Assignment 1

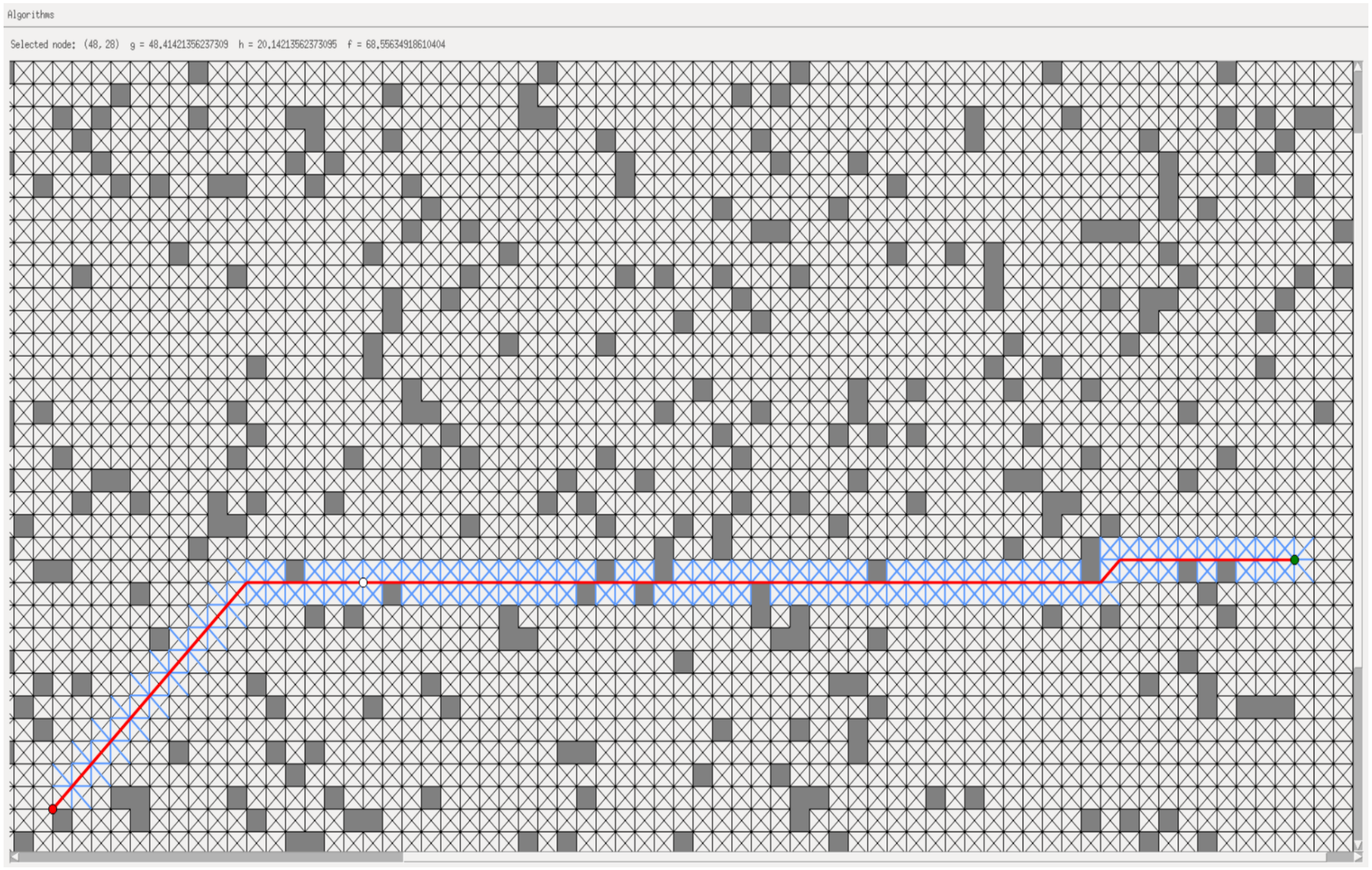
Problem Solving

Christopher Naporlee (cmn134), Michael Nelli (mrn73), Timothy Walker (tpw32)

Problem 1

a. Overview:

- Our UI displays a 100 x 50 grid with blocked cells grayed out. At the top of the program, we allow the user to select which path-finding algorithm they want to choose, along with the h, g, and f values of the selected node in the grid. As the algorithm runs, it highlights searched edges in blue, and draws the final path in red.

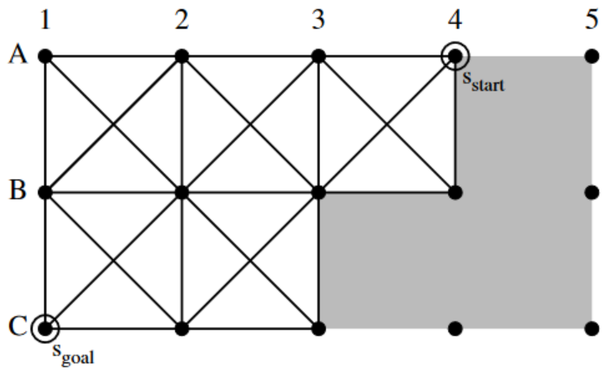


Selected node: (48, 28) g = 48.41421356237309 h = 20.14213562373095 f = 68.55634918610404

- Capabilities:
  - Displays start (green) and end (red) positions
  - Highlights searched edges
  - Highlights final path
  - Scrollable canvas to see the grid in parts
  - Auto-scroll when searching to follow the path
  - Drop-down menu to change path-finding algorithm
  - Selectable corners to visualize the corresponding node's h, g, and f values

- Implementation:
  - We created a Window class that used the Tkinter library for all visualizations.
  - To create the graphs, we have a gen_graph script that creates an 8-neighbor graph with 10% randomly chosen blocked nodes and stores the information into a file.
  - For the graph structure, we have a Graph class along with an Edge and Node class. The graph is initiated with a generated graph file. The window then draws lines by iterating through all of the edges of the graph. If a node read from the generated graph file is marked as blocked, then we create a gray square for that tile, with the blocked node serving as the top left corner of the cell.
  - To draw the paths in “real time,” we added a delay to each draw call using the after() Tkinter function. This prevents everything from being drawn instantaneously.

