CS440 Assignment

Manel Bermad and Robert Bonagrua

Task 1

Environment - The grid of n*n cells.
State Space - Array of cells that represent a graph.
Actions - Move left, right, up or down to another cell based on the number of jumps available to the current cell.
Perception/Observations - The neighboring cells the agent is able to currently jump to.
Transition Function - Moving to a neighbor based on the number of jumps at the current cell, thus moving into a new state.
Evaluation metric - K value. The total number of jumps to get to goal state.

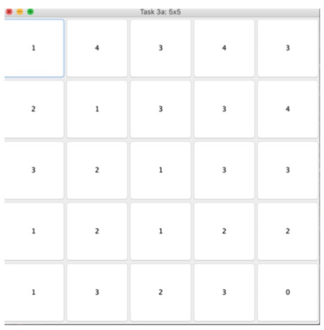
Task 2

Sample Output from program:

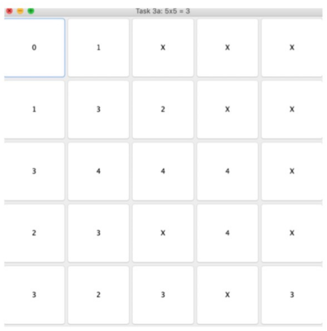
```
TASK 2:
Please enter any size n [1..100] to generate a new graph and a
visualization of its solution.
(NOTE: values under 40 are preferable in order to see visualization
clearly).

10
Creating 10-by-10 sized puzzle...
Finished creating 10-by-10 sized puzzle.
The solution is: 7
The path to the solution is: (8 , 0)(9 , 0)(2 , 0)(3 , 0)(3 , 9)(7 , 9)(9
, 9)
Found in: 0.19305 ms
```

Task 3

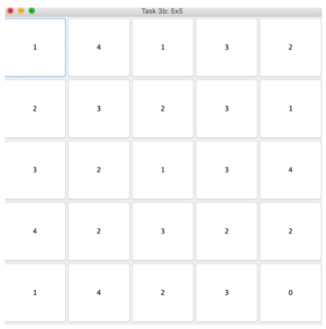
Puzzles generated based on Task 3 instructions:


Task 3a: 5x5

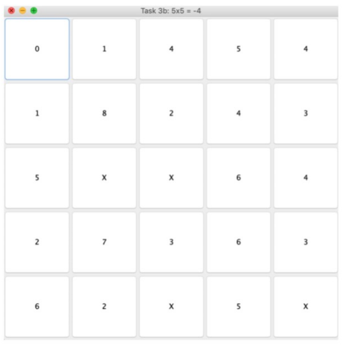
1	4	3	4	3
2	1	3	3	4
3	2	1	3	3
1	2	1	2	2
1	3	2	3	0


Task 3a: 5x5 = 3

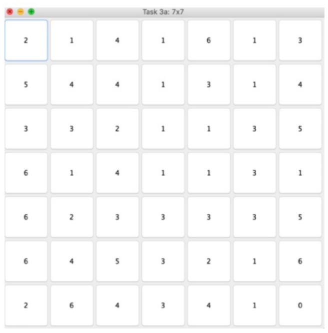
0	1	X	X	X
1	3	2	X	X
3	4	4	4	X
2	3	X	4	X
3	2	3	X	3


Task 3b: 5x5

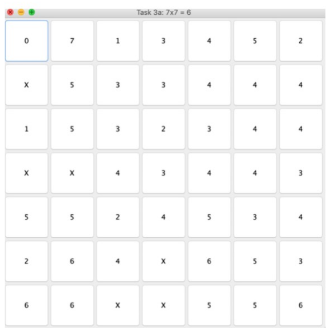
1	4	1	3	2
2	3	2	3	1
3	2	1	3	4
4	2	3	2	2
1	4	2	3	0


Task 3b: 5x5 = -4

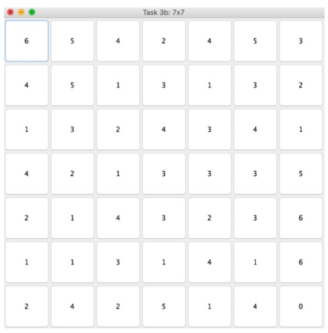
0	1	4	5	4
1	8	2	4	3
5	X	X	6	4
2	7	3	6	3
6	2	X	5	X


Task 3a: 7x7

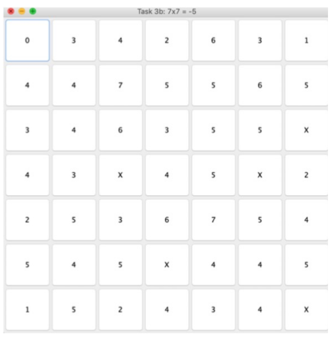
2	1	4	1	6	1	3
5	4	4	1	3	1	4
3	3	2	1	1	3	5
6	1	4	1	1	3	1
6	2	3	3	3	3	5
6	4	5	3	2	1	6
2	6	4	3	4	1	0


Task 3a: 7x7 = 6

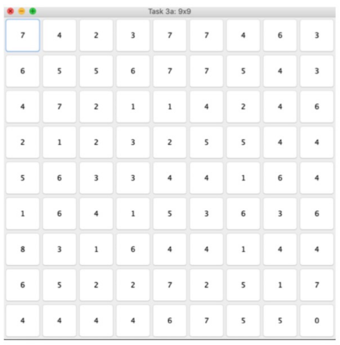
0	7	1	3	4	5	2
X	5	3	3	4	4	4
1	5	3	2	3	4	4
X	X	4	3	4	4	3
5	5	2	4	5	3	4
2	6	4	X	6	5	3
6	6	X	X	5	5	6


Task 3b: 7x7

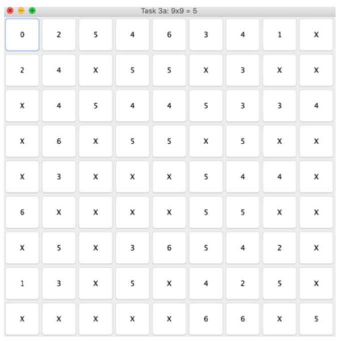
6	5	4	2	4	5	3
4	5	1	3	1	3	2
1	3	2	4	3	4	1
4	2	1	3	3	3	5
2	1	4	3	2	3	6
1	1	3	1	4	1	6
2	4	2	5	1	4	0


