The Plant Propagation Algorithm for Discrete Optimisation

Birsen İrem Selamoğlu

A Thesis Submitted for the Degree of

Doctor of Philosophy

Department of Mathematical Sciences

University of Essex

August 2017

Dedicated to

My beloved mother, father and brother.

SUMMARY

The thesis is concerned with novel Nature-Inspired heuristics for the so called NP-hard problems of optimisation. A particular algorithm which has been recently introduced and shown to be effective in continuous optimisation is the Plant Propagation Algorithm or PPA. Here, we intend to extend it to cope with combinatorial optimisation. In order to show that our extension is viable and effective, we consider three types of problems which are good representatives of the whole topic. These are the Travelling Salesman Problem or TSP, the Knapsack Problem or KP and the scheduling problem of Berth Allocation as arises in container ports or BAP. Because PPA is a population-based search heuristic, we devote a chapter to the important issue of generating good and yet computationally relatively light initial populations of solutions to kick start the search process. In the case of the TSP we revisit and extend the Strip Algorithm (SA). We introduce the 2-Part SA and show that it is better than the classical SA. We also introduce new variants such as the Adaptive SA and the Spiral SA which cope with clustered cities and instances with cities concentrated around the center of the unit square, respectively. In the case of KP we adapt the Roulette Wheel selection approach to generate solutions to start with PPA. And in the case of BAP, we introduce a number of simple heuristics which consider a schedule as a flat box with one side being the processing time and the other the position of vessels on the wharf. The heuristics try to generate schedules by avoiding overlap as much as possible. All approaches and algorithms are implemented and tested against well established algorithms. The results are recorded and discussed extensively. The thesis ends with a conclusion and ideas for further research.

DECLARATION

The work in this thesis is based on research carried out at the Department of Mathematical Sciences, University of Essex, United Kingdom. No part of this thesis has been submitted elsewhere for any other degree or qualification, and it is all my own work, unless referenced, to the contrary, in the text.

ACKNOWLEDGEMENTS

First, I would like to thank my family for their continuous support and endless patience with me during this long journey. They always believed in me and encouraged me especially, when I lost my confidence in my studies. Without their support, this study would not have been completed. This thesis is dedicated to them.

I am deeply grateful to my supervisor, Professor Abdellah Salhi. I greatly appreciate his guidance, support and how much he influenced me with his wisdom and knowledge throughout my PhD studies.

I would also thank my external examiner, Professor El-Ghazali Talbi and my internal examiner, Dr. Andrew Harrison. Their comments and feedback enhanced the quality of this thesis.

My study has been financially funded by the Turkish government through the Ministry of National Education. I would like to thank the Ministry of National Education.

I would like to thank the academic and administrative staff in the Department of Mathematical Sciences, University of Essex, UK.

Next, I would like to thank my friends whom I shared my tears and laughter during this journey. Special thanks to Hulya Kocagul Yuzer, Canan Ugur Rizzi, Didem Sanver, Gokce Caylak Kayaturan, Derya Tekin and Demet Caltekin. Also, I would like to thank Ezgi Curuk, Gizem Basak Berk and Hande Gul Atasagun for supporting me regardless of the distance between us.

Finally, I would like to thank my ex-supervisor from the Department of Industrial Engineering at Cukurova University, Associate Professor Oya Hacire Yuregir who has always been a great influence on me since 2005.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS & ABBREVIATIONS

2-PSA= 2-part Strip Algorithm

ABC= Artificial Bee Colony Algorithm

ASA= Adaptive Strip Algorithm

BA=Bat Algorithm

BAP= Berth Allocation Problem

BB=Branch and Bound

CABC= Combinatorial Artificial Bee Colony Algorithm

CS= Cuckoo Search Algorithm

CSA=Classical Strip Algorithm

DABC= Discrete Artificial Bee Colony Algorithm

DBA=Discrete Bat Algorithm

DCS= Discrete Cuckoo Search Algorithm

DP= Dynamic Programming

DPSO= Discrete Particle Swarm Optimization

EDFA= Evolutionary Discrete Firefly Algorithm

ESH= Edge Seeking Heuristic

FF= Firefly Algorithm

GA= Genetic Algorithm

H-PPA-SbPPA= Hybrid PPA and SbPPA

ICS= Improved Cuckoo Search Algorithm

KP=Knapsack Problem

LP= Linear Programming

MA= Memetic Algorithm

MABC= Modified Artificial Bee Colony Algorithm

MKP=Multiple Knapsack Problem

MPPA= Modified Plant Propagation Algorithm

New DFA= Discrete Firefly Algorithm for TSP: A new movement scheme

NP= Non-deterministic Polynomial

P= Polynomial time

PPA= Plant Propagation Algorithm

PPA-C= Plant Propagation Algorithm for Constrained Optimisation

PSO= Particle Swarm Optimization

qCABC= Quick Combinatorial Artificial Bee Colony Algorithm

RBAP=Robust Berth Allocation Problem

RW= Roulette Wheel Method

SA= Strip Algorithm

SbPPA= Seed-based Plant Propagation Algorithm: The Feeding Station Model

SiA= Simulated Annealing

SSA=Spiral Strip Algorithm

TS= Tabu Search

TSP= Travelling Salesman Problem

CHAPTER 1

NATURE-INSPIRED ALGORITHMS FOR DISCRETE OPTIMIZATION

1.1 Introduction

Optimization is finding the best possible solutions for the given problems under given sets of constraints. In general, this is achieved using mathematical methods. Three types of approaches to solve optimisation problems are discussed; exact methods, approximation algorithms and heuristic/metaheuristic methods. Exact methods guarantee optimality. However, they often involve demanding calculations. Approximation algorithms tend to be computationally less expensive at the expense of optimality. They try to generate solutions within predefined bounds, which may exclude the optimum. Like exact methods they usually have strong mathematical underpinnings. Heuristic/metaheuristic approaches have become popular in the last few decades since exact methods are often inconvenient, as they are computationally too expensive for large scale real-world problems. These, generally, do not guarantee to find the optimum solution. But, they are usually able to find near-optimum solutions in reasonable computational times. These algorithms are often Nature-inspired. Nature has inspired scientists with its ability to deal with problems over millions of years. Hence the rapid development in the so called Nature-inspired algorithms or heuristics. This thesis is concerned with one such method, the Plant Propagation Algorithm or PPA, which implements the way the strawberry plant propagates; PPA is also referred to as the Strawberry Algorithm, [9]. It is organised as follows.

Chapter 2 discusses the implementations of PPA on unconstrained and constrained continuous optimisation problems. First, we explain the basic PPA, then a modified and improved version of it is discussed. The main modification is in the way the initial population is generated. Then, the implementation of PPA to solve constrained optimisation problems is discussed. A new methodology to generate the initial population is adopted in this variant too. As the strawberry plant can propagate by seeds as well, another implementation of PPA implements this propagation via seeds. Here, seed dispersion is assumed to be done by animals and birds, mainly. This version has shown a better exploration capability. Considering the good exploitation characteristics of strawberry plant propagation by runners and its good exploration characteristic by seeds, a new hybrid algorithm which combines the two heuristics recently proposed in the literature is discussed as the final variant of PPA. The final section introduces the discrete extension of PPA and gives the basics of its implementation.

Chapter 3 is devoted to the methodology for generating initial populations to start with the algorithm under investigation, i.e. PPA for the different problem types considered. Note that, being a population-based search, starting PPA, and any such algorithm for that matter, with a good population is very important. Three cases will be considered: generating good tours for TSP problems using strip algorithms, generating good item combinations for Knapsack problems, and generating good schedules for the robust Berth Allocation problem as arise in container ports.

In Chapter 4, we propose a discrete variant of PPA to solve TSP. Defining the notations neighborhood and distance are addressed. Representation of tours as plants, computing the distance between two plants and the implementation of short and long runners are explained. Finally, the experimental results are added to show the performance of Discrete PPA compared to that of some well-known metaheuristic methods.

In Chapter 5, we use PPA to solve another well-known family of discrete optimisation problems, namely the Knapsack Problem and some of its variants. First, we describe the different forms of KP. Then we discuss the issues of the implementation of PPA to solve them. The methodologies to implement short and long runners are given and experimental results are reported.

In Chapter 6, we implement PPA to solve a scheduling problem, the Robust Berth Allocation problem that arises in container ports. After explaining the mathematical model we are using, we give strategies and details for implementing short and long runners. We conclude the chapter with our experimental investigation and computational results.

Chapter 7 is the conclusion and suggestions for worthwhile research on Nature-Inspired algorithms and in particular the Plant Propagation Algorithm.

1.2 The Optimisation Problem: General Definition

Optimisation is the operation of maximising or minimising an objective function possibly subject to constraints. The mathematical representation of such a problem is

$$\begin{aligned} &\textit{maximize/minimize} \quad f(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n \\ &\textit{subject to} \quad \phi_j(\mathbf{x}) = 0, \quad (j = 1, 2, ..., M), \\ &\lambda_k(\mathbf{x}) \leq 0 \quad \textit{or} \quad \lambda_k(\mathbf{x}) \geq 0, \quad (k = 1, 2, ..., N), \end{aligned} \tag{1.1}$$

where $f(\mathbf{x})$, $\phi_j(\mathbf{x})$ and $\lambda_k(\mathbf{x})$ are functions of $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$.

In (1.1), the entries of vector $\mathbf{x} = (x_1, x_2, \ldots, x_u)^T$ are the decision variables. $f(\mathbf{x})$ is called the objective function and, $\phi_j(\mathbf{x}) = 0$ and $\lambda_k(\mathbf{x}) \leq 0$ (or $\lambda_k(\mathbf{x}) \geq 0$) are called the equality and the inequality constraints, respectively. The feasible region/search space consists all $\mathbf{x}$ points that satisfy all the constraints, [10, 11].

Decision variables can be continuous, discrete or mixed, i.e. some are real and the others integer. In continuous optimization problems the decision variables take real values, and in the discrete ones, they take integer values. If the optimization problem does not have any constraints, it is called an unconstrained optimisation problem. Otherwise, it is constrained, [10, 11].

Let a real valued function f be defined over a feasible set $S \subset \mathbb{R}^n$, then:

Definition 1.2.1. *The solution $\mathbf{x}^*$ is said to be in the neighborhood of $\mathbf{x}$ if $\mathbf{x}$ satisfies $|\mathbf{x} - \mathbf{x}^*| < \epsilon$.*

For continuous optimisation problems, ϵ is usually a small positive number. For combinatorial problems, it can, for instance, be defined as the number of changes in a permutation, [11].

Definition 1.2.2. *A solution $\mathbf{x}^*$ is said to be a global maximum if $f(\mathbf{x}^*) \geq f(\mathbf{x})$, $\forall\ \mathbf{x} \in S \subset \mathbb{R}^n$. If $f(\mathbf{x}^*) \leq f(\mathbf{x})$, $\forall\ \mathbf{x} \in S \subset \mathbb{R}^n$, the solution $\mathbf{x}^*$ is said to be a global minimum, [12].*

Definition 1.2.3. *A solution $\mathbf{x}^*$ is said to be a local maximum if $f(\mathbf{x}^*) \geq f(\mathbf{x})$, $\forall\ \mathbf{x} \in N(\mathbf{x}^*, \epsilon)$. If*

$f(\mathbf{x}^*) \leq f(\mathbf{x})$, $\forall\ \mathbf{x} \in N(\mathbf{x}^*, \epsilon)$, then $\mathbf{x}^*$ is said to be a local minimum, where $N(\mathbf{x}^*, \epsilon)$ denotes the ϵ-neighborhood of $\mathbf{x}^*$, [12].

1.3 NP-Completeness: A Brief Introduction

The NP-Completeness framework allows us to believe that it is very unlikely that polynomial-time (P) algorithms, in other words, efficient algorithms exist for common problems such as the TSP, KP or Scheduling for instance. It also allows us to extrapolate and state that if we were to find a polynomial-time algorithm for one of the above listed problems and any one of the many known NP-Complete problems, then all of them can be solved efficiently including finding mathematical proofs, and translating in real-time from language to language, to name of few of the most rewarding problems a breakthrough in solving one single NP-Complete problem from a long list that started in the famous book of Garey and Johnson, 1979, [13], will bring. Here, NP refers Non-deterministic polynomial time.

The theory of NP-Completeness assumes that problems which are solvable in polynomial-time are tractable. Recall that the big-O notation is used to describe the asymptotic running-time of an algorithm. It is also important to note that a running time of $O(n^{50})$ is not very encouraging, polynomial times have some nice characteristics that support this assumption. Moore's Law stipulates that the speed of computers doubles every 18 months. This means that for problems that are polynomially solvable, whenever computer speed doubles, larger by a multiplicative factor in size such problems can be solved. This is contrasted with only an additive increase for problems which require exponential time for their solution, [14]. To illustrate consider a problem which is solvable in $O(n^3)$ (polynomial) and one which is solvable in $O(2^n)$ (exponential). If the speed of the computer we use doubles, then we can solve problems which are $\sqrt[3]{2}$ larger than the former problem as the speed will now be twice faster and, $(\sqrt[3]{2} \times n)^3 = 2 \times n^3$. For the latter one, only single variable larger problems can be solved. This is because $2 \times 2^n = 2^{n+1}$. Note that n is the size of a given problem, i.e. the number of variables it involves.

NP-Complete problems are decision problems, i.e. problems with a "yes" or "no" answer. Optimisation problems are easy to convert into decision ones by comparing the solution value to some threshold value. For instance, in the case of the TSP, the decision version of it asks if a given tour has length less than a given value L. In the case of the Knapsack problems, we have:

$$KP - Optimization(p, w, c) = \left\{ \begin{array}{c} max \displaystyle\sum_{j=1}^{n} p_j x_j \\ subject\ to \displaystyle\sum_{j=1}^{n} w_j x_j \leq c, \\ x_j \in \{0, 1\},\ j = 1, \ldots, N. \end{array} \right\} \tag{1.2}$$

where p_j denotes the profit of item j, w_j denotes the weight of item j, c denotes the knapsack capacity and N denotes the total number of items.

$$KP - Decision(p,w,c,t) = \left\{ \begin{array}{c} \text{there exists an } \mathbf{x} \text{ with} \\ \sum_{j=1}^{n} p_j x_j \geq t, \\ subject\ to \sum_{j=1}^{n} w_j x_j \leq c, \\ x_j \in \{0,1\},\ j = 1,\ldots,N. \end{array} \right\} \tag{1.3}$$

where p_j denotes the profit of item j, w_j denotes the weight of item j, c denotes the knapsack capacity and t is a lower bound predetermined.

To be precise, we say that an algorithm accepts a decision problem in polynomial time if there exists an algorithm which runs in time polynomial in the input size and for every instance of the decision problem, if it is a "yes″ instance, then the algorithm should print out "yes″ in polynomial time. Note that no restrictions are put on "no″ instances. This means that for "no″ instance, the algorithm may or may not print "no″ in polynomial time. In fact, it may not terminate at all, for “no″ instance. It is also important to note that optimisation problems whose decision versions are solvable in polynomial time often are themselves solved efficiently.

Problems the decision versions of which can be solved in polynomial time are said to be in the P class. Current knowledge and wisdom among researchers in Operational Research, Mathematics and Theoretical Computer Science is that KP-Decision is NOT in the P class, [13]. However, there is no proof that a polynomial time algorithm does not exist for it. Therefore, we cannot exclude it from the P class for certain. The same can be said of the TSP decision, Scheduling etc. . . .

As we said earlier, only the decision versions of problems are said to be NP-Complete, or in the NPC class; or the P class for that matter. The full optimisation versions are said to be NP-Hard. This is because they are at least as difficult as their decision versions and can be a lot harder. Before, leaving this brief introduction, let us add that the NP class is the class of Nondeterministic polynomially solvable problems. In other words, it contains the problems for which a solution can be guessed in polynomial time and checked in polynomial time. For example, in the case of the TSP decision version, one can guess a permutation of cities in polynomial time; the 1st and last cities are then joined up to make a Hamiltonian cycle or tour. The length of this tour can be computed in polynomial time since it is only a sequence of n additions if n is the number of cities. The resulting length can be compared to see if it is less or equal to a given value L. TSP is therefore in the NP class. The same can be said of other problems whether they are in P or not. In fact $P \in NP$.

1.4 Solution Approaches to Optimisation Problems

In this section, exact methods, approximation algorithms and metaheuristic methods will be reviewed.

1.4.1 Exact Methods

Exact methods are used to find the optimum solution for a given combinatorial optimization problem. The drawback of these methods is, as the size of the instance increases, the total computation time also increases excessively. Nevertheless, instances of small size can be solved efficiently by these methods. Some of them will be referred in the following.

Branch and Bound (B&B), [15] and Dynamic Programming (DP), [16] are two of the classical methods that give exact optimum solutions by partially searching the feasible solution set. In B&B, a branch is a subset of solutions of the partitioned problem, and the bound is the lower bound computed that helps to find the optimum, [11,17]. As DP solves subproblems, it keeps the solutions as a future reference, i.e if it finds the same subproblem uses the result that was stored. In order to find the optimum value it starts from the bottom subproblem and goes to the main problem, which also guarantees that the subproblems are solved, [11,18,19]

The cutting-plane method is another exact approach for combinatorial optimization problems. In this method, the feasible region of the Linear Programming (LP) relaxation of the problem is renewed by adding linear inequalities at each iteration. These inequalities, referred as cuts, try to get close to the convex hull of the integer solution set. This method is inefficient, and has low convergence rate, [20,21]. However, it was combined with B&B; the Branch and Cut Algorithm is the result, [20,21].

1.4.2 Approximation Algorithms

Approximation algorithms for combinatorial optimization problems do not necessarily provide an optimal solution. However, they approximate the optimum solution to a guaranteed error value α, [22,23]. Greedy and local search algorithms are two standard approximation algorithms. In the greedy algorithm each step guarantees that the solution provided is locally optimal. The local search algorithm, on the other hand, starts with an initial solution and iteratively improves it by making changes to find a better local optimum solution, [1].

The nearest addition algorithm is an example of the greedy approaches. It starts with

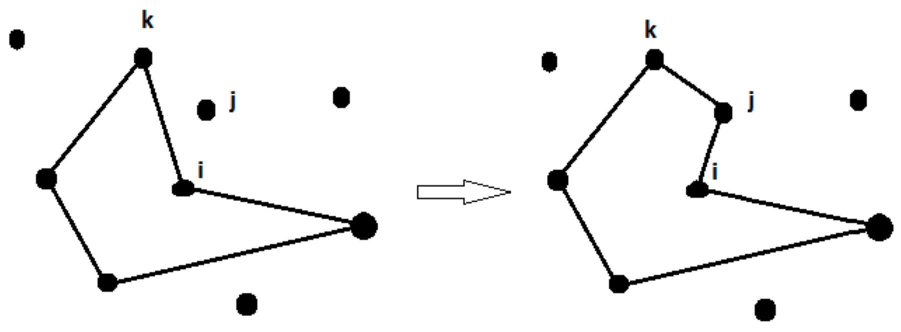

Figure 1.1: *Illustration of a nearest addition algorithm for TSP, [1].*

connecting two points with minimum cost. Then at each step, a new point which gives the shortest addition to the current set is inserted. An illustration of the method for TSP can be seen in Figure 1.1. Here, the cheapest new link is found between points i, which is already inserted and j, which is a new point. The new link is created by breaking the link between k and i. This procedure continues until all points are inserted to form a tour. This method is proved to be a 2-approximation algorithm for the metric TSP, i.e in the worst case the algorithm finds tours twice as long as the optimal tour, [1]. The Christofides Algorithm is a $\frac{3}{2}$-approximation algorithm for TSP, [24].

The deterministic rounding algorithm is another example of such algorithms. It consists in solving the LP relaxation of the integer programming model of the problem, i.e. as a linear programming model and then getting integer solutions by rounding the obtained results, [25].

Rounding a dual solution method is explained on the set covering problem in [26]. The algorithm guarantees a f-times approximation of the optimum value in $O(n^3)$ operations, where f is the maximum row sum of an $m \times n$ binary matrix, each column of which is the incidence vector of one of the sets. This is proved by considering the dual model of the LP relaxation of the problem, [26].

The primal-dual method explained in [25], is applied to the set covering problem. However, as it does not require to solve the dual of the LP relaxation problem, it is faster.

The randomised rounding algorithm solves the LP relaxation of the main problem by rounding the solution to integers with a probability. For instance, one way of doing it is using the fractional part of the solution as the probability value, [1, 22, 27]

1.4.3 Metaheuristics

Real life problems are hard to solve. Thus, exact algorithms are inefficient and costly, especially when the problem size is large. Instead of finding the optimum solution, metaheuristics generally find good approximations to it in acceptable computational times. For this reason, they are widely used in the last few decades, [28]. They are, in general, a combination of random search and local search, [23, 29, 30]. Some well-known metaheuristic methods and the ones developed recently will be reviewed in the following.

1.4.4 A Brief Review

As said earlier, combinatorial or discrete optimisation problems are often computationally demanding. They more often than not belong to the so-called NP-hard class of problems, [13], [31]. As such, it is not reasonable to expect exact solutions to perform well when solving large and practical instances. Thus, approximation methods, heuristics and metaheuristics, are almost the norm when it comes to solving them, [11, 32, 33].

1.4.4.1 Simulated Annealing

Simulated Annealing (SiA) was introduced by Kirkpatrick et al. in 1983, [34]. It is inspired by the annealing process of metals which involves slowing down the cooling of molten metal. It is important to choose the appropriate initial temperature and the cooling down rate to avoid imperfections. In the pseudo-code of SiA, the temperature is denoted by t, k_B is the Boltzmann's constant, [28], and the current candidate solution c_{old} is replaced by c_{new}, if the newly generated solution is better. But, if c_{new} is worse than the current solution, then it may be accepted with probability P given by the below formula.

$$P(t, c_{new}, c_{old}) = e^{\frac{-(value(c_{new}) - value(c_{old}))}{k_B t}}, \ t \geq 0, \tag{1.4}$$

to replace the current solution c_{old}. This is what allows SiA to escape from the local optima. The pseudo-code of SiA is given as Algorithm 1.

Algorithm 1 Simulated Annealing, [34]

```
1:  $t \leftarrow$ initially a high temperature;
2:  $c_{old} \leftarrow$ some initial guess;
3:  $c_{best} \leftarrow c_{old}$;
4:  **Repeat**
5:      $c_{new} \leftarrow$ update ($c_{old}$);
6:      **If** value($c_{new}$)<value($c_{old}$) or $rand[0,1] < P$ **Then**
7:          $c_{old} \leftarrow c_{new}$;
8:          Reduce the temperature $t$;
9:      **End If**
10:     **If** value($c_{old}$)<value($c_{best}$) **Then**
11:         $c_{best} \leftarrow c_{old}$;
12:     **End If**
13: **Until** Best solution is found, or termination criterion is reached, or $t \leq 0$;
14: **Return** Best solution $c_{best}$ as the candidate optimum solution.
```

1.4.4.2 Genetic Algorithm

The Genetic Algorithm (GA) was developed by Holland in 1975, [35]. It is based on the idea of natural selection. The algorithm works with three operators; crossover, mutation and reproduction. Basics of GA are discussed below.

Initial Population The predetermined number of individuals is randomly generated to form an initial population. The basic GA starts with this population.

Fitness Function This measure is essential for the implementation of GA. It allows to rank individual solutions in the population. It is often the objective function of the optimisation problem.

Selection of Parents Choosing suitable individuals from the population to be parents to new individuals is essential in GA. The latter are expected to be better than their parents, as a result. There are different selection methods such as the Roulette Wheel and Tournament Selection, [36].

Genetic Operators: There are three such operators.

Crossover The crossover operator selects a random point on chosen individuals representations. Then, parts of the two selected individuals are exchanged to generate two new individuals. This procedure is called a single-point crossover. Two-point and multi-point crossovers are possible. In the two-point variant, two random positions are selected and parts of those parents are exchanged, to form new offspring, [36].

Mutation A predetermined number of individuals are mutated. This is done by changing/flipping some of the entries of an individual. This operator helps exploration in GA.

Reproduction This copies good individuals into the new population as they are.

Stopping Criteria The algorithm stops when the number of generations reaches a predetermined maximum number of generations. Another commonly used stopping criterion is the maximum number of generations without improvement in the current best, [23,35]. The pseudo-code of the algorithm is as Algorithm 2.

Algorithm 2 Genetic Algorithm, [35]

```
1: $f \leftarrow$ Objective function;
2: Generate an initial random population of individuals (Parents);
3:     Repeat
4:         Select the number of individuals based on the rate;
5:         Generate new offspring using crossover (with probability $p_c$),
           mutation (with probability $p_m$), or reproduction (with $1 - p_c - p_m$);
6:         Evaluate the fitness of the children;
7:         Update the population;
8:         Update the generation counter;
9:     Until The stopping criteria are met.
10: Return Current best solution as candidate optimum.
```

1.4.4.3 Discrete Particle Swarm Optimisation

PSO was introduced by Kennedy and Eberhart in 1995, [37]. It is based on flocking birds, fish schooling and any animals moving as a group. Each particle in a swarm represents a solution. Each particle moves in a multidimensional search space for exploration and exploitation. In Discrete PSO or DPSO, and binary DPSO, in particular, [38], each particle is considered as a position in an N-dimensional space and each entry of a particle position can take value 1 or 0 which mean ″included" and ″not included", respectively. Each particle also has a velocity vector attached to it, [39]. The velocity vector is updated at each iteration using *two* pieces of information. One is the current best, *pbest*, that a particle achieved and the other is the best kept in the memory from the beginning of the algorithm, *nbest*. The equations below are used to update the velocity and position vectors v_i and X_i, respectively [39].

$$v_i(t+1) = v_i(t) + \rho_1 C_1 (pbest_i - X_i(t)) + \rho_2 C_2 (nbest_i - X_i(t)), \tag{1.5}$$

where v_i denotes the velocity of the i^{th} particle, and t denotes time. ρ_1 and ρ_2 are random values between $[0, 1]$ and C_1 and C_2 are learning factors.

$$X_i(t+1) = \begin{cases} 1 & \text{if } sig(v_i(t+1)) > r_i, \\ 0 & \text{otherwise} \end{cases} \tag{1.6}$$

where $sig(v_i(t+1))$ is the sigmoid function,

$$sig(v_i(t+1)) = \frac{1}{1 + \exp(-v_i(t+1))}. \tag{1.7}$$

The pseudo-code of DPSO is given as Algorithm 3.

Algorithm 3 Discrete Particle Swarm Optimisation, [38,39]

```
1:  Initialize with a randomly generated $N - dimensional$ swarm with $P$ particles;
2:  Repeat
3:      For all swarm $i$
4:          If $f(X_i) > f(pbest_i)$ Then $pbest_i = X_i$; End If
5:          If $f(pbest_i) > f(nbest_i)$ Then $nbest_i = pbest_i$; End If
6:      End For
7:      For all swarm $i$
8:          Update the velocity and the position vectors $v_i$ and $X_i$;
9:      End For
10: Until The stopping criterion is reached.
11: Return Best solution as candidate optimum.
```

1.4.4.4 The Combinatorial Artificial Bee Colony Algorithm

The Artificial Bee Colony (ABC) algorithm was introduced by Karaboga, [40]. The idea of the algorithm is based on the foraging behaviour of bees living in a colony. There are three types of bees represented in this process; worker bees, onlooker bees, and scout bees. Although the algorithm was first used on continuous optimisation, Karaboga and Gorkemli, [41] have implemented it for combinatorial optimization problems. They have adapted the Greedy Sub Tour Mutation (GSTM) operator, which increases the capability of GA to find the shortest length tours in TSP, as proposed by Albayrak and Allahverdi, [42]. The fitness value of solutions is estimated by the equation

$$P_i = \frac{0.9 fit_i}{fit_{best}} + 0.1. \tag{1.8}$$

The pseudo-code of the Combinatorial ABC or CABC is given as Algorithm 4.

Algorithm 4 **CABC algorithm, [41]**

```
1: Initialize the parameters (colony size (cs), maximum number of iterations (maxit));
2: Initialize the positions of the food sources $X_i$ , $i = 1, 2, \dots, cs$;
3: Evaluate the fitness of the population of solutions;
4: Keep the best solution;
5: $c = 0$;
6:     **Repeat**
7:         Each worker bee produces a new solution $v_i$ in the neighborhood of $X_i$
           and evaluates it. Apply greedy selection to choose between $v_i$ and $X_i$;
8:         Each onlooker bee produces a new solution $v_i$ from $X_i$ which is
           selected depending on its fitness value, $P_i$ and evaluates it. Apply
           greedy selection to choose between $v_i$ and $X_i$;
9:         Keep the best solution so far;
10:        Food sources that are not good are abandoned and replaced by new ones
           discovered by worker bees that have become scouts;
11:        $c = c + 1$;
12:    **Until** $c$=maxit
13: **Return** Best solution as candidate optimum solution.
```

The Quick Artificial Bee Colony Algorithm: The Quick Artificial Bee Colony Algorithm (qABC) was proposed by Karaboga et al., [43]. They have observed that in real life, worker bees and the onlooker bees are not using the same way to select food sources in the search space. Therefore, they have updated the way an onlooker bee chooses the food source. This modified version was implemented to solve TSP and gave better results than CABC, [43].

1.4.4.5 The Discrete Firefly Algorithm

The Firefly (FF) algorithm is inspired by the flashing of fireflies trying to attract mates, [44]. The main assumptions of the algorithm are as follows. All fireflies are attracted towards each other regardless of their sex. Attraction is proportional to the brightness of fireflies; the brighter a firefly is, the more attractive it will be for other members of the population. The less bright member will move towards the brighter ones. The brightness of fireflies represents the objective function. The discrete version of this algorithm was proposed by Jati et al., [45] and implemented to solve TSP. Light intensity is defined as an inverse proportion of the total tour length for TSP. The distance between two fireflies was defined as the number of different edges between them. The attractiveness of any member of the population is denoted by β and the distance between any two fireflies is denoted by r. The relationship between β and r is given as follows,

$$\beta = \beta_0 \cdot e^{-\gamma r^2}, \tag{1.9}$$

where β_0 is the value of attractiveness at $r = 0$ and γ represents the variation of the attractiveness, [44]. Equation (1.9) guarantees that the attractiveness reduces as the distance between fireflies increases. The movement procedure guarantees that the distance between the fireflies decreases when one moves towards the brighter one, [45]. The pseudo-code of the Discrete FF is given as Algorithm 5.

1.4.4.6 The Discrete Cuckoo Search Algorithm

The Cuckoo Search (CS) algorithm is another Nature-inspired metaheuristic. It simulates the phenomenon of brood parasitism in cuckoo species, [46]. It has three basic rules, [46]:

Algorithm 5 The Discrete Firefly Algorithm, [45]

```
1: Define population size, $\gamma \leftarrow$ light absorbtion coefficient, updating index $m$ and the
   objective function;
2: Create a random population of fireflies and compute their function values;
3: Repeat
4:       Find the most attractive firefly;
5:       Attract fireflies and generate new solutions by using edge-movement scheme or
         a random move (which is applicable);
6:       Update attractiveness of fireflies according to $e^{-\gamma r^2}$;
7:       Evaluate the new solutions;
8:       Sort the fireflies and select the global best;
9: Until The stopping criteria is not reached.
10: Return Best solution is returned as the candidate optimum.
```

1. Each cuckoo lays one egg at a time, and dumps its egg in a randomly chosen nest;

2. The best nest is selected based on the high quality of its eggs, and is carried forward to the next generation;

3. There are fixed numbers of host nests. An egg laid by a cuckoo is discovered by the host bird with a probability $P_a \in (0, 1)$. The host bird can either eliminate the discovered egg or abandon the whole nest.

The Discrete Cuckoo Search Algorithm (DCS) was introduced by Jati et al., [47]. They have proposed two new schemes: a discrete step size and the cuckoo's updating scheme. The discrete step size uses the number of different arcs to define the distance between two cuckoos. The updating scheme uses the discrete step size as the updating scheme of the best cuckoo and also uses a local random walk by adding a randomly generated value between $[0, s/2]$ for exploitation, where s denotes the problem size, i.e the number of cities for TSP. In the basic CS, a random step length is drawn from the Lévy distribution, [47]. The pseudo-code of DCS is given as Algorithm 6.

