MHO – ASSIGNMENT 2

CONTENTS:

1.1 Tsp 2 opt local search

1.1.1 Basic 2 opt local search…….. 2

1.1.2 Variant 1……3

1.1.3 Variant 2……3-4

Evaluation……..5-8

Conclusion……..8

1.2Run Time Distributions……9-12

MHO – ASSIGNMENT 2

1.1 TSP 2 OPT LOCAL SEARCH

2-Opt is a Local search technique, which helps us in getting to a good solution. Idea here is to consider a route which is crossing over itself and then re-order the it, such that there is no crossover. 2-Opt search will introduce 2 new edges by removing other 2 edges. We will have only 2 new costs to be computed and added to the original cost and remove the 2 old costs (where the edges are removed).

1.1.1 Basic 2-Opt:

This is also called as complete 2-opt search, which iteratively selects an edge and compares every possible combination (without altering the first city) of edges to swap with.

For each edge, all possible combinations are searched, and only the best possible move is applied for the tour. This is continued until all edges are compared with all other edges. So, it searches exhaustively for all edge combinations to swap with

1.1.2 Variant1:

In this, we are following below steps:

1. Select an edge at random
2. Search all other edges to swap with the selected random edge
3. Apply only the best move to the whole tour at the randomly selected edge

Termination Criteria: Max iterations or max number of non-improving moves

We have used Max iterations as termination criteria, where at each iteration steps 1 to 3 are followed

We need any one of the above said termination criteria, because if the search gets stuck in local optima (Fails to improve upon the previous best move), it goes on and on for ever.

Pseudo Code:

```
tour = InitializeTour()
for iteration in range(max_iterations)
{
        opt = False
        curr_best = tour.cost
        Select random edge
        While i < n
        {
                new_cost = swapCost(indices)
                if new_cost < curr_best
                {
                        best_move = indices
                        curr_best = new_cost
                        opt = True
                }
                i += 1
        }
        if opt
        {
                updateTour(tour, best_move)
        }
}
```

1.1.3 Variant2:

This is almost same as the Variant1, but instead of applying the best move to the tour, the first improvement is applied and will break out the loop of searching the better edges. Below are the steps that we follow:

1. Select an edge at random
2. Search other edges to swap with the selected random edge
3. Apply the first best improvement and break out the loop in the 2nd step

Like for variant1, in this case also, we may get stuck in local optima and we need termination criteria for this variant as well. We have considered max iterations as termination criteria and at each iteration, a random edge is selected from the last updated tour, to search and apply the first best improvement to the tour.

MHO – ASSIGNMENT 2

Pseudo Code:

```
tour = InitializeTour()
for iteration in range(max_iterations)
{
        curr_best = tour.cost
        Select random edge
        While i < n
        {
                new_cost = swapCost(indices)
                if new_cost < curr_best
                {
                        best_move = indices
                        curr_best = new_cost
                        updateTour(tour, best_move)
                        break
                }
                i += 1
        }
}
```

Let's see with an example how these variants work:

Let's say we have 10 cities as shown below:

Indices	0	1	2	3	4	5	6	7	8	9
Cities	A	B	C	D	E	F	G	H	I	J

1. First randomly selected an edge

Indices	0	1	2	3	4	5	6	7	8	9
Cities	A	B	C	D	E	F	G	H	I	J

   Let's say E-F is a randomly selected edge which is highlighted above

2. Starting with index 0, we search all other edges to swap with the edge E-F

Indices	0	1	2	3	4	5	6	7	8	9
Cities	A	B	C	D	E	F	G	H	I	J

Indices	0	1	2	3	4	5	6	7	8	9
Cities	A	B	C	D	E	F	G	H	I	J

3. As shown in the above two figures, the highlighted edges will be considered to swap with the randomly selected edge in Step 1, depending on the index of the edge we are

considering to swap with, such that at any point in time, the link between the randomly selected edge (E-F in this case) is swapped with any other edge

