Prog. Assignment 1: Any-Angle Path Planning for Computer Games

Libraries which produce stunning graphics are widely available and thus games no longer gain all of their competitive advantage from cool graphics. This is why artificial intelligence becomes increasingly important for computer games, many of which require autonomous path-planning capabilities and depend on search algorithms. For example, consider the terrain in a strategy game as the one shown in Figure 1. Grids with blocked and unblocked cells are often used to represent terrains. Grid-based discretizations are also popular in robotic applications. 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. Heuristic search algorithms, such as A*, 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].

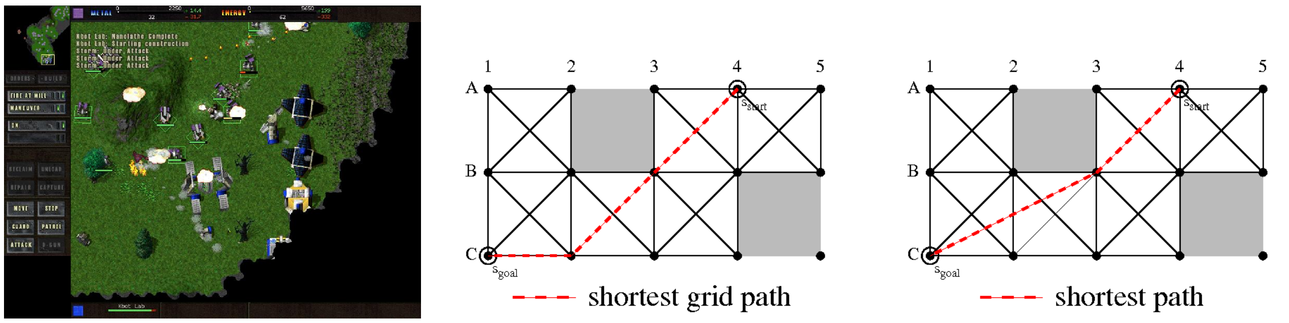


Figure 1: (a) Total Annihilation by Cavedog, (b) Shortest Grid Path, (c) Shortest Any-Angle Path

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 unidirectionally from the start vertex to the goal vertex.

