Assignment 1, A* θ* visualization

Yousef Attia, Ameel Jani, Fauzan Amjad

CS 440

Visualization

We did this assignment in Python 3.8. For the visualization we chose to use pygame and pygame_gui.

To generate the gird we iterate from the upper left corner of the grid to the end of the grid in steps of the cell size. In each iteration we check the 2d array outputted by the file reading function to see if the cell is blocked, if the cell is unblocked we create a vertex object at each corner of the cell, if a vertex object is already present in that corner, we edit that vertex instead of creating a new one, after initializing all of the vertex objects we set the neighbors of each vertex. This way we make sure there are no paths over blocked cells. Afterwards we draw lines in between every vertex and all of its neighbors.

The vertex class is our graph node implementation, it stores all the important graph utilities along with the image coordinates and grid coordinates. The draw vertex method creates a pygame_gui.UIButton which is styled with theme.json. when the button is clicked a text box displaying the vertex information is drawn, when another button is pressed the text box is hiddenAfter the algorithm sets the parents of the vertices on the path we call draw path on the goal index which will draw a line connecting each vertex to its parent. The button on the left switches between A* and theta star. The program clears the previous image by drawing a cached image of the blank graph over the current graph, then the next algorithm is called and we call draw path again.

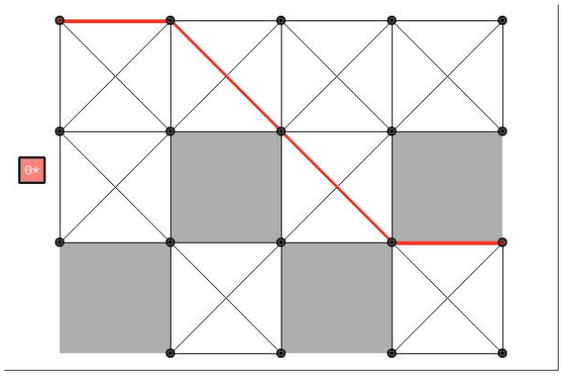


Figure 1: visualization of A* on 4x3 grid

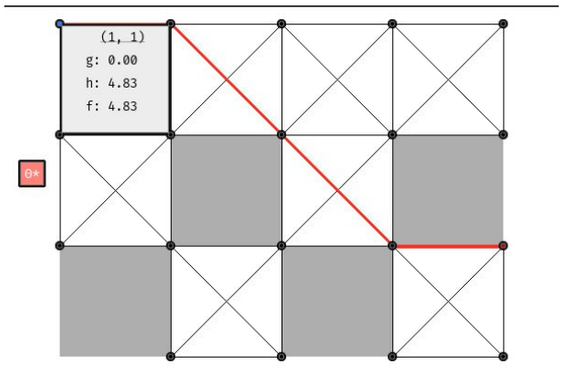


Figure 2: text box displaying vertex information

A * Implementation

In order to have this algorithm working, I had to implement several things. In this algorithm, we have to keep track of the nodes (vertices) that have been closed, the vertices that are in the open list/fringe, and the blocked cells. We also have to keep track of the coordinates of each vertex, the parent of each vertex in the path, the neighbors of each vertex, and the g, f, and h values of each vertex. I accomplished this by essentially making a class called "Vertex" in python and having it contain the boolean field "isclosed"; the floating-point fields "gvalue", "hvalue", and "fvalue"; and the tuple "coords" field. I also had each Vertex object contain a Vertex reference to its parent as well as a reference to a list of vertices that Vertex was adjacent to (the neighbors list). I used the python heapq library to implement the fringe/open list of nodes in the algorithm. I included a comparator —lt— function in the Vertex class to help the heapq class order vertices in priority of their g values. I initialized every Vertex's g value to infinity and is—closed to false.

In order to implement the algorithm itself, I first wrote a function to read the text file provided describing the grid. I went through the file line-by-line, first recording the coordinates of the starting and ending points. While reading the file, I also used a global 2D array called temparr to keep track of the cells that were blocked; when any line in the file ended with a 1, I set temparr[x][y] to 1. For every cell encountered, I added every possible vertex in it to a global list of vertices. After finishing reading the file and with the temparr array filled with the appropriate values, I went through the vertices array and updated each vertex's neighbors list with that vertex's valid neighbors.

At the beginning of the actual A star algorithm, I went through the entire vertices list and set each vertex's h-value to the value returned by the given heurstic function. I then set the starting vertex's parent value to itself. I also initialized an empty heapq array to represent the fringe and pushed the starting vertex into it. I then used a while loop and began removing (popping) items from the fringe in order of least to greatest f-value (order maintained by heapq) while there were still items in the fringe. For each vertex popped from the fringe, I marked its is—closed value as true and went through all the neighbors in its neighbors list. For each neighbor whose is—closed value was false, I calculated its straight-line (g value) distance from the start vertex. If this value was less than its current g value, I updated this vertex's g, h, f, and parent values (set its parent to the popped vertex). I then popped it from the fringe and then reinserted it into the fringe to maintain/update the order of vertices in the fringe. If the goal vertex ended up being popped from the fringe at any point, a path from the start to the end vertex had been found, and I made the algorithm exit.

