Quadratic Assignment Problem

Comparison and Evaluation of Heuristics and Metaheuristics

By Mr. Lok Tze Lun

Table of Contents

1. Introduction

The quadratic assignment problem (QAP) was first introduced by Koopsman and Beckmann in 1957 to model the allocation of indivisible resources [1]. In linear programming, when trying to assign factories to locations, the transportation cost between the factories is ignored and only the rental charges are considered. However, QAP considers both facets in its allocation. In general, in a QAP, a total of n facilities are to be assigned to n locations, with a flow matrix F, whereby f_{ij} is defined as the cost for the activities between facility i and j, and a distance matrix D, with d_{kl} defined as the cost of separation between location k and l. In Koopsman and Beckmann's version, they also include the matrix B which is the cost of assigning facility i to location k, given as b_{ik}. The goal in QAP is to minimize the cost of the assignment. The general equation of the QAP is given below:

$$\min_{\emptyset \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{i\emptyset(i)} + \sum_{i=1}^{n} b_{i\emptyset(i)} \ , i, j = 1, \dots \dots, n \tag{1}$$

The symbol, $\emptyset \in \mathbb{R}^{1 \times n}$ is the permutation in which each column of the vector represents the index of the location and the elements of the vector represents the index of the facility assigned to a location. S_n is the set of all possible permutations for the given assignment problem. In this report, the cost of assigning the facility to a location is not considered, so the term b_{ik} can be omitted. Throughout this report, the objective function of the above QAP is frequently represented in terms of the permutation as $G(\emptyset)$, whereby this function gives the objective function value.

Besides facilities allocation, there are other applications of QAP, namely backboard wiring [2], campus planning [3], ergonomic electronics design [4], scheduling [5] and ranking task [6]. There are few approaches to solve a QAP. One of them is called the exact solutions methods, in which an exact algorithm is used to yield the global optimum solution, such as branch and bound, cutting plane, and branch and cut algorithms. The QAP formula can be linearized into linear formulations to be solved as a mixed integer linear programming problem (MILP). This is possible by introducing new variables and constraints to the problem, with existing linearization methods such as Lawler's linearization [7], Kaufmann and Broeckx's linearization [8], Frieze and Yadegar's linearization [9], and Adams and Johnson's linearization [10], in which [10] gets the lower bound for the branch and bound method.

However, according to Sahni and Gonzalez, QAP is a NP-hard problem and even approximation methods cannot solve it within polynomial time [11]. The exact algorithms from earlier aren't able to solve problems of size $n > 20$ within a reasonable computational time. It is also not efficient to solve the QAP with MILP given the additional constraints and variables imposed to the problem. Therefore, researchers looked into heuristics which are able to solve larger QAP problem at a better time compare to that using exact algorithms. The problem of some heuristics is they tend to get trap in the local optima, which such occurrence is reduced or solved through the use of metaheuristics, a subset of heuristics. Some examples of heuristics are the particle swarm optimization (PSO), ant colony optimization (ACO), genetic algorithm (GA), tabu search (TS), iterative improvement search (II), simulated annealing (SA), Greedy Randomized Adaptive Search Procedure (GRASP) and iterative local search (ILS). These heuristics usually take over the search task by using an initial solution, and this solution can be generated using trivial or sophisticated methods.

The aim in this report is to compare and evaluate four different heuristics method, which are the iterative-improvement-based local search with dynamic 2-/3-Opt, simulated annealing, genetic algorithm and tabu search, using problem instances from QAPLIB [12]. Four of these algorithms will receive initial solutions generated from two different methods. All four algorithms and the initial solution generators are written from scratch with C++ programming language using an object-oriented approach such that the users can easily use the algorithms by including the QAPSolver header file. Examples of code is given in the Appendix.

2. Designing the Algorithm

This section gives a brief introduction to every algorithm and explains the structure and concept of every algorithm's implementation. A pseudocode is included for every algorithm.

2.1 Initial Solution Generator

Intuitively, the initial solution has an influence over the performance of the heuristics. A good starting solution can help the algorithm to converge to an optimum solution quickly and vice versa. Despite that, the initial solution might cause certain algorithms to start from a position that eventually traps themselves in a local optimum. Further comments on the initial solutions will be given in the later sections. In general, there are three methods to generate an initial solution, such as the random generator, greedy algorithm, and the trivial method. The trivial method is usually just the assignment of permutations without any computations. For example, for a QAP of size $n = 12$, the trivial solution can be given as the permutation, $\emptyset = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)$. Random generators are good estimators such that they are easy to implement. Greedy algorithms are algorithms that uses a greedy strategy in which they take a path that is believed to minimize the cost of the QAP. In this report, both random generator and greedy algorithms are used to generate the initial solution.

2.1.1 Iterative Random Generator

As mentioned earlier, random generator is easy to implement. Moreover, they are fast if the generator only runs once. However, as the generator is heavily dependent on randomness, the goodness of the initial solution fluctuates and this fluctuation is larger if the first solution from the generator is taken directly as the initial solution. Hypothetically, it is possible to reduce such fluctuations by running the generator over multiple iterations and choose the best permutation. This simple generator can be devised by randomly shuffling the current permutation at every iteration and if the current permutation gives a smaller objective function value than the current best one, it will replace the current best permutation. This algorithm is also implemented in [13]. In fact, it is possible to achieve the global optimum if this generator is run for infinitely long time. Below shows the pseudocode for the iterative random generator (IRG):

Algorithm 1 : *Iterative Random Generator*

```
1:   Input: number of iterations, $iter$
2:   $\emptyset \leftarrow$ Generate the initial permutation
3:   $\emptyset_{best} \leftarrow \emptyset$
4:   $G_{best} \leftarrow$ a large number
5:   While $iter$ not reached do
6:       $\emptyset \leftarrow$ Shuffle the elements of $\emptyset$ randomly
7:       If $G(\emptyset)$ is smaller than $G_{best}$ then
8:           $G_{best} \leftarrow G(\emptyset)$
9:           $\emptyset_{best} \leftarrow \emptyset$
10:      End if
11:  End while
12:  Return $\emptyset_{best}$
```

2.1.2 Greedy Generator

There are many greedy strategies one can impose to generate the initial solution. For instance, in the travelling salesman problem (TSP), one greedy strategy would be to select the next location that has the shortest distance from the current location. In this report, a very simple and similar greedy heuristics is used to guide the assignment, that is, the facility with the minimum sum of flow is assigned to a location that has a maximum sum of distance. This strategy is called the Minimum Flow to Maximum Distance (MFMD) greedy strategy. To get a better understanding, an example is demonstrated in Figure 1.

Flow matrix, F

Facility 1	0	12	8	10	Sum = 30
Facility 2	12	0	5	6	Sum = 23
Facility 3	8	5	0	9	Sum = 22
Facility 4	10	6	9	0	Sum = 25

Distance matrix, D

Location 1	0	100	50	125	Sum = 275
Location 2	100	0	150	30	Sum = 280
Location 3	50	150	0	90	Sum = 290
Location 4	125	30	90	0	Sum = 245

Figure 1: An example to demonstrate the concept of MFMD.

Based on the information from Figure 1, Facility 3 will be first assigned to Location 3, then Facility 2 with Location 2 and so on. The resulting permutation would be $(4, 2, 3, 1)$. If there is duplicate minimum or maximum sum, the facility or location is selected based on their order of precedence in the array. Unlike IRG, the proposed greedy generator will always generate the same permutation given the same problem instance. This consistency is very valuable especially when trying to get an accurate performance comparison between the same heuristic algorithm but with different hyperparameters. Below shows the high-level pseudocode for the proposed greedy generator (GG). Note that the facility index and location index start from 1 to n, whereas array indexing starts from 0 to $n-1$ in C++ or any similar programming language. To avoid confusion, in the pseudocode (including future pseudocode), array indexing is treated to start from 1 to n, as well, like that in MATLAB.

Algorithm 2 : *Greedy Generator*

```
1:   $F_{sum} \leftarrow$ array containing the sum of each row of matrix $F$
2:   $D_{sum} \leftarrow$ array containing the sum of each row of matrix $D$
3:   $I_F \leftarrow$ array containing the facility index for each element of $F_{sum}$
4:   $I_D \leftarrow$ array containing the location index for each element of $D_{sum}$
5:   Initialize $\emptyset$ as an empty array of size, $n$
6:   While not all facilities are assigned a location do
7:       $I_{Fmin} \leftarrow$ value of $I_F$ corresponding to the minimum of $F_{sum}$
8:       $I_{Dmax} \leftarrow$ value of $I_D$ corresponding to the maximum of $D_{sum}$
9:       $\emptyset[I_{Dmax}] \leftarrow I_{Fmin}$
10:      Remove element associated with $I_{Fmin}$ from $F_{sum}$ and $I_F$
11:      Remove element associated with $I_{Dmax}$ from $D_{sum}$ and $I_D$
12:  End while
13:  Return $\emptyset$
```

2.2 Heuristics & Metaheuristics

With the initial solution generated from the IRG or the GG algorithm, the algorithms in this subsection will try and converge to an optimum solution. Four algorithms will compete with each other to determine which algorithm gives the best solution and run time. The beauty of heuristics and metaheuristics is the needlessness of defining a good set of constraints, and at the same time, able to outperform exact solution methods if tuned correctly and used for the right QAP.

2.2.1 Local Search with Dynamic Neighbourhood Operators

The local search in this report is based on an iterative improvement algorithm that keeps on searching the neighbourhood until no better solution can be attained. The algorithm starts with an initial solution as the current and best solution and a neighbourhood operator is executed to yield all the neighbours of the current solution. Each neighbour is a permutation actively compared to the current solution and if the neighbour is found to have lower cost than the best solution, it will be made the best solution. Current solution is updated with the best solution based on some improvement methods. One of them updates the current solution as soon as any solution from the neighbourhood surpasses the best solution. This is called the first improvement. It is faster than its peer, the best improvement, which only update the current solution after searching through the whole neighbourhood to find the best candidate. Clearly, best improvement is likely to outperform first improvement since best improvement doesn't overlook undiscovered search space within the neighbourhood. There is another improvement method called the Heider's rule [14] but it will not be used in this report. The neighbourhood operators used in here are the 2-Opt and 3-Opt methods, which is briefly but intuitively illustrated in Figure 2.

Conventionally, the algorithm stops as soon as the current solution is still the same as the best solution. However, the proposed algorithm here interchanges the neighbourhood operator when the improvement plateaus off in the hope that the algorithm could explore a different search space for new improvements. The algorithm could continue the same neighbourhood operator perpetually if there are always improvements and terminate once changing the neighbourhood operator still yield futile results.

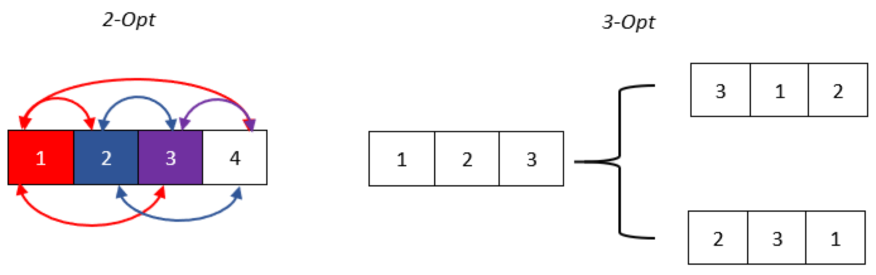


Figure 2: Simple permutation demonstrating the 2-Opt and 3-Opt neighbourhood operators.

The 2-Opt operator takes a simple pair-wise exchange whereas for the 3-Opt method, there are two ways to exchange three indices, which yield two different neighbours. The proposed algorithm uses the approach that leads to the top right solution as illustrated in Figure 2. To conclude this subsection, the pseudocode for the local search (LS) with iterative improvement and dynamic neighbourhood operator algorithm is given. The inputs to the algorithm are the initial solution, $init$, improvement method, $method$ and the starting neighbourhood operator, $mode$.

Algorithm 3 : *Local Search*($init, method, mode$)

```
1:  Initialize current solution, $S_c \leftarrow init$
2:  Initialize best solution, $S_{best} \leftarrow init$
3:  Initialize $repeat \leftarrow 1$
4:  While search not done do
5:      $S \leftarrow N(S_c)$ using neighbourhood operator from $mode$
6:      For each $s \in S$ do
7:          If $G(s)$ is smaller than $G(S_{best})$ then
8:              $S_{best} \leftarrow s$
9:              If $method$ is first accept then
10:                 Terminate the for loop
11:             End if
12:         End if
13:     End for
14:     If $S_c$ is equal to $S_{best}$ then
15:         Increment $repeat$ by 1
16:         Change $mode$ to the other neighbourhood operator
17:     Else
18:         Reset $repeat$ to 1
19:         $S_c \leftarrow S_{best}$
20:     End if
21:     If $repeat$ is equal to 3 then
22:         Terminate the search
23:     End if
24: End while
25: Return $S_{best}$
```

2.2.2 Simulated Annealing

Simulated annealing is a physics-inspired metaheuristic, in which the term "annealing" refers to a heated solid will be susceptible to changes till it cools down. In the case of solving a combinatorial optimization problem like QAP, simulated annealing allows poor solutions to be accepted with a probability that decreases with temperature. This gives the algorithm a chance to escape from a local optimum through this hill-climbing method. The probability for the acceptance criterion is given by the Metropolis criterion [15] in (2), given the current temperature, T, the best solution, S_{best} and a neighbour solution, s:

$$Probability\ of\ Acceptance, P = \begin{cases} 1, & if\ G(s) < G(S_{best}) \\ exp^{\left(-\frac{G(s) - G(S_{best})}{T}\right)}, & if G(s) > G(S_{best}) \end{cases} \tag{2}$$

In this report, the simulated annealing algorithm is devised by first accepting an initial solution, which is also treated as the best solution. At the current temperature, T, a 2-Opt neighbourhood operator is used to search the neighbours of the best solution. Since at any temperature level, the solid could undergo multiple changes instead of just one, this simulated annealing algorithm takes in a user-defined number of iterations, $iter$ as input. For "$iter$" number of times at temperature T, a neighbour from the current solution space is selected randomly and it is compared to the best solution. Using the acceptance criteria in (2), the algorithm decides whether to update the best solution with the neighbour solution. Once $iter$ number of comparisons are done, the next temperature level is reached by reducing the current temperature with an annealing schedule, in this case, the algorithm uses a geometric cooling schedule as simple as that highlighted in (3), with alpha α being a user-defined parameter between 0 to 1:

$$T_{n+1} = \alpha T_n \tag{3}$$

This process continues until T reaches its final temperature. The special point of this simulated annealing algorithm is that once the final temperature is reached, the algorithm restarts itself with the final best solution as the new initial solution and the number of restarts is also a user-defined hyperparameter. The pseudocode for the proposed simulated annealing (SA) algorithm is given below. The inputs of the algorithm are the initial solution, $init$, number of restarts, $nRestart$, number of iterations per temperature level, $iter$, initial temperature, T, final temperature, T_{final} and the rate of cooling, α.

