

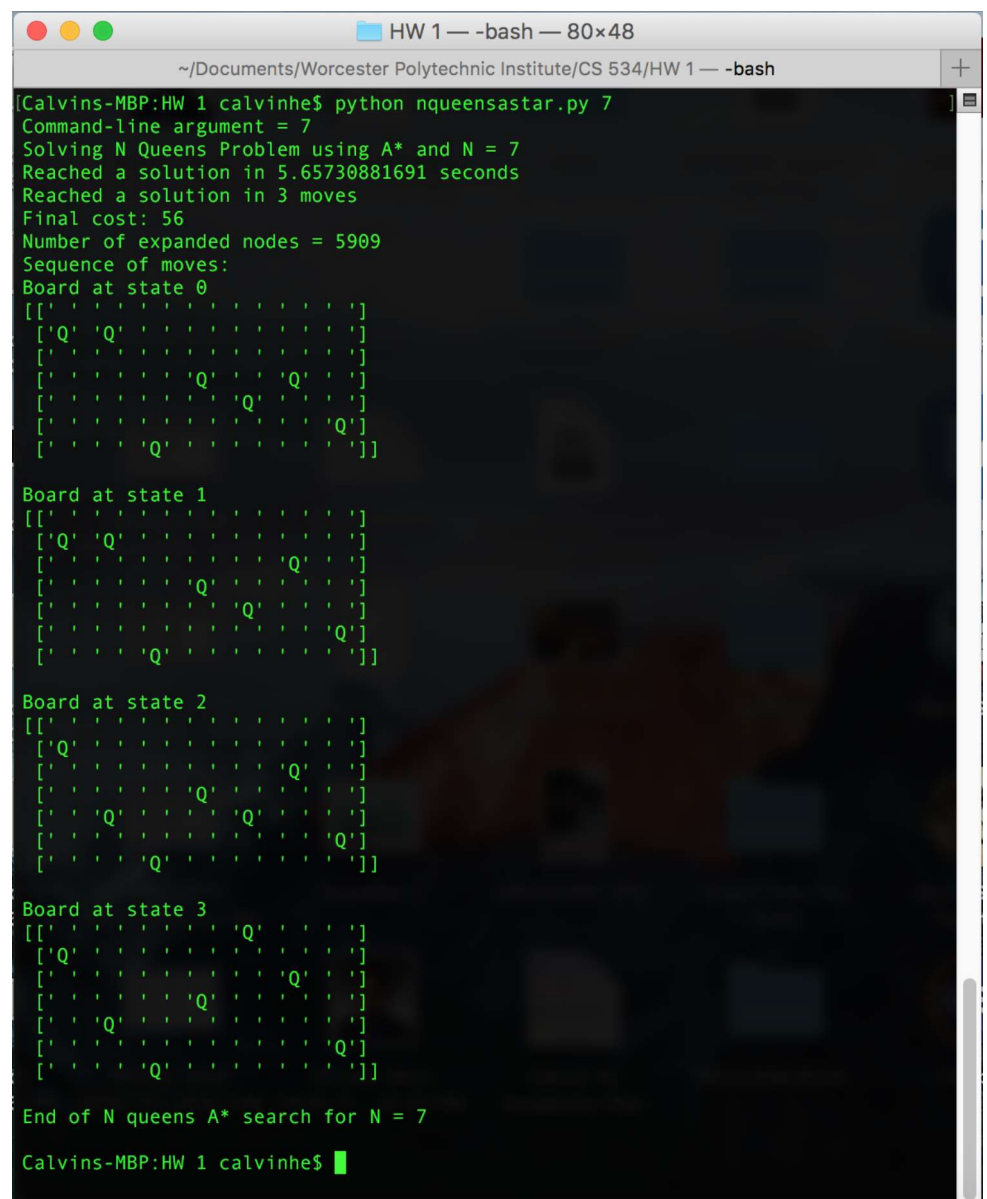
Calvin He, Max Li, Ilya Lifshits, Rohit Voleti
CS 534 Artificial Intelligence Spring 2018
Group 11
Assignment 1 Writeup
February 14, 2018

Problem 1:

Instructions to run code: To execute the program for Heavy N Queens problem using just A* Search, the filename of the Python script is **nqueensastar.py**, and the following is how to run the program on the command line or terminal:

\$ python nqueensastar.py N (where N is the number of queens)

The program would output some print statements onto the terminal after running A* search and would also use matplotlib to plot the start state and final state as a table figure for better visualization. There would be two figures that would show up with one on top of the other, so use your cursor to drag the top figure out of the way to see the figure underneath it. Close both figures and the script will finish. Here are screenshots of running nqueensastar.py with the argument N = 7 as an example:



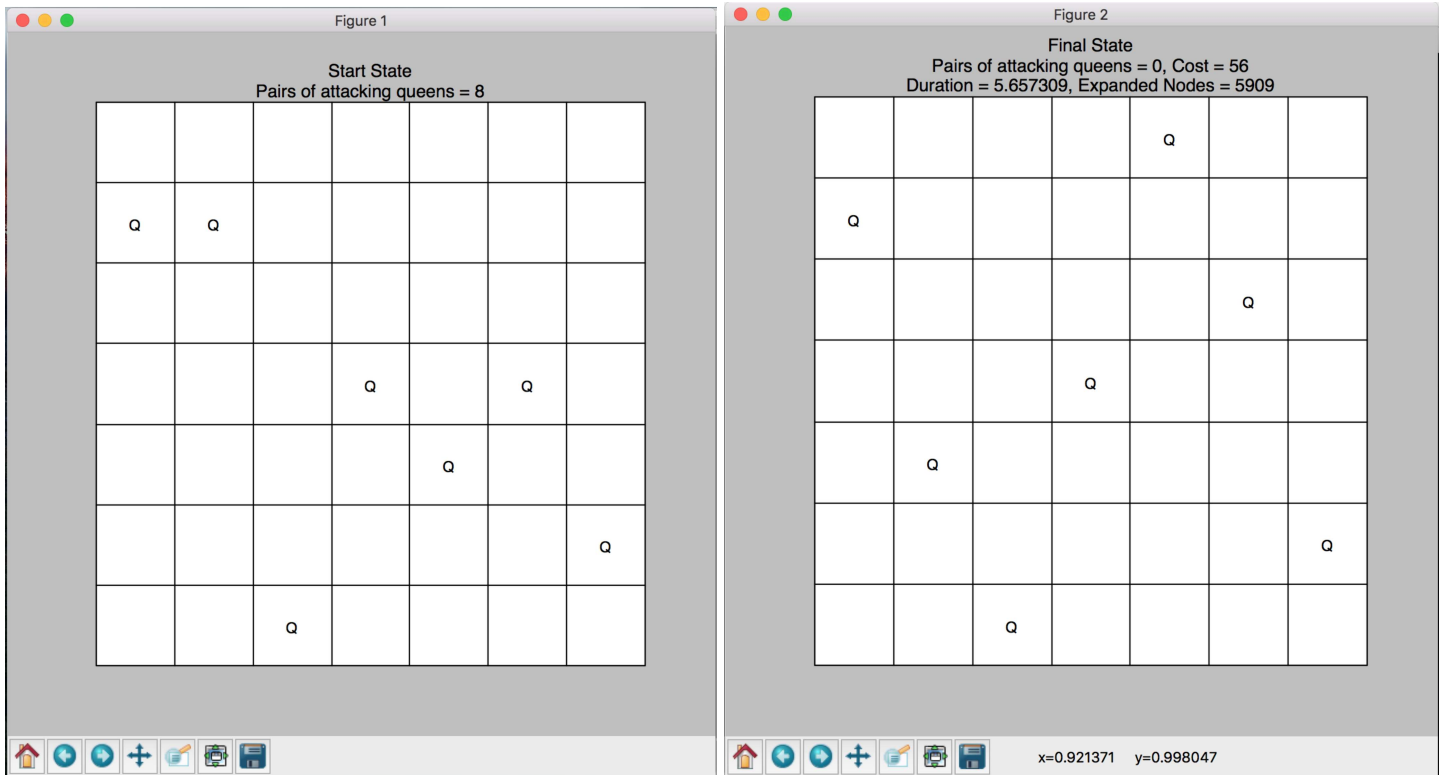
```
HW 1 — -bash — 80x48
~/Documents/Worcester Polytechnic Institute/CS 534/HW 1 — -bash
[Calvins-MBP:HW 1 calvinhe$ python nqueensastar.py 7
Command-line argument = 7
Solving N Queens Problem using A* and N = 7
Reached a solution in 5.65730881691 seconds
Reached a solution in 3 moves
Final cost: 56
Number of expanded nodes = 5909
Sequence of moves:
Board at state 0
[[ ' ' ' ' ' ' ' ' ' ' ]
 [ 'Q' 'Q' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ ' ' ' ' ' ' ' ' 'Q' ]
 [ ' ' ' ' 'Q' ' ' ' ' ' ]

Board at state 1
[[ ' ' ' ' ' ' ' ' ' ' ]
 [ 'Q' 'Q' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ ' ' ' ' ' ' ' ' 'Q' ]
 [ ' ' ' ' 'Q' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ' ' ]

Board at state 2
[[ ' ' ' ' ' ' ' ' ' ' ]
 [ 'Q' ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ ' ' 'Q' ' ' 'Q' ' ' ' ]
 [ ' ' ' ' 'Q' ' ' 'Q' ]
 [ ' ' ' ' ' ' ' ' 'Q' ]
 [ ' ' ' ' 'Q' ' ' ' ' ' ]

Board at state 3
[[ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ 'Q' ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' 'Q' ' ' ' ]
 [ ' ' ' ' 'Q' ' ' ' ' ' ]
 [ ' ' 'Q' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' 'Q' ]
 [ ' ' ' ' 'Q' ' ' ' ' ' ]

End of N queens A* search for N = 7
Calvins-MBP:HW 1 calvinhe$
```



