วิธีฮิวริสติกค้นแบบนกกาเหว่าและอาณาจักรผึ้งเทียมแบบเพิ่มประสิทธิภาพสำหรับปัญหาการจัดเส้นทางเดินรถซึ่งมีข้อจำกัดด้านรถเที่ยวกลับและหน้าต่างเวลา

นายธนวรรธน์ วรวัชร์ทวีชัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรดุษฎีบัณฑิต
สาขาวิชาคณิตศาสตร์ประยุกต์และวิทยาการคณนา ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2560

CUCKOO SEARCH AND ENHANCED ARTIFICIAL BEE COLONY HEURISTIC METHODS FOR VEHICLE ROUTING PROBLEM WITH BACKHAUL AND TIME WINDOW CONSTRAINTS

Mr. Tanawat Worawattawechai

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy Program in Applied Mathematics and Computational Science
Department of Mathematics and Computer Science
Faculty of Science
Chulalongkorn University
Academic Year 2017

Thesis Title	CUCKOO SEARCH AND ENHANCED ARTIFICIAL BEE COLONY HEURISTIC METHODS FOR VEHICLE ROUTING PROBLEM WITH BACKHAUL AND TIME WINDOW CONSTRAINTS
By	Mr. Tanawat Worawattawechai
Field of Study	Applied Mathematics and Computational Science
Thesis Advisor	Assistant Professor Boonyarit Intiyot, Ph.D.
Thesis Co-Advisor	Associate Professor Chawalit Jeenanunta, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

.....Dean of the Faculty of Science
(Associate Professor Polkit Sangvanich, Ph.D.)

THESIS COMMITTEE

.....Chairman
(Assistant Professor Krung Sinapiromsaran, Ph.D.)

.....Thesis Advisor
(Assistant Professor Boonyarit Intiyot, Ph.D.)

.....Thesis Co-Advisor
(Associate Professor Chawalit Jeenanunta, Ph.D.)

.....Examiner
(Associate Professor Phantipa Thipwiwatpotjana, Ph.D.)

.....Examiner
(Kitiporn Plaimas, Ph.D.)

.....External Examiner
(Associate Professor Jirachai Buddhakulsomsiri, Ph.D.)

ธนวรรธน์ วรวัชร์ทวีชัย : วิธีฮิวริสติกค้นแบบนกกาเหว่าและอาณาจักรผึ้งเทียมแบบเพิ่มประสิทธิภาพสำหรับปัญหาการจัดเส้นทางเดินรถซึ่งมีข้อจำกัดด้านรถเที่ยวกลับและหน้าต่างเวลา (CUCKOO SEARCH AND ENHANCED ARTIFICIAL BEE COLONY HEURISTIC METHODS FOR VEHICLE ROUTING PROBLEM WITH BACKHAUL AND TIME WINDOW CONSTRAINTS) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร. บุญฤทธิ์ อินทิยศ, อ.ที่ปรึกษาวิทยานิพนธ์ร่วม: รศ. ดร. ชวลิต จีนอนันต์, หน้า.

ปัญหาการจัดเส้นทางเดินรถซึ่งมีข้อจำกัดด้านรถเที่ยวกลับและหน้าต่างเวลามีจุดประสงค์เพื่อหาเส้นทางเดินรถที่เป็นไปได้ที่ทำให้ระยะทางในการเดินทางโดยรวมมีค่าน้อยที่สุด โดยมีข้อจำกัดในด้านความจุของรถ รถเที่ยวกลับ และ หน้าต่างเวลา ในดุษฎีนิพนธ์นี้ ได้นำเสนอตัวแบบทางคณิตศาสตร์ของปัญหาการจัดเส้นทางเดินรถซึ่งมีข้อจำกัดด้านรถเที่ยวกลับและหน้าต่างเวลา เพื่อใช้ในการหาผลเฉลยที่เหมาะที่สุด นอกจากนี้ ยังได้นำเสนอวิธีฮิวริสติกผู้แข่งขันที่เร่งด่วนและใกล้ที่สุด (nearest urgent candidate หรือ NUC) ซึ่งใช้เทคนิคในการจัดลำดับความเร่งด่วนและการเลือกผู้แข่งขัน และวิธีฮิวริสติกการเลือกเพื่อนบ้านใกล้ที่สุดด้วยวงล้อรูเล็ตต์ (nearest neighbor with roulette wheel selection หรือ NNRW) ซึ่งเป็นการผสมผสานวิธีการเลือกด้วยวงล้อรูเล็ตต์กับการเลือกเพื่อนบ้านที่ใกล้ที่สุดที่ปรับปรุงแล้ว เพื่อนำมาใช้แก้ปัญหานี้ นอกจากนี้ ยังได้นำเสนอวิธีเมตาฮิวริสติกสองวิธี เพื่อหาผลเฉลยที่เหมาะที่สุดหรือใกล้จะเหมาะที่สุด วิธีแรกคือขั้นตอนวิธีค้นแบบนกกาเหว่า (cuckoo search หรือ CS) ซึ่งได้ถูกนำมาใช้กับปัญหานี้เป็นครั้งแรก วิธีที่สองคือขั้นตอนวิธีอาณาจักรผึ้งเทียมแบบเพิ่มประสิทธิภาพ (enhanced artificial bee colony หรือ EABC) ซึ่งใช้เทคนิครายชื่อต้องห้าม การค้นหาอย่างเป็นลำดับของผึ้งเฝ้ารัง และการผสมผสานของเทคนิคต่าง ๆ ในการค้นคำตอบใกล้เคียง ผลการวิจัยเชิงคำนวณชี้ให้เห็นว่า ขั้นตอนวิธีที่ถูกนำเสนอมาทั้งหมดนั้นมีสมรรถภาพที่ดีในเชิงคุณภาพโดยเฉพาะขั้นตอนวิธีอาณาจักรผึ้งเทียมแบบเพิ่มประสิทธิภาพ ซึ่งได้ 33 ผลเฉลยที่เทียบเท่าหรือผลเฉลยที่ดีที่สุดอันใหม่จาก 45 ปัญหาโดยเปรียบเทียบกับผลเฉลยที่ดีที่สุดที่รวบรวมมาจากงานวิจัยต่างๆ ดังนั้นขั้นตอนวิธีที่นำเสนอข้างต้นจึงเป็นวิธีที่มีประสิทธิภาพในการแก้ปัญหาการจัดเส้นทางเดินรถซึ่งมีข้อจำกัดด้านรถเที่ยวกลับและหน้าต่างเวลา

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต	.....
สาขาวิชา	คณิตศาสตร์ประยุกต์และวิทยาการคณนา	ลายมือชื่อ อ.ที่ปรึกษาหลัก	.....
		ลายมือชื่อ อ.ที่ปรึกษาร่วม	.....
ปีการศึกษา	2560		

5672859623 : MAJOR APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCE
KEYWORDS: VEHICLE ROUTING PROBLEMS / BACKHAUL / TIME WINDOW / HEURISTICS / ARTIFICIAL BEE COLONY / CUCKOO SEARCH

TANAWAT WORAWATTAWECHAI: CUCKOO SEARCH AND ENHANCED ARTIFICIAL BEE COLONY HEURISTIC METHODS FOR VEHICLE ROUTING PROBLEM WITH BACKHAUL AND TIME WINDOW CONSTRAINTS. ADVISOR: ASST. PROF. BOONYARIT INTIYOT, Ph.D., CO-ADVISOR: ASSOC. PROF. CHAWALIT JEENANUNTA, Ph.D., pp.

The vehicle routing problem with backhauls and time windows (VRPBTW) aims to find a feasible vehicle route that minimizes the total traveling distance while imposing capacity, backhaul, and time-window constraints. In this dissertation, a mathematical model of VRPBTW is introduced to obtain an optimal solution. The heuristics, namely the nearest urgent candidate (NUC), which applies the urgency priority and candidate techniques, and the nearest neighbor with roulette wheel selection (NNRW) which is a combination of a roulette wheel selection method and the improved nearest neighbor heuristic, are also presented to solve this problem. Moreover, two metaheuristic methods are presented to obtain the optimal or near optimal solutions. The first is a cuckoo search (CS) algorithm, which is applied to this problem for the first time. The second is the enhanced artificial bee colony (EABC) algorithm which uses a forbidden list, the sequential search for onlookers, and the combination of neighborhood search techniques. The computational results indicate that proposed algorithms yield good performance in terms of solution quality, especially EABC. It obtained 33 ties or new best known solutions out of 45 instances comparing with the best known solutions found in the literature. Hence, the proposed algorithms are the effective ways to solve the VRPBTW.

Department:	Mathematics and Computer Science	Student's Signature .....
		Advisor's Signature .....
Field of Study:	Applied Mathematics and Computational Science	Co-Advisor's Signature .....
Academic Year:	2017	

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Assistant Professor Dr. Boonyarit Intiyot, my dissertation advisor, Associate Professor Dr. Chawalit Jeenanunta, my dissertation co-advisor, for their kind supervision and encouragement with their patience and knowledge throughout my dissertation. Without their constructive suggestions and knowledgeable guidance in this study, this research would never have successfully been completed.

Sincere thanks and deep appreciation are also extended to Assistant Professor Dr. Krung Sinapiromsaran, Associate Professor Dr. Phantipa Thipwiwatpotjana, Dr. Kitiporn Plaimas, and Associate Professor Dr. Jirachai Buddhakulsomsiri, my dissertation committees, for their comments and suggestions. Besides, I would like to thank all teachers who have taught me all along.

I am also grateful to Development and Promotion of Science and Technology Talents Project (DPST) for granting me financial support to do this research.

In particular, I would like to thank my dear friends for giving me good advice about my dissertation and experiences at Chulalongkorn University. Finally, I take this opportunity to express the profound gratitude from my deep heart to my beloved parents for their love and encouragement throughout my study.

CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 General Vehicle Routing Problem

Since business has always been a highly competitive environment, many companies employ strategies for optimizing their logistics system to make their business more efficient. To effectively improve logistic service quality, several problems have been studied including vehicle routing problem (VRP).

The vehicle routing problem (VRP) is a transportation problem which is widely studied in operations research. The objective of VRP is to find an optimal set of routes, which minimizes total cost, for delivering goods located at the central depot to all customers who have placed demands for goods. This problem is widely applied in many applications such as logistics distribution, school bus routing, and mailing system. Many types of vehicle routing problem models have been developed due to varieties of real-world situations, namely the capacitated vehicle routing problem (CVRP) [1-2], the open vehicle routing problem (OVRP) [3-4], the vehicle routing problem with simultaneous delivery and pickups (VRPDP) [5-6], the vehicle routing problem with backhauls (VRPB) [6-10], the vehicle routing problem with time window (VRPTW) [11-18], etc. Since the VRP is an NP-hard combinatorial optimization problem [20], the exact algorithm is not always possible to find an optimal solution within a limited time. For larger problems, heuristics and metaheuristics are more appropriate than exact methods.

A heuristic is an optimization technique which explores the space of the feasible solutions to rapidly get satisfactory solution. There are well-known heuristics such as nearest neighbor algorithm, sweep algorithm, and cluster algorithm. Furthermore, there are also metaheuristic methods which are higher-level heuristics designed for finding a near optimal feasible solution. Examples of these algorithms are genetic algorithm (GA) [14, 21], particle swarm optimization (PSO) [22-23], ant colony optimization

(ACO) [19, 24], cuckoo search (CS) [55-56], artificial bee colony algorithm (ABC) [57-62], and firefly algorithm (FA) [33-34] and bat algorithm (BA) [35-36].

In this dissertation, a mathematical model of the vehicle routing problem with backhauls and time windows (VRPBTW) is introduced. Moreover, two new heuristics and two new metaheuristic methods for solving VRPBTW are proposed and their solutions to the benchmark problems are compared with those of several other methods.

1.2 Motivations

Since VRPBTW is one of vehicle routing problem types which is NP-hard combinatorial optimization problem, it is too difficult to be solved by an exact method within a limited time. In this dissertation, two heuristics and two metaheuristic methods for VRPBTW are proposed to solve this problem. The heuristics are used to obtain high quality feasible initial solutions in the brief time and the metaheuristics explore a larger area of the solution space to achieve good optimization results.

The two heuristics are the nearest urgent candidate (NUC) and the nearest neighbor with roulette wheel selection (NNRW) heuristics. For NUC, the basic idea is that the most urgent customer should be firstly served. When generating an initial solution, the urgency of their delivery is considered the first priority while the closeness is the second one. However, NUC yields only one initial solution which is not suitable for population-based metaheuristics while NNRW can generate many good initial solutions. The basic idea of NNRW is to generate routes by using roulette wheel strategy where the nearer customers have higher probability to be selected as the next customer in the current route. Although these methods can provide decent feasible initial solutions and solve the problem in a short time, the solutions obtained from these methods always get stuck in the local optima.

The two metaheuristic methods are the cuckoo search (CS), and the enhanced artificial bee colony algorithm (EABC). CS is inspired by an aggressive breeding behavior of cuckoo birds. The female cuckoos lay eggs in the nest of other host birds to let them hatch and brood young cuckoo chicks. To the best of our knowledge, CS algorithm had never been applied to VRPBTW. Thus, we propose a CS algorithm for VRPBTW in this dissertation. Moreover, CS is that it requires less parameters

compared with other metaheuristics. CS has only 2 parameters while GA [14], ACO [19], and PSO [22] have 3, 6, 8 parameters respectively. Since the results of metaheuristics are affected by parameter tuning, the less parameters is more desirable because it makes parameter tuning easier. However, the disadvantage of CS is that when there are many duplicated initial solutions, they are not properly dispersed in the solution space and that can easily lead to trapping in a local optimum. EABC is an enhanced version of ABC, which is inspired from the intelligent finding food sources behavior of the honey bees around the hives. Although the original ABC has a strategy for preventing premature convergence of the solutions, it has the same problems as CS when there are many duplicated initial solutions. Hence, the enhanced version (EABC) is proposed to solve this problem by using the strategy called *forbidden list*. Moreover, the sequential search strategy, and the intra-route and inter-route exchange combination strategy are applied in EABC to extend the exploration on the solution space to obtain better solutions.

1.3 Research Objectives

The objectives of this research study are stated as follows:

1. To propose a mathematical model of the vehicle routing problem with backhauls and time windows.
2. To develop heuristic methods for solving the vehicle routing problem with backhauls and time windows and compare the performance with the existing heuristics using benchmark problems.

1.4 Scope of the Research

In this research study, the VRPBTW is the single-trip VRPBTW for each delivery vehicle. Each vehicle starts from the depot, which is assumed to be the only one depot located in the city, and then serves a set of customers before going back to the depot, which is considered as the destination. A time window refers to a fixed time window in which the associated customer must be served. Note that the traveling time from customer A to customer B is equal to the Euclidean distance between them. A vehicle is allowed to pick up the goods from backhaul customers and then back to the

depot only after all linehaul customers are served. A vehicle is also allowed to deliver goods to linehaul customers and then go back to the depot without any pick up. The aim of our research is to minimize total distance for VRPBTW. Thus, there is no additional cost for adding vehicles. The performance of the heuristics are measured based on the benchmark problem sets developed by Gelinas et al. [37].

1.5 Overview of Dissertation

The dissertation is divided into five chapters, namely introduction, literature review, VRPBTW and heuristic approaches, computational experiment, and conclusion.

The first chapter is the introduction about VRP in general, the research objectives, and the scope of this research study.

The second chapter is the literature review about the VRP and its variants including VRPB, VRPTW, and VRPBTW. Furthermore, we also review the solution approaches to VRP which can be categorized into mathematical optimization approaches, heuristic approaches, and metaheuristic approaches.

The third chapter describes VRPBTW, the proposed mathematical models, and the solution approaches. Firstly, VRPBTW is described and then a proposed mathematical model for the problem is introduced. Next, the nearest neighbor approaches are described including a general nearest neighbor heuristic (NN), an improved nearest neighbor heuristic (INN), a new nearest urgent candidate heuristic (NUC), and a new nearest neighbor with roulette wheel selection method (NNRW). The last two approaches (NUC and NNRW) are our proposed methods in this study. The last part presents our proposed metaheuristic approaches, which includes the cuckoo search (CS), and the enhanced artificial bee colony algorithm (EABC).

The fourth chapter presents the computational experiments and results. Firstly, the descriptions of test problems are described. Then, a small study on parametrization is explained in the parameter setting section. Next, the computational results from the mathematical model, the heuristics, and the metaheuristics are presented. Moreover, the comparisons of our methods with the existing algorithms for solving the VRPBTW are

also shown in the same section. In addition, we also study about rate of convergence and discuss about results.

The fifth chapter is the conclusion of this study.

CHAPTER 2
LITERATURE REVIEW

2.1 Vehicle Routing Problem

The vehicle routing problem (VRP) is a generalization of the travelling salesman problem (TSP) which is a non-deterministic polynomial-time hard (NP-hard) problem in computational complexity theory. It is a combinatorial optimization and integer programming problem aiming to minimize the total distance or the total number of vehicles used. The basic VRP consists of a depot, a set of customers who require goods from the depot, and a fleet of vehicles. Each vehicle starts from the depot and serves a set of customers before going back to the depot. It was first introduced by Dantzig and Ramser [38]. The formulation of VRP has been extended with various constraints to reflect real-world applications such as capacity, time windows, pickup and delivery, cross-docking, and backhaul.

The capacitated vehicle routing problem (CVRP) is a classical version of VRP. The objective of CVRP is to find an optimal route set which minimizes the total cost for a fleet of homogeneous vehicles to serve a set of customers while being restricted by the capacity of vehicles. All vehicle routes begin and end at the depot and all customers are visited. Each customer is visited exactly once by exactly one vehicle. The total demand of each route must not exceed the vehicle capacity. More details of CVRP can be found in a VRP textbook such as [39].

2.1.1 Vehicle Routing Problem with Backhauls

The vehicle routing problem with backhauls (VRPB) is one of the interesting variations of VRP where a vehicle does not only deliver goods to the linehaul customers but also picks up goods from the backhaul customers before going back to the depot. A vehicle is also allowed to deliver goods to linehaul customers and then goes back to the depot without any pick up. The benefit of doing so is to utilize the unused capacity of empty vehicle on the way back to the depot after delivery. For example, a coffee

company delivers the goods to its customers and picks up their raw materials on their way back to its factory [40].

Toth and Vigo [7] proposed a heuristic which used the information of a Lagrangian relaxation to obtain the formation of clusters of customers for generating feasible routes. To improve the quality of the solution, intra-route and inter-route methods were applied.

Osman and Wassan [8] presented a reactive tabu search which was a new way to exchange neighborhood structures for VRPB. In their study, two algorithms were used to find the initial solutions, namely the saving-insertion heuristic (SIH), and the saving-assignment heuristic (SAH). For SIH heuristic, a set of vehicle routes for linehauls was constructed by using saving algorithm, and then backhauls were inserted into those routes while preserving the vehicle constraints. SAH started by generating two sets of vehicle routes: one for linehauls and another for backhauls. The initial solutions were constructed by using the 2-opt and 3-opt improvement heuristics to merge linehauls routes and backhauls routes. In their tabu search algorithm, the λ-interchange was used as the neighborhood search. The results showed that this algorithm was robust, and gave the better solutions than Toth and Vigo [7].

Brandao [9] presented a new tabu search algorithm (TSA) for the VRPB. For the initial solution construction, two methods were introduced. The first method was the open initial solution (TSA-open), which considered the two sets of customers (linehauls and backhauls) separately, each with their own VRP, and then linked the two solutions from both VRPs to form the initial solutions. The second method was the K-tree initial solution (TSA-K-tree), which selected the best 10 lower bounds from Lagrangian relaxation to create 10 initial solutions. The computational results showed that TSA algorithm outperformed the heuristic of Toth and Vigo [7] and the average results of TSA-open were almost identical to Osman and Wassan (2002).

A memetic algorithm with different local search methods was presented by Tavakkoli-Moghaddam et al. [10]. The concept of this algorithm was the simple population was used instead of complicated structured populations. Many types of evolutionary operators were used in this algorithm, namely partial-mapped crossover (PMX), order crossover (OX), position based crossover (PBX), and order-based

crossover (OBX). The results showed that this algorithm was better than the heuristic of Toth and Vigo [7].

Gajpal and Abad [11] presented multi-ant colony system which used pheromone data to generate the solutions. They divided ants into two types, namely vehicle-ants and route-ants, to construct a feasible solution. Each elitist ant was equally important and distinct to prevent trapping at a local minimum. Moreover, the solutions (elitist ants) were abandoned when the solutions were not improved in limited time. This algorithm gave some better solutions than the others and five new best known solutions for the benchmark problem instances available in the literature.

2.1.2 Vehicle Routing Problem with Time Windows

The vehicle routing problem with time windows (VRPTW) is a VRP with a specified time slot that a delivery is allowed for each customer. A waiting time occurs if a vehicle arrives before the specified time window. VRPTW is commonly found in distribution planning, material transportation, and E-grocery delivery.

Chiang and Russell [12] proposed a reactive tabu search metaheuristic for the VRPTW. They applied the intensification and diversification strategies to obtain the high-quality solutions. To improve the solutions, the λ-interchange was used as the neighborhood search. Large scale for real-world problems and test problems from the literature were used in computational results report.

Berger and Barkaoui [13] presented a new memetic algorithm in the serial and parallel versions to address the VRPTW. Later, they presented a new parallel hybrid genetic algorithm for VRPTW [14]. Two sets of populations (solutions) were evolved in different directions; the first one focuses on minimizing total distance, and the second one focuses on minimizing temporal constraint violation to find a new feasible solution. The master–slave message–passing paradigm was used for parallel method. The master processing element controlled the parent selection process while reproduction and mutation operators were managed by the slave processing elements. The results showed that this algorithm was highly competitive and provided some new best known solutions.

Bräysy and Gendreau [15] presented research survey on the tabu search algorithms for VRPTW. In the comparison with the current best approaches by using Solomon's benchmarks [16], they concluded that tabu search algorithm is one of the best techniques to tackle this problem.

Gong et al. [17] proposed a two-generation (father and children) ant colony system for VRPTW. In the children generation, the sub-routes were generated by minimizing the total distance while preserving the time window constraints. Then, the feasible solutions were composed from sub-tours in the father generation. For small test problems, this algorithm reduced the vehicles in use but it increased the total distance and a little break of the time window when comparing with the ant colony algorithm in other literature.

A hybrid version which consisted of ant colony optimization (ACO) and tabu search (TS) was presented by Yu et al. [18]. The initial solutions were constructed by using ACO, and TS maintained the diversity of the current solutions as well as explores the new solutions. Using Solomon's test problems [16], this algorithm obtained 41 best known solutions out of 56 instances. They concluded that this hybrid version was an effective tool for VRPTW.

Ding et al. [19] presented the hybrid ant colony optimization (HACO) for VRPTW. The ACO was combined with the saving algorithm and λ-interchange mechanism to increase the convergence speed. Furthermore, the strategy of candidate list was adopted to reduce the time to compute the transition probabilities when ants selected the next customer in construction phase. In addition, the pheromone approach, which was based on Min-Max ant system, and a disaster operator were applied to prevent trapping in local optima. The results indicated that HACO was competitive with existing heuristics in literature and also found new best known solutions for some instances.

2.1.3 Vehicle Routing Problem with Backhauls and Time Windows

The vehicle routing problem with backhauls and time windows (VRPBTW) is an extension of the vehicle routing problem with backhaul (VRPB) by imposing a specific service time window for each customer. There are three main types of

constraints in this problem: capacity, time window, and backhaul. The capacity constraints ensure that the total demand in each vehicle does not exceed its capacity. The time window constraints confirm that each vehicle arrives at each customer within his or her specified time slot. The backhaul constraints guarantee that each vehicle serves linehaul customers before backhaul customers and eventually goes back to the depot.

An increasing number of publications on heuristic approaches for vehicle routing problem have been developed for the past two decades. However, only few studies have been devoted to the VRPBTW.

Gelinas et al. [37] proposed a new branching strategy for branch-and-bound approaches based on column generation for the VRPTW. Two main strategies were time window divisions and branching strategies. For time window division, time window was divided into two subintervals to create two new problems, and then some conditions were added to each problem to eliminate routes. For branching strategies, there were two techniques to choose a network node on which to branch, namely choosing a time window division, choice of the network node for branching (elimination of cycles, number of visits, and flow values). To test these strategies, 45 benchmarks for VRPBTW were constructed based on Solomon data set [16]. Results showed that this method successfully solved 34 problems optimally with up to 100 customers.

Thangiah et al. [41] introduced a heuristic approach to VRPBTW called the push-forward insertion heuristic (PFIH), which was used for generating routes one by one. A customer which was nearest to depot was selected to be a customer seed for current route, and then unassigned customers were inserted at the best feasible positions. If there were no feasible insertion places in the current route, the algorithm would repeat by finding a new customer seed to generate a new route. To improve the solutions, the λ-interchange and 2-opt* methods were used as the local search. For the computational results, PFIH solutions were compared against known optimal solutions, and were within 2.5% of the optima on the average.

Potvin et al. [21] proposed a genetic algorithm (GA) for VRPBTW. The initial solutions were constructed using the greedy insertion heuristic, which was derived from Solomon's work [16], during the route construction. For recombination, the OX, MX1,

MX2, and 1X operators were applied. The techniques called remove-and-reinsert, swap, and last-will-be-first were introduced for mutation. The computational results showed that the solutions of GA were within 1% of the optima on average.

Duhamel et al. [42] presented a tabu search heuristic for the VRPBTW. They focused on the minimization of fleet sizes first and the minimization of schedule times (which include travel times, service times, and waiting times) next while Thangiah et al. [41] considered the minimization travel times only as the second goal. An initial solution was generated by using insertion heuristic [16]. To improve the solution, tabu search and local search and enhance algorithms were applied. The tabu search heuristic is tested on problems where customers were distributed normally over the service area. For the computational results, the solutions of this method were within 0.5% of the optima on average, and better than GA [21] and PFIH [41].

Reimann et al. [43] presented the insertion based ant system for the VRPBTW. The core of this algorithm was the incorporation of insertion heuristic [16] as the solution construction method within the ant system. The swap and move operators were applied as the solution improvement. The computational result showed that this algorithm outperforms a custom-made heuristic proposed by Thangiah et al. [41].

Zhong and Cole [44] presented a guided local search heuristic (GLSA) to solve the VRPBTW. The algorithm was divided into two phases based on the idea of a cluster-first route-second algorithm. For the first phase, an initial solution was generated by sweep algorithm, and then a guided local search heuristic (2-opt, 1-move, and 1-exchange) was used to improve the solution. For the second phase, a new strategy called section planning was applied which inserted new routes until a feasible solution was obtained and arranged customers within routes to decrease the total distance. In this phase, the feasibility constraints were soft in early iterations and hard later. For experiment results, although GLSA underperformed GA [21] algorithm for VRPBTW problems, GLSA did find a better solution than GA [21] for some instances.

Pisinger and Ropke [45] proposed a unified heuristic called ALNS to solve several variants of the VRP including the VRPBTW. The VRPBTW was transformed into the VRPB while the routes were ordered according to time window constraints. The results showed that it can improve 183 best known solutions out of 486 benchmark tests especially in large problems.

Aghdaghi and Jolai [46] presented a goal programming approach and a heuristic algorithm to solve the vehicle routing problem with backhauls and soft time windows (VRPSBTW). The different between the soft time window and the hard time window was the lower and upper bounds of the time window was not necessary to be met, but could be violated with the penalty. The proposed heuristic was a two-phase algorithm based on the idea of a cluster-first route-second algorithm. During the first phase, the partition sets of customers, called zones, were created. In the second phase, the feasible routes were generated by using input data from the first phase. For computational results, this algorithm results were close to the optimal results for some instances.

Liu et al. [47] presented a genetic algorithm and a tabu search method to solve the vehicle routing problem with mixed backhauls and time windows (VRPMBTW). For VRPMBTW, a vehicle could serve linehauls and backhauls in a mixed order. The algorithms were tested on benchmark problems and better than the best-known solutions in the literature.

Küçükoğlu and Öztürk [48] proposed a differential evolution algorithm which is similar to a genetic algorithm. The main difference between the genetic algorithm and the differential evolution algorithm was the process of creating the improved solution: genetic algorithm relied on crossover while differential evolution relies on mutation. The results showed that this algorithm could obtain some new best known solutions. However, this algorithm lost to more than half of the current best known solutions for the large problems. Later, Küçükoğlu and Öztürk [49] proposed an advanced hybrid meta-heuristic algorithm for VRPBTW. This algorithm was a structured combination of simulated annealing which helped to escape from local optima and tabu search which helped to avoid cycling. The algorithm was tested on the benchmark set of Gélinas et al. [37]. The results indicated that this algorithm had superior performance compared with the existing algorithms in the literature. However, one of the disadvantages of the hybrid algorithm was that it took a lot of computational time.

2.2 Solution Approaches to Vehicle Routing Problem

2.2.1 Exact Approaches

Since the VRP is an NP-Hard problem, it is complicated to find the optimal solution especially for large problems. However, various exact methods have been developed to solve this problem. The exact methods for the VRP is divided into three categories, namely Lagrange relaxation-based methods, column generation, and integer programming.

2.2.1.1 Lagrange Relaxation

Lagrange relaxation method is a method which relaxes the original problem by reducing some hard inequality constraints and adding Lagrange multiplier in the objective function to penalize violations of the constraints. Generally, the relaxed problem is easier than the original one.

Koul and Madsen [50] introduced an optimization algorithm based on the Lagrange relaxation for solving VRPTW. The constraints that ensured each customer was served exactly once are relaxed. The Lagrange multiplier was added to the objective function to enforce that every customer was serviced once. This algorithm could solve several previously unsolved problems with competitive computational times.

Fisher et al. [51] presented two optimization algorithms for solving VRPTW. The problem was formulated as a K-tree problem with degree 2K on the depot and degree 2 on the customers. The relaxation constraints were the constraints which ensured that exactly one vehicle visits and leaves each customer. The problem was solved by a K-tree relaxation and a Lagrangain decomposition with variable splitting where the problem was divided into two sub-problems, namely a series of shortest path problems and a semi-assignment problem. This method was tested on the Solomon benchmark problems [16] with up to100 customers and it was very effective with clustered problems.

2.2.1.2 Column Generation

Column generation is an efficient method to solve a larger linear programming problem. The algorithm splits the problem into two problems: the (restricted) master problem and the sub-problem. The (restricted) master problem is the original problem which considers only a subset of variables. After the master problem is solved, the information of dual prices for each of the constraints in the master problem is utilized in the objective function of the sub-problem to identify a new entering variable. When the sub-problem is solved, the negative reduced cost variables are added to the master problem. The master problem is re-solved to obtain new information of dual prices for the re-solving sub-problem. This process is repeated until no negative reduced cost variables are identified.

Agarwal et al. [52] proposed an exact algorithm based on the set-partitioning formulation for VRP. The column generation was applied for solving this problem. The results showed that the optimal results were slightly different from optimal solution because all distance data in this program were represented in integer.

Desrochers et al. [53] proposed the column generation approach for VRPTW for the first time, and later the improved version of the same model was presented by Desrochers et al. [54]. They concluded that the column generation approach is capable of solving large problems.

2.2.1.3 Integer Programming

Integer programming is a technique which can be used to solve a complex problem by breaking it down into a number of sub-problems. In this way, the optimal solutions of a large problem can be obtained from the smaller sub-problems. Branch and bound technique is a general algorithm for solving various discrete optimization problems. It consists of a systematic enumeration of all candidate solutions which formed as a rooted tree. The subsets of solution are branches of the tree. Each branching solution is checked by lower and upper estimated bound. If the solution is not better than the old one, it will be discarded.

The name “branch and bound” first appeared in the work of Little et al. [55] and used to solve the traveling salesman problem. All feasible solutions are divided into

small subsets. This process is called branching. Then a lower bound on the length (objective value) of the feasible solutions is calculated therein in each subset. This process repeats until a subset contains a single feasible solution whose length is less than or equal to some lower bound for every feasible solution. Their method can reasonably solve the extended problem size without using special techniques.