Task 3b: 7x7 = -5

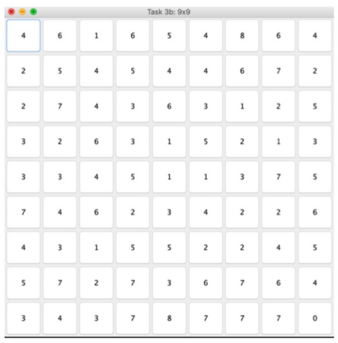
0	3	4	2	6	3	1
4	4	7	5	5	6	5
3	4	6	3	5	5	X
4	3	X	4	5	X	2
2	5	3	6	7	5	4
5	4	5	X	4	4	5
1	5	2	4	3	4	X


Task 3a: 9x9

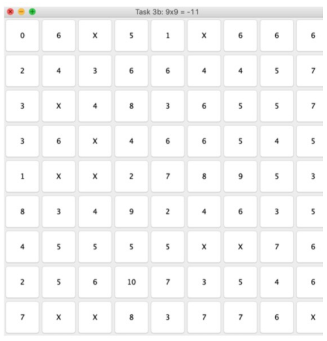
7	4	2	3	7	7	4	6	3
6	5	5	6	7	7	5	4	3
4	7	2	1	1	4	2	4	6
2	1	2	3	2	5	5	4	4
5	6	3	3	4	4	1	6	4
1	6	4	1	5	3	6	3	6
8	3	1	6	4	4	1	4	4
6	5	2	2	7	2	5	1	7
4	4	4	4	6	7	5	5	0


Task 3a: 9x9 = 5

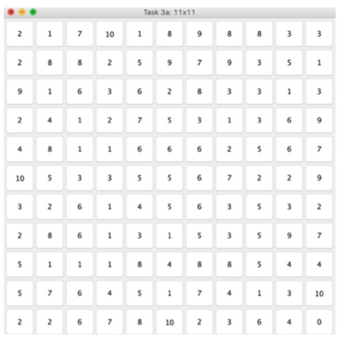
0	2	5	4	6	3	4	1	X
2	4	X	5	5	X	3	X	X
X	4	5	4	4	5	3	3	4
X	6	X	5	5	X	5	X	X
X	3	X	X	X	5	4	4	X
6	X	X	X	X	5	5	X	X
X	5	X	3	6	5	4	2	X
1	3	X	5	X	4	2	5	X
X	X	X	X	X	6	6	X	5


Task 3b: 9x9

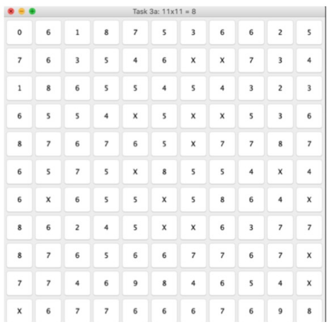
4	6	1	6	5	4	8	6	4
2	5	4	5	4	4	6	7	2
2	7	4	3	6	3	1	2	5
3	2	6	3	1	5	2	1	3
3	3	4	5	1	1	3	7	5
7	4	6	2	3	4	2	2	6
4	3	1	5	5	2	2	4	5
5	7	2	7	3	6	7	6	4
3	4	3	7	8	7	7	7	0


Task 3b: 9x9 = -11

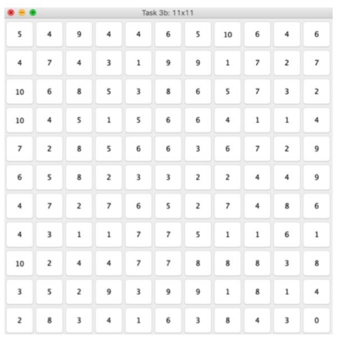
0	6	X	5	1	X	6	6	6
2	4	3	6	6	4	4	5	7
3	X	4	8	3	6	5	5	7
3	6	X	4	6	6	5	4	5
1	X	X	2	7	8	9	5	3
8	3	4	9	2	4	6	3	5
4	5	5	5	5	X	X	7	6
2	5	6	10	7	3	5	4	6
7	X	X	8	3	7	7	6	X


Task 3a: 11x11

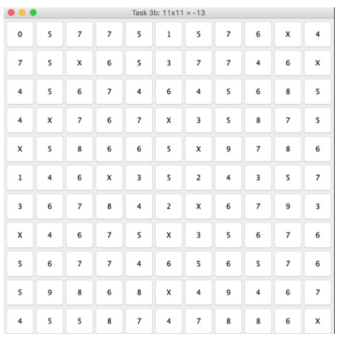
2	1	7	10	1	8	9	8	8	3	3
2	8	8	2	5	9	7	9	3	5	1
9	1	6	3	6	2	8	3	3	1	3
2	4	1	2	7	5	3	1	3	6	9
4	8	1	1	6	6	6	2	5	6	7
10	5	3	3	5	5	6	7	2	2	9
3	2	6	1	4	5	6	3	5	3	2
2	8	6	1	3	1	5	3	5	9	7
5	1	1	1	8	4	8	8	5	4	4
5	7	6	4	5	1	7	4	1	3	10
2	2	6	7	8	10	2	3	6	4	0


Task 3a: 11x11 = 8

0	6	1	8	7	5	3	6	6	2	5
7	6	3	5	4	6	X	X	7	3	4
1	8	6	5	5	4	5	4	3	2	3
6	5	5	4	X	5	X	X	5	3	6
8	7	6	7	6	5	X	7	7	8	7
6	5	7	5	X	8	5	5	4	X	4
6	X	6	5	5	X	5	8	6	4	X
8	6	2	4	5	X	X	6	3	7	7
8	7	6	5	6	6	7	7	6	7	X
7	7	4	6	9	8	4	6	5	4	X
X	6	7	7	6	6	6	7	6	9	8


