Optimization of Transportation Problem

AE 755

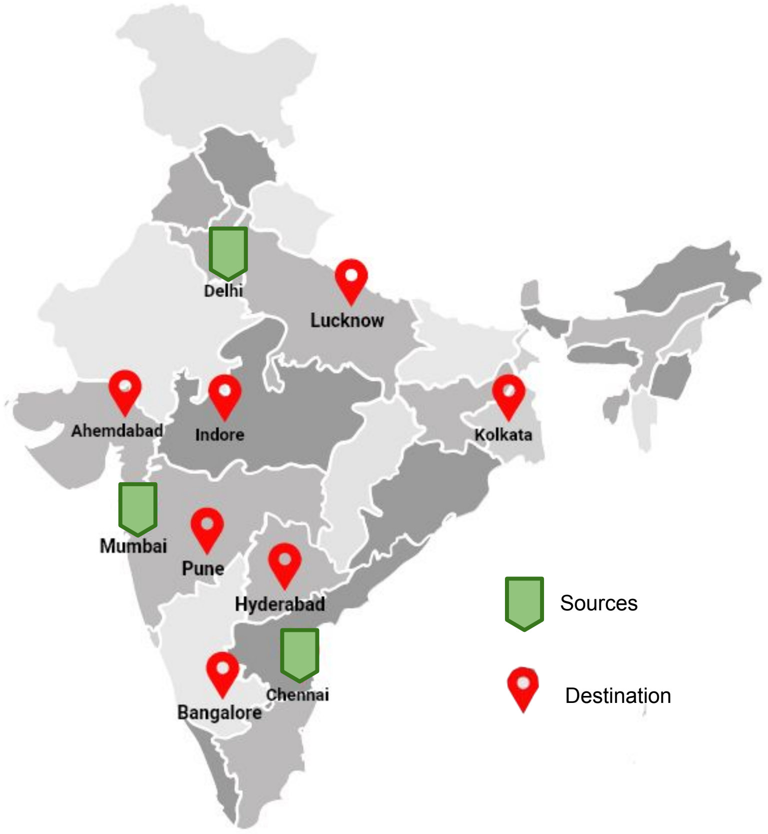
Anand Karunan - 180010007
Anjali Dhananjay Katake - 180010008
Chinmay Choudhari - 180010020
Pranav K Das - 180010043
Prathmesh More - 170010016
Prince Sharma - 170010017

Department of Aerospace Engineering,
IIT Bombay

Introduction

The transportation problem is a special type of linear programming problem where the objective is to minimize transportation cost of a given commodity from a number of sources or origins to a number of destinations.

Eg. Real life transportation problems such as oxygen cylinder transport in this pandemic situation



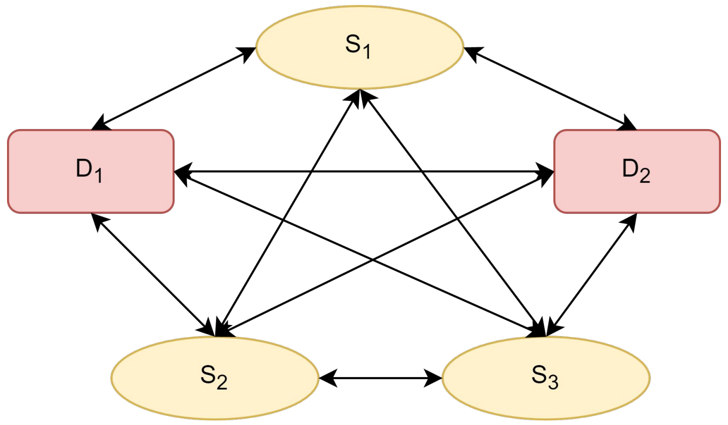
Problem Statement

In a region, there are a limited number of water reservoirs which meet the water demands of the cities in the region. The amount of water transported from each of these water reservoirs to each city in the region needs to be optimised such that the water demands of each city is met and the cost of transporting the water is minimized.

Model Description and Parameters

Let the water supply points be denoted as S_i ; i = 1,2,3 .. m and demand points as D_j ; j = 1,2,3 .. n

Parameters :
● Cost of transporting unit volume of water in each pipeline. C_{ij} = cost for transporting from node i to node j, where $i \neq j$
● Locations of water reservoirs and cities with respect to each other
● Water demand of each city, D_k,where k = 1,2,3
● Water capacity of each reservoir, S_l,where l = 1,2



Advanced transportation model

Objective Function And Constraints

Objective Function :

The total cost of transporting water in the entire regional water pipeline network.

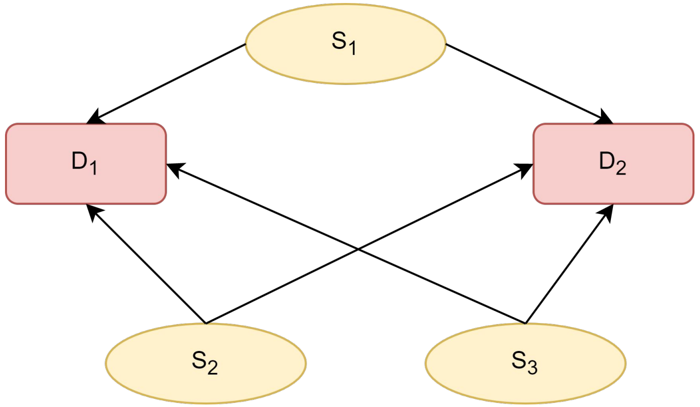
$J(D_k,S_l,V_{ij},C_{ij}) = \sum\sum V_{ij}C_{ij}$, where $i \neq j$ and $i,j=1,2,...,5$

Constraints :

- C_{ij} and V_{ij} are non-negative real values
- Net demand is always less than or equal to net capacity of water reservoirs, $\sum D_k - \sum S_l \leq 0$
- $V_{ij} - \alpha_{ij} \leq 0$, total water transported in each pipeline has some upper limit $\alpha_{ij} > 0$

Assumptions

- Water transport is taking place only between source and demand nodes
- The amount of water transported is a real-valued number
- Total Demand = Total Supply, i.e, balanced transport.
- No transport with the outer world other than the input network
- The cost of transportation only depends on the distance between the nodes.



Simplified linear transportation model used for GA and Simplex

Algorithms Used

1. Genetic Algorithm
2. Simplex Big-’M’ Method
3. Karmarkar Algorithm

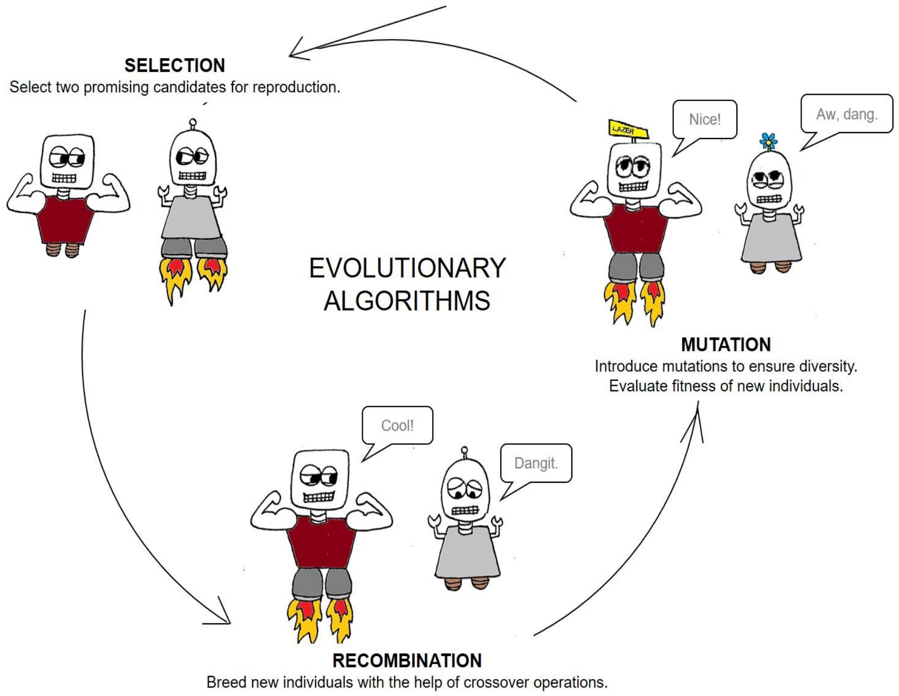
Genetic Algorithm

Introduction

Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics and is subset of a much larger branch of computation known as Evolutionary Computation.

