CHAPTER 2

METAHEURISTICS

2.1 Introduction

Many combinatorial optimization problems have been proven to be NP-hard, and sometimes even NP-complete. It is well known that a combinatorial optimization can be classified as P, NP-hard and NP-complete, which has been discussed in great length in the previous chapter. Metaheuristic has contributed significantly in solving many combinatorial optimization problems which were once difficult to solve. The growth of applying metaheuristics in solving combinatorial problems is very fast as these combinatorial problems are important to the industrial world.

Examples of some of the very well known classical combinatorial problems include, but are not limited to, the china postman problem, travelling salesman problem, vehicle routing problem, minimum spanning tree problem, integer programming, supply chain problem, inventory routing problem and scheduling problem. Due to the inherent complexity of these problems, metaheuristics is often used to find a good or near-optimal (optimal) solution.

The term metaheuristics is a combination of two Greek words, *heuristic* and *meta where* the word *heuristic* taken from the verb *heuriskein* means "to find", whilst the word *meta* means "beyond, in an upper level" (Glover, 1986). Glover and Laguna (1997) then extended the definition as metaheuristics which "refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest of local optimality".

A heuristics is a technique (consisting of a rule or a set of rules) which seeks (and hopefully finds) good solutions within a reasonable computational cost. Heuristics is an approximation method in the sense that it provides a good solution for relatively little effort, but does not guarantee optimality (Voss, 2001). However, large problems could be difficult to be solved by using the heuristics method until recently when a better and more efficient way of solving large scale neighbourhoods have been found (Pisinger and Ropke, 2010). The effectiveness of very large scale neighbourhood search algorithm depends on a good neighbourhood structure as the structure will determine and guide the search.

An alternative definition for metaheuristic is “an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, in which learning strategies are used to structure information in order to find efficiently near-optimal solutions” as given in (Osman and Laporte, 1996).

Voss (2001) elaborated further by stating that “a meta-heuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iterations. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method”.

Another definition given by Stutzel (1999) as cited in Blum and Roli (2003) is “Metaheuristics is a typically high-level strategy which guides an underlying and more problem specific heuristic to increase its performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descents by allowing the local search to escape from local optima”. This is achieved by either worsening moves or generating new starting solutions for the local search in a more “intelligent” way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly.

This bias can be of various forms and can be casted as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on the probabilistic decisions made during the search. But the main difference to pure random search is that in metaheuristic algorithms randomness is not used blindly but in an intelligent, biased form.

Many researchers (Soriano and Gendreau, 1996; Toth and Vigo, 2003) have reported the success in finding near-optimal solutions by applying metaheuristics in many hard problems. Currently metaheuristics includes (but is not limited to) Evolutionary Computation (EC), Tabu Search (TS), Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Variable Neighborhood Search (VNS), Simulated Annealing (SA), Iterated Local Search (ILS), Scatter Search (SS) and Guided Local Search (GLS). Most of the methods that are applied in hard problems produce good results; however not all methods can be applied to all kinds of problems. The question is how to determine the method that can be used to solve combinatorial optimization problems and the method that works best for various problems.

Blum and Roli (2003) did a research on the importance of metaheuristics from the conceptual approach. They introduced a framework called Intensification and Diversification (I&D) to find the relationship between intensification and diversification in metaheuristics. The I & D frame shows and explains the metaheuristics method in a conceptual way (emphasizing in the strong point criterion), which helps in understanding the strength of the metaheuristics. It is important to know the strength of a metaheuristics, which will help in choosing the right method for a specific problem. Both I & D are the two very important concepts as they determine the behaviour of a metaheuristic. Diversification refers to the exploitation of the search space whilst intensification refers to the exploitation of the accumulated search experience.

Finding the balance between diversification and intensification is crucial and very important, as diversification will search the region of solution space with high quality solutions, whilst intensification ensures that not too much time is spent in the region of solution space which has been explored or provides poor quality solutions.

2.2 Classifications of Metaheuristics

A good metaheuristic will produce near optimal solution in a reasonable computational time. There are several similarities and dissimilarities between metaheuristics, which allow either grouping them together or separating the algorithms. Metaheuristics can be classified or divided in several ways based on similarities or dissimilarities. Seeing the metaheuristics method from some selective characteristics or properties to differentiate among them can be a way of classifying it.

Gendreau and Potvin (2005) are adopted in this research as they categorize metaheuristics into single solution vs. population based algorithm. The same classification has been mentioned in Blum and Roli (2003) where the single solution algorithm is referred to as the trajectory method.

A. Single Solution Based Metaheuristics

Single Solution based metaheuristics which is also known as the trajectory method works on improving a single solution. The trajectory move can be viewed as ‘walking’, the term coined by Talbi (2009). An iterative improvement is employed to move from the current solution to an improved (better) solution in the solution space. As described in Blum and Roli (2003), the search process of trajectory can be viewed as discrete dynamical system. The algorithm starts from an initial solution and describes a trajectory in the state space. The dynamical system depends on the strategy used. One of the common strategies often used in combinatorial optimization is the iterative improvement method which includes hill climbing and random restart hill climbing. Single solution metaheuristics includes SA, TS, Greedy Randomized Adaptive Search Procedures (GRASP) and VNS. These methods will be elaborated in the following section. Figure 2.1 shows an example of an improvement when trajectory moves are applied to an initial solution in search of space S.

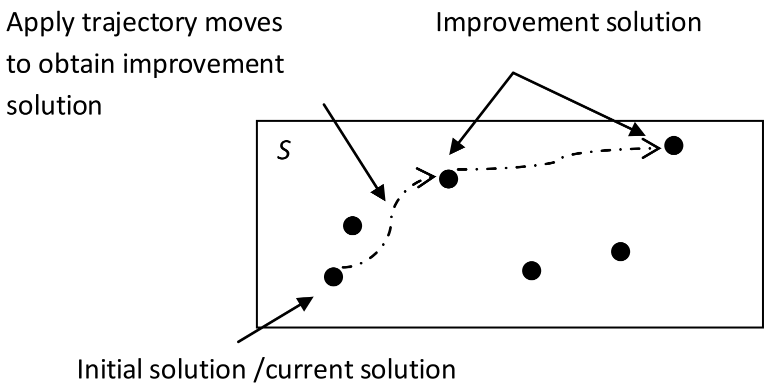
Apply trajectory moves to obtain improvement solution
Improvement solution
S
Initial solution /current solution

Figure 2.1: Example of trajectory moves.

B. Population Based Metaheuristics

Population based metaheuristics handles a set of solution at every iteration. The examples of population based metaheuristics include ACO, Evolutionary Algorithms (EAs) and Scatter Search (SS). In population based metaheuristics, several candidates are evaluated in each iteration. The main aim is to optimize each candidate globally compared to the local search which is to optimize selected candidates locally.

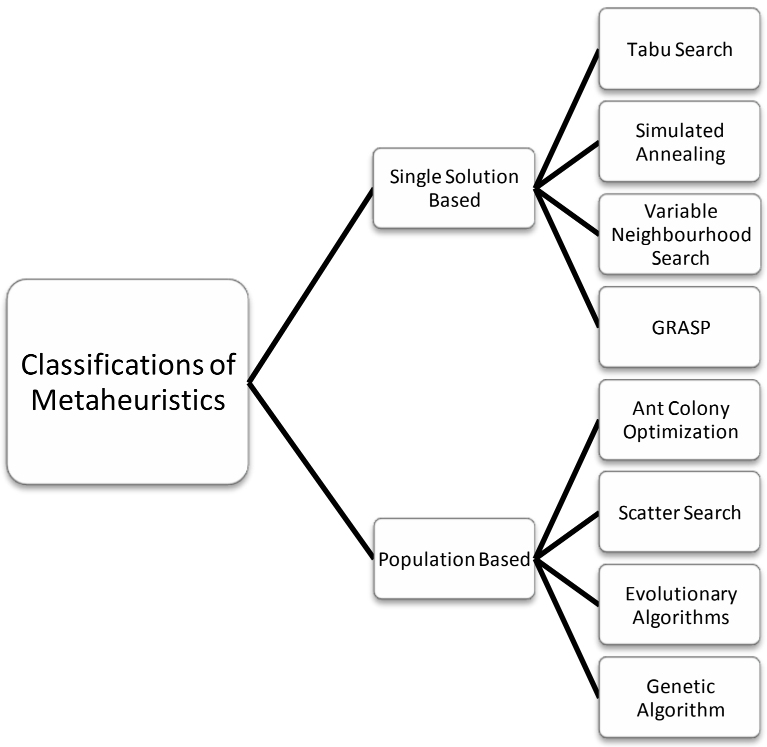

Figure 2.2: Classifications of metaheuristics and its examples.