Christofides and Eilon [56] proposed a branch and bound method for the CVRP. They suggested using a shortest spanning 1-tree bound instead of computing bound at every node of the search tree. The efficacy of this technique was tested by two CVRP problems: a 6-city problem and a 13-city problem. They compared three approaches, namely the branch-and-bound approach, the savings approach, the 3-optimal tour method. The result reported that the 3-optimal tour method was superior to the other two methods.

An exact branch-and-bound algorithm was proposed by Fischetti et al. [57] for solving an asymmetric capacitated vehicle routing problem. They presented the two new additive approach bounding procedures, namely ADD_DISJ and ADD_FLOW. Each procedure computed a sequence of non-decreasing lower bounds by solving different relaxation problems. For branching technique, they adapted of the well-known scheme called subtour elimination [58]. Their method was tested on both real-world and random test problems and compared with the previous algorithms from the literature. The result showed that all instances were solved to optimality by this algorithm within acceptable computing time.

Baldacci and Mingozzi [59] proposed a new branch-and-cut algorithm for the CVRP to find the optimal solution. A branch-and-cut algorithm was the method which was applied the cutting plane to decrease the feasible solution space in the linear programming relaxations while running a branch and bound algorithm. The computational results showed the instances are solved to optimality by this method, and it obtained new lower bounds which were better than the best lower bounds reported in the literature.

Cordeau [60] presented a branch-and-cut algorithm for the dial-a-ride problem which minimized the vehicle routes while preserving the capacity, time window, and ride time constraints. For the computational results, the branch-and-cut method could solve small- and medium-sized problems. It explored fewer nodes and uses less

computational time than the branch-and-bound method appearing in the version 8.1 of CPLEX.

Dell'Amico et al. [61] introduced a branch-and-price algorithm for the vehicle routing problem with simultaneous distribution and collection for the first time. A branch-and-price was a hybrid method between branch and bound and column generation methods. Each branch of a tree was applied with a column generation method to improve the linear programming relaxation. The bi-directional search and bounded number of steps were used to enhance the algorithm performance. Various branching strategies were also studied. In computational experiments, the exact programming and state space relaxation were compared in small- and medium-sized problems. They concluded that branch and price was a practicable approach to solve the vehicle routing problem with simultaneous distribution and collection for small- and medium-sized problems.

Gutiérrez-Jarpa et al. [62] presented a branch-and-price algorithm for the VRP with deliveries, selective pickups and time windows. The algorithm was an extension version of the algorithm proposed by Dell'Amico et al. [40]. This algorithm was applied to solve five variants of the problem, namely single demands with mixed routes, single demands with backhauls, combined demands with single visits, combined demands with multiple visits and mixed routes, and combined demands with multiple visits and backhauls. They found the optimal solutions for the instances containing up to 50 customers.

Ropke and Cordeau [63] introduced a new branch-and-cut-and-price algorithm for the pickup and delivery problem with time windows. A branch-and-cut-and-price algorithm was similar to branch-and-price but cutting planes were also applied to tighten LP relaxations within a branch-and-price algorithm. An elementary and a non-elementary shortest path problems were considered in the column generation algorithm as the pricing sub-problems. The results of adding valid inequalities indicating the 2-path cut was the most successful among the valid inequalities tested. Computational experiments showed the branch-and-cut-and-price algorithm outperforms a recent branch-and-cut algorithm.

Pessoa et al. [64] presented robust branch-cut-and-price algorithms for vehicle routing problems. This robust version would not change the structure of the pricing sub-

problem and kept its pseudo-polynomial pricing complexity. They were quite successful on current typical instances up to 100 customers.

2.2.2 Heuristic Approach

A heuristic method is simply an algorithm which is sufficient to find immediate solutions but may not guarantee the optimal solution. This method is suitable for solving a complex problem because it speeds up the process and obtains satisfactory solutions.

2.2.2.1 Two-phase heuristics

Two-phase heuristics divide a problem into two phases. The first phase is a cluster phase, which groups customers into subsets. Then, the second phase builds routes on each subset.

Gillett and Miller [65] introduced sweep algorithm for solving medium- and large-scale vehicle-dispatch problems. The polar-coordinate angle for each customer was used to build up each route by rotating the line centered at the depot in a circle. Each customer which touched the line was added to the route. A new route was started when the customer could not be added because of constraint violations. The results reported that the algorithm results were slightly better than Christofides and Eilon's results [56]. However, its computationally efficiency was slightly less so than Christofides and Eilon's [56].

Solomon [16] adapted the sweep method for solving VRPTW. The cluster phase of the algorithm was the same as the original version of the sweep algorithm [65] without scheduling. For the route phase, scheduling solution was created in each sector using some insert heuristic criteria. The computational results indicated that the algorithm was very efficient in terms of the quality for the cluster instance type, but other types were quite disappointing.

Fisher and Jaikumar [66] presented a generalized assignment heuristic for vehicle routing problems based on the idea of two-phase process. The algorithm started by clustering the customers using generalized assignment problem. It then created the routes by applying means of a travelling salesman problem algorithm on each cluster.

The results indicated that it outperformed the best existing heuristics on a sample of standard test problems.

Another heuristic called route first–cluster second was presented by Beasley [67]. The algorithm first formed a giant tour in the route phase to ensure that the customers who were in close proximity to one another were also near one another in the giant tour. In the clustering phase, an algorithm for the shortest path problem was applied. They found that five out of ten problems were the great solutions which three of them are optimal solutions.

2.2.2.2 Constructive heuristics

Constructive heuristics is a method which creates a solution by repeating some processes until the solution is completed.

Clarke and Wright [68] introduced the saving heuristic for scheduling vehicles from a central depot to a number of delivery points. In this heuristic, the saving value between any two customers was defined as the distance that was saved due to putting them in the same route instead of having them in separate routes. The routes with largest saving value were connected if they did not violate the constraints. The process would be repeated again and again. This algorithm was simple because no parameters are required. It also speeded up the selection of an optimum or near-optimum route.

Solomon [16] presented the nearest-neighbor heuristic. It started by adding an unassigned customer which was closest to the depot. Then, the next unassigned customer which was closest to the previous customer was added to the route if the constraints were still preserved. If this violates the constraints, the heuristic started a new route. The closeness in this paper was computed by the following formula: $closeness_{ij} = \alpha c_{ij} + \beta h_{ij} + \gamma v_{ij}$, where $\alpha + \beta + \gamma = 1$, $\alpha, \beta, \gamma \geq 0$, c_{ij} denoted the distance expressed as time from customer i to customer j, h_{ij} denoted the idle time before servicing customer j after customer i, and v_{ij} denoted the urgency of delivery to customer j after customer i expressed as the time remaining until the vehicle's last possible service started for customer j. In addition, the insertion heuristic was also introduced as a constructive heuristic. There were three types of insertion criteria. The

insertion criteria type (i) focused on minimizing the extra distance and extra time required after insertion. The insertion criteria type (ii) focused on minimizing total distance and time. The last insertion criteria type (iii) focused on the urgency of servicing a customer. Moreover, the saving heuristic [68] was adapted in this method to handle VRPTW. The computational results indicated that the insertion heuristic type (i) showed excellent performance both in terms of quality and computational time. However, type (ii) and (iii) did not perform well in general and the results were quite disappointing.

Potvin and Rousseau [69] presented insertion heuristic for the vehicle routing and scheduling problem with time windows (VRSPTW). The heuristic was a parallel version of Solomon's insertion heuristic type (i) [15]. The measure of a gap between the best and the second best insertion places for a customer overall routes was a criterion for choosing the next customer to be inserted on the route. The Computational results reported that the parallel approach was better than the sequential approach all instances especially on pure clustered problems.

Balakrishnan [70] introduced three heuristics which were based on nearest-neighbor and Clarke-Wright savings algorithm [68] for the vehicle routing problem with soft time windows (VRPSTW). Each heuristic differed only in the way the first customer was chosen for a route and the way the next customer was chosen during the route construction. The different measures were used to compare the cost of waiting and potential penalty which was a linear function of the amount of time window violation. The best solutions from the three heuristics had smaller number of vehicles used and total distances than the solutions from other methods in the literature.

Dullaert [71] introduced a new time insertion criteria to solve the VRPTW with relatively few customers per route. The results indicated that the algorithm saved the cost but it increased the number of customers per route. They concluded that the algorithm can improve the quality of the heuristic for short-routed VRPTW.

Ioannou et al. [72] presented a greedy look-ahead heuristic for the VRPTW. They introduced new criteria for choosing and inserting customers which were based on a greedy look-ahead heuristic which proposed by Atkinson [73]. The results showed that the algorithm suits the problems which required the number of vehicles used, and daily real-life scheduling problems.

Pang [74] presented an adaptive parallel route construction heuristic for the VRPTW. The heuristic was motivated from nearest-neighbor heuristic [16] which considered three cost factors: distance, urgency and waiting cost. The weights of the cost factors were adjusted based on the problem characteristics which were randomly uniform distribution, clustered distribution, and the mixture of random and clustered distribution. The results indicated that the algorithm was useful for the initial route construction.

The dynamic nearest neighbor policy for the multi-vehicle pick-up and delivery problem was presented by Sheridan et al. [75]. They introduced new policy that maintained closest customer-to-vehicle assignments because of its ability to divert/re-assign vehicles when another vehicle became available or a new customer call arrived. The results showed that this algorithm outperformed the existing NN, and it could minimize the longest customer waiting times in realistic scenarios.

A nearest neighbor heuristic is one of classical route construction heuristics which is easy to implement and fast to execute. The algorithm starts by adding the closest unassigned customer to the depot into the route and then repeats adding the next closest unassigned customer until some constraint is violated. If it fails to add any customers into the route because of some infeasibility, it will start a new route from the depot and continue the process until all customers are scheduled on some routes.

Solomon [17] proposed a time-oriented nearest-neighbor heuristic that considered both the capacity constraints and the time window constraints. In this approach, he computed the "closeness" from three factors, namely the direct distance between the two customers, the urgency of the delivery of the next customer, and the time remaining until the vehicle's last possible service starting.

Küçükoğlu and Öztürk [49] also applied nearest-neighbor algorithm, called improved nearest-neighbor heuristic, as a constructive heuristic to solve VRPBTW. This heuristic was computed the closeness using the same three factors as a time-oriented nearest-neighbor heuristic proposed by Solomon [16]. This is an inspiration to propose new algorithms based on nearest-neighbor heuristic in this dissertation, namely a nearest urgent candidate heuristic (NUC), in which all customers are ordered according to the urgency of their delivery, and a nearest neighbor with roulette wheel

selection (NNRW) method which is a combination of a roulette wheel selection method and the improved nearest-neighbor heuristic.

2.2.3 Metaheuristic Approach

A metaheuristic method is the optimization technique that explores a larger area of the solution space to achieve good optimization results. It has proven to be the methods of choice for many researchers to get an approximate, and near-optimal solutions. The main difference between metaheuristic and heuristic is the metaheuristic is a high-level problem-independent algorithm which guides the search process while heuristic is a problem-dependent algorithm.

Metaheuristic methods are divided into three types, namely local search (e.g. tabu search, simulated annealing), population search (e.g. cuckoo search, artificial bee colony), and learning mechanism (e.g. neural network, swarm intelligence). Two metaheuristics, namely cuckoo search (CS) algorithm, and artificial bee colony (ABC) algorithm, are used to solve VRPBTW in this dissertation. Thus, only the literature reviews on the two methods will be discussed.

CS is a metaheuristic method introduced by Yang and Deb [25]. It is inspired from aggressive breeding behavior of cuckoo birds. Although this algorithm was originally designed for solving continuous problem, a hybrid cuckoo search algorithm with greedy randomized adaptive search procedure was first proposed by Zheng et al. [26] to solve discrete problems like VRP. A path relinking strategy which was used as a way of exploring trajectories between high-quality solutions was applied to CS instead of Lévy flight in the original CS. Moreover, swap and inversion strategies were also used in a local search. The results showed that this algorithm was effective for solving the VRP. However, the computational time of the algorithm increased significantly in the large-scale problems.

ABC is inspired by the intelligent food source finding behavior of the honey bees around the hives and was proposed by Karaboga [27]. It was firstly applied to the CVRP by Szeto et al. [28] with some enhancements. The results showed that the enhanced version of ABC algorithm outperformed the original one, and it could produce good solutions when compared with the existing heuristics. Alzaqebah et al.

[29] presented the modified ABC for the VRPTW. In this study, the list of abandoned solutions was used to generate new solutions. The results showed that the modified ABC algorithm obtained good results when compared with the best-known results. An improved artificial bee colony algorithm for a real case in Dalian was introduced by Yu et al. [30]. In this version of ABC algorithm, three strategies were applied, namely an adaptive strategy, a crossover operation, and a mutation operation. The results showed that some solutions were better than the best-known solutions when tested on benchmark problems [16] for VRPTW.

There are many reasons that motivate us to apply the CS and ABC algorithm to solve VRPBTW in this dissertation. Firstly, these algorithms were successfully applied to VRP as described [26, 28-30]. Secondly, these algorithms are metaheuristics, which means the exploring area of the solution space is larger than non-metaheuristics (e.g. PFIH, unified heuristic). Thus, they can achieve good optimization results, especially in the large-sized problems. Thirdly, these algorithms are a population-based heuristic which starts with a number of initial solutions. Therefore, they can explore more in the solution space and have more chance to obtain the better solutions than non-population-based heuristic (e.g. HMA). Moreover, a population-based heuristic can be enhanced with parallel computing or distributed computing. Finally, these algorithms can prevent the search from premature convergence problem which is the weakness of other population-based heuristics (e.g. GA and DEA). This is because, for ABC algorithm in the scout bee stage, the stalled solutions are removed from the population and new randomly generated solutions are added to the population; and, for CS algorithm, the solutions are abandoned with a probability and then completely new solutions are built. This process also amplifies global search capability.

To the best of our knowledge, CS algorithm had never been applied to VRPBTW. Thus, we propose a CS algorithm for VRPBTW in this dissertation. Although ABC algorithm was successfully applied to several variations of the VRP [28-30], there are a few studies [31-32] that apply ABC for solving VRPBTW. Tuntitippawan and Asawarungsaengkul [31] applied ABC to small and medium problems and Tuntitippawan and Asawarungsaengkul [32] applied ABC to small, medium, and large problems. However, the results showed that it still underperformed the existing heuristics in many instances, especially in the large-scale problems. It is

necessary to extend the exploration on the solution space or, equivalently, to expand the capability of the neighborhood search. Therefore, we propose the enhanced artificial bee colony algorithm (EABC) by applying a forbidden list strategy to prevent duplicated initial solutions (which initially extends the exploration on the solution space), the sequential search strategy for onlookers to explore the neighborhood near the high-quality food source, and the intra-route and inter-route exchange combination strategy to obtain the better solutions.

CHAPTER 3
THE VEHICLE ROUTING PROBLEM WITH BACKHAULS AND TIME WINDOWS

3.1 Problem Description

The vehicle routing problem (VRP) is a well-known combinatorial optimization problem designed to find the minimum distance or fleet size required to satisfy the demands located at a set of geographically dispersed customers from one or more depots. Many variations of VRP have been formulated by applying constraints to add realism such as capacity, time windows, pickup and delivery, cross-docking, and backhaul constraints. In this dissertation, we focus on the VRP with backhaul and time window (VRPBTW). Here, customers either require items to be delivered from the depot (linehauls) or they need items returned to the depot (backhauls). Moreover, there are restrictions on the times that a vehicle can arrive at the customers. The VRPBTW has three main constraints: 1) the capacity constraints where the total demand in each vehicle does not exceed its capacity, 2) the time window constraints where each vehicle arrives at each customer within the customer's specified time window, and 3) the backhaul constraints that ensures linehauls customers are served before backhauls customers.

3.2 Mathematical Model

We propose a mathematical model for VRPBTW which are modified from mathematical formulation for fleet size and mixed vehicle routing problem with backhauls (FSMVRPB) proposed by Salhi et al. [76] and vehicle routing problem with backhauls and time windows (VRPBTW) presented by Küçükoğlu and Öztürk [49]. The VRPBTW can be formulated into a mixed-integer program model as follows. (Note that the depot is considered a node indexed by 0.)

Notations:

L = number of linehaul customers (indexed by $1,\ldots,L$).

B = number of backhaul customers (indexed by $L+1,\ldots,n$).

n = total number of customers ($L+B$).

K = number of vehicles (indexed by $1,\ldots,K$)

u^k = capacity of vehicle $k : k \in \{1,\ldots,K\}$.

d_i = demand of customer i.

c_{ij} = distance between node i and node j.

a_i = earliest arrival time at customer i.

b_i = latest arrival time at customer i.

s_i = service time for customer i.

t_{ij} = travel time between node i and node j.

M = a large scalar.

$T_{\max}$ = maximum route time allowed for every vehicle.

φ_{ij} = the vehicle load on the arc from customer i to customer j.

w_i^k = service start time of vehicle k for customer i. (w_0^k means the time that vehicle k returns to the depot.)

x_{ij}^k = 1 if vehicle k travels from customer i to customer j, 0 otherwise.

The model:

$$\text{Min} \quad \sum_{k=1}^{K}\sum_{i=0}^{n}\sum_{j=0}^{n} c_{ij}x_{ij}^k \tag{1}$$

Subject to

$$\sum_{i=0}^{n}\sum_{k=1}^{K} x_{ij}^k = 1 \ , \ j = 1,\ldots,n \tag{2}$$

$$\sum_{j=0}^{n}\sum_{k=1}^{K} x_{ij}^k = 1, \ i = 1,\ldots,n \tag{3}$$

$$\sum_{i=0}^{n} x_{ip}^k = \sum_{j=0}^{n} x_{pj}^k \ , \ k = 1,\ldots,K \ \ p = 0,\ldots,n \tag{4}$$

$$\sum_{i=0}^{L} \varphi_{ij} = \sum_{l=0}^{n} \varphi_{jl} + d_j, \qquad j = 1,\ldots,L \tag{5}$$

$$\sum_{l=L+1}^{n} \varphi_{jl} + \varphi_{j0} = d_j + \sum_{i=1}^{n} \varphi_{ij}, \ j = L+1,\ldots,n \tag{6}$$

$$\varphi_{ij} = 0, \ i = 0,\ldots,L \ \ j = 0 \ \& \ j = L+1,\ldots,n \tag{7}$$

$$\varphi_{ii} = 0, \ i = 0,\ldots,n \tag{8}$$

$$\sum_{i=L+1}^{n} \varphi_{i0} = \sum_{i=L+1}^{n} d_i \tag{9}$$

$$\sum_{j=1}^{L} \varphi_{oj} = \sum_{j=1}^{L} d_j \tag{10}$$

$$x_{ij}^{k} = 0, i = L+1,\ldots,n; \ \ j = 1,\ldots,L; \ k = 1,\ldots,K \tag{11}$$

$$\varphi_{ij} \le \sum_{k=1}^{K} x_{ij}^{k} u^{k}, \ i \ne j = 0,\ldots,n \tag{12}$$

$$t_{0j} - w_j^k \le M(1 - x_{0j}^k), \ j = 1,\ldots,n; \ k = 1,\ldots,K \tag{13}$$

$$w_i^k + s_i + t_{i0} - w_0^k \le M(1 - x_{i0}^k), \ i = 1,\ldots,n; \ k = 1,\ldots,K \tag{14}$$

$$w_i^k + s_i + t_{ij} - w_j^k \le M(1 - x_{ij}^k), \ i = 1,\ldots,n \ ; \ j = 1,\ldots,n; \ k = 1,\ldots,K \tag{15}$$

$$a_i \le w_i^k \le b_i, \ i = 1,\ldots,n; \ k = 1,\ldots,K \tag{16}$$

$$0 \le w_0^k \le T_{\max}, \ k = 1,\ldots,K \tag{17}$$

$$x_{ij}^k \in \{1,0\}, \ i = 0,\ldots,n; \ j = 0,\ldots,n; \ k = 1,\ldots,K \tag{18}$$

$$\varphi_{ij} \ge 0, \ i = 0,\ldots,n; \ j = 0,\ldots,n \tag{19}$$

$$w_i^k \ge 0, \ i = 0,\ldots,n; \ k = 1,\ldots,K \tag{20}$$

In the above model, equation (1) is the objective function which refers to minimizing the total route distances. Constraints (2) and (3) ensure that each customer is visited exactly once and by one vehicle only. Constraint (4) guarantees that a vehicle leaves from the same customer it has entered. Constraint (5) confirms the precedence relationship and the delivery satisfies the demands of linehaul customers. Constraint (6) is also a precedence constraint which ensures that the backhaul pickups are satisfied.

For example, we assume that the next linehaul customer is A, its demand equals 25, and the vehicle load before serving customer A is 75. After servicing customer A, the vehicle load becomes 50. On the other hand, if A is a backhaul customer, the vehicle load becomes 100 after picking up the goods from customer A. Constraint (7) ensures that the load carrying from linehaul customers to backhaul customers or the depot is empty. Constraint (8) confirms that there is no load between the same customers. Constraint (9) guarantees that the total of the loads on vehicles returning from backhaul customers to the depot is equal to the sum of the demands of the backhaul customers. Constraint (10) ensures that the total of the loads on vehicles departing from the depot is equal to the sum of the demands of the linehaul customers. Constraint (11) prevents vehicles from going from a backhaul customer to a linehaul customer. Constraint (12) ensures that the vehicle load from customer i to customer j does not exceed the capacity of the vehicle going from node i to node j. Constraint (13) guarantees that if there is a vehicle from the depot to a customer, the travelling time between the depot and customer j must not exceed the start service time at node j. Constraint (14) states that, if a vehicle is traveling from node i to the depot, the arrival time at the depot must be greater than summation of the start service time at node i, service time at node i, and travelling time between node i and the depot. Constraint (15) states that, if a vehicle is traveling from node i to node j, the arrive time at node j must be greater than summation of the start service time at node i, service time at node i, and travelling time between node i and node j. Constraints (16) and (17) ensure that the time window requirements are satisfied. Finally, constraints (18), (19), and (20) define the restrictions on the decision variables.

We use this model to obtain the exact optimal solutions using CPLEX version 12.6. The exact solutions are used for comparison with heuristic solutions and metaheuristic solutions in terms of their quality and computational times.

3.3 Heuristic Approach

The VRPBTW belongs to the class of NPC (Nondeterministic Polynomial-time Complete) problems because it is an extension of the traveling salesman problem which has been shown to belong to the class of NPC problems [41]. This means that all known algorithms that define an optimal solution require exponentially increasing computational time as the number of customers increases. Therefore, heuristic methods which provide approximate solutions are justified and are required for realistic-sized problems. In this section, we introduce the heuristics which are based on nearest neighbor algorithm, namely a nearest neighbor heuristic (NN), an improved nearest neighbor heuristic (INN), a nearest urgent candidate heuristic (NUC), and a nearest neighbor with roulette wheel selection method (NNRW). Note that the NUC and NNRW heuristics are new proposed algorithms in this study.

3.3.1 Common Elements

The heuristics explained in this section share the following common elements: solution representation, quality measure of a solution, and neighborhood search.

3.3.1.1 Solution Representation

A solution of the VRPBTW contains the tour for each vehicle. A tour is a path from a depot to a subset of customers and then back to the depot. Assume there are N customers which are denoted by integers 1 through N, and the depot denoted by 0. Also, assume that there are K vehicles to deliver the products. A solution, which consist of K tours, is described by a vector of length (N+K+1) that contains each customer exactly once and (K+1) zeros that indicates the start of a new tour (a solution always starts with a zero). The sequence of integers between two zeros represents one vehicle tour in the solution. For example, if there are 6 customers (N=6) and 3 vehicles (K=3), a possible solution is represented by the vector (0, 1, 2, 0, 3, 4, 5, 0, 6, 0). Notice the length is 10 (N+K+1=10) and it means that the first vehicle serves customers 1 and 2, the second vehicle serves customers 3, 4, and 5, and the last vehicle serves only customer 6.

3.3.1.2 Quality Measure of a Solution

The quality measure of a solution is determined by the reciprocal of fitness function which is represented by a real number. In this dissertation, the fitness value of a solution is the total traveled distance where the distance between any customers a and b is the Euclidian distance between them, which is calculated by the following formula [50],

$$c_{ab} = \frac{\left\lfloor 10\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} \right\rfloor}{10}$$

where (x_i, y_i) is the Cartesian coordinate of customer i. This formula is designed to round down the distance to one decimal place. Therefore, the quality measure of a solution is represented by

$$\left(\sum_{k=1}^{k} (\text{total distance traveled by vehicle } k) \right)^{-1}$$

3.3.1.3 Neighborhood Search

In VRPBTW, a neighbor of a solution is generated by changing the order of visited customers. A neighborhood search (also known as a local search) is a heuristic method for finding neighbors of a solution that are feasible and have better quality. In this dissertation, the 1-move intra-route exchange (Chiang and Russell [12]) and the λ-interchange (Osman [8]) are used.

1-move Intra-route Exchange

The idea of 1-move intra-route exchange is to randomly remove one customer (linehaul or backhaul) from a route and insert it back to the same route in a different position. The solution is accepted if it can reduce the total cost while the capacity constraints, the time windows constraints, and the backhaul constraints are not violated. An example of 1-move is shown in Figure 1.

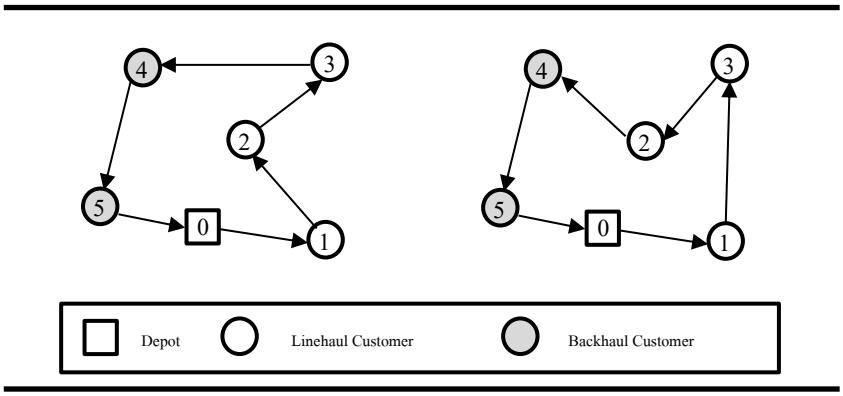


Figure 1 An Example of 1-move

λ -interchange

The λ -interchange local search heuristic is one of the best methods to solve the VRP problem types. The λ -interchange is an effective technique because it combines many methods such as insertion (randomly choose a customer from a route and then insert him/her in a different route), swap (randomly select two customers from different routes and then swap their positions), insert section (randomly choose a subset of customers from a route and then insert the subset in a different route), and swap section (randomly select two subsets of customers from different routes and then swap their positions).

The idea of λ -interchange is to interchange customers between routes where λ is a number of customers that are exchanged. In this method, the operator (λ_1, λ_2) on routes (p, q) is defined as exchanging λ_1 customers on route p with λ_2 customers on route q, where $\lambda_1, \lambda_2 \leq \lambda$. The customers on each route are selected either systematically or randomly. The improved solution is accepted if the total cost is decreased while maintaining the capacity feasibility, time window feasibility, and backhaul feasibility. In order to simplify the mechanism, we accept the first improved solution with the cost lower than the current solution. An example of the operator (1, 0) which removes customer 4 in the first route and then adds it in another route is given in Figure 2. This operator is similar to the insertion algorithm. As shown in Figure 3, the operator (1, 2) exchanges customer 4 in the first route with customer 8 and customer 9 in the second route. This operator is similar to the swap section algorithm.

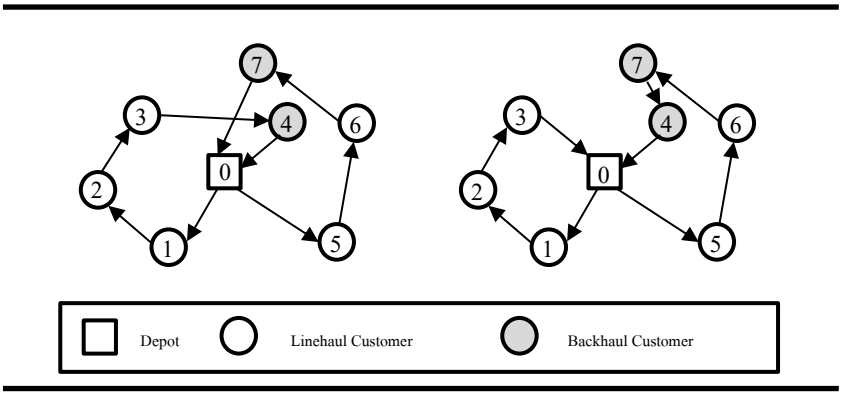


Figure 2 An Example of operator (1, 0)

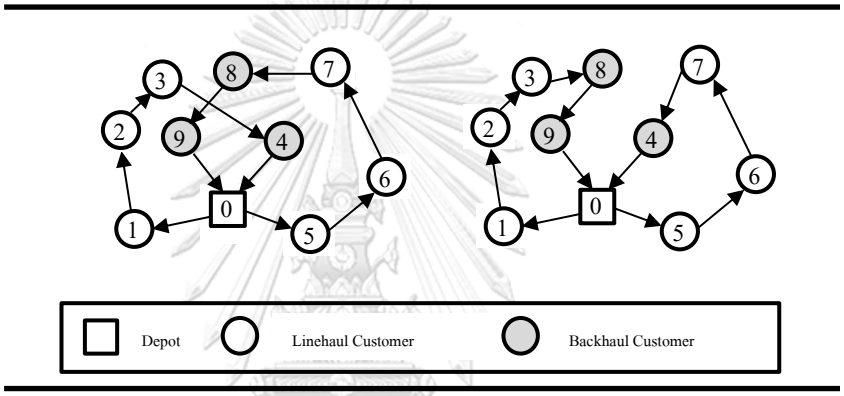
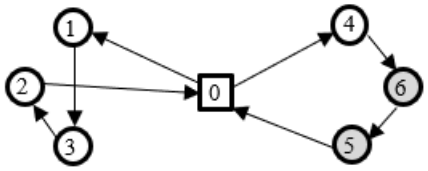
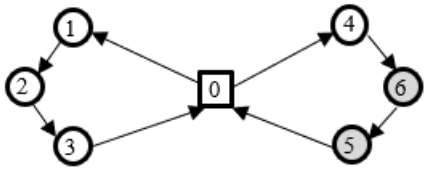


Figure 3 An Example of operator (1, 2)



Customer	Available Time	Demand
1	13.00-15.00	30
2	15.00-1700	30
3	15.00-17.00	30
4	10.00-12.00	50
5	13.00-14.00	40
6	14.00-15.00	10



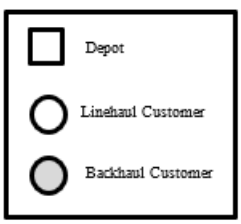


Figure 4 Example of case study

Since the λ-interchange is the method which interchange customers between routes, it cannot improve the solution in some cases. For example, in Figure 4, we assume that the first solution is represented by the vector (0, 1, 3, 2, 0, 4, 5, 6, 0) and the second solution which is a better solution compared with the first one is represented by the vector (0, 1, 3, 2, 0, 4, 5, 6, 0). When we try to apply λ-interchange to improve the first solution, we can notice that any operators of λ-interchange cannot be applied to improve solution without violating constraints. Thus, the first solution will be our result in stead of the second one which its total distance is less than the first one. To solve this problem, 1-move intra-route exchanges should be applied in this case. Hence, in this dissertation, we propose the intra-route and inter-route exchange combination strategy to obtain the better solutions.

3.3.2 Nearest Neighbor (NN) Heuristic

The nearest neighbor (NN) heuristic is one of the classic methods for solving the VRPBTW. This method finds a solution by choosing the closest customer to the last node to be next customer in the route while preserving the capacity, time windows, and backhaul feasibilities.

