Habib University CS351:

Artificial Intelligence

Fall 2023 - Homework 02

Muhammad Najeeb Jilani, mj06879

Sajeel Nadeem Alam, sa06840

Q.1.a. [5 points] A* does not guarantee optimal results if the underlying heuristic is not admissible. What is an admissible heuristic? Why do we need it in A* to ensure an optimal solution?

An admissible heuristic in the context of the A* search algorithm is a heuristic function that never overestimates the cost to reach the goal from any given state. In other words, it provides a lower-bound estimate of the remaining cost to reach the goal. Mathematically, for any state s, an admissible heuristic $h(s)$ satisfies the condition:

$$h(s) \leq \text{actual cost from } s \text{ to the goal}$$

A* combines the actual cost to reach a state $(g - cost)$ with the heuristic estimate $(h - cost)$ to prioritize expanding states with the lowest f-cost $(f = g + h)$

When the heuristic is admissible, A* is guaranteed to find the optimal solution, meaning it will find the least-cost path to the goal. This guarantee is because the f-cost of any path is an underestimation of the true cost to reach the goal, ensuring that A* explores the most promising paths first. As a result, A* can prune away suboptimal paths early in the search, leading to efficiency in finding the optimal solution.

Q.1.b. [5 points] What different factors control exploration and exploitation in simulated annealing and how? Write your answer in three to four sentences. Cite references if you use any other external material besides the AIMA book and lecture notes.

There are two factors that control the exploration exploitation:

- Temperature (T)
- Absolute Difference in current best and explored value (delta E)

If the state is very bad means delta value is high hence probability of selecting it is low. If the temperature value is low, we become more selective hence the probability of selecting the bad state is low. When the temperature is high the probability of accepting a bad state is higher; the reason being that we want to explore other regions of the search space. But we can't stay like this because we need to arrive at an optimum which is why we reduce the temperature to increase the selection pressure (exploitation) and reduce the probability of selecting a bad candidate. What matters the most is how we control the temperature.

Q.2. [20 points] [Source: TBA] Refer to the pseudocode (and the flowchart) for Simulated Annealing given in the AIMA book and on lecture slides. Given a function $f(x, y) = x^2 + y$, minimize the function using Simulated Annealing. Assume the starting values for (x, y) to be $(-1, 14)$, where each trial (neighbor) is generated by adding 0.5 to both x and y respectively.

Run three iterations showing the values of SA to minimize the given function. Let the starting temperature T_0 be 1 and the temperature is reduced after every two iterations as $T_n = 0.8\, T_n - 1$.

Use the following random numbers (see the snippet of the pseudocode) from left to right, wherever required.

0.593054	0.29857	0.504246	0.376256	0.942043	0.496899	0.093536	0.093536

```
m := exp ( delta / temperature )
p := random number in range [0...1]
if p < m then
        current = trial
end if
```

Solution:

We applied Simulated annealing algorithm on the function $f(x, y)$ in according to the instructions mentioned in the question. We run the program for 3 iterations and in each iteration, there are two attempts as the temperature gets updated after every 2 attempts. So basically got 6 iterations.

Here is the table of findings:

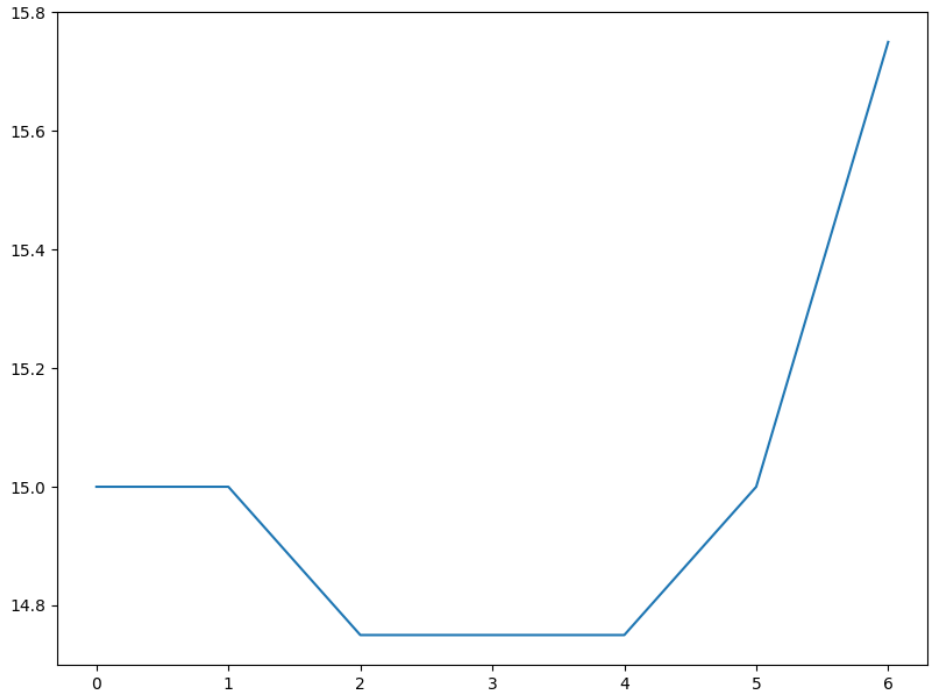
Temperature	Best solution (X, Y)	Current solution (X, Y)	Best fitness	$F(x, y) = x^2 + y$ (Current fitness)	E (delta)	M	P
1	(-1, 14)	(-1,14)	15	15	0		
1	(-1, 14)	(-0.5, 14.5)	15	14.75 (best solution)	0.25		
-0.2	(-0.5, 14.5)	(0, 15)	14.75	15 (worse) --> didn't accepted	0.25	0.29	0.59
-0.2	(-0.5, 14.5)	(0, 15)	14.75	15 (worse) --> didn't accept	0.25	0.29	0.3
-1.16	(-0.5, 14.5)	(0, 15)	14.75	15 (worse) --> accepted	0.25	0.81	0.5

-1.16	(0, 15)	(0.5, 15.5)	15	15.75 (worse) ---> accepted	0.75	0.52	0.38

Simulated Annealing values:

1. 14.75 --> 15
2. 15 --> 15.75

Best fitness: 15.75



In the above attempt, for p we were given a list of values to select. But I attempt using $p = random.random()$ function and here are the results.

