Jack Charbonneau
Jannik Haas
Jian Liu
Mario Arduz
Joan Wong

Project 1 Write Up - Group 1

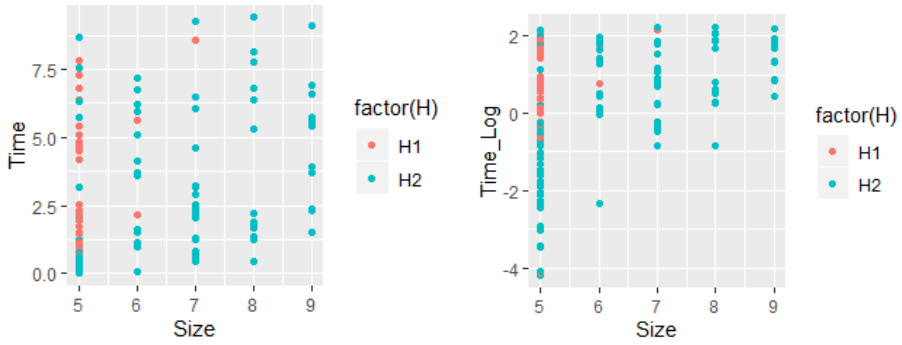
Part 1. Heavy N-queens

Question #1

How large of a puzzle can your program typically solve within 10 seconds using A*? Using greedy hill climbing?

To determine how large of a puzzle the program can typically solve within 10 seconds, we have tested 25 random board configurations, from 5 to 10 dimensions, queens that have a random integer weight in the range of [1,9], with both heuristic functions, using A* and greedy hill climbing.

The data points with successful completion of the given board configuration are shown below, and they indicate the board size and time that it was required to solve the problem.

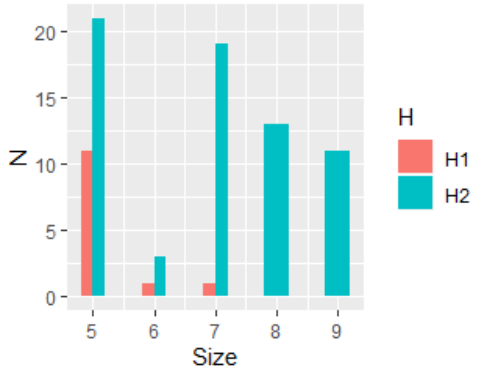


The previous two plots have the intention to show the exponential time consumption of the search based on the dimension of the size of the board. This relation is more clear at the right graph, where the y axis corresponds to the natural logarithm of the execution time.

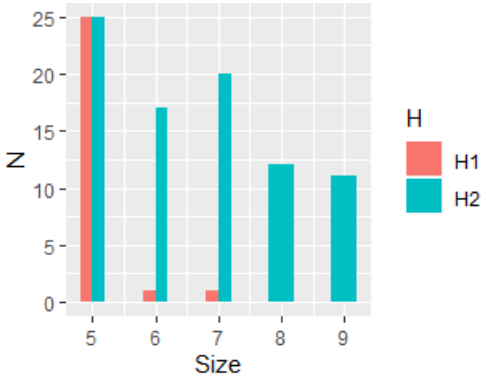
The following graphs show the number of successful attempts given the size of the board and an approach, from the sample of 25 boards. The main conclusion for both A* and greedy hill

climbing is that H1 is not able to perform in board sizes that are bigger than 5, and that the maximum size that H2 can solve is 7.

A*



Greedy Hill Climbing



With respect to the greedy algorithms, it has to be stated that there were allowed as many restarts as needed, as well as many sideways moves, but the length of the solution is at most 1.25 times the dimension of the dimension of the board. Even though aming for short solutions reduces the percentage of successful sets of moves to solve the initial board, the cost function under this heavy queens problem is highly sensitive to the number of moves and is a decisive argument to opt for a short solution.