There are also other classifications suggested by Blum and Roli (2003) such as *Nature-inspired vs. non-nature inspired, Population-based vs. single point search, Dynamic vs. static objective function, One vs. various neighbourhood structures and Memory usage vs. memory-less methods.* The details of the description can be found in Blum and Roli (2003).

Nature-inspired vs. non-nature inspired.

Population-based vs. single point search.

Dynamic vs. static objective function.

One vs. various neighbourhood structures.

Memory usage vs. memory-less methods.

Figure 2.3: Innovative suggestions on classifying the metaheuristics method.

2.2.1 Single Solution Based Metaheuristics

A. Simulated Annealing (SA)

Simulated Annealing (SA) was first introduced independently by Cerny (1985) and Kirkpatrick (1983) as cited in Blum and Roli (2003) and Gendreau and Potvin (2005). The underlying principle was to allow movement that was worse than the current solution in order to escape the local optima in the hope of finding the global optima. SA is based on the analogy of the physical annealing process where crystalline solid is heated by increasing its temperature and then allowed to cool very slowly until it achieves its most regular possible crystal lattice configuration, which is when it attains low energy states. If the cooling schedule is done very slowly, the resulting crystal has an excellent structure and is free of crystal defects. On the contrary, if the process is done too fast, the structure will be unsmooth and often contain a lot of defects.

In combinatorial optimization, the search process is done in two phases. The first phase is to allow a random search with higher probability, thus giving the opportunity for a worse solution to be accepted. The probability (which depends on parameter *T*, the global defined temperature) is gradually decreased and the algorithm enters a second phase which employs an iterative improvement method to intensify the search area. The algorithm of SA is represented as follows (Blum and Roli, 2003):

Step 1 *Initialization.*

Select an initial solution $\omega \in \Omega$, temperature change counter $k = 0$, temperature cooling schedule,t_k, initial temperature $T = t_0 \geq 0$. Select a maximum repetition schedule, M_k, that defines the number of iterations executed at each temperature, t_k.

Step 2 *Repeat below until stopping criterion is met.*

Set repetition counter $m = 0$

```
Repeat below until $m=M_k$
        Generate a solution $\omega' \in N(\omega)$
        Calculate $\Delta_{\omega,\omega'} = f(\omega') - f(\omega)$
        If $\Delta_{\omega,\omega'} \leq 0$, then $\omega \leftarrow \omega'$
        If $\Delta_{\omega,\omega'} > 0$, then $\omega \leftarrow \omega'$ with probability $\exp(-\Delta_{\omega,\omega'}/t_k)$
        $m \leftarrow m+1$
        $k \leftarrow k+1$
```

The SA formulation results in $M_0 + M_1 + \cdots + M_k$, $\sum_{i=0}^{n} M_i$ total iterations being executed, where *k* corresponds to the value for t_k at which some stopping criterion is met (for example, a pre-specified total number of iterations has been executed or a solution of a certain quality has been found). In addition, if $M_k = 1$ for all *k*, then the temperature changes at each iteration.

B. Tabu Search (TS)

Tabu Search (TS) is a metaheuristics method proposed by Fred Glover in 1986, based on his previous work (Blum and Roli, 2003). The term *tabu* (or *taboo)* is from a Polynesia's language: Tongan of Tonga Island (Glover and Laguna, 1997) which refers to things that cannot be touched because it is sacred. TS is a deterministic local strategy that escapes local optima by allowing a substitution of current solution with the new solution found; even if it is far (worse) from the current solution. This allows non-improving solution to be kept.

Glover and Laguna (1997) argued that for an algorithm to qualify as intelligent, it must incorporate an adaptive memory feature. In the context of TS, this adaptive memory is implemented in short term and long term memories. In the short term memory, all the moves that have been examined are stored in a tabu list to prevent revisiting or cycling.

In the implementation of TS, four main strategies are of great importance: short-term memory, long-term memory, diversification and intensification. Many different variants of these strategies have been described in detail by Glover and Laguna (1997).

Gendreau and Potvin (2010) claimed that TS is an extension of a Local Search (LS). And Simple TS is just a LS combined with a short term memory. And *tabus* is the distinctive element that actually differentiates TS and LS. As mentioned earlier, the *tabus* or Tabu list is to prevent cycling back when the search process is done through a non-improving solution. Preventing the *cycling* needs the use of memory to track its steps and origins and eventually avoiding it from happening again. The listed *tabus* is usually restricted to a certain quantity (usually decided by the user). And usually in TS, a partial specified solution is stored rather than a complete solution as the complete solution is expensive in terms of memory and running time.

The use of memory in TS is based on four principal dimensions; quality, influence, recency and frequency. Quality is to assure that the recently visited solution is leading to a good solution or bad solution. For a bad solution, some mechanism will be applied to penalize from pursuing the search space from there. Influence is the impact of a quality solution and the structure of search; in other words, the influence of searching for a decision that guides to good and bad solutions. This gives TS additional information and the ability to learn; which guides throughout the search. Recency is the ability to keep track of the preceding recent solution.

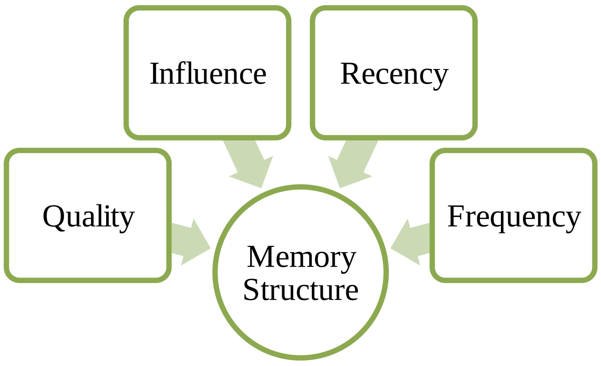

Figure 2.4: Memory structure of the Tabu Search.

An algorithm of TS taken from Gendreau (2003) is presented here. The notation used in the algorithm is defined first followed by the algorithm.

Notation :

S	: the current solution,
S^*	: the best-known solution,
f^*	: value of S^*
$N(S)$	: the neighbourhood of S,
$N(\tilde{S})$	: the “admissible” subset of $N(S)$ (i.e., non-tabu or allowed by aspiration).

```
Step 1 Initialization. Choose (construct) an initial solution, $S_0$
    Set $S = S_0$ , $f^* = f(S_0)$ , $S^* = S_0$ , $T = \emptyset$
Step 2 Search Process
    While termination criterion is not satisfied do
        Select $S$ in $\mathrm{argmin}[f(S')]$; such that $S' \in \tilde{N}(S)$
        If $f(S) < f^*$, then set $f^* = f(S), S^* = S$
        Record tabu for the current move in T (delete oldest entry if
        necessary)
    End of while
```

The termination criterion for TS is not specified, but the most commonly used criteria are maximum number of iteration, maximum *cpu* time allowed, and the maximum number of iterations without an improvement in the objective function value and if the objective value reaches a pre-specified threshold value(Gendreau, 2003).

C. Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) was first introduced by Pierre Hansen and Nenad Mladenovic in 1997. This metaheuristic method is based on the systematic changes of neighbourhood within the local search algorithms (Mladenovic and Hansen, 1997). Exploration of the search space is carried out by the local search which allows the algorithm to jump from one neighbourhood to another. This allows the algorithm to escape from the local optimum.

Some of the interesting properties of the algorithm are simplicity, coherence, efficiency, effectiveness, robustness, user-friendliness, and innovation. The simplicity can be found in the basic principle of VNS that is a systematic change of the neighbourhood. Another aspect of VNS is the simple algorithm which allows embedding any local search that is suitable for specific combinatorial problems or else, it can be decided to not have any local search at all. The entire criteria can be found in the algorithm itself (Hansen *et al.*, 2008).

There are three phases of the main VNS: *Shaking, Local Search,* and *Move or Not.* The details regarding the three phases are discussed in the following section.

Algorithm of VNS

The algorithm of the VNS as taken from Hansen and Mladenovic (2001) is given as follows:

Step 1 *Initialization*

Select the set of neighbourhood structures N_k, $k=1,\ldots, k_{max}$, that will be used in the search; find an initial solution x; choose a stopping condition;

Step 2 *Repeat the following until the stopping condition is met:*

(i) *Set $k \leftarrow 1$;*

(ii) *Until $k=k_{max}$, repeat the following steps:*

(a) Shaking. Generate a point x' at random from the k^{th} neighbourhood x ($x' \in N_k(x)$);

(b) *Local Search. Apply some local search method with* x' *as initial solution denote with x” the so obtained local optimum;*

(c) *Move or not. If this local optimum is better than the incumbent, move there* $(x \leftarrow x'')$*, and continue the search with* $N_1(k \leftarrow 1)$*; otherwise, set* $k \leftarrow k+1$

In step 1, the neighbourhood structure N_k is initialized. The maximum number of neighborhood k_{max} is then defined and the stopping conditions that will be used throughout the entire search process are decided upon. After that the initial solution x is initialized or found.