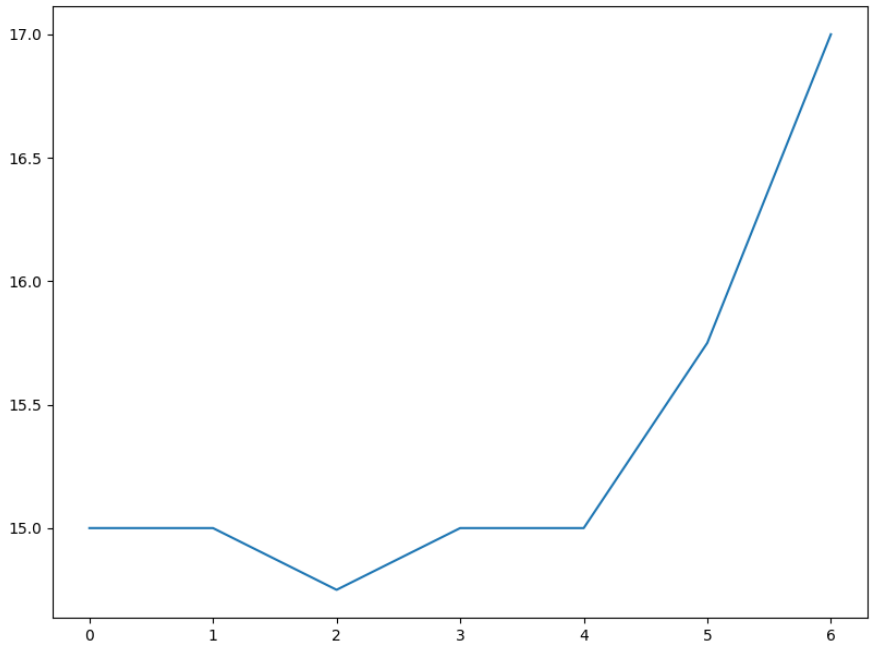
Temperature	Best solution (X, Y)	Current solution (X, Y)	Best fitness	$F(x, y) = x^2 + y$ (Current fitness)	E	M (probability)	P (probability)
1	(-1, 14)	(-1,14)	15	15	0		
1	(-1, 14)	(-0.5, 14.5)	15	14.75 (best solution)	0.25		
-0.2	(-0.5, 14.5)	(0, 15)	14.75	15 (worse) --> accepted	0.25	0.29	0.02

-0.2	(0, 15)	(0.5, 15.5)	15	15.75 (worse) --> didn't accept	0.75	0.02	0.58
-1.16	(0, 15)	(0.5, 15.5)	15	15.75 (worse) ---> accepted	0.75	0.52	0.11
-1.16	(0.5, 15.5)	(1, 16)	15.75	17 (worse) ---> accepted	1.25	0.34	0.14

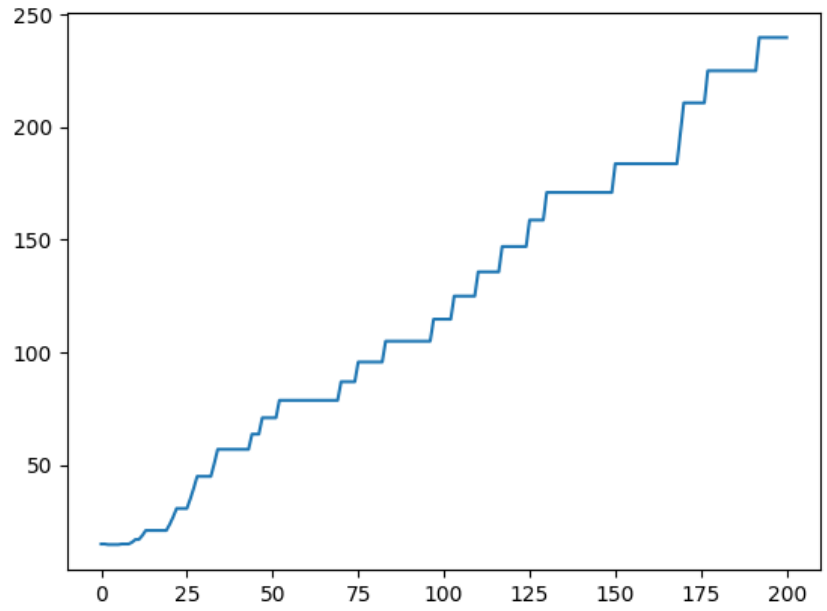
Best fitness: 17

<u>Simulated Annealing values:</u>

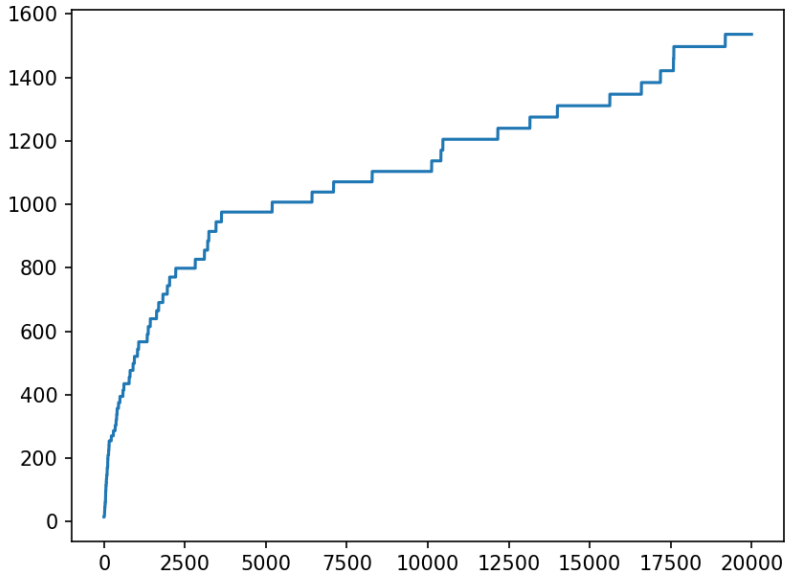
1. 14.75 --> 15
2. 15 --> 15.75
3. 15.75 --> 17