Main steps of NN

The steps of the NN algorithm for solving the VRPBTW problem can be described as follows:

Step 1 Build a new route by starting from the depot. Add the closest unassigned customer to the route.

Step 2 Consider the closest unassigned customer j to the currently assigned customer i to be next node in the route by checking the feasibility constraints. If they are not violated, the customer is added to the route. The closeness of customer i to customer j, denoted by $closeness_{ij}$, is set to be the reciprocal of $proximity_{ij}$, which is defined as: $proximity_{ij} = c_{ij}$, where c_{ij} denotes the Euclidean distance expressed as time from customer i to customer j.

Step 3 Repeat Step 2 until no more customers can be added to the current route, in which case the route is finished.

Step 4 If there remain unassigned customers, go back to Step 1. Otherwise, go to Step 5.

Step 5 Improve the solution by the 1-move intra-route exchange, and the λ -interchange where the algorithm is operated with $\lambda = 4$. The solution is accepted if the total cost is lower than the current best one while maintaining the capacity feasibility, time window feasibility, and backhaul feasibility.

Step 6 Repeat Step 5 until the number of iterations reaches the maximum, in which case the algorithm finishes.

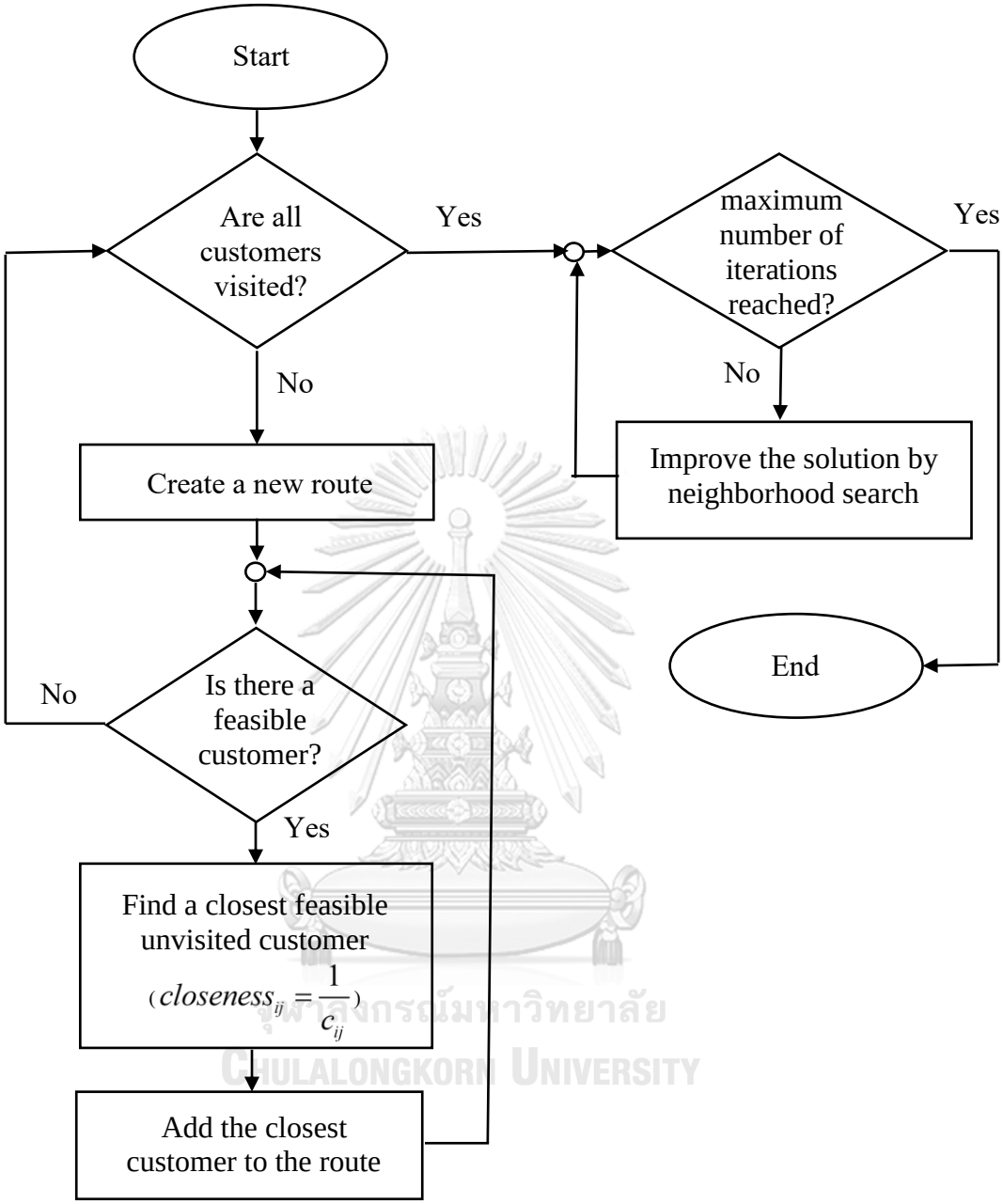


Figure 5 Flowchart of NN algorithm

3.3.3 Improved Nearest Neighbor (INN) Heuristic

Solomon [16] proposed a time-oriented nearest-neighbor heuristic that considers not only the capacity constraints but also the time window constraints for VRP with time window (VRPTW). In this approach, the closeness is computed from three factors, namely the direct distance between the two customers, the urgency of the delivery of the next customer, and the time remaining until the vehicle's last possible service start. This algorithm was later improved and applied to VRPBTW by Küçükoğlu and Öztürk [49]. It is called an improved nearest neighbor heuristic (INN). The closeness in INN is computed from the same three factors as a time-oriented nearest-neighbor heuristic proposed by Solomon [16]. However, it considers not only the capacity constraints and the time window constraints but also the backhaul constraints.

Main steps of INN

The steps of the INN algorithm for solving the VRPBTW problem can be described as follows:

Step 1 Build a new route by starting from depot. Add the closest unassigned in the route.

Step 2 Consider the closest unassigned customer j to the currently assigned customer i to be next node in the route by checking the feasibility constraints. If they are not violated, the customer is added to the route. The closeness of customer i to customer j, denoted by $closeness_{ij}$, is set to be the reciprocal of $proximity_{ij}$, which is defined as: $proximity_{ij} = \alpha c_{ij} + \beta h_{ij} + \gamma v_{ij}$, where $\alpha + \beta + \gamma = 1$, $\alpha, \beta, \gamma \geq 0$, c_{ij} denotes the distance expressed as time from customer i to customer j, h_{ij} denotes the idle time before servicing customer j after customer i, and v_{ij} denotes the urgency of delivery to customer j after customer i expressed as the time remaining until the vehicle's last possible service start for customer j.

Step 3 Repeat Step 2 until no more customers can be added to the current route, in which case the route is finished.

Step 4 If there remain unassigned customers, go back to Step 1. Otherwise, go to step 5.

Step 5 Improve the solution by 1-move intra-route exchange, and λ -interchange where the algorithm is operated with $\lambda = 4$. The solution is accepted if the total cost is lower than the current best one while maintaining the capacity feasibility, time window feasibility, and backhaul feasibility.

Step 6 Repeat Step 5 until the number of iterations reaches the maximum, in which case the algorithm finishes.

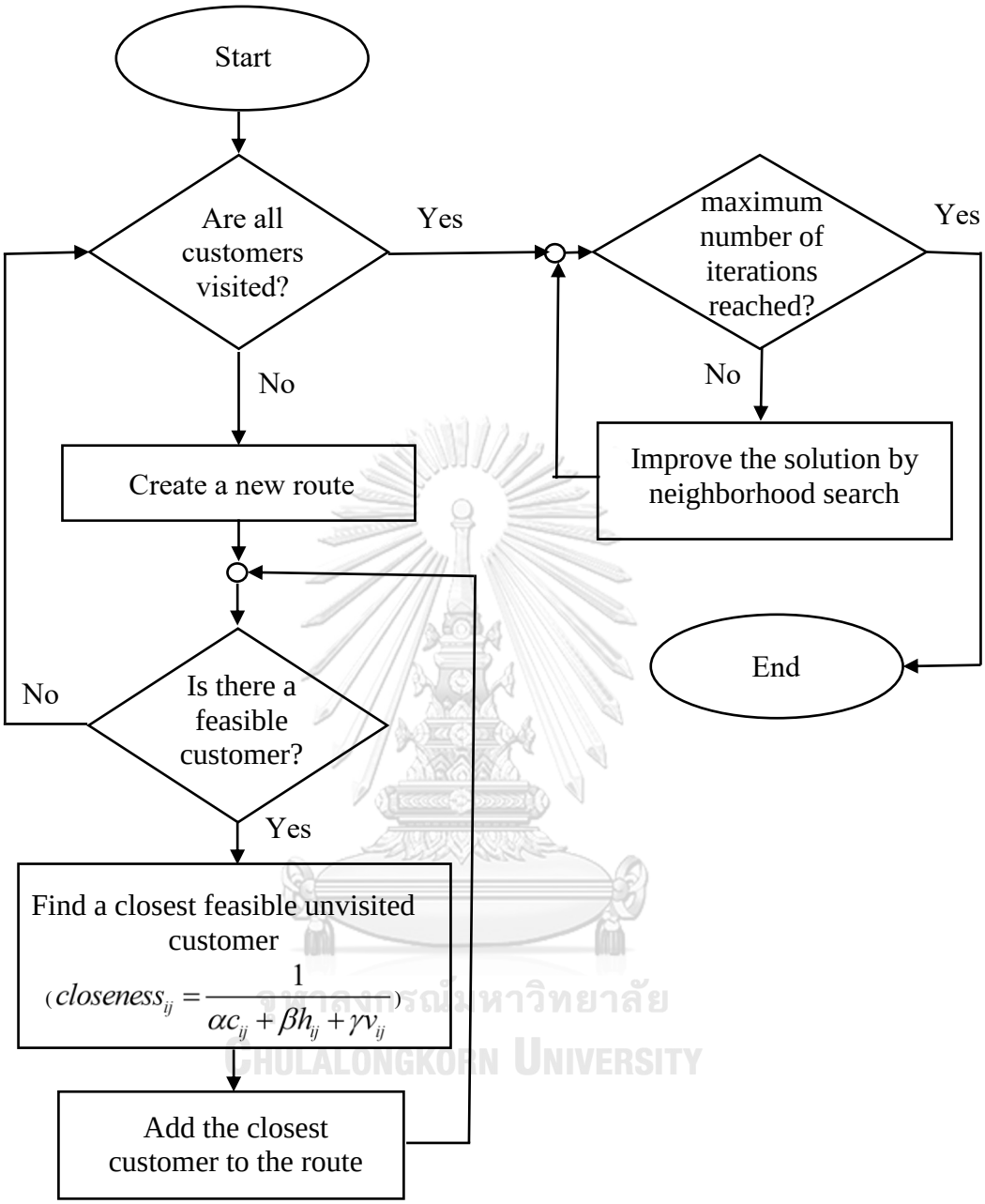


Figure 6 Flowchart of INN algorithm

3.3.4 Nearest Urgent Candidate (NUC) Heuristic

We propose a nearest urgent candidate (NUC) heuristic, in which all customers are ordered according to the urgency of their delivery. This idea comes from a common-sense management that the most urgent customer is served before the others. However, the cost, which consists of the traveling time and waiting time, must be taken into account as well. Thus, we use a candidate technique [14] to maintain our concept and reduce the cost in the same time. We speculate that a high quality initial solution will be obtained from this algorithm.

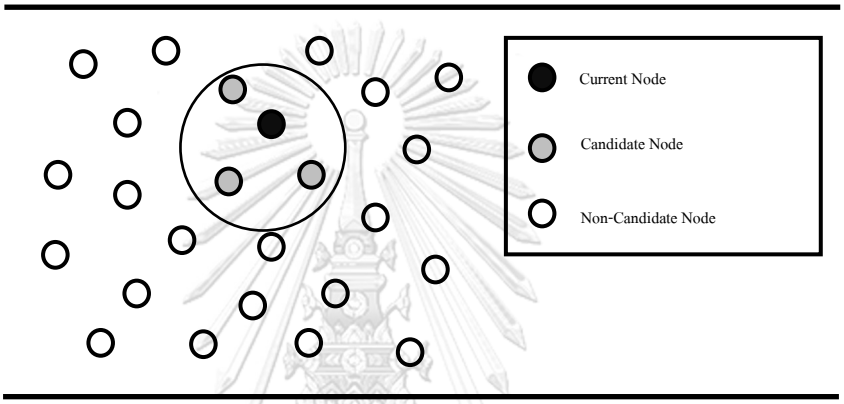


Figure 7 An Example of candidate list

NUC starts with sorting the customers by their urgency (latest arrival times) in ascending order. The customers that have urgent due times will be considered first but they must compete using their closeness to be assigned into the current route. Our version of closeness is computed from only two factors, which are the direct distance from the current customer i to the next customer j, c_{ij}, and the maximum idle time between servicing customers i and j, h_{ij}. Since we suppose that the traveling time between customer i and customer j are equal to the direct distance between them, we can assume the direct distance is the traveling time. Formally, $closeness_{ij} = \alpha c_{ij} + \beta h_{ij}$, where $\alpha + \beta = 1$, $\alpha \geq 0, \beta \geq 0$. The number of urgent customers allowed to compete must be limited. Otherwise, all customers can compete and the urgency becomes meaningless. Therefore, a candidate technique [14] is applied to solve this problem. In this technique, a candidate list of a fixed size is created. The list contains the chosen customers allowed to compete for the next node in the route. However, the proper size

of the list depends on the problem being considered. Figure 6 depicts an example of a problem with 25 customers and a candidate list of size 3.

Main steps of NUC

The steps of the NUC algorithm for solving the VRPBTW problem can be described as follows:

Step 1 Order all customers according to latest arrival time from the most urgency of delivery to the least. The NUC algorithm considers customers to add into a route by this order.

Step 2 Build a new route by starting from depot. Choose the first customer in the sequence to be the first customer in this route.

Step 3 Consider the next customer j in the sequence to be a potential next node in the route after the currently assigned customer i by checking the feasibility constraints. If they are not violated, the customer is added to candidate list. Consider the next customer in the sequence until the predetermined number of candidates is reached, or no further candidate is found. Compute the closeness value of each candidate j from the latest customer i in the route according to the following formula:

$$closeness_{ij} = \alpha c_{ij} + \beta h_{ij}$$

where $\alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$. The closest candidate is then selected as the next node to be visited in the route and also removed from the urgency sequence.

Step 4 Repeat Step 3 until no more customers can be added to the route, in which case the route is finished.

Step 5 If there are unassigned customers, go back to Step 2. Otherwise, go to Step 6.

Step 6 Improve the solution by 1-move intra-route exchange, and λ -interchange where the algorithm is operated with λ = 4. The solution is accepted if the total cost is lower than the current best one while maintaining the capacity feasibility, time window feasibility, and backhaul feasibility.

Step 5 Repeat Step 5 until the number of iterations reaches the maximum, in which case the algorithm finishes.

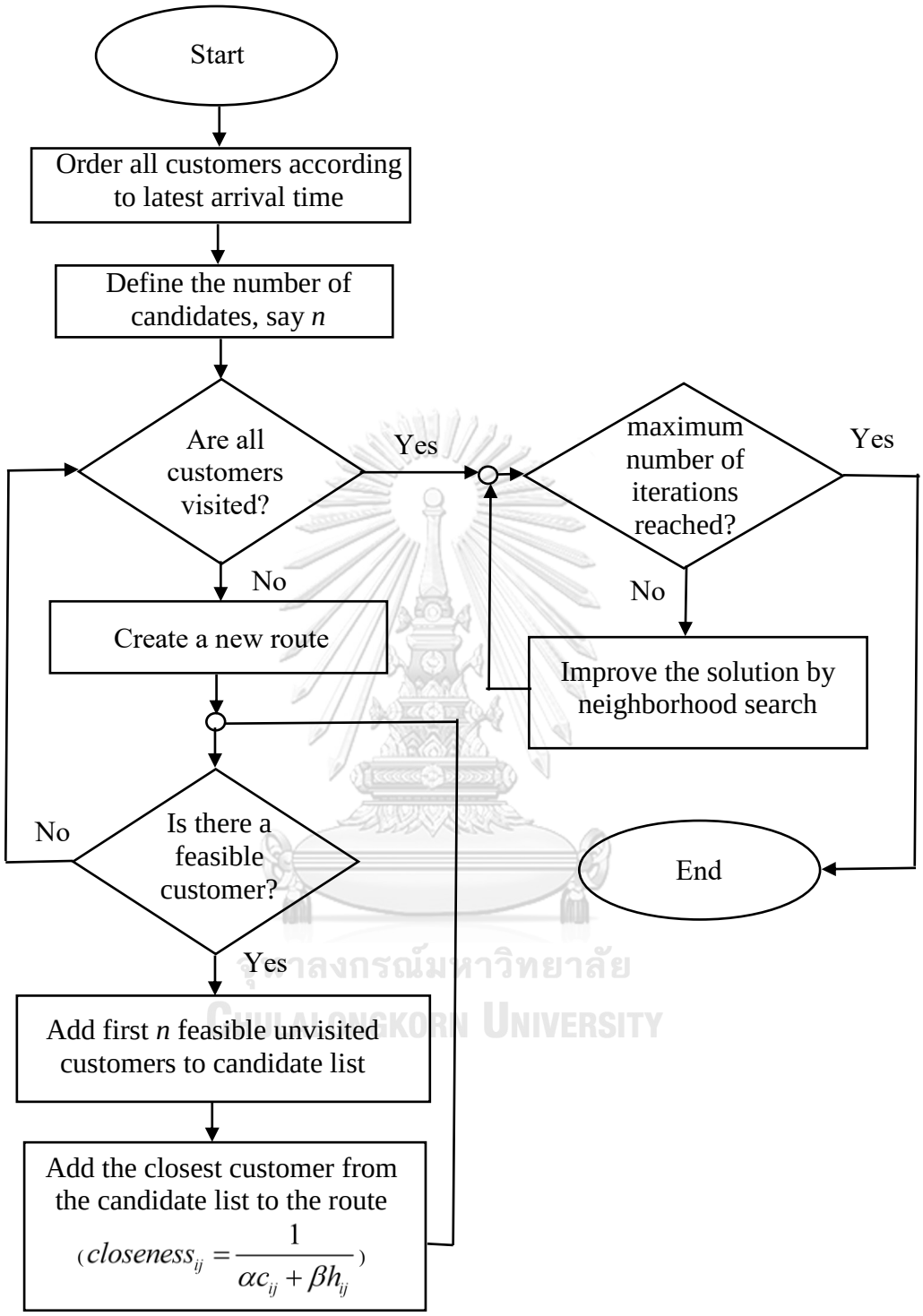


Figure 8 Flowchart of NUC algorithm

3.3.5 Nearest Neighbor with Roulette Wheel Selection (NNRW) Heuristic

We propose a nearest neighbor with roulette wheel selection (NNRW) method which is a combination of a roulette wheel selection method [77] and the INN heuristic [49] for generating the initial solutions. The idea of NNRW is to combine the advantage of INN, which finds the next customer by choosing the best closeness one, and roulette wheel selection, which finds the next customer by giving a chance to all customers with more probability for the customers with better closeness. Moreover, NNRW is a population-based heuristic. Therefore, it can explore more in the solution space and get more chance to obtain the better solutions than a heuristic with a single initial solution.

In this algorithm, the $closeness_{ij}$, which is the reciprocal of $proximity_{ij}$, is defined the same way the INN heuristic describes. The NNRW heuristic can be explained as follows.

During a route construction where customer i is our current customer, let p_j be the selection probability of customer j to be served next after customer i. Let U be the set of all unassigned customers with arbitrary order, say, $U = \{1, 2, \ldots, N\}$ where $N = |U|$. Then, for $j \in U$, p_j is calculated by:

$$p_j = \frac{closeness_{ij}}{\sum_{h \in U} closeness_{ih}}$$

We define $q_j = \sum_{h=1}^{j} p_h$ for $j \in U$. Then a random number r which ranges between 0 and 1 is selected for spinning the roulette wheel. If $r \le q_j$, then choose the first customer in U to be the next customer for the vehicle. Otherwise, if $q_{j-1} < r \le q_j$, then choose the j^{th} customer in U to be the next customer where $2 \le j \le |U|$. The assigned customers are discarded from U to prevent duplicate customers in a route.

The initial solution construction always starts a route with the depot, and then finds the next customer by the nearest neighbor with roulette wheel selection method. If the next customer violates the constraints (the capacity constraints, the time windows constraints, and the backhaul constraints), we spin the roulette wheel again to find a

new one. If the new one is still not feasible, we end this route and begin a new route. This process is repeated until all customers are served.

Main steps of NNRW

The steps of the NNRW algorithm for solving the VRPBTW problem can be described as follows:

Step 1 Build a new route by starting from depot. Set the current "customer" to be the depot.

Step 2 Find the next customer j to be the next node in the route after the currently assigned customer i by spinning the roulette wheel, which can be described as follows. Select a random number r which ranges between 0 and 1. Compute p_j and q_j according to the following formulas:

$$p_j = \frac{closeness_{ij}}{\sum_{h \in U} closeness_{ih}} \text{ and } q_j = \sum_{h=1}^{j} p_h \text{ for } j \in U$$

where U is the set of all unassigned customers, and $closeness_{ij}$ is defined the same way the INN heuristic describes. If $r \le q_1$, then choose the first customer in U to be the next potential customer for the route. Otherwise, if $q_{j-1} < r \le q_j$, then choose the j^{th} customer in U to be the next potential customer where $2 \le j \le |U|$.

Step 3 If the next potential customer is feasible, assign it to the route, delete the new assigned customer from U and go back to Step 2. Otherwise, go to Step 4.

Step 4 Repeat Step 2 one time to find a new next potential customer. If the new one is still not feasible, we end this route and go to Step 5. Otherwise, go to Step 3.

Step 5 If there remain unassigned customers, go back to Step 1. Otherwise, go to Step 6.

Step 6 Improve the solution by 1-move intra-route exchange, and λ -interchange where the algorithm is operated with λ = 4. The solution is accepted if the total cost is lower than the current best one while maintaining the capacity feasibility, time window feasibility, and backhaul feasibility.

Step 7 Repeat Step 6 until the number of iterations reaches the maximum, in which case the algorithm finishes.

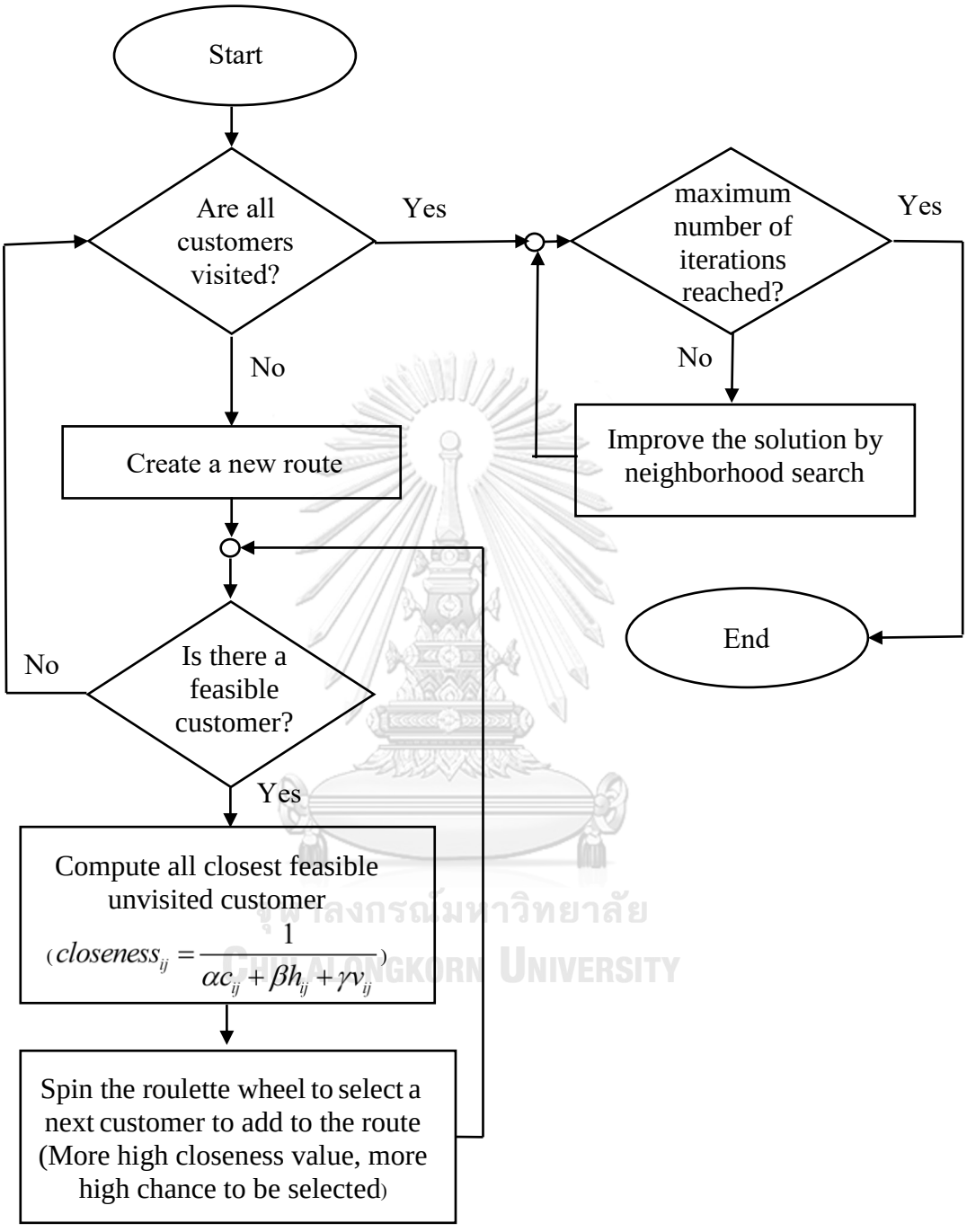


Figure 9 Flowchart of NNRW algorithm

3.4 Metaheuristic Approach

Bio-inspired intelligence known as metaheuristic methods is widespread for solving the class of NPC including VRPBTW. Metaheuristic algorithms, which are the optimization technique that explore a larger area of the solution space to achieve good optimization results with independence of the problem, have been proven to be the methods of choice for many researchers to get an approximate, and near-optimal in some cases, solutions. In this section, we present metaheuristics, namely a cuckoo search algorithm, and an artificial bee colony algorithm to solve the VRPBTW.

The common elements described in Section 3.3.1 (the solution representation, the quality measure of solution, and the neighborhood search) are also used in the following metaheuristics.

3.4.1 Cuckoo Search (CS) Algorithm

CS is a metaheuristic method introduced by Yang and Deb [27]. Inspiration of this algorithm is the parasitic spawn behavior of some cuckoo species. This algorithm was originally designed for solving continuous problem. Although discrete versions of CS have been applied to the travelling salesman problem (Ouaarab et al. [78]) and VRP (Zheng et al. [26]), to the best of our knowledge, it had never been applied to VRPBTW.

There are three reasons that we propose the CS algorithm for VRPBTW in this research study. First, To the best of our knowledge, CS algorithm had never been applied to VRPBTW. Second, the CS requires fewer parameters compared with other metaheuristics, so its solution is less affected by parameter tuning. The last reason is the CS has a process of generating new solutions which prevents the search from premature convergence problem.

3.4.1.1 The General Concept of CS

A cuckoo is an extraordinary bird because of its aggressive breeding behavior. The female cuckoos lay eggs in the nest of other host birds to let them hatch and brood young cuckoo chicks. If the host birds discover that the eggs are not theirs, they can

either get rid of the cuckoo eggs or abandon their nests and build new ones. However, some cuckoo species can mimic color and pattern of eggs in a few chosen host species to reduce chance of their eggs being abandoned. In addition, a cuckoo chick always mimics the call of the host chick to gain more feeding opportunity.

The cuckoo search starts by generating a number of host eggs (initial solutions) and assign them to nests. In the simplest approach, each nest can always have only a single egg. A cuckoo randomly selects a host nest and lays its egg (neighborhood search) into the nest. The aim is to replace a not-so-good solution with a new and better solution (cuckoo egg). A cuckoo egg will be abandoned and the host bird will build a completely new one (generating a new solution) when it discovers the egg is not its own. In summary, there are three ideal rules for this: (1) each cuckoo lays one egg at a time and selects a nest randomly; (2) the best nest with a high quality egg will be carried over to the next generation; (3) the number of host nests is fixed and a cuckoo egg is discovered by the host with a probability $p_a \in [0,1]$.

3.4.1.2 Main Steps of CS

The steps of the CS can be described as follows:

Step 1 Generate a set of initial solutions (host eggs) by the NNRW method (Section 3.3.4) and assign each egg to a host nest.

Step 2 Evaluate the fitness of each solution and record the global best solution.

Step 3 Choose randomly a host nest and then apply the neighborhood search on the host egg to generate a cuckoo egg. The host egg will be replaced with the cuckoo egg if the new cuckoo egg is better.

Step 4 With the probability p_a , abandon the worse nest and generate a new one.

Step 5 Update the global best solution if a solution has better quality than the current best one and go to Step 3. Otherwise, the algorithm ends and returns the global best solution in hand.

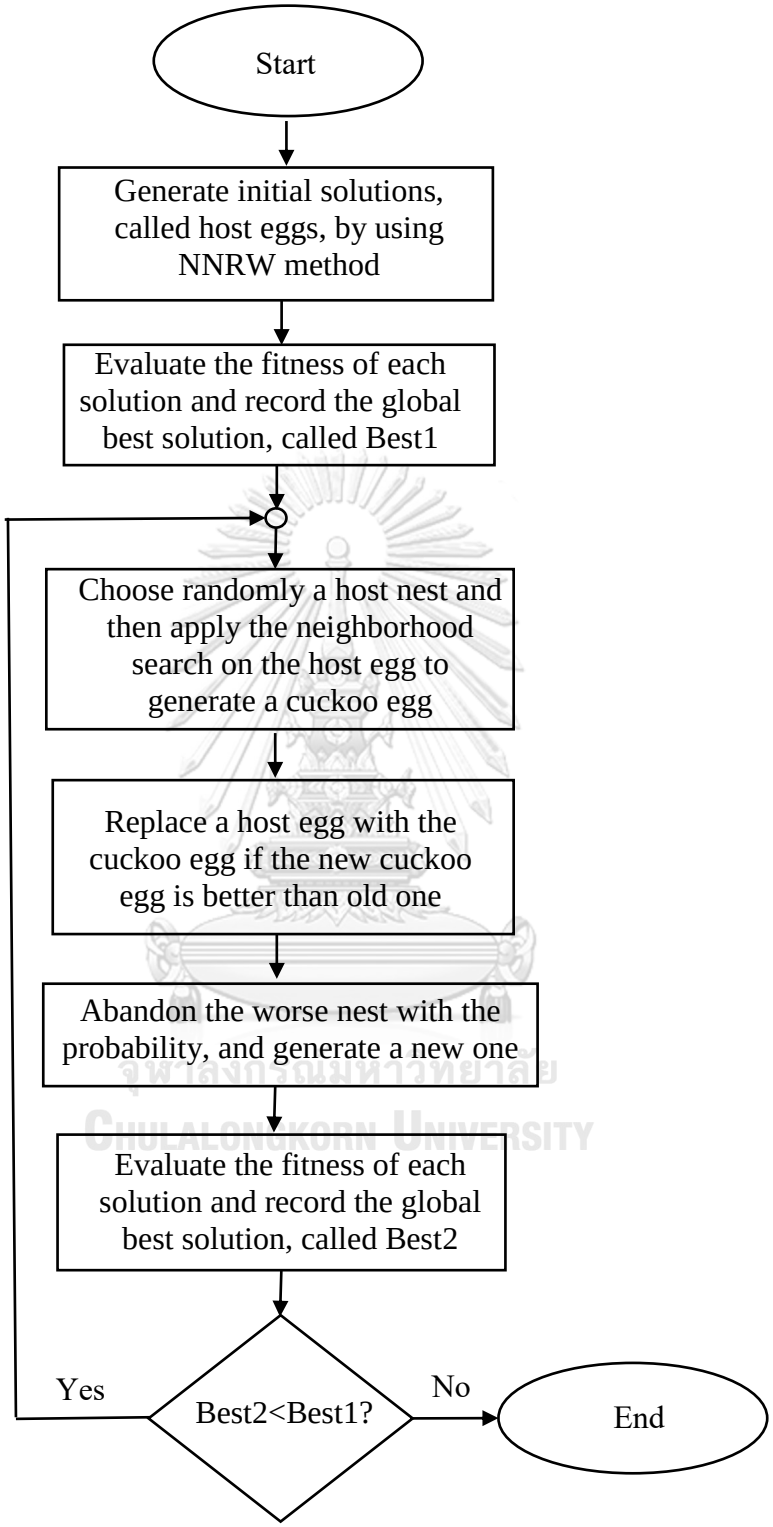


Figure 10 Flowchart of CS algorithm

3.4.2 Artificial Bee Colony (ABC) Algorithm

Artificial bee colony (ABC) algorithm is another metaheuristic method that has been applied to VRP. There are a few studies (Tuntitippawan and Asawarungsaengkul [31-32]) that apply ABC algorithm for solving VRPBTW. Hence, we propose an enhanced ABC algorithm by applying a forbidden list strategy to prevent duplicated initial solutions (which initially extends the exploration on the solution space), the sequential search strategy for onlookers to explore the neighborhood near the high-quality food source, and the intra-route and inter-route exchange combination strategy to obtain the better solutions.

