Assignment 1 - Complete

Problem Solving

Problems 1,b and 1,e will be graded given your initial submission by Wednesday, September 28, end of day. We will also provide feedback whether we can run your software on iLab.
The deadline for the assignment will be Sunday, October 16, end of day.
Perfect score: 100 points (20 extra credit points available + LaTeX 6% bonus)

Assignment Instructions:

Teams: Assignments should be completed by groups of students (up to 3 - from any section). No additional credit will be given for students working individually or in groups of 2. You are encouraged to form a team of three students. Please inform the TAs as soon as possible about the members of your team so they can update the scoring spreadsheet and no later than Sunday, September 18. For this purpose you can use the following Google form: `https://forms.gle/z4vtYEq4ZnoBaBRU8`. Only one member of the team should submit the assignment. You can find the TAs' contact information on the course's syllabus.

Report Submission Rules: Submit your reports electronically as a PDF document through Cavnas. For your reports, do not submit Word documents, raw text, or hardcopies etc. Make sure to generate and submit a PDF instead regardless of how you generated the material. *Each team of students should submit only a single copy of their solutions and indicate all team members (name, netID and section) on their submission*.

Program Evaluation: For the programming component, you need to also submit a compressed file via Canvas, which contains your results and/or code as requested by the assignment. You will need to make sure that your program is executable by the TAs by following the instructions indicated here.

Late Submissions: No late submissions are allowed with the exception of extreme circumstances. Any delay in submitting the assignment will impact your ability to prepare for the midterm.

Start working on the assignment early!

Extra Credit for LaTeX: You will receive 6% extra credit points if you submit your answers as a typeset PDF (i.e., one that has been prepared by using LaTeX, in which case you should also submit electronically the source LaTeX code for the report). There will be a 3% bonus for electronically prepared answers (e.g., on MS Word, etc.), which are not typeset with LaTeX.Resources on how to use LaTeX are available here: `https://robotics.cs.rutgers.edu/pracsys/courses/intro-to-computational-robotics/#latex`

The bonuses are computed as functions of your score, i.e., if you were to receive 50 points out of 100 and you typesetted your answer with LaTeX, then you will get a score of 53. If you want to submit a handwritten report, scan it and submit a PDF but you will not receive any extra credit points. If you choose to submit PDFs of 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. We will not accept hardcopy submissions.

Precision: Try to be precise in your description and thorough in your evaluation. Have in mind that you are trying to convince a very skeptical reader (and computer scientists are the worst kind...) 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. If you are asked to implement an algorithm yourself, you should do so and not use external code. You must indicate any external sources you have used in the preparation of your solution. 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 syllabus). Failure to follow these rules may result in failure in the course.

Problem 1:
Any-Angle Path Planning

Grids with blocked and unblocked cells are often used to represent terrains in computer games, virtual/augmented reality systems, robotic or human-assistive navigation systems. For instance, in games, when the human player selects with the mouse a destination for a game character, then a reasonable, short path has to be computed. Classical search algorithms can find paths on reasonable size grids quickly but these **grid paths** consist of grid edges and their possible headings are thus artificially constrained, which results in them being longer than minimal and unrealistic looking. Any-angle path planning avoids this issue by propagating information along grid edges, to achieve small run-times, without constraining the paths to grid edges, to find **any-angle paths** [1].

1 Project Description

We consider a 2D continuous terrain discretized into a grid of square cells (with side lengths one) that are either blocked or unblocked. We assume that the grid is surrounded by blocked cells. We also assume a static environment (i.e., blocked cells remain blocked and unblocked cells remain unblocked). We define vertices to be the corner points of the cells, rather than their centers. Agents are points that can move along (unblocked) paths. Paths cannot pass through blocked cells or move between two adjacent blocked cells but can pass along the border of blocked and unblocked cells. For simplicity (but not realism), we assume that paths can also pass through vertices where two blocked cells diagonally touch one another. The objective of path planning is to find a short and realistic looking path from a given unblocked start vertex to a given goal vertex. In this project, all algorithms search uni-directionally from the start vertex to the goal vertex.