For 100 iterations:



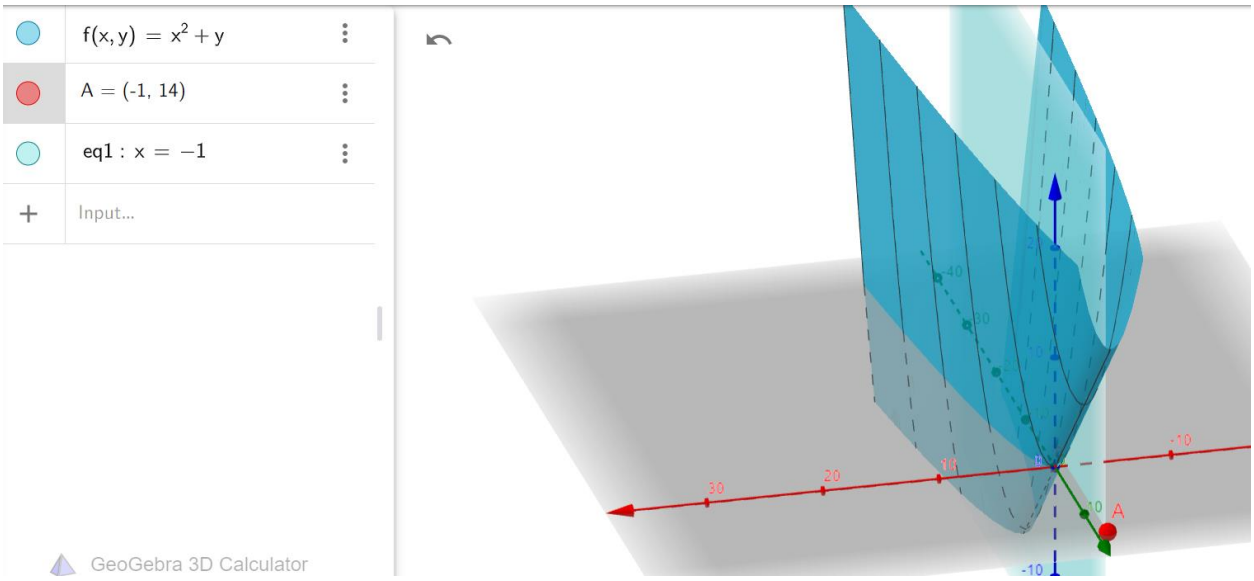
For 10,000 iterations:

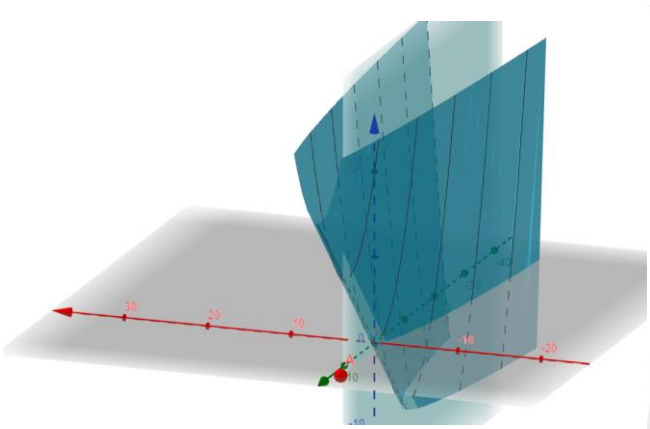


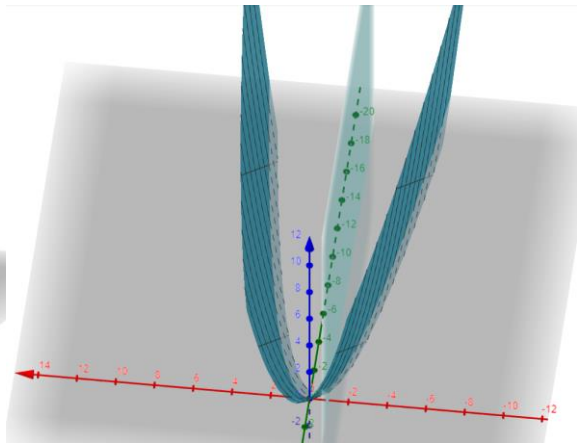
Findings:

It does seem like simulated annealing isn't beneficial in this case. As we are not getting the minimum but every time graph with going upwards. Like in the solutions, only dropped happened was on 14.75 but as we are accepting worse solution, it is impossible to get back on the previous solution. It will be even remained same or go for the worse.

This is all happening due to the function and by time behavior of the function defined in question. Lets see how the function looks like:







And for every exploration of neighbor, it is moving upward. As of 0.5 addition on both x & Y and the function is inclinedly moving upward.

