A Project on Any-Angle Path Planning for Computer Games for "Introduction to Artificial Intelligence" Classes

Sven Koenig Kenny Daniel Alex Nash
Department of Computer Science
University of Southern California
300 Henry Salvatori Computer Science Center (SAL)
941 W 37th Street
Los Angeles, CA 90089-0781
{skoenig, kfdaniel, anash}@usc.edu

Abstract

This stand-along path-planning project for an undergraduate or graduate artificial intelligence class relates to video game technologies and is part of our effort to use computer games as a motivator in projects without the students having to use game engines. Heuristic search and, in particular, A* are among the important single-agent search techniques and thus good candidates for a project in artificial intelligence. In this project, the students need to code A* and then extend it to Theta*, an any-angle algorithm, to plan paths for game characters in known gridworlds. Theta*, similar to A*, propagates information along the edges of the gridworld (to achieve a small runtime) but, different from A*, does not constrain the paths to the edges (to find shorter paths than A*). This project requires students to develop a deep understanding of A* and heuristics to answer questions that are not yet covered in textbooks. The project is versatile since it allows for theoretical questions and implementations. We list a variety of possible project choices, including easy and difficult questions.

Introduction

The Department of Computer Science at the University of Southern California created a Bachelor's Program in Computer Science (Games) and a Master's Program in Computer Science (Game Development), which not only provide students with all the necessary computer science knowledge and skills for working anywhere in industry or pursuing advanced degrees but also enable them to be immediately productive in the game-development industry. They consist of regular computer science classes, game-engineering classes, game-design classes, game cross-disciplinary classes and a final game project. See Zyda and Koenig, Teaching Artificial Intelligence Playfully, Proceedings of the AAAI-08 Education Colloquium, 2008 for more information. The undergraduate and graduate versions of the "Introduction to Artificial Intelligence" class at the University of Southern California are regular computer science classes that are part of this curriculum. We are now slowly converting these classes to use games as the domain for projects because games motivate students, which we believe increases enrollment and retention and helps us to educate better computer scientists.

Artificial intelligence becomes more and more important for computer games, now that games use graphics libraries that produce stunning graphics and thus no longer gain much of a competitive advantage via their graphics capabilities. Many games need path-planning capabilities and thus use search algorithms. Some games already use machine learning or planning algorithms. For example, “Black and White” uses a combination of inductive learning of decision trees and reinforcement learning with neural networks, and “F.E.A.R” uses goal-oriented action planning. Thus, games can be used to illustrate many areas of artificial intelligence.

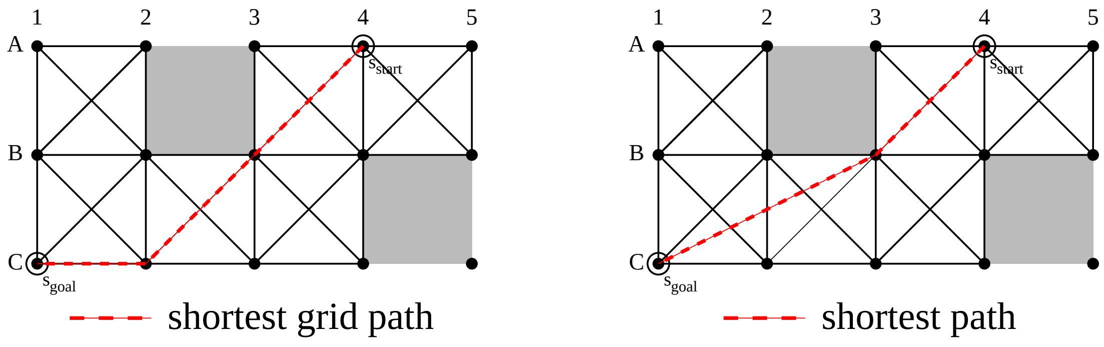
It was tempting for us to use a publicly available game engine throughout the class and then ask the students to perform several projects in it, such as a search project to plan the paths of the game characters and a machine-learning project to make them adapt to the behavior of their opponents. However, students familiar with game development already have plenty of exposure to game engines while students not interested in game development do not need the additional overhead. To put all students on the same footing and allow them to concentrate on the material taught in class, we decided to go with several small projects that do not need large code bases.