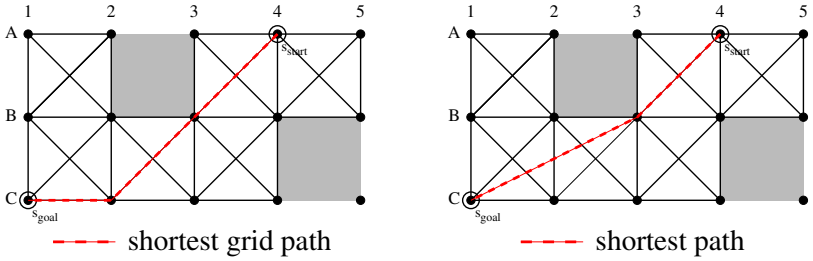


Figure 1: (a) Shortest Grid Path, (b) Shortest Any-Angle Path

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 any-angle path (dashed red line). To understand better which paths are unblocked, assume that cell B3-B4-C4-C3 in Figure 1 is also blocked in addition to cells A2-A3-B3-B2 and B4-B5-C5-C4. Then, the path [A1, B2, B3, A3, A5, C1, C2, A4, B5, B1, C3] is unblocked even though it repeatedly passes through a vertex where diagonally touching blocked cells meet. On the other hand, the following paths are blocked: [B4,C4] and [A4,C4] (paths cannot move between two adjacent blocked cells), [A2,A3] and [A1,A4] (paths cannot move between two adjacent blocked cells and the grid is surrounded by blocked cells - not shown in the figure), [C3,B4] and [C2,B4] and [C1,B5] (paths cannot pass through blocked cells).

2 Review of the A* algorithm (will be covered in class)

The pseudo-code of A* is shown in Algorithm 1. A* is described in your artificial intelligence textbook, covered during the lectures and therefore described only briefly in the following, using the following notation:

- S denotes the set of vertices.
- $s_{start} \in S$ denotes the start vertex (= the current point of the agent), and
- $s_{goal} \in S$ denotes the goal vertex (= the destination point of the agent).
- $c(s, s')$ is the straight-line distance between two vertices $s, s' \in S$.
- Finally, $succ(s) \subseteq S$ 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.

```
Main()
    g(s_start) := 0;
    parent(s_start) := s_start;
    fringe := ∅;
    fringe.Insert(s_start, g(s_start) + h(s_start));
    closed := ∅;
    while fringe ≠ ∅ do
        s := fringe.Pop();
        if s = s_goal then
            return "path found";
        closed := closed ∪ {s};
        foreach s' ∈ succ(s) do
            if s' ∉ closed then
                if s' ∉ fringe then
                    g(s') := ∞;
                    parent(s') := NULL;
                UpdateVertex(s, s');
    return "no path found";

UpdateVertex(s,s')
    if g(s) + c(s, s') < g(s') then
        g(s') := g(s) + c(s, s');
        parent(s') := s;
        if s' ∈ fringe then
            fringe.Remove(s');
        fringe.Insert(s', g(s') + h(s'));
```

Algorithm 1: A*

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. First, the g-value $g(s)$ is the distance from the start vertex to vertex s.
2. Second, the parent $parent(s)$, which is used to identify a path from the start vertex to the goal vertex after A* terminates.

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* (and, later, Theta*) 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 1-1]. It sets the g-value of the start vertex to zero and the parent of the start vertex to itself [Lines 1-1]. 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 [1-1]. A* then repeatedly executes the following statements: If the fringe list is empty, then A* reports that there is no path [Line 1]. Otherwise, it identifies a vertex s with the smallest f-value in the fringe list [Line 1]. 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 1]. 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 1] and expands it by inserting the vertex into the closed list [Line 1] 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 1]. 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 1-1]. 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 uniform-first search (or breadth-first search in this case since all the edges have the same cost).

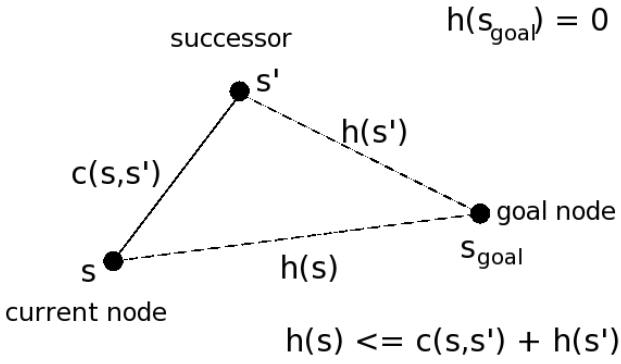


Figure 2: The consistency property for heuristics presented as a triangular inequality.

3 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.