b. A* and Theta* Manual trace for Figure 7



A* Table

🟨 = Node being explored

Step	Frontier	Closed	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3
0	[A4]	[]	N/A	N/A	Parent: A4 g(s): 1 h(s): 2.83 f(s): 3.83	■	N/A	N/A	Parent: A4 g(s): 1.41 h(s): 2.24 f(s): 3.65	Parent: A4 g(s): 1 h(s): 3.16 f(s): 4.16	N/A	N/A	N/A
1	[B3, A3, B4]	[A4]	N/A	Parent: B3 g(s): 2.83 h(s): 2.41 f(s): 5.24	Parent: A4 g(s): 1 h(s) 2.83 f(s): 3.83	DONE	N/A	Parent: B3 g(s): 2.41 h(s): 1.41 f(s): 3.83	■	Parent: A4 g(s): 1 h(s): 3.16 f(s): 4.16	N/A	Parent: B3 g(s): 2.83, h(s): 1 f(s): 3.83	Parent: B3 g(s): 2.41 h(s): 2 f(s): 4.41
2	[C2, A3, B2, B4, C3, A2]	[A4, B3]	N/A	Parent: B3 g(s): 2.83 h(s): 2.41 f(s): 5.24	Parent: A4 g(s): 1 h(s): 2.83 f(s): 3.83	DONE	Parent: C2 g(s): 5.24 h(s): 1 f(s): 6.24	Parent: B3 g(s): 2.41 h(s): 1.41 f(s): 3.83	DONE	Parent: A4 g(s): 1 h(s): 3.16 f(s): 4.16	Parent: C2 g(s): 3.83 h(s): 0 f(s): 4.83	■	Parent: B3 h(s): 2.41 g(s): 2 f(s): 4.41
3	[B2, A3, B4, C3, C1, A2, B1]	[A4, B3, C2]	Parent: B2 g(s): 3.83, h(s): 2 f(s): 5.83	Parent: B3 g(s): 2.83 h(s): 2.41 f(s): 5.24	Parent: A4 h(s): 1 g(s): 2.83 f(s): 3.83	DONE	Parent: B2 g(s): 3.41 h(s): 1 f(s): 4.41	■	DONE	Parent: A4 g(s): 1 h(s): 3.16 f(s): 4.16	Parent: C2 g(s): 3.83 h(s): 0 f(s): 3.83	DONE	Parent: B3 g(s): 2.41 h(s): 2 f(s): 4.41
4	[C1, A3, B4, C3, C1, A2, B1]	[A4, B3, C2, B2]											

- At step 2, choose C2 over B2 and A3 because C2 has larger g value
- At step 3, choose B2 over A3 because B2 has larger g value
- During this step we check if B2 has a better path to C1 than C2. It doesn't, so we do not update it.
- At step 4, return path found since we pop the goal off the queue