There are three main reasons that the EABC is proposed in this research study. First, the EABC algorithm applies the combination of intra-route and inter-route exchange as the neighborhood search. Thus, this strategy can extend the regions of the search space to increase the chance for finding a better solution. Second, the high-quality solutions are used more often than the low-quality ones to produce an improved solution in the onlooker bee stage through sequential search technique. Therefore, the regions of the search space are searched in detail. Third, the stalled solutions are removed from the population and a new randomly generated solution is added to the population in the scout bee stage. This process provides global search ability and prevents the search from premature convergence problem.

3.4.2.1 The General Concept of ABC

The artificial bee colony is inspired by the intelligent finding food sources behavior of the honey bees around the hives proposed by Karaboga [27]. A colony of the bees consists of three types of bees: employed bees, onlookers and scouts. The employed bees search for available nectar sources and share this information with the onlookers via a waggle dance at the dancing area. The onlookers select the food sources by evaluating quality of nectar sources from the waggle dance to be further explored. When the quality of food sources is not improved within a time limit, the employed bees abandon the food source and turn into scout bees to find new food sources.

The ABC algorithm starts by generating a number of nectar sources (initial solutions) and assigning an employed bee to each food source. Each employed bee

explores a new food source near its original food source (neighborhood search) and measures the nectar amounts (fitness value). If the nectar quality of the new source is better than the old one, the old one will be replaced by the new one. After the employed bees update the food sources, they return to the hive with the information of the food sources. The information is shared with the onlookers by the waggle dance. Each onlooker selects a food source with a probability that depends on the nectar amounts (the roulette wheel method). In particular, a food source with higher nectar amounts has a higher probability to be selected by an onlooker than ones with lower nectar amounts. After selecting a food source, each onlooker finds a new food source around the selected food sources (neighborhood search) and evaluates the amount of nectar. The employed bee will abandon its old food source and go to the new one if it has more nectar. In the case that the quality of food source is not improved within a time limit, the employed bee will also abandon the old food source and become a scout bee that searches for the new food source by randomly generating a new solution. After the scout bee finds a new food source, it becomes an employed bee again. This process will repeat until a stopping criterion is reached.

3.4.2.2 Enhanced Artificial Bee Colony (EABC) Algorithm

Since the ABC algorithm was successfully to applied in VRP and VRPTW, these motivate us to apply this algorithm to solving the VRPBTW in this dissertation. Although the ABC algorithm was firstly applied to the VRPBTW by Tuntitippawan and Asawarungsaengkul [31-32], the computational results show that it underperforms the existing heuristics in many instances, especially in the large-scale problem. Since the ABC is often easily trapped in local optima, it is necessary to extend the exploration on the solution space and, equivalently, to expand the capability of neighborhood search. Therefore, in this dissertation, we introduce the enhanced artificial bee colony (EABC) algorithm by applying a forbidden list strategy to prevent the duplicated initial solution which extends the exploration on the solution space, and the sequential search strategy for onlookers to explore the neighborhood near the high-quality food source.

Forbidden List Strategy

In the process of generating the initial solution, a forbidden list strategy was applied in this section to prevent the duplication of the initial solution. After a feasible initial solution is obtained, the solution will be checked with the forbidden list of solutions. If the solution is not in the list, then add it to the list. Otherwise, the solution will be abandoned. The process repeats until the number of solutions in the forbidden list reaches the defined number. This strategy is applied to EABC algorithm whereas original version is executed without this strategy.

Sequential Search Strategy for Onlookers

In the onlooker bee process of the original version, if there are many onlooker bees selecting the same food source, each onlooker individually searches for a new food source and the old food source is replaced by the best of those new food sources. In EABC algorithm, if there are many onlooker bees selecting the same food source, each onlooker searches for a new food source in sequence as follows. If the previous onlooker bee finds a new food source, the next onlooker bee will start from the newly found food source and look for a better one. Otherwise, the next onlooker bee will start from the same food source as the previous one. In this way, the quality food source will be given opportunities to be further explored in good regions of the solution.

Intra-route and Inter-route Exchange Combination Strategy

The local search in the ABC proposed by Tuntitippawan and Asawarungsaengkul [32] only uses the λ-interchange, which is an inter-route operator that considers two routes at once. To extend the search ability, the EABC can either randomly apply λ-interchange or 1-move intra-route exchange, which work on a single route, for its neighborhood search. Since the 1-move operator improves the solution by deleting a customer and then inserting it into the same route, it helps rearranging the customer in the route. The experimental parameter testing discussed in Section 4.2.6 indicates that this setting gives better solution than using λ-interchange alone (See Figure 13).

3.4.2.3 Main Steps of EABC

The steps of the EABC algorithm for solving the VRPBTW model can be described as follows:

Step 1 Generate a set of initial solutions (food sources) by the NNRW method (Section 3.3.4). The forbidden list strategy is also applied in this process. Then assign each food source to each employed bee.

Step 2 Evaluate the fitness of each solution and record the global best solution.

Step 3 Apply the neighborhood search on each food source. An employed bee abandons its old food source if a new one with better fitness is found. Otherwise, increment the time counter of the food source.

Step 4 For each onlooker, select a food source by using the fitness-based roulette wheel selection method and improve the food source by the neighborhood search. If the onlooker bee finds a new one with better fitness, the employed bee associated with the food source abandons the old food source and go to the new one.

Step 5 Check the time counter of each food source. If it reaches the predetermined limit, the food source is replaced by a new randomly generated solution.

Step 6 Update the global best solution if a solution has better quality than the current best one and go to Step3. Otherwise, the algorithm ends and returns the global best solution in hand.

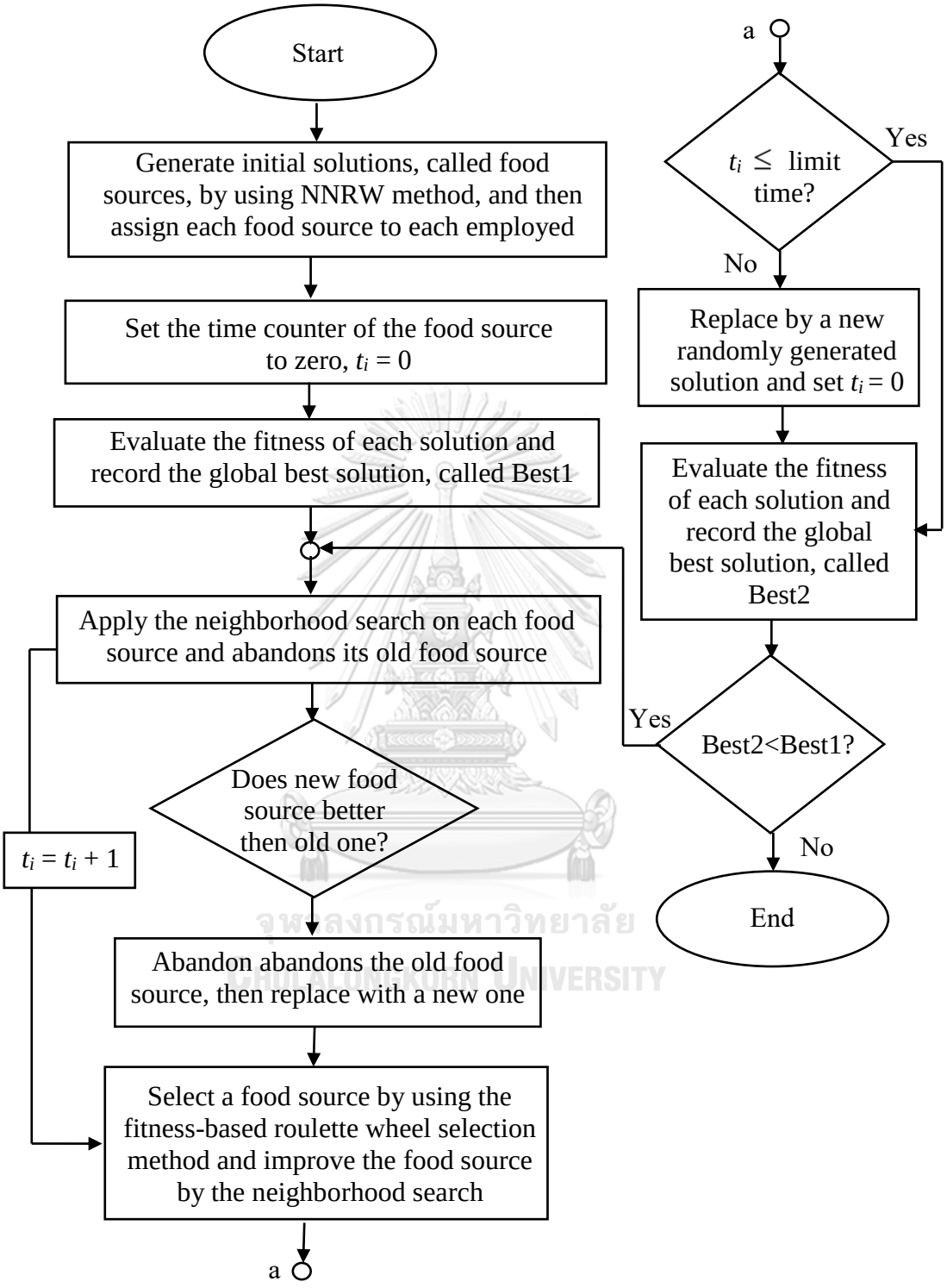


Figure 11 Flowchart of EABC algorithm

CHAPTER 4
COMPUTATIONAL EXPERIMENT

The proposed mathematical model for the VRPBTW and heuristic method was programed in CPLEX version 12.6 and in Microsoft Visual C# 2010 Express respectively. They were executed on a computer PC with 2.4 GHz Intel i7 Duo Core CPU and 8 GB memory.

4.1 Test Problems

The EGBA was tested on the widely accepted set of benchmark instances for VRPBTW proposed by Gelinas at el. [37] that were modified from R101-105 of Solomon's R1-type problems [16]. These problems had 100 customers that were located uniformly over the service area. They had a short scheduling horizon and the vehicle capacity was 200 units. Problems were generated by randomly selecting 10%, 30% and 50% of the 100 customers to be backhaul customers without any changes to the other attributes. Moreover, smaller problems were obtained by considering the first 25 and 50 customers.

4.2 Parameter Setting

A small study on parametrization of our algorithm is shown in this section. The crucial parameters are varied and their solutions are compared by carrying out on the large problem R101 with 10% backhauls selected randomly.

4.2.1 Nearest Neighbor (NN) Heuristic

The parameters were assigned as follows: the size of λ -interchange operator = 4, and maximum number of iterations = 300.

4.2.2 Improved Nearest Neighbor (INN) Heuristic

The parameters α, β, and γ , which are the weights associated with distance, idle time, and urgency of delivery, respectively, are set to α=0.4, β=0.3, γ=0.3 following Küçükoğlu and Öztürk [49]. The other parameters are assigned as follows: the size of λ -interchange operator = 4, and maximum number of iterations = 300.

4.2.3 Nearest Urgent Candidate (NUC) Heuristic

Recall that the parameters α, and β are the weights associated with distance, and idle time, respectively. In Figure 11, the relationship between the heuristic solution and ratio of α to β indicates that the heuristic solution is relatively better when the ratios of α to β are 0.5: 0.5, 0.6: 0.4, and 0.7: 0.3. Hence, we select the values $\alpha = 0.6$ and $\beta = 0.4$ for our parameters in all instances. The other parameters are assigned as follows: the size of λ -interchange operator = 4, and maximum number of iterations = 300.

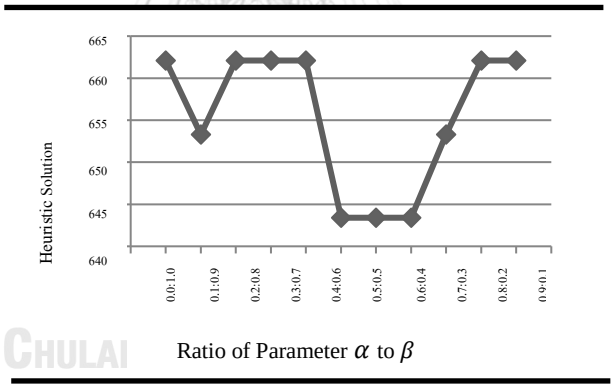


Figure 12 The relationship between the heuristic solution and ratio of α to β

4.2.4 Nearest Neighbor with Roulette Wheel Selection (NNRW) Heuristic

Recall that the parameters α, β, and γ are the weights associated with distance, idle time, and urgency of delivery, respectively, in the calculation of proximity that drives the probability in the roulette wheel used to generate solutions to start EGBA. From the suggestions of Küçükoğlu and Öztürk [49], the relationship of these parameters should be $\alpha+\beta+\gamma$ = 1 where α=0.4, β=0.3, γ=0.3. To evaluate this fact, the experiments based on the ratio of these parameters were performed and the results are shown in Figure 12. The results indicate that the performance of this algorithm is better

when parameter α is weighted more than the others. It produced the best solution when α=0.4, β=0.3, γ=0.3 as suggested in [49]. Thus, these parameters are set as α =0.4, β=0.3, γ=0.3 for the remaining of this dissertation. The other parameters are assigned as follows: the size of λ -interchange operator = 4, and maximum number of iterations = 300.

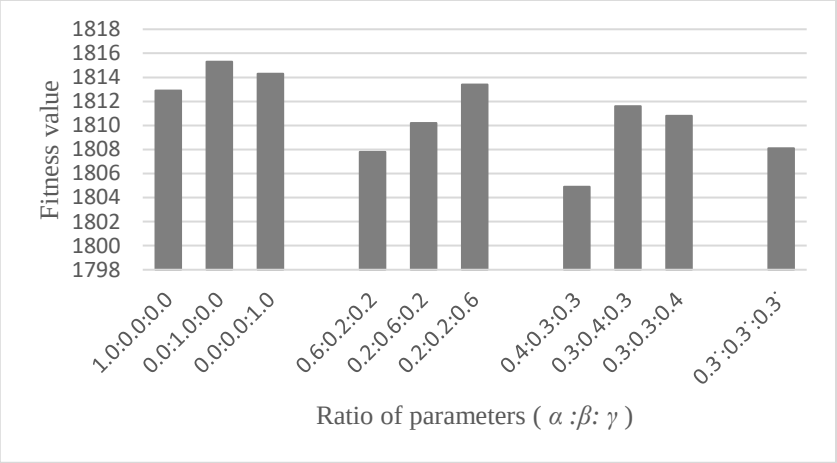


Figure 13 The relationship between fitness value and the ratio of parameters α, β, and γ

4.2.5 Cuckoo Search (CS) Algorithm

The CS algorithm parameters are assigned as follows: the number of host nests = 15, $\alpha = 0.4$, $\beta = 0.3$, $\gamma = 0.3$ (the weights associated with distance, idle time, and urgency of delivery, respectively), P_a = 0.25 (the suggestions of Yang and Deb [27]), the size of λ -interchange operator = 4, maximum number of iterations = 300.

4.2.6 Artificial Bee Colony (ABC) Algorithm

The relationship between the fitness value and parameter λ is shown in Figure 13. The smaller λ is, the more difficult it is for our algorithm to obtain a better solution since the number of customers to be exchanged between routes is limited. Thus, the value of parameter λ = 4 is set in this paper. Moreover, the comparison of λ -interchange with and without 1-move intra-route is also shown in this figure. The experiment indicated that the λ -interchange with 1-move intra-route can produce better solution when compared with the λ -interchange without 1-move intra-route. Thus, the 1-move intra-route can help improve the algorithm performance.

The number of employed bees and the number of onlooker bees are set to be the same, which is 50. This idea is recommended on the performance of artificial bee colony (ABC) algorithm which proposed by Karaboga and Basturk [79]. The *the limit time* parameter was set as 20.

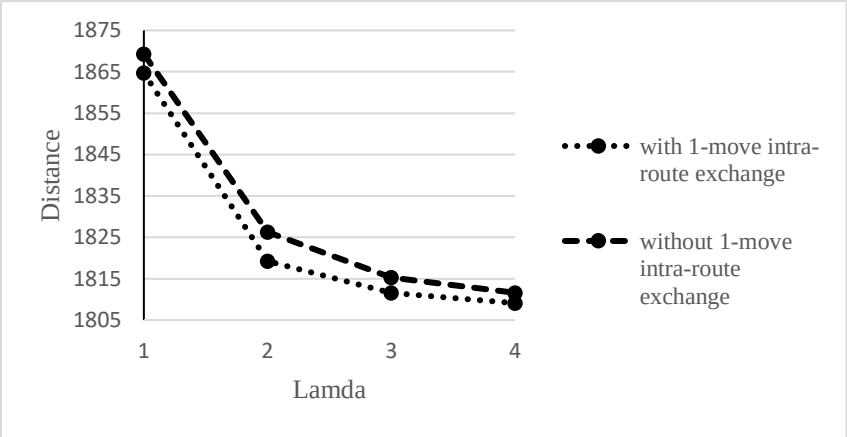


Figure 14 The relationship between the total distance and parameter λ, and comparison λ-interchange between with and without 1-move intra-route.

4.3 Results and Comparison

In Tables 1-6, the first column represents the number of customers. The second column shows the problem name. BH. (%) denotes the percentage of backhauls. Distance shows the total distance of solution. Best Distance means the total distance of the best solution from 20 replications performed using different starting solutions. NV indicates the number of vehicles used and time represents the computational time in seconds.

The computational results of the mathematical model, NUC, NNRW, CS and EABC for 25, 50, 100 customers for VRPBTW are reported in Tables 1-3 respectively. The empty slots mean the results cannot be found within 2 hours by solving the problems with CPLEX program. Some optimal solutions are found only in the small-sized problems, and its computational time is much higher than all proposed methods. This indicates that the exact method is too difficult to solve the VRPBTW within a reasonable time. Moreover, the NUC used the lowest computational time whereas the NNRW used the highest computational time.

Table 1 Computational results of the model, NUC, NNRW, CS, and EABC for 25 customers in VRPBTW.

Size	Problem	BH. (%)	Optimal Solution			NUC Solution			NNRW Solution			CS Solution			EABC Solution		
			Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time
n=25	R101	10	643.4	9	3.77	643.4	9	0.13	643.4	9	0.25	643.4	9	3.10	643.4	9	3.39
		30	-	-	-	721.8	10	0.14	721.8	10	0.26	721.8	10	1.95	721.8	10	2.00
		50	-	-	-	676.8	10	0.14	676.8	10	0.29	676.8	10	1.74	676.8	10	1.90
	R102	10	-	-	-	563.5	7	0.19	563.5	7	0.31	563.5	7	3.68	563.5	7	5.22
		30	-	-	-	628.1	9	0.16	628.1	9	0.45	628.1	9	3.29	628.1	9	4.11
		50	-	-	-	584.4	8	0.17	584.4	8	0.35	584.4	8	3.72	584.4	8	4.99
	R103	10	-	-	-	479.2	6	0.19	478.8	6	0.27	478.8	6	4.82	476.6	5	5.71
		30	507.0	7	60.11	507.0	7	0.19	507.0	7	0.25	507.0	7	2.63	507.0	7	3.21
		50	-	-	-	483.0	6	0.19	488.8	6	0.29	483.0	6	2.93	483.0	6	3.73
	R104	10	-	-	-	452.5	5	0.21	453.4	5	0.27	452.8	5	2.37	452.5	5	3.34
		30	-	-	-	468.5	6	0.20	476.3	6	0.22	473.1	6	3.91	468.5	6	4.40
		50	446.8	5	26.49	447.7	5	0.19	447.7	5	0.19	446.8	5	2.79	446.8	5	3.59
	R105	10	565.1	7	16.75	565.1	7	0.17	565.1	7	0.24	565.1	7	3.07	565.1	7	3.70
		30	623.5	8	72.06	632.9	8	0.14	628.9	8	0.26	623.5	8	2.89	623.5	8	3.09
		50	591.1	8	8.46	591.1	8	0.16	591.1	8	0.20	591.1	8	1.61	591.1	8	2.02

Table 2 Computational results of the model, NUC, NNRW, CS, and EABC for 50 customers in VRPBTW.

Size	Problem	BH. (%)	Optimal Solution			NUC Solution			NNRW Solution			CS Solution			EABC Solution		
			Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time
n=50	R101	10	-	-	-	1149.2	14	0.17	1133.3	15	0.43	1133.3	15	22.40	1133.3	15	39.03
		30	-	-	-	1204.7	16	0.20	1196.4	16	0.39	1191.6	16	21.19	1191.6	16	36.50
		50	-	-	-	1183.9	16	0.22	1183.9	16	0.38	1183.9	16	23.79	1183.9	16	36.80
	R102	10	-	-	-	976.5	12	0.30	977.0	12	0.52	976.5	12	35.83	976.5	12	46.47
		30	-	-	-	1058.1	14	0.23	1054.7	14	0.34	1054.6	14	33.07	1054.6	14	44.88
		50	-	-	-	1061.6	14	0.17	1059.7	14	0.32	1059.7	14	26.92	1059.7	14	43.89
	R103	10	-	-	-	829.8	9	0.20	828.7	10	0.49	818.8	9	43.79	812.3	9	54.49
		30	-	-	-	888.0	11	0.22	894.4	11	0.38	894.4	11	32.77	886.2	11	49.79
		50	-	-	-	885.4	10	0.28	892.2	10	0.44	889.0	10	36.85	883.0	11	49.04
	R104	10	-	-	-	703.2	7	0.27	700.9	7	0.38	698.2	7	46.82	685.9	7	55.17
		30	-	-	-	737.1	8	0.28	742.8	8	0.48	742.3	8	68.46	734.8	8	85.68
		50	-	-	-	739.4	8	0.33	754.0	8	0.50	734.5	8	69.60	733.6	8	81.31
	R105	10	-	-	-	992.1	11	0.16	983.3	12	0.38	972.8	11	68.51	976.2	11	76.07
		30	-	-	-	1052.8	12	0.18	1031.8	13	0.34	1027.1	13	71.08	1019.9	12	69.15
		50	-	-	-	1009.1	11	0.25	998.9	12	0.42	993.4	11	44.91	993.4	11	56.28

Table 3 Computational results of the model, NUC, NNRW, CS, and EABC for 100 customers in VRPBTW.

Size	Problem	BH. (%)	Optimal Solution			NUC Solution			NNRW Solution			CS Solution			EABC Solution		
			Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time	Best Distance	NV	Time
n=100	R101	10	-	-	-	1814.7	24	0.49	1811.6	24	1.21	1805.7	24	107.14	1809.1	24	220.99
		30	-	-	-	1885.2	24	0.84	1888.8	24	1.04	1886.9	24	111.69	1885.0	24	272.27
		50	-	-	-	1927.3	25	1.98	1929.4	26	3.54	1924.3	25	117.73	1921.2	25	270.70
	R102	10	-	-	-	1633.8	19	0.75	1628.8	21	1.14	1624.1	20	122.09	1620.8	20	309.01
		30	-	-	-	1716.8	21	0.98	1716.0	23	1.45	1705.6	22	131.93	1693.4	21	341.00
		50	-	-	-	1764.9	23	0.47	1765.2	22	0.97	1757.8	22	127.87	1738.7	21	270.14
	R103	10	-	-	-	1377.5	17	0.52	1391.6	18	1.02	1379.7	17	153.43	1355.2	17	345.50
		30	-	-	-	1426.7	17	1.01	1435.6	17	2.07	1407.7	16	157.91	1408.6	17	293.69
		50	-	-	-	1475.4	18	1.08	1490.1	19	1.78	1474.7	19	161.27	1463.7	18	264.91
	R104	10	-	-	-	1142.0	13	1.28	1146.0	13	2.31	1145.2	13	147.25	1119.6	13	294.54
		30	-	-	-	1166.7	14	2.15	1166.4	14	4.21	1167.8	14	105.59	1148.6	14	296.11
		50	-	-	-	1187.7	13	0.83	1229.9	14	2.22	1197.3	14	181.47	1207.4	14	278.64
	R105	10	-	-	-	1555.8	19	0.44	1541.9	18	1.19	1523.7	18	199.23	1514.3	18	254.66
		30	-	-	-	1618.4	18	0.72	1626.5	19	1.68	1602.2	19	143.47	1594.5	17	203.89
		50	-	-	-	1643.8	19	0.94	1647.2	19	1.83	1629.6	19	137.77	1607.2	18	215.98

Table 4 Comparison results of the NN, INN, NUC, NNRW, CS, ABC, and EABC for 25 customers in VRPBTW.

Size	Problem	BH. (%)	NN	INN	NUC	NNRW	CS	EABC	ABC[a]	%GAP_IMP						
			Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	NUC vs NN	NUC vs INN	NUC vs NNRW	NNRW vs NN	NNRW vs INN	EABC vs ABC	EABC vs CS
n=25	R101	10	662.1	662.1	643.4	643.4	643.4	643.4	643.4	**-2.82%**	**-2.82%**	**0.00%**	**-2.82%**	**-2.82%**	**0.00%**	**0.00%**
		30	735.3	721.8	721.8	721.8	721.8	721.8	721.8	**-1.84%**	**0.00%**	**0.00%**	**-1.84%**	**0.00%**	**0.00%**	**0.00%**
		50	693.1	678.8	676.8	676.8	676.8	676.8	676.8	**-2.35%**	**-0.29%**	**0.00%**	**-2.35%**	**-0.29%**	**0.00%**	**0.00%**
	R102	10	564.2	563.5	563.5	563.5	563.5	563.5	563.5	**-0.12%**	**0.00%**	**0.00%**	**-0.12%**	**0.00%**	**0.00%**	**0.00%**
		30	629.6	628.1	628.1	628.1	628.1	628.1	628.1	**-0.24%**	**0.00%**	**0.00%**	**-0.24%**	**0.00%**	**0.00%**	**0.00%**
		50	591.6	586.4	584.4	584.4	584.4	584.4	584.4	**-1.22%**	**-0.34%**	**0.00%**	**-1.22%**	**-0.34%**	**0.00%**	**0.00%**
	R103	10	507.1	488.8	479.2	478.8	478.8	476.6[#]	476.6	**-5.50%**	**-1.96%**	0.08%	**-5.58%**	**-2.05%**	**0.00%**	**-0.46%**
		30	534.8	534.0	507.0	507.0	507.0	507.0	507.0	**-5.20%**	**-5.06%**	**0.00%**	**-5.20%**	**-5.06%**	**0.00%**	**0.00%**
		50	535.2	497.4	483.0*	488.8	483.0	483.0	483.0	**-9.75%**	**-2.90%**	**-1.19%**	**-8.67%**	**-1.73%**	**0.00%**	**0.00%**
	R104	10	486.2	465.5	452.5*	453.4	452.8	452.5[#]	453.8	**-6.93%**	**-2.79%**	**-0.20%**	**-6.75%**	**-2.60%**	**-0.29%**	**-0.07%**
		30	517.4	513.3	468.5*	476.3	473.1	468.5[#]	468.5	**-9.45%**	**8.73%**	**-1.64%**	**-7.94%**	**-7.21%**	**0.00%**	**-0.97%**
		50	506.5	500.5	447.7	447.7	446.8	446.8	446.8	**-11.61%**	**-10.55%**	**0.00%**	**-11.61%**	**-10.55%**	**0.00%**	**0.00%**
	R105	10	579.6	565.1	565.1	565.1	565.1	565.1	565.1	**-2.50%**	**0.00%**	**0.00%**	**-2.50%**	**0.00%**	**0.00%**	**0.00%**
		30	633.4	632.9	632.9	628.9	623.5	623.5	628.0	**-0.08%**	**0.00%**	0.64%	**-0.71%**	**-0.63%**	**-0.72%**	**0.00%**
		50	639.2	635.5	591.1	591.1	591.1	591.1	591.1	**-7.53%**	**-6.99%**	**0.00%**	**-7.53%**	**-6.99%**	**0.00%**	**0.00%**

[a] Obtained from Tuntitippawan and Asawarungsaengkul [32]

* NUC is significantly better than NNRW (at 95% confidence level)

[#] EABC is significantly better than CS (at 95% confidence level)

Table 5 Comparison results of the NN, INN, NUC, NNRW, CS, ABC, and EABC for 50 customers in VRPBTW.

