ECM2423 Written Answers

Note the submission file contains a Readme. Please consult this when marking.

Question 1

1.1

I would frame this as a search problem that is looking for the shortest path to get from the start state to the goal state. This can further be split into more searching by saying that we are trying to find the most optimistic path to take at each intermediate state. We are using a function to search for which path is most likely going to give us an optimal solution.

1.2

1.2.1

The A* algorithm I am creating will take a start state and a goal state. It will then produce a path from one to the other. This should be the optimal path you can take. To do this is will generate the possible moves it can make from its current state, starting with the start state. It will then add the current state into traversed and the new states into waiting and then evaluate which of these moves is closest to the goal state. It will then start again with the current state being the one that is closets to the goal state in waiting. It will repeat this process until the goal has been reached. It prevents loops and errors by checking that the move generated is a viable move, not moving the empty state out of the puzzle and that it hasn't already been added to the traversed or waiting lists. The evaluation of each state will take place by a heuristic, one of the two described bellow. The output will be all the traversed nodes.

1.2.2

The admissible functions I am going to be using are going to be Hamming and Manhattan

Hamming

Hamming heuristic is where you measure the tiles in the 8-puzzle which are currently out of place. Hamming is admissible as it never overestimates the cost to the goal. This is because the cost it evaluates to will be the sum of the tiles currently in the wrong place, all that are currently in the wrong place will have to be moved at least once to get to the right place. This makes the h(g) ≤ the actual path.

Manhattan

Manhattan heuristic is where you measure how far a each tile is from where it should be. It will be an admissible heuristic as it will count the sum of moves each tile has to make to get to its own goal, these will all have to be made to get to the goal in the actual path. And this is only assuming we only have to make the exact distance to get to the goal node, omitting moving any other tiles in the process. Therefore the cost of the actual path will certainly never be any higher than our estimation making it admissible.

I have decided to use these functions as they demonstrate measurements of what it means to be getting to the goal. Is it to be in or out of the right place or to be getting closer to the goal itself.

1.2.3

The code is available in the A*Python.py file. The code should be commented so you can follow the reasoning I have used. I have decided to have the user pick which heuristic they would like to use before the code runs instead of having 2 scripts as this will limit the amount of repeated code.

Testing

I have decided to pick my own start state, and "." Will represent my empty state

```
My start state will be    4 7 2
                          1 6 5
                          3 8 .
```

And my goal state will be the goal in the specification

These are more than 5 moves apart

Proved here

```
. 1 2     1 . 2     1 4 2     1 4 2     1 4 2     1 4 2     . 4 2
3 4 5     3 4 5     3 . 5     3 7 5     3 7 5     . 7 5     1 7 5
6 7 8     6 7 8     6 7 8     6 . 8     . 6 8     3 6 8     3 6 8

4 . 2     4 7 2     4 7 2     4 7 2
1 7 5     1 . 5     1 6 5     1 6 5
3 6 8     3 6 8     3 . 8     3 8 .
```

1.2.4

Results

The raw results can be found the A* Raw Data file in the submission file

The summary of results

When running the A* algorithm both of the heuristics managed to find the goal state. They also both did it in the same amount of moves. The only difference was the time. Manhattan was significantly quicker on my machine.

This was completely difference from what I expected as the Manhattan has more computations to do. However these are mathematical computations not comparative computations so this does make sense. I also expected the Manhattan to do it in less moves. This may only occur for particular start states.

In terms of the results, they both seem to have followed the path that has gotten them to the goal state so they are both successful. Although Manhattan's time complexity is more favourable.

1.3

This version of the code in the A*PythonGeneral.py file. Here the user is prompted to enter the type of heuristic they would like to use followed by the goal and start state of their choice in the form :

```
X X X
X X X
X X X
```

Where one X is a . and the others are all unique numbers

Question 2

2.1

Design of my EA

In the design of my EA I have made many design decisions, here I am going to try to explain a few of them.

The first thing my EA does is it takes the information from the file and puts it into a viable form. I have decided to put the solutions into the form of a list of numbers. This solution had different pros and cons but it was particularly useful when solving the problem of not wanting to change some of the numbers during initial population. These solutions then went into a list (making it a 2D list) which is the population.