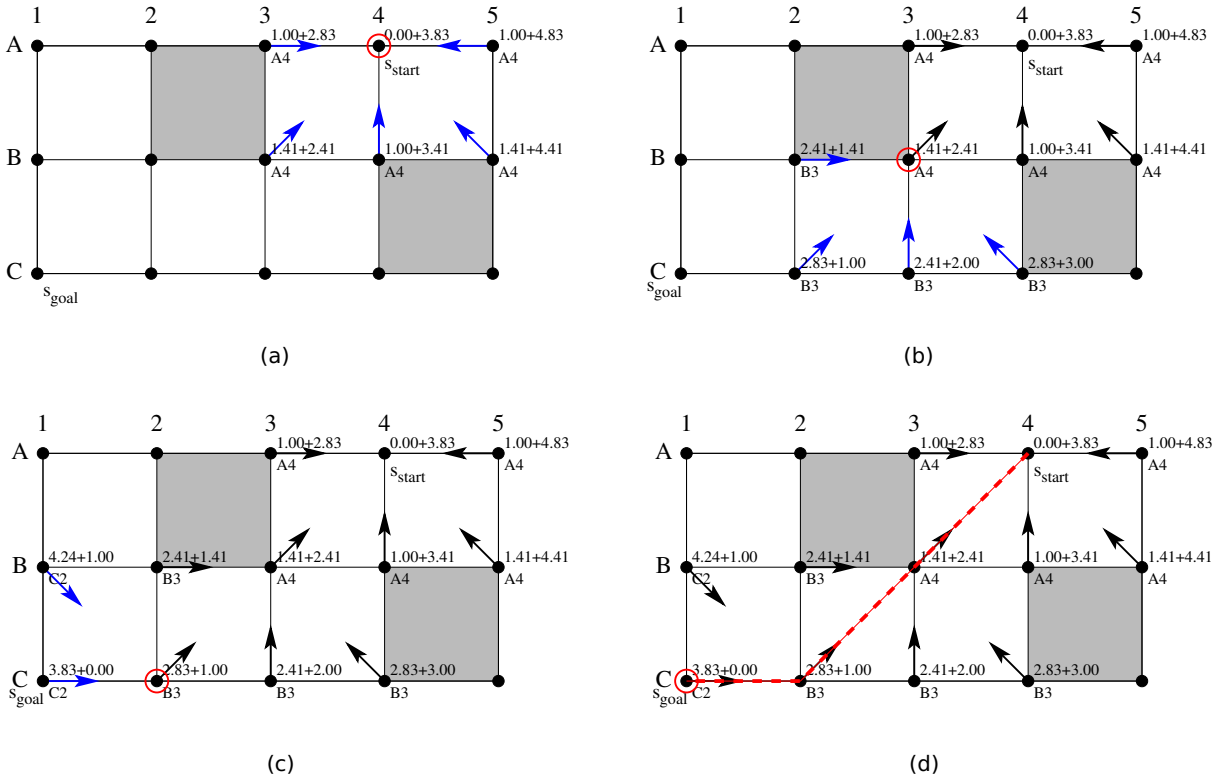


Figure 3: Example Trace of A*

4 Theta*

Theta* is a version of A* for any-angle path planning that propagates information along grid edges without constraining the paths to grid edges. The key difference between the two algorithms is that in Theta* the parent of a vertex can be any other vertex, while in A* the parent of a vertex has to be a successor [2].

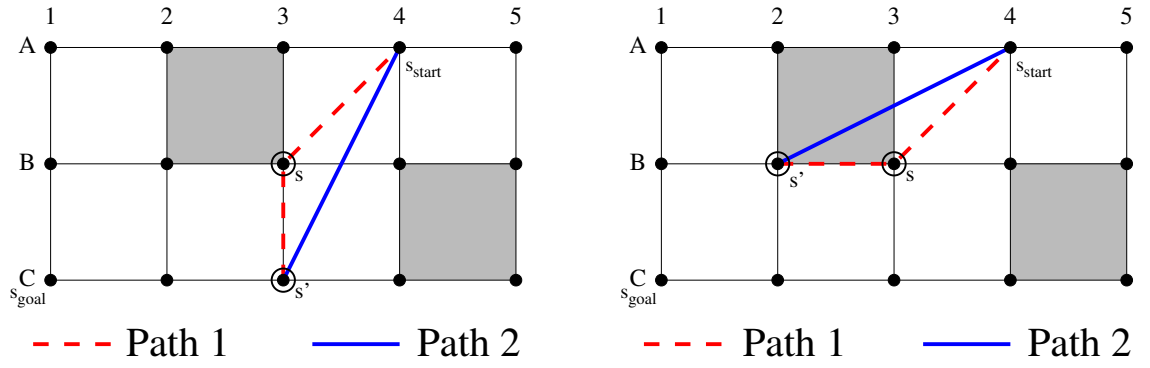
Algorithm 2 shows the pseudo-code of Theta*, as an extension of A* in Algorithm 1. Procedure Main is identical to that of A* and thus not shown. Theta* considers two paths instead of only the one path considered by A* when it updates the g-value and parent of an unexpanded successor s' in procedure UpdateVertex. In Figure 4, Theta* 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. Theta* considers the following two paths:

```
UpdateVertex(s,s')
    if LineOfSight(parent(s), s') then
        /* Path 2 */
        if g(parent(s)) + c(parent(s), s') < g(s') then
            g(s') := g(parent(s)) + c(parent(s), s');
            parent(s') := parent(s);
            if s' ∈ open then
                open.Remove(s');
            open.Insert(s', g(s') + h(s'));
    else
        /* Path 1 */
        if g(s) + c(s, s') < g(s') then
            g(s') := g(s) + c(s, s');
            parent(s') := s;
            if s' ∈ open then
                open.Remove(s');
            open.Insert(s', g(s') + h(s'));
```