This technical report describes one particular project from the undergraduate and graduate versions of the “Introduction to Artificial Intelligence” class. Heuristic search and, in particular, A* are among the important single-agent search techniques and thus good candidates for a project in artificial intelligence. We therefore developed a project where the students use a generalization of A*, Theta*, to plan any-angle paths for game characters. The students need to code A* and then develop a deep understanding of A* and heuristics to answer questions that are not yet covered in textbooks and require computational thinking when they extend A* to Theta*. The search project is versatile since it allows for theoretical questions and for implementations. We list a variety of possible project choices, including easy and difficult questions. Teachers need to select among them since we list too many of them for a project of a reasonable size. (Questions tagged with asterisks are the ones that we tend to use in our own projects.) Teachers also need to provide information on which programming language(s) the students are allowed to use, how they can determine the runtime of their programs, what they need to submit for their projects (for example, whether they need to submit their programs or traces of their programs on example search problems), how they should submit their projects (for example, on paper or electronically) and by when they need to submit their projects.

If you are using this assignment in your class or have any questions, comments or corrections, please send us an email at skoenig@usc.edu. If you need to cite this assignment (in your own publications or to give us credit), please cite this technical report as Sven Koenig, Kenny Daniel and Alex Nash, A Project on Any-Angle Path Planning for Computer Games for “Introduction to Artificial Intelligence” Classes, Technical Report, Department of Computer Science, University of Southern California, Los Angeles (California, USA). Updates, errata and supporting material for the project (including the latex version of this document to allow teachers to customize this assignment) can be found on our webpages idm-lab.org/gameai, where we will also release additional projects in the future. If there is sufficient interest, we will create sample solutions for teachers at accredited colleges and universities.

Acknowledgments

This initiative was partly supported by a grant from the Fund for Innovative Undergraduate Teaching at the University of Southern California. We thank Ariel Felner for earlier joint research that strongly influenced the creation of this project. We thank Xiaoxun Sun for helpful comments that improved our project text. This project is still undergoing testing in our classes and thus might contain some undetected mistakes.



(a) (b)

Figure 1: Shortest Grid Path versus Shortest Any-Angle Path

1 Project: Any-Angle Path Planning for Computer Games

Grids with blocked and unblocked cells are often used to represent terrain in computer games and robotics. A* finds paths on 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 runtimes, without constraining the paths to grid edges, to find **any-angle paths**.

We study path-planning problems where a two-dimensional continuous terrain is 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. Blocked cells remain blocked and unblocked cells remain unblocked. We define vertices to be the corner points (rather than the center points) of the cells. Agents are points that can move along (unblocked) paths. (Unblocked) 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 (unblocked) paths can also pass through vertices where diagonally touching blocked cells meet. The objective of path planning is to find a short and realistic looking (unblocked) path from a given (unblocked) start vertex to a given goal vertex. In this project, all search algorithms search from the start vertex to the goal vertex.

Figure 1 gives 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 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, no unblocked path can move from vertex B4 to vertex C4 in a straight line or from vertex A4 to C4 in a straight line since unblocked paths cannot move between two adjacent blocked cells. No unblocked path can move from vertex A2 to vertex A3 in a straight line or from vertex A1 to vertex A4 in a straight line since unblocked paths cannot move between two adjacent blocked cells and the grid is surrounded by blocked cells (not shown in the figure). Finally, no unblocked path can move from vertex C3 to vertex B4 in a straight line, from vertex C2 to vertex B4 in a straight line or from vertex C1

```
Main()
    $g(s_{start}) := 0$;
    $parent(s_{start}) := s_{start}$;
    $open := \emptyset$;
    $open.Insert(s_{start}, g(s_{start}) + h(s_{start}))$;
    $closed := \emptyset$;
    while open $\neq \emptyset$ do
        $s := open.Pop()$;
        if $s = s_{\text{goal}}$ then
            return "path found";
        $closed := closed \cup \{s\}$;
        foreach $s' \in$ succ($s$) do
            if $s' \notin$ closed then
                if $s' \notin$ open then
                    $g(s') := \infty$;
                    $parent(s') := NULL$;
                UpdateVertex($s, s'$);
    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 $s' \in$ open then
            $open.Remove(s')$;
        $open.Insert(s', g(s') + h(s'))$;
end
```

Algorithm 1: A*

to vertex B5 in a straight line since unblocked paths cannot pass through blocked cells.