Theta* implementation

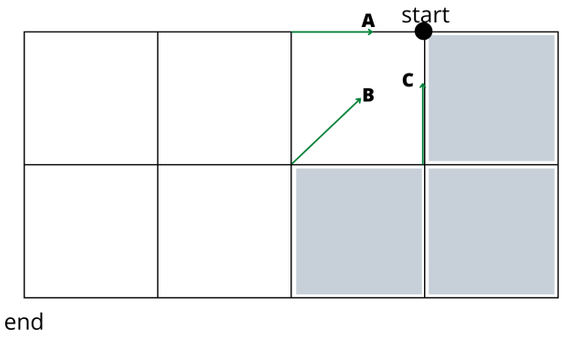
I ran the Theta* algorithm on the grid after running the A* algorithm on it. Before running this algorithm, I reset every vertex's g value to infinity, parent to null, h value and f value to its Euclidean distance from it to the end vertex, and its is—closed value to false. I also implemented a separate line of sight function to determine if 2 vertices are within the line of sight of each other using the pseudocode logic provided (I didn't implement anything extra for this part).

To implement the Theta* algorithm itself, I first set the starting vertex's parent value to

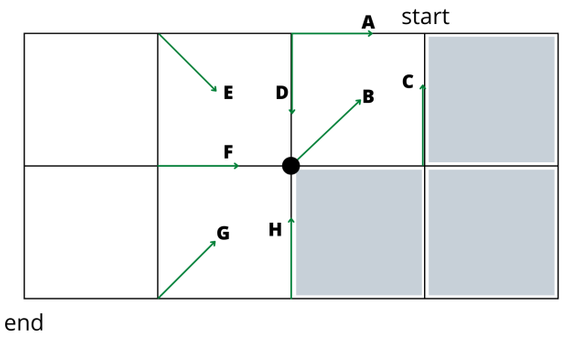
itself. I also initialized an empty heapq array to represent the fringe and pushed the starting vertex into it. I then used a while loop and began removing (popping) items from the fringe in order of least to greatest f-value (order maintained by heapq) while there were still items in the fringe. For each vertex popped from the fringe, I marked its is—closed value as true and went through all the neighbors in its neighbors list. For each neighbor whose is—closed value was false, I examined 2 paths. First, I checked if the line of sight function between each neighbor and the parent of the popped vertex returned true. If it did, I checked if the any-angle distance from the neighbor to the start vertex involving this particular path was less than its current g value. If this value was less than its current g value, I updated this vertex's g, h, f, and parent values (set its parent to the popped vertex). I then popped it from the fringe and then reinserted it into the fringe to maintain/update the order of vertices in the fringe if the line of sight function returned false, I calculated each neighbor's calculated its straight-line (g value) distance from the start vertex. If this value was less than its current g value, I updated this vertex's g, h, f, and parent values (set its parent to the popped vertex). I then popped it from the fringe and then reinserted it into the fringe to maintain/update the order of vertices in the fringe. If the goal vertex ended up being popped from the fringe at any point, a path from the start to the end vertex had been found, and I made the algorithm exit.