Algorithm 2: Theta*

- **Path 1:** Theta* 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 2]. 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:** Theta* also considers the path from the start vertex to the parent of vertex s [$= g(parent(s))$] and from the parent of vertex s to vertex s' in a straight line [$= c(parent(s), s')$], resulting in distance $g(parent(s)) + c(parent(s), s')$ [Line 2]. This path is not considered by A* and allows Theta* to construct any-angle paths. 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, Theta* 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, Theta* chooses Path 1 over Path 2. Figure 4(b) gives an example. Theta* 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 [$= g(s')$].



(a) Path 2 is Unblocked

(b) Path 2 is Blocked

Figure 4: Paths Considered by Theta*

LineOfSight$(parent(s), s')$ on Line 2 is true iff the straight line between vertices $parent(s)$ 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 the straight line passes through. Thus, the straight line is unblocked iff none of the plotted points correspond to

```
LineOfSight(s, s')
    x_0 := s.x;
    y_0 := s.y;
    x_1 := s'.x;
    y_1 := s'.y;
    f := 0;
    d_y := y_1 − y_0;
    d_x := x_1 − x_0;
    if d_y < 0 then
        d_y := −d_y;
        s_y := −1;
    else
        s_y := 1;
    if d_x < 0 then
        d_x := −d_x;
        s_x := −1;
    else
        s_x := 1;
    if d_x ≥ d_y then
        while x_0 ≠ x_1 do
            f := f + d_y;
            if f ≥ d_x then
                if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                    return false;
                y_0 := y_0 + s_y;
                f := f − d_x;
            if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                return false;
            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
                return false;
            x_0 := x_0 + s_x;
    else
        while y_0 ≠ y_1 do
            f := f + d_x;
            if f ≥ d_y then
                if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                    return false;
                x_0 := x_0 + s_x;
                f := f − d_y;
            if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                return false;
            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
                return false;
            y_0 := y_0 + s_y;
    return true;
```

Algorithm 3: Adaptation of the Bresenham Line-Drawing Algorithm

blocked cells. This allows Theta* 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, an adaptation of the Bresenham line-drawing algorithm [3]. $s.x$ 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.

5 Example Trace of Theta*

Figure 5 shows a trace of Theta* with the h-values $h(s) = c(s, s_{goal})$ to give you data to test your implementation, similar to the trace of A* from Figure 3. First, Theta* 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, Theta* expands vertex B3 with parent A4, as in Figure 5(b). The straight line between

unexpanded successor B2 of vertex B3 and vertex A4 is blocked. Theta* 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. Theta* thus updates vertices C2, C3 and C4 according to Path 2 and sets their parents to vertex A4. Third, Theta* expands vertex B2 with parent B3, as in Figure 5(c). (It can also expand C2, since B2 and C2 have the exact same f-value, and might then find a path different from the one below.) Fourth, Theta* terminates when it selects the goal vertex C1 for expansion, as in Figure 5(d). Theta* 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 to the goal in reverse, which is a shortest any-angle path.