In step 2, the neighbourhood of an initial solution must be first defined. There are several ways of defining neighbourhood. An illustration of the neighbourhood structure is shown in Figure 2.5.

As mentioned earlier in step 2, the algorithm can be divided into three phases: *Shaking, Local Search* and *Move or Not.* In the *shaking* step, a solution x' is picked randomly from the k^{th}neighbourhood of the current solution, x. This will ensure that the solution is not far from the current best solution x. The solution will then be perturbed using the local search in the local search phase in the hope of finding a better solution. Figure 2.6 shows an example of the *shaking* step and *local search* applied where the shaded area represents the local search applied. The local optimum produced by the local search is represented by x” (triangular shape).

In the *local search* step, any local search method can be adopted until an improvement has been found or until it is terminated by some predefined stopping condition. If there is an improvement when the new solution x'' is compared with the initial solution x, x'' will replace x in *Move or Not* phase. A new neighbourhood of x'' will be defined and the algorithm starts again with N_1 the first neighbourhood of the new solution. If no better solution is found, the algorithm will continue with the $k + 1^{th}$ neighbourhood. Note that the change in neighbourhood only happens if no new improvement is found. Figure 2.7 and 2.8 show an example of the *Move or Not* step.

Defining the stopping condition can differ from one program to the other. Most algorithms adopt the maximum number of iterations k_{max} as the stopping condition. Other criteria such as maximum running time or *cpu* time allowed, or maximum number of iterations between two improvements can also be defined in the algorithm. Often successive neighbourhoods N_k will be nested (Mladenovic and Hansen, 1997). This simply means that a better solution can be found in the neighbourhood of the current solution which is relatively far (still in the neighbourhood) from the current solution. So it is termed as successive nested neighbourhoods. The new found solution will be used as the current solution, allowing for a substantial jump from the current solution. The neighbourhood structure is illustrated in Figures 2.5, 2.6 and 2.7.

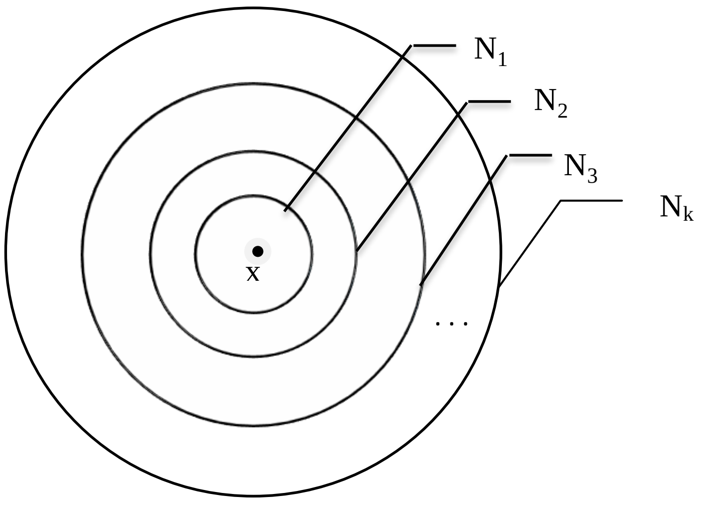

Figure 2.5: Illustration of neighbourhood structure for $N_1(x) \rightarrow N_k(x)$.

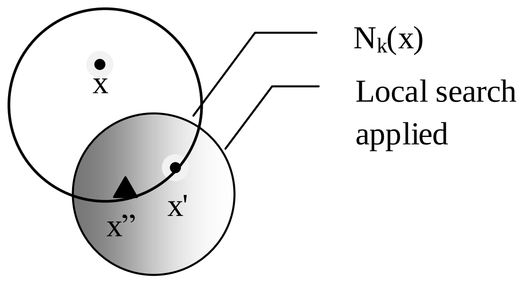

Figure 2.6: Illustration of the *Shaking* step.

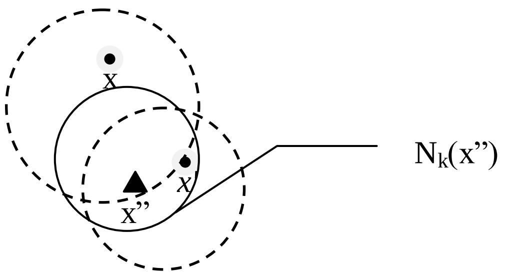

Figure 2.7: Illustration of the new neighborhood structure N_1.

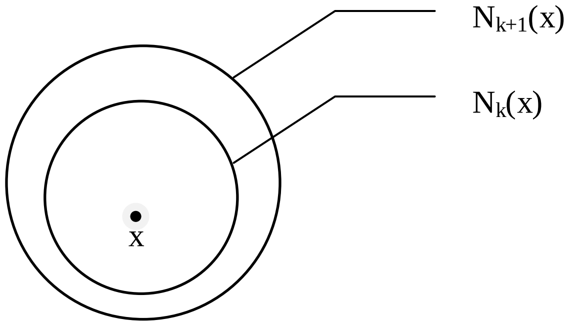

Figure 2.8: Illustration of the N_{k+1} neighbourhood structure.

Note that a local optimum for one neighbourhood is not necessarily a local optimum for another neighbourhood. This means that the optimum solution can differ from one neighbourhood to another. The local optimums for several neighbourhoods are usually close to each other.

Neighbourhood Structure

The neighbourhood structure can be defined in several ways and it is very much dependent on the nature of the problem. The neighbourhood structure can be generated using for instance 1-interchange (or vertex substitution), vertex deletion or addition, node based or path based, k-edge exchange or comparing the symmetric difference between two solutions or Hamming distance. The choice of the neighbourhood should exploit different properties and characteristics of the search space, that is, the neighbourhood structures should provide different abstractions of the search space (Blum and Roli, 2003).

In this research, the neighbourhood k , N_k is defined as a distance function, that is the cardinality of the symmetric difference between any two solutions $V_1$and V_2 written as $\rho(V_1, V_2) = |V_1, V_2|$ or $\rho = |V_1 \Delta V_2|$ where Δ denotes the symmetric difference operator. Note that the change of neighbourhood can be achieved by introducing local search heuristics. The choices of the neighbourhood will be described in great detail in Chapters 3 and 4.

Variants of VNS

In this section, several variants of VNS are presented, focusing on those related to this research. These variants have been widely applied in solving many *NP*-hard or *NP*-complete problems. The Variable Neighbourhood Descent or VND is an example of deterministic changes of neighbourhood. The algorithm of VND is shown as follows:

Step 1 *Initialization*

Select the set of neighbourhood structures N_k, $k=1,\ldots, k_{max}$, that will be used in the search; find an initial solution x; choose a stopping condition;

Step 2 *Repeat the following until no improvement is obtained:*

(i) *Set $k \leftarrow 1$;*

(ii) *Until $k=k_{max}$, repeat the following steps:*

(a) *Exploration of neighbourhood. Find the best neighbour x' of x $(x' \in N_k(x))$;*

(b) *Move or not. If the solution thus obtained x' is better than x, set $x \leftarrow x'$; otherwise, set $k \leftarrow k+1$.*

Another variant of VNS is known as Reduced VNS (RVNS) where the random (solution) point is selected from neighbourhood N_k. This algorithm is very useful if the local search is very costly. Note that there is no *Local Search* step applied in the algorithm. The improvement depends on the neighbourhood structure itself. This algorithm is often adopted to solve large scale problems as applying local search is very expensive and often increases the processing time drastically. The algorithm of RVNS is shown as follows: (Hansen and Mladenović, 2001)

Step 1 *Initialization*

Select the set of neighbourhood structures N_k, $k=1,\ldots, k_{max}$, that will be used in the search; find an initial solution x; choose a stopping condition;

Step 2 *Repeat the following sequence until the stopping condition is met:*

(i) *Set $k \leftarrow 1$;*

(ii) *Repeat the following steps until $k = k_{max}$:*

(a) *Shaking. Generate a point x' at random from the k^{th} neighbourhood of x $(x' \in N_k(x))$;*

(b) *Move or not. If this point is better than the incumbent, move $x \leftarrow x'$, and continue the search with $N_1(k \leftarrow 1)$; otherwise, set $k \leftarrow k + 1$.*

Other variants of VNS include Skewed VNS (SVNS), Variable Neighbourhood Decomposition Search and Parallel VNS. Refer to Hansen and Mladenovic (1997, 2001, 2003) for a comprehensive explanation of these methods.

