PROGRAMMING ASSIGNMENT - 2

Kshitij Srivastava
MT18099

QUESTION - 1 :

- **Assumptions** :
  - Human player always moves first
  - Human has been allocated “x”
  - Human player doesn’t take any invalid moves, i.e., once a cell is occupied, the human player doesn’t select that cell again.

- **Methodology for MINIMAX** :
  - After, the human takes the turn, the computer makes the moves using MINIMAX algorithm.
  - The algorithm is as follows:

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

- **<u>Methodology for ALPHA-BETA</u>** :
  - After, the human takes the turn, the computer makes the moves using MINIMAX algorithm.
  - The algorithm is as follows:

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

TIC TAC TOE RESULTS

MINIMAX:

```
Play your move : 9
| || || |
---------
| || || |
---------
| || ||x|
---------
Time taken to make this move = 0.12117362022399902 seconds
SOPHIA'S MOVE
| || || |
---------
| ||o|| |
---------
| || ||x|
---------
Play your move : 3
| || ||x|
---------
| ||o|| |
---------
| || ||x|
---------
Time taken to make this move = 0.005606412887573242 seconds
SOPHIA'S MOVE
| || ||x|
---------
| ||o||o|
---------
| || ||x|
---------
Play your move : 4
| || ||x|
---------
|x||o||o|
---------
| || ||x|
---------
Time taken to make this move = 0.0004973411560058594 seconds
SOPHIA'S MOVE
|o|| ||x|
---------
|x||o||o|
---------
| || ||x|
---------
Play your move : 7
|o|| ||x|
---------
|x||o||o|
---------
|x|| ||x|
---------
Time taken to make this move = 7.867813110351562e-05 seconds
SOPHIA'S MOVE
|o|| ||x|
---------
|x||o||o|
---------
|x||o||x|
---------
```

```
Play your move : 2
|o||x||x|
---------
|x||o||o|
---------
|x||o||x|
---------
DRAW!
```

ALPHA -BETA :

```
Play your move : 9
| || || |
---------
| || || |
---------
| || ||x|
---------
Time taken to make this move = 0.014664411544799805 seconds
SOPHIA'S MOVE
| || || |
---------
| ||o|| |
---------
| || ||x|
---------
Play your move : 3
| || ||x|
---------
| ||o|| |
---------
| || ||x|
---------
Time taken to make this move = 0.0024344921112060547 seconds
SOPHIA'S MOVE
| || ||x|
---------
| ||o||o|
---------
| || ||x|
---------
Play your move : 4
| || ||x|
---------
|x||o||o|
---------
| || ||x|
---------
Time taken to make this move = 0.00044608116149902344 seconds
SOPHIA'S MOVE
|o|| ||x|
---------
|x||o||o|
---------
| || ||x|
---------
Play your move : 7
|o|| ||x|
---------
|x||o||o|
---------
|x|| ||x|
---------
Time taken to make this move = 9.1552734375e-05 seconds
SOPHIA'S MOVE
|o|| ||x|
---------
|x||o||o|
---------
|x||o||x|
---------
```

```
Play your move : 2
|o||x||x|
---------
|x||o||o|
---------
|x||o||x|
---------
DRAW!
```

- **Observation** :
  - There's quite a visible difference in the time taken to make the first move between minimax and alpha-beta.
  - However, it gets decreased with the advancement of the game.
- **Inference** :
  - The number of states that minimax visits for the initial moves is quite large as compared to alpha-beta, hence , the large amount of time for making the initial moves.
  - However, with the game's advancement, this number gets reduced and both perform almost equally towards the end.

GENETIC, MEMETIC, CSP

QUESTION - 2 :

- **Assumptions** :
  - Professors have been mapped to courses uniformly.
  - The total number of slots to fill in a week is a hyper parameter.
  - Number of lecture halls = 4
  - Number of professors = 20
  - Number of courses = 40
  - Number of days/week = 5
  - Number of slots/day = 8
- **Logical Constraints :**
  - A professor cannot be present in two lecture halls for a particular day during the same slot in a chromosome.
  - Two courses cannot be allotted the same lecture hall for a particular day during the same slot.