Algorithm 6 The Discrete Cuckoo Search (DCS) Algorithm, [47]

```
1: $f \leftarrow$ Objective function;
2: Initialise a random population of cuckoos;
3: Define probability coefficient $P_a$ and number of evolution $m$;
4: For each cuckoo, generate an initial solution;
5: **Repeat**
6:         Select a random cuckoo;
7:         Generate new solution for $m$ times;
8:         Evaluate the new solution;
9:         Get the best cuckoo;
10:        Generate new solution $m$ times with the best cuckoo;
11:            **If** ($rand < P_a$) **Then**
12:                Get the worst cuckoo;
13:                Generate new solution for $m$ times;
14:                Remove the worst cuckoo from the population.
15:            **End if**
16:        Store nests containing quality solutions;
17:        Select the $n$ best cuckoos;
18: **Until** Stopping criterion is met.
19: **Return** Best solution as candidate for optimum solution.
```

1.4.4.7 Discrete Bat Algorithm

The Bat Algorithm is a population-based metaheuristic first introduced by Xin-She Yang in [48]. It is based on echolocation used by microbats to find their way, hunt and avoid obstacles. Bats fly randomly with velocity v_i at position x_i with a fixed frequency f_{min}, varying wavelength λ and loudness A_0 to search for prey. They can automatically adjust the wavelength (or frequency) of their emitted pulses and adjust the rate of pulse emission $r \in [0, 1]$, depending on the proximity of their target. Although the loudness can vary in many ways, it is assumed that the loudness varies from a large (positive) to a minimum constant value, from A_0 to A_{min}, [48].

The discrete bat algorithm has been proposed in [49] and used to solve symmetric and asymmetric TSP. The main parameters of the Bat Algorithm have been modified to adapt it to solve discrete optimisation problems. The frequency parameter f was not used in the implementation to simplify the implementation. The velocity parameter, v_i of a bat at time step t was modified as a random number between 1, and the difference between this bat and the best bat of the flock. The difference is found by Hamming Distance. This parameter determines how many times 2-opt and 3-opt rules are going to be executed to find the new solution. These methods look for an improvement in the solution by removing 2 edges and adding 2 new edges for the former one and similarly, removing 3 edges and adding 3 new ones for the latter one. The best result is taken as the new solution after the corresponding rule is executed v_i times on the current bat. The variation of the loudness, is between A_0 and A_{min} and the rate of pulses emission r, where $r \in [0, 1]$, remains the same as in [48]. If $rand > r_i$ one solution is selected among the best ones, and a local solution is generated around this one. To generate this local solution, the best neighbor of the chosen bat is selected using also the 2-opt and 3-opt rules. If $A_i < rand$, the new solution is accepted, then r_i is increased and A_i is decreased as the bat is getting closer to the prey. The pseudo-code of the proposed algorithm is given as Algorithm 7.

1.4.4.8 Other Metaheuristic Methods

There are many other metaheuristic methods for combinatorial optimisation, [23, 29, 30]. Ouaarab et al., [50] have introduced the Improved Cuckoo Search (ICS) and also two types of Discrete Cuckoo Search (DCS) algorithms, one of which is based on the original CS and the other on the improved CS, to solve TSP. The performance of the improved DCS was tested on 41 TSP problem instances with the number of cities ranging from 51 to 1379. The experimental results show that the proposed method is superior to Genetic Simulated

Algorithm 7 The Discrete Bat Algorithm (DBA), [49]

```
$F \leftarrow$ Objective function, $n \leftarrow$ number of nodes of the instance;
$r_i \leftarrow$ rate of pulses, $A_i \leftarrow$ loudness, $v_i \leftarrow$ velocity;
Initialise $r_i$, $v_i$ and $A_i$ for all bat $i$, $i = 1, 2, \ldots, n$;
**Repeat**
    **For** Each bat $i$ **do**
        Generate a new solution;
        **If** ($v_i^t < n/2$) **Then**
            $x_i \leftarrow 2 - opt(x_i^{t-1}, v_i^t)$ ;
        **Else**
            $x_i \leftarrow 3 - opt(x_i^{t-1}, v_i^t)$ ;
        **End if**
        **If** (rand $> r_i$) **Then**
            Select one solution amongst the best ones;
            Generate a new bat using the 2-opt or the 3-opt rule, selecting the best
            neighbour around the chosen bat;
        **End if**
        **If** (rand$< A_i$ and $f(\mathbf{x}_i) > f(\mathbf{x}^*)$) **Then**
            Select new solution;
            Increase pulse rate $r_i$ and reduce loudness $A_i$;
        **End if**
        Rank all solutions and select the best solution.
    **End for**
**Until** The stopping criterion is met.
**Return** Best solution as candidate for optimum solution.
```

Annealing Ant Colony System with Particle Swarm Optimization Technique (GSiA-ACS-PSOT) and DPSO [50]. Saenphon et al., [51] have developed the Fast Opposite Gradient Search method and combined it with ACO. The proposed method has been compared to TS, GA, PSO, ACO, PS-ACO and GA-PS-ACO on TSP instances. Their algorithm has achieved better results. Mahi et al. have developed an algorithm based on PSO, ACO and 3-opt algorithms for TSP, [52]. The new hybrid method was tested against some well-known algorithms such as ACO+2-opt, SiA-ACO-PSO and ACO with ABC. The experimental results show that it outperforms the other algorithms in terms of solution quality.

Whale swarm algorithm is recently proposed and mimics the social behavior of humpback whales [53]. The fruit fly optimisation algorithm was proposed by Pan, [54]. It follows the food finding procedure of fruit flies. This algorithm has been extended to solve the multidimensional knapsack problem in [55]. Bitam et al. proposed a bees life algorithm for cloud computing services selection, [56]. Mehrabian et al., [57] proposed yet another algorithm, the invasive weed optimisation algorithm inspired by the invasive spread and growth property of weeds. The discrete implementation of this algorithm to solve TSP can be found in [58].

1.5 Summary

We have introduced the subject matter of this thesis and reviewed important material on which we intend to build up. This includes key Nature-inspired algorithms such as the Genetic Algorithm, Simulated Annealing, the Firefly, Particle Swarm Optimisation to name a few. It is clear that there is a profusion of new methods. Some active researchers are not satisfied with this because of the lack of analysis and proper investigation of the new additions. However, it is also a mark of necessity and activity in the field which must be praised and encouraged. More importantly, it is a sign that we are facing harder and more challenging problems demanding newer and more effective approaches. In the following we will extend and investigate such an algorithm.

CHAPTER 2

THE PLANT PROPAGATION ALGORITHM: VARIANTS AND IMPLEMENTATIONS

2.1 Introduction

After describing the state-of-art Nature-Inspired algorithms, here we focus on a specific algorithm namely the Plant Propagation Algorithm or PPA [9]. We start by introducing the original and basic algorithm otherwise known as the Strawberry Algorithm. We then look at implementations of it to handle different continuous global optimisation problems, and in particular constrained ones. We proceed to describe a PPA algorithm which emulates propagation based entirely on seeds. This version uses the feeding station model of queueing theory. Since the particular plant of interest in PPA is the strawberry plant, and this plant uses both runners and seeds to propagate, it is reasonable to consider a hybrid which uses runners and seeds to search for the global optimum. Such a hybrid has been introduced and successfully tested. We will also explain it here. Finally we will extend PPA to discrete optimisation.

2.2 PPA for Continuous Optimisation Problems

2.2.1 The Basic Plant Propagation Algorithm

The Plant Propagation Algorithm (PPA) introduced by Salhi and Fraga, [9], emulates the strategy that plants deploy to survive by colonising new places which have good conditions for growth. Plants, like animals, survive by overcoming adverse conditions using often basic but effective strategies. The strawberry plant, for instance, has a survival and expansion strategy which is to send short runners to exploit the local area if the latter has good conditions, and to send long runners to explore new and more remote areas, i.e. to run away from a not so favourable current area. The mechanism of the basic PPA is explained in this section, [9].

Algorithm 8 Pseudo-code of PPA, [9]

```
1:  Generate a population $P = X_i,\ i = 1, \dots,\ NPop$ of plants;
2:  $g \leftarrow 1$
3:  for $g = 1 : g_{\max}$ do
4:      Compute $N_i = f(X_i), \forall\ X_i \in P$;
5:      Sort $P$ in ascending order of fitness values $N$ (for minimization);
6:      Create new population $\Phi$;
7:      for each $X_i,\ i = 1, \dots,\ NPop$ do
8:          $r_i \leftarrow$ set of runners where both the size of the set and the distance for each runner
            (individually) are proportional to the fitness value $N_i$;
9:          $\Phi \leftarrow \Phi \cup\ r_i$ (append to population; death occurs by omission);
10:     end for
11:     $P \leftarrow \Phi$ (new population);
12: end for
13: return Best solution as the candidate for optimum.
```

The algorithm starts with a population of plants each of which represents a solution in the search space. X_i denotes the solution represented by plant i in an n-dimensional space. $X_i \in \mathbb{R}^n$, i.e. $X_i = [x_{ij}]$, for $j = 1, \dots, n$ and $x_{ij} \in \mathbb{R}$. $NPop$ is the population size.

This iterative process stops when g, the counter of generations, reaches its given maximum value $g_{\max}$.

Individuals/plants/solutions are evaluated and then ranked (sorted in ascending or descending order) according to their objective (fitness) values and whether the problem is a min or a max problem. The number of runners of a plant is proportional to its objective value and conversely, the length of each runner is inversely proportional to the objective value, [9]. For each X_i, $N_i \in (0,1)$ denotes the normalized objective function value. The number of runners for each plant to generate is

$$n_r^i = \lceil (n_{\max} \, N_i \, \beta_i) \rceil, \tag{2.1}$$

where n_r^i shows the number of runners and $\beta_i \in (0,1)$ is a randomly picked number. For each plant, the minimum number of runners is set to 1. The distance value found for each runner is denoted by dx_j^i. It is:

$$dx_j^i = 2(1 - N_i)(r - 0.5), \ for \ j = 1, \dots, n, \tag{2.2}$$

where $r \in [0,1]$ is a randomly chosen value. Calculated distance values are used to position the new plants as follows:

$$y_{ij} = x_{ij} + (b_j - a_j) \, dx_j^i, \ for \ j = 1, \dots, n, \tag{2.3}$$

where y_{ij} shows the position of the new plant, and $[a_j, b_j]$, $j = 1, \dots, n$ are the bounds of the search space.

The new population that is created by appending the new solutions to the current population is sorted. In order to keep the number of population constant, the solutions that have lower objective values are dropped. Figure 2.1 depicts a strawberry plant with its runners and plantlets.

After the introduction and successful implementation of the basic PPA , further studies

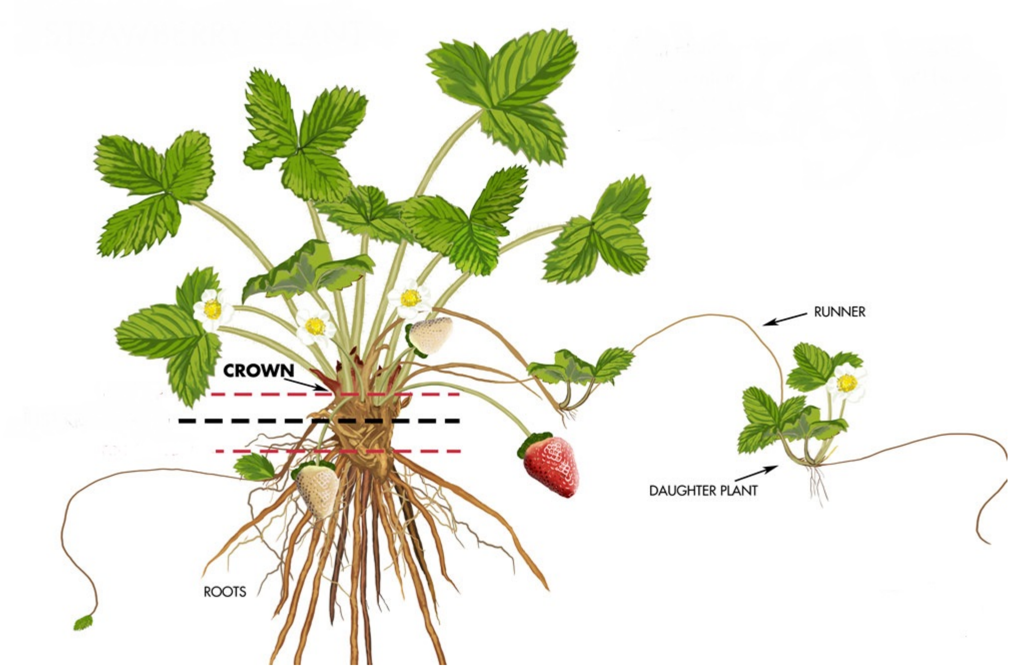

Figure 2.1: *Mother plant, runners and plantlets, [2]*

have been carried out to solve continuous optimisation problems, [59–62]. Results show that PPA works well both on constrained and unconstrained continuous optimisation problems.

2.2.2 The Modified and Improved PPA

In [62], Sulaiman et al. have introduced MPPA, a modified version of PPA. They have proposed an alternative implementation of the propagation phase. The algorithm has been stated as a robust, easy to implement for non-linear, non-convex high dimensional continuous optimisation problems. The proposed method is explained as follows. First, the population is generated randomly using the Equation (2.4):

$$x_{ij} = a_j + (b_j - a_j)\alpha_j,\ j = 1, ..., n, \tag{2.4}$$

where $\alpha_j \in (0,1)$ is a randomly generated real number for each j, and $[a_j, b_j],\ j = 1, \ldots, n$ are the bounds of the search space. After the population is set, MPPA proceeds to generate for every member in the population n_r runners. These runners lead to new solutions as per Equations (2.5-2.7) below:

$$y_{ij} = x_{ij} + \beta_j x_{ij},\ j = 1, \ldots, n, \tag{2.5}$$

where $\beta_j \in [-1,1]$ is a randomly generated number for each j. The term $\beta_j x_{ij}$ is the length with respect to the j^{th} coordinate of the runner, and $y_{ij} \in [a_j, b_j]$. If the bounds of the search domain are violated, the point is adjusted to be within the domain. The generated individual Y is evaluated according to the objective function and is stored in Φ. Equation (2.5) helps in exploring the neighbourhood of x_{ij}. As the search becomes refined, that is the algorithm is in exploitation mode, the coordinates produced by Equation (2.5) become smaller and smaller. In MPPA, if this newly created solution by Equation (2.5) is not improving the objective function, then another individual is created as a runner based on Equation (2.6).

$$y_{ij} = x_{ij} + \beta_j b_j,\ j = 1, \ldots, n, \tag{2.6}$$

where b_j is the j^{th} upper bound and here again $y_{ij} \in [a_j, b_j]$. This can be considered as a solution at the end of a long runner. Again, if the generated individual does not improve the objective value, another runner is created by Equation (2.7),

$$y_{ij} = x_{ij} + \beta_j a_j,\ j = 1, \ldots, n, \tag{2.7}$$

where a_j is the j^{th} lower bound and $y_{ij} \in [a_j, b_j]$. Similar to Equation (2.6), this can be also considered as a solution at the end of a long runner.

Equations (2.5-2.7) are implemented in MPPA in turn if any of the search equations fails to improve the current solution. MPPA maintains a better balance between exploration and exploitation of the search space. Note that the above equations may lead to infeasibility. In these situations, the offending entry is set by default to the boundary, lower or upper as per the concerned equation. To keep the size of the population constant, the plants with ranks bigger than *NPop* (population size) after sorting, are eliminated [62].

2.2.3 PPA for Constrained Optimisation

In [59], Sulaiman et al. show the superiority of PPA to some well-known algorithms. In the study, well-known, hard constrained engineering problems are solved. In order to start the algorithm with a good initial population, it has been run r times with a randomly generated population. If $r = NPop$, where *NPop* denotes the size of population, there are enough individuals to start the algorithm. In the case of mixed integer problems, the j^{th} entries of the solution vectors are fixed when they are showing a trend to converge to some values; here $j = 1, 2, \cdots, n$. This trend is monitored by calculating the number of these entries which have not changed after a number of iterations.

Let P be a general matrix containing the randomly generated population of a given run. Its rows correspond to individuals. Equation (2.4) is used to generate a random population for each of the initial *NPop* runs.

Algorithm 9 Modified Plant Propagation Algorithm (MPPA), [62]

```
1: Create a random population of plants $pop = \{X_i \mid i = 1, 2, ..., NPop\}$,
   $f \leftarrow$ Objective function , $\Phi \leftarrow$ Temporary population of runners;
2: Evaluate the population;
3: Assume a fixed number of runners as $n_r = 3$;
4: while the stopping criteria is not satisfied do
5:     Create $\Phi$;
6:     for all plants $i = 1$ to $NPop$ do
7:         $\Phi_i = X_i$;
8:         for $k = 1$ to $n_r$ do
9:             Generate a new solution $Y$ according to Equation (2.5);
10:            Evaluate it and store it in $\Phi$;
11:            Calculate diff=$\mid f(Y) \mid - \mid f(X_i) \mid$;
12:            if diff $\geq 0$ then
13:                Generate a new solution Y according to Equation (2.6);
14:                Evaluate it and store it in $\Phi$;
15:                Compute diff=$\mid f(Y) \mid - \mid f(X_i) \mid$;
16:                if diff $\geq 0$ then
17:                    Generate a new runner using Equation (2.7);
18:                    Evaluate it and store it in $\Phi$;
19:                end if
20:            end if
21:        end for
22:    end for
23:    Append $\Phi$ to current population $pop$;
24:    Sort the population in ascending order of the objective values and omit the solutions
       with rank > $NPop$;
25:    Update current best;
26: end while
27: return Best solution as candidate for optimum.
```

In each generation, generated individuals are kept in Φ. Each new plant is generated following three rules. The first two rules are valid if $r \leq NPop$, the last one is used otherwise. For the first two rules, there is a fixed modification parameter P_m, which after a number of carried out experiments, is set to $P_m = 0.8$ as the algorithm has a better convergence rate with this value. The first two rules are implemented if the population is initialized randomly. Rule 1 uses Equation (2.8) below to update the population.

$$x^*_{ij} = x_{ij}(1 + \beta_j), j = 1, ..., n, \tag{2.8}$$

Algorithm 10 Pseudo-code of PPA for constrained optimisation problems, [59]

```
1: Set $g_{\max} \leftarrow$ Max. number of generations $NPop \leftarrow$ pop. size $r \leftarrow$ no. of trial runs;
2: if $r \leq NPop$ then
3:     Generate a random population $P = X_i,\ i = 1, \dots, NPop$ using Equation (2.4),
       gather best solutions.
4: end if
5: while $r > NPop$ do
6:     Use $P_g$ that is created by collecting best individuals from each run, then
       calculate $IN_j$ value for each individual plant;
7: end while
8: Evaluate the population;
9: Set $n_r = 3$, $n_{\text{gen}} = 1$, where $n_r$ indicates the number of runners;
10: while do$(n_{\text{gen}} < g_{\max}) || (n_{\text{eval}} < max_{\text{eval}})$
11:     Create new population $\Phi$;
12:     for $i = 1, \dots, NPop$ do
13:         for $k = 1, \dots, n_r$ do
14:             if $r \leq NPop$ then
15:                 if $rand \leq P_m$ then
16:                     Generate a new solution $X^{*(1)}$ using Rule 1;
17:                     Evaluate it and append in $\Phi$;
18:                 end if
19:                 if $rand \leq P_m$ then
20:                     Generate a new solution $X^{*(2)}$ using Rule 2;
21:                     Evaluate it and append to $\Phi$;
22:                 end if
23:             else
24:                 for $j = 1, \dots, n$ do
25:                     if $(IN_j \leq 4) || (rand \leq P_m)$ then
26:                         Update the $j^{th}$ entry of $X_i,\ i = 1, \dots, NPop$ using Rule 3;
27:                     end if
28:                     Evaluate it and append to $\Phi$;
29:                 end for
30:             end if
31:         end for
32:     end for
33:     $P \leftarrow \Phi$ (new population);
34:     Sort the population in ascending order;
35:     Update current best;
36: end while
37: return Best solution as candidate for optimum.
```

where $\beta_j \in [-1, 1]$ and $x^*_{ij} \in [a_j, b_j]$.

The generated individual $X_i^{*(1)}$ is evaluated according to the objective function and is stored in Φ. In rule 2 another individual is created using the same modification parameter $P_m = 0.8$ with Equation (2.9),

$$x^*_{ij} = x_{ij} + (x_{lj} - x_{kj})\beta_j, j = 1, ..., n, \tag{2.9}$$

where $\beta_j \in [-1, 1]$, $x^*_{ij} \in [a_j, b_j]$. The indices l, k are mutually exclusive and are different from i, [63]. The generated individual $X_i^{*(2)}$ is evaluated according to the objective function and is stored in Φ. The first two rules are applicable for $r \leq NPop$. r is a counter of the number of runs. For $r > NPop$ the algorithm also tries to recognise entries which are settling to their final values through a counter IN which is an n-dimensional vector containing counts for each column j. If the j^{th} entries in the current population matrix has a low IN_j value then it is modified by implementing Equation (2.10) as Rule 3. Otherwise it is left as is. The value of IN_j is set to 4 after carrying out a number of experiments over a number of problems. Equation (2.10) is as below.

$$x^*_{ij} = x_{ij} + (x_{ij} - x_{kj})\beta_j, j = 1, ..., n, \tag{2.10}$$

where $\beta_j \in [-1, 1]$, $x^*_{ij} \in [a_j, b_j]$, and k is different from i, [63].

2.3 The Seed-Based PPA: The Feeding Station Model

PPA is particularly concerned with the strawberry plant. Strawberry plants propagate via seeds as well as runners. In [60], Sulaiman et al. studied a Seed-based PPA or SbPPA which is a new variant of PPA based on propagation by seeds. The dispersion of seeds is provided by animals and birds in particular. Since plants produce fruit for the purpose of dispersing their seeds, there is a timing element and a set up which must attract the dispersing agents. This obviously calls for the feeding station analogy. Indeed, one can draw the reasonable parallel with restaurants for instance which are set up to attract customers in need of feeding, but will then contribute to the business owner. A queueing model is used to imitate the system.

A queuing system has two basic components:
(1) the rate at which agents arrive at the (feeding station) strawberry plants,
(2) the rate at which the agents eat fruit and leave the plants to disperse the seeds. The agents arrive at plants in a random process. The conditions under which the system operates are listed below, [64]:

- **Orderliness:** Plants are visited at most by one agent (bird/animal in this case) at a time.
- **Stationarity:** The probability of arrivals of agents to the plants remains the same for a particular period of time.
- **Independence:** Arrivals are independent from each other.

Based on these assumptions, it is concluded that the probability of arrival of k agents during a cycle c of fruit production by strawberry plants can be denoted by random variable X', [64]. The probability distribution of a Poisson random variable X' can be expressed mathematically as

$$P(X' = k) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}, \tag{2.11}$$

where λ denotes the mean arrival rate of agents per unit time, and t the length of the time interval. The time taken by agents in successfully eating fruit and leaving to disperse its seeds, in other words the service time for agents, are expressed by a random variable which follows the exponential probability distribution, [65]. This can be expressed as follows,

$$S(\rho) = \mu e^{-\mu\rho}, \tag{2.12}$$

where μ is the average number of agents that can feed at time ρ.

Assume that the arrival rate of agents is less than the fruits available on all plants per unit of time, therefore $\lambda < \mu$ and the system is in steady state. Let D denote the average number of agents in the strawberry field (some already eating and the rest waiting in the queue to feed), and D_q the average number of agents waiting to get the chance to eat, i.e waiting in the queue. If the average number of agents eating fruits is denoted by $\frac{\lambda}{\mu}$, then by Little's formula, [66],

$$D = D_q + \frac{\lambda}{\mu}. \tag{2.13}$$

Since the plant strives to maximise dispersion, this is equivalent to having a large D_q in Equation (2.13). Therefore, from this equation, the following problem should be solved,

$$\textit{Maximize } D_q = D - \frac{\lambda}{\mu}, \tag{2.14}$$

subject to

$$\begin{aligned} &g_1(\lambda, \mu) = \lambda < \mu + 1, \\ &\lambda > 0,\ \mu > 0. \end{aligned} \tag{2.15}$$

Frugivores may travel far away from the plants and hence will disperse the seeds far and wide. This feeding behaviour typically follows a Lévy distribution, [60, 67].

Lévy distribution

The Lévy distribution is a probability density function of a random variable. Here the random variable represents the directions of flights of arbitrary birds. This function ranges over real numbers in the domain represented by the problem search space, [60]. The flight lengths of the agents served by the plants follow a heavy tailed power law distribution, [68], represented by,

$$L(s) \sim |s|^{-1-\beta}, \tag{2.16}$$

where s is a step size drawn from the Lévy distribution and $L(s)$ denotes the Lévy distribution with index $\beta \in (0, 2)$. Lévy flights are unique arbitrary excursions whose step lengths are drawn from (2.16). An alternative form of Lévy distribution is, [68],

$$L(s,\gamma,\mu) = \begin{cases} \sqrt{\frac{\gamma}{2\pi}}\left(\frac{1}{(s-\mu)}\right)^{\frac{3}{2}} \exp\left[-\frac{\gamma}{2(s-\mu)}\right], & 0 < \mu < s < \infty \\ 0 & \text{Otherwise.} \end{cases} \tag{2.17}$$

This implies that

$$\lim_{s\to\infty} L(s,\gamma,\mu) \approx \sqrt{\frac{\gamma}{2\pi}}\left(\frac{1}{s}\right)^{\frac{3}{2}}. \tag{2.18}$$

The steps $L(s)$ are generated by Mantegna's algorithm, [68]. This algorithm ensures that the behaviour of Lévy flights is symmetric and stable. It is assumed that the arrival of different agents (birds and animals) to the plants to feed follows a Poisson distribution, [64].

As already stated, balancing the exploration and exploitation is essential in this implementation. To this end, in [60] a threshold value of the Poisson probability is chosen that dictates how much exploration and exploitation is done during the search. The probability $Poiss(\lambda) < 0.05$ means that exploitation is covered. In this case, Equation (2.19) is used, which is helping the algorithm to search locally,

$$x^*_{ij} = \begin{cases} x_{ij} + \xi_j(x_{ij} - x_{lj}) & \text{if } PR \le 0.8;\ j = 1,2,\cdots,n; \\ & \quad i,l = 1,2,\cdots,NPop;\ i \ne l \\ x_{ij} & \textit{Otherwise}, \end{cases} \tag{2.19}$$

where PR denotes the rate of dispersion of the seeds locally, around the strawberry plant;

x^*_{ij} and $x_{ij} \in [a_j, b_j]$ are the j^{th} coordinates of the seeds X^*_i and X_i respectively; where a_j and b_j are the j^{th} lower and upper bounds defining the search space of the problem, and $\xi_j \in [-1, 1]$, [63]. The indices l and i are mutually exclusive. If, on the other hand, $Poiss(\lambda) \geq 0.05$ then global dispersion of seeds becomes more prominent. This is implemented by using the following equation,

$$x^*_{ij} = \begin{cases} x_{ij} + \mathrm{L_i}(x_{ij} - \theta_j) & \text{if } PR \leq 0.8,\ \theta_j \in [a_j\ b_j] \\ & i = 1, 2, \cdots, NPop;\ j = 1, 2, \cdots, n \\ & \\ x_{ij} & \textit{Otherwise}, \end{cases} \tag{2.20}$$

where $\mathrm{L_i}$ is a step drawn from the Lévy distribution, [68], θ_j is a random coordinate within the search space. Equations (2.19) and (2.20) perturb the current solution.

As mentioned in Algorithm 10, first collect the best solutions are collected from the first *NPop* trial runs to form a population of potentially good solutions denoted by pop_{best}. The statistics values best, worst, mean and standard deviation are calculated based on pop_{best}.

The seed based propagation process of the strawberry plant can be represented in the following steps:

1. The dispersal of seeds in the neighbourhood of the strawberry plant is carried out either by fruit fallen from strawberry plants after they become ripe or by agents. The step lengths for this phase are calculated using Equation (2.19).

2. Seeds are spread globally through agents. The step lengths for these travelling agents are drawn from the Lévy distribution, [68].

3. The probabilities, $Poiss(\lambda)$, that a certain number k of agents will arrive to strawberry plants to eat fruit and disperse it, is used as a balancing factor between exploration and exploitation.

For implementation purposes, it is assumed that each strawberry plant produces one fruit, and each fruit is assumed to have one seed; solution X_i means the current position of the i^{th} seed to be dispersed. The number of seeds in the population is denoted by *NPop*. Initially a random population of *NPop* seeds is generated using Equation (2.4).

Algorithm 11 Seed-based Plant Propagation Algorithm (SbPPA), [60]

```
1:  $NPop \leftarrow$ Population size, $r \leftarrow$ Counter of trial runs, $MaxExp \leftarrow 30$;
2:  for r=1 : MaxExp do
3:      if $r \leq NPop$ then
4:          Create a random population of seeds $pop = \{X_i \mid i = 1, 2, ..., NPop\}$,
            using Equation (2.4) and collect the best solutions from each trial run, in $pop_{best}$;
5:          Evaluate the population $pop$.
6:      end if
7:      while $r > NPop$ do
8:          Use updated population $pop_{best}$;
9:      end while
10:     while ( the stopping criteria is not satisfied) do
11:         for $i = 1$ to $NPop$ do
12:             if $Poiss(\lambda) \geq 0.05$ then            ▷ (Global or local seed dispersion)
13:                 for $j = 1$ to $n$ do                  ▷ ($n$ is number of dimensions)
14:                     if rand $\leq PR$ then
15:                         Update the current entry according to Equation (2.20);
16:                     end if
17:                 end for
18:             else
19:                 for $j = 1$ to $n$ do
20:                     if rand $\leq PR$ then
21:                         Update the current entry according to Equation (2.19);
22:                     end if
23:                 end for
24:             end if
25:         end for
26:         Update current best;
27:     end while
28:     Return: Updated population and global best solution.
29: end for
```

2.4 The Hybrid PPA-SbPPA for Continuous Optimisation

In [61], Sulaiman et al. have proposed and investigated a hybrid metaheuristic algorithm referred to as H-PPA-SbPPA , which captures the overall propagation process of the strawberry plant. They combined PPA and SbPPA to handle constrained and unconstrained optimisation problems. The study of PPA showed that exploration is not prominent; this often results in premature convergence. The hybridised algorithm is expected to address the weak exploration characteristic of PPA with the exploration capability of SbPPA. Exploitation is performed through sending many short or few long runners in the neighbourhood of the parent, [9]. Furthermore, there is no gauge defined in PPA which can decide how much exploration and exploitation should be carried out to cover the search space well. They used the Poisson probability as in [60] to balance the two characteristics. In the original implementation of PPA, the runners are generated according to a normalised fitness value of the objective function. These runners are either long or short depending on the fitness value of positions of their parent plants.

At the initialization phase, H-PPA-SbPPA generates random populations for the first *NPop* (population size) trial runs. To preserve feasibility, it gathers all the global best solutions of the first *NPop* experiments and forms a new population of size *NPop*. The random spots where to grow *NPop* strawberry plants are generated as in [59]. This new algorithm has all its components already explained in the previous sections of this chapter.

Quality of the Current Spot

The quality of a spot is calculated according to Equation (2.21), [9],

$$f(X) = \frac{f_{max} - f(X)}{f_{max} - f_{min}}, \tag{2.21}$$

where f_{min} and f_{max} are respectively the minimum and maximum objective function values in the current population. If $f_{max} - f_{min} < \epsilon$, where ϵ is a small positive real number, then all positions in the current population are given a fitness of 0.5, [9].

