CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Department of Computer Science

Assignment 1 – NP Completeness and Genetic Algorithms

Module Name: Metaheuristic Optimization – COMP9058

Student Name: Sreekanth Palagiri - R00184198

Table of Contents

Part 1: NP-completeness

Convert formula F into a 3SAT formula F'

Last digit of student ID is 8. Formula selected is: $F = (q_1 \vee q_4) \wedge (\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4 \vee \neg q_5)$

Formula f is can be divided into two clauses:

$F = C_1 \wedge C_2$ where, $C_1 = (q_1 \vee q_4)$ and $C_2 = (\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4 \vee \neg q_5)$

Conversion of the formula F from SAT to 3SAT can be done by converting C1 and C2 and replacing them back into the formula F.

Clause C_1:

We need to introduce one additional variable x_1 to make this as 2 clause 3SAT formula:

$$C_1 \longrightarrow (q_1 \vee q_4) \longrightarrow (q_1 \vee q_4 \vee x_1) \wedge (q_1 \vee q_4 \vee \neg x_1)$$

Clause C_2:

We need to introduce two additional variables-> x_2 and x_3 to make this as 3 clause 3SAT formula:

$$C_2 \longrightarrow (\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4 \vee \neg q_5) \longrightarrow (\neg q_1 \vee q_2 \vee x_2) \wedge (\neg x_2 \vee q_3 \vee x_3) \wedge (\neg x_3 \vee \neg q_4 \vee \neg q_5)$$

F can be now stated in 3SAT as:

$$F' = (q_1 \vee q_4 \vee x_1) \wedge (q_1 \vee q_4 \vee \neg x_1) \wedge (\neg q_1 \vee q_2 \vee x_2) \wedge (\neg x_2 \vee q_3 \vee x_3) \wedge (\neg x_3 \vee \neg q_4 \vee \neg q_5)$$

Find a solution to F' and verify that this is a solution to F

Formula F' is satisfiable, if we can assign Boolean values to literal such that F' is True.

We will divide F' into clauses C_1, C_2, C_3, C_4, C_5 as below.

$C_1 = (q_1 \vee q_4 \vee x_1)$, $C_2 = (q_1 \vee q_4 \vee \neg x_1)$, $C_3 = (\neg q_1 \vee q_2 \vee x_2)$, $C_4 = (\neg x_2 \vee q_3 \vee x_3)$, $C_5 = (\neg x_3 \vee \neg q_4 \vee \neg q_5)$

F' can be rewritten as:

$F' = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$

Below is truth table of few such valid assignments, assignment of variables x would not have any effect on F':

q_1	q_2	q_3	q_4	q_5	$\neg q_1$	$\neg q_2$	$\neg q_3$	$\neg q_4$	$\neg q_5$	C_1	C_2	C_3	C_4	C_5	F'
T	T	T	F	T	F	F	F	T	F	T	T	T	T	T	T
T	T	T	F	F	F	F	F	T	T	T	T	T	T	T	T
T	T	T	T	F	F	F	F	F	T	T	T	T	T	T	T

Same are also valid for F. F can be rewritten as below:

$F = C_1 \wedge C_2$, where $C_1 = (q_1 \vee q_4)$ and $C_2 = (\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4 \vee \neg q_5)$

