CS440
2/9/22
Raisha Monga, Mini Sinha, Alex Giacobbe

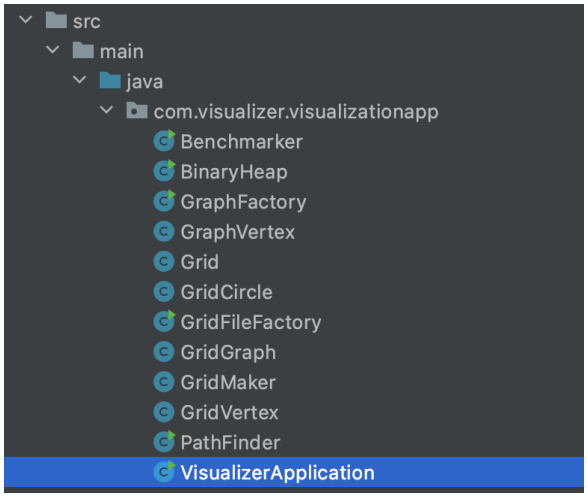
Problem 1: Any Angle-Path Planning

Part A:

The visualization was created with JavaFX. Before getting into the GridMaker class that is used to generate the grids in the Path Planning Visualizer, we'll first take a look at the supplementary class factories that were used to generate a random solvable 100 x 50 Grid.

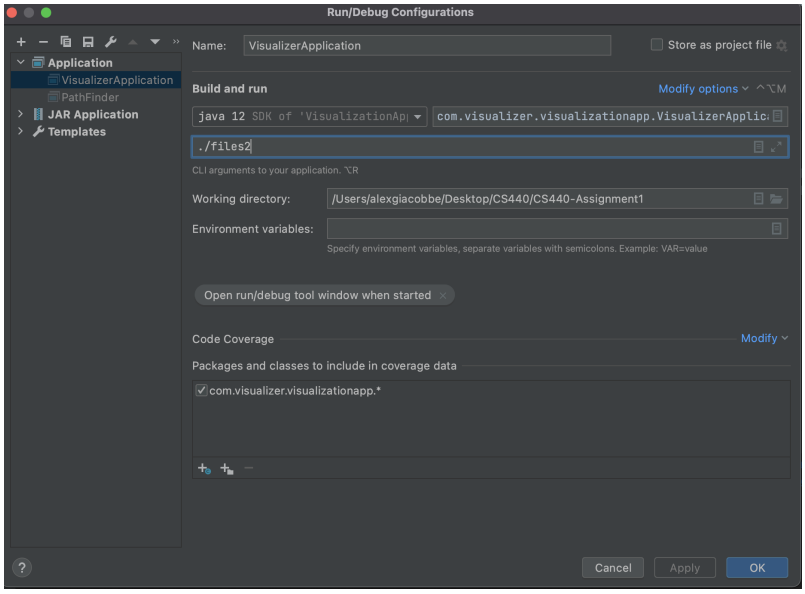
Run Directions (because we couldn't figure out how to create a JAR from a JavaFX project with Maven ¯_(ツ)_/¯)

1. Open the project in IntelliJ (available on the iLabs)
2. Right-click on src/main/java/com.visualizer.visualizationapp/VisualizerApplication



3. Select the "Modify Run Configurations..." option.\
4. A window should popup titled "Edit Run Configuration: VisualizerApplication"

In the program arguments, add the directory path to where you have your test grid files.



5. click Apply.
6. Close the dialog and then right-click again on VisualizerApplication.
7. Select the option "Run VisualizerApplication.main()" to run the application.

GridFileFactory

This is the class that generates the files used to create the grid. Given the rows and columns of the grids, a directory to save to, and the number of files to create, make() will run a method generateGridFile() for each file. 10% of the cells are blocked by randomly selecting an x coordinate in the array space. We evenly distribute the blocked cells per row to ensure a greater dispersal. Once the Grid is created as an object, we translate it to a graph as a 2d adjacency list. Breadth-First search[1] is run on the graph from the randomly chosen start and goal vertices. If BFS finds any path from start to goal, the grid passes and can be transferred into a text file.

GraphFactory

This class generates a graph from a grid object or from the provided file. The graph that is generated is what A* and Theta* operates on to find the shortest path. It is also used to ensure that there is a path from the start vertex to the goal vertex using BFS in the GridFileFactory.

The data structure is a 2d adjacency linked list, where at every index i,j is a linked list of neighbor vertices, each which are GraphVertex objects. The most difficult aspect of creating the Graph from the grid was how to know which neighboring vertices were blocked. The solution was the method **setNeighborsForGridGraph()**. Given a grid reference and the graph, we iterate over every index i, j. At each index, we created a LinkedList and using the grid reference, check if the index (i,j) (i-1, j) (i, j-1) and (i-1, j-1) are valid, and then check if the index is blocked and if it's (i+1, j+1) diagonal is valid. Valid checks are to check if the index being calculated exists within the grid. We can then add all of these

[1] Originally I implemented a recursive DFS to do this. It worked with smaller sized grids, but the 100x50 grids would cause a stack overflow. I left the implementation as part of the work.