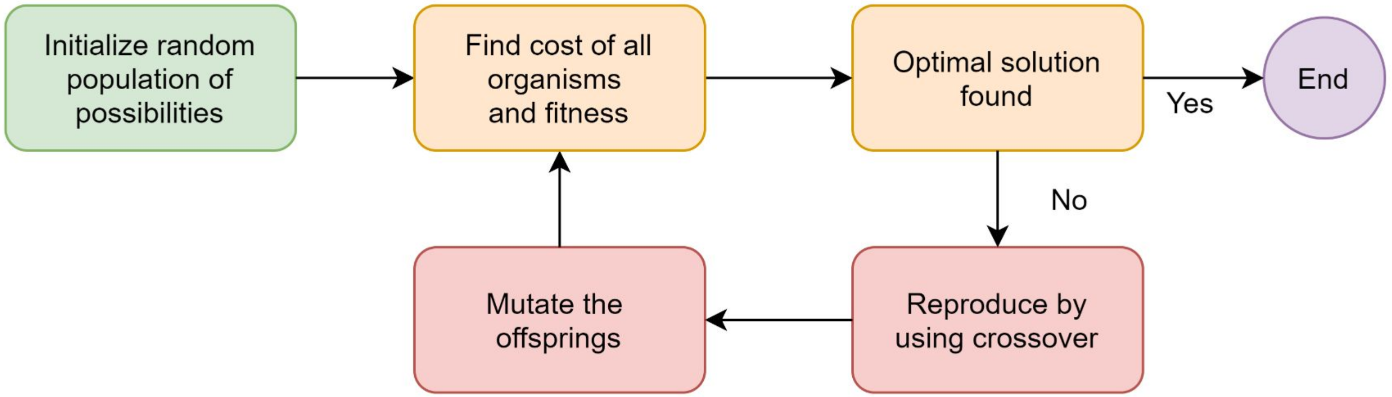
The algorithm mainly consists of three steps:

- Selection
- Crossover
- Mutation


SELECTION
Select two promising candidates for reproduction.
EVOLUTIONARY ALGORITHMS
LAZER
Nice!
Aw, dang.
MUTATION
Introduce mutations to ensure diversity.
Evaluate fitness of new individuals.
Cool!
Dangit.
RECOMBINATION
Breed new individuals with the help of crossover operations.

Introduction

Flow Chart of Genetic Algorithm


Initialize random population of possibilities
Find cost of all organisms and fitness
Optimal solution found
Yes
End
No
Reproduce by using crossover
Mutate the offsprings

Encoding for GA

There are many ways of encoding:

Encoding Methods	Description
Binary encoding:	Representing a gene in terms of bits (0s and 1s).
Real value encoding:	Representing a gene in terms of values or symbols or string.
Permutation (or Order) encoding:	Representing a sequence of elements
Tree encoding:	Representing in the form of a tree of objects

Selection methods for GA

Rank Selection
Rank Selection sorts the population first according to fitness value and ranks them. Then every chromosome is allocated selection probability with respect to its rank and are selected as per their selection probability. This type is mainly helpful in preventing too quick convergence but populations must be sort on every cycle

Tournament Selection (TOS)
Strategy to select the fittest parent in the list. A tournament is conducted on a certain amount of selected parents and only the fittest candidate will move to the next generation. When the tournament size is changed we can easily adjust selection pressure. The time complexity is very less in TOS and no fitness or sorting required but there is no guarantee that best solution is obtained.

Truncation Selection (TRS)
The parents or the candidates are sorted according to their fitness. Then, only a certain portion p of the fittest individuals are selected and reproduced np times. It is less used in practice than other techniques, except for very large population.

Elitism Selection
In elitism selection, the candidates are arranged in the decreasing order and the selection of best two chromosomes are done and transferred directly to the next generation without any modification. This process improves the algorithm because it passes the best individuals to the next generation.

Roulette Wheel Selection (RWS)
The role of roulette wheel is the area covered by the entire chromosome in a population as per the fitness value. A proportion of the wheel is assigned to each of the possible selections based on their fitness value. Then a random selection is made similar to how the roulette wheel is rotated. The best thing about this method is all the candidates get selected. Time complexity is $O(n^2)$.

Stochastic Universal Sampling (SUS)
It is similar to roulette wheel selection, but instead of using a single selection point and turning the roulette wheel again and again until all needed individuals have been selected, we turn the wheel only once and use multiple selection points that are equally spaced around the wheel. This way, all the individuals are chosen at the same time.

Selection methods for GA

Rank Selection
Rank Selection sorts the population first according to fitness value and ranks them. Then every chromosome is allocated selection probability with respect to its rank and are selected as per their selection probability. This type is mainly helpful in preventing too quick convergence but populations must be sort on every cycle

Tournament Selection (TOS)
Strategy to select the fittest parent in the list. A tournament is conducted on a certain amount of selected parents and only the fittest candidate will move to the next generation. When the tournament size is changed we can easily adjust selection pressure. The time complexity is very less in TOS and no fitness or sorting required but there is no guarantee that best solution is obtained.

Truncation Selection (TRS)
The parents or the candidates are sorted according to their fitness. Then, only a certain portion p of the fittest individuals are selected and reproduced np times. It is less used in practice than other techniques, except for very large population.

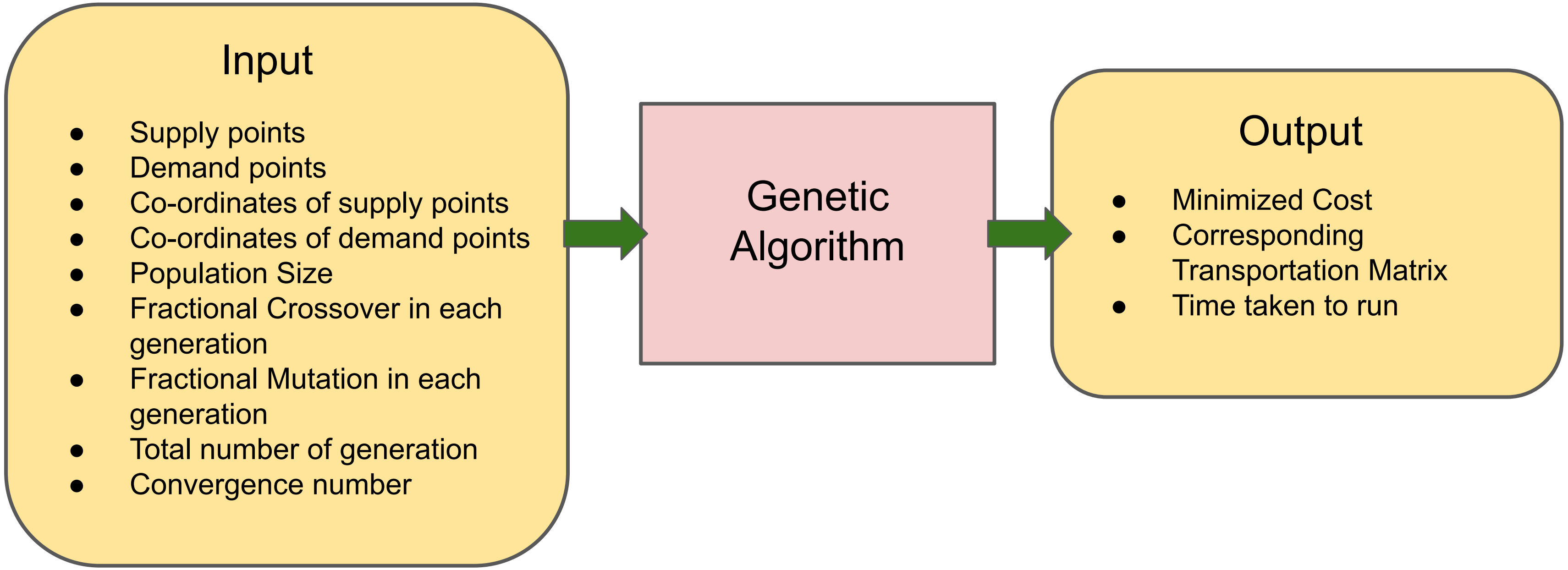
Elitism Selection
In elitism selection, the candidates are arranged in the decreasing order and the selection of best two chromosomes are done and transferred directly to the next generation without any modification. This process improves the algorithm because it passes the best individuals to the next generation.

Roulette Wheel Selection (RWS)
The role of roulette wheel is the area covered by the entire chromosome in a population as per the fitness value. A proportion of the wheel is assigned to each of the possible selections based on their fitness value. Then a random selection is made similar to how the roulette wheel is rotated. The best thing about this method is all the candidates get selected. Time complexity is $O(n^2)$.

APPROVED

Stochastic Universal Sampling (SUS)
It is similar to roulette wheel selection, but instead of using a single selection point and turning the roulette wheel again and again until all needed individuals have been selected, we turn the wheel only once and use multiple selection points that are equally spaced around the wheel. This way, all the individuals are chosen at the same time.

Our implementation



Input

- Supply points
- Demand points
- Co-ordinates of supply points
- Co-ordinates of demand points
- Population Size
- Fractional Crossover in each generation
- Fractional Mutation in each generation
- Total number of generation
- Convergence number

Genetic Algorithm

Output

- Minimized Cost
- Corresponding Transportation Matrix
- Time taken to run

Some methods used

Genetic Operators and Facilitators

Roulette Wheel Based Selection

1. Calculate the total fitness of the population
   $$F = \sum_{i=1}^{population} eval(X_i)$$
2. Calculate selection probability p_i for each chromosome,
   $$p_i = \frac{F - eval(X_i)}{F \times (population - 1)}$$
3. Calculate the cumulative probability q_i for each chromosome
   $$q_i = \sum_{j=1}^{i} p_j$$
4. Generate a random number r from the range [0,1]
5. If $q_{i-1} < r < q_i$, then chromosome X_i is selected

Crossover

1. Use Roulette's wheel to select parents X_1 and X_2
2. Create D and R matrix such that
   $d_{ij} = [(x_{ij}^1 + x_{ij}^2)/2)]$ and $r_{ij} = (x_{ij}^1 + x_{ij}^2) \bmod 2$
3. Split R into R_1 and R_2 such that $R = R_1 + R_2$ and
   $$\sum_{i=1}^{len(source)} r_{ij}^1 = \sum_{i=1}^{len(source)} r_{ij}^2 = \tfrac{1}{2} \sum_{i=1}^{len(source)} r_{ij},\ i = 1, 2 .. len(dest)$$
   $$\sum_{i=1}^{len(dest)} r_{ij}^1 = \sum_{i=1}^{len(dest)} r_{ij}^2 = \tfrac{1}{2} \sum_{i=1}^{len(dest)} r_{ij},\ i = 1, 2 .. len(source)$$
4. Offspring1 = D+R1 and Offspring2 = D+R2
5. Remove copies from the list of offsprings

