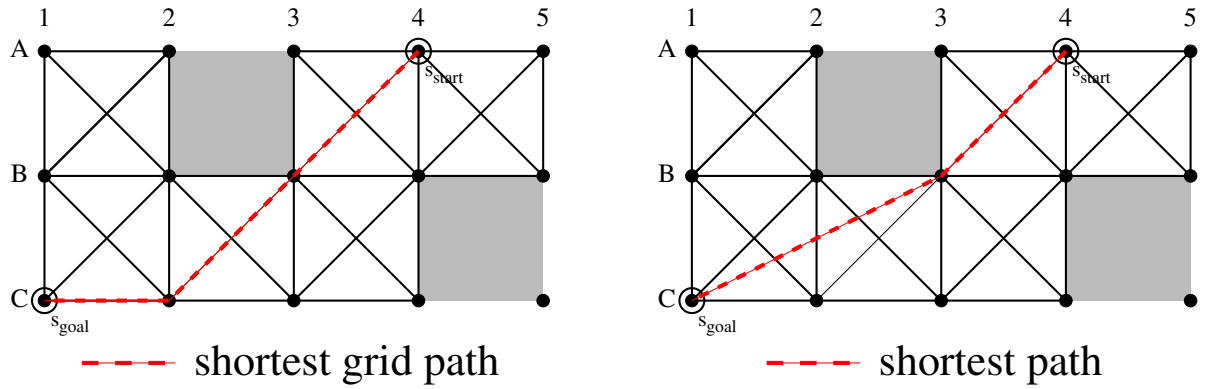


Figure 1: (a) Shortest Grid Path, (b) Shortest Free-Direction Path

Figure 1 provides an example. White cells are unblocked and grey cells are blocked. The start vertex is marked s_{start} and the goal vertex is marked s_{goal}. Figure 1(a) shows the grid edges for an eight-neighbor grid (solid black lines) and a shortest grid path (dashed red line). Figure 1(b) shows the shortest path, which allows the robot to move freely along any direction (dashed red line).

Algorithm 1: A*

```
1  Main()
2      g(s_start) := 0;
3      parent(s_start) := s_start;
4      fringe := ∅;
5      fringe.Insert(s_start, g(s_start) + h(s_start));
6      closed := ∅;
7      while fringe ≠ ∅ do
8          s := fringe.Pop();
9          if s = s_goal then
10             return "path found";
11         closed := closed ∪ {s};
12         foreach s' ∈ succ(s) do
13             if s' ∉ closed then
14                 if s' ∉ fringe then
15                     g(s') := ∞;
16                     parent(s') := NULL;
17                 UpdateVertex(s, s');
18     return "no path found";
19 UpdateVertex(s,s')
20     if g(s) + c(s, s') < g(s') then
21         g(s') := g(s) + c(s, s');
22         parent(s') := s;
23         if s' ∈ fringe then
24             fringe.Remove(s');
25         fringe.Insert(s', g(s') + h(s'));
```

2.1 Review of A*

The pseudo-code of A* is shown in Algorithm 1 and uses the following notation:

- S denotes the set of vertices;
- $s_{start} \in S$ denotes the start vertex (= the current point of the robot);
- $s_{goal} \in S$ denotes the goal vertex (= the destination point of the robot);
- $c(s, s')$ is the straight-line distance between two vertices $s, s' \in S$;
- $succ(s) \subseteq S$ is the set of successors of vertex $s \in S$, which are those (at most eight) vertices adjacent to vertex s so that the straight lines between them and vertex s are unblocked.

You can also check the AI book [1].
For example, the successors of vertex B3 in Figure 1 are vertices A3, A4, B2, B4, C2, C3 and C4. The straight-line distance between vertices B3 and A3 is one, and the straight-line distance between vertices B3 and A4 is $\sqrt{2}$. A* maintains two values for every vertex $s \in S$:

1. the g-value $g(s)$ is the distance from the start vertex to vertex s;
2. the parent $parent(s)$, which is used to identify a path from the start vertex to the goal vertex upon termination.