Algorithm 12 **Hybridised PPA-SbPPA (H-PPA-SbPPA), [61]**

```
1:  $NPop \leftarrow$ Population size, $r \leftarrow$ Counter of trial runs, $MaxExp \leftarrow 30$;
2:  for r=1 : $MaxExp$ do
3:      if $r \leq NPop$ then
4:          Create a random population of plants $pop = \{X_i \mid i = 1, 2, ..., NPop\}$,
            using Equation (2.4) and collect the best solutions from each trial run, in $pop_{best}$;
5:          Evaluate the population $pop$.
6:      end if
7:      while $r > NPop$ do
8:          Use updated population $pop_{best}$;
9:      end while
10:     while (the stopping criteria is not satisfied) do
11:         for $i = 1$ to $NPop$ do
12:             if $Poiss(\lambda) \geq 0.05$ then                ▷ (Global or local search)
13:                 for $j = 1$ to $n$ do                     ▷ ($n$ is number of dimensions)
14:                     Update the current entry according to Equation (2.19);
15:                 end for
16:             else
17:                 for $j = 1$ to $n$ do
18:                     Update the current entry according to Equation (2.3);
19:                 end for
20:             end if
21:         end for
22:     end while
23:     Record the best of this experiment.
24: end for
25: return Best solution as candidate for optimum.
```

In each iteration, a plant produces new child plants through seeds or runners. The probability $Poiss(\lambda) \geq 0.05$ means that exploration is covered. In this case, Equation

(2.19) is used, which is helping the algorithm to search globally. If the quality of new spot X_i^* is better than the quality of its parent location X_i, then the parent location is replaced, otherwise the new solution is ignored. If, on the other hand, $Poiss(\lambda) < 0.05$ then propagation through runners becomes prominent. Depending on their sports, each plant either sends many short runners or a few long runners in their neighborhood. This is done by using Equations (2.2) and (2.3), [9]. To keep the size of the population constant, the plants with ranks bigger than *NPop* after sorting, are eliminated.

2.5 Extension of PPA for Discrete Optimisation

After discussing various successful implementations of PPA on constrained and unconstrained continuous optimisation problems, its extension to solve discrete optimisation problems will now be discussed. The idea is again to use short runners for exploitation and long runners for exploration of the search space, [9]. The main issues with the implementation of PPA to solve discrete optimization problems are [6]:

1. Finding/Defining the equivalent of a distance between two solutions in the discrete solution space. In other words defining a metric for the search space.
2. By extension, defining the neighbourhood of a discrete solution.

2.5.1 The Representation of a Solution as a Plant

The solution representation as a plant varies depending on the discrete optimisation problem type. In the next chapters, we will explain implementations to solve the Travelling Salesman Problem (TSP), the Knapsack Problem (KP) and, Scheduling Problems, in partic-

1	4	6	9	12	20	5	7	8	11	15	3	17	16	2	10	14	18	13	19

Figure 2.2: *The permutation representation of a plant as a tour*

ular the Robust Berth Allocation Problem (RBAP) as arise in container ports. For TSP, the represenation is depicted as a permutation of cities. Each plant therefore indicates a tour.

In Figure 2.2, the entries of the array represent cities. City 6 and city 9 are successive in the depicted tour and the notation 6-9 defines the edge between them.

For the single binary KP, the represenation of a solution as a plant is basically a binary string, $X_i = [0\ 1\ 0\ 0\ 1\ 1\ 0\ 1]$ which represents a knapsack where $n = 8$ and items x_2, x_5, x_6 and x_8 are to be included in the knapsack. For the multiple knapsack problem, a matrix representation is used where each row represents a knapsack. A sample set of 3 knapsacks with 6 items in total can be shown as below, X_{ij}, where x_{ij} in the set X_{ij} denotes the j^{th} item in the i^{th} knapsack:

$$X_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}, \quad i = 1, \ldots, 3, \quad j = 1, \ldots, 6. \tag{2.22}$$

Here, items x_{11}, x_{15}, and x_{16} are in the first knapsack, items x_{22} and x_{24} are in the second bag, and items x_{33} and x_{36} are in the 3^{rd} knapsack.

The third type of representation is for schedules. Here, we will look at the schedules for the Robust Berth Allocation Problem. For each vessel, a starting time and a position is needed to configure a schedule. We represent it as an array of length $2n$, where n denotes the number of vessels. The first n numbers on the string show processing starting times and the rest shows the positions each vessel is assigned to. Table 2.1 depicts a schedule for 3 vessels. The first vessel starts being processed in time unit 5 and is placed in position unit 0.

Table 2.1: *Representation of a schedule as a plant*

5	16	23	0	2	6
T_1	T_2	T_3	P_1	P_2	P_3

2.5.2 Short and Long Runners

Determining strategies for short and long runners is at the heart of the implementation. While implementing the strategies chosen, it is essential to cover the search space extendedly.

Let *gen* be the number of generations, *pop* be the size of population, r and R be the number of short and long runners, respectively. Let short runners be sent from the top γ solutions of the population and S denote the search space that is examined. The total number of short runners is calculated as:

$$SR = gen \times r \times \frac{pop}{\gamma}, \tag{2.23}$$

The total number of long runners generated from the rest of the population is:

$$LR = gen \times R\ (pop - \frac{pop}{\gamma}), \tag{2.24}$$

Therefore, assuming that there are no repetitions, the number of all individuals ($SR + LR$) examined by the algorithm is:

$$S = gen \times pop\ (\frac{r + (\gamma - 1)R}{\gamma}) + pop. \tag{2.25}$$

The sketch in Figure 2.3 shows that the cumulative effect of short runners is a long runner or even a longer runner. Following this idea, it can be assumed that the algorithm does not get stuck into a local optimum. If there is a local optimum at the exploitation area, it is found. Unless the local optimum is in fact the global optimum, after a certain number of generations, long runners will help move from the local optimum found by its

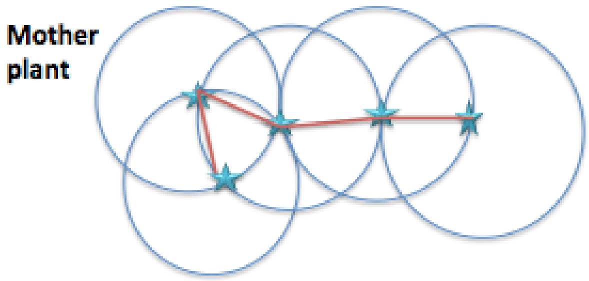

Figure 2.3: *A sketch of short runners in different generations*

exploration property.

2.5.3 Termination of the Algorithm

The termination criterion of the algorithm is two fold:

1. The maximum number of generations, *maxgen*;
2. The number of iterations without improvement in the current best solution.

Note that both approaches can be combined and used together.

2.6 Summary

In this chapter, we discussed variants and implementations of PPA, which has been successfully implemented for both constrained and unconstrained continuous optimisation problems. First, we explained the basic algorithm [9]. We then reviewed recent implementations of it. We started with MPPA which offers a new mechanism for short and long runners. Then, we discussed PPA for constrained optimisation problems. Well-known and hard engineering problems were solved in this study [59]. The novelty of this variant is that the inital population is generated running the algorithm *NPop* times, where *NPop* denotes the size of the population, and keeping the best solution of each run. Next, a study that uses a different approach for the propagation of the strawberry plant has been discussed. In this algorithm, SbPPA, instead of runners, the propagation is achieved via seeds. Seeds are transported and dispersed by animals and birds in particular. The feeding station model of queueing systems has been used to implement SbPPA. Succesful results were observed after solving well-known benchmark problems. As SbPPA was a success, then the hybrid version of SbPPA and PPA [61] was studied. This variant was aimed at improving the exploration capability of PPA by using SbPPA.

In the last section, the extension of PPA to solve discrete optimisation problems is discussed. Representation methods for discrete optimisation problems were shown for different problem types. The main issues of the implementation were addressed and the importance of choosing good strategies for implementing short and long runners was highlighted. Then, the search mechanisms of short and long runners were discussed. It was underlined that short runners have the ability to find local optima, if they happen to be in their exploitation search areas and long runners have the ability to escape from getting stuck at local optima. Finally, the stopping criteria of the extenstion of PPA for discrete optimisation was discussed. The discrete extensions of PPA to solve various combinatorial optimisation problems will be discussed in Chapter 4, Chapter 5 and Chapter 6 in detail.

CHAPTER 3

INITIAL POPULATION GENERATION IN POPULATION-BASED ALGORITHMS

3.1 Introduction

Population-based algorithms maintain a population from generation to generation and tries to improve it. Some of the well-known population-based heuristics are GA [35], ACO [69], ABC [40], Swarm intelligence [70], and Harmony Search [71].

Starting an algorithm with a population of individuals/solutions enhances its exploration capabilities, [23]. However, starting with a good initial population is crucial since it directly affects the performance of the algorithm, [23]. In order to boost the quality of the initial population we include in it a number of individuals which are not randomly generated. This chapter is concerned with ways to generate good individual cheaply. Some of the methods used to initialize populations are as follows, [23].

1. Random generation
2. Sequential diversification

3. Parallel diversification

4. Heuristic initialization

Here, we consider the situation of applying PPA to a set of well-known discrete optimisation problems. We will treat each case on its own, showing how the initial population can be generated.

3.2 Strip Algorithms to Generate Good Tours for TSP

Cheap but not necessarily robust algorithms are always needed. Also, since, often, hard problems encountered in the real world are not required to be solved exactly, approximate solutions are enough particularly when time is crucial. The TSP is one such problem. So, cheap algorithms were devised for it. Such algorithms are known as Strip Algorithms, [72–74]. They are simple and efficient heuristics for large Euclidean TSPs, [75]. Their approach is to cut the plane into vertical or horizontal strips, then group the cities within strips according to their coordinates, and find Euclidean distances by following the order of the cities. Eventually the total length of a tour is obtained, [76]. It can be implemented in various ways. Furthermore, we believe that it has a lot of scope for improvement.

3.2.1 Ideas Behind the Strip Algorithm

Without loss of generality, we consider the Euclidean or 2*D* TSP. The idea is to connect nodes which are within a reduced area of the 2*D* problem. This is not very different from approaches which build tours from minimum spanning trees and other greedy methods, [77]. However, it allows for ”less obvious" links to be taken into consideration and discourages cris-crossings from appearing in the tour. It is well-known that good tours do not have any edge crossings.

Besides well-known construction heuristics such as, the nearest neighbour, the greedy algorithm, and insertion heuristics, there are different approaches for solving TSP instances. In some situations, such as when large TSP instances are involved, the speed of an algorithm is the priority, rather than the quality of solutions, [78]. Therefore, for large Euclidean TSP instances, various fast but not necessarily very accurate algorithms have been introduced. Some of these heuristics are the Space Filling Curves, Strip Heuristics and Decomposition Approaches, [75,79].

Few [72], has proved that for n points, with $n \geq 2$, the path length through them cannot be larger than $\sqrt{2n} + 1.75$ by using the strip idea. Beardwood et al., [73] have studied the shortest path for large numbers of points on a k-dimensional Euclidean space. They have assumed $k = 2$. They have shown that the optimum tour length is almost always proportional to $\sqrt{nv}$, where n is the number of points and v is the area of the plane defined. So, in the unit square this proportion has been defined as

$$\lim_{n \to \infty} \frac{c_{\text{opt}}}{\sqrt{nv}} = C, \tag{3.1}$$

where c_{opt} is the optimal solution, and $C \leq 0.9212$ is a constant.

In [80], the constant C found by Beardwood et al. was improved and the new constant was defined as $C \leq 0.894$ again by using the strip idea. Supowit et al. [74,76], have studied the length of the shortest TSP tour in the worst case, through n cities in the unit square. They have found the worst case tour length to be $\alpha\sqrt{n} + o(\sqrt{n})$, where $1.075 \leq \alpha \leq 1.414$. They introduced the strip algorithm which was inspired from the study in [73]. In their heuristic, the unit square is divided into $r = \lceil \sqrt{(\frac{n}{2})} \rceil$ vertical strips. Here, the number of

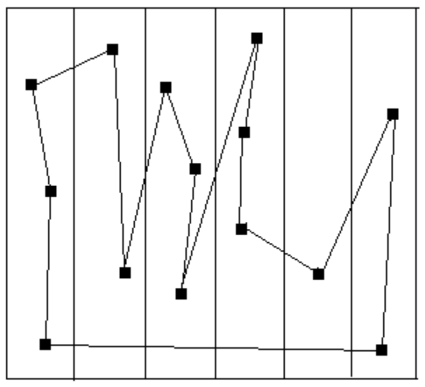

Figure 3.1: *An Illustration of the Classical Strip Algorithm.*

strips is guaranteed to be an integer value by using the ceiling function. The configuration of their strip algorithm can be seen in Figure 3.1. The first tour T_1 is formed by linking the points starting from the city which has the smallest y-axis value on the leftmost strip. The path goes up along that strip, then jumps to the next strip and goes down, and so on. The last city is connected to the first one to complete the tour. The second tour T_2 is formed in the same way. However, in order to construct the second tour the strips are shifted by $\frac{1}{2r}$ while the width of each strip is kept the same. The algorithm returns the tour with shortest length, [74]. The complexity of this algorithm is $O\ (n\ \log\ n)$, [81].

In [3], Dazango has introduced a new strategy for the expected length of tours by considering zones of various shapes and the density of vertices. The expected tour length was given as

$$D \approx \phi\sqrt{AN} \tag{3.2}$$

as $N \rightarrow \infty$, where N is the number of points uniformly distributed in the plane, A is the area of the plane, and ϕ is taken as 0.75 for the Euclidean metric. Further details are in [3].

In [82], Karloff has introduced the Local Strips method and proved that the upper bound for the shortest length of a TSP tour for n cities in a unit square cannot be larger than

$\alpha\sqrt{n}+11$, where n is the number of points and $\frac{\alpha}{\sqrt{(2)}} < 0.984$. Reinelt et al. [75], defined the strip algorithm as in Algorithm 13.

Algorithm 13 Reinelt's Algorithm, [75]

```
1: $s \leftarrow$ number of strips,
2: Split the planar graph into $s$ vertical strips of equal width and calculate the tour length;
3: Split the planar graph into $s$ vertical strips each having the same number of cities and
   calculate the tour length;
4: Split the planar graph into $s$ horizontal strips of equal height and calculate the tour
   length;
5: Split the planar graph into $s$ horizontal strips each having the same number of cities
   and calculate the tour length;
6: **Return** Best solution is chosen as the result.
```

The running time for this algorithm is estimated to be $\Theta\ (n\ \log\ n)$, and the number of strips is $s = \frac{\sqrt{n}}{2}$, where n indicates the total number of points. In [83], Johnson et al. modified the strip algorithm to overcome the problem that occurs with clustered data points/ cities. They ran the algorithm twice for each problem; once using vertical strips and once horizontal strips. They, then, chose the one which gives the better result. This method is called ″2-Way Strip Algorithm". Compared to the Classical Strip Algorithm, it is reported that this method has only minor improvement on average.

3.2.1.1 The Appropriate Number of Strips

In their algorithm, Supowit et al., [74], used $r = \lceil \sqrt{(\frac{n}{2})} \rceil$ strips for the vertices uniformly distributed vertices in the unit square. We applied this algorithm to some TSP problems taken from [4], but we used different numbers of strips chosen arbitrarily [8]. We also used the number of strips suggested in the original paper for comparison purposes. Results can be seen in Table 3.1. Instances with 100, 200, 280, 442, 575 and 1084 points have been solved. The numbers of strips were chosen arbitrarily as 12, 20, 30, 40. But, we also used $\lceil \sqrt{(\frac{n}{2})} \rceil$

Table 3.1: *CSA with various numbers of strips, [8]*

TSP Instances	True Solution	Average errors for strip numbers (%): r=12	r=20	r=30	r=40	r=$\lceil\sqrt{(n/2)}\rceil$
rd100	7910	63.80	129.45	188.26	231.51	33.93
kroA200	29368	37.67	60.55	92.90	133.30	38.70
a280	2579	57.31	51.49	84.18	119.83	57.31
pcb442	50778	37.49	56.22	82.83	126.42	48.77
rat575	6773	60.54	38.95	39.72	57.63	38.29
vm1084	239297	73.83	53.23	55.65	77.23	49.19

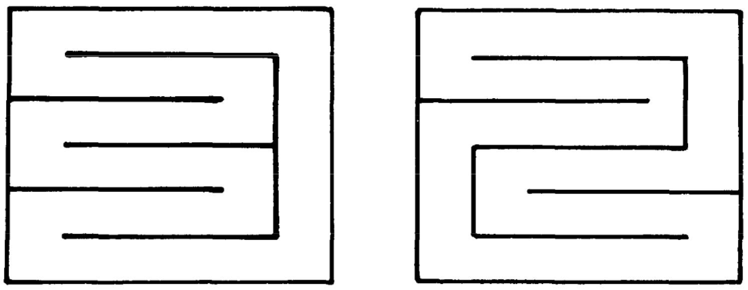
Figure 3.2: *Dazango's configuration of the Strip Algorithm, [3].*

strips.

Results show that, better approximations can be obtained using different numbers of strips. For kroA200 and pcb442 the algorithm with 12 strips gives better results. For 280 points, 20 strips give a better result. Reinelt et al. [75] defined the number of strips as $r = \frac{\sqrt{n}}{2}$, where n indicates the total number of nodes. Another study of the optimum number of strips can be found in [3]. Dazango [3], claimed that, for any rectangle containing the cities, the width of strips, i. e. their number, affects the quality of solution. If width w is too small then there will be extra length to connect points. But, if w is too large, then there will be zigzags which increase the tour length. For the configuration in his study, where A is the area, r is the number of strips and σ is the density of points in the unit area the optimum width was found to be $w = \sqrt{(\frac{3A}{\sigma})}$. Dazango's configuration of the strip algorithm can be seen in Figure 3.2.

3.2.2 2-PSA: the 2-Part Strip Algorithm

The difference between CSA, [74], and 2-PSA is that in the latter the plane is divided into two by one horizontal line. Furthermore, the number of strips can be changed in the given interval, so more alternatives can be compared in one method. In practical implementations, this algorithm aims at minimising the distance between the last visited point and the starting point. In both upper and lower part CSA is applied. It starts from the bottom part and follows the ascending or descending order of y-axis values to connect to the next strip. When it goes through the upper part, the same procedure is used and the last point meets the starting point. The time complexity of this algorithm is $O\ (n \log n)$, because of the sorting that has been performed with respect to the vertical coordinates. The 2-PSA can be described as in Algorithm 14, [8]. Note that r can be chosen between $r = \lceil \sqrt{(\frac{n}{2})} \rceil$ and width $w = \sqrt{(\frac{3A}{\sigma})}$ or through some experimentation. An example output of 2-PSA for a large TSP instance can be seen in Figure 3.3.

Algorithm 14 2-Part Strip Algorithm, [8]

```
1: $r \leftarrow$ number of strips;
2: Divide the grid into two horizontal parts each having roughly equal number of points;
   then divide each into $r$ vertical strips;
3: Proceed as in the basic strip algorithm from either of the horizontal parts, but reverse
   the sense of travel when at the end of the first part;
4: Connect the last point of the tour to the starting point to complete the Hamiltonian
   tour;
5: **Return** Return the tour and its length.
```

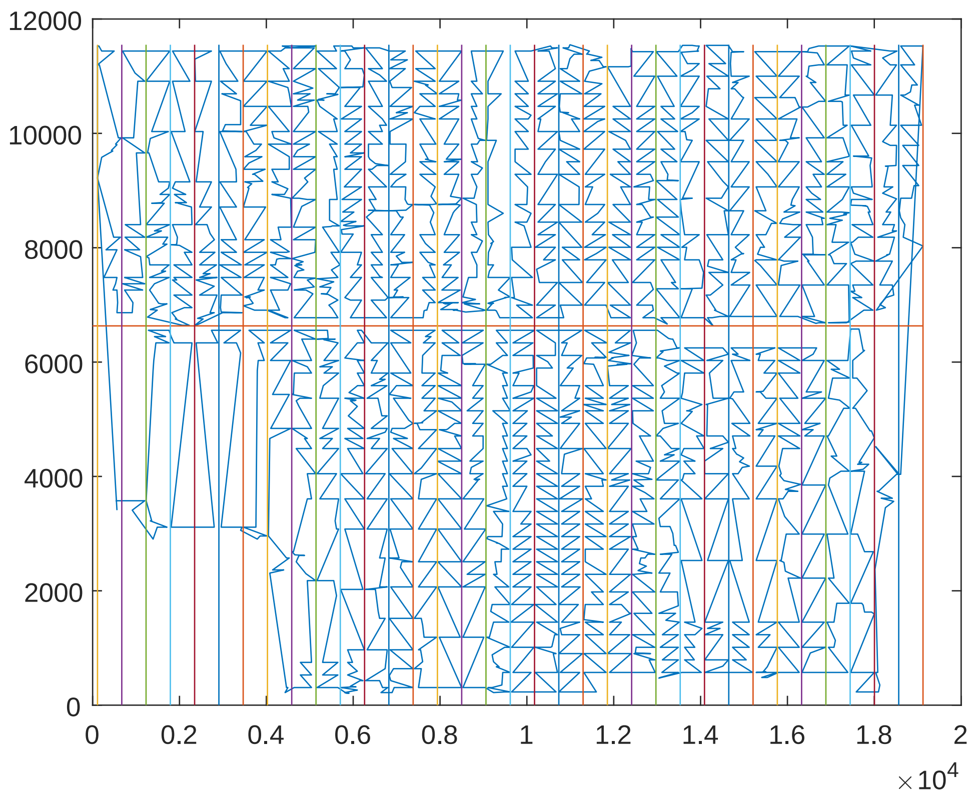

Figure 3.3: *Representation of the solution found by 2-PSA for TSP problem rl5915, [4].*

3.2.3 Worst-Case Analysis of 2-PSA and an Upper Bound on the Minimum Tour Lengths Returned

Since 2-PSA, [8], is a refined version of CSA, [74], we carried out its analysis in the same way. The L_2 metric has been used. Let there be n points uniformly distributed in the unit square. Construct two tours, T_1 and T_2 according to the procedure described in Algorithm 14 and let the lengths of these tours be LT_1 and LT_2, respectively. In order to define upper bounds on the optimum tour length, create two tours BT_1 and BT_2, the former following the median of a strip that is used to construct T_1 and the latter that of a shifted strip to the right by $\frac{1}{2r}$, that is used to construct T_2. Tours BT_1 and BT_2 are generated using the same

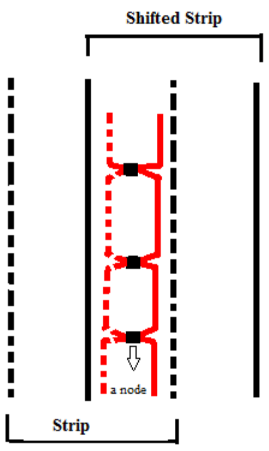

Figure 3.4: *The paths BT1 and BT2 for visiting points*

procedure as that of 2-PSA. However, since each path goes up or down through the median of the strip and its shifted couterpart, in order to connect the points, there is a jut that goes horizontally to the middle of the intersection of the two strips. This can be seen in Figure 3.4. A similar representation can be found in [74].

Assuming that the horizontal line cuts the unit square from the midpoint of the leftmost and the rightmost sides, the vertical length of the upper and lower parts becomes $\frac{1}{2}$ each. The total length of BT_1 and BT_2 are calculated in order to define an upper bound for the 2-PSA. Since the total vertical length for BT_1 is $\frac{r}{2}$ for the lower part and $\frac{r}{2}$ for the upper part, it is therefore, r for the total tour length. Similarly for BT_2, it is $\frac{r+1}{2} + \frac{r+1}{2} = r + 1$. Since the strips have been shifted to construct the second tour, the total number of strips has increased by 1. The total horizontal length for BT_1 to connect one strip to another is, $1 - \frac{1}{r}$ for the lower part, because on the leftmost and the rightmost strips, there are gaps of $\frac{1}{2r}$, and the same applies for the upper part, the total tour length would be therefore $2 - \frac{2}{r}$. For BT_2, this length is 2. In order to calculate the total horizontal length of visiting each point and coming back to the median line, let us assume that points have been placed $\frac{1}{4r}$ apart from each half strip. Therefore, from each path, which is on the median, there is a horizontal length of $2\frac{1}{4r}$. For both BT_1 and BT_2, the total horizontal length is $\frac{n}{r}$. Finally, the length of connecting the last visited point to the starting point is 1 in the worst case for both tours. By using the triangle inequality and assuming $r = \lceil \sqrt{(\frac{n}{3})} \rceil$, we can write

$$\begin{aligned} LT_1 + LT_2 &\leq length(BT_1) + length(BT_2), \\ &\leq r + r + 1 + \frac{n}{r} + (2 - \frac{2}{r}) + 2 + 1 + 1, \\ &\leq \frac{n}{r} + 2r + O(1), \\ &\leq 5\frac{\sqrt{3n}}{3} + O(1). \end{aligned} \tag{3.3}$$

Therefore, as in [74], either LT_1 or LT_2 is less or equal to $5\frac{\sqrt{3n}}{6} + O(1)$ which is the worse upper bound in terms of cost. However, experiments show that assuming that we know the optimal width of the strips, we can deduce that

$$length(T_1^{2-PSA}) \leq length(T_1^{CSA}).$$

In other words, 2-PSA generates better tours than CSA. But, referring to experimental evidence again, CSA is much faster. Note that, $O(1)$ denotes the constant execution time that does not depend on the size of the problem.

3.2.4 Other Implementations of the Strip Algorithm

Following the analysis of 2-PSA, we have developed other variants of the strip algorithm and investigated them. These variants and the results obtained with them are given below.

3.2.4.1 The Adaptive Strip Algorithm (ASA)

The classical strip algorithm is not effective on instances with clustered cities. To generate good quality initial tours for such TSP instances in a cheap way, a new idea has been explored. This idea consists in generating strips with approximately similar numbers of cities in them. A threshold of cities is arbitrarily chosen and then any strip with a number of cities greater than this threshold is subdivided into 2 new strips. The process is started with the initial grid containing all cities. Since the threshold is set well below the number of given cities, this initial grid or box is vertically split into two. The choice of splitting vertically first is arbitrary. Each resulting strip or box is then checked for the number of cities it has. If this number is higher than the threshold, the box is split further. The process continues. Note that any box which is a result of a vertical split is split horizontally, and any box which is the result of a horizontal split is divided vertically. This approach results in small boxes where there is a high density of cities and larger ones where the density is low. In each box, a representative city is then chosen at random. These representatives are linked up using CSA. Then in every box a path linking all cities is generated. These paths fall into the overall path which links the boxes, to form a Hamiltonian path. The last city in the last box is then linked to the first city in the first box to complete a tour. Note that, boxes with no cities in them are ignored. The Adaptive Strip Algorithm or ASA can be described as in Algorithm 15.

The graphical result of ASA for solving greece9882 problem can be seen in Figure 3.5. A 50-city TSP problem was solved by using both ASA and CSA. Graphical results can be seen in Figures 3.6 and 3.7. In this set of cities, there are clusters which cause CSA to work

Algorithm 15 **The Adaptive Strip Algorithm, [8]**

```
1: Split the unit square into horizontal and vertical strips. Each strip should contain a
   predetermined number of cities at most;
2: Choose a city from each strip as a representative of the cities in that strip;
3: Apply CSA to all representative cities to link them into a path;
4: Apply CSA to all cities in each strip;
5: Return Return the tour by linking up all paths within strips to each other and calculate
   its length.
```

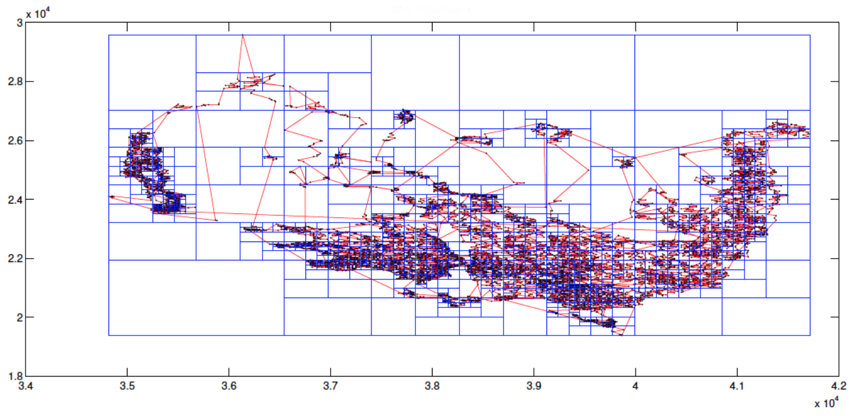

Figure 3.5: *ASA solution of greece9882, [5]*

ineffectively. Applying ASA, the same problem was solved by hand. The total length of the tour formed with CSA is 6.33. With ASA it is 5.50. This shows that the clustered TSP instances can be solved more efficiently using the new algorithm.

3.2.4.2 The Spiral Strip Algorithm (SSA)

In this approach, the plane is cut into a number of horizontal and vertical strips proportional to the number of cities. Assume that r is the number of vertical strips and p the number of horizontal strips. Therefore, $r \times p$ cells are created. Both r and p are calculated as $\lceil \frac{number\ of\ cities}{t} \rceil$, where t is a positive integer used as the number of strips. The application starts from the leftmost upper corner cell and follows a spiral and ends up about the middle of the plane. The Hamiltonian cycle is completed by connecting the ending point with the starting one.