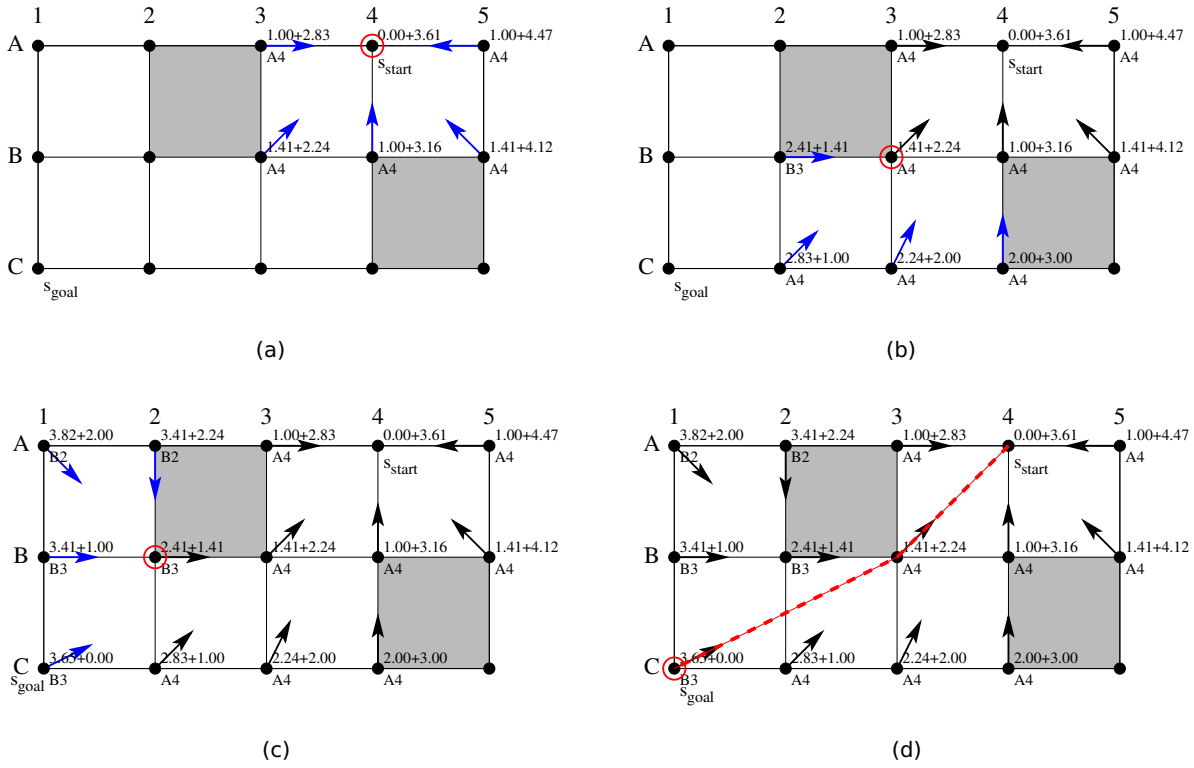


Figure 5: Example Trace of Theta*

6 Implementation Details and Optimization of your Code

Your implementation of A* should use a binary heap [4] to implement the open list. The reason for using a binary heap is that it is often provided as part of standard libraries and, if not, it is easy to implement. At the same time, it is also reasonably efficient in terms of processor cycles and memory usage. You will get extra credit if you implement the binary heap from scratch, that is, if your implementation does not use existing libraries to implement the binary heap or parts of it.

Do not use code written by others 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 (GRAPH-SEARCH), 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. All critical applications place limitations on the resources that search algorithms have available. 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 your program might be used on large grids. Make sure that your implementation does not determine membership in the closed list by iterating through all vertices in it.

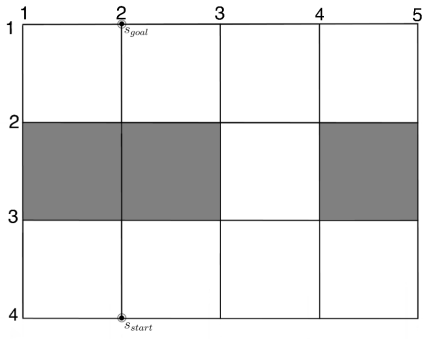
Numerical precision is important since the g-values, h-values and f-values are floating point values. An implementation of A* with the h-values from Equation 1 can achieve high numerical precision by representing these values in the form $m + \sqrt{2}n$ for integer values m and n. Nevertheless, your implementations of A* and Theta* can use 64-bit floating point values ("doubles") for simplicity, unless stated otherwise.

7 Questions and Deliverable

In this project you are asked to implement A* and Theta*, to run a series of experiments and report your results and conclusions. You can work in groups of at most 3. Make sure to include the name of every member of your group in both submissions. Only one member of the team should submit the assignment. Please inform the TAs who are the members of your team by using the corresponding Google Form: `https://forms.gle/z4vtYEq4ZnoBaBRU8`.

