CS344 - Assignment 1

Maya Barathy, Sammi Scalese, Kelci Mensah

February 25, 2022

Problem Solving

1 - Any-Angle Path Planning

1g.

During the first implementation of the code, we actually added optimizations to the code that were not shown in the pseudo code. In the pseudo code, for each new node popped off the priority queue to be searched, a loop cycles through all of its neighbors and checks the entire priority queue to see if that neighbor is added, i.e. search time of queue O(n) multiplied by the number of neighbors for each node checked. To eliminate this time cost, we added a boolean flag to each node called 'notAdded' that was defaultly false but would become true when the node was added to the priority queue; this way we could check whether a node was added to the queue or not in constant time. Another optimization we made was to also add a boolean 'newNeighbors' flag that was defaultly false but would become true when that node added a new, not yet added neighbor to the priority queue. This was done to mitigate the time spent on backtracking in the final trace of the shortest path for A* runs. If a node did not add any new, unchecked neighbors to the open list, that means the search got trapped in a corner or blocked area, and the node with the next smallest f-value in the priority queue should be checked to continue the path correctly around the block. The node would not be added to the closed list if it did not contribute a new neighbor to the open list, but it would still be marked as added, so it was not checked again. This eliminated the need for another search at the end of the algorithm because we could ensure the closed list was the shortest path.

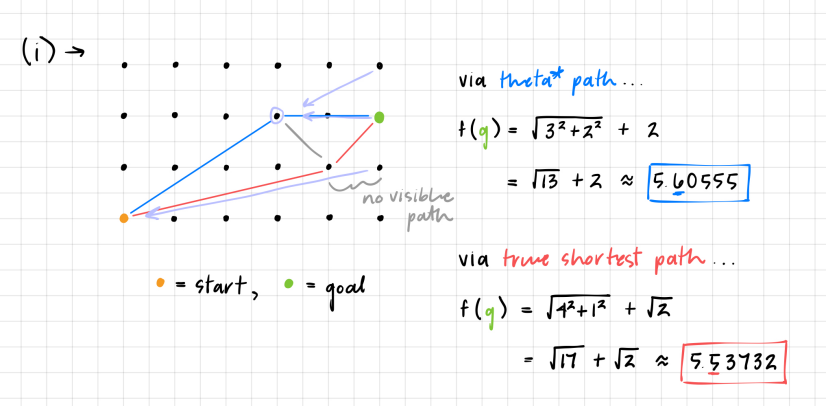
1h.

For grid-spanning algorithms, there are a few main heuristic equations that are best for finding the shortest path between two points. There is: the "Manhattan" or block distance (for 4-point directional movement, like up, down, left, or right), the diagonal distance (for 8-point directional movement that also includes diagonals), and the Euclidean or straight-line distance (for any-angle directional movement). The Euclidean distance is a good heuristic since it will always be admissible, meaning that, for any and every case, the equation's result will always be either less than or equal to the actual cost of the path, which is true of the Euclidean distance because it is the shortest possible distance between two points. However, Euclidean distance is not necessarily optimal for certain use-cases (i.e. 4-point or 8-point directional movement) because it is an overly broad heuristic that would require searching way more nodes, which is neither time nor space efficient. To mitigate these inefficiencies, a more realistic heuristic to use for the A* algorithm would be the diagonal distance, which is more accurate to its possible range of movement while still being admissible. Although it is not perfect in terms of backtracking when obstacles are introduced, it still requires less searching and backtracking compared to using the Euclidean distance for the A* use-case. For the Theta* algorithm, however, the diagonal distance would not be admissible for its any-angle neighbors because, for a best case use for Theta*, the actual cost can be equal to the Euclidean distance between the start and goal points. Therefore, the best possible heuristic for Theta* while still being admissible is the Euclidean distance. Although these heuristics are not perfect in the face of introducing obstacles, backtracking around obstacles cannot be avoided, and these heuristics actually ensure the most efficient workarounds, i.e. the least amount of backtracking. Thus, the given heuristics, the diagonal distance for A* and the

Euclidean distance for Theta*, are the best possible heuristic equations for the specific goal of each algorithm.