GENETIC ALGORITHM

- **Methodology** :
  - Firstly, I have mapped all the courses to professors uniformly.
  - Then, randomly created a gene having the given structure.

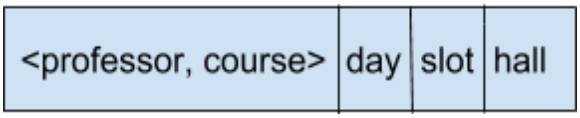


Fig. Structure of a single gene

  - Then created a chromosome of size = 8 * 5 * (number of lecture halls).
  - The chromosome is 1-D array of genes.
  - Now, similarly, we create a random population of genes.
  - Then calculate the fitness of each chromosome in the population, and sort in non-decreasing order.
  - Then selected top - k chromosomes for crossover and created another population.
  - Now, merged both, the initial as well as the crossover population.
  - Then calculated its fitness and again sorted.
  - Then, cut the population size to 100.
  - Performed mutation on these 100 chromosomes.
  - Now, added the top- k performing parents from the unmutated population to this new population.
  - Make this the new population and repeat the process for it.

- **FITNESS EVALUATION** (same for both GA and MA):
  - Initially, the fitness score is zero.
  - For each conflict in a chromosome the score is increased by 1.

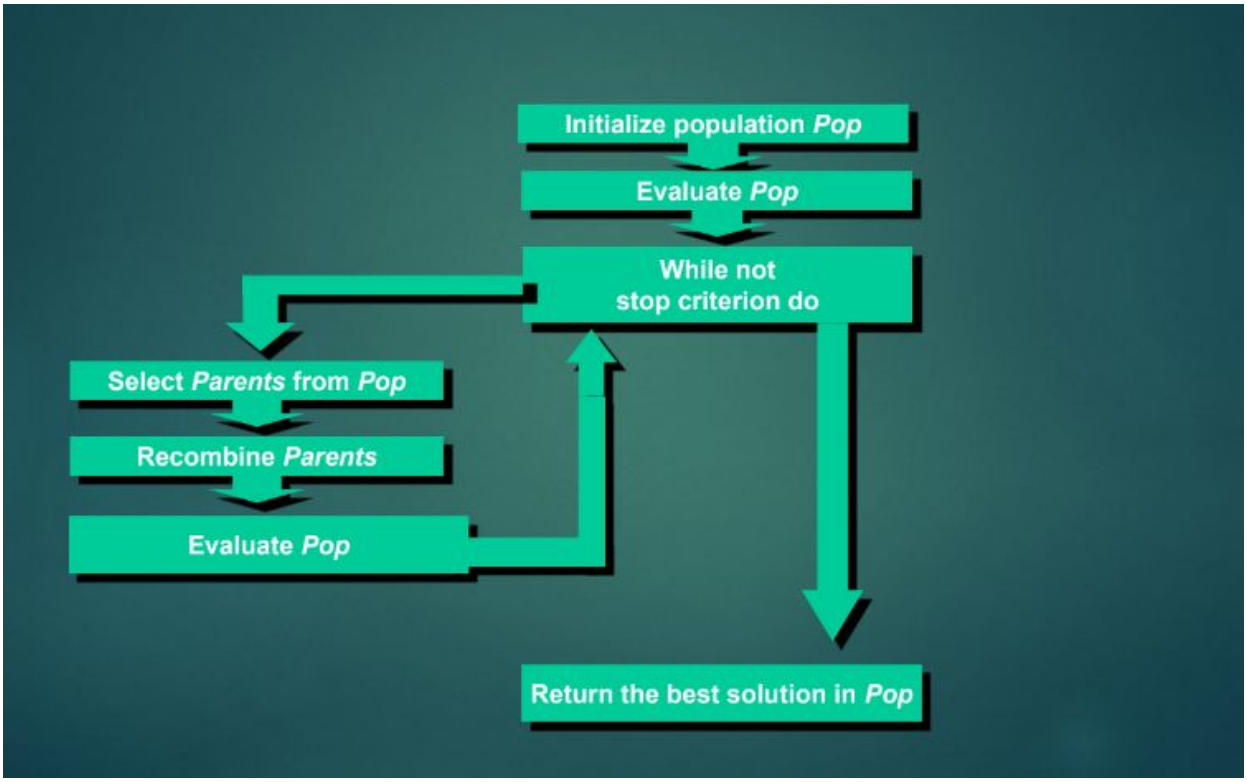
```
def calc_fitness(chromosome):
    fitness = 0
    for i in range(n_slots):
        gene = []
        gene.append(chromosome[i])
        test1 = list([])
        test1.append([gene[0][0][0], gene[0][1], gene[0][2]])
        test2 = list([])
        test2.append([gene[0][1], gene[0][2], gene[0][3]])

        for j in range(i + 1, n_slots):
            sche_slot_test1 = list([])
            sche_slot_test2 = list([])
            sche_slot_test1.append([chromosome[j][0][0], chromosome[j][1], chromosome[j][2]])
            sche_slot_test2.append([chromosome[j][1], chromosome[j][2], chromosome[j][3]])

            if test1 == sche_slot_test1 or test2 == sche_slot_test2:
                fitness += 1
                break

    return fitness
```