Answer the following questions under the assumption that A* and Theta* are used on eight-neighbor grids. Average all experimental results over the same 50 eight-neighbor grids of size 100×50 with 10 percent randomly blocked cells and randomly chosen start and goal vertices so that there exists a path from the start vertex to the goal vertex. You need to generate these grids yourself. To facilitate your experiments, save each grid into a file. Remember that we assumed that paths can pass through vertices where diagonally touching blocked cells meet. All search algorithms search from the start vertex to the goal vertex unidirectionally.

A grid is saved in a file using the following format. The first three lines contain two integers (x y) each, specifying the start vertex, goal vertex, and the dimensions of the grid in terms of columns ×rows (the number of cells, *not* the number of vertices) respectively. Next, there are $x \times y$ lines each with three numbers: the x-coordinate, the y-coordinate and either 0 for a free cell or 1 for a blocked cell. Your experiments have to be on grids of 100×50 but this format allows you to start testing with smaller grids. Refer to the figure below for a small example grid with its corresponding text file.



(a)

```
1 2 4
2 2 1
3 4 3
4 1 1 0
5 1 2 1
6 1 3 0
7 2 1 0
8 2 2 1
9 2 3 0
10 3 1 0
11 3 2 0
12 3 3 0
13 4 1 0
14 4 2 1
15 4 3 0
```

(b)

Figure 6: (a) Example grid of 4×3 with three blocked cells and start and goal vertices. (b) Corresponding text file (line numbers shadowed), numbering starts from the upper left cell.

Different from our examples, A* and Theta* 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. Hint: Priorities can 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

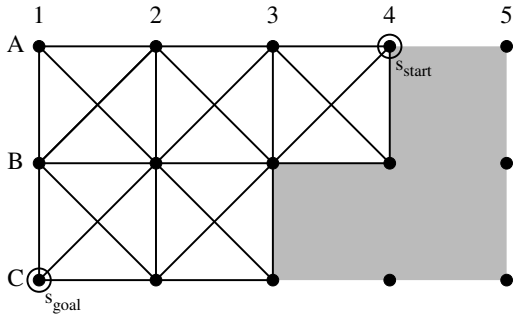


Figure 7: Example Search Problem

constant larger than the largest f-value of any generated vertex (for example, larger than the longest path on the grid).

You are asked to submit part of your submission (questions a, b, c, d, e, f) by September 25. The final submission of this programming assignment as well as followup homework questions will be two weeks later.

Your submission must run and will be tested on ilab (`https://resources.cs.rutgers.edu/docs/instructional-lab/`). You are free to use any programming language already available on ilab. TAs will not upgrade or install anything. The TAs will evaluate your software by using different grid maps than yours, which follow the above format. Submit a compressed file containing your source code as well as a *README* file with instructions on how to test your submission (i.e. compilation, command to execute, etc).
You are asked to address the following questions:

a) Create an interface so as to create and visualize the 50 eight-neighbor grids you are going to use for the experiments.

   Your software should be able to visualize: the start and the goal location, the path computed by an A*-family algorithm. Visualize the values h, g and f computed by A*-family algorithms on each cell (e.g., after selecting with the mouse a specific cell, or after using the keyboard to specify which cell's information to display). Use the images in this report from the traces of algorithms as a guide on how to design your visualization.

   Highlight in your report your visualization, its capabilities and what you implemented for it.

   (10 points - due date: Sept. 28)

b) Read the chapter in your artificial intelligence textbook on uninformed and informed (heuristic) search and then read the project description again. Make sure that you understand A*, Theta* and the Bresenham line-drawing algorithm. Manually compute and show a shortest grid path and a shortest any-angle path for the example search problem from Figure 7. Manually compute and show traces of A* with the h-values from Equation 1 and Theta* with the h-values $h(s) = c(s, s_{goal})$ for this example search problem, similar to Figures 3 and 5.

   (5 points - due date: Sept. 28)

c) Implement the A* algorithm for a given start and goal location for the grid environments.

   Describe in your report what you had to implement in order to have the A* algorithm working.

   (5 points - due date: Sept. 28)

d) Implement Theta*.

   Describe in your report what you had to implement in order to have the Theta* algorithm working.

   (10 points - due date: Sept. 28)

e) Give a proof (= concise but rigorous argument) why A* with the h-values from Equation 1 is guaranteed to find shortest grid paths.