The fitness function I chose was counting through the horizontal, vertical line and boxes to check how many unique numbers are in each. The sum of these is then the fitness function. This makes a maximal fitness of 243 (81*3). I felt this was the best solution as it takes in the rules for the game and what a actual solution should look like. It was also a better alternative than just giving it a target solution as this would most likely not be a unique solution and constraining the program like this would give less of a chance of getting a viable solution. The program achieves this fitness through a system of loops.

For my crossover I have gone quite simple. My crossover function will pick a random pair from the best of the population(from select pop) and then will pick a random one of these to put into the next generation or put in a new solution. I have decided to do it this way because when I was testing different methods I was finding that anything where you take the numbers from both sides, (with by randomness or by fitness) and add them into one solution caused the fitness to be drastically reduced. This is because if you take a horizontal line you are removing unique solutions from the vertical lines and blocks and vice versa. This leads to a very repetitive solution and a low fitness, this way the population fitness has a greater chance of increasing, and we are still taking something from the population above. The function also has a chance of putting in a new solution. The chance is 1/4 of the chance of putting in an existing one. When this occurs it will run the same process that was used to populate the solutions at the beginning but this time it only performs it for one solution.

I have tried many different ways to do crossover and found that many of the methods decreased the fitness that was produced and this still does happen with this method from time to time as it is based largely around randomness, however this method appears to be where this occurs least.

When I mutate my functions I select an amount that I would like to mutate, this is at 25-generation(up to 24) meaning as you get further along your generations you start to converge to an optimal answer and stop mutating as much. It then loops through this many times selected a index of the list of which I am able to change, meaning it isn’t in the permanent index list, and then will change that number to a different randomly generated number.

To initialise the population I have taken the reading of the file (where I have removes the characters we won’t need), and used this as a template. I also have made a list of the indexes of the numbers that cannot change throughout the course of the program. With these 2 things I then populate the templates for the amount needed (population size). I then fill these templates out by adding random numbers into the indexes that I am able to change. These new solutions are then added to a list to make up the total population.

To select my population ready for the next run I have taken the population size which will vary from one test to another, and then have a truncation rate of 0.8. Meaning when you run select it will get the fitnesses along with the list in a zipped list of tuples and then sort these to get the highest fitness first. This list will then be cut off by 80% of the population size and the remaining 20% will go through breeding and mutation to get the next generation. I found that 0.8 was the best truncation rate as it allowed for only the best fitnesses to be picked and also allowed room for more mutation of higher fitnesses and higher levels of new solutions in the crossover process.

Other design decisions I have made include having 24 generations. I have made this decision because I found that after this we had pretty much reached a steady state and there were no more improvement. This will be my termination criteria.

2.2

The code for my EA can be found in the SudukoEA.py file. The code is currently set up to run the tests in the specification.

These are the results of my test :

Grid 1

Population Size	Run 1 Average	Run 2 Average	Run 3 Average	Run 4 Average	Run 5 Average	Overall Average
10	165.375	163.417	165.625	162.417	163.75	164.1168
100	165.542	166	163.083	165.625	165.417	165.1334
1000	163.333	165.333	164.458	165.958	164.167	164.6498
10000	163.792	164.292	164.125	163.417	164.875	164.1002

Grid 2

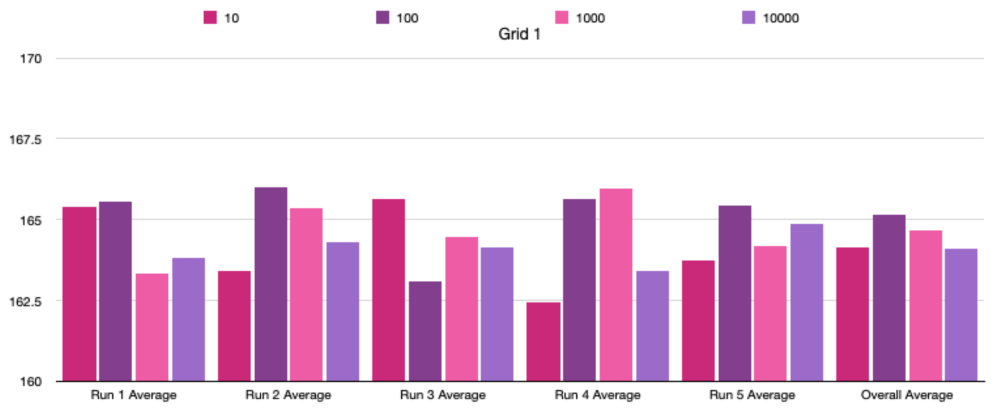
Population Size	Run 1 Average	Run 2 Average	Run 3 Average	Run 4 Average	Run 5 Average	Overall Average
10	168.667	167.833	169.167	167.667	168.25	168.3168
100	169.167	168.75	169.333	169.208	167.917	168.875
1000	167.833	169.042	168.542	167.25	168.791	168.2916
10000	169.833	168.917	167	168.208	166.708	168.1332