Algorithm 4 : *Simulated Annealing*($\boldsymbol{init, nRestart, iter, T, T_{final}, \alpha}$)

```
1:   Initialize best solution, $S_{best} \leftarrow init$
2:   **While** $nRestart$ not reached **do**
3:       **While** $T_{final}$ not reached **do**
4:           $S \leftarrow N(S_{best})$ using a neighbourhood operator
5:           **For** $iter$ number of times **do**
6:               $s \leftarrow$ randomly pick one neighbour from $S$
7:               **If** $G(s)$ is smaller than $G(S_{best})$ **then**
8:                   $S_{best} \leftarrow s$
9:               **Else**
10:                  $P \leftarrow \exp(-[G(s) - G(S_{best})]/T)$
11:                  $S_{best} \leftarrow s$ with a probability of $P$
12:              **End if**
13:          **End for**
14:          $T \leftarrow \alpha T$
15:      **End while**
16:  **End while**
17:  **Return** $S_{best}$
```

2.2.3 Genetic Algorithm

Genetic algorithm is one of the population-based search algorithms inspired by Darwin's theory on the survival of the fittest. There are many articles and techniques corresponding to genetic algorithm and a good review journal on these topics was done by [16]. The population in genetic algorithm is comprised of chromosomes and each chromosome carries genes/alleles. Conventionally, the genetic algorithm uses natural selection, in which the strongest chromosome carrying the best genes will survive in the population and is a candidate for breeding. However, the proposed algorithm here takes an unbiased and random selection process which is detailed in the next few paragraphs.

The slogan, "the strong will prevail", is very much a survival instinct for mankind. However, in search algorithms, emphasizing only on strong solutions might lead the algorithm into a tunnel vision. In general, a good balance between "exploration" and "exploitation" is required to avoid trapping oneself in a local optimum. Exploration aims to create a diverse population whereas exploitation aims to steer the chromosomes towards a stronger population. Generically, a genetic algorithm begins by creating an initial population of size to be determined by the users. The chromosome here is the permutation, $\emptyset$ and the genes represent the facility index whereas the index of the gene represents the location index. Next, two parents will be selected from the list for breeding via crossover. Through this process, two offspring are born and mutation will occur either probabilistically or certainly. The algorithm will decide how and whether to fit the offspring into the population by displacing the incumbent chromosomes in the list. These procedures continue until a stopping criterion is met, in this case, once the maximum number of generations is reached. Each process is detailed below:

Generating the Initial Population:

The proposed genetic algorithm promotes fairness. Hence, the initial chromosomes/permutations that make up the population are generated by the same iterative random generator (IRG) detailed in Algorithm 1. So, if the population size is 50, the IRG algorithm is run 50 times to generate 50 chromosomes/permutations for the population.

Crossover:

Crossover exchanges genes between two parents from the population and usually only the strongest/fittest chromosome is selected as parents. In QAP, the fittest chromosome is the one with the lowest objective function value. Hence, the objective function value is used as the fitness value. However, the parents from the population are selected randomly for this algorithm, to promote diversification and to avoid premature convergence to a local optimum. As for the crossover process, a gene is selected randomly from parent 1 and it will be exchanged with the gene from parent 2 located at the same index as the gene from parent 1. Since the chromosomes must contain genes that are not repeatable in the permutation, the repeated genes in parent 1 and parent 2 must be transposed too, as shown in Figure 3. The algorithm takes in the number of crossovers as one of the parameters such that the selected parents could exchange genes randomly for multiple times.

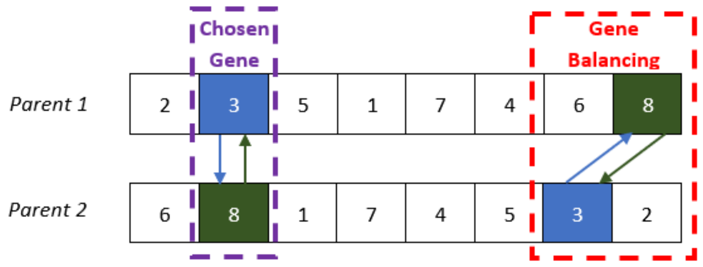


Figure 3: The crossover method employed in the proposed genetic algorithm.

Mutation:

Through the crossover process, parent 1 and parent 2 will become child 1 and child 2 or offspring, whichever one prefers. Mutation is also important in diversifying the population by discovering new search spaces. The written genetic algorithm allows user to define the probability of mutation, to control the frequency of mutation. The mutation process itself is just a simple local pair-wise exchange between two randomly selected gene in each child chromosome. Like the crossover process, there is an input parameter where users can choose how many times mutation occurs within a child.

Fitting the Child into the Population:

Fitting a child into the population requires removing one existing chromosome in the population. This isn't a trivial decision since replacing the chromosomes randomly with the child might slow down improvement and convergence. At the end of the day, some form of elitism is needed to ensure the algorithm is exploring and improving. In this algorithm, both children don't directly replace their parents, instead they follow two acceptance rules:

1. If any child is fitter (lower cost) than the existing chromosome with the highest fitness value (highest cost) in the population, the child immediately replaces that chromosome.
2. If not, the child could still replace the weakest chromosome in the list, but with a probability.

From Rule 2, the acceptance criterion is heavily inspired from Simulated Annealing, to encourage diversity in the population. This criterion is what makes this genetic algorithm unbiased since everyone is given a chance to prevail. The acceptance probability here isn't temperature dependent, but it decreases periodically based on the generation numbers. For this algorithm, given a total number of generations, $nGen$ and a starting acceptance probability, P_{accept} of 1, the probability P_{accept} is decreased by 0.25 after every $nGen/5$ generations. In other words, if $nGen$ is 500, then the probability is reduced at the 100th, 200th, 300th, 400th and 500th generation, which implies that exploitation is emphasized as the algorithm approaches the end.

Below is a high-level pseudocode for the proposed random and unbiased genetic algorithm (GA). The inputs of the algorithm are the max number of generations, $nGen$, population size, s, mutation rate, r_{mut}, number of crossovers, N_{cross}, number of mutations, N_{mut} and the number of iterations for initial solution generator, $iter$.

Algorithm 5 : *Genetic Algorithm*($\boldsymbol{nGen, s, r_{mut}, N_{cross}, N_{mut}, iter}$)

```
1:   Generate population, $P$ of size $s$ using iterative random generator (IRG)
2:   $f \leftarrow$ the fitness value for each chromosome in the population
3:   Initialize acceptance probability, $P_{accept} \leftarrow 1$
4:   **While** $nGen$ not reached **do**
5:       $p_1, p_2 \leftarrow$ Select randomly two parents from $P$
6:       $c_1, c_2 \leftarrow$ crossover($p_1$), crossover($p_2$) for $N_{cross}$ times
7:       **If** mutation is triggered with probability of $r_{mut}$ **then**
8:           $c_1, c_2 \leftarrow$ mutate($c_1$), mutate($c_2$) for $N_{mut}$ times
9:       **End if**
10:      $P_{accept} \leftarrow P_{accept} - 0.25$ after every period of $nGen/5$ generations
11:      **For** each child, $c \in \{c_1, c_2\}$ **do**
12:          **If** $G(c)$ is smaller than max($f$) **then**
13:              Replace element of max($f$) in $P$ with $c$
14:              Update $f$ to include $G(c)$
15:          **Else**
16:              Replace element of max($f$) in $P$ with $c$ with a probability $P_{accept}$
17:              Update $f$ to include $G(c)$ if accepted
18:          **End if**
19:      **End for**
20:  **End while**
21:  **Return** $\emptyset \in P$ that has the min($f$)
```

2.2.4 Tabu Search

This is a metaheuristics introduced by Glover and Laguna [17] in which “tabu” refers to the word taboo or prohibition. Most of the details were explained in their handbook, which is a good source for deeper understanding on the algorithm. Essentially, tabu search makes a move or a solution taboo such that the algorithm cannot choose the same moves or solutions temporary or permanently. This was to encourage the algorithm not to cycle back to the same search space, in which such cyclic actions inhibit exploration and the potential of finding a better solution. The tabu search in this report uses a recency-based attributive memory, which is a short-term memory that tracks the recent moves undertaken by the algorithm which are tabu-active (a term to mean that the moves are prohibited). As it is an attributive memory, only the moves are tracked.

In summary, tabu search operates by first using a move operator to find the neighbours of the current solution, whereby each of these neighbours are added to the candidate list, under the condition that they are not tabu-active or if they satisfy an aspiration criterion. Then, the candidates in that list are compared to the best solution and if there exists a candidate which surpasses the best solution, that candidate will replace the current best solution. Otherwise, the best candidate from the list will be made the current solution which will be explored by the algorithm in the next iteration and the move that gets this candidate will be branded tabu-active. The search stops after a number of iterations. Being a short-term memory tabu search, tabu-active moves are temporary and the period of being tabu-active is called a tabu tenure. Comparing to normal descent methods, where the algorithm stops once no better solution is found, tabu search continues searching using the best candidates available, even though they aren’t better than the best solution. Now that the structure of the tabu search is explained, four domains constituting the search algorithm, namely the move operator, tabu attributes, tabu tenure and the aspiration criterion are explored in detail.

<u>Move operator and tabu attributes:</u>

The move operator used in this tabu search is just a 2-Opt pair-wise exchange operator. This move will be made tabu-active if it is the move that gets the best candidate in the current iteration, hence the move is the tabu attribute. Let’s show a minimization problem example of how a move is made tabu.

		Moves		Candidates	Score	Best Score
Current Solution (1, 2, 3, 4, 5, 6, 7, 8)	→	(3, 6)	→	(1, 2, 6, 4, 5, 3, 7, 8)	32	19
		(2, 8)	→	(1, 8, 3, 4, 5, 6, 7, 2)	54	
		(5, 6)	→	(1, 2, 3, 4, 6, 5, 7, 8)	20	
		(1, 3)	→	(3, 2, 1, 4, 5, 6, 7, 8)	41	

According to the example, the best candidate (in red) is via move (5, 6) with a score (objective function value) of 20. Although the score isn’t lower (better) than the best score, it will still be chosen as the solution to be explored in the next iteration. Therefore, the move (5, 6) and its mirror pair (6, 5) will be added to a tabu list to imply that they are tabu. As long as this status is active, if the 2-Opt operator reaches one of these moves, it will ignore it in a way the move will not be executed, unless an aspiration criterion is reached, which will be covered next.

<u>Aspiration Criterion:</u>

By default, aspiration criterion has the power to remove the tabu-active status of a move, should there be no moves left to take if every move is tabu-active. However, the aspiration criterion doesn’t necessarily work this way. In this tabu search, at any iterations, if a move is found to be tabu, the objective function value of the resulting solution of the move is still evaluated and if this value is found to be better than that from the best solution, the move is revoked off its tabu status and will be added to the candidate list. This is known as aspiration by objective and it will be used in this tabu search.

Tabu tenure:

The tabu tenure here is a semi-random systematic dynamic tenure, whereby the tenure values come from a range of values. For instance, given a minimum tenure value, $t_{min} = 4$ and a maximum tenure value, $t_{max} = 10$, the tabu tenure will have a range of { 4, 5, 6, 7, 8, 9, 10 }. The tabu-active move and its mirror pair will be assigned the same tenure value in sequential order. Once the end of the tabu tenure range is reached, the current sequence of tenure values is randomly shuffled and the tenure values are assigned sequentially again. To facilitate a more efficient tenure checking, the following shows how the tenure value is assigned to each tabu-active move at any iteration, $iter$.

$$tabutenure(move) = iter + tenure\ value$$

Compare to assigning the tenure values directly, the above method retains memory of when the move was branded tabu. The tabu-active status of the move is revoked when the following condition is met:

$$current\ iteration > tabutenure(move)$$

Below shows the high-level pseudocode for the tabu search (TS) algorithm. This algorithm only takes in the initial solution, $init$, the maximum number of iterations, i_{max}, the minimum tabu tenure, t_{min} and the maximum tabu tenure, t_{max} as the input arguments.