❖ **Algorithm** :



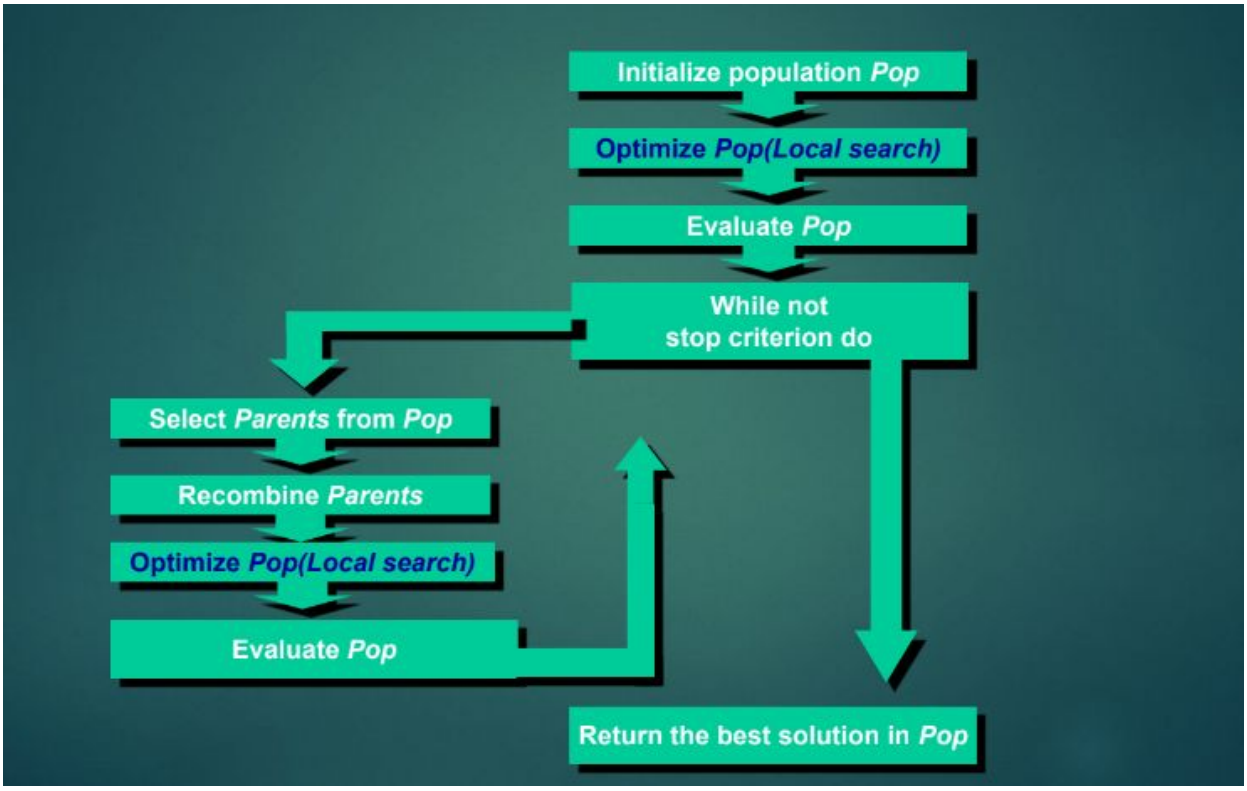
❖ **Aspects of the process followed** :