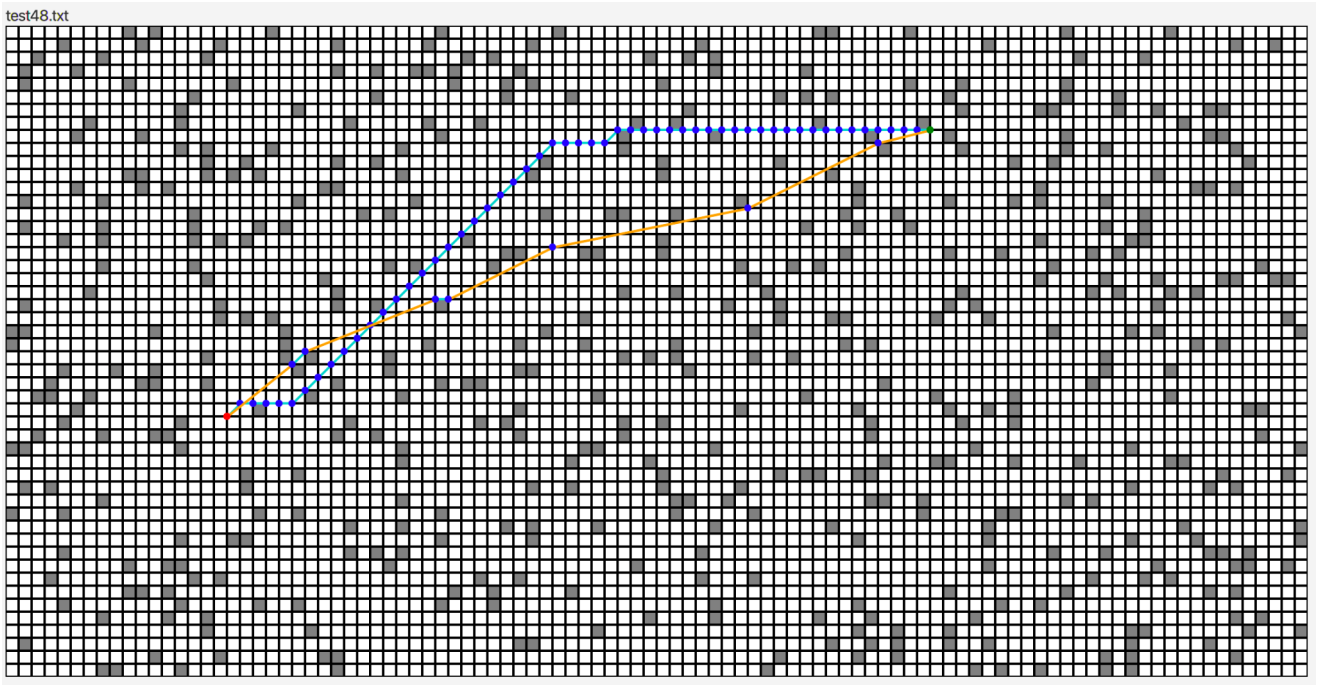
vertices to the LinkedList at that index, knowing that because if the index is unblocked and its diagonal is valid, it can reach all of these vertices. Once this procedure is completed, the graph data structure is created.

GridMaker
The GridMaker generates the visualization for each file in the directory passed into its constructor. For each file in the directory, we run two read operations - one to generate the visual base grid, and one to generate a graph to solve the a* and theta* path. Once these are completed, we draw the path from the graph onto the grid. The start vertex is represented by green and the goal is red. All vertices in between are blue. The a* path itself is a light blue color, and the theta* path (where it spans multiple vertices) is orange.

VisualizerApplication
Here is where we run the application. In the command line arguments, simply pass the directory path to the java file. When the application is run successfully, it will generate grids with the shortest path visualized. Clicking on the vertices in the path will give you the calculated g(n) h(n) and f(n) values in the upper left corner.

Possible Optimizations
In the GridMaker make() method, we are reading from the file twice, once to make the grid and once to make the graph and solve it. An improvement would be to read from the file once, and then create the grid and solve the path asynchronously on different threads. Also, to improve memory usage, these two separate instances of a Graph and the Grid could be combined into one object.

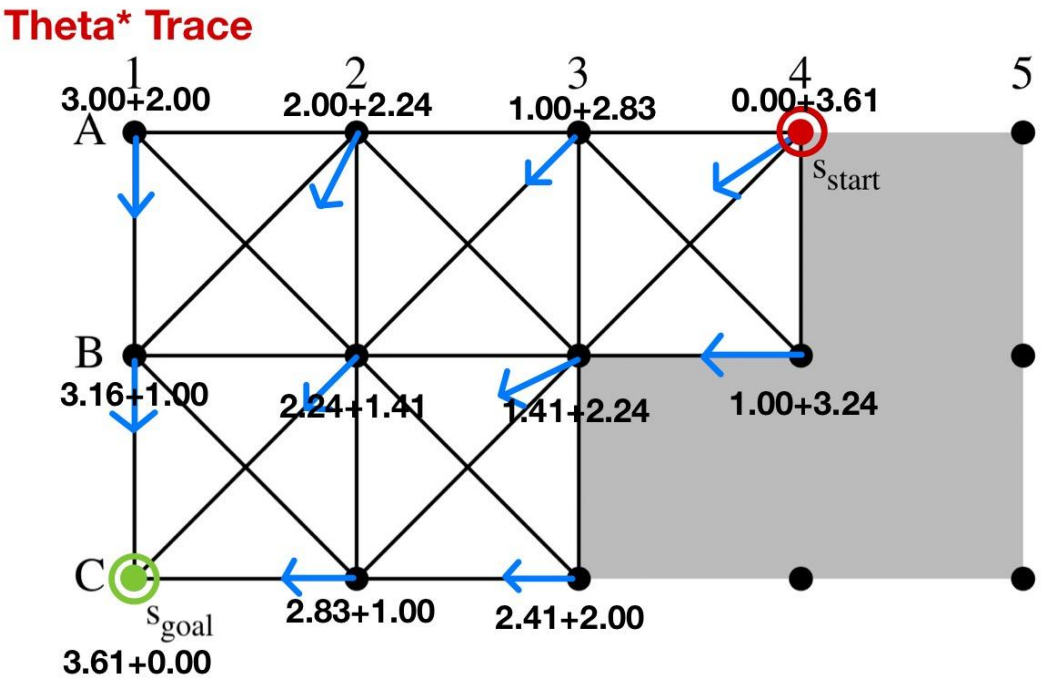
Other **main()**s
We left the other main methods in some of the classes to document how we tested.