Part B

A* Method

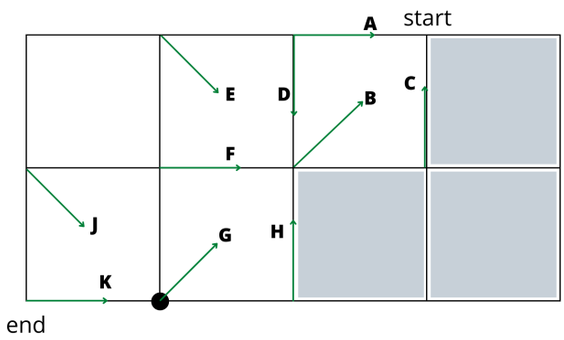


A = 1.00 + 2.83; B = 1.41 + 2.41; C = 1.00+ 3.41

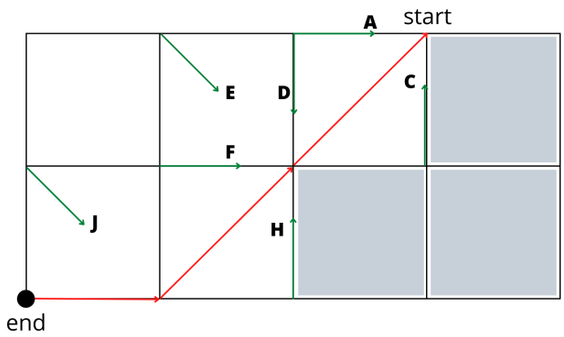


D = 2.41 + 2.83; E = 2.83 + 2.41

F = 2.41 + 1.41; G = 2.83 + 1.00; H = 2.41 + 2.00

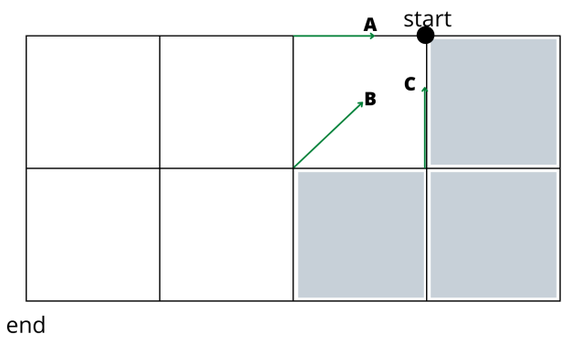


J = 4.24 + 1.00; K = 3.83 + 0.00

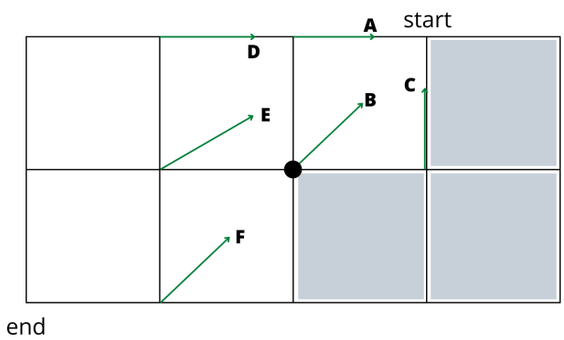


Shortest grid path found via A*

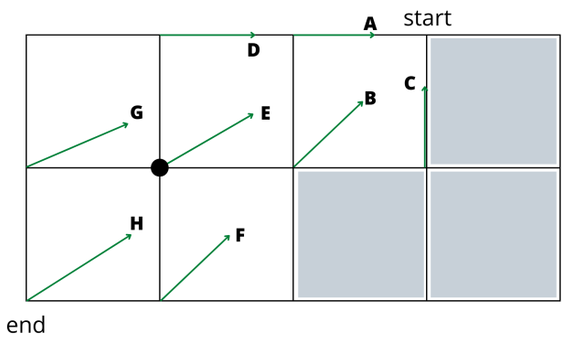
Theta* Method



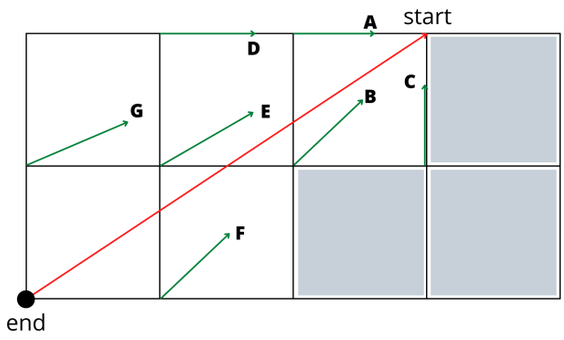
A = 1.00 + 2.83; B = 1.41 + 2.41; C = 1.00 + 3.41



D = 2.00 + 2.24; E = 2.24 + 1.41; F = 2.83 + 1.00



G = 3.162 + 1.00; H = 3.61 + 0.00; I = 2.83 + 1.00



Shortest any angle path found via Theta*

Part E

A* with the h-values from Equation 1 is guaranteed to find shortest grid paths because it implements a admissible heuristic, otherwise a heuristic function that doesn't overestimate the cost from the start point to the respective endpoint. Using a smaller number of nodes to get to an endpoint doesn't mean it's the smallest path as the distance between the nodes play the determining role, and A* accounts for that.

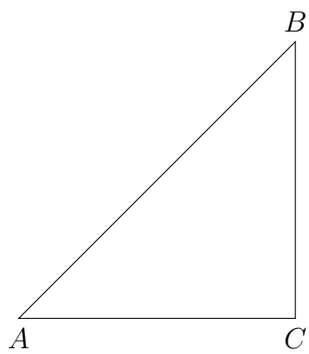


Figure 8

Utilizing Figure 8, let's get from Point A to Point B in the shortest path. The distance from A to B is 10, from A to C is 5, and from C to B is 3. Let's look at the neighbors of A and the distance between them from A. To determine which path to take, we will add "distance from A" and "distance from endpoint - B." If we went to B, it would be 10 + 0, which is equal to 10. If we went to C, it would be 5 + 3, which is equal to 8. We choose to go the route of C, and then finally achieve endpoint B - the shortest possible path (8 + 0 = 8).

The reason why A* is optimal in finding the shortest path as demonstrated via the example with Figure 8 despite its space complexity is because every step the algorithm takes, it estimates the distance it's already taken combined with the distance remaining for each

eligible path it can take. Let's say that for every endpoint, the algorithm calculates $x + y$, where x is the distance from the start point and y is the distance from the end point. If the algorithm and heuristic is developed correctly, this would mean that while x and y will change over the duration of the path, $x + y$ should remain constant, up until $y = 0$ and x = the distance from the start point and endpoint. Thus, it is guaranteed that the shortest path is found via the A* algorithm.

