

CS 359 Lab Assignment 4 (Red Deer Algorithm)

Tarun Gupta
180001059
April 23, 2021

1 Introduction

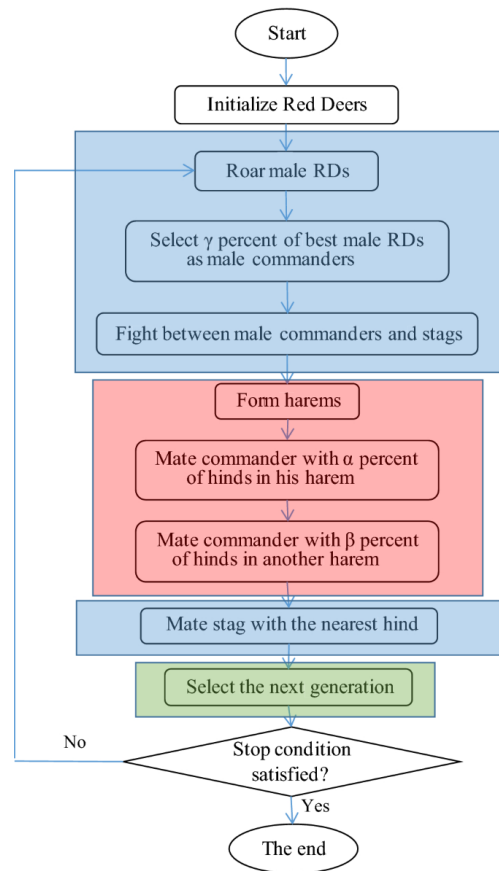
Red Deer Algorithm (RDA) is a metaheuristic inspired from an unusual mating behavior of Scottish red deer in a breeding season. The Red Deer Algorithm (RDA) starts with an initial population called red deers (RDs). They are divided into two types: hinds and male RDs. Besides, a harem is a group of female RDs. The general steps of this evolutionary algorithm are considered by the competition of male RDs to get the harem with more hinds via roaring and fighting behaviors.

Travelling Salesman Problem: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization.

In this assignment, we have to use RDA algorithm to solve the Travelling Salesman Problem.

2 Design of Genetic Algorithm

Figure 1: Flow Diagram explaining the RDA algorithm. (Fathollahi-Fard et al.)



The RDA Algorithm can be divided into following steps:

- **Create the population:** In this step we create a random population.
- **Roar male RDs:** Male red deers roar. Roaring is associated with fitness and strength. This is a mating call for hinds.
- **Selection of best male RDs:** We select γ percentage of best male RDs as male commanders.
- **Fight between male commanders and stags:** Each commander fights with stags randomly.
- **Form harems:** A harem is a group of hinds in which a male commander seized them.
- **Mate commander with α percentage of hinds in his own harem.**
- **Mate commander with β percentage of hinds in another harem.**
- **Mate stag with the nearest hind.**
- **Repeat till stop criterion satisfied.**

3 Encoding for Travelling salesman problem

For encoding scheme of TSP, **random key method** is used as defined in paper: *Lawrence V. Snyder, Mark S. Daskin, A random-key genetic algorithm for the generalized traveling salesman problem.*

In the random key method, we assign each gene a random number drawn uniformly from $[0, 1)$. To decode the chromosome, we visit the nodes in ascending order of their genes.

For example: **Random key** $\{0.42 \ 0.06 \ 0.38 \ 0.48 \ 0.81\}$, **Decodes** as $\{3 \ 1 \ 2 \ 4 \ 5\}$.

Nodes that should be early in the tour tend to evolve genes closer to 0 and those that should come later tend to evolve genes closer to 1

4 Parallelizing the Genetic Algorithm

I have used MPI for parallelizing my code. I have used collective communication calls such as MPI_Scatter, MPI_Gather and MPI_Bcast majorly for communication between processes. As an example, pseudo code for parallelization of male-roaring procedure has been show below. In this code I have used MPI_Scatter and MPI_Gather to divide and combine the global population into different sub-populations. These sub-populations are divided among different processes and each process works on its set of sub-population.

```

1      comm.Scatter(males, males_scattered, root=0)
2      # roaring of male deer
3      for i in range(local_num_males):
4          new_male = males_scattered[i].copy()
5          r1 = np.random.random() # r1 is a random number in [0, 1]
6          r2 = np.random.random() # r2 is a random number in [0, 1]
7          r3 = np.random.random() # r3 is a random number in [0, 1]
8          if r3 >= 0.5:
9              new_male += r1 * (((UB - LB) * r2) + LB)
10         else:
11             new_male -= r1 * (((UB - LB) * r2) + LB)
12
13         if obj_function(new_male, graph) < obj_function(males_scattered[i], graph):
14             males_scattered[i] = new_male
15     comm.Gather(males_scattered, males, root=0)
```