Task 3b: 11x11

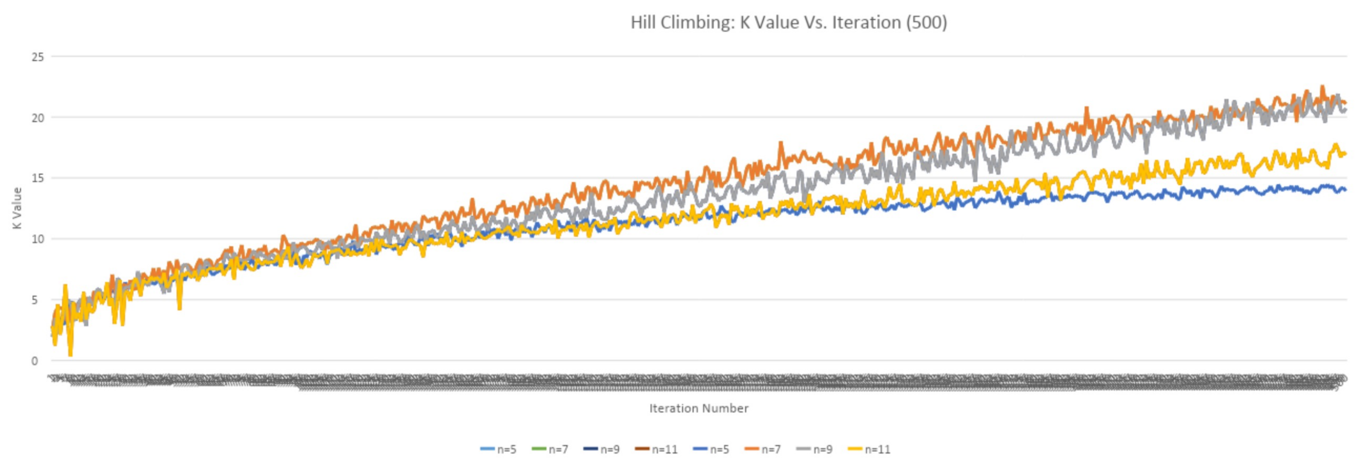
5	4	9	4	4	6	5	10	6	4	6
4	7	4	3	1	9	9	1	7	2	7
10	6	8	5	3	8	6	5	7	3	2
10	4	5	1	5	6	6	4	1	1	4
7	2	8	5	6	6	3	6	7	2	9
6	5	8	2	3	3	2	2	4	4	9
4	7	2	7	6	5	2	7	4	8	6
4	3	1	1	7	7	5	1	1	6	1
10	2	4	4	7	7	8	8	8	3	8
3	5	2	9	3	9	9	1	8	1	4
2	8	3	4	1	6	3	8	4	3	0


Task 3b: 11x11 = -13

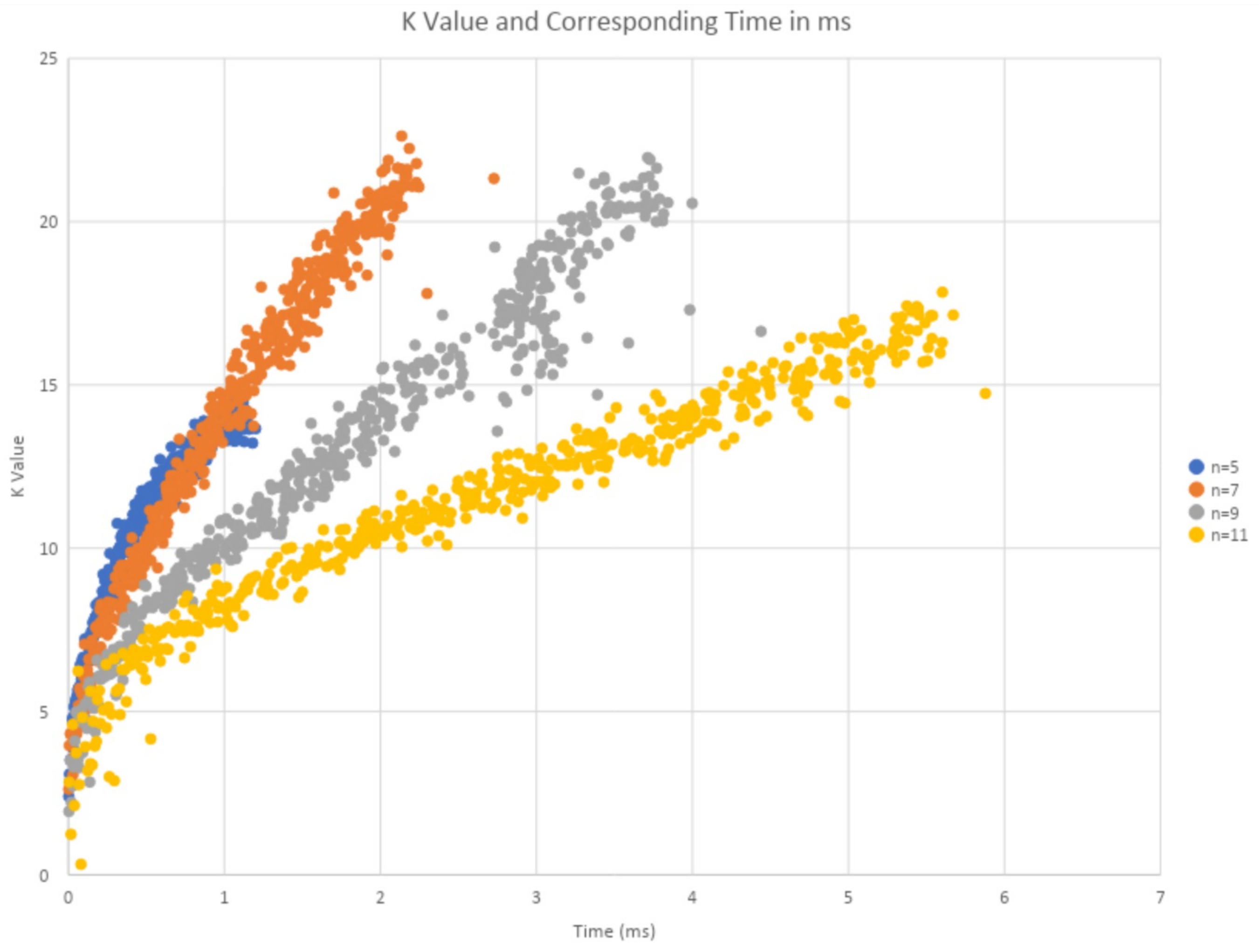
0	5	7	7	5	1	5	7	6	X	4
7	5	X	6	5	3	7	7	4	6	X
4	5	6	7	4	6	4	5	6	8	5
4	X	7	6	7	X	3	5	8	7	5
X	5	8	6	6	5	X	9	7	8	6
1	4	6	X	3	5	2	4	3	5	7
3	6	7	8	4	2	X	6	7	9	3
X	4	6	7	5	X	3	5	6	7	6
5	6	7	7	4	6	5	6	5	7	6
5	9	8	6	8	X	4	9	4	6	7
4	5	5	8	7	4	7	8	8	6	X

Task 4

The graph below shows how the value function increases as the number of iterations increases.



