Name: Sidhu Arakkal, Eoin O'Hare, Mohammed Awais Zubair
Date: 10/16/2022
Fall 2022

# Assignment 1 (CS 440)

Final Submission

1. Any-Angle Path Planning

   (a) For our interface, we decided to use matplotlib and matplotlib.pyplot. We represented the starting location with a blue circle on the vertex, and the goal location with an orange circle on the vertex. After computing the path, the path is represented through a red dotted line drawn between each of the vertices on the path, as shown in the figure below. The blocked cells are represented with a gray cell. The visualization is capable of representing the 50

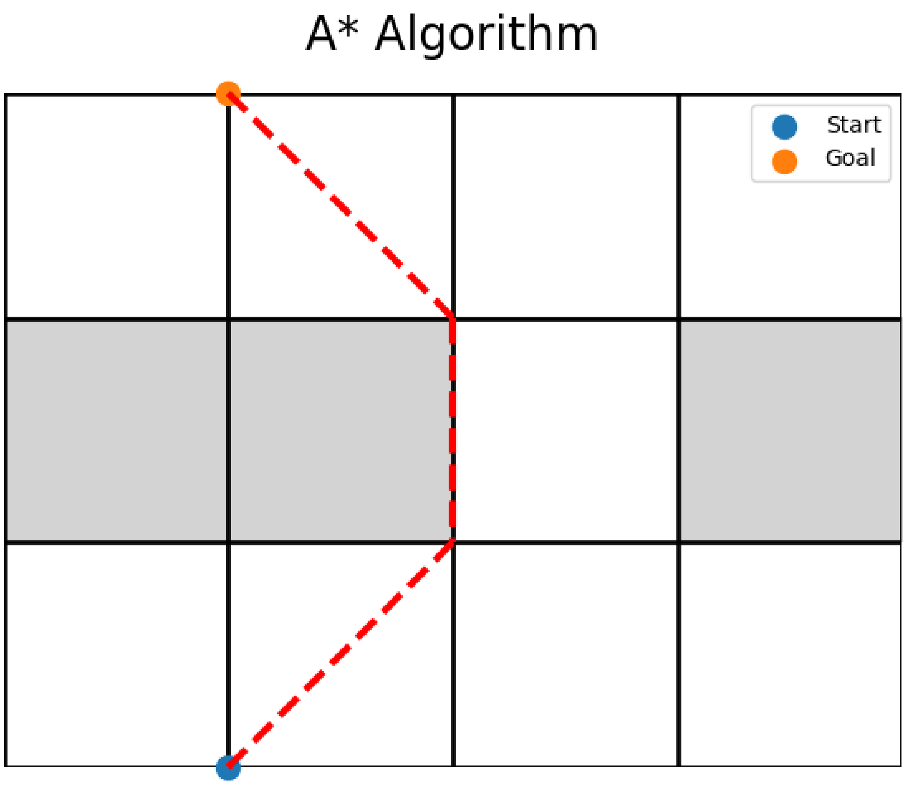
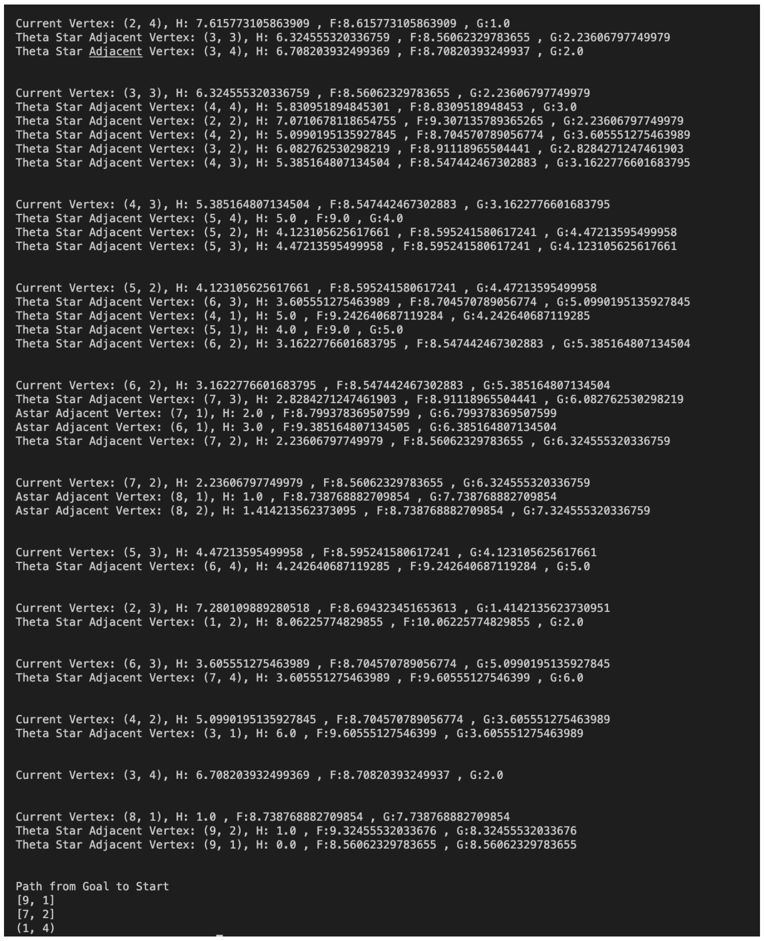


Figure 1: A* Trace Visualization

eight-neighbor grids for the experiments, and is already capable of showing the blocked cells and optimal path from the start and end goal. Currently, we are using the terminal to represent the h, f, and g values of the vertices. While computing the path, the values of the vertex after it is popped from the fringe will be printed and then all the neighboring vertices' values will be printed. We will add the capability for buttons by the full submission of the project to better represent the values of the experiments. An example of the terminal's output is shown below:



```
Current Vertex: (2, 4), H: 7.615773105863909 , F:8.615773105863909 , G:1.0
Theta Star Adjacent Vertex: (3, 3), H: 6.324555320336759 , F:8.56062329783655 , G:2.23606797749979
Theta Star Adjacent Vertex: (3, 4), H: 6.708203932499369 , F:8.70820393249937 , G:2.0


Current Vertex: (3, 3), H: 6.324555320336759 , F:8.56062329783655 , G:2.23606797749979
Theta Star Adjacent Vertex: (4, 4), H: 5.830951894845301 , F:8.8309518948453 , G:3.0
Theta Star Adjacent Vertex: (2, 2), H: 7.0710678118654755 , F:9.307135789365265 , G:2.23606797749979
Theta Star Adjacent Vertex: (4, 2), H: 5.0990195135927845 , F:8.704570789056774 , G:3.605551275463989
Theta Star Adjacent Vertex: (3, 2), H: 6.082762530298219 , F:8.91118965504441 , G:2.8284271247461903
Theta Star Adjacent Vertex: (4, 3), H: 5.385164807134504 , F:8.547442467302883 , G:3.1622776601683795


Current Vertex: (4, 3), H: 5.385164807134504 , F:8.547442467302883 , G:3.1622776601683795
Theta Star Adjacent Vertex: (5, 4), H: 5.0 , F:9.0 , G:4.0
Theta Star Adjacent Vertex: (5, 2), H: 4.123105625617661 , F:8.595241580617241 , G:4.47213595499958
Theta Star Adjacent Vertex: (5, 3), H: 4.47213595499958 , F:8.595241580617241 , G:4.123105625617661


Current Vertex: (5, 2), H: 4.123105625617661 , F:8.595241580617241 , G:4.47213595499958
Theta Star Adjacent Vertex: (6, 3), H: 3.605551275463989 , F:8.704570789056774 , G:5.0990195135927845
Theta Star Adjacent Vertex: (4, 1), H: 5.0 , F:9.242640687119284 , G:4.242640687119285
Theta Star Adjacent Vertex: (5, 1), H: 4.0 , F:9.0 , G:5.0
Theta Star Adjacent Vertex: (6, 2), H: 3.1622776601683795 , F:8.547442467302883 , G:5.385164807134504


Current Vertex: (6, 2), H: 3.1622776601683795 , F:8.547442467302883 , G:5.385164807134504
Theta Star Adjacent Vertex: (7, 3), H: 2.8284271247461903 , F:8.91118965504441 , G:6.082762530298219
Astar Adjacent Vertex: (7, 1), H: 2.0 , F:8.799378369507599 , G:6.799378369507599
Astar Adjacent Vertex: (6, 1), H: 3.0 , F:9.385164807134505 , G:6.385164807134504
Theta Star Adjacent Vertex: (7, 2), H: 2.23606797749979 , F:8.56062329783655 , G:6.324555320336759


Current Vertex: (7, 2), H: 2.23606797749979 , F:8.56062329783655 , G:6.324555320336759
Astar Adjacent Vertex: (8, 1), H: 1.0 , F:8.738768882709854 , G:7.738768882709854
Astar Adjacent Vertex: (8, 2), H: 1.414213562373095 , F:8.738768882709854 , G:7.324555320336759


Current Vertex: (5, 3), H: 4.47213595499958 , F:8.595241580617241 , G:4.123105625617661
Theta Star Adjacent Vertex: (6, 4), H: 4.242640687119285 , F:9.242640687119284 , G:5.0


Current Vertex: (2, 3), H: 7.280109889280518 , F:8.694323451653613 , G:1.4142135623730951
Theta Star Adjacent Vertex: (1, 2), H: 8.06225774829855 , F:10.06225774829855 , G:2.0


Current Vertex: (6, 3), H: 3.605551275463989 , F:8.704570789056774 , G:5.0990195135927845
Theta Star Adjacent Vertex: (7, 4), H: 3.605551275463989 , F:9.60555127546399 , G:6.0


Current Vertex: (4, 2), H: 5.0990195135927845 , F:8.704570789056774 , G:3.605551275463989
Theta Star Adjacent Vertex: (3, 1), H: 6.0 , F:9.60555127546399 , G:3.605551275463989


Current Vertex: (3, 4), H: 6.708203932499369 , F:8.70820393249937 , G:2.0


Current Vertex: (8, 1), H: 1.0 , F:8.738768882709854 , G:7.738768882709854
Theta Star Adjacent Vertex: (9, 2), H: 1.0 , F:9.32455532033676 , G:8.32455532033676
Theta Star Adjacent Vertex: (9, 1), H: 0.0 , F:8.56062329783655 , G:8.56062329783655


Path from Goal to Start
[9, 1]
[7, 2]
(1, 4)
```