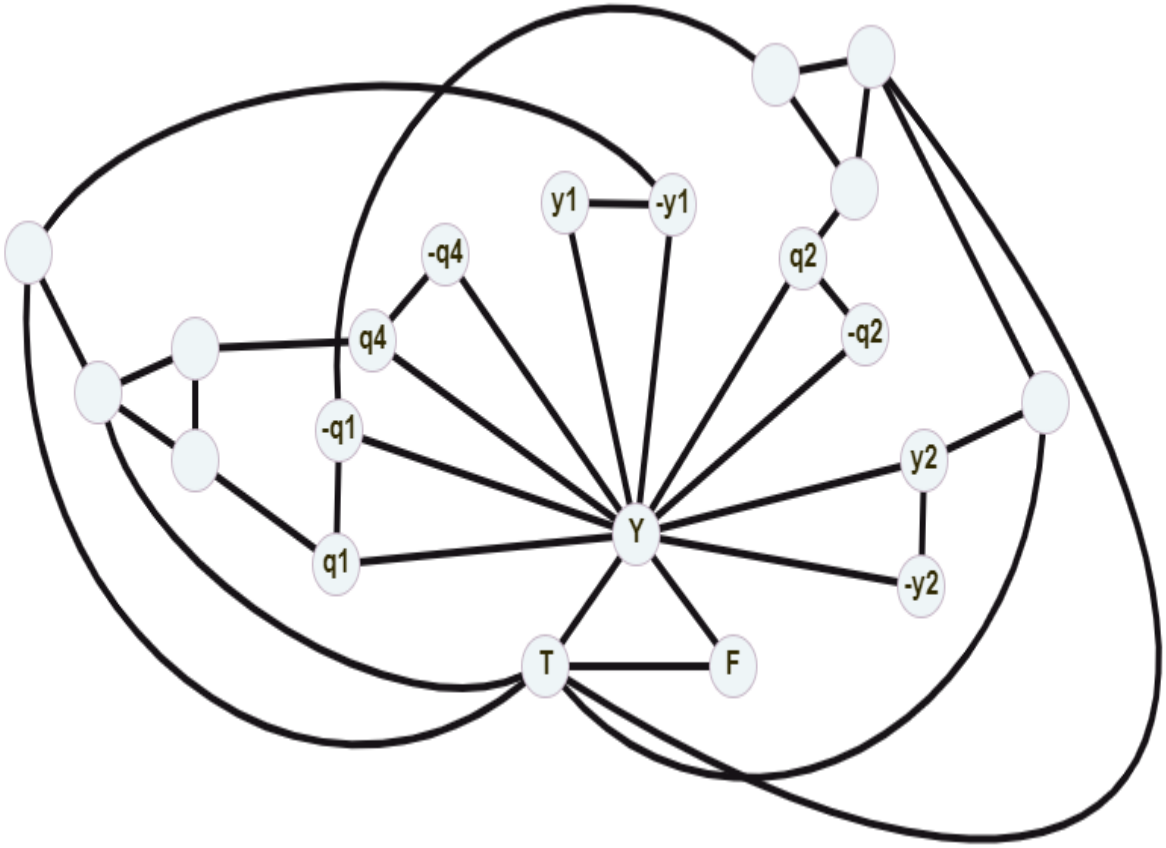
q_1	q_2	q_3	q_4	q_5	$\neg q_1$	$\neg q_2$	$\neg q_3$	$\neg q_4$	$\neg q_5$	C_1	C_2	F
T	T	T	F	T	F	F	F	T	F	T	T	T
T	T	T	F	F	F	F	F	T	T	T	T	T
T	T	T	T	F	F	F	F	F	T	T	T	T

Convert subclauses in your F' to a 3Col graph

First letter of first name is S, clauses selected are 2 and 3. Below is selected clause:

$$3COL= (q_1 \vee q_4 \vee -x_1) \wedge (-q_1 \vee q_2 \vee x_2)$$

Graph for 3COL:



Note: Graph has been created using http://graphonline.ru

Part 2: Genetic Algorithms - Description

Genetic algorithms belong to class of evolutionary algorithms which are inspired by process of natural selection. Algorithms reflect process of natural selection where fittest individuals are selected to reproduce offspring of next generations. The process is iterated many times till a criterion or maximum number of iterations is reached. The result is population of fittest individuals. Individual with highest fitness can be considered as a solution.

Genetic algorithms are one of the many techniques used in optimization of search problems. Genetic algorithms are being used in fields of automotive design, traffic and shipment routing, healthcare, robotics and others to fasten the design process.

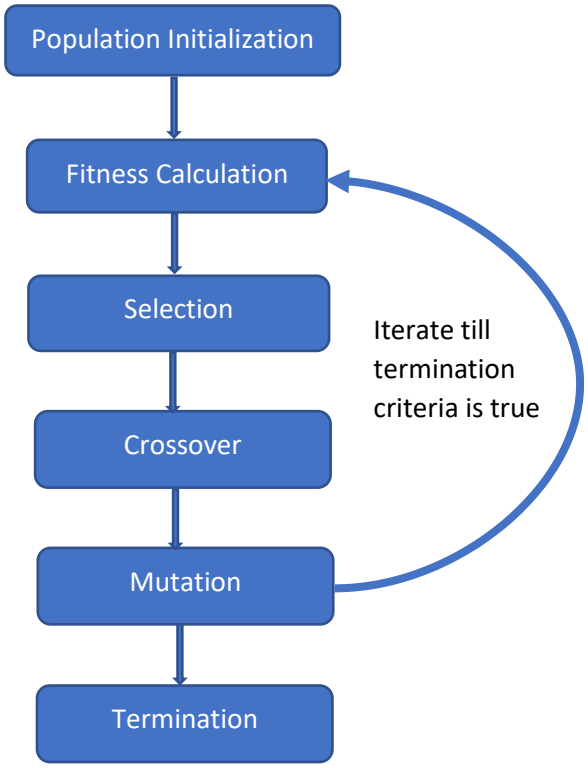
Genetic Algorithm and Travelling Salesmen Problem (TSP)

TSP problem statement:

"Given a list of cities and the co-ordinates of cities, what is the shortest possible route that visits each city once and returns to the origin city?"