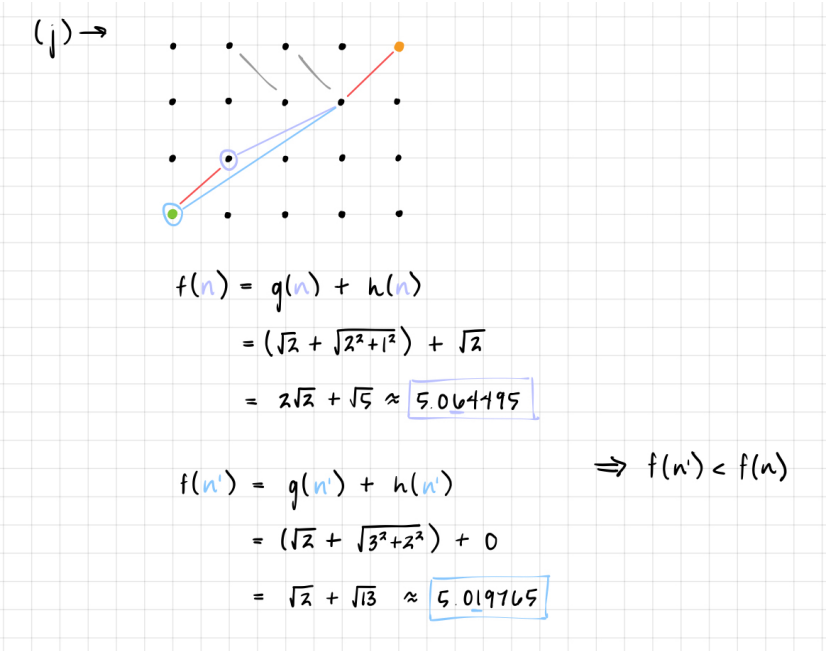
1i.

The Theta* algorithm is not actually guaranteed to find the shortest path, like the A* algorithm is able to ensure. This is because, in Theta*, the parent of a node has to be either a visible neighbor or the parent of a visible neighbor, which is not always a property of any-angle true shortest paths. While remaining time and space efficient, meaning without searching the entire grid for all possible paths, the any-angle algorithm cannot ensure the shortest path is found.



1j.

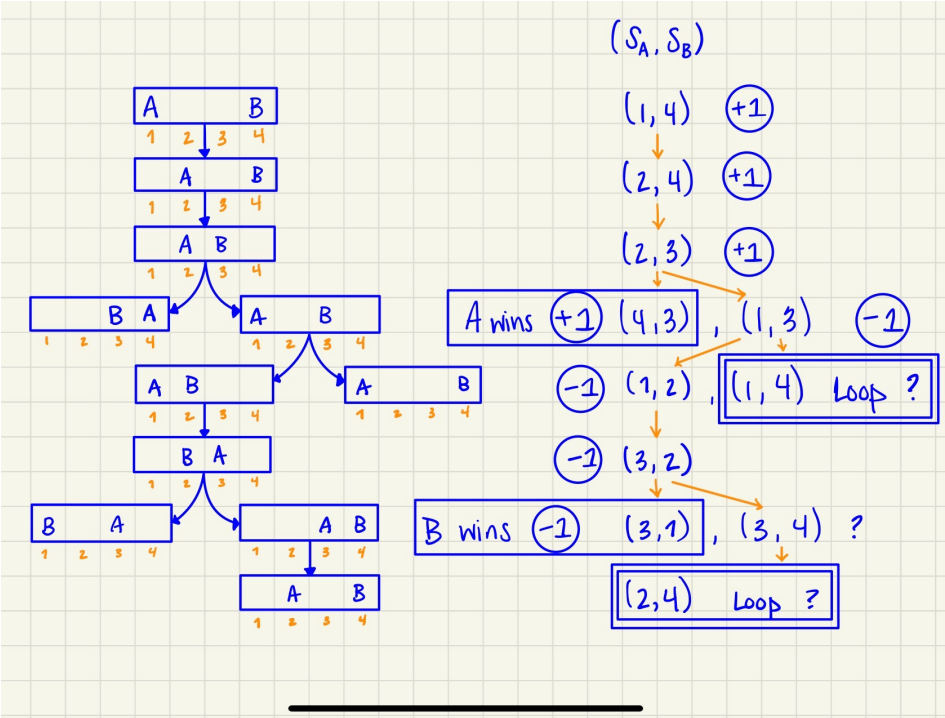
Due to Theta* using any-angle paths, it is possible for a node n to expand a neighboring node n‘ (n prime) whose total f-value is less than node n’s f-value due to the algorithm using line of sight.