A* also maintains two global data structures:

1. First, the fringe (or open list) is a priority queue that contains the vertices that A* considers to expand. A vertex that is or was in the fringe is called generated. The fringe provides the following procedures:
   - Procedure *fringe*.*Insert*(s, x) inserts vertex s with key x into the priority queue *fringe*.
   - Procedure *fringe*.*Remove*(s) removes vertex s from the priority queue *fringe*.
   - Procedure *fringe*.*Pop*$()$ removes a vertex with the smallest key from priority queue *fringe* and returns it.
2. Second, the closed list is a set that contains the vertices that A* has expanded and ensures that A* expand every vertex at most once (i.e., we are executing `GRAPH-SEARCH`).

A* uses a user-provided constant h-value (= heuristic value) $h(s)$ for every vertex $s \in S$ to focus the search, which is an estimate of its goal distance (= the distance from vertex s to the goal vertex). A* uses the h-value to calculate an f-value $f(s) = g(s) + h(s)$ for every vertex s, which is an estimate of the distance from the start vertex via vertex s to the goal vertex.

A* sets the g-value of every vertex to infinity and the parent of every vertex to NULL when it is encountered for the first time [Lines 15-16]. It sets the g-value of the start vertex to zero and the parent of the start vertex to itself [Lines 2-3]. It sets the fringe and closed lists to the empty lists and then inserts the start vertex into the fringe list with its f-value as its priority [4-6]. A* then repeatedly executes the following statements: If the fringe list is empty, then A* reports that there is no path [Line 18]. Otherwise, it identifies a vertex s with the smallest f-value in the fringe list [Line 8]. If this vertex is the goal vertex, then A* reports that it has found a path from the start vertex to the goal vertex [Line 10]. A* then follows the parents from the goal vertex to the start vertex to identify a path from the start vertex to the goal vertex in reverse [not shown in the pseudo-code]. Otherwise, A* removes the vertex from the fringe list [Line 8] and expands it by inserting the vertex into the closed list [Line 11] and then generating each of its unexpanded successors, as follows: A* checks whether the g-value of vertex s plus the straight-line distance from vertex s to vertex s' is smaller than g-value of vertex s' [Line 20]. If so, then it sets the g-value of vertex s' to the g-value of vertex s plus the straight-line distance from vertex s to vertex s', sets the parent of vertex s' to vertex s and finally inserts vertex s' into the fringe list with its f-value as its priority or, if it was there already, changes its priority [Lines 21-25]. It then repeats the procedure.

Thus, when A* updates the g-value and parent of an unexpanded successor s' of vertex s in procedure UpdateVertex, it considers the path from the start vertex to vertex s [= $g(s)$] and from vertex s to vertex s' in a straight line [= $c(s, s')$], resulting in distance $g(s) + c(s, s')$. A* updates the g-value and parent of vertex s' if the considered path is shorter than the shortest path from the start vertex to vertex s' found so far [= $g(s')$].

A* with consistent h-values is guaranteed to find shortest grid paths (optimality criterion). H-values are consistent (= monotone) iff (= if and only if) they satisfy the triangle inequality, that is, iff $h(s_{goal}) = 0$ and $h(s) \leq c(s, s') + h(s')$ for all vertices $s, s' \in S$ with $s \neq s_{goal}$ and $s' \in succ(s)$. For example, h-values are consistent if they are all zero, in which case A* degrades to Dijkstra's search (or breadth-first search in this case, since all the edges have the same cost).

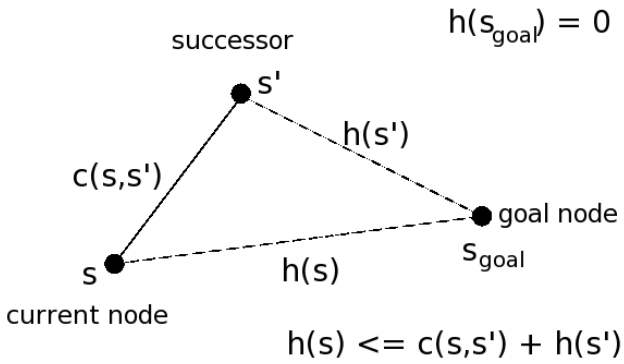


Figure 2: The consistency property for heuristics presented as a triangular inequality.