Algorithm 6 : *Tabu Search*($\mathbf{init}, \mathbf{i_{max}}, \mathbf{t_{min}}, \mathbf{t_{max}}$)

```
1:   Initialize best solution and current solution, $S_{best}, S_c \leftarrow init$
2:   Initialize tabu list, $L_{tabu}$ and tabu tenure list, $L_{tenure}$
3:   Initialize candidate list, $L_{cand}$ and move list, $L_{move}$
4:   While $i_{max}$ not reached do
5:       Update($L_{tenure}, L_{tabu}$) for tenure expiration
6:       For $s \in N(S_c)$ using a neighbourhood operator do
7:           $m \leftarrow$ the current move producing $s$
8:           If $m$ and its mirror pair, $m^*$ are not in $L_{tabu}$ then
9:               Add $s$ to $L_{cand}$ and $m$ to $L_{move}$
10:          Else if aspiration criterion is met
11:              Update($L_{tenure}, L_{tabu}$) to remove $m$ and $m^*$
12:              Add $s$ to $L_{cand}$ and $m$ to $L_{move}$
12:          End if
13:      End for
14:      $b \leftarrow$ best($L_{cand}$)
15:      $m_b \leftarrow$ move $\in L_{move}$ corresponding to solution $b$
16:      If $G(b)$ is smaller than $G(S_{best})$ then
17:          $S_{best} \leftarrow b$
18:      End if
19:      $S_c \leftarrow b$
20:      Add $m_b$ and $m_b{}^*$ to $L_{tabu}$
21:      $t_1, t_2 \leftarrow$ TenureAssign($m_b, t_{min}, t_{max}$), TenureAssign($m_b{}^*, t_{min}, t_{max}$)
22:      Add $t_1$ and $t_2$ to $L_{tenure}$
23:      Empty($L_{cand}, L_{move}$)
24:  End while
25:  Return $S_{best}$
```

3. Tuning the Algorithm

Some of the algorithms presented in this report have many hyperparameters. The value these hyperparameters take will influence the performance of the algorithm. Hence, the algorithm must be tuned to its optimum operation and the data used for tuning will be these four problem instances from QAPLIB : Had12 by Hadley [18], Rou12 by Roucairol [19], Nug12 by Nugent [20] and Scr12 by Scriabin [21]. These problem instances already have their global optimum solution found, so it is good to use for evaluation purposes. Each algorithm is tuned to perform as well as possible in these four problem instances respectively.

There must be a score to evaluate how well the algorithm is performing. Just simply recording the time or attempts the algorithm takes to reach global optimum is unfair because the use of randomness in many of the algorithms here will cause inconsistency in their performance. The best practise here is to run each algorithm over again and get the average score. Three scoring criteria will be used, which are the average deviation from the global optimum, Dev, the average run time, $Runtime$ and the frequency of reaching global optimum, $fOpt$. Average deviation from the global optimum is calculated as shown below, with G being the objective function value or the solution value and N is the number of iterations a test is rerun:

$$Dev = \frac{1}{N}\frac{\sum_{i=1}^{N}\left(G_i - G_{opt}\right)}{G_{opt}} \times 100\% \tag{4}$$

The frequency of reaching global optimum is calculated as shown below:

$$fOpt = \frac{number\ of\ times\ algorithm\ reach\ global\ optimum}{N} \times 100\% \tag{5}$$

When selecting the best hyperparameters for each algorithm, there are three important standards to adhere to:

1. If possible, the configuration that yields the global optimum at least once is preferrable. This is prioritised first before evaluating the scores.
2. The deviation, Dev is the most prioritised score among the rest. In general, the lowest Dev of an algorithm will be selected as the best.
3. However, should there be a relatively equal score in terms of Dev but a drastically different run time, the combination of tuning parameters that yield the faster run time is preferred.

The $fOpt$ score is there to evaluate the consistency of the algorithm in achieving the global optimum, which is greatly affected by the randomness of the algorithm, hence it isn't used to evaluate the best tuning parameters for the algorithm. The average run time includes both the time needed to generate initial solution and the time to solve the QAP and it will be longer than normal run time because the program will be run in "Debug" mode in Visual Studio 2019. If debugging is not needed, run the program in "Release" mode instead. Despite that, the run time for each algorithm is still reflecting the actual one, so it is still valid for comparison.

Throughout every tuning test, the test is repeated for $N = 100$ times and every result depicted in the table is an average value of that. As this is the tuning stage, the best permutation and best solution will not be included in the table.

3.1 Iterative Random Generator

The goal here is to tune the random generator such that a good starting solution is generated without exhausting significant computational time. The generator is tuned over iterations of 1, 10, 100, 500, 1000 and 10,000. Below shows the result of the tuning.

Table 1: Tuning results for the Iterative Random Generator (IRG)

Num of	*Had12*			*Rou12*			*Nug12*			*Scr12*		
Iterations	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	11.958	0.000018	0	26.480	0.000015	0	42.907	0.000017	0	44.998	0.000021	0
10	7.431	0.000094	0	23.481	0.000115	0	23.875	0.000097	0	40.624	0.000091	0
100	5.633	0.000782	0	15.977	0.000861	0	17.907	0.000969	0	30.640	0.000908	0
500	5.380	0.005117	0	15.116	0.003936	0	19.031	0.004289	0	20.999	0.004552	0
1000	4.435	0.008745	0	10.370	0.009121	0	13.869	0.008145	0	17.201	0.008552	0
10000	3.103	0.079203	0	9.774	0.087839	0	8.536	0.095125	0	13.885	0.087760	0

As expected, increasing the number of iterations will yield solutions of lower cost, as depicted by the decreasing deviation in Table 1. Since with 100 iterations, a reasonable deviation is attainable with a run time of roughly 1 millisecond, which will not deteriorate the performances of most of the algorithms used in this study, this number of iterations will be the standard parameter used for the IRG to generate initial solutions for the search algorithm.

3.2 Greedy Generator

There is no hyperparameter to tune for the greedy generator. Moreover, scores depicted in the result table, Table 2 are what the generator could consistently output.

Table 2: Quality of Initial Solution from the Greedy Generator (GG)

Problem	Dev (%)	Runtime (s)	fOPT (%)
Had12	16.420	0.000151	0
Rou12	24.178	0.000131	0
Nug12	39.792	0.000141	0
Scr12	101.216	0.000128	0

From Table 2, it clearly shows that IRG outperforms GG in terms of the quality of initial solution and computational time. However, when tuning the search algorithms in the later section, Tabu Search was found to obtain global optimum without fail with GG as the initial solution generator when tested with the Rou12 problem instance. Therefore, GG will not be discarded and shall be involved in tuning the search algorithms.

3.3 Local Search with Dynamic Neighbourhood Operators

Like GG, it isn't cumbersome to tune the local search algorithm as it doesn't exactly have hyperparameters to tune. One of the tuning operations is to compare the first improvement method to the best improvement method and find out which methods give the best performances. Besides, the neighbourhood solution is different for the 2-Opt and 3-Opt transposition. So, experiments were conducted to test whether starting the local search with a 2-Opt or 3-Opt transposition will influence the performances. Table 3 shows the result of both tuning experiments.

Table 3: Tuning result of Local Search (LS). Top Left : Local search with IRG and start with 2-Opt. Top Right : LS with IRG and start with 3-Opt. Bottom Left : LS with GG and start with 2-Opt. Bottom Right : LS with GG and start with 3-Opt.

LS with IRG (start with 2-Opt)				
Problem	Acceptance	Dev (%)	Runtime (s)	fOPT (%)
Had12	First	7.143	0.001828	0
	Best	1.472	0.004824	0
Rou12	First	17.155	0.001408	0
	Best	4.143	0.007080	0
Nug12	First	8.305	0.001428	0
	Best	2.076	0.005928	0
Scr12	First	16.199	0.001420	0
	Best	10.767	0.006403	0

LS with IRG (start with 3-Opt)				
Problem	Acceptance	Dev (%)	Runtime (s)	fOPT (%)
Had12	First	4.019	0.001176	0
	Best	0.467	0.007574	0
Rou12	First	13.259	0.001188	0
	Best	6.376	0.007727	0
Nug12	First	13.149	0.001552	0
	Best	6.308	0.007497	0
Scr12	First	17.447	0.001906	0
	Best	5.265	0.007554	0

LS with GG (start with 2-Opt)				
Problem	Acceptance	Dev (%)	Runtime (s)	fOPT (%)
Had12	First	9.080	0.001293	0
	Best	0.726	0.006893	0
Rou12	First	18.480	0.000631	0
	Best	9.484	0.005386	0
Nug12	First	16.263	0.001253	0
	Best	5.536	0.006295	0
Scr12	First	21.579	0.000885	0
	Best	7.202	0.003926	0

LS with GG (start with 3-Opt)				
Problem	Acceptance	Dev (%)	Runtime (s)	fOPT (%)
Had12	First	7.506	0.000799	0
	Best	1.090	0.009948	0
Rou12	First	20.394	0.000493	0
	Best	6.796	0.006780	0
Nug12	First	17.647	0.000844	0
	Best	3.806	0.007602	0
Scr12	First	13.009	0.001568	0
	Best	6.272	0.006263	0

From Table 3, it is proven that best improvement method yields better solution quality than first improvement method, but at a cost of a higher computational time. Using the results from LS with initial solution generated by GG as they are always consistent, the performance of LS on Rou12, Nug12 and Scr12 improves when the algorithm uses 3-Opt initially. In general, it is also notable that the algorithm runs slightly slower when it starts with the 3-Opt transposition. Given the solution from GG, it is clear that the proposed LS algorithm cannot reach a global optimum. On the other hand, there are very small occasions whereby the LS with a random initial solution is able to reach global optimum several times, when the test is repeated few times outside the main experiment. Such phenomenon isn't due to the LS's innate ability to escape from the local optima, instead the IRG had generated the perfect initial solution to channel the LS to the direction of the global optimum. Rows highlighted in green are the best configuration of the LS for the given problem instances and initial solution generator. Table 4 shows the summary of these configurations.

Table 4: Summary of LS's best configurations for each problem instance given the initial solution generator it uses.

Problem Instances	**LS with IRG**	**LS with GG**
Had12	Best accept. Start with 3-Opt	Best accept. Start with 2-Opt
Rou12	Best accept. Start with 2-Opt	Best accept. Start with 3-Opt
Nug12	Best accept. Start with 2-Opt	Best accept. Start with 3-Opt
Scr12	Best accept. Start with 3-Opt	Best accept. Start with 3-Opt

3.4 Simulated Annealing

There are many hyperparameters to tune for Simulated Annealing (SA). Best practise is to perform a grid-search based tuning process which screens through all possible combinations of hyperparameters available in the algorithm, at a hefty computational cost. To save time, only the main hyperparameters are tuned. These are the number of restarts, $numRestart$, the maximum number of iterations, $\max iter$, and the initial starting temperature, $T_{initial}$. The $numRestart$ allows the solution to converge to a better one if there are enough restarts but will also alter an already reasonable solution if the algorithm restarts in excess. The purpose of tuning $\max iter$ is to find the number of iterations on each temperature level such that enough neighbours are picked randomly to draw out the successor to the current best solution. Lastly, $T_{initial}$ controls the leniency of accepting a bad solution. If $T_{initial}$ is high, the algorithm will lose information on many good solutions, especially when it restarts with a good solution from the previous annealing process.

The remaining hyperparameters are kept constant, which are $T_{final} = 0.01$ and $\alpha = 0.85$. As SA is very stochastic, it can be seen in the results of some problem instances that a much larger number of restarts is needed for the algorithm to converge, which will take exponentially longer to compute. In such cases, the tuning process will not continue further and the best configuration is selected, even though the deviation is huge. The $numRestart$ and $\max iter$ are tuned first, with $T_{initial} = 100$ for all problem instances. The $T_{initial}$ is tuned if and only if multiple good performing configurations are identified and further shortlisting is desired. Table 5 to Table 8 shows the tuning result using SA with IRG and SA with GG. Ones in yellow are entitled for initial temperature tuning.

