Q1. Tic Tac Toe

a) Minimax

This is the average time computer took for every step of the game.
[Calculated over 10 games]

- 5.200206756591797 seconds for the first move
- 0.09863662719726562 seconds for second
- 0.009058237075805664 seconds
- 0.00244140625 seconds

Hence we can infer that, since in the beginning the tree is huge, the computer takes a lot of time to make its decision. At every time step, the tree becomes smaller and smaller and hence the decision making is faster.

b) Using Alpha beta Pruning

This is the average time computer took for every step of the game.
[Calculated over 10 games]

- Computer took 0.26935601234436035 seconds
- Computer took 0.02106332778930664 seconds
- Computer took 0.005557537078857422 seconds
- Computer took 0.0010161399841308594 seconds

Hence we can infer that alpha beta pruning greatly helps reduce the number of possible states. It is more prominent when the state space is huge.(Look at the difference in performance for the first move as compared to the last move.)

Q2. GA vs MA vs CSP

a) Logical Constraints:

- One course cannot be taught by two different professors.
- One course can be taught max once per day.
- One course needs to be taught at least twice in a week.
- One professors teaches max 2 slots in a day.
- One professor teaches at max 2 courses.
- There should be atleast a period gap between two classes to be taken by the same professor.

Please note that these are all desirable conditions and are not carved in stone. Our algorithm must be able to find the best solution even if not all these constraints are followed. Hence, in our fitness function, the chromosome is penalized for violation of these conditions. We attempt to minimize this constraint.

b)

GENETIC ALGORITHM

IMPORTANT DEFINITIONS:

- Gene
  - g = <course><prof><hall>
- Chromosome
  - g11,g12, .. , g18, g21, … , …, g58
  - Hence a chromosome has 40 genes.
  - An ordered sequence which refers to slot for day1 slot1, day2 slot2 and so on.
- Initial Population
  - <cph>, … , $<cph>_{40}$
  - Where c, p, and h are chosen randomly from [1,m], [1,p], and [1,n] respectively.
- Fitness Function
  - Sum of all the violated rules.
  - Please refer to the code and comments in it for further detail
- Crossover
  - A simple crossover is used for mixing genes.

Memetic Algorithm

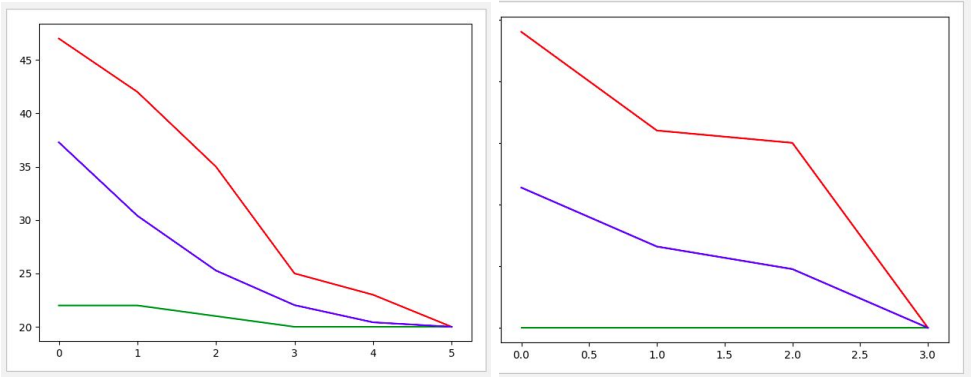
Exactly same as above with two major differences:

- The fitness function returns a set of all genes violating the constraints.

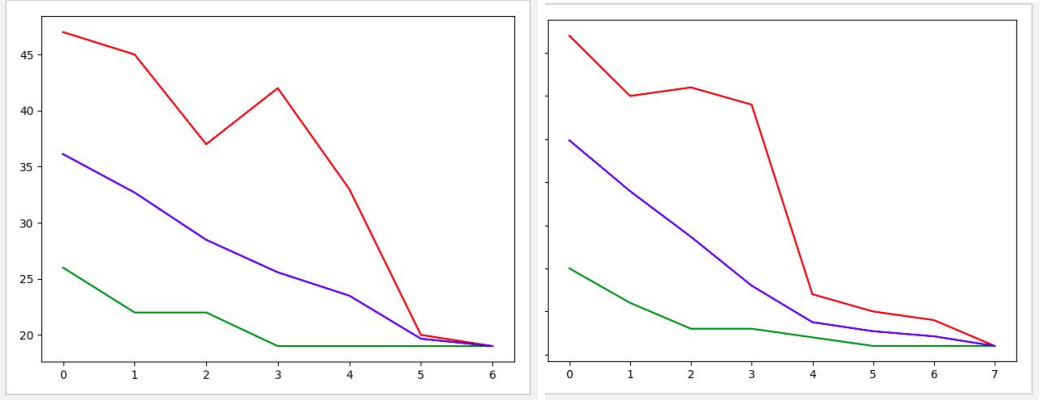
- We use a simple randomized version of hill climbing search to update all the chromosomes our population at every iteration.
    - While the score keeps on improving I replace every problematic gene by an epsilon-greedy method. With a small probability epsilon, I replace it by random values and otherwise I replace by the least used node.

Performance Contrast

Both Genetic and Memetic learning were able to solve the problem in an amazingly small number of generations.
However on an average Memetic Learning tended(it's all probabilistic) to outperform Genetic Algorithm by some generations mark over a wide range of values of M, N and P. The following chart shows a toy example with the following values.
# M courses, N Halls, P professors
M = 30
N = 80
P = 20



Memetic Learning : Penalty vs Generation Curve



Genetic Learning : Penalty vs Generation Curve

However, we must state that MA took a lot of more time to converge as as compared to MA(at-least for smaller input size). This is because of our probabilistic Hill climb search approach. For example in the

above case GA converged in 0.25665807723999023 seconds whereas MA converged in 1.3037867546081543 seconds.

CSP

I use recursive backtracking search to find the solution of the problem.

```
BackTracking(Level)
        If all variables assigned:
                Successful
        For every tuple in <cph>:
                If satisfyConstraint() == True:
                        BackTracking(Level + 1)
        Unsuccessful. No possible solution forward in this path.
```

But since my constraints are not absolute but just some factors that I've decided and want to minimize it, I define a **threshhold** variable. This variable refers to the scope of penalty allowed. Now in order to find a correct solution, I kept on increasing value of **threshhold** till I find a solution.

```
For i in range(100000):
        threshhold = i
        print("Running for threshhold of ",threshhold)
        backtrack(0)
        print("Failed to find a solution for this threshhold")
```

Testing this code on the above test case, we find the solution for threshhold = 21. This is also evident from the graph. Hence our results are consistent.