Part G

We made several modifications to our original submission as well the given pseudocode to make the algorithm more efficient. To start with, we implemented the closed list as well as the fringe/open list as Python sets. This allowed us to check whether a particular node was closed or was in the fringe in constant $O(1)$ time. We also, in a change from our original Feb 9 submission, had the vertexes nodes themselves store their own g, h, f, and parent values. This allowed us to access and update a node's g, h, f, and parent values at various stages of the algorithm in constant $O(1)$ time without having to iterate over all the vertices. We also made sure to only have the algorithm explicitly iterate over all the vertices once when first initializing them from the file inputs. Our code also never explicitly iterates over all the vertices between the start of multiple searches made on an identical grid, unlike in our previous submission. These changes/optimizations applied to both our A star and Theta star implementations.

Running Time for Our new version (A star):

d: depth of the shortest path

b: maximum branching factor= 8 in this case

V: number of valid vertices in grid

Calculating the f, g, parent, and h values for the starting vertex in the beginning and placing it into the fringe: $O(1)$ time

Iterating until fringe queue is empty: $O(b^d)$

Inside this iteration:

Removing Node from Fringe: $O(logd)$ Iterating through this Node's neighbors: $O(b)$ For every neighbor, seeing if it's closed: $O(1)$ For every neighbor, seeing if it's not in the fringe: $O(1)$ For every neighbor, seeing if it's g, f, parent, or h values need to be updated: $O(1)$ For every neighbor (worst case), updating it's g, f, parent, and h values : $O(1)$

Total Running Time for our new version of A star: $O(b^d * logd)$

Running Time for Our new version (Theta star):

d: depth of the shortest path

b: maximum branching factor= 8 in this case

V: number of valid vertices in grid

Calculating the f, g, parent, and h values for the starting vertex in the beginning and placing it into the fringe: O(1) time

Iterating until fringe queue is empty: $O(b^d)$

Inside this iteration:

Removing Node from Fringe: $O(logd)$ Iterating through this Node's neighbors: $O(b)$ For every neighbor, seeing if it's closed: $O(1)$ For every neighbor, seeing if it's not in the fringe: $O(1)$ For every neighbor, seeing if it's g, f, parent, or h values need to be updated: $O(1)$ For every neighbor (worst case), updating it's g, f, and h values : $O(1)$ For every neighbor (worst case), calculating the parent distance and line of sight (constant): $O(c)$

Total Running Time for our new version of Theta star: $O(b^d * clogd)$

Running Time for Our Old version (A star):

d: depth of the shortest path

b: maximum branching factor= 8 in this case

V: number of valid vertices in grid

Calculating the f, g, parent, and h values for the starting vertex in the beginning and placing it into the fringe: O(1) time

Iterating until fringe queue is empty: $O(b^d)$

Inside this iteration:

Removing Node from Fringe: $O(logd)$ Iterating through this Node's neighbors: $O(b)$ For every neighbor, seeing if it's closed (you would have to iterate through the entire fringe here): $O(d)$ For every neighbor, seeing if it's not in the fringe: $O(d)$

For every neighbor, seeing if it's g, f, parent, or h values need to be updated (in the pseudocode version, this would be $O(d)$, as you'd have to iterate over everything in the fringe) : $O(1)$ For every neighbor (worst case), updating it's g, f, parent, and h values (in the pseudocode version, this would be $O(d)$, as you'd have to iterate over everything in the fringe) : $O(1)$

Total Running Time for our old version of A star: $O(b^d * dlogd)$

Running Time for Our Old version (Theta star):

d: depth of the shortest path

b: maximum branching factor= 8 in this case

V: number of valid vertices in grid

Calculating the f, g, parent, and h values for the starting vertex in the beginning and placing it into the fringe: O(1) time

Iterating until fringe queue is empty: $O(b^d)$

Inside this iteration:

Removing Node from Fringe: $O(logd)$ Iterating through this Node's neighbors: $O(b)$ For every neighbor, seeing if it's closed (you would have to iterate through the entire fringe here): $O(d)$ For every neighbor, seeing if it's not in the fringe: $O(d)$

For every neighbor, seeing if it's g, f, parent, or h values need to be updated (in the pseudocode version, this would be $O(d)$, as you'd have to iterate over everything in the fringe) : $O(1)$ For every neighbor (worst case), updating it's g, f, parent, and h values (in the pseudocode version, this would be $O(d)$, as you'd have to iterate over everything in the fringe) : $O(1)$

For every neighbor (worst case), calculating the parent distance and line of sight (constant): $O(c)$

Total Running Time for our old version of Theta star: $O(b^d * c * dlogd)$

As you can see, for one search, our new algorithms are more efficient than our old versions or the pseudocode.