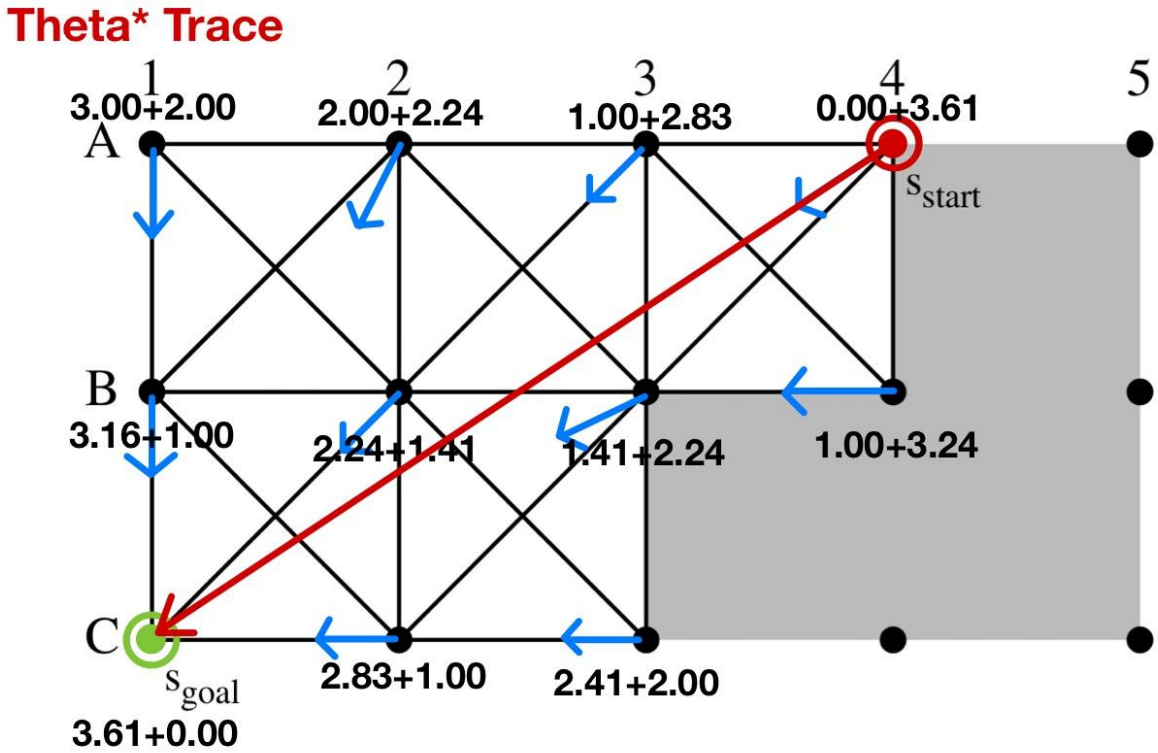
Example 100x50 Grid with A* and Theta* path planning.

Part B:

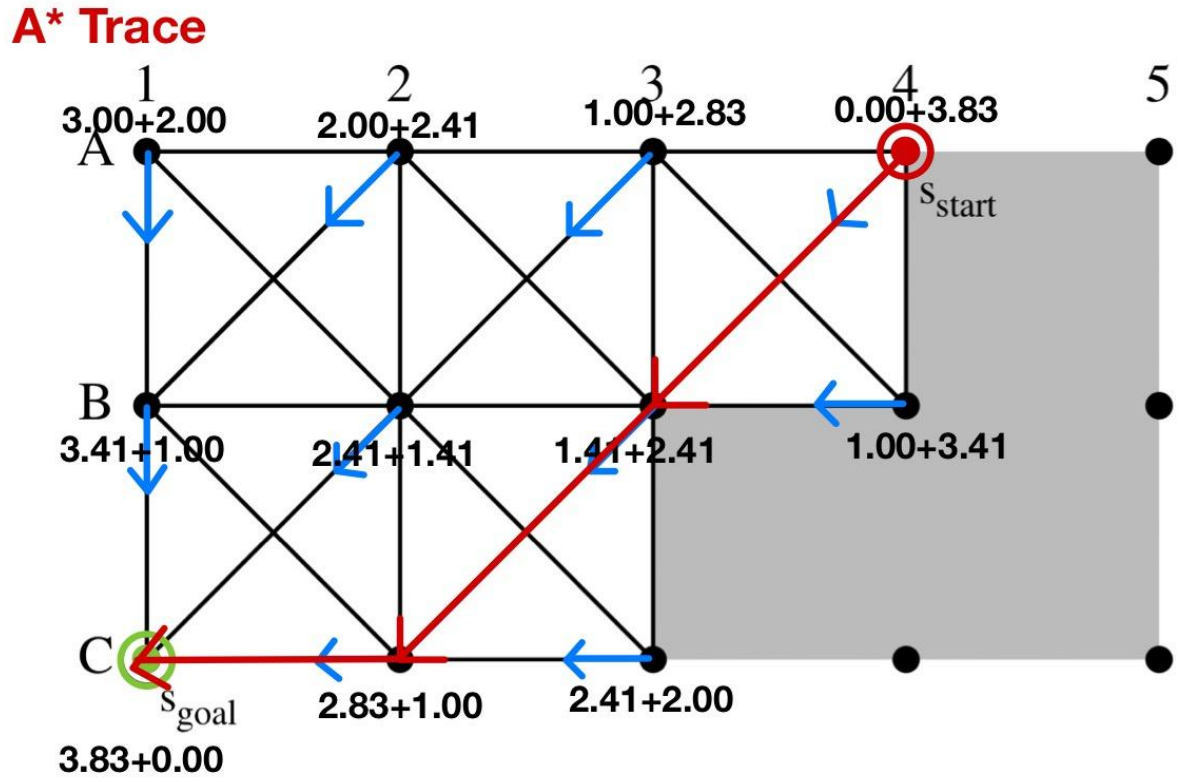
Theta* Trace:



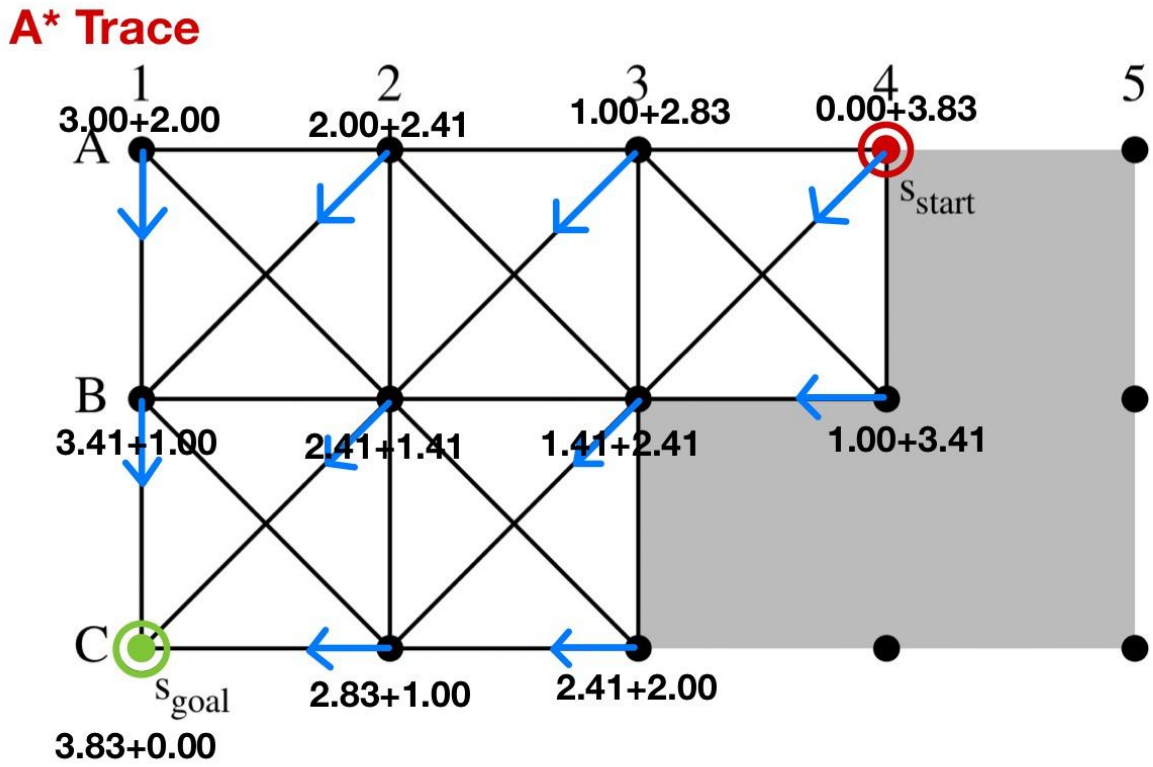
Theta*: Shortest Path:



A*: Shortest Path



A* Trace:



Part C:
For the implementation of A* We used the pseudocode pretty much every step of the way. For us to implement both A* and theta*, we had three data structures, a priority queue called _fringe, an arrayList called _pathList, and another arrayList named closed (to represent a closed list for the A* algorithm) and a gridGraph called _Graph. The GridGraph is an object which represents the grid. It includes rows, columns, start and goal vertex. _Graph is the object that a* runs on and it is a 2d array with a linked list of neighbors at every index. _pathList is the list that we return which includes the path that we return from start to goal in reverse order[2].

Initially, we get our start and goal by using the getter method we defined in GridGraph.java. Next, we set our initial G value to 0 because from the start, the g value is 0. Next we calculate the heuristic value by using the formula provided in the pdf. The specific method to calculate the heuristic formula is in GraphVertex.java. We then add the starting vertex to our fringe. Now, after adding the starting vertex, we can simply just have a loop that runs until the fringe is empty. At the end of this loop we return pathList. Regardless of us finding a path or not, we always return pathList because if we don't find a path between two vertices, we just return an empty list which represents that there is no path between our start and goal vertex. If the fringe is not empty, we will pop the current vertex and check if it equals the goal. If the current vertex is equal to the goal, we add the vertices to pathList by calling the pathFromGoal method and we return the pathList. However, if the current node that we look at is not the goal, we add it to the closed list and go through all of its neighbors. For each neighbor, we check if it's in the closed list and if it's in the fringe. If it's not in the fringe, this is the first time we've seen this vertex, so we update the G value to infinity to ensure that the next time we call updateVertex the current vertex's G value plus the distance to its neighbor will be less than the g Value of the neighbor (which is infinity). We make sure to calculate the heuristic for the neighbor node (in the event that it hasn't been calculated yet) and call updateGValFromFringe()[3] and then call updateVertex.

UpdateVertex gets called on the current vertex and its neighbor that we are currently iterating over. If the current vertex's G value + the straight line distance to the neighbor is less than the neighbors G Value, we recalculate the neighbors G value to be the current vertex G value plus the straight line distance. This guarantees the shorter path is chosen. Then if the neighbor is in the fringe, we remove it, and then add it back with the new g value. This process continues until either all nodes are closed or the goal vertex is found, in which case we run pathFromGoal() to walk back from the goal up the parent chain to the start vertex and return _pathList.

Part D:
For the implementation of theta*, we extended our implementation of a*. If you see in our pathfinder.java, we have a method called aStarPath which takes in a parameter called isTheta. If isTheta is true, updateVertex is changed to use updateThetaVertex. The key difference between a* and theta* is that theta* allows a vertex to be a parent of any other vertex within line of sight, while A* only allows a neighbor to be the parent of the vertex. For this reason, we implemented our theta* in the same class as

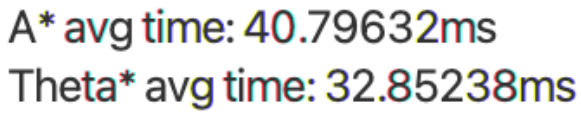
---

[2] The pathlist is in reverse order because we start from the goal and work our way back from each vertex to its parent.

[3] Because our implementation of GridGraph stores copies of neighbor vertices in lists, the G Value for neighbor nodes for a new vertex will always be 0. We can update its G Value for this copy by checking the fringe if it exists. It's a definite performance hit and a future optimization would be to use pointers instead of copies, so the data is up to date always.