Figure 2: Visualization of Values on Terminal

(b) Search Algorithms

i. *Shortest Grid Path*
The shortest grid path based on the sample problem in Figure 1 would be the following path: [A4, B3, C2, C1]. The weights of the path are: $\sqrt{2}$, $\sqrt{2}$, and 1. Thus the total distance would be $1 + 2\sqrt{2}$. Similarly, path [A4, B3, B2, C1] and [A4, A3, B2, C1] would have the same distance.

ii. *Shortest Any-Angle Path*
The shortest any-angle path based on the sample problem in Figure 1 would be the direct path from $A4 \rightarrow C1$. The distance of this path is calculated via the distance formula, which results in a distance of $\sqrt{13}$.

iii. *A* Trace*
Using the formula provided, Figure 2 shows the trace of the A* algorithm. The labels of the vertices are f-values (written as the sum of g- and h- values) and parents. We must assume that all numbers are precisely calculated even though the diagram has everything rounded to two decimal places. The arrows point to parents and red circles indicate expanded vertices. The A* algorithm follows the parents from the goal vertex, C1, to the start vertex, A4, via the dashed red path [A4, B3, C2, C1], which is one of the shortest grid paths found earlier.

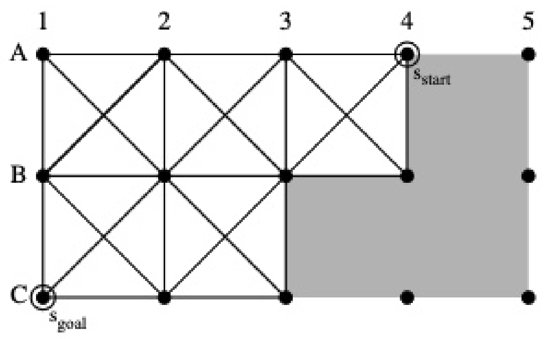


Figure 3: Sample Search Problem

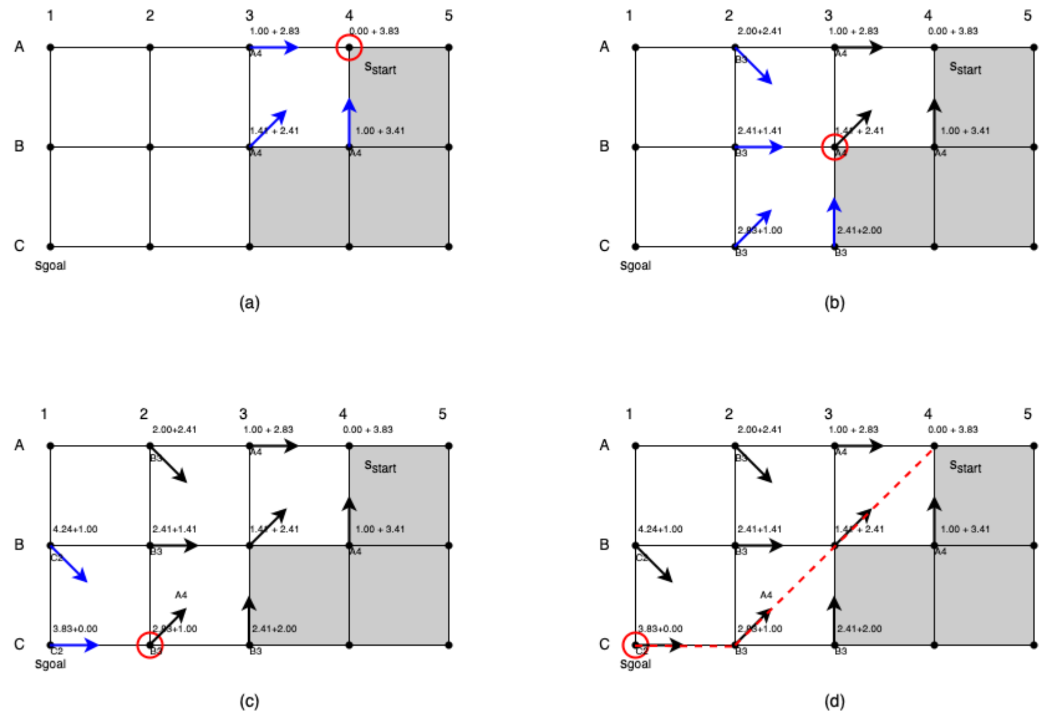


Figure 4: A* Trace

iv. *Theta* Trace*

Using the formula provided, Figure 3 shows the trace of the Theta* algorithm. Firstly, the algorithm expands vertex A4 as seen in Figure 3(a) and sets the parent of the un-expanded successors of vertex A4 to be A4. Then Theta* expands to vertex B3 with parent A4, as seen in Figure 3(b), and the only straight line that is blocked is between C3, thus its parent is updated to B3. The remaining vertexes (A2, B2, C2) have their parents remain A4 since the path is unblocked. Thirdly, the algorithm expands to vertex B2 with parent A4 and sees that all the remaining points (A1, B1, and C1) still have a line of sight to the starting point A4, thus their parent remains A4. However, since C1 is the goal node, the algorithm expands to C1 and identifies the direct dashed path from C1 to A4, which is a shortest any-angle path gotten above.

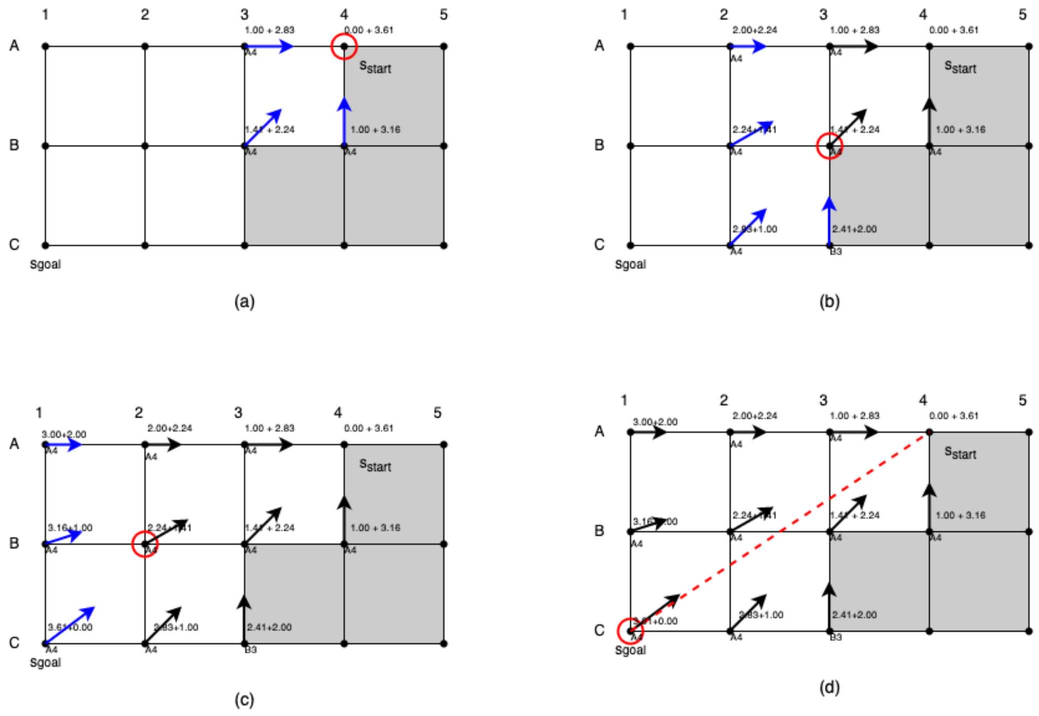


Figure 5: Theta* Trace

(c) As the difference between the A* algorithm and the Theta* algorithm lies in the updating of the vertices, we decided to create a common function that popped vertices out of the fringe (our min-Heap priority queue) and added all of the adjacent vertices into the fringe that were not added to it already. We implemented a closed set to ensure that vertices would not be added to the fringe multiple times and only be expanded a maximum of one time. In order to add all adjacent vertices, we implemented an adjacency list. This adjacency list is a python dictionary where the indexes are the coordinates of a vertex and the accompanying terms are the neighboring vertices. In order to achieve this, we first added the popped vertex's coordinates as a string to be a term in the dictionary. Then, we ran a function that verified that the edges of the neighboring vertices were acceptable through checking various conditional statements. If the edge between the vertices was passing diagonally through a blocked cell, the edge was invalid and the neighboring vertex was not added to the adjacency list. Also, if the edge was passing vertically or horizontally between two blocked cells, that was also a violation and the neighbor would not be added to the adjacency list of the current vertex. If the adjacent vertex had a valid edge, it was then determined whether the vertex was in the closed set. If not, then the fringe would be checked if it contained the neighboring vertex. If the fringe does not contain the neighboring vertex, then the g value of the vertex is set to infinity so all other g values would be less then it. Also, the parent would be set equal to null. Then, if the command-line argument was set to "aStar", the update vertex function for the A* algorithm would be used.