This approach might be influencing the performance of H1 more than H2, because H1 is not sensitive to the number of queens that attack each other, but rather is focused on the minimum weight of one of the queens in conflict. H1 has the problem of not being a good estimation of the distance to the solution because many board alternatives can be regarded as equivalent, whereas they are different in the minimum number of steps to find the goal. H1 would need a bigger number of sideways moves to arrive at a result, which under this execution is not freely allowed since there is a maximum number of steps for each call of the greedy function. Nevertheless,

allowing for more steps does not solve the problem of not differentiating alternatives. Given the set of parameters and restrictions, this is an example where a non admissible heuristic (as it will be shown in point 5) is prefered to an admissible one.

Question #2

What is the effective branching factor? For this computation, perform 10 runs of a puzzle half the maximum size you calculated for step #1.

The following calculations are carried in board sizes that have 5 dimensions, considering that neither of both approaches were able to solve more than 50% of the trials at a 9 dimension board.

a. Compute the branching factor using H1 for A* and greedy search
   Effective branching factor for H1 with A*: 1226.6
   Effective branching factor for H1 with Greedy search: 38.4
b. Compute the branching factor using H2 for A* and greedy search
   Effective branching factor for H2 with A*: 638.3
   Effective branching factor for H2 with Greedy Search: 1.8.

The effective branching factor of A* is much bigger than that of Greedy search. Since A* considers all possible nodes into consideration and Greedy search only expands the nodes which have the lowest heuristic value, there is a remarkable difference in the number of nodes that were expanded.

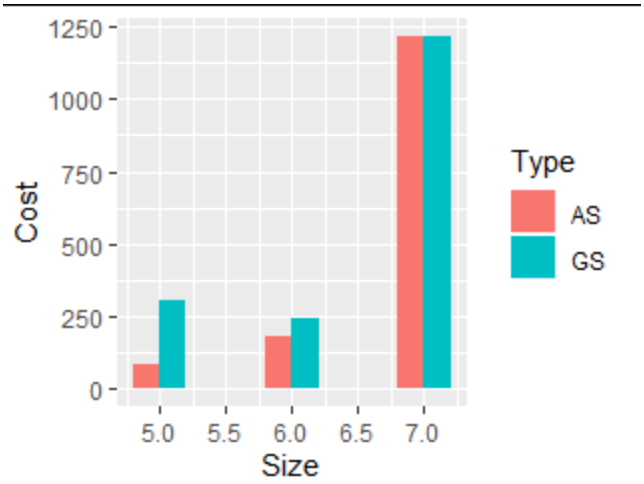
As it was addressed in the previous question, there is a noticeable difference in the performance of the algorithm based on the election of the heuristic function. H1 has a higher branching factor because there are several boards with the same value of the heuristic function, whereas in H2, the search can be directed in a fewer number of nodes.

Question #3

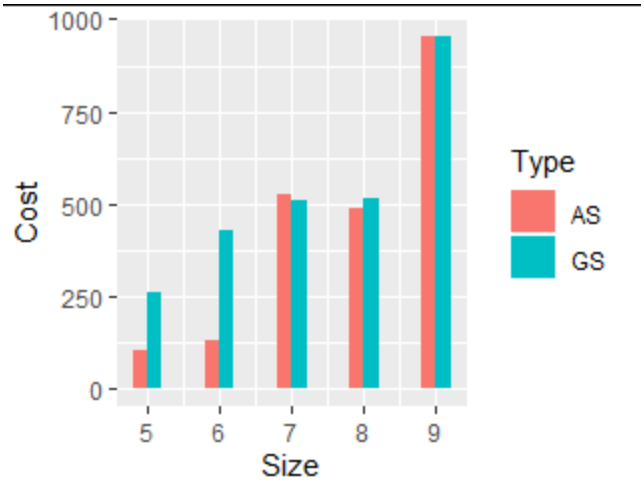
Which approach comes up with cheaper solution paths? Why?

It is expected that the algorithm with the cheapest solution would be A*, but as it will be shown in the next graphs, under large board configurations there does not seem to be a stark difference between both approaches. Nevertheless, one also has to remember that the number of successful executions of the algorithms is lower.

