2. Genetic algorithm

(1) Implement algorithm

* Data structure

In this part, we defined a class called Map to store the information of the inputted terrain map. We also defined a class called Population to store the information of each different state. In Population, there’s a dictionary called zones, and it will store the coordinates of locations of each zone. The Population has methods such as add one zone, remove one zone, and the most important, calculate the score of this specific urban map. That means, we are not going to change the raw terrain map for each state, we will only refer the information from the terrain map.

Another important class is priority queue. With this queue, we are able to get the best states or worst states quite fast in order to implement elitism and culling.

* Fitness

In this part, the fitness function is easy to define. We calculate the bonus, penalty value and the cost for every map as heuristic, also, fitness.

* Selection

In order to make the algorithm find better states faster, we can not only use common random select method. The way we implement selection function is, one of the parents will be chosen from the best k1 states, i.e., from the elitism queue. Thus, all of the individuals in present population (not including the best k1) will produce two children with one of the best k1 individuals.

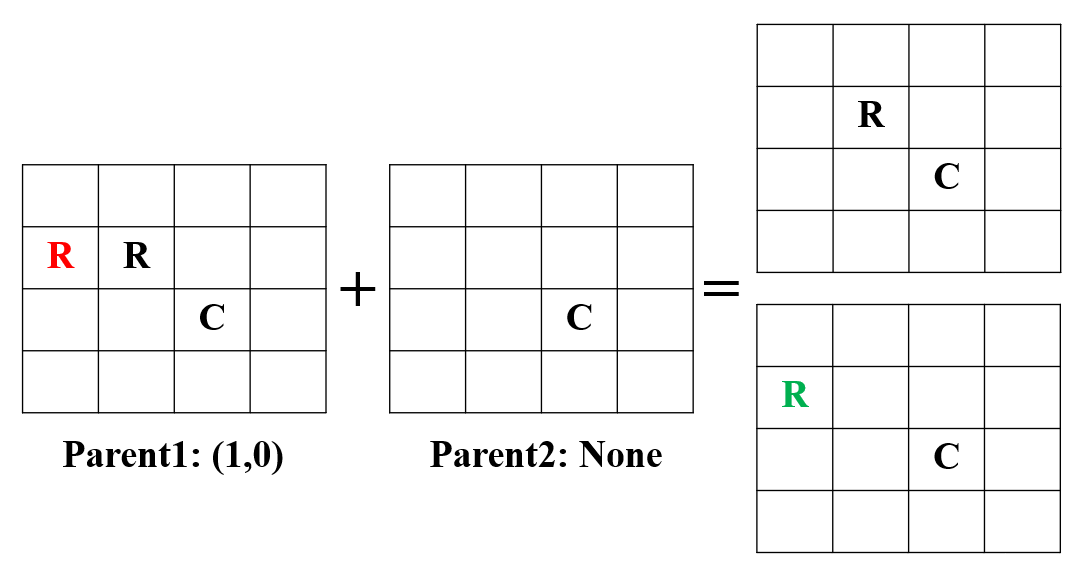
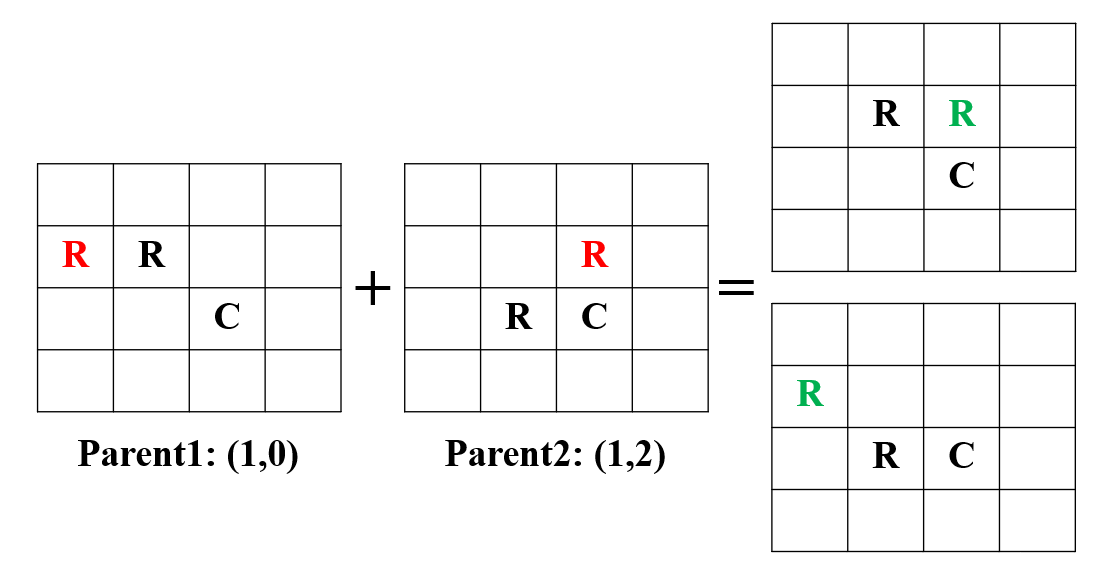
* Crossover