Mutation

1. Use Roulette's wheel to select parents X_1 and X_2
2. Select random number of rows and columns from parent
3. To obtain mutated offspring, populate these rows and columns randomly such that row and column sum remains same
4. Remove copies from the list of offsprings

Elitism

1. Combine the list of species obtained after crossover, mutation and mix them with the parents
2. Sort the list based on the values of their cost in ascending order
3. Choose the first population number of species as parents for the next generation

Pseudo Code for our GA implementation

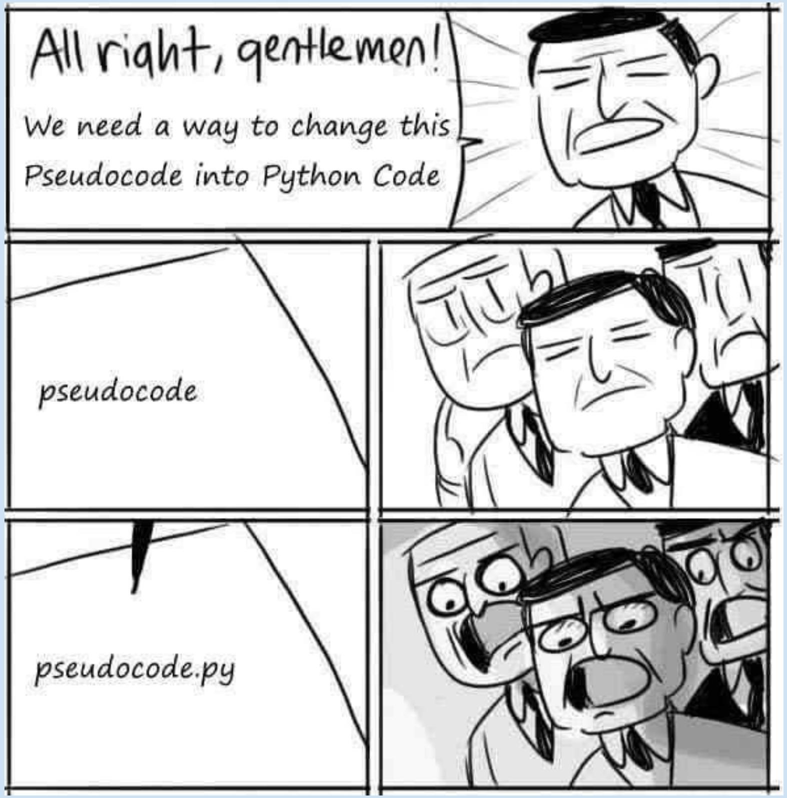
```
1.  Input total number of supply and demand points, supply and demand coordinates and parameters like population, crossover and mutation fraction, num_gen and convergence number
2.  Using coordinates of supply and demand points
    a.  Distance between a selected source and destination calculated
    b.  Cost deduced using distance function
3.  Initialize parent generation of size population given in input.
    a.  Sort parents based on their cost
    b.  Remove copies from them based on the cost

5.  for i in range(num_gen):
    a.  Create offsprings by crossover between parents until we get crossover_fraction*population number of offsprings.
    b.  Create offsprings by mutation of parentsuntil we get mutation_fraction*population number of offsprings
    c.  Create a temporary list which contains all parents, all crossover offsprings and all mutated offsprings.
    d.  Sort this list based on the cost of it.
    e.  Now assign first population number of species from the temporary list to parents
    f.  If the number of the most minimum value still goes more than the convergence number, the final answer is found. Exit loop. Else go on the loop.
```

Output : **Minimized cost**
Transportation matrix corresponding to minimised cost
Time taken

Problems faced while code development

Issue	How we resolved it
Finding different combinations of matrix given the row and column sum	Modified the method given for zeroth generation initialization to use
Row sum and column sum of offsprings violates the constraints resulting in extremely low cost functions	Added extra checks while creating offsprings so that such offsprings are not included.
Optimization converges too quickly without undergoing much generations due to accumulate of copies of members over generations.	Introduced a function to remove copies of members with same cost in a generation
Sorting the combined list of offsprings and parents to perform elitism gives error	Wrote an improved version of bubble sort which resolves this issue

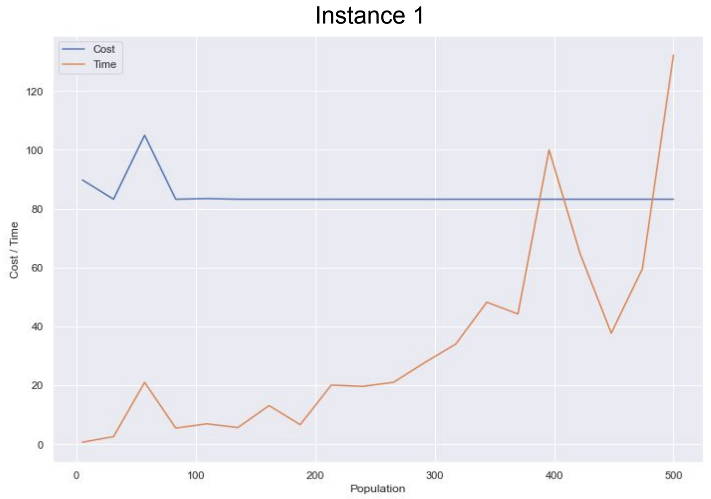


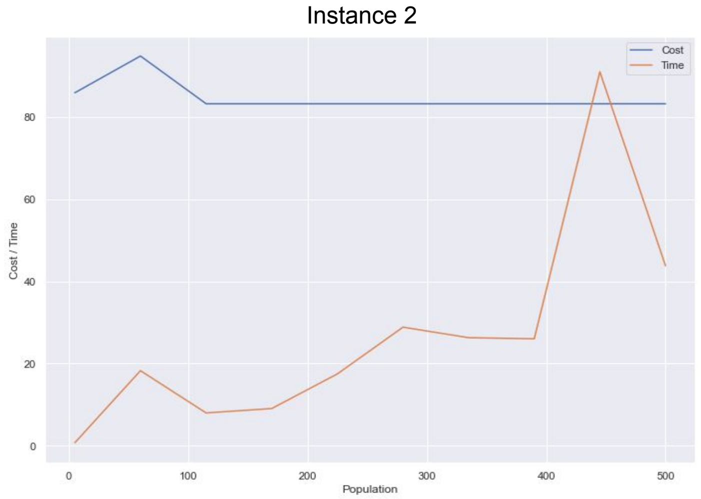
Tuning parameters of GA

There are mainly 5 tunable parameters in our implementation :

Parameter	Used to control
● Population	Number of members in one generation
● Crossover fraction	Number of parents selected for crossover
● Mutation fraction	Number of parents selected for mutation
● Number of generations	Max number of generations the GA runs
● Convergence number	Number of times a minimized cost should repeat itself to identify that the GA has found the optimum solution.

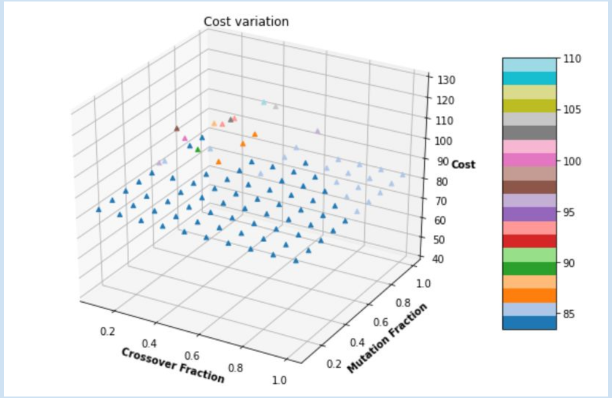
Tuning parameters of GA : Population

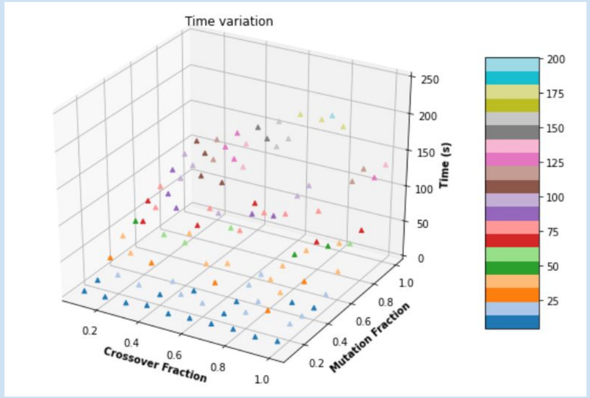


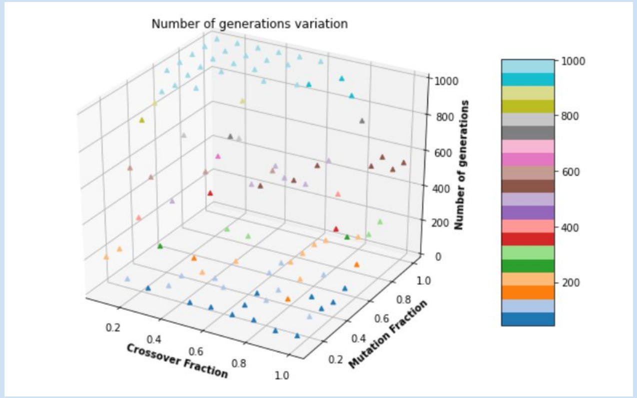


Therefore, a population size of 100-150 gives us an accurate result along with reasonably fast convergence time. This is verified