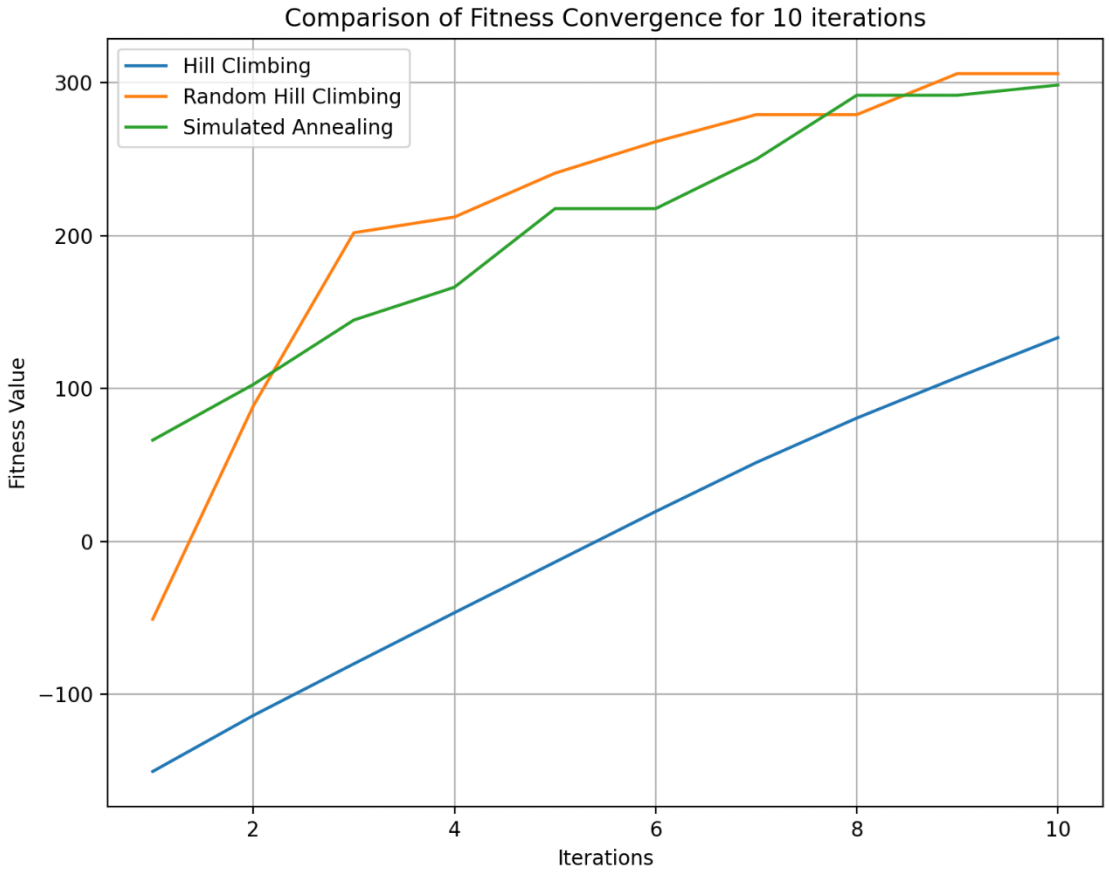
Note: all the simulations have been generated by code. The link to github is attached here:
Source code: https://github.com/mj06879/AI-CS-351-Assignments/blob/main/Homework_2/Simulated_annealing_on_f(x%2Cy).py

Q.3. [30 points] One of the benchmark functions to test Optimization techniques is the Rosenbrock function (https://en.wikipedia.org/wiki/Rosenbrock_function) A bivariate version is given as follows:

$$f(x,y) = 100 * (x^2 - y^2) + (1 - x)^2;\ -2 \leq x \leq 2, -1 \leq y \leq 3.$$

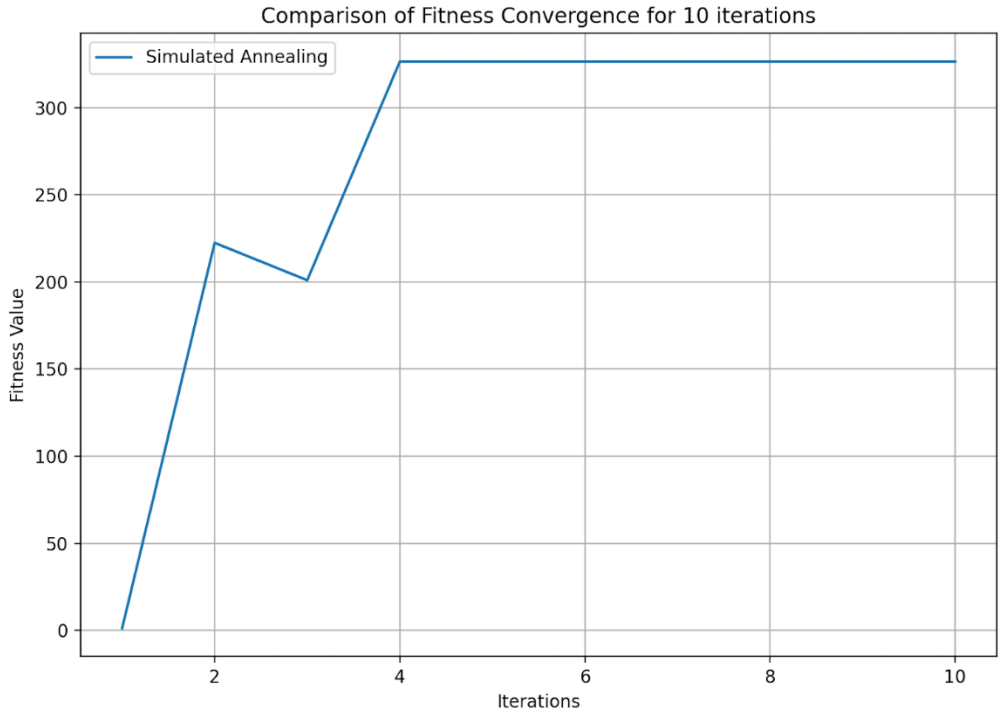
Implement **Hill Climbing**, **Random Hill Climbing**, and **Simulated Annealing** (in Python/C++/Java) to find the global maximum/minimum given the ranges for x and y. Comment on the performance of three techniques against different # of iterations (e.g., try for 10, 100, 500 iterations each) to see how fast the three techniques converge.

