Supplementary Material

Appendix I.

1. Definition of the cost function.

Input: $solution_1 \ . \ . \ . \ solution_N$

Output: $cost\ (Total\ cost\ of\ the\ solution)$

$function\ fitness\_function(solution[\ \ ],\ dest)$

$total_{price} = 0,\ last_{arrival} = 0$ # 0:00 time

$first_{departure} = 1439$ # 23:59 for initialization

$flight_{id} =- 1$

$for\ i \leftarrow 1\ to\ \frac{length(solution)}{2}\ do$

$origin = people[i][1]$

$flight_{id}\ += 1$

$going = flights[(origin,\ dest\ )]\left[solution\left[flight_{id}\right]\right]$

$flight_{id}\ += 1$

$returning = flights[(dest,\ origin)]\left[solution\left[flight_{id}\right]\right]$

$total_{price}\ += going[2]$

$total_{price} += returning[2]$

$if\ \ last_{arrival} < get\_minutes(going[1])\ then$ # Find last arrival

$last_{arrival} = get\_minutes(going[1])$

end if

$if\ first_{departure} < get\_minutes(returning[0])\ then$ # Find first departure

$first_{departure} = get\_minutes(returning[0])$

end if

end for

$total_{wait} = 0$

$flight_{id} =- 1$

$for\ i \leftarrow 1\ to\ \frac{length(solution)}{2}\ do$

$origin\ =\ people[i][1]$

$flight_{id} += 1$

$going\ =\ flights[(origin, dest)]\left[solution\left[flight_{id}\right]\right]$

$flight_{id} += 1$

$returning\ =\ flights[(dest, origin)]\left[solution\left[flight_{id}\right]\right]$

$\#\ Waiting\ time\ for\ all\ arrived$

$total_{wait} += last_{arrival} - get\_minutes(going[1])$

$\#\ Waiting\ time\ for\ all\ to\ depart\ and\ reach\ location$

$total\_wait\ \ +=\ \ get\_minutes(returning[0])\ -\ first\_departure$

end for

$\#\ 3PM\ -\ 10AM$

$\#\ 11AM\ -\ 3PM$

$if\ last_{arrival} > \ first_{departure}\ \ then$

$\#\ Penalize\ if\ arrival\ and\ departure\ are\ not\ on\ same\ days$

$total_{price}\ +=\ 50$

end if

$return\ total_{price}\ +\ total_{wait}\ \ \ \#\ The\ total\ cost\ associated$

$end\ function$

Appendix II.

1.Time Complexity Derivations

In the following Appendix we derive the Time Complexity of our algorithms, by first calculating the running time and then derive the upper bounds(worst case) by setting maximum values of N.

1.1 Cost Function

$S \rightarrow$ Length of initial Solution/Individual

Time Complexity Derivation:
$T(S) = S/2 + S/2;$

$\therefore$ $Time\ Complexity\ is\ O(N)$

1.2 OnePoint Mutation and Crossover

By ignoring the operation of copying N elements $O(N)$, gene selection then takes $O(1)$, as random.randint uses the **Mersenne-Twister** algorithm which is $O(1)$**.**
$\therefore$Time Complexity is **O(1).**

1.3 Random Search

$E \rightarrow$ Epochs
$D \rightarrow$ Length of Domain
$S \rightarrow$ Length of initial Population/Solution

Time Complexity:
$T(N) = D + E * (S/2) + (E - 1) * D;\ O(T(N)) = O(ES/2 + E - 1 * D)$;
$T(N) = N^2/2$
$\therefore Time\ complexity\ is\ O(N^2)$

1.4 Hill Climbing

E$\rightarrow$ Epochs
D$\rightarrow$ Length of Domain
S$\rightarrow$ Length of initial Population/Solution
n$\rightarrow$ Number of neighboring solutions

$T(N) = D + S/2 + n * (S/2)$;
$\therefore Time\ Complexity\ is\ O(N^2)$

1.5 Standard GA Configuration

$P \rightarrow$ Population Size
$G \rightarrow$ Number of Generations'
$D \rightarrow$ Length of Domain
$S \rightarrow$ Length of Solution/Individual
$C \rightarrow$ Length of array of Costs
$P_m \rightarrow$ Probability of Mutation
$P_c \rightarrow$ Probability of Crossover, $P_c = 1 - P_m$

$T(N) = P * D + G * ((P + 1) * S/2 + ClogC + P * (P_m * 1 + P_c * 1))$

this can be simplified to $T(N) = N^3/2 + N^3 logN$,

$\therefore Time\ Complexity\ is\ O(N^3)$

The choice of sorting function is responsible for $log\ N$. Python's default $Tim\ Sort$ instead of doing Heapify operations which reduces worst case complexity from $O(N^3 logN)\ to\ O(N^3)$.