Part H

after observing A^* and θ^*'s behaviour, it is clear that θ^* will usually find a shorter path and in the worst case it will default to the same path found by A*. But, that shorter path does come with a higher time complexity, from my observations, it seems the extra time complexity is only worth it when traveling larger distances, as you can see in figures 3 and 4 and 4x3 grids.

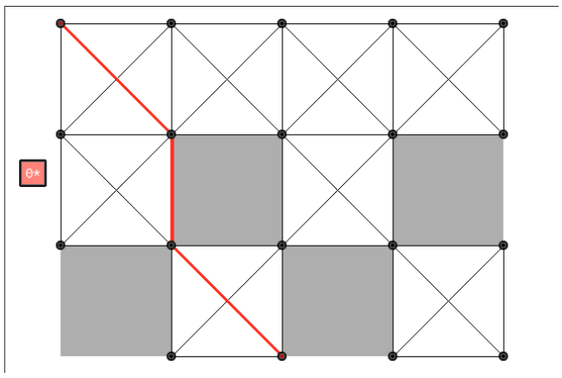


Figure 3: A* starting at point (3,4) and ending at point (1,1)

θ^* did find the true shortest path with an overall distance of 3.64, but the distance found by A* was only 0.19 units longer at 3.83. However as you can see in figures 5 and 6 on 100x50 girds, θ^* found the true shortest path with a distance of 34.41. But this time A*'s path is 2.32 units longer at 36.73. If I were to decide between these two algorithms, I would take into consideration the distance to travel, to make sure that extra computation time goes to good use, since over short distances the difference isn't significant. comparing the h-values of each algorithm, there doesn't seem to be much difference between the euclidean distance and the h function provided in the write up, all that really matter in the end is whether or not the function is admissible. However as you can see in figures 5 and 6 on 100x50

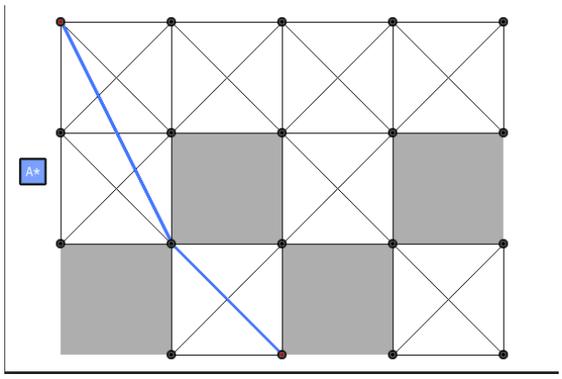


Figure 4: θ* starting at point (3,4) and ending at point (1,1)

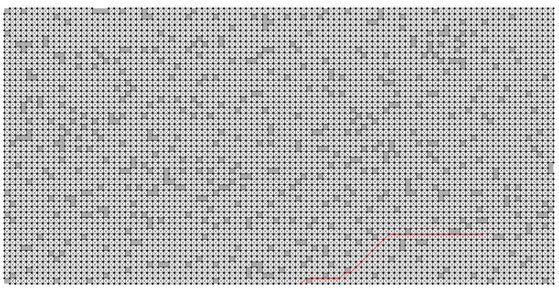


Figure 5: A* starting at point (55,51) and ending at point (88,42)

girds, θ* found the true shortest path with a distance of 34.41. But this time A*'s path is 2.32 units longer at 36.73. If I were to decide between these two algorithms, I would take into consideration the distance to travel, to make sure that extra computation time goes to good use, since over short distances the difference isn't significant. comparing the h-values of each algorithm, my observations prove that the choice of h-value was reasonable for both functions respectively, the h-value for A* could have overestimated the distances for θ* since it considers going through a straight line as well as a diagonal line.

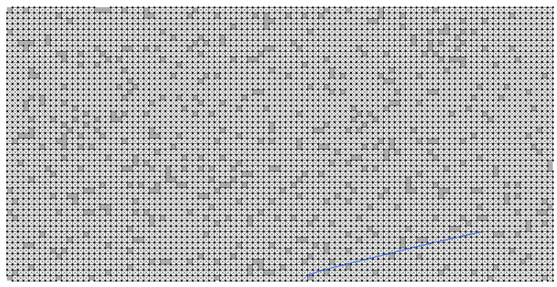
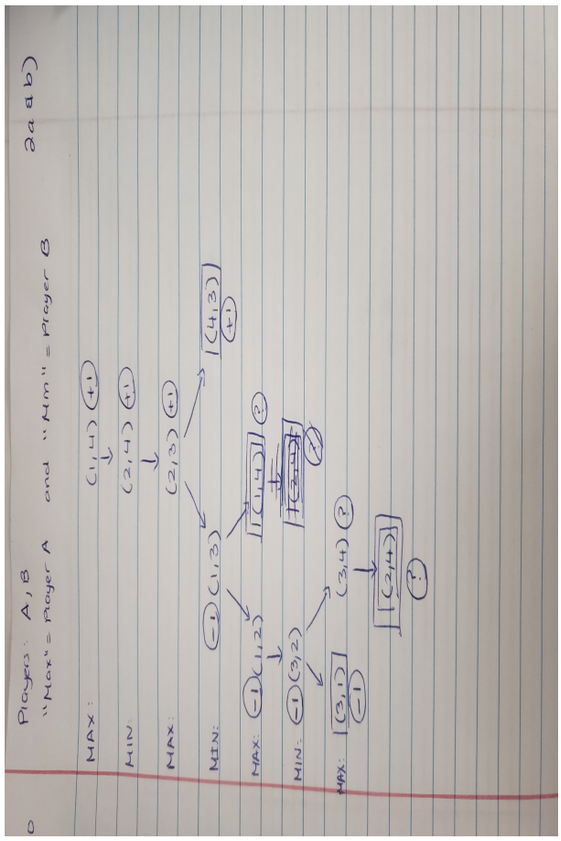