Size	Problem	BH. (%)	NN	INN	NUC	NNRW	CS	EABC	ABC[a]	%GAP_IMP						
			Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	NUC vs NN	NUC vs INN	NUC vs NNRW	NNRW vs NN	NNRW vs INN	EABC vs ABC	EABC vs CS
n=50	R101	10	1175.5	1173.2	1149.2	1133.3	1133.3	1133.3	1134.0	**-2.24%**	**-2.05%**	1.40%	**-3.59%**	**-3.40%**	**-0.06%**	**0.00%**
		30	1223.2	1218.8	1204.7	1196.4	1191.6	1191.6	1191.6	**-1.51%**	**-1.16%**	0.69%	**-2.19%**	**-1.84%**	**0.00%**	**0.00%**
		50	1203.1	1190.5	1183.9	1183.9	1183.9	1183.9	1183.9	**-1.60%**	**-0.55%**	**0.00%**	**-1.60%**	**-0.55%**	**0.00%**	**0.00%**
	R102	10	994.3	987.8	976.5*	977.0	976.5	976.5	976.5	**-1.79%**	**-1.14%**	**-0.05%**	**-1.74%**	**-1.09%**	**0.00%**	**0.00%**
		30	1091.1	1081.2	1058.1	1054.7	1054.6	1054.6	1054.6	**-3.02%**	**-2.14%**	0.32%	**-3.34%**	**-2.45%**	**0.00%**	**0.00%**
		50	1100.9	1100.3	1061.6	1059.7	1059.7	1059.7	1059.7	**-2.57%**	**-3.52%**	0.18%	**-3.74%**	**-3.69%**	**0.00%**	**0.00%**
	R103	10	877.8	874.9	829.8	828.7	818.8	812.3#	821.6	**-5.47%**	**-5.15%**	0.13%	**-5.59%**	**-5.28%**	**-1.13%**	**-0.79%**
		30	955.3	951.7	888.0*	894.4	894.4	886.2#	887.1	**-7.04%**	**-6.69%**	**-0.72%**	**-6.37%**	**-6.02%**	**-0.10%**	**-0.92%**
		50	947.6	939.1	885.4*	892.2	889.0	883.0#	885.1	**-6.56%**	**-5.72%**	**-0.76%**	**-5.85%**	**-4.99%**	**-0.24%**	**-0.67%**
	R104	10	792.1	784.8	703.2	700.9	698.2	685.9#	-	**-11.22%**	**-10.40%**	0.33%	**-11.51%**	**-10.69%**	-	**-1.76%**
		30	795.8	785.6	737.1*	742.8	742.3	734.8#	-	**-7.38%**	**-6.17%**	**-0.77%**	**-6.66%**	**-5.45%**	-	**-1.01%**
		50	771.7	771.6	739.4*	754.0	734.5	733.6#	739.3	**-4.19%**	**-4.17%**	**-1.94%**	**-2.29%**	**-2.28%**	**-0.77%**	**-0.12%**
	R105	10	1091.5	1091.5	992.1	983.3	972.8	976.2	985.2	**-9.11%**	**-9.11%**	0.89%	**-9.91%**	**-9.91%**	**-0.91%**	0.35%
		30	1084.3	1075.6	1052.8	1031.8	1027.1	1019.9#	1024.7	**-2.91%**	**-2.12%**	2.04%	**-4.84%**	**-4.07%**	**-0.47%**	**-0.70%**
		50	1078.4	1059.2	1009.1	998.9	993.4	993.4	993.4	**-6.43%**	**-4.73%**	1.02%	**-7.37%**	**-5.69%**	**0.00%**	**0.00%**

[a] Obtained from Tuntitippawan and Asawarungsaengkul [32]

* NUC is significantly better than NNRW (at 95% confidence level)

EABC is significantly better than CS (at 95% confidence level)

Table 6 Comparison results of the NN, INN, NUC, NNRW, CS, ABC, and EABC for 100 customers in VRPBTW.

Size	Problem	BH. (%)	NN	INN	NUC	NNRW	CS	EABC	ABC[a]	%GAP_IMP						
			Distance	Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	Best Distance	NUC vs NN	NUC vs INN	NUC vs NNRW	NNRW vs NN	NNRW vs INN	EABC vs ABC	EABC vs CS
n=100	R101	10	2072.7	1914.5	1814.7	1811.6	1805.7	1809.1	1818.6	**-12.45%**	**-5.21%**	0.17%	**-12.60%**	**-5.37%**	**-0.52%**	0.19%
		30	2091.2	1978.7	1885.2*	1888.8	1886.9	1885.0	1904.5	**-9.85%**	**-4.73%**	**-0.19%**	**-9.68%**	**-4.54%**	**-1.02%**	**-0.10%**
		50	1992.0	1990.2	1927.3*	1929.4	1924.3	1921.2	1928.2	**-3.25%**	**-3.16%**	**-0.11%**	**-3.14%**	**-3.05%**	**-0.36%**	**-0.16%**
	R102	10	1687.8	1671.8	1633.8	1628.8	1624.1	1620.8	1640.7	**-3.20%**	**-2.27%**	**0.31%**	**-3.50%**	**-2.57%**	**-1.21%**	**-0.20%**
		30	1755.7	1733.7	1716.8	1716.0	1705.6	1693.4	1717.3	**-2.22%**	**-0.97%**	0.05%	**-2.26%**	**-1.02%**	**-1.39%**	**-0.72%**
		50	2001.8	1891.2	1764.9*	1765.2	1757.8	1738.7	1752.2	**-11.83%**	**-6.68%**	**-0.02%**	**-11.82%**	**-6.66%**	**-0.77%**	**-1.09%**
	R103	10	1457.4	1454.2	1377.5*	1391.6	1379.7	1355.2#	-	**-5.48%**	**-5.27%**	**-1.01%**	**-4.51%**	**-4.30%**	-	**-1.78%**
		30	1478.8	1459.0	1426.7*	1435.6	1407.7	1408.6	-	**-3.52%**	**-2.21%**	**-0.62%**	**-2.92%**	**-1.60%**	-	0.06%
		50	1563.5	1519.5	1475.4*	1490.1	1474.7	1463.7#	-	**-5.63%**	**-2.90%**	**-0.99%**	**-4.69%**	**-1.93%**	-	**-0.75%**
	R104	10	1206.3	1152.3	1142.0*	1146.0	1145.2	1119.6#	-	**-5.33%**	**-0.89%**	**-0.35%**	**-5.00%**	**-0.55%**	-	**-2.24%**
		30	1210.7	1201.7	1166.7	1166.4	1167.8	1148.6#	-	**-3.63%**	**-2.91%**	0.03%	**-3.66%**	**-2.94%**	-	**-1.64%**
		50	1274.8	1274.7	1187.7*	1229.9	1197.3	1207.4	-	**-6.83%**	**-6.83%**	**-3.43%**	**-3.52%**	**-3.51%**	-	0.84%
	R105	10	1632.1	1627.6	1555.8	1541.9	1523.7	1514.3#	-	**-4.67%**	**-4.41%**	0.90%	**-5.53%**	**-5.27%**	-	**-0.62%**
		30	1626.3	1621.7	1618.4*	1626.5	1602.2	1594.5#	-	**-0.49%**	**-0.20%**	**-0.50%**	0.01%	0.30%	-	**-0.48%**
		50	1724.2	1699.8	1643.8*	1647.2	1629.6	1607.2#	-	**-4.66%**	**-3.29%**	**-0.21%**	**-4.47%**	**-3.09%**	-	**-1.37%**

[a] Obtained from Tuntitippawan and Asawarungsaengkul [32]

* NUC is significantly better than NNRW (at 95% confidence level)

EABC is significantly better than CS (at 95% confidence level)

Table 7 Comparison results of the NUC, NNRW, CS, EABC, EGB, and other heuristics for 25 customers in VRPBTW

Size	Problem	BH. (%)	BKS	NUC	NNRW	CS	EABC	DEA[d]	HMA[c]	%GAP_BKS					
			Distance	Distance	Best Distance	Best Distance	Best Distance	Distance	Distance	NUC	NNRW	CS	EABC	DEA[a]	HMA[c]
n=25	R101	10	643.4[a]	643.4	643.4	643.4	643.4	643.4	643.4	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
		30	717.0[b]	721.8	721.8	721.8	721.8	721.8	721.8	0.67%	0.67%	0.67%	0.67%	0.67%	0.67%
		50	676.8[d]	676.8	676.8	676.8	676.8	676.8	676.8	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	R102	10	563.5[c]	563.5	563.5	563.5	563.5	565.3	563.5	**0.00%**	**0.00%**	**0.00%**	**0.00%**	0.32%	**0.00%**
		30	628.1[c]	628.1	628.1	628.1	628.1	629.0	628.1	**0.00%**	**0.00%**	**0.00%**	**0.00%**	0.14%	**0.00%**
		50	584.4[c]	584.4	584.4	584.4	584.4	585.3	584.4	**0.00%**	**0.00%**	**0.00%**	**0.00%**	0.15%	**0.00%**
	R103	10	476.6[e]	479.2	478.8	478.8	476.6	489.0	478.8	0.55%	0.46%	0.46%	**0.00%**	2.60%	0.46%
		30	507.0[c]	507.0	507.0	507.0	507.0	510.9	507.0	**0.00%**	**0.00%**	**0.00%**	**0.00%**	0.77%	**0.00%**
		50	483.0[c]	483.0	488.8	483.0	483.0	495.0	483.0	**0.00%**	1.20%	**0.00%**	**0.00%**	2.48%	**0.00%**
HMA[c]	R104	10	453.8[c]	452.5	453.4	452.8	452.5	459.1	453.8	**-0.29%**	**-0.09%**	**-0.22%**	**-0.29%**	1.17%	**0.00%**
		30	468.5[c]	468.5	476.3	473.1	468.5	469.6	468.5	**0.00%**	1.66%	0.98%	**0.00%**	0.23%	**0.00%**
		50	446.8[c]	447.7	447.7	446.8	446.8	458.7	446.8	0.20%	0.20%	**0.00%**	**0.00%**	2.66%	**0.00%**
	R105	10	565.1[a]	565.1	565.1	565.1	565.1	565.1	565.1	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
		30	623.5[c]	632.9	628.9	623.5	623.5	630.2	623.5	1.51%	0.87%	**0.00%**	**0.00%**	1.07%	**0.00%**
		50	591.1[c]	591.1	591.1	591.1	591.1	598.5	592.1	**0.00%**	**0.00%**	**0.00%**	**0.00%**	1.25 %	0.17%

[a] Obtained from Potvin et al. [21]
[b] Obtained from Thangiah et al. [41]
[c] Obtained from Küçükoğlu and Öztürk [49]
[d] Obtained from Küçükoğlu and Öztürk [48]
[e] Obtained from Tuntitippawan and Asawarungsaengkul [32]

Table 8 Comparison results of the NUC, NNRW, CS, EABC, EGB, and other heuristics for 50 customers in VRPBTW.

Size	Problem	BH. (%)	BKS	NUC	NNRW	CS	EABC	DEA[d]	HMA[c]	%GAP_BKS					
			Distance	Distance	Best Distance	Best Distance	Best Distance	Distance	Distance	NUC	NNRW	CS	EABC	DEA[d]	HMA[c]
n=50	R101	10	1134.0[e]	1149.2	1133.3	1133.3	1133.3	1138.3	1135.8	1.34%	**-0.06%**	**-0.06%**	**-0.06%**	0.38%	0.16%
		30	1191.6[c]	1204.7	1196.4	1191.6	1191.6	1245.8	1191.6	1.10%	0.40%	**0.00%**	**0.00%**	0.67%	**0.00%**
		50	1183.9[a]	1183.9	1183.9	1183.9	1183.9	1183.9	1183.9	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	R102	10	976.5[e]	976.5	977.0	976.5	976.5	978.7	976.8	**0.00%**	0.05%	**0.00%**	**0.00%**	0.23%	0.03%
		30	1024.8[b]	1058.1	1054.7	1054.6	1054.6	1046	1046	3.25%	2.92%	2.91%	2.91%	0.14%	2.07%
		50	1059.7[a]	1061.6	1059.7	1059.7	1059.7	1153	1061.6	0.18%	**0.00%**	**0.00%**	**0.00%**	0.15%	0.18%
	R103	10	815.5[c]	829.8	828.7	818.8	812.3	831.1	815.5	1.75%	1.62%	0.40%	**-0.39%**	2.60%	**0.00%**
		30	887.1[e]	888	894.4	894.4	886.2	895.1	889.3	0.10%	0.82%	0.82%	**-0.10%**	0.77%	0.25%
		50	885.1[e]	885.4	892.2	889	883	887.7	887.7	0.03%	0.80%	0.44%	**-0.24%**	2.48%	0.29%
HMA[c]	R104	10	687.7[c]	703.2	700.9	698.2	685.9	688.7	687.7	2.25%	1.92%	1.53%	**-0.26%**	1.17%	**0.00%**
		30	736.8[c]	737.1	742.8	742.3	734.8	737.7	736.8	0.04%	0.81%	0.75%	**-0.27%**	0.23%	**0.00%**
		50	738.2[c]	739.4	754.0	734.5	733.6	742.2	738.2	0.16%	2.14%	**-0.50%**	**-0.62%**	2.66%	**0.00%**
	R105	10	972.8[d]	992.1	983.3	972.8	976.2	972.8	978.5	1.98%	1.08%	**0.00%**	0.35%	**0.00%**	0.59%
		30	1024.7[e]	1052.8	1031.8	1027.1	1019.9	1030	1026.7	2.74%	0.69%	0.23%	**-0.47%**	1.07%	0.20%
		50	993.4[e]	1009.1	998.9	993.4	993.4	1022.2	996.2	1.58%	0.55%	**0.00%**	**0.00%**	1.08%	0.28%

[a] Obtained from Potvin et al. [21]
[b] Obtained from Thangiah et al. [41]
[c] Obtained from Küçükoğlu and Öztürk [49]
[d] Obtained from Küçükoğlu and Öztürk [48]
[e] Obtained from Tuntitippawan and Asawarungsaengkul [32]

Table 9 Comparison results of the NUC, NNRW, CS, EABC, EGB, and other heuristics for 100 customers in VRPBTW.

Size	Problem	BH. (%)	BKS	NUC	NNRW	CS	EABC	DEA[d]	HMA[c]	%GAP_BKS					
			Distance	Distance	Best Distance	Best Distance	Best Distance	Distance	Distance	NUC	NNRW	CS	EABC	DEA[a]	HMA[c]
n=100	R101	10	1811.6[d]	1814.7	1811.6	1805.7	1809.1	1811.6	1811.6	0.17%	**0.00%**	**-0.33%**	**-0.14%**	**0.00%**	**0.00%**
		30	1891.1[c]	1885.2	1888.8	1886.9	1885.0	1925.9	1891.1	**-0.31%**	**-0.12%**	**-0.22%**	**-0.32%**	1.84%	**0.00%**
		50	1905.9[a]	1927.3	1929.4	1924.3	1921.2	1930.2	1911.2	1.12%	1.23%	0.97%	0.80%	1.27%	0.28%
	R102	10	1623.7[c]	1633.8	1628.8	1624.1	1620.8	1649.8	1623.7	0.62%	0.31%	0.02%	**-0.18%**	1.61%	**0.00%**
		30	1724.0[c]	1716.8	1716.0	1705.6	1693.4	1758.2	1724.0	**-0.42%**	**-0.46%**	**-1.07%**	**-1.77%**	1.98%	**0.00%**
		50	1746.0[b]	1764.9	1765.2	1757.8	1738.7	1777.1	1759.8	1.08%	1.10%	0.68%	**-0.42%**	1.78%	0.79%
	R103	10	1346.9[c]	1377.5	1391.6	1379.7	1355.2	1356.3	1346.9	2.27%	3.32%	2.44%	0.62%	0.70%	**0.00%**
		30	1385.9[c]	1426.7	1435.6	1407.7	1408.6	1389.2	1385.9	2.94%	3.59%	1.57%	1.64%	0.24%	**0.00%**
		50	1456.5[f]	1475.4	1490.1	1474.7	1463.7	1465.0	1465.0	1.30%	2.31%	1.25%	0.49%	0.58%	0.58%
HMA[c]	R104	10	1084.2[f]	1142.0	1146.0	1145.2	1119.6	1105.4	1093.4	5.33%	5.70%	5.63%	3.27%	1.96%	0.85%
		30	1136.6[c]	1166.7	1166.4	1167.8	1148.6	1146.5	1136.6	2.65%	2.62%	2.75%	1.06%	0.87%	**0.00%**
		50	1189.6[c]	1187.7	1229.9	1197.3	1207.4	1199.6	1189.6	**-0.16%**	3.39%	0.65%	1.50%	0.84%	**0.00%**
	R105	10	1516.0[c]	1555.8	1541.9	1523.7	1514.3	1527.7	1516.0	2.63%	1.71%	0.51%	**-0.11%**	0.77%	**0.00%**
		30	1581.5[c]	1618.4	1626.5	1602.2	1594.5	1582.6	1581.5	2.33%	2.85%	1.31%	0.82%	0.07%	**0.00%**
		50	1604.1[c]	1643.8	1647.2	1629.6	1607.2	1608.6	1604.1	2.47%	2.69%	1.59%	0.19%	0.28%	**0.00%**

[a] Obtained from Potvin et al. [21]
[b] Obtained from Thangiah et al. [41]
[c] Obtained from Küçükoğlu and Öztürk [49]
[d] Obtained from Küçükoğlu and Öztürk [48]
[e] Obtained from Tuntitippawan and Asawarungsaengkul [32]
[f] Obtained from Pisinger and Ropke [45]

Tables 4-6 report comparison results of proposed algorithms (NUC, NNRW, CS, EABC) and the other methods (NN, INN, ABC [32]) for VRPBTW with 25, 50, and 100 customers, respectively. The %Gap_IMP column denotes the gap percentage between the considered solution and the compared solution. A negative number in this column means the considered solution is better than the compared solution, zero value indicates that both are equal, and a positive value indicates the considered solution is worse than the compared solution. Specifically, the %Gap_IMP is computed by the formula:

$$\%\text{Gap\_IMP} = \frac{(the\ considered\ solution)-(the\ compared\ solution)}{the\ compared\ solution} \times 100.$$

The numbers in bold face font in these tables indicate that the considered solution is equivalent or better comparing with compared solution for the same problem. Note that the empty slots in ABC [32] column mean the results were not reported in their paper. The results obtained from the comparison in Tables 4-6 can be summarized in Table 10.

Table 10 The summary of the result comparisons for each algorithm.

versus	NN	INN	NNRW	NUC	ABC	CS	EABC
NN	-	1/ 45 (2.22%)	1/ 45 (2.22%)	0/ 45 (0.00%)	-	-	-
INN	**44/ 45 (97.78%)**	-	1/ 45 (2.22%)	0/ 45 (0.00%)	-	-	-
NNRW	**44 / 45 (97.78%)**	**44 / 45 (97.78%)**	-	10/ 45 (22.22%)	-	-	-
NUC	**45 / 45 (100%)**	**45 / 45 (100%)**	**30/ 45 (66.67%)**	-	-	-	-
ABC	-	-	-	-	-	5/ 34 (14.71%)	0/ 45 (0.00%)
CS	-	-	-	-	**29 / 34 (85.29%)**	-	4/ 45 (8.89%)
EABC	-	-	-	-	**34 / 34 (100%)**	**41 / 45 (91.11%)**	-

Each fraction x/y in Table 10 means that the row algorithm obtained x equivalent or better solutions out of the total y instances when compared with the column algorithm. Moreover, these fractions are also shown as percentages in parentheses. For example, in the entry (NUC, NNRW) shows the fraction 30/45. This means, the NUC obtained 30 equivalent or better solutions out of 45 problems (66.67%)

when compared with the NNRW solution. The bold numbers in this table highlight the outcomes with at least 50%.

To determine if the considered solution and compared solutions are significantly different from each other, the Mann–Whitney U test is applied. This test is a nonparametric test which does not require a special distribution of the dependent variable in the analysis. The Mann–Whitney U value is the smaller value of U_1 and U_2 which are computed from the formula:

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2}, U_2 = R_2 - \frac{n_2(n_2+1)}{2}$$

where R_1 and R_2 are the sum of the ranks in samples 1 and 2, respectively; and n_1 and n_2 are the number of samples 1 and 2, respectively. In this dissertation, we used a two-tailed test with confidence interval at the 95% confidence so U critical value when $n_1 = 20$, and $n_2 = 20$ is $U_0 = 105$. The number which is marked with the star symbol (*) in Tables 4-6 indicates that the NUC solution is significantly better than the NNRW solution at 95% confidence level, and the number which is marked with the octothorpe symbol (#) indicates that the EABC solution is significantly better than the CS solution at 95% confidence level. In Tables 4-6 report that there are 18 problems that the NUC is significantly better than the NNRW, whereas there are 17 problems that that the EABC is significantly better than the CS.

To evaluate the efficiency of the proposed algorithms, the comparison between the solutions obtained from the proposed algorithms in this dissertation and the best known solutions in the literature is also shown in %Gap_BKS column of Tables 7-9. The %Gap_BKS in these tables is the relative difference in percentage between the considered solution and the best known solution. %Gap_BKS is computed by:

$$\%\text{Gap\_BKS} = \frac{(the\ considered\ solution)-(the\ best\ known\ solution)}{the\ bestknown\ solution} \times 100.$$

The %Gap_BKS can be negative, zero, or positive. Since this is a minimization problem, if %Gap_BKS is negative, the considered solution is better than the best known solution. If it is zero, the two are equal. If it is positive, the best known solution is better. For example, in the %Gap_BKS column, the negative numbers in the EGBA

sub-columns means the EABC obtained better solutions than the current best known solutions. The results obtained from the comparison can be summarized in Table 11.

Each fraction x/y in Table 11 means that the algorithm obtained x equivalent or better solutions out of the total y instances when compared with the best known solutions. Moreover, these fractions are also shown as percentages in parentheses. For example, the EABC obtained 33 equivalent or better solutions out of 45 problems (73.33%) when compared with the best known solutions. The bold numbers in this table highlight outcomes from the proposed algorithms. These results indicate that the EABC algorithm outperformed the other proposed algorithms in terms of solution quality in many problems.

Table 11 The summary of comparison between each algorithm solutions and best known solutions

Algorithm solution	Best known solution
EABC	**33/45 (73.33%)**
HMA	29/45 (64.44%)
CS	**23/45 (51.11%)**
NUC	**16/45 (35.56%)**
NNRW	**12/45 (26.67%)**
DEA	6/45 (13.33%)

In order to evaluate the potentiality of EABC algorithm, from Tables 8-10, the EABC algorithm is also compared with the DEA, which is a population-based heuristic, and the HMA, which is a non-population-based heuristic. The results obtained from the comparison can be summarized as follows.

- When compared with the DEA, the EABC algorithm obtained 38 equivalent or better solutions out of 45 problems (84.44%).
- When compared with the HMA, the EABC algorithm obtained 37 equivalent or better solutions out of 45 problems (82.22%).

In summary, the EABC outperformed the existing algorithms in terms of solution quality in many problems as it obtained 33 equivalent or new best known solutions out of 45 instances (73.33%) while others did not perform as well (NN 0.00%, INN 6.67%, NNRW 26.67%, NUC 35.56%, CS 51.11%, HMA 53.33%, and DEA

13.33%). In addition, EGB algorithm obtained 15 new best known solutions, and found the optimal solutions for some instances. Moreover, EABC still displayed superior performance on the problems where the optimal solution is still unknown.

4.4 Rate of Convergence

In order to consider the convergence of the proposed algorithms, the graphs between the fitness value and the number of iterations for each instance are plotted in section A of the appendix. The results obtained from those graphs can be summarized in Table 12.

Table 12 The average number of iterations until the start of the convergence.

Proposed algorithms	The average number of iterations until the start of convergence			
	Small problem (25 customers)	Medium Problem (50 customers)	Large problem (100 customers)	Overall (45problems)
NNRW	16.20	25.27	29.93	23.80
NUC	12.93	24.27	34.13	23.78
CS	5.93	8.07	8.27	7.42
EABC	2.47	5.00	5.93	4.47

In Table 12, on average, the proposed algorithm which used the least number of iterations was EABC for all sizes of problems. The NNRW used the most number of iterations on average in all sizes of problems except large problems where NUC used the most number of iterations. For overall problems, the NUC and the NNRW have approximately the same convergent rate while EABC has faster convergent rate than CS by 60.18 % on average.

4.5 Results Discussion

The computational results show that the optimal solution from mathematical model cannot be found in many instances, especially the medium- and large-sized problems, and its computation time is much higher than other heuristics. This is because the VRPBTW is an NP-hard combinatorial optimization problem, thus, the exact method is not always possible to find an optimal solution within a limited time.

The proposed heuristics (NUC, NNRW) perform better than NN and INN. We speculate that a candidate technique in NUC helps to obtain a good initial solution by properly selecting the customers in construction phase, while roulette wheel selection in NNRW helps to extend the exploration on the solution space by increasing the number of initial solutions. The computational time of NNRW algorithm is also the highest among the other heuristics with this reason. However, the NNRW underperforms the NUC in many problems. This may be concluded that good initial construction is more important than the extension of the exploration on the solution space by increasing the number of initial solutions.

The CS results indicate that it is better than other presented heuristics in this dissertation except for EABC. It can produce better solutions than the best-known solutions for the majority of small- and medium-sized instances. However, it does not perform as well for large problems. We speculate that EABC algorithm contains many techniques to extend the exploration on the solution space and to escape from local optima while the CS does not.

When comparing the results of enhanced version of ABC with the original one proposed by Tuntitippawan and Asawarungsaengkul [32], the EABC algorithm is superior to original ABC algorithm in terms of the solution quality. We speculate that the forbidden list strategy in generating process, the sequential search strategy for onlooker bees, and the intra-route and inter-route exchange combination strategy for the local search in the EABC algorithm indeed helps extend the exploration on the solution space to obtain the better solutions. Note that although the sequential search of onlookers increases the chance of finding great solutions, it also leads to larger computational time. Further study is needed to analyze the tradeoffs and compare the computational time with the original ABC algorithm.

When comparing the results of EABC algorithm with the other methods in terms of solution quality, we find that the performance of our algorithm is better than the HMA and DEA for small- and medium-sized problems while comparable with the HMA and the DEA in the large-sized problems. We speculate that there are four main reasons EABC algorithm contributes the successful results. First, the EABC algorithm is a population-based heuristic which starts with a number of unduplicated initial solutions. Therefore, it can explore more in the solution space and get more chance to

obtain the better solutions. Second, the EABC algorithm applied the combination of intra-route and inter-route exchange as the neighborhood search. Hence, this strategy can extend the regions of the search space to increase the chance for finding a better solution. Third, the high-quality solutions are used more often than the low-quality ones to produce an improved solution in the onlooker bee stage. Thus, the regions of the search space are searched in shorter time and in detail. Forth, the stalled solutions are removed from the population and a new solution from random generating is added to the population in the scout bee stage. This process provides global search ability and prevents the search from premature convergence problem.

CHAPTER 5
CONCLUSION

The vehicle routing problem with backhauls and time windows (VRPBTW) is an extension of the vehicle routing problem with backhaul (VRPB) by imposing a specific service time window for each customer. The objective of this problem is to find a set of feasible vehicle routes that minimizes the total distance while imposing capacity, backhaul, and time window constraints. In this dissertation, a mathematical model of VRPBTW is introduced to obtain an optimal solution. It is formulated as a mixed-integer programming model by modifying the mathematical formulation for fleet size and mixed vehicle routing problem with backhauls (FSMVRPB) proposed by Salhi et al. [76] and adding time window constraints from [49]. The aim of this model is to minimize total distance for VRPBTW. (There is no additional cost for adding vehicles.) The VRPBTW model is solved using CPLEX. However, the optimal solutions of many problems (especially the medium- and large-sized problems) cannot be found within two hours because the complexity of VRPBTW depends on the number of customers in the problem. Hence, the alternative methods, which are heuristic and metaheuristic methods, are presented to solve this issue.

For NUC algorithm, all customers are initially ordered according to the urgency of delivery. The closeness is computed from only two factors, namely, the direct distance and the waiting time. With a candidate list, we can preserve the urgency order while constructing the initial solutions. Then, the local search heuristics, i.e. 1-move and the λ-interchange, are applied to improve the solution.

NNRW heuristic is a combination of a roulette wheel selection method and the INN heuristic for generating the initial solutions. The closeness is computed from three factors in the same way as described in the INN heuristic.

Moreover, two metaheuristic methods are studied to obtain the optimal or near optimal solutions.

The first metaheuristic is the cuckoo search (CS) algorithm. It starts by generating a number of initial solutions called host eggs by applying NNRW algorithm. Then the algorithm assigns the host eggs to nests. In this dissertation, a nest always contains only a single host egg. The neighborhood search is randomly applied to a host egg to create a cuckoo egg. A solution (host egg) is replaced by a new solution (cuckoo egg) if the new one is better. A cuckoo egg will be abandoned, and the host bird will build a completely new one (generating a new solution) when it discovers the egg is not its own.

Second one is an enhanced artificial bee colony (EABC) algorithm. The ABC algorithm starts by generating a number of initial solutions called nectar sources by NNRW. Then the algorithm assigns an employed bee to each food source. The neighborhood search is applied to each solution before it is selected by a roulette wheel selection method. When the quality of a solution is not improved within a time limit, the employed bee abandons the food source (an old solution) and turn into a scout bee to find a new food source (generating a new solution). Three strategies are proposed in EABC, which are a forbidden list, the sequential search for onlookers, and the combination of the 1-move intra-route exchange and the λ-interchange technique.

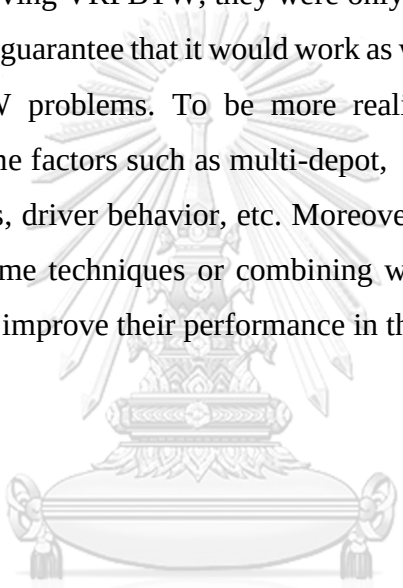
The proposed algorithms were tested on the classical set of benchmark instances (25, 50, 100 customers) proposed by Gélinas at el. [37] to evaluate the efficiency of each algorithm.

For heuristics, NNRW and NUC algorithms are compared with the general nearest neighbor algorithm (NN) and the improved nearest neighbor algorithm (INN) through the benchmark instances. The results show that both proposed heuristics are superior to NN and INN heuristic in terms of solution quality. In addition, in terms of quality, the NUC outperforms the NNRW in many problems, and its computational time is also lower than the NNRW algorithm. Although the convergent rate of NNRW is the slowest in the small and medium problem sizes, the NNRW has the same convergent rate as NUC in overall problems.

For metaheuristics, the enhanced version of ABC is superior to original version in terms of solution quality in all problems. Moreover, the results indicate that EABC algorithm outperforms the cuckoo search in terms of solution quality in many problems. In addition, the comparison between the solutions of the proposed algorithms (EABC

and CS) and the best known solutions in the literature is made. Results show that proposed algorithms yield the best results for most instances, especially EABC, which obtained 33 equivalent or new best known solutions out of 45 problems (73.33%). There are 15 new best known solutions found and the optimal solutions are obtained for some instances. Furthermore, the convergent rate of EABC is the fastest among the proposed algorithms. Hence, the proposed algorithms are effective ways to solve the VRPBTW.

Although the results in dissertation shows that the proposed algorithms are effective choices for solving VRPBTW, they were only tested on the set of benchmark instances so it could not guarantee that it would work as well on the real-world problems or other non-VRPBTW problems. To be more realistic, the problems should be extended by adding some factors such as multi-depot, mixed size of the vehicle fleet, traffic congestion levels, driver behavior, etc. Moreover, the algorithms also could be enhanced by adding some techniques or combining with other algorithms to reduce their disadvantages and improve their performance in the future work.



REFERENCES

[1] Jin, J., T.G. Crainic, and A. Løkketangen, *A parallel multi-neighborhood cooperative tabu search for capacitated vehicle routing problems.* European Journal of Operational Research, 2012. **222**(3): p. 441-451.

[2] Santos, L., J. Coutinho-Rodrigues, and J.R. Current, *An improved heuristic for the capacitated arc routing problem.* Computers & Operations Research, 2009. **36**(9): p. 2632-2637.

[3] Atefi, R., et al., *The open vehicle routing problem with decoupling points.* European Journal of Operational Research, 2018. **265**(1): p. 316-327.

[4] Yu, V.F., P. Jewpanya, and A.A.N.P. Redi, *Open vehicle routing problem with cross-docking.* Computers & Industrial Engineering, 2016. **94**: p. 6-17.

[5] Gajpal, Y. and P. Abad, *An ant colony system (ACS) for vehicle routing problem with simultaneous delivery and pickup.* Computers & Operations Research, 2009. **36**(12): p. 3215-3223.

[6] Zachariadis, E.E., C.D. Tarantilis, and C.T. Kiranoudis, *A hybrid metaheuristic algorithm for the vehicle routing problem with simultaneous delivery and pick-up service.* Expert Systems with Applications, 2009. **36**(2): p. 1070-1081.

[7] Toth, P. and D. Vigo, *A heuristic algorithm for the symmetric and asymmetric vehicle routing problems with backhauls.* European Journal of Operational Research, 1999. **113**(3): p. 528-543.

[8] Osman, I.H. and N.A. Wassan, *A reactive tabu search meta-heuristic for the vehicle routing problem with back-hauls.* Journal of Scheduling, 2002. **5**(4): p. 263-285.