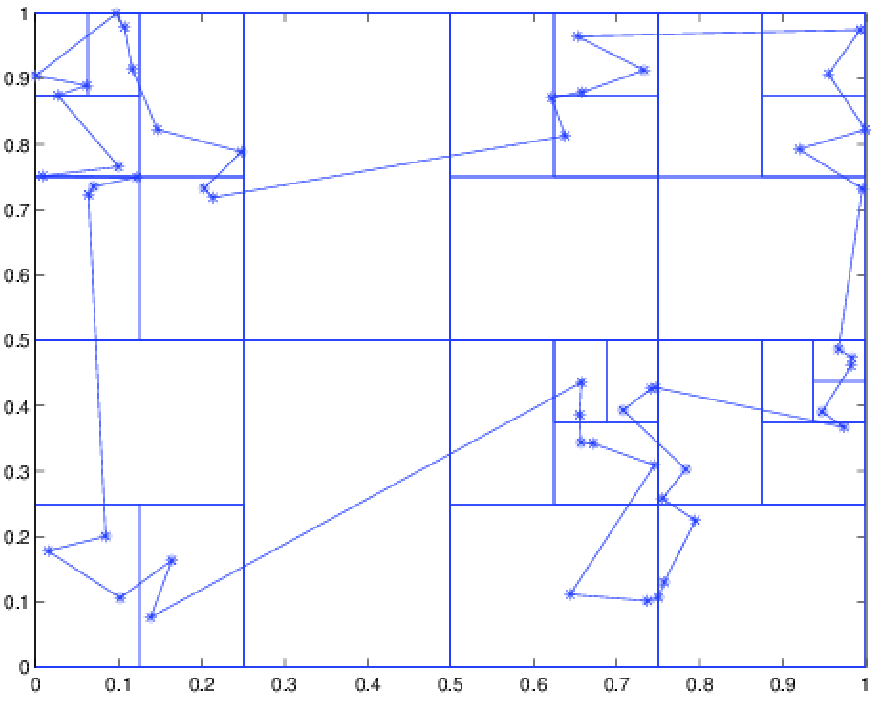

Figure 3.6: *The Adaptive Strip Algorithm on the 50-city problem*

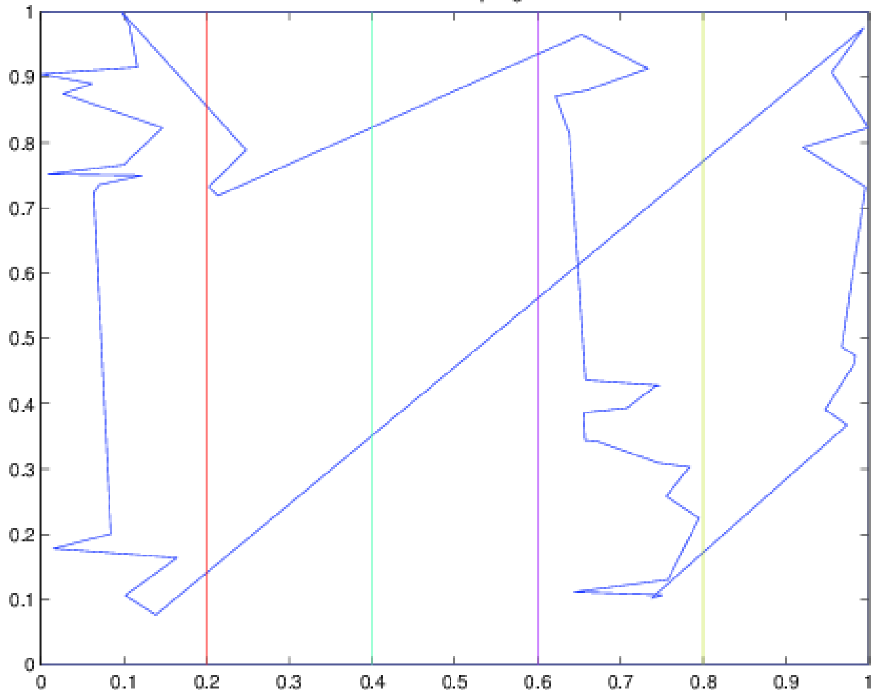

Figure 3.7: *The Classical Strip Algorithm on the 50-city problem*

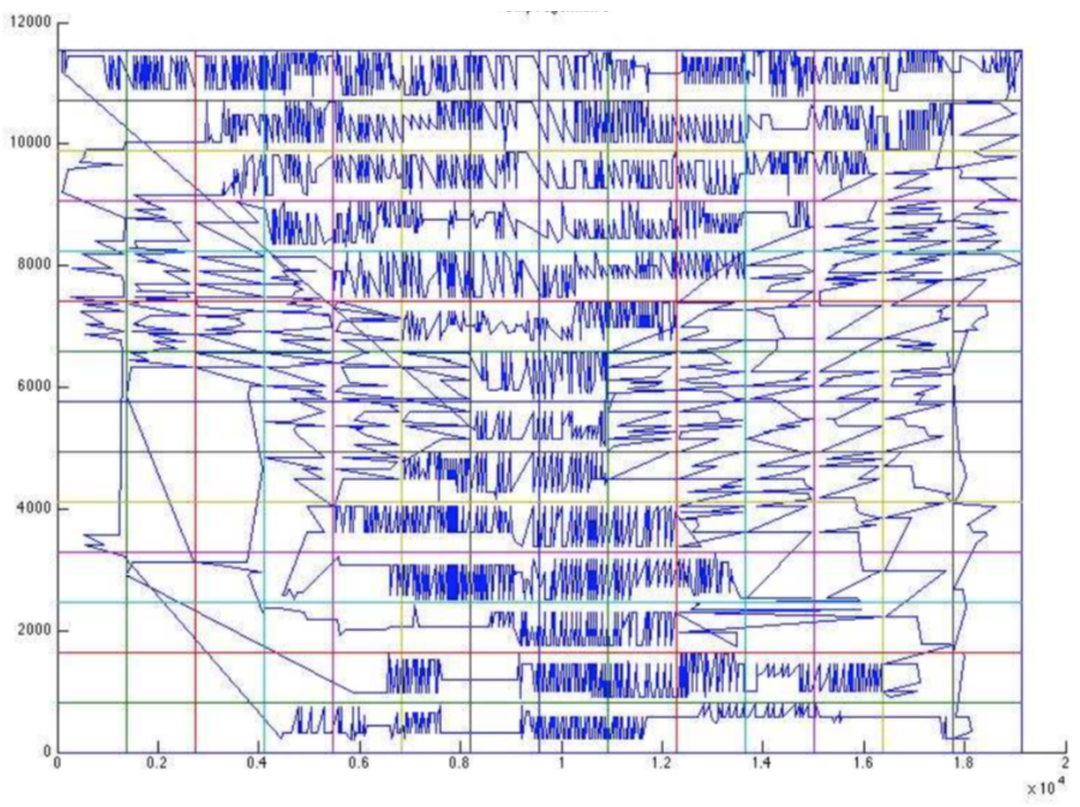

Figure 3.8: *SSA solution of problem rl5915, [4].*

One similar approach can be found in [84]. To justify the investigation of the SSA, we compare the tour length it produces with that of the CSA, for instance. Let consider two identical unit squares with n number of cities spread over each. In order to complete the Hamiltonian cycle, let solve each using CSA and SSA, respectively. If we use r strips to solve the problem with CSA, the total tour length in the worst case will be $r + \frac{r-1}{r} + \sqrt{2}$. Similarly, to solve the problem using SSA with r vertical and r horizontal strips, then, the total tour length in the worst case will be $\frac{3}{r}(r-1) + \frac{(r-2)(r-1)}{r} + \frac{\sqrt{2}}{2}$. The first part is the total length of three edges of the unit square. The second part is the total length of the inner path. Note that the spiral path is getting smaller as it gets closer to the centre. Finally, the last part is the length of the connection between the centre and the starting point. In both cases, it is assumed that nodes are exactly on the strips. Comparison of the results of each problem proves that

$$length\ (T^{SSA}) \leq length\ (T^{CSA}). \tag{3.4}$$

An example output of the implementation can be seen in Figure 3.8.

3.2.5 Computational Results

We have implemented 2-PSA and other similar schemes and tested them on standard TSP problems ranging from 51 to 5915 cities, [4]. The computing platform is an Intel core I5 PC with a 3.40 GHz processor and 16 Gigabytes RAM, running Windows 7. All algorithms are coded in Matlab R2014a. The results show the value of the strip approach at least as a tool for generating computationally cheap but better tours than those generated randomly. Note that it was not expected to generate solutions too close to the optimum, hence the relatively large errors observed. The aim is to have something which is better than just random tours. Speed is the essence. For comparison purposes, the CPU time of 2-PSA for the TSP with 9882 cities is less than $\frac{1}{10}^{th}$ of a second, while that of the greedy algorithm is around 8 minutes. Comparative results between the different variants of the strip algorithm and random permutation considered have also been recorded in Table 3.2; the superiority of 2-PSA over CSA, SSA and the random permutation is very clear.

In Table 3.3, real-life instances [5] have been used to compare the performance of CSA, 2-PSA, ASA and the random permutation. Note that 2-PSA and ASA give better results than CSA. Table 3.4 records the total computation time for each algorithm. ASA has solved all problems in a reasonable time. Results show that 2-PSA runs 7 to 20 times longer than CSA, however it reduces the error from 10% to nearly half that. ASA is faster than 2-PSA, and for large instances, it gives better tour lengths than both CSA and 2-PSA. Results demonstrate

Table 3.2: *Deviation (in %) from the optimum of CSA, SSA, 2-PSA and Random Solutions*

TSP instances	No. of cities	Opt. Sol.	CSA	SSA	2-PSA	Rand. Perm.
eil51	51	426	21.67	40.63	19.62	303.28
st70	70	675	31.94	47.93	17.75	443.46
rd100	100	7910	32.84	48.18	23.80	598.31
a280	280	2579	41.50	62.19	29.71	1190.02
rat575	575	6773	47.70	37.54	20.83	1604.73
vm1084	1084	239297	51.74	519.27	40.38	3498.61
vm1748	1748	336556	55.58	671.69	44.19	4342.15
rl5915	5915	565530	78.24	163.03	64.11	7436.97

Table 3.3: *Deviation (in %) from the optimum of CSA, 2-PSA, ASA and Random Solutions*

TSP instances	No. of cities	Opt. Sol.	CSA	2-PSA	ASA	Rand. Perm.
usca50	50	14497	108.79	54.69	97.61	451.08
zimbabwe929	929	95345	68.13	57.27	61.91	1319.46
canada4663	4663	1290319	158.17	114.88	86.11	1793.95
greece9882	9882	300899	90.81	90.12	87.75	12529.2

that although the random permutation is the fastest, it gives the worst approximate solution compared to the others.

The Strip Heuristic is cheap yet effective in finding good tours in a short time. The proposed algorithms, 2-PSA and ASA have given better results than CSA. Although the returned solutions are far from the optimum solutions in terms of quality, they have been obtained quickly. This makes them potential providers of initial populations for other meta-

Table 3.4: *CPU time of each algorithm on large problems*

CPU Time (s)	usca50	zimb929	ca4663	gre9882
CSA	0.008	0.003	0.006	0.012
2-PSA	0.028	0.061	0.117	0.083
ASA	0.006	0.009	0.020	0.047
Random Permutation	0	0	0	0.001

Table 3.5: *An Example of a RW Selection*

Items	Fitness	Normalised Fit.	Probabilities (%)
1	5	0.05	5%
2	10	0.1	10%
3	42	0.42	42%
4	20	0.2	20%
5	23	0.23	23%

heuristics such as the Plant Propagation Algorithm or the Strawberry Algorithm [6,9,59,60] and the Genetic Algorithm [85].

3.3 KP Solutions: Roulette Wheel Approach

The Roulette Wheel (RW) method is one of the most common and easy-to-implement approaches used in the parents' selection process of GA [86], in particular. Here, we explain how it can be used in the context of PPA for the Knapsack problem. The approach works as follows. For each item a probability is assigned based on their $\frac{p_j}{w_j}$ proportion, where p_j and w_j denote the value and weight of item j, respectively. Then, by using the RW approach sets of solutions are generated. For multiple knapsack problems, the procedure starts with assigning items to the knapsack with the smallest capacity and then proceed in the same way until all knapsacks are filled, [87]. Some of the sets are generated by using random generation to ensure diversity in the population. A simple example and an illustration of a roulette wheel are given in Table 3.5 and Figure 3.9, respectively.

Assuming that the fitness values represent $\frac{p_j}{w_j}$ proportion values for each item j to be placed in a single KP. This proportion denotes the profit of item j per unit weight. Then the normalised fitness values are calculated using $\frac{fitness_j}{\sum fitness}$ and probabilities to be chosen

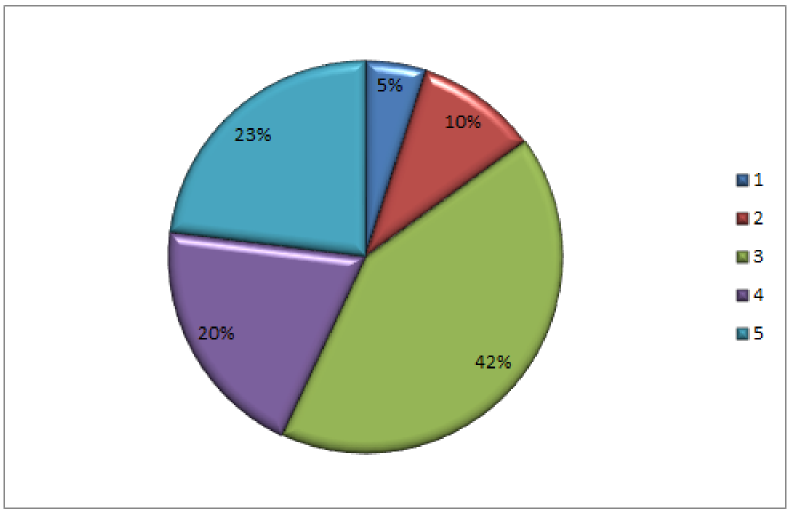

Figure 3.9: *Proportionate fitness representation on a roulette wheel.*

are assigned by multiplying each normalised fitness value by 100. During the selection process cumulative probabilities are calculated by adding up probabilities and reaching 1. Then, a random value between 0 and 1 is generated and by checking where the value falls in the cumulative probability, the corresponding item is chosen. According to the given example in Table 3.5, it is expected to have item 3 in the knapsack with a priority since it has the largest probability to be chosen.

3.3.1 Complexity of the Roulette Wheel Approach

In RW approach, we search the target value within a list of assigned proportions. Each element is checked sequentially to find into which slot the randomly generated value falls. This is called linear search, [88]. Checking for each random value takes $O(n)$ time. However, as the algorithms needs to be performed for all n items, it requires n spins. Therefore, the time complexity of this method is $O(n^2)$, [86].

3.4 Edge Seeking Heuristics for Scheduling

In this section, we present some new heuristics we refer to as Edge Seeking Heuristics (ESH) designed to generate quickly feasible or near feasible schedules for population-based heuristics such as GA and PPA when applied to scheduling problems of the Berth Allocation Problem (BAP) type. Five heuristics are considered. It is assumed that a schedule is a box or rectangle with time on its horizontal edge and space on its vertical one, and a ship is a rectangular plank. Ships must be placed in the box without any overlap. However, these heuristics do not guarantee feasible solutions. All heuristics have been introduced and compared with different size of problems ranging from 5 to 30 vessels. All data has been generated randomly. While generating vessels $Area\ of\ box \geq \sum Area\ of\ vessels$ is considered. The maximum length and the maximum time are fixed while running the experiments depending on the size of the problem. Here, the maximum width corresponds to the wharf length, and the maximum time is total processing time of all vessels found in the worst case, i.e handling of a vessel cannot be started before its predecessor task is completed. Illustrations of a 10-vessel problem and its solution with each ESHs are provided later.

3.4.1 Edge Seeking Heuristic 1

ESH1 is the simplest version of these algorithms. If $T_v - 0 \leq Maximum\ time - T_v - h_v$, where T_v and h_v are berthing time and handling time of a vessel, respectively, it is moved to the left and its arrival time becomes 0. Otherwise, it is placed at the point $Maximum\ time - h_v$. Similarly, the same procedure is applied in the vertical line, if $P_v - 0 \leq Maximum\ length -$

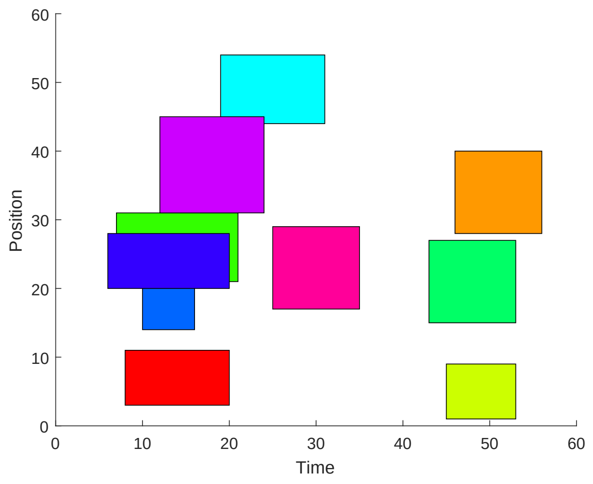

Figure 3.10: *Initial positions of a 10-vessel Problem*

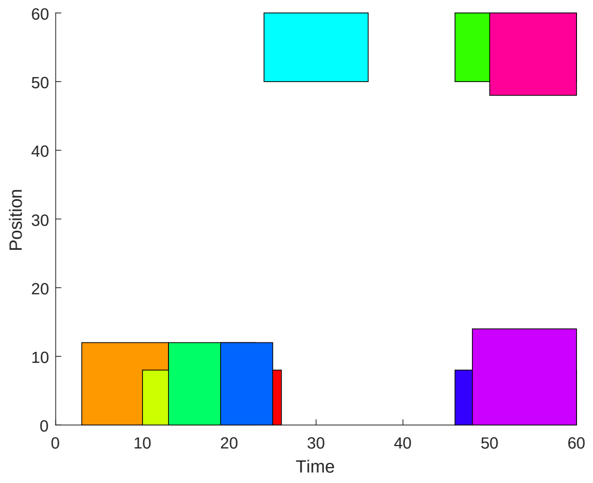

Figure 3.11: *Example ESH1 output for a 10-vessel problem*

$P_v - L_v$, where P_v and L_v are berthing position and length of a vessel, respectively, it is pushed to 0, otherwise, it is placed at $\textit{Maximum length} - L_v$. In the end, vessels that are starting before their arrival times are pushed back to them. An illustration is shown in Figures 3.10 and 3.11. The working mechanism of ESH1 is shown in Figure 3.12.

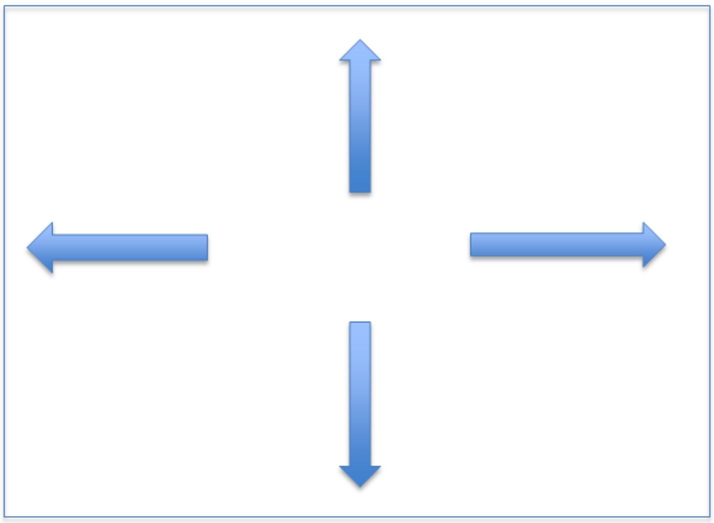

Figure 3.12: *The working mechanism of ESH1*

3.4.2 Edge Seeking Heuristic 2

This approach starts with implementing ESH1. In ESH2, it is assumed that the main box consists of four equally divided boxes. The mid-coordinates of each vessel are then found. ESH1 is implemented in each box and each vessel takes a place determined by their mid-point coordinates, i.e the vessel is placed in that corresponding box where the coordinates fall. In the end, vessels that are starting before their arrival times are pushed back to them. The working mechanism of ESH2 is shown in Figure 3.13. An example implementation is shown in Figure 3.14.

3.4.3 Edge Seeking Heuristic 3

This heuristic is slightly different from ESH2. Here, again, the main box is assumed to consist of 4 equal boxes. Each vessel is assigned to the boxes where their mid-point coordinates fall. Then, both vertical and horizontal positions are determined for each vessel implementing ESH1 in each box. In the end, vessels that are starting before their

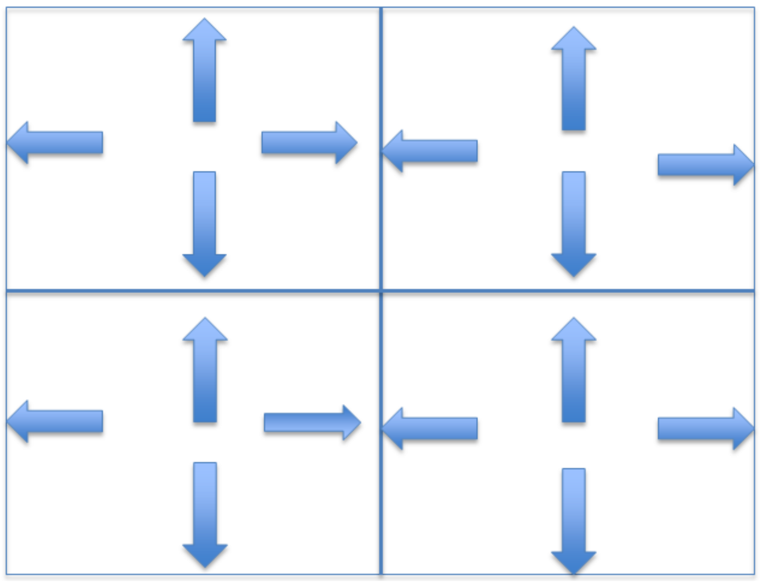

Figure 3.13: *The working mechanism of ESH2*

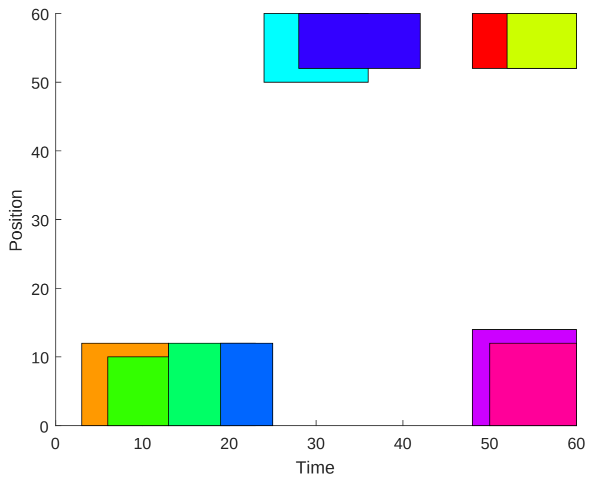

Figure 3.14: *Example ESH2 output for a 10-vessel problem*

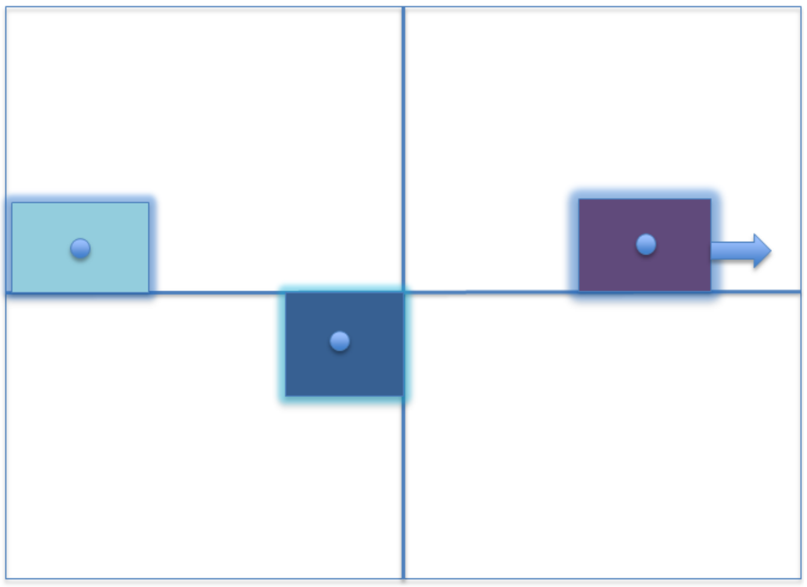

Figure 3.15: *The working mechanism of ESH3*

arrival times are pushed back to them. The working mechanism of ESH3 is shown in Figure 3.15. An example implementation of it to solve a 10-vessel problem is shown in Figure 3.16.

3.4.4 Edge Seeking Heuristic 4

In this approach, a smaller box is placed inside the main box. The size of the small box is $1/r$ of the main box, where r is a pre-determined number. Then, each vessel is checked and placed in the box where their mid-point falls. Note that, it is assumed that the small box in the middle of the main box is equally far from all edges. The main box is divided into four small boxes. First, ESH1 is applied inside the small box for the vessels that fall in there, and then, the same approach is implemented for the four sub boxes of the main box for the rest of the vessels. The working mechanism of ESH4 is illustrated in Figure 3.17. An example implementation of it to solve a 10-vessel problem is shown in Figure 3.18.

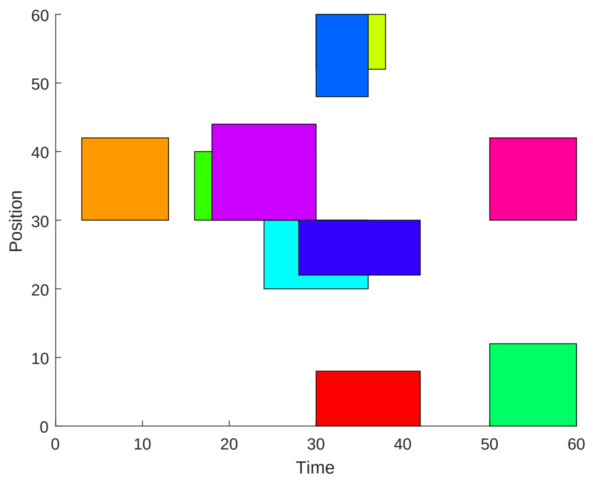

Figure 3.16: *Example ESH3 output for a 10-vessel problem*

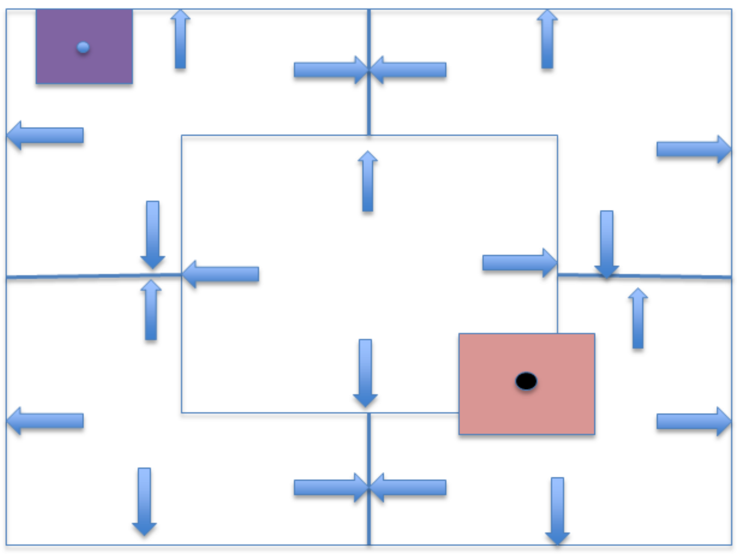

Figure 3.17: *The working mechanism of ESH4*

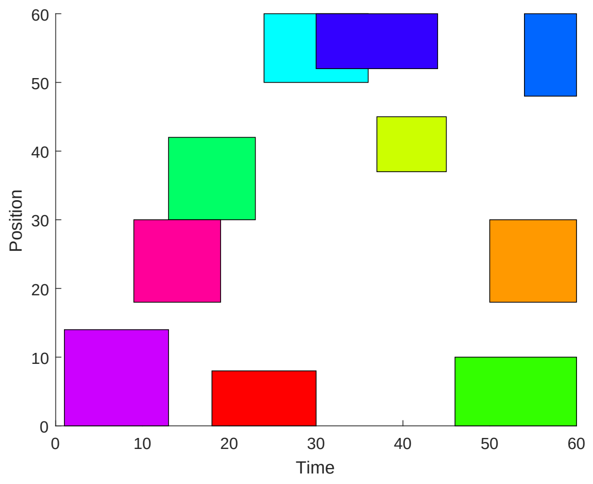

Figure 3.18: *Example ESH4 output for a 10-vessel problem*

3.4.5 Edge Seeking Heuristic 5

ESH5 is slightly different from ESH4. Similarly in ESH4, a small box is placed inside the main box. The size of the small box is $1/r$ of the main box, where r is a pre-determined value. Each vessel is checked and placed in the box where their mid-point falls. Here, again, it is assumed that the small box in the middle of the main box is equally far from all edges. The main box is divided into four small boxes. This time, the inner box is also divided into four. First, ESH1 approach is applied inside the small box for the vessels that fall in there also considering the four boxes obtained when it is divided, and then, the same approach is implemented for the four sub boxes of the main box for the rest of the vessels. The working mechanism of ESH5 is shown in Figure 3.19. An example implementation of it to solve a 10-vessel problem is shown in Figure 3.20.

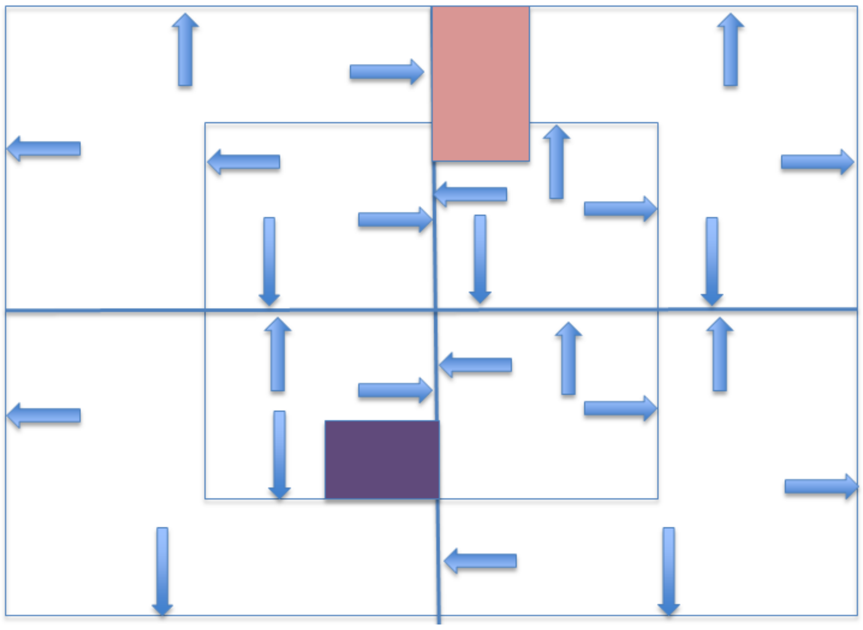

Figure 3.19: *The working mechanism of ESH5*

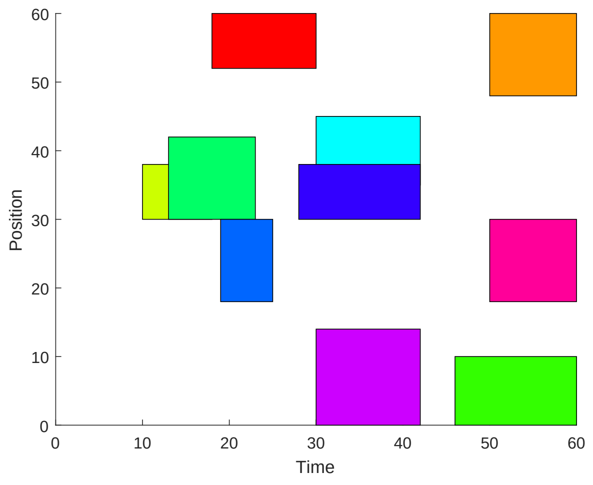

Figure 3.20: *Example EHS5 output for a 10-vessel problem*

3.4.6 Complexity Analysis of ESH

Let us consider ESH1. The aim is to push a vessel to its nearest edge of the main box which represents a schedule. This heuristic checks for each vessel how far it is from each edge to find the nearest and then moves it there by altering its position. Clearly, this is an $O(n)$ procedure, where n denotes the number of vessels. The same analysis shows that the other heuristics have complexity $O(mn)$ since the main box is partitioned into m boxes and the procedure is replicated for each small box.

3.4.6.1 Experimental Results and Comparisons

All algorithms were tested on problems with sizes ranging from 5 to 30 vessels. Each algorithm was run a 100 times. In Table 3.6, "Box size" is "maximum length × maximum time". "Tolerance" shows the overlapping percentage that is acceptable depending on the number of vessels. Results show that for 5 vessels, all algorithms work well. However, as the number of vessels increases ESH 1, 2 and 3 show very low performance. ESH5 gives the best results compared to the rest of the variants. Here, we should remember that these are simple but effective and cheap heuristics introduced to generate good initial populations and/or to be used as part of long runners.

3.5 Summary

In this chapter we have discussed the importance of initializing population-based heuristics with good populations. Although diversity is important for exploration, with a good set of initial individuals mixed with a number of randomly generated ones, the total computation

Table 3.6: *Experimental result of ESH for various sizes of problems (All experiments are repeated 100 times.)*

	Box size	Tolerance	Prob. Size	% with Tol.	No overlap	Time (s)
	60x60	20%	5	40%	4%	0.0018
	60x60	30%	10	1%	0%	0.0037
ESH1	90x90	40%	20	0%	0%	0.009
	120x120	40%	30	0%	0%	0.0158
	120x120	50%	30	0%	0%	0.0158
	60x60	20%	5	41%	3%	0.0018
	60x60	30%	10	1%	0%	0.0037
ESH2	90x90	40%	20	0%	0%	0.0089
	120x120	40%	30	0%	0%	0.0158
	120x120	50%	30	0%	0%	0.0158
	60x60	20%	5	50%	9%	0.0016
	60x60	30%	10	36%	1%	0.0035
ESH3	90x90	40%	20	1%	0%	0.0086
	120x120	40%	30	0%	0%	0.0154
	120x120	50%	30	1%	0%	0.0154
	60x60	20%	5	58%	17%	0.0015
	60x60	30%	10	33%	0%	0.0034
ESH4	90x90	40%	20	10%	0%	0.0086
	120x120	40%	30	3%	0%	0.0154
	120x120	50%	30	12%	0%	0.0154
	60x60	20%	5	59%	12%	0.0016
	60x60	30%	10	28%	1%	0.0035
ESH5	90x90	40%	20	13%	0%	0.0085
	120x120	40%	30	16%	0%	0.0155
	120x120	50%	30	36%	0%	0.0155

time can be reduced. We proposed two cheap yet effective heuristics to generate the initial populations for TSP and RBAP. For KP, we used the well-known Roulette Wheel approach. First, we discussed how to generate good initial populations for TSP. We picked the Strip Algorithm and proposed new variants of it. We provided the worst-case analysis of the new variants, carried out experiments and compared the results obtained with the variants on randomly generated solutions. We then explained how the RW approach is used to generate good initial solutions for KP. Finally, we proposed 5 easy heuristics to generate good initial populations for RBAP in container ports. The aim was to generate good schedules, i.e placing vessels in a designated area without overlaps. We ran experiments to demonstrate the performance of these heuristics. In the following chapters we will show how to implement PPA to solve iconic problems of TSP, KP and RBAP taking advantage of the material expanded in this chapter, namely good initial population of plants.