Table 5: Tuning Result for number of restart and max iterations for SA with IRG

Problem instance : *Had12* ; Initial Solution Generator : IRG; T_initial : 100; T_final : 0.01; alpha : 0.85

numRestart / max Iter	1			10			50			100		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	3.377	0.018843	0	3.992	0.211156	0	3.500	1.035210	0	3.677	2.053200	1
5	1.280	0.021610	0	1.412	0.242559	6	1.226	1.169920	6	1.410	2.336270	4
20	0.890	0.031920	0	0.900	0.346754	9	0.805	1.717100	11	0.765	3.544490	17
50	0.833	0.055621	0	0.799	0.544843	13	0.812	2.906990	10	0.832	5.795090	14

Problem instance : *Rou12* ; Initial Solution Generator : IRG; T_initial : 100; T_final : 0.01; alpha : 0.85

numRestart / max Iter	1			10			50			100		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	13.519	0.018888	0	5.978	0.221433	0	5.354	0.959765	2	5.792	1.951340	1
5	6.413	0.021014	0	5.063	0.195667	0	5.118	1.170360	1	5.353	2.361850	0
20	6.498	0.029902	0	5.801	0.309683	0	4.480	1.680870	1	4.382	3.630960	1
50	6.202	0.052975	0	5.274	0.548687	0	4.171	2.869760	1	3.472	5.826490	1

Problem instance : *Nug12* ; Initial Solution Generator : IRG; T_initial : 100; T_final : 0.01; alpha : 0.85

numRestart / max Iter	1			10			50			100		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	9.855	0.019965	0	12.678	0.214505	0	10.934	1.028410	0	11.744	2.070230	1
5	6.540	0.021923	0	5.574	0.242688	0	5.709	1.170290	4	6.024	2.247990	3
20	2.284	0.030788	0	3.727	0.363634	2	3.540	1.705130	2	3.595	3.512390	3
50	4.000	0.061163	0	3.242	0.577965	10	3.391	2.861580	1	3.263	5.754030	4

Problem instance : *Scr12* ; Initial Solution Generator : IRG; T_initial : 100; T_final : 0.01; alpha : 0.85

numRestart / max Iter	1			10			50			100		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	14.693	0.018446	0	6.606	0.183878	18	6.346	1.014670	6	5.773	2.036730	10
5	5.654	0.020675	0	6.506	0.234703	8	5.649	1.208900	4	5.002	2.307740	10
20	5.944	0.030973	0	5.472	0.330519	8	4.493	1.750080	11	4.181	3.550370	14
50	8.423	0.063077	8	5.170	0.590151	12	4.458	2.877510	14	3.869	5.770350	13

Table 6: Tuning Result for number of restart and max iterations for SA with GG

Problem instance : *Had12* ; Initial Solution Generator : GG; T_initial : 100; T_final : 0.01; alpha : 0.85												
numRestart	1			10			50			100		
max Iter	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	4.747	0.018697	0	2.975	0.188598	0	3.674	1.071540	0	3.259	2.125660	0
5	1.770	0.021639	0	1.338	0.227514	6	1.280	1.190890	5	1.401	2.442530	2
20	1.007	0.030336	4	1.041	0.346443	14	0.903	1.745060	5	0.935	3.709840	6
50	1.144	0.048856	0	0.866	0.582138	18	0.797	2.954430	9	0.689	5.957050	15

Problem instance : *Rou12* ; Initial Solution Generator : GG; T_initial : 100; T_final : 0.01; alpha : 0.85												
numRestart	1			10			50			100		
max Iter	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	7.175	0.018192	0	6.460	0.207838	0	6.165	1.042630	0	6.329	2.029280	0
5	5.218	0.020127	0	5.448	0.241347	0	6.084	1.214760	1	6.057	2.440280	0
20	8.721	0.029855	0	6.287	0.346669	2	5.730	1.813430	1	5.342	3.723790	2
50	8.035	0.059322	0	7.004	0.578367	0	5.458	3.081890	1	4.536	5.885940	2

Problem instance : *Nug12* ; Initial Solution Generator : GG; T_initial : 100; T_final : 0.01; alpha : 0.85												
numRestart	1			10			50			100		
max Iter	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	14.747	0.017793	0	11.478	0.204005	0	12.211	1.021020	0	12.280	2.113360	0
5	10.633	0.021445	0	5.121	0.233590	0	5.599	1.176550	4	5.723	2.340060	2
20	3.325	0.042720	4	4.768	0.352855	0	3.986	1.784750	3	4.062	3.460970	3
50	3.277	0.060146	20	3.125	0.581207	7	3.228	2.967640	2	3.349	5.927830	1

Problem instance : *Scr12* ; Initial Solution Generator : GG; T_initial : 100; T_final : 0.01; alpha : 0.85												
numRestart	1			10			50			100		
max Iter	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1	16.413	0.018422	0	7.857	0.194818	0	6.053	1.015890	2	6.579	2.041160	2
5	10.469	0.020516	0	8.401	0.235638	0	6.106	1.172840	2	5.611	2.353360	2
20	8.823	0.041218	0	7.291	0.345888	0	5.167	1.752230	3	4.734	3.478390	2
50	7.880	0.059134	0	5.403	0.579547	0	4.902	2.894310	3	4.589	5.726560	3

The initial temperature test for the yellow-highlighted configurations:

Table 7: Tuning Result for Initial Temperature for SA with IRG and SA with GG (Had12 and number of restarts : 10)

Problem instance : *Had12* ; Initial Solution Generator : IRG; numRestart : 10; T_final : 0.01; alpha : 0.85									
max Iter	5			20			50		
T_initial	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
55	1.039	0.229478	0	0.817	0.330277	11	0.697	0.544015	17
75	1.061	0.218844	0	0.774	0.329662	11	0.895	0.570339	16
100	1.412	0.242559	6	0.900	0.346754	9	0.799	0.544843	13

Problem instance : *Had12* ; Initial Solution Generator : GG; numRestart : 10; T_final : 0.01; alpha : 0.85									
max Iter	5			20			50		
T_initial	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
55	1.213	0.221623	8	0.837	0.303378	6	0.786	0.505848	6
75	1.263	0.233077	10	0.703	0.331455	5	0.737	0.496295	13
100	1.338	0.227514	6	1.041	0.346443	14	0.866	0.582138	18

Table 8: Tuning Result for Initial Temperature for SA with IRG (number of restarts : 10) and SA with GG (number of restarts : 1 and 10) using Nug12

Problem instance : *Nug12* ; Initial Solution Generator : IRG; numRestart : 10; T_final : 0.01; alpha : 0.85						
max Iter / T_initial	20			50		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
55	3.543	0.333515	4	3.000	0.517734	5
75	4.048	0.335382	4	3.149	0.557835	3
100	3.727	0.363634	2	3.242	0.577965	10

Problem instance : *Nug12* ; Initial Solution Generator : GG; T_final : 0.01; alpha : 0.85									
max Iter / T_initial	**numRestart : 1**						**numRestart : 10**		
	20			50			50		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
55	4.848	0.027232	9	5.529	0.045386	0	3.422	0.503040	5
75	4.035	0.028193	0	5.758	0.051746	0	3.356	0.540727	5
100	3.325	0.042720	4	3.277	0.060146	20	3.125	0.581207	7

(Note : due to the nature of how the algorithm is written, a restart of 1 actually means the algorithm runs once without restarting)

Based on the results, it is certain that SA is capable of achieving the global optimum, unlike LS which depends highly on its initial solution. From Table 5 and 6, it can be inferred that increasing the number of restart and/or increasing the maximum number of iterations per temperature level will lead to a decreasing average deviation, but a longer run time. Regardless of how the initial solution is generated, for the Rou12 and Scr12 problem instances, the SA will need larger number of restarts to converge to a better average deviation, which might take more than 10 seconds of run time for just one search, which is why the tuning stops there and the best configuration is selected without undergoing an initial temperature tuning test. From Table 7 and 8, changing the starting temperature doesn't exhibit a clear trend but doing so did bring improvements to some. In Table 8, for SA with GG, the configuration with 10 restarts is chosen instead of better ones (shorter run time) without restart because other results proven that running SA without restarts tend not to achieve global optimum, implying that the achievements here might be a fluke. The summary of all configurations of the SA algorithm for each problem instances is given in Table 9.

Table 9: Summary of SA's best configurations for each problem instance given the initial solution generator it uses.

Problem Instances	**SA with IRG**	**SA with GG**
Had12	$T_{initial} = 55, T_{final} = 0.01, \alpha = 0.85, numRestart = 10, \max iter = 50$	$T_{initial} = 75, T_{final} = 0.01, \alpha = 0.85, numRestart = 10, \max iter = 20$
Rou12	$T_{initial} = 100, T_{final} = 0.01, \alpha = 0.85, numRestart = 100, \max iter = 50$	$T_{initial} = 100, T_{final} = 0.01, \alpha = 0.85, numRestart = 100, \max iter = 50$
Nug12	$T_{initial} = 55, T_{final} = 0.01, \alpha = 0.85, numRestart = 10, \max iter = 50$	$T_{initial} = 100, T_{final} = 0.01, \alpha = 0.85, numRestart = 10, \max iter = 50$
Scr12	$T_{initial} = 100, T_{final} = 0.01, \alpha = 0.85, numRestart = 100, \max iter = 50$	$T_{initial} = 100, T_{final} = 0.01, \alpha = 0.85, numRestart = 100, \max iter = 50$

3.5 Genetic Algorithm

Like SA, genetic algorithm (GA) also has a lot of hyperparameters. Like earlier, not all hyperparameters will be tuned. The hyperparameters to be kept fixed are mutation rate (probability of mutation), $r_{mut} = 0.5$, number of crossovers, $N_{cross} = 2$, number of mutations, $N_{mut} = 2$ and the number of iterations for the IRG to generate the initial population, $iter = 100$. The hyperparameters, number of generations, $nGen$ and the population size, s are the major influencers to the performance and must be tuned. Table 10 shows the results of tuning the GA. Note that the GA doesn't use GG to generate its initial population, so the tuning result is lesser than usual.

Table 10: Tuning results of GA.

Problem instance : *Had12* ; MutRate : 0.5; nMut : 2; nCrossover : 2; iter (IRG) : 100

num Gen / Population Size	500			**1000***			**10000***		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
25	0.695	0.037627	0	0.355	0.051720	21	0.203	0.326976	66
75	0.550	0.086356	9	0.303	0.093998	18	0.165	0.382083	75
125 / **200***	1.631	0.142116	0	1.214	0.242124	0	0.071	0.571037	87
200 / **500***	1.944	0.225448	0	1.688	0.556321	0	0.011	1.001690	97

Problem instance : *Rou12* ; MutRate : 0.5; nMut : 2; nCrossover : 2; iter (IRG) : 100

num Gen / Population Size	1000			10000			50000		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
25	5.605	0.051650	0	3.043	0.327960	2	2.817	1.564610	6
75	4.471	0.112917	0	2.351	0.348194	6	1.710	1.653910	15
200	6.526	0.234481	0	1.420	0.591198	19	1.451	2.158310	16
500	6.420	0.555985	0	1.261	1.018490	20	0.769	3.015120	28

Problem instance : *Nug12* ; MutRate : 0.5; nMut : 2; nCrossover : 2; iter (IRG) : 100

num Gen / Population Size	1000			10000			50000		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
25	4.796	0.058399	0	2.609	0.316571	7	2.301	1.575060	7
75	3.958	0.106720	0	2.131	0.368513	12	2.069	1.736290	7
200	5.841	0.235784	0	1.661	0.569715	6	1.287	2.122670	16
500	6.273	0.558295	0	1.253	0.990448	15	0.993	2.973940	30

Problem instance : *Scr12* ; MutRate : 0.5; nMut : 2; nCrossover : 2; iter (IRG) : 100

num Gen / Population Size	1000			10000			50000		
	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
25	5.486	0.045676	0	3.644	0.305922	25	3.366	1.523910	20
75	6.493	0.105413	0	2.751	0.379202	33	2.752	1.754130	33
200	7.558	0.211091	0	1.015	0.558847	64	1.350	2.058100	60
500	9.467	0.572955	0	1.004	0.965243	61	0.172	3.195020	93

From Table 10, GA is able to perform extremely well on all four problem instances, with all best average deviations within 1%. GA also achieve global optimum very frequently at higher number of generations. Interestingly, increasing the population size too much will be detrimental to the performance, as shown in Had12 in which the GA performed poorly with population size of 200 and 500 as oppose to 25 and 75 with the maximum number of generations at 500. In other problem instances, with 10000 number of generations, using a population size of 500 didn't improve vastly from using population size of 200, until the generation number is increased to 50000. This plateau phenomenon is as if the number of generations is the limiting factor to the performance. One more thing to note is the results highlighted in yellow. These results show that GA could achieve less than 1% deviation at a very good run time, defeating both LS and SA in terms of solution quality for the Had12 problem instance. Table 11 shows the summary of configurations of GA for the four problem instances.

Table 11: Summary of GA's best configurations for each problem instance

Problem Instances	**GA**
Had12	$nGen = 10000, s = 200,$ $r_{mut} = 0.5, N_{cross} = 2,$ $N_{mut} = 2, iter = 100$
Rou12	$nGen = 50000, s = 500,$ $r_{mut} = 0.5, N_{cross} = 2,$ $N_{mut} = 2, iter = 100$
Nug12	$nGen = 50000, s = 500,$ $r_{mut} = 0.5, N_{cross} = 2,$ $N_{mut} = 2, iter = 100$
Scr12	$nGen = 50000, s = 500,$ $r_{mut} = 0.5, N_{cross} = 2,$ $N_{mut} = 2, iter = 100$

3.6 Tabu Search

Compare to SA and GA, the proposed tabu search (TS) is the easiest metaheuristic to tune. It only has the maximum number of iterations, $\boldsymbol{i_{max}}$, the minimum tabu tenure, $\boldsymbol{t_{min}}$ and the maximum tabu tenure, $\boldsymbol{t_{max}}$ to tune. The minimum and maximum tabu tenure can be treated as a tenure range when tuning, which further simplifies the tuning process. Table 12 shows the tuning results for TS with IRG and TS with GG.