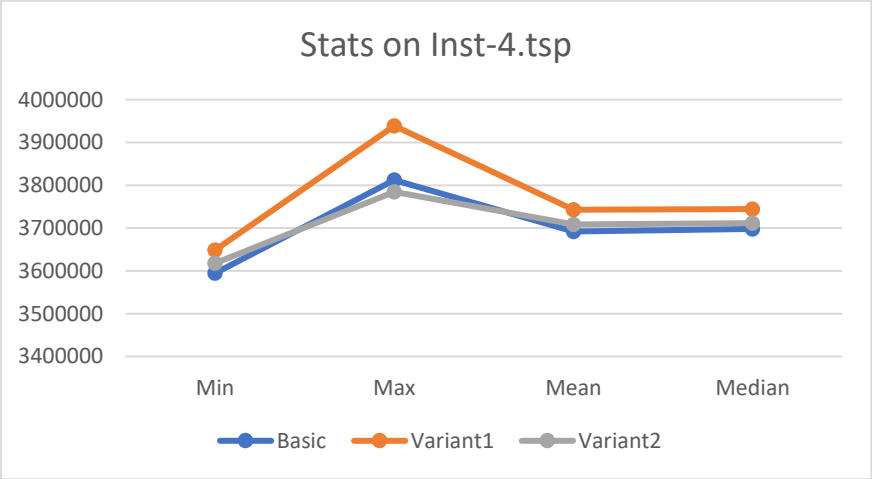
Due to the randomness involved in selecting an edge to swap with, it will get stuck in local optimum and hence the above steps 1,2 and 3 are repeated for either max_iterations or max number of non-improving updates.

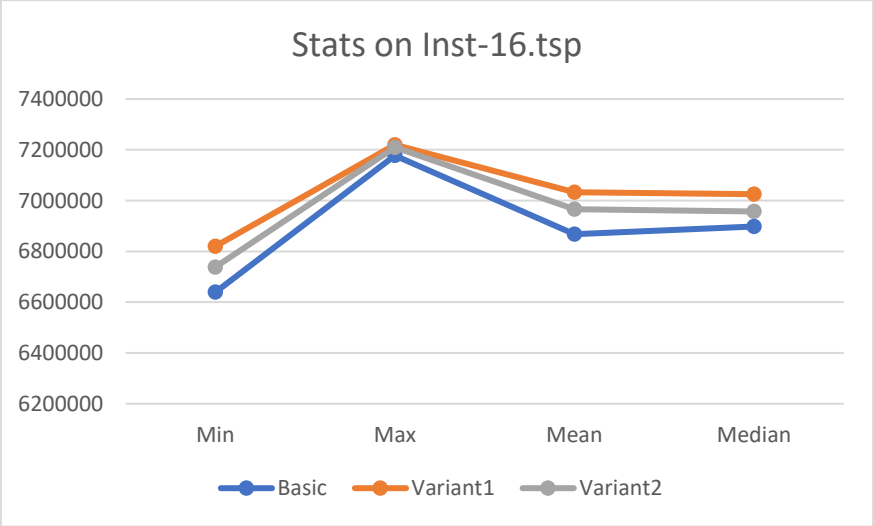
Variant1 and Variant2 works exactly same except the tour is updated with the best move for Variant1 and first improved move in the Variant2, after which the next random edge is selected to swap with all other edges as explained in the above example

EVALUATION OF ALL THE ABOVE THREE VARIANTS:

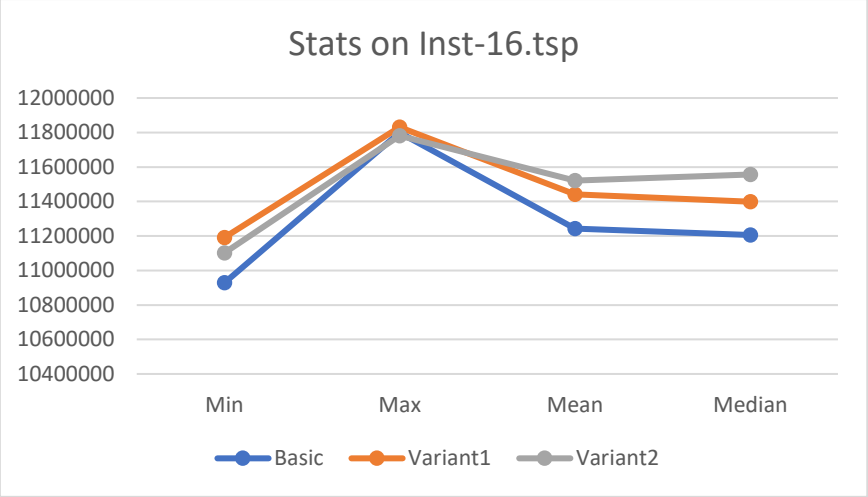
All of the above variants of 2-opt algorithm have been run on inst-4.tsp, inst-16.tsp and inst-6.tsp, each for 10 runs and below are the results:

OVERALL PERFORMANCE:





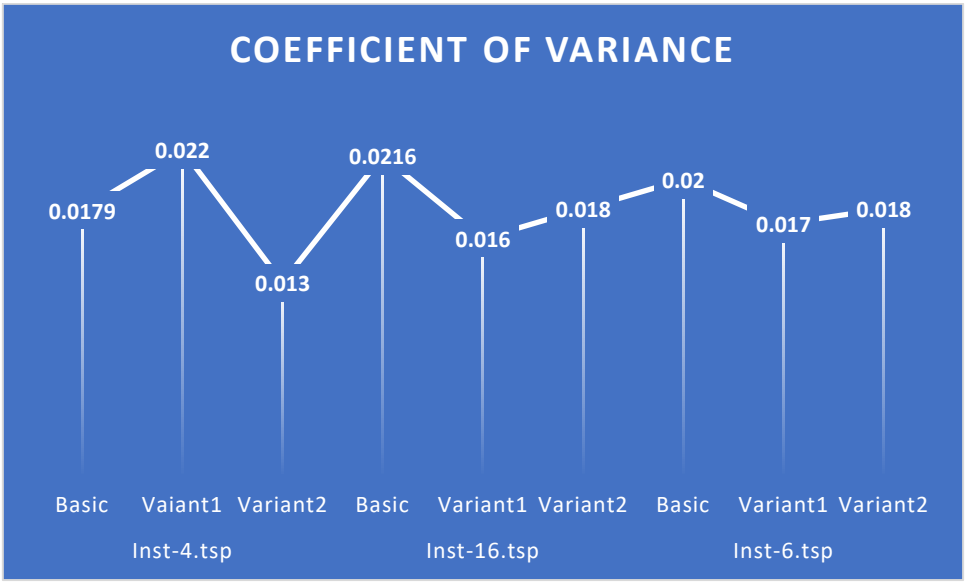
MHO – ASSIGNMENT 2



From above graphs, we can see that the Basic version of 2-opt algorithm, is giving us better results (converging / getting the tour with less cost) when compared to the 2 variants

If we sort the algorithms from best to worst, Basic algo comes on top, followed by variant1 and then at last variant2

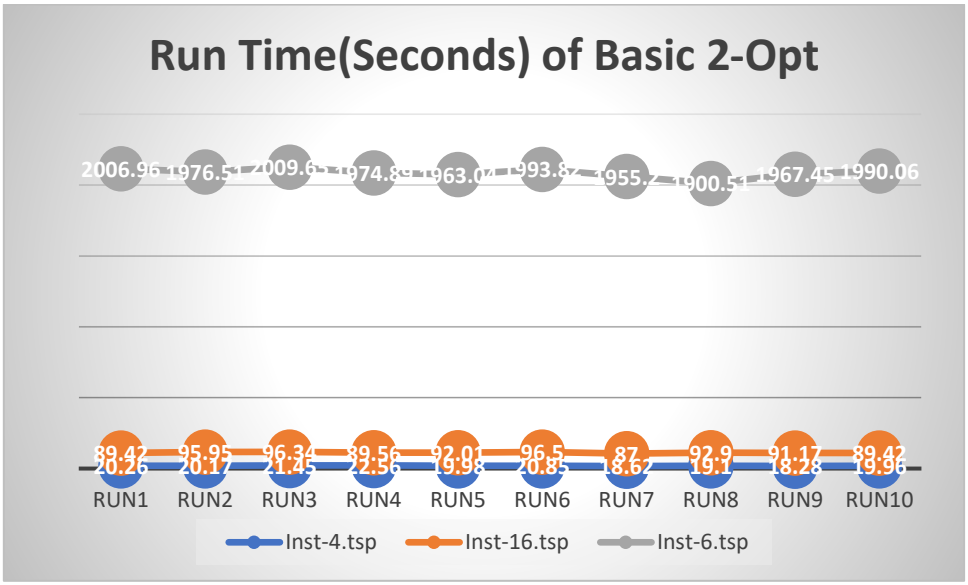
COEFFICIENT OF VARIANCE:

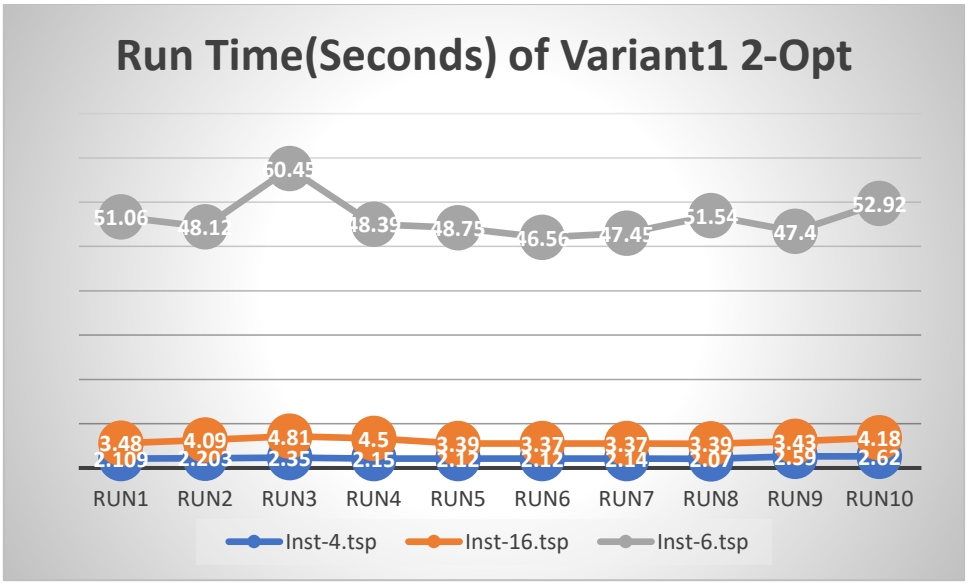


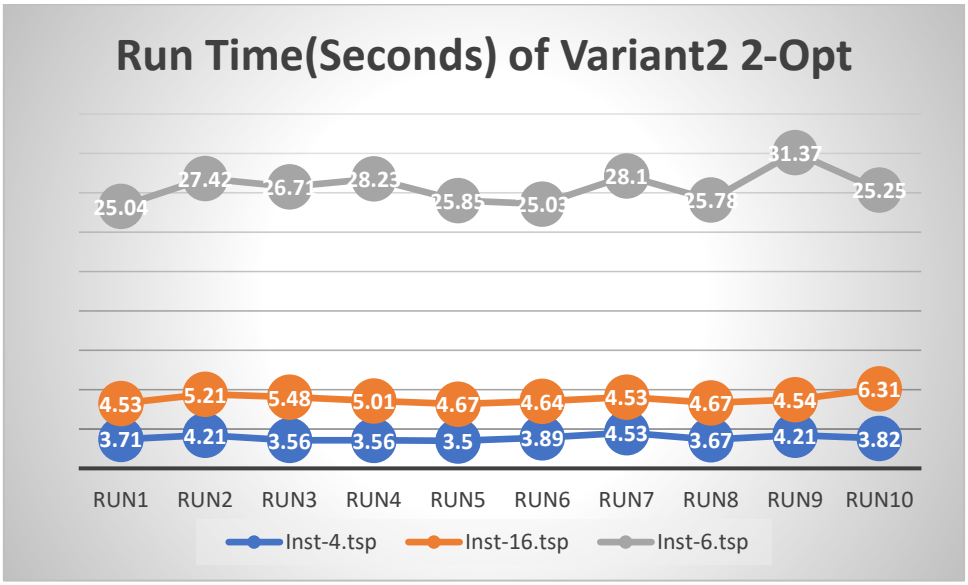
Coefficient of variance for each variant, across the runs for each instance files, varies in the range 0.01 to 0.02. This indicates the consistency in the performance of the algorithms.