CHAPTER 4

THE PLANT PROPAGATION ALGORITHM FOR THE TRAVELLING SALESMAN PROBLEM

4.1 Introduction

After giving a brief description of discrete PPA in the previous chapters, in this chapter we investigate its implementation to solve the Travelling Salesman Problem (TSP), [6].

4.2 TSP: A Brief Review

A notoriously difficult and yet easy to state representative of NP-hard problems is the well known Travelling Salesman Problem, or TSP, [77]. The aim is to find the Hamiltonian cycle of shortest length in the complete weighted graph that represents a fully connected set of cities where the edges represent the connections between each pair of cities; an edge weight is the distance (time, cost ...) that separates a pair of cities linked by the edge. Depending on the properties of these weights, we get different types of TSP. When the weights c_{ij} are put together in a square matrix $C = c_{ij}, \forall\ i, j$ and $c_{ij} = c_{ji}, \forall\ i, j$, then we have a symmetric TSP or STSP, by virtue of matrix C which is symmetric. It is asymmetric if this property does not hold. If entries of C fulfil the triangle inequality, i.e. $c_{ik} \leq c_{ij} + c_{jk}, \forall\ i, j, k$, the TSP is called metric. When c_{ij} is given as the Euclidean distances between nodes, the TSP is said to be Euclidean, [77,89].

There are various methods to solve the TSP. As for other intractable combinatorial optimisation problems, exact algorithms are available, but only for relatively small instances; they do not work for large instances for efficiency reasons. Therefore, many heuristic approaches have been proposed in this respect, [90,91].

Some of the exact algorithms for TSP are the branch and bound [92], branch and cut algorithms [93] and the Held-Karp algorithm which is based on dynamic programming [94]. Some of the popular heuristic algorithms are Lin-Kernighan Local Search [95] and the strip algorithm [74]. Metaheuristics have become popular since they are able to find near optimal results in reasonable time even for large instances [33,70,75]. These are Nature-inspired algorithms which are now widely used to solve both continuous and discrete optimization problems. Most of these have been applied to TSP. Early examples are the Genetic Algorithm (GA) [96], Simulated Annealing (SiA) [97], Ant Colony Optimisation (ACO) [69], Discrete Particle Swarm Optimisation (DPSO) [98,99] and Tabu Search (TS) [100].

In the following we review some of the most recently introduced Nature-inspired algorithms. Tsai et al. [101] have developed an algorithm called Heterogeneous Selection Evolutionary Algorithm (HeSEA). The algorithm has been tested on 16 TSP benchmark problems ranging from 318 to 13509-city problems. The algorithm is able to find the optimum result for up to 3038-city problem. The average errors are 0.05%, 0.01% and 0.74% for problems that have 4461, 5915 and 13509 cities, respectively. Song and Yang [102] have proposed an improved ACO introducing new strategies in order to increase the quality of the original algorithm. The new approach has given better results than the classical ACO and has found even better results than the best known solutions for some TSPs. Marinakis et al. [103] have proposed a hybrid algorithm to solve the Euclidean TSP problem. Their approach combines Honey Bees Mating Optimization algorithm (HBMOTSP), the Multiple Phase Neighborhood Search-Greedy Randomized Adaptive Search Procedure (MPNS-GRASP) and the Expanding Neighborhood Search Strategy. Experiments have been run on 74 benchmark TSP instances and have given competitive results.

Karaboga and Gorkemli [41] implemented the Combinatorial Artificial Bee Colony algorithm (CABC). They have adapted the Greedy Sub Tour Mutation (GSTM) operator proposed by Albayrak and Allahverdi [42], which increases the capability of GA to find the shortest length in TSP. The algorithm was used to solve two TSP instances with 150 and 200 cities, respectively. In [43], Gorkemli et al. have introduced the Quick Combinatorial Artificial Bee Colony Algorithm (qCABC) and improved CABC by changing the behaviour of onlooker bees. The new algorithm was tested against 9 heuristic methods including the CABC on the same instances. The qCABC outperforms all algorithms except CABC on the 150-city problem. In [104], Li et al. have developed a Discrete Artificial Bee Algorithm (DABC) and applied it to TSP. They used the Swap Operator to represent the basic ABC for discrete problems. The performance of the algorithm was compared to that of PSO algorithm. Experimental results show that DABC outperforms PSO.

Table 4.1: *A compilation of recent notable results, [6]*

Authors	Year	Algorithm	TSP size	Avg. Error (%)
Karaboga et al.	2011	CABC	150 and 200 cities	0.9% and 0.6% respctively
Li et al.	2011	DABC	14 to 130 cities	Changing from 0.55% to 6.41%
Jati et al.	2011	EDFA	16 to 666 cities	0% up to 225-city instances and the 666-city problem, less than 12% for instances of 225,280 and 442 cities
Karaboga et al.	2013	qCABC	150 and 200 cities	0.7% and 0.5% respctively
Jati et al.	2013	New EDFA	16 to 666 cities	0% up to 225-city instances and the 666-city problem, less than 12% for instances of 225, 280 and 442 cities
Ouaarab et al.	2014	DCS	51 to 1379 cities	0% for 13 out of 41 instances, less than 4.78% as the worst for the 1379-city instance
Saenphon et al.	2014	FOGS-ACO	48 to 200 cities	0% for the instance of 51-city, changing from 0.062% to 1.64% for the other instances
Mahi et al.	2015	PSO-ACO-3Opt	51 to 200 cities	Changing from 0.00% to 0.95%
Zhou et al.	2015	DIWO	48 to 2392 cities	Changing from 0.00% to 3.1%
Osaba et al.	2016	IBA	30 to 1002 cities	Changing from 0.00% to 7.5%

Jati and Suyanto [105] have introduced the Evolutionary Discrete Firefly Algorithm (EDFA) and tested it against the Memetic Algorithm (MA). EDFA was found to be better than MA on TSP instances. An improved version of EDFA was developed later by Jati et al. [70]; it uses a new movement scheme. This new version has outperformed the previous one in terms of efficiency. Zhou et al. [58] proposed a discrete invasive weed optimisation (DIWO) to solve TSP. The performance of the algorithm was tested on twenty benchmark TSP instances. The experimental results showed that DIWO can find results close to the optimal values within a reasonable period of time, and it also has strong robustness. Osaba et al. [49] developed an improved discrete bat algorithm (IBA) for symmetric and asymmetric TSP. They compared the performance of the algorithm in 37 instances with the performance of five different methods in the literature. Results show that IBA has outperformed all the other alternatives in most of the cases. Table 4.1 is a compilation of a set of results reported in fairly recent papers. The last column records the performance of the concerned algorithm on a set of TSP problems.

4.3 Implementation of PPA for TSP

In any algorithm and in particular in population-based ones, the representation of individuals/solutions is a key aspect of their implementation. The issue here is the representation of a plant which itself represents a solution. A solution here is any Hamiltonian cycle (tour) of the complete graph representation of the TSP. Note that representation affects the way the search/optimisation process as well as any stopping criteria which are implemented.

1	4	6	9	12	20	5	7	8	11	15	3	17	16	2	10	14	18	13	19

Figure 4.1: *The permutation representation of a plant as a tour, [6]*

4.3.1 The Representation of a Tour

A plant in the population of plants maintained by PPA is a tour/solution represented as a permutation of cities. X_i is tour i, $i = 1, \ldots, NP$. The size of the population of plants is NP. Tours/plants are ranked according to their lengths. The tour length of plant i is denoted by N_i; it is a function of X_i. Without loss of generality, the Euclidean TSP is considered here. Tour lengths, therefore, are calculated according to the Euclidean distance

$$d_{x,y} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{4.1}$$

In Figure 4.1, the entries of the array represent cities. City 1 and city 4 are successive in the depicted tour and the notation 1-4 defines the edge between them.

4.3.2 The Distance Between Two Plants

One of the issues in implementing PPA is defining the distance that separates tours. Here it is defined as the number of exchanges to transform one tour into another. After sorting the tours by their tour lengths, a pre-determined number of the tours is taken amongst the ones that have good short lengths; short runners are then sent from these plants, i.e. new neighbouring tours are generated from them. The 2-opt rule is used for this purpose since it require the minimum number of changes to create new tours. The 2-opt move is implemented by removing two edges from the current tour and exchanging them with two other edges, [106].

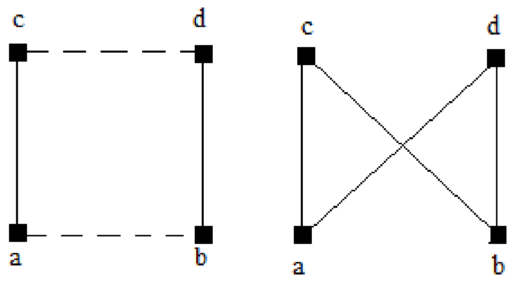

Figure 4.2: *2-opt Exchange of a 4-city tour, [6]*

An illustration of a 2-opt exchange is shown in Figure 4.2. There, tour **a-b-d-c-a** has been transformed into **a-d-b-c-a** by exchanging edges **a-b** and **d-c**.

Similarly, long runners are implemented by applying a k-opt rule with $k > 2$. In fact, this is pretty much the Lin-Kernighan algorithm (LK) [95]. It changes k edges in a tour, with k other edges. If, in this process, shorter tours are preferred and kept, then it will converge to potentially better solutions than it started with, [107].

This is not the only way available to measure the distance separating any two tours or permutations. However, in the literature, this method is widely used because of its efficiency and ease of implementation, [23, 95]. An alternative approach can be found in [108], where a metric space of permutations defined using the k-opt rule has been studied.

4.3.3 Short and Long Runners

As mentioned earlier, in the basic PPA, a plant sends many short runners when it represents a good solution (exploitation move), or a few long runners when representing a poor solution (exploration move). Short runners are implemented using the 2-opt rule, and long runners, the k-opt rule, with $k > 2$.

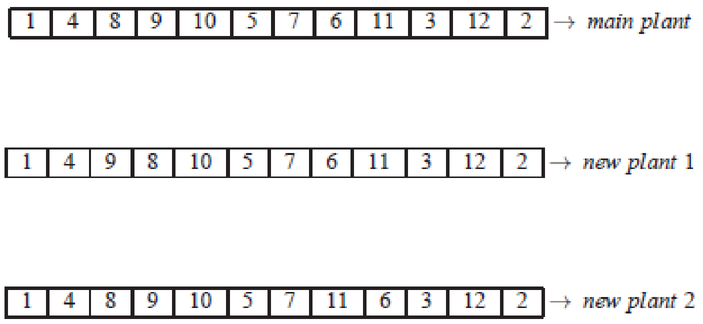
1 4 8 9 10 5 7 6 11 3 12 2 → main plant
1 4 9 8 10 5 7 6 11 3 12 2 → new plant 1
1 4 8 9 10 5 7 11 6 3 12 2 → new plant 2

Figure 4.3: *Illustration of short runner generation from a main plant, [6]*

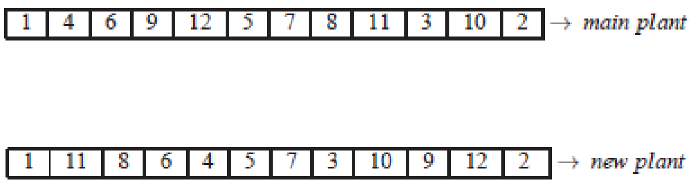
1 4 6 9 12 5 7 8 11 3 10 2 → main plant
1 11 8 6 4 5 7 3 10 9 12 2 → new plant

Figure 4.4: *Illustration of one long runner generation from a main plant, [6]*

An illustration of 2-opt rule implementing short runners can be seen in Figure 4.3. In the figure, only two 2-opt neighbours of the main plant are shown. The first new plant was generated by exchanging the edges **4-8** and **8-9** and for the second one the edges **7-6** and **6-11** were exchanged. Note that the exchanged edges do not have to be adjacent.

Those tours in the population deemed to be representing poor solutions send one long runner each to explore the search space for better solutions. This is reasonable since a plant in a poor spot can hardly afford to send many long runners as sending a long runner requires a lot of energy consumption and it is not easy for a plant which is trying to survive in a poor spot. For both short and long runner cases, if the new tours have better results from these exchanges they are adopted and kept as new tours. Otherwise they are ignored.

An illustration of a new plant produced by sending a long runner can be seen in Figure 4.4. The new plant is a 6-opt neighbour of the main plant. A 6-opt move can either be achieved by exchanging 6 edges chosen with other 6 edges or by implementing at most of six 2-opt moves sequentially, [109].

4.4 Pseudo-code of Discrete PPA

To the light of the general idea of implementing discrete PPA, we aimed at keeping the total computation time as short as possible. Therefore, it has been decided to start the algorithm with a good population of plants (tours). Diversity in the initial population is assumed to be guaranteed by the random processes used to generate tours. There are various such processes. Here, the initial population is generated using the greedy algorithm, random permutation or the strip algorithm, [8,75]. See also Chapter 3.

For short runners, the 2-opt algorithm has been implemented using speed-up techniques such as, “don’t look bits", and “fixed radius search". Fixed radius search can be achieved either finding a given number of nearest cities to each city, or using specially defined measures to find the nearest cities [110]. The main idea of the don’t look bits strategy is to restrict the nodes which have been chosen to be searched and have not given an improved solution [106, 110]. For long runners, the 2-opt has been implemented sequentially many times in order to complete the number of exchanges a k-opt rule with $k > 2$ would achieve, [107].

4.5 Computational Results and Discussion

Four sets of experiments have been carried out. The first set compares PPA with GA and SiA. The second compares it with modified PSO and the third with New DFA. Final set of experiments carried out to see the performance of discrete PPA on large TSPs and the

Algorithm 16 **Pseudo-code of Discrete PPA for TSP, [6]**

```
1: Generate a population $P = \{X_i,\ i = 1, \dots, NP\}$ of valid tours; choose values for $g_{\max}$
   and $y$;
2: g = 1
3: while g < $g_{\max}$ do
4:     Compute $N_i = f(X_i), \forall\ X_i \in P$;
5:     Sort $N = \{N_i,\ i = 1, \dots, NP\}$ in ascending order (for minimization);
6:     for $i = 1 : E(NP/10)$, Top 10% of plants do
7:         Generate $\lceil (y/i) \rceil$ short runners for plant $i$ using 2-opt rule, where y is an
           arbitrary parameter;
8:         if $N_i > f(r_i)$ then
9:             $X_i \leftarrow r_i$;
10:        else
11:            Ignore $r_i$;
12:        end if
13:    end for
14:    for $i = E(NP/10) + 1 : NP$ do
15:        $r_i = 1$ runner for plant $i$ using $k$-opt rule, $k > 2$, 1 long runner for each plant
           not in the top 10%;
16:        if $N_i > f(r_i)$ then
17:            $X_i \leftarrow r_i$;
18:        else
19:            Ignore $r_i$;
20:        end if
21:    end for
22: end while
23: return the best solution as candidate for optimum.
```

Table 4.2: *Parameters used in PPA experimental results, [6]*

Problem size	Pop. size	Max. gen[a]	Max. runner[b]	Short runners	Long runners
14 to 51	40	100	10	10 (each a 2-opt)	1 (3 seq. 2-opts)
51 to 101	40	100	10	10 (each a 2-opt)	1 (4 seq. 2-opts)
101 to 666	100	100	10	10 (each a 2-opt)	1 (6 seq. 2-opts)

[a] Maximum number of generations
[b] Maximum number of runners a plant can produce
seq.= sequential

effect of using 2-PSA to generate a part of the initial population. Therefore, we solved a set of problems with the discrete PPA and the discrete PPA+ 2-PSA, seperately and compared their results. Each of these is discussed below.

4.5.1 PPA versus GA and SiA

The discrete implementation of PPA has been applied to 10 TSP instances ranging from 14 to 101 cities, [4]. These well known problem have also been solved with GA and SiA elsewhere, in particular, in [96, 111] The key parameters used in our experiments can be found in Table 4.2, below. For algorithms GA and SiA, the relevant parameter values can be found in Table 4.4 and Table 4.5. Note that these particular values have been chosen through experimentation.

Two termination criteria have been used: the first is the maximum number of generations set to 100, and the second is the number of iterations without any change in the best tour found so far, which is set to 10. In these experiments, PPA outperformed both GA and SiA on all instances. All algorithms have been coded in Matlab and each algorithm was run 5 times. Results of the comparison with GA and SiA have been recorded in Table 4.3.

Table 4.3: *Comparison of GA, SiA and, PPA on standard TSP Instances, [4, 6]*

		GA		SiA		Discrete PPA	
Problem	Optimum	Av. Dv.(%)	Av. Time(s)	Av. Dv.(%)	Av. Time(s)	Av. Dv.(%)	Av. Time(s)
burma14	30.8785	0.7	6.95	0.53	8.34	0	1.55
ulysses16	73.9876	0.27	8.26	0.17	42.28	0	2.71
ulysses22	75.3097	1.56	9.83	1.16	97.12	0	4.02
att48	33524	4.97	41.23	31.48	10.52	0.6	5.71
eil51	426	4.46	44.45	18.17	423.26	1.54	5.12
berlin52	7542	8.67	42.99	36.37	11.44	2.1	7.87
st70	675	11.62	66.17	24.89	232.21	1.66	9.23
eil76	538	6.84	74.9	33.34	1162.02	4.1	9.42
pr76	108159	6.25	94.02	35.91	254.2	1.2	10.26
eil101	629	10.37	143.99	50.11	220.71	4.29	14.37

Table 4.4: *Parameters of Genetic Algorithm*

Parameter	**Value**
Population size	50
Maximum number of generations	20
The rate of crossover	0.95
The rate of mutation	0.075
The length of the chromosome	50×8-bits
The number of points of crossover	2

Note that although the results are very encouraging, further testing on larger instances and comparison with other algorithms are needed to draw useful conclusions on performance. The really interesting aspects of PPA which have not been discussed yet are: (a) it is very simple to understand; (b) it involves less parameters than GA and SiA, for instance, [6].

Claim (a), above, is justified if we note that the algorithm is based on a universal principle which is “to stay in a favourable spot" (exploitation) and “to run away from an unfavourable spot" (exploration). That is all that the algorithm implements really. But, that is all a global search algorithm requires too. Claim (b) is equally easy to justify. Let us consider the list of parameters that are arbitrarily set in GA.

Table 4.5: *Parameters of Simulated Annealing*

Parameter	**Value**
Maximum temperature	20
Minimum temperature	1
$\alpha\%$	0.5%
P	5
Number of iterations at each temperature	20
Temperature set	[20,10,5,3,1]

1. The population size;
2. The maximum number of generations;
3. The number of generations without improvement to stop;
4. The rate of crossover;
5. The rate of mutation;
6. The length of the chromosome;
7. The number of points of crossover.

Now, compare the above list to that of PPA. We can start the algorithm with a single plant that will then produce more plants unlike GA where a population with more than one individual is required. The number of runners can be decided by the objective value of each plant; indeed in Nature, some plants may have no runners at all because they are in desperate conditions while other may have 1, 2 or more depending on where they are and the corresponding value they give to the objective function. If we accept this, then PPA requires no more than a mechanism to stop, i.e a stopping criterion. Hence, the comparatively short list of its parameters below.

1. The maximum number of generations;
2. The number of generations without improvement to stop;

3. The maximum number of runners any plant can have.

For ease of implementation more parameters are used.

Simulated Annealing is not as extravagant as GA when it comes to arbitrary parameters. However, it still requires at least five parameters.

1. The maximum temperature;
2. The minimum temperature;
3. Parameter α: Percentage improvement in the objective value expected in each move;
4. The maximum number of moves without achieving α% of objective function value improvement (P);
5. Number of iterations at each temperature.

   To this, one can add the temperature set as in the last row of Table 4.5.

It is, therefore, fair to say that PPA compares well against GA and SiA even if only small instances of TSP have been considered. Note that the proliferation of arbitrary parameters makes the concerned algorithms less usable since it is difficult to find good default parameters when the list is long. More arbitrary parameters also mean more uncertainty. Based on this comparison approach, it is fair too to say that PPA will match most heuristics and hyper-heuristics. Further work may be to design a more realistic algorithm comparison methodology which not only takes into account raw performance, but also what is required in terms of parameter setting.

4.5.2 PPA versus Modified PSO

A second set of experiments has been conducted and the results compared to those obtained by the modified PSO, [112]. There are four versions of PSO was studied. All algorithms have been applied to four TSP instances with 14 to 76 cities, [4]. Each algorithm was run 10 times for each problem. The results of the comparison can be found in Table 4.6. The parameters values used in the PPA experiments can be found in Table 4.2. Those of PSO can be found in Table 4.7.

Table 4.6: *Comparison of modified PSO and PPA on standard TSP Instances, [4]*

	PSO-TS	PSO-TS-2opt	PSO-TS-CO	PSO-TS-CO-2opt	Discrete PPA
Problem	Av. Dv.(%)	Av. Dv.(%)	Av. Dv.(%)	Av. Dv.(%)	Av. Dv.(%)
burma14	9.12	0	10	0	0
eil51	35.47	6.81	16.34	2.54	1.84
eil76	9.98	5.46	12.86	4.75	3.76
berlin52	7.37	5.22	10.33	2.12	1.84

PSO-TS : The PSO based on Space Transformation
PSO-TS-2opt: PSO-TS combined with 2-opt local search
PSO-TS-CO: PSO-TS with chaotic operations
PSO-TS-CO-2opt: PSO-TS combined with CO and 2-opt.

In these experiments, PPA outperformed all modified PSO algorithms on all instances in terms of solution quality. Arbitrarily set parameters for modified PSO are as listed below, [112]:

1. The number of particles;
2. The value of $V_{\max}$;
3. The values for learning factors, c1 and c2;

Table 4.7: *Parameters of Modified PSO*

Parameter	**Value**
The number of particles	50
The value of $V_{\max}$	0.1
The values for learning factors, c1 and c2	c1=c1=2
The inertia coefficient	1
$P_{\max}$	1
Local search probability	0.01
Disipitive Probability	0.001
The maximum number of generations	2000

4. The inertia coefficient;
5. A positive real number $P_{\max}$, to express the range of the activities for each particle;
6. Local search probability;
7. Disipitive Probability;
8. The maximum number of generations;
9. The number of generations without improvement to stop.

Figure 4.5 depicts the tours found in generations 1, 3, 5 and 8 of PPA when applied to a 22-city instance [4]. The last figure shows the optimal tour. Figures 4.6-4.9 show the evolution curve diagrams of the algorithm on four TSP instances with the number of cities ranging from 48 to 225.

4.5.3 PPA versus New DFA

Another set of experiments has been conducted and the results compared to those obtained with the Discrete Firefly Algorithm (New DFA) described in [45,70]. Both algorithms have been applied to 7 TSP instances with 16 to 666 cities, [4]. The parameter values used in PPA experiments can be found in Table 4.2. For New DFA, they can be found in Table 4.9. Note

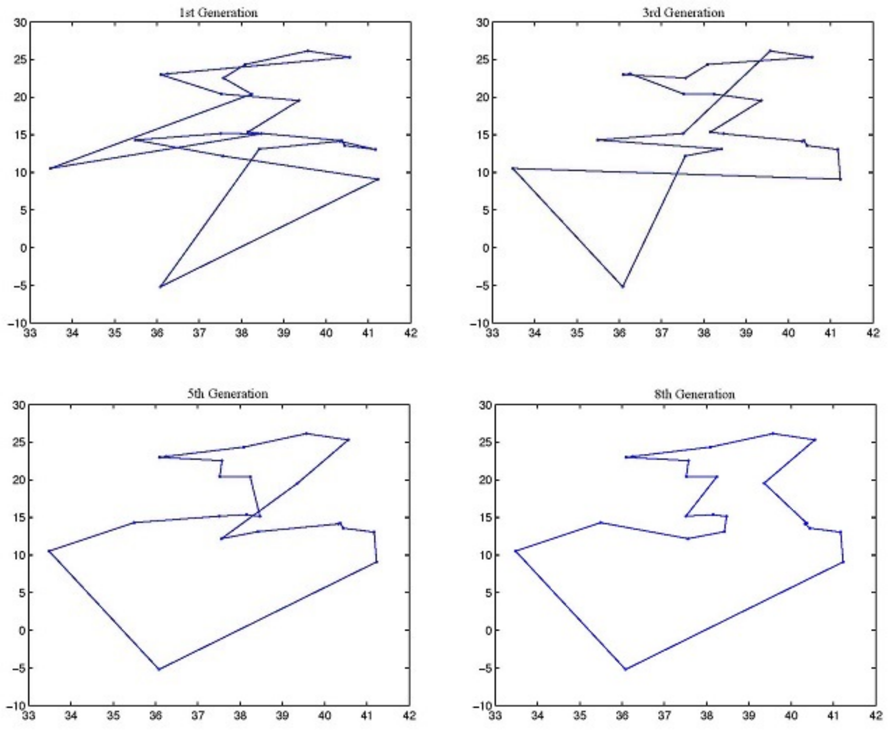

Figure 4.5: *A 22-city problem-* 1st, 3rd, 5th *and the* 8th *Generations*

that the average accuracy was calculated using the equation below.

$$Avg.\ accuracy = \frac{Best\ known\ solution}{Avg.\ solution\ found} \times 100. \tag{4.2}$$

Each algorithm was run 50 times. The results of the comparison can be found in Table 4.8. In these experiments, in terms of solution quality, PPA outperformed New DFA on 3 out of 7 instances. On the remaining 4 instances both algorithms have found the optimum solution. Arbitrarily set parameters required by New DFA are as listed in Table 4.9 with the values used, [45].

4.5.4 Discrete PPA on large TSP instances

Here, we demonstrated the performance of PPA on large TSP instances by applying PPA on its own and the combination of PPA and 2-PSA. In the implementation of PPA on its own,

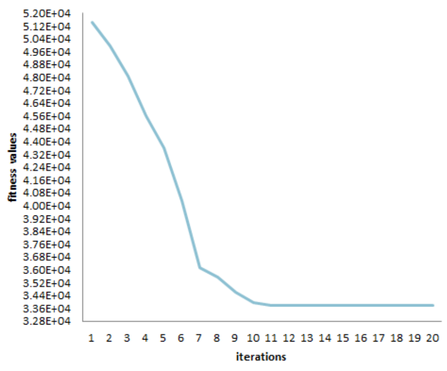

Figure 4.6: *Evolution curve diagram of att48*

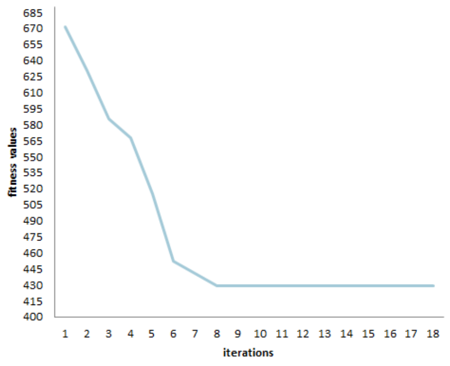

Figure 4.7: *Evolution curve diagram of eil51*

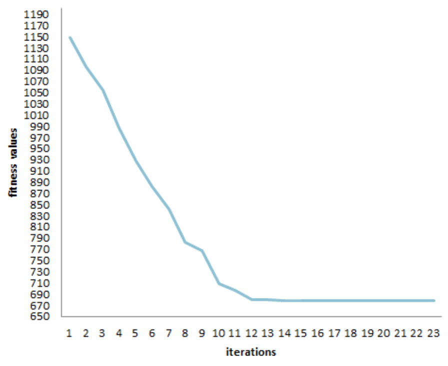

Figure 4.8: *Evolution curve diagram of st70*

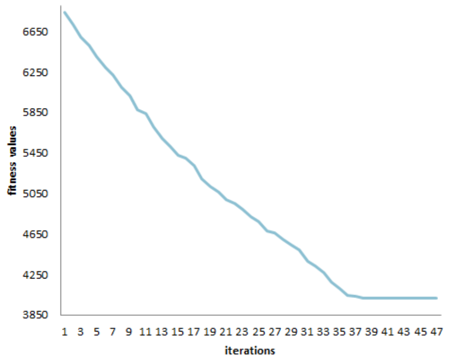

Figure 4.9: *Evolution curve diagram of tsp225*

Table 4.8: *PPA versus New DFA on standard TSP Instances, [4]*

		New DFA	PPA
Problem	Optimum	Av. Acc.(%)	Av. Acc.(%)
ulysses16	73.9876	100	100
ulysses22	75.3097	100	100
gr202	549.99	100	100
tsp225	3845	88.332	94.242
a280	2578	88.297	93.392
pcb442	50778	88.505	93.985
gr666	3952.53	100	100

Table 4.9: *Parameters of New DFA*

Parameter	**Value**
The number of maximum generations	Between 100 to 500 generations
The population size	5
The light absorption coefficient	0.001
The updating index	Between 1 and 16

$\frac{1}{4}$ of the initial population was generated using the greedy algorithm and the rest of the population was generated using random permutation. In the combined version, $\frac{1}{4}$ of the initial population was generated using the greedy algorithm, second $\frac{1}{4}$ of it was generated using 2-PSA and the rest of it was generated using random permutation.

Both algorithms were applied to 7 TSP instances ranging from 280 to 5915 cities, [4]. The comparison in terms of total computation time and average errors is recorded in Table 4.10. For the instances ranging from 280 to 666 cities, each algorithm was run 5 times with the population size of 40. The number of short and long runners was set to 10 and 6, respectively. For the instances ranging from 1304 to 3038 cities, each algorithm was run 3 times with the population size of 100. The number of short and long runners was set to 30 and 20, respectively. For the instance with 5915 cities, each algorithm was run once with the population size of 200; the number of short and long runners was set to 30 and

50, respectively. These parameters were set after a number of experiments. As instance size increases, PPA+2-PSA outperforms PPA in terms of execution time. This shows that starting with a good population is important.

Table 4.10: *PPA versus PPA+2-PSA on large TSP instances, [4]*

		PPA		PPA+2-PSA	
Problem	Optimum	Av. Dv.(%)	Av. Time(s)	Av. Dv.(%)	Av. Time(s)
a280	2578	6.71	42.4	7.11	42.08
pcb442	50778	7.41	86.35	6.12	80.92
gr666	3952.53	0	291.45	0	273.237
rl1304	252948	7.02	1129.37	7.69	1011.77
rl1889	316536	7.68	1921	7.76	1772.1
pcb3038	137694	9.02	4906.27	8.44	2308.9
rl5915	565530	8.1	7.45(h)	9.6	4.16(h)

h: hour

4.6 Summary

In this chapter, we showed how to implement PPA to solve TSP. We addressed the issues of defining distance and neighborhood. We used the Euclidean distance to compute the tour length. The distance between plants is the number of changes in the permutation, i.e tours. Short runners are generated using the $2-opt$ rule and long runners are generated using the $k-opt$ rule, with $k > 2$. We used a variant of the Strip Algorithm, to generate some individuals of the initial population. We also used the greedy approach and, the random generation to provide diversity. One of the advantages of PPA is that it has relatively fewer parameters compared to other heuristics such as GA, SiA and PSO. It was demonstrated that, PPA combined with the variant of the Strip Algorithm, 2-PSA which generates good initial populations, is an effective way to handle large TSP instances in terms of computation time. Experiments conducted to demonstrate the performance of the proposed algorithm, produced results which are very encouraging and overall in favour of PPA.

CHAPTER 5

THE PLANT PROPAGATION ALGORITHM FOR THE KNAPSACK PROBLEM

5.1 Introduction

The second implementation of PPA is to solve Knapsack Problems (KP). They are a class of well-known combinatorial optimization problems. They are widely studied and encountered in real-life. They belong to the class of *NP-hard* problems, [113].