Questions:

- Using an A* search approach, the largest puzzle that the nqueensastar.py script was able to solve was $N = 7$, in which after test runs of 10 trials, the average execution time was approximately 5.68 seconds. When running the program with $N = 8$, the execution can vary from 15 seconds to a couple of minutes, but a solution where there are no attacking queens can still be reached.
- For A* search, half of max $N = 7$ is rounded up to 4, but solving a 4 queens problem is trivial in terms of the execution time and number of solutions. Thus, I would like to test $N = 5$. Here is a table that summarizes the runs of ten trials using $N = 5$:

A* Search:

Trial Number of N = 5	Elapsed Time (seconds)	Number of Expanded Nodes	Depth (Length of Path)
1	0.051745	127	3
2	0.057030	129	2
3	0.053239	139	3
4	0.056773	148	2

5	0.053276	138	2
6	0.055435	104	3
7	0.045632	102	3
8	0.085933	221	3
9	0.040202	97	3
10	0.061123	141	3

The formal definition of the Effective Branching Factor is

$$N = b + (b)^2 + \dots + (b)^d$$

where N is the total number of nodes expanded, d is the depth where the solution was founded, and b is the effective branching factor. A close guess to the solution of calculating b would be looking at just $N \geq (b)^d$, which is equivalent to $b \leq N^{(1/d)}$. Using the data from the test trials, we have the average number of expanded nodes to be 134.6 and the average depth to be 2.7. So, $N^{(1/d)}$ would be $(134.6)^{(1/2.7)}$, which means that the effective branching factor is **b = 6.145**.

3. The A* Search approach would come up with the cheaper solution paths due to it being a more informed and more optimal algorithm than Greedy Hill Climbing. A* makes use of a priority queue to keep in memory all nodes and their successors that A* would traverse through. By using a cost and heuristic function as a priority, the queue would always pop out a successor that uses the least cost, but if this cost so far would be greater than another node's cost within the queue, then A* would backtrack to that node and consider its path to see if the end goal yields a less overall cost. In the case of Greedy Hill Climbing, any greedy algorithm is generally not complete nor optimal, but due to its nature of selecting the first node that it sees with the best heuristic (in the case of the N queens, it is the board space with the least number of pairs of attacking queens), Greedy Hill Climbing would reach a solution much faster, even if it is not the most optimal one.
4. Greedy Hill Climbing typically takes less time because, as a greedy algorithm, it would accept a non-optimal result in its fast performance. Since A* requires the expansion of nodes in a Breadth First Search manner while also computing its cost and heuristics as a priority to be saved in a priority queue, its time and space complexity are both exponential.