2.2 Example Trace of A*

Figure 3 shows a trace of A* with the h-values

$$h(s) = \sqrt{2} \cdot \min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + \max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - \min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) \quad (1)$$

to give you data that you can use to test your implementation. s^x and s^y denote the x- and y-coordinates of vertex s, respectively. The labels of the vertices are their f-values (written as the sum of their g-values and h-values) and parents. (We assume that all numbers are precisely calculated although we round them to two decimal places in the figure.) The arrows point to their parents. Red circles indicate vertices that are being expanded. A* eventually follows the parents from the goal vertex C1 to the start vertex A4 to identify the dashed red path [A4, B3, C2, C1] from the start vertex to the goal vertex in reverse, which is a shortest grid path.

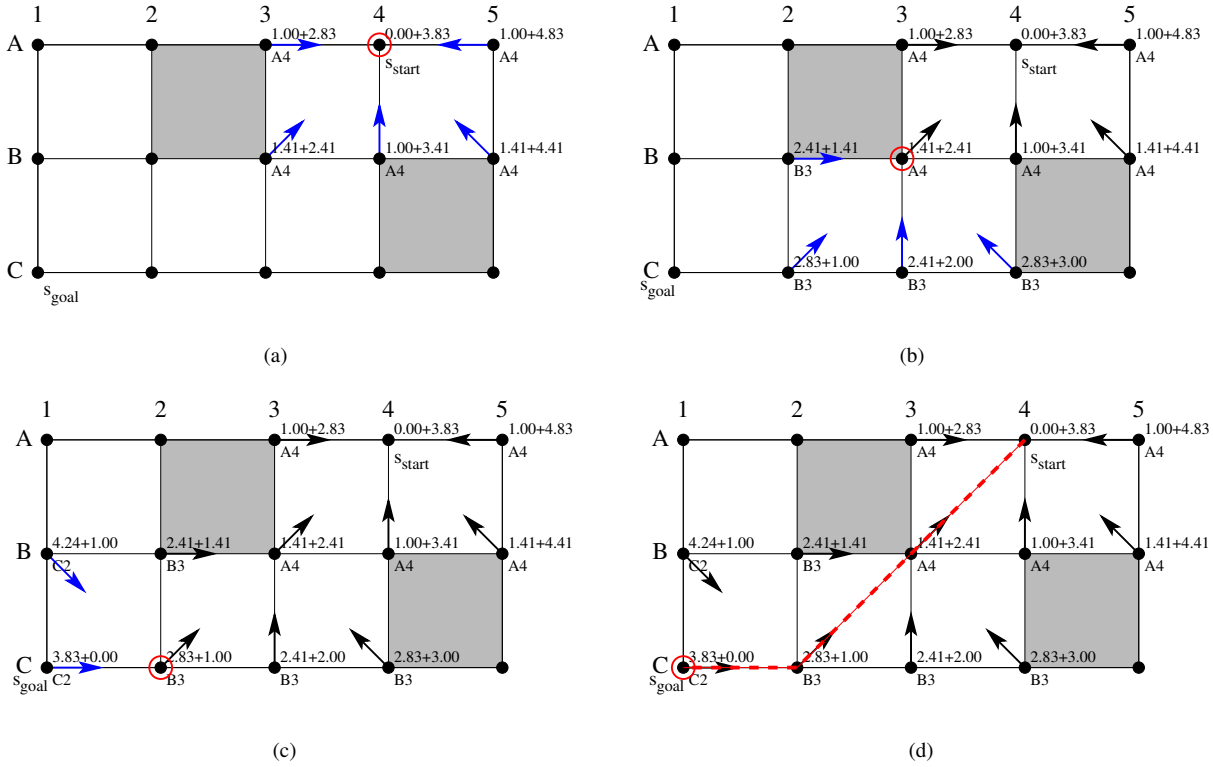


Figure 3: Example Trace of A*

2.3 Free-Direction A*

Consider an extension of A* that allows the robot to move along any direction by propagating information along grid vertices without constraining the paths to lie on grid edges. Lets call this method, Free-Direction A* or FDA*. The key difference between the two algorithms is that in FDA* the parent of a vertex can be any other vertex, while in A* the parent of a vertex has to be a successor.