(5 points - due date: Sept. 28)

f) Extra credit: Implement your own binary heap and use it in both algorithms. Discuss your implementation in the final report. If you do execute this extra credit step, make sure that any runtimes you report in your answers below how your own implementation of a binary heap has influenced the running times.

Describe in your report what you had to implement for your own binary heap implementation.

(10 extra credit points - due date: Sept. 28)

For September 28: Submit a report answering the above questions and a copy of your code that is executable on iLab together with instructions on how it should be executed.

g) Optimize your implementation of A* and Theta* in terms of running time and space requirements relatively to your original submission. Discuss your optimizations in your final report and show graphically the difference in running times and space requirements by running experiments.

(5 points - due date: Oct. 16)

h) Compare A* with the h-values from Equation 1 and Theta* 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 and explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions). In particular, average all experimental results over the same 50 eight-neighbor grids of size 100×50 with 10 percent randomly blocked cells and randomly chosen start and goal vertices so that there exists a path from the start vertex to the goal vertex. You need to generate these grids yourself. To facilitate your experiments, save each grid into a file.

(10 points - due date: Oct. 16)

i) Discuss whether it is appropriate for A* and Theta* to 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.

(5 points - due date: Oct. 16)

j) Identify cases where the paths found by Theta* with the h-values $h(s) = c(s, s_{goal})$ are longer than the true minimal path in the continuous case. Report how much longer than the true optimal paths are the Theta* paths and how much longer are the A* paths.

(5 points - due date: Oct. 16)

k) Extra credit: To determine the true shortest paths you can use visibility graphs [5]. A visibility graph contains the start vertex, goal vertex and the corner points of all blocked cells. Two vertices of the visibility graph are connected via a straight line iff the straight line is unblocked. Figure 8, for example, shows the visibility graph for the example search problem from Figure 1. A shortest path from the start vertex to the goal vertex on the visibility graph is guaranteed to be the true shortest path in the continuous space. Show your experimental results in your final report. You already have all the necessary components available for implementing the visibility graph: the line of sight check to define which pairs of vertices are visible and thus which edges exist on the visibility graph, and then the A* algorithm for finding the shortest path on the visibility graph. Note that if you decide to do this extra credit question, you can also use examples where the output of the visibility graph and the Theta* algorithm differ to answer the previous question j).

(10 extra credit points - due date: Oct. 16)

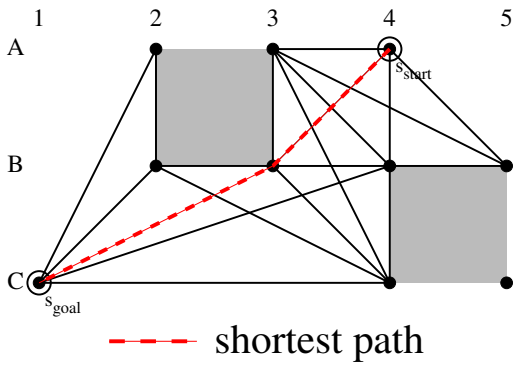


Figure 8: Visibility Graph

l) 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'. Theta* does not have this property. Construct an example search problem where Theta* expands a vertex whose f-value is smaller than the f-value of a vertex that it has already expanded. Do not forget to list the h-values of Theta* and argue that your h-values are indeed consistent. Then, show a trace of Theta* for this example search problem, similar to Figures 3 and 5.

(5 points - due date: Oct. 16)

Together with the final report you will have to submit a copy of your code.

Beyond Heuristic Search

35 points - these problems are due Oct. 16

Problem 2 - Adversarial Search: Consider the two-player game described in Figure 9.

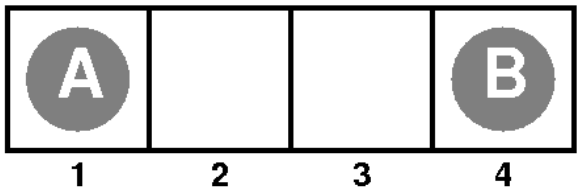


Figure 9: The start position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in *either direction*. If the opponent occupies an adjacent space, than a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is +1; if player B reaches space 1 first, then the value of the game to A is -1.

a. Draw the complete game tree, using the following conventions:

- Write each state as (s_A, s_B) where s_A and s_B denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since it is not clear how to assign values to loop states, annotate each with a "?" in a circle.

b. Now mark each node with its backed-up minimax value (also in circle). Explain how you handled the "?" values and why.

c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

d. This 4-square game can be generalized to n squares for any $n > 2$. Prove that A wins if n is even and loses if n is odd.