As another example, I have also presented the parallel portion of the algorithm for the fight between male commanders and stags. As it can be observed from the pseudo code, MPI_Scatter and MPI_Gather to divide and combine the global population into different sub-populations. These sub populations are divided among different processes and each process works on its set of sub-population.

```

1
2 comm.Scatter(coms, coms_scattered, root=0)
3 comm.Scatter(stags, stags_scattered, root=0)
4
5 # fight between male commanders and stags
6 for i in range(local_num_coms):
7
8     chosen_com = coms_scattered[i].copy()
9     chosen_stag = random.choice(stags_scattered)
10
11     r1 = np.random.random()
12     r2 = np.random.random()
13
14     # Eq. (6)
15     new_male_1 = (chosen_com + chosen_stag) / 2 + r1 * (((UB - LB)* r2) + LB)
16
17     # Eq. (7)
18     new_male_2 = (chosen_com + chosen_stag) / 2 - r1 * (((UB - LB)* r2) + LB)
19
20     fitness = np.zeros(4)
21     fitness[0] = obj_function(chosen_com, graph)
22     fitness[1] = obj_function(chosen_stag, graph)
23     fitness[2] = obj_function(new_male_1, graph)
24     fitness[3] = obj_function(new_male_2, graph)
25
26     bestfit = np.max(fitness)
27     if fitness[0] < fitness[1] and fitness[1] == bestfit:
28         coms_scattered[i] = chosen_stag.copy()
29     elif fitness[0] < fitness[2] and fitness[2] == bestfit:
30         coms_scattered[i] = new_male_1.copy()
31     elif fitness[0] < fitness[3] and fitness[3] == bestfit:
32         coms_scattered[i] = new_male_2.copy()
33
34 comm.Gather(coms_scattered, coms, root=0)
35 comm.Gather(stags_scattered, stags, root=0)

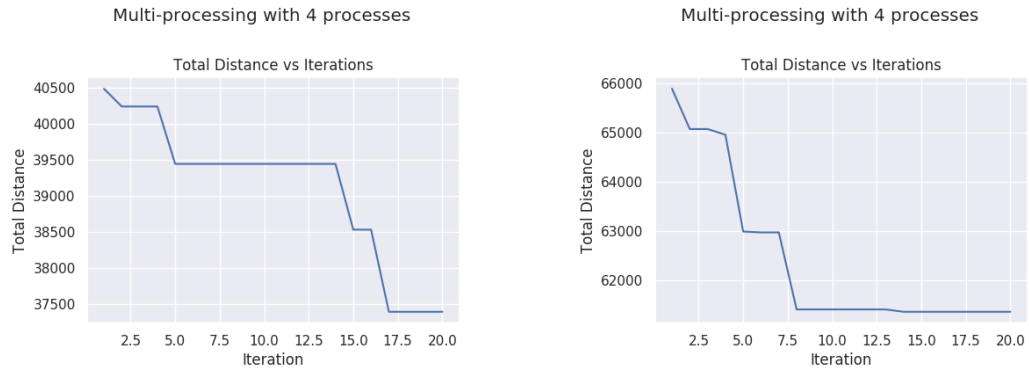
```

In a similar fashion, mating of commander with hinds in his harem and mating of stag with nearest hind has been parallelised using MPI.

5 Results obtained

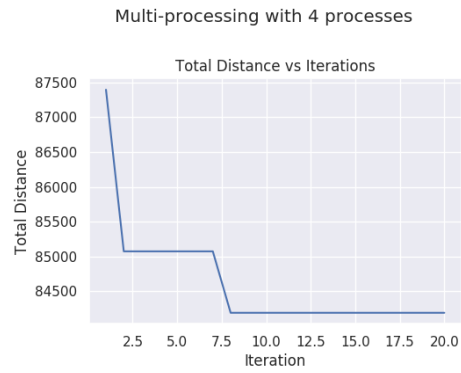
5.1 Converge Graphs

Figure 2: Convergence graphs for different number of cities, with population size = 1200 and number of iterations=20.