2.2.2 Population Based Metaheuristics

A. Evolutionary Computation (EC)

Evolutionary Computation (EC) is the algorithm that is inspired by nature. Nature has the capabilities to evolve and adapt to its environment. The EC mimics the evolutionary process in a computational way where instead of using 'iteration', the term 'generation' is used as it mimics the real process of evolutionary. Borrowing the term from nature, a population of chromosomes is created where each chromosome represents a potential solution to the problem on hand. The population will then undergo several processes such as recombination and mutation to produce a new population. The recombination operator is a process where two individuals (or more) are combined to produce new individuals, whilst the mutation operator modifies an individual and acts as a safety net to recover information that has been lost through the process of recombination.

The key to the recombination process is the process of selection. The selection process selects individuals based on the fitness value where it gives a measure of the objective value with respect to the problem on hand. The process of selection is biased on fit individuals where they are allocated more copies to contribute to the next generation. This is inspired by the principle of survival of the fittest in natural evolution, where the *fitness* is measured as the ability to adapt to the environment.

Many new variants of EC have evolved over the years and they can be grouped in three different categories: Evolutionary Programming (Fogel, 1962; Fogel*et al.*, 1966; cited in Blum and Roli, 2003), Evolutionary Strategies (Rechenberg, 1973; cited in Blum and Roli, 2003), and Genetic Algorithm (GA) (Holland, 1975; cited in Blum and Roli, 2003). EC introduces a small perturbation into the algorithm through mutation. However, recently, many researchers introduced the concept of crossover while the demarcation line that separated between the EC and GA has become very blurred. The algorithm of EC can be expressed as follows (Blum and Roli, 2003):

```
Step 1 Generate an initial population, P
Step 2 Evaluate the fitness value (objective function) of P
Step 3 While termination condition is not met does:
    Apply Recombination operator $(P' \leftarrow P)$
    Apply Mutation operator $(P'' \leftarrow P)$
    Evaluate fitness value of P”
    Apply Selection Process with $P'' \cup P$ to obtain the new P
```

B. Genetic Algorithm (GA)

GA was first proposed by John Holland in 1975 and his book entitled “Adaptation in Natural and Artificial Systems” outlines the general principals of GA (Reeves, 2003). This algorithm is based on Darwin’s principal of “survival of the fittest” where the best individuals will survive and reproduce. His book describes how to adapt the principals in the optimization method and since then GA has been a powerful technique in solving many problems of optimization (Reeves, 2003; Michalewicz, 1996).

Holland’s student Ken DeJong details GA in his doctorial thesis entitled “An Analysis of the Behavior of a Class of Genetic Adaptive Systems” (Reeves, 2003). In 1989, Holland’s former student, David Goldberg produced a book with a complete guidance to GA and his book entitled “Genetic Algorithms in Search, Optimization and Machine Learning” has become one of the basic references in many researches in GA.

GA is motivated by the biological mechanism, so the solution or string is represented as chromosomes. In biological evolution, chromosomes are made up of genes, and a gene encodes specific properties, for example the specific characteristics can be an individual’s eye and hair colour (Sivanandam and Deepa, 2008). Other concepts of natural evolution such as selection, crossover and mutation are also adopted in the artificial algorithm.

Encoding

Encoding or representation of an individual or solution plays an important role in any GA. It has such a strong impact on the alteration process (crossover and mutation operator). The chromosomes should contain information about the solution it represents (Sivanandam and Deepa, 2008). This will determine how close the solutions are to the real solutions. The genotype and phenotype mapping are two important terms in GA. Genotype is gene setting or genetic constitution of a cell while phenotype is the physical expression of the genotype.

Chromosomes or solution representation can be encoded in several ways such as gray coding, binary representation, octal representation, and integer representation which are among the most forms reported in the literature. Binary and integer representations can usually be categorized as vector representation. The simplest representation is binary. Figure 2.9 shows an example of the binary representation. Each bit in the binary represents some characteristics of the solution.

Chromosome 1: 010001111
Chromosome 2: 110110110
Chromosome 3: 111000010

Figure 2.9: Binary representation.

An example of the integer representation is represented in Figure 2.10. Permutation representation is usually used in problem with ordering or sequencing. For example, this representation is used in classical China Postman Problem or Travelling Salesman Problem (TSP). The chromosome is represented as permutations of integer where these integers represent the order in which the cities are visited.

Chromosome 1: 123456789
Chromosome 2: 124376958
Chromosome 3: 231457689

Figure 2.10: Permutation representation.

Operators in GA

Selection

Selection is the process of determining the number of copies that will be allocated to a chromosome for the next operation. The number of copies is often determined by the fitness value of the chromosome. Each chromosome has its corresponding fitness value where higher values signify better chromosomes in a maximization case and vice versa. The relative fitness value is considered so that only the best chromosomes are given a higher chance of being selected to go to the next process (Goldberg, 1989).

The simplest form of selection is the Roulette Wheel Selection (RWS). RWS selects parents by spinning a biased weighted roulette wheel. The weighted roulette is divided according to the chromosome's fitness value, where a fitter chromosome represents more area in the wheel. Hence, this represents higher probability for a particular chromosome to be chosen as the wheel spins. However, RWS is biased as the same chromosomes may be selected each time and it is not guaranteed that the best chromosome will be selected during the process. The area constructed is proportional to $\frac{f_i}{\sum_{j=1}^{n} f_j}$ where f_i is the fitness value of chromosome i and n is the number of chromosomes in the population.

Figure 2.11 shows an example of the RWS. Consider a population consisting of 5 chromosomes, with the probabilities of selection proportional to the area of the sector of the roulette wheel. Numbers 0.00, 0.25, 0.40, 0.60 and 0.85 shown are the cumulative probabilities. A single pointer (arrow) has to be spun 5 times for selection.

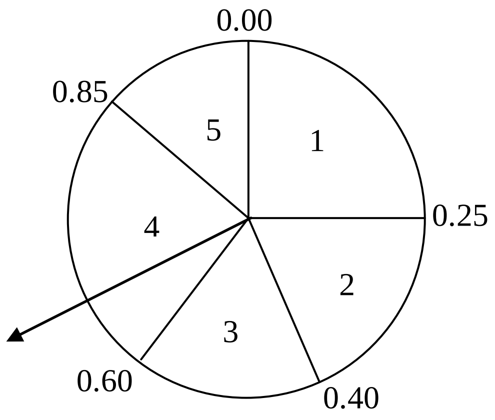

Figure 2.11: Roulette Wheel selection.

Stochastic Universal Sampling (SUS) is introduced to overcome the weakness of the RWS. As in the RWS, SUS constructs the wheel based on the fitness value of each of the chromosome in the population. Instead of using a one pointer that has to be spun n times to get n chromosomes, SUS employs n equidistant pointers that is spun once to obtain n chromosomes. SUS is known to provide zero bias and minimum spread. An example of SUS is shown in Figure 2.12. Instead of having one pointer, SUS has 5 equally spaced pointers that are to be spun once for all the 5 selections.

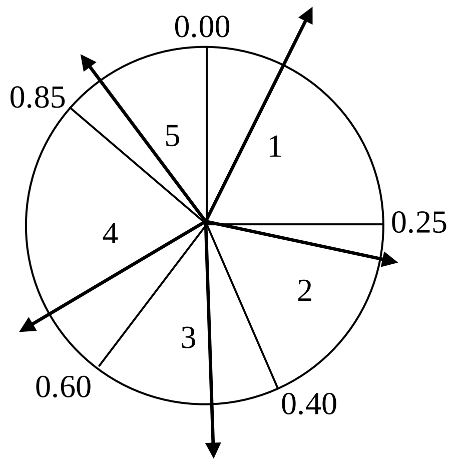

Figure 2.12: Stochastic Universal Sampling selection.

Crossover

Crossover is one of the most important operators. This operator allows information exchange between two chromosomes. Two good chromosomes will exchange information which allows good qualities from both parents to be inherited in the new offspring (child). By doing this, the child will (it is hoped) surpass the parents or in other words a better chromosome will be born. There are many types of crossovers for example, One Point Crossover, Partially Mapped Exchange (PMX), Order Based Crossover (OX), Path Based Crossover, Edge Recombination Operator (ERO) and Enhanced Edge Recombination Operator (EERO). EERO is an operator that emphasizes adjacency information instead of the order or position in the sequences. EERO is adapted in one of the applications in this research and a detailed description of it will be given in the next chapter.