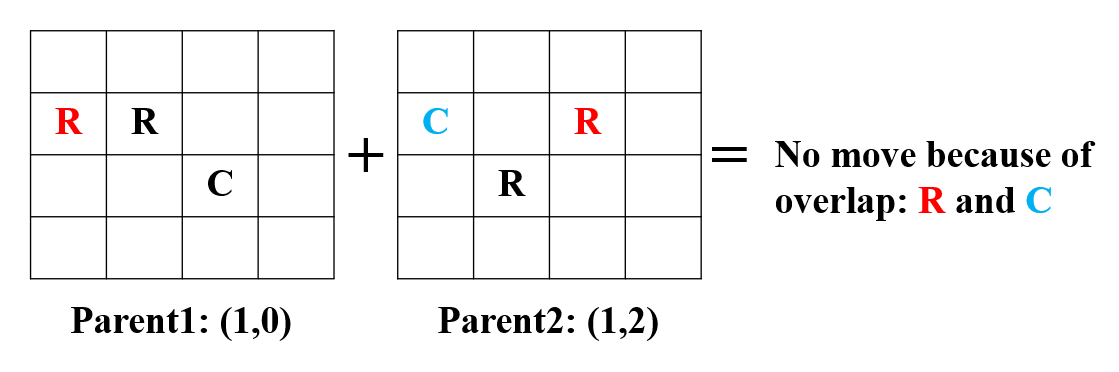
There are many different methods for crossover, but we already saw in class that some methods may cause the offspring not acceptable (for example, some of the numbers appear twice in the TSP). The checking and revising part of such offspring can be time-consuming. Thus, we planned to use single-point crossover method.

The idea of single-point crossover in our work is, each time after we choose two “parents”, we randomly select one kind of zones, such as residential zone. Then, we are only going to swap the locations in this kind of zone, unless both of the parents have the same arrangement. That is to say, we will not allow the change from one kind of a zone to another. The actions we allow only contains: to swap zones of same kind between parents, and to randomly add one zone if it’s acceptable (actually it is included in mutation function). This solution intuitively avoids the unacceptable cases stated above.

Of course, there can be cases that the position a zone is moving to is already occupied by another zone. Then no move will occur, we will go directly to mutation process. This doesn’t mean the method is not complete, because the mutation process also has the chance to change both of the children.

Here are some examples for single-point crossover in our work.





* Elitism

Elitism is to keep the top k1 of the population during the entire process of genetic algorithm. This technique keeps the performance of genetic algorithm monotonic. We keep the top k1 states in another priority queue, thus the best state is first one in that queue. After every iteration of genetic algorithm, the top k1 states in the new generated offspring are extracted, and merged into elitism queue. Then, only the best k1 states in this new queue are kept, becoming the new elitism queue. As stated above, this elitism queue is used in selection process every iteration.