(a) For number of ciites = 100

(b) For number of ciites = 150



(c) For number of ciites = 200

5.2 For different number of cities:

In this set of experiments, I have changed the number of cities, while keep other parameters constant. Number of iterations = 20 and population size = 600

Table 1: For number of cities = 100

Number of Processes	Computation Time (sec)	Speed-up
1	20.14	-
2	10.34	1.95
3	9.09	2.21
4	9.71	2.07

Table 2: For number of cities = 70

Number of Processes	Computation Time (sec)	Speed-up
1	13.33	-
2	8.21	1.62
3	7.17	1.85
4	6.65	2.01

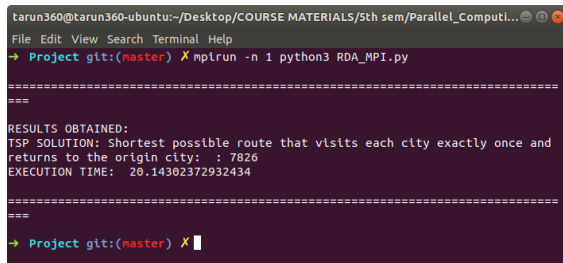
Table 3: For number of cities = 40

Number of Processes	Computation Time (sec)	Speed-up
1	9.56	-
2	5.64	1.69
3	4.41	2.16
4	4.47	2.13

As it can be observed from above results, the speed-up is usually higher when number of cities is larger. This is because for small number of cities, the synchronisation cost becomes significant. Also, for more number of cities, computation time is higher.

5.3 Terminal screenshots of results obtained for different number of cities

Figure 3: Terminal Outputs for For number of cities = 100.

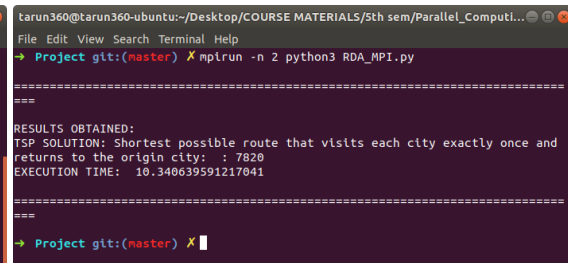


```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...
File Edit View Search Terminal Help
→ Project git:(master) X mpirun -n 1 python3 RDA_MPI.py

=====
RESULTS OBTAINED:
TSP SOLUTION: Shortest possible route that visits each city exactly once and
returns to the origin city: : 7826
EXECUTION TIME: 20.14302372932434

=====
→ Project git:(master) X
```

(a) For number of processes = 1

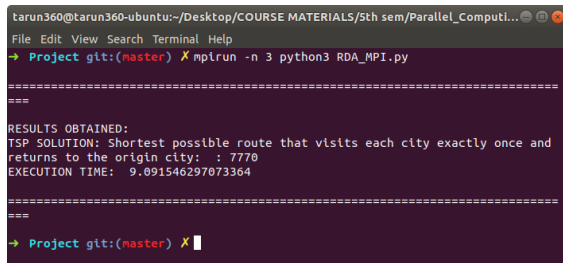


```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...
File Edit View Search Terminal Help
→ Project git:(master) X mpirun -n 2 python3 RDA_MPI.py

=====
RESULTS OBTAINED:
TSP SOLUTION: Shortest possible route that visits each city exactly once and
returns to the origin city: : 7820
EXECUTION TIME: 10.340639591217041

=====
→ Project git:(master) X
```

(b) For number of processes = 2

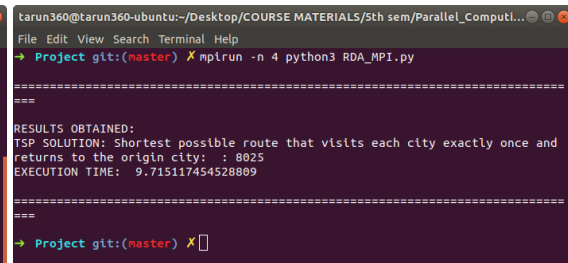


```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...
File Edit View Search Terminal Help
→ Project git:(master) X mpirun -n 3 python3 RDA_MPI.py

=====
RESULTS OBTAINED:
TSP SOLUTION: Shortest possible route that visits each city exactly once and
returns to the origin city: : 7770
EXECUTION TIME: 9.091546297073364

=====
→ Project git:(master) X
```

(c) For number of processes = 3



```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...
File Edit View Search Terminal Help
→ Project git:(master) X mpirun -n 4 python3 RDA_MPI.py

=====
RESULTS OBTAINED:
TSP SOLUTION: Shortest possible route that visits each city exactly once and
returns to the origin city: : 8025
EXECUTION TIME: 9.715117454528809

=====
→ Project git:(master) X
```

(d) For number of processes = 4

5.4 For different population size:

In this set of experiments, I have changed the total population size, while keep other parameters constant. Number of iterations = 20 and number of cities = 40

