Intro to AI 440, Assignment 1

Finn Jensen faj21, Jane McGrath ljm209, Andres Uribe aju24

Feb 09 2022

Question 1

a)
For this project, the python tkinter library was used in the visualization, called in the files "draw.py" and "griddisplay.py". Additionally, the methods "readin.py", "Vertex.py", and "Graph.py" were written in order to create the underlying vertices and graphs in this project.

Eight neighbor grids are represented by a model that uses a Graph class and a Vertex class.

The Vertex class represents nodes on the eight neighbor grid and stores their g, h, and f-values. Vertex objects are identified by a unique key in the form "$x|y$". The class also maintains a neighbors dictionary with neighboring vertices as the key and that edges weight as the value.

The graph class maintains a list of all of the participating vertices, the starting and goal vertices and a dictionary that keeps track of which cells are blocked and unblocked.

These grids interact with the front end visualization (which consists of "griddisplay.py" and "draw.py") by providing a Graph object that represents the eight neighbor grid with a start and goal node. The path is generated later by the A* or Theta* algorithm class.

The file "griddisplay.py" contains a class "Display". A Display object is created with a graph, an integer "scale", and an integer "radius". The "scale" integer takes care of the visibility of the displayed grid, and the "radius" integer determines the size of the average vertex. The Display object will have its own Tkinter root window, a frame, a canvas, two scrollbars (one for horizontal scrolling and one for vertical scrolling), and it will also configure all these features (the window, the canvas, the scrollbars). "Display" also contains the methods "coordinate_producer", "display_info", and "bind_canvas".

The "coordinate_producer" method only takes a Display object. First, it obtains the start and goal vertices with the graph methods "start_node_key" and "goal_node_key", scales them up according to the Display object's "scale" attribute, and then calls the "points" method (from "draw.py") to draw them with a larger radius in the color "red". The "coordinate_producer" method then iterates over each vertex in the graph, and then over each neighbor of that vertex. It obtains the coordinates

of every vertex using the Vertex method "getKeyCoordinates", and then calls "points_and_lines" (from "draw.py"). Each vertex will have the tag "vertex". Once every vertex and its neighbors have been drawn, this method is finished.

Then, the "bind_canvas" method will use tkinter's "tag_bind" method to bind all objects that are tagged with "vertex" to the event of a mouse click, and to the "display_info" method. When a vertex (tagged "vertex") is clicked on, "display_info" is called. It obtains the x and y coordinates of the mouse's position, and then iterates over the vertices in the Display object's graph. The vertices that have been drawn on the canvas have a certain radius determined by the Display object's vertex_radius; they have also been scaled up according to the Display object's scale attribute. So, for each vertex in the graph, this method checks if the mouse-click-coordinates are within a range of "vertex_radius" of that original vertex's coordinates. It obtains the original vertex coordinates through "getKeyCoordinates", of course. It scales the value of the original vertex by the Display object's attribute "scale" to do so. The exact coordinates which correspond to a vertex are very difficult to hit, so this allows the entire drawn region that a vertex occupies to be clicked on instead. If the coordinates are within range, this method then obtains the g-, h-, and f-values of the vertex, and prints them to the terminal, along with the vertex's coordinates.

"draw.py" contains very simple methods: "points", "lines", "draw_path", and "points_and_lines". The method "points_and_lines" is generally called first, and takes two sets of coordinates (four integers) and a Display object (defined in "griddisplay.py") as its parameters. With these parameters, it calls the "lines" and "points" methods. The "lines" method simply uses the "create_line" method from tkinter, and colors it with the passed fill color. It obtains the Display object's canvas to create this line, with the passed coordinates.

Unlike the other methods, the "points" method needs a specific canvas and radius passed, along with an x coordinate, a y coordinate, a string for a tag, and a string for a fill color. With the radius provided, "points" calculates the appropriate integers to pass to tkinter's "create_oval" method, which will then fill the circle created with the provided fill, and tag it with the provided tag, on the provided canvas.

The "points" method is called by the Display class once specifically for the start and goal vertices, which must be larger than the average vertex. The need to make these vertices larger is the reason why the "points" method currently does not take a Display object as a parameter, and instead requires a specific canvas and a specific radius.