One Point Crossover

Figure 2.13 illustrates an example of the One Point Crossover. Here, a binary representation is used. The parents will be cut at randomly selected points (cutting points are represented as a dashed line) and two new offspring are born. The first offspring will be from Parent 1 until the cutting point and then it will continue with Parent 2.

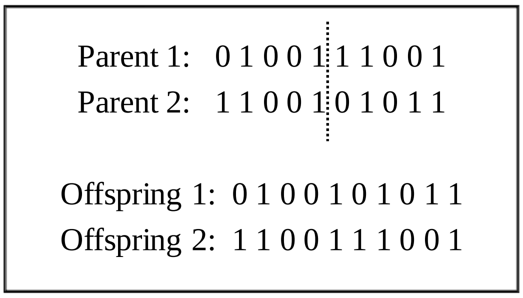
Parent 1: 0 1 0 0 1 1 1 0 0 1
Parent 2: 1 1 0 0 1 0 1 0 1 1
Offspring 1: 0 1 0 0 1 0 1 0 1 1
Offspring 2: 1 1 0 0 1 1 1 0 0 1

Figure 2.13: One point crossover.

Two Point Crossover

Figure 2.14 illustrates an example of a two point crossover. The dashed line represents the crossover point.

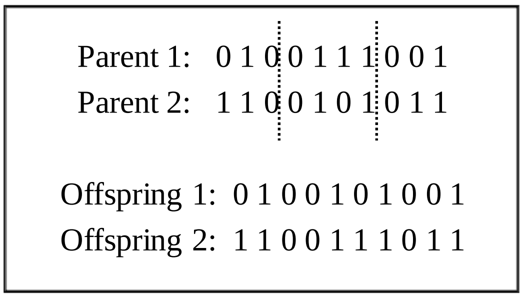
Parent 1: 0 1 0 0 1 1 1 0 0 1
Parent 2: 1 1 0 0 1 0 1 0 1 1
Offspring 1: 0 1 0 0 1 0 1 0 0 1
Offspring 2: 1 1 0 0 1 1 1 0 1 1

Figure 2.14: Two point crossover.

Edge Recombination Operator (ERO)

ERO is a heuristics operator that emphasizes on the edges of parents instead of the vertex. This is the method proposed by Grefenstette (1987) and is sometimes known as Edge Recombination Crossover (ERX). Basically an offspring is made up exclusively from the edges of a pair of parent. This is done by listing all the edges of both parents and then reconnecting them (using certain conditions) again in the offspring. The algorithm of ERO is stated as follows:

Step 1 *Randomly select a city to be the current city of the offspring.*

Step 2 *Select four edges (two from each parent) incident to the current city.*

Step 3 *Define a probability distribution over selected edges based on their cost. The probability for the edge associated with a previously visited city is 0,*

Step 4 *Select an edge. If at least one edge has non-zero probability, selection is based on the above distribution; otherwise, selection is random (from unvisited cities),*

Step 5 *The city on 'the other end' of the selected edge becomes the current city.*

Step 6 *If the tour is complete, stop; otherwise, go to step 2.*

Mutation

Mutation is the process of applying a small perturbation to the solutions encoded in the chromosomes that may result in a huge effect. Applying mutation can introduce diversity in the search space and it acts as a safety net to the information that might have been lost during the process of selection and crossover. An example of a mutation by changing one bit of a binary string is shown in Figure 2.15 where the representation used is a vector representation of a binary number.

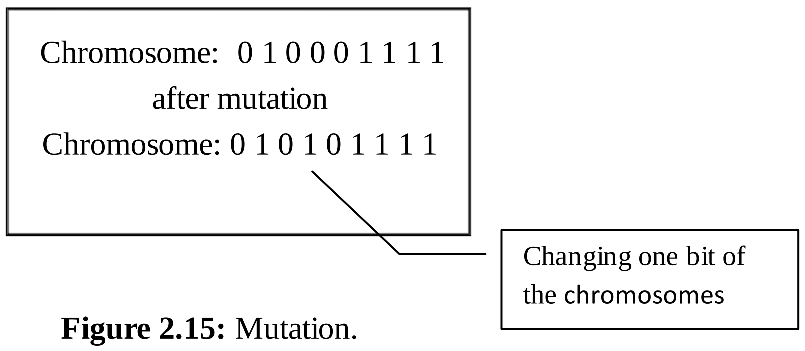
Chromosome: 0 1 0 0 0 1 1 1 1
after mutation
Chromosome: 0 1 0 1 0 1 1 1 1
Changing one bit of the chromosomes

Figure 2.15: Mutation.

In this research, the inversion and swap have been applied as the mutation operator where the chromosomes are represented as permutation of integers. These operators will be described in detail in the next chapter.

Hybrid GA

It is widely acknowledged that the main weakness of GA is that it often fails to compete successfully with approaches that are specifically designed for a particular problem. Suh and Van Gucht (1987) have recognized the need to incorporate some local search heuristics or problem specific knowledge into the GA framework known as the hybrid GA if a more competitive technique is desired. One of the interesting features of the GA paradigm is its domain independence; the fact that it operates on a coding of a problem and needs to know nothing more about the problem itself, creating a hybrid algorithm which almost inevitably means losing this characteristic because of the introduction of some problem specific knowledge in the algorithm.

Due to the robust nature of the GA framework, local search heuristics or problem specific knowledge can be introduced in several ways. Many researchers (for example: Braysy, Berger and Barkaoui, 2000) modified either the crossover operators or the mutation operators to include problem specific knowledge. Local search techniques have also been adopted to act as mutation operators in some of the problems (Land,1998; Preux and Talbi, 1999; Mühlenbein, 1992). Alternatively, the local search can also be used to generate a set of initial solutions (seeded population) rather than starting randomly (Sivanandam and Deepa, 2008). Many examples of the hybrid GA have been shown to give better solutions and improve efficiency (Gendreau and Potvin, 2005).

Several attempts have been made to hybridize GA with other metaheuristics such as the Ant Colony Optimization (ACO) method, Artificial Neural Network (ANN), Tabu Search (TS) and Simulated Annealing (SA). For example, in the hybridization with ACO, GA is employed to find the solutions globally and ACO is used as a local search in order to improve the quality of the solutions found (Bilchev and Parmee, 1995). Another example with ACO is done by Chen and Lu (2005). In their paper, GA and ACO were combined by using the characteristics in both algorithms, where the convergence and diversity in ACO were exploited using the multiplicity of population in GA, known as HGACO.

2.3 Local Search

Local search is generally an approach that is used to estimate solutions for hard problems. Most of the metaheuristics embed local search in the algorithm in order to improve the incumbent solution and increase efficiency. With the inclusion of a good local search method, the algorithm is able to find almost near optimal (or optimal solution) in a shorter time.

A local search algorithm is an iterative improvement method that can be applied generally to any problem that has a well defined solution space Ω, an evaluation function $f(S)$, where $S \in \Omega$, and a neighbourhood generation mechanism, $N(S)$. Starting from an initial configuration, which is often randomly generated, the algorithm generates a series of solutions by repeatedly searching the neighbourhoods of the current solution for a better configuration to replace the existing solution. Without loss of generality, assuming that the problem is one of minimization, the aim is to find an optimal solution $S^* \in \Omega$ such that $f(S^*) \leq f(S)$, $\forall S \in \Omega$. A solution S' is accepted if it produces a better objective value, i.e. $f(S') < f(S)$. The new solution can be selected in several ways such as accepting the first improved solution or choosing solutions with the greatest improvement in the neighbourhood, known as steepest descent in a minimization problem (steepest ascent in the context of maximization).

Local search works by starting with an initial solution that may be generated randomly. Small changes may be applied until it reaches optimum or until no further changes can be made. The algorithm often keeps track of one state at a time and it is a deterministic improvement.

Many of the combinatorial optimization problems are *NP-hard* (Garey and Johnson, 1979), which means that the optimal solution for an *NP-hard* problem cannot be found within the polynomial bounded computation times (Aarts and Lenstra, 2003). Many local search techniques have been developed to address this problem.

The local search is problem dependent and is selected based on the characteristics of the problems. The quality of the solution is very much dependent on the choice of the initial solution. Therefore, in many cases the algorithm is repeatedly applied with different initial configurations, a method which will asymptotically result in optimal solutions if all the solutions are used as starting solutions, which is almost impossible in many real world problems.

Local search heuristics can be divided into two main categories: Constructive Heuristics and Improvement Heuristics. Constructive heuristics constructs a solution by adding the element one at a time while improvement heuristics tries to improve the current solution.