➢ **Genetic Algorithm :**

- **Gene generation** : It is random in the sense that random combinations of <professor, course>, day, slot and hall are being used to form a gene.
- **Chromosome generation** : Whatever random genes are formed, they are added to form a chromosome.
- **Population generation** : A fixed number of chromosomes are generated using the above method for creating a population.
- **Crossover methods** : Single-point crossover is being used by randomly generating the crossover point.
- **Evaluation Function** : A +1 is added to the fitness value for each conflict.
- **Mutation method** : A fixed number of genes are chosen at random and replaced with new ones.

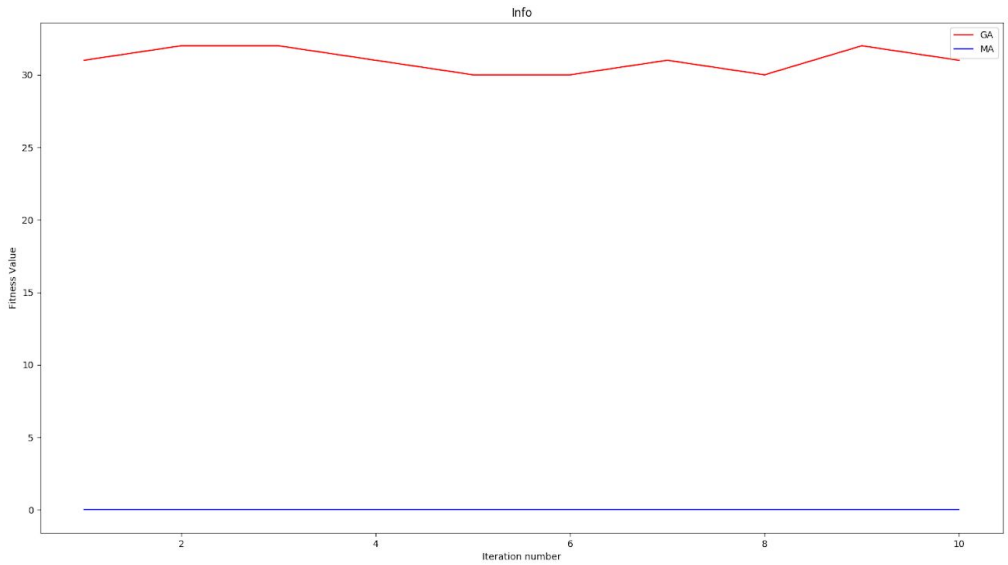
MEMETIC ALGORITHM

- **Methodology** :
  - The generation of gene and chromosome is the same as for genetic algorithm.
  - However, initially, only 10 chromosomes are being generated.
  - If there are any conflicting genes in these chromosomes, they are replaced with new ones.
  - The process doesn't move forward unless there's some improvement in its fitness.
  - After this, crossover is done to form a population.
  - This population is then sorted according to the fitness values of its chromosomes.
  - And then again, top 10 chromosomes are selected for the process to repeat.
- **Algorithm** :