[9] Brandão, J., *A new tabu search algorithm for the vehicle routing problem with backhauls.* European Journal of Operational Research, 2006. **173**(2): p. 540-555.

[10] Tavakkoli-Moghaddam, R., A.R. Saremi, and M.S. Ziaee, *A memetic algorithm for a vehicle routing problem with backhauls.* Applied Mathematics and Computation, 2006. **181**(2): p. 1049-1060.

[11] Gajpal, Y. and P.L. Abad, *Multi-ant colony system (MACS) for a vehicle routing problem with backhauls.* European Journal of Operational Research, 2009. **196**(1): p. 102-117.

[12] Chiang, W.-C. and R.A. Russell, *A reactive tabu search metaheuristic for the vehicle routing problem with time windows.* INFORMS Journal on computing, 1997. **9**(4): p. 417-430.

[13] Berger, J. and M. Barkaoui. *A memetic algorithm for the vehicle routing problem with time windows*. in *The 7th International Command and Control Research and Technology Symposium*. 2002.

[14] Berger, J. and M. Barkaoui, *A parallel hybrid genetic algorithm for the vehicle routing problem with time windows.* Computers & Operations Research, 2004. **31**(12): p. 2037-2053.

[15] Bräysy, O. and M. Gendreau, *Tabu search heuristics for the vehicle routing problem with time windows.* Top, 2002. **10**(2): p. 211-237.

[16] Solomon, M.M., *Algorithms for the vehicle routing and scheduling problems with time window constraints.* Operations research, 1987. **35**(2): p. 254-265.

[17] Gong, W., X. Liu, J. Zhang, and Z. Fu, *Two-Generation Ant Colony System for Vehicle Routing Problem with Time Windows*. in *2007 International Conference on Wireless Communications, Networking and Mobile Computing*. 2007.

[18] Yu, B., Z.Z. Yang, and B.Z. Yao, *A hybrid algorithm for vehicle routing problem with time windows.* Expert Systems with Applications, 2011. **38**(1): p. 435-441.

[19] Ding, Q., X. Hu, L. Sun, and Y. Wang, *An improved ant colony optimization and its application to vehicle routing problem with time windows.* Neurocomputing, 2012. **98**: p. 101-107.

[20] Tasan, A.S. and M. Gen, *A genetic algorithm based approach to vehicle routing problem with simultaneous pick-up and deliveries.* Computers & Industrial Engineering, 2012. **62**(3): p. 755-761.

[21] Potvin, J.-Y., C. Duhamel, and F. Guertin, *A genetic algorithm for vehicle routing with backhauling.* Applied Intelligence. **6**(4): p. 345-355.

[22] Ai, T.J. and V. Kachitvichyanukul, *Particle swarm optimization and two solution representations for solving the capacitated vehicle routing problem.* Computers & Industrial Engineering, 2009. **56**(1): p. 380-387.

[23] Goksal, F.P., I. Karaoglan, and F. Altiparmak, *A hybrid discrete particle swarm optimization for vehicle routing problem with simultaneous pickup and delivery.* Computers & Industrial Engineering, 2013. **65**(1): p. 39-53.

[24] Yu, B., Z.-Z. Yang, and B. Yao, *An improved ant colony optimization for vehicle routing problem.* European Journal of Operational Research, 2009. **196**(1): p. 171-176.

[25] Yang, X.-S. and S. Deb. *Cuckoo search via Lévy flights*. in *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*. 2009. IEEE.

[26] Zheng, H., Y. Zhou, and Q. Luo, *A hybrid Cuckoo Search Algorithm-GRASP for Vehicle Routing Problem.* Journal of Convergence Information Technology, 2013. **8**(3): p. 821-828.

[27] Karaboga, D., *An idea based on honey bee swarm for numerical optimization*. 2005, Technical report-tr06, Erciyes university, engineering faculty, computer engineering department.

[28] Szeto, W.Y., Y. Wu, and S.C. Ho, *An artificial bee colony algorithm for the capacitated vehicle routing problem.* European Journal of Operational Research, 2011. **215**(1): p. 126-135.

[29] Alzaqebah, M., S. Abdullah, and S. Jawarneh, *Modified artificial bee colony for the vehicle routing problems with time windows.* SpringerPlus, 2016. **5**(1): p. 1298.

[30] Yu, S., et al., *An improved artificial bee colony algorithm for vehicle routing problem with time windows: A real case in Dalian.* Advances in Mechanical Engineering, 2016. **8**(8): p. 1687814016665298.

[31] Tuntitippawan, N., and Asawarungsaengkul, K. 2016a. *An Artificial Bee Colony Algorithm for the Vehicle Routing Problem with Backhauls and Time Windows: International Conference on Industrial Engineering and Operations Management, Kuala Lumpur, Malaysia*. p.2788

[32] Tuntitippawan, N. and K. Asawarungsaengkul, *An artificial bee colony algorithm with local search for vehicle routing problem with backhauls and time windows.* Engineering and Applied Science Research, 2016. **43**: p. 404-408.

[33] Pan, F., et al., *Research on the Vehicle Routing Problem with Time Windows Using Firefly Algorithm.* Journal of Computers, 2013. **8**(9).

[34] Goel, R. and R. Maini, *A hybrid of ant colony and firefly algorithms (HAFA) for solving vehicle routing problems.* Journal of Computational Science, 2018. **25**: p. 28-37.

[35] Wang, G. and L. Guo, *A Novel Hybrid Bat Algorithm with Harmony Search for Global Numerical Optimization.* Journal of Applied Mathematics, 2013. **2013**: p. 1-21.

[36] Zhou, Y., J. Xie, and H. Zheng, *A Hybrid Bat Algorithm with Path Relinking for Capacitated Vehicle Routing Problem.* Mathematical Problems in Engineering, 2013. **2013**: p. 1-10.

[37] Gelinas, S., et al., *A new branching strategy for time constrained routing problems with application to backhauling.* Annals of Operations Research, 1995. **61**(1): p. 91-109.

[38] Dantzig, G.B. and J.H. Ramser, *The truck dispatching problem.* Management science, 1959. **6**(1): p. 80-91.

[39] Toth, P. and D. Vigo, *Vehicle routing: problems, methods, and applications*. Vol. 18. 2014: Siam.

[40] Casco, D., B. GOLDEN, and E. WASIL, *Vehicle routing with backhauls: Models, algorithms and case studies. Vehicle Routing: Methods and Studies. Studies in management science and systems-Volume 16.* Publication of: Dalctraf.

[41] Thangiah, S.R., J.-Y. Potvin, and T. Sun, *Heuristic approaches to vehicle routing with backhauls and time windows.* Computers & Operations Research, 1996. **23**(11): p. 1043-1057.

[42] Duhamel, C., J.-Y. Potvin, and J.-M. Rousseau, *A Tabu Search Heuristic for the Vehicle Routing Problem with Backhauls and Time Windows.* Transportation Science, 1997. **31**(1): p. 49-59.

[43] Reimann, M., K. Doerner, and R.F. Hartl, *Insertion Based Ants for Vehicle Routing Problems with Backhauls and Time Windows*, in *Ant Algorithms: Third International Workshop, ANTS 2002 Brussels, Belgium, September 12–14, 2002 Proceedings*, M. Dorigo, G. Caro, and M. Sampels, Editors. 2002, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 135-148.

[44] Zhong, Y. and M.H. Cole, *A vehicle routing problem with backhauls and time windows: a guided local search solution.* Transportation Research Part E: Logistics and Transportation Review, 2005. **41**(2): p. 131-144.

[45] Pisinger, D. and S. Ropke, *A general heuristic for vehicle routing problems.* Computers & Operations Research, 2007. **34**(8): p. 2403-2435.

[46] Aghdaghi, M. and F. Jolai, *A goal programming model for vehicle routing problem with backhauls and soft time windows.* Journal of Industrial Engineering, International, 2008. **4**(6): p. 7-18.

[47] Liu, R., X. Xie, V. Augusto, and C. Rodriguez, *Heuristic algorithms for a vehicle routing problem with simultaneous delivery and pickup and time windows in home health care.* European Journal of Operational Research, 2013. **230**(3): p. 475-486.

[48] Küçükoğlu, İ. and N. Öztürk, *A differential evolution approach for the vehicle routing problem with backhauls and time windows.* Journal of Advanced Transportation, 2014. **48**(8): p. 942-956.

[49] Küçükoğlu, İ. and N. Öztürk, *An advanced hybrid meta-heuristic algorithm for the vehicle routing problem with backhauls and time windows.* Computers & Industrial Engineering, 2015. **86**: p. 60-68.

[50] Kohl, N. and O.B. Madsen, *An optimization algorithm for the vehicle routing problem with time windows based on Lagrangian relaxation.* Operations Research, 1997. **45**(3): p. 395-406.

[51] Fisher, M.L., K.O. Jörnsten, and O.B. Madsen, *Vehicle routing with time windows: Two optimization algorithms.* Operations Research, 1997. **45**(3): p. 488-492.

[52] Agarwal, Y., K. Mathur, and H.M. Salkin, *A set-partitioning-based exact algorithm for the vehicle routing problem.* Networks, 1989. **19**(7): p. 731-749.

[53] Desrosiers, J., F. Soumis, and M. Desrochers, *Routing with time windows by column generation.* Networks, 1984. **14**(4): p. 545-565.

[54] Desrochers, M., J. Desrosiers, and M. Solomon, *A new optimization algorithm for the vehicle routing problem with time windows.* Operations research, 1992. **40**(2): p. 342-354.

[55] Little, J.D.C., K.G. Murty, D.W. Sweeney, C. Karel, *An Algorithm for the Traveling Salesman Problem.* Operations research, 1963. **11**(6): p. 972-989.

[56] Christofides, N. and S. Eilon, *An algorithm for the vehicle-dispatching problem.* Journal of the Operational Research Society, 1969. **20**(3): p. 309-318.

[57] Fischetti, M., P. Toth, and D. Vigo, *A Branch-and-Bound Algorithm for the Capacitated Vehicle Routing Problem on Directed Graphs.* Operations Research, 1994. **42**(5): p. 846-859.

[58] Bellmore, M. and J.C. Malone, *Pathology of traveling-salesman subtour-elimination algorithms.* Operations Research, 1971. **19**(2): p. 278-307.

[59] Baldacci, R., E. Hadjiconstantinou, and A. Mingozzi, *An Exact Algorithm for the Capacitated Vehicle Routing Problem Based on a Two-Commodity Network Flow Formulation.* Operations Research, 2004. **52**(5): p. 723-738.

[60] Cordeau, J.-F., *A branch-and-cut algorithm for the dial-a-ride problem.* Operations Research, 2006. **54**(3): p. 573-586.

[61] Dell'Amico, M., G. Righini, and M. Salani, *A branch-and-price approach to the vehicle routing problem with simultaneous distribution and collection.* Transportation Science, 2006. **40**(2): p. 235-247.

[62] Gutiérrez-Jarpa, G., G. Desaulniers, G. Laporte, and V. Marianov, *A branch-and-price algorithm for the Vehicle Routing Problem with Deliveries, Selective Pickups and Time Windows.* European Journal of Operational Research, 2010. **206**(2): p. 341-349.

[63] Ropke, S. and J.-F. Cordeau, *Branch and cut and price for the pickup and delivery problem with time windows.* Transportation Science, 2009. **43**(3): p. 267-286.

[64] Pessoa, A., M.P. De Aragão, and E. Uchoa, *Robust branch-cut-and-price algorithms for vehicle routing problems,* in *The vehicle routing problem: Latest advances and new challenges*. 2008, Springer. p. 297-325.

[65] Gillett, B.E. and L.R. Miller, *A Heuristic Algorithm for the Vehicle-Dispatch Problem.* Operations Research, 1974. **22**(2): p. 340-349.

[66] Fisher, M.L. and R. Jaikumar, *A generalized assignment heuristic for vehicle routing.* Networks, 1981. **11**(2): p. 109-124.

[67] Beasley, J.E., *Route first—cluster second methods for vehicle routing.* Omega, 1983. **11**(4): p. 403-408.

[68] Clarke, G. and J.W. Wright, *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points.* Operations Research, 1964. **12**(4): p. 568-581.

[69] Potvin, J.-Y. and J.-M. Rousseau, *A parallel route building algorithm for the vehicle routing and scheduling problem with time windows.* European Journal of Operational Research, 1993. **66**(3): p. 331-340.

[70] Balakrishnan, N., *Simple Heuristics for the Vehicle Routeing Problem with Soft Time Windows.* Journal of the Operational Research Society, 1993. **44**(3): p. 279-287.

[71] Dullaert, W., *A sequential insertion heuristic for the vehicle routing problem with time windows with relatively few customers per route*. 2000.

[72] Ioannou, G., M. Kritikos, and G. Prastacos, *A greedy look-ahead heuristic for the vehicle routing problem with time windows.* Journal of the Operational Research Society, 2001. **52**(5): p. 523-537.

[73] Atkinson, J.B., *A greedy look-ahead heuristic for combinatorial optimization: an application to vehicle scheduling with time windows.* Journal of the Operational Research Society, 1994. **45**(6): p. 673-684.

[74] Pang, K.-W., *An adaptive parallel route construction heuristic for the vehicle routing problem with time windows constraints.* Expert Systems with Applications, 2011. **38**(9): p. 11939-11946.

[75] Sheridan, P.K., E. Gluck, Q. Guan, T. Pickles, B. Balcıog˘lu, and B. Benhabib, *The dynamic nearest neighbor policy for the multi-vehicle pick-up and delivery problem.* Transportation Research Part A: Policy and Practice, 2013. **49**: p. 178-194.

[76] Salhi S, Wassan N, Hajarat M. *The fleet size and mix vehicle routing problem with backhauls: Formulation and set partitioning-based heuristics*. Transportation Research Part E, 2013; **56**: p. 22-35.

[77] Holland, J.H., *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 1992: MIT press.

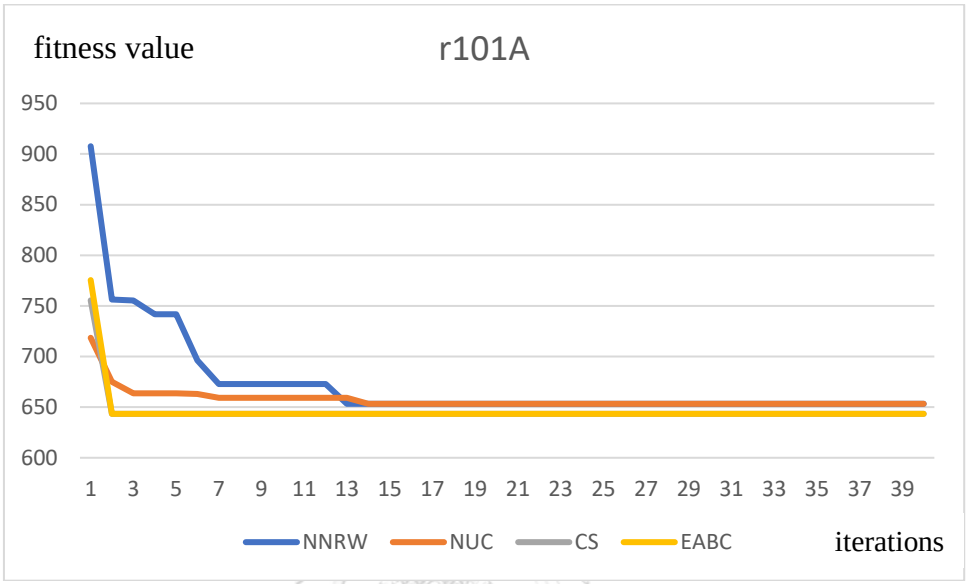
[78] Ouaarab, A., B. Ahiod, and X.-S. Yang, *Discrete cuckoo search algorithm for the travelling salesman problem.* Neural Computing and Applications, 2014. **24**(7-8): p. 1659-1669.

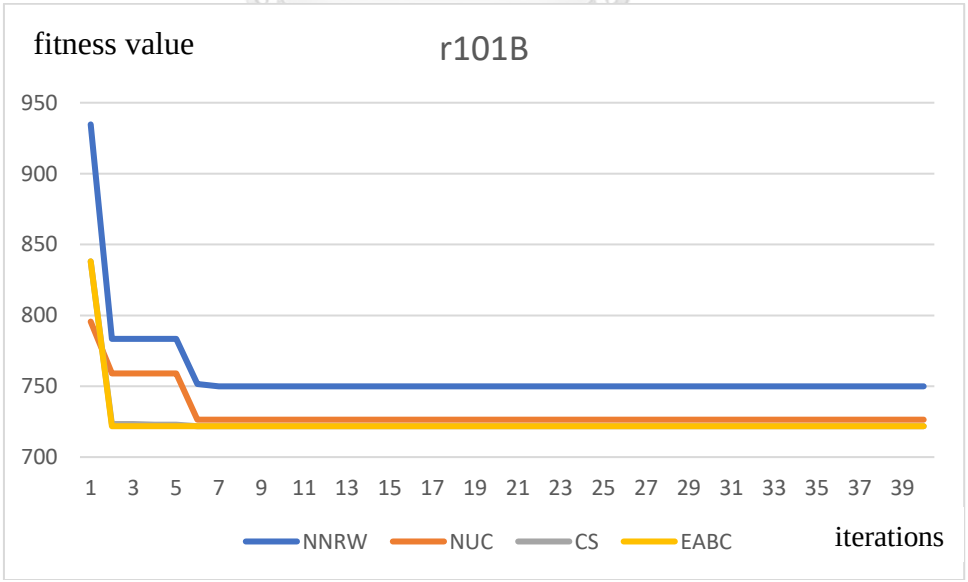
[79] Karaboga, D. and B. Basturk, *On the performance of artificial bee colony (ABC) algorithm.* Applied soft computing, 2008. **8**(1): p. 687-697.

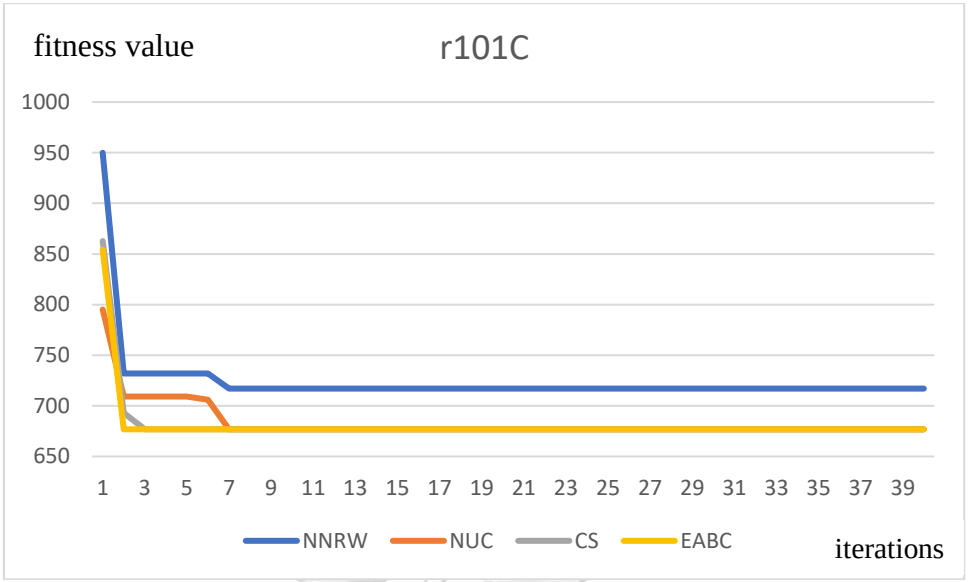
APPENDIX

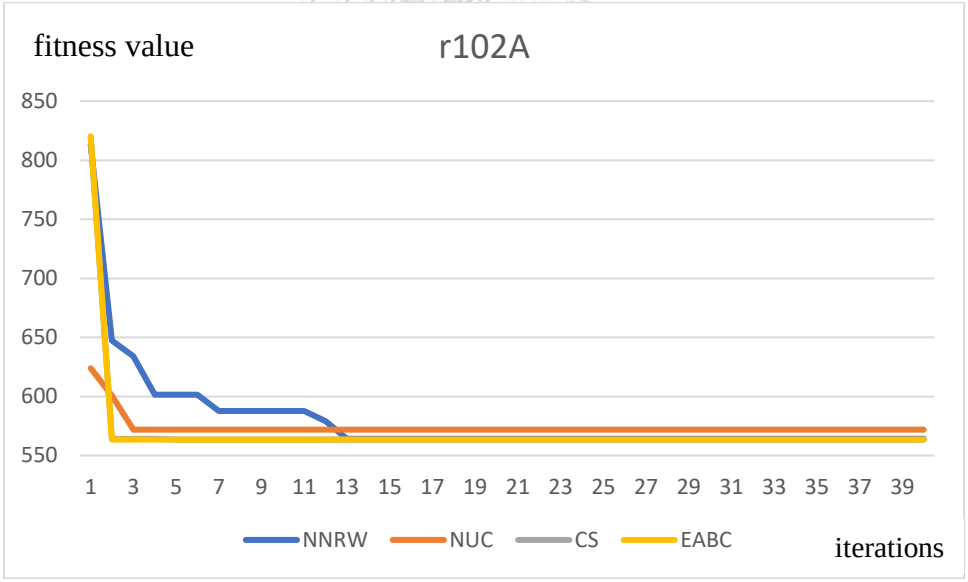
A. The 45 plots showing the relationship between the fitness value of each proposed algorithm and its number of iterations for each instance

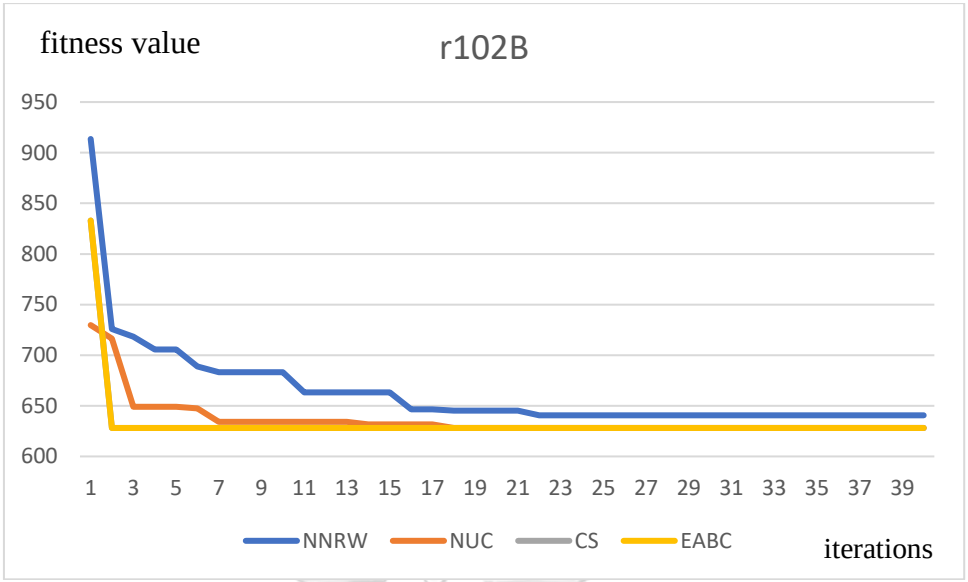
Small problems (25 customers)

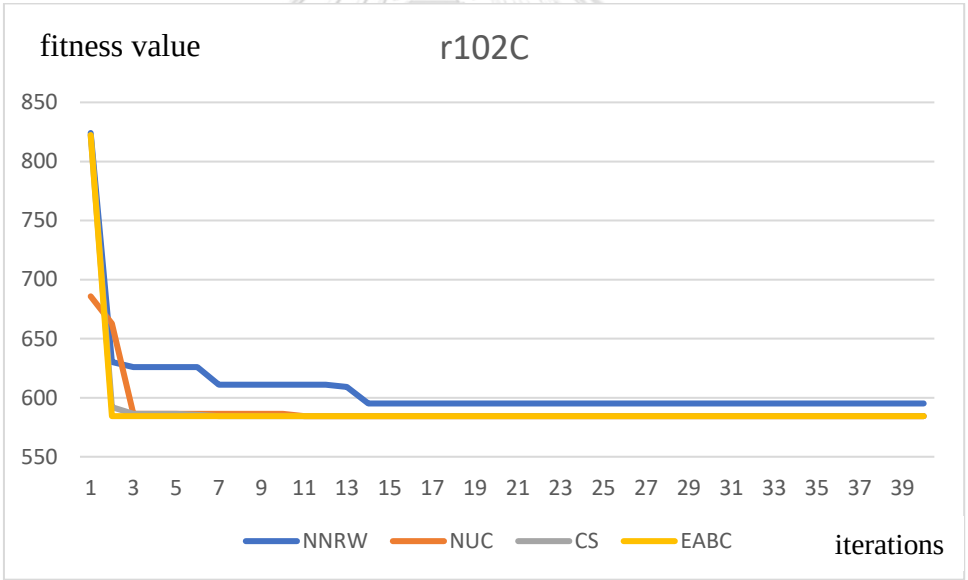


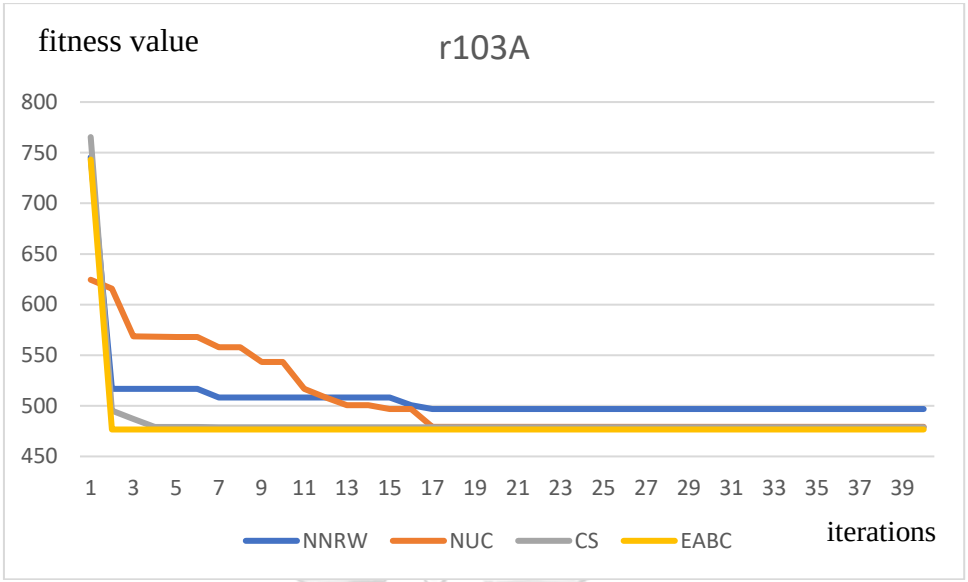


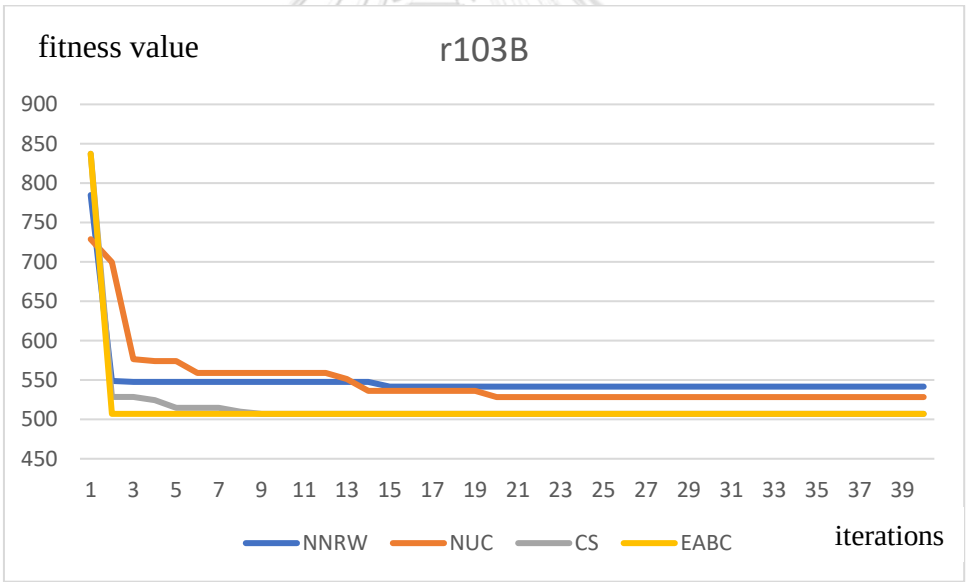

fitness value
r101C
1000
950
900
850
800
750
700
650
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

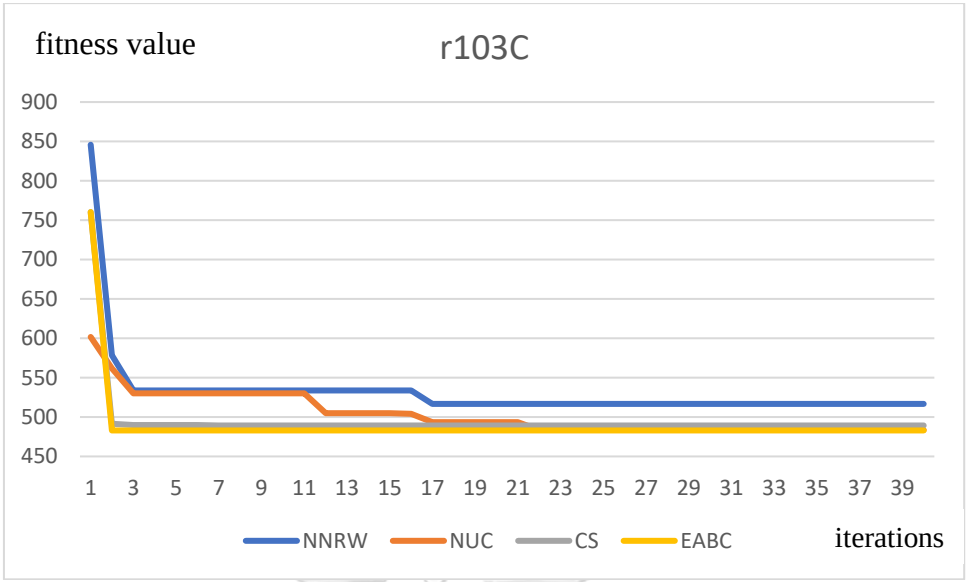

fitness value
r102A
850
800
750
700
650
600
550
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

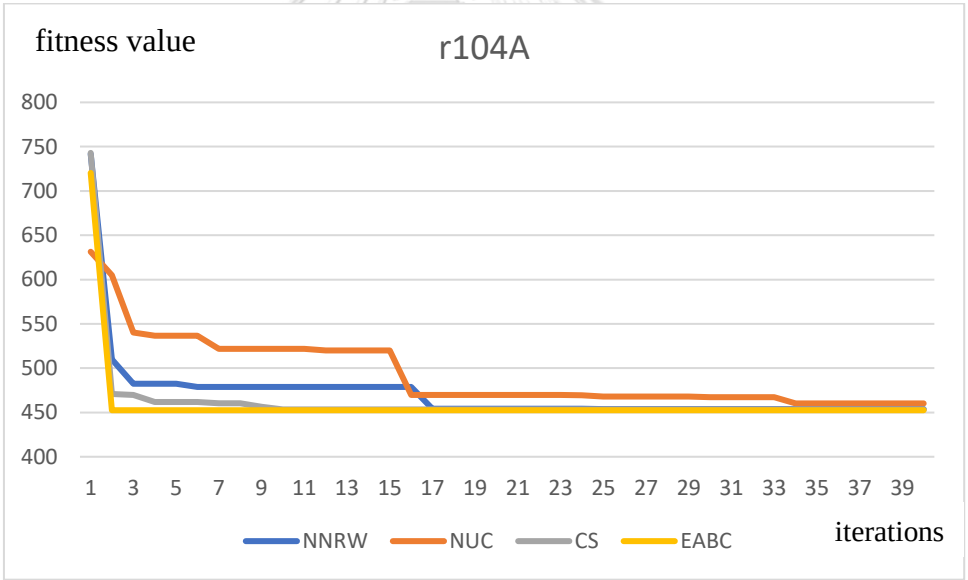

fitness value
r102B
950
900
850
800
750
700
650
600
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