A. Constructive Heuristics

Constructive heuristics in general are the fastest approximate methods. These heuristics start with an empty solution, then extend or collect information until the solution is completely constructed. The extending process is repeated until the solution is complete or some stopping criteria are met. The algorithm of the constructive heuristics is given as follows:

```
Step 1 Set an empty variable, S=[].
Step 2 Determine all the possible extension for S, N(S).
Step 3 While $N(S) \neq 0$
          Find S' from N(S)
          Extend S by appending solution component S' in it.
          Determine N(S) that is all possible extension of new S.
Step 4 Complete constructed solution.
```

An example of Constructive Heuristics is the Nearest Neighbour Method that is a greedy heuristics, Insertion Heuristics and Christofieds method. Generalized Insertion Method (GENI) is also included in the Constructive Heuristics.

B. Improvement Heuristics

Improvement heuristics basically helps improve the solution. Improvement heuristics is usually applied to trap a local optimal solution by allowing deterioration span. Improvement heuristics starts with a complete solution (a complete tour can be constructed using construction heuristics) and then tries to improve by using (any) heuristic algorithms.

In this research, a few improvement heuristics are used, for example *2-opt*, *3-opt* and or-opt to improve the solution. These heuristics are embedded as a local search in the hope of finding a better solution. By using improvement heuristics, the structure of the complete solution is manipulated. The algorithm of Improvement Heuristics is given as follows:

```
Step 1 Initialization. Generate an initial solution $x_0$. Set the current solution
          $x = x_0$ and objective value $Z = Z_0$.
```

Step 2 *Set* $x' = x$ *and* $Z' = Z$

Step 3 *Repeat until* $Z < Z'$

Find candidate solution $x" \in N(x')$ *with objective value* $Z"$ *where* $N(x')$*is the neighbourhood of solution* x'.

If $Z" < Z$. *Set* $x = x"$ *and* $Z = Z"$.

2.3.1 Local Search for Travelling Salesman Problems and Related Problems

Traveling Salesman Problem (TSP) is a classical problem which represents many real life applications. It has been one of the most intensive areas that have been investigated with many local search algorithms developed. TSP can easily be adapted to many routing problems which is the subject of this research.

Given a list of cities, N and the distance from each city, solving TSP is to find the shortest possible tour that visits each city exactly once. With a little modification, changing the idea city to others, for example customers, suppliers or trucks, and the idea distance to cost or travelling time, TSP can be viewed as other problems or appear to be a sub-problem in other combinatorial optimization problems such as, Inventory Routing Problem and Vehicle Routing Problem.

Many local searches have been developed for TSP, for example swap, insertion, *2-opt*, *3-opt* and *k*-opt. Usually, for large combinatorial optimization problems, the running time is too long, so a simple local search will be applied to reduce the running time. Some of the local searches in TSP which include the simplest to apply like swap and insertion and the more innovative like GENI are described in the following section.

Swap

The swap method is included in the improvement heuristics. This method will randomly swap between any two cities in a tour. The swap is not limited to only two cities; the swapping can also be done on several more cities. Example of swap heuristics is shown in Figure 2.16 where Figure 2.16 (a) shows two cities selected to go through the swap, while (b) represents the resulting tour after the swap by deleting the old edges (represented by the red line) and substituting them with the new edges (represented by the dotted lines).

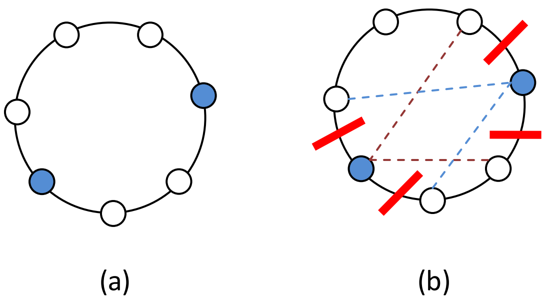

Figure 2.16: A swap heuristics.

Insertion

Insertion heuristic involves selecting a city randomly to form a tour. Insertion heuristic can belong to both the categories of constructive and improvement heuristics depending on its application. The constructive heuristics starts off with a city (it can sometimes be a sub-tour) and then the insertion heuristics is applied until all the listed cities are in the tour (forms a complete tour). While the improvement heuristics starts from a complete solution (complete tour), selects a city randomly to leave the tour (by some heuristics) and then inserts randomly in the other places in the tour. There are many variants of insertion heuristics which can easily be adapted to others (other combinatorial problems). An example of the *Nearest Insertion* is outlined as follows:

Step 1 *Select a sub-tour with minimum cost possible.*

Step 2 *Select a city that is not in the sub-tour that connects to any cities in the* *sub-tour with minimum distance.*

Step 3 *Insert the selected city in the sub-tour so that the insertion is minimum.*

Step 4 *Repeat Steps 2and 3 until a complete tour is obtained.*

Nearest Neighbour Method (NNM)

NNM is a method adopted from Skellam (1952) which is applied to determine if the data are clustered by using the ratio and mean value of the nearest neighbour distances. NNM is categorized in constructive heuristics. It starts with a node then continues to insert the nearest neighbour to the current node (city) to form a tour. This will form a tour, but it will not necessarily be an optimal tour. Here, the term *nearest* can be in many scale forms (example: cost and distance), and the most common scale used is the Hamiltonian distance.

2-*opt*

2-opt algorithm is basically deleting two edges of a tour, causing the tour to break off into two paths and then reconnecting both those paths back. The resulting tour must be a valid tour. The processes of deleting and reconnecting the tour are done until no further *2-opt* improvements can be found. It is said that the tour is now optimal. *2-opt* also falls into both categories with a little modification. In Figure 2.17, the original tour is shown on the left. The dotted lines are the two edges to be deleted. The one on the right shows a valid tour after reconnection.

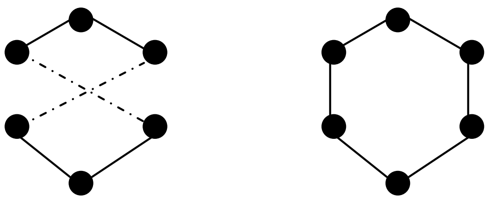

Figure 2.17: An example of *2-opt* algorithm.

3-opt

3-opt algorithm is similar to the *2-opt* algorithm; but instead of deleting two edges, *3-opt* will delete three edges. The tour is then broken into three paths and the paths are then reconnected such that the resulting tour is valid. Figure 2.18 is an example of the *3-opt*. The original tour is on the left where the dash dotted lines are the three edges to be deleted. The right shows two ways of reconnecting (dotted dash lines) to produce valid tour.

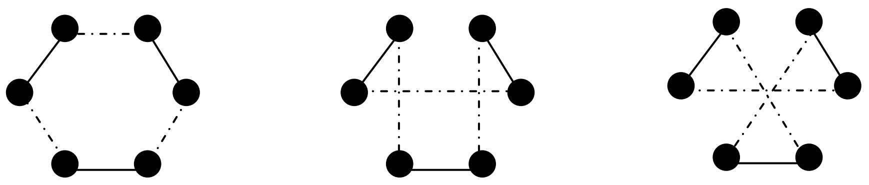

Figure 2.18: An example of *3-opt* algorithm.

Or-opt

Or-opt was first introduced by Or (1976). Or–opt is actually a node exchange heuristics. Usually, Or-opt is considered as a special case of *3-opt,* where three arcs are moved and substituted by other 3 arcs. Or-opt will delete 2 arcs when removing a chain of consecutive nodes, while the third is deleted when reinserting the chain back into the tour. Compared to *3-opt,* Or-opt has a lesser number of changes which results in less computational time. The algorithm of the Or-opt can be described as follows:

Step 1 *Consider an initial tour and set* $t = 1$ *and* $s = 3$.

Step 2 *Remove from the tour a chain of* s *consecutive vertices, starting with the vertex in position* t*, and tentatively insert it between all the remaining pairs of consecutive vertices on the tour.*

(i) *If the tentative insertion decreases the cost of tour, implement it immediately, thus defining a new tour. Set* $t = 1$, *and repeat Step 2.*

(ii) *In no tentative insertion decreases the cost of tour, set* $t = t + 1$. *If* $t = n + 1$ *then proceed to Step 3, otherwise repeat Step 2.*

Step 3 *Set* $t = 1$ *and* $s = s - 1$. *If* $s > 0$ *go to Step 2, or else Stop.*

k-opt algorithm

*2-opt*and *3-opt* are actually k-opt algorithm with k=2 and k=3.There is also a *4-opt* move which was once called *the crossing bridges* (Johnson, McGeoch and Rothberg, 1996). Another well known version of the *k-opt* is a special case of *3-opt* where the edges are not disjointed (two of the edges are adjacent to one another).