Algorithm 2: Free-Direction A*

```
1  UpdateVertex(s,s')
2      if LineOfSight(parent(s), s') then
3          /* Path 2 */
4          if g(parent(s)) + c(parent(s), s') < g(s') then
5              g(s') := g(parent(s)) + c(parent(s), s');
6              parent(s') := parent(s);
7              if s' ∈ open then
8                  open.Remove(s');
9              open.Insert(s', g(s') + h(s'));
10     else
11         /* Path 1 */
12         if g(s) + c(s, s') < g(s') then
13             g(s') := g(s) + c(s, s');
14             parent(s') := s;
15             if s' ∈ open then
16                 open.Remove(s');
17             open.Insert(s', g(s') + h(s'));
```

Algorithm 2 provides the pseudo-code for FDA*, as an extension of A* in Algorithm 1. Procedure Main is identical to that of A* and thus not shown. FDA* considers two paths instead of a single path considered by A* when it updates the g-value and parent of an unexpanded successor s' in procedure UpdateVertex. In Figure 4, FDA* is in the process of expanding vertex B3 with parent A4 and needs to update the g-value and parent of unexpanded successor C3 of vertex B3. FDA* considers the following two paths:

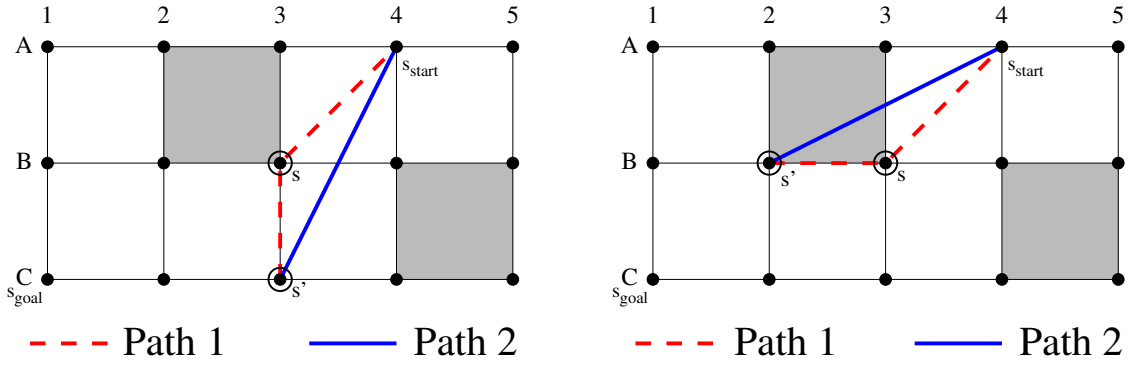
Algorithm 3: Adaptation of the Bresenham Line-Drawing Algorithm

```
1  LineOfSight(s, s')
2      x_0 := s.x;
3      y_0 := s.y;
4      x_1 := s'.x;
5      y_1 := s'.y;
6      f := 0;
7      d_y := y_1 − y_0;
8      d_x := x_1 − x_0;
9      if d_y < 0 then
10         d_y := −d_y;
11         s_y := −1;
12     else
13         s_y := 1;
14     if d_x < 0 then
15         d_x := −d_x;
16         s_x := −1;
17     else
18         s_x := 1;
19     if d_x ≥ d_y then
20         while x_0 ≠ x_1 do
21             f := f + d_y;
22             if f ≥ d_x then
23                 if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
24                     return false;
25                 y_0 := y_0 + s_y;
26                 f := f − d_x;
27             if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
28                 return false;
29             if d_y = 0 AND grid[x_0 + ((s_x − 1)/2), y_0] AND grid[x_0 + ((s_x − 1)/2), y_0 − 1] then
30                 return false;
31             x_0 := x_0 + s_x;
32     else
33         while y_0 ≠ y_1 do
34             f := f + d_x;
35             if f ≥ d_y then
36                 if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
37                     return false;
38                 x_0 := x_0 + s_x;
39                 f := f − d_y;
40             if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
41                 return false;
42             if d_x = 0 AND grid[x_0, y_0 + ((s_y − 1)/2)] AND grid[x_0 − 1, y_0 + ((s_y − 1)/2)] then
43                 return false;
44             y_0 := y_0 + s_y;
45     return true;
```