Table 12: Tuning result for TS with IRG

Problem instance : *Had12* ; Initial Solution Generator : IRG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	0.528	0.076834	48	0.354	0.170583	48	0.363	0.742452	54
7 to 10	0.171	0.127680	58	0.295	0.266111	58	0.257	1.240030	46
5 to 12	0.534	0.104835	38	0.349	0.212837	37	0.232	1.285030	51
20 to 30	0.268	0.195916	45	0.225	0.406213	50	0.222	2.891990	52

Problem instance : *Rou12* ; Initial Solution Generator : IRG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	1.294	0.139937	15	1.630	0.265998	4	0.559	1.370230	40
7 to 10	1.918	0.212289	8	1.025	0.452035	18	0.061	2.344960	77
5 to 12	1.787	0.220614	8	1.485	0.435767	15	0.086	2.420880	73
20 to 30	1.765	0.378572	8	1.266	0.850456	20	0.162	5.035770	63

Problem instance : *Nug12* ; Initial Solution Generator : IRG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	1.415	0.065322	24	1.225	0.157236	30	0.965	0.995821	36
7 to 10	1.232	0.108410	26	0.879	0.262821	36	0.145	1.775370	84
5 to 12	1.294	0.120557	5	0.799	0.249907	47	0.208	1.754660	79
20 to 30	1.391	0.218706	23	1.232	0.489552	21	0.256	3.571410	73

Problem instance : *Scr12* ; Initial Solution Generator : IRG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	3.370	0.060475	17	2.600	0.143984	34	2.779	0.970513	34
7 to 10	3.687	0.105239	10	2.436	0.250815	43	0.727	1.739970	80
5 to 12	1.926	0.103539	53	3.073	0.231907	19	0.673	1.676910	81
20 to 30	2.719	0.184946	21	1.743	0.476428	40	0.160	3.218170	93

Table 13: Tuning results for TS with GG

Problem instance : *Had12* ; Initial Solution Generator : GG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	0.709	0.081363	0	0.586	0.131832	0	0.479	0.715036	0
7 to 10	0.492	0.122910	0	0.479	0.235319	0	0.479	1.304720	0
5 to 12	0.504	0.110119	0	0.479	0.226478	0	0.479	1.303300	0
20 to 30	0.697	0.203420	0	0.632	0.490486	0	0.419	2.733540	12

Problem instance : *Rou12* ; Initial Solution Generator : GG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	1.490	0.136075	7	1.153	0.266748	6	0.489	1.350050	39
7 to 10	0.743	0.213837	29	0.321	0.432783	70	0.021	2.318760	92
5 to 12	0.786	0.214367	29	0.622	0.446960	42	0.036	2.340370	88
20 to 30	0	0.361350	100	0	0.840623	100	0	4.924320	100

Problem instance : *Nug12* ; Initial Solution Generator : GG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	1.287	0.097412	7	1.087	0.158560	23	1.066	1.009530	23
7 to 10	1.495	0.188735	0	0.941	0.267151	31	0.194	1.965150	81
5 to 12	1.453	0.142380	0	0.858	0.258480	26	0.131	1.751050	88
20 to 30	1.730	0.255924	0	1.239	0.558913	17	0.291	3.509230	69

Problem instance : *Scr12* ; Initial Solution Generator : GG									
Max Iteration	50			100			500		
Tenure Range	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)	Dev (%)	Runtime (s)	fOPT (%)
1 to 4	4.065	0.069215	0	3.888	0.142180	0	3.871	0.713161	0
7 to 10	2.262	0.125735	14	1.394	0.303338	47	0.225	1.243010	91
5 to 12	2.104	0.115256	20	1.777	0.310013	32	0.199	1.402890	92
20 to 30	2.630	0.201579	0	1.953	0.572530	24	0.233	3.227270	89

From Table 12, the TS was able to achieve global optimum at least once for all problem instances at any tenure range and iterations used in the experiments. The yellow-highlighted result for Rou12 in Table 12 shows that TS could outperform GA in terms of solution quality and run time even though this isn't TS's best configuration in that problem instance. The other yellow-highlighted results in Table 12 and Table 13 are just to show alternatives if one prefers to trade in some accuracy for a shorter run time. Using GG in Table 13, the TS didn't perform well for the Had12 problem instance. However, with GG, the TS will always solve Rou12 with perfect scores at a reasonably short run time. Despite that, a slightly poorer configuration is chosen for TS with GG on the Rou12 data as that perfect configuration is unlikely to perform equally well on other problem instances. Table 14 gives the summary of TS's best configurations for each problem instance.

Table 14: Summary of TS's best configurations for each problem instance given the initial solution generator it uses.

Problem Instances	**TS with IRG**	**TS with GG**
Had12	$\boldsymbol{i_{max}} = 50$, $t_{min} = 7, t_{max} = 10$	$\boldsymbol{i_{max}} = 500$, $t_{min} = 20, t_{max} = 30$
Rou12	$\boldsymbol{i_{max}} = 500$, $t_{min} = 7, t_{max} = 10$	$\boldsymbol{i_{max}} = 500$, $t_{min} = 7, t_{max} = 10$
Nug12	$\boldsymbol{i_{max}} = 500$, $t_{min} = 7, t_{max} = 10$	$\boldsymbol{i_{max}} = 500$, $t_{min} = 5, t_{max} = 12$
Scr12	$\boldsymbol{i_{max}} = 500$, $t_{min} = 20, t_{max} = 30$	$\boldsymbol{i_{max}} = 500$, $t_{min} = 5, t_{max} = 12$

4. Discussion

With the tuning stage completed, using all the best configurations for each algorithm given the problem instances, the four algorithms are tested using the same problem instances, but with larger sizes. This time, the best solution and permutation will be included. Like the tuning test, each result is the average of 100 independent runs. Table 15 to Table 18 shows the result of these experiments. Solution values highlighted in **green** are global optimum values or optimum values (OPT) for the given problem instance as noted on the QAPLIB website.

Local Search:

Table 15: Top: Results of Local Search with IRG on problem instances. Bottom : Results of Local Search with GG on problem instances.

	LS with IRG				
Problem	**Dev (%)**	**Runtime (s)**	**fOPT (%)**	**Best Sol**	**Best Permutation**
Had12	0.467	0.007574	0	1656	(3, 10, 5, 2, 12, 11, 7, 1, 8, 6, 4, 9)
Had14	0.220	0.011775	0	2730	(8, 13, 10, 5, 2, 12, 14, 6, 3, 11, 7, 1, 9, 4)
Had16	0.147	0.027383	19	**3720**	(9, 4, 16, 1, 7, 8, 6, 11, 15, 14, 12, 10, 5, 3, 2, 13)
Had18	1.176	0.055754	0	5370	(13, 5, 10, 11, 12, 6, 14, 18, 2, 7, 1, 8, 15, 4, 16, 9, 3, 17)
Had20	0.867	0.106220	0	6934	(9, 4, 14, 16, 19, 18, 7, 11, 1, 12, 10, 17, 2, 20, 5, 3, 15, 8, 6, 13)
Rou12	4.143	0.007080	0	245286	(12, 2, 5, 8, 1, 11, 3, 4, 6, 7, 9, 10)
Rou15	8.775	0.013004	0	369226	(8, 1, 6, 15, 14, 7, 13, 4, 12, 9, 11, 5, 3, 10, 2)
Rou20	4.567	0.064410	0	737392	(3, 20, 13, 18, 10, 7, 6, 16, 9, 5, 15, 8, 19, 14, 2, 4, 12, 1, 17, 11)
Nug12	2.076	0.005928	0	590	(3, 1, 8, 4, 9, 11, 7, 12, 2, 10, 6, 5)
Nug14	5.834	0.010498	0	1072	(11, 12, 7, 3, 10, 9, 8, 13, 14, 6, 1, 5, 2, 4)
Nug18	3.851	0.044186	0	1962	(6, 5, 1, 7, 17, 10, 12, 8, 15, 18, 3, 14, 11, 2, 4, 13, 9, 16)
Scr12	5.265	0.007554	0	31884	(3, 6, 7, 10, 2, 9, 5, 1, 11, 8, 4, 12)
Scr15	7.105	0.020755	0	54142	(2, 10, 8, 9, 11, 13, 14, 15, 12, 6, 1, 5, 4, 7, 3)
Scr20	7.998	0.072081	0	112664	(20, 8, 12, 7, 9, 4, 3, 2, 16, 11, 17, 10, 18, 15, 19, 13, 14, 5, 6, 1)

	LS with GG				
Problem	**Dev (%)**	**Runtime (s)**	**fOPT (%)**	**Best Sol**	**Best Permutation**
Had12	0.726	0.006893	0	1664	(9, 4, 1, 7, 11, 6, 12, 5, 8, 2, 10, 3)
Had14	1.468	0.016177	0	2764	(4, 9, 7, 1, 6, 14, 11, 5, 8, 12, 10, 2, 13, 3)
Had16	3.387	0.020171	0	3846	(3, 6, 14, 2, 7, 5, 12, 11, 1, 10, 16, 8, 13, 15, 9, 4)
Had18	1.605	0.042339	0	5444	(3, 10, 17, 18, 2, 14, 12, 5, 15, 7, 6, 11, 16, 4, 1, 9, 13, 8)
Had20	0.000	0.095077	100	**6922**	(8, 15, 16, 14, 19, 6, 7, 17, 1, 12, 10, 11, 5, 20, 2, 3, 4, 9, 18, 13)
Rou12	6.796	0.006780	0	251534	(3, 9, 6, 1, 2, 7, 10, 8, 12, 5, 4, 11)
Rou15	3.690	0.015003	0	367282	(15, 8, 1, 4, 9, 14, 10, 3, 6, 13, 2, 12, 11, 7, 5)
Rou20	3.493	0.108787	0	750866	(18, 8, 19, 2, 7, 16, 5, 20, 15, 13, 6, 9, 11, 14, 10, 3, 12, 1, 17, 4)
Nug12	3.806	0.007602	0	600	(3, 9, 12, 1, 2, 11, 7, 8, 10, 6, 5, 4)
Nug14	7.692	0.012131	0	1092	(10, 14, 12, 11, 6, 1, 2, 13, 9, 5, 4, 3, 7, 8)
Nug18	3.316	0.056167	0	1994	(17, 8, 18, 11, 4, 7, 12, 15, 2, 16, 5, 1, 10, 14, 9, 6, 13, 3)
Scr12	6.272	0.006263	0	33380	(11, 6, 10, 5, 7, 1, 2, 3, 9, 8, 12, 4)
Scr15	11.259	0.029741	0	56898	(13, 5, 8, 1, 9, 2, 6, 10, 3, 11, 14, 15, 4, 7, 12)
Scr20	4.959	0.076052	0	115486	(7, 6, 13, 5, 8, 1, 2, 3, 16, 11, 19, 10, 20, 15, 12, 18, 14, 17, 9, 4)

From Table 15, the LS is unable to reach global optimum for most problem instances and the average deviation from the global optimum for most problem instances is quite large. Using GG to generate initial solution, the LS performed flawlessly on Had20, and this isn't repeatable if the LS starts the search with 3-Opt or using IRG instead, indicating the location of the starting point is more crucial than its quality, considering GG produces initial solutions of quality lower than that from IRG. Similarly, the LS achieved global optimum on Had16 19 times using IRG because the IRG generated the right initial solution for the LS to converge in the correct direction. On the positive note, the run time of LS in every problem instance is remarkably fast.

Simulated Annealing:

Table 16: Top: Results of Simulated Annealing with IRG on problem instances. Bottom : Results of Simulated Annealing with GG on problem instances.

	SA with IRG				
Problem	**Dev (%)**	**Runtime (s)**	**fOPT (%)**	**Best Sol**	**Best Permutation**
Had12	0.697	0.544015	17	**1652**	(3, 10, 11, 2, 12, 5, 7, 6, 8, 1, 4, 9)
Had14	0.427	0.664545	29	**2724**	(8, 13, 10, 11, 12, 5, 2, 14, 3, 6, 7, 1, 9, 4)
Had16	0.091	0.858894	44	**3720**	(9, 4, 16, 1, 7, 8, 6, 11, 15, 14, 12, 10, 5, 3, 2, 13)
Had18	0.425	1.085690	9	**5358**	(8, 15, 16, 14, 7, 18, 6, 11, 1, 10, 12, 5, 3, 13, 2, 17, 9, 4)
Had20	0.427	1.270200	7	**6922**	(8, 15, 1, 14, 6, 19, 7, 11, 16, 12, 10, 17, 2, 20, 5, 3, 4, 9, 18, 13)
Rou12	3.472	5.826490	1	**235528**	(6, 5, 11, 9, 2, 8, 3, 1, 12, 7, 4, 10)
Rou15	5.109	8.774450	0	359748	(12, 13, 8, 5, 14, 15, 3, 6, 2, 1, 9, 10, 4, 7, 11)
Rou20	3.361	14.431100	0	729066	(6, 7, 17, 18, 10, 20, 15, 16, 9, 11, 5, 14, 8, 1, 13, 3, 12, 2, 4, 19)
Nug12	3.000	0.517734	5	**578**	(2, 10, 6, 5, 1, 11, 8, 4, 3, 9, 7, 12)
Nug14	3.004	0.686411	1	**1014**	(9, 8, 13, 2, 1, 11, 7, 14, 3, 4, 12, 5, 6, 10)
Nug18	2.650	1.070260	1	**1930**	(10, 3, 14, 2, 18, 6, 7, 12, 15, 4, 5, 1, 11, 8, 17, 13, 9, 16)
Scr12	3.869	5.770350	13	**31410**	(2, 7, 10, 11, 3, 12, 8, 4, 9, 6, 1, 5)
Scr15	7.791	8.602030	4	**51140**	(12, 10, 11, 14, 1, 13, 9, 5, 15, 6, 4, 2, 8, 7, 3)
Scr20	7.144	14.796800	0	110058	(13, 14, 10, 15, 17, 16, 18, 19, 2, 6, 8, 7, 1, 5, 9, 4, 3, 12, 11, 20)