Tuning parameters of GA : Crossover and Mutation fractions







Try to chose values of Crossover and Mutation fraction such that the following function is minimum

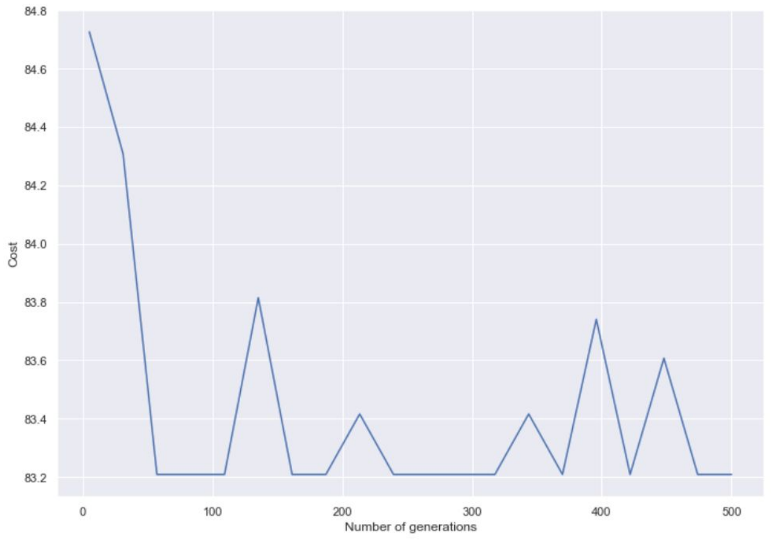
$$\text{Weighted function}_{\text{minimize}} = (100 * \text{cost}) + (50 * \text{time}) + (200 * \text{num\_generation})$$

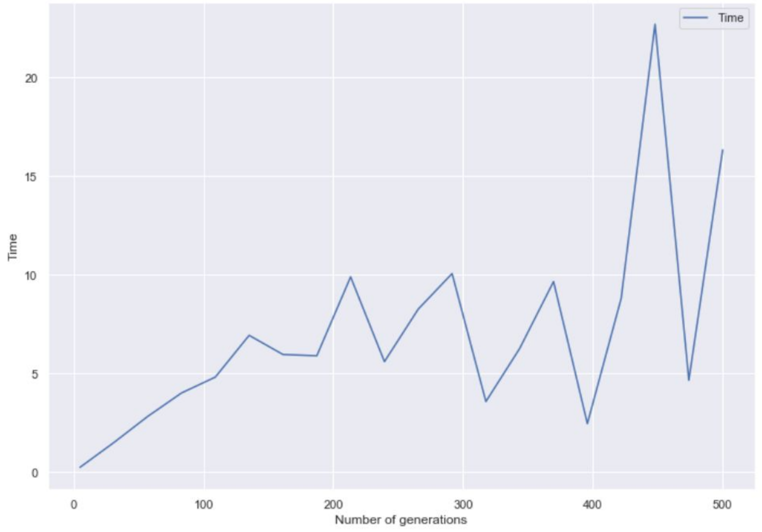
Tuning parameters of GA : Crossover and Mutation fractions

These are the candidate crossover and mutation fractions that we have :

Cross_fraction , Mut_fraction	Weighted function value
[0.9 , 0.1]	17460.34
[1.0 , 0.1]	18566.64
[0.7 , 0.2]	21716.22
[0.9 , 0.4]	21901.44
[1.0 , 0.4]	22133.88

Tuning parameters of GA : Number of generations and Convergence number





From graph, we can conclude that the effect of number of generations on the convergence is quite random and a high value is preferred so that the chances of finding global optima increases

For convergence number, through excessive testing, we were able to arrive at a value of 25-30.

GA benchmarking

Test example 1 :

```
source = [8,4,12,6]
dest = [3,5,10,7,5]
supply_coords = [[7., 6.],[2., 7.],[1., 6.],[6., 8.]]
demand_coords = [[3., 9.],[4., 6.],[5., 7.],[5., 3.],[2., 9.]]
```

Results from GA :

```
Optimized cost: 83.20907295926658
Solution matrix:  [[0. 0. 4. 4. 0.]
                   [3. 0. 0. 0. 1.]
                   [0. 5. 0. 3. 4.]
                   [0. 0. 6. 0. 0.]]
Time taken: 8.504777193069458 sec
```

Results from PULP :

```
Optimized cost:  83.20907295926656
Solution matrix:  [[0. 0. 4. 4. 0.]
                   [3. 0. 0. 0. 1.]
                   [0. 5. 0. 3. 4.]
                   [0. 0. 6. 0. 0.]]
Time taken =  0.022297859191894531
```

GA benchmarking

Test example 2 :

Note : Here the co-ordinates are randomly initialized

```
source1 = [13,12,10,5,3,8,1,2,17]
dest1 = [9, 16,8,9,2,5,18,2,2]
supply_coords1 = []
demand_coords1 = []
for node_s in range(len(source1)):
        supply_coords1.append([50*random.random(),50*random.random()])
for node_d in range(len(dest1)):
        demand_coords1.append([50*random.random(),50*random.random()])
```

GA benchmarking

Results from GA :

```
Optimized cost: 670.5537978024969
Solution matrix:  [[ 0.  0.  0.  0.  0.  0. 13.  0.  0.]
                   [ 0. 12.  0.  0.  0.  0.  0.  0.  0.]
                   [ 0.  0.  5.  0.  0.  5.  0.  0.  0.]
                   [ 0.  0.  0.  0.  0.  0.  1.  2.  2.]
                   [ 0.  0.  2.  0.  1.  0.  0.  0.  0.]
                   [ 0.  4.  0.  0.  0.  0.  4.  0.  0.]
                   [ 0.  0.  0.  0.  1.  0.  0.  0.  0.]
                   [ 0.  0.  1.  1.  0.  0.  0.  0.  0.]
                   [ 9.  0.  0.  8.  0.  0.  0.  0.  0.]]
 Time taken: 59.4437689781189 sec
```

Results from PULP :

```
Optimized cost: 669.0626832445316
Solution matrix:  [[ 0.  0.  0.  0.  0.  0. 13.  0.  0.]
                   [ 0. 12.  0.  0.  0.  0.  0.  0.  0.]
                   [ 0.  0.  4.  1.  0.  5.  0.  0.  0.]
                   [ 0.  0.  0.  0.  0.  0.  1.  2.  2.]
                   [ 0.  0.  2.  0.  1.  0.  0.  0.  0.]
                   [ 0.  4.  0.  0.  0.  0.  4.  0.  0.]
                   [ 0.  0.  0.  0.  1.  0.  0.  0.  0.]
                   [ 0.  0.  2.  0.  0.  0.  0.  0.  0.]
                   [ 9.  0.  0.  8.  0.  0.  0.  0.  0.]]
Time taken =  0.04002571105957031
```

GA benchmarking

Test example 3:

```
source2 = list(range(1,25))
dest2 = list(range(1,25))
supply_coords2 = []
demand_coords2 = []
for node_s in range(len(source2)):
        supply_coords2.append([50*random.random(),50*random.random()])
for node_d in range(len(dest2)):
        demand_coords2.append([50*random.random(),50*random.random()])
```

Note : Here the sources,demand and destination co-ordinates are randomly initialized

GA benchmarking

Results from GA :

```
Optimized cost: 6254.20919718621
Solution matrix: [[ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.]
                  [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  4.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.]
                  [ 0.  2.  0.  0.  0.  3.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
                  [ 0.  0.  0.  4.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  3.  0.  2.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 10.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 11.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 13.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  5.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  2.  0.  0.  0.  2.  0.  0.  0.  0.  0.  0.  0.  0.  0. 11.  0.  0.  0.  0.]
                  [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  0.  0. 13.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  5.  0.  0.  1.  0.  0.  0.  0. 11.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  2.  1.  0.  0.  0.  0.  4.  0.  6.  0.  0.  0.  3.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 19.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  0.  0. 16.  0.  0.  0.  0.  0.  2.  0.  0.]
                  [ 1.  0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  7.  6.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  0.  0.  0.  0. 20.  0.  0.]
                  [ 0.  0.  2.  0.  0.  0.  0.  7.  4.  0.  5.  0.  0.  0.  0.  0.  0.  0.  0.  3.  2.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  4.  2.  5.  0.  0.  0.  0. 10.  0.  0.  0.  0.  0.  3.  0.]]
 Time taken: 372.4835443496704 sec
```

Results from PULP :

```
Optimized cost: 2996.8764750970413
Solution matrix: [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  3.  0.  0.]
                  [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  3.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  5.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 10.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  3.  0.  0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  3.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 12.  0.  2.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 15.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  8.  0.  0.  0.  8.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0. 12.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 11.  0.  7.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 19.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 13.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 21.  0.]
                  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  2.  0. 20.  0.  0.  0.  0.]
                  [ 0.  2.  0.  0.  0.  0.  7.  8.  0.  0.  6.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
                  [ 0.  0.  0.  4.  5.  0.  0.  0.  0.  0.  0.  0.  0.  0.  3.  0.  0.  0.  9.  0.  0.  0.  0.  3.]]
Time taken =  0.15760064125061035
```

Reason for non-convergence could be due to less number of generations available for finding better results

Observations

- Worst case Time complexity for the algorithm is approximately: O(no. of generation*population^2)
- We are using real valued encoding in our implementation
- Roulette wheel selection is efficient to select the candidates and elitism selection method is used to size the candidate matrix in every generation.
- Ideal value of population number is found to be between 100-150
- For crossover and mutation fraction, a high value of crossover fraction and a low value of mutation fraction seems ideal as crossover brings in best characteristics of the parents and mutation brings in diversity
- The effect of number of generations is quite random and hence a high value (like 1000) is preferable so that the chances of convergence within this much generations is high. For examples with high number of supply and demand points like test example 3, a very high value of 3000+ generations is preferable.
- For convergence number, through excessive testing, we were able to arrive at a value of 25-30.
- Implementations of GA take slightly higher time than the benchmarking algorithms for small number of inputs and the time taken increases with increasing number of inputs generally.