The graph below shows how computation time increases as the difficulty (or k value) of the puzzle increases



Both of the graphs shown for Task 4 are a result of the HillCimbing function being run 50 times, at every iteration *i* from 1 to 500. The values plotted are the values averaged over 50 repetitions.

Task 5

The SPF algorithm, as expected, runs longer than the BFS. This is likely due to the fact that there is larger overhead used for creating and implementing the priority queue.

As expected, as the size or difficulty of the problem increase, so does the runtime.

Task 6

We were initally surprised to see that the BFS algorithm outperformed the A* Search algorithm. This was the case for all sizes n = {5, 7, 9 11}. Our best guess is that this is likely do again to the overhead cost of implementing a priority queue in Java.

Sample output for BFS, SPF, and A* respectively with size 11:

```
The solution is: 6
The path to the solution is: (9 , 0)(7 , 0)(7 , 5)(7 , 8)(7 , 10)(10 , 10)
Found in: 0.206202 ms

The solution is: 6
The path to the solution is: (9 , 0)(7 , 0)(7 , 5)(7 , 8)(7 , 10)(10 , 10)
Found in: 0.774601 ms

The solution is: 6
The path to the solution is: (9 , 0)(7 , 0)(7 , 5)(7 , 8)(7 , 10)(10 , 10)
Found in: 0.239938 ms
```

Sample output for BFS, SPF, and A* respectively with size 500:

```
The solution is: 10
The path to the solution is: (122 , 0)(122 , 182)(122 , 155)(122 , 170)
(122 , 100)(122 , 11)(122 , 93)(182 , 93)(182 , 199)(199 , 199)
Found in: 5.526452 ms

The solution is: 10
The path to the solution is: (122 , 0)(122 , 182)(122 , 155)(122 , 170)
(122 , 100)(122 , 11)(122 , 93)(182 , 93)(182 , 199)(199 , 199)
Found in: 14.907976 ms

The solution is: 10
The path to the solution is: (122 , 0)(122 , 182)(122 , 155)(122 , 170)
(122 , 100)(122 , 11)(122 , 93)(182 , 93)(182 , 199)(199 , 199)
Found in: 7.671918 ms
```

Task 7

Genetic Algorithm Methodology

The Genetic Algorithm takes in 3 parameters: *n* (which is the dimension of the nxn puzzle), *p* (which is the number of graphs we start with, or the size of initial population), and *j*, the number of cycles (which is the number of generation or, the number of times the genetic ALG will run for). The ALG first populates the intialPopulation (array of graphs) by generating *p* random graphs of size *n* by *n*. They are then sorted in descending orders based on their *k* value (fitness value), and if *p* is odd, the last graph in that order is dropped (this makes coupling later on easy).

For Selection (GeneticSelection method), we take as input the initial population array. Elitism is applied by keeping the highest ranking graph unchanged, and then we couple graphs based on their ranking in the sorted array. For example, graph1 with graph2, graph3 with graph4 …etc. This type of pairing increases the chance of producing new puzzles that are equally as difficult— or more.

For Cross-Over (GeneticCrossOver method), we iterate through the pairs (parent1, parent2) formed in Selection and create 2 new graphs: graph1 first half of rows will be identical to the ones in parent1, and the second half of rows is identical to the ones in parent2. We do the opposite for graph2.

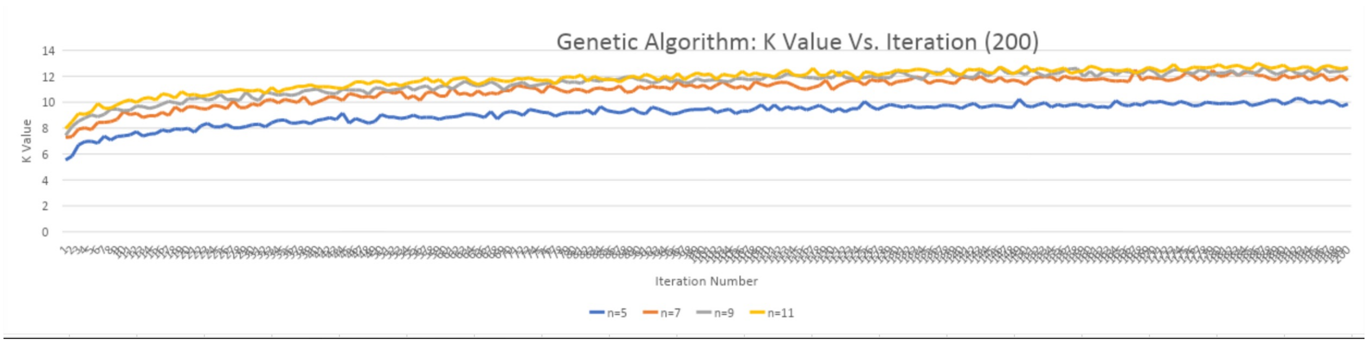
For Mutation (GeneticMutation method), we take the array of CrossOvers and mutate each graph with a probability of 90% (a random generator used). The lower this probability, the lower the chances of having a better population.

This results in a new population and the 3 genetic steps are repeated j amount of times.