Cost of Solution Maintaining H1



Cost of Solution Maintaining H2



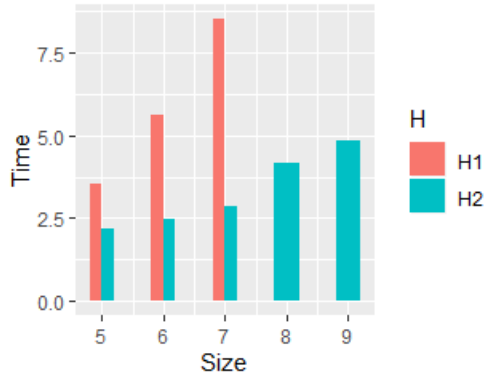
From the graphs above, we can see that given a small size of the board, A* search comes up with cheaper solution paths. But when the size of board gets bigger, the cost of A* search and Greedy search seems similar. When the board is big, it is more likely to be far from the solution, so both A* and Greedy search will have a big cost. The advantage of A* will become smaller given the time constraints that the problem is enforcing.

Question #4

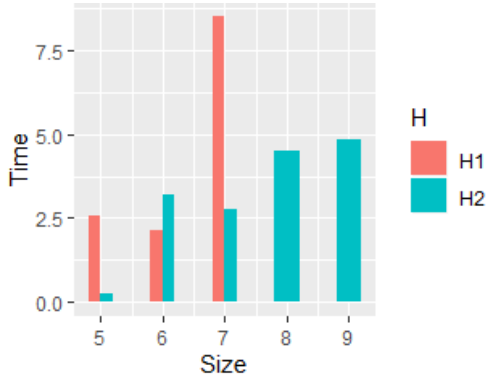
Which approach typically takes less time? Why?

Greedy search typically takes less time to arrive at a solution because it does not expand all the possible nodes of a given board configuration. However, similarly to the comparison of the costs, this difference has less evidence in boards of a large size.

A*



Greedy Hill Climbing



The biggest differences in time are shown at a board of size 5, utilizing H2 as an heuristic function, and at a board of size 6 with H1 as the heuristic function, whereas the rest of the comparisons with larger boards does not show a significant difference. However, it also has to be noted once again that the number of successful executions of the search programs decreases with larger board sizes.

Question #5

Either prove the heuristic H2 for A* is admissible, or provide a counterexample demonstrating that it is not.

To demonstrate that H2 is not admissible, consider the following board configuration in a 4 by 4 table:

-	-	-	-
9	-	9	-
-	-	-	9
-	9	-	-

The heuristic cost under this example is 162, because there are two pairs attacking each other, and all the queens in the board have a weight equal to 9. Nevertheless, the solution of the game only requires one move to arrive at a board configuration where no queens attack each other. This move has a true cost of 81, which is lower than the value of the heuristic function and thus, proves that H2 is not admissible.

Part 2. Urban Planning

Question #1

Explain how you traded off the number of restarts and the schedule for decreasing temperature. We are not expecting a dissertation-level analysis, but you should provide a couple of experiments explaining how you selected your parameters. Running a few trials works much (!) better than just making the numbers up.

We decided that since we would be running the algorithm for a set time limit, we would let the simulated annealing algorithm restart as many times as it could during this set time. This means that for larger boards, the number of restarts will decrease since each restart will take longer. Now we only had to decide on the schedule for cooling the temperature. When trying the logarithmic function we were not able to obtain good results, but with the geometric schedule we were able to obtain the maximum score close to 100% of the time. We decided to start with a temperature of 10 and cooling it each iteration by multiplying the current temperature by a constant c and ending when the temperature reached 1. We started with a constant of 0.9 and saw that for both test maps, each restart took less than 0.01 seconds, allowing for a lot of restarts in the 10 second time frame, however there was not enough exploration so it only reached the maximum score 40% of the time. Increasing the constant to 0.99 increased the time for each restart to about 0.03 for the urban2 map, and slightly less for the urban1 map, again allowing for plenty of restarts, while also increasing exploration which increased the success rate to 95%. However, we saw some anomalies where even though it had plenty of restarts, it still got stuck at some local maximum. Therefore we increased the constant to 0.999. Now each restarts takes about 0.25 seconds for the urban2 map, allowing for about 40 restarts, and even more for the urban1 map. This has been shown to be a great balance between exploration and exploitation and got us 100% success rate when running the algorithm 1000 times even when not allowing any

restarts. We believe the geometric constant of 0.999 provides the best balance of exploration and exploitation while allowing for enough restarts in case the algorithm does get stuck at a local maximum. For much larger boards we may have to reconsider this constant as each restart will take much longer, but we believe even without any restarts, the algorithm will provide a good, if not optimal solution due to the balance of exploration and exploitation.