- **Aspects of the process followed** :
  - **Memetic Algorithm :**
    - **Gene generation** : It is random in the sense that random combinations of <professor, course>, day, slot and hall are being used to form a gene.
    - **Chromosome generation** : Whatever random genes are formed, they are added to form a chromosome.
    - **Population generation** : A fixed number of chromosomes are generated using the above method for creating a population.
    - **Local Search** : A conflicting gene is replaced by a newly generated one, and the chromosome is checked for any improvement in its fitness.
    - **Crossover methods** : Single-point crossover is being used by randomly generating the crossover point.
    - **Evaluation Function** : A +1 is added to the fitness value for each conflict.

- **Comparison between GA and MA** :
  - **Running GA and MA for randomly generated datasets:**



- **Observation :**
  - GA averages out at a worse fitness score than MA.
  - MA is able to completely solve the problem at hand.

- **Inference :**
  - The reason for GA averaging at a worse score is because of crossover, there is some localisation of genes and hence, the entire population might settle at some average score, i.e. , achieve convergence.
  - MA is able to solve the problem completely because with the help of local search we are able to target the exact conflicting genes and deterministically improve the fitness score.

CSP

- **Assumptions** :
  - Professors have been mapped to courses uniformly.
  - The problem is to fill in a given number of slots.
  - The CSP problem here has been solved for 150 slots.
- **Methodology** :
  - We have a list of the given number of slots.
  - We randomly generate 10 tuples and place one of them into the schedule list.
  - Put the other 9 in a backstack.
  - If the inserted tuple is valid, i.e, it satisfies the given constraints, we move on to the next slot and repeat the process.
  - In case the tuple doesn't satisfy the constraints, we pop a new tuple from the backstack and insert it into the schedule and check for validity.
  - We repeat this process until all the slots are filled with valid tuples.

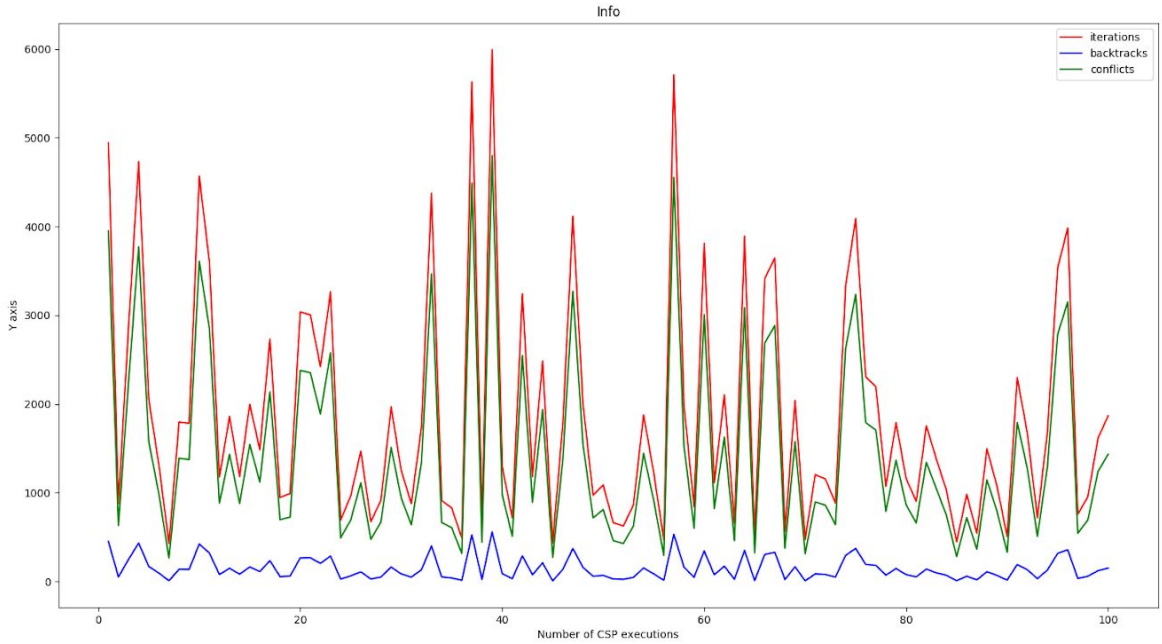
- **Algorithm** :