2 A*

The pseudocode 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 on an eight-neighbor grid with the property 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 $s \in S$. First, the g-value $g(s)$ is the length of the shortest path found so far from the start vertex to vertex s and thus an estimate of its start distance (= the distance from the start vertex to vertex s). Second, the parent $parent(s)$ is used to identify a path from the start vertex to the goal vertex after A* terminates. We explain below how A* does that.

A* also maintains two global data structures: First, the open list is a priority queue that contains the vertices that A* considers to expand. A vertex that is or was in the open list is called generated. We explain below what it means to generate or expand a vertex. Procedure *open.Insert*(s, x) inserts vertex s with key x into the priority queue *open*, *open.Remove*(s) removes vertex s from the priority queue *open* and *open.Pop*() removes a vertex with the smallest key from priority queue *open* and returns it. 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.

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 open and closed lists to the empty lists and then inserts the start vertex into the open list with its f-value as its priority [4-6]. A* then repeatedly executes the following statements: If the open 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 open 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 pseudocode]. Otherwise, A* removes the vertex from the open 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 21]. 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 open list with its f-value as its priority or, if it was there already, changes its priority [Lines 22-26]. 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 $g(s) + c(s, s')$ [Line 39]. 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. H-values are consistent 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 breadth-first search.

3 Example Trace of A*

Figure 2 shows a trace of A* with the h-values

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

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

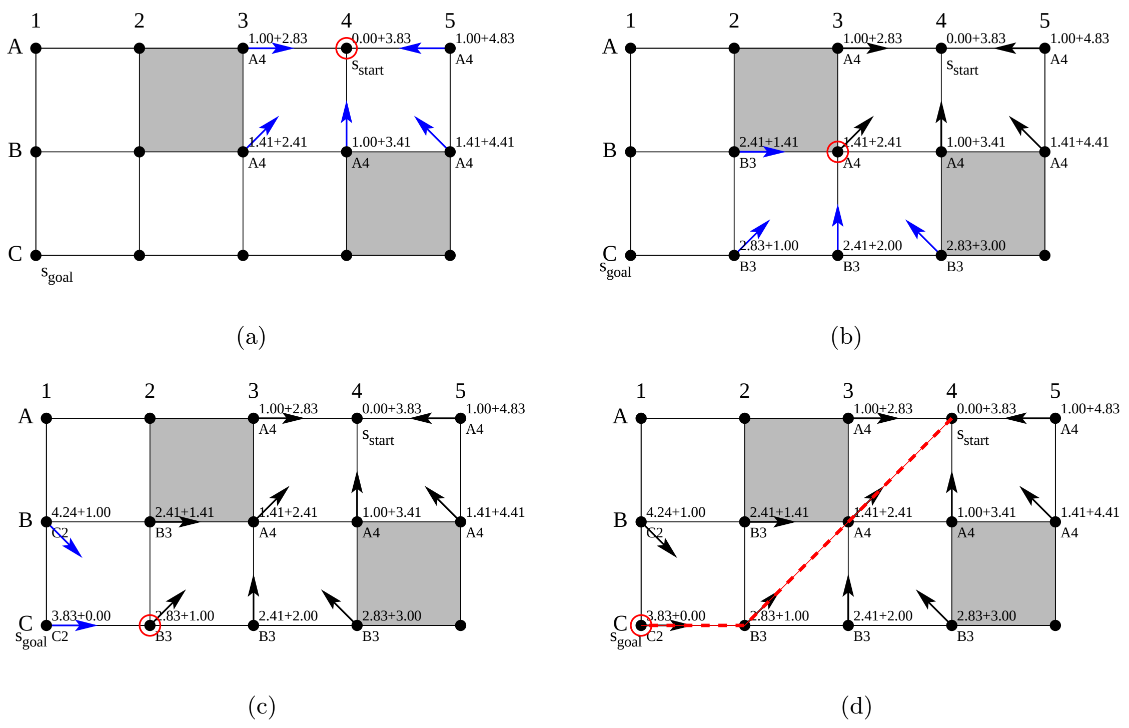
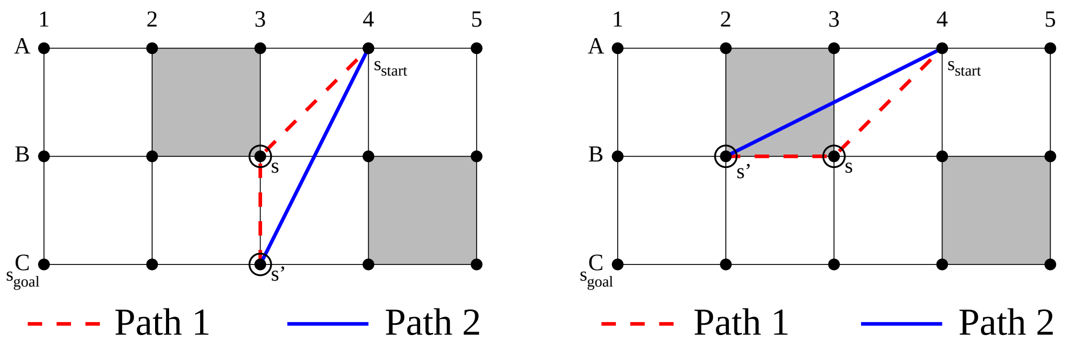