Question #2

Explain how your genetic algorithm works. You must describe your selection, crossover, elitism, culling, and mutation approaches. Give some data to explain why you selected the values that you did. How do elitism and culling affect performance?

The general idea of genetic algorithm is combining elements from two solutions, or parents, together in the hope to get a better solution, mixing algorithmic concepts of exploration (fitness, mutation) and exploitation (fitness, elitism, culling). The algorithm begins with a population of k randomly generated states, which in this case are terrain/map configurations resulted from random assignments of site locations. From the population, we *select* two states as parents semi-randomly, *combine* (crossover) them to generate a child terrain state, and *mutate* the children terrain state with some probability, p, until the population is full. For every iteration of the algorithm, we applied *elitism* and *culling* to keep solutions that were close to the best solution and remove weakest solutions that were farther away from the best solution correspondingly. We will describe our selection, crossover, elitism, culling, and mutation approaches to our implementation in more detail below.

<u>Selection</u>

For this urban planning problem, our team decided to use the terrain scores as their respective weights when randomly choosing two parent states. In other words, higher scores will more likely be chosen and vice versa. Because of this, there is a chance that two identical parent terrains will be selected by our algorithm, resulting in an identical child terrain. However, we believe that by introducing mutation in crossover, we will mitigate this issue of having multiple (or too many) identical terrains in the population. Note: Fitness will be explained in (4).

<u>Crossover</u>

Crossover is the process of combining the two parent states, p_1 and p_2, selected as discussed above. The operation of crossover takes into consideration two types of information from both parent terrains: the number of each site and the locations of each site. First, we randomly choose the number of *each* site from either parents,

$n_I = n_I\ (p_1)$ or $n_I\ (p_2)$,

$n_C = n_C\ (p_1)$ or $n_C\ (p_2)$, and

$n_R = n_R\ (p_1)$ or $n_R\ (p_2)$.

Therefore, for every site type, J, $n_J \leq \max(n_J\ (p_1), n_J\ (p_2))$.

After determining the number of each site, the next step is determining the positions of them on the child terrain. This is done by randomly choosing n_J locations of site, J, from p_1 AND p_2. To note, it is possible that the algorithm chooses the same location (or an occupied location), such that the location will be replaced with another site.

Elitism

To prevent the drawback of combining states in losing a "near" solution through random combination, our team implemented genetic algorithm Version Three discussed in class, where we preserve the k * *elite_factor*, where *elite_factor* < 1, most fit states. The best parents will go straight to our next generation (or iteration).

Culling

Our team also applied the concept of culling from genetic algorithm Version Three, which we removed k * *culling_factor*, where *culling_factor* < 1, weakest states from the population. The worst parents will be removed before crossover is performed (and thus, will not be inherited by the child).