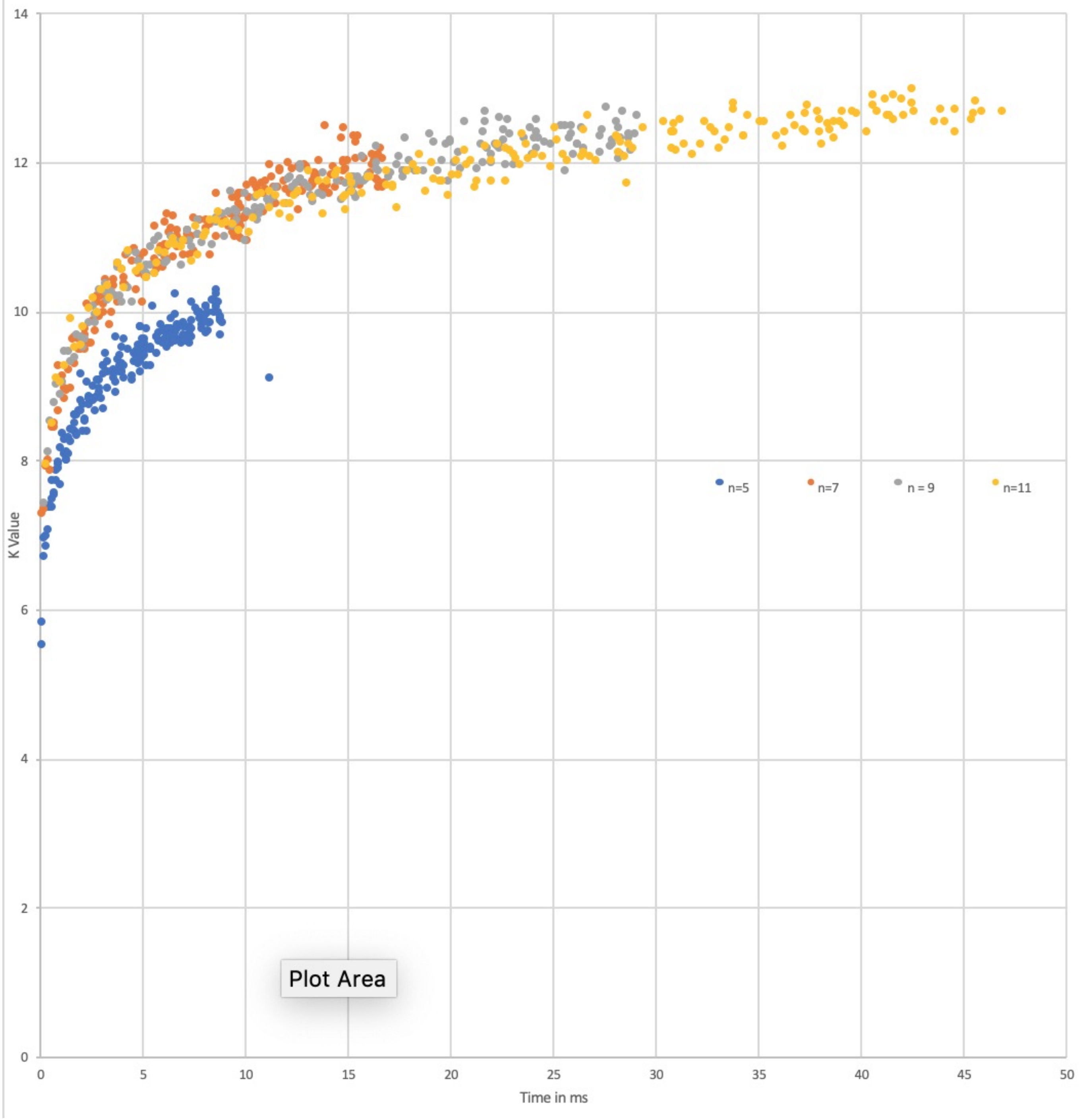
The final return statement of the algorithm is again, the highest ranked graph on the last population.

Graphs outputted by Genetic Algorithm:

The first graph below shows how as the number of iterations increases (up to 200), the K value increases only slightly.



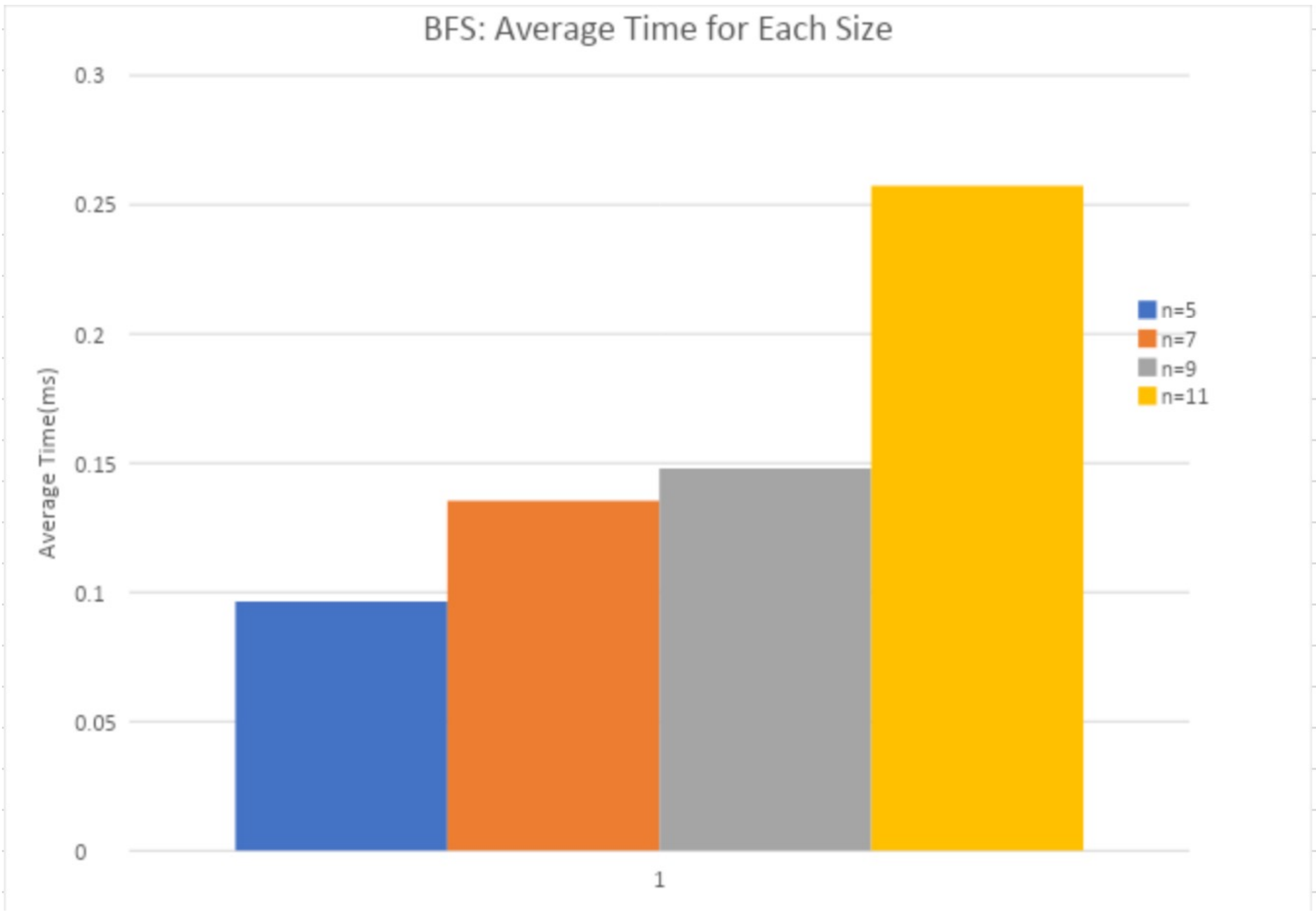
The next graph shows when the K value approaches around 12, the computation time begins to grow much faster than the K value.