- **Path 1:** FDA* considers the path from the start vertex to vertex s [= $g(s)$] and from vertex s to vertex s' in a straight line [= $c(s, s')$], resulting in distance $g(s) + c(s, s')$ [Line 12]. This is the path also considered by A*. It corresponds to the dashed red lined from vertex A4 via vertex B3 to vertex C3 in Figure 4(a).
- **Path 2:** FDA* also considers the path from the start vertex to the parent of vertex s [= $g(\textit{parent}(s))$] and from the parent of vertex s to vertex s' in a straight line [= $c(\textit{parent}(s), s')$], resulting in distance $g(\textit{parent}(s))+c(\textit{parent}(s), s')$ [Line 4]. This path is not considered by A* and allows FDA* to construct paths along any direction. It corresponds to the solid blue line from vertex A4 to vertex C3 in Figure 4(a).

Path 2 is no longer than Path 1 due to the triangle inequality. Thus, FDA* chooses Path 2 over Path 1 if the straight line between the parent of vertex s and vertex s' is unblocked. Figure 4(a) gives an example. Otherwise, FDA* chooses Path 1 over Path 2. Figure 4(b) gives an example. FDA* updates the g-value and parent of vertex s' if the chosen path is shorter than the shortest path from the start vertex to vertex s' found so far [= $g(s')$].

LineOfSight($\textit{parent}(s), s'$) on Line 2 is true iff the straight line between vertices $\textit{parent}(s)$ and s' is unblocked. Performing a line-of-sight check is similar to determining which points to plot on a raster display when drawing a straight line between two points. Consider a line that is not horizontal or vertical. Then, the plotted points correspond to the cells that



(a) Path 2 is Unblocked

(b) Path 2 is Blocked

Figure 4: Paths Considered by FDA*

the straight line passes through. Thus, the straight line is unblocked iff none of the plotted points correspond to blocked cells. This allows FDA* to perform the line-of-sight checks with standard line-drawing methods from computer graphics that use only fast logical and integer operations rather than floating-point operations. Algorithm 3 shows the pseudo-code of such a method. $s.x$ and $s.y$ denote the x- and y-coordinates of vertex s, respectively. The value $grid[x, y]$ is true iff the corresponding cell is blocked. Note that the statement $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ is equivalent to $grid[x_0 + ((s_x = 1)?0 : -1), y_0 + ((s_y = 1)?0 : -1)]$ using a conditional expression (similar to those available in C) since s_x and s_y are either equal to -1 or 1. Floating point arithmetic could possibly result in wrong indices.