MHO – ASSIGNMENT 2

RUN TIME COMPARISION:



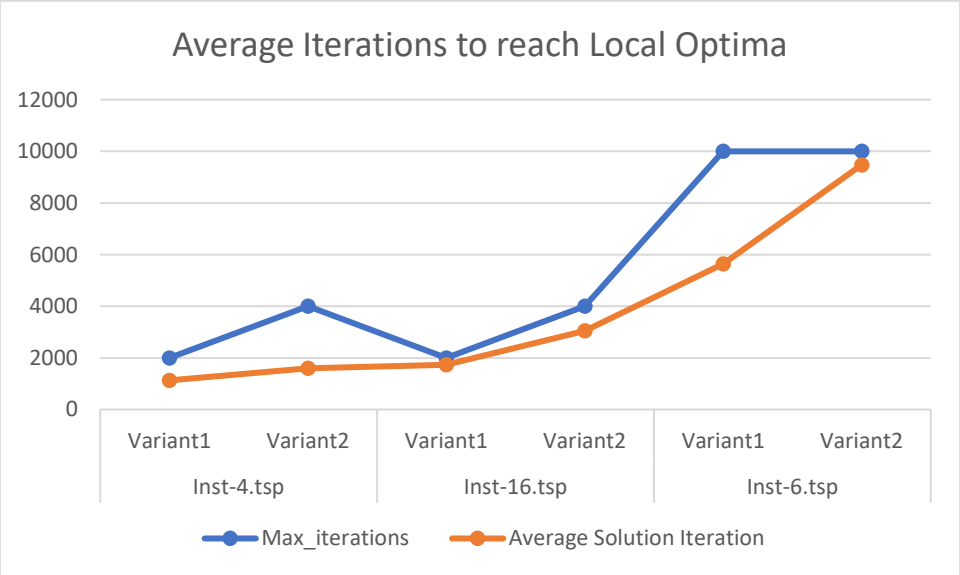




MHO – ASSIGNMENT 2

Since basic 2opt searches exhaustively, run time is on the higher side, variant1 and variant2 are faster versions, and run times are very less when compared to basic 2opt.

ITERATIONS TAKEN BY VARIANT1 & VARIANT2 TO GET TO A GOOD SOLUTION:



Variant2 takes a greater number of iterations to get to a good solution when compared to Variant1. This can be seen with significant difference with the larger instance file Inst-6.tsp.

Conclusion:

1. To conclude, out of the three variations, basic algorithm gives us better results compared to the other two variants, but at the cost of run time (wait time).
2. Variant1 does not search exhaustively all the edges, instead it searches for specified number of max_iterations
3. Variant2 also, does not search exhaustively, but it is much faster than Basic and variant1 version of the algorithm
4. Both the variants give acceptable results, excluding the basic version
5. If runtime is a constraint, then one can go with variant1 or variant2 as per their convenience.

MHO – ASSIGNMENT 2

1.2 RUN TIME DISTRIBUTION

We are solving N-Queens and recording the run time for all the successful runs along with the total number of steps it took for us to solve the problem.

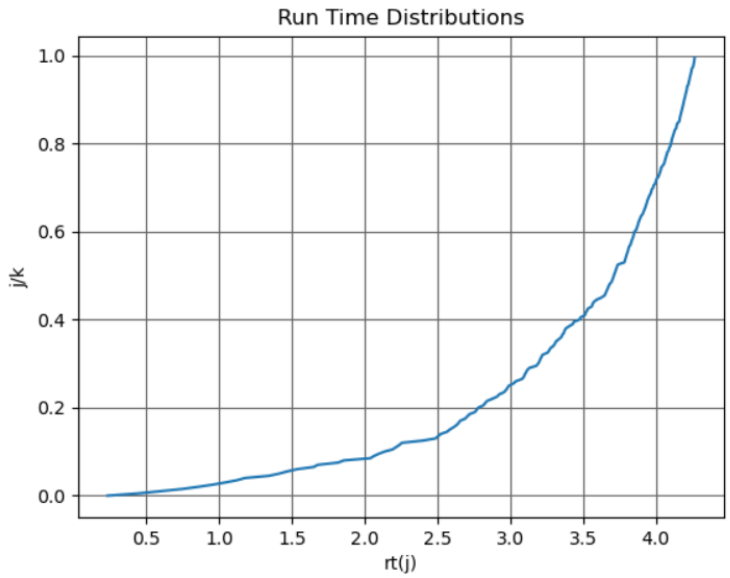
There are two ways to solve the MinMax problem in this case, one allowing sideways moves (to avoid getting stuck at the **shoulder point**) and one without allowing sideways moves.