Savings Algorithm

Another example of the constructive heuristics is the Clarke and Wright (CW) savings algorithm (Clarke and Wright, 1964). The algorithm is based on the notion of savings, and it is a very simple and easy to apply algorithm. It simply merges two routes$(0, \dots, i, 0)$ and $(0, \dots, i, 0)$ into a single route $(0, \dots, i, j, \dots, 0)$with a distance saving of$s_{ij} = c_{i0} + c_{0j} - c_{ij}$. The algorithm is stated as follows (Laporte and Semet, 2002) :

Step 1 *Savings Computation. Compute the savings* $s_{ij} = c_{i0} + c_{0j} - c_{ij}$ *for* $i, j = 1, \dots, n$ *and* $i \neq j$. *Creates* n *vehicles route* $(0, i, 0)$ *for* $i = 1, \dots, n$. *Order the savings in a nonincreasing fashion.*

Step 2 *Route Extension. Consider in turn each route* $(0, i, \dots, j, 0)$. *Determine the first saving* s_{ki} *or* s_{jl} *that can feasibly be used to merge the current route with another route containing an arc or edge* $(k, 0)$ *or an arc or* *edge $(0, l)$.Implement the merge and repeat this operation to the current route. If no feasible merge exists, consider the next route and reapply the same operations. Stop when no route merge is feasible.*

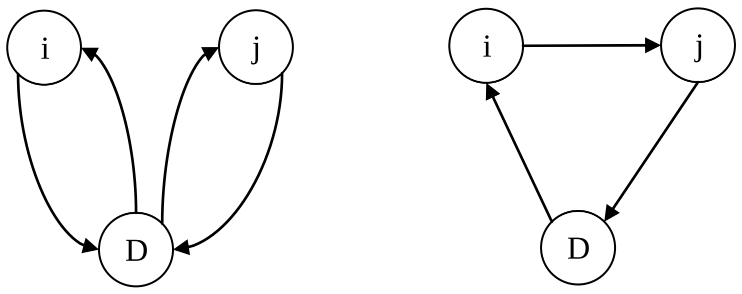

Figure 2.19: An illustration of savings algorithm.

Figure 2.19 shows an illustration of the savings algorithm; where the basic savings is by joining two routes into a route (if and only if the connection produces smaller distance). Another version of the savings algorithm is also available where the merging yielding the largest saving is implemented. This saving is called parallel saving algorithm. In this algorithm, multiple routes are considered at one particular time. The algorithm is given as follows:

Step 1 *Same as Step 1 in original CW savings algorithm.*

Step 2 *Best Feasible Merge. Starting from the top of the savings list, execute the following. Given a saving s_{ij}, determine whether there exist two routes, one containing arc or edge $(0, j)$ and the other containing arc or edge $(i, 0)$, that can feasibly be merged. If so, combine these two routes by deleting $(0, j)$ and $(i, 0)$ and introducing (i, j).*

Sweep Algorithm

The sweep algorithm is introduced by Gillett and Miller (1974). A simple implementation of this method is as follows. Assume each vertex $i$is represented by its polar coordinates (θ_i, ρ_i), where θ_i is the angle and ρ_i, is the ray length. Assign a value $\theta_i^* = 0$ to an arbitrary vertex $i^*$and compute the remaining angles from $(0, i^*)$. Rank the vertices in increasing order of their θ_i.

Step 1 *Route Initialization. Choose an unused vehicle*k.

Step 2 *Route Construction. Starting from the unrouted vertex having the smallest angle, assign vertices to vehicle* k*as long as its capacity or the maximal route length is not exceeded. If unrouted vertices remain, go to* ***Step 1***.

Step 3 *Route Optimization. Optimize each vehicle route separately by solving the corresponding TSP (exactly or approximately).*

2.4 Shortest Path Heuristics

A shortest path problem is one of the common problems when solving network flow. The problem applications arise in many combinatorial and network optimization problems. The applications include data packet, phone call, vehicle routing problem and supply chain problem where there is usually the need to send or deliver products (material) from one destination to another. An efficient heuristics for the shortest path problem must be able to send data (product) through the network as fast as possible with the lowest costs possible (Ahuja *et al.*, 1993).

The shortest path problem is first defined. Then this continues with algorithm that can be used or solve it. The definition of the Shortest Path Problem here is taken from Ahuja et.al (1993). It is defined as: Consider a directed network $G = (N, A)$, with an arc length (or arc cost) c_{ij} associated with each arc $(i, j) \in A$. The network has a distinguished node s, called the source node. Let *A(i)* represent the arc adjacency list of node i and let $C = max\{c_{ij} : (i, j) \in A\}$. The shortest path problem is to determine every non-source node $i \in N$ as the shortest length directed path from node s to node i.

There are several shortest path heuristics that have been proposed in the literature including but not limited to Dijkstra Algorithm, Bellman Ford Algorithm and Floyd Algorithm (also known as Floyd-Warshall algorithm).

A. Dijkstra Algorithm

Dijkstra algorithm is proposed by Dijkstra (1959). The problem is to construct the tree of minimum total length between the n nodes. (A tree is a graph with one and only one path between every two nodes). The use of the term branches and nodes are in the algorithm for easy understanding. The branches for this algorithm are subdivided into three sets. The sets are as follows:

I. the branches are definitely assigned to the tree under construction (they will form a sub-tree) ;

II. the branches from which the next branch to be added to set I, will be selected;

III. the remaining branches will be rejected or not considered yet.

The nodes used in the algorithm are subdivided into two sets. The sets are as follows:

A. the nodes connected by the branches of set I,

B. the remaining nodes (one or only one branch of set II will lead to each of these nodes).

The construction algorithm starts by choosing any arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. Start with set I which is empty. Then, perform these steps repeatedly.

Step 1 *The shortest branch of set II is removed from this set and added to set I. As a result, one node is transferred from set B to set A.*

Step 2 *Consider the branches leading from the node that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.*

Then return to Step 1 and repeat the process until sets II and B are empty. The tree required is to be formed by the branches in set I.

An illustration of the Dijkstra Algorithm

Considered graph of 6 nodes, A, B, C, D and E with weighted cost for the corresponding edge, as given in Figure 2.20. The objective is to find the shortest path from source node A to destination node F.

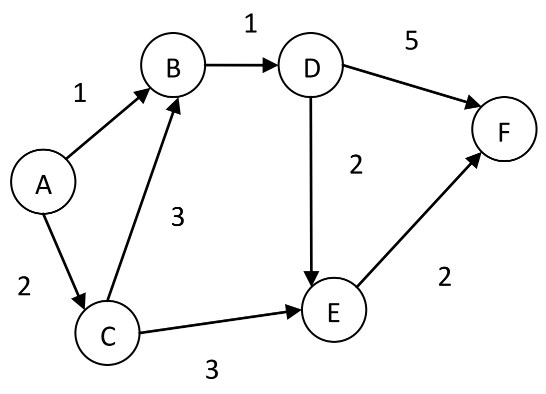

Figure 2.20: A graph with weighted costs.

Shown in Figure 2.21 Considered node A, then find the minimum distance from the unconsidered node that connects to A. From the figure we can see it is B. Now B will be in the considered nodes. In Figure 2.22 finds the next node that connects to considered node A and B. Here are C and D, but D is minimum compared to C, so D is now in the considered node list.

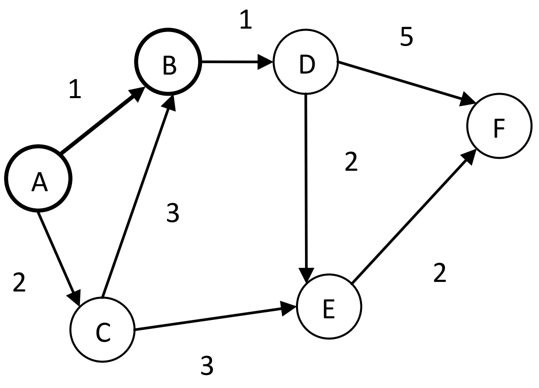

Figure 2.21: Consider node A, and the next considered node found is node B.

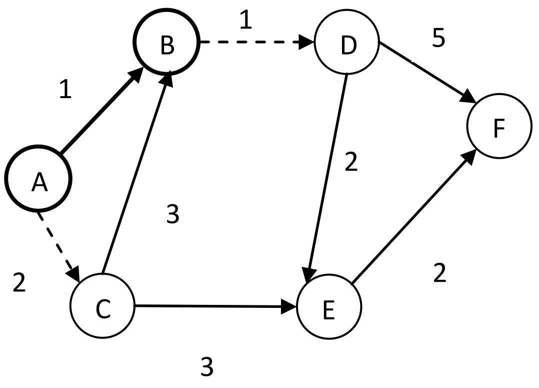

Figure 2.22: Find the next unconsidered node that connects to node A and B.