(10 points)

Problem 3 - Local Search: Simulated annealing is an extension of hill climbing, which uses randomness to avoid getting stuck in local maxima and plateaux.

a) For what types of problems will hill climbing work better than simulated annealing? In other words, when is the random part of simulated annealing not necessary?

b) For what types of problems will randomly guessing the state work just as well as simulated annealing? In other words, when is the hill-climbing part of simulated annealing not necessary?

c) Reasoning from your answers to parts (b) and (c) above, for what types of problems is simulated annealing a useful technique? What assumptions about the shape of the value function are implicit in the design of simulated annealing?

d) As defined in your textbook, simulated annealing returns the current state when the end of the annealing schedule is reached and if the annealing schedule is slow enough. Given that we know the value (measure of goodness) of each state we visit, is there anything smarter we could do?

(e) Simulated annealing requires a very small amount of memory, just enough to store two states: the current state and the proposed next state. Suppose we had enough memory to hold two million states. Propose a modification to simulated annealing that makes productive use of the additional memory. In particular, suggest something that will likely perform better than just running simulated annealing a million times consecutively with random restarts. [Note: There are multiple correct answers here.]

(10 points)

Problem 4 - Constraint Satisfaction Problem: Consider the game Sudoku, where we try to fill a 9×9 grid of squares with numbers subject to some constraints: every row must contain all of the digits $1, \ldots, 9$, every column must contain all of the digits $1, \ldots, 9$, and each of the 9 different 3×3 boxes must also contain all of the digits $1, \ldots, 9$. In addition, some of the boxes are filled with numbers already, indicating that the solution to the problem must contain those assignments. Here is a sample board:
Each game is guaranteed to have a single solution. That is, there is only one assignment to the empty squares which satisfies all the constraints. For the purposes of this homework, let's use $n_{i,j}$ to refer to the number in row i, column j of the grid. Also, assume that M of the numbers have been specified in the starting problem, where $M = 29$ for the problem shown above.

	1		4	2				5
		2		7	1		3	9
							4	
2		7	1					6
				4				
6					7	4		3
	7							
1	2		7	3		5		
3				8	2		7	

Figure 10: Sample Sudoku board

a. This is an instance of a Constraint Satisfaction Problem. What is the set of variables, and what is the domain of possible values for each? How do the constrains look like?

b. One way to approach the problem, is through an incremental formulation approach and apply backtracking search. Formalize this problem using an incremental formulation. What are the start state, successor function, goal test, and path cost function?

   Which heuristic for backtracking search would you expect to work better for this problem, the degree heuristic, or the minimum remaining values heuristic and why?

   What is the branching factor, solution depth, and maximum depth of the search space? What is the size of the state space?

c. What, is the difference between “easy” and “hard” Sudoku problems? [Hint: There are heuristics which for easy problems will allow to quickly walk right to the solution with almost no backtracking.]

d. Another technique that might work well in solving the Sudoku game is local search. Please design a local search algorithm that is likely to solve Sudoku quickly, and write it in pseudocode. You may want to look at the WalkSAT algorithm for inspiration. Do you think it will work better or worse than the best incremental search algorithm on easy problems? On hard problems? Why?

(10 points)

Problem 5 - Logic-based Reasoning: Consider the following sequence of statements, which relate to Batman’s perception of Superman as a potential threat to humanity and his decision to fight against him.

For Superman to be defeated, it has to be that he is facing an opponent alone and his opponent is carrying Kryptonite. Acquiring Kryptonite, however, means that Batman has to coordinate with Lex Luthor and acquire it from him. If, however, Batman coordinates with Lex Luthor, this upsets Wonder Woman, who will intervene and fight on the side of Superman.

a. Convert the above statements into a knowledge base using the symbols of propositional logic.

b. Transform your knowledge base into 3-CNF.

c. Using your knowledge base, prove that Batman cannot defeat Superman *through an application of the resolution inference rule* (this is the required methodology for the proof).

(5 points)

References

[1] S. Koenig, K. Daniel, and A. Nash, “A project on any-angle path planning for computer games for ’introdction to artificial intelligence’ classes,” Department of Computer Science, University of Southern California, Los Angeles, Tech. Rep., 2008.

[2] A. Nash, K. Daniel, S. Koenig, and A. Felner, “Theta*: Any-angle path planning on grids,” in *Proc. of the AAAI Conf. on Artificial Intelligence (AAAI)*, 2007, pp. 1177–1183.

[3] Bresenham, “Algorithm for computer control of a digital plotter,” *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, September 2009, third Edition.

[5] M. De Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer, 1998.