Travelling salesperson is a NP-Hard problem. The easiest way to solve this problem is find all possible combinations of cities, find distances using co-ordinates and find the route with least distance. This is also very expensive as number of possible combinations would be (N-1)!. TSP is treated as optimization problem and metaheuristics such as genetic algorithms are used to find the optimal solutions, these may not provide the best possible solution but can provide a sufficiently good solution.

Steps in Genetic Algorithms:



Population Initialization

Population is set of individuals, each individual being a single solution to the problem. An individual is characterized by a set of variables called genes. Usually, each gene in genetic algorithm is a characterized by number, either binary or a digit depending on the type of the problem. Population initialization is the first step of genetic algorithms. Initial population is created either randomly by creating different combinations of genes or using heuristic methods where we randomly pick one gene and then one nearest to the first gene as next in sequence.

For TSP, each individual is a single route satisfying problem conditions. Each gene is city represented by a number. Population P is collection of N possible routes, where N is population size. Each route is formed either through random combination of cities or by nearest neighbour method. In nearest neighbour method, we pick a city and then city nearest to first city and so on till we choose all the cities. We validate both methods as part of our assignment.

Nearest Neighbour Method

In the nearest neighbour method that we use is Euclidean Distance as metric. Euclidean distance is defined as below:

$$d(q,p)=d(p,q)=\sqrt{\sum(q_1-p_1)^2+(q_2-p_2)^2+..+(q_n-p_n)^2}$$

where q and p are coordinates of a node or city.

Nearest neighbour is used during population initialization in the program to select an individual. Below is the pseudo code of nearest neighbour method:

```
Nearestneighbour(gene)
    Gene is list of indexes of a ciy
    temp =gene.copy()
    e=gene[e]
    newlist=[e]
    temp.remove(e)
    while temp not empty:
       e=getnearest(e,temp)
       temp.remove(e)
       newlist.append(e)
    return newlist
getnearest(e,temp):
   e is element
   temp is list of elements
   bestdist=infinity
   for i in temp:
      dist=Euclidean(i,e)
      if dist < bestdist
         bestdist=dist
         index=i
   return i
```

Fitness Calculation

Fitness function of genetic algorithm evaluates optimality of a solution for the problem. In other words, it determines how fit a solution is. Fitness function used is dependent on type of problem. As part of this step, fitness of each individual of the population is evaluated.

For TSP, distance of the route determines fitness. We can use one of the distance functions to calculate the fitness. We are considering Euclidean distance as part of this assignment. TSP is a minimisation problem i.e.; we must find a solution with least distance. So, we will consider inverse function i.e. 100000000/Euclidean distance as fitness. Individual with best fitness across the iterations is accepted as solution.

Selection

Selection is the stage of the algorithm where individuals are chosen from the population for breeding and mutation later. These individuals are called parents. Selection can be random, or we can apply genetic operators like Roulette wheel selection, Stochastic Universal Sampling or Tournament selection. The selected population are then transferred to mating pool.

Random Selection

In random selection, we choose parents randomly. We don't evaluate fitness of individuals while selecting parents. This strategy is usually least opted. We evaluate this method as part of our assignment.

Fitness Proportionate Selection

Fitness proportionate selection is one of the more popular selection methods. Probability of an individual getting selected is proportional to its fitness value. In this method, individuals with highest probabilities have higher changes of getting picked. There are two variations of this method:

1) Roulette Wheel Selection
2) Stochastic Universal Sampling

Roulette Wheel Selection:

In this method, we consider a circular roulette wheel. Numbers on the wheel are in the range of zero to size of population. Wheel is divided into many pies, each pie occupying portion of the wheel proportional to its fitness value of an individual. A fixed point is chosen on the wheel and the wheel is rotated. The individual corresponding to region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.

For implementation, a random number is generated each time and individual related to pie having this random number is chosen as parent. Disadvantage of roulette wheel selection is that few individuals might dominate the mating pool as we iterate.

Stochastic Universal Sampling (SUS)

SUS is an improvement over roulette wheel method since it reduces selection bias and does not allow the mating pool to be dominated by only few fittest members of the population. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness value. Equally spaced N pointers, where N is equal to number of parents required for mating, are placed over the line. N pointers are spaced at distance P=F/N, F is total sum of fitness. Individuals at the pointers are considered as parents.

As part of this assignment, we are using SUS for our TSP problem. **Complexity of SUS is $O(n^2)$** where n is size of gene.