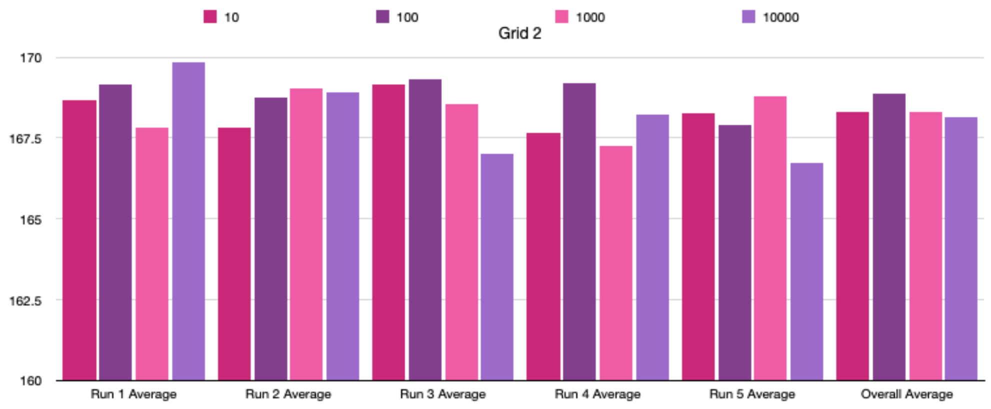
Grid 3

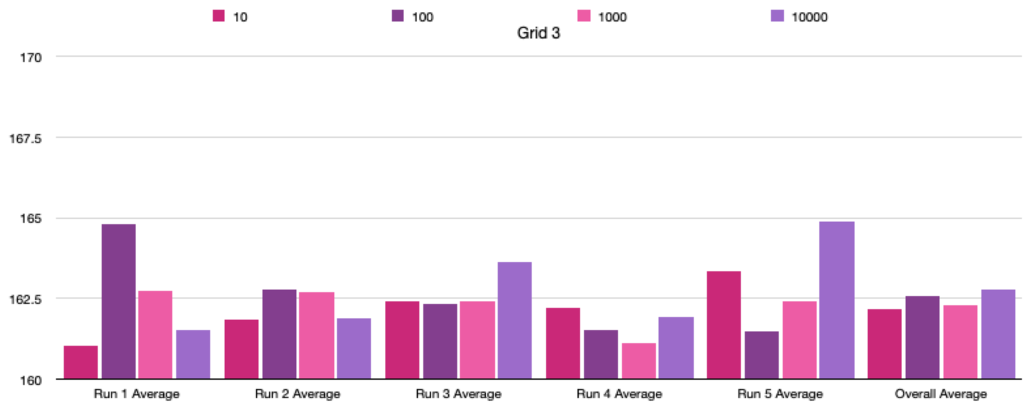
Population Size	Run 1 Average	Run 2 Average	Run 3 Average	Run 4 Average	Run 5 Average	Overall Average
10	161.042	161.833	162.417	162.208	163.333	162.1666
100	164.792	162.792	162.333	161.542	161.5	162.5918
1000	162.75	162.708	162.417	161.125	162.417	162.2834
10000	161.547	161.875	163.625	161.917	164.875	162.7678

The raw data from these tests is included in the zip submission for more information

I have also created some graphs to better demonstrate these numbers







2.2.1

Overall 100 had the best average. However 10000 achieved the maximum score. However as the 10000 seems to be a bit of a fluke based on the information I have available, I would say that 100 is better based on its more consistent outcomes.