Figure 2: Example Trace of A*

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

Algorithm 2: Theta*

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.



(a) Path 2 is Unblocked

(b) Path 2 is Blocked

Figure 3: Paths Considered by Theta*

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 A* and Theta* is that the parent of a vertex can be any vertex when using Theta*, while the parent of a vertex has to be a successor of the vertex when using A*. Theta* was first described in Nash, Daniel, Koenig and Felner, Theta*: Any-Angle Path Planning on Grids, Proceedings of the AAAI Conference on Artificial Intelligence, 1177-1183, 2007.

Algorithm 2 shows the pseudocode of Theta*, as an extension of the pseudocode of A* in Algorithm 1. Procedure Main is identical to that of A* and thus not shown. Theta* is identical to A* except that it 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' of vertex s in procedure UpdateVertex. In Figure 3, 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:

- **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 39]. 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 3(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 31]. 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 3(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 3(a) gives an example. Otherwise, Theta* chooses Path 1 over Path 2. Figure 3(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 [$= g(s')$].

```
46 LineOfSight(s, s')
47     $x_0 := s.x$;
48     $y_0 := s.y$;
49     $x_1 := s'.x$;
50     $y_1 := s'.y$;
51     $f := 0$;
52     $d_y := y_1 - y_0$;
53     $d_x := x_1 - x_0$;
54     if $d_y < 0$ then
55         $d_y := -d_y$;
56         $s_y := -1$;
57     else
58         $s_y := 1$;
59     if $d_x < 0$ then
60         $d_x := -d_x$;
61         $s_x := -1$;
62     else
63         $s_x := 1$;
64     if $d_x \geq d_y$ then
65         while $x_0 \neq x_1$ do
66             $f := f + d_y$;
67             if $f \geq d_x$ then
68                 if $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
69                     return $false$;
70                 $y_0 := y_0 + s_y$;
71                 $f := f - d_x$;
72             if $f \neq 0$ $AND$ $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
73                 return $false$;
74             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
75                 return $false$;
76             $x_0 := x_0 + s_x$;
77     else
78         while $y_0 \neq y_1$ do
79             $f := f + d_x$;
80             if $f \geq d_y$ then
81                 if $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
82                     return $false$;
83                 $x_0 := x_0 + s_x$;
84                 $f := f - d_y$;
85             if $f \neq 0$ $AND$ $grid[x_0 + ((s_x - 1)/2), y_0 + ((s_y - 1)/2)]$ then
86                 return $false$;
87             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
88                 return $false$;
89             $y_0 := y_0 + s_y$;
90     return $true$;
91 end
```