A*. The only difference was in our updateThetaVertex method. UpdateThetaVertex uses a method called lineOfSight. LineOfSight is a boolean method that takes in two vertices as its parameters. Next, we get the x and y coordinates of the two vertices. Next, we get the difference between the x and y coordinates of the two vertices. Based on the difference of the x and y coordinates we set the offset for the x and y coordinates. This is important because the offset allows us to check if the current cell that we are going to is blocked. UpdateThetaVertex calls on the lineOfSight method to check whether or not there is a path from the start to goal despite there being blocked cells. If the lineOfSight method returns true, we go to updateVertex with the current vertex's parent and the next node, where we calculate the G value, distance, and update the shortest path to use the next nodes parent has the current vertex's parent.

Benchmark times for A* and Theta* (screenshot from application).
The averages are from 50 random 100x50 grids generated by GridFileFactory.



Part E:
Assume h is admissible.
Let n be a node such that n.State = goal and n is not on the optimal
path to the goal,
Let n' be a node on the optimal path,
Then A∗ expands n' before n.

Since A* expands the path with the underestimated cost, it will always expand the most optimal path and ignore the non-optimal path. The optimal path with the goal has a lower underestimated path cost compared to the non optimal path with the goal, which has a higher underestimated path cost.

Part F:
For the binary heap, we had to implement ArrayList in such a way that it acts as a priority queue. For this program to be possible I first imported arrayList, serializable, and the comparator libraries. I made the class Binary heap of type E which extends comparable and a super class. To create the binary heap I created three array lists, Listkey, ListIn and ListOut. Listkey is the only list that holds the actual values whereas listIn and listOut include the indices of that value. I also included a boolean called heapMin which I would use later on in different methods throughout the file. The most important methods I would use it in was in goingUp (recursive method that would find the parent of the node and keep finding the parent until root is reached to find the minimum), smallerNodeIfMin (to compare two indices with the size of listKey and each other to find the minimum heap) and on goingDown(a recursive method that goes down to find smallest child by going down in the list). Next, I have a private int method that compares two objects of type E which is the same type that the binary heap is. The next method I have is a constructor to initialize the heap. The next method I have is a binary heap method that makes sure that the three lists have enough capacity to add all the elements to it and copies all the items that are in arr and its indices into the three lists. The next method I have is increaseCap which increases the capacity of the

heap to make sure that there is enough space in our three lists without having to worry about reallocation. Next method I have is add which returns the value of the index it is handling within the three keys. The add method also calls on going up to basically rebalance the heap after inserting a new value into the three lists. Next, we have the remove method, which first checks if the value that we are looking to remove is in listKey and then if it is then it retrieves the index, removes it from the three lists and then calls on going down to rebalance the tree. The next method contains which simply just checks if the value we are looking for is in the listKey and returns a true or false within the method. Next, I have a comparator constructor. The next method I have is doingHeap which is just a method that is similar to heapify, it goes through half of the listKey and then goes down half of the listKey list to rebalance it. Next, I have the going up method which basically climbs up the tree to find the compare the minimum and rebalance the tree. The next method I have is dataExchange which simply just swaps two values with each other's position in the three lists. The next two methods (smallerNodeIfMin and goingDown) have been explained above. The next method I have is PopMinIndex which simply returns the index of the minimum lament within the three lists and then rebalances the tree. Next method I have is removeMin which simply removes the minimum element in the list and then rebalances the tree by calling goingDown. Next I have a simple toString method to get the string representation of the contents of the three lists. Next, I have the method daddy which simply just returns index of the parent of a child node, childLeft returns the left child of a node and childRight returns the index of the right child of a node and then I have an isEmpty method just checks if the method is empty and finally I have a main method used to check if my method works.

Part G:

In doing the optimization, I quickly realized that there was a mistake made in the GridFileFactory class. Upon choosing the random start and goal vertices, I was allowing indices of 0 to be chosen, which in the file format doesn't exist. This was causing a small percentage of created grids to have out of bounds start and goals. The BFS was not catching this case - which I did not have time to look further into. The generated grids were unsolvable, causing the pathfinder to enumerate every possible path and causing the mean benchmark time to increase dramatically. After fixing the GridFileFactory, my times (unoptimized) were as follows:

A* avg time: 10.94626ms
Theta* avg time: 3.31424ms

For the optimization, the main issue was the data structures for the GridGraph. The _vertices 2d array was a linked list of neighbors, each with its own set of instances of graph vertices. That meant that anytime a new vertex was explored and its neighbors were visited, we would need to recalculate the G value. This was done by getting the vertex from the fringe, and updating the current instance's G value. With the priority queue, this meant I had to pop out all the vertices from the fringe, update the one I needed, and add them back.

To fix this issue, I added a wrapper class for the GraphVertex, called VertexInfo, which became the object in the 2d array. It consists of a single GraphVertex that represents the current vertex on the grid, and an array list of neighboring vertices. That single vertex would contain the GValue and HValue used in the A* and Theta*, allowing us to remove the GVal recalculation. Another small optimization with our Priority

Queue implementation was to simplify the Binary Heap class to only use what we needed for our Pathfinder. Below are the average benchmark times.

No optimizations:

A* avg time: 10.94626ms
Theta* avg time: 3.31424ms

Optimized with our Priority Queue BinaryMinHeap:

A* avg time: 1.07012ms
Theta* avg time: 1.26246ms

Optimized with Java's Priority Queue:

A* avg time: 0.80104ms
Theta* avg time: 1.0484ms

BinaryMinHeap (changes)

We remade our priority queue implementation as BinaryMinHeap. This solution was to update the old implementation to match the methods used by Java's Priority Queue to make it easy to switch. Our PriorityMinHeap uses one ArrayList<E> with a comparator to organize the heap.

The add() method adds an element to the end of the list. Then, using the index of the added element, we loop through the parents (a parent node is located at the [i - 1] / 2 index) and check if the element is greater or equal to its parent's value. If it is, the element is in the right spot, otherwise we swap() the element to its parent location, and the parent becomes the child.

The poll() method returns the element at the top (or front) of the heap. The way that we did this is by swapping it with the last element (max element) in the list, and then removing it from the end of the list. Then we run heapify() on the top element of the heap. Since it's the max element, this element will move back down through the heap, getting swapped with the smaller values along the way, rebalancing the heap.