Table 4: For population size = 1800

Number of Processes	Computation Time (sec)	Speed-up
1	54.54	-
2	26.97	2.02
3	20.67	2.64
4	18.61	2.93

Table 5: For population size = 1200

Number of Processes	Computation Time (sec)	Speed-up
1	31.72	-
2	14.50	2.19
3	11.04	2.87
4	11.00	2.66

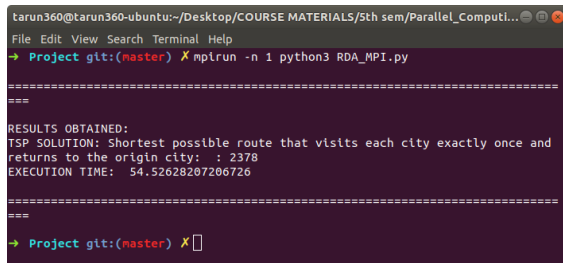
Table 6: For population size = 600

Number of Processes	Computation Time (sec)	Speed-up
1	9.34	-
2	5.74	1.62
3	4.41	2.12
4	4.42	2.11

As it can be observed from above results, the speed-up is much higher when total population size is larger. This is because for small population size, the synchronisation cost becomes significant.

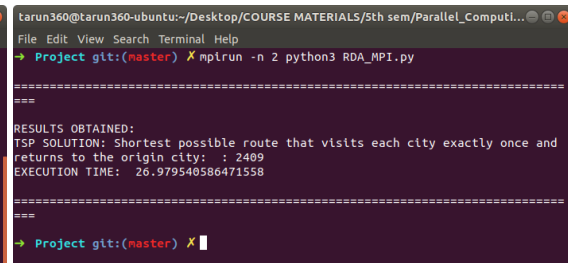
5.5 Terminal screenshots for results obtained for different population size.

Figure 4: Terminal Outputs for For number of cities = 40.



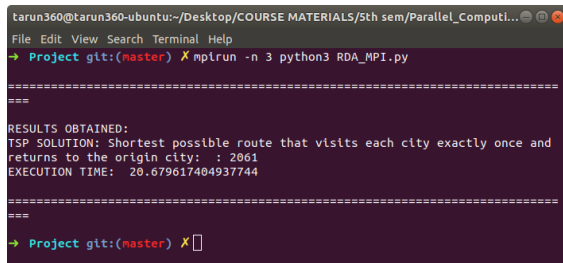
```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...  
File Edit View Search Terminal Help  
→ Project git:(master) X mpirun -n 1 python3 RDA_MPI.py  
=====  
RESULTS OBTAINED:  
TSP SOLUTION: Shortest possible route that visits each city exactly once and  
returns to the origin city: : 2378  
EXECUTION TIME: 54.52628207206726  
=====  
→ Project git:(master) X
```

(a) For number of processes = 1



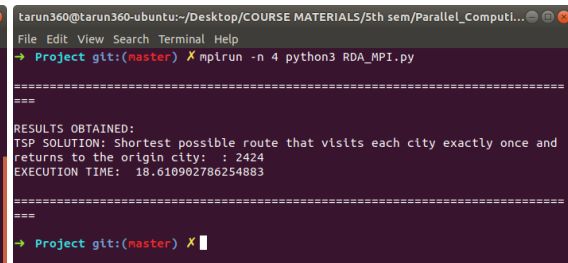
```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...  
File Edit View Search Terminal Help  
→ Project git:(master) X mpirun -n 2 python3 RDA_MPI.py  
=====  
RESULTS OBTAINED:  
TSP SOLUTION: Shortest possible route that visits each city exactly once and  
returns to the origin city: : 2409  
EXECUTION TIME: 26.979540586471558  
=====  
→ Project git:(master) X
```

(b) For number of processes = 2



```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...  
File Edit View Search Terminal Help  
→ Project git:(master) X mpirun -n 3 python3 RDA_MPI.py  
=====  
RESULTS OBTAINED:  
TSP SOLUTION: Shortest possible route that visits each city exactly once and  
returns to the origin city: : 2061  
EXECUTION TIME: 20.679617404937744  
=====  
→ Project git:(master) X
```

(c) For number of processes = 3



```
tarun360@tarun360-ubuntu:~/Desktop/COURSE MATERIALS/5th sem/Parallel_Computi...  
File Edit View Search Terminal Help  
→ Project git:(master) X mpirun -n 4 python3 RDA_MPI.py  
=====  
RESULTS OBTAINED:  
TSP SOLUTION: Shortest possible route that visits each city exactly once and  
returns to the origin city: : 2424  
EXECUTION TIME: 18.610902786254883  
=====  
→ Project git:(master) X
```

(d) For number of processes = 4