Algorithm 3: Adaptation of the Bresenham Line-Drawing Algorithm

LineOfSight$(parent(s), s')$ on Line 31 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 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 pseudocode of such a method, an adaptation of the Bresenham line-drawing algorithm, which is described in Bre-

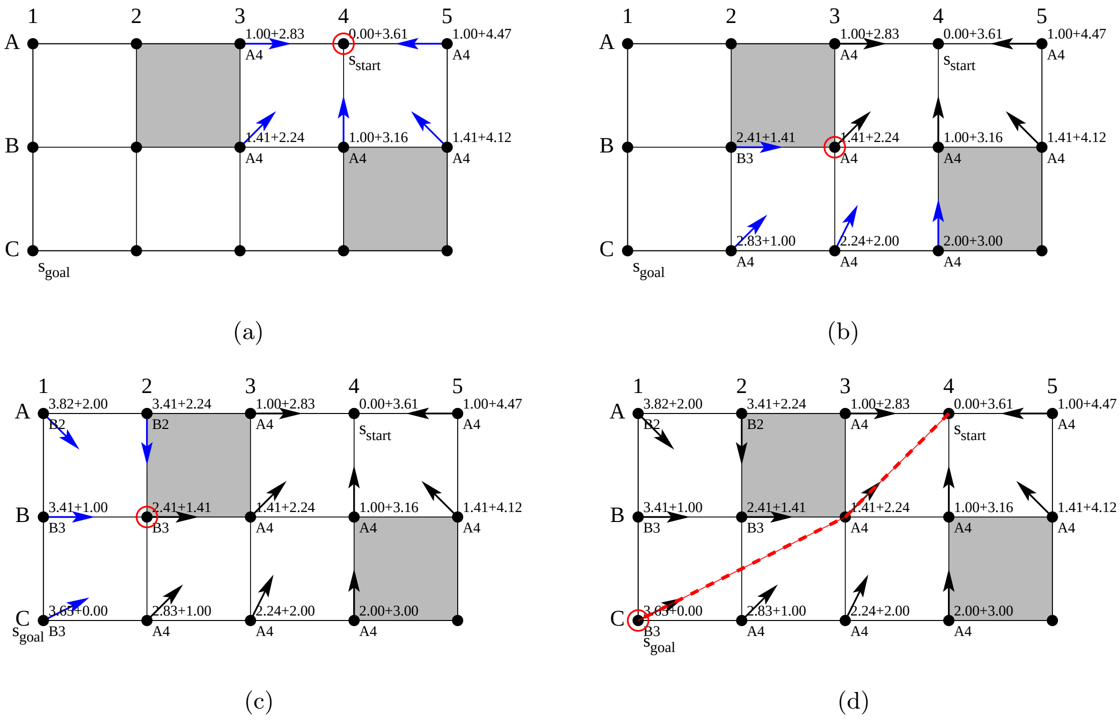


Figure 4: Example Trace of Theta*

senham, Algorithm for computer control of a digital plotter, IBM Systems Journal, Vol. 4, No. 1, 25–30, 1965 and most graphics textbooks. It is exactly the Bresenham line-drawing algorithm except that it checks whether a cell is blocked whenever the Bresenham line-drawing algorithm would plot the corresponding point and handles horizontal and vertical lines differently. $s.x$ and $s.y$ denote the x- and y-coordinates of vertex s, respectively. The array $grid$ in the pseudocode represents the grid, and $grid[x, y]$ 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 4 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 2. First, Theta* expands start vertex A4, as shown in Figure 4(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 4(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 4(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 4(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 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, that 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. We give you this extra credit to allow you to make connections to other classes and experience first hand how helpful the algorithms and data structure class that you once took can be. You can read up on binary heaps, for example, in Cormen, Leiserson and Rivest, Introduction to Algorithms, MIT Press, 2001.

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, and make sure that A* with consistent h-values finds shortest grid paths.