The remove() method removes an element from the heap at the provided index. It is the same procedure as poll() with the exception of removing an element at any index. We swap() the element with the last element, remove it, and then heapify() the swapped element to rebalance the tree.

The contains() method simply uses the array list contains() method to check if an element exists in the list.

The heapify() method compares an element at a provided index with its children (2i + 1 and 2i + 2) if its larger than its children (starting with the left one) we swap() it with that child and then recursively call heapify() on its new position. Heapify() is a private method used to balance the tree.

The isEmpty() method checks if the heap is currently empty using the ArrayList isEmpty() implementation.

Part H:

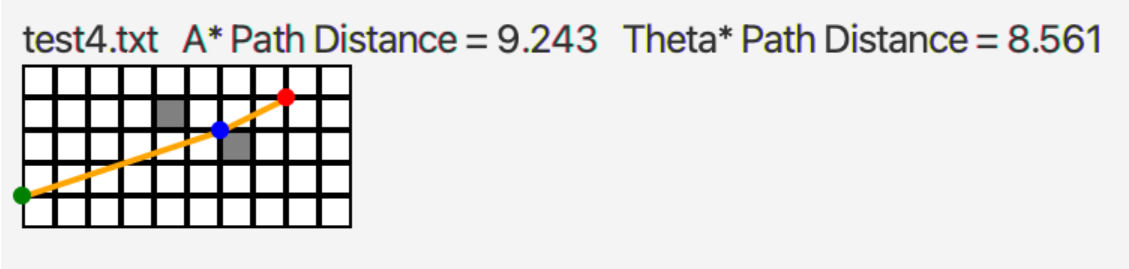
The A* and Theta* h-valuesIn order to get A* path lengths and Theta* path lengths, we used the resulting

path to calculate the distance between each vertex. In all of our test grids, A* ultimately had a longer path distance. In the average run time case, A* was quicker. I believe this is partially due to how the heuristic is calculated, which is simple comparison and addition/subtraction, whereas Theta* uses the distance formula to calculate the exact distance between two points, which requires getting the square of Δx and Δy, and then getting the square root of the sum. In addition to being more computationally heavy, lineOfSight(s) in Theta* loops through vertices to find the longest line of sight to the goal.

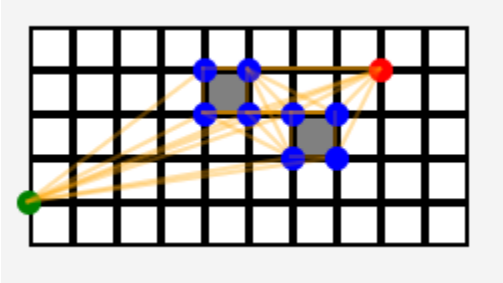
It is fair that A* and Theta* use different h-values. In both cases, a more accurate heuristic will result in better outcomes. Calculating the distance between two vertices for Theta* is more accurate, since the path between vertices follows line of sight, which is equivalent to the straight line distance between the two points, thus it makes sense to use a different heuristic value.

Part I:

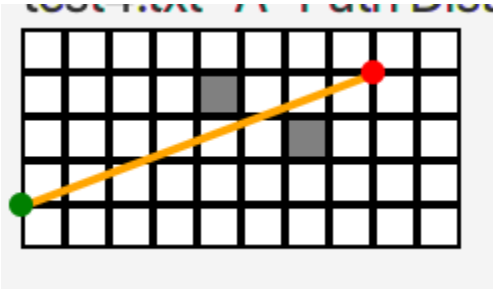
Below is a screenshot from the visualizer application using a grid that has h-values longer than the minimal path, resulting in a longer than minimal path. The minimal path length (which is a straight line) in this case should be approximately 8.55, but the Theta* path is slightly longer with a difference of .011. This happens because "...a vertex p can only become the parent of another vertex s if either p is a neighbor of s or p is the parent of a neighbor of s." (Daniel & Nash & Koenig, 2010, pg. 3).