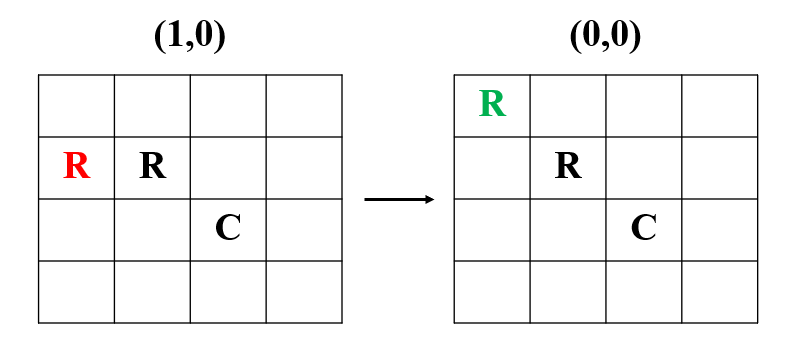
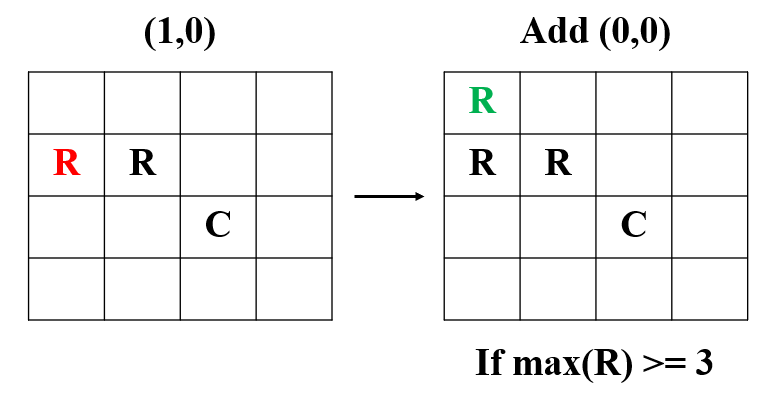
* Culling

Culling is to drop the lowest k2 of the population. This technique will eliminate states with low scores in order to keep the quality of population. It can be easily implemented by popping the lowest k2 states from the priority queue.

* Mutation

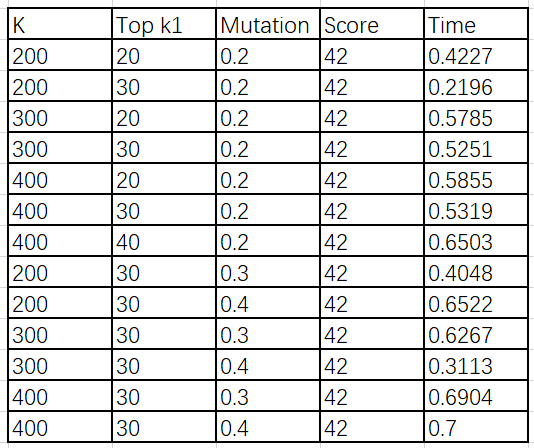
Mutation is to randomly change some bits in an individual, which brings a chance that a better individual will occur (to get out of local optimum). The probability to mutate is an important parameter in this function, it is used at the beginning of this function. A number is generated randomly, and if it’s greater than the probability of mutation, the function will directly return and no mutation will occur. However, if it’s lower, that means we are going to accept mutation, we will select one kind of zone to mutate. The possible outcomes of a mutation are, to move a zone from one position to another, or to generate a new zone.

Here are also some examples for mutation process.

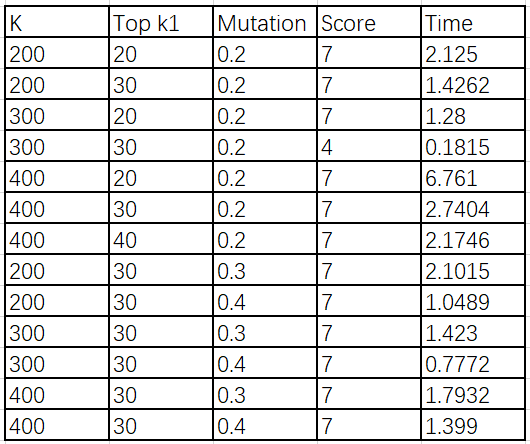
 

1. Parameter selection

6\*5 problem, max capacity = 5 zones



7\*7 problem, max capacity = 9 zones



From the statistics, most of the cases we get optimal arrangement of the terrain. As the size of the entire population (K) increases, the average time for finding the best answer also increases. Intuitively if we want to find the answer faster, we should choose a smaller K value. But this is not always the case. We can see when we increase the size of the problem, there may have chances that the algorithm won’t find correct answer, which is absolutely what we don’t expect. Thus, with a lower K, we may get stuck in a local optimum, and will not find a correct answer. Therefore, we are not going to using K = 200.