	SA with GG				
Problem	**Dev (%)**	**Runtime (s)**	**fOPT (%)**	**Best Sol**	**Best Permutation**
Had12	0.703	0.331455	5	**1652**	(3, 10, 11, 2, 12, 5, 7, 6, 8, 1, 4, 9)
Had14	0.570	0.414266	15	**2724**	(8, 13, 5, 10, 12, 11, 2, 14, 3, 6, 7, 1, 9, 4)
Had16	0.219	0.547823	26	**3720**	(9, 4, 16, 1, 7, 8, 6, 14, 15, 11, 12, 10, 5, 3, 2, 13)
Had18	0.457	0.690008	1	**5358**	(8, 15, 16, 6, 7, 18, 14, 11, 1, 10, 12, 5, 13, 3, 2, 17, 9, 4)
Had20	0.249	0.814842	6	**6922**	(8, 15, 16, 6, 14, 19, 7, 11, 1, 12, 10, 5, 3, 20, 2, 17, 4, 9, 18, 13)
Rou12	4.536	5.885940	2	**235528**	(6, 5, 11, 9, 2, 8, 3, 1, 12, 7, 4, 10)
Rou15	5.094	8.677300	0	356654	(12, 11, 4, 14, 7, 3, 5, 2, 15, 1, 9, 10, 8, 13, 6)
Rou20	3.009	14.329500	0	726920	(2, 1, 19, 14, 6, 3, 15, 5, 9, 20, 7, 4, 8, 13, 17, 16, 12, 11, 18, 10)
Nug12	3.125	0.581207	7	**578**	(12, 7, 9, 3, 4, 8, 11, 1, 5, 6, 10, 2)
Nug14	3.535	0.686705	0	1016	(12, 7, 3, 4, 10, 9, 5, 13, 14, 6, 11, 8, 2, 1)
Nug18	2.947	1.170780	1	**1930**	(10, 3, 14, 2, 18, 6, 7, 12, 15, 4, 5, 1, 11, 8, 17, 13, 9, 16)
Scr12	4.589	5.726560	3	**31410**	(2, 7, 10, 11, 3, 12, 8, 4, 9, 6, 1, 5)
Scr15	6.614	8.811230	2	**51140**	(15, 7, 11, 8, 1, 4, 3, 2, 12, 6, 13, 5, 14, 10, 9)
Scr20	4.171	14.512900	1	**110030**	(17, 6, 9, 7, 1, 5, 2, 3, 15, 10, 19, 12, 18, 14, 13, 20, 16, 8, 11, 4)

As expected from a metaheuristic, SA managed to get a global optimum in most problem instances. As explain during the tuning stage, SA could still be tuned using larger number of restarts for Rou12, Nug12 and Scr12 for a smaller deviation but the tuning stopped there as the SA will be taking extremely long to run, which is one of the disadvantages of SA. Even though the deviation for the three problem instances category is huge, it still manages to find the global optimum for most of these problem instances at the very least.

<u>Genetic Algorithm:</u>

Table 17: Results of Genetic Algorithm on problem instances

	GA				
Problem	**Dev (%)**	**Runtime (s)**	**fOPT (%)**	**Best Sol**	**Best Permutation**
Had12	0.071	0.571037	87	**1652**	(3, 10, 11, 2, 12, 5, 6, 7, 8, 1, 4, 9)
Had14	0.005	0.646634	97	**2724**	(8, 13, 10, 5, 12, 11, 2, 14, 3, 6, 7, 1, 9, 4)
Had16	0.042	0.742293	41	**3720**	(9, 4, 16, 1, 7, 8, 6, 14, 15, 11, 12, 10, 5, 3, 2, 13)
Had18	0.278	0.884251	18	**5358**	(8, 15, 16, 6, 7, 18, 14, 11, 1, 10, 12, 5, 3, 13, 2, 17, 9, 4)
Had20	0.358	0.991219	13	**6922**	(8, 15, 16, 6, 19, 14, 7, 17, 1, 12, 10, 11, 5, 20, 2, 3, 4, 9, 18, 13)
Rou12	0.769	3.015120	28	**235528**	(6, 5, 11, 9, 2, 8, 3, 1, 12, 7, 4, 10)
Rou15	2.628	3.439760	5	**354210**	(12, 6, 8, 13, 5, 3, 15, 2, 7, 1, 9, 10, 4, 14, 11)
Rou20	2.409	4.583320	0	727322	(19, 7, 5, 2, 11, 14, 10, 8, 9, 15, 18, 3, 20, 13, 1, 4, 12, 16, 17, 6)
Nug12	0.993	2.973940	30	**578**	(2, 10, 6, 5, 1, 11, 8, 4, 3, 9, 7, 12)
Nug14	0.789	3.224390	5	**1014**	(9, 8, 13, 2, 1, 11, 7, 14, 3, 4, 12, 5, 6, 10)
Nug18	1.477	4.266380	3	**1930**	(10, 3, 14, 2, 18, 6, 7, 12, 15, 4, 5, 1, 11, 8, 17, 13, 9, 16)
Scr12	0.172	3.195020	93	**31410**	(3, 6, 11, 10, 2, 9, 5, 1, 12, 7, 4, 8)
Scr15	2.378	3.417240	55	**51140**	(15, 7, 11, 8, 1, 4, 3, 2, 12, 6, 13, 5, 14, 10, 9)
Scr20	3.037	4.488080	6	**110030**	(20, 7, 12, 6, 4, 8, 3, 2, 14, 11, 18, 9, 19, 15, 16, 17, 13, 5, 10, 1)

According to Table 17, GA performed impressively on most problem instances, with greater consistency in hitting the global optimum based on fOPT and smaller average deviation than SA. From an overall perspective, GA also runs faster than SA, especially if the SA is to be tuned to match the deviation of GA. Like SA, for problem instances of larger size, the GA must be implemented with a larger generation count, to achieve the global optimum more frequently.

<u>Tabu Search:</u>

Table 18: Top: Results of Tabu Search with IRG on problem instances. Bottom : Results of Tabu Search with GG on problem instances.

	TS with IRG				
Problem	**Dev (%)**	**Runtime (s)**	**fOPT (%)**	**Best Sol**	**Best Permutation**
Had12	0.171	0.127680	58	**1652**	(3, 10, 11, 2, 12, 5, 7, 6, 8, 1, 4, 9)
Had14	0.250	0.158901	66	**2724**	(8, 13, 10, 11, 12, 5, 2, 14, 3, 6, 7, 1, 9, 4)
Had16	0.423	0.241281	24	**3720**	(9, 4, 16, 1, 7, 8, 6, 14, 15, 11, 12, 10, 5, 3, 2, 13)
Had18	0.465	0.319916	31	**5358**	(8, 15, 16, 6, 7, 18, 14, 11, 1, 10, 12, 5, 13, 3, 2, 17, 9, 4)
Had20	0.630	0.399330	9	**6922**	(8, 15, 1, 6, 14, 19, 7, 11, 16, 12, 10, 5, 3, 20, 2, 17, 4, 9, 18, 13)
Rou12	0.061	2.344960	77	**235528**	(6, 5, 11, 9, 2, 8, 3, 1, 12, 7, 4, 10)
Rou15	0.573	2.147140	65	**354210**	(12, 6, 8, 13, 5, 3, 15, 2, 7, 1, 9, 10, 4, 14, 11)
Rou20	0.466	4.649350	5	**725522**	(1, 19, 2, 14, 10, 16, 11, 20, 9, 5, 7, 4, 8, 18, 15, 3, 12, 17, 13, 6)
Nug12	0.145	1.775370	84	**578**	(3, 9, 7, 12, 1, 11, 8, 4, 2, 10, 6, 5)
Nug14	0.519	1.781610	39	**1014**	(9, 8, 13, 2, 1, 11, 7, 14, 3, 4, 12, 5, 6, 10)
Nug18	0.485	3.457740	22	**1930**	(9, 3, 10, 6, 13, 14, 2, 12, 7, 5, 18, 15, 8, 1, 17, 16, 4, 11)
Scr12	0.160	3.218170	93	**31410**	(5, 7, 2, 3, 11, 4, 8, 12, 1, 6, 9, 10)
Scr15	1.212	4.649420	71	**51140**	(15, 7, 11, 8, 1, 4, 3, 2, 12, 6, 13, 5, 14, 10, 9)
Scr20	1.169	8.868210	17	**110030**	(1, 14, 9, 15, 17, 13, 18, 19, 7, 10, 3, 12, 2, 6, 5, 4, 8, 16, 11, 20)

	TS with GG				
Problem	Dev (%)	Runtime (s)	fOPT (%)	Best Sol	Best Permutation
Had12	0.419	2.733540	12	**1652**	(3, 10, 11, 2, 12, 5, 7, 6, 8, 1, 4, 9)
Had14	0.734	3.821360	0	2744	(4, 9, 7, 6, 14, 2, 12, 11, 1, 5, 10, 3, 13, 8)
Had16	0.070	5.142580	0	3722	(3, 5, 2, 10, 12, 11, 14, 8, 13, 6, 7, 15, 16, 4, 1, 9)
Had18	0.242	7.077460	0	5370	(13, 5, 10, 11, 12, 6, 14, 18, 2, 7, 1, 8, 15, 4, 16, 9, 17, 3)
Had20	0.000	8.655790	100	**6922**	(8, 15, 16, 14, 19, 6, 7, 17, 1, 12, 10, 11, 5, 20, 2, 3, 4, 9, 18, 13)
Rou12	0.021	2.318760	92	**235528**	(6, 5, 11, 9, 2, 8, 3, 1, 12, 7, 4, 10)
Rou15	1.116	2.196720	40	**354210**	(12, 6, 8, 13, 5, 3, 15, 2, 7, 1, 9, 10, 4, 14, 11)
Rou20	0.484	4.604330	3	**725522**	(1, 19, 2, 14, 10, 16, 11, 20, 9, 5, 7, 4, 8, 18, 15, 3, 12, 17, 13, 6)
Nug12	0.131	1.751050	88	**578**	(12, 7, 9, 3, 4, 8, 11, 1, 5, 6, 10, 2)
Nug14	0.371	1.839530	13	**1014**	(9, 8, 13, 2, 1, 11, 7, 14, 3, 4, 12, 5, 6, 10)
Nug18	0.369	3.442580	10	**1930**	(9, 3, 10, 6, 13, 14, 2, 12, 7, 5, 18, 15, 8, 1, 17, 16, 4, 11)
Scr12	0.199	1.402890	92	**31410**	(8, 6, 11, 10, 2, 9, 5, 1, 12, 7, 4, 3)
Scr15	3.820	2.298260	1	**51140**	(15, 7, 11, 8, 1, 4, 3, 2, 12, 6, 13, 5, 14, 10, 9)
Scr20	2.652	4.540790	3	**110030**	(17, 6, 9, 7, 1, 5, 2, 3, 15, 10, 19, 12, 18, 14, 13, 20, 16, 8, 11, 4)

Based on Table 18, TS managed to find the global optimum for all problem instances with IRG. The average deviations from the global optimum attained by TS regardless of the initial solution generator for most if not all problem instances are always less than 1.0%, not only implying the solutions are superior in quality but also the algorithm's consistency of converging the initial solution to such solutions. Although TS with GG failed to find the global optimum of Had14, Had16 and Had18, despite running it with a costly configuration, most of its solution are very close to the global optimum, as shown by the average deviation and the proximity of the best solution to the global optimum. However, for problem instances of large sizes (Scr20 for TS with IRG and Had20 for TS with GG) that require the most expensive configuration of TS to solve, the run time of TS is very long, compare to GA.

<u>Comparison of Performance:</u>

To give an explicit comparison between each algorithm, Figure 4 to Figure 7 are graphs of the average deviation for all four algorithms vs the size of each problem instance, respectively. Figure 8 to Figure 11 are graphs of the average run time for all algorithms vs the size of each problem instance, respectively.

Average Deviation Comparison for Hadley's problem instances (Had12, 14, 16, 18, 20)

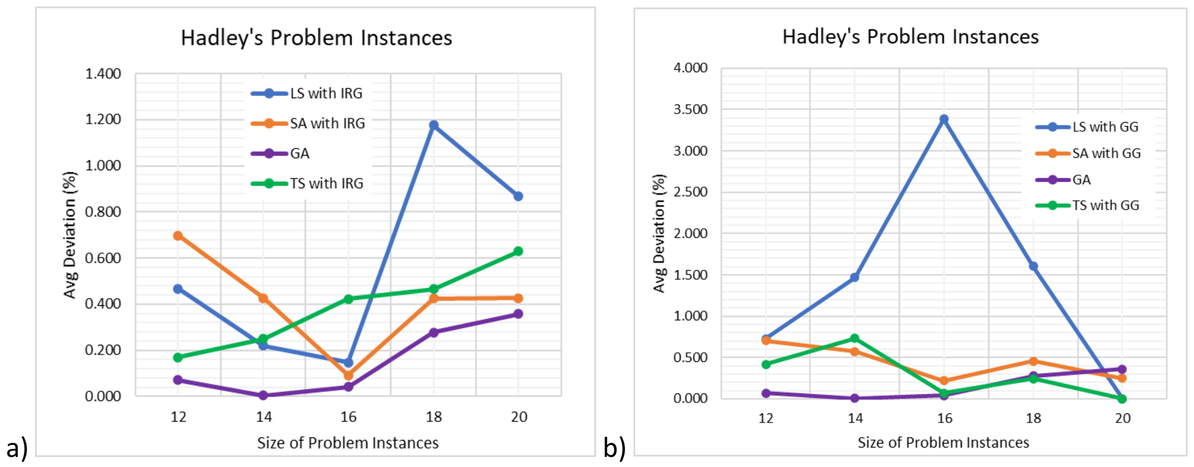


Figure 4: Comparison of average deviation between four algorithms with a) IRG and b) GG on Hadley's problem instances.