Scope for improvement and References

Scope for improvement

- Instead of bubble sort ($O(n^2)$), try quick_sort or other sorting algorithms which have better worst case time complexity
- Try to implement a more efficient method to remove copies of candidates as a sorted array is also a requirement for removing copies in our implementation
- Stochastic Universal Sampling (SUS) can also be used for selection process.
- For a large number of population, truncation selection is more efficient

Major References :

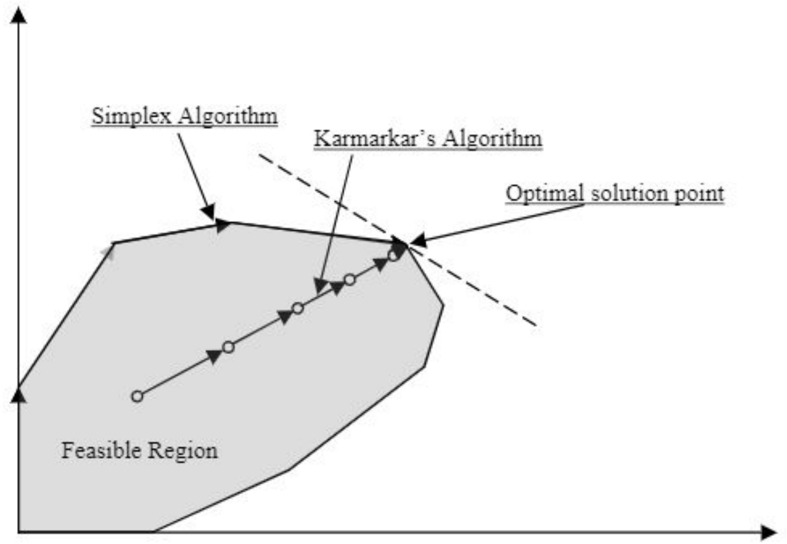
- Page 280, Genetic Algorithms and Engineering Design by Mitsuo Gen and Runwei Chen
- A genetic algorithm for the generalised transportation problem by W. Ho and P. Ji*
- Selection Methods for Genetic Algorithms by Khalid Jebari, Mohammed Madiafi
- Wikipedia

Karmarkar’s Algorithm

Interior Point Methods for Linear Programming

Karmarkar's algorithm is an algorithm introduced by Narendra Karmarkar in 1984 for solving linear programming problems. It was the first reasonably efficient algorithm that solves these problems in polynomial time.

- Points generated are in the “interior" of the feasible region
- Polynomial time and efficient
- Some interior points methods:
  - Affine Scaling
  - Karmarkar’s Method



Karmarkar's Method

Assumptions:

- The problem is in homogeneous form:
- Optimum objective function value is 0

Idea:

1. Use projective transformation to move an interior point to the centre of the feasible region
2. Move along projected steepest descent direction

$$\begin{aligned} \min \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \mathbf{0} \\ & \mathbf{1}^T \boldsymbol{x} = 1 \\ & \boldsymbol{x} \geq \mathbf{0} \end{aligned}$$

Karmarkar’s Method

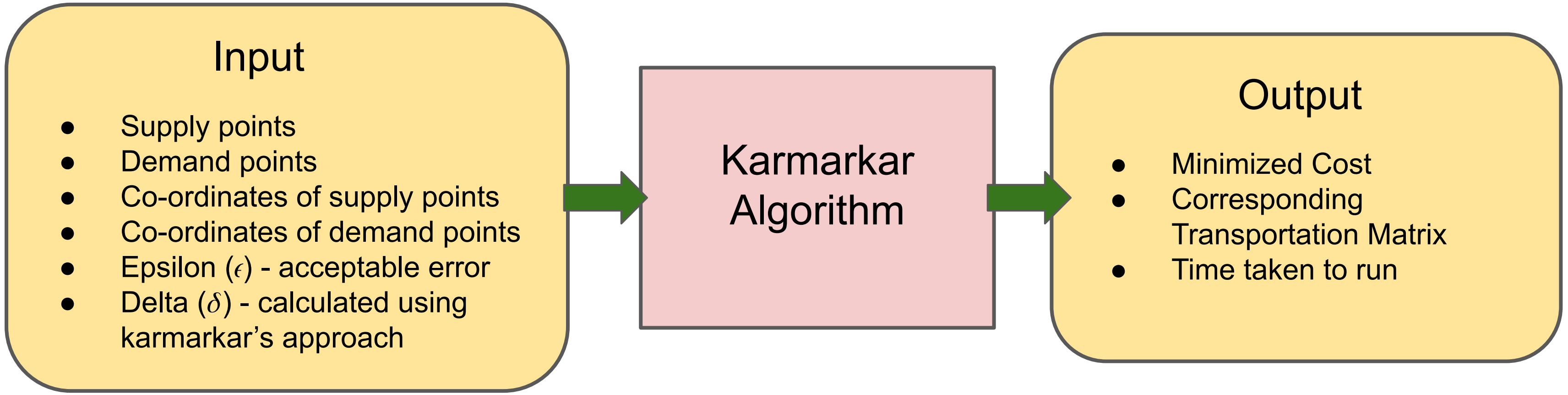
Using the transformation,

$$\boldsymbol{x} = \frac{X^k \boldsymbol{y}}{\mathbf{1}^T X^k \boldsymbol{y}}$$

$$\begin{array}{ll} \min & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} & A\boldsymbol{x} = \mathbf{0} \\ & \mathbf{1}^T\boldsymbol{x} = 1 \\ & \boldsymbol{x} \geq \mathbf{0} \end{array} \equiv \begin{array}{ll} \min & \dfrac{\boldsymbol{c}^T X^k \boldsymbol{y}}{\mathbf{1}^T X^k \boldsymbol{y}} \\ \text{s.t.} & AX^k\boldsymbol{y} = \mathbf{0} \\ & \mathbf{1}^T\boldsymbol{y} = 1 \\ & \boldsymbol{y} \geq \mathbf{0} \end{array} \equiv \begin{array}{ll} \min & \boldsymbol{c}^T X^k \boldsymbol{y} \\ \text{s.t.} & AX^k\boldsymbol{y} = \mathbf{0} \\ & \mathbf{1}^T\boldsymbol{y} = 1 \\ & \boldsymbol{y} \geq \mathbf{0} \end{array}$$

- Equivalence based on the assumption: Optimal objective function value is 0 and $\mathbf{1}^T X^k \boldsymbol{y} > 0$

Karmarkar Algorithm



Input

- Supply points
- Demand points
- Co-ordinates of supply points
- Co-ordinates of demand points
- Epsilon (ϵ) - acceptable error
- Delta (δ) - calculated using karmarkar's approach

Karmarkar Algorithm

Output

- Minimized Cost
- Corresponding Transportation Matrix
- Time taken to run

Pseudo Code for our Karmarkar Algorithm implementation

```
1. Input total number of supply and demand points, supply quantity and
   demand required for each point
2. Input coordinates of supply and demand points
3. Build the cost matrix using distance:
     a. distance function
          Returns the distance between a selected source and destination
     b. cost function
          for i in range(len(source)):
             for j in range(len(dest)):
                cost deduced using distance function
4. Convert into homogeneous LP satisfying karmarkar's assumptions A,c,$\epsilon$
```

```
5. Set $k := 0,\ x^k = (1/n) * 1$
6. $X^k = diag(x^k)$
7. while $c^T x^k > \epsilon$:
     (a) Find the projected steepest descent direction $d^k$
     (b) $y^{k+1} = \frac{1}{n}1 + \frac{\delta}{\sqrt{n(n-1)}} \frac{d^k}{\|d^k\|} \quad (\delta = \frac{1}{3})$
     (c) $x^{k+1} = T^{-1}(y^{k+1})$
     (d) $X^{k+1} = diag(x^{k+1})$
     (e) $k := k + 1$
     Endwhile
Output : Optimal x - $x^* = x^k$
     Cost function ( minimised cost at $x^*$) = $z = c^T * x$
```

https://youtu.be/LWXXhBIlj0o
https://nptel.ac.in/content/storage2/courses/105108127/pdf/Module_3/M3L6_LN.pdf

Observations

$$\begin{aligned} &\text{Minimize} && Z = 2x_2 - x_3 \\ &\text{subject to:} && x_1 - 2x_2 + x_3 = 0 \\ &&& x_1 + x_2 + x_3 = 1 \\ &&& x_1, x_2, x_3 \geq 0 \end{aligned}$$