1.6 GA with Reverse Operations

$P \rightarrow$ Population Size
$G \rightarrow$ Number of Generations
$D \rightarrow$ Length of Domain
$S \rightarrow$ Length of Solution/Individual
$C \rightarrow$ Length of array of Costs
$P_c \rightarrow$ Probability of Crossover
$P_m \rightarrow$ Probability of Mutation $P_m = 1 - P_c$

$T(N) = P * D + G * ((P + 1) * S/2 + ClogC + P * (P_c * 1 + P_m * 1)$

this is of the form

$O(N) = N^3/2 + N^3 logN$,

$\therefore O(N) = N^3$

1.7 GAs with Reversals

$P \rightarrow$ Population Size
$R \rightarrow$ Number of Reversals.
$G \rightarrow$ Number of Generations
$step_{length} \rightarrow$ The number of reverse steps/epochs

The number of reversals is calculated as follows:
$R = G/n_k - 1$

$T_R(N) = C; \quad if\ step_{length} = 1\ else$
$T_R(N) = (step_{length-1)} * (C + S/2 + P * (Pc * 1 + Pm * 1)$

Now actual Time Complexity of GA with Reversals is,

$T(N) = T(N)) + T_R(N)$
$T(N) = P * D + G * ((P + 1) * S/2 + ClogC + R * C + P * (Pm * 1 + Pc * 1));$
$if\ step_{length} = 1\ else$

$T(N) = P * D + G * ((P + 1) * S/2 + ClogC + R * ((step_{length-1)} * (C + S/2 + P * (Pc * 1 + Pm * 1)))$
$+ P * (Pm * 1 + Pc * 1))$

This further reduces to these 2 forms:
$T(N) = N^3/2 + N^3 = 3/2N^3 = N^3;\ if\ step_{length} = 1\ else$
$T(N) = N^3/2 + N^4,\ therefore\ Time\ Complexity\ is\ O(N^3);$

1.8 Iterated Chaining

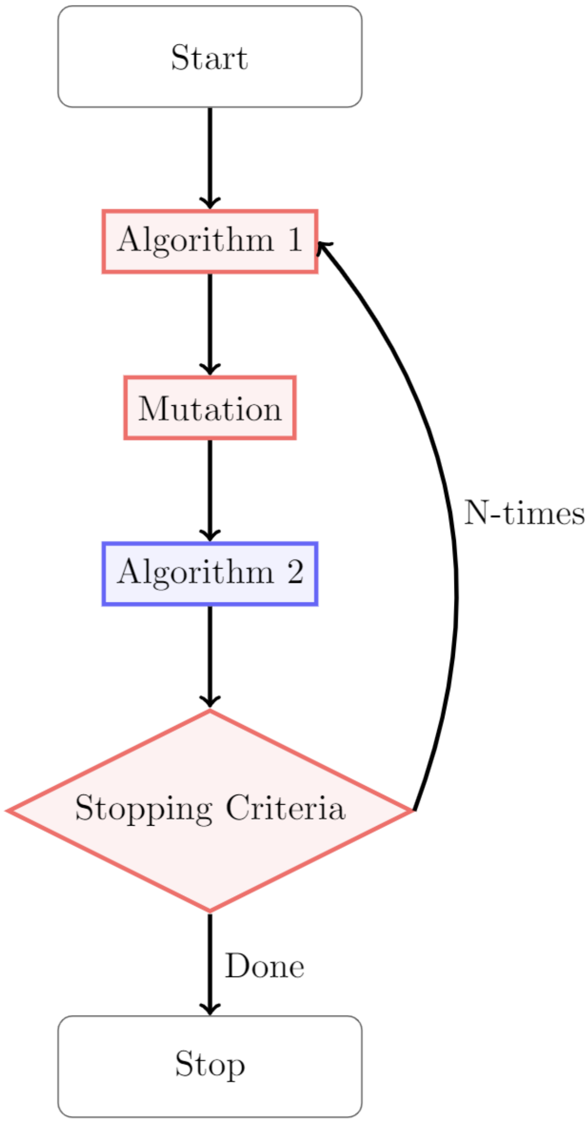
$Rounds$→ The number of iterated Chaining rounds