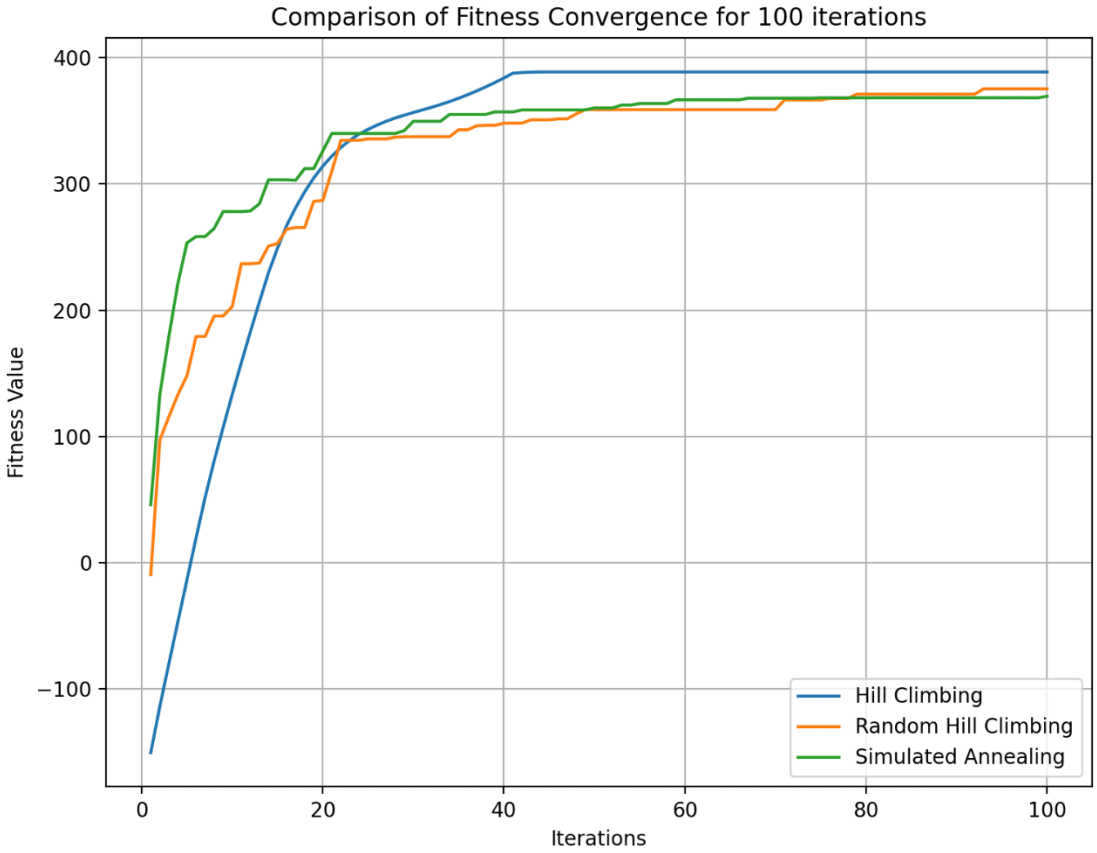
Answer) Since we were getting varying results every time we ran either of the techniques, we decided to run each technique for its set number of iterations (10, 100 and 500) ten times and then averaged the fitness values to get more accurate and reliable results. All the following graphs and observations are for the case of maximizing the function.