Problem 2:

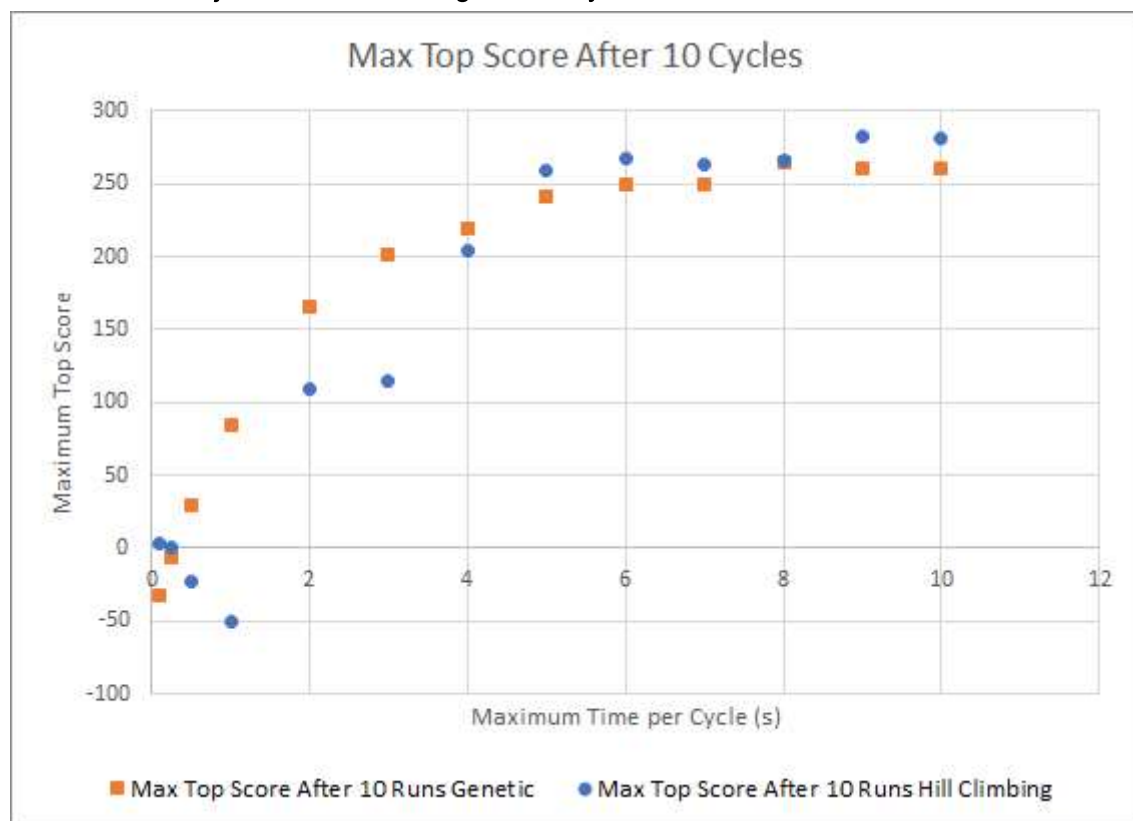
1. The genetic algorithm is centered around a helper class called GeneticChild, that contains the map candidate, the locations of buildings, and the time that the child was produced. The list containing the locations of buildings is the representation chosen to facilitate crossover. In the first generation, location lists are randomly chosen. In all other generations, the best 6 results from the last generation were found and saved. This is done by making a list of

tuples containing scores and the indices of the GeneticChild objects that have that score. This list is then sorted by score, and the top 6 results are compared to the objects that are currently saved. If the new object has a higher utility score than any of the objects previously saved, it is added to the list of saved objects. This is not strictly necessary for the algorithm to work, but it enables preservation of the time the utility score was first found.

Using the same sorted list of tuples, the worst 5 results from the last generation are deleted. This remaining population is used to draw parents from. The selection method is described in the response to question 4.