Pseudo code of the algorithm implemented in this assignment is below:

```
StochasticUniversalSampling()
    matingpool=[]
    F= sum(fitness of population Populationpool)
    N= popsize
    P= (F/N) # distance between the pointers
    rn= random number between 0 and P
    Pointers= [rn + i*P for i in range(popsize)]
    for i in pointers[1… i]
        j,fitsubtotal= 0
        while fitsubtotal is less than pointers[i]
            j++
            add Populationpool[j].fitness to fitsubtotal
        Append Populationpool[j] to matingpool
     Return matingpool
```

Ordinal Based Methods

In ordinal based methods, probability of selecting an individual depends on its relative order or ranking compared with other methods. One of the more popular ordinal based algorithms is tournament selection.

Tournament selection

In K-Way tournament selection, we select K individuals from the population at random and select the best among these to become a parent. The same process is repeated for selecting the N parents. We can choose to replace the parent back in the draw or remove it completely. Without replacement will result in less variance but not necessarily individual with better fitness.

Crossover

Crossover is a genetic operator used to combine the genetic information of two parents to generate new offspring. There are many crossover operators that can used based on the problem. As part of this assignment we have implemented uniform order-based crossover and PMX crossover.

Uniform Order Based Crossover

In uniform order-based crossover, two parents are randomly selected, say P1 and P2 and a random binary template is generated. Genes for offspring C1 are filled by taking the genes from parent P1 where there is a one in the template. The genes of parent P1 in the positions corresponding to zeros in the template are taken and sorted in the same order as they appear in parent P2. The sorted list is used to fill the gaps in C1. Offspring C2 is created by using a similar process.

Complexity of algorithm is $O(n^2)$ where n is size of gene. Pseudo code for Uniform Order Based Crossover implemented in this assignment is below:

```
uniformCrossover(indA, indB):
    indA,indB – Individuals being used for mating
    selector=Random binary list of gene size
    copy selector into A replace elements from indA where selector value is 1
    copy selector into B replace elements from indB where selector value is 1
    For each element i in indB
       If i in A, Pass:
       Else For j in range(len(A))
               If A[j] is empty replace by i
    For each element i in indA
       If i in B, Pass:
       Else For j in range(len(B))
               If B[j] is empty replace by i
    indA=A
    indB=B
```

PMX Crossover

In PMX crossover we take two random points of the gene. Portion between two segments is then passed to the offspring and remaining information exchanged such that order and position of as many genes in the individual are preserved from another parent.

Complexity of algorithm is $O(n^2+m)$, where n is segment size, m is size of the gene. Pseudo code for PMX crossover is given below:

```
PMXCrossover(indA, indB):
    indA,indB – Individuals being used for mating
    indexA=random integer between 0 and len(gene)
    indexB=random integer between 0 and len(gene)
    i=min(indexA,indexB)
    j=max(indexA,indexB)
    A=make a list copying indA and replace values of elements outside I,j with ‘’
    B=make a list copying indB and replace values of elements outside I,j with ‘’
    For a,b in zip(indA.genes[i:j], indB.genes[i:j]):
       #Find position of elements in selected segments
       if b not in A:
           x = index of a in indB.genes
           while A[x]!='':
              x = index of A[x] in indB.genes
           A[x]=b
       if a not in B:
           x = index of b in indA.genes
           while B[x]!='':
              x = index of B[x] in indA.genes
           B[x]=a
     For i in range(self.genSize):
        Copy element from indB in A[i] if A[i] is empty
        Copy element from indA in B[i] if B[i] is empty

     indA.genes=A
     indB.genes=B
```

Mutation

A mutation is a genetic operator which when applied changes part of the gene. Mutation is important to introduce new information into individual and is used to maintain genetic diversity from one generation to other. Mutation rate determines how many genes or individual undergo mutation. Mutation rate is expressed as probability of mutation occurring at any given point of time.

As part of this assignment we have used Inversion mutation and reciprocal exchange.