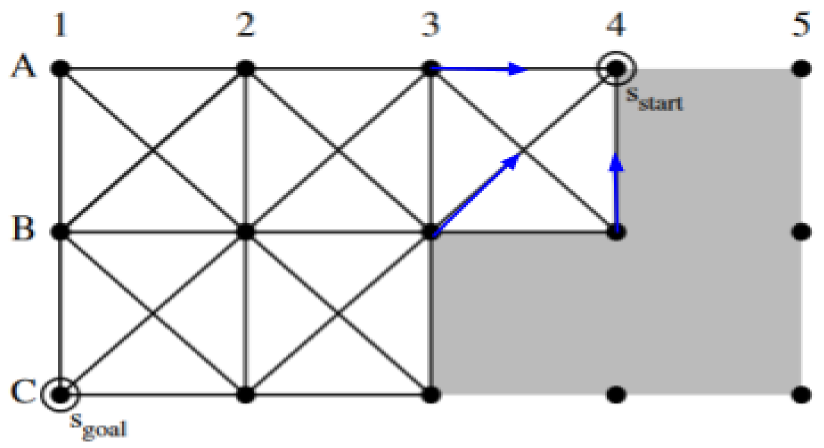
Theta* Table

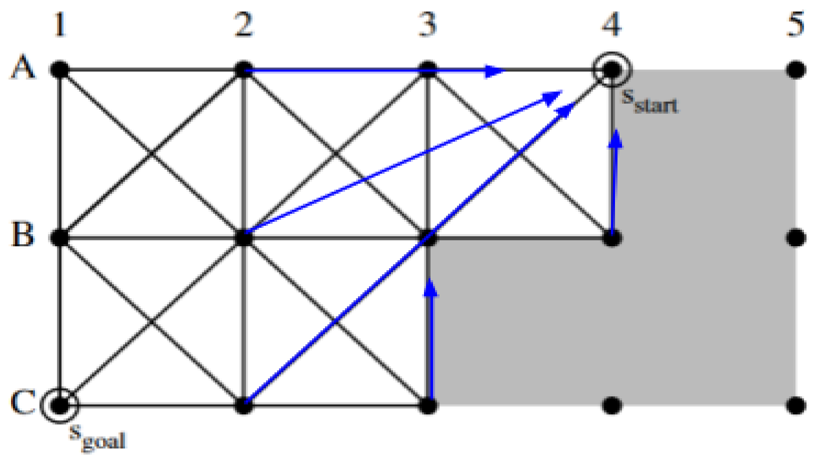
🟨 = Node being explored

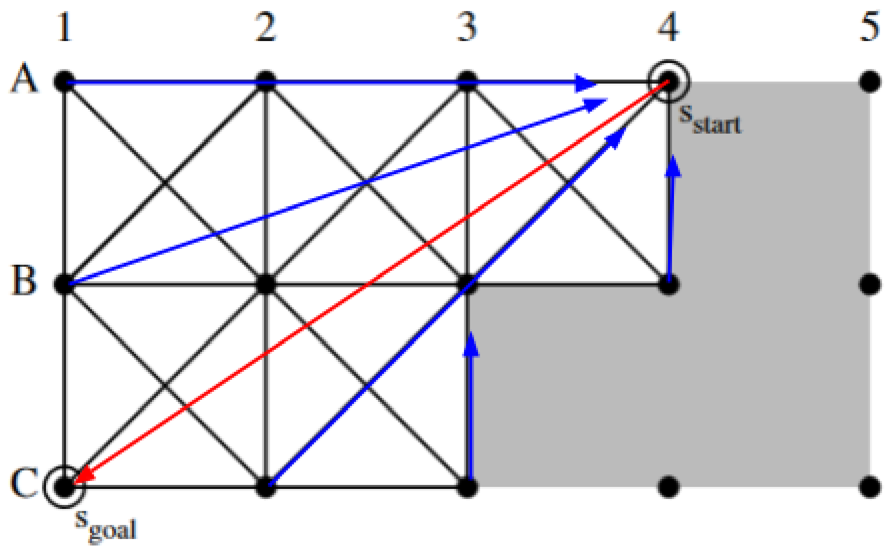
Step	Frontier	Closed	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3
0	[A4]	[]	N/A	N/A	Parent: A4 g(s): 1 h(s): 2.83 *f*: 3.83	■	N/A	N/A	Parent: A4 g(s): $\sqrt{2}$ h(s): 2.24 *f*: ~3.65	Parent: A4 g(s): 1 h(s): 3.16 *f*: 4.16	N/A	N/A	N/A
1	[B3, A3, B4]	[A4]	N/A	Parent: A4 g(s): 2 h(s): 2.41 f(s): 4.41	Parent: A4 g(s): 1 h(s): 2.83 *f*: 3.83	DONE	N/A	Parent: A4 g(s): $\sqrt{5}$ h(s): 1.41 f: 3.646	■	Parent: A4 g(s): 1 h(s): 3.16 *f*: 4.16	N/A	Parent: A4 g(s): $2\sqrt{2}$ h(s): 1 F: 3.83	Parent: B3 g(s): 1+ $\sqrt{2}$ h(s): 2 f: 4.414
2	[B2, C2, A3, B4, C3, A2]	[A4, B3]	Parent: A4 g(s): 3 h(s): 2 f: 5	Parent: A4 g(s): 2 h(s): 2.41 f(s): 4.41	Parent: A4 g(s): 1 h(s): 2.83 *f*: 3.83	DONE	Parent: A4 g(s): 3.162 h(s): 1 f: 4.162	■	DONE	Parent: A4 g(s): 1 h(s): 3.16 *f*: 4.16	Parent: A4 g(s): $\sqrt{13}$ h(s): 0 f: ~3.61	Parent: A4 g(s): $2\sqrt{2}$ h(s): 1 F: 3.83	Parent: B3 g(s): 1+ $\sqrt{2}$ h(s): 2 f: 4.414
3	[C1, C2, A3, B1, B4, C3, A2, A1]	[A4, B3, B2]											

- It is important to notice that, in comparison to A*, Theta* constantly updates its parent to the origin. This is because of the LineOfSight() algorithm that it uses to get direct paths from one vertex to another
- In the following Theta* Visual Trace, you can see how the algorithm updates the best paths by (instead of pointing back to a neighbor) points as furthest back as possible. In this case, some nodes pointing all the way back to the start.

Theta* Visual Trace (Blue lines for explored, red line for solution):







c. A* implementation

Our A* function takes the following inputs:

- *window* - our UI
- *start* - the start node
- *goal* - the goal node
- *nodes* - 2D list that represents the grid
- *edges* - a dictionary that stores a tuple of tuples (start & end coordinates) and the weight of the resulting edge

Our implementation of the A* algorithm begins with creating the variable *fringe* which is a Priority Queue with the format $[(f,\ (x, y))]$ where *f* is a nodes f(s) value and (x, y) is a tuple of nodes coordinates on the grid.

Three other variables will also be created:

- *closed* - A list of closed nodes
    - initialized to be empty
- *cost_so_far* - A dictionary used to track the cost so far, for an already explored node
    - *cost_so_far[start] = 0*
- *parent* - Another dictionary that will contain the best parent for a given node
    - *parent[start] = start*

We start executing the algorithm with a loop that will run while the fringe is not empty and use the start node as the first element *s*. If we have reached the goal we iterate through *nodes* and calculate *h(s)* and *f(s)*. *g(s)* will only be calculated if the current node is found in *cost_so_far*. Then we print the fastest path, its total cost, and return the path to main.py to be drawn.

If our goal has not been reached, we append *s* to *closed* and check its neighbors. If a neighboring node cannot be reached, or is in *closed*, we automatically move to the next neighbor. If a neighbor is new, *cost_so_far* and *parent* of said neighbor are set to *float('inf'),* and *None* respectively. Regardless of whether the neighbor is new, we can now draw the edge in our UI and use a function *update_vertex()* to update all necessary information.

In *update_vertex()*, we can calculate the cost to reach a target from a given node and if it is less than the target's cost so far we overwrite it, set the parent of target to be the start node, and add the target node to the fringe with its new *f(s)* value.

d. Theta* implementation

Similarly to A*, our Theta* implementation utilizes functions to update a vertex, calculate heuristics, and check valid edges.

We also take the same inputs as before:

- *window* - our UI
- *start* - the start node
- *goal* - the goal node
- *nodes* - 2D list that represents the grid
- *edges* - a dictionary that stores a tuple of tuples (start & end coordinates) and the weight of the resulting edge

Our Theta* implementation begins with creating the same variable *fringe* which is a Priority Queue with the same format as above. The A* variables *cost_so_far*, *parent*, and *closed*, are also created and initialized in the same manner here. The start node

We start executing the algorithm with a loop that will run while the fringe is not empty and use the start node as the first element *s*. If we have reached the goal, then we print the fastest path by starting at the goal node and working our way to the start node by looking at the current node's parent.

If our goal has not been reached, we append *s* to *closed* and check its neighbors the same way they would be checked in A*. If a neighbor is new, *cost_so_far* and *parent* of said neighbor are set to *float('inf'),* and *None* respectively. Regardless of whether the neighbor is new, we can now draw the edge in our UI and use a function *update_vertex()* to update all necessary information.

Unlike A*, our Theta* function to update a vertex uses a new function we created, named *line_of_sight()*. This function determines if two given nodes have a straight, unobstructed path to one another. Taking a beginning and end node as inputs, it attempts to move in a straight line from beginning to end and determine if a path is possible.

- If a path from the beginning node's parent to the end is found, and its cost is less than the end node's previous cost, the end node's cost is updated, its parent is set to be the parent of the beginning node, and the end is added to the fringe. This effectively cuts out the middleman and allows a more direct path to be taken.
- If this path is not found, and the cost to reach the end from the beginning is less than the end's previous cost, we update the fringe and the end's cost; its parent is set to be the beginning node, not its parent. In this situation a more direct path is not possible and the middleman is needed.

e. Proof

Given Equation (1):

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

Prove: A* with the h-values from Equation 1 is guaranteed to find the shortest grid paths.