Your implementations should be efficient in terms of processor cycles and memory usage since 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 pseudocode 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

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 (unrealistically) that paths can pass through vertices where diagonally touching blocked cells meet. All search algorithms search from the start vertex to the goal vertex. 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

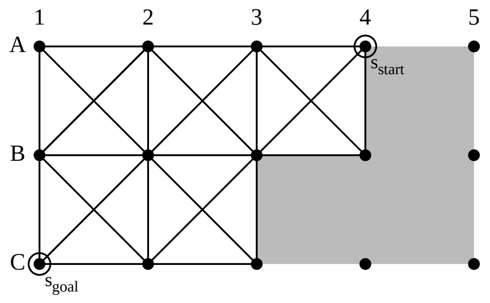


Figure 5: Example Search Problem

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).

Question 0: Easy

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. Show a shortest grid path and a shortest any-angle path for the example search problem from Figure 5. 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 by hand and submit figures similar to Figures 2 and 4.

* Question 1: Moderately Difficult

A* needs to break ties to decide which vertex to expand next if several vertices have the same smallest f-value. It can either break ties in favor of vertices with smaller g-values (version 1) or in favor of vertices with larger g-values (version 2). Both versions break remaining ties randomly. Compare both versions of A* with the h-values from Equation 1 with respect to their numbers of vertex expansions and the lengths of the paths found. (Do not count the goal vertex as expanded.)

As part of your writeup, show all of your experimental results and explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions). For example, explain why or why not you observed a large difference in the number of vertex expansions and/or the lengths of the paths found and give the underlying reasons for these observations.

Hints: For the explanation part, consider which vertices both versions of A* expand on grids without blocked cells with start vertex A1 and goal vertex E9. Numerical precision is important since small numerical errors can make f-values unequal that should otherwise be equal, making it unimportant how ties are broken. Thus, the f-values of vertices in the binary heap that are almost equal should be considered equal. Explain how your implementation achieves this objective.

Question 2: Moderately Difficult

H-values are admissible iff they do not overestimate the goal distances. Note that the goal distances for an A* search can be different from the goal distances for a Theta* search. For example, the goal distance of the start vertex in Figure 5 is about 3.83 for A* but only about 3.61 for Theta*. Thus, the answer whether h-values are admissible for A* and Theta* could be different. H-values are consistent iff 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 $s' \in succ(s)$. Thus, the answer whether h-values are consistent for A* and Theta* have to be the same. Prove or disprove:

1. The h-values $h(s) = c(s, s_{goal})$ are admissible for an A* search on an eight-neighbor grid.
2. The h-values from Equation 1 are admissible for an A* search on an eight-neighbor grid.
3. The h-values $h(s) = c(s, s_{goal})$ are admissible for a Theta* search on an eight-neighbor grid.
4. The h-values from Equation 1 are admissible for a Theta* search on an eight-neighbor grid.
5. The h-values $h(s) = c(s, s_{goal})$ are consistent for an A* and a Theta* search on an eight-neighbor grid.
6. The h-values from Equation 1 are consistent for an A* and a Theta* search on an eight-neighbor grid.

Be cautious. For example, textbooks sometimes state in the context of an A* search that consistent h-values are guaranteed to be admissible but they only discuss A* search, not Theta* search. Thus, you need to think about whether this property continues to hold in the context of a Theta* search.

* Question 3: Moderately Difficult

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

Question 4: Very Difficult

Give a proof (= concise but rigorous argument) why Theta* is guaranteed to find an unblocked (any-angle) path if it finds a path. Then, give a proof why Theta* is guaranteed to find an unblocked any-angle path if one exists. (The first proof is easy. The second one is more difficult.)

Question 5: Moderately Difficult

A* with consistent h-values guarantees that the sequence of f-values of the expanded vertices is monotonically nondecreasing. 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 2 and 4.

Question 5: Difficult

A* with consistent h-values guarantees that the sequence of f-values of the expanded vertices is monotonically nondecreasing. In other words, $f(s) \leq f(s')$ if A* expands vertex s before vertex s'. Explain the significance of this property in the context of A*. Theta* does not have this property. What effects might this have on the paths found by Theta*? Explain your reasoning in detail.