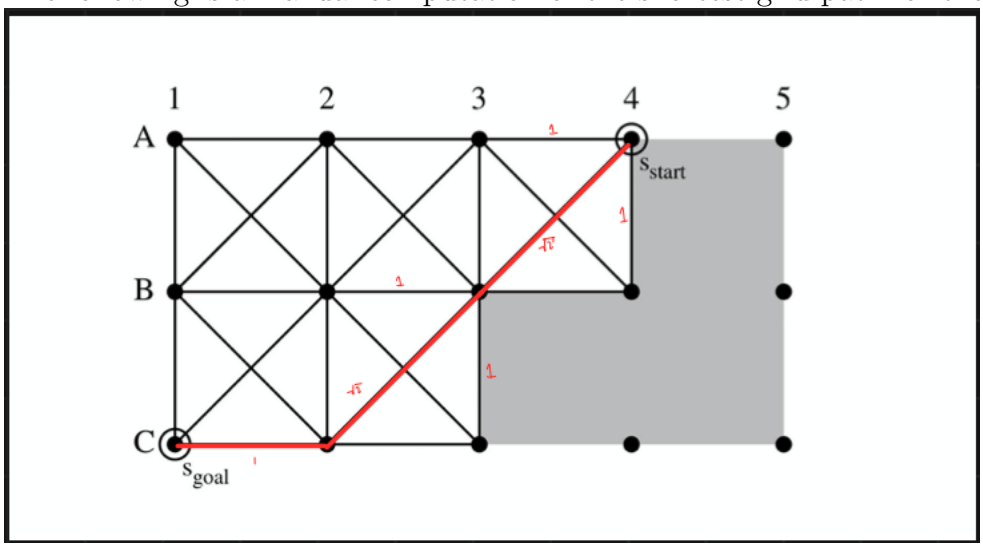
These three methods, "points_and_lines", "points", and "lines", are all in charge of drawing the vertices and edges of the graph. The method "draw_path", on the other hand, can be called with the path that the A* family algorithms return. It takes a Display object, the path which an algorithm returns, and some fill color. The "draw_path" method iterates over each pair of coordinates in the path that is passed to it, which are scaled up according to the Display object's scale attribute, and then calls the "lines" method on these scaled-up coordinates.

When the appropriate graph is displayed, the "start" and "goal" vertices should appear outlined in red, and the path that an algorithm takes should also be in red. When clicking on a vertex, a user

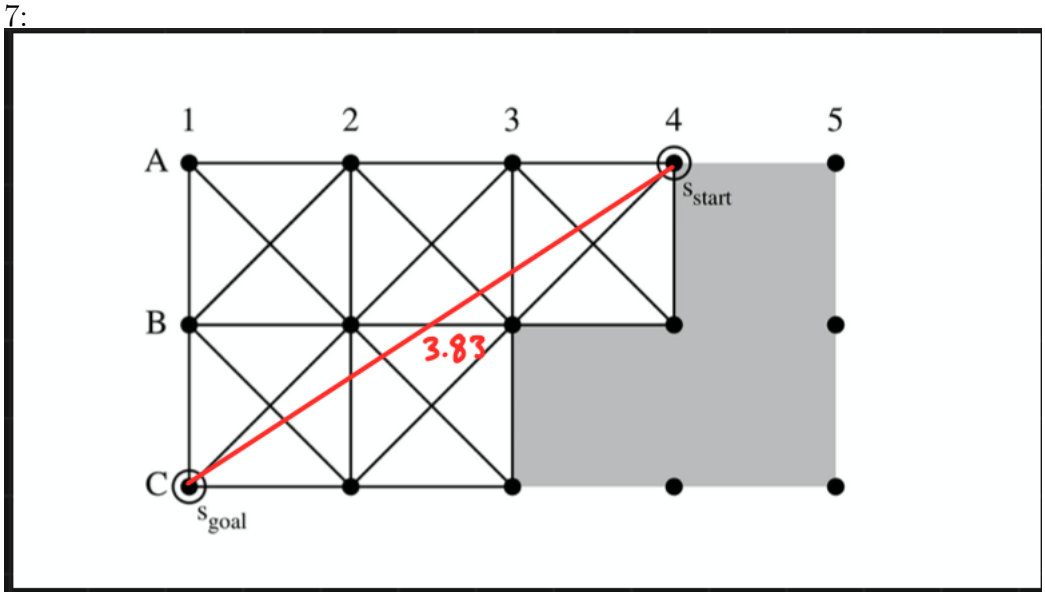
should take care to click precisely within the filled-in black part of the vertex, and not outside or on any lines. The "draw_path" method is called after the vertices have all been drawn, and so the line it draws will appear above the vertices of the graph. Such regions will not allow for "display_info" to be called appropriately, because the lines are essentially obscuring the click-able portions of the vertices.

b)