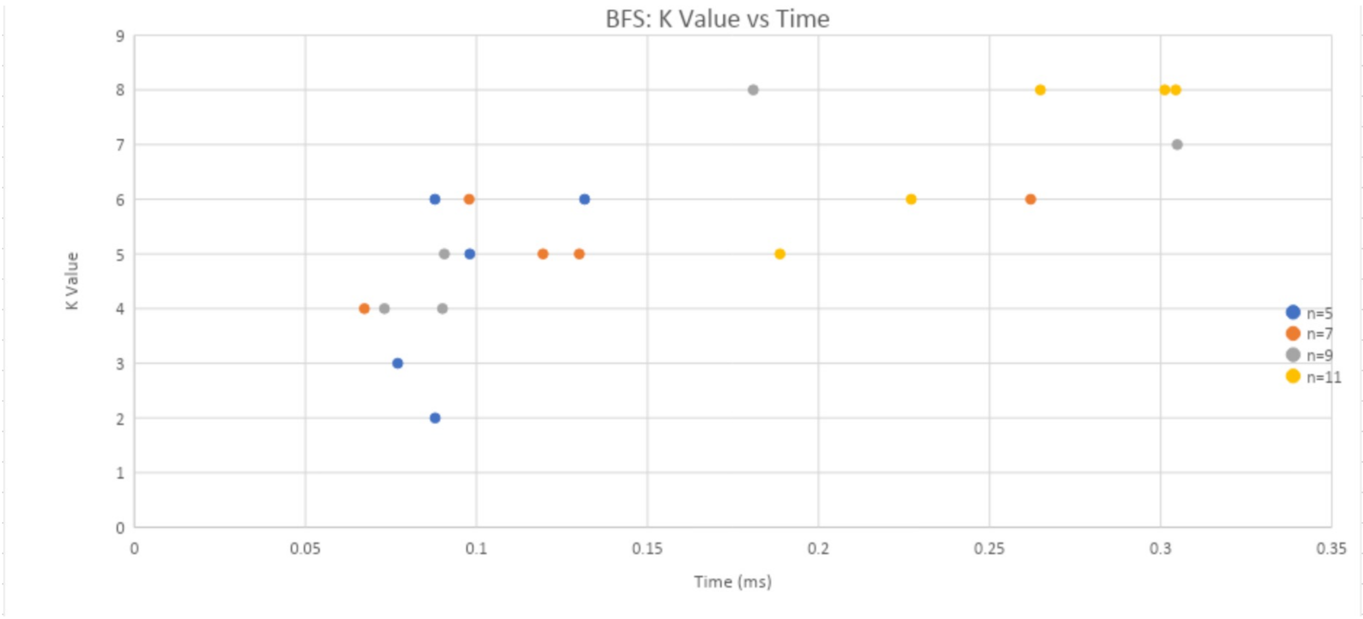


These two graphs show that despite incresing the number of iterations, the K value does not continue to grow much higher, and instead only the computation time required to generate the puzzle continues to grow.