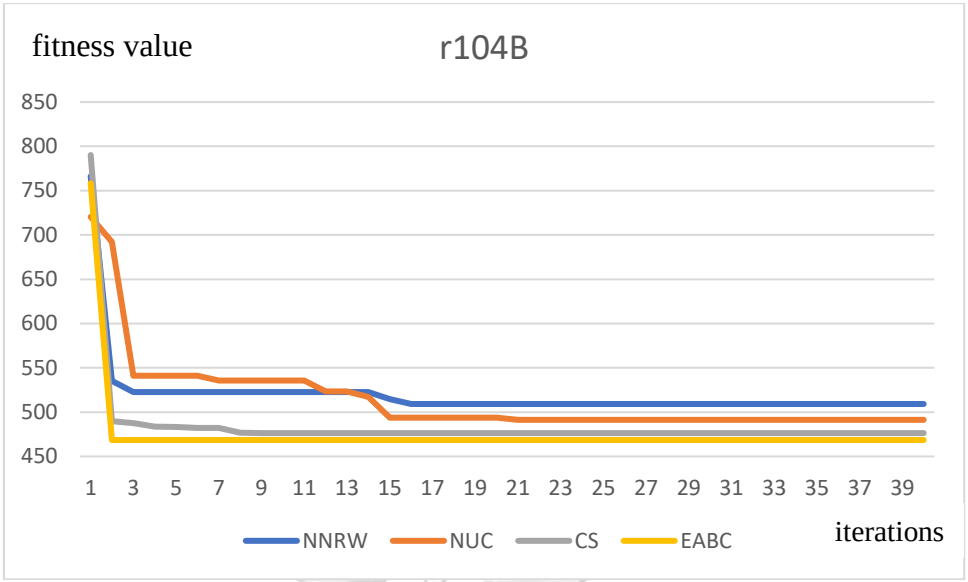

fitness value
r102C
850
800
750
700
650
600
550
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

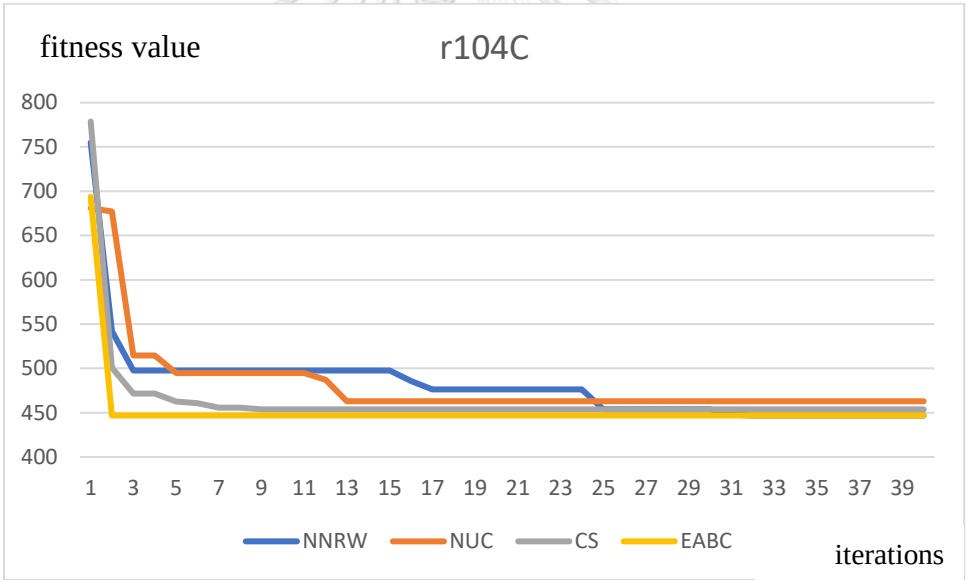

fitness value
r103A
800
750
700
650
600
550
500
450
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

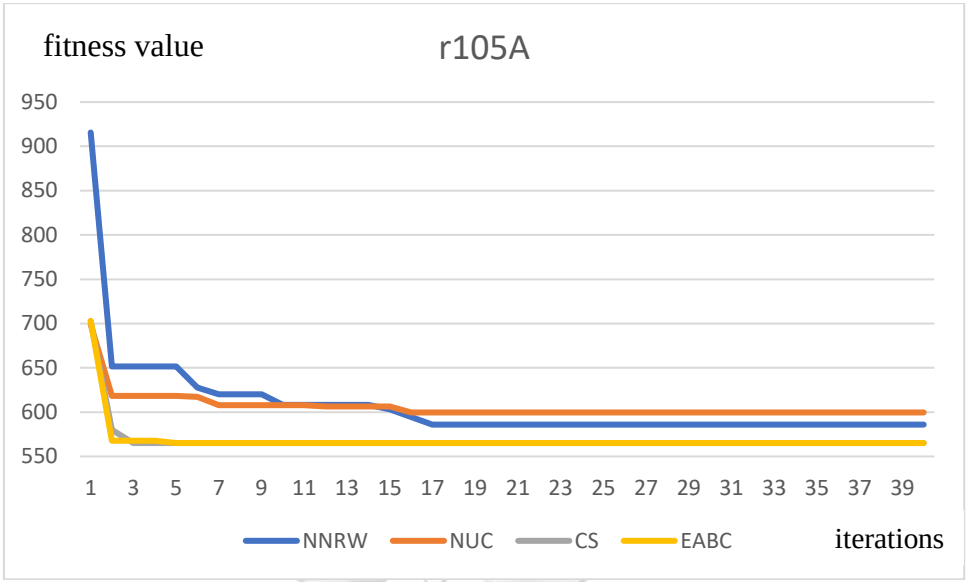

fitness value
r103B
900
850
800
750
700
650
600
550
500
450
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

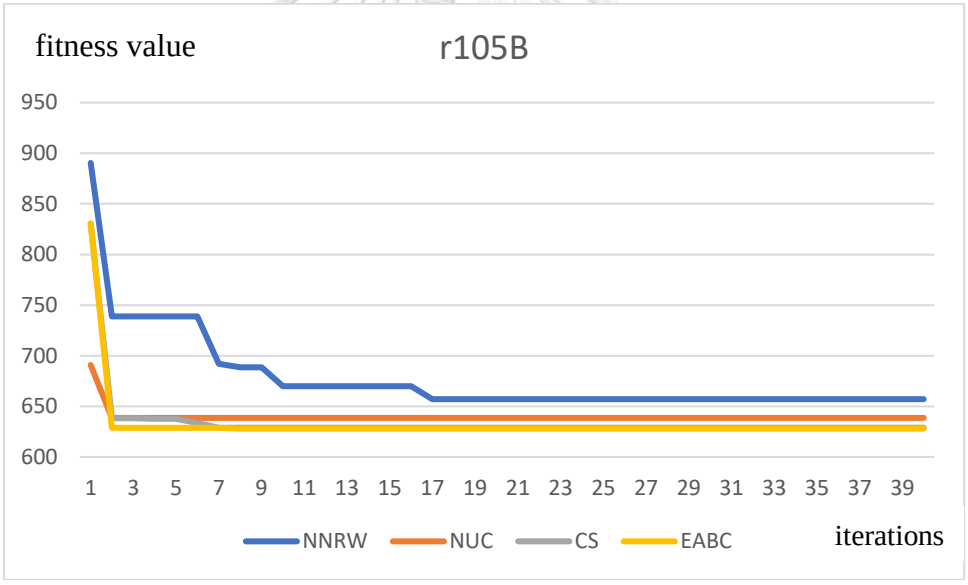

fitness value
r103C
900
850
800
750
700
650
600
550
500
450
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

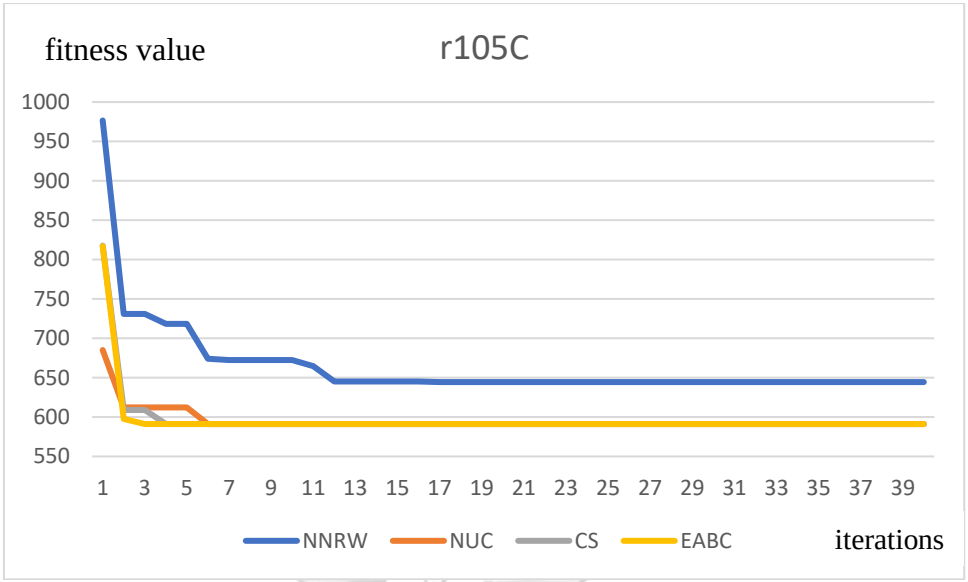

fitness value
r104A
800
750
700
650
600
550
500
450
400
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations


fitness value
r104B
850
800
750
700
650
600
550
500
450
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

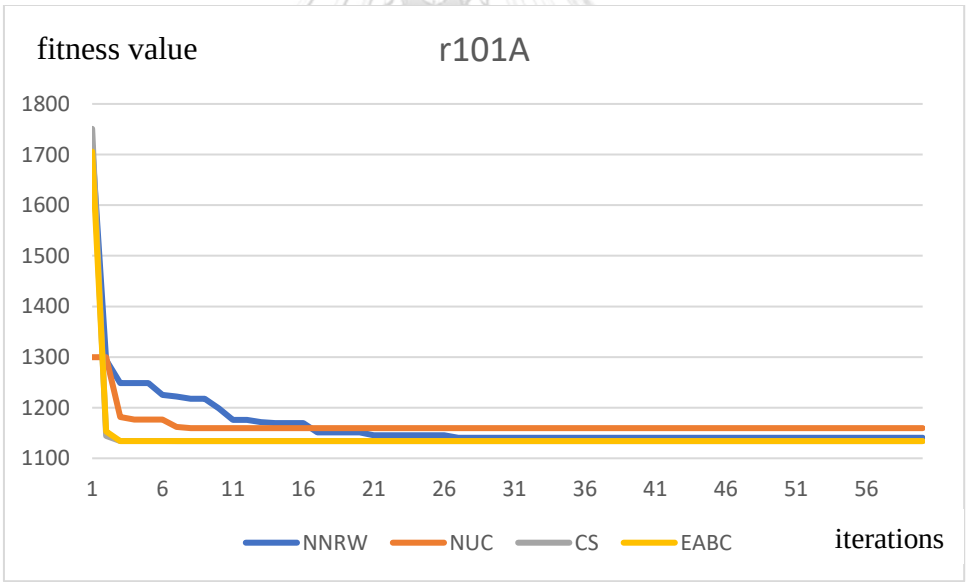

fitness value
r104C
800
750
700
650
600
550
500
450
400
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

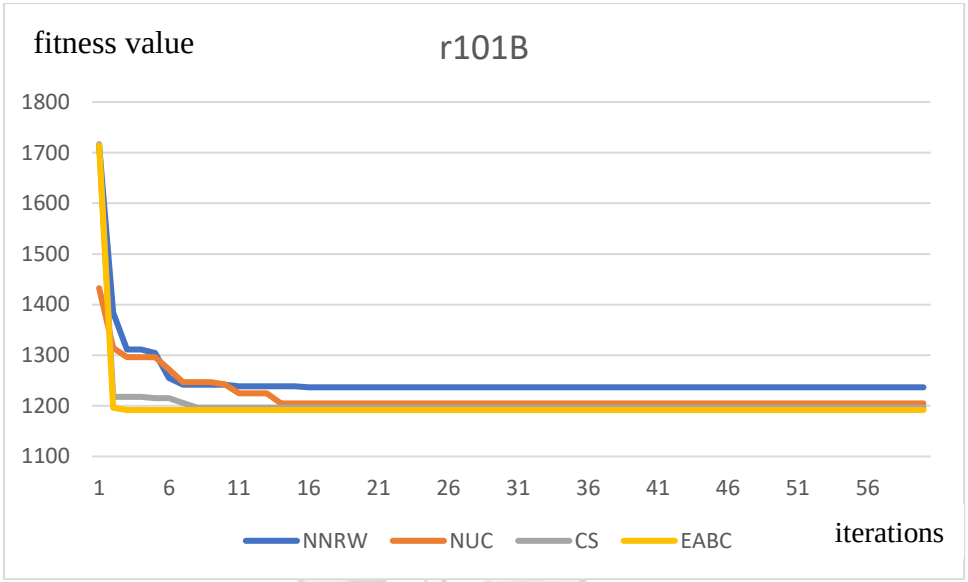

fitness value
r105A
950
900
850
800
750
700
650
600
550
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

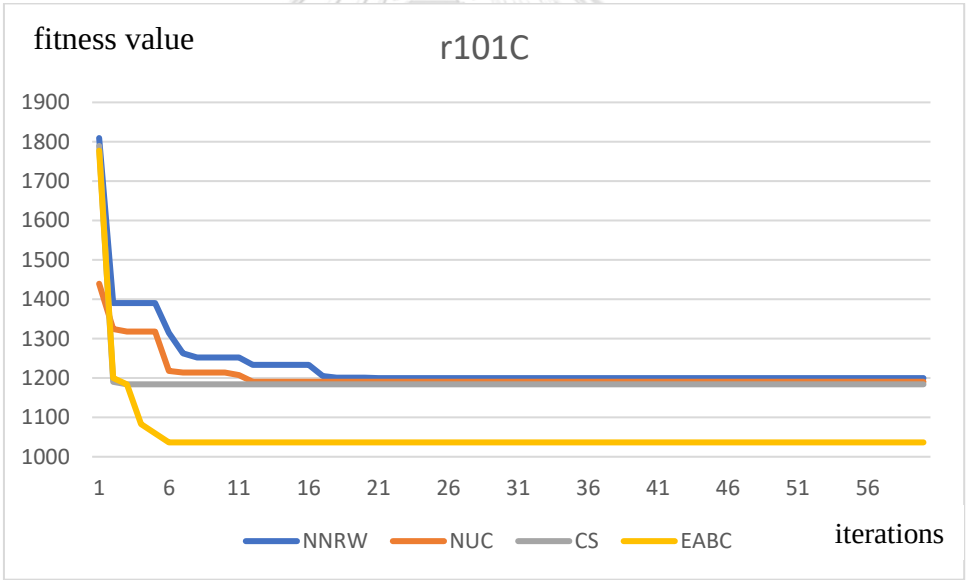

fitness value
r105B
950
900
850
800
750
700
650
600
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
NNRW
NUC
CS
EABC
iterations

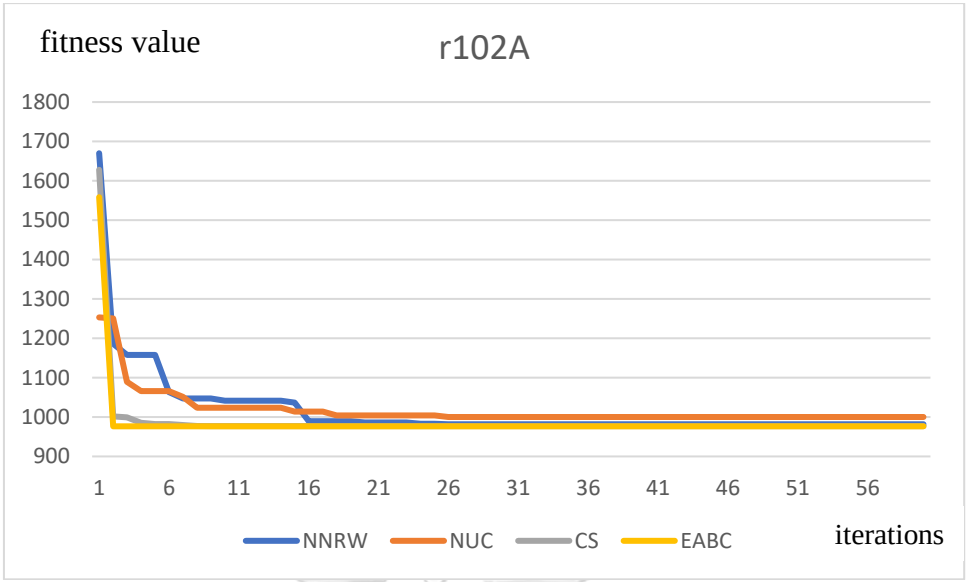


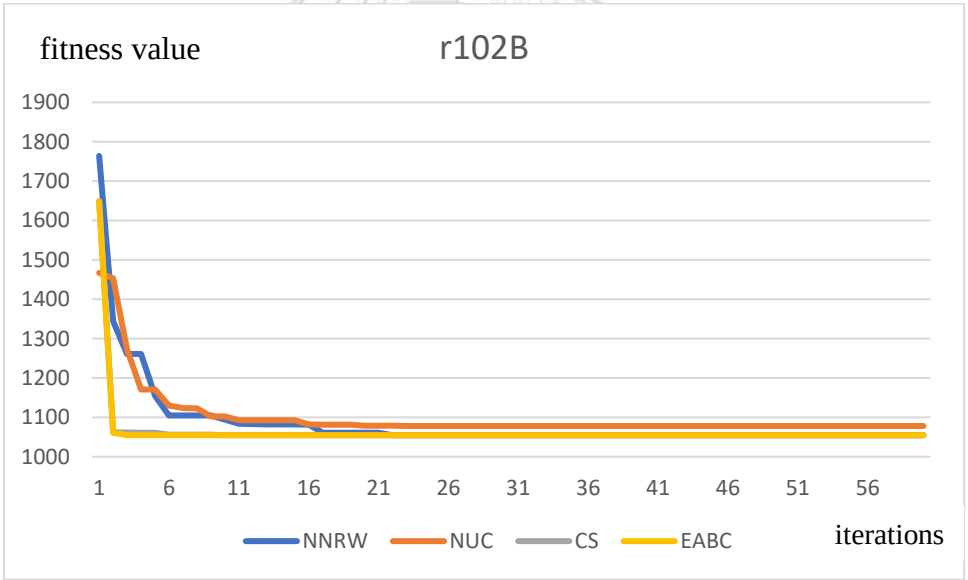
Medium problems (50 customers)

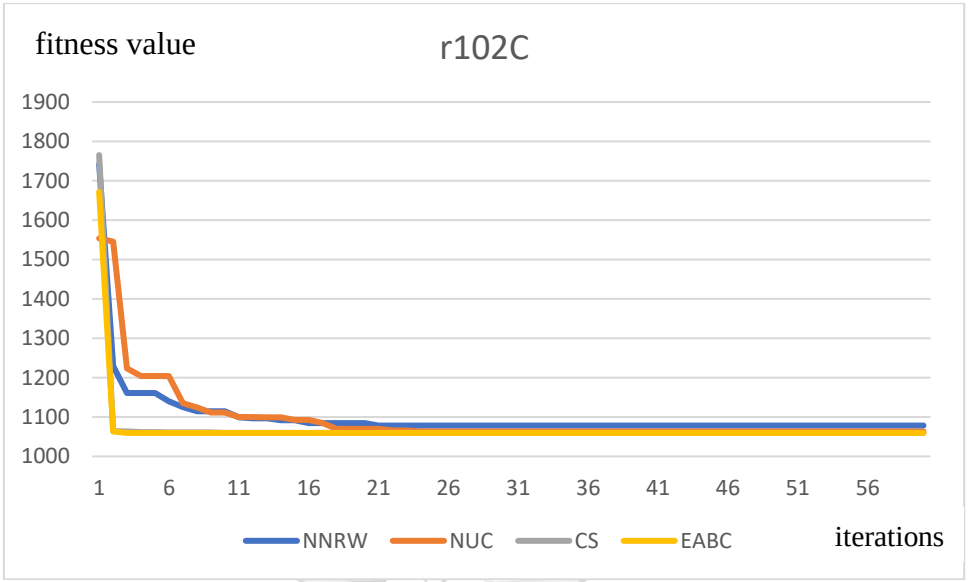


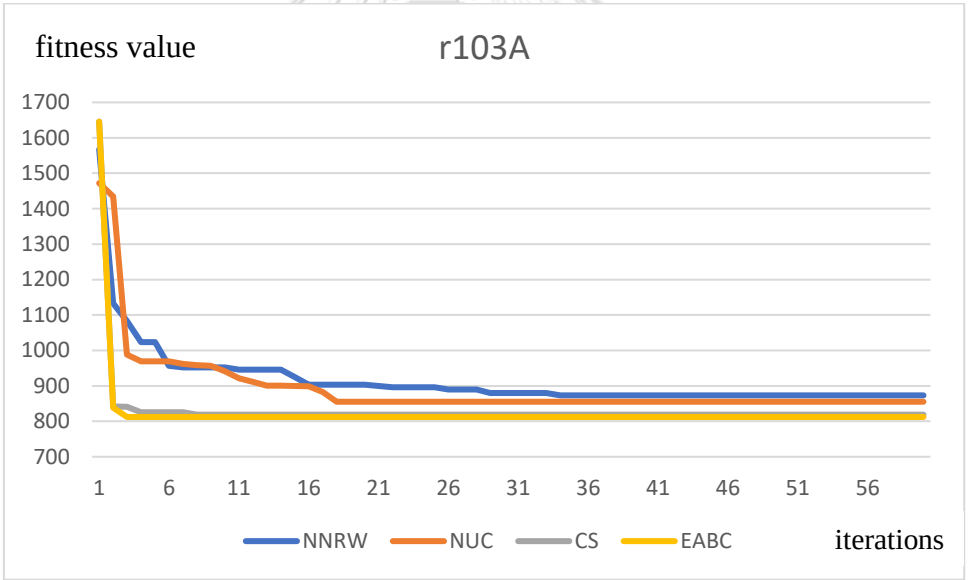

fitness value
r101B
1800
1700
1600
1500
1400
1300
1200
1100
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

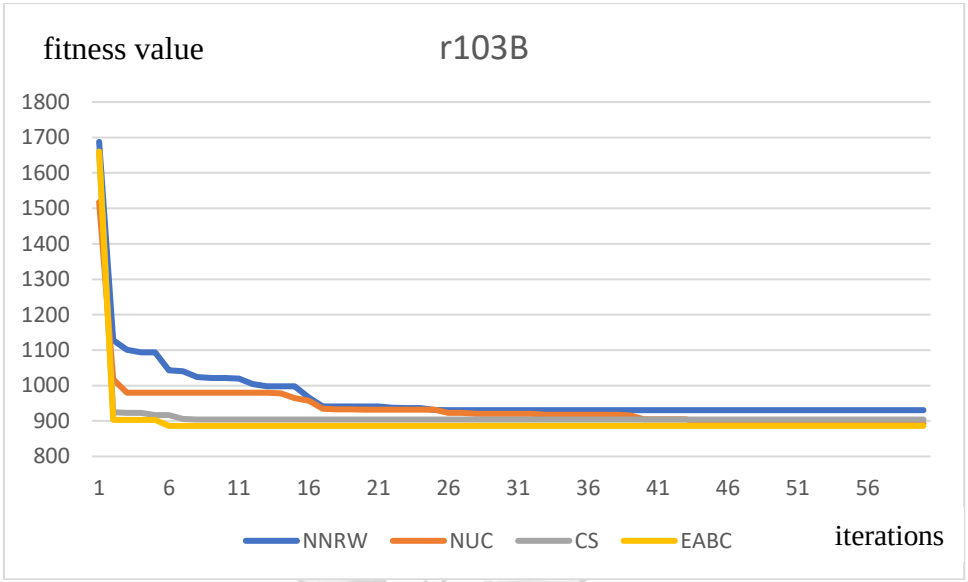

fitness value
r101C
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

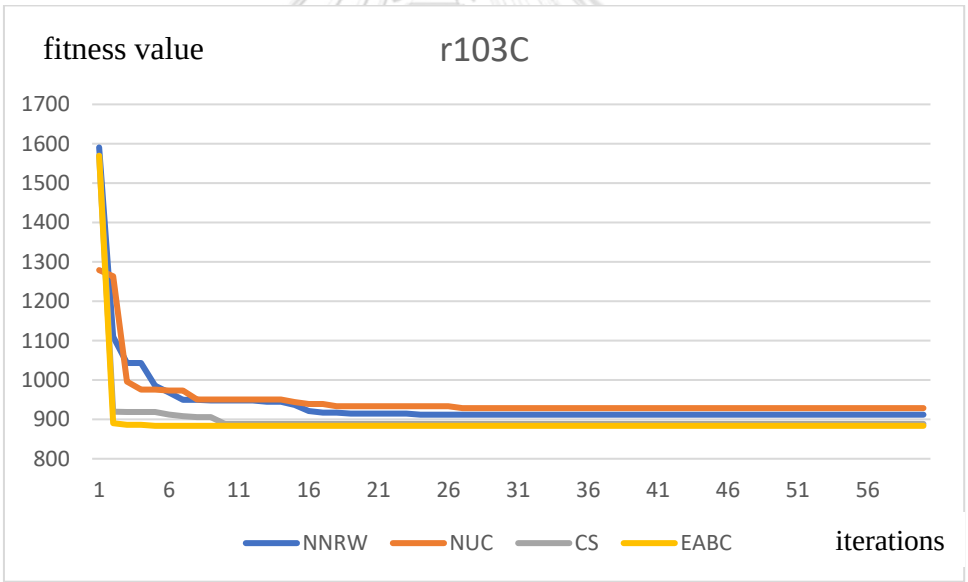

fitness value
r102A
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

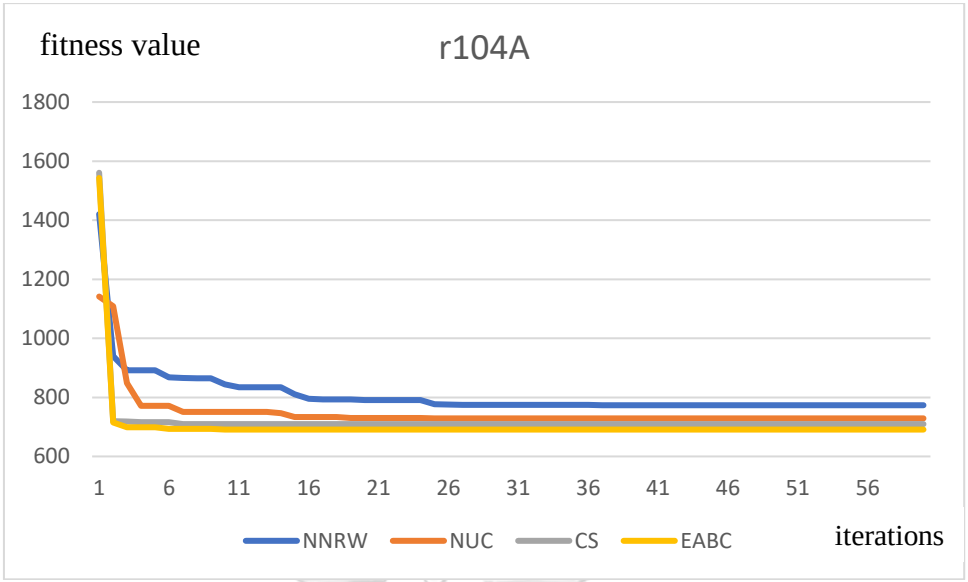

fitness value
r102B
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

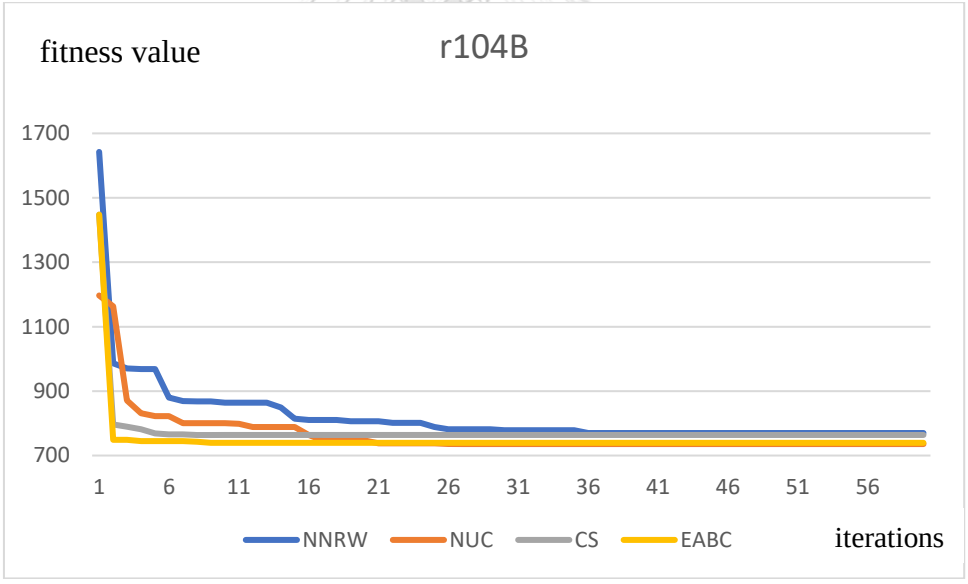

fitness value
r102C
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

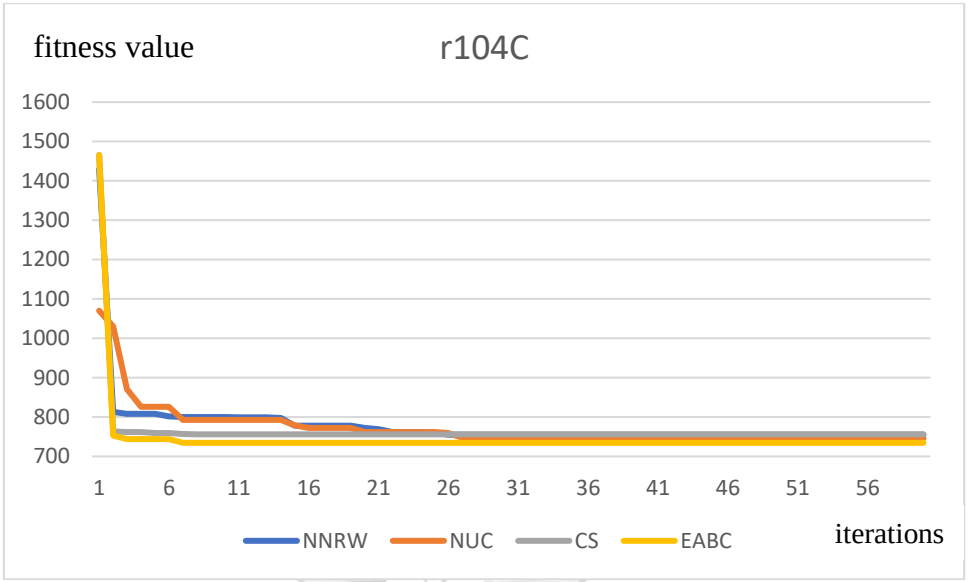

fitness value
r103A
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

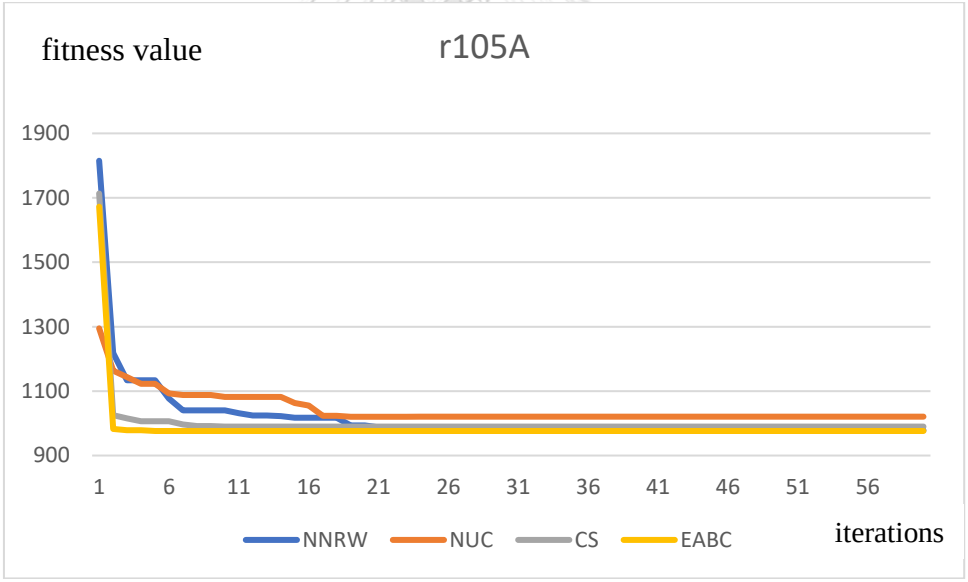

fitness value
r103B
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

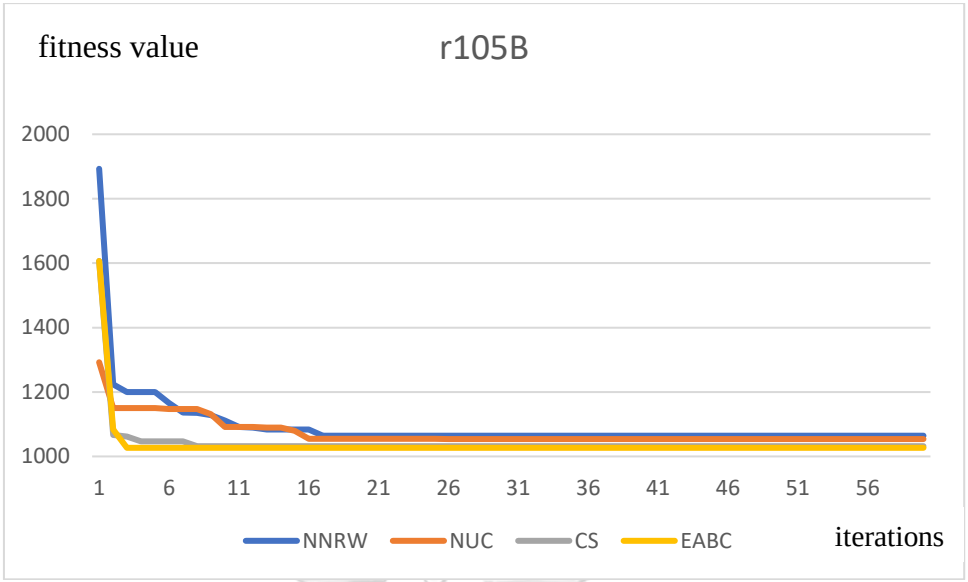

fitness value
r103C
1700
1600
1500
1400
1300
1200
1100
1000
900
800
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