Figure 6: θ^* starting at point (55,51) and ending at point (88,42)

Question 2

a and b picture:



b continued:

As can be seen in the picture above, I implemented the MINIMAX algorithm on this game tree. While doing so, however, I had to make several assumptions on how to handle the loop states and the ? values in the tree. To start with, I assumed that the ? values in the tree were terminal states. I made this assumption because if the loop states are treated as actual infinite repeating paths in the game tree, the recursive algorithm would never end, as its base case (terminal state) wouldn't be found for some branches of the tree.

Upon making this assumption, I also made the assumption that the ? values represent a utility of 0 at the corresponding loop state. I made this the case for 2 reasons. First, having the game terminate at a loop means that neither player wins the game (the game is a draw essentially at this point). I thought that the best way to represent this would be to set Player

A's utility at this state to be 0. Second, having the ? values represent a utility of 0 would make it possible to easily compare the utilities of the successors of each node to determine their maximum or minimum in the MINIMAX algorithm.

c:

The standard MINIMAX algorithm will fail on this game tree for 2 reasons. The first reason is the fact that the tree contains loop states which essentially correspond to an infinite looping path in the standard application of the algorithm on this tree. This means that the game tree itself has infinite paths too. When the standard MINIMAX algorithm is run on this particular tree, it would never end, as its base case (terminal state) wouldn't be found for some branches of the tree. Second, without a comparison mechanism for comparing ? state utilities to +1 or -1 utilities, the MINIMAX algorithm is unable to compare the utilities of the successors of each node to determine their maximum or minimum.

There are some changes that can be made to the algorithm to rectify this, though. To do this, you first have to assume that each loop state is, in effect, a terminal state.

Algorithm: MINIMAX(n):

```
if n is terminal or loop state:
        if utility(n)=='?':

          n.value=0

        else:

            n.value=utility(n)
        return

else if n is a MAX Node:

          max=min double value

          for succ in n.successors:
             val=MINIMAX(succ)
             if  val>max

                max=val

          n.value=max

else if n is a MIN Node:

          min=max double value

          for succ in n.successors:
             val=MINIMAX(succ)
             if val<min

                min=val

          n.value=min
```

I believe that this algorithm will return the optimal solution for all games with loops, provided that some conditions are true. It will specifically achieve this goal if the branching factor of the game tree is finite (since infinite paths are already handled by the algorithm), if both players are assumed to be playing optimally, and if an optimal solution to the game exists in the tree.

d:

I will prove this through a combination of direct proof/proof by construction.

The goal in this game is for each player to be the first to reach the other end of the array (the opposite index). In order to do this, logically, one player must jump over the other. For the winning player to actually win the game, this crossover must occur at a point where the winning player will be closer to their goal than the losing player.

If n is even, then the number of squares separating A and B is even (n=4 means 2 squares, n=6 means 4 squares, etc.) while if n is odd, then the number of squares separating A and B is odd (n=3 means 1 square, n=5 means 3 squares, etc.)

At the very beginning of the game, A always has the first move. A has to move to the next square right (because there's no other valid move). On B's first move, B has to move to the next square left for the same reason (if n=3, then B crosses over A into position 1 and wins the game). The number of squares separating the 2 are still even if n is even and odd if n is odd in this case.

Now, way for A or B to achieve their goal is to continue to move toward each other in order to have the opportunity to eventually jump over each other at a point where they are closer to their goal square than their opponent after the cross over. To get to this goal, at each of their respective moves/turns (when crossing over is not possible), A and B can choose to either move toward the other side or backward toward their own side. However, at any point, a player moving backward toward their own side (if it wasn't their only legal move) takes them farther away from their goal and leads the other player to continue moving toward them. This also ensures that the crossover (regardless of which player does the actual crossing over) occurs closer to that player's starting position than vice-versa, which enables the opposing player to ultimately win further down that particular path in the game tree. In short, it is not really optimal for a player to voluntarily move back toward their own side at any point in the game if all players are assumed to be playing optimally. Therefore, given the fact that each player plays optimally in the MINIMAX algorithm, they will almost never choose to move back toward their own side (unless it's their only legal move) prior to a crossing over.