Figures 1(b)-(c) provide 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(b) shows the grid edges for an eight-neighbor grid (solid black lines) and a shortest grid path (dashed red line). Figure 1(c) 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).

```
Main()
    $g(s_{start}) := 0$;
    $parent(s_{start}) := s_{start}$;
    $fringe := \emptyset$;
    $fringe.Insert(s_{start}, g(s_{start}) + h(s_{start}))$;
    $closed := \emptyset$;
    while fringe $\neq \emptyset$ do
        $s := fringe.Pop()$;
        if $s = s_{\text{goal}}$ then
            return "path found";
        end
        $closed := closed \cup \{s\}$;
        foreach $s' \in$ succ($s$) do
            if $s' \notin$ closed then
                if $s' \notin$ fringe then
                    $g(s') := \infty$;
                    $parent(s') := NULL$;
                end
                UpdateVertex($s, s'$);
            end
        end
    end
    return "no path found";
end

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

Algorithm 1: A*

2 Review of A*

The pseudo-code of A* is shown in Algorithm 1. A* is described in your artificial intelligence textbook 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.

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).

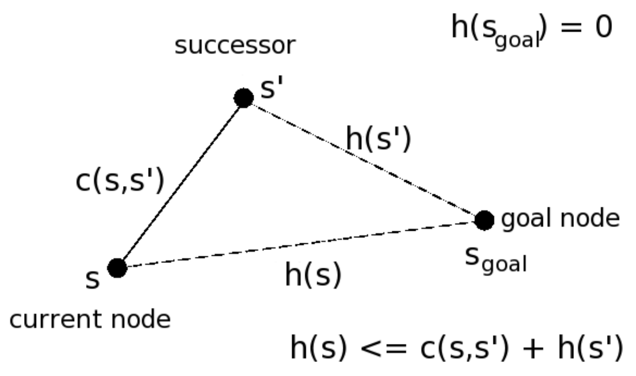


Figure 2: The consistency property for heuristics presented as a triangular inequality.

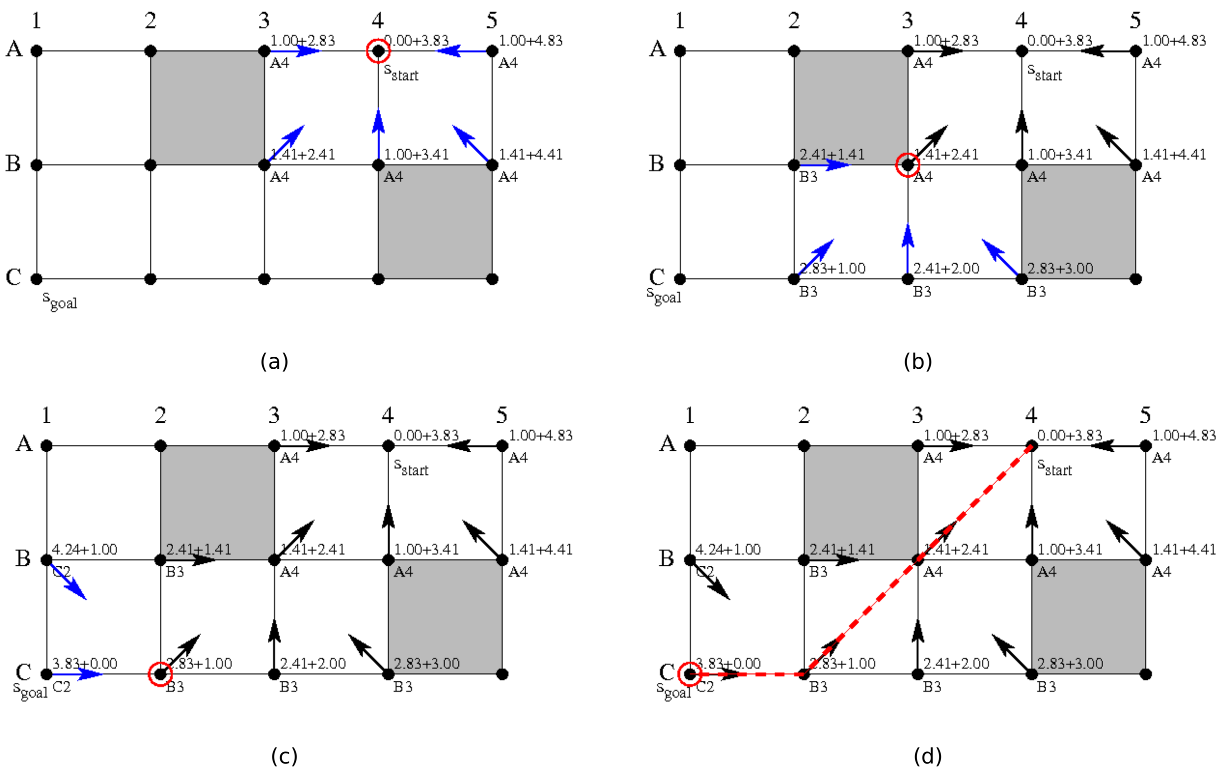


Figure 3: Example Trace of A*

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.

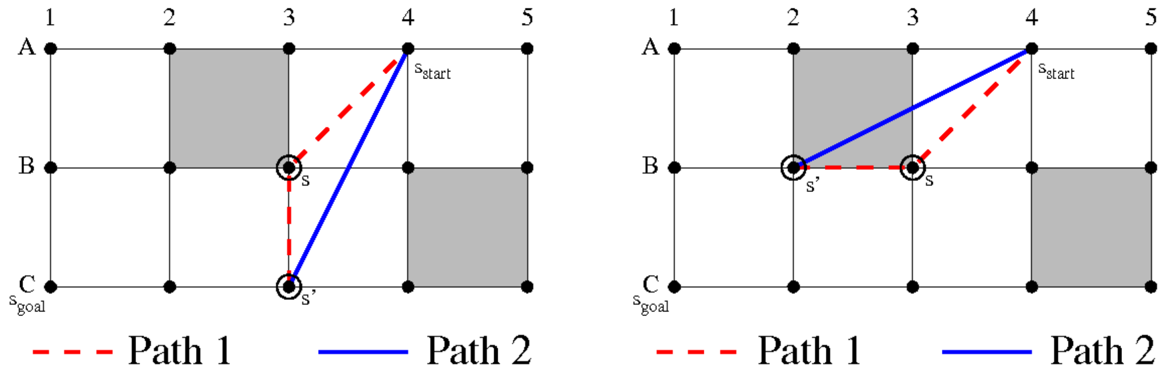
4 Theta*

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 $g(\text{parent}(s)) + c(\text{parent}(s), s') < g(s')$ then
            $g(s') := g(parent(s)) + c(parent(s), s')$;
            $parent(s') := parent(s)$;
            if $s' \in$ open then
                $open.Remove(s')$;
            end
            $open.Insert(s', g(s') + h(s'))$;
        end
    else
        /* Path 1 */
        if $g(s) + c(s, s') < g(s')$ then
            $g(s') := g(s) + c(s, s')$;
            $parent(s') := s$;
            if $s' \in$ open then
                $open.Remove(s')$;
            end
            $open.Insert(s', g(s') + h(s'))$;
        end
    end
end
```