Theta* path



Visibility Graph for test4.txt

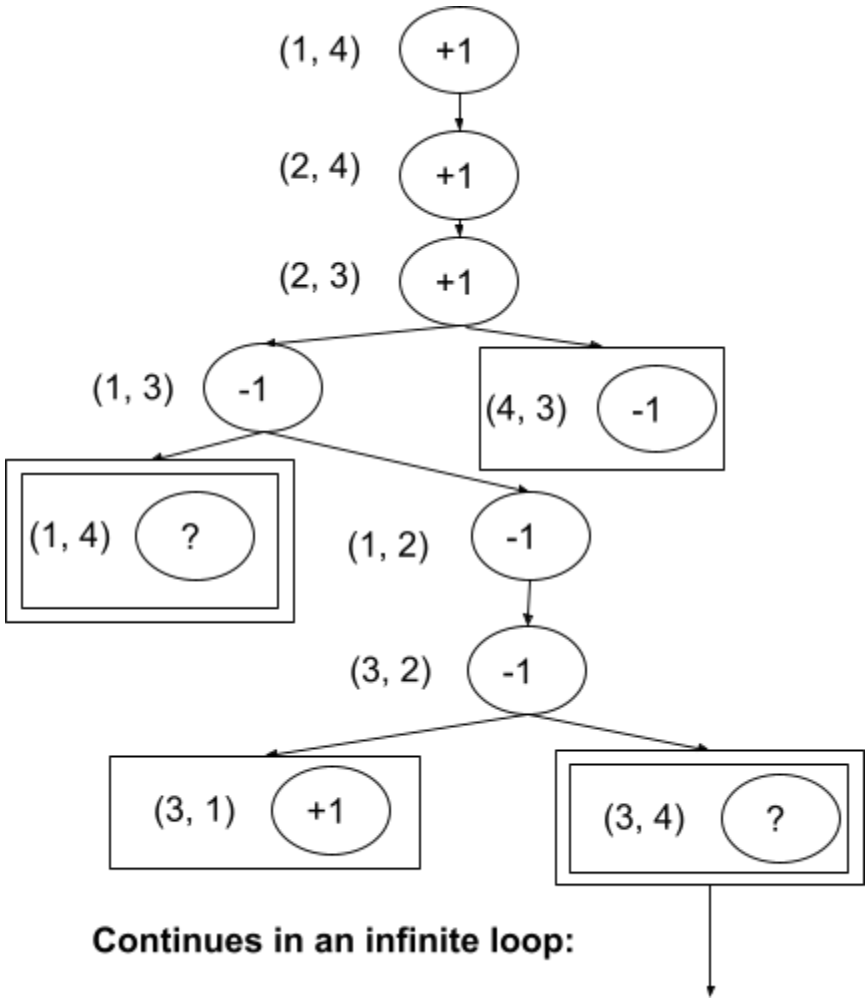


True minimum distance path

Visibility Graph Implementation

In the Visualizer Application, we included a visibility graph for the first grid to be solved. For implementing the visibility graph, we simply collected the start vertex, goal vertex, and all vertices that are corners to blocked cells in an ArrayList<VertexInfo>. Each VertexInfo object holds an ArrayList<GraphVertex> of "neighbors", which will be any other corner vertex that is blocked in the direct line of sight of the current vertex (or the start and goal vertices). In order to find those in direct line of sight of a vertex, for each vertex in ArrayList<VertexInfo>, iterate over the ArrayList<VertexInfo> and use the LineOfSight(s, s') algorithm provided for Theta*. If it is in line of sight, add it to the "neighbors" list. Then for the display, we modified our line drawing algorithms for tracing the path to consume an ArrayList<VertexInfo> and iterate over the list and each VertexInfo neighbor list and draw the line segments and place the vertices on the grid.

Problem 2:



a.

b. For the states where both A and B are in a position they were previously in simultaneously, the value is ?.
c. The standard minimax algorithm recursively searches through the game tree to find the most optimal move for a player; however, this game will run on an infinite loop, and the standard minimax algorithm will fail on this game tree.
d. As shown in the example above, A reached its goal state in n/2 turns, given n is even. A reaches its goal state in n/2 tries when n is even, and B reaches its goal state in n/2 tries when n is odd.

Problem 3:

a. Since hill climbing cannot go downhill or move off of local extrema, but simulated annealing does, hill climbing would be favorable when there is no local extrema.
b. Hill climbing part of simulated annealing would not be necessary when all the neighboring states are of around the same value. Since the values are similar, there is no hill to climb
c. Simulated annealing is best used in situations that have local extrema and require random walk in the beginning.
d. You can keep track of each state that you have visited and track the best states(closest to your goal state) you have already visited. Therefore, instead of going back to a state that may not be

the best state and starting over the annealing schedule at an inefficient state, you can restart the annealing schedule at a state that will reach the goal state quicker.

e. One way to perform simulated annealing with two million states without performing simulated annealing a million times is to use a data structure (such as array, linked lists, or hashmaps) to store the current state after every annealing, and to pass that information to the next annealing, but within one simulated annealing.

Problem 4:

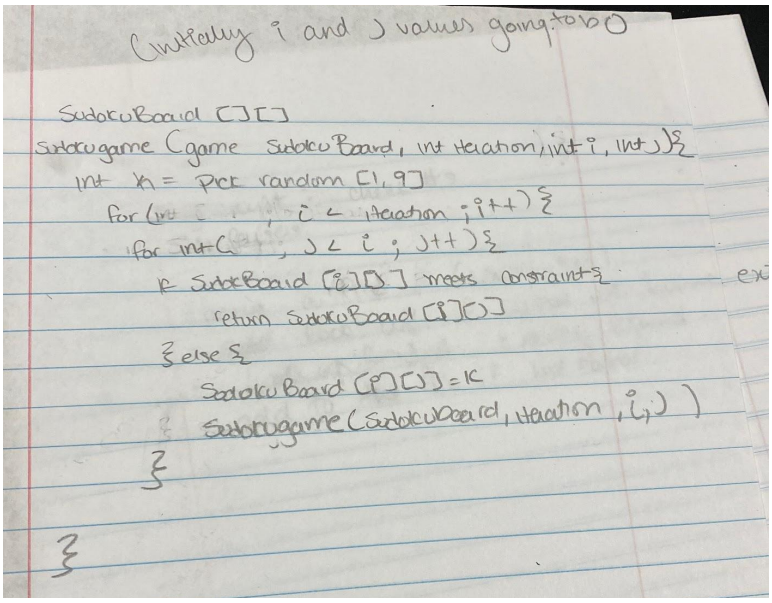
Consider the game Sudoku, where we try to fill a 9 × 9 grid of squares with numbers subject to some constraints: every row must contain all of the digits 1,...,9, every column must contain all of the digits 1,...,9, and each of the 9 different 3× 3 boxes must also contain all of the digits 1, . . . , 9. In addition, some of the boxes are filled with numbers already, indicating that the solution to the problem must contain those assignments. Each game is guaranteed to have a single solution. That is, there is only one assignment to the empty squares which satisfies all the constraints. For the purposes of this homework, let's use n,j to refer to the number in row , column j of the grid. Also, assume that M of the numbers have been specified in the starting problem, where M = 29 for the problem shown above.