For the mutation probability, increasing in probability doesn’t always means increasing in searching time. We can easily observe that in some cases, the algorithm finds the answer faster when using a higher mutation probability.

Above all, we can see with K = 300, k1 = 30, mutation probability = 0.4, we get an overall best performance. Thus, we are going to use this set of parameters in experiment.

Culling is a little bit tricky. We implemented culling part, but it doesn’t accept an input parameter k2. We are trying to keep the population in a consistent size K. That means no matter how many offspring are generated (let’s say, N), only top K children are kept, the rest N – K individuals are all culled out. Thus, no parameter is needed here, the culling part only depends on population size K. This is a good way to keep time complexity consistent.

1. Elitism and culling

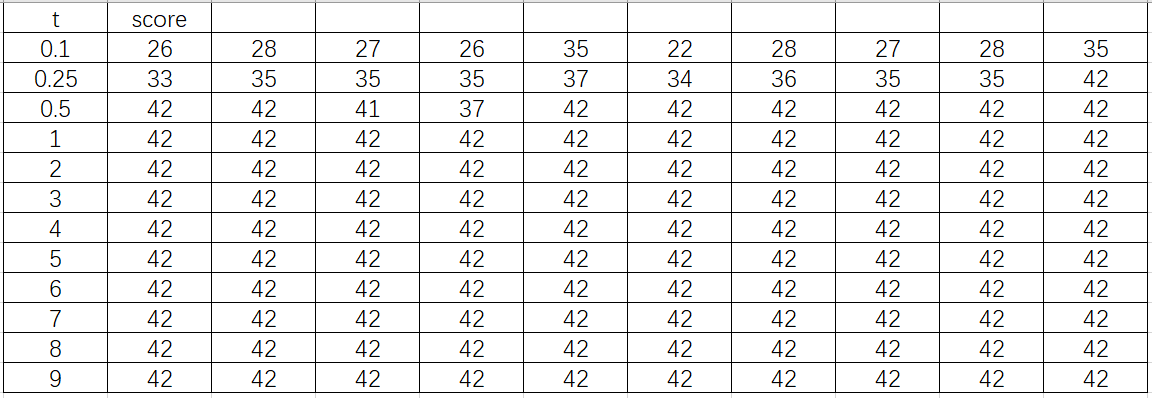
We are going to use a 7\*7 terrain to see how these techniques affect performance.