Task 8

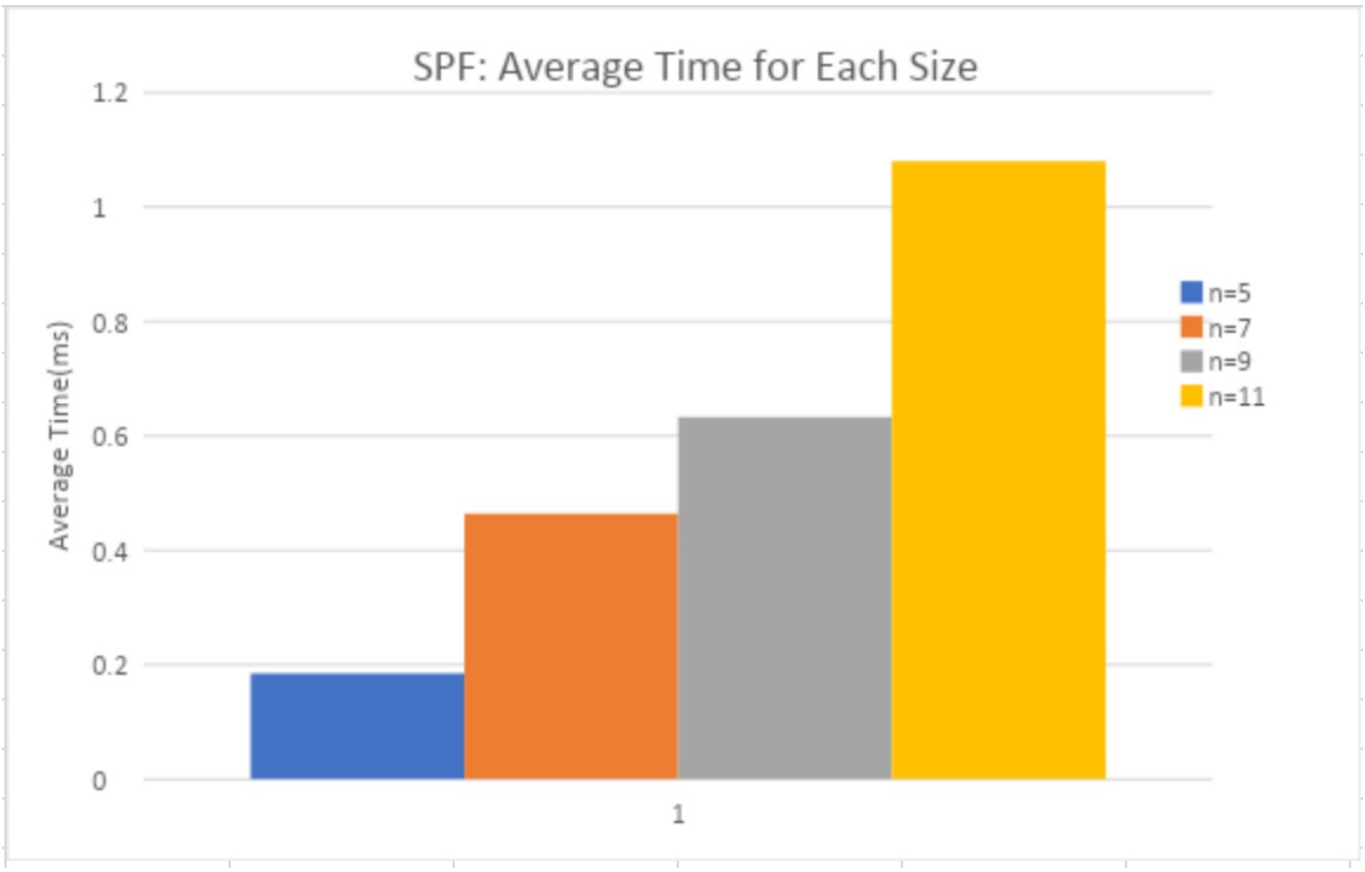
BFS

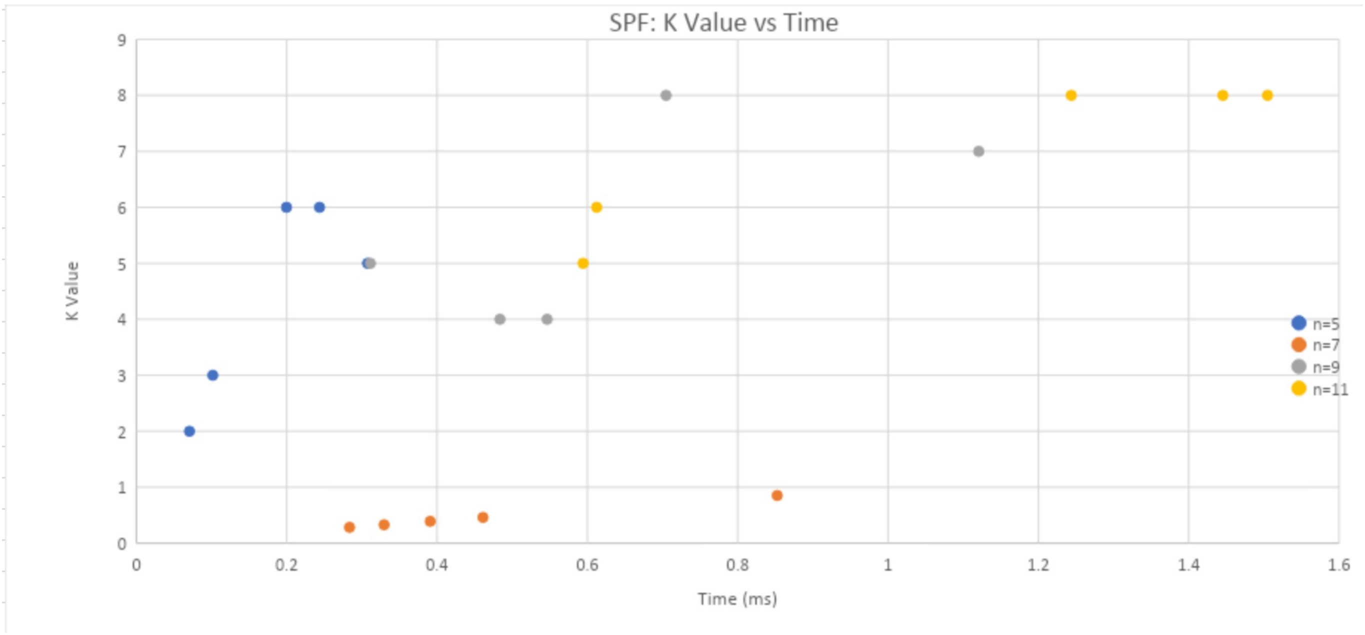




BFS is the fastest search algorithm we implemented on puzzles of these sizes.