Algorithm 2: Theta*



(a) Path 2 is Unblocked (b) Path 2 is Blocked

Figure 4: Paths Considered by 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 [$= g(s')$].

$LineOfSight(parent(s), s')$ 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

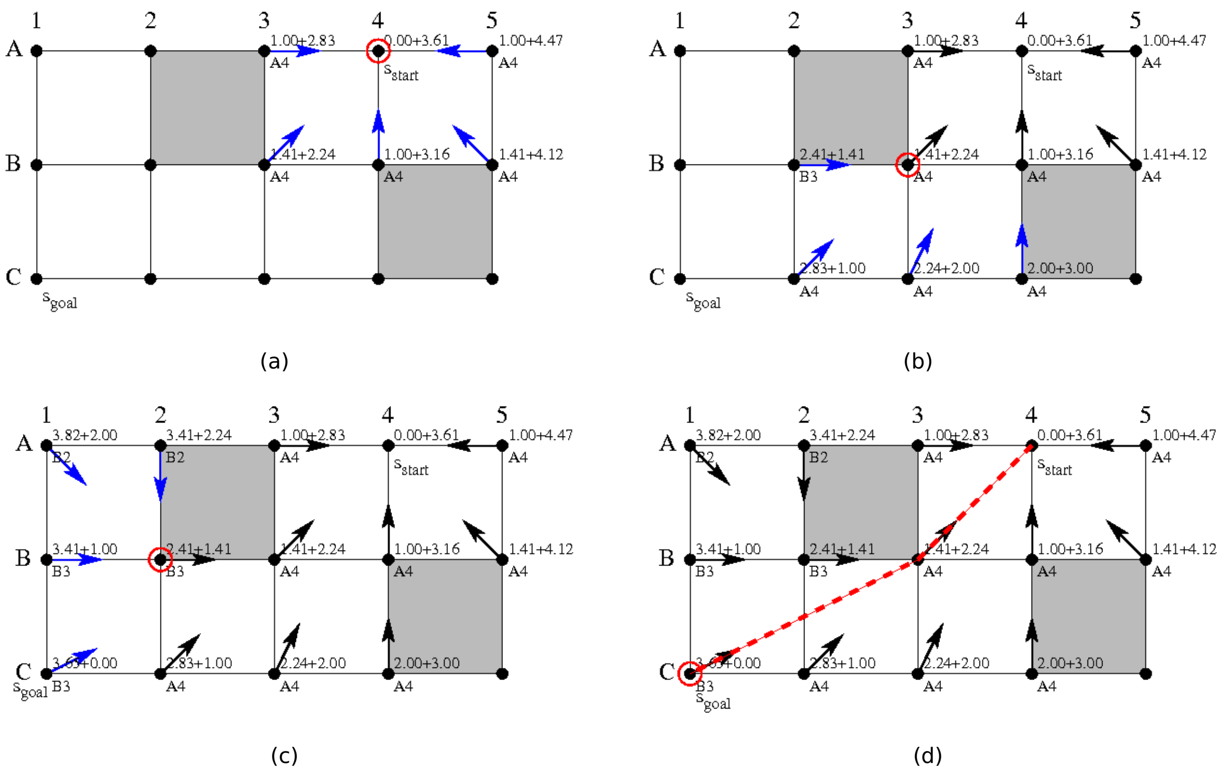


Figure 5: Example Trace of Theta*

passes through. Thus, the straight line is unblocked iff none of the plotted points correspond to 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 that you can use 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 shown 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 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, Theta* terminates when it selects the goal vertex C1 for expansion, as shown 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 vertex to the goal vertex in reverse, which is a shortest any-angle path.

6 Implementation Details

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. After all game companies place limitations on the resources that path planning has 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. However, 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 use the programming language of your preference to implement and visualize the results of your algorithms. You can work in pairs. Please inform the instructor who are the members of your team.

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. 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.