Beyond Heuristic Search

2 - Adversarial Search

2a. Draw the complete game tree:



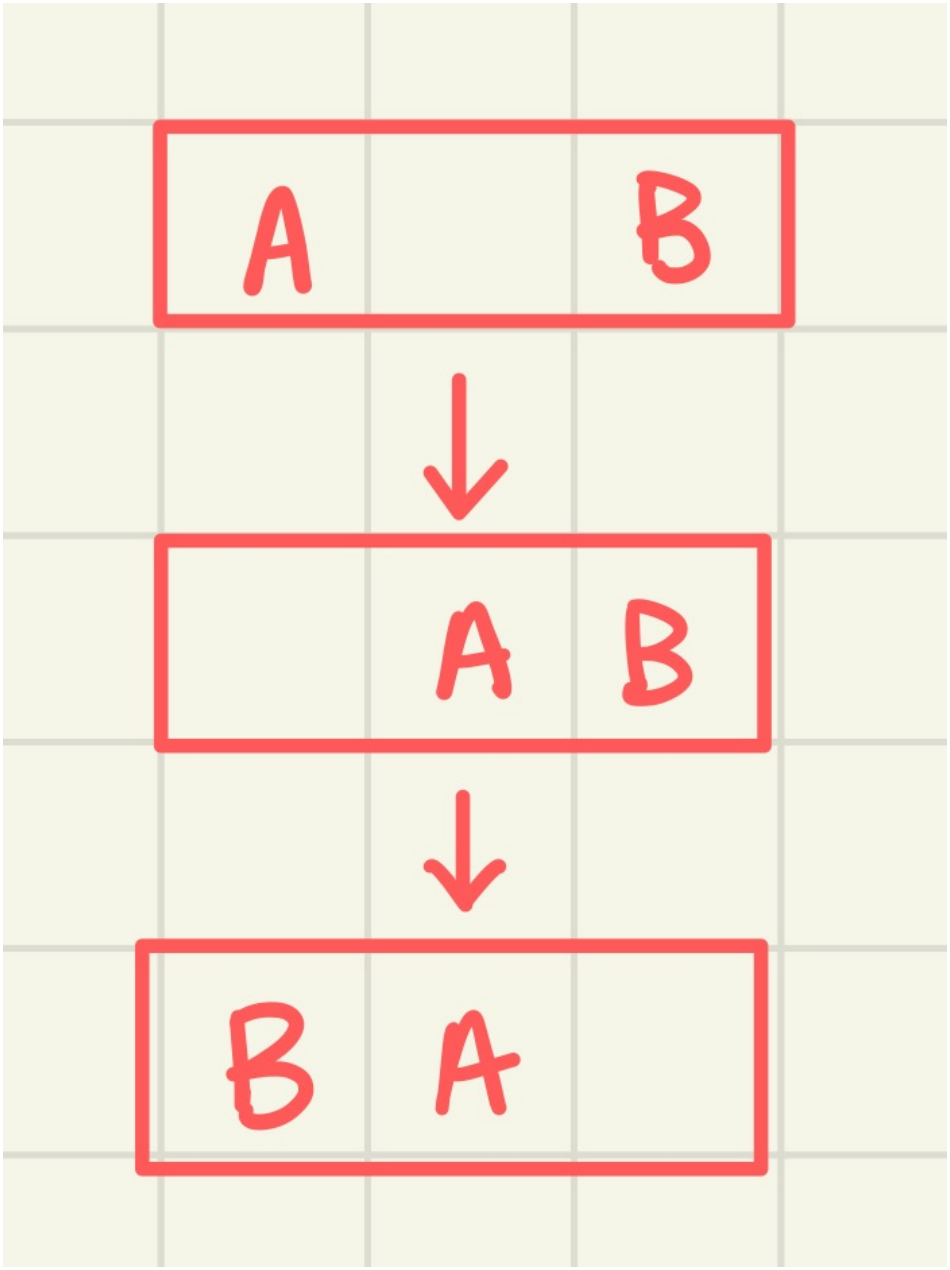
2b.