In the update vertices function for the A* algorithm, the distance between the popped vertex (s) and the adjacent vertex (s') would be calculated. This distance would then be used for comparing the g value of the adjacent vertex with the g value of s and the c(s,s'). If the g value and c(s,s') (or distance) was less than the g value of the adjacent vertex, then the g value of the adjacent vertex would be re-assigned to be g(s) + c(s,s'). Then, the parent of the adjacent vertex would be set as s. The H-value will then be set to equal Equation 1 in the assignment, where it shows the minimal possible path through using the most possible 45 degree angle diagonal lines from the adjacent vertex to the goal vertex. If the fringe already contained the adjacent vertex, then it would be removed from the fringe. Afterwards, the adjacent vertex's h and f values will be set and the adjacent vertex would be inserted into the fringe. The rest of the current vertex's valid neighbors will be checked against the closed set, fringe, have its values updated, and then be inserted into the fringe if meeting the conditional statement. Finally, the current vertex would be fully expanded and the next vertex would be popped from the fringe and this process would continue until either the fringe is empty or the vertex being popped is the goal vertex. The path would then be printed onto the grid and visualized, completing the A* algorithm.

(d) As mentioned in part c, the algorithm leading up to the updating of the vertices is the same for both the A* and Theta* algorithms. Therefore, our implementation of Theta* uses the same method of popping from the fringe, generating the adjacency list, examining the closed set, and then checking conditionals of whether to update the values of the vertex before being added to the fringe. If the command-line argument was set to "thetaStar," then the update vertex function for the Theta* algorithm would be used. However, one of the biggest difference between A* and Theta* is the change in H-values. For Theta*, the H-value is now the straight line distance from the current vertex to the goal, as Theta* is not restricted to only 45 degree angles and can have straight lines between unblocked vertices at any angle.

In order to update the vertices according to the Theta* algorithm, a line of sight function had to be implemented. The line of sight function returned a conditional statement of whether the neighboring node was in the line of sight of the current node's parent. The function used a variety of conditional statements to ensure that if the path between the vertices crossed through a blocked cell at any-angle, then the neighboring vertex would be considered to be out of the line of sight of the current node's parent and return a false value. If the line of sight function returned a false value, then the vertex being added to the fringe would be updated according to the A* algorithm. If the line of sight function returned a true value, then the g value of the parent and the distance between the parent and the neighboring vertex would be calculated. If the calculated value is less than the g value of the neighboring vertex in the line of sight, then g value of the neighboring vertex would be set to the calculated value and the parent of the vertex would be set to the parent of the current node popped from the fringe. Then, if the fringe contains the neighboring vertex, then the vertex would be removed from the fringe. Afterwards, the vertex would have its h and f values set before adding the vertex to the fringe and then returning the fringe. The current vertex would be fully expanded and have each neighboring vertex checked by the line of sight function with its parents. Finally, the next vertex would be popped from the fringe and this process would continue until either the fringe is empty or the vertex being popped is the goal vertex. The path would then be printed onto the grid and visualized, completing the Theta* algorithm.

(e) We can prove that A* with the h-values from Equation 1 is guaranteed to find the shortest path by utilizing the properties of the given h-value(s). From lecture, we know our heuristic must be admissible. It can also be consistent, because we know a heuristic that is consistent. must also be admissible. A consistent heuristic has consistent h-values if and only if they satisfy the triangular inequality:

$$\text{iff } h(s_{goal}) = 0 \text{ and } h(s) \leq c(s,s') \text{ for all vertices s, s'} \in S$$

Additionally, Equation 1 is taking the max distance ( $max(|s^k - s^k_{goal}|)$ ) and the minimum distance of the diagonals. This must guarantee the shortest path, since it cannot over predict due to the inequality shown above. Since each edge costs one unit and going past one block would two edges (i.e. two units) we must minimize the edge use. Using diagonals, which cost $\sqrt{2}$ units would lower the overall cost assuming there is no blocking of diagonals. This best case scenario created through analyzing Equation 1 incorporates some number of edges and some number of diagonals to minimize total cost and result in the shortest path. Equation 1 is shown below:

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

The maximum distance between the current point and the goal point would require taking a path where no diagonals are used, as the edge cost of 2 units whenever there is a vertical and horizontal difference would be higher than $\sqrt{2}$ (which is around 1.4). Therefore, the heuristic equation in this problem cannot neglect diagonals, or it gives the possibility of over-estimating and would thus not be admissible. The only situation in which to never over-estimate would to use every diagonal possible and never use the 2 edge cost path. Therefore, the minimum between the horizontal and vertical difference of the current point and the goal point would provide the number of diagonals that it would take to either completely remove the vertical difference or horizontal difference. After the vertical or horizontal difference is removed, the only edge costs remaining would be the 1 edge cost path to the goal point. While this equation does not account for blocked cells through using every diagonal possible, it fulfils the admissible property of the heuristic equation by leaving no possibility that the h value is greater than the optimal path.

(f) In our project, we decided to implement our own binary heap to use in both the A* and Theta* implementations. Our min-heap priority queue is located in the priorityQueue.py file in the classes folder. In order to create the priority queue for the algorithms, we chose to use a minimum Heap. The key of the min-Heap would be the f-value of the vertices being expanded upon. Our min-Heap has functions to find the parent of a given index, the right child of a given index, and the left child of a given index. Using these functions, we were able to create a min-Heapify function where the min-heap is ensured to be following the principle where the parent nodes have smaller f-values than its two children. Therefore, if the parent is smaller than any children, the min-Heapify function switches the parent's position with the child with the lower f value and then checks the rest of the min-Heap recursively. We provided functions to switch the position of two vertices in the heap, pop the minimum value of the heap, and to insert a vertex into the heap. After inserting into a heap, our code ensures that the heap is still following the behavior of a min-heap, where every parent has a smaller f-value. Finally, we provided functions to check whether the heap is empty, see if a heap contains a certain vertex, and to initialize the class.

(g) In our first implementation of code, we attempted to create an adjacency list for the A* and Theta* algorithms. An adjacency list would add a space complexity of O(8 * n * m) = O(n*m), where 8 is the maximum number of neighbors, n is the length of the grid, and m is the width. However, it would allow us to determine a vertex's neighbors with a run-time complexity of O(1). After the initial submission, we were able to successfully implement our adjacency lists and our A* and Theta* algorithms overall, through comparisons with the example traces given in the assignment and our own test cases. After verifying that our implementations worked, we first decided to optimize the run-time.

Through our binary heap, we were able to optimize the requirement of having the fringe. Through our implementation of using a Python List, we were able to get the parent of the current vertex in the fringe with a run-time complexity of O(1). Also, through our implementation, the most costly operation (min-heapifying the min-heap) was only called when popping a vertex from the fringe. Our function for inserting a node iterates through the heap, swapping the parent with the newly added node if it has a lower f value. This effective implementation of our binary heap allowed our run-times to be lower. Our next approach towards optimizing the run-time was to first identify any functions that were running more times than necessary. We identified that with each vertex that is being popped from the fringe, we ran our function to add the vertex to the vertices dictionary (our adjacency list) and then ran our function to add all the adjacent vertices as values. Therefore, these two functions only need to run once per node. We added a conditional statement to only run these functions if the popped vertex is not a key in the adjacency list dictionary. As these functions were in our algosMain function, the basis for both the A* and Theta* algorithms, we ran our experiments to graph the difference in run-time for both algorithms. First, we ran our A* algorithm on our 50 grids in a series of 10 tests and then averaged the average run-times per test. The graph is shown below:

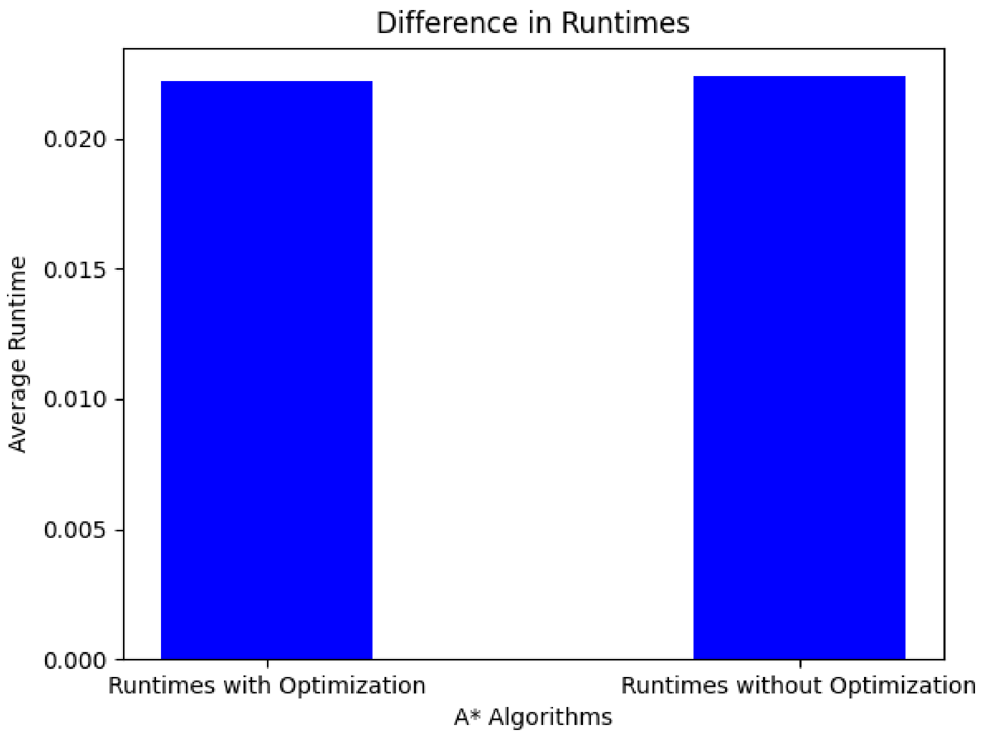


Figure 6: Analysis of Improvements in A* Algorithm

The difference in run-time is shown below. The average run-time of the optimized A* algorithm was 0.022236095918927872 seconds, and the average run-time of the non-optimized A* algorithm was 0.022390179523021456 seconds. The difference was 0.0001540836, which was minimal in the A* algorithm but appeared to more significant in the Theta* algorithms. The graph of the average run-times of the series of 10 tests of the 50 grids with the Theta* algorithms are shown below:

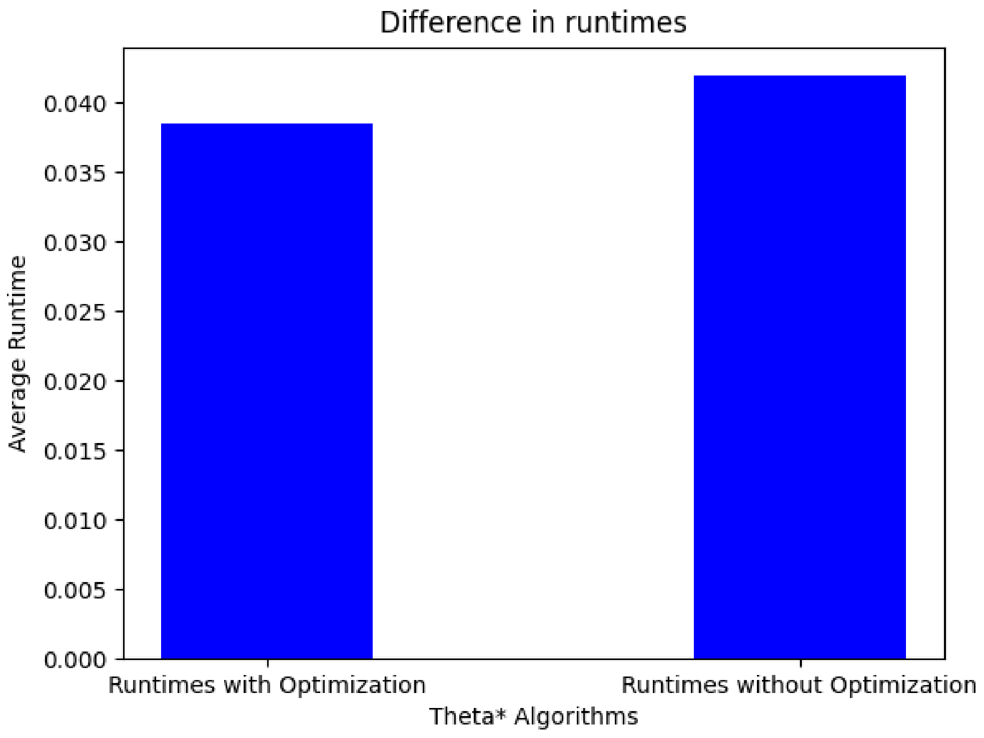


Figure 7: Analysis of Improvements in Theta* Algorithm

The run-time of the optimized Theta* algorithm was 0.038438252175611164 seconds, while the run-time of the non-optimized Theta* algorithm was 0.04190716614458296 seconds. The change accounted for a difference of 0.00346891396 seconds, or 8.28 percent. After optimizing run-time, we shifted our focus onto memory. The adjacency list accounts for the most additional memory beyond the essential data, such as the array of the blocked and unblocked cells, the priority queue minimum heap (fringe), and the Vertex objects for each vertex. Thus, we created an additional version of our algorithm that replaced the adjacency list with a Boolean. Therefore, for each vertex popped from the fringe, a function ran to see which each of the 8 potential adjacent vertices were not blocked and then inserted those adjacent vertices into the fringe if they were not already in the closed set. Through this method, no data structure was used for the memory, and efficiency was sacrificed for minimizing memory space. While we knew that the run-time would be considerably slower, we wanted to see if the run-times were close enough for the algorithm to be usable in a situation where memory space was limited. The averages of the results of the series of 10 tests of the 50 grids are shown below:

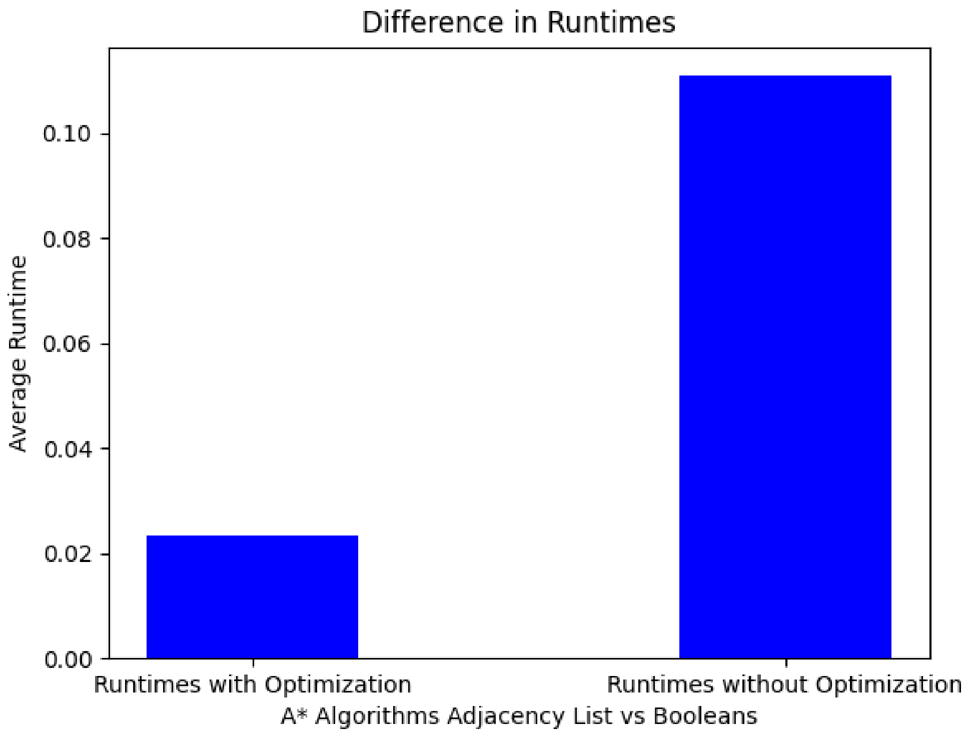


Figure 8: A* Analysis of Run-time of Speed vs Memory Algorithms

The average run-time of the Boolean version of the A* algorithm was 0.11083632871442371 seconds. The average run-time of the adjacency list version of the A* algorithm was 0.023501533253041523 seconds. The difference was very significant with a difference of 0.08733479546 seconds, or the adjacency list version was equal to 21.2 percent of the Boolean algorithm. Therefore, in most situations, the memory saved by this implementation of the A* algorithm would not be worth the huge cost to its efficiency. The average run-times of the Theta* versions are shown below:

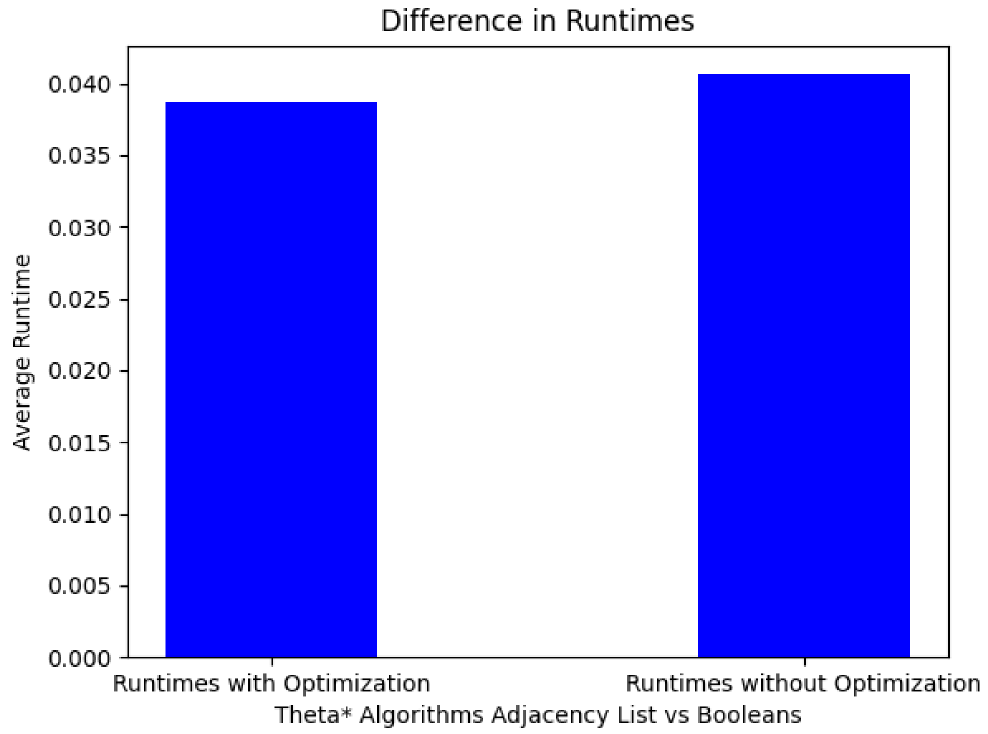


Figure 9: Theta* Analysis of Run-time of Speed vs Memory Algorithms

The average run-time of the Boolean algorithm was 0.0386651707936847, while the average run-time of the adjacency list algorithm was 0.04061230101850298. The difference was 0.00194713022, which is far more acceptable than the difference with the A* algorithm. Therefore, this algorithm works as a optimized version for memory when Theta* is available for use. In addition to testing the Boolean method, any unused variables in our functions were deleted albeit, saving a very small amount of memory.