Question 6: Difficult

Read the following explanation of why Theta* uses a closed list:

> A* with consistent h-values guarantees that the sequence of f-values of the expanded vertices is monotonically nondecreasing. In other words, $f(s) \leq f(s')$ if A* expands vertex s before vertex s'. As a consequence, A* cannot find a shorter path from the start vertex to a vertex once it has expanded the vertex and thus does not need a closed list to guarantee that it expands every vertex at most once. Theta* with consistent h-values does not have these properties. As a consequence, Theta* can find a shorter path from the start vertex to a vertex after it has expanded the vertex.

The closed list prevents Theta* from re-expanding vertices that it has expanded already. Change your implementation of Theta* so that it can re-expand vertices that it has expanded already (but, in this case, does not re-initialize their g-values or parents). Compare the original and modified versions of Theta*, both with the h-values $h(s) = c(s, s_{goal})$, with respect to their runtimes (or, equivalently, numbers of expanded vertices) and the resulting path lengths. As part of your writeup, describe how you modified the pseudocode of Theta*, show all of your experimental results and explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions). In case you find that your modified version of Theta* re-expands only a small number of vertices, explain why this is so and which changes you can make to increase that number and find even shorter paths.

* Question 7: Moderately Difficult

Give four example grids with different configurations of blocked cells that all show that Theta* with the h-values $h(s) = c(s, s_{goal})$ is not guaranteed to find shortest any-angle paths. Also, give four example grids with different configurations of blocked cells that all show that Theta* with the h-values $h(s) = 0$ is not guaranteed to find shortest any-angle paths. For the second part, reuse the configurations that you used in the first part, if possible.

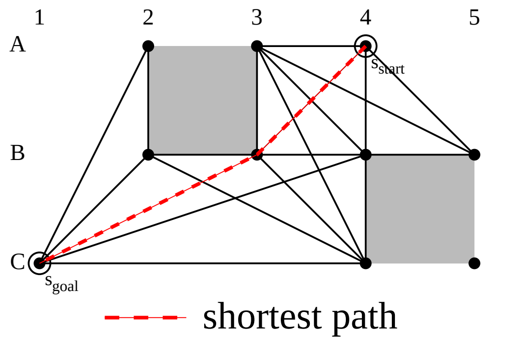


Figure 6: Visibility Graph

Question 8: Moderately Difficult

Determine experimentally the numbers of vertex expansions and the lengths of the paths found for the following eight test scenarios: Test both A* and Theta* with the h-values $h(s) = c(s, s_{goal})$, the h-values $h(s) = c(s, s_{goal})$ multiplied by 0.5 (that is, $h(s) = 0.5c(s, s_{goal})$), the h-values from Equation 1 and h-values from Equation 1 multiplied by 0.5. Assume that 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. (One of the eight scenarios is identical to part of your results from Question 1. Obviously, the experiments do not need to be redone for this case.)

As part of your writeup, show all of your experimental results and explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions). For example, explain why or why not you observed a large difference in the number of vertex expansions and/or the lengths of the paths found and give the underlying reasons for these observations. Also explain in which of these scenarios one is guaranteed to find shortest paths and why.

Question 9: Moderately Difficult

Determine experimentally the numbers of vertex expansions and the lengths of the paths found for the following five test scenarios: Test Theta* with the h-values $h(s) = 0.00 \cdot c(s, s_{goal}) = 0.00$, $h(s) = 0.25 \cdot c(s, s_{goal})$, $h(s) = 0.50 \cdot c(s, s_{goal})$, $h(s) = 0.75 \cdot c(s, s_{goal})$ and $h(s) = 1.00 \cdot c(s, s_{goal}) = c(s, s_{goal})$. Assume that Theta* breaks ties among vertices with the same f-value in favor of vertices with larger g-values. (This question is similar in parts to Question 8. Obviously, the experiments do not need to be redone for the common cases.)