The ? states only appear when the only option is between either looping back or winning the game. So when an agent is presented with (-1, ?) for min, then the agent will choose -1 and for max (+1, z) that is +1. If the successors after the node that is ?, then the backed up minimax value is ?.

2c.

Standard minimax follows a depth first search algorithm hence causing it to go into an infinite loop. Yes, our modified algorithm provides optimal decisions for all games with loops because anytime a node loops, it goes back to the previous time it was at that state.

2d.



Using proof by induction we can assume that the base case is when n is equal to 3, A loses. From the above diagram we can conclude that B wins. In part A, n is equal to 4 and A wins. So whenever A reaches n-1 before B reaches 2 A wins and this only happens when n is even. So following this pattern A wins when n is even and B wins when n is odd.

3 - Local Search

3a.

Hill Climbing works better than Simulated Annealing for any problem in which there is only a global maximum or global minimum, or when the difference between local optimum and local optima is minimal. Simulated Annealing is similar, however the probability of its randomization can bring the solution to a point where the succeeding randomization is less optimal than the previous.

3b.

They would work the same and the randomization would not be necessary when the fitness function is entirely made of shoulders or plateaus, i.e. a straight line. This is because there would be no "better" or more optimal state".

3c.

Simulated annealing is a useful technique for problems where the fitness function has a lot of local extrema points.

3d.

Since we know the value of each state we visit the algorithm will recall the best value. So instead of returning the current state we could return the best state.

3e.

Now that we have additional space because of the additional memory we could let the annealing step choose between the step that has the best score and the local step. This is obviously more productive because we are not only limited to the current state and the proposed next state.

4 - Constraint Satisfaction Problem

4a.

Variables
m = number of variables that have already been assigned A1,A2,A3,A4,A5,A6,A7,A8,A9
B1,B2,B3,B4,B5,B6,B7,B8,B9
C1,C2,C3,C4,C5,C6,C7,C8,C9
D1,D2,D3,D4,D5,D6,D7,D8,D9
E1,E2,E3,E4,E5,E6,E7,E8,E9
F1,F2,F3,F4,F5,F6,F7,F8,F9
G1,G2,G3,G4,G5,G6,G7,G8,G9
H1,H2,H3,H4,H5,H6,H7,H8,H9
I1,I2,I3,14,I5,I6,I7,I8,I9

Domain
A1 -A9 = 1,2,3,4,5,6,7,8,9
B1 -B9 = 1,2,3,4,5,6,7,8,9
C1 -C9 = 1,2,3,4,5,6,7,8,9
D1 -D9 = 1,2,3,4,5,6,7,8,9
E1 -E9 = 1,2,3,4,5,6,7,8,9
F1 -F9 = 1,2,3,4,5,6,7,8,9
G1 -G9 = 1,2,3,4,5,6,7,8,9
H1 -H9 = 1,2,3,4,5,6,7,8,9
I1 -I9 = 1,2,3,4,5,6,7,8,9

Constraints
A1 != A2 != A3 != A4 != A5 != A6 != A7 != A8 != A9
B1 != B2 != B3 != B4 != B5 != B6 != B7 != B8 != B9
C1 != C2 != C3 != C4 != C5 != C6 != C7 != C8 != C9
D1 != D2 != D3 != D4 != D5 != D6 != D7 != D8 != D9
E1 != E2 != E3 != E4 != E5 != E6 != E7 != E8 != E9
F1 != F2 != F3 != F4 != F5 != F6 != F7 != F8 != F9
G1 != G2 != G3 != G4 != G5 != G6 != G7 != G8 != G9