(h) In order to analyze the differences between A* with the H values from Equation 1 and Theta* with the H values from a straight line from the current vertex to the goal, we first created 50 grids with each cell having a 10 percent chance of being blocked. In our functions that we used to create the grid, we used conditional statements to ensure that the starting and goal vertices were not surrounded by blocked cells. We also ensured that they were not equal to each other. After verifying that the test grids met the requirements, we began our tests. First, we conducted experiments to compare the run-time of A* and the run-time of Theta*. We collected the average run-times of both the Theta* and A* algorithms in a series of 10 tests, where all 50 grids were searched by both algorithms in each test. The final result is shown below:

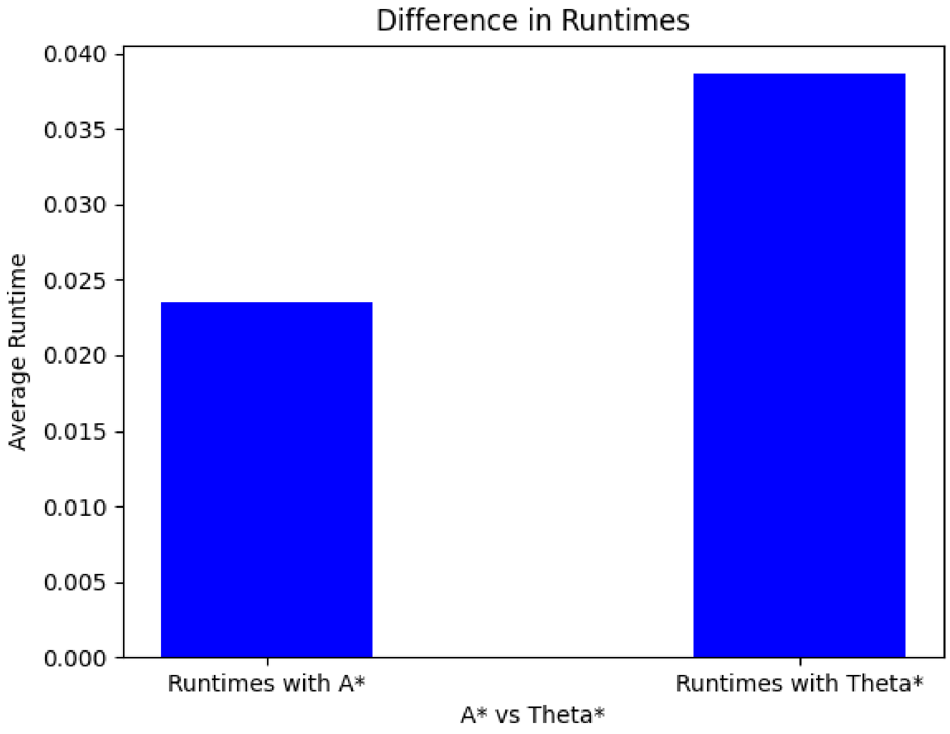


Figure 10: Run-times of Theta* and A*

The average run-time of A* was calculated to be 0.023555940647919972, and the average run-time of Theta* was calculated to be 0.03863190359709755. The difference between the algorithms was 0.01507596294, or A* was 1.64 times faster than Theta*. As both of these algorithms use the algoMain() function that provides the basis for both A*-family algorithms, they both benefit from the fringe from our custom priority queue. These run-times were both decreased through our implementation limiting the number of times that the min-heap is completely resorted. We then generated both a line plot of the different path sizes by algorithm for all 50 grids, and then plotted a bar graph of the averages. The line plots and bar graph are shown below:

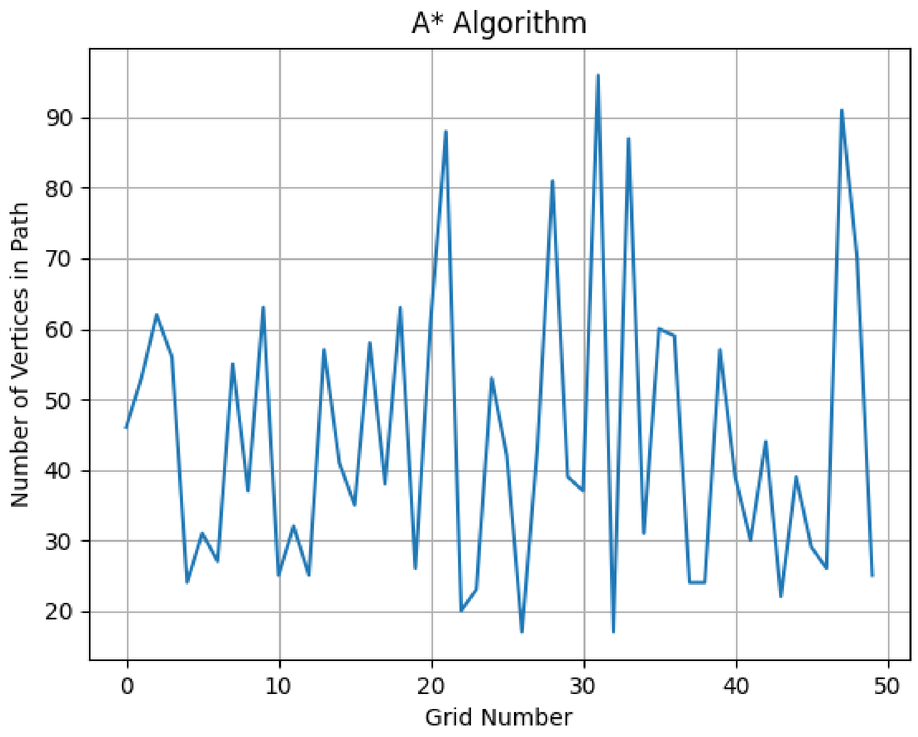


Figure 11: Paths generated from A*

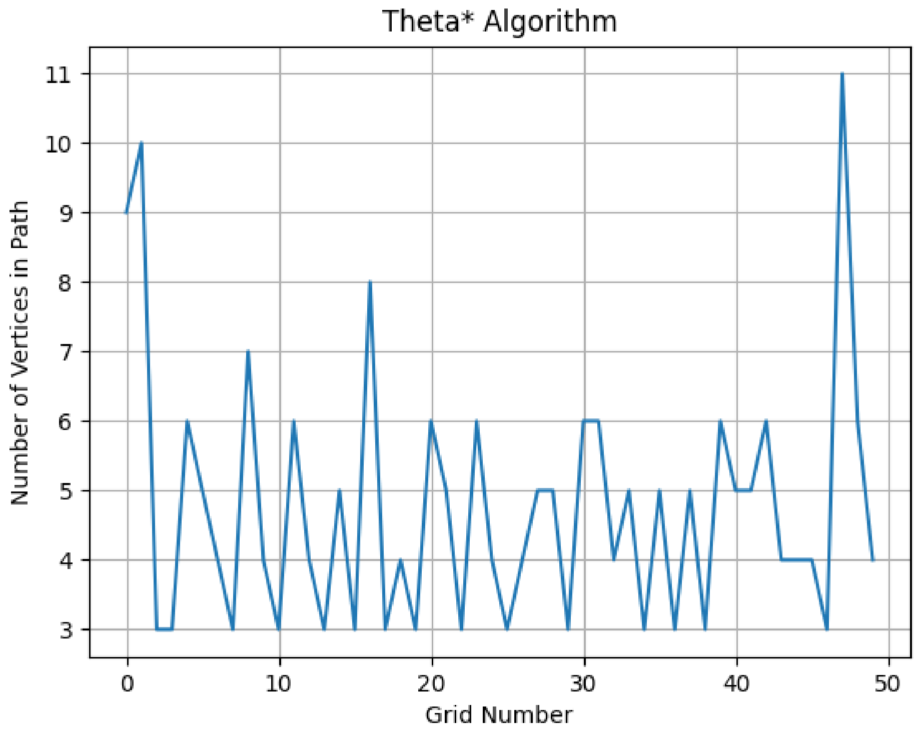


Figure 12: Paths generated from Theta*

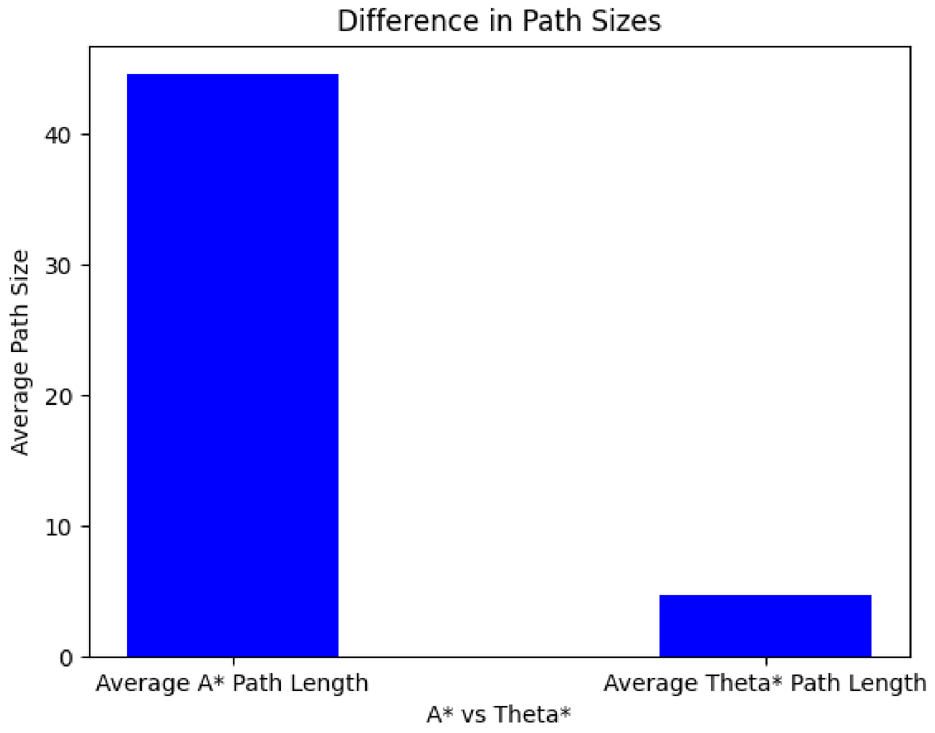


Figure 13: Average Path Size by Algorithm

Through these experiments, we determined that while A* had a faster run-time than Theta*, it also had a much higher average path length. A* had an average path length of around 45 (44.56), and Theta* had an average path length of around 5 (4.76). As A* has so many more vertices in their paths, there will be a lot more variance between A* algorithms than in Theta* algorithms. This is due to the path determining on which vertices are looked at first. If two or more adjacent vertices are calculated to have the same F values, then the vertex that is examined first, will be expanded first. These differences between paths found by different A* algorithms will only compound as more and more different vertices are expanded first. However with Theta*, even if two or more adjacent vertices are calculated to have the same F values, if they are both in the line of sight of a vertex with the lowest F value, the path will be the same as the path would now be from the previous vertex to the new vertex. Therefore, we conclude that while the A* algorithm is faster, the Theta* algorithm is much more consistent in finding an optimal path.