KP can be defined as the problem of assigning items to knapsacks in a way that would maximize the total value of items put in each knapsack while not exceeding their predetermined capacities. Here, the weight and the value per item are given as input data of the problem. We intend to establish that PPA can also work on Knapsack Problems.

5.2 The Knapsack Problem: A Brief Review

In a KP, let the total number of items be n and the total number of knapsacks be m. For each item j, its weight and profit are given in the problem and denoted by w_j and p_j, respectively. For each knapsack i, given capacity values are represented by c_i, [87].

Sahni [114], has shown approximate algorithms to solve (0-1) KP and in his work each algorithm guarantees a certain minimal closeness to the optimal solution value. Gherboudj et al. [115] have proposed a discrete Binary Cuckoo Search (BCS) algorithm to solve the (0-1) Knapsack and the Multidimensional Knapsack Problems. They have used a sigmoid function to generate binary solutions. They have compared the performance of discrete BCS with other algorithms. The experimental results show that in the most cases BCS gives results close to the optimal.

Shao et al. [116], have developed the greedy genetic algorithm by hybridising the greedy algorithm with GA. The new approach was used to solve the (0-1) KP to demonstrate the algorithm feasibility and viability.

Zou et al. [117], have proposed a novel global Harmony Search Algorithm(NGHS) to solve the (0-1) KP. Their algorithm has two key operations. The first one is position updating. It enables the worst harmony of the harmony memory to move to the global best harmony in a fast way, in each iteration. The second one is a genetic mutation with a small probability that helps NGHS to run away from the local optimum. Computational experiments show that NGHS can be an efficient alternative for solving the (0-1) KP.

Pulikanti et al. [118], have proposed a new hybrid approach combining artificial bee colony algorithm with a greedy heuristic and a local search for the quadratic knapsack problem. The quadratic knapsack problem belongs to the knapsack problem family and it is an extension of the well-known (0-1) KP. In this problem profits are associated with pairs of objects as well as with individual objects. As this problem is an extension of the (0-1) KP, it is also NP-Hard. The performance of the algorithm on standard quadratic knapsack problem instances is compared with other heuristic techniques. The results obtained show that the proposed algorithm is superior to the techniques in many aspects.

Hristakeva et al. [119], have presented a comparative study of brute force, dynamic programming, memory functions, branch and bound, greedy, and genetic algorithms. They discussed the complexity of each algorithm in terms of time and memory requirements, and in terms of required programming efforts. The results show that the most promising approaches are dynamic programming and genetic algorithms. The paper examines in more details the specifics and limitations of these two paradigms.

Sundar et al. [120], presented an Artificial Bee Colony (ABC) algorithm for the (0-1) Multidimensional Knapsack Problem, (0-1) MKP. The objective of (0-1) MKP is to find a subset of a given set of n objects in such a way that the total value of the objects included in the subset is maximized, while a set of knapsack constraints remains satisfied. Computational results demonstrate that the ABC algorithm not only produces better results but converges very rapidly in comparison with other swarm-based approaches.

5.2.1 The 0-1 Knapsack Problem

The (0-1) KP is the classical, well-known KP. There are two variants of this problem: single and multiple KP. Given a set of n items and a set of m knapsacks $(m \leq n)$; the general form of the (0-1) KP is as follows [87].

$$max\, z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij}$$

$$\begin{aligned} s.t. \sum_{j=1}^{n} w_j x_{ij} \leq c_i,\ i \in M = \{1, \dots, m\}, \\ \sum_{i=1}^{m} x_{ij} \leq 1,\ j \in N = \{1, \dots, n\}, \\ x_{ij} = 0\ or\ 1,\ i \in M,\ j \in N, \end{aligned} \tag{5.1}$$

where, $\{x_1, x_2, \dots, x_n\}$ denotes the set of n items. $x_{ij} = 1$ if item j is in the knapsack i and 0, otherwise. If $m = 1$, the problem is called (0-1) Single Knapsack Problem, otherwise, if $m > 1$, then it is called the (0-1) Multiple Knapsack Problem, or (0-1) MKP, [113].

5.2.2 The Bounded Knapsack Problem

This variant of the KP allows choosing the same item x_j multiple times. An upper limit u_j denotes the maximum number of item x_j that can be selected. The mathematical formulation of the problem is as follows [113].

$$max\, z = \sum_{j=1}^{n} p_j x_j$$

$$\begin{aligned} s.t. \sum_{j=1}^{n} w_j x_j \leq c, \\ 0 \leqslant x_j \leqslant u_j \text{ and integer}, j \in N. \end{aligned} \tag{5.2}$$

5.2.3 The Unbounded Knapsack Problem

The knapsack problem is said to be unbounded when there is no upper limit on an item to be selected. The problem can be formulated as follows [121].

$$\begin{aligned} max\ z &= \sum_{j=1}^{n} p_j x_j \\ s.t. \sum_{j=1}^{n} w_j x_j &\leq c, \\ 0 \leqslant x_j,\ j \in N, and\ x_j &\in \mathbb{Z}. \end{aligned} \tag{5.3}$$

5.2.4 The Multidimensional Knapsack Problem

The multidimensional KP has more than one aspect that should be considered while assigning items to knapsacks. For instance, if w_{1j} denotes the weight of each item, w_{2j} denotes the volume of each item [121]. These constraints are defined in the mathematical programming formulation of the (0-1) multidimensional knapsack problem as follows.

$$\begin{aligned} max\ z &= \sum_{j=1}^{n} p_j x_j \\ s.t. \sum_{j=1}^{n} w_{1j} x_j &\leq c_1, \\ \sum_{j=1}^{n} w_{2j} x_j &\leq c_2, \\ x_j &\in \{0,1\},\ j \in N. \end{aligned} \tag{5.4}$$

5.2.5 The Quadratic Knapsack Problem (QKP)

This variant of KP considers an additional value an item will have if it is transported with another particular item. The mathematical model of QKP for a single knapsack is as follows [122].

$$max\ z = \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}x_ix_j$$

$$\begin{aligned} s.t. \sum_{j=1}^{n} w_jx_j &\leq c, \\ x_j \in \{0,1\},\ j &\in N. \end{aligned} \tag{5.5}$$

There are other variants, such as the Multiple-choice Knapsack Problems, the Set-Union Knapsack Problems and others.

5.3 Implementation of PPA to solve the Knapsack Problem

In the previous section, variants of the KP are discussed. In this section, PPA is implemented to solve the KP. As discussed in the previous chapter, representation of individuals/solutions is a key aspect of their implementation. The issue here is the representation of a plant which itself represents a solution. A solution here is the set of items placed in one or many knapsacks in a way that would maximise the total value of the knapsack or knapsacks.

5.3.1 The Representation of Knapsacks

A plant in the population of plants maintained by PPA is a solution represented as a set of 0s and 1s that depicts whether an item is in a knapsack or not. X_i is set $i, i = 1, \ldots, NP$. Here, NP denotes the size of the population. Plants are ranked according to their total values. Total value of a plant i is denoted by Z_i; it is a function of X_i. Here, while maximising the total value of a knapsack or multiple knapsacks, the given capacity constraints should not be violated. Recall that the mathematical programming formulation of the (0-1) KP is given by Equation (5.1).

For the single KP, the representation of items in a knapsack is an array of 0s and 1s. To illustrate, $X_i = [0\ 1\ 0\ 0\ 1\ 1]$ represents in where $n = 6$ and items x_2, x_5 and x_6 are in the knapsack. For a multiple knapsack problem, a matrix representation is used where each row represents a knapsack. A sample set of 3 knapsacks with 5 items in total can be shown as :

$$X_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad i = 1, \ldots, 3, \quad j = 1, \ldots, 5. \tag{5.6}$$

Here, items x_{11} and x_{15} are in the first knapsack, items x_{22} and x_{24} are in the second bag and items x_{33} is in the third knapsack. For this variant, it is important not to violate the capacity constraint for each knapsack while maximising the total value.

The initial population is generated by the Roulette-wheel approach, as discussed in Chapter 3. For each item a probability is assigned based on their $\frac{p_j}{w_j}$ ratio. For multiple knapsack problems, the procedure starts with assigning items to the knapsack with the smallest capacity [87]. Some of the sets are generated by using random generation to ensure diversity in the population.

5.3.2 The Distance Between Two Plants

The distance between two plants is defined as the number of exchanges to transform one plant/solution into another. After sorting sets by their total values, a pre-determined number of them is taken amongst the ones that have maximum values; short runners are then sent from these plants, i.e. new neighbouring sets are generated from them. For the rest of the set, long runners are sent to explore for better solutions. For this purpose, mutation is used. For the single knapsack problem, the change is applied to the vector and for the multiple KP, the mutation is applied to the matrix.

5.3.3 Short and Long Runners

The mechanism of PPA works by sending short and long runners to exploit and explore the search space to find better solutions. After generating the initial population, a predetermined number of the best solutions are taken, and short runners are sent from them. Three different approaches are used to implement short and long runners.

5.3.3.1 The Hamming Distance Approach (PPA1)

The Hamming distance is the total number of point changes between two solutions [123]. For short runners, the 2-point Hamming distance is used since it requires the minimum number of changes to create a new solution. For a 5-item single knapsack problem, the change from $0-1-0-0-1$ to $1-1-0-0-0$ demonstrates the 2-point change. If the solution is improved and the capacity constraint is not violated, the new plant is kept as a child plant; otherwise, it is tried for the second time and kept regardless of the satisfaction of the constraints. Long runners are implemented by using k-point mutation since it requires many changes from one plant to another to keep the distance far from the main plant. For long runners there is no control mechanism. All child plants are added up to the temporary population set.

For the MKP, the change is made in the matrix, i.e. the set of knapsacks. The number of short runners sent from each plant is determined depending on the quality of solutions. A maximum number of runners is set in the beginning. This guarantees to send many short runners from good plants whereas, sending only 1 or 2 long runners from plants with bad quality results.

Once this process is completed, solutions that exceed the capacity limits of knapsacks are eliminated. The rest of the solutions are sorted considering their solution quality and *NP* of them are kept for the next cycle.

5.3.3.2 Adding / Removing a Small Real Value (PPA2)

The binary vector or string can be considered as a binary representation of some value. It can be altered to implement short and long runners. In this approach, a small value ε is added into binary strings after they are converted into decimals. See Equation 5.7. Then the procedure is completed once new decimal values are converted back to binary values and considered as runners.

$$bin2dec(0100110)\ \pm\varepsilon\ =\ dec2bin(value) \tag{5.7}$$

5.3.3.3 Hamming Distance + Deep Search (PPA3)

In this approach, randomization is reduced compared to the first approach. Some items are chosen randomly and some depending on their weights and value, i.e. the cheapest element in the knapsack should be removed in case of overloading. In the case of short runners again two items are changed following the rule of searching. For long runners *k*-items are changed, this process can be done either randomly or by following the rule defined below.

Algorithm 17 Hamming distance+ Deep search Procedure

```
1: Generate the first set of items to be changed randomly;
2:        If Available space < 0 Then
3:            Remove the item with the least value and check whether
              the conditions are satisfied;
4:        Else
5:            Add the most profitable item available.
6:        End If
7: Return New solutions
```

5.3.4 Termination of the Algorithm

There are two ways to terminate the algorithms. The first one is by setting a maximum number of generations. When the program reaches this number, it stops. The other one is keeping a list of best solutions found. If the best solution found repeats itself for a predetermined number of times, the program terminates and outputs that solution as the optimum.

Algorithm 18 Pseudo-code of Discrete PPA for KP

```
1: Generate a population $P = \{X_i,\ i = 1, \ldots, NP\}$ of valid solutions by RW approach;
2: Choose values for $g_{\max}$, $maxrunner$ and $y$;
3: g = 1
4: while g < $g_{\max}$ do
5:     Compute $N_i = f(X_i), \forall\ X_i\ \in\ P$;
6:     Sort $N = \{N_i,\ i = 1, \ldots, NP\}$ in descending order (for maximisation);
7:     Normalise $N_i$ and assign $N_i \times maxrunner$ runners to plant $i$;
8:     if Number of runners > $y$ then
9:         Generate short runners and add to the new population $\Phi$;
10:    else
11:        Generate long runners and add to the new population $\Phi$;
12:    end if
13: end while
14: return Best solution as candidate for optimum.
```

Table 5.1: *PPA versus BCSA and NgHS*

Test	Size	Optimum	BCSA	NgHS	PPA1
f1	10	295	295	295	295
f2	20	1024	1024	1024	1024
f3	4	35	35	35	35
f4	4	23	23	23	23
f5	15	481.0694	481.0694	481.0694	481.0694
f6	10	52	52	50	52
f7	7	107	107	107	107
f8	23	9767	9767	9767	9767
f9	5	130	130	130	130
f10	20	1025	1025	1025	1025

5.4 Computational Experiments and Results

The proposed algorithm has been implemented using various strategies as mentioned in the previous section. Five sets of experiments have been carried out. The first set compares PPA1 with Binary Cuckoo Search Algorithm (BCSA) [115], and the Novel Global Harmony Search Algorithm (NgHS) for solving the (0-1) single Knapsack Problem instances with item sizes range from 4 to 23, [117]. The first set of test results can be seen in Table 5.1. This table only shows that PPA1 is as robust as the other algorithms considered. We cannot say more since we only have access to the results of these algorithms.

In the second set of experiments the same problem instances were solved using all variants of PPA for the KP and compared to GA. Results can be seen in Table 5.2. These results show that PPA3 outperforms all algorithms by finding the optimum result in all runs. The size of population is set to 10 and the number of generations to 100 for all algorithms. For GA, the crossover and mutation rates are set to 0.5 and 0.1, respectively and 1-point cross-over is applied. For the variants of PPA for KP, the maximum number of runners is set to 4 and for short and long runners the neighborhood is defined as 2 and 6-element changes, respectively.

In the third set of experiments variants of PPA were compared with Binary ABC, Binary PSO and Improved Binary PSO , [117, 124, 125]. Here, the size of the problem instances is increased. PPA3 has outperformed all other algorithms in terms of solution quality, including PPA1 and PPA2, on both instances.

In the fourth set of experiments randomly generated large KP instances are solved with both PPA and GA. The size of population is set to 100 and the number of generations to 200 for all algorithms. For GA, the crossover and mutation rates are set to 0.5 and 0.1, respectively and 1-point cross-over is applied. For variants of PPA for KP, the maximum number of runners is set to 10 and for short and long runners the neighborhood is defined as 2 and 12-element changes, respectively. Results can be seen in Table 5.4.

In the last set of experiments PPA is compared with Binary Firefly Algorithm and Binary PSO, [125, 126]. For the first two instances all algorithms except BPSO find the optimum. PPA3 and BFA show similar performances. Results are shown in Table 5.5.

5.5 Summary

In this chapter we explained how PPA can be implemented to solve KP. First we gave a brief review of the topic of interest and reviewed different variants of KP. Second, we extended PPA to solve KP. We discussed the representation of solutions as a plant. For short and long runners, we used three different approaches and compared their effects on the performances of the algorithm. The one which converts binary solution representations to decimal

Table 5.2: *PPA variants versus GA*

			PPA1	Avg. Time	PPA2	Avg. Time	PPA3	Avg. Time	GA	Avg. Time
Test	Size	Optimum	Avg. Results		Avg. Results			Avg. Results		Avg. Results
f1	10	295	295	0.03762 (50/50)	295	0.06575(50/50)	295	0.01929(50/50)	294.8	0.01336 (38/50)
f2	20	1024	1024	0.04548 (50/50)	1023.3	0.2352 (46/50)	1024	0.01303 (50/50)	1023.88	0.06341(49/50)
f3	4	35	35	0.082 (50/50)	35	0.0121 (50/50)	35	0.0033 (50/50)	34.32	0.0094 (43/50)
f4	4	23	23	0.084 (50/50)	23	0.0095(50/50)	23	0.0042 (50/50)	22.96	0.0093 (48/50)
f5	15	481.0694	481.0694	0.0349 (50/50)	481.0694	0.07959 (50/50)	481.0694	0.01878(50/50)	481.0694	0.06786 (50/50)
f6	10	52	52	0.01839 (50/50)	50	0.0342 (50/50)	52	0.0144 (50/50)	51.64	0.0112 (41/50)
f7	7	107	107	0.0386 (50/50)	107	0.0886(50/50)	107	0.0112 (50/50)	106.84	0.0585 (46/50)
f8	23	9767	9767	0.4269 (46/50)	9766.1	3.128(23/50)	9767	0.9171(50/50)	9766.9	0.1124 (49/50)
f9	5	130	130	0.01285 (50/50)	130	0.0329 (50/50)	130	0.0066 (50/50)	130	0.017 (50/50)
f10	20	1025	1025	0.03956 (50/50)	1024.8	0.53 (46/50)	1025	0.02354 (50/50)	1025	0.0617 (50/50)

Table 5.3: *PPA versus BABC, BPSO and IBSO*

Test	Size	Opt.	PPA1	PPA2	PPA3	BABC	BPSO	IBPSO
f12	50	3103	3091	2953	3103	3087.5	3093	3078
f13	80	5183	5181.5	N/A	5183	5147.7	5132.6	N/A

Table 5.4: *PPA versus GA*

Test	Size	PPA1	PPA2	PPA3	GA
f14	500	22637	N/A	22697	19014
f15	1000	48452	N/A	48700	35446
f16	1500	76290	N/A	76649	53102

Table 5.5: *PPA versus BFA and BPSO*

Test	Size	Opt.	PPA1	PPA2	PPA3	BFA	BPSO
f17	8	3924400	3924400	3924400	3924400	3924400	3921857.19
f18	12	5688887	5688887	5688887	5688887	5688887	5683694.29
f19	20	10727049	10712735.38	10703865.24	10724854.4	10727049	10707360.9
f20	24	12233713	12200679	12144380.48	12211026.56	12208229.7	12205346.2
f21	16	9352998	9342627.22	9351524.64	9352998	9352998	9334408.62

and vice versa has the disadvantage of implementation as Matlab cannot handle binary numbers larger than 52 digits. Further tests were carried out to measure the performance of the proposed algorithm. We used 21 different problem instances and compared the results of the variants of PPA with that of some well-known metaheuristic methods found in the literature. Large instances were generated randomly. Overall PPA3 performed better than other variants.

CHAPTER 6

THE PLANT PROPAGATION ALGORITHM FOR SCHEDULING PROBLEMS

6.1 Introduction

Scheduling is a vast area of applications and research. It spans timetabling in all its variety (course, exam, tournament . . .) and other scheduling problems such as planning and sequencing. Here we consider a practical and representative scheduling problem which is Berth Allocation in container ports. Moreover, we consider a practical version which handles uncertainty, hence the more appropriate reference to the Robust Berth Allocation Problem, or RBAP.

Container terminal problems are complex real-life problems. They are varied and numerous. The Berth Allocation Problem (BAP) is just one of them. Others include Quay Crane Assignment Problem, Quay Crane Scheduling, Yard Crane Scheduling and Workforce Planning Problem [127], to name a few. BAP is widely studied in the literature, [127–131]. The aim of the problem is assigning the best berthing time and berthing position for each ship that arrives at the port, to minimise the total time spent at the wharf, [132]. This chapter is concerned with implementing PPA to solve RBAP, [133].

6.1.1 The Berth Allocation Problem

It is the problem of finding the most appropriate docking place and berthing time that optimize a given objective function, [132]. Imai et. al have classified BAP in terms of wharf utilization and berthing time [128, 129]. Wharf configuration is divided into three main categories: Discrete, continuous, and hybrid types [129]. In the discrete setup, the wharf is split into berths and vessels are placed in these specified areas. The continuous setup allows each vessel to dock at any point on the wharf as long as the wharf length is not exceeded. In the hybrid configuration, the wharf is partitioned into berth spaces for vessels; however, it is more flexible than the discrete setup type, i.e. a vessel is allowed to moor and take more space than is needed for one vessel, [129, 132]. Berthing times are divided into 2 categories: Static and dynamic arrivals, [128]. If arrivals are static, it is assumed that all ships are ready to be moored immediately, i.e. there are no fixed arrival times. On the other hand, if arrivals are dynamic, each ship has a particular arrival time which directly affects their berthing times.

6.1.2 Robust Berth Allocation Problem

In real life problems, it is not realistic to expect to have fixed arrival and handling times. As uncertainty has a high impact on real life, schedules that are more flexible considering real life situations, rather than rigid may be preferable. In [134], Xu et. al studied RBAP. In their study, they have proposed a robust model by adding buffer time between two

Figure 6.1: *Port of Felixstowe-Berths 8 and 9, [7]*

successive vessels to overcome uncertainties regarding arrival and handling times. The objective function of their model targets is to minimise total tardiness. A recent study of RBAP by Alsoufi et. al, [133] considers optimal berth position as well as tardiness. In addition to total tardiness, their objective function also gauges the total distance between the vessels current positions and their desired positions. This is an important aspect of container port problems as it impacts on yard management. In this model, the continuous wharf is used.

6.1.2.1 The Mathematical Model of RBAP

The problem is represented as a mixed integer programming model, [132, 133]. It aims at generating robust plans by mitigating uncertainty. This is rendered by finding the optimal time for starting the process, the optimal position a vessel is allocated by considering their desired position, the time buffer needed between a vessel and its successor, etc.

Let $V = [v_1\, v_2\, \ldots, v_n]$ denote a set of vessels. If a segment of quay is occupied, no other ships are allowed to be positioned there. It is assumed that there is a safety distance between a vessel and its successor. A vessel cannot leave before its processing has finished. A ship can be allocated to any point on the continuous wharf considering its arrival time and the available space in wharf.

The parameters of the model are listed below:

- W: the length of wharf
- A_v: arrival time of vessel v
- d_v: requested departure time of a vessel v
- h_v: handling time of vessel v
- L_v: Length of vessel v
- b_v: desired point for vessel v to be allocated
- $c1_v$: tardiness cost of vessel v
- $c2_v$: distance cost of vessel v if it is not placed to its desired point
- IN_v: Insatiability in arrival time of v
- PP_v: proportion of the processing time of vessel v over total processing time
- $R_v : IN_v + PP_v$
- λ_v: time buffer value

- M: a large number (positive)

There are eight decision variables; four of which are binary,

$$\delta_{v_iv_j} = \begin{cases} 1, & \text{if the processing time of } v_j \text{ starts later than finishing time of } v_i \\ 0, & \text{otherwise} \end{cases}$$

$$\sigma_{v_iv_j} = \begin{cases} 1, & \text{if } v_i \text{ is located below } v_j \\ 0, & \text{otherwise} \end{cases}$$

$$\xi_{v_iv_j} = \begin{cases} 1, & \text{if } v_j \text{ occupies part of berthing location of } v_i \\ 0, & \text{otherwise} \end{cases}$$

$$\zeta_{v_iv_j} = \begin{cases} 1, & \text{if } v_j \text{ starts later than the finishing time of } v_i \text{ and occupies part of its space} \\ 0, & \text{otherwise} \end{cases}$$

and the rest, T_{v_i}, θ_{v_i}, P_{v_i} and $\tau_{v_iv_j}$ are continuous. T_{v_i} denotes berthing time of v_i and P_{v_i} denotes berthing position of v_i. Time buffer is indicated by $\tau_{v_iv_j}$ and minimum buffer time between a vessel and other vessels is denoted by θ_{v_i}. The mathematical model of the problem is as follows, [133]. The objective function aims at minimising total cost of tardiness and position distance.

$$min \sum_{v=1}^{V} C_{1v}(T_v + h_v - d_v)^+ + \sum_{v=1}^{V} C_{2v}|P_v - b_v| \tag{6.1}$$

s.t

$$T_{v_i} + h_{v_i} + R_{v_i}\lambda_{v_i} - \theta_{v_i} \leq T_{v_j} + M(1 - \delta_{v_iv_j}) \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.2}$$

$$P_{v_i} + L_{v_i} \leq P_{v_j} + M(1 - \sigma_{v_iv_j}) \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.3}$$

$$\sigma_{v_iv_j} + \sigma_{v_jv_i} + \delta_{v_iv_j} + \delta_{v_jv_i} \geq 1 \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.4}$$

$$T_v \geq A_v \qquad \forall v \tag{6.5}$$

$$0 \leq P_v + L_v \leq W \qquad \forall v \tag{6.6}$$

$$\xi_{v_iv_j} = 1 - (\sigma_{v_iv_j} + \sigma_{v_jv_i}) \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.7}$$

$$\zeta_{v_iv_j} \geq \delta_{v_iv_j} + \xi_{v_iv_j} - 1 \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.8}$$

$$\tau_{v_iv_j} \leq M(1 - \zeta_{v_iv_j}) + T_{v_j} - T_{v_i} - h_{v_i} \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.9}$$

$$\theta_{v_i} \leq \tau_{v_iv_j} + M(1 - \zeta_{v_iv_j}) \qquad \forall v_i, v_j; v_i \neq v_j \tag{6.10}$$

$$\xi_{v_iv_j}, \zeta_{v_iv_j}, \sigma_{v_iv_j}, \delta_{v_iv_j} \in \{0, 1\} \tag{6.11}$$

$$P_{v_i}, T_{v_i}, \tau_{v_iv_j}, \theta_{v_i} \geq 0 \tag{6.12}$$

The first part of the function corresponds to the time factor and the second part corresponds to the position factor. Constraint (6.2) makes sure that if v_j starts after finishing time of v_i, its berthing time should be larger than completion time of v_i including the buffer time. If the current time between these two vessels are larger than buffer time, then it is ignored. Constraint (6.3) guarantees that v_j is positioned above v_i, if v_i is below v_j. Constraint

Table 6.1: *Representation of a schedule as a plant for three vessels*

4	12	17	0	5	1
T_1	T_2	T_3	P_1	P_2	P_3

(6.4) eliminates overlaps in the 2-d space. Constraint (6.5) guarantees that a vessel cannot start being processed before its arrival time. Constraint (6.6) guarantees that the total wharf length cannot be exceeded. Constraints (6.7-6.8) are on binary variables. Constraints (6.9-6.10) calculates the current buffer time and the minimum buffer time between vessels.

6.2 Solving RBAP with PPA

Implementing the PPA paradigm to solve RBAP requires a solution representation, the generation of good initial solutions (schedules). implementing the concepts of short and long runners and stopping. It is also necessary to have define a useful objective function.

6.2.1 Solution Representation

As discussed in the previous chapters, the representation of individuals/solutions is a key aspect of the implementation. The issue here is the representation of a plant which itself represents a solution. A solution here is a string that shows berthing times and positions of vessels. Table 6.1 illustrates this for the case of three vessels.

6.2.2 Generating Initial Population

It is important to start with a good set of initial solutions as mentioned in previous chapters. For this problem a number of plants has been generated using a three-step method comprising a deterministic step, semi-deterministic step and a random generation step.

- *Deterministic Step:* Vessels are set in an ascending order according to their arrival times. Berthing time for the earliest ship is set as its arrival time. All vessels are then checked and grouped if their arrival time is earlier than the earliest ship finishing time and the total length of ships does not exceed the wharf length. Vessels in the same group are placed along the wharf in a random order. Once the wharf is fully occupied the next earliest available vessel is placed and the same procedure is repeated until all vessels are docked. Berthing times are calculated in a way that would avoid overlapping without considering the robustness factor.

- *Semi-Deterministic Step:* Vessels are set in an ascending order according to their arrival times. The berthing time for the earliest ship is set as its arrival time. All vessels are then checked and grouped if their arrival time is earlier than the earliest ship finishing time and the total length of ships does not exceed the wharf length. The number of vessels and the order of the vessels that share the wharf to be docked are decided randomly. Once the determined number of ships are placed the next earliest available vessel is placed and the same procedure is repeated until all vessels are placed. Berthing times are calculated in a way that would avoid overlapping without considering robustness factor.

- *Random Step:* This step consists of two parts. All berthing times and positions are generated randomly considering arrival time and wharf length constraints. Half of these are added to the initial population after applying ESH3, 4 or 5. The rest is kept as random.

Once the initial population has been generated, handling times and finishing times are calculated for each vessel [133] as follows :

Handling time : $C_v = h_v + Rv_i \lambda_{v_i} - \theta_{v_i}$

Finishing time : $F_v = T_v + C_v$.

6.2.3 The Fitness Function Calculation

The fitness function introduced in [133] is used as a solution quality indicator for PPA. A penalty value is added to the normalised fitness value to find the final fitness value of a plant. Here, the penalty value is calculated by finding the overlapped area. This is achieved by multiplying two overlapped quantities in Cartesian time and place. In order to decide whether it is necessary to add a penalty value, each solution is checked against constraints (6.2), (6.3) and (6.4). The penalty value is denoted by $\gamma_{ij} = (A_{ij} \times B_{ij})^{0.5}$. Here, A_{ij} corresponds to the positional overlapped area and B_{ij} corresponds to the timewise overlapped area. These two terms are calculated as follows:

$$\begin{aligned} A_{ij} &= Max(\frac{L_i + L_j}{2} - |\frac{P_i + P_i + L_i}{2} - \frac{P_j + P_j + L_j}{2}|,\ 0), \\ B_{ij} &= Max(\frac{C_i + C_j}{2} - |\frac{T_i + F_i}{2} - \frac{T_j + F_j}{2}|,\ 0). \end{aligned}$$

The maximum feasible error is defined as $\gamma_{ij}^{max} = (A'_{ij} \times B'_{ij})^{0.5}$ where $A'_{ij} = \frac{L_i + L_j}{2}$ and $B'_{ij} = \frac{C_i + C_j}{2}$. The objective function Z of the model is considered in the fitness of individuals as $\frac{1}{Z}$. The ratio of $\frac{\sum \gamma_{ij}}{\sum \gamma_{ij}^{max}}$ is subtracted from the normalised objective function value Z^{-1} to find the fitness values for each plant. This is maximised. Although, mathematically speaking, Z when minimised can hit its lowest value, zero, which can cause numerical problems during the optimisation process, in practice this is unlikely to happen. This is because it is almost impossible to allocate every arriving vessel its ideal position in the wharf and also it is unlikely that all vessels will depart at exactly their expected departure time. This means that $Z > 0$ in practice. That is what we worked with. In the extreme cases, we can of course consider the normalised objective function values as $\frac{1}{\exp(Z)}$. So, there is no real issue in implementing and solving the model. The fitness values of plants are sorted in descending order and the pre-determined number of short or long runners are sent from each plant.

$$Fitness = Z^{-1} - \frac{\sum \gamma_{ij}}{\sum \gamma_{ij}^{max}}.$$

6.2.4 Short and Long Runners Implementations

The maximum number of runners is given at the beginning of the procedure. The number of runners assigned to each plant depends on their fitness value. This calculation starts with normalising all fitness values as follows.

$$Normalised\ fitness(i) = \frac{fitness_i - min(fitness)}{max(fitness) - min(fitness)},\ i = 1, \ldots, npop,$$

where *npop* denotes the size of the population.

The number of runners for each mother plant is calculated as follows:

$$Number\ of\ runners(i) = \lceil((max.\ number\ of\ runners - 1) * Normalised\ fitness(i))\rceil + 1,$$

where *npop* denotes the size of the population and $i = 1, \ldots, npop$.

6.2.4.1 Short Runners

There are two possible ways to determine the number of plants that sends short runners. One of them is taking $k\%$ of the solutions in the population, where k is a pre-determined value. The other option is taking plants, where $Number\ of\ runners\ (i) \geq n_r$ is satisfied, where n_r is a pre-determined value. Short runners are implemented in two steps:

1. The first part involves the investigation of overlapped vessels. As we are dealing with goodish solutions, we do not expect many overlapping ships in this subset of the population. If there are overlapping ships, to implement short runners, one of the overlapping pairs is selected randomly and one vessel is either moved slightly to the right/left or up/down without violating constraints (6.5) and (6.6). An example is illustrated in Figure 6.2. In Figure 6.3, Vessel 3 has been pushed upwards to eliminate overlapping.

2. If there is no overlapping, we use a list that shows the differences $T_v - A_v$ and $P_v - b_v$ for each vessel. A vessel that is changed is randomly selected from this list. If a randomly chosen point shows berthing time, then it is updated and pushed to the left to get as close as possible to the arrival time of the corresponding vessel. If a randomly chosen point shows a position, then this position is pushed as close as possible to its desired point. During this process, constraints (6.5) and (6.6) are verified for feasibility. An illustration can be seen in Figure 6.4. In the figure the randomly chosen Vessel 4 is pushed down to get close to its desired position.