The first part of Equation 1 ($\sqrt{2}\min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$ determines the maximum distance that our path can go diagonally. This is important because going diagonal is always faster than taking a combination of horizontal and vertical paths (triangle inequality theorem). Then, the next two parts ($\max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + \min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$) determine the rest of the distance needed to reach the point by going in a straight line to the goal. In other words, the equation is determining how far it can go diagonally followed by a straightaway. Since A* must follow the grid paths, this guarantees that we are less than or equal to the actual distance to the goal. This makes our *h(s)* **admissible**.

Next, a heuristic function h is said to be consistent if for all *(n, a, n')*:

h(n) <= c(n, a, n') + h(n')

where *c(n, a, n')* is the step cost for going from *n* to *n'* using action *a*.
In our case, we can present our heuristic function on our graph as a triangle inequality;

h(s) <= c(s, s') + h(s')

In other words, the heuristic value of *s* is guaranteed to be less than the cost to a successor node plus the successor's *h(s')* value. This makes our *h(s)* **consistent** for each vertex on the graph.

Finally, we can use the following theorems:

If h is admissible then the tree search A is optimal*

and

If h is consistent then the graph search A is optimal*

This means that consistent *h(s)* values will have A* expand the nodes on the optimal path before those on the nonoptimal. All these properties combined guarantee that our equation will give us the shortest grid paths.

f. *Did not implement extra credit*

g. Optimizations:
   - Calculate the heuristic once at the start of A* and Theta*
     - Originally, we were calculating it twice
   - Replaced the closed list with a "closed" variable inside of each node, giving constant time access rather than O(n) for checking if a node is closed.

Tests:
We ran both the optimized and unoptimized algorithms across all 50 100x50 graphs, averaging their runtimes. The results are as follows:

	Avg Optimized Time (s)	Avg Unoptimized Time (s)
A*	.0134	.0144
Theta*	.0147	.0160

Table 1: average runtime comparison across 50 100x50 graphs

These results show that on average, optimized A* was ~7.5% faster, while optimized theta* was ~9% faster. Considering that the closed list never gets too large on a 100x50 graph, we expected that the improvements would be marginal.

We also ran both programs over 1000x50 graphs, to test how well our optimization works over larger graphs. Over 25 trials each, we found the following results:

	Avg. Optimized Time (s)	Avg. Unoptimized Time (s)
A*	.2493	.2793
Theta*	8.4938	21.1926

Table 2: average runtime comparison across 25 1000x50 graphs

Here, optimized theta* is ~160% faster than its unoptimized counterpart–a significant improvement. A* on the other hand only had a 12% difference here, which is still more significant than 7.5% as seen in the 100x50 graphs.

Command used for testing:

```
stat data >/dev/null && rm data; touch data; for i in `seq 0 49`; do python main.py
graphs/graph$i.txt >> data; done && awk '{s+=$1} END {print s/NR}' data;
```

h. Using the 50 graphs from part g, we have the same average runtimes (from Table 1). We now incorporate the average path length of each algorithm across all 50 graphs in the table below:

	Avg. Time (s)	Avg. Path Length
A*	.0134	47.2452
Theta*	.0147	44.9

Table 3: average runtime and path length of A* and Theta*

In our experiments, we observed that Theta* had shorter paths in 46 out of the 50 tested graphs, with its path lengths being ~5% shorter on average. However, we saw that Theta* consistently ran slower than A*, performing ~10% slower on average (see Table 3). This is expected, however, as Theta* needs to do additional checks for line of sight to determine optimal paths.

The h-values given to A* and Theta* derived from their respective equations are fair, as both of their h(s) equations are built for their respective path finding behavior. A* uses the only the grid path edges, while Theta* can take a straight path to its target. Thus, it makes sense that A* would use the heuristic equation:

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

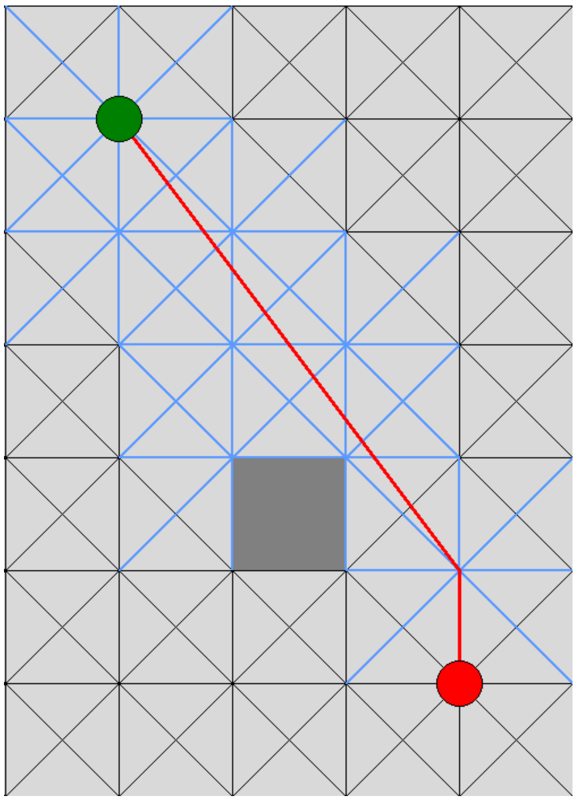
This equation finds the maximum length it can go diagonally, followed by a straight line. This makes sense for A* since it's path must align to the grid lines.

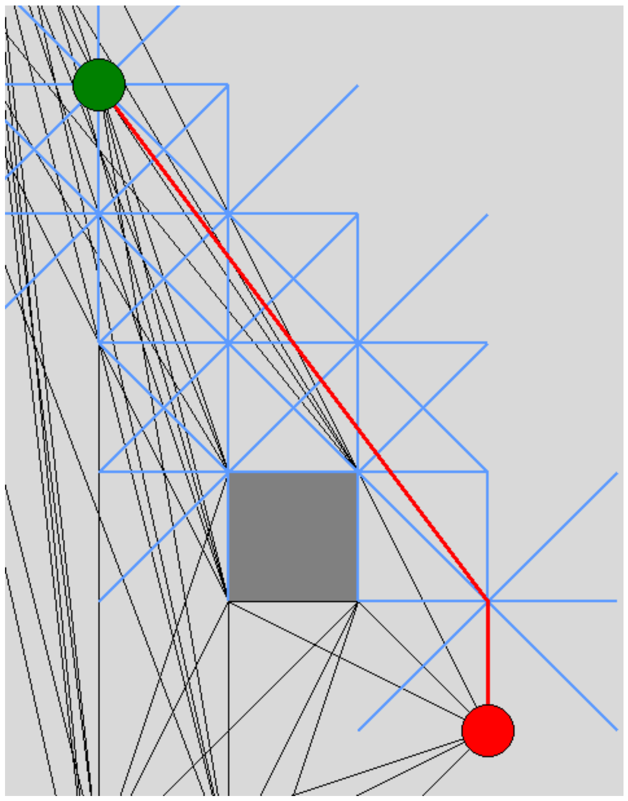
As for Theta* the use of: *h(s) = c(s, s')* is well suited as (in the best case) Theta* can find the straight line to its goal. These h-values give straight line estimates to the goal which help Theta* find a near direct path to its goal. In other words, the euclidean distance is the maximum of the underestimated (optimistic) heuristics, making it the best suited heuristic.