Thus, $n = 3$, $\mathbf{C} = \begin{bmatrix} 0 \\ 2 \\ -1 \end{bmatrix}$, $\mathbf{A} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$, $\mathbf{X}_0 = \begin{bmatrix} 1/3 \\ 1/3 \\ \vdots \\ 1/3 \end{bmatrix}$, $r = \dfrac{1}{\sqrt{n(n-1)}} = \dfrac{1}{\sqrt{3(3-1)}} = \dfrac{1}{\sqrt{6}}$,

$$\alpha = \frac{(n-1)}{3n} = \frac{(3-1)}{3 \times 3} = \frac{2}{9}.$$

Observations

Iteration #	Theoretical Output	Our Output

0	X = [0.2692 0.3333 0.3974]
	Z = 0.2692
		X = [0.2691833 0.33333333 0.39748336]
		Z = 0.2691833

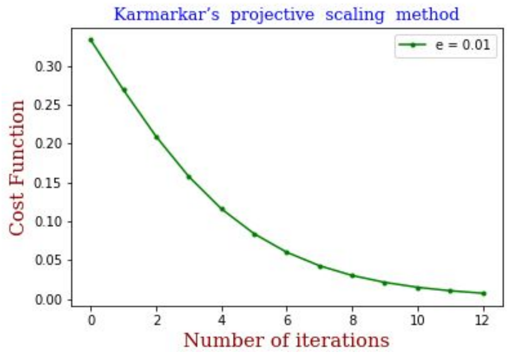
1	X = [0.2092 0.3334 0.4574]
	Z = 0.2094
		X = [0.20925762 0.33333333 0.45740904]
		Z = 0.20925762

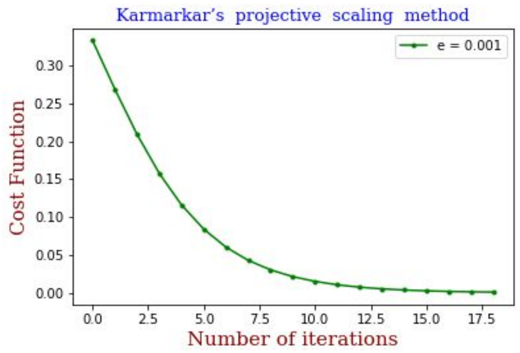
.	.	.
.	.	.

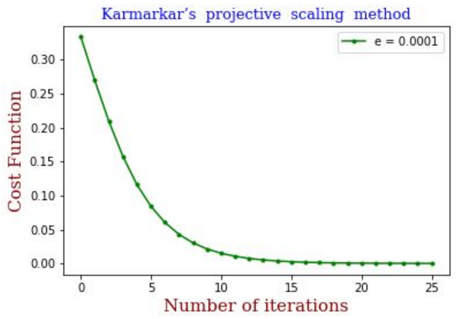
n(18)	X = [0.000873 0.3333 0.66578]
	Z = 0.000873
		X = [0.00087479 0.33333333 0.66579188]
		Z = 0.00087479

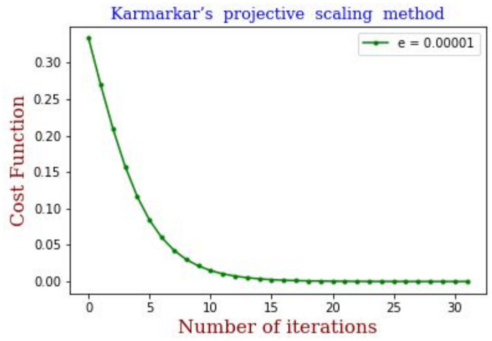
Cost Function Analysis

The variation of cost function with number of iteration for different stopping conditions is as follows



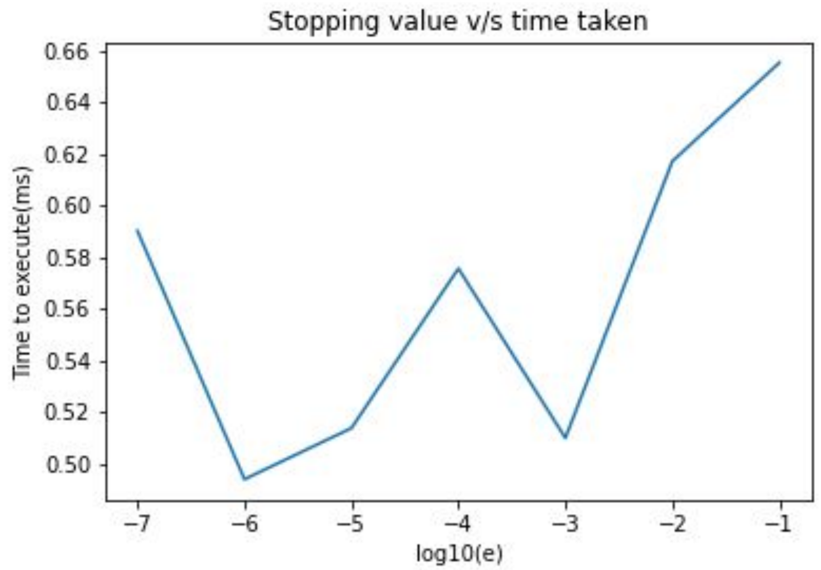






Time Analysis

Variation of time taken to execute for different stopping values is plotted below :

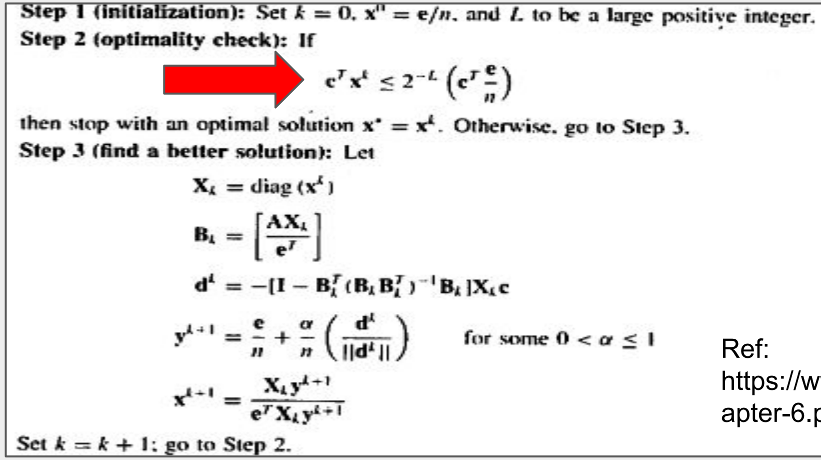


Stopping conditions confusion

$$f(\mathbf{x}) = \sum_j \ln\left(\frac{\mathbf{c}^T\mathbf{x}}{x_j}\right)$$

$$\rightarrow \quad f(\mathbf{x}^{(k+1)}) \leqq f(\mathbf{x}^{(k)}) - \delta$$

Ref: Karmarkar, N. (1984). "A new polynomial-time algorithm for linear programming". *Combinatorica*. **4** (4): 373–395



Step 1 (initialization): Set $k = 0$, $\mathbf{x}^0 = \mathbf{e}/n$, and L to be a large positive integer.

Step 2 (optimality check): If

$$\rightarrow \quad \mathbf{c}^T\mathbf{x}^k \leq 2^{-L}\left(\mathbf{c}^T\frac{\mathbf{e}}{n}\right)$$

then stop with an optimal solution $\mathbf{x}^* = \mathbf{x}^k$. Otherwise, go to Step 3.

Step 3 (find a better solution): Let

$$\mathbf{X}_k = \operatorname{diag}(\mathbf{x}^k)$$

$$\mathbf{B}_k = \begin{bmatrix} \mathbf{A}\mathbf{X}_k \\ \mathbf{e}^T \end{bmatrix}$$

$$\mathbf{d}^k = -[\mathbf{I} - \mathbf{B}_k^T(\mathbf{B}_k\mathbf{B}_k^T)^{-1}\mathbf{B}_k]\mathbf{X}_k\mathbf{c}$$

$$\mathbf{y}^{k+1} = \frac{\mathbf{e}}{n} + \frac{\alpha}{n}\left(\frac{\mathbf{d}^k}{\|\mathbf{d}^k\|}\right) \qquad \text{for some } 0 < \alpha \leq 1$$

$$\mathbf{x}^{k+1} = \frac{\mathbf{X}_k\mathbf{y}^{k+1}}{\mathbf{e}^T\mathbf{X}_k\mathbf{y}^{k+1}}$$

Set $k = k + 1$; go to Step 2.

Ref: https://www.ise.ncsu.edu/fuzzy-neural/wp-content/uploads/sites/9/2015/08/LPchapter-6.pdf

Non-homogeneous form

- Consider a strictly interior starting point, **a**
- Consider the transformation, $\mathbf{x}' = T(\mathbf{x})$ where $\mathbf{x} \in \mathbf{R}^n$, $\mathbf{x}' \in \mathbf{R}^{n+1}$ are defined by

$$x_i' = \frac{x_i/a_i}{\sum_j (x_j/a_j) + 1} \quad i = 1, 2, \dots, n$$

$$x_{n+1}' = 1 - \sum_{i=1}^{n} x_i'.$$

- Then,

$$A_i' = a_i A_i, \quad i = 1, \dots, n$$

$$A_{n+1}' = -\mathbf{b}.$$

$$\sum_{i=1}^{n+1} A_i' x_i' = \mathbf{0}.$$

$$c_i' = a_i c_i \quad i = 1, 2, \dots, n$$

$$c_{n+1}' = 0.$$

Non-homogeneous form

The original problem is then converted to the following homogeneous form,

$$\begin{aligned} &\text{minimize} \quad \mathbf{c}'^T\mathbf{x}' \\ &\text{subject to} \quad \begin{array}{l} A'\mathbf{x}' = \mathbf{0} \\ \mathbf{x}' \geqq \mathbf{0}, \ \sum_{i=1}^{n+1} x_i' = 1. \end{array} \end{aligned}$$

Where the inverse transformation is given by,

$$x_i = \frac{a_i x_i'}{x_{n+1}'} \quad i = 1, 2, \ldots, n.$$

Problems faced in non-homogeneous part

- Singular matrices in the inverse matrix calculations
- Underflow
- Reaches infeasible space due to interference with slack variables

