# Introduction

The treatment of Combinatorial Optimization Problems (COP) with Evolutionary Computation is a welcome and timely addition to my academic research in Reversible Logic Synthesis of Quantum Circuits. Akin to the problem at hand, *Quadratic Assignment Problem,* reversible logic synthesis problems of small number of variables can be easily solved through exhaustive permutation of input/output mapping. However, as the number of variables increase, the solution becomes unfeasible as the number of permutations grows factorially. Benefiting from the experience gained from this project, I plan on extending my current algorithm for synthesis of reversible circuits to employ evolutionary computing and explore the impact of such treatment on circuit optimization.

For this project, in addition to implementing the genetic algorithm, I implemented a permutative algorithm, which was easily able to find the optimal solution for this 10 variable problem. However, when I raised my limit to 20 variables, the permutation algorithm became impossible to compute within the allotted time for this project, and hence, abandoned the effort. I wanted to observe the performance of the GA as the problem size increased and whether it would discover, or how close would it be to, the global optima within a reasonable time.

# Observation

The algorithm used for this project is a standard GA with a constant population size of 100 using two point cross over with elitism as a recombination operator and a two element swap as a mutation operator. Four hundred generations were generated for twenty runs where the best fitness of the runs is plotted in Figure 1. For the problem at hand, each solution (individual) consists of an array of N integers where N = 10. The index of the array represents a location (plot of land) and the content of each element of the array represent the facility occupying that location.

facility

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 9 | 1 | 10 | 2 | 4 | 5 | 7 | 6 | 8 |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |

Genotype

Location

Figure Fitness vs. Generation of Best run.

Figure 1 shows the plot of the fitness per generation for the best run out of the twenty runs we did. The fitness function used is the reciprocal of the cost function specified in the problem statement, as follows:

(for facility *i* in location *k* and facility *j* in location *p*)

The function measures the fitness of a solution in minimizing the cost function which measures the best allocation of facilities to locations such that the accumulative flow of material per unit of distance between facilities is minimized. We chose the fitness function above as we were assured by our permutation shortcut that the minimum cost would never drop below 188, and hence, no division by zero is possible. Notice in the figure that the fitness function saturates early in the process at the 46th generation. Since our termination criteria was a fixed limit of 400 generations, the algorithm could not improve further. For this run, the maximum fitness of *0.005128* corresponds to the minimum cost of 188. This cost was easily discovered through our side experiment where all permutation of facilities to locations were calculated and the minimum cost of all combination was determined to be 188 - the same as discovered by our GA. The permutative algorithm also concluded that there is only a single combination which yields the minimum cost as shown below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **location** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **facility** | 6 | 9 | 7 | 8 | 5 | 10 | 4 | 2 | 3 | 1 |

Figure Optimal Solution at cost of 188.

For the sake of discovery, we modified some of the parameters of the GA to understand the impact of such changes upon the speed and probability of convergence. As seen in Figure 1, the GA quickly converged to a minimum within the first few generations and held steady for the rest of the run. The same dynamic repeated for the GA even though the algorithm did not discover this global minimum where for most runs, the GA discovered a local minimum early in the process and held steady at that point. We increased the number of generation to 3000 generations and in small number of cases, the GA was able to dislodge out of a local minimum and discover another solution with better fitness qualification. Better outcomes, however, were attained by increasing the probability of mutation from the 0.05 used for the results shown here to a high value of 0.25 and 0.45. For both such mutation probabilities, the GA was able to quickly discover local minimum and more than 50% of the time leap to a better minimum and finally to the global minimum. We also attempted various probabilities for recombination ranging from 0.6 to 0.9 with no visible improvement over 20 runs of 400 generations each.

# The Algorithm

This project specifies minimizing the cost function for locating 10 facilities on 10 plots of land where each plot can hold a single facility and for a different rate of flow of material between each pair of facilities. Naturally, the total cost of this ecosystem is proportional to the amount of flow of material between pairs of facilities and the distance between such facilities. Ideally, facilities with larger inter-transaction should be located closer to one another. Minimizing the cost function specified in the problem is equivalent to maximizing the fitness function declared above.