Inversion Mutation

In inversion mutation, we select a subset of genes and we merely invert(reverse) the entire string in the subset. Algorithm has constant complexity O(1). Below is pseudo code of inversion mutation:

```
inversionmutation(ind):
    ind – Individual selected for mutation
    rn = random decimal between 0 and 1
    if rn > 0.1
       return
    indexA=random integer between 0 and length(gene)
    indexB=random integer between 0 and length(gene)
    i=min(indexA,indexB)
    j=max(indexA,indexB)
    temp =ind[i:j]
    ind[i:j]=tmp.reverse()
```

Reciprocal Exchange

In reciprocal exchange, we select two random genes and we swap them with each other. Algorithm has constant complexity O(1). Below is pseudo code:

```
reciprocalexchangemutation(ind):
    ind – Individual selected for mutation
    rn = random decimal between 0 and 1
    if rn > 0.1
       return
    indexA=random integer between 0 and length(gene)
    indexB=random integer between 0 and length(gene)
    temp=ind.genes[indexA]
    ind.genes[indexA]= ind.genes[indexB]
    ind.genes[indexB]= temp
```

Termination

The algorithm terminates if the population has converged (offspring are not significantly different from the previous generation) or it has reached maximum of iterations. The best solution achieved is treated as near optimal solution.

Evaluation of GA

In this section we evaluate GA with different crossover and mutation methods. We also evaluate mutation rate and population parameters. We have used dataset inst-16.tsp. **Due to computational issue we kept population at 100 for configuration evaluation and the fitness values are a fraction of fitness that can be achieved with higher population.**

Basic Evaluation

In this part we will evaluate below two basic configurations:

Configuration	Initial Solution	Crossover	Mutation	Selection
1	Random	Uniform Crossover	Inversion Mutation	Random Selection
2	Random	PMX Crossover	Reciprocal Exchange	Random Selection

We used parameters population at 100 and mutation rate at 0.1.

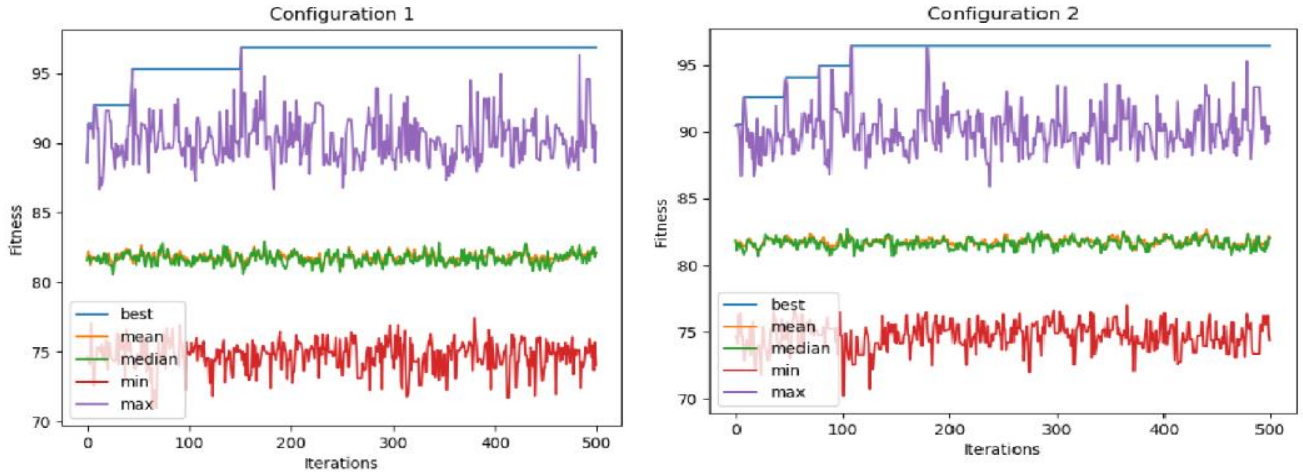


Figure 1: Basic Evaluation

Figure 1 above is the plotting of mean, median, min and max fitness per each iteration along with best fitness till that iteration. Based on above, we can conclude that both configurations are exhibiting almost similar behaviour. However, best fitness is achieved in big steep steps for configuration 2 (PMX and Reciprocal) than configuration 1. This could be because we preserve entire segment in PMX crossover and transfer it to the child whereas in uniform crossover random elements are transferred creating a more complex offspring in single generation. PMX crossover would need many iterations to achieve the same.

Configuration	Average Time per iteration	Average Execution Time(S)	Average Least Distance	Average Best Fitness
Configuration 1	0.43	214.30	103121614.8	96.98
Configuration 2	0.24	120.36	103315440	96.80

Table 1: Comparison of results from 3 iterations

In terms of performance, there is no considerable difference in best fitness or distance of the best route but configuration 2 is taking half the time of configuration 1. This is due to uniform crossover which has time complexity of $O(n^2)$ where n is size of gene. PMX has complexity of $O(m^2+n)$, where m is size of the segment selected and n size of gene. And since m is always less than n sometimes a fraction of n, we see the big difference in performance of configuration.