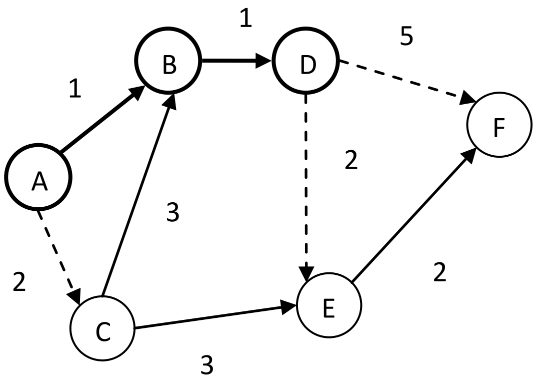

Figure 2.23: Find the next unconsidered node that connects to node A, B and D.

We then consider the next unconsidered node that connects to A, B and D. From Figure 2.23, the next unconsidered node that connects to A, B and D are C, E and F; with C and E are of the same distance. In addition, from A to E through B and D is 4, and through C is 5, and A to F through B and D is 7, hence E will be considered next in the node list.

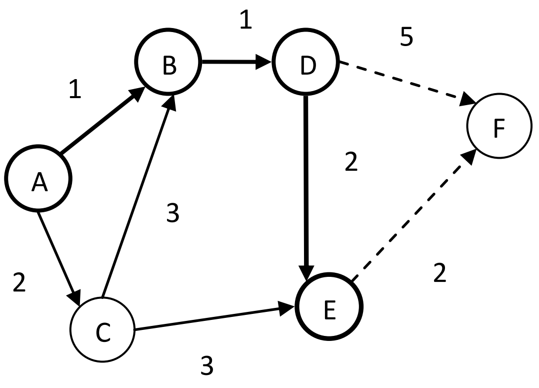

Figure 2.24: Node E is chosen among the unconsidered nodes.

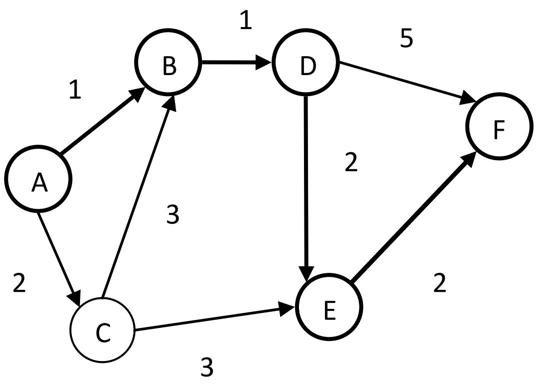

Figure 2.25: Shortest path for the given graph.

Figure 2.24 illustrates the next connection to be considered. The shortest distance to connect F is through nodes D and E with distance 5 and 2, respectively. The minimum distance is from node E, hence node E will be connected to F. Figure 2.25 shows the shortest path found using Dijkstra algorithm.

B. Bellman Ford's Algorithms

Bellman Ford's Algorithm finds the shortest path from a graph not only for positive weighted graphs but also for negative weighted ones. If negative weighted graphs are not dealt with then the Dijkstra algorithm is a more efficient choice as Dijkstra runs at a faster time. The structure of Bellman Ford is similar to Dijkstra's, except instead of greedily selecting the next minimum weight node to relax, Bellman Ford will *relax* the entire nodes (except the source node) for $|V|-1$ times where V is the number of vertex (node). *Relax* here means checking the edge of the considered nodes (check for the shortest one).

2.5 Minimum Spanning Tree (MST)

Minimum spanning tree (MST) is finding the minimum cost that spans all nodes from a weighted graph. Consider a graph, *G*. A weighted graph is a graph where each edge has a weight (usually represented by a real number). Commonly, the weight is cost or distance.

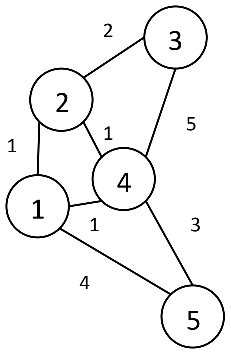

Figure 2.26: Example of a weighted graph.

Spanning tree T of G is a connected acyclic sub-graph that spans all the nodes. The problems of MST are: Given an undirected graph $G=(N,A)$ with $n=|N|$ nodes and $m=|A|$ arcs and the length or cost c_{ij} associated with each arc $(i,j)\in A$ (Ahuja *et al.*, 1993). Shown in Figure 2.27, the leftmost figure represents a graph and the middle and the rightmost represent a spanning tree respectively.

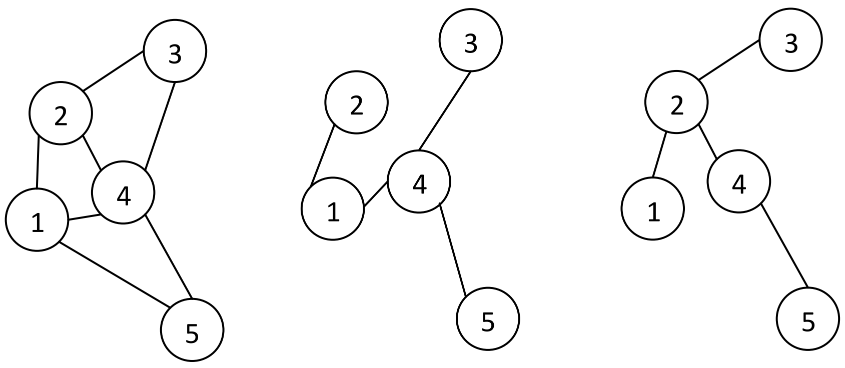

Figure 2.27: Example of a graph and its spanning tree.

Ahuja *et al.* (1993) stated that MST is different with the shortest path problem in two different ways. First, the arc of MST is undirected which can be represented by$arc\ (i,j) =\ arc\ (j,i)$. Second, is in the calculation of the minimum *cost;* in MST the arc is calculated exactly once while in the Shortest path algorithm, the cost is calculated according to the number of paths from the root node that pass through that arc.

A. Kruskal's Algorithm

Kruskal's Algorithm is stated as follows. Let V be an arbitrary but fixed (nonempty) subset of the vertices of$G$. Then, perform the following step as many times as possible: among the edges of G which are not yet chosen but which are connected either to a vertex of V or to an edge already chosen pick the shortest edge which does not form any loops with the edges already chosen. The set of edges eventually chosen forms a spanning tree of G, and in fact it forms a shortest spanning tree.

B. Prim's Algorithm

Prim's algorithm is similar to Kruskal's algorithm, where it starts with a vertex V, then finds the node with the shortest path (edge) that connects to V. The basic idea is to expand the tree by adding the shortest edge (path) to the tree (one at a time).

2.6 Concluding Remarks

In this chapter, the introduction of metaheuristics methods was presented. It was first introduced by several definitions from various authors, focusing on some well known metaheuristics that were commonly developed for combinatorial optimization problems.

The method of classifying metaheuristics proposed by Gendreau and Potvin (2005) and Blum and Roli (2003) was also presented which was classified by solution based; Single solution based metaheuristics (including trajectory search) or populations based metaheuristics. There were also some other classifications proposed by Blum and Roli (2003).

This was then continued with some explanations on some other well known metaheuristics, such as TS, SA, EC, GA and VNS. Each of this metaheuristic method was explained with the basic idea and the algorithm given. However, GA and VNS were explained in great detail with illustrations of the algorithm as used in this research.

Then, the local search heuristics which was commonly used in TSP was discussed. Basically the local search heuristics could be categorized into two: Constructive heuristics and Improvement heuristics. The definitions of both the constructive and improvement heuristics were given, along with the algorithms. Construction heuristics occurred by adding one node at a time to a single node to form a tour, while improvement heuristics worked with a complete tour, and then continued with an improvement heuristics to amend the tour. Local search heuristics reviewed in Section 2.3.1 included but not limited to insertion, swap, *2-opt*, *3-opt*, *k-opt*, *or-opt* and savings algorithm.

The topics of the shortest path heuristics and minimum spanning tree were discussed in Sections 2.4 and 2.5 respectively. Two examples on shortest path algorithm were reviewed, which were: Dijkstra algorithm and Bellman Ford’s algorithm. The same was done to minimum spanning tree, in which the Kruskal’s algorithm and Prim’s algorithm were reviewed. These two subtopics consisting of the shortest path algorithm and minimum spanning tree were important as both of them were used in this research.