**Initialize:**

1. N := 10;

Pm:= 0.05;

Pr := 0.6;

P(0) := {p1 , p2, … p**100**};

**while** (g < 400)

2. F(g) = {f(p1), f(p2), … f(p**100**)};

O1(g) = max(F(g)); // Keep Elite Parent

3. **for** i:=2 to 100 // Generate rest of offspring

P1,P2 = **SelectWithRoulette**(P(g),F(g))

**if** (randn < Pr)

Oi(g) = **RecombineWithRepair (**P1, P2); // 2-point or 1-point xover

**else**

Oi(g) = **MaxFit(**P1, P2);

**end if**

**end for**

4. **for** i:=1 to 100

**if** (randn < Pm)

**MutateUsingSwap(**Oi(g));

**end if**

**end for**

5. P(g+1) = O(g);

g :**=** g + 1;

**end while**

Step 1 is the initialization step where the control parameters of the algorithm are initialized and the initial population is randomly created. The initial population is created by initializing an array of 10 elements to the numbers 1 through 10 and performing a random shuffle to create enough diversity in this initial population. In step 2, the fitness of each individual is computed and the best fit individual is copied to the next generation. Step 3 generates the remaining 99 offspring of the next generation using fitness proportional roulette wheel selection method of two parents and either recombining or selecting the best fit based on the recombination probability ratio, Pr. Two-point crossover recombination operator was used for the results reported in this paper. Step 4 performs mutation on all the offsprings depending on the mutation probability parameter Pm. Finally step 5 purges all the parents of the past generation and reinitializes the population with the new offspring as the parents for the next generation.

# Analysis

For this particular data set of N=10 of the QAP, discovering the solution with exhaustive permutation of all facilities with respect to locations was a simple implementation which yielded the optimal configuration within few seconds of processing. A permutative method for solving the QAP problem becomes unfeasible, however, as the search spaces increases factorially with a linear increase of the data set size N. For our example, 10! permutations were easily computed but when we attempted an N=20 problem, the computation time suddenly became unfeasible for the due date of this project, 20!. For N=10, we learned from the permutative method that our solution is optimal at a cost of 188 and that it is the only solution at this minimal cost.

As indicated above, the algorithm quickly converged to a solution, minimal or otherwise. We also noted that increasing the probability of mutation to an exaggerated amount (0.25 to 0.45) allowed for greater rate of convergence to the global minimum. Introducing such a large mutation parameter most likely allowed the algorithm to escape whatever local optima it would typically land in, and jump to a better local optima until finally reaching the global optima.

For a termination criteria we chose a fixed limit of 400 generations and increasing such limit to 3000 only yielded scant improvement. For this specific problem, it would have been prudent to terminate each run after a number of generations M where no new cost is discovered.

For the results indicated here, we used a 2-point recombination operator. The crossover operator selects a partition from one of the parent's gene to replace the same slice in the other parent's gene. The QAP problem imposes an inherent constraint that there is a *one-to-one* and *onto* relationship between the facilities and locations, and hence, a repair is necessary after recombination. In effect, the selected slice is a specific permutation of the original gene's slice, where a allele of the contributing parent are placed in the corresponding slot of the receiving parent *iff* that same allele existed within the original corresponding partition of the receiving parent. Any other alleles which create a duplicate are *repaired (i.e., replaced)* by selecting one of the alleles which are not present after the recombination. In effect, the new genotype has a unique facility for each location.

We also implemented the single point crossover for recombination which discovered the minimal cost of 188 *more often* than the two point crossover. This can be explained by the assumption that a single point crossover modifies the genotype of the offspring more drastically than the 2-point crossover. On average, a two point crossover operator affects a *smaller* partition of the genotype than the 1-point crossover operator. Such drastic change, of 1-point crossover, has an impact similar to our experimentation of the exaggerated mutation size, noted above, which allows the algorithm to escape local optima to finally discover the global optimum.