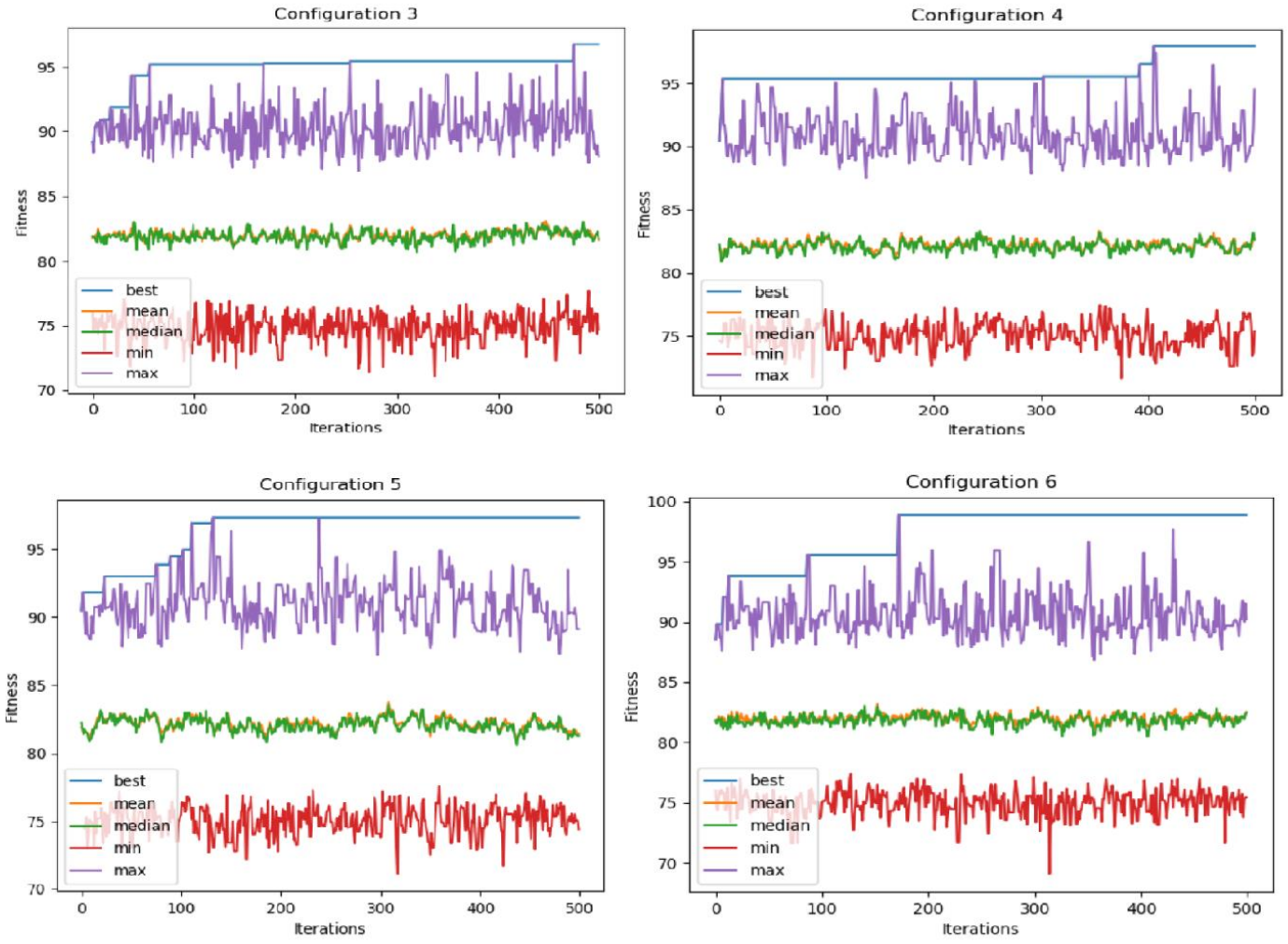


Figure 2: Extensive evaluation – Configurations 3,4,5 and 6

Extensive Evaluation

In this part, we will first evaluate below four configurations. All the configurations have stochastic universal sampling as selection method and random initial solution.

Configuration	Initial Solution	Crossover	Mutation	Selection
3	Random	Uniform Crossover	Reciprocal Exchange	Stochastic Universal Sampling
4	Random	PMX Crossover	Reciprocal Exchange	Stochastic Universal Sampling
5	Random	PMX Crossover	Inversion Mutation	Stochastic Universal Sampling
6	Random	Uniform Crossover	Inversion Mutation	Stochastic Universal Sampling

Figure 2 in page 10 shows the plotting of mean, median, min and max fitness per each iteration along with best fitness till that iteration. Our results from basic evaluation are consistent with extensive evaluation with respect to crossover methods. Configuration 4, 5 and 7(with PMX) take small steps towards best solution while 3,6 and 8 take big steps. With addition of Stochastic universal sampling in place of random selection, we see that best fitness values are achieved in less number of iterations. We also see that range of max fitness values is higher compared to random selection. We don't see any major difference in mean fitness values, but population size is not big enough to be exact with analysis.