i. Theta*, in some cases, will not find the shortest path between two nodes on a graph. This problem lies in the fact of how LineOfSight() is implemented. When a node is expanded it finds a neighboring node and attempts to reparent that neighbor with its parent.

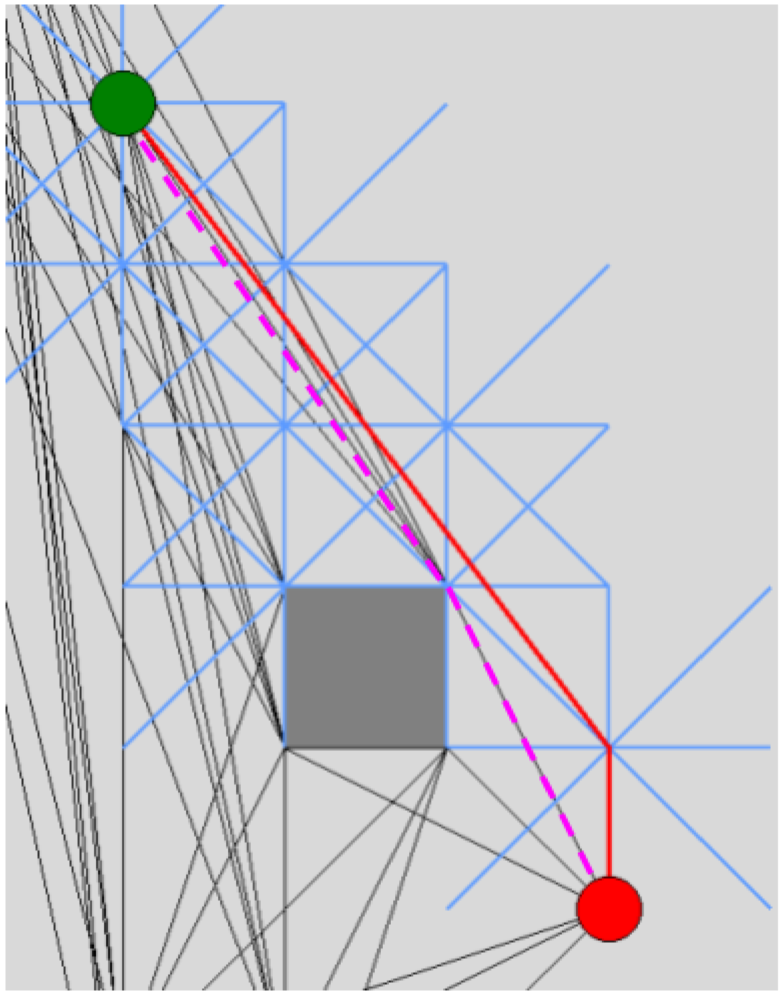
   For example, if you're expanding *s* with parent *a*, it will attempt to reparent neighbor *s'* with *a*. If this reparent process is not possible, then it will simply parent *s'* with *s*.

   However, the important part of this process is that *s* is **never** reevaluated once it is expanded. The consequences of this are if *s'* finds the goal and it's parent does not have LineOfSight, then it will pivot to the goal from s*'*! It will not try to backtrack to see if a previous node in the path has LineOfSight. The following example expands on this idea.





In this example we can see that Theta* actually fails to find the most optimal path from the start (in green) to the goal (in red). It may not be immediately noticeable in the standard grid graph, but the visibility graph (on the right) shows a different story. The visibility graph clearly shows that we have an almost direct path from the start to the goal that barely detours around the blocked cell. However, it was not picked up by Theta*. The correct answer is denoted in the following picture.



In this picture, the correct answer is highlighted by the dashed purple line. Additional visibility graph information is documented in the README.

With our complete correct answer we can see what happened. The vertex right above the goal (which we will denote as *p*) was reparented to the starting node. Then when *p* saw the goal node in its neighbors, it attempted to have it's parent (the starting point) to have LineOfSight to the goal. The blocked cell keeps us from having LineOfSight which causes the Theta* line to be pivoted from *p*. Since we never revisit closed nodes in Theta*, the correct path from the corner of the closed cell to the goal is never found.