6.2.4.2 Long Runners

For the rest of the population long runners are implemented. To do this, first all overlapping ships are listed. Then, depending on their berthing times and positions, ships are moved to left or right and pushed up or down considering constraints (6.5) and (6.6). A large

Algorithm 19 **Short Runners**

```
1:  $n_r \leftarrow$ The number of runners to be sent from the mother plant
2:  Separate the pre-determined number of plants in the population to send short runners.
3:    For Each mother plant
4:      For $i = 1 : n_r$
5:        If There are overlapping ships then
6:          Choose one ship randomly and move slightly to eliminate overlapping;
7:        else
8:          Make a list that shows the differences $T_v - A_v$ and $P_v - b_v$ for each vessel;
9:          Choose one of the starting times or the positions randomly from the list;
10:           If The randomly chosen point shows berthing time, then it is updated
            and pushed to the left to get as close as possible to the arrival time
            of the corresponding vessel then;
11:           else
12:             It shows the position. So, it is pushed as close as possible to
              its desired point;
13:           EndIf
14:       EndIf
15:     Keep the generated plant in a new list.
16:     EndFor
17:   EndFor
18: Return List of new plants.
```

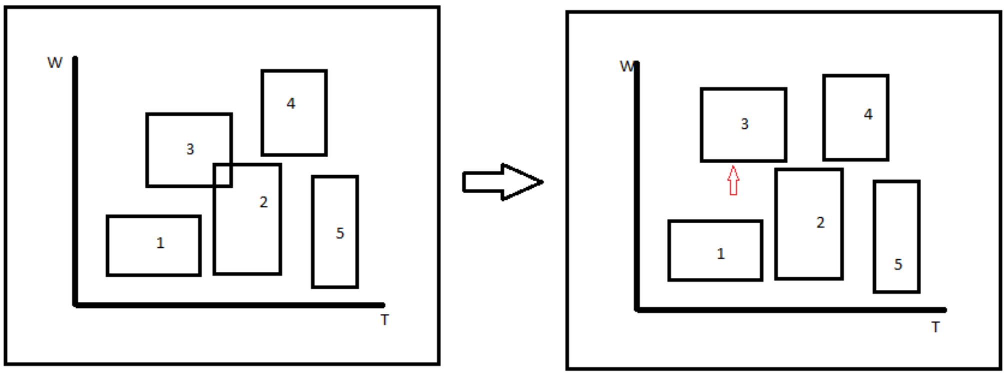

Figure 6.2: *Implementation of short runners in the case of overlapping*

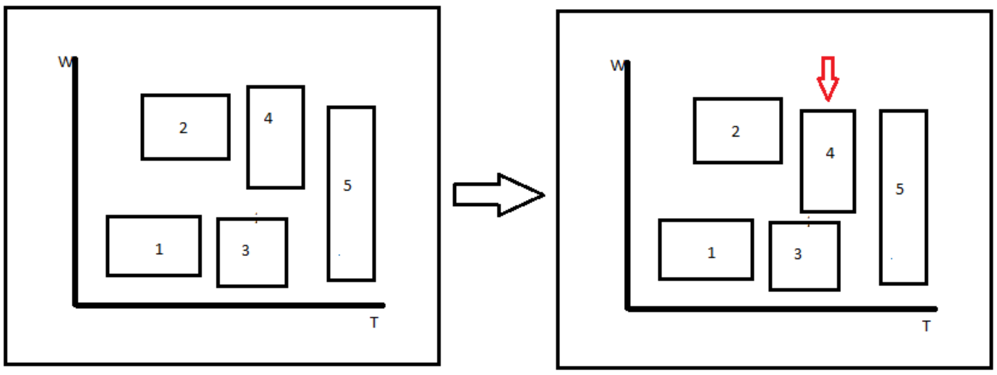

Figure 6.3: *Implementation of short runners when there is no overlapping*

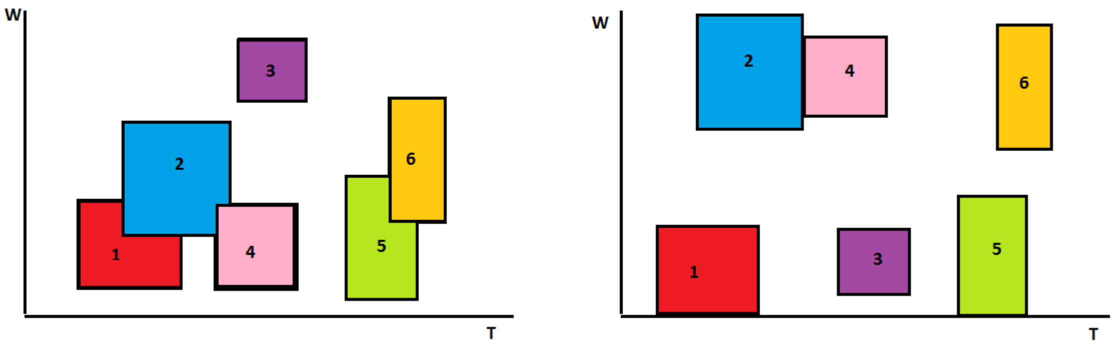

Figure 6.4: *Implementation of long runners when there is overlapping*

number of changes means that the new plant is far from the mother plant. Swapping is also used for vessels that are in the same horizon, period T. The procedure to implement long runners is explained below. A further investigation of implementing long runners is discussed in the next section. An illustration of an implementation of a long runner can be seen in Figure 6.4.

In Figure 6.3 overlapped pairs 1-2 and 5-6 are pushed to the bottom and to the top , for this example swapping is also implemented for vessels 3 and 4. Although the new layout does not have any overlaps when we do not consider the robustness factor, there may be new overlapping pairs once the long runner is generated. A sample schedule obtained by PPA for a 20-vessel problem can be seen in Figure 6.5.

Algorithm 20 Long Runners

```
1: $n_r \leftarrow$ The number of runners to be sent from the mother plant
2: Separate the pre-determined number of plants in the population to send long runners.
3:     For Each mother plant
4:         $n_r = 1$;
5:         Find the list of overlapping ships;
6:           If rand>0.5 & there are overlapping ships then
7:             First, all overlapping ships are moved in vertical space. The lower ship is
             pushed to level 0 and the upper one is moved to $W - Lv_{shipabove}$;
8:             Second, both ships are moved either to the left of right by $(T_v + 0.5 * r)$, where
             $T_v$ is the berthing time and $r$ is a random value between $[0, 1]$.
             Constraints (6.5) and (6.6) are considered;
9:           else
10:              If rand1>0.5 then
11:                $T_v - A_v$ and $P_v - b_v$ for each vessel are found as in the implementation of short
                runners. Each vessel that is not starting on its arrival time and/or not placed
                at their desired position is moved accordingly;
12:              else
13:                Swap either overlapping and/or non-overlapping situations by changing
                any two randomly selected vessels on the same vertical line;
14:              EndIf
15:           EndIf
16:      Keep the generated plant in a new list.
17:     EndFor
18: Return List of new plants.
```

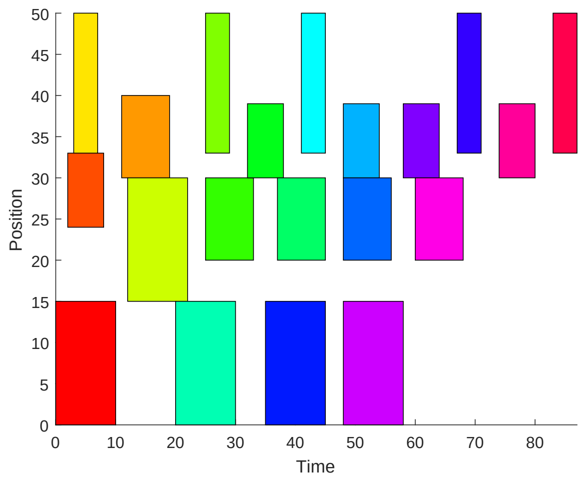

Figure 6.5: *A Sample Schedule (1) for a 20-vessel Problem*

6.2.5 Further Investigation of the Initial Population and Runners

A further study is carried out to find optional approaches to generate the initial population or to implement long runners. For this purpose, we use the Edge Seeking Heuristics discussed in Chapter 3.

6.3 PPA Application

In this section we will give the pseudo-code of PPA to solve RBAP.

Algorithm 21 Pseudo-code of Discrete PPA for RBAP

```
1: $npop \leftarrow$ population size, $gmax \leftarrow$ maximum num. of generations, $max_run \leftarrow$ maximum
   number of runners.
2: Generate a population $P = \{X_i,\ i = 1, \dots, npop\}$ of valid solutions using the three-step
   approach;
3: $g = 1$;
4: while $g <= g_{max}$ do
5:     Find fitness values and sort in descending order;
6:     Assign number of runners to be sent using $n_r = \lceil (max_{run} - 1) * n_{fit} \rceil + 1$;
7:     for $n_r > a$ do                                  ▷ $a$ is a pre-determined number.
8:         Apply short runners;
9:     end for
10:    Set $n_r = 1$ for the rest of the plants;
11:    Apply long runners;
12:    Keep all new plants in a temporary set $\Phi$;
13:    Compute fitness values and append to mother plants; $\Phi$;
14:    Keep top $npop$ of the solutions
15: end while
16: return The best solution as the candidate for optimum.
```

Table 6.2: *Example input data of a 5-vessel problem*

Arrival time	5	9	6	8	7
Estimated handling time	8	2	9	4	5
Expected departure time	12	10	14	11	11
Length of vessel	5	8	4	9	6
Desired position	5	3	7	1	0
Cost of tardiness	1	1	1	1	1
Distance cost	1	1	1	1	1
Instability in arrival	0.2	0.9	0.5	0.7	0.3
Proportion of processing time	0.28	0.07	0.23	0.14	0.17

6.4 Experimental Investigation

The experiments were carried out with problems of size ranging from 5 to 20 [1]. Example input data can be seen in Table 6.2.

Wharf length, W and a large number, M is also given.

Following the mathematical model given in Section 6.1.2, the total number of variables and the total number of constraints are found as follows:

Given $i = 1, \ldots, n$ and $j = 1, \ldots, m$, where m and n denote the number of vessels, in total there are 8 variables in the model. Five of them are in the form of matrices of $m \times n$ size and three of them are arrays of n size. These can be found in constraints (6.11) and (6.12). As $m = n$, and $i = j$ situations are not taken into account, total number of variables are $5n^2 - 5n + 3n$ or $5(n^2 - n) + 3n$. Similarly, for 12 constraints out of 17 (total number of constraints), $i = j$ situations are not taken into account. Therefore, there are total of $12(n^2 - n) + 5n$ constraints.

[1]All randomly generated problem sets were provided by Ghazwan Alsoufi in private communication.

Table 6.3: *PPA versus B&C and GA*

Size	B&C	GA		PPA	
		Best obj.	Avg. Obj.	Best obj.	Avg. Obj.
5	51	61	88	51	52.8
6	22	37	49	22	22
9	144	183	300	174	175.4
20	152	263	423	211	220.4
5	58	64	76	58	59.8
6	37	42	52	37	37
9	163	209	243	190	194
20	239	424	471	394	397.6

6.5 Experimental Results and Conclusion

Four instances of RBAPs have been solved using PPA. Experiments have been repeated twice for two different buffer time values. The results of the experiments are compared with those of Branch and Cut method and the Genetic Algorithm, [133].

In our experiments, the maximum number of runners is set to 10. For small instances population size is set to 100 and for large instances it is set to 200. Each problem was solved five times.

For the first four tests, λ is set to 5 and for the rest it is set to 10. Results show that PPA outperforms GA on all instances based on the average objective function value for both λ 5 and 10.

Another set of tests was run for a 6-vessel problem. The results of PPA were compared to those of the Branch and Cut algorithm. Results show that PPA has found the optimum for different values of λ at least 70% of all runs. Results for B&C are all optimum since it is an exact method. CPU times of PPA are given in Table 6.4. This time is expected to

Table 6.4: *PPA versus B&C for a 6-vessel Problem*

	B&C	PPA			
λ	Optimum	Best found	Avg. Result	CPU Time(s)	Success Rate
5	83	83	83	6.8	10/10
10	90	90	94.3	7.3	7/10
15	103	103	106.3	8.9	8/10
20	120	120	122.2	9.7	7/10

be around 1-2 minutes for B&C considering the results given in [133]. The success rate indicates how many times PPA achieved the optimum over 10 runs.

6.6 Summary

In this chapter we implemented PPA to solve RBAP one of the scheduling problems that arises in container ports. In order to generate the initial population of solutions/schedules, we used three approaches: deterministic, semi-deterministic and random generations. Strategies for short and long runners are discussed extensively. We used Edge Seeking Heuristics or ESH methods embedded in a random generation step to improve the solution quality of half of the randomly generated initial population.

CHAPTER 7

CONCLUSION AND FURTHER RESEARCH

7.1 Summary

This thesis is mainly concerned with the extension of the Plant Propagation Algorithm to handle discrete optimisation problems. Considering the importance of having a good set of population as well as randomly generated ones, we focused on methods to generate good initial population. First of all we revisited the well-known Strip Algorithm and we developed new variants of it to generate cheap good tours for TSP. For instance, we introduced the 2-Part Strip Algorithm that works well on uniformly distributed instances and the Adaptive Strip Algorithm that is aimed to give good initial solutions for clustered TSP instances. We also carried on the analysis of one of the variants. For scheduling problems, the robust berth allocation in particular has been considered. We proposed a set of heuristics referred to as Edge Seeking Heuristics (ESH) to generate cheap, near feasible and feasible schedules. These are used to improve the initial population which is generated on the whole randomly. The aim of ESH is to reduce or completely remove overlapping.

In order to show the efficiency of PPA on discrete problems, we extended it to solve various and different types of problems. One of them is the well-known TSP. Here, we discussed the parameters of well-known algorithms and showed the advantage of PPA, which has only a few parameters. This makes our algorithm easy to implement. The second problem is the Knapsack Problem and the last one is the Robust Berth Allocation Problem. We carried out extensive experiments and compared the results with those of well-established algorithms and heuristics. These comparative results show that, on the whole, discrete PPA is a robust and efficient algorithm, which compares well with other Nature-inspired approaches to optimisation.

7.2 Future Research

This thesis is concerned with a Nature-inspired metaheuristic referred to as PPA. Our main contribution is to show that it works well on a variety of discrete optimisation problems such as TSP, KP and RBAP. However, discrete PPA only uses the runners approach to solve problems. In the literature, there is a seed-based strawberry plant propagation algorithm and a hybrid of PPA with SbPPA, [60,61]. The discrete PPA can be modified to implement the propagation via seeds as well. Discrete PPA can be used to solve other discrete type problems as found in the literature to show its efficiency. It can also be implemented to solve multi-objective optimisation problems and other scheduling problems as arise in timetabling applications, [135]. The performance of PPA can be tested on very large scale discrete problems. Finally, in this thesis MATLAB is used for all implementations of PPA. Other programming languages may improve the efficiency of the algorithm. Parallel implementation can be another useful approach to increase the efficiency of the algorithm. Given that PPA is fairly recent and has very attractive attributes such as simplicity and low number of parameters, it is worth while extending it to handle different optimisation problems.

BIBLIOGRAPHY

[1] D. P. Williamson and D. B. Shmoys, *The design of approximation algorithms*. Cambridge University Press, 2011.

[2] B. Plants, “Growing strawberries.” `https://bonnieplants.com/growing/growing-strawberries/`. Accessed April 06, 2017.

[3] C. F. Daganzo, “The length of tours in zones of different shapes,” *Transportation Research Part B: Methodological*, vol. 18, no. 2, pp. 135–145, 1984.

[4] G. Reinelt, “TSPLIB - A T.S.P. library,” Tech. Rep. 250, Universität Augsburg, Institut für Mathematik, Augsburg, 1990.

[5] “TSP test data.” `http://www.math.uwaterloo.ca/tsp/data/`.

[6] B. İ. Selamoğlu and A. Salhi, “The plant propagation algorithm for discrete optimisation: The case of the travelling salesman problem,” in *Nature-Inspired Computation in Engineering*, pp. 43–61, Springer, 2016.

[7] P. of Felixstowe, "Container operations- berths 8 and 9 for mega vessels." `https://www.portoffelixstowe.co.uk/port/container-operations-berths-8-9/`. Accessed April 06, 2017.

[8] B. İ. Selamoğlu, A. Salhi, and M. Sulaiman, "Strip algorithms as an efficient way to initialise population-based metaheuristics," in *Recent Developments in Metaheuristics* (F. Y. L. Amodeo, E-G. Talbi, ed.), Springer, 2017 (to appear).

[9] A. Salhi and E. Fraga, "Nature-inspired optimisation approaches and the new plant propagation algorithm," *Proceedings of the ICeMATH2011*, pp. K2–1 to K2–8, 2011.

[10] X.-S. Yang, *Introduction to Mathematical Optimization: From Linear Programming to Metaheuristics*. Cambridge International Science Publishing, 2008.

[11] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[12] G. C. Goodwin, M. M. Seron, and J. A. De Doná, *Constrained control and estimation: an optimisation approach*. Springer Science & Business Media, 2006.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[14] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[15] E. L. Lawler and D. E. Wood, "Branch-and-bound methods: A survey," *Operations research*, vol. 14, no. 4, pp. 699–719, 1966.

[16] R. Bellman, "Dynamic programming and lagrange multipliers," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 42, no. 10, p. 767, 1956.

[17] B. Korte, J. Vygen, B. Korte, and J. Vygen, *Combinatorial optimization*. Springer, 2002.

[18] P. Festa, "A brief introduction to exact, approximation, and heuristic algorithms for solving hard combinatorial optimization problems," in *Transparent Optical Networks (ICTON), 2014 16th International Conference on*, pp. 1–20, July 2014.

[19] T. H. Cormen, *Introduction to algorithms*. 2009.

[20] R. S. Garfinkel and G. L. Nemhauser, *Integer programming*, vol. 4. Wiley New York, 1972.

[21] J. E. Mitchell, "Branch-and-cut algorithms for combinatorial optimization problems," *Handbook of applied optimization*, pp. 65–77, 2002.

[22] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of computer and system sciences*, vol. 9, no. 3, pp. 256–278, 1974.

[23] E.-G. Talbi, *Metaheuristics: from design to implementation*, vol. 74. John Wiley & Sons, 2009.

[24] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," tech. rep., DTIC Document, 1976.

[25] R. Bar-Yehuda and S. Even, "A linear-time approximation algorithm for the weighted vertex cover problem," *Journal of Algorithms*, vol. 2, no. 2, pp. 198–203, 1981.

[26] D. S. Hochbaum, “Approximation algorithms for the set covering and vertex cover problems,” *SIAM Journal on computing*, vol. 11, no. 3, pp. 555–556, 1982.

[27] N. E. Young, “Randomized rounding without solving the linear program.,” in *SODA*, vol. 95, pp. 170–178, 1995.

[28] X.-S. Yang and L. Press, “Nature-inspired metaheuristic algorithms second edition,” 2010.

[29] E.-G. Talbi, “A taxonomy of hybrid metaheuristics,” *Journal of heuristics*, vol. 8, no. 5, pp. 541–564, 2002.

[30] L. Jourdan, M. Basseur, and E.-G. Talbi, “Hybridizing exact methods and metaheuristics: A taxonomy,” *European Journal of Operational Research*, vol. 199, no. 3, pp. 620–629, 2009.

[31] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, “Proof verification and the hardness of approximation problems,” *J. ACM*, vol. 45, pp. 501–555, May 1998.

[32] M. Yagiura and T. Ibaraki, “On metaheuristic algorithms for combinatorial optimization problems,” *Systems and Computers in Japan*, vol. 32, no. 3, pp. 33–55, 2001.

[33] A. Salhi and J. A. V. Rodríguez, “Tailoring hyper-heuristics to specific instances of a scheduling problem using affinity and competence functions,” *Memetic Computing*, vol. 6, no. 2, pp. 77–84, 2014.

[34] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[35] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[36] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[37] R. Eberhart and J. Kennedy, “A new optimizer using particle swarm theory,” in *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pp. 39–43, IEEE, 1995.

[38] J. Kennedy and R. C. Eberhart, “A discrete binary version of the particle swarm algorithm,” in *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, vol. 5, pp. 4104–4108, IEEE, 1997.

[39] H. Izakian, B. T. Ladani, A. Abraham, V. Snasel, *et al.*, “A discrete particle swarm optimization approach for grid job scheduling,” *International Journal of Innovative Computing, Information and Control*, vol. 6, no. 9, pp. 1–15, 2010.

[40] D. Karaboga and B. Basturk, “A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm,” *Journal of global optimization*, vol. 39, no. 3, pp. 459–471, 2007.

[41] D. Karaboga and B. Gorkemli, “A combinatorial artificial bee colony algorithm for traveling salesman problem,” in *Innovations in Intelligent Systems and Applications (INISTA), 2011 International Symposium on*, pp. 50–53, IEEE, 2011.

[42] M. Albayrak and N. Allahverdi, “Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms,” *Expert Systems with Applications*, vol. 38, no. 3, pp. 1313–1320, 2011.

[43] D. Karaboga and B. Gorkemli, "A quick artificial bee colony-qabc-algorithm for optimization problems," in *Innovations in Intelligent Systems and Applications (INISTA), 2012 International Symposium on*, pp. 1–5, IEEE, 2012.

[44] X.-S. Yang, "Firefly algorithm, stochastic test functions and design optimisation," *International Journal of Bio-Inspired Computation*, vol. 2, no. 2, pp. 78–84, 2010.

[45] G. K. Jati, R. Manurung, and Suyanto, "13 - discrete firefly algorithm for traveling salesman problem: A new movement scheme," in *Swarm Intelligence and Bio-Inspired Computation* (X.-S. Y. C. X. H. G. Karamanoglu, ed.), pp. 295 – 312, Oxford: Elsevier, 2013.

[46] X.-S. Yang and S. Deb, "Cuckoo search via lévy flights," in *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pp. 210–214, IEEE, 2009.

[47] G. K. Jati, H. M. Manurung, and S. Suyanto, "Discrete cuckoo search for traveling salesman problem," in *Computing and Convergence Technology (ICCCT), 2012 7th International Conference on*, pp. 993–997, IEEE, 2012.

[48] X.-S. Yang, "A new metaheuristic bat-inspired algorithm," in *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pp. 65–74, Springer, 2010.

[49] E. Osaba, X.-S. Yang, F. Diaz, P. Lopez-Garcia, and R. Carballedo, "An improved discrete bat algorithm for symmetric and asymmetric traveling salesman problems," *Engineering Applications of Artificial Intelligence*, vol. 48, pp. 59 – 71, 2016.

[50] A. Ouaarab, B. Ahiod, and X.-S. Yang, "Improved and discrete cuckoo search for solving the travelling salesman problem," in *Cuckoo Search and Firefly Algorithm*, pp. 63–84, Springer, 2014.

[51] T. Saenphon, S. Phimoltares, and C. Lursinsap, "Combining new fast opposite gradient search with ant colony optimization for solving travelling salesman problem," *Engineering Applications of Artificial Intelligence*, vol. 35, pp. 324–334, 2014.

[52] M. Mahi, O. K. Baykan, and H. Kodaz, "A new hybrid method based on particle swarm optimization, ant colony optimization and 3-opt algorithms for traveling salesman problem," *Applied Soft Computing*, vol. 30, pp. 484 – 490, 2015.

[53] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software*, vol. 95, pp. 51–67, 2016.

[54] W.-T. Pan, "A new fruit fly optimization algorithm: taking the financial distress model as an example," *Knowledge-Based Systems*, vol. 26, pp. 69–74, 2012.

[55] L. Wang, X.-l. Zheng, and S.-y. Wang, "A novel binary fruit fly optimization algorithm for solving the multidimensional knapsack problem," *Knowledge-Based Systems*, vol. 48, pp. 17–23, 2013.

[56] S. Bitam, M. Batouche, and E.-G. Talbi, "A bees life algorithm for cloud computing services selection," in *Multidisciplinary Computational Intelligence Techniques: Applications in Business, Engineering, and Medicine*, pp. 31–46, IGI Global, 2012.

[57] A. R. Mehrabian and C. Lucas, "A novel numerical optimization algorithm inspired from weed colonization," *Ecological informatics*, vol. 1, no. 4, pp. 355–366, 2006.

[58] Y. Zhou, Q. Luo, H. Chen, A. He, and J. Wu, "A discrete invasive weed optimization algorithm for solving traveling salesman problem," *Neurocomputing*, vol. 151, pp. 1227–1236, 2015.

[59] M. Sulaiman, A. Salhi, B. I. Selamoglu, and O. B. Kirikchi, "A plant propagation algorithm for constrained engineering optimisation problems," *Mathematical Problems in Engineering*, 2014. Article ID 627416, 10 pages, doi:10.1155/2014/627416.

[60] M. Sulaiman and A. Salhi, "A seed-based plant propagation algorithm: the feeding station model," *The Scientific World Journal*, vol. 2015, 2015.

[61] M. Sulaiman and A. Salhi, "A hybridisation of runner-based and seed-based plant propagation algorithms," in *Nature-Inspired Computation in Engineering*, pp. 195–215, Springer, 2016.

[62] M. Sulaiman, A. Salhi, E. S. Fraga, W. K. Mashwani, and M. M. Rashidi, "A novel plant propagation algorithm: Modifications and implementation," *Science International*, vol. 28, no. 1, pp. 201–209, 2016.

[63] S. Rahnamayan, H. R. Tizhoosh, and M. M. Salama, "Opposition-based differential evolution," *IEEE Transactions on Evolutionary computation*, vol. 12, no. 1, pp. 64–79, 2008.

[64] J. A. Lawrence and B. A. Pasternack, *Applied management science: modeling, spreadsheet analysis, and communication for decision making*. Wiley New York, 2002.

[65] A. Ang, "Hs. and tang, wh (2007). probability concepts in engineering," 2004.

[66] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[67] A. Reynolds, "How many animals really do the lévy walk? comment," *Ecology*, vol. 89, no. 8, pp. 2347–2351, 2008.

[68] X.-S. Yang, *Nature-Inspired Optimization Algorithms*. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 1st ed., 2014.

[69] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *BioSystems*, vol. 43, no. 2, pp. 73–81, 1997.

[70] X.-S. Yang, Z. Cui, R. Xiao, A. H. Gandomi, and M. Karamanoglu, *Swarm intelligence and bio-inspired computation: theory and applications*. Newnes, 2013.

[71] Z. W. Geem, J. H. Kim, and G. Loganathan, "A new heuristic optimization algorithm: harmony search," *Simulation*, vol. 76, no. 2, pp. 60–68, 2001.

[72] L. Few, "The shortest path and the shortest road through n points," *Mathematika*, vol. 2, no. 02, pp. 141–144, 1955.

[73] J. Beardwood, J. H. Halton, and J. M. Hammersley, "The shortest path through many points," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 55, pp. 299–327, Cambridge Univ Press, 1959.

[74] K. J. Supowit, E. M. Reingold, and D. A. Plaisted, "The travelling salesman problem and minimum matching in the unit square," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 144–156, 1983.

[75] G. Reinelt, *The Traveling Salesman: Computational Solutions for TSP Applications*. Berlin, Heidelberg: Springer-Verlag, 1994.

[76] K. J. Supowit, D. A. Plaisted, and E. M. Reingold, "Heuristics for weighted perfect matching," in *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pp. 398–419, ACM, 1980.

[77] G. Laporte, "The traveling salesman problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, vol. 59, no. 2, pp. 231–247, 1992.

[78] C. A. Floudas and P. M. Pardalos, *Encyclopedia of optimization*, vol. 1. Springer Science & Business Media, 2008.

[79] G. Laporte, "A concise guide to the traveling salesman problem," *Journal of the Operational Research Society*, vol. 61, no. 1, pp. 35–40, 2010.

[80] S. Steinerberger, "A new upper bound for the traveling salesman constant," *unpublished note*, 2013.

[81] D. Avis, "A survey of heuristics for the weighted matching problem," *Networks*, vol. 13, no. 4, pp. 475–493, 1983.

[82] H. J. Karloff, "How long can a euclidean traveling salesman tour be?," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 1, pp. 91–99, 1989.

[83] D. S. Johnson and L. A. McGeoch, "Experimental analysis of heuristics for the stsp," in *The traveling salesman problem and its variations*, pp. 369–443, Springer, 2007.

[84] M. Iri, K. Murota, and S. Matsui, “Heuristics for planar minimum-weight perfect metchings,” *Networks*, vol. 13, no. 1, pp. 67–92, 1983.

[85] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[86] D. E. Goldberg and K. Deb, “A comparative analysis of selection schemes used in genetic algorithms,” *Foundations of genetic algorithms*, vol. 1, pp. 69–93, 1991.

[87] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[88] D. E. Knuth, *The art of computer programming: sorting and searching*, vol. 3. Pearson Education, 1998.

[89] M. Bellmore and G. L. Nemhauser, “The traveling salesman problem: a survey,” *Operations Research*, vol. 16, no. 3, pp. 538–558, 1968.

[90] K. L. Hoffman, M. Padberg, and G. Rinaldi, “Traveling salesman problem,” in *Encyclopedia of Operations Research and Management Science*, pp. 1573–1578, Springer, 2013.

[91] A. Salhi, “The ultimate solution approach to intractable problems,” in *Proceedings of the 6th IMT-GT Conference on Mathematics, Statistics and its Applications*, pp. 84–93, 2010.

[92] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *Econometrica: Journal of the Econometric Society*, pp. 497–520, 1960.

[93] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM review*, vol. 33, no. 1, pp. 60–100, 1991.

[94] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied Mathematics*, pp. 196–210, 1962.

[95] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Oper. Res.*, vol. 21, pp. 498–516, Apr. 1973.

[96] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht, "Genetic algorithms for the traveling salesman problem," in *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pp. 160–168, Lawrence Erlbaum, New Jersey (160-168), 1985.

[97] E. H. Aarts, J. H. Korst, and P. J. van Laarhoven, "A quantitative analysis of the simulated annealing algorithm: A case study for the traveling salesman problem," *Journal of Statistical Physics*, vol. 50, no. 1-2, pp. 187–206, 1988.

[98] M. Clerc, "Discrete particle swarm optimization, illustrated by the traveling salesman problem," in *New optimization techniques in engineering*, pp. 219–239, Springer, 2004.

[99] X. H. Shi, Y. C. Liang, H. P. Lee, C. Lu, and Q. Wang, "Particle swarm optimization-based algorithms for tsp and generalized tsp," *Information Processing Letters*, vol. 103, no. 5, pp. 169–176, 2007.

[100] M. Malek, M. Guruswamy, M. Pandya, and H. Owens, "Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem," *Annals of Operations Research*, vol. 21, no. 1, pp. 59–84, 1989.

[101] H.-K. Tsai, J.-M. Yang, Y.-F. Tsai, and C.-Y. Kao, "An evolutionary algorithm for large traveling salesman problems," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 34, no. 4, pp. 1718–1729, 2004.

[102] X. Song, B. Li, and H. Yang, "Improved ant colony algorithm and its applications in tsp," in *Intelligent Systems Design and Applications, 2006. ISDA'06. Sixth International Conference on*, vol. 2, pp. 1145–1148, IEEE, 2006.

[103] Y. Marinakis, M. Marinaki, and G. Dounias, "Honey bees mating optimization algorithm for the euclidean traveling salesman problem," *Information Sciences*, vol. 181, no. 20, pp. 4684–4698, 2011.

[104] L. Li, Y. Cheng, L. Tan, and B. Niu, "A discrete artificial bee colony algorithm for tsp problem," in *Bio-Inspired Computing and Applications*, pp. 566–573, Springer, 2012.

[105] G. K. Jati *et al.*, *Evolutionary discrete firefly algorithm for travelling salesman problem*. Springer, 2011.

[106] D. S. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," *Local search in combinatorial optimization*, vol. 1, pp. 215–310, 1997.

[107] K. Helsgaun, *An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic*. PhD thesis, Roskilde University. Department of Computer Science, 2006.