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

❖ NUMBER OF ITERATIONS OF CSP VS PARAMETERS PLOT



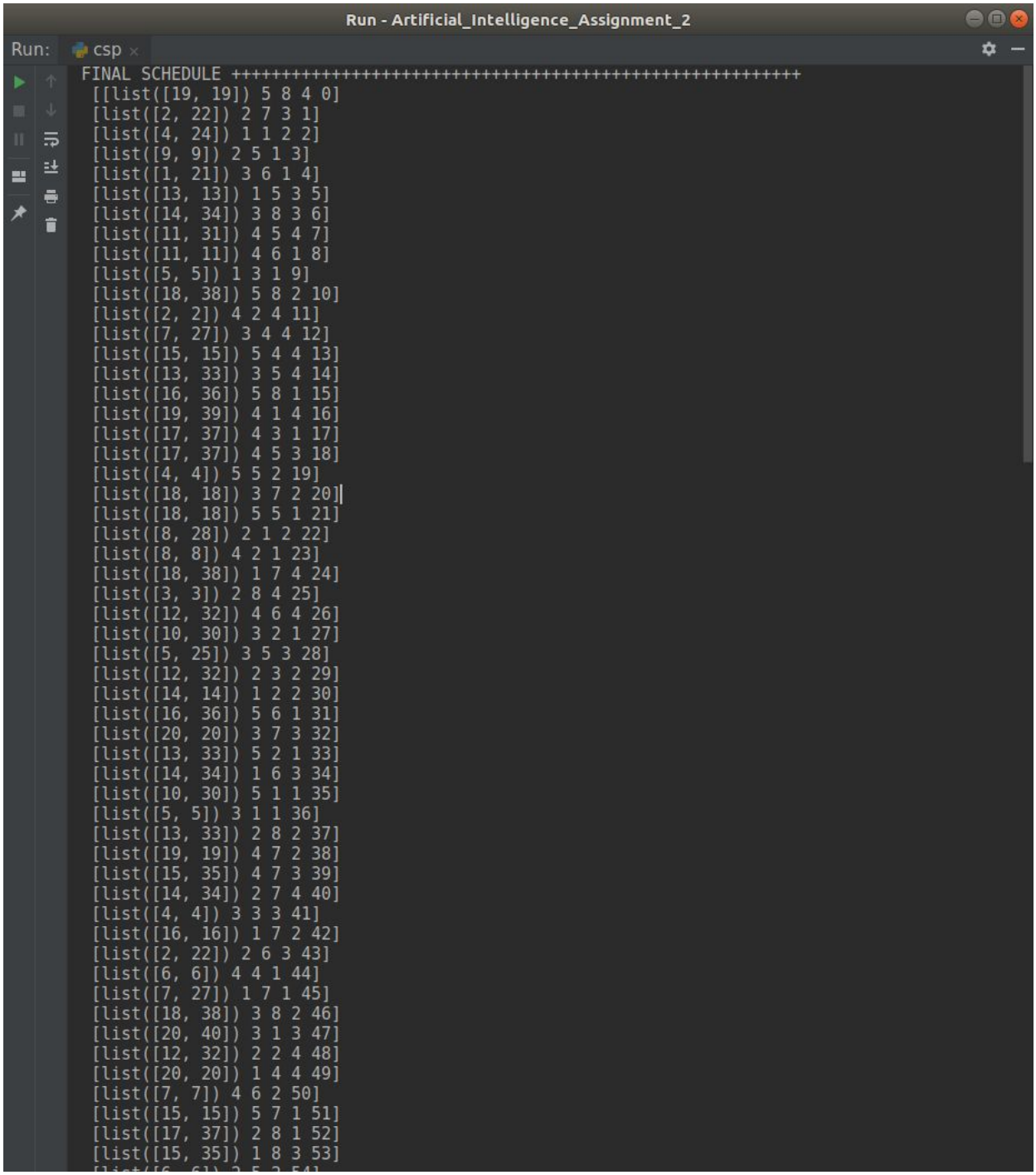
❖ **Observations :**

- ➢ Here we observe that for each iteration of CSP, the number of conflicts are quite close to the total number of iterations performed to solve that particular problem.
- ➢ Also, that the number of times that we have to backtrack is quite less as compared to the total number of conflicts encountered.

❖ **Inference** :

- ➢ The total conflicts are close to the number of iterations as the genes are being generated randomly and the probability of a conflict is quite high.
- ➢ However, the times that we actually backtrack is quite less than the number of conflicts because some gene from the backstack will eventually be a valid one.

❖ **Result** :


Run - Artificial_Intelligence_Assignment_2
Run:
csp
FINAL SCHEDULE ++
[[list([19, 19]) 5 8 4 0]
[list([2, 22]) 2 7 3 1]
[list([4, 24]) 1 1 2 2]
[list([9, 9]) 2 5 1 3]
[list([1, 21]) 3 6 1 4]
[list([13, 13]) 1 5 3 5]
[list([14, 34]) 3 8 3 6]
[list([11, 31]) 4 5 4 7]
[list([11, 11]) 4 6 1 8]
[list([5, 5]) 1 3 1 9]