Simplex Big-’M’ Method

Simplex Big-'M' Method

- **A variation of the simplex method designed for solving problems typically having 'greater than' or 'less than' inequality constraints**
- **It introduces surplus and artificial variables to convert all inequalities in standard form**
- **The method will be valid only if it is possible to force these variables at zero level when the optimum solution is attained**

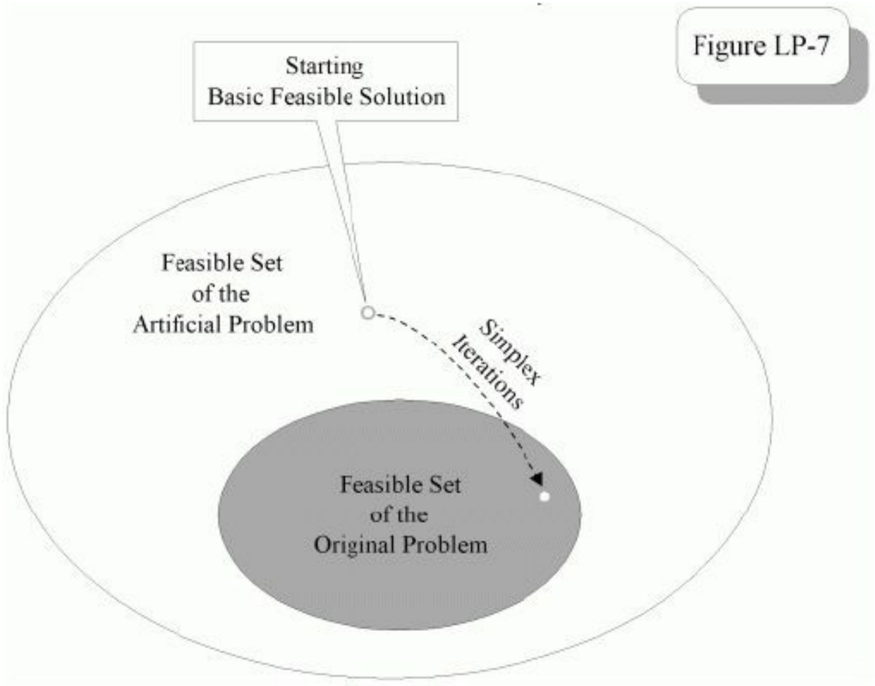
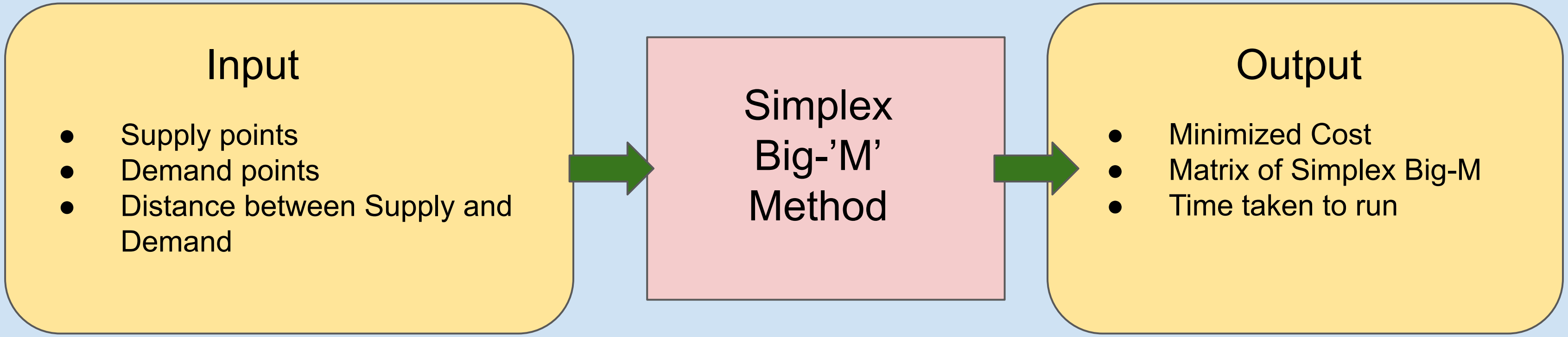
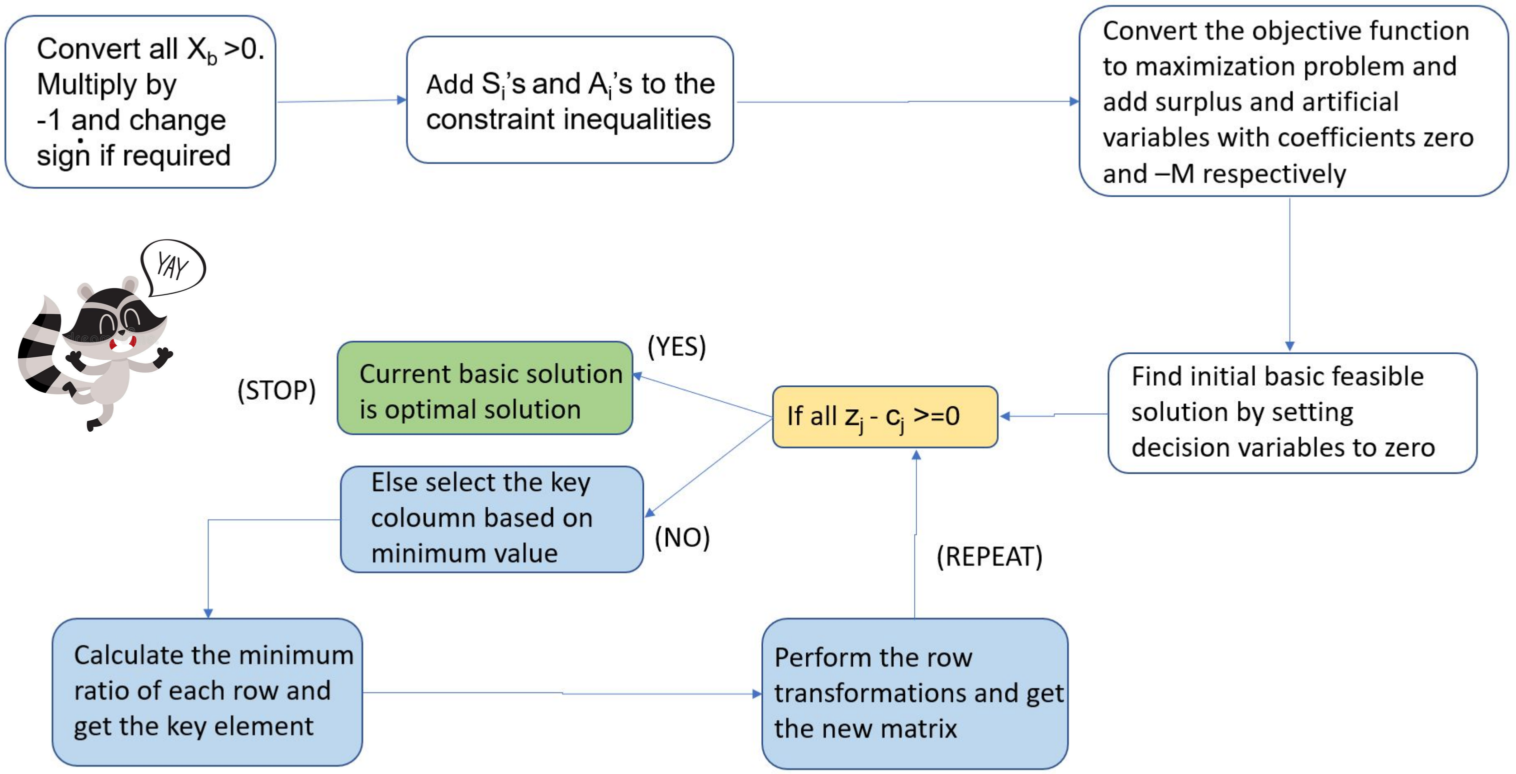


Figure LP-7
Starting Basic Feasible Solution
Feasible Set of the Artificial Problem
Simplex Iterations
Feasible Set of the Original Problem

Simplex Big-’M’ Method



Simplex Big-’M’ Method



Convert all $X_b > 0$. Multiply by -1 and change sign if required

Add S_i’s and A_i’s to the constraint inequalities

Convert the objective function to maximization problem and add surplus and artificial variables with coefficients zero and –M respectively

Find initial basic feasible solution by setting decision variables to zero

If all $z_j - c_j >= 0$

(YES)

Current basic solution is optimal solution

(STOP)

(NO)

Else select the key coloumn based on minimum value

Calculate the minimum ratio of each row and get the key element

Perform the row transformations and get the new matrix

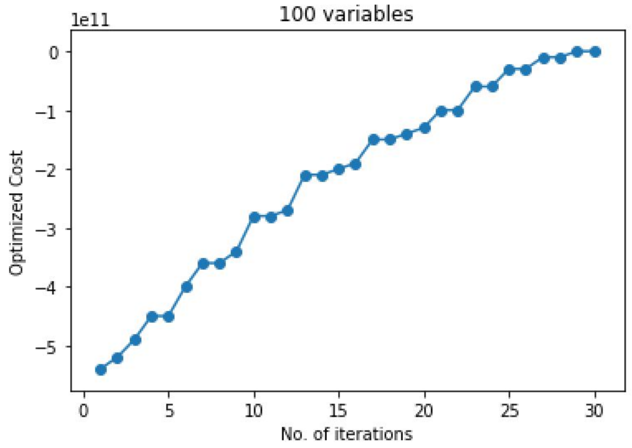
(REPEAT)