(i) It is appropriate for the Theta* and A* algorithms to use different heuristic equations. This is due to the A* algorithm being confined to horizontal, vertical, and 45 degree diagonal lines, whereas the Theta* algorithm can have horizontal, vertical, and any-angle diagonal lines. Therefore, the below heuristic equation does not have the true minimal Theta* path and would over-predict:

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

As discussed in lecture, a heuristic function can only be optimal if it never over-predicts the cost to reach the goal. The heuristic equation for Theta* that should be used is shown below:

h(s) $= \sqrt{(s^x - s^x_{goal})^2 + (s^y - s^y_{goal})^2}$

The probability that the path between $s$ and $s_{goal}$ to have 0 blocked cells between them is low (with a probability of (0.9) raised to the power of $(s^x - s^x_{goal}) * (s^y - s^y_{goal})$, or the area between the points)), but it is possible. Therefore, the only way to not over-predict would be to have the heuristic equation be equal to the the distance between $s$ and $s_{goal}$, and the absolute minimum path would only have the starting and ending point with a direct line between them. When using the A* algorithm to update the neighboring vertex's values when it is not in sight of the current vertex's parent, it should use the same heuristic equation that was used in the algorithm with only A*. This is due to the minimum path of A* being restricted to horizontal, vertical, and 45 degree diagonal lines and thus, it would not be possible for the minimum path to be an any-angle straight line between the goal and the vertex being examined.

(j) Through our visibility graph, we were able to determine two different types of scenarios where the paths found by the Theta* algorithm using h(s) = c(s,sGoal) values were longer than the true optimal paths. One scenario is that there are blocked cells in between the goal and starting vertex that create a tight window. One vertical and one horizon example of this scenario are shown below:

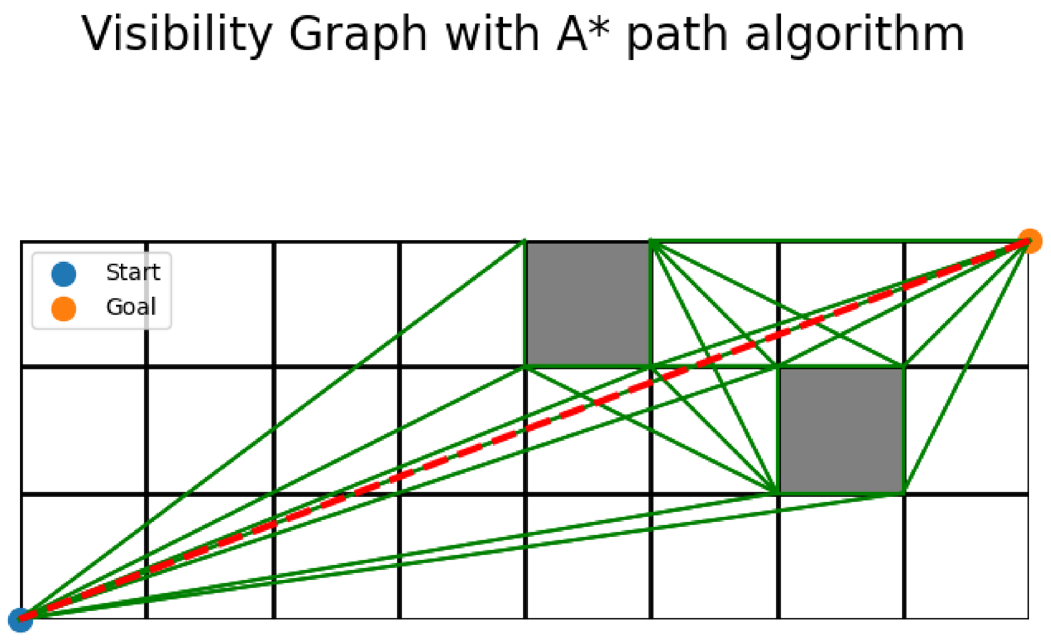


Figure 14: Horizontal Tight Window True Path

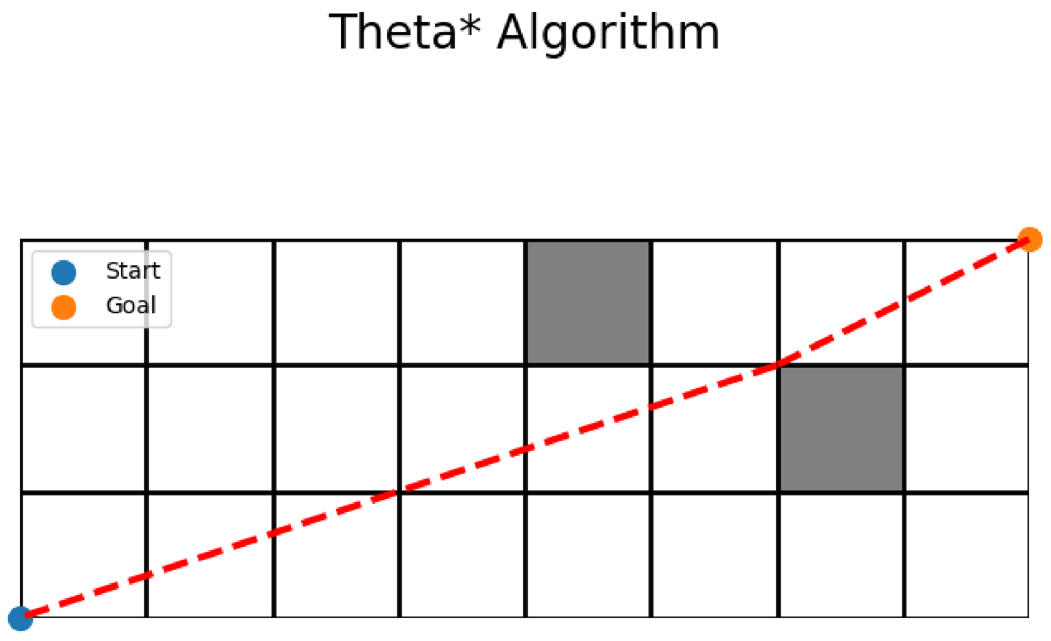


Figure 15: Horizontal Tight Window Theta* Error

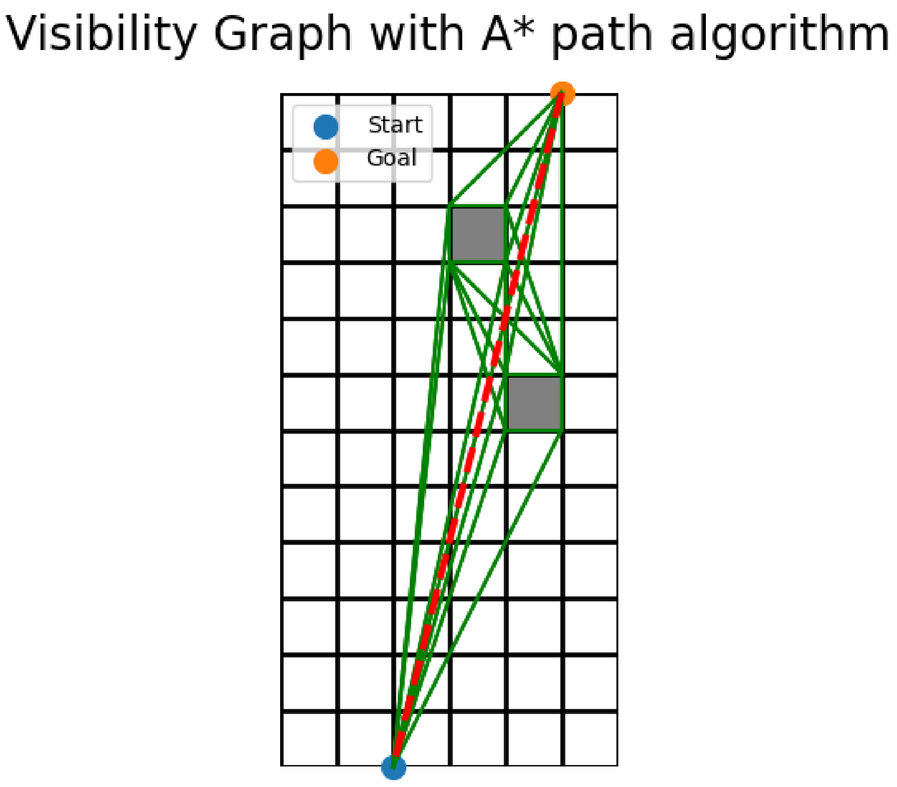


Figure 16: Vertical Tight Window True Path

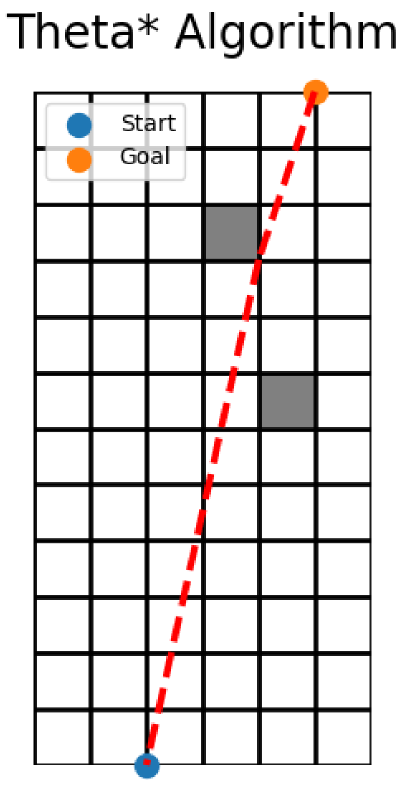


Figure 17: Vertical Tight Window Theta* Error

The distance of the true path in the horizontal case is equal to 8.544, whereas the Theta* distance and A* distance are 8.561 and 9.243 respectively. Therefore, the Theta* distance is 0.017 off, or 0.199 percent off the true shortest distance. Also, the A* distance is 0.699 off, or 7.562 percent higher than the true shortest distance. In the vertical case, the distance of the true shortest path is 12.369, whereas the Theta* distance and A* distance are 12.382 and 12.464 respectively. Therefore, the Theta distance is 0.013 off, or 0.105 percent off. Also, the A* distance is 0.095 off, or 0.762 percent higher than the true shortest path.

(k) Using our A* algorithm, Line of Sight function, and adjacency list functions, we were able to successfully implement the visibility graph. First, we added the starting vertex, goal vertex, and the 4 corners of each of the blocked cells as keys of the adjacency list. Then, we utilized our line of sight function to determine whether an edge should be created between the vertices. For example, even if a vertex in the line of sight of the next vertex, if the vertex is not in the line of sight of at least one of the vertices on either side, then the vertex being analyzed is in between two blocked cells. The edges of the visibility graph are shown in green and the true shortest path is shown in red. Examples of our visibility graph are shown below:

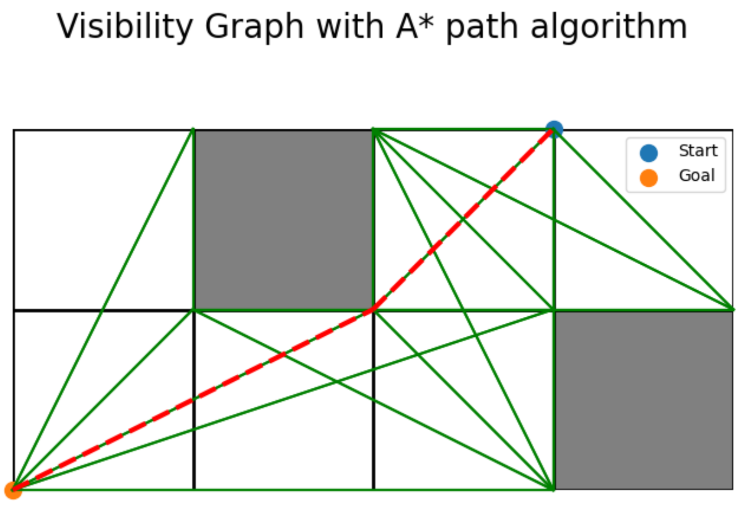


Figure 18: Visibility Graph Given Example

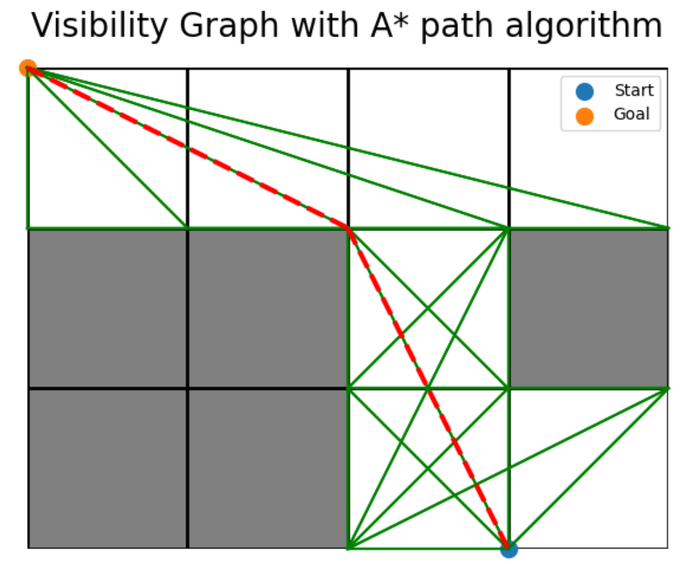


Figure 19: Visibility Graph with Blocked Edge Cases

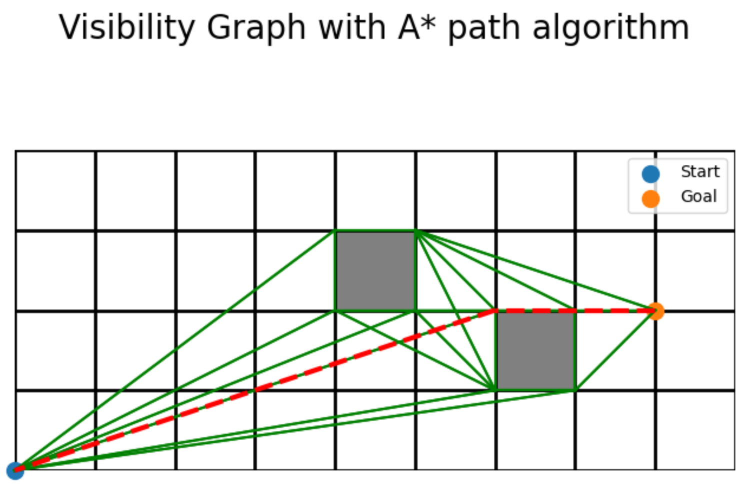


Figure 20: Visibility Graph

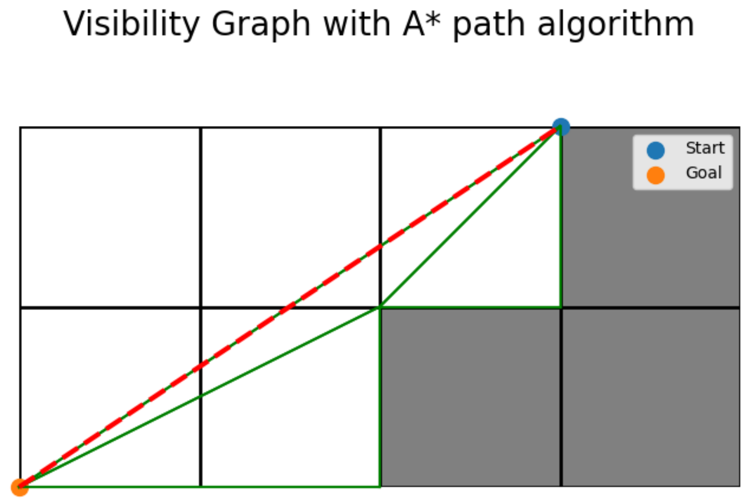


Figure 21: Visibility Graph

(l) We were unable to solve Question L.

---

2. Adversarial Search

   (a) Game Tree

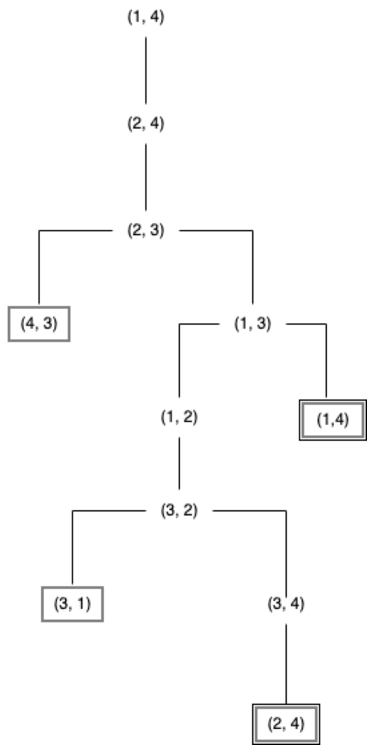


Figure 22: Game Tree

(b) Game Tree with Minimax

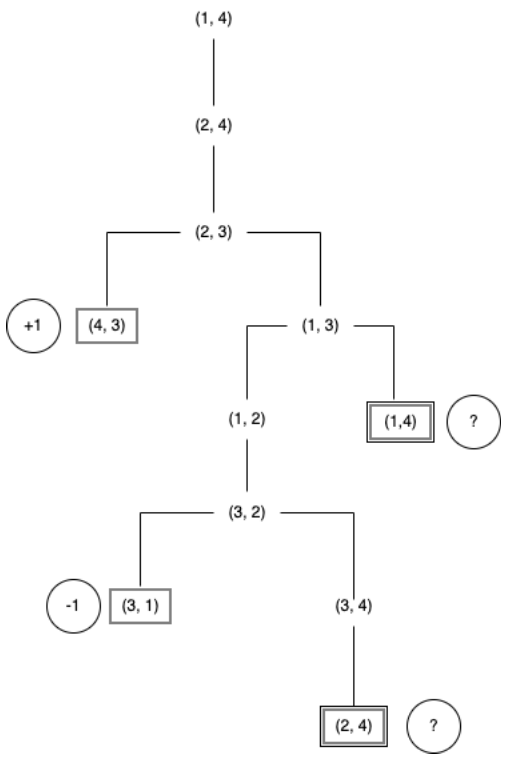


Figure 23: Complete Game Trees

In order to handle the "?" values, we must make the assumption that whenever a choice is given between a "?" value and a win, the agent will always choose to win. Therefore, the only case where the "?" is present is when all of the successors are "?". If it is min(-1,?) or max(+1,?), the -1 and +1 will be chosen respectively.