As part of your writeup, show all of your experimental results and explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions). For example, explain why or why not you observed a large difference in the number of vertex expansions and/or the lengths of the paths found and give the underlying reasons for these observations.

* Question 10: Easy

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 visibility graphs. 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 6, 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 a shortest any-angle path. Show all of your experimental results.

* Question 11: Easy

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. As part of your writeup, show all of 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.

Question 12: Very Difficult

Field D* is an any-angle path planning algorithm different from Theta*. Read up on Field D* in D. Ferguson and A. Stentz, Using Interpolation to Improve Path Planning: The Field D* Algorithm, Journal of Field Robotics, Vol. 23, No. 2, 2006, pp. 79-101. Compare Field D* and Theta*, both with the h-values $h(s) = c(s, s_{goal})$, with respect to their runtimes and the resulting path lengths. As part of your writeup, show all of your experimental results, explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions), speculate on how your experimental results are likely going to affected in case the percentage of blocked cells is changed and in case different unblocked cells have different traversal costs, as assumed in the Field D* paper, and argue why your speculations are reasonable.

* Question 13: Moderately Difficult

Assume that Theta* expands vertex s and that vertex s' is a successor of vertex s. Theta* considers not only the path from the start vertex to vertex s and from vertex s to vertex s' in a straight line (Path 1) but also the path from the start vertex to the parent of vertex s and from the parent of vertex s to vertex s' in a straight line (Path 2). Change your implementation of Theta* so that it also considers the path from the start vertex to the grand parent of vertex s and from the grand parent of vertex s to vertex s' in a straight line, the path from the start vertex to the great grand parent of vertex s and from the great grand parent of vertex s to vertex s', and so on. Compare the original and modified versions of Theta*, both with the h-values $h(s) = c(s, s_{goal})$, with respect to their runtimes and the resulting path lengths.

As part of your writeup, describe how you modified the pseudocode of Theta*, show all of your experimental results, explain them in detail (including your observations, detailed explanations of the observations and your overall conclusions) and explain both whether the original version of Theta* can find shorter paths than the modified version of Theta* (if so, give an example grid) and why the path lengths of the modified version of Theta* are not much smaller than the ones of the original version of Theta* and are often identical even though the modified version of Theta* checks many more paths (possibly resulting in shortcuts) than the original version of Theta*.

* Question 14: Moderately Difficult

We assumed that paths can pass through vertices where diagonally touching blocked cells meet. Describe all changes that need to be made to Theta* to drop this unrealistic assumption. (In this case, for example, it is no longer true that a straight line is unblocked iff none of the points plotted by the Bresenham line-drawing algorithm correspond to blocked cells.) Implement your changes and make sure that the resulting version of Theta* indeed finds short any-angle paths that do not pass through such vertices.

Extra Credit 1: Moderately Difficult

We used A* as baseline with its open list implemented as binary heap. However, there are alternatives. For example, the open list of A* can be implemented with data structures other than a binary heap, for example, with buckets (an array of sets of vertices that is indexed with the f-values of the vertices). Compare both versions of A* for finding shortest grid paths with respect to their runtimes or, equivalently, numbers of expanded vertices. (Do not count the goal vertex as expanded.) The version of A* that uses a binary heap should break ties among vertices with the same f-value in favor of vertices with larger g-values. (If you want to be thorough, see the comments on the importance of numerical precision in Question 1.) The version of A* that uses buckets can break ties in a similar (but not completely identical) way by implementing the sets as last-in first-out queues.

Extra Credit 2: Difficult

Performance differences between search algorithms can be systematic in nature or only due to sampling noise (= bias exhibited by the selected test cases since the number of test cases is always limited). One can use statistical hypothesis tests to determine whether they are systematic in nature. Read up on statistical hypothesis tests (for example, in Cohen, Empirical Methods for Artificial Intelligence, MIT Press, 1995) and then describe for one of the experimental questions above exactly how a statistical hypothesis test could be performed and what its outcome is for your experimental data.

Extra Credit 3: Open Ended

Theta* is one particular modification of A* that allows it to find short any-angle paths. Think of other modifications of A* that achieve the same objective, implement them and compare them experimentally against A*.