$T(N) = Rounds - 1 * T_{algo_1}(N) + Rounds * T_{algo_2}(N)$

The authors chose $algo_1\ as\ Random\ Search$ and $algo_2\ as\ HillClimbing$, thus:

$T(N) = Rounds - 1 * T_{RS}(N) + Rounds * T_{algo_2}(N)$
$T(N) = Rounds - 1 * D + E * (S/2) + (E - 1) * D + Rounds * D + S/2 + n * (S/2)$
$T(N) = N^3 - 1 + N^3$
$\therefore Time\ Complexity\ is\ O(N^3)$

Appendix III.



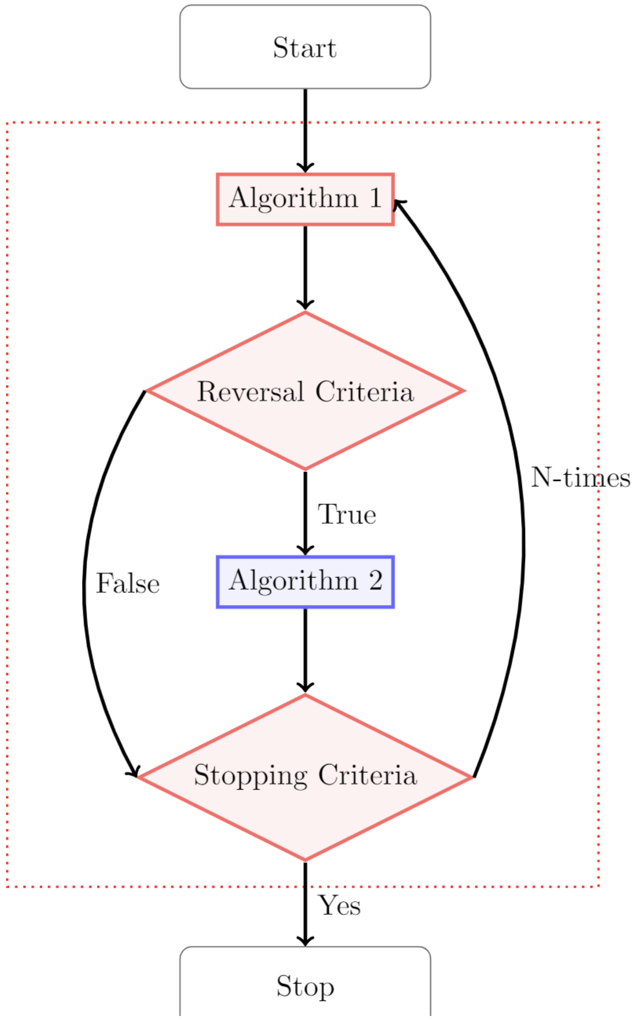
Iterated Chaining Algorithm

ALGORITHM : ITERATED CHAINING WITH EARLY STOPPING
Input: *Domain* D_i*....... *D_N*, rounds, fitness_function,* n_{obs}*, tolerance*
Output:$soln_{final}$,$best_{cost}$,*scores, NFE ,nfe,seed*

```
scores ← [ ], NFE ← 0 //Global record of cost and no. of function evaluations
for i ← rounds do
    if i==0 then
        soln, cost, scores, nfe, seed ←algorithm_1(domain, fitness_function, seed)
        soln ← OnePointMutation(domain, random.randint(0, 1), soln)
        scores← append cost to list
        NFE ← nfe+1
    end if
    else if i == rounds – 1 then
        soln_final, cost, scores, nfe, seed ←algorithm_2(domain, fitness_function, seed)
        scores← append cost to list
        return soln_final, scores [-1], scores, NFE
        NFE ← nfe+1
    end if
    else
        soln, cost, scores, nfe, seed ← algorithm_1(domain, fitness_function, seed)
        soln ← OnePointMutation(domain, random.randint(0, 1), soln)
        scores← append cost to list
        NFE ← nfe+1
    end else
soln_final, cost, scores, nfe, seed ← algorithm_2(domain, fitness_function, seed)
scores← append cost to list
NFE ← nfe + 1

if rounds ==1 then
    return soln, scores [-1], scores, NFE
end if
if cost - random.randint(tolerance, 100) > int(sum(scores[-n_obs:]) / n_obs) then
    return soln_final, scores [-1], scores, NFE
end if
```