Average Deviation Comparison for Roucairol’s problem instances (Rou12, 15, 20)

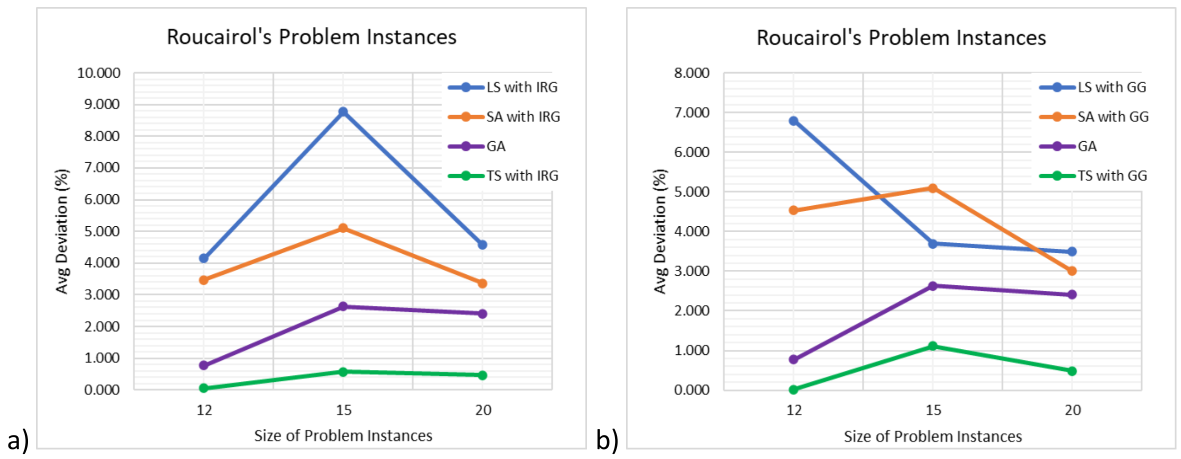


Figure 5: Comparison of average deviation between four algorithms with a) IRG and b) GG on Roucairol's problem instances.

Average Deviation Comparison for Nugent’s problem instances (Nug12, 14, 18)

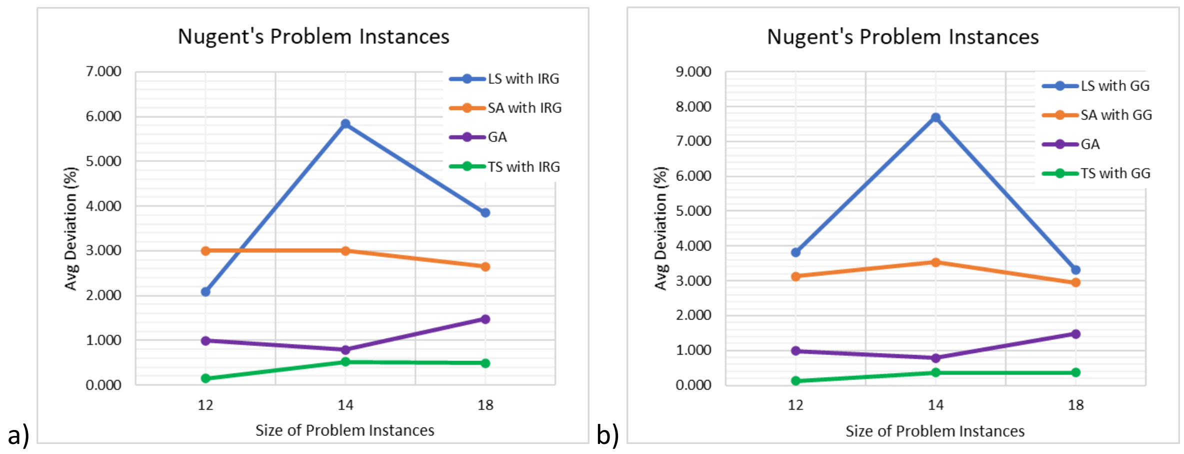


Figure 6: Comparison of average deviation between four algorithms with a) IRG and b) GG on Nugent's problem instances.

Average Deviation Comparison for Scriabin’s problem instances (Scr12, 15, 20)

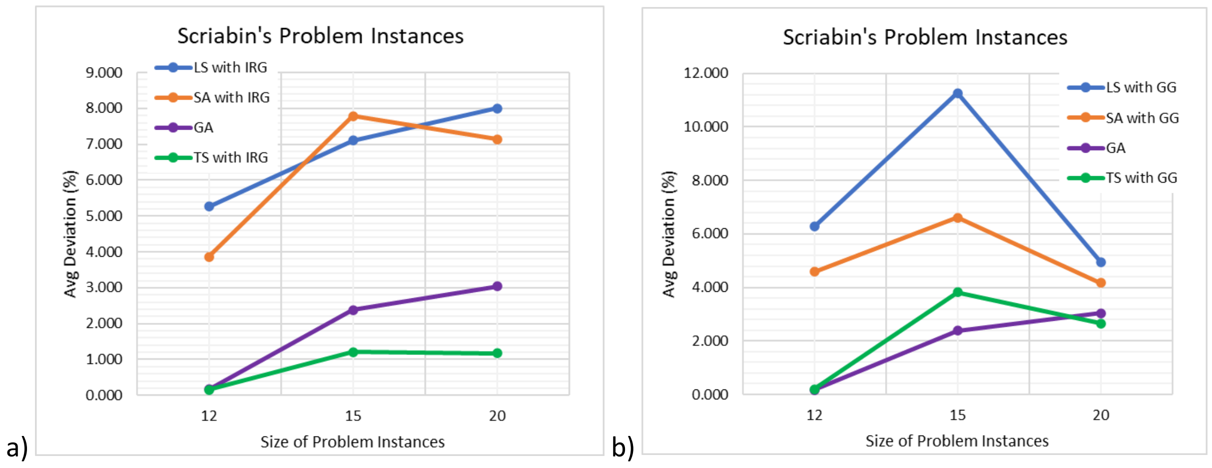


Figure 7: Comparison of average deviation between four algorithms with a) IRG and b) GG on Scriabin's problem instances.

Average Run Time Comparison for Hadley’s problem instances (Had12, 14, 16, 18, 20)

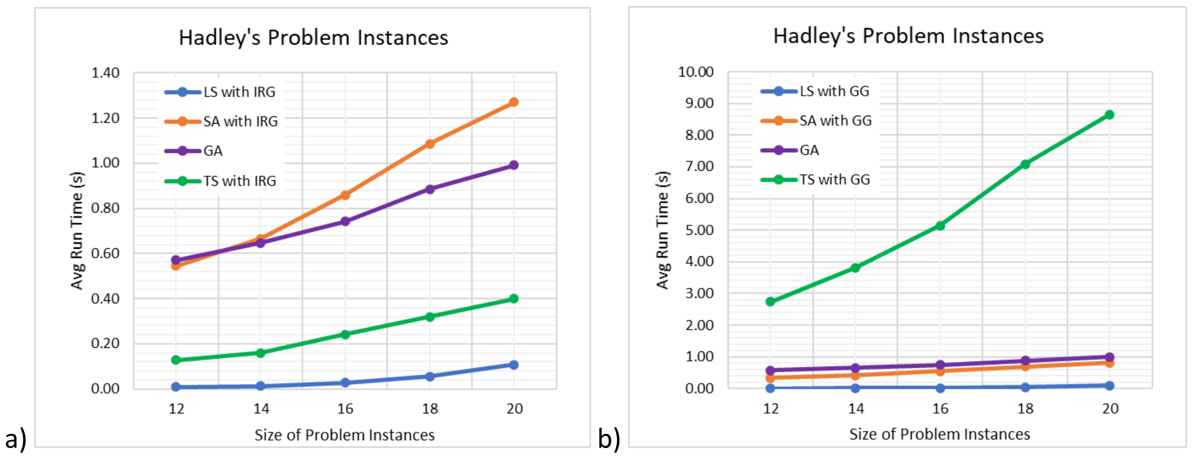


Figure 8: Comparison of average run time between four algorithms with a) IRG and b) GG on Hadley's problem instances.

Average Run Time Comparison for Roucairol’s problem instances (Rou12, 15, 20)

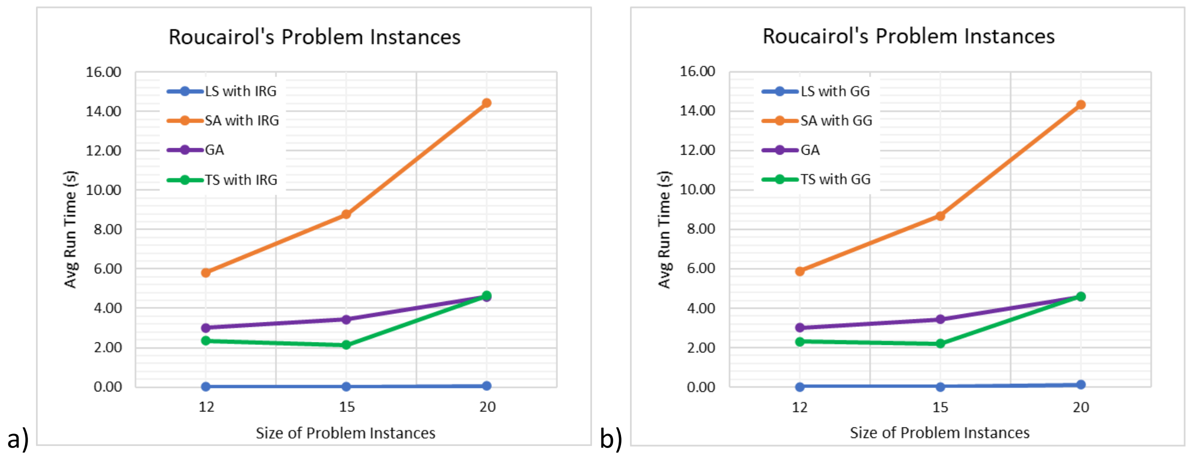


Figure 9: Comparison of average run time between four algorithms with a) IRG and b) GG on Roucairol's problem instances.

Average Run Time Comparison for Nugent’s problem instances (Nug12, 14, 18)

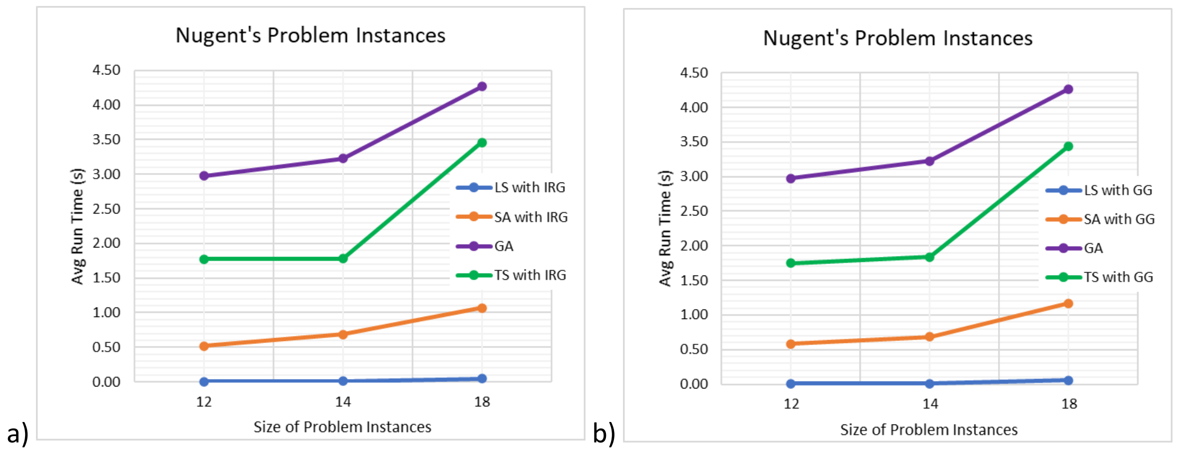
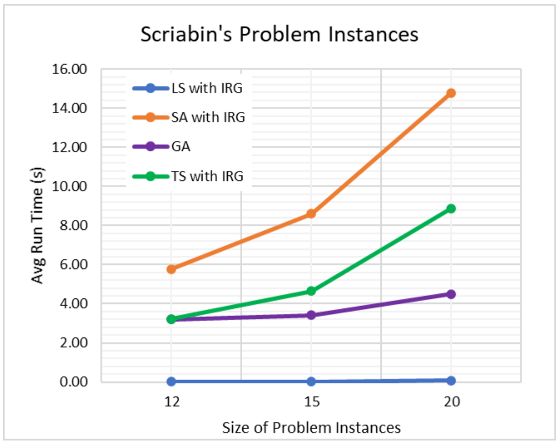


Figure 10: Comparison of average run time between four algorithms with a) IRG and b) GG on Nugent's problem instances.

Average Run Time Comparison for Scriabin's problem instances (Scr12, 15, 20)

a) 

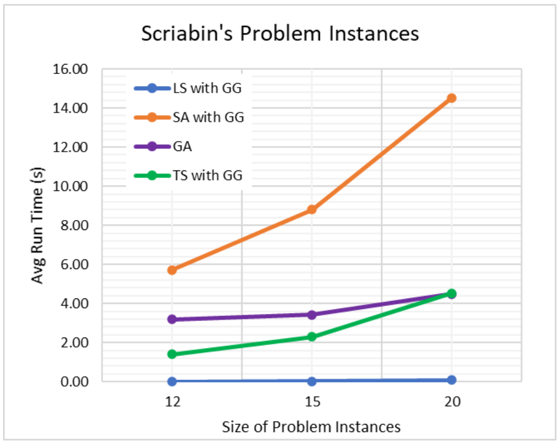
b) 

Figure 11: Comparison of average run time between four algorithms with a) IRG and b) GG on Scriabin's problem instances.