Crossover is implemented on the location lists of the two parents. For each location, there is a 6% chance that a random location will be selected for the child. This is done by drawing a number from a uniform probability from 0 to 1 and using values greater than 0.94 as an indication that a mutation occurred. If this did not happen, a number is once again drawn from a uniform probability from 0 to 1 and using values greater than 0.5 as an indication that the child should take the location from parent 2, and using values less than 0.5 as an indication that the child should take the location from parent 1. This is repeated for each location to be populated.

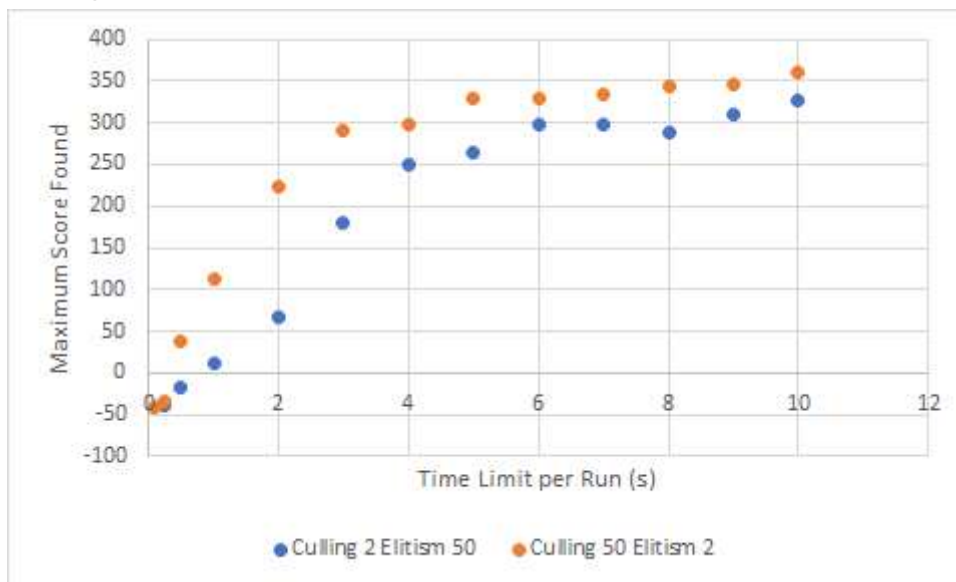
Once the new generation is completely created, the last generation is deleted, and the current generation is passed back in as the parent generation. Once time runs out, the GeneticChild object that had the highest utility value is returned.



- 2.
3. Elitism forces the genetic algorithm to approach the maximum by having the algorithm keep the top 'x' number of children. Doing this prevents children that may be close to the correct answer from being randomly removed. This also makes the top score a monotonically

increasing function over time. The smaller the 'x' the more likely the algorithm is to approach towards the closest maximum, be it a local or the global maximum. Increasing the number of elite children kept forces the algorithm to approach a maximum more slowly, but it does have better chance of finding the global maximum. This result makes intuitive sense, as saving a larger set of high-scoring results is more likely to contain a result that has some portion of the global maximum, but children that approach a local maximum are more likely to continue to be saved, increasing time that these are explored.

Culling determines the number of weakest children to be discarded and not be used in creating future generations. This has the opposite effect of elitism. As culling increases the algorithm approaches quicker towards its closest maximum, which could be local and not global. A large number of culled children can be thought of as having the genetic algorithm operate in a more greedy manner, prioritizing exploiting the currently high-scoring maps over exploring the lower-scoring options. If the goal of the algorithm is to find any maximum quickly then it is best to decrease the number of the best solutions that are kept and increase the number of worst solutions that are discarded. If the goal of the algorithm is to find the global maximum then it is best to keep more of the elite as well as low scoring children. The graph below shows how our algorithm performance at two extremes with culling and elitism set to 50 and 2 respectively the algorithm reaches the maximum faster than culling and elitism set to 2 and 50.



4. The utility function provided by the assignment is capable of providing negative responses. Because of this, the method shown in class is impractical to use. Instead, a tournament-based selection method was used. To select each parent, a subpopulation of the selection pool is randomly chosen. From these, the option with the highest utility value is chosen. In the current configuration, the subpopulation has a size of five. This is the simplest implementation of a tournament selection method. A modification that could be added would be to make the selection of the highest utility value dependent on some probability p , introducing the ability to select the second or third best result instead. This added

probabilistic feature could potentially improve performance.