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 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 present the progress of your project (questions a, b, c, d) on. Feb. 10 during a 10 minutes demo. Please schedule your meeting as soon as possible. The available time slots are 1pm-5pm on Feb. 10. Coordinate with the instructor if you are not available on that day. The final presentation of your project to the instructor will take place on Feb. 24 (same time slots available). Your final report is due before Lecture 9 (Feb. 21). If you typeset your final report in LaTeX, you will receive 5% extra credit. Check the course's website regarding information on how to use LaTeX.

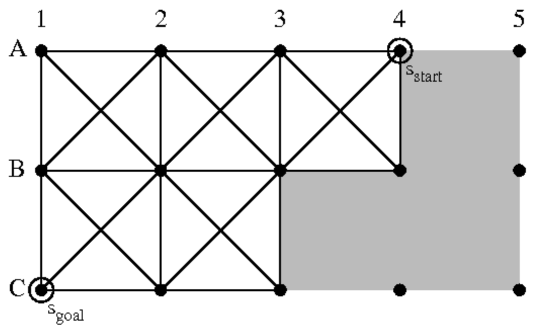


Figure 6: Example Search Problem

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 also be able to visualize: the start and the goal location, the path computed by a heuristic search 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.

   (10 points - due date: Feb. 10)

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 6. 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.

   (10 points - due date: Feb. 10)

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

   (10 points - due date: Feb. 10)

d) Implement Theta*.

   (10 points - due date: Feb. 10)

e) Optimize your implementation of A* and Theta*. Discuss your optimizations in your final report.

   (15 points - due date: Feb. 24)

f) 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, explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions) and discuss in detail whether it is fair that A* and Theta* 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 - due date: Feb. 24)

g) Determine experimentally by how much the paths found by Theta* with the h-values $h(s) = c(s, s_{goal})$ are longer than minimal. To this end, determine the shortest any-angle paths with

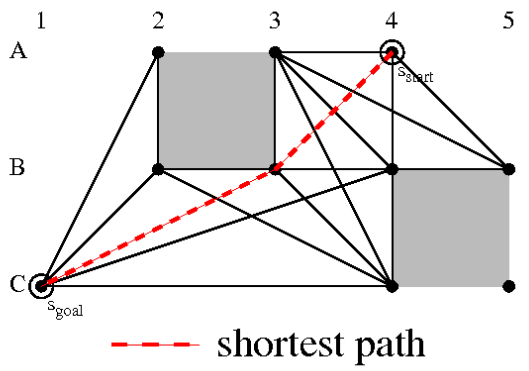


Figure 7: Visibility Graph

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 7, 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 shortest any-angle path. Show your experimental results in your final report.

(15 points - due date: Feb. 24)

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

(10 points - due date: Feb. 24)

i) 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.

(10 points - due date: Feb. 24)

j) Implement your own binary heap and use it in both algorithms. Discuss your implementation in the final report. Make sure that any runtimes you report in your answers to the above questions correspond to runs that have used your own implementation of a binary heap.

(Extra credit: 10 points - while the due date is Feb. 24, it will probably be advantageous for your team if you aim to have the binary heap ready on Feb. 10 together with the initial implementations of A* and Theta*)

Your code should be compiling and running either on a laptop that you will bring together in the meetings with the instructor or on a publicly accessible computer in the SEM that you will use to present your results to the instructor.

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.

```
**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
        $d_y := -d_y$;
        $s_y := -1$;
    end
    else
        $s_y := 1$;
    end
    if $d_x < 0$ then
        $d_x := -d_x$;
        $s_x := -1$;
    end
    else
        $s_x := 1$;
    end
    if $d_x \geq d_y$ then
        while $x_0 \neq x_1$ do
            $f := f + d_y$;
            if $f \geq d_x$ then
                if $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
                    return $false$;
                end
                $y_0 := y_0 + s_y$;
                $f := f - d_x$;
            end
            if $f \neq 0$ $AND$ $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
                return $false$;
            end
            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 $false$;
            end
            $x_0 := x_0 + s_x$;
        end
    end
    else
        while $y_0 \neq y_1$ do
            $f := f + d_x$;
            if $f \geq d_y$ then
                if $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
                    return $false$;
                end
                $x_0 := x_0 + s_x$;
                $f := f - d_y$;
            end
            if $f \neq 0$ $AND$ $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
                return $false$;
            end
            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 $false$;
            end
            $y_0 := y_0 + s_y$;
        end
    end
    return $true$;
end
```

Algorithm 3: Adaptation of the Bresenham Line-Drawing Algorithm