Allowing sideway moves will yield us more solved instances than not allowing ones.

RTDs are obtained for solved instances against run time and number of steps it took to get the solution, the results are as shown below:

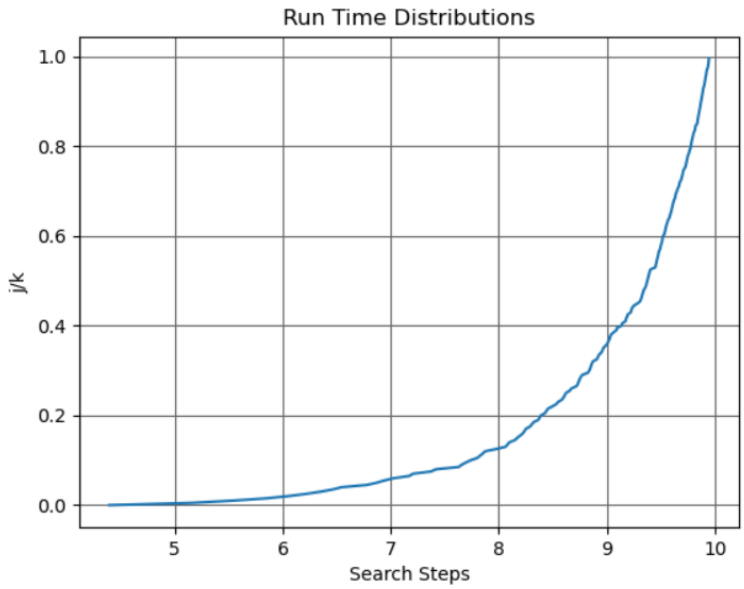
With Sideways movement allowed:

Below output is obtained for 200 runs, with 100 iterations each and 10 restarts for n = 57 (50 + last digit of student id)



This is a semi log view, applied over runtime rt(j) (along x-axis)

MHO – ASSIGNMENT 2



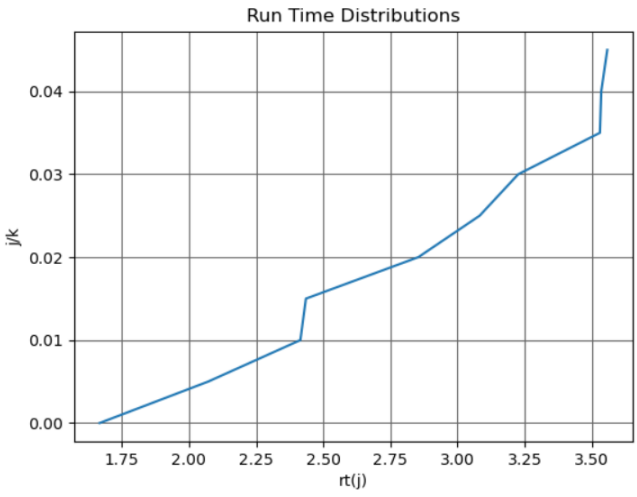
This is a semi log view, applied over search steps (along x -axis)

We are getting solution at every run, when we are allowing the sideways movement, because of the nature of the update that happens due to this.

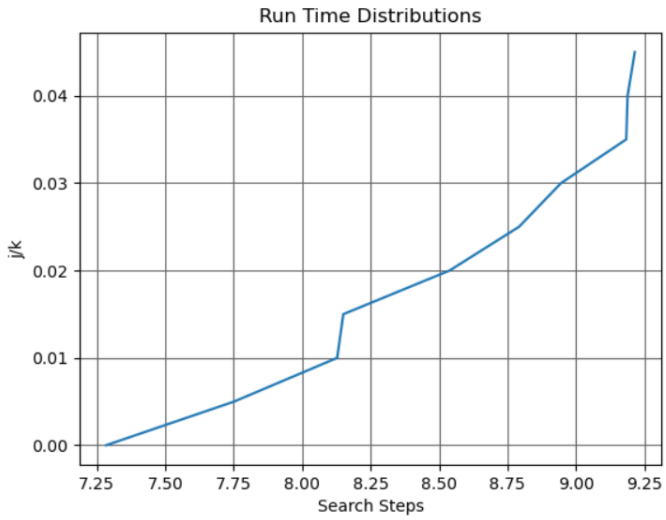
<u>Without Sideways movement allowed:</u>

Below output is obtained for 200 runs, with 200 iterations each and 10 restarts for n = 57 (50 + last digit of student id).