H1 != H2 != H3 != H4 != H5 != H6 != H7 != H8 != H9
I1 != I2 != I3 != I4 != I5 != I6 != I7 != I8 != I9
A1 != B1 != C1 != D1 != E1 != F1 != G1 != H1 != I1
A2 != B2 != C2 != D2 != E2 != F2 != G2 != H2 != I2
A3 != B3 != C3 != D3 != E3 != F3 != G3 != H3 != I3
A4 != B4 != C4 != D4 != E4 != F4 != G4 != H4 != I4
A5 != B5 != C5 != D5 != E5 != F5 != G5 != H5 != I5
A6 != B6 != C6 != D6 != E6 != F6 != G6 != H6 != I6
A7 != B7 != C7 != D7 != E7 != F7 != G7 != H7 != I7
A8 != B8 != C8 != D8 != E8 != F8 != G8 != H8 != I8
A9 != B9 != C9 != D9 != E9 != F9 != G9 != H9 != I9
A1 != A2 != A3 != B1 != B2 != B3 != C1 != C2 != C3
D1 != D2 != D3 != E1 != E2 != E3 != F1 != F2 != F3
G1 != G2 != G3 != H1 != H2 != H3 != I1 != I2 != I3
A4 != A5 != A6 != B4 != B5 != B6 != C4 != C5 != C6
D4 != D5 != D6 != E4 != E5 != E6 != F4 != F5 != F6
G4 != G5 != G6 != H4 != H5 != H6 != I4 != I5 != I6
A7 != A8 != A9 != B7 != B8 != B9 != C7 != C8 != C9
D7 != D8 != D9 != E7 != E8 != E9 != F7 != F8 != F9
G7 != G8 != G9 != H7 != H8 != H9 != I7 != I8 != I9

4b.

- Start State: For the given problem, the start state has 29 out of the 81 cubes filled
- Successor Function: This refers to the function that goes from the current state to the next state. For the given problem, the successor function fills the next empty cube.
- Goal Test: tests whether the agent satisfies each constraint. The goal test for this problem would be to make sure that each of the 9 rows and 9 columns are filled with numbers from 1-9 and they do not repeat themselves.
- Path Cost Function: Cost of taking each function. It is same in all states. For this problem, the path cost is 1.
- Degree Heuristic assigns a value to the variable with the most constraints whereas the remaining minimum value heuristic assigns values to the variables with the fewest possible variables. In this case using minimum remaining values heuristic is more beneficial because the variables that are unassigned will be filled with the remaining values till there are no more unassigned variables.
- The branching factor is the number of children for each node which in this case is 9
- Solution depth is the length of the shortest path from the initial state to the goal state which is 52 in this problem
- The maximum depth is the number of nodes from the root to the furthest node which is also 52
- The size of the state space is the set of all possible values for a system which in this case is 9.

4c.

Easy Sudoku problems have a larger value for m whereas harder Sudoku problems have a smaller value for m. This is because if the value for m is large then the solution path is smaller and hence becomes easier to solve the problem.

4d.

function localSearch
if variable = null
assign = assigns each variable with a number from 1 to 9
if assign adheres to the constraints
return assign
This technique will be better for easy problems because there are less number of variables that are empty.

5 - Logic-Based Reasoning

5a.

Variables

```
D -> superman is defeated
A -> superman is alone
K -> Kryptonite
B -> Batman
S -> Superman
W -> Wonder Woman
L -> Lex Luthor
U -> Wonder Woman upset
```

5b.

Statements

```
K AND A --> D       (CNF Statement 1)
~K OR ~A OR D       (3-CNF Statement 1)

K --> B AND L       (CNF Statement 2)
~K OR ~(~B OR ~L)   (3-CNF Statement 2)

B AND L --> U       (CNF Statement 3)
~B OR ~L OR U       (3-CNF Statement 3)

U --> W AND S       (CNF Statement 4)
~U OR ~(~W OR ~S)   (3-CNF Statement 4)
```

5c.

We must prove $\sim D$. We know . . .

```
A AND K OR ~D
```

From our knowledge base, we can infer . . .

```
K --> L
L --> U
U --> W
W --> ~B
```

We also know if . . .

```
P -> Q = ~P OR Q
```

Converting the above statements using our knowledge

```
~K OR L
~L OR U
~U OR W
~W OR ~A
```

```
The resolution inference rule states that (p OR q) AND (~p OR r)  ->  q OR r

Using this we can conclude that
(~U OR W) AND (~W AND ~A) --> ~U OR ~A
(~L OR U) AND (~U AND ~A) --> ~L OR ~A
(~K OR L) AND (~L AND ~A) --> ~K OR ~A
~K OR A = ~(K AND A)


Using distributive property we know
~D OR (K AND A)
~D OR ~(K AND A)
~D
```