2.2.2

I think that the reason for this is because when you use a higher population size the power of the fittest answers gets lost when you create and mutate offspring. If you have a very good answer it can be hard to take what is good about that solution and pass it onto the next generation as there are many more to pick from when producing offspring and the gene pool gets too full. I also think that when you have a higher population it becomes harder for the solution to begin to converge into a good solution, preventing one dominant path from being taken by the algorithm. You don't have so many of these issues with smaller population sizes. However I think with the 10 population size we drastically reduce the amount of solutions we generate so much that it cannot explore a vast enough area before it is converging to a solution. There are only 10 unique solutions at the beginning and this number really doesn't go up enough during the course of the program so it lacks the large numbers for the randomness in the code to produce good results.

2.2.3

The data I have from my EA appears to suggest that the 2nd grid was the easiest and the 3rd was the hardest as these appear to have the best and worst results retrospectively. However I think my EA has also improved the partial solution more for grid3 from the partial solution we started with.

2.2.4

I think these results are because of how many inputs the puzzles start with:

Grid1 having 29 start numbers and having an end overall score of 164

Grid2 having 33 start numbers and having an end overall score of 168

Grid3 having 23 start numbers and having an end overall score of 162

So this could mean a couple of things.

First it appears that given a more solved puzzle my EA will end up with a better solution. It has the higher fitness and therefore is nearer to the goal.

But also it appears that during the same amount of runs in the exact same conditions the runs that started out with a less solved grid have filled in more of that grid. This makes sense as given more opportunity to mutate and fill in new entries to these spaces, we then get more of an increase in the amount that is filled in correctly.

I think these results actually show that in the end the solution from more filled out starts is going to give a more overall accurate solution, however if we were to take away the starting fitness of the starting solutions, the less solved starters would come out on top.

2.2.5

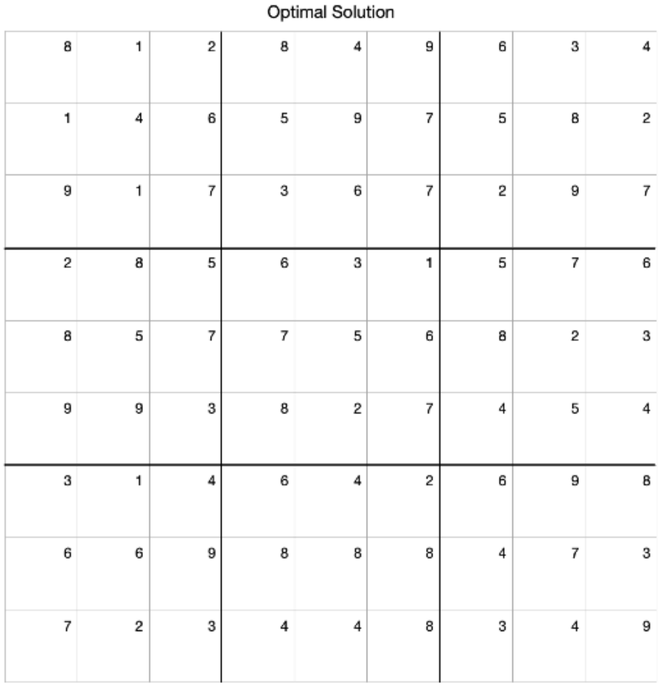
During this development and design process I tested many different ways of mutation and crossover functions. The one I finished with was the best one I could think of during this time. It works, however I'm sure it is not the most optimal as it runs mostly on randomness. Maybe further testing into better offspring creation functions would yield a better result.

Also I have been testing using different generation amounts, and different truncation percentages. I found these to be the best numbers I could come up with, however maybe test what would happen if you were to use these in a more complex way. Base them off what population size you have or your best fitness from the first generation etc. These may also yield better results, particularly for larger population sizes.

Conclusion

At the end of my experiment this was my most optimal solution from my most optimal test. There are a few repeats in it but overall I think its pretty good.

Optimal Solution



8	1	2	8	4	9	6	3	4
1	4	6	5	9	7	5	8	2
9	1	7	3	6	7	2	9	7
2	8	5	6	3	1	5	7	6
8	5	7	7	5	6	8	2	3
9	9	3	8	2	7	4	5	4
3	1	4	6	4	2	6	9	8
6	6	9	8	8	8	4	7	3
7	2	3	4	4	8	3	4	9