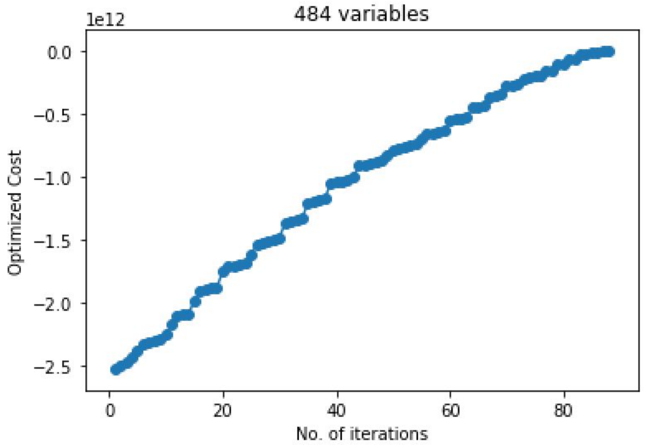
Pseudo code for Simplex Big-'M' Method

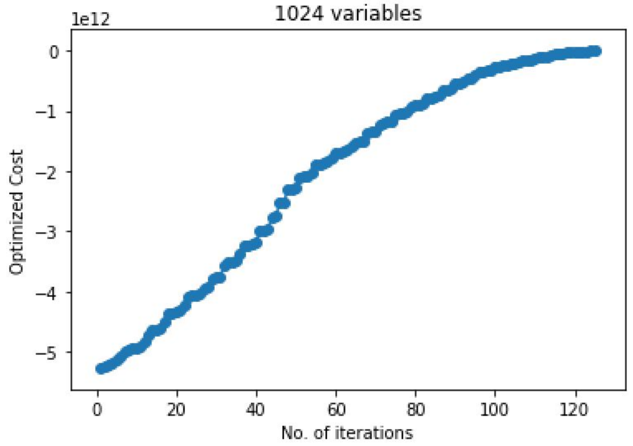
```
1.  Input quantity of each supply and demand point
2.  Input distance between each node
3.  Modify the objective function by adding surplus
    and additional variables
    $(c \sum YiXi) - M\sum Ai$
4.  Create the matrix-
    Cost function array-
        for i in range (len(S)):
            for  j in range (len(D)):
    Matrix rows for surplus variables -
        for i in range (len(S) + len(D)):
```

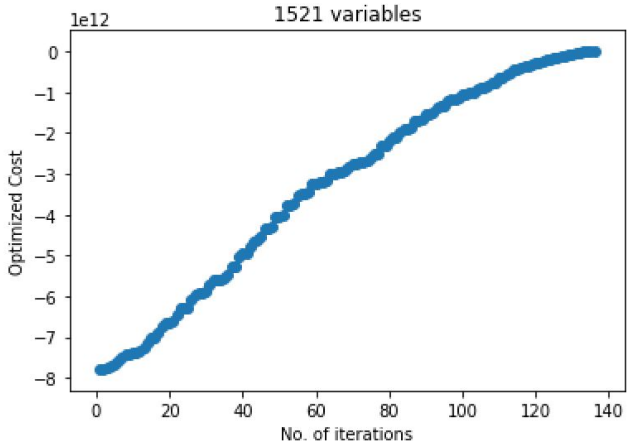
```
5.  Create function for min ratio for each row :
    Min ratio = min(Xi/Yi)
6.  Create zj-cj (array a):  zj-cj = $\sum YjCbj$ - Cj
7.  To reach the optimum solution all Zj-Cj>=0
     for i in range (len(a))
     if(a[i]<0)
        a.  Find key element using the min ratio and
            minimum zj-cj column
        b.  Apply row transformations
        c.  Calculate zj-cj array 'a' of the transformed matrix
        d.  Repeat
     else
            Break. All zj-cj>=0, optimum solution reached
8.   Output-  $\sum CbjXbj$
```

Variation of cost function with no. of iterations required









Observations and Conclusions

- The graph starts starts with a large negative value due to the contribution of -M in ΣXbCb (value of solution)
- The number of variables (input) is proportional to the number of iterations required as can be seen from the plot. (more the input variables more the number of iterations required)
- Worst case Time complexity for the algorithm is -

  O(no. of supply points * no. of demand points)
- Time required was less than puLP

Simplex Big-M Benchmarking

Test Example 1:

source = [3, 6]

dest = [2, 4]

Dist = [[1, 3], [2, 5]]

Corresponding Co-ordinates Chosen for Pulp Algorithm

supply_coords = [[7., 6.], [6., 8.]]

demand_coords = [[7, 5], [8.538, 3.513]]

Results from Simplex Big-M :

```
Optimized cost: 35.0
Solution Matrix = [[2,  1],
                   [0, 6]]
Time required- 0.0005102157592773438 seconds
Difference- 2.01%
```

Results from PULP :

```
Optimized cost:  35.72
Solution Matrix = [[1.         2.92414312]
                   [3.16227766 5.15505703]]
Time required- 0.020310163497924805 seconds
```

Simplex Big-M Benchmarking

Test Example 2:

Source = [8,4,12,6]

Dest = [3,5,10,7,5]

Dist =[[1, 2, 3, 4, 5], [2, 3, 4, 5, 6],
[3, 4, 5, 6, 7], [4, 5, 6, 7, 8]]

Corresponding Co-ordinates Chosen for Pulp Algorithm

supply_coords = [[0,0], [0,-1] ,[0,-2], [0,-3]]

demand_coords = [[0,1], [0,2], [0,3], [0,4], [0,5]]

Results from Simplex Big-M :

Optimized cost: 142.0

Time required- 0.003450632095336914 seconds

Difference-0%

Results from PULP :

Optimized cost: 142.0

Time required- 0.021148681640625 seconds

Advantages & Disadvantages of simplex big M

Advantages

1. Overcomes the initialization issue of simplex which requires basic feasible solution to start from
2. Gives satisfactory results for problems with inequality constraints

Disadvantages-

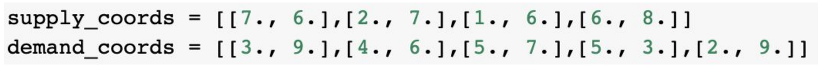
1. Determining M is a challenge as too big or too less values leads to loss of precision or computational errors
2. Feasibility is not known until optimality

Scope for improvement

1. Code can be generalised (designed for our problem statement)
2. Should be capable to deal with non-integer distance
3. Use of numpy array instead of lists - to improve efficiency

Source = [8,4,12,6]

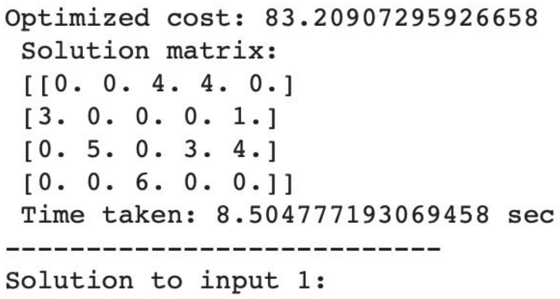
Dest = [3,5,10,7,5]



```
supply_coords = [[7., 6.],[2., 7.],[1., 6.],[6., 8.]]
demand_coords = [[3., 9.],[4., 6.],[5., 7.],[5., 3.],[2., 9.]]
```

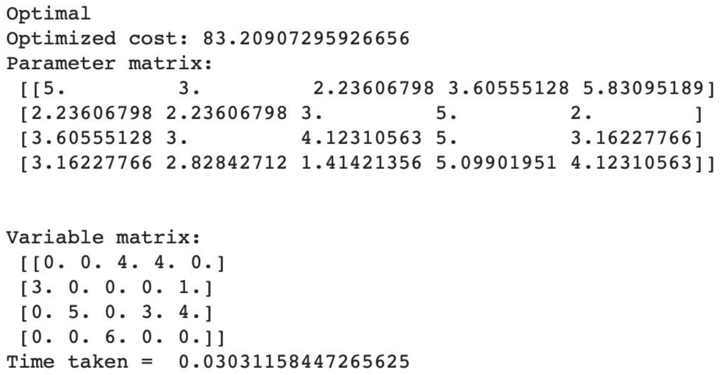
Dist = [5, 3, 2.236068, 3.60555, 5.830952], [2.236068, 2.236068, 3, 5, 2], [3.60555, 3, 4.1231, 5, 3.162278], [3.162278, 2.82843, 1.41421, 5.09902, 3.60555]

Genetic result



```
Optimized cost: 83.20907295926658
 Solution matrix:
 [[0. 0. 4. 4. 0.]
 [3. 0. 0. 0. 1.]
 [0. 5. 0. 3. 4.]
 [0. 0. 6. 0. 0.]]
 Time taken: 8.504777193069458 sec
---------------------------
Solution to input 1:
```

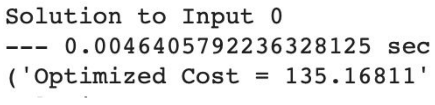
puLP result



```
Optimal
Optimized cost: 83.20907295926656
Parameter matrix:
 [[5.         3.         2.23606798 3.60555128 5.83095189]
 [2.23606798 2.23606798 3.         5.         2.        ]
 [3.60555128 3.         4.12310563 5.         3.16227766]
 [3.16227766 2.82842712 1.41421356 5.09901951 4.12310563]]


Variable matrix:
 [[0. 0. 4. 4. 0.]
 [3. 0. 0. 0. 1.]
 [0. 5. 0. 3. 4.]
 [0. 0. 6. 0. 0.]]
Time taken =  0.03031158447265625
```

big-M result



```
Solution to Input 0
--- 0.004640579223632812 5 sec
('Optimized Cost = 135.16811'
```

The End