CSE 5360 Programming Assignment 1

Daniel Tam (1001724986)

November 7, 2021

Question 1:

For the first question, I implemented the missionaries and cannibals' problem in LISP. The idea behind the problem is that we have several missionaries and cannibals on one side of the river, and we must move them to the opposite bank. There is a singular boat that is used to shuttle either the missionaries or cannibals across, but the boat has limited spots and we must ensure that the cannibals do not outnumber the missionaries on either side. When all the missionaries and cannibals are on the other side, the problem is solved.

In my implementation, I use a BFS search in order to find the next best path. I expand each node and remove the nodes that cannot be used next. Examples of this would be if after we shuttle missionaries and/or cannibals across, we need to make sure that the cannibals do not outnumber the missionaries on either side. We also remove the nodes that are completely not possible such as adding too many or subtracting too many missionaries or cannibals from either side.

We take our results that are viable and pass it recursively back in and begin expanding our nodes. Since we are using a BFS method, we expand all the nodes on the same level. In order to correctly do this, I only expand the nodes which were passed in per recursive loop. For example, if nodes such as ((10, 5, 1), (10, 4, 1)) were passed in, I would only expand these two nodes. This ensures that we correctly use a BFS method to expand.

While initially testing, with smaller groups of missionaries and cannibals and smaller boat sizes, it runs how you would expect. It moves only a few at a time back and forth and slowly moves everyone to the opposite bank. When running with the assignment of 15 missionaries and cannibals and a boat size of 6, I noticed the behavior was to move all the people in even numbers. That is, it moves 3 missionaries and 3 cannibals at a time each. With the test of 20 missionaries and cannibals, the behavior is the same: with a boat size of 6, it moves 3 missionaries and 3 cannibals each time to keep numbers even on both sides.

There are not checks in place to ensure valid input, but for the sake of our program, we will assume that all inputs are valid. From testing, it appears that if the boat is too small compared to the number of missionaries and cannibals, the program will cause a crash from a stack overflow. When testing, I found that if I used a missionary and cannibal size of 4 and a boat size of 2, it causes it to crash. It appears a boat size of 2 can only run properly on a group size of 3, and a boat size of 3 can only run properly up to a group size of 5. For a boat size of 4, we can go up well above 100 for our group size. It appears that only the smaller boat sizes are causing issues.

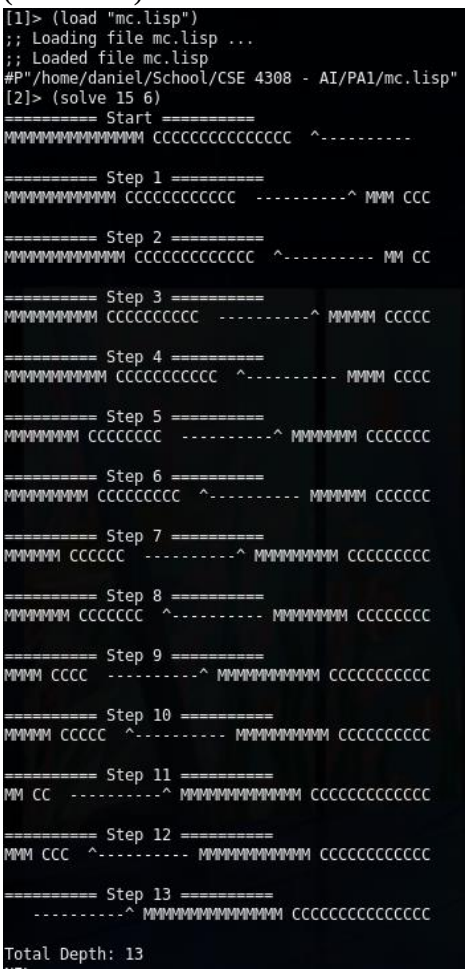
When attempting to try with the original problem, which is 3 cannibals and 3 missionaries with a boat size of 2, it performs as expected except for the last step. Though, the result is valid, but it may be how BFS is implemented and how it cycles through the expansions. It may target a different expansion first which is also valid, but the order may be different from other solutions. This may be an unintended bug in

my implementation. Verifying all the steps, it all appears valid. No rules were broken, but this may cause later attempts to be higher than normal expansions.

We utilize a struct to keep track of our expansions and their parents. This way, we can continue to expand the nodes without having to keep track of any parents. When we hit our goal of moving everyone to the other side, we can take the winning node, and trace it backwards through it's parents until we hit the starting node (the original parent). We use this method when we print our nodes out for display. All information is stored in the final goal node.

The LISP program was tested and verified to run in CLISP GNU 2.49.93+. To run the program after loading in the file, we can simply call: (solve 15 6). The program will dynamically adjust to the input numbers, the first number: 15 in will be the group of missionaries and cannibals, while the second number: 6 is the size of the boat. If we wanted to for example, run with 20 missionaries and cannibals with a boat size of 6, the command will be: (solve 20 6).

An example run with:
(load "mc.lisp")
(solve 15 6)



We start both missionaries and cannibals on the left side. The missionaries and cannibals are denoted by M's and C's respectively. The ^ represents the boat, and the -'s is the river. We can see the boat switch sides every along with the missionaries and cannibals they shuttle across. While I printed the boat's

location manually, if required, one can find each boat's position as it is stored in the struct containing the current node, the parent node, and the boat's position.