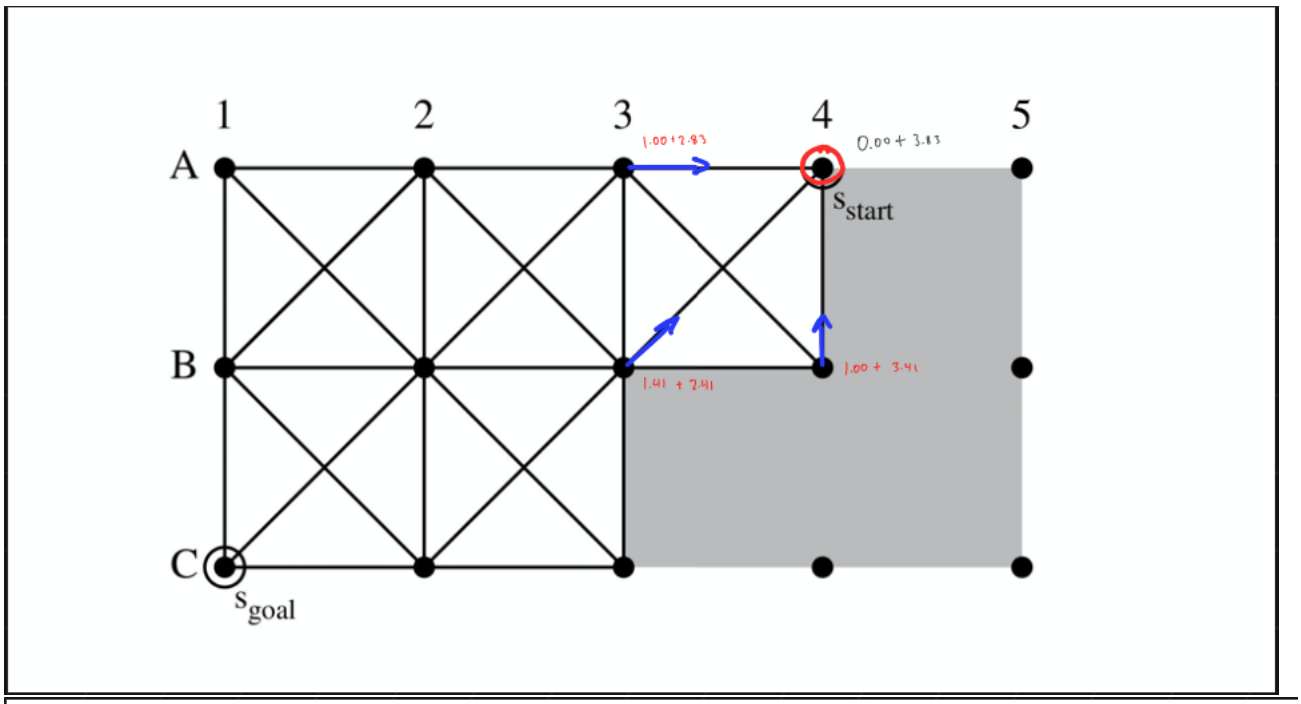
The following is a manual computation of the shortest grid path for the problem from Figure 7:

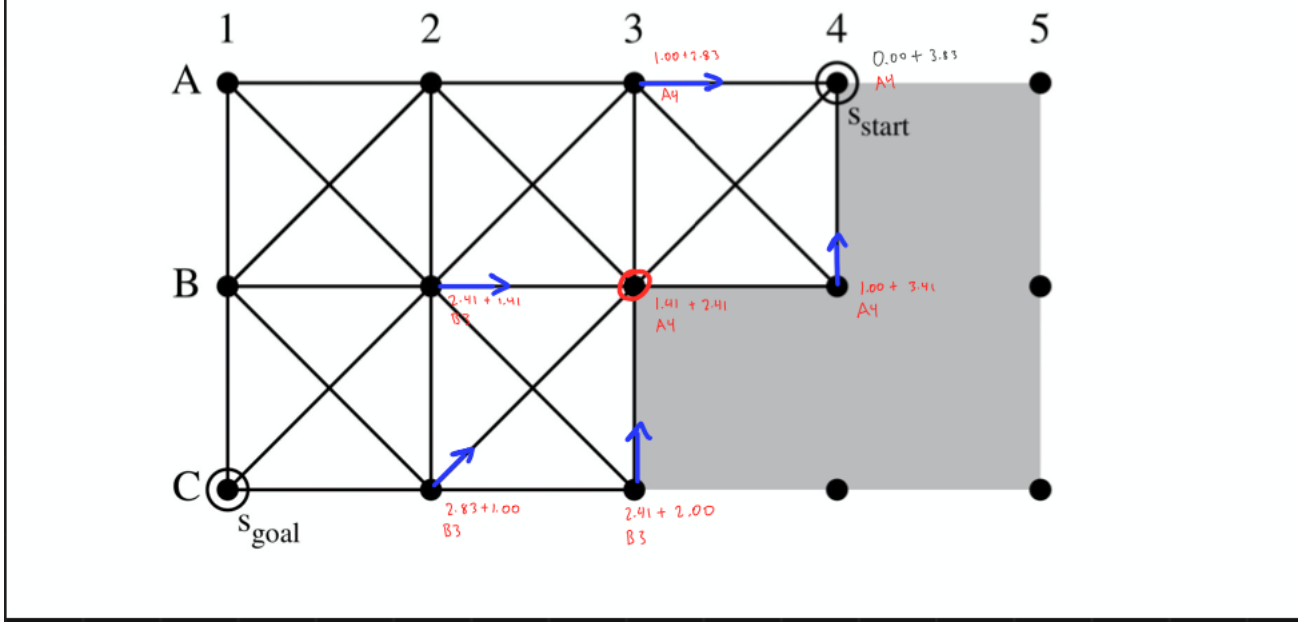


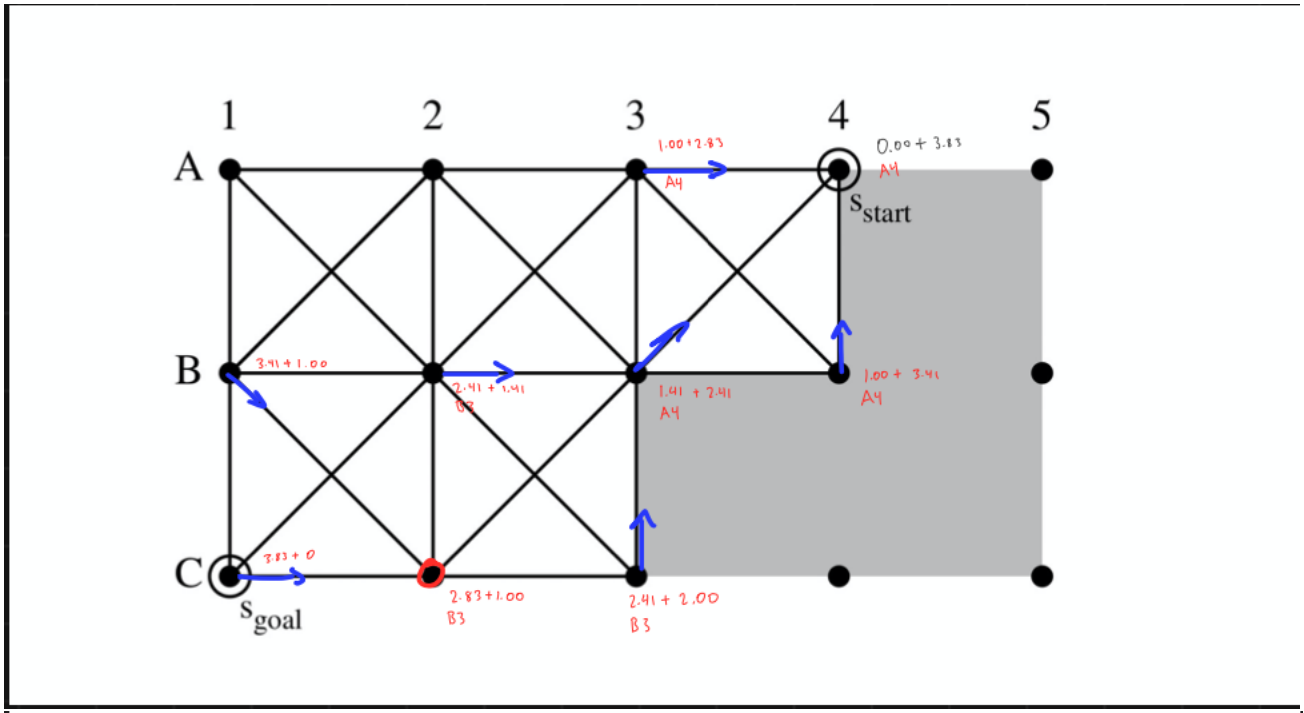
The following is a manual computation of the shortest any-angle path for the problem from Figure 7:

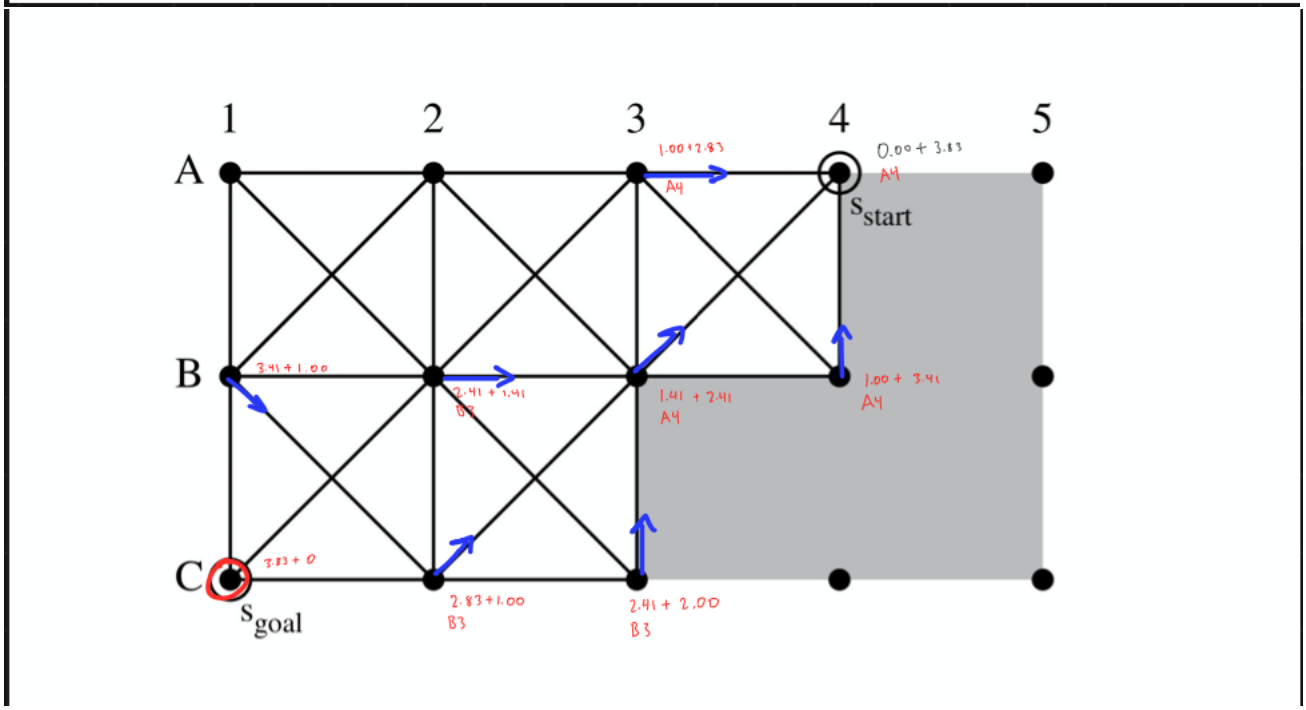


The following images depict a manually computed trace of A* with the h-values from Equation 1:

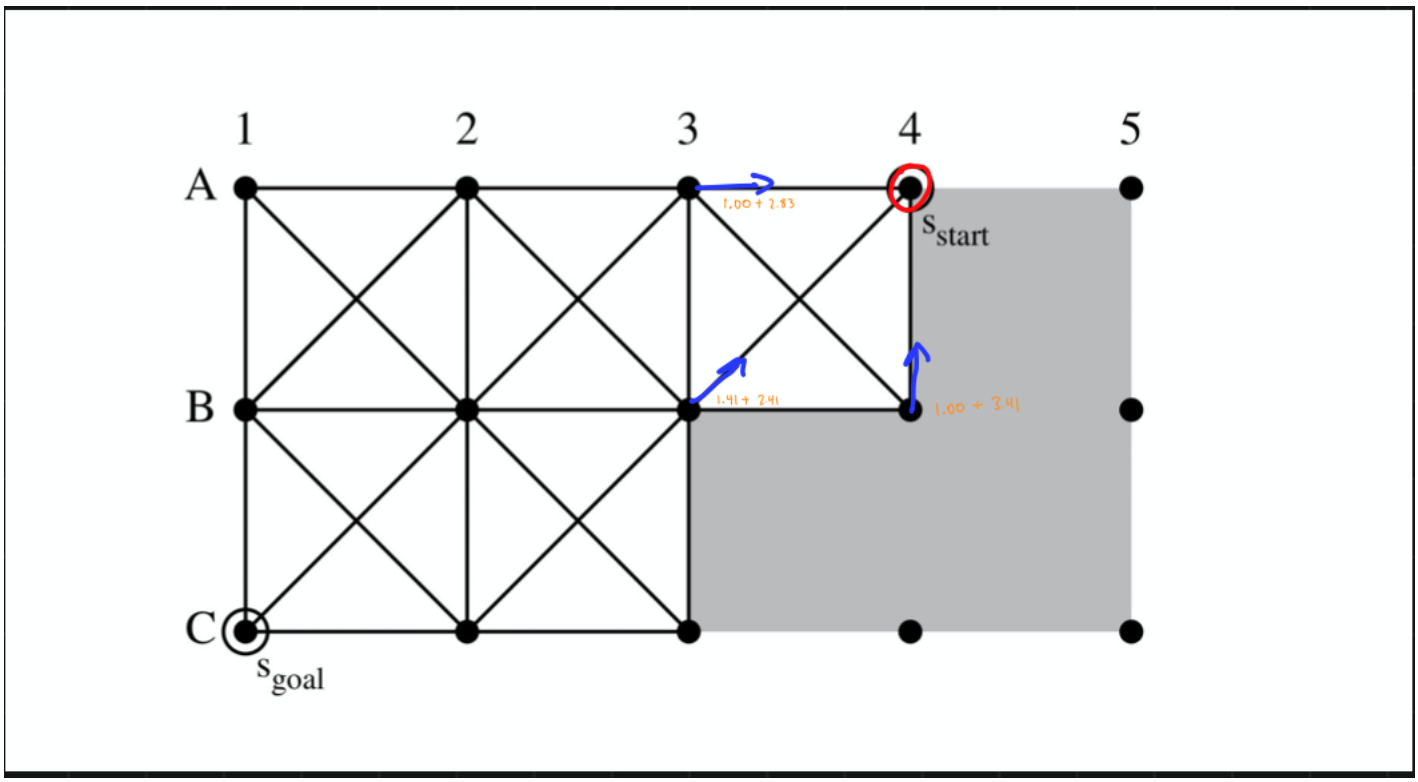

1
2
3
4
5
A
B
C
1.00+2.83
0.00+3.83
Sstart
1.00+3.41
1.41+2.41
Sgoal


1
2
3
4
5
A
B
C
1.00+2.83
A4
0.00+3.83
A4
Sstart
2.41+1.41
B2
1.41+2.41
A4
1.00+3.41
A4
2.83+1.00
B3
2.41+2.00
B3
Sgoal