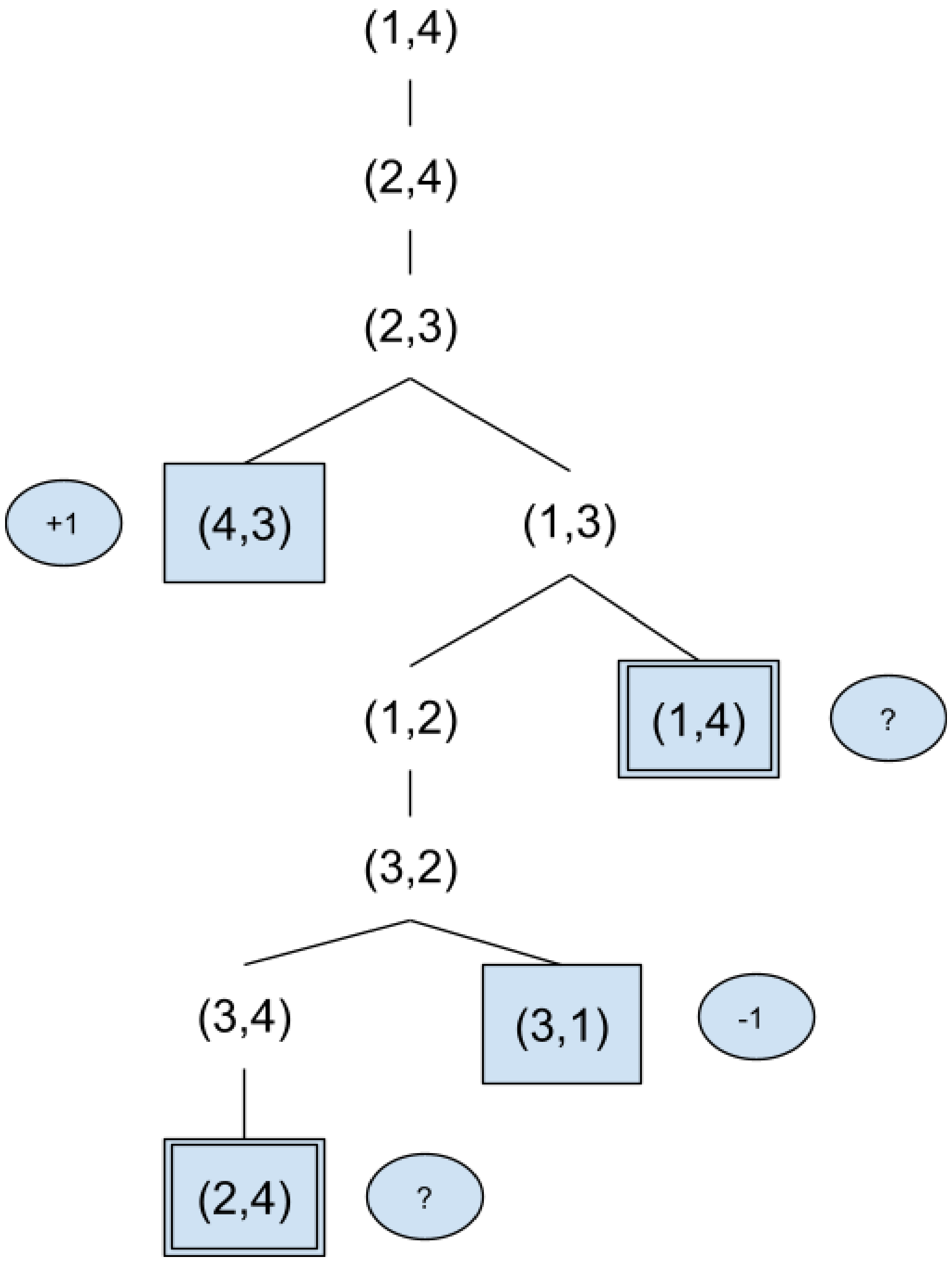
As for the differences between them:

- A* length $\rightarrow 3\sqrt{2} + 2 \approx 6.242$
- Theta* length $\rightarrow \sqrt{25} + 1 = 6$
- Actual Best length $\rightarrow \sqrt{13} + \sqrt{5} \approx 5.8416$

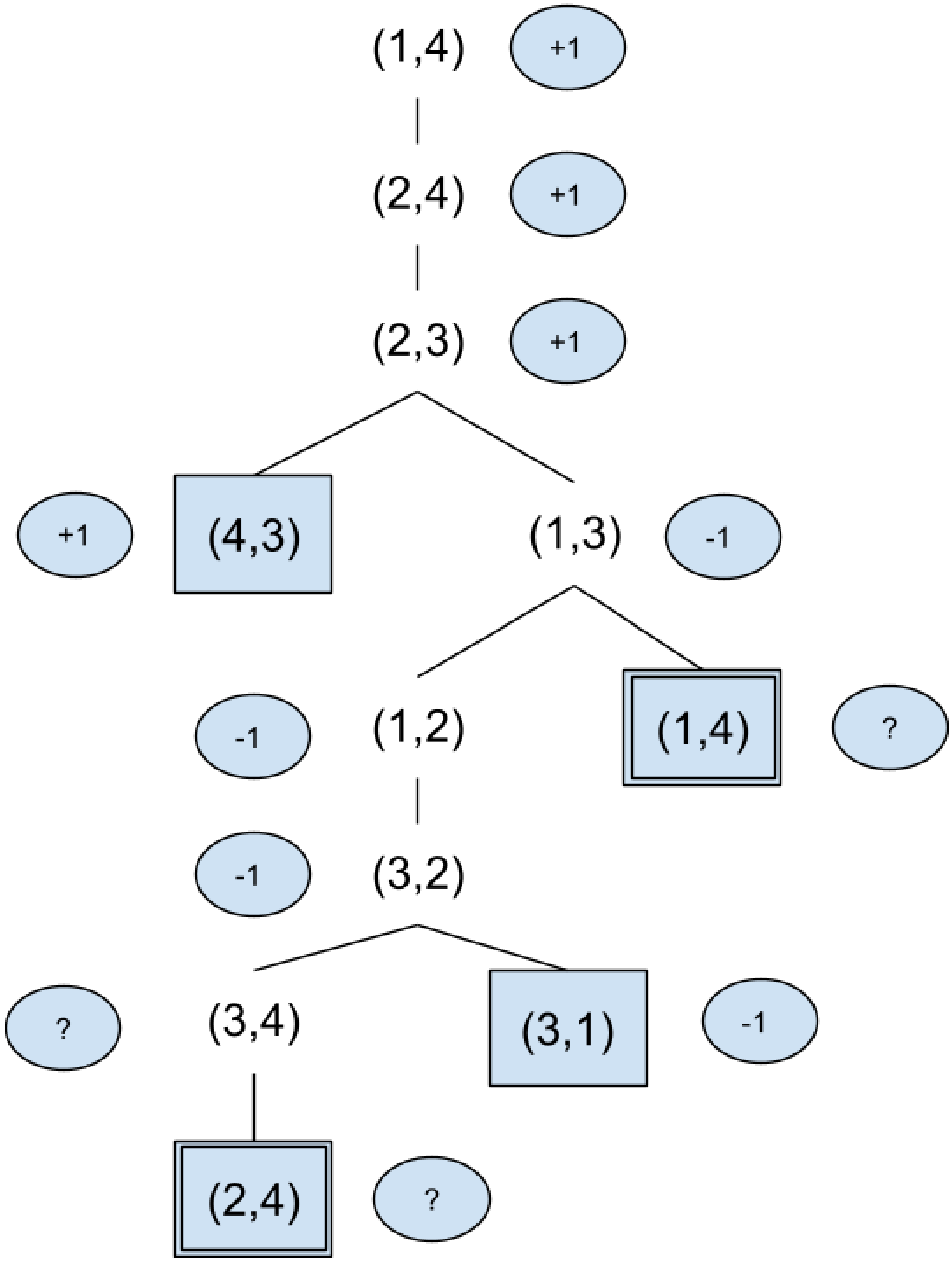
j. *Did not do EC*

Problem 2:

a. Game tree:



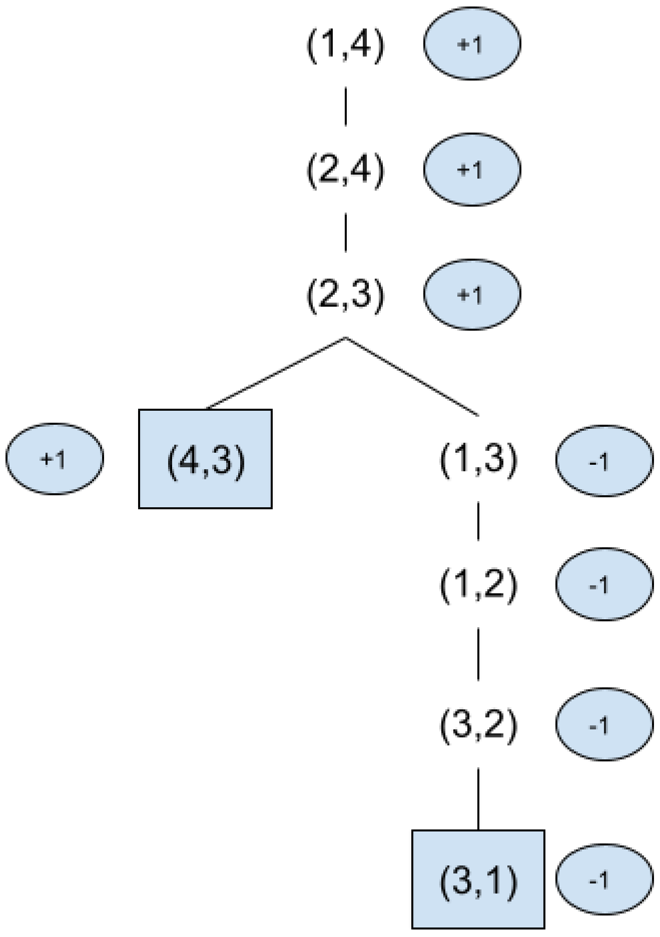
b. Minimax values:



At each point for the “?” values, B is left with the choice of taking min(-1, ?). Since -1 is the utility value that gives him the win, B will always take -1 over “?”.

c. The standard minimax algorithm would fail because it uses DFS. DFS will follow the loop states (the ones marked with '?') and infinitely loop. This will cause a solution to never be found as the algorithm will be stuck in 1 branch of the tree forever.

   Potential fix (Prune off the loop branches):



   This will not give optimal solutions for all games with loops. One of the main cruxes to this solution are games that use randomness. For example, a game that rolls dice or flips a coin. This randomness factor can not be pruned away and therefore can cause the game state to have loops. Our solution of pruning off the looping branches only works for games that have deterministic choices.

d. Proof by Strong Induction
Let n denote number of cells on the board
Let P(n) be true when A wins if n is even else loses

Base case: n = 3 => 3 is odd, A loses
Start state: (1, 3)
Moves: (2, 3) → (2, 1) B wins
Base case: n = 4 => 4 is even, A wins
Start state: (1, 4)
Moves: (2, 4) → (2, 3) → (4, 3) A wins

Inductive Step: For n >= 3, let P(t) be true for any 3 <= t <= n such that P(t) => P(t + 1)

P(t + 1) is the case of t +1 cells on the board. Thus, A and B start at position would be denoted as: (1, t + 1)

When A and B move inward on their first turn, the board turns into sub board: [2, t] which can be represented as: P(t - 1). (Because two cells were removed, one from the beginning and one from the end). Therefore, we went from a t + 1 board to a t - 1 board within one step. From this fact, we can relate future larger boards to smaller, already proven to be true, boards.

If t + 1 is even, we know that subboard t - 1 is also even. (Any even number - 2 = another even number) This new subboard, P(t - 1), was assumed true by the inductive hypothesis. Therefore, P(t + 1) must also be true because the t + 1 board is the same board as the t - 1 board (aside from the initial steps inward).

Similarly, if t + 1 is odd, we know that t - 1 is also odd. Which is, in our hypothesis denoted by P(t - 1), also true.

Thus, through use of strong induction, we know that P(t) holds for all n >= 3. We can now officially say that A wins when the number of cells is even, and loses when the number of cells is odd.

Problem 3:

a. Hill climbing works better for functions that have no local maxima (a single global maxima). This is because it will be nudged in the uphill direction on the only hill that exists, meaning the right answer is guaranteed.

b. Randomly guessing would work just as well as simulated annealing on functions that have no structure: there are many plateaus and bumps that make it hard to discover the global maxima.

c. Simulated annealing is useful when our value function has several areas that are not our goal state. For example, if we are trying to find the max of a function we do not want to get stuck on a local max. Simulated annealing remedies this problem by getting us out of these local max (or mins) that would normally be returned by standard local search.

d. Simulated annealing returns the current state when the temperature reaches 0. When this condition is met, we hope we have hit our largest goal state. However, since simulated annealing could possibly take a worse state (based on probability) it would be smart to keep a variable that keeps track of our best state we have seen so far. When the temperature reaches 0, we can either return to the state we are at or the best state we have seen so far depending on which is better.

e. Originally we stored the current state and a single proposed next state. Instead, we can store 2 million randomly proposed states and compare them to each other. We can then select the state with the highest increase in value; if all states are worse, then we select the best of these states with a probability that is proportional to the change in value. With this method, we are more likely to find higher value states, than rely on probability that we may find one within our “cooling” duration. Also, storing more states will prevent the need for an extremely high starting “temperature”, allowing less loops through the algorithm.