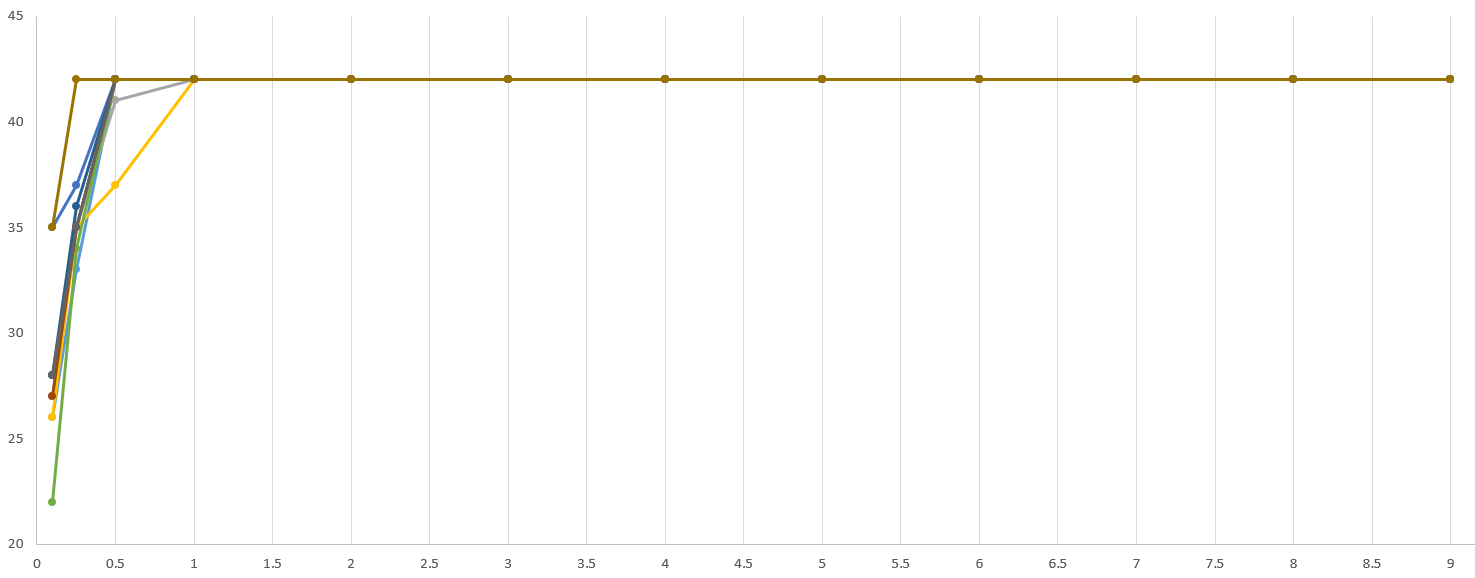
Elitism: set k1 = 1, let’s see what’s going on. (Has to be greater than 0, or the core of our algorithm will not work). Other parameters are all as before. We test the algorithm, and it still finds a correct answer, but will spend 2.041 seconds, slower than previous tests.

Culling: modify the code, and set the culling number k2 = 20. Other parameters are all as before. This time, it even doesn’t find the best answer in 10 seconds.

From these results, we can conclude that elitism surely will keep the performance monotonic, and will find the correct answer faster. And culling will greatly improve the speed.

3.Performance over time of GA





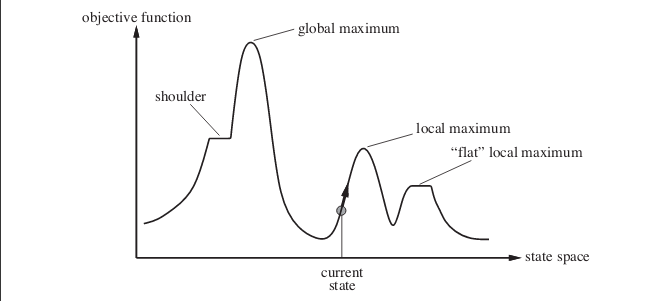
4. I think this part is already answered in question 2

**Simulated Annealing**

**Number of Restarts Trade-off**

**Background**

Random restarts are an important feature in simulated annealing because they allow the program to explore the problem space from various initial configurations. In order to understand this, consider the simple two-dimensional hill-climbing graph shown in the figure below.



Hill Climbing Example

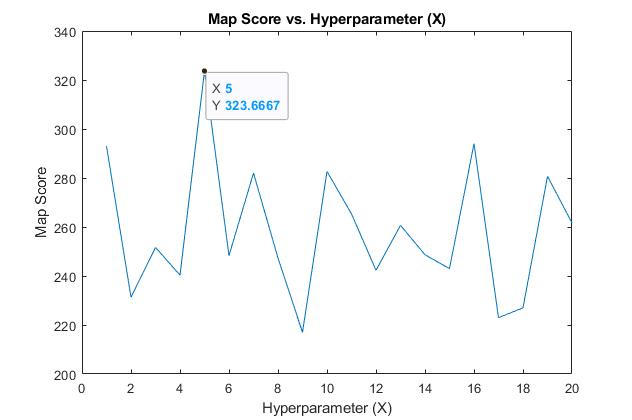
In such an example, Simulated Annealing would allow actions, such as traversing valleys between local maximums, which increases the probability of reaching the global maximum. However, suppose that using some initial configuration the program would have to allow several “valley” traversals in order to reach the global maximum. If this is the case, then it is unlikely that the global maximum will be reached. Overall, we can make the general statement that the probability of reaching the global maximum decreases with the distance between the initial configuration and the optimal solution.