2.4 Example Trace of FDA*

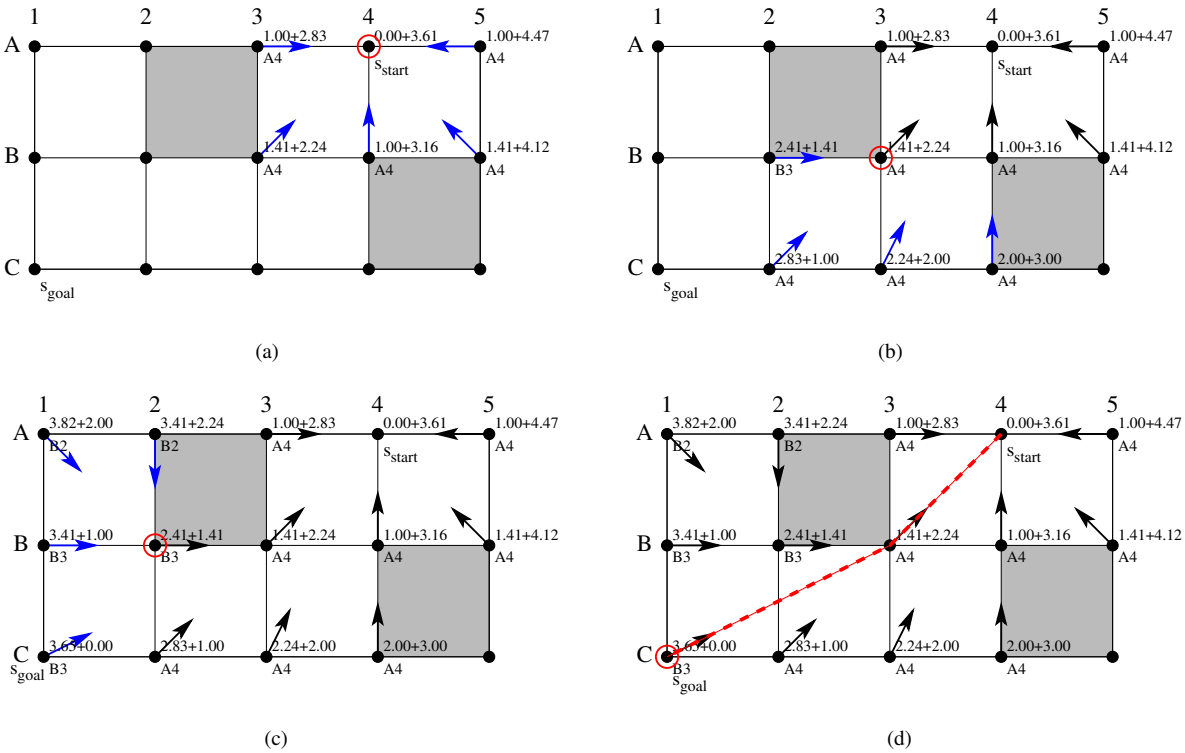


Figure 5: Example Trace of FDA*

Figure 5 shows a trace of FDA* with the h-values $h(s) = c(s, s_{goal})$ to give you data that you can use to test your implementation, similar to the trace of A* from Figure 3. First, FDA* expands start vertex A4, as shown in Figure 5(a). It sets the parent of the unexpanded successors of vertex A4 to vertex A4. Second, FDA* expands vertex B3 with parent A4, as shown in Figure 5(b). The straight line between unexpanded successor B2 of vertex B3 and vertex A4 is blocked. FDA* thus updates vertex B2 according to Path 1 and sets its parent to vertex B3. On the other hand, the straight line between unexpanded successors C2, C3 and C4 of vertex B3 and vertex A4 are unblocked. FDA* thus updates vertices C2, C3 and C4 according to Path 2 and sets their parents to vertex A4. Third, FDA* expands vertex B2 with parent B3, as shown in Figure 5(c). (It can also expand vertex C2, since vertices B2 and C2 have the exact same f-value, and might then find a path different from the one given below.) Fourth, FDA* terminates when it selects the goal vertex C1 for expansion, as shown in Figure 5(d). FDA* then follows the parents from the goal vertex C1 to the start vertex A4 to identify the dashed red path [A4, B3, C1] from the start vertex to the goal vertex in reverse, which is a shortest free-direction path.

2.5 Implementation Details

You are allowed to use existing code for the implementation of the A* although you are encouraged to write your own. You are required, however, to provide your own implementation of Algorithms 2 and 3 for FDA*. You can extend whatever A* framework (yours or existing) for the implementation of FDA*. But do not use code written by others for Algorithms 2 and 3 and test your implementations carefully. For example, make sure that the search algorithms indeed find paths from the start vertex to the goal vertex or report that such paths do not exist, make sure that they never expand vertices that they have already expanded (i.e., avoid repeated states), and make sure that A* with consistent h-values finds shortest grid paths. Use the provided traces to test your implementation.