Problem 4:

a. The set of variables are the cells within each row and column (81 total)
   Using A-I for rows and 1-9 for columns, the default domains are {1-9}. For pre-filled cells, we have the following domains:

   A2: {1}
   A4: {4}
   A5: {2}
   A9: {5}
   B3: {2}
   B5: {7}
   B6: {1}
   B8: {3}
   B9: {9}
   C8: {4}
   D1: {2}
   D3: {7}
   D4: {1}
   D9: {6}
   E5: {4}
   F1: {6}
   F6: {7}
   F7: {4}
   F9: {3}
   G2: {7}
   H1: {1}
   H2: {2}
   H4: {7}
   H5: {3}
   H7: {5}
   I1: {3}
   I5: {8}
   I6: {2}
   I8: {7}

   Considering that a single number cannot be found more than once in a given row, column, and 3x3 square, we have 27 different constraints: 9 for each row, 9 for each column, and 9 for each square.
   A row constraint would look something like:

   diff(A1,A2,A3,A4,A5,A6,A7,A8,A9)

   A column constraint would look something like:

   diff(A1,B1,C1,D1,E1,F1,G1,H1,I1)

A box constraint would look something like:

diff(A1,A2,A3,B1,B2,B3,C1,C2,C3)

b. M = number of given cells (in our question M is given to be 29)
   i. Incremental Formulation
      1. Start State → The board with M filled in cells
      2. Successor function → Pick neighbor cell, assign value within the domain {1..9}
      3. Goal test → All cells filled with constraints satisfied
      4. Path cost function → 1
   ii. Which heuristic?
      1. Minimum remaining values because by going for these nodes first we can quickly prune off a large amount of branches. Imagine if one of the 3x3 boxes on the board had {1, 2, 3, 4, 5, 6, 7, 8} already filled in. It would make most sense to fill in 9 to that last cell (the cell with least remaining values) instead of going for one that has a large amount of values remaining to check. As there is only one value to fill in there to satisfy the constraint. In short, this quick fill will also constrain surrounding cells (the other variables), thus bringing us closer to an answer much faster.
   iii. Search space:
      1. Branching factor → 9
      2. Solution depth → 81 - M
      3. Max depth → 81 - M
      4. Size of state space → $9^{(81 - M)}$

c. The difference between “easy” and “hard” sudoku problems is the number of given cells that are already filled in. (Therefore, a larger M value) Take for example a board that has only one number filled in for each of the 9 3x3 boxes on the board. It will take a significant amount of more time (even for a computer) to think of all possible places to put your other numbers to fill the board. In contrast, imagine a board where the 9 3x3 boxes on the board are all filled but one cell in each. It only takes a minute at most to determine what number will go into that box to satisfy the constraint of that row, column, and box. This is because the other cells (or variables) are already assigned and you know those assignments are correct answers. Leaving you with significantly less boxes to fill in, and thus less numbers to test.

d.

```
localSearchSudoku(Sudoku table):
    -   randomly assign a value $x \mid 1 \leq x \leq 9$ to all initially empty cells
    -   while !GoalTest(table):
        -   select an initially empty cell that is now violating a constraint
        -   if a given probability is struck:
            -   change the cell's value to a new random value x
        -   else:
            -   for each initially empty cell:
                -   check how many constraint violations will occur if the
                    current assignment changes
                -   change the cell that will result in the lowest number of
                    violations
    -   return table
```

This algorithm should work better than the best incremental search for hard problems and worse for easier problems. If we have an easy problem, incremental search will likely need little time to reach a solution because there's few values to choose from, allowing a heuristic such as minimum remaining value to shine. On the other hand, the randomness of our algorithm and the rechecking of cells will often lead to more time being wasted and thus take longer to reach the same answer.

In the case of a hard problem, incremental searches will most likely take longer to find the solution, giving our algorithm a better chance to reach the solution faster. We hypothesize this because our algorithm works by favoring values that result in the lowest number of violations. This allows all of the violations across the board to be slowly dwindled down, or in other words, doing local search for the minimum number of violations. As opposed to incremental search, that mainly focuses on its neighbor cells and might run into a violation further down the line setting it back.

Problem 5:

a. Knowledge base:

  A is true when superman is alone
  D is true when superman is defeated
  K is true when kryptonite is being used
  L is true when Lex Luthor was involved
  W is true when wonder woman gets involved

  In propositional logic, our knowledge base looks like:

  $D \Leftrightarrow A \wedge K$
    Superman's defeat comes from fighting alone AND Kryptonite being used
  $K \Rightarrow L$
    If Kryptonite is obtained/used, then collusion with lex luthor must have happened
  $L \Rightarrow W$
    If collusion with Lex Luthor happens, then wonder woman gets involved
  $W \Rightarrow \sim A$
    If Wonder woman gets involved, then superman will not be alone

b. $K \Rightarrow L$, $L \Rightarrow W$, $W \Rightarrow \sim A := K \Rightarrow \sim A$ through hypothetical syllogism

  This leaves us with:

  $D \Leftrightarrow A \wedge K$
  $K \Rightarrow \sim A$

  3CNF conversion for $D \Leftrightarrow A \wedge K$:

  $((D \Rightarrow A \wedge K) \wedge (A \wedge K \Rightarrow D))$ // Biconditional elimination
  $((\sim D \vee (A \wedge K)) \wedge (\sim(A \wedge K) \vee D)$ // Transposition (implication elim)
  $((\sim D \vee (A \wedge K)) \wedge (\sim A \vee \sim K \vee D)$ // DeMorgan's Law
  $(\sim D \vee A) \wedge (\sim D \vee K) \wedge (\sim A \vee \sim K \vee D)$ // Distribution

  3CNF conversion for $K \Rightarrow \sim A$:

  $\sim K \vee \sim A$

  So our 3CNF knowledge base is:

  $(\sim D \vee A) \wedge (\sim D \vee K) \wedge (\sim A \vee \sim K \vee D) \wedge (\sim K \vee \sim A)$

c. Want to prove that Batman cannot defeat superman. We can use a proof by contradiction within the satisfiability algorithm to solve this.

   Let $\alpha = \sim D$ (Superman cannot be defeated)
   KB $\wedge$ $\sim\alpha$
   $(\sim D \vee A) \wedge (\sim D \vee K) \wedge (\sim A \vee \sim K \vee D) \wedge (\sim K \vee \sim A) \wedge D$
   $(\sim D \vee A) \wedge (\sim D \vee K) \wedge (\sim K \vee \sim A) \wedge D$ // Absorption Rule
   $(F \vee A) \wedge (F \vee K) \wedge (\sim K \vee \sim A) \wedge T$ // Resolve out D
   $A \wedge K \wedge (\sim K \vee \sim A)$ // Simplify
   $K \wedge \sim K$ // Resolve out A
   * // Resolve out K

   We are then left with an empty clause. By the satisfiability algorithm we found KB $\wedge \sim\alpha$ to be unsatisfiable. Therefore, KB $\models \sim D$. In other words, our knowledge base entails that Superman can not be defeated.