(c) The standard minimax algorithm would fail because it is a depth first search. When a depth first search is run, the game runs infinitely and thus will not end. In order to fix this, a possible solution would be to compare the current state to the stack of previous states. In the case of a repeat, return the "?" value with similar properties as (b). The only downside is this method is not foolproof, as a game tree that has a tie position would break the logic for determining "?" vs a draw (since a draw is neutral, we cannot assume a move).

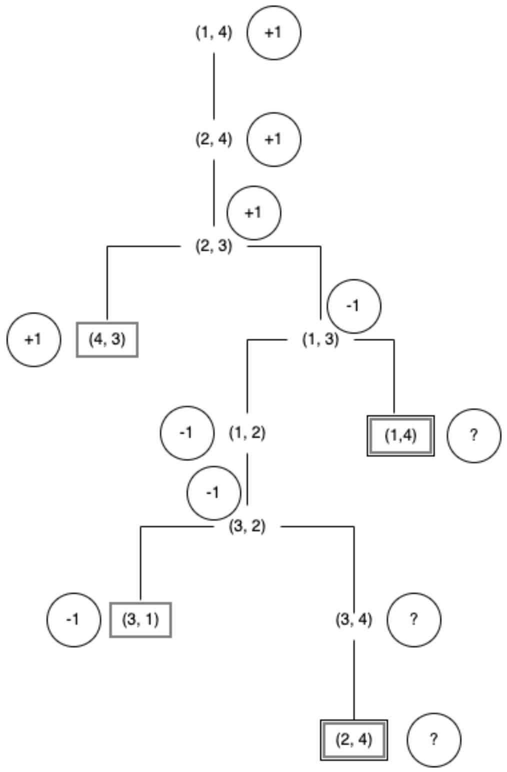


Figure 24: Complete Game Tree with Minimax Values

(d) In order to prove that A wins whenever $n$ is even and loses whenever $n$ is odd, provided $n > 2$, we will do a proof by induction. There are two bases cases, when $n = 3$ and $n = 4$. We have proven that when $n = 4$, which has been proven in the previous parts (particularly the game tree). We can use the same method to prove that when $n = 3$ which uses the same method.

Now to the inductive step, for any game where $n > 4$, the first steps of the game remain constant: Player A will always move to position two and Player B will move to position $n - 1$. As a result, the board size is now $n - 2$ (but both players have an extra move which accounts for moving to a previous position). Thus, as the game continues to be played at size $n-2$, the winner of the size game will always reach the second to last square, which is one away from winning the game.

To complete this proof, we must examine the case of moving backwards. We know that the winning player will always play optimally, thus the projected winner will always move towards their winning position. On the other hand, the projected loser does have the option to move to their previous position. However, even if they do move to a previous position, the opposing player will continue to move closer; thus, the size of the game remains $n-2$. Therefore the proof holds as true: the projected winner will always win, and this projected winner is dependent on whether $n$ is odd or even.

---

3. Local Search

   (a) The problems where hill climbing will work better than simulated annealing are problems where the cost function is smooth and has only a global maxima. If there are local maxima, ridges, or plateaus the hill climbing method is sub-optimal. In the case of local maxima, the hill climbing algorithm will approach the peak of a local maxima and then remain stuck there, even if it is not the global maxima. Similarly, ridges are a series of local maxima so the hill climbing search method will get stuck at one of the ridges regardless of whether it is the global maxima. Lastly is the case of plateaus, where there is no peak; thus, the algorithm will wander indefinitely since no maxima is to be found. If any of these cases are present, the simulated annealing method will be better.

   (b) The case where randomly guessing the state works just as well as simulated annealing is when the structure of the cost function is relatively standard. This means that regardless of how the cost function is defined, the regions surrounding the global maximum is similar to the regions of any given location. You would never be able to tell if you are approaching the solution, nor would the shape of the structure guide you towards the solution. In this case, both randomly guessing and simulated annealing work at similar efficiencies.

   (c) Based on our previous answers, simulated annealing is most useful when the problem has some form of general structure as well as local maxima. The general structure makes simulated annealing more efficient than randomly guessing, while the existence of local maxima means that the hill climbing method is not optimal. When these two criteria are met, the simulated annealing technique is most optimal.

   (d) The simulated annealing method returns the very last current state at the end of the annealing schedule. To optimize this, rather than returning the last state, we should keep track of the highest value of the states that we visit and return that at the end of the annealing schedule. In this case, we can ensure the state we return has the highest value, as there is no guarantee that the simulated annealing method will return such a state.

   (e) If we had enough memory to store two million states, rather than the two needed for simulated annealing, one way to modify the technique would be to perform an annealing on each of the two million states, but instead of just moving to the next scheduled state, allow the algorithm to move to the state with the highest score, regardless of whether it was local or not. In this way, we could efficiently utilize the provided memory to come to a solution more efficiently.

---

4. Constraint Satisfaction Problem

(a) *i.* Set of variables: each square is a variable, which can be represented as $X = S[1,1], S[1,2], ..., S[9,9]$

*ii.* The domain of possible values for each variable would be any number from 1-9: 1, . . . ,9

*iii.* The constraints would look extensive as they extend for the row, column, and block. These can all be made into one relation called 'Different' which ensures all values are different. There are 9 constraints from each row, column, and block which gives us a total of 27 (9x3) constraints. The constraints and there values are shown below:

Row Constraints: $(i\forall = 1, 2, ..., 9)$
$C = Different(S[i,1], S[i,2], ..., S[i,9])$

Column Constraints: $(j\forall = 1, 2, ..., 9)$
$C = Different(S[1,j], S[2,j], ..., S[9,j])$

Block Constraints: $C = Different(S[1,1], S[1,2], ..., S[9,9])$

The last possible set of constraints would be for the squares that are already filled and these would be limited to their specified value. No other value would be allowed in these pre-filled squares.

(b) One way to formalize the Sudoku search problem would be to begin with the starting board with all the pre-filled squares as the start state. The successor function would pick an empty square on the start state board and fill it in. This function will assign a number from 1-9 for each square and it will do this for each consecutive empty square. The goal test would be to test if all squares are filled and all constraints are met. For example, if there are repeating numbers in a row then the constraints/goal test are not met. Lastly, the path cost function would measure the path cost when moving from square to square, whether it be left to right or top to bottom. In this case, the path cost will always be equal to one.

(c) The difference between "easy" and "hard" problems is that the easier Sudoku problems generally have more "clues" or starting digits that a player can work with to solve the game. Harder Sudoku problems will omit some of this information which leaves the player with a larger number of possible solutions. In terms of heuristics, this can be put so that easy Sudoku problems have heuristics that provide more information and allow for less backtracking which makes the game easier. Meanwhile, heuristics that give less information will require more backtracking and in turn be more difficult to solve because they require a more sophisticated search over a larger space.

(d) Pseudocode:

```
function localSearchSudokuGame (SudokuGame game, int iterations)
assignment = random value assignment from 1, . . . ,9 to variables
for i = 1 to iterations do
if assignment meets constraints then return assignment
num = random variable in a row, column, or square that is empty (not met constraints)
assign random value to num
```

This approach would definitely work better than the best incremental search algorithm for easy Sudoku problmes, however, it would not work as well for hard Sudoku problmes. This is because the time complexity would be inefficient when searching over such a large space.

---

5. Logic-Based Reasoning

   (a) Knowledge Base:
   i. $Superman \land\ (Batman \land Kryptonite) \Rightarrow\ Superman\ Defeated$
   ii. $Kryptonite\ \iff\ Batman\ \land\ Lex\ Luther$
   iii. $Batman \land\ Lex\ Luther\ \ \Rightarrow\ Upset\ Wonder\ Woman$
   iv. $Upset\ Wonder\ Woman \Rightarrow\ Wonder\ Woman\ Intervention\ \land\ Superman$

   (b) Convert Knowledge Base to CNF:
   i. $(\neg Superman\ \lor\ \neg Batman\ \lor\ \neg Kryptonite)\ \lor\ Superman\ Defeated$
   ii. $(Batman\ \land\ Lex\ Luther)\ \lor\ \neg Krytonite$
   iii. $(\neg Batman\ \lor\ \neg Lex\ Luther)\ \lor\ Upset\ Wonder\ Woman$
   iv. $\neg Upset\ Wonder\ Woman\ \lor\ (Superman\ \land\ Wonder\ Woman\ Intervention)$

   (c) Prove: Batman cannot Defeat Superman
   Let's start by taking the following statement:
   $\neg P\ \lor\ \neg Q$, where P = Batman and Q = Defeat Superman

   We know that $\neg P\ and \neg Q$ essentially have the same representation because the negation on P removes in from question and the negation on an already negated Q value cancels out the negations on Q. This helps us such that we may bring the following equation into consideration: $P\ \lor\ R$, where $R = Superman$. Thus, we know that $P and \neg P$ cannot both be true such that both equations are true. The only possible solution would be for $R \lor \neg Q$ to be true through the resolution inference rule. Thus, the resulting logic shows that Batman cannot defeat Superman because we are left with 'Superman' and the opposite of 'Defeat Superman' after our proof. This result was found with the utilization of the resolution inference rule in conjunction with the logic statements that we comprised above.