Please note the number of iterations here, which is more than the number of iterations we used while allowing sideways movement. This is because, to obtain more solutions with cost = 0, we need more iterations to try and push it out of the local optima.



MHO – ASSIGNMENT 2



With 200 iterations, we got the N-Queens problem solved for 10 times for below runs:

```
Total time taken for 200 runs:  1406.765625

solved_runs:  [2, 31, 34, 47, 85, 104, 119, 127, 130, 190]
```

EXPERIMENTS:

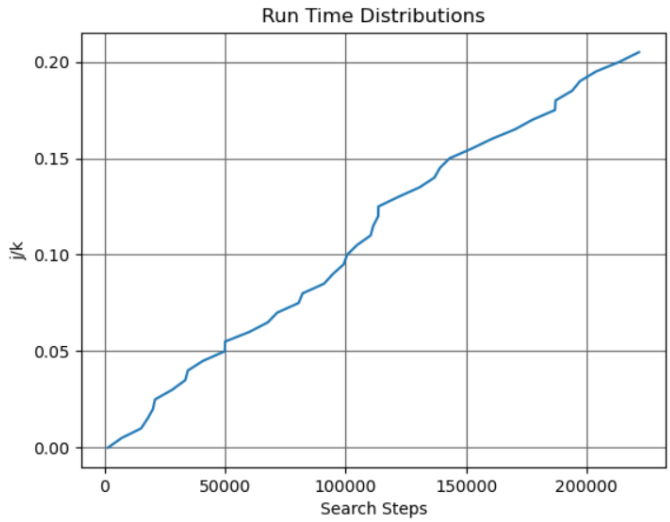
4 different experiments are carried out for the version which does not allow sideways movement, keeping number of runs fixed at **200** and below are the results obtained:

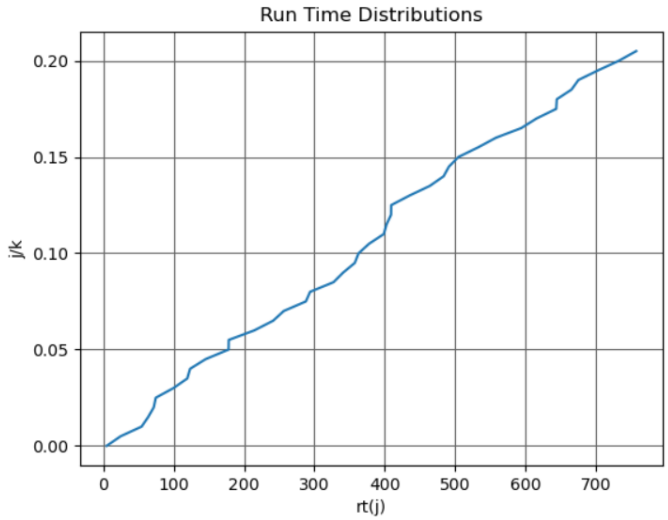
Total Runs = 200			
SL/NO	Iterations	Restarts	Solved Runs/Execution
1	200	10	10
2	100	10	7
3	100	**50**	**27**
4	200	**50**	**42**

With increasing restarts, number of solved instances are also increasing, this is because restart helps it to move out of the local optimum plateau and converge to a better solution.

Below graphs shows the RTDs with respect to search steps and also the runtime:

MHO – ASSIGNMENT 2





1. Only about 5% of solved instances are obtained in first 50000 search steps and up to 200,00 search steps (or completion of the execution), only 20% of the instances are solved (where cost = 0)
2. With respect to runtime, we can see that between 400 and 410 seconds (cumulative times for only solved instances) we are getting roughly about 2.5% of solved instances between the range 0.10 and 0.15 (More solved instances)

CONCLUSION:

1. Run time distribution helps in identifying the run time when the more instances of the problem are reaching solved state and there by helping us in making the decision on how much should we wait.
2. Indirectly this also helps in identifying the random seed value for which we are getting the problem solved in quick time, if we change the random seed value in proportionate with the number of runs and some constant.