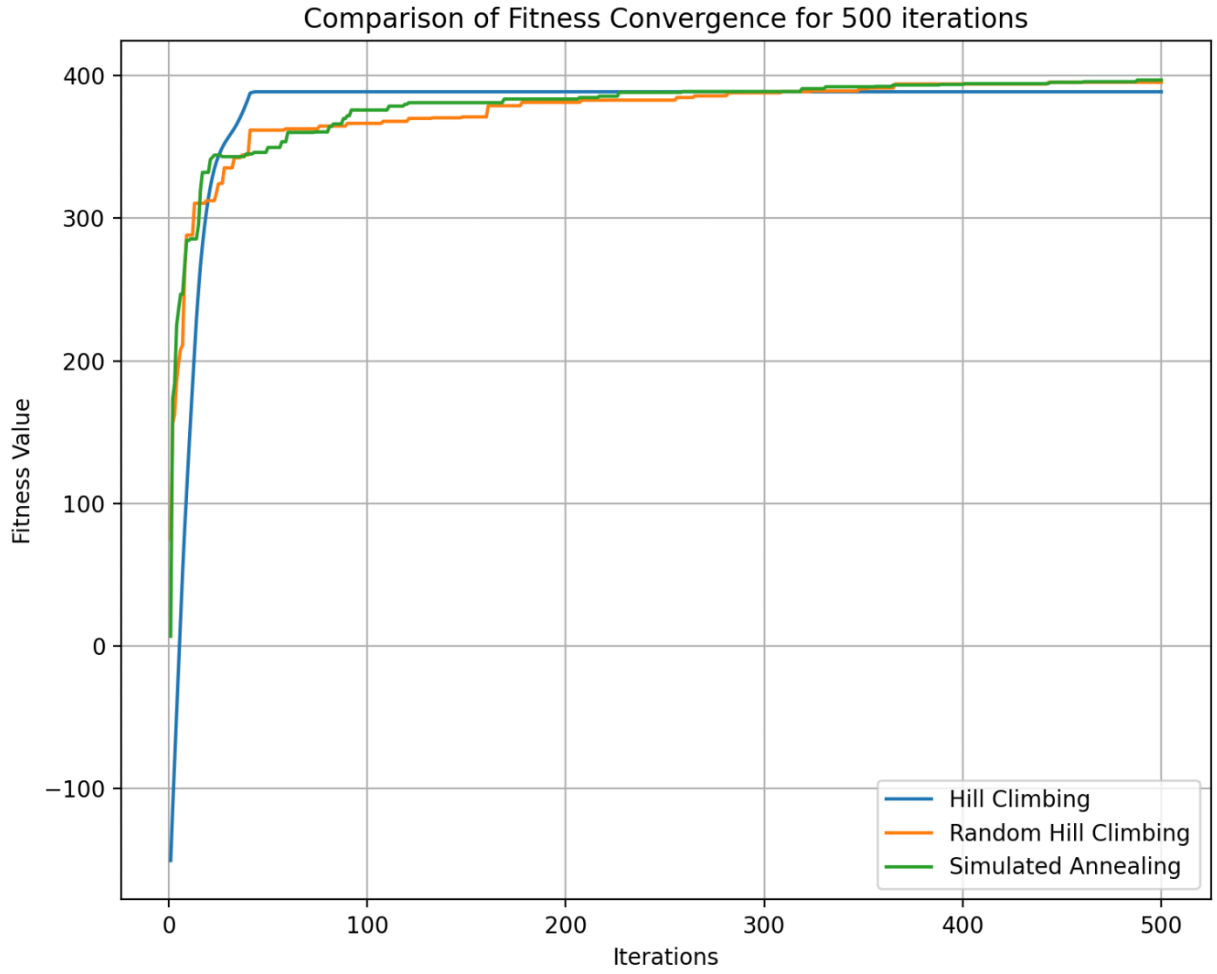
Explanation: In 10 iterations, we observe that for all three techniques there is a general increasing trend and none of the graphs have converged. For hill climbing, the graph increases smoothly since we look for candidates in the local vicinity of our current point and select the neighbor with the highest fitness if it is greater than the current fitness. Since we are searching in the local vicinity of our current point, the fitness value of any neighbor with a higher fitness will only be slightly higher than the current fitness which is why the graph increases smoothly and we can expect to find a candidate with a higher fitness every time. On the other hand, the graphs of

random hill climbing and simulated annealing have a similar trend where the fitness values either increase or stay the same. This makes sense for random hill climbing because in every iteration we consider a random candidate and select it if it has a higher fitness. Since this random point may not be close to our current point, its fitness can be considerably higher which explains the upward slope sections of the graph. However, if the fitness of the candidate is lower than the current fitness, we stick with the current point which explains the plateaus. In simulated annealing we again consider a random candidate and select it if it has a higher fitness than the current point which explains the upward slope sections (the higher fitness value can be considerably higher) however, if it has a lower fitness we select it with a probability. The reason there are always plateaus in the case of lower fitness candidates instead of some drops is because we have made the graph by running the technique 10 times and using average fitness values. If we do not use average values, then drops in the fitness value can be observed which is when simulated annealing accepts a bad state. This is shown by the following graph:

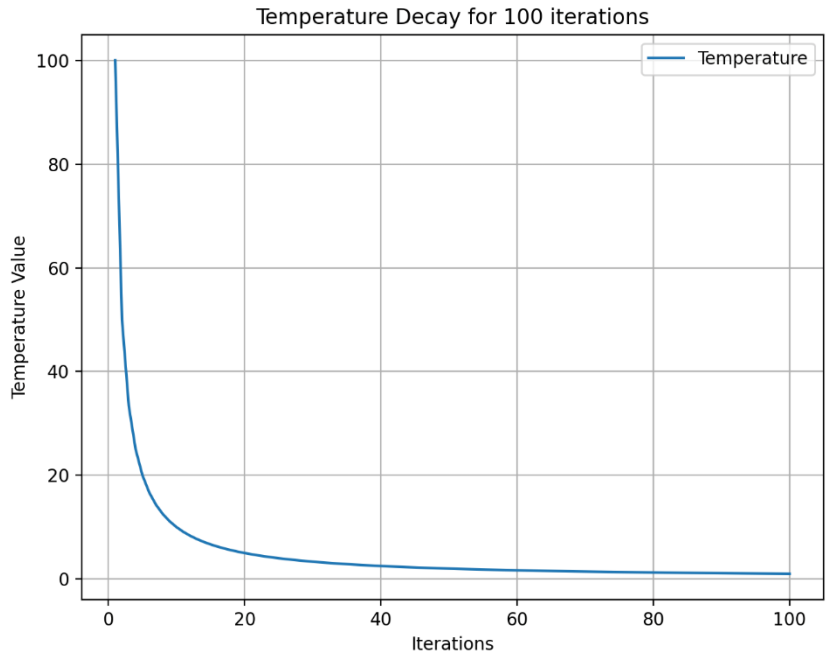


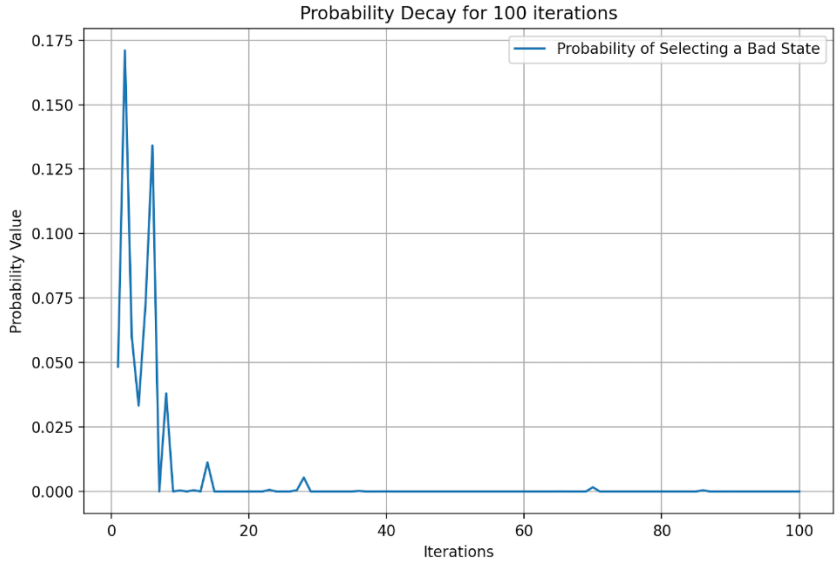


Explanation: We can observe that in the first 20 iterations all three techniques have a sharp increase in fitness values. Hill climbing's increase is smooth whereas the increase for random hill climbing and simulated annealing is rugged. The reason for this was explained in the previous explanation. After 20 iterations, the rate of increase for all three techniques decreases since they have now reached high fitness values which makes it less probable to find points with higher fitnesses. After 40 iterations, hill climbing converges as depicted by the flat line.The reason being that no other point near the current point has a higher fitness, thus hill climbing cannot move any further and is stuck at an optima. Random hill climbing converges after 60 iterations. The reason may be that it may not be able to find a random candidate with a higher fitness value and thus stays at the current solution. Lastly, simulated annealing maintains a gradual increase till the end which may be because it keeps exploring random points and has the possibility of encountering points that have a slightly higher fitness value. The reason why simulated annealing does not select a bad state in later iterations is because the temperature value becomes very low which reduces the probability of it selecting a state with a lower fitness value.



Explanation: In this case, hill climbing again converges the earliest at around 40 iterations since it possibly gets stuck in a local optima. On the other hand, random hill climbing and simulated annealing have similar trends as they maintain a gradual increase until the end. This is because they keep exploring random points and have the possibility of encountering points that have a slightly higher fitness value. Thus in conclusion hill climbing converges the quickest at a local optima whereas random hill climbing and simulated annealing converge late at a higher fitness values.





In Simulated Annealing we set an initial temperature value of 100 which was reduced after every iteration. This can be seen in the temperature decay curve as the temperature starts off at a value of 100 and then decreases exponentially to a value of less than 10 after 20 iterations, and then continues to decrease. What this achieves is that initially when the temperature is high, simulated annealing is more explorative since a high temperature value leads to a higher probability of accepting a bad state as can be seen in the probability decay graph. This ensures that a majority of the search space is explored. However, in order to converge to a value, after every iteration the temperature value drops which reduces the probability of accepting a bad state. After 20 iterations, when the temperature value is less than 10 we notice that the probability of accepting a bad state gets close to 0 or sometimes even 0. The reason being that the temperature value is very less and also that when the algorithm encounters a state that is better than the current state,

the probability of selecting a bad state is automatically 0. Thus as the iterations increase, simulated annealing becomes less explorative and more exploitative which results in it converging at an optima eventually.

Source Code: https://github.com/mj06879/AI-CS-351-Assignments/blob/main/Homework_2/hillClimbing-RandomHillClimbing-SimulatedAnnealing(q3).py

Q.4. [20 points] Given a function $f(x) = x^2$, find the maximum of the function using genetic algorithms (GA) for the range $-50 \leq x \leq 50$. Since we are doing it by hand, start with a small initial population of candidate solutions or chromosomes, say four (04), where each chromosome is represented by an 8-bit string representation within the given range for values of x. The **fitness function** is just the value of $f(x)$ for a given value of x. Start with four (04) chromosomes and then show the working of the genetic algorithm for two successive generations. Use a roulette-wheel **selection** (as discussed in the class), and a one-point (or, a two-point) **crossover** to create offspring for a pair of parents, with a 1% **mutation** rate, to randomly modify some bits in the chromosomes. **Replace** the old generation with the new generation. Repeat for two successive generations only.

Answer)

Population Initialization (chromosomes represented as 8 bit binary strings where the leftmost bit is the sign bit):

1. -12 represented as 10001100
2. 34 represented as 00100010
3. -24 represented as 10011000
4. 19 represented as 00010011

Fitness Evaluation:

1. f (-12) = $(-12)^2$ => 144 thus 10001100 has a fitness of 144
5. f (34) = $(34)^2$ => 1156 thus 00100010 has a fitness of 1156
2. f (-24) = $(-12)^2$ => 576 thus 10011000 has a fitness of 576
3. f (19) = $(19)^2$ => 361 thus 00010011 has a fitness of 361

Generation 1

Parent Selection using Roulette Wheel:

Using the normalized fitness values of each chromosome, we will construct ranges for each chromosome. We will then generate a random number between 0 and 1, and select the

chromosome that has the range within which the random number lies in. Thus chromosomes that have a higher fitness value will have a greater chance of being selected to become the parent.

n	Chromosome	Fitness	Normalized Fitness	Lower Bound	Upper Bound
1	10001100	144	0.06437192669	0	0.064371
2	00100010	1156	0.5167635226	0.06437192669	0.581135
3	10011000	576	0.2574877068	0.5811354493	0.838623
4	00010011	361	0.161376844	0.8386231561	1

Generating a random number between 0 and 1: 0.1140535161135724
This lies in the range of chromosome 2 thus our first parent is 00100010

Generating another random number which gives a different chromosome: 0.728993564882427

This lies in the range of chromosome 3 thus our second parent is 10011000

Crossover:

Parent 1: 00100010
Parent 2: 10011000

Generating a random index: 3

Selecting bits from index 0 to 2 from parent 1: 001
Selecting bits from index 3 to 7 from parent 2: 11000
Combining them together to create offspring 1: 00111000

Selecting bits from index 0 to 2 from parent 2: 100
Selecting bits from index 3 to 7 from parent 1: 00010
Combining them together to create offspring 2: 10000010