a. This is an instance of a Constraint Satisfaction Problem. What is the set of variables, and what is the domain of possible values for each? How do the constrains look like?
   i. Set of variables:
      1. One variable for each empty or full Square $S_{(i,j)}$ where i is the row and column is j
   ii. Set of domains: {1,2,3,4,5,6,7,8,9}
   iii. Constraints:
      1. The variables of each row cannot contain any overlapping values
         a. $S_{(1,j)} \neq S_{(2,j)} \neq S_{(3,j)} \neq S_{(4,j)} \neq S_{(5,j)} \neq S_{(6,j)} \neq S_{(7,j)} \neq S_{(8,j)} \neq S_{(9,j)}$
      2. The variables of each column cannot contain any overlapping values
         a. $S_{(i,1)} \neq S_{(i,2)} \neq S_{(i,3)} \neq S_{(i,4)} \neq S_{(i,5)} \neq S_{(i,6)} \neq S_{(i,7)} \neq S_{(i,8)} \neq S_{(i,9)}$
      3. The variables of each box in each 3x3 block cannot contain any overlapping values
         a. $S_{(1,1)} \neq S_{(1,2)} \neq S(_{1,3)} \neq S_{(2,1)} \neq S_{(2,2)} \neq S_{(2,3)} \neq S_{(3,1)} \neq S_{(3,2)} \neq S_{(3,3)}$

b. One way to approach the problem, is through an incremental formulation approach and applying backtracking search. Formalize this problem using an incremental formulation. What are the start state, successor function, goal test, and path cost function? Which heuristic for backtracking search would you expect to work better for this problem, the degree heuristic, or the minimum remaining values heuristic and why? What is the branching factor, solution depth, and maximum depth of the search space? What is the size of the state space?
   i. Start state: 52 empty cubes in a visualization of total 81 cubes where 29 are filled
   ii. Successor function: fill the neighboring box of the current box with a number between 1-9. Neighbors being empty cells near the current cell (top, bottom, left, right)
   iii. Goal test: all cubes are filled with the constraint being met where every column contains all of the digits (1-9) and each of the 9 different 3 x 3 boxes also contain all of the digits 1-9
   iv. Path cost function: each path costs 1 unit (?)

v. Heuristic for backtracking: since all possible values will eventually decrease and succumb to 0, the minimum remaining values heuristic would work better
vi. Branching factor: 9
vii. Solution depth: 52 (because 52 empty cubes on the starting board)
viii. Maximum depth: 52
ix. Size of state space : 9^52
    1. Branching factor of the search problem is 9
    2. The solution depth is 52
    3. So the size of the search space is b^d
        a. 9^52

c. What, is the difference between “easy” and “hard” Sudoku problems? [Hint: There are heuris- tics which for easy problems will allow to quickly walk right to the solution with almost no backtracking.]
    i. The difference between an easy and hard Sudoku problem is the numbers given in the sudoku problem. The fewer numbers that are included within the sudoku problem the more logical decisions are required and less immediate solvable clues are available during progression making it more probable for us to encounter a bottleneck situation which would force us to backtrack and which in the end makes it harder for us to solve the problem.

d. Another technique that might work well in solving the Sudoku game is local search. Please design a local search algorithm that is likely to solve Sudoku quickly, and write it in pseudo- code. You may want to look at the WalkSAT algorithm for inspiration. Do you think it will work better or worse than the best incremental search algorithm on easy problems? On hard problems? Why?
    i. We would first go to a the first empty box and assign it to a random value between 1- 9 , then we would loop through the whole sudoku placements for each square in row i and col j, we would first assign it to a random value between 1-9 and then check if the assignment satisfied the constraints. However, if there's a placement that does not satisfy the constraint we assign that to a different number.
    ii. 

iii. This algorithm is worse than the best incremental search algorithms in both easy and hard problems because the algorithm given works best when a legal move is given so there is no alternative answer for that row, column or table. If there are multiple answers, the code will continue even though there is a logical error present forcing it to backtrack in later steps when there are no other possible moves. The need for backtracking is what makes the algorithm inefficient

Problem 5:

Consider the following sequence of statements, which relate to Batman's perception of Superman as a potential threat to humanity and his decision to fight against him. 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.

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

- One thing to note that the opponent that we are talking about here is batman so:
- Superman is defeated if he is facing an opponent alone and his opponent has a kryptonite
    - Superman defeated => Facing opponent alone Opponent carrying kryptonite
        - SD => O BK
- Acquiring Kryptonite, means that Batman has to coordinate with Lex Luthor and acquire it from him
    - Batman Acquire Kryptonite => Batman Lex Luthor
        - BK => B L
- If batman coordinates with Lex Luthor, Wonder woman is upset
    - Coordinate with Lex Luthor => Wonder woman upset
        - B L => WU
- If wonder woman is upset, she will intervene and fight on side of Superman
    - Wonder Woman Upset => Superman and Wonder Woman fight together
        - WU => S W
- If wonder woman fights on the side of superman, superman cannot be defeated
    - Superwoman and wonder woman fight tother => ¬ Superman Defeated
        - S W=> ¬ SD

b. Transform your knowledge base into 3-CNF.

- Clauses from 5a. Combined:
    - (S O BK) ∨ (¬ SD)
    - ¬ BK ∨ (B L)
    - ¬ (B L) ∨ WU
    - ¬ WU ∨ (S W)
- The 3-CNF form of the clauses

- ¬S ∨ ¬O ∨ ¬BK ∨ SD ¬B ∨¬L ∨ WU

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

- Using formula p =>q = ¬p ∨ q we can eliminate multiple implications
  - SD => O BK = ¬SD ∨ (O BK)
  - BK => B L = ¬BK ∨ (B L),
  - B => WU = ¬(B L) ∨ WU,
  - WU => SW = ¬WU ∨ SW,
  - SW =>¬SD = ¬ SW∨ ¬(SD),
- The first statement can be further simplified into through the use of distribution law
  - ¬SD ∨ (O BK) = (¬SD ∨ O) (¬SD ∨ BK)
- The CNF Form now looks like
  - (¬SD ∨ O) (¬SD ∨ BK)
  - (¬BK ∨ (B L))
  - (¬(B L) ∨ WU)
  - (¬WU ∨ SW)
  - (¬ SW∨ ¬SD)
- We can now turn the first statement (¬SD ∨ O) (¬SD BK) into a clause as it is now in CNF form
  - (¬SD ∨ O) , (¬SD ∨ BK)
- We can now eliminate literals with their negations through the use of resolution inference
  - (¬SD ∨ O) , (¬SD ∨ BK) , (¬BK ∨ (B L)) = (¬SD ∨ O), (¬SD ∨ BK) , (¬BK ∨ ¬O)
  - (¬SD ∨ (O BK)), (¬(O BK)
- Simplifying the statement we get (¬SD) which proves that superman is undefeated

Works Cited:

https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated2/koenig_thetastart_aaai07a.pdf