[108] K.-T. Mak and A. J. Morton, “Distances between traveling salesman tours,” *Discrete Applied Mathematics*, vol. 58, no. 3, pp. 281 – 291, 1995.

[109] F. Mendivil, R. Shonkwiler, and M. C. Spruill, “Analysis of random restart and iterated improvement for global optimization with application to the traveling salesman problem,” *Journal of Optimization Theory and Applications*, vol. 124, no. 2, pp. 407–433, 2005.

[110] H. Hoos and T. Sttzle, *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[111] A. Salhi and Ö. Töreyen, “A game theory-based multi-agent system for expensive optimisation problems,” in *Computational Intelligence in Optimization*, pp. 211–232, Springer, 2010.

[112] W. Pang, K.-P. Wang, C.-G. Zhou, L.-J. Dong, M. Liu, H.-Y. Zhang, and J.-Y. Wang, “Modified particle swarm optimization based on space transformation for solving traveling salesman problem,” in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 4, pp. 2342–2346, IEEE, 2004.

[113] D. Pisinger, “Where are the hard knapsack problems?,” *Computers & Operations Research*, vol. 32, no. 9, pp. 2271–2284, 2005.

[114] S. Sahni, “Approximate algorithms for the 0/1 knapsack problem,” *Journal of the ACM (JACM)*, vol. 22, no. 1, pp. 115–124, 1975.

[115] A. Gherboudj, A. Layeb, and S. Chikhi, "Solving 0-1 knapsack problems by a discrete binary version of cuckoo search algorithm," *International Journal of Bio-Inspired Computation*, vol. 4, no. 4, pp. 229–236, 2012.

[116] Y. Shao, H. Xu, and W. Yin, "Solve zero-one knapsack problem by greedy genetic algorithm," in *Intelligent Systems and Applications, 2009. ISA 2009. International Workshop on*, pp. 1–4, IEEE, 2009.

[117] D. Zou, L. Gao, S. Li, and J. Wu, "Solving 0âĂŞ1 knapsack problem by a novel global harmony search algorithm," *Applied Soft Computing*, vol. 11, no. 2, pp. 1556 – 1564, 2011. The Impact of Soft Computing for the Progress of Artificial Intelligence.

[118] S. Pulikanti and A. Singh, "An artificial bee colony algorithm for the quadratic knapsack problem," in *International Conference on Neural Information Processing*, pp. 196–205, Springer, 2009.

[119] M. Hristakeva and D. Shrestha, "Solving the 0-1 knapsack problem with genetic algorithms," in *Proceedings of the 37th midwest instruction and computing symposium, Morris, MN*, 2004.

[120] S. Sundar, A. Singh, and A. Rossi, "An artificial bee colony algorithm for the 0–1 multidimensional knapsack problem," in *International Conference on Contemporary Computing*, pp. 141–151, Springer, 2010.

[121] H. Kellerer, U. Pferschy, and D. Pisinger, *Introduction to NP-Completeness of knapsack problems*. Springer, 2004.

[122] D. Pisinger, "The quadratic knapsack problemâĂŤa survey," *Discrete Applied Mathematics*, vol. 155, no. 5, pp. 623 – 648, 2007.

[123] L. Null, J. Lobur, *et al.*, *The essentials of computer organization and architecture*. Jones & Bartlett Publishers, 2014.

[124] K. K. Bhattacharjee and S. Sarmah, "A binary firefly algorithm for knapsack problems," in *Industrial Engineering and Engineering Management (IEEM), 2015 IEEE International Conference on*, pp. 73–77, IEEE, 2015.

[125] F. He, "An improved particle swarm optimization for knapsack problem," in *2009 International Conference on Computational Intelligence and Software Engineering*, pp. 1–4, Dec 2009.

[126] L. Wei, N. Ben, and C. Hanning, "Binary artificial bee colony algorithm for solving 0-1 knapsack problem," *Advances in Information Sciences & Service Sciences*, vol. 4, no. 22, 2012.

[127] C. Bierwirth and F. Meisel, "A survey of berth allocation and quay crane scheduling problems in container terminals," *European Journal of Operational Research*, vol. 202, no. 3, pp. 615–627, 2010.

[128] A. Imai, E. Nishimura, and S. Papadimitriou, "The dynamic berth allocation problem for a container port," *Transportation Research Part B: Methodological*, vol. 35, no. 4, pp. 401–417, 2001.

[129] A. Imai, X. Sun, E. Nishimura, and S. Papadimitriou, “Berth allocation in a container port: using a continuous location space approach,” *Transportation Research Part B: Methodological*, vol. 39, no. 3, pp. 199–221, 2005.

[130] J.-F. Cordeau, G. Laporte, P. Legato, and L. Moccia, “Models and tabu search heuristics for the berth-allocation problem,” *Transportation science*, vol. 39, no. 4, pp. 526–538, 2005.

[131] E. Nishimura, A. Imai, and S. Papadimitriou, “Berth allocation planning in the public berth system by genetic algorithms,” *European Journal of Operational Research*, vol. 131, no. 2, pp. 282–292, 2001.

[132] S. S. Ganji, A. Babazadeh, and N. Arabshahi, “Analysis of the continuous berth allocation problem in container ports using a genetic algorithm,” *Journal of marine science and technology*, vol. 15, no. 4, pp. 408–416, 2010.

[133] G. Alsoufi, X. Yang, and A. Salhi, “Robust berth allocation using a hybrid approach combining branch-and-cut and the genetic algorithm,” in *International Workshop on Hybrid Metaheuristics*, pp. 187–201, Springer, 2016.

[134] Y. Xu, Q. Chen, and X. Quan, “Robust berth scheduling with uncertain vessel delay and handling time,” *Annals of Operations Research*, vol. 192, no. 1, pp. 123–140, 2012.

[135] M. Cheraitia, S. Haddadi, and A. Salhi, “Hybridizing plant propagation and local search for uncapacitated exam scheduling problems,” *International Journal of Services and Operations Management*, 2017. To appear.

APPENDIX A

Matlab Code for The Adaptive Strip Algorithm

```

%An example implementation of the Adaptive Strip Algorithm

clear all
close all

cit=[ -6890.91       2242.06
    -5632.62       2838.55
    -5096.27       2947.10
    -7369.09       2424.18
    -4303.32       5701.62
    -5216.07       2805.86
    -7036.07       2433.68
    -10357.5       4229.90
    -5785.13       2920.73
    -5704.14       2459.86
    -5709.93       2658.69
    -5830.85       2331.90
    -5142.33       2719.89
    -5664.12       2312.69
    -4821.50       3061.67
    -6753.60       2091.31
    -8223.61       2444.14
    -5293.59       2714.79
    -4752.28       3095.56
    -6298.38       2104.00
    -5885.55       2924.21
    -5796.36       3012.19
    -6502.02       2078.80
    -5341.75       3053.49
    -8463.32        3369.09
    -8448.44        2616.77
    -7496.88        3163.43
    -6141.59        2100.22
    -5245.63        2908.83
    -5997.68        2316.13
    -6963.69        3234.25
    ];

[nnn col]=size(cit);
added=ones(nnn,2);
added=added.*(1.15*10^4);
cit=cit+added;
cit=sortrows(cit,1);
box_l=[];
%n=100;
limit=15;
%cit=n*rand(n,2);
cont=0;
x1=max(cit(:,1))+0.01;
x2=min(cit(:,1))-0.01;
y1=max(cit(:,2))+0.01;
y2=min(cit(:,2))-0.01;
completed=[];
al=[];
box=[x2;y2;x1;y2;x1;y1;x2;y1]';
[a,b]=count(box',cit);
draw_seg(box)
cell_all=[];
if b>limit
    cont=cont+1;
else
end
r=1;
Partition=box;

while cont>=1

    if size(Partition,2)<2*cont*8
        Partition(:,end+1:(2*cont*8))=0;
    end

    Temp1=Partition(r,:);
    if mod(r,2)==~0
        i=1;
        Temp=[];

        while i<=size(Partition,2)
            [a,b]=count(Temp1(i:i+7)',cit);
            a=sortrows(a,2);
            if b>limit
                coord=Split(Temp1(i:i+7)');
                Temp=[Temp,coord];
            else
            end
            i=i+8;
        end
        if size(Temp,2)<size(Partition,2)
            Temp(end+1:size(Partition,2))=0;
            Partition=[Partition;Temp];
        else
            Partition=[Partition;Temp];
        end
    else
        Temp=[];

        l=1;

        while l<=size(Partition,2)
            [p,b]=count(Temp1(l:l+7)',cit);
            p=sortrows(p,1);
            if b>limit;
                coord=Split_h(Temp1(l:l+7)');
                Temp=[Temp,coord];
            else

            end
            l=l+8;
        end
        if size(Temp,2)<size(Partition,2)
            Temp(end+1:size(Partition,2))=0;
            Partition=[Partition;Temp];
        else
            Partition=[Partition;Temp];
        end
    end
    r=size(Partition,1);
    Temp2=Partition(r,:);
    cont=0;
    j=1;
    gr=[];

    while j<=size(Partition,2)
        [c,k]=count(Temp2(j:j+7)',cit);
        Temp3=Temp2(j:j+7);
        if mod(r,2)==~0
            c=sortrows(c,1);
        else
            c=sortrows(c,2);
        end
        %
        if size(c,1)<=limit
            al=[al;Temp3];
        else
        end

        j=j+8;
        gr=[gr;k];

        if size(c,1)<=limit & size(c,1)>0
```

```
            box_l=[box_l;size(c)];
            completed=[completed;c];
        end
    end

    a=find(gr>limit);
    cont=size(a,1);
end

completed;
box_l;
set=[];
set1=[];
distance=0;
d=1;
setx=[];
for i=1:size(box_l,1)
    if box_l(i)==1
        set=[completed(1,:);completed(1,:)]
    else
        set=completed(1:box_l(i),:);
    end
    set1=[set1;set(1,:);set(end,:)];
    nc=[];

    for j=1:size(set,1)
        nc(1:j,1)=d;
    end
    setx=[setx;set,nc];
    d=d+1;
    completed(1:box_l(i),:)=[];
end
alx=al(1,:);
st_w=alx(1,3)-alx(1,1);

coord_box=[];
strip_box_tours=[];
coord_box1=[];
strip_box_tour=[];
a=rand();
for i =1:2:size(set1,1)
    set2=set1(i:i+1,:);
    if a>=0.5
        coord_box=[coord_box;set2(1,:)];
    else
        coord_box=[coord_box;set2(2,:)];
    end
end
cb=1:size(coord_box);
coord_box1=[coord_box cb'];
strip21m
bb;
bb(end,:)=[];
k=[];
for i=1:size(bb,1)
```

```
    for j=1:size(coord_box1,1)
        if coord_box1(j,1)==bb(i,1) & coord_box1(j,2)==bb(i,2)
            strip_box_tour=[strip_box_tour;bb(i,:) coord_box1(j,end)];
        end
    end
end
strip_box_tour;
setx;
strip_box_tour1=strip_box_tour(:,3:4);
k=[];
for j=1:size(strip_box_tour1,1)
    temp=[];
    for i=1:size(setx,1)
        if strip_box_tour1(j,2)==setx(i,3)
            temp=[temp;setx(i,:)];
        end
    end
    if strip_box_tour1(j,1)==0
        temp=sortrows(temp,[2]);
    else
        temp=sortrows(temp,[-2]);
    end

    k=[k;temp];
end
b=[];
hold on
k(end+1,:)=k(1,:);
plot(k(:,1),k(:,2))
for ii=1:length(k)-1
    jj=ii+1;
    distance=distance+(sqrt([k(ii,1)-k(jj,1)]^2+[k(ii,2)-k(jj,2)]^2));
end
distance=distance+(sqrt([k(1,1)-k(end,1)]^2+[k(1,2)-k(end,2)]^2));

aa=[];
hold on
plot(cit(:,1),cit(:,2),'*')
for i=2:size(Partition,1)
    a=Partition(i,:);
    b= find(a==0);
    a(b)=[];
    a;

    for t=1:8:size(a,2)
        box=a(t:t+7);
        plot( [box(1) box(1)] ,[box(2) box(8)] )
        hold on
        plot( [box(3) box(3)] ,[box(2) box(8)] )
        plot( [box(3) box(1)], [box(2) box(2)] )
        plot([box(3) box(1)],[box(6) box(6)])
        hold on
    end

end
```

```
distance
```

```
function [cities1,number]=count(coord,cit)
number=[];
if coord(1)==0
    cities=[cit(find(cit(:,1)>=coord(1) &  cit(:,1)<=coord(3)),:)];
else
    cities=[cit(find(cit(:,1)>coord(1) &  cit(:,1)<=coord(3)),:)];
end
if coord(2)==0
    cities1=cities(find((cities(:,2))>=coord(2) & (cities(:,2))<=coord(6)),:);
else
    cities1=cities(find((cities(:,2))>coord(2) & (cities(:,2))<=coord(6)),:);
end
number=[number;length(cities1)];
end
```

```
function draw_seg(coordinates)
x2=coordinates(1);
y2=coordinates(2);
x1=coordinates(3);
y1=coordinates(6);
plot([x1 x1] ,[y1 y2])
hold on
plot([x2 x2], [y1 y2])
plot([x1 x2] ,[y1 y1])
plot([x1 x2], [y2 y2])
end
```

```
function new_coord = Split(coord)
new_coord(1)=coord(1);
new_coord(2)=coord(2);
new_coord(3)=(coord(3)+coord(1))/2;
new_coord(4)=coord(2);
new_coord(5)=(coord(5)+coord(7))/2;
new_coord(6)=coord(6);
new_coord(7)=(coord(7));
new_coord(8)=coord(8);
new_coord(9)=new_coord(3);
new_coord(10)=new_coord(4);
new_coord(11)=coord(3);
new_coord(12)=coord(4);
new_coord(13)=coord(5);
new_coord(14)=coord(6);
new_coord(15)=new_coord(5);
new_coord(16)=new_coord(6);
end
```

```
function new_coord = Split_h(coord)
new_coord(1)=coord(1);
new_coord(2)=coord(2);
new_coord(3)=coord(3);
new_coord(4)=coord(4);
new_coord(5)=(coord(3));
new_coord(6)=(coord(6)+coord(4))/2;
new_coord(7)=(coord(7));
new_coord(8)=(coord(8)+coord(2))/2;
new_coord(9)=new_coord(7);
new_coord(10)=new_coord(8);
new_coord(11)=new_coord(5);
new_coord(12)=new_coord(6);
new_coord(13)=coord(5);
new_coord(14)=coord(6);
new_coord(15)=coord(7);
new_coord(16)=coord(8);
end
```

```
%The Strip Algorithm to connect boxes

city_coord= coord_box;
[nn co]=size(city_coord);
%r=ceil(sqrt(nn/2))
city_coord1=sortrows(city_coord,[1]);
city_coord1;
%sort the first column
%[city_coordi sorted_index]=sort(city_coord(:,1))
w=(st_w);
x1p=(x1-x2);
r=floor((x1p)/st_w);
%((max(city_coord1(:,1)+1))/r); %width of each strip
it=1;
st=1;
bb=[];
%j=1;
%city_coord1(n+1,2)=max(city_coord1(:,1))+2;
while (st <= r)
    bl=[];
    witness=0;
    while city_coord1(it,1) <= (st*w)
        witness=1;
        bl=[bl;city_coord1(it,:)];
        if it<nn
            it=it+1;
        else
            break
        end
    end
    if witness==1
        if mod(st,2)==0
            bb=[bb;sortrows(bl,[-2]),ones(size(bl,1),1)];
        else
            bb=[bb;sortrows(bl,[2]),zeros(size(bl,1),1)];
        end
        bb;
    end
    st=st+1;
end
bb(end+1,:)=bb(1,:);
hold on
plot1m
```

```
b=[];
for x=x2:w:x1
    b(end+1,:)=[x];
end
b;
%plot(bb(:,1),bb(:,2));
hold all
% plot([b b], [0 (max(max(bb(:,2))))+0.0001])
hold off
title('Adaptive Strip Algorithm')
```

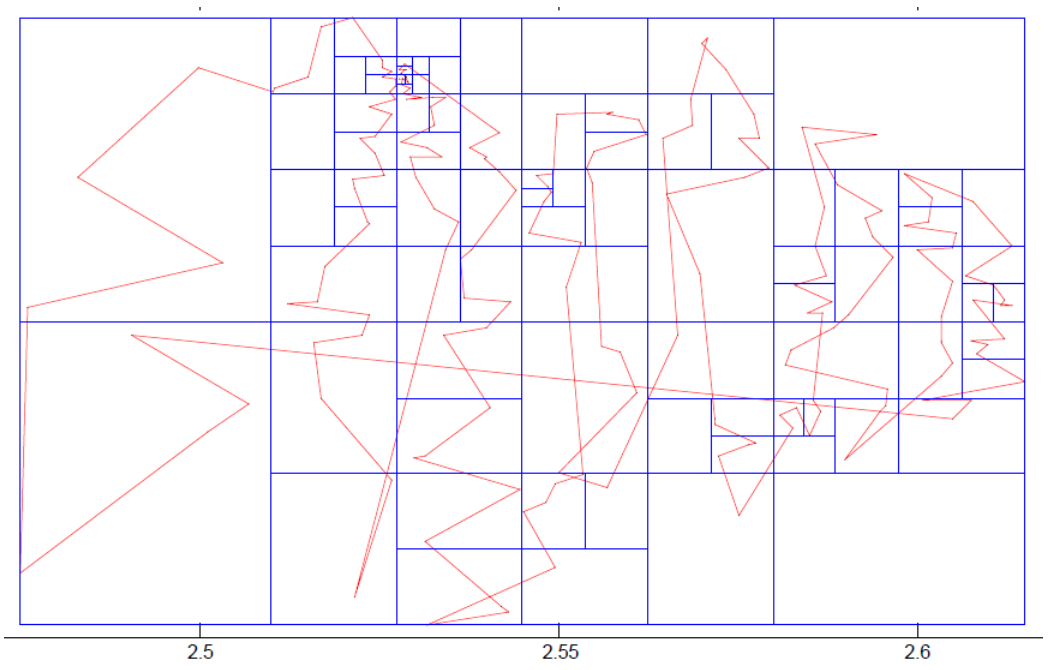

Figure 7.1: *An illustration of the output of ASA for a 194-city Qatar map*

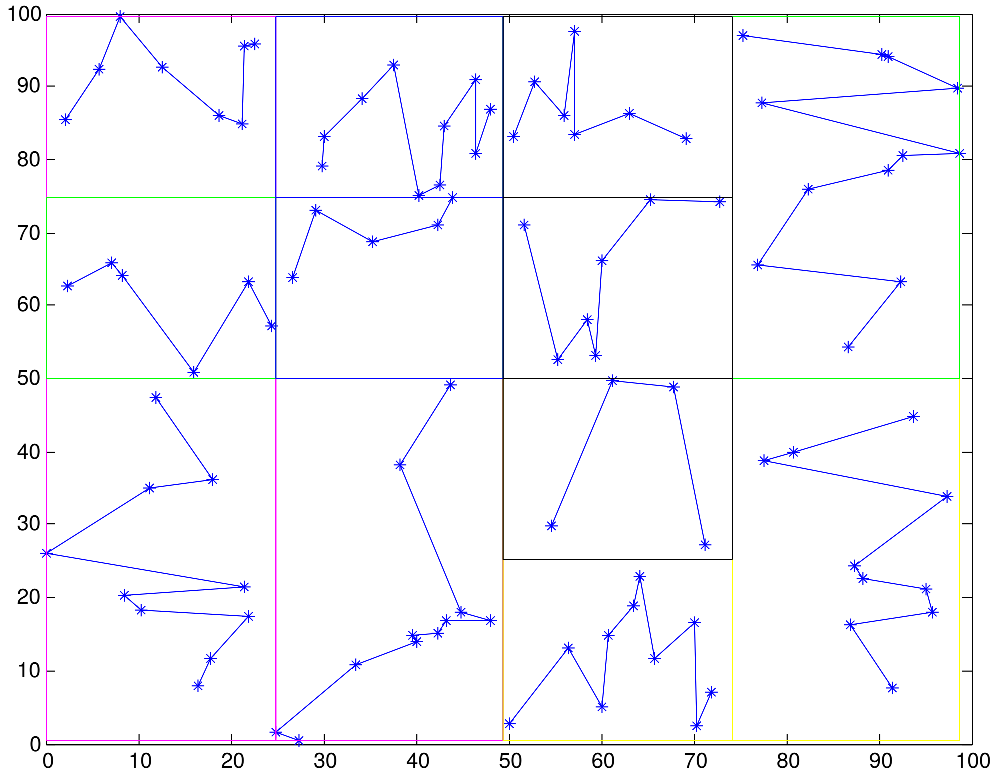

Figure 7.2: *An illustration that shows the interconnection inside boxes*

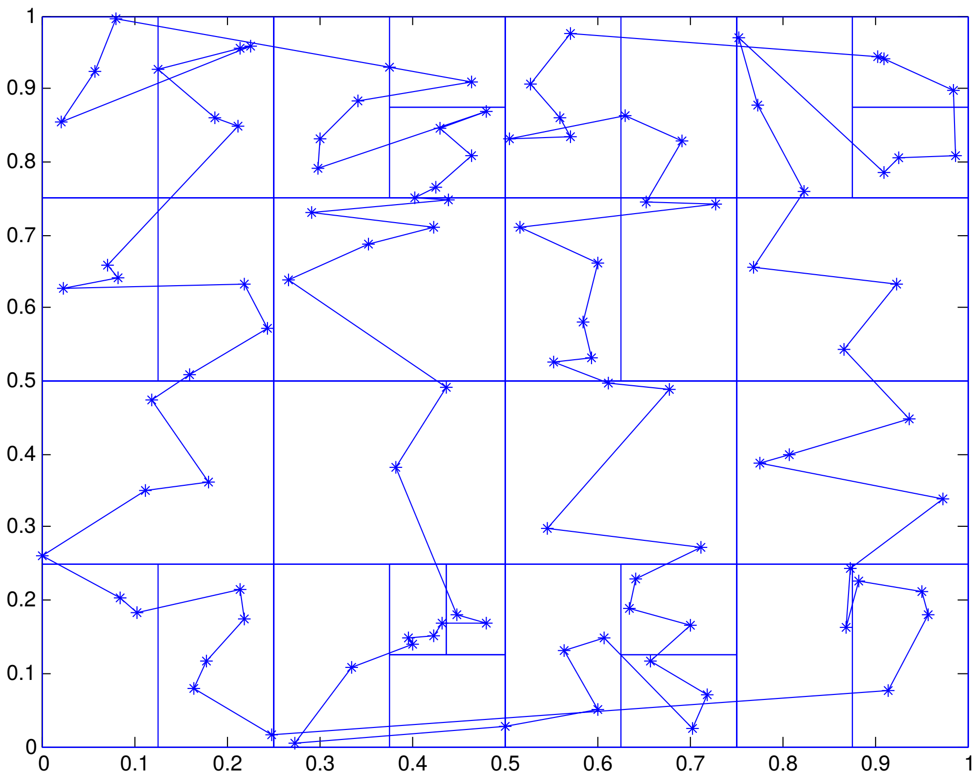

Figure 7.3: *An illustration of an output of ASA*

APPENDIX B

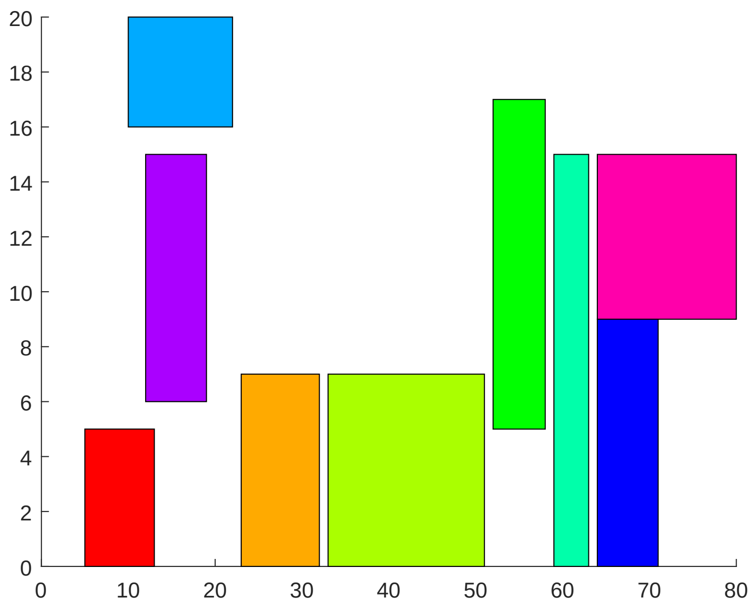

Figure 7.4: *An initial plant(schedule) generated using the deterministic approach for a 9-vessel problem*

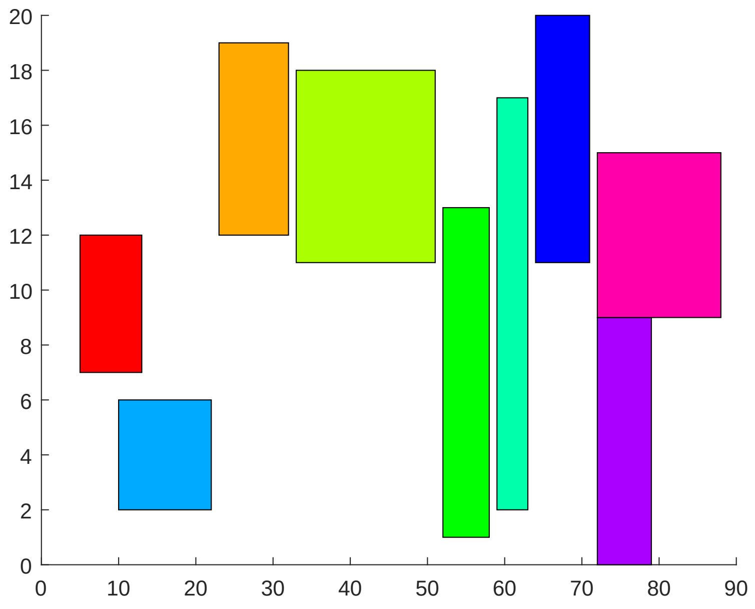

Figure 7.5: *An initial plant(schedule) generated using the semi-deterministic approach for a 9-vessel problem*

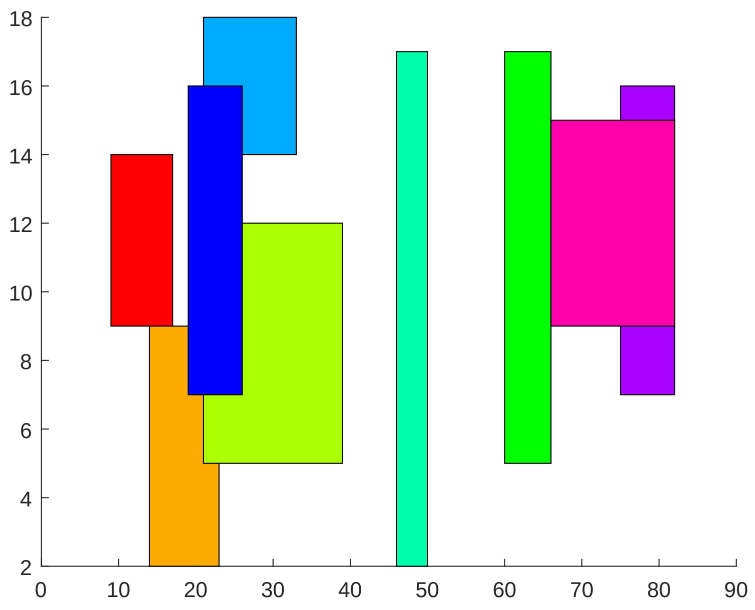

Figure 7.6: *An initial plant(schedule) generated using the random approach for a 9-vessel problem*

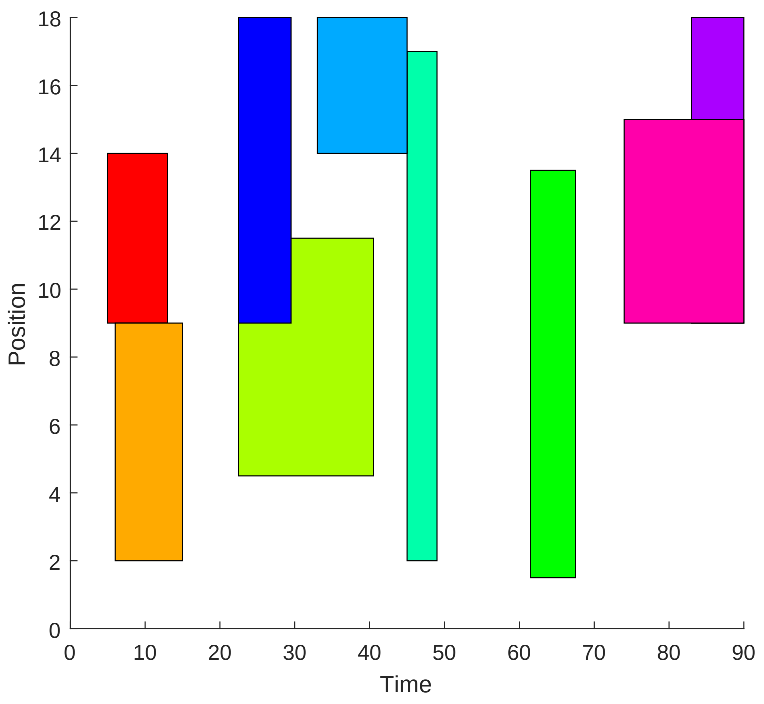

Figure 7.7: *An initial plant(schedule) generated using ESH5 for a 9-vessel problem*

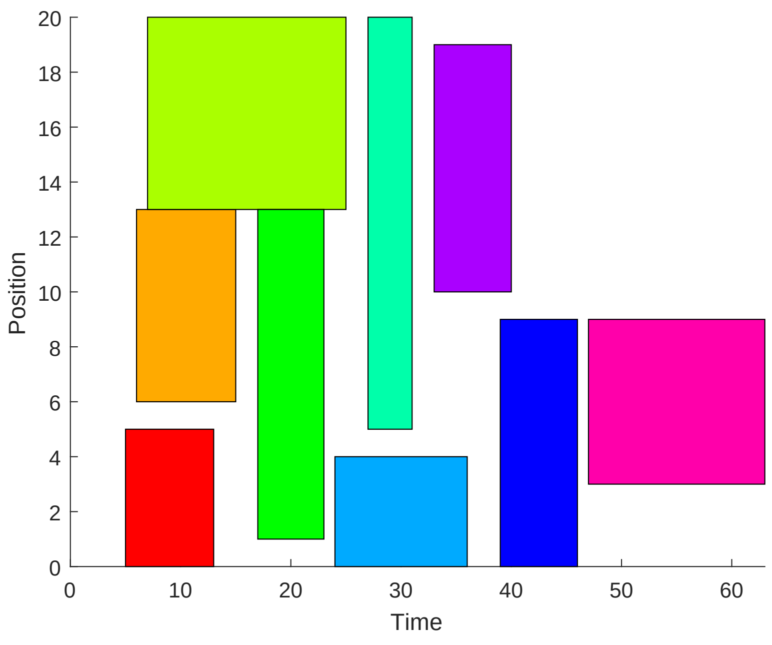

Figure 7.8: *Output of PPA for a 9-vessel problem*

PUBLICATIONS

[1] M. Sulaiman, A. Salhi, B. I. Selamoglu, and O. B. Kirikchi, "A Plant Propagation Algorithm for Constrained Engineering Optimisation Problems," Mathematical Problems in Engineering, vol. 2014, Article ID 627416, 10 pages, 2014. doi:10.1155/2014/627416

[2] B.I. Selamoglu and A. Salhi, 'The Plant Propagation Algorithm for Discrete Optimisation: The Case of the Travelling Salesman Problem,'Nature-Inspired Computation in Engineering', Vol. 637 of the series Studies in Computational Intelligence, pp 43-61, 2016.

[3] B.I. Selamoglu, A. Salhi and M. Sulaiman, 'Strip Algorithms as an Efficient Way to Initialise Population-based Metaheuristics', Eds. L. Amodeo, E-G. Talbi, F. Yalaoui, chap., Recent Developments of Metaheuristics, Springer, awaiting publication 2017.