SPF

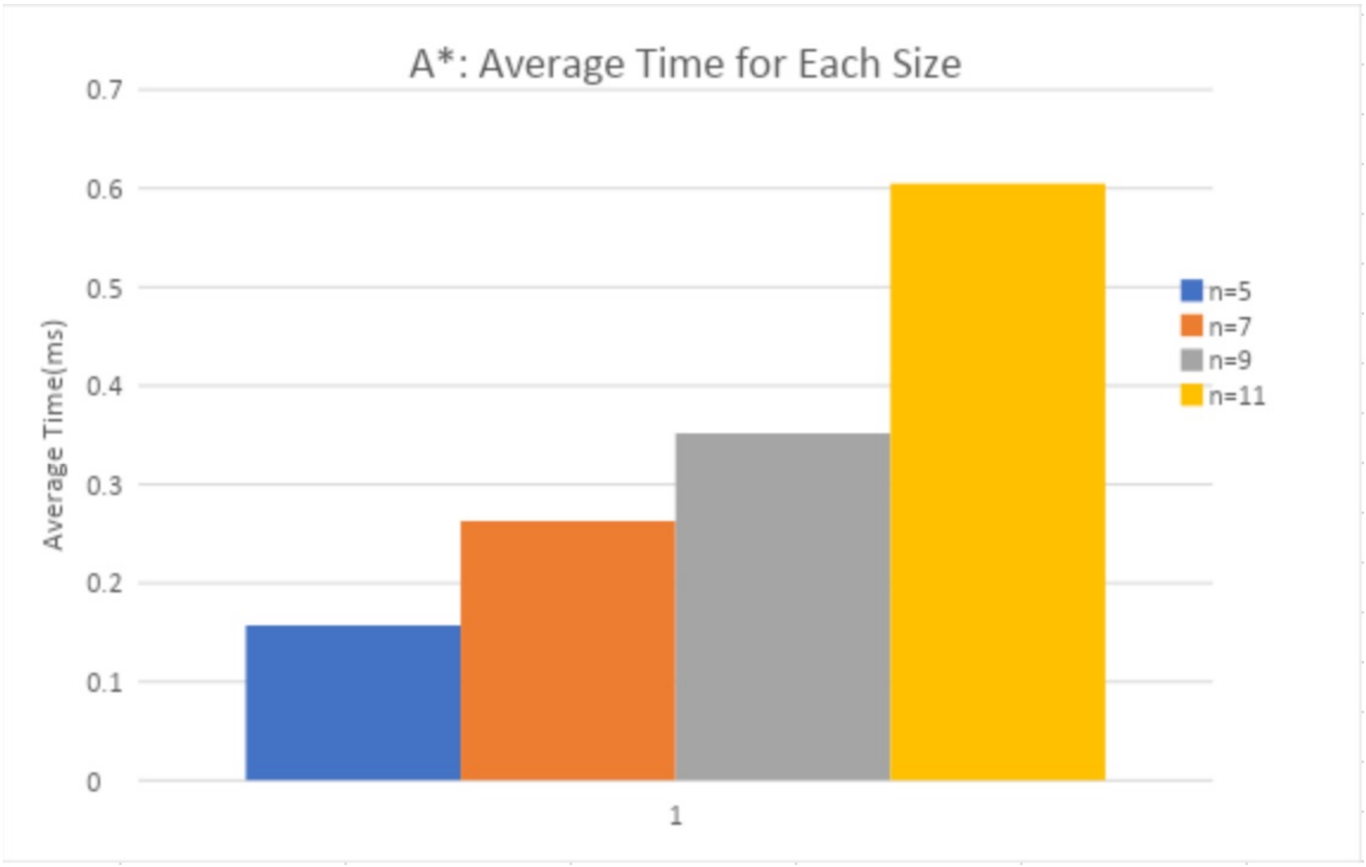


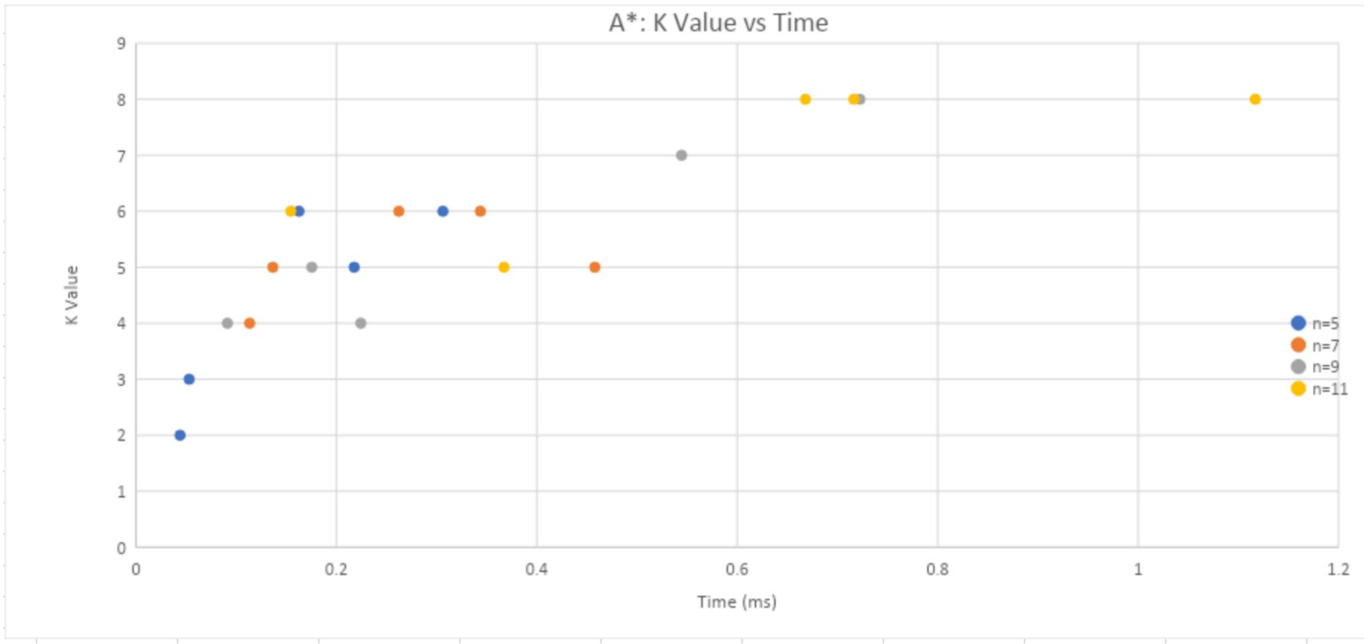


The use of SPF was always the slowest. This makes sense because with unweighted edges, it should mirror BFS exactly, with one exception: The cost of creating a Priority Queue.

Therefore, SPF should always run slower than BFS on these puzzles.

A* Search





The fact that A* Search runs faster than SPF gives us reason to believe that cost of overhead associated with the Priority Queue was it's bottleneck. Because it outperformed SPF, which is fundamentally the same algorithm as BFS when edges are weightless, we believe that we if we could omit the overhead cost of Java's PrioritQueue class in A*, we would see it run faster than BFS.

Task 9

Sample output for 1000 x 1000 sized puzzle:

```
TASK 9:
Performing Hill Climbing algorithm to create a difficult 1000-by-1000
```

```
puzzle...

BFS search on result of HillClimbing
The solution is: 16
The path to the solution is: (255 , 0)(255 , 559)(255 , 828)(680 , 828)
(680 , 413)(680 , 434)(208 , 434)(788 , 434)(661 , 434)(29 , 434)(937 ,
434)(908 , 434)(285 , 434)(285 , 31)(999 , 31)(999 , 999)
Found in: 138.045876 ms

SPF search on result of HillClimbing
The solution is: 16
The path to the solution is: (255 , 0)(255 , 559)(255 , 828)(680 , 828)
(680 , 413)(680 , 434)(208 , 434)(788 , 434)(661 , 434)(29 , 434)(937 ,
434)(908 , 434)(285 , 434)(285 , 31)(999 , 31)(999 , 999)
Found in: 1237.655531 ms

A* search on result of HillClimbing
The solution is: 16
The path to the solution is: (255 , 0)(255 , 559)(255 , 290)(534 , 290)
(534 , 383)(150 , 383)(150 , 680)(579 , 680)(579 , 777)(622 , 777)(688 ,
777)(688 , 999)(795 , 999)(190 , 999)(947 , 999)(999 , 999)
Found in: 496.9399 ms
```