This is where random restarts come into play. A random restart allows the program to explore the problem-space, beginning at some initial configuration, until it reaches a local maximum, at which point it will restart from some new random configuration. As a result, as the number of random restarts increases the probability of one of the configurations, leading to an optimal solution, increases.

**Implementation**

When deciding the number of restarts, we must consider the cooling schedule. As explained in further detail later, slower cooling schedules are more likely to lead to optimal solutions. Additionally, they decrease the number of restarts that can be performed within a given amount of time. We must take this into consideration when determining how many restarts to perform. In doing so, I use the size of the problem and then number of steps taken without taking progressing, to determine when and how often to restart. I use the general assumption that the number of restarts should be dependent upon the size of the urban map, to develop a simple equation to determine the number of iterations, without progress, to perform before restarting. This equation can be seen below.

In order to estimate the hyperparameter (x), I ran the program given various values of x and a geometric cooling schedule (which will be discussed later), on a 10 x 10 map. Then, I observed the data and selected the value which resulted in the best average score given. Below the plot of the algorithm performance with various values for the hyperparameter can be seen.



10 x 10 map score vs. Hyperparameter

The value of 5 lead to the best performance for the experiment, so I hard coded this value into the program. The program performs well on maps of various sizes, which makes me believe that this method generalizes well given different map boundaries. Although this is not full proof, overall it leads to near optimal results in a reasonable amount of time.

**Cooling Schedule Tradeoff**

**Background**

Simulated Annealing will always move to some next state if the state results in a greater score than the current state. On the other hand, if the next state results in a worse score, then the program will move to it with some probability. As previously mentioned, this allows the program to escape local maximums by allowing “valley” traversals. In Simulated Annealing, the temperature is used to determine the probability that a worse move will be taken. When the temperature is great then it is likely that a worse move will be taken; however, as the temperature decreases, this likelihood also decreases. Due to the characteristics of Simulated Annealing, a slow temperature decrease increases the chance of attaining an optimal solution. However, if we have a slow temperature decrease then we will spend more time exploring a given configuration; and thus, decrease the number of restarts to perform.

**Implementation**

In order to determine the best cooling schedule to implement, I ran the algorithm several times using various cooling schedules, given a 6 x 5 and 10 x 10 map. The results from the simulations can be seen below.



Map scores for 6 x 5 map



Map Scores for 10 x 10 map

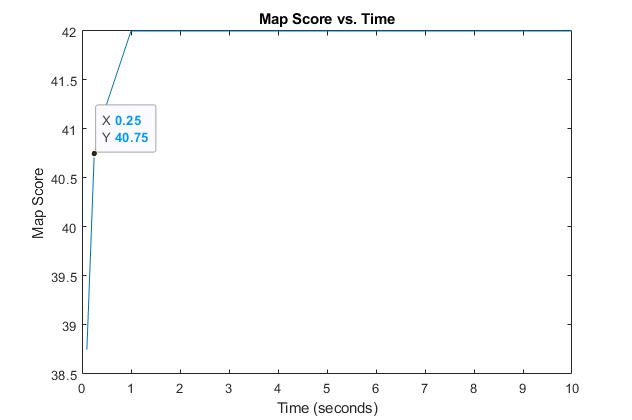
The geometric schedule had the best performance given both experiments; so, I implemented it in the program. By running the program on different size maps, we are ensured that the hyperparameter generalizes well for different maps.

**Program Performance**

The performance of Simulated annealing with restarts, over time can be seen in the figures below.



Simulated Annealing Scores for 6 x 5 map



Simulated Annealing Scores for 6 x 5 map

Within 0.1 seconds the program can obtain an average score of 38 / 42. Additionally, the program can consistently find an optimal score of 42 / 42. This makes me confident that I selected parameters which work well on various sized maps.

**Simulated Annealing vs. Genetic Algorithms Performance**