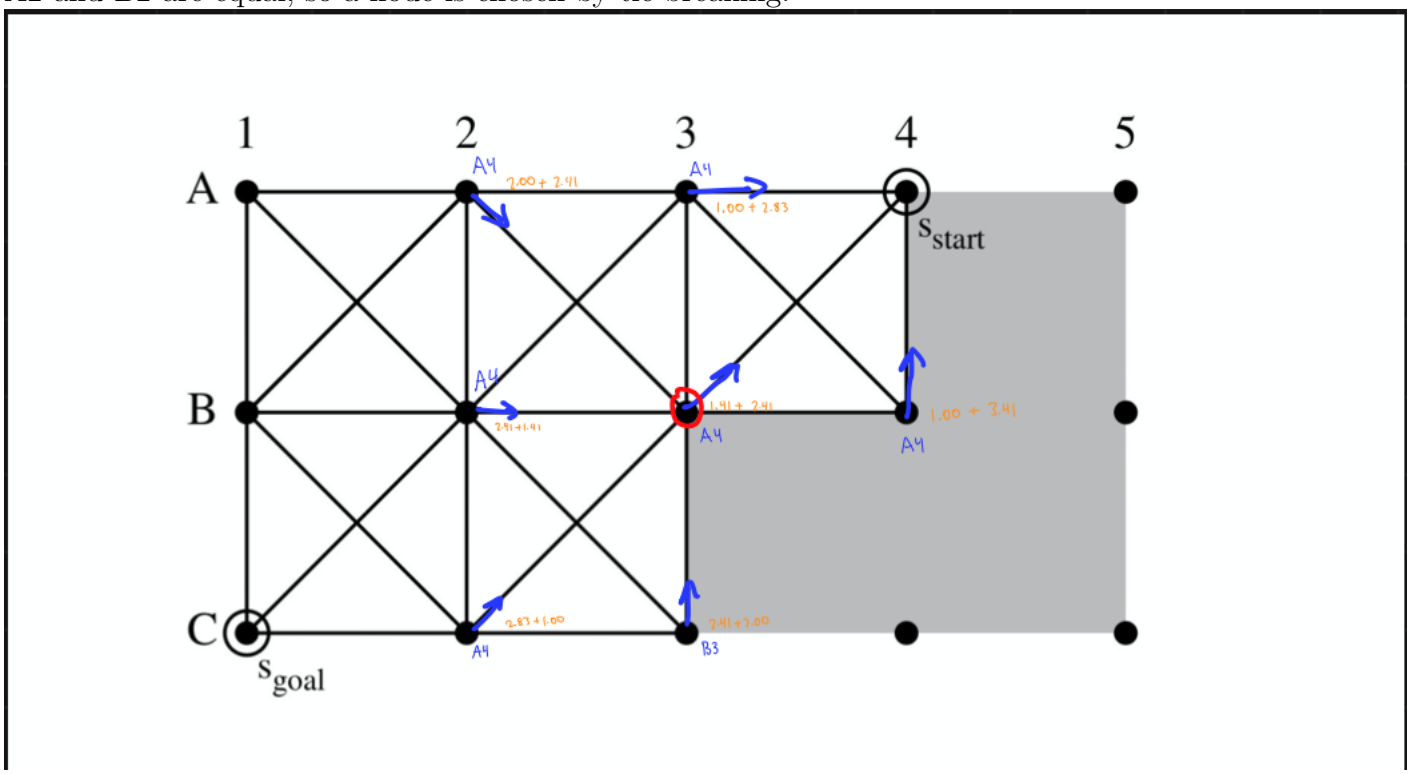


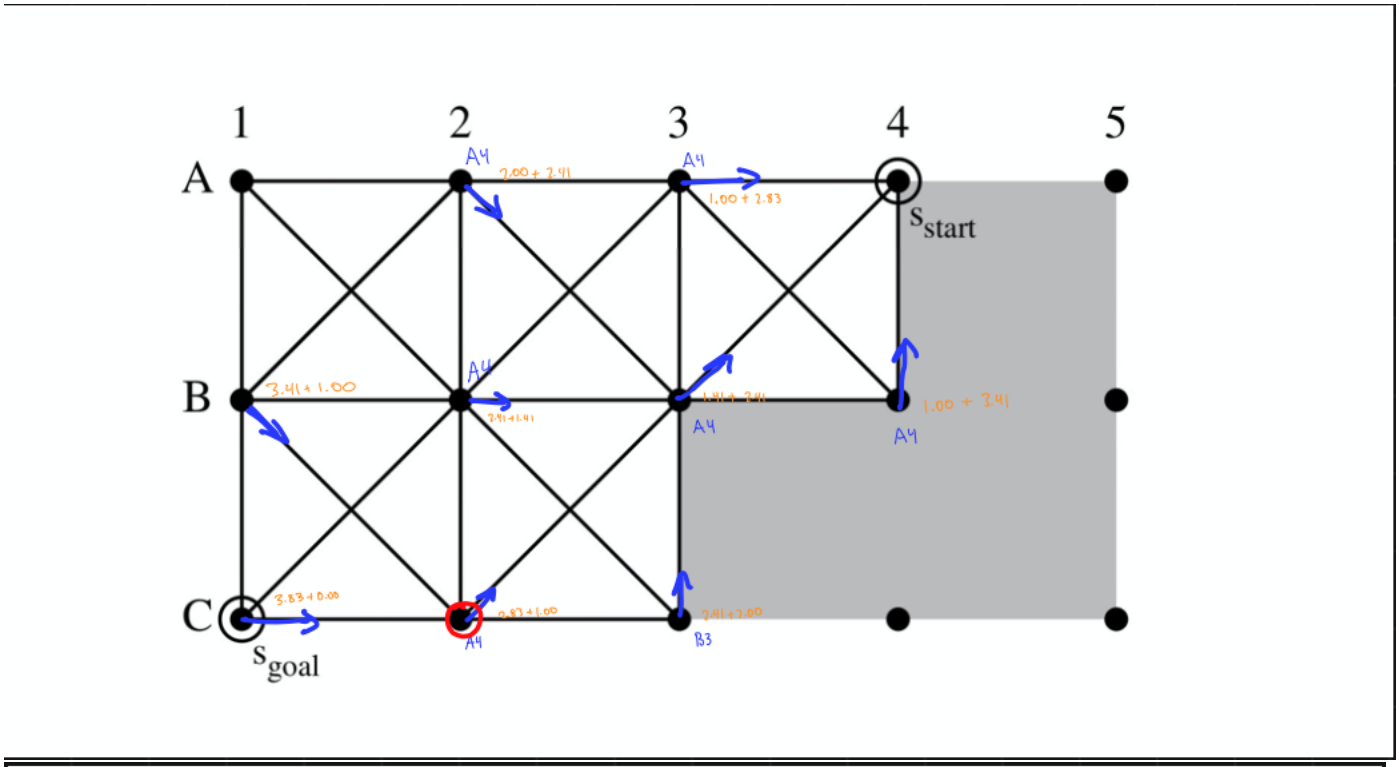


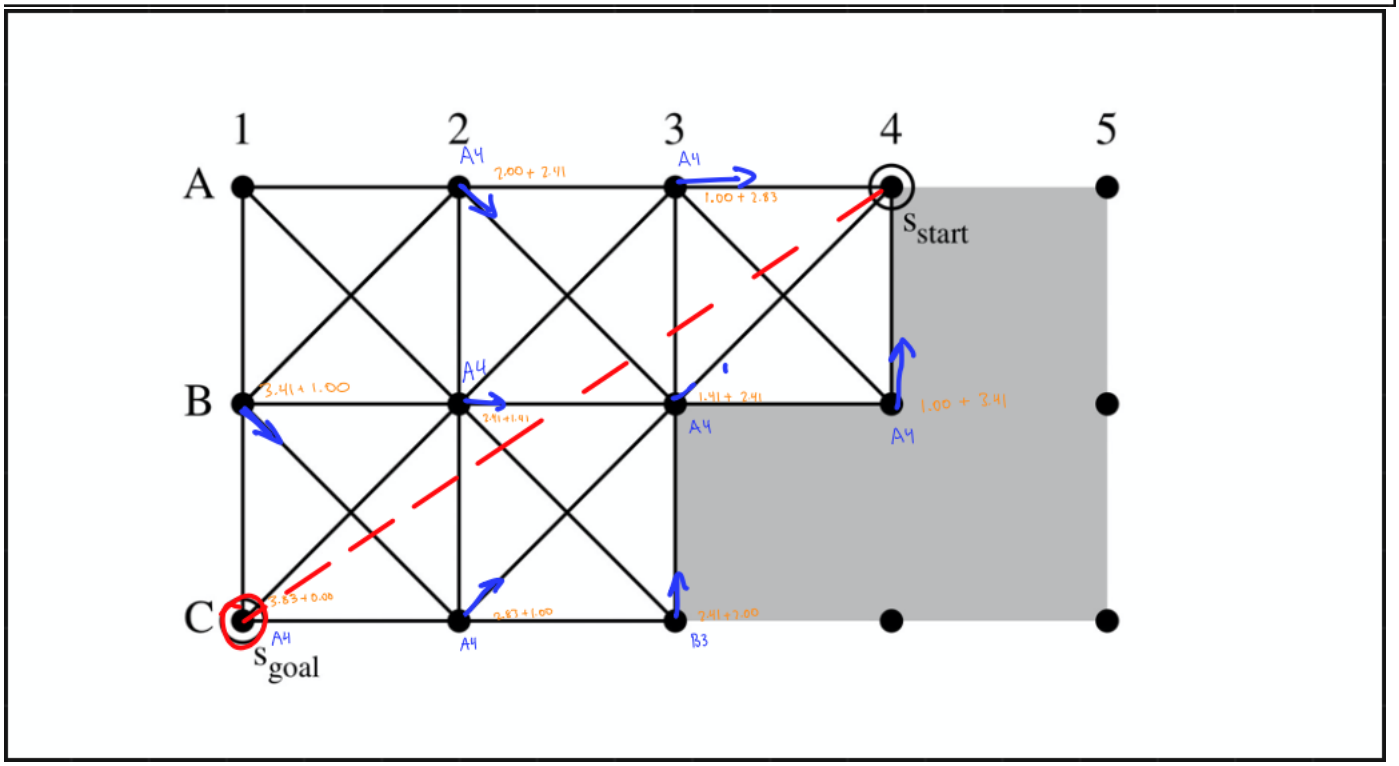
The following are a manually computed trace of Theta* with the h-values $h(s) = s(s, s_{goal})$