Your implementations should be efficient in terms of processor cycles and memory usage. Thus, it is important that you think carefully about your implementations rather than use the given pseudo-code blindly since it is not optimized. For example, make sure that your implementations never iterate over all vertices except to initialize them once at the beginning of a search (to be precise: at the beginning of only the first search in case you perform several searches in a row) since you may need to evaluate your program on large grids. Make sure that your implementation does not determine membership in the closed list by iterating through all vertices in it.

Numerical precision is important since the g-values, h-values and f-values are floating point values. An implementation of A* with the h-values from Equation 1 can achieve high numerical precision by representing these values in the form $m + \sqrt{2}n$ for integer values m and n. However, you can follow a more standard choice of implementing A* and FDA* by using 64-bit floating point values ("doubles").

3 Questions and Deliverable

In this project you are asked to implement and evaluate A* , FDA* and a visibility graph approach. Use eight-neighbor grids. You are asked to run a series of experiments and report your results and conclusions. For the experiments and the comparisons use the 5 polygonal environments provided by the TAs and define 10 different start and goal locations per environment, for a total of 50 different benchmarks that you use to compare the different algorithms. You can use the programming language of your preference to implement your path planning algorithm as long as you can interface with the provided Gazebo simulation environment.

Break ties among vertices with the same f-value in favor of vertices with larger g-values and remaining ties in an identical way, for example randomly (this may be different from the examples here). Priorities can also be single numbers rather than pairs of numbers. For example, you can use $f(s) - c \times g(s)$ as priorities to break ties in favor of vertices with larger g-values, where c is a constant larger than the largest f-value of any generated vertex (for example, larger than the longest path on the grid).

Your final report is due on September 30. You are asked to demonstrate your implementations to the TAs during the week of Oct. 1-5 during a 12-15' meeting, which will be coordinated by the TAs.

You are asked to address the following questions:

1. Consider the grid problem shown in Figure 6. Show a shortest grid path and a shortest free-direction path for the example search problem in Figure 6. Show traces of A* with the h-values from Equation 1 and FDA* with the h-values $h(s) = c(s, s_{goal})$ for this example search problem by hand and submit figures similar to Figures 2 and 4. (5 points)

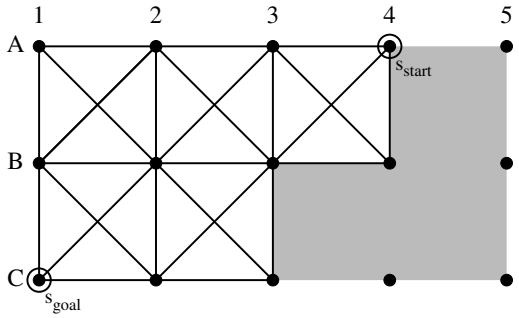


Figure 6: Example Search Problem

2. Explain in your report how you have decided to define the grid given the input polygonal representation of the obstacles. You will need to define a 2D grid of square cells that are either blocked or unblocked. The grid should be surrounded by blocked cells. You should compute a path on the grid that does not pass through blocked cells or moves between two adjacent blocked cells. A solution path, however, can pass along the border of blocked and unblocked cells. It is up to you to decide whether it makes sense for a solution path to pass through vertices where two blocked cells diagonally touch one another.

   You will have to pick the length of the cells appropriately as well as their values so as to guarantee that collision-free paths on the grid remain collision-free for the Turtlebot. Provide a visualization of the grip you consider for each of the polygonal environments that you have received.

   Explain your choices in your report and argue why the grid representation choice allows the computation of short but collision-free paths in the simulated world (i.e., the Turtlebot should not be too far away from obstacles but should avoid collisions). (5 points)

3. Demonstrate the A* algorithm for different start and goal locations in the polygonal environments that you have received. Argue that you are computing the true shortest path on the underlying grid. (10 points)

   Hint: as a debugging and demonstration tool, it may be a good idea to visualize in a 2D interface: the start and the goal location, the path computed by the algorithm, the values h, g and f considered by the algorithm.