Where scores[-1] ***is*** $best_{cost}$, *i.e the final cost; NFE is the global list of costs*


Start
Algorithm 1
Reversal Criteria
True
False
N-times
Algorithm 2
Stopping Criteria
Yes
Stop

ALGORITHM : GENETIC ALGORITHM WITH REVERSALS

Input: *Domain* D_i ……. D_N, $P_{mutation}$, $P_{crossover}$,, n_k , $step_{length}$, $num_{generations}$ *, fitness_function,* n_{obs}

,$population_{size}$

Output: $soln_{final}$, $best_{cost}$, *scores,nfe, seed*

for *i ←*$num_{generations}$ ***then***

```
population← Initialize population randomly
if i/n_k == 0 and i ≠ 0 then
    if step_length == 1 then
        Sort costs list in descending order instead
        rev ←rev+1
    end if
    else
        rev ←rev+1
        while i ← step_length do
            costs.sort(reverse=True) //Decreasing order of costs
            ordered_individuals = [individual for (cost, individual) in costs]
            population ← Get to a list of top n_eltisim from ordered_individuals
            scores←fitness_function(population[0])
            nfe ←nfe+1
            while length of population list < population_size do
                if random.random() <P_mutation then
                :
                    population ← Append result of Crossover of 2 randomly
                    chosen individuals from ordered_individuals
                end if
                else
                    population ← Append result of OnePointMutation of  a randomly
                    chosen individual from ordered_individuals
                end else
            end while
        end while
    end else
end if
else
    costs.sort() //Increasing order of costs
    ordered_individuals = [individual for (cost, individual) in costs]
    population ← Get to a list of top n_eltisim from ordered_individuals
    scores←fitness_function(population[0])
    nfe ←nfe+1
```

while *length of population list* $<$ *population_size* **do**

  if *random.random() <* $P_{mutation}$ **then**

    population ← Append result of ***Crossover*** *of 2 randomly chosen individuals from ordered_individuals*

  end if

  else

    population ← Append result of ***OnePointMutation*** *of a randomly chosen individual from ordered_individuals*

  end else

end while

end else

end for

return $soln_{final}$, $best_{cost}$, *scores,nfe, seed*

Algorithm : Genetic Algorithm with Random Search Reversals
Input: *Domain* D_i ……. D_N, $P_{mutation}$, $P_{crossover}$,, n_k , $step_{length}$, $num_{generations}$, *fitness_function,* n_{obs}
,$population_{size}$
Output: $soln_{final}$, $best_{cost}$, *scores,nfe, seed*

for *i* ←$num_{generations}$ **then**

population← Initialize population randomly

if i/n_k *== 0* **and** *i ≠ 0* **then**

  if *step_length == 1* **then**

    Sort costs list in descending order instead

    rev ←rev+1

  end if

  else

    rev ←rev+1

    while *i* ← $step_{length}$ **do**

      costs.sort(reverse=True) //Decreasing order of costs

      soln ← Randomly initialize within U.B and L.B of D

      population ← Get to a list of top $n_{eltisim}$ *from ordered_individuals*

      scores←fitness_function(population [0])

```
            nfe ←nfe+1
            if cost > best_cost then
            :
                best_cost ← cost
                best_solution ← solution
            end if
            scores← Append best_cost
            population←Append best_soln
        end while
    end else
end if
else
    costs.sort() //Increasing order of costs
    ordered_individuals = [individual for (cost, individual) in costs]
    population ← Get to a list of top n_eltisim from ordered_individuals
    scores←fitness_function(population [0])
    nfe ←nfe+1
    while length of population list < population_size do
        if random.random() <P_mutation then
            population ← Append result of Crossover of 2 randomly
            chosen individuals from ordered_individuals
        end if
        else
            population ← Append result of OnePointMutation of  a randomly
            chosen individual from ordered_individuals
        end else
    end while
end else
end for
return soln_final, best_cost, scores,nfe, seed
```