So, for the sake of simplicity, we can assume that both players try to move toward each other on every turn (and if you draw out the game trees and assume no repeated ? paths are chosen, this happens to be the actual game tree path followed by the players according to the MINIMAX algorithm). Then, there are 2 cases:

Case 1:

If n is even and thus the gap between A and B is an even number of boxes. This means there are an even number of boxes on either side of the array's midpoint line/edge. There are an even and equal number of boxes on either side of this line/edge, and A and B are equidistant from it (in terms of boxes) Since A and B take turns moving, when moving toward each other, they will each traverse an equal and even number of squares. Eventually, A and B will end up on opposite sides of the midpoint line of the array. Then, it will be A's turn and A will jump over B and thus be one square closer to its goal than B (since it jumped 2 squares in the crossover), thus ensuring an A win down this optimal path.

Example: n=4: A starts at 1, B starts at 4 A moves to 2 B moves to 3 A and B are next to each other, and then it's A's turn and A jumps over B A has reached its goal

n=6: A starts at 1, B starts at 6 A moves to 2 B moves to 5 A moves to 3 B moves to 4 A and B are next to each other, and then it's A's turn and A jumps over B. B moves to 3 A can reach its goal in 1 move and win while B needs 2 moves to do so. A wins!

n=8: A starts at 1, B starts at 8 A moves to 2 B moves to 7 A moves to 3 B moves to 6 A moves to 4 B moves to 5 A and B are next to each other, and then it's A's turn and A jumps over B. B moves to 4 A can reach its goal in 2 moves and win while B needs 3 moves to do so. A wins!

Case 2:

If n is odd and thus the gap between A and B is an odd number of boxes. This means there is a midpoint box and an even number of boxes on either side of this box. There are an even and equal number of boxes on either side of this line/edge, and A and B are equidistant from it (in terms of boxes) Since A and B take turns moving, when moving toward each other, they will each traverse an equal and even number of squares. Then, A and B will move toward each other until A occupies the exact midpoint of the array and B is on the square to its right. At this point, it'll be B's turn and B will jump over A. B will thus be one square closer to its goal than A (since A will still be at the array midpoint while B will be at the square left of it after the crossover), thus ensuring an B win down this optimal path.

Examples: n=3: A starts at 1, B starts at 3 A moves to 2 B crosses over A to move to 1. B wins!

n=5: A starts at 1, B starts at 5 A moves to 2 B moves to 4 A moves to 3 B crosses over A to move to 2 B is now only 1 move away from its goal while A is 2 moves away. B wins!

n=7: A starts at 1, B starts at 7 A moves to 2 B moves to 6 A moves to 3 B moves to 5 A moves to 4 B crosses over A to move to 3 B is now only 2 moves away from its goal while A is 3 moves away. B wins!

As you can see, A wins if n is even and B wins if n is odd provided that both players play optimally.

Question 3

Note: For these problems, I'm assuming that we're looking for a global maximum (as in the TOP hill of an evaluation function graph)

a:

Hill climbing will work better than simulated annealing mainly when the random part of simulated annealing, which helps to prevent the algorithm from getting stuck in local maximums (assuming you are searching for the global maximum), is not needed. This occurs when a problem's state-space landscape graph with the states on its x-axis and the values of

the evaluation function for those states on its y-axis essentially takes the shape of a single hill/bell curve. A graph like this would have no local maxima (that are not also global maxima), just one global maximum. On a state-space landscape graph like this, there is no need to use the random part of simulated annealing, which is pretty inefficient, since hill climbing by itself will always find and return the global maximum (because there are no local maxima for it to get stuck in and return instead).

b:

Randomly guessing the state will work just as well as simulated annealing when the hill-climbing part, which helps the algorithm get to a local maximum, of simulated annealing is not necessary. This occurs, in my opinion, in 2 cases mainly. It can occur when a problem's state-space landscape graph with the states on its x-axis and the values of the evaluation function for those states on its y-axis essentially takes the shape of a flat line. It can also occur in the case in which the problem's cost function is not defined clearly, meaning that the area around the global maximum would look similar in value to the area around it. In both of these cases, the main feature of hill-climbing, which is moving states based on a neighboring state having a higher evaluation function value, is not needed, since the evaluation functions of the states are all equal or unclear.

c:

Based on the answers above, we can conclude that simulated annealing works best on problems which have clearly defined cost/evaluation functions for its states and have a state-space landscape graph with the states on its x-axis and the values of the evaluation function for those states on its y-axis that is not a straight flat line, is concave, and has a great deal of local maxima, plateaus, and shoulders. Problems with a clearly defined evaluation/cost function and a state-space landscape graph that is not a straight and flat line have "hills" in their state-space landscape graph. In order to reach the actual top of these hills efficiently, you would want to partially choose which state to move towards based on whether it gets you closer to the top, which the hill climbing part of simulated annealing does. However, using just hill climbing on problems whose state-space landscape graph include multiple "hills" of different sizes, local maxima, plateaus, and shoulders is not guaranteed to return a global maximum and is thus not complete. Using the random walk portion of simulated annealing combined with the hill climbing simulated annealing, however, makes it possible to prevent getting stuck at local maxima, shoulders, and plateaus while retaining the efficiency provided by the hill climbing portion of simulated annealing on such graphs. All in all, the 2 parts of simulated annealing complement each others' strengths and weaknesses to provide a complete and efficient solution to problems with he properties described above, and those are the kinds of problems it is designed for.