Configuration	Average Time per iteration (s)	Average Execution Time (s)	Average Least Distance	Average Best Fitness
Configuration 3	0.44	220.13	101458658.26	98.59
Configuration 4	0.22	108.60	102607542.74	97.46
Configuration 5	0.22	110.80	99955678.63	100.26
Configuration 6	0.43	215.64	103295507.98	96.83

Table 2: Comparison of results from 3 iterations

From above table, we see that with stochastic sampling best fitness values across three iterations are higher than using random selection. We also don't see stochastic universal sampling impacting performance of the program. The program execution times are dependent on crossover techniques that we used.

Impact of Nearest Neighbour

In this part we evaluate below two configurations.

Configuration	Initial Solution	Crossover	Mutation	Selection
7	Nearest Neighbour	PMX Crossover	Reciprocal Exchange	Stochastic Universal Sampling
8	Nearest Neighbour	Uniform Crossover	Inversion Mutation	Stochastic Universal Sampling

Through these configurations, we analyse the impact of nearest neighbour on our genetic algorithms. Impact of nearest neighbour heuristic is not significant on performance of the program. The program with configuration 7 and 8 completes in line with other configurations.

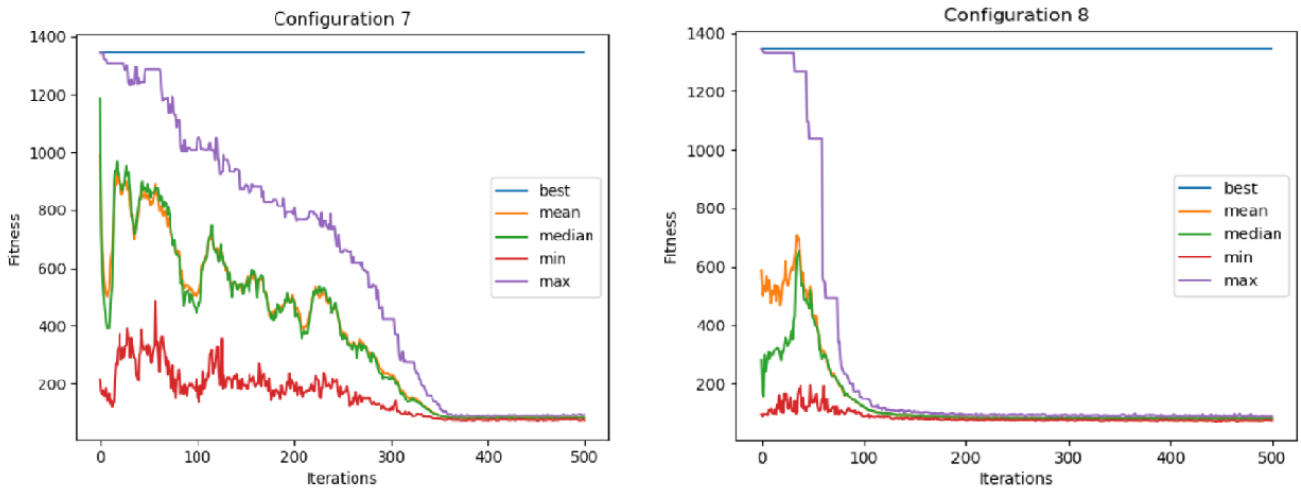


Figure 3: Extensive evaluation – Configurations 7 and 8

Figure 3 shows the plotting of mean, median, min and max fitness per each iteration along with best fitness till that iteration. With nearest neighbour as initial solution, we see that high-quality solutions are often identified very early and fitness measure decreasing with iterations. **This is because of insufficient population size.** Having high value for population parameters, usually in multiples of gene size, would solve the above problem. With higher population parameter we can also see that best and max fitness values of configuration with and without heuristic as initial solution converge. **This is shown by running GA on toy dataset of 10 cities with initial population as 1000 in the below figure 4:**