Mutation:

Since the mutation rate is 1%, in order to check if an offspring will be mutated we will iterate over it and for each but we will generate a random number between 0 and 1. If the random number is less than or equal to 0.01 (meaning 1%) then the bit at that position will be flipped. The resultant mutated chromosome will be returned and this process will be carried out for both the offsprings. The code that I used to implement mutation is stated below:

```
def mutation(chromosome):
    chromosome = list(chromosome)
    for bit in range(len(chromosome)):
        randomNum = random.random()
        if randomNum <= 0.01:
            if chromosome[bit] == '0':
                chromosome[bit] = '1'
            elif chromosome[bit] == '1':
                chromosome[bit] = '0'

    chromosome = "".join(chromosome)
    return chromosome
```

When I passed both of the offsprings through this function, I got the resultant chromosomes as:

```
Offspring 1 after mutation =  00110000
Offspring 2 after mutation =  10000010
```

This shows that offspring 1 got mutated from 00111000 to 00110000 and offspring 2 did not get mutated.

Fitness Evaluation of Off-springs:

1. Offspring 1 => 00110000 which is 48 has a fitness value of $f(48) = 48^2$ => 2304
2. Offspring 2 => 10000010 which is -2 has a fitness value of $f(-2) = -2^2$ => 4

Adding these to the population, we get the population as:

1. -12 represented as 10001100 with fitness value of 144
2. 34 represented as 00100010 with fitness value of 1156
3. -24 represented as 10011000 with fitness value of 576
4. 19 represented as 00010011 with fitness value of 361
5. 48 represented as 00110000 with fitness value of 2304
6. -2 represented as 10000010 with fitness value of 4

Survivor Selection:

In order to keep the population size the same, we select the best 4 candidates thus our updated population is now:

1. 34 represented as 00100010

2. -24 represented as 10011000
3. 19 represented as 00010011
4. 48 represented as 00110000

Generation 2

Parent Selection using Roulette Wheel:

n	Chromosome	Fitness	Normalized Fitness	Lower Bound	Upper Bound
1	00100010	1156	0.2629065272	0	0.262906
2	10011000	576	0.130998408	0.2629065272	0.393904
3	00010011	361	0.0821014328	0.3939049352	0.476006
4	00110000	2304	0.523993632	0.476006368	1

Generating a random number between 0 and 1: 0.7689723135733967
This lies in the range of chromosome 4 thus our first parent is 00110000

Generating another random number which gives a different chromosome:
0.08263203555145104
This lies in the range of chromosome 1 thus our second parent is 00100010

Crossover:

Parent 1: 00110000
Parent 2: 00100010

Generating a random index: 5

Selecting bits from index 0 to 4 from parent 1: 00110
Selecting bits from index 5 to 7 from parent 2: 010
Combining them to create offspring 1: 00110010

Selecting bits from index 0 to 4 from parent 2: 00100
Selecting bits from index 5 to 7 from parent 1: 000
Combining them together to create offspring 2: 00100000

Mutation:

After passing both the offsprings through the mutation function that we used in the first generation we get the resultant chromosomes as:

```
Offspring 1 after mutation =  00110010
Offspring 2 after mutation =  00100000
```

This shows that neither of the offspring got mutated.

Fitness Evaluation of Off-springs:

1. Offspring 1 => 00110010 which is 50 has a fitness value of $f(50) = 50^2$ => 2500
2. Offspring 2 => 00100000 which is 32 has a fitness value of $f(32) = 32^2$ => 1024

Adding these to the population, we get the population as:

1. 34 represented as 00100010 with fitness value of 1156
2. -24 represented as 10011000 with fitness value of 576
3. 19 represented as 00010011 with fitness value of 361
4. 48 represented as 00110000 with fitness value of 2304
5. 50 represented as 00110010 with fitness value of 2500
6. 32 represented as 00100000 with fitness value of 1024

Survivor Selection:

In order to keep the population size the same, we select the best 4 candidates thus our updated population is now:

1. 34 represented as 00100010
2. 48 represented as 00110000
3. 50 represented as 00110010
4. 32 represented as 00100000

Conclusion: It can be observed that after every generation the population consists of candidates with better fitness values meaning that the average fitness of the population increases. We were also lucky enough to get the global optimum (x = 50) after just 2 generations.

Source Code: https://github.com/mj06879/AI-CS-351-Assignments/blob/main/Homework_2/mutation(q4).py

Q.5. [10 points] Which of the following are true and which are false? Justify your choice in one to two sentences only.

1. Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.

2. $h(n) = 0$ is an admissible heuristic for the 8-puzzle problem.

3. A* is of no use in robotics because percepts, states, and actions are continuous.

4. Breadth-first search is complete even if zero step costs are allowed.

5. Assume that a rook can move on a chessboard any number of squares in a straight line, vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rook from square A to square B in the smallest number of moves.

Solution:

1. **False.** Depth-first search does not always expand as many nodes as A* with an admissible heuristic. A* with an admissible heuristic is designed to prioritize exploring paths that are likely to lead to an optimal solution, which often results in fewer nodes being expanded compared to depth-first search.
2. **True.** $h(n) = 0$ is an admissible heuristic for the 8-puzzle problem because it provides a lower-bound estimate of zero for the remaining cost to reach the goal state, which is always true since no moves are needed from the goal state
3. **False.** A* can be applied in robotics even when percepts, states, and actions are continuous. Techniques like discretization or approximation can be used to make A* suitable for continuous spaces.
4. **True.** A* can be applied in robotics even when percepts, states, and actions are continuous. Techniques like discretization or approximation can be used to make A* suitable for continuous spaces.
5. **True.** A* can be applied in robotics even when percepts, states, and actions are continuous. Techniques like discretization or approximation can be used to make A* suitable for continuous spaces.