4. Implement DFA* and similarly demonstrate the algorithm. Argue that you are computing the true free-direction shortest path. (10 points for CS560 and 15 points for CS460)

5. Optimize your implementation of A* and DFA*. Discuss your optimization in your report (10 points)

6. Compare A* with the h-values from Equation 1 and FDA* with the h-values $h(s) = c(s, s_{goal})$ with respect to their runtimes and the resulting path lengths. In your final report show your experimental results, explain them in detail (including your observations, explanations of the observations and your overall conclusions).

   Discuss whether it is fair that A* and FDA* should use different h-values. If you think it is, explain why. If you think it is not, explain why not, specify the h-values that you suggest both search algorithms use instead and argue why these h-values are a good choice. (10 points for CS560 and 15 points for CS460)

7. A* with consistent h-values guarantees that the sequence of f-values of the expanded vertices is monotonically non-decreasing. In other words, $f(s) \leq f(s')$ if A* expands vertex s before vertex s'. FDA* does not have this property. Construct a (small) example search problem where FDA* expands a vertex whose f-value is smaller than the f-value of a vertex that it has already expanded. List the h-values of FDA* and argue that your h-values are indeed consistent. Then, show a trace of FDA* for this example search problem, similar to Figures 3 and 5. (10 points)

8. CS460 alternative: Compute the visibility graph of the polygonal worlds that you are provided. A visibility graph contains the start vertex, goal vertex and the vertices of the polygons. Two vertices of the visibility graph are connected via a straight line iff the straight line does not intersect any of the polygons.

   CS560 alternative: Compute the reduced visibility graph of the polygonal worlds that you are provided. A reduced visibility graph contains the start vertex, goal vertex and the reflex vertices of the polygons. Two vertices of the

reduced visibility graph are connected via a straight line iff the straight line does not intersect any of the polygons and belongs in one of the following three categories: a) connects two consecutive vertices of the same polygon, or b) is a supporting line segment for the two polygons corresponding to the vertices, or c) is a separating line segment for the two polygons.

In either case: Your implementation for computing the (reduced) visibility graph does not have to be efficient but it has to be correct. Visualize the (reduced) visibility graph for the polygonal worlds you are provided. Demonstrate the shortest collision-free paths that you can compute for the Turtlebot by executing A* on top of the (reduced) visibility graph. Make sure that the path you compute for the Turtlebot is collision-free. Explain how you dealt with the fact that the Turtlebot has a geometry and it is not a point robot. (20 points)

9. Evaluate experimentally by how much the paths found by FDA* with the h-values $h(s) = c(s, s_{goal})$ are longer than those found by running A* on the (reduced) visibility graph. Evaluate the time to run FDA* versus running A* on the (reduced) visibility graph. What if you had to also compute the visibility graph on the fly? Discuss. (10 points)

10. Extra credit for CS460 but required for CS560: Implement the A* search over the (reduced) visibility graph in an implicit manner. That is, do not first explicitly compute the (reduced) visibility graph and then run A* on it. Instead start the A* with a priority queue that contains just the start node and without knowledge of a grid or of a visibility graph. Then, every time you expand a node from the priority queue dynamically detect the neighbors of that node from the (reduced) visibility graph, which have not been expanded yet (still keep a closed list). To detect the neighbors, you will need to consider the vertices of the polygons and the goal node. But you will have to perform dynamically the same test for the existence of the line segment from the expanded node to the neighbor as you do when you build the (reduced) visibility graph. Evaluate your results in terms of time and path length against the FDA* and the A* executed on an explicitly constructed (reduced) visibility graph. (10 points)

11. Extra credit: Implement the rotational sweep algorithm and argue that your implementation for computing the (reduced) visibility graph has indeed $O(n^2 logn)$ time complexity. (10 points)

References

[1] S. J. Russell and N. P., *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.