Summary of Comparison:

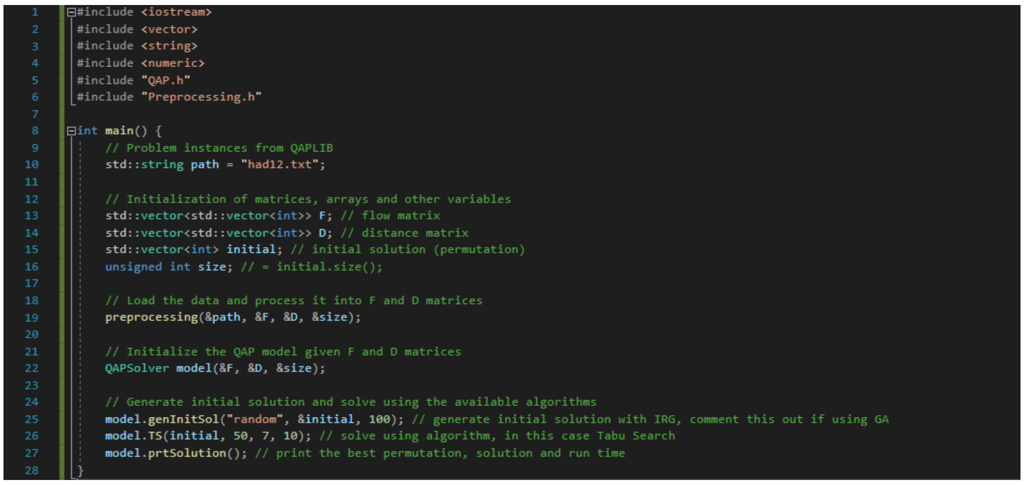
In general, TS and GA output a stronger solution than SA and LS, with LS having the largest deviations. Except Hadley's problem instances, TS had the smallest deviation in majority of the problem instances, be it with IRG or GG. On the topic of which initial solution generator gave the best facilitation to the algorithm's performance, both the IRG and GG had their contributions. IRG and GG gave roughly the same trend in performance in terms of average deviation. Although GG has poor starting solution, using it could boost the performance of TS in the Hadley's problem instances, as shown in Figure 4. Moving forward, out of all the algorithms, LS has the shortest run time whereas SA has the longest run time in most problem instances.

5. Conclusion

In this report, two initial solution generation methods were proposed, which are the Iterative Random Generator (IRG) and the Greedy Generator (GG). IRG, if tuned reasonably, outperformed GG in solution quality and computational time. The solutions from these generators will be used by the Local Search with Dynamic Neighbor Operator (LS), Simulated Annealing (SA) and Tabu Search (TS) to solve a quadratic assignment problem (QAP) using problem instances from QAPLIB. As a bonus, a random and unbiased Genetic Algorithm (GA) was also imposed into the competition. All the algorithms were tuned using the Had12, Rou12, Nug12 and Scr12 problem instances. Once the best configurations for the algorithms were identified for each of the problem instances, the real competition began by solving myriad problems of larger sizes using the algorithms with these configurations. Overall, TS is the victor in terms of solution quality, whereas LS has the poorest solution, despite possessing the fastest run time. This reinforces the standard conjecture in combinatorial optimization problems, whereby stronger algorithms tend to take longer run time and vice versa, a trade off worth contemplating about. Besides, as the size of the problem instances increases, the average run time and average deviation for every algorithm deteriorate. Despite that, all the metaheuristics in this report could achieve the global optimum for most problem instances. Although IRG has the better initial solution, there are occasions where using the solution from GG could solve certain problem instances flawlessly. There are some improvements can be made, such as tuning the algorithms by running it for more than 100 iterations to obtain a more accurate average. Other hyperparameters of the SA and GA could also be tuned, instead of iteration parameters only. Lastly, there is no clear winner in this experiment. Pairing different initial solution generators with different search algorithms will lead to different possibilities, and there will exist a configuration that works best for one's problem.

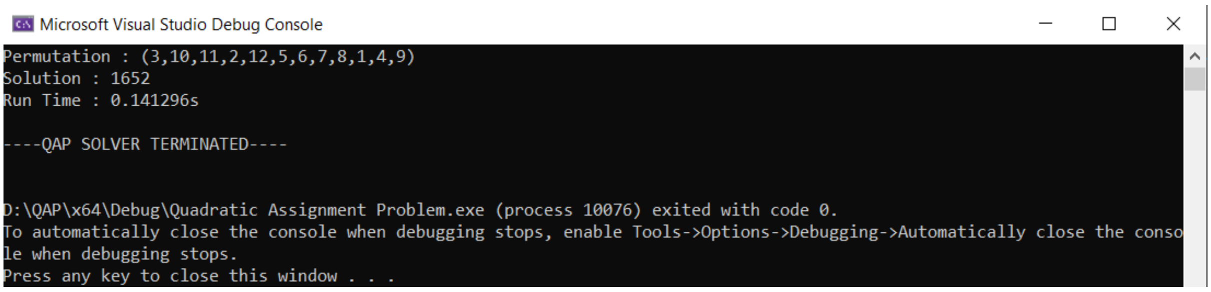
Appendix

A short section demonstrating the intuitiveness of the QAPSolver, written from scratch with C++ using Visual Studio 2019 Community Edition. The QAPSolver is very easy to setup with little knowledge on C++ and can be used readily with the code below, using the preprocessing code to process data from text files into matrices:



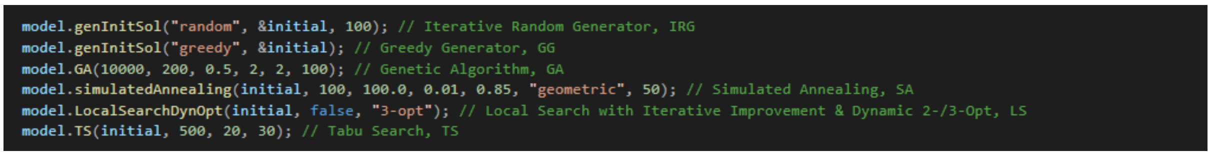
```cpp
#include <iostream>
#include <vector>
#include <string>
#include <numeric>
#include "QAP.h"
#include "Preprocessing.h"

int main() {
    // Problem instances from QAPLIB
    std::string path = "had12.txt";

    // Initialization of matrices, arrays and other variables
    std::vector<std::vector<int>> F; // flow matrix
    std::vector<std::vector<int>> D; // distance matrix
    std::vector<int> initial; // initial solution (permutation)
    unsigned int size; // = initial.size();

    // Load the data and process it into F and D matrices
    preprocessing(&path, &F, &D, &size);

    // Initialize the QAP model given F and D matrices
    QAPSolver model(&F, &D, &size);

    // Generate initial solution and solve using the available algorithms
    model.genInitSol("random", &initial, 100); // generate initial solution with IRG, comment this out if using GA
    model.TS(initial, 50, 7, 10); // solve using algorithm, in this case Tabu Search
    model.prtSolution(); // print the best permutation, solution and run time
}
```

The data in ".txt" file format shares the same layout as that in the ".dat" file format from QAPLIB. Below shows what to expect from the output:



```
Permutation : (3,10,11,2,12,5,6,7,8,1,4,9)
Solution : 1652
Run Time : 0.141296s

----QAP SOLVER TERMINATED----


D:\QAP\x64\Debug\Quadratic Assignment Problem.exe (process 10076) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Below shows how all six algorithms can be called once a QAPSolver model is initialized, the function arguments are explained in the QAP header file.



```cpp
model.genInitSol("random", &initial, 100); // Iterative Random Generator, IRG
model.genInitSol("greedy", &initial); // Greedy Generator, GG
model.GA(10000, 200, 0.5, 2, 2, 100); // Genetic Algorithm, GA
model.simulatedAnnealing(initial, 100, 100.0, 0.01, 0.85, "geometric", 50); // Simulated Annealing, SA
model.LocalSearchDynOpt(initial, false, "3-opt"); // Local Search with Iterative Improvement & Dynamic 2-/3-Opt, LS
model.TS(initial, 500, 20, 30); // Tabu Search, TS
```

There are other methods the QAPSolver model is equipped with besides prtSolution(), but in general the above demo code is more than enough to get the solver up and running to solve Quadratic Assignment Problems from QAPLIB.

References

[1] T. C. Koopmans and M. Beckmann, “Assignment Problems and the Location of Economic Activities,” *Econometrica*, vol. 25, no. 1, p. 53, Jan. 1957, doi: 10.2307/1907742.

[2] L. Steinberg, “The Backboard Wiring Problem: A Placement Algorithm,” *http://dx.doi.org/10.1137/1003003*, vol. 3, no. 1, pp. 37–50, Jul. 2006, doi: 10.1137/1003003.

[3] J. W. Dickey and J. W. Hopkins, “Campus building arrangement using topaz,” *Transp. Res.*, vol. 6, no. 1, pp. 59–68, Mar. 1972, doi: 10.1016/0041-1647(72)90111-6.

[4] R. E. Burkard and J. Offermann, “Entwurf von Schreibmaschinentastaturen mittels quadratischer Zuordnungsprobleme,” *undefined*, vol. 21, no. 4, Aug. 1977, doi: 10.1007/BF01918175.

[5] A. M. Geoffrion and G. W. Graves, “Scheduling Parallel Production Lines with Changeover Costs: Practical Application of a Quadratic Assignment/LP Approach,” *https://doi.org/10.1287/opre.24.4.595*, vol. 24, no. 4, pp. 595–610, Aug. 1976, doi: 10.1287/OPRE.24.4.595.

[6] J. Krarup and P. M. Pruzan, “Computer-aided layout design,” *Math Progr. Study*, no. 9, pp. 75–94, 1978, doi: 10.1007/BFB0120827.

[7] E. L. Lawler, “The Quadratic Assignment Problem,” *http://dx.doi.org/10.1287/mnsc.9.4.586*, vol. 9, no. 4, pp. 586–599, Jul. 1963, doi: 10.1287/MNSC.9.4.586.

[8] L. Kaufman and F. Broeckx, “An algorithm for the quadratic assignment problem using Bender’s decomposition,” *Eur. J. Oper. Res.*, vol. 2, no. 3, pp. 207–211, May 1978, doi: 10.1016/0377-2217(78)90095-4.

[9] A. M. Frieze and J. Yadegar, “On the quadratic assignment problem,” *Discret. Appl. Math.*, vol. 5, no. 1, pp. 89–98, Jan. 1983, doi: 10.1016/0166-218X(83)90018-5.

[10] W. Adams and T. Johnson, “Improved linear programming-based lower bounds for the quadratic assignment problem,” pp. 43–75, Aug. 1994, doi: 10.1090/DIMACS/016/02.

[11] S. Sahni and T. Gonzalez, “P-Complete Approximation Problems,” *J. ACM*, vol. 23, no. 3, pp. 555–565, Jul. 1976, doi: 10.1145/321958.321975.

[12] R. E. Burkard, S. E. Karisch, and F. Rendl, “QAPLIB – A Quadratic Assignment Problem Library,” *J. Glob. Optim. 1997 104*, vol. 10, no. 4, pp. 391–403, 1997, doi: 10.1023/A:1008293323270.

[13] F. Ahyaningsih, “A combined strategy for solving quadratic assignment problem,” *AIP Conf. Proc.*, vol. 1867, no. 1, p. 020006, Aug. 2017, doi: 10.1063/1.4994409.

[14] C. H. Heider, “A Computationally Simplified Pair-Exchange Algorithm for the Quadratic Assignment Problem.” 01-Nov-1972.

[15] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of State Calculations by Fast Computing Machines,” *J. Chem. Phys.*, vol. 21, no. 6, p. 1087, Dec. 2004, doi: 10.1063/1.1699114.

[16] S. Katoch, S. S. Chauhan, and V. Kumar, “A review on genetic algorithm: past, present, and future,” *Multimed. Tools Appl.*, vol. 80, no. 5, pp. 8091–8126, Feb. 2021, doi: 10.1007/S11042-020-10139-6/FIGURES/8.

[17] F. Glover and M. Laguna, “Tabu Search,” *Handb. Comb. Optim.*, pp. 2093–2229, 1998, doi: 10.1007/978-1-4613-0303-9_33.

[18] S. W. Hadley, F. Rendl, and H. Wolkowicz, “A New Lower Bound Via Projection for the Quadratic Assignment Problem,” *https://doi.org/10.1287/moor.17.3.727*, vol. 17, no. 3, pp. 727–739, Aug. 1992, doi: 10.1287/MOOR.17.3.727.

[19] Catherine Roucairol, “DU SEQUENTIEL AU PARALLELE : LA RECHERCHE ARBORESCENTE ET SON APPLICATION A LA PROGRAMMATION QUADRATIQUE EN VARIABLES 0.1 (Book, 1987) [WorldCat.org],” 1987.

[20] C. E. Nugent, T. E. Vollmann, and J. Ruml, “An Experimental Comparison of Techniques for the Assignment of Facilities to Locations,” *https://doi.org/10.1287/opre.16.1.150*, vol. 16, no. 1, pp. 150–173, Feb. 1968, doi: 10.1287/OPRE.16.1.150.

[21] M. Scriabin and R. C. Vergin, “Comparison of Computer Algorithms and Visual Based Methods for Plant Layout,” *http://dx.doi.org/10.1287/mnsc.22.2.172*, vol. 22, no. 2, pp. 172–181, Oct. 1975, doi: 10.1287/MNSC.22.2.172.