[list([18, 38]) 5 8 2 10]
[list([2, 2]) 4 2 4 11]
[list([7, 27]) 3 4 4 12]
[list([15, 15]) 5 4 4 13]
[list([13, 33]) 3 5 4 14]
[list([16, 36]) 5 8 1 15]
[list([19, 39]) 4 1 4 16]
[list([17, 37]) 4 3 1 17]
[list([17, 37]) 4 5 3 18]
[list([4, 4]) 5 5 2 19]
[list([18, 18]) 3 7 2 20]
[list([18, 18]) 5 5 1 21]
[list([8, 28]) 2 1 2 22]
[list([8, 8]) 4 2 1 23]
[list([18, 38]) 1 7 4 24]
[list([3, 3]) 2 8 4 25]
[list([12, 32]) 4 6 4 26]
[list([10, 30]) 3 2 1 27]
[list([5, 25]) 3 5 3 28]
[list([12, 32]) 2 3 2 29]
[list([14, 14]) 1 2 2 30]
[list([16, 36]) 5 6 1 31]
[list([20, 20]) 3 7 3 32]
[list([13, 33]) 5 2 1 33]
[list([14, 34]) 1 6 3 34]
[list([10, 30]) 5 1 1 35]
[list([5, 5]) 3 1 1 36]
[list([13, 33]) 2 8 2 37]
[list([19, 19]) 4 7 2 38]
[list([15, 35]) 4 7 3 39]
[list([14, 34]) 2 7 4 40]
[list([4, 4]) 3 3 3 41]
[list([16, 16]) 1 7 2 42]
[list([2, 22]) 2 6 3 43]
[list([6, 6]) 4 4 1 44]
[list([7, 27]) 1 7 1 45]
[list([18, 38]) 3 8 2 46]
[list([20, 40]) 3 1 3 47]
[list([12, 32]) 2 2 4 48]
[list([20, 20]) 1 4 4 49]
[list([7, 7]) 4 6 2 50]
[list([15, 15]) 5 7 1 51]
[list([17, 37]) 2 8 1 52]
[list([15, 35]) 1 8 3 53]