Question 2:

In the second question, I implemented the 8-puzzle problem in LISP. For the 8-puzzle problem, we have 8 tiles in a 9-tile box. One of the tiles is empty, and we use this empty tile space to shift items around. Because they are tiles, they are locked in movement; they can only travel (if allowed) up, down, left, or right. We also cannot go outside the box, so in order to reach our goal, we move one tile at a time in a direction towards our intended goal.

As required by the assignment, I used the A* algorithm to search and find the best path to the goal. I find the heuristic value of each node by calculating the misplaced tiles. By utilizing this, along with the g-score, which is our current depth in, we can calculate the f-score. The lowest f-score will determine which path we take, then we implement the strategy again to find the next move.

By utilizing an opened and closed list, we can move our items around from opened to close if it is the correct path. We also need to ensure that we do not repeat old moves, so we will always need to check against our closed list.

Since we know we have limited movement, and we have a set 3x3 tile space, I manually calculated the possible movement positions. For example, if we're in position 1, that is the top left, we know that we can only move right or down. Since the assignment did not specify or require a dynamic tile space, the possible movements are set in code.

Each possible movement is kept track of, and its h-score, g-score, and f-score are calculated. We keep track of each state in a struct that contains the current position, its parent position, the h-score, g-score, and f-score. Since we must calculate all of these at every step, when it has returned to choose the next position, we can find the lowest f-score, choose it to move the tile, then repeat the process.

As a requirement of the assignment, I also have it checked to make sure the puzzle is solvable. I calculate initially the number of inversions, and if it's even, then it is solvable. If it's odd, it is not. This is all calculated before we perform any actions in order to ensure we have a smoother runtime.

When testing, I found I was able to solve most of the puzzles that were given. The assigned puzzle ((E,1,3), (4,2,5), (7,8,6)) was quickly and easily solvable. The assignment did not specify having a dynamically changing goal, so the only goal that was ever set was ((1,2,3), (4,5,6), (7,8,E)). I have not tested anything other than this goal, and I do not believe this program will work properly if the goal is different. I do not believe how I check for feasibility will work with any other goal as it does the inversion in a specific way. The rest of the program and calculating may work, but it has yet to be tested.

When attempting additional puzzles for testing, I found that ones that are supposed to work worked just fine, while ones that are infeasible are noted immediately. Puzzles such as ((1,8,2), (E,4,3), (7,6,5)) or ((E,1,3), (4,2,5), (7,8,6)), will solve quickly and not have any issues. I did find a puzzle that was solvable but would crash on my program. It appears that when the depth is too large, it will cause a crash. When testing this puzzle, I used an online solver and found that it takes a depth of over 600 to solve, and my program does not seem to appreciate this large depth. When solving with the smaller depths, it does not cause any issues.

Similar to question one, there are no checks in place for invalid input. We will assume that the input is correct and that we have the correct list. I do not have any checks to check if there is a 0 in the input. We must only pass correct into for the program.

The LISP program was tested and verified to run in CLISP GNU 2.49.93+. To run the program after loading in the file, we can simply call: (solve (list ‘0 ‘1 ‘3 ‘4 ‘2 ‘5 ‘7 ‘8 ‘6)). We use a single 2D list to represent our tiles, as we know it is a 3x3 tile set, we can split every three to represent the next line. In place of the empty tile, we also use the number 0 instead. As stated before, since the goal was not specified to be dynamic, the is set statically in the program. In order to test more starting lists, you can simply change the list to be different, for example: (solve (list ‘1 ‘8 ‘2 ‘0 ‘4 ‘3 ‘7 ‘6 ‘5)).

An example with:
(load “8_puzzle.lisp”)
(solve (list '0 '1 '3 '4 '2 '5 '7 '8 '6))

```
[1]> (load "8_puzzle.lisp")
;; Loading file 8_puzzle.lisp ...
;; Loaded file 8_puzzle.lisp
#P"/home/daniel/School/CSE 4308 - AI/PA1/8_puzzle.lisp"
[2]> (solve (list '0 '1 '3 '4 '2 '5 '7 '8 '6))
Start    Goal
0 1 3    1 2 3
4 2 5    4 5 6
7 8 6    7 8 0
------
0 1 3
4 2 5
7 8 6
------
1 0 3
4 2 5
7 8 6
------
1 2 3
4 0 5
7 8 6
------
1 2 3
4 5 0
7 8 6
------
1 2 3
4 5 6
7 8 0
------
Total nodes expaned: 4
```

I printed the output in an easy-to-read format the represents the 3x3 tile set. At the top, we can easily see our starting position and our goal position. Then we can see the printout of each movement as we traverse towards our goal. The final board is the same as the goal board, then we also display how many nodes were expanded in order to reach our goal.

In order to print out, we can easily traverse our closed list since that list has only the ones, we use for going forward. But like my first question, the struct for each step contains its parents’ nodes. If required, we can traverse backwards through the final node to reach the first node, though not required this time because we have opened and closed lists keeping track of everything in the A* algorithm.