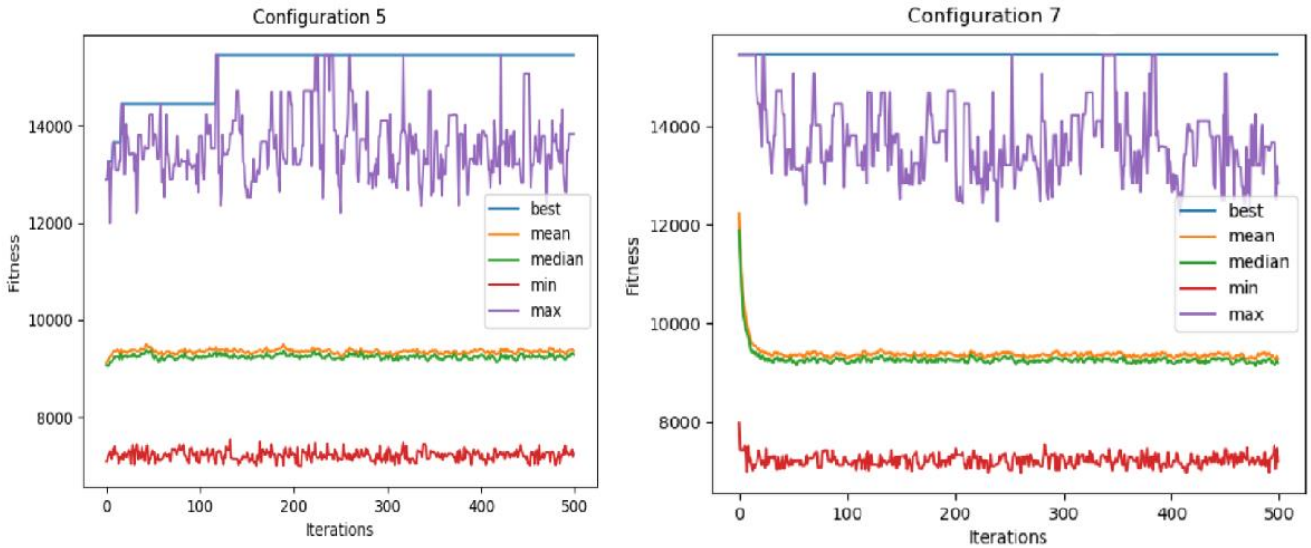


Figure 3: Toy problem of 10 cities and 1000 initial population

Using heuristic methods for creating initial population creation with high population size will result in identifying fitter solutions very early and creation of newer generation with better fitness measure. Elite survival if implemented together with initial heuristic can result in having high mean of fitness measure.

Impact of Population Size

We analyse impact of population on GA by varying population parameters. We use configuration 5 (['Random Initial Solution', 'PMX Crossover', 'Reciprocal Exchange Mutation', 'Stochastic Universal Sampling']) for this analysis and we range our population across 5 values. ([100,200,300,500,1000). Due to computational issues we limited our population to 1000.

Population	Average Execution Time	Best Fitness
100	103.80	97.33
200	119.07	96.45
300	200.76	98.23
500	245.33	98.99
1000	348.41	100.04

Table 3: Comparison of Populations

We can see from table 3 that with increase in population, best fitness and completion time of the program also increase. Increasing population size further would increase the fitness further as we will select, create and mutate more members in new generations. Population size can be in increased to few multiples of gene size to give enough variations. Increasing population size beyond a point might only increase computational complexity and might not provide a significant improvement in solutions.

Impact of Mutation Rate

We analyse impact of mutation rate by varying mutation parameters. We use configuration 5 (['Random Initial Solution', 'PMX Crossover', 'Reciprocal Exchange Mutation', 'Stochastic Universal Sampling']) for this analysis and we range our mutation rate across 5 values. ([0.01,0.05,0.1,0.2,0.5]). We have kept population parameter constant at 300. Mutation rate has very minor impact on performance of the program.

Mutation Rate	Average of Best Fitness
0.01	99.21
0.05	99.29
0.1	98.23
0.2	98.69
0.5	97.19

Table 4: Mutation Evaluation

Above table doesn't show much impact of mutation rate on fitness, we need to analyse this with higher population and more iterations. But keeping mutation rate low will allow gradual changes in population composition. It will also be in line with evolutionary principle of nature where mutation is not frequent. Ideal mutation rate is between 0.05 and 0.1. Best mutation rate can be arrived with help of experimentation.

Elitism

Elitism is a way to carry best of this generation to future. This ensure that the most successful individuals persist even in future generation. There are many ways to achieve this, but simplest way is by replacing 10% least fit individuals of new generation with 10% of previous generation. Elitism is not part of the evaluations done during this assignment and program developed doesn't include elitism.

Conclusion

Through evaluation of experiments done during this assignment, we can conclude that, A genetic algorithm can be used to arrive at reasonable solutions for Travelling Sales Person Problem. High quality solutions can be achieved by having high population size, having initial solution heuristic and keeping mutation rate at 0.05.