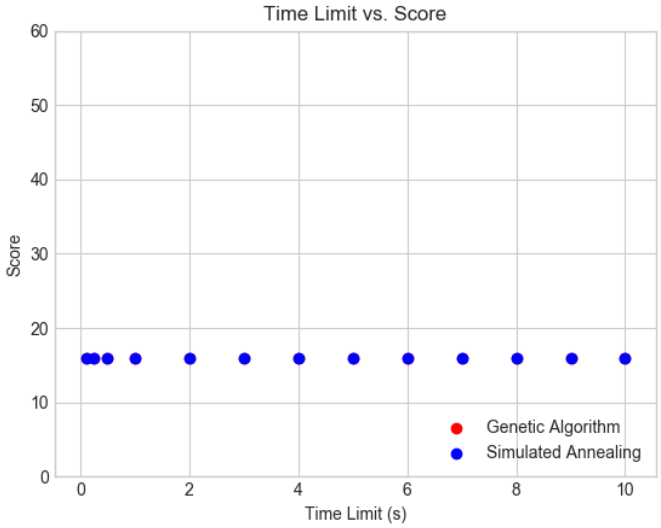
Mutation

Our team used mutation to add more randomness to our genetic algorithm. After performing crossover, a child terrain state is mutated with a probability, p. We have implemented three different types of mutation: site relocation, site removal, and site building. The type of mutation performed is randomly chosen. In site relocation, the location of an existing site inherited in child terrain is mutated or relocated in any random direction by a distance of 0 or 1. In site removal, an existing site on the child terrain is randomly chosen to be removed. In site building, a new, random site (that is required and is not present or is low in number) will be added to a random, available (non-occupied) location on the child terrain. In the case where a mutation fails to happen when expected, such as when there are no more sites left to add, the algorithm is forced to randomly choose another mutation to perform.

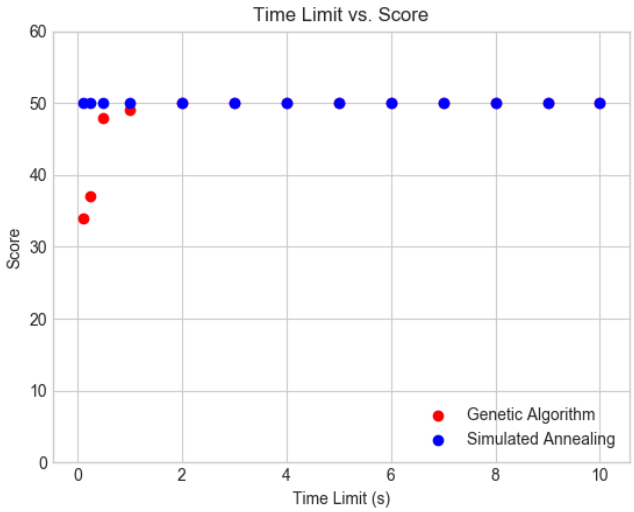
Question #3

Create a graph of program performance vs. time. Run your program 10 times and plot how well hill climbing and genetic algorithms perform after 0.1, 0.25, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 seconds. If you only had 0.25 seconds to make a decision, which technique would you use? Does your answer change if you have 10 seconds?

Urban 1:



Urban 2:



These plots display the score returned and the time we let the algorithm run for both the genetic algorithm (GA) as well as simulated annealing with restarts (SA). We can see that given only a very short time limit, the SA algorithm outperforms the GA. This is due to the fact that SA only generates one random start state and immediately starts searching, whereas the GA generates a large pool of states which takes longer before it can start actually searching for a solution. Given more time, both algorithms consistently reached the max state for both test maps, therefore if we

only had less than 2 seconds to reach an answer we would choose SA, and if we had 10 seconds to reach an answer, either algorithm will reach the maximum score.

Question #4

How did you perform selection and crossover for your population? Find some known method for doing selection, other than the one described in the slides. Explain what you did.

Our team performed selection and crossover differently than described in the slides to make it more applicable and suitable for our urban planning problem. We will give further insight to our modifications of each procedure below.

Selection

In the N-queen problem we did in class, the fitness score is calculated based on known "limits" (the best possible maximum score and the worst possible state) of the board, allowing us to take the difference and compare how close the score is to 0. However, for the urban planning problem, we cannot do this. It is not possible for us to determine the maximum score of any given *new* map beforehand. Therefore, in our case, we favor the parent terrains with higher scores instead. Yet, we cannot *only* select parent terrains with better scores because this might introduce the problem of having too much elitism and not enough exploration in our algorithm. Thus, our definition of fitness is in the form of weights (how likely a parent terrain is to be chosen), which are based on terrain scores. That way we are breeding "good" parent terrains but also not neglecting other possibilities that "bad" parent terrains can produce. This procedure is explained in more detail in (2).

Crossover

This procedure is explained in more detail in (2). Our team performed crossover differently than described in the slides. Firstly, the crossover process produces one child terrain state as opposed to two. Secondly, the features inheritance from parent terrains include the number and the locations of each site type. In the N-queen example we did in class, crossover can easily be performed by splitting the board in half and switching the two halves together to produce two children. However, we cannot do that in our case. Since our crossover procedure can randomly produce many children, we have decided to only generate one child from two parents, that way we can explore more parent combinations.