Note that the f-values are displayed rounded to 2 decimal places. At this step the real f-values for A2 and B2 are equal, so a node is chosen by tie-breaking.







c)
A* and Theta* are implemented as classes with a main method closely following their respective pseudocodes.

The AStar class is interacted with through its main method. It accepts a graph object as a parameter and then when the main method is run [$AStar(graph).main()$] it returns the shortest path as a list of nodes from start node to goal node. AStar implements $main$, g, h, $shortest\_path$, and $UpdateVertex$.

The AStar class also uses heapq from python in order to implement a heap which is then used throughout the class, with the fringe and closed list.

In shortest_path(end_node), given the final node after running, A* traces the path to that node through its parents, and returns a reversed list.

Note: All other methods are implemented exactly as described in the project description.

d)
In order to get Theta* working, there were some slight changes that had to be made to the A* algorithm.

The first change occurred in the updateVertex() method. On top of the path that was considered by A*, Theta* also considered the following path: start vertex to parent of vertex s, and parent of vertex s to successor vertex s'. This was achieved by computing the straight line distance between the parent of vertex s and its successor s'.

Theta* also checks if this straight line is even possible by checking whether it crosses through a blocked cell or not. This was achieved in the LineOfSight() method by calling a helper method cellsBlocked(). This method uses a dictionary where the key is a vertex and the value is 0 or 1, where 0 and 1 correspond to an unblocked and a blocked cell, respectively.

e)
The A* algorithm must be consistent in order to guarantee that it finds the shortest grid paths. Therefore,

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

must be consistent for A* to find the shortest grid paths.

For h(s) where s=goal, h(s) must be zero. This means

$$h(s) = sqrt[2] * min(0, 0) + max(0, 0) - min(0, 0)$$

Assuming h(s) is indeed consistent, $h(s) \leq c(s, s') + h(s')$ must be true.

If s' were the goal vertex, and s is a vertex one step away, it could be said that $h(s) \leq c(s, s_{goal}) + h(s_{goal})$

However, $h(s_{goal})$ is zero, so $h(s) \leq c(s, s_{goal})$

This means that for h(s), $h(s) = \sqrt{2} * min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$ is still less than the straight line distance between the current vertex and the goal vertex. This sort of heuristic ensures that the current vertex should indeed be chosen, being one step away from the goal.

s' is a vertex which is a successor to vertex s. Assume $s' = s_{goal}$, and that the heuristic is consistent.

Then it can be said that $g(s') = g(s) + c(s, a, s')$

Additionally, $f(s') = g(s') + h(s')$

It can also be said that $f(s') = g(s') + h(s') = g(s) + c(s, a, s') + h(s')$, which is all greater than $g(s) + h(s) = f(s)$

However, $h(s_{goal})$ is zero.

This means that $g(s') = g(s) + c(s, a, s')$ and that $f(s') = g(s') + 0$, so $g(s') = f(s')$

It can then be said that

$$f(s') = g(s') = g(s) + c(s, a, s') \geq g(s) = f(s)$$

$$f(s') = f(s) + c(s, a, s') \geq f(s)$$

$$f(s') = f(s) + c(s, a, s')$$

Because f(s') is a successor to s, and because f(s) is less than f(s'), this proves the values of f(s) on any path are nondecreasing.