d)

There are some smarter things we can do to make the process of simulated annealing more accurate and/or efficient. One thing we could do is to keep track of a maximum value

throughout the algorithm. This value is initially set to null and keeps track of the maximum valued state visited during the algorithm. It would be updated every time the algorithm visited a state that was "better" than it. At the end of the algorithm, when the annealing schedule is reached, the algorithm could return this value instead of the current state, guaranteeing that the state returned would be the best state that the algorithm has actually traversed.

e)

If we had this much memory available, we can make one change to help the algorithm function more accurately, but at some loss for efficiency. We can store the states visited by the algorithm in memory. At each particular visited state, instead of having the algorithm pick only 1 neighbor to explore randomly, we can have it explore say 3 neighbors. We can then have run the simulated annealing algorithm on each one of these states as if the algorithm had picked each one of them as the next state to visit . We would store the results of the algorithm run on all 3 neighbor states for every single state the algorithm as a whole visits. Then, when this process is repeated for a certain forward depth (say 5 levels or termination, whichever comes first), you can backtrack through the "tree" of simulated annealing possibilities to find the best path to follow/neighbor to explore at each step, and then have the whole algorithm follow that path. Essentially, this modification to the algorithm would basically, as a result of the extra memory available, incorporate DFS and minimax (but for 1 player)-style comparison backtracking to determine the best path for the simulated annealing algorithm to follow at each level based on where picking that state is likely to lead to in the future of the algorithm. I believe that this modification will help simulated annealing perform better at finding the global maximum state in the problem's state space.

Question 4

a)

the variables are each of the cells in the 9x9 Sudoku grid, each cell has a domain of $\{1:9\}$. The constraints are: no repeated numbers in any row, column or 3x3 box.

b)

Start state: grid with M predetermined variables, chose a random predetermined variable to be $n_{i,j}$ the current variable
Successor Function: Use arc consistency to make inferences from $n_{i,j}$, change the domains of every unassigned variable on row i, column j, and the remaining variables in the 3x3 box $n_{i,j}$ is located in, remove $n_{i,j}$'s value from the domains of every variable visited, then assign a value to a neighboring value which will become $n_{i,j}$.
Cost Function: number of filled in cells
Heuristic Function: the minimum remaining value heuristic will work better than the

maximum degree heuristic for this problem. Every variable in this problem has similar degrees, combined with the arc consistency in the successor function, minimum remaining value will prove very helpful in deciding which neighbor to select. Maximum degree will make the program start with the values in the middle of the grid, which wont do much.
Branching Factor: the size of the domain of the current variable (Maximum of 9).
Size of state space: 9^{81-M}
solution depth you will be iterating through all 81 cells for a solution
max solution depth you will iterate over all 81 cells but you may backtrack n times

c)

in easy problems, M is larger, so using a Least Constraining Value heuristic, will allow you to quickly find a solution with less Backtracking

d)

function sudokuSolve(constraints,max_flips)
inputs: constraints: a set of constraints for each row column and 3x3 box,
max_flips: maximum amount of iterations
model $\rightarrow$ a randomly filled out Sudoku solution
for $i = 1$ to max_flips **if** model satisfies constraints then return model **otherwise:**
constraint $\rightarrow$ a randomly selected constraint from constraints that is marked false
for each variable in model that violates constraint
choose a value that minimizes the number of violations
endfor
endfor
return failure

Question 5

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

Part A

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

Opponent$\wedge Kryptonite \wedge Superman \implies Superman's Defeat$
$Kryptonite \implies Batman \wedge LexLuthor$
$Batman \wedge LexLuthor \implies UpsetWondenWoman$

$UpsetWonderWoman \implies Superman \land WonderWomanIntervention$

Part B

Transform your knowledge base into 3-CNF.

$\neg(Opponent \land Kryptonite \land Superman) \lor (Superman'sDefeat)$
$\neg(Kryptonite) \lor (Batman \land LexLuthor)$
$\neg(Batman \land LexLuthor) \lor (UpsetWondenWoman)$
$\neg(UpsetWonderWoman) \lor (Superman \land WonderWomanIntervention)$

Part C

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

Superman'sDefeat $\lor \neg Opponent \lor \neg Opponent$
$UpsetWonderWomen \lor \neg LexLuther \lor Batman$

Basically Batman cannot defeat Superman because he would need to collaborate with Lex Luthor, and by collaborating with Lex Luthor, he upsets Wonder Women, and finally Wonder Woman intervenes to help Superman, hence why Superman can be defeated.