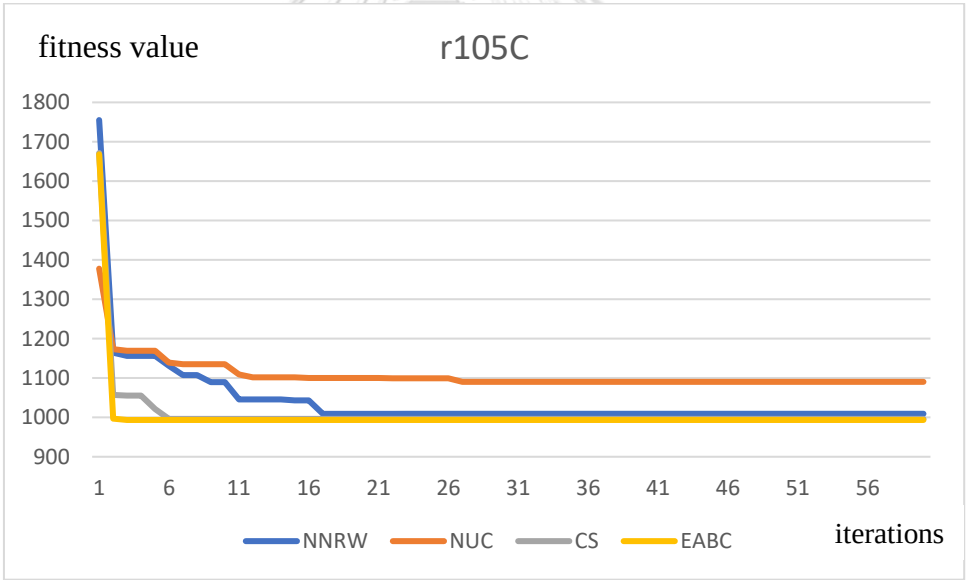

fitness value
r104A
1800
1600
1400
1200
1000
800
600
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations


fitness value
r104B
1700
1500
1300
1100
900
700
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

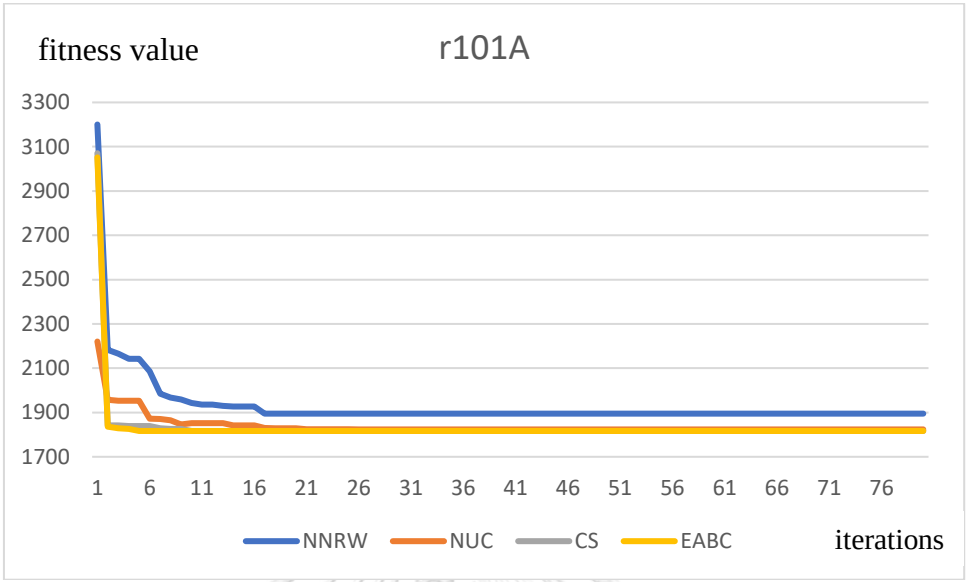

fitness value
r104C
1600
1500
1400
1300
1200
1100
1000
900
800
700
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

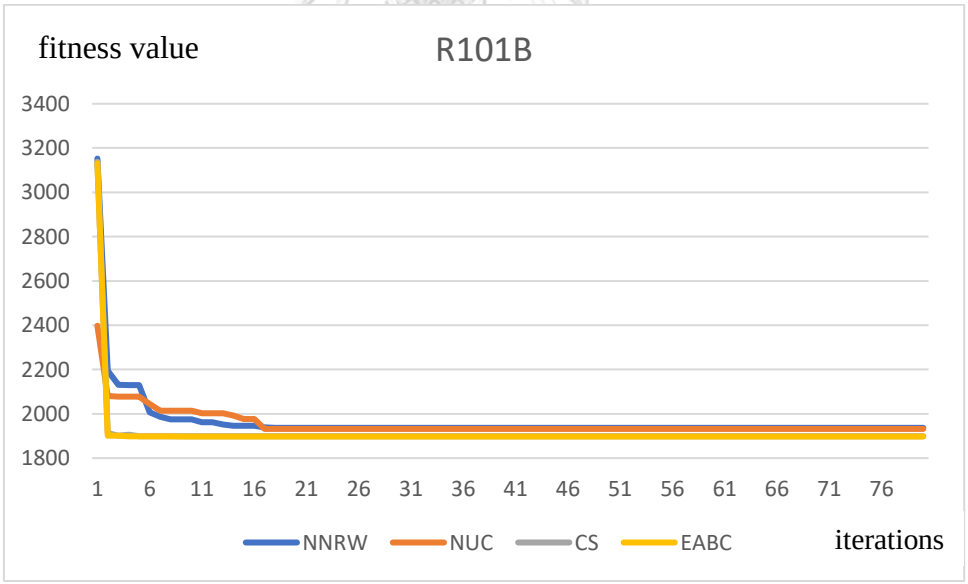

fitness value
r105A
1900
1700
1500
1300
1100
900
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

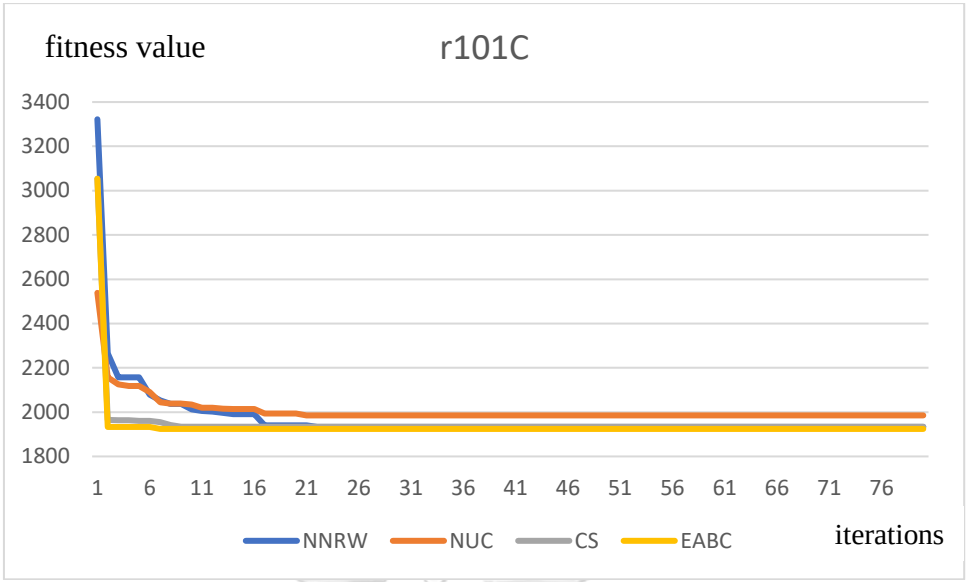

fitness value
r105B
2000
1800
1600
1400
1200
1000
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

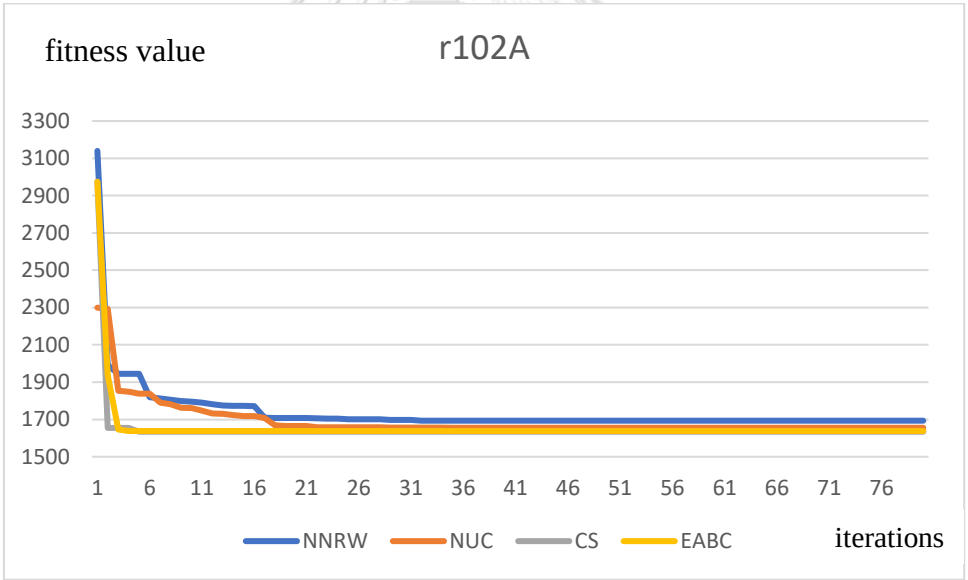

fitness value
r105C
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
1
6
11
16
21
26
31
36
41
46
51
56
NNRW
NUC
CS
EABC
iterations

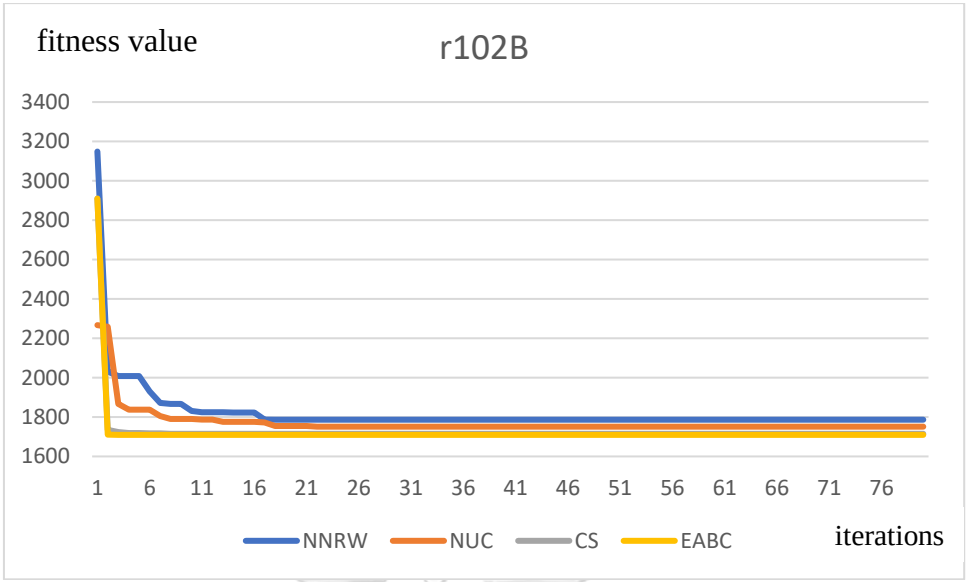
Large problems (100 customers)

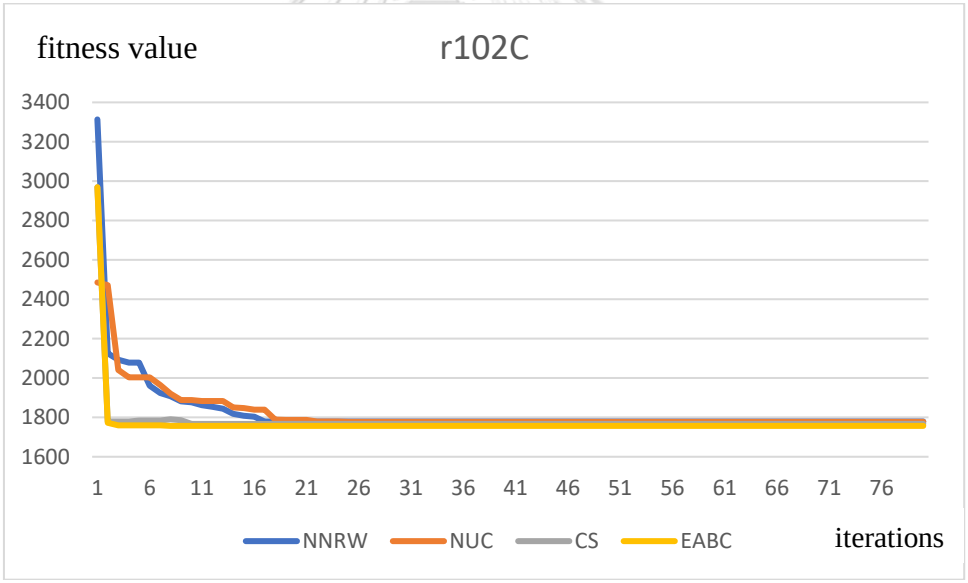


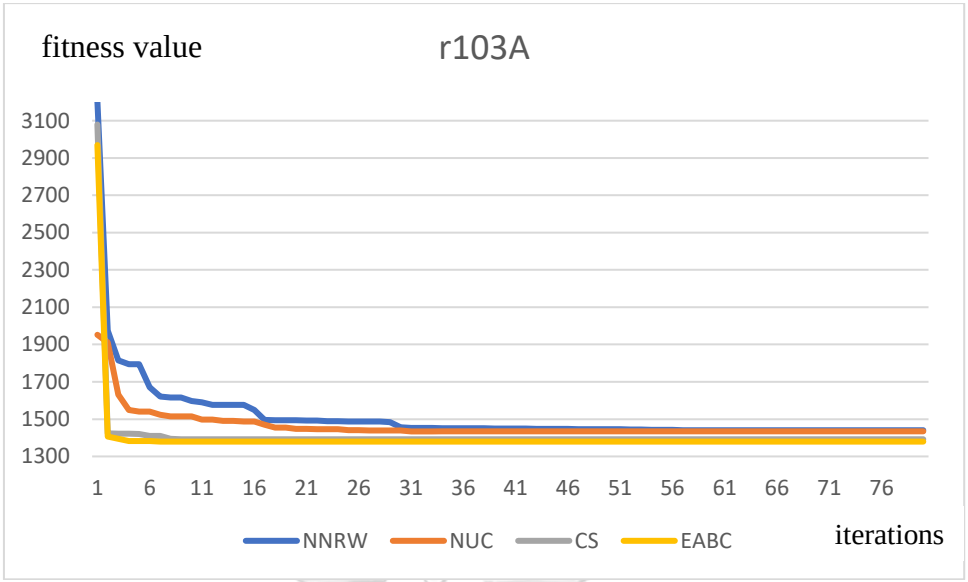


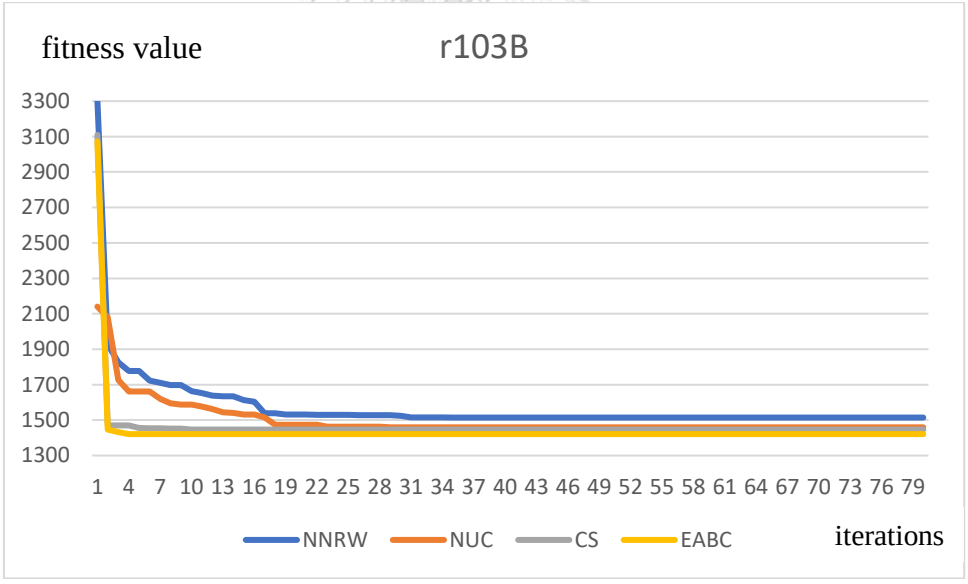

fitness value
r101C
3400
3200
3000
2800
2600
2400
2200
2000
1800
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations

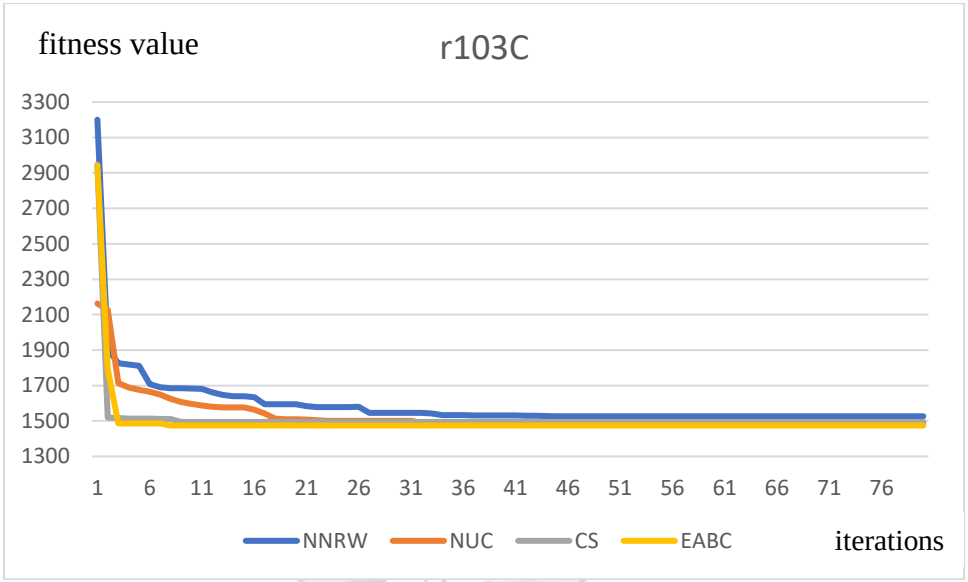

fitness value
r102A
3300
3100
2900
2700
2500
2300
2100
1900
1700
1500
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations

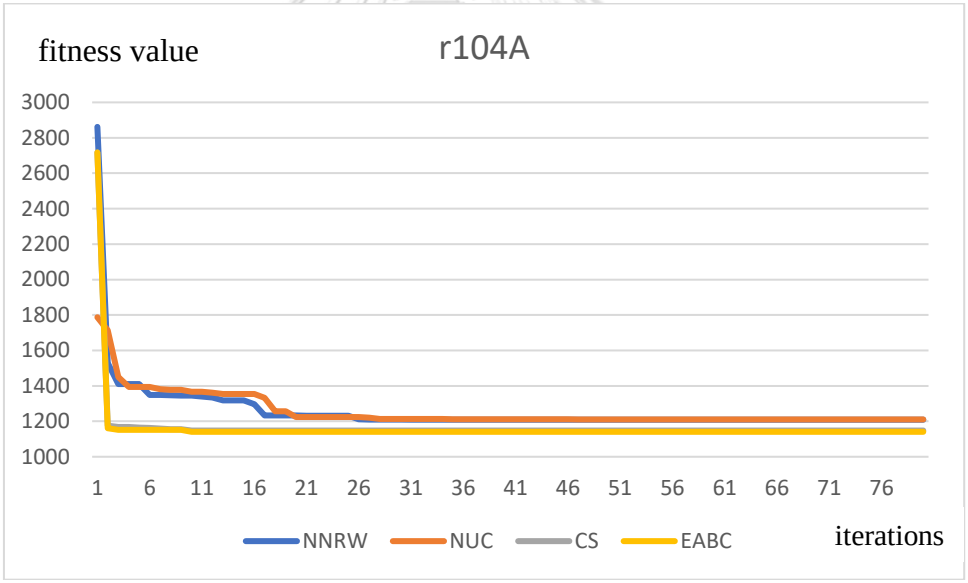

fitness value
r102B
3400
3200
3000
2800
2600
2400
2200
2000
1800
1600
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations

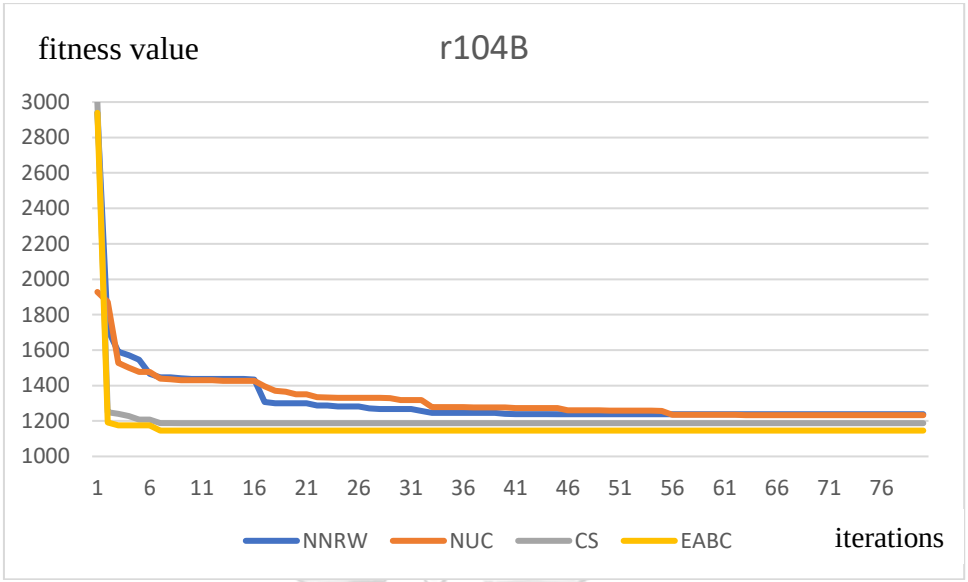

fitness value
r102C
3400
3200
3000
2800
2600
2400
2200
2000
1800
1600
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations

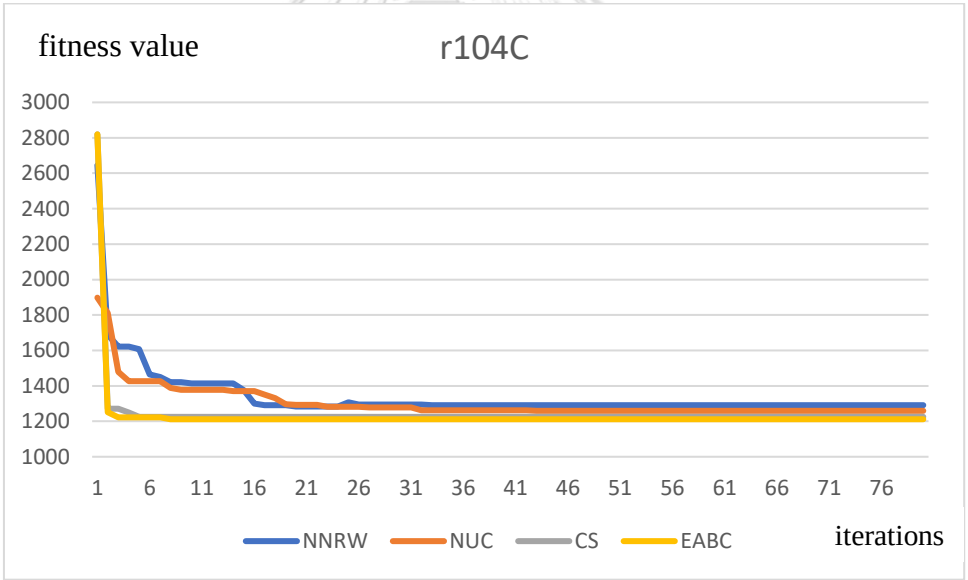

fitness value
r103A
3100
2900
2700
2500
2300
2100
1900
1700
1500
1300
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations

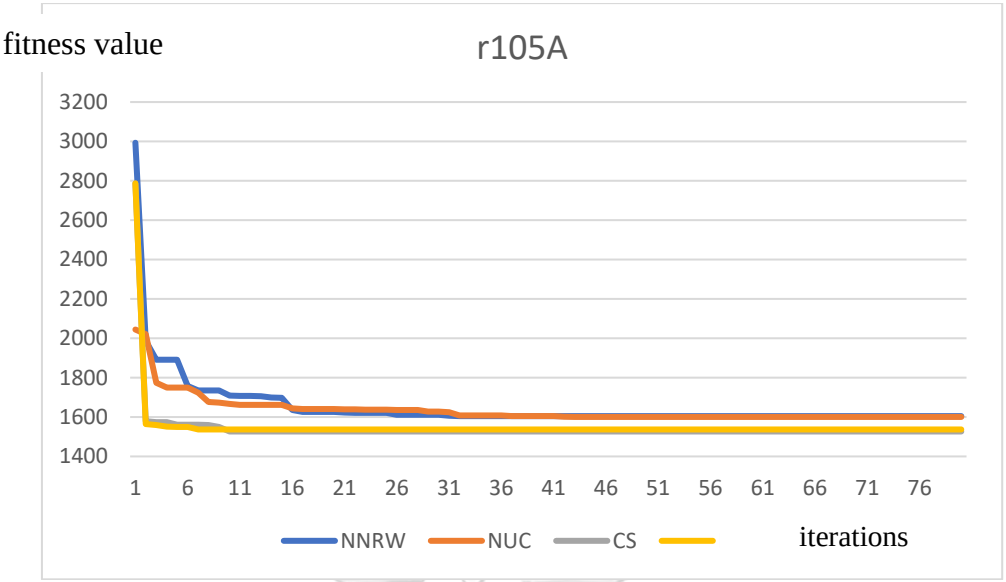

fitness value
r103B
3300
3100
2900
2700
2500
2300
2100
1900
1700
1500
1300
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79
NNRW
NUC
CS
EABC
iterations

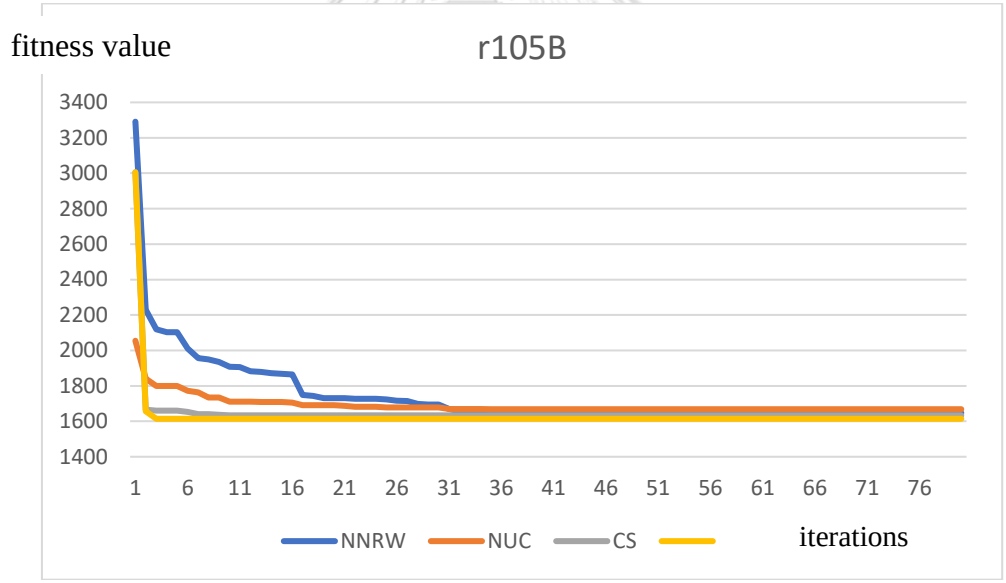

fitness value
r103C
3300
3100
2900
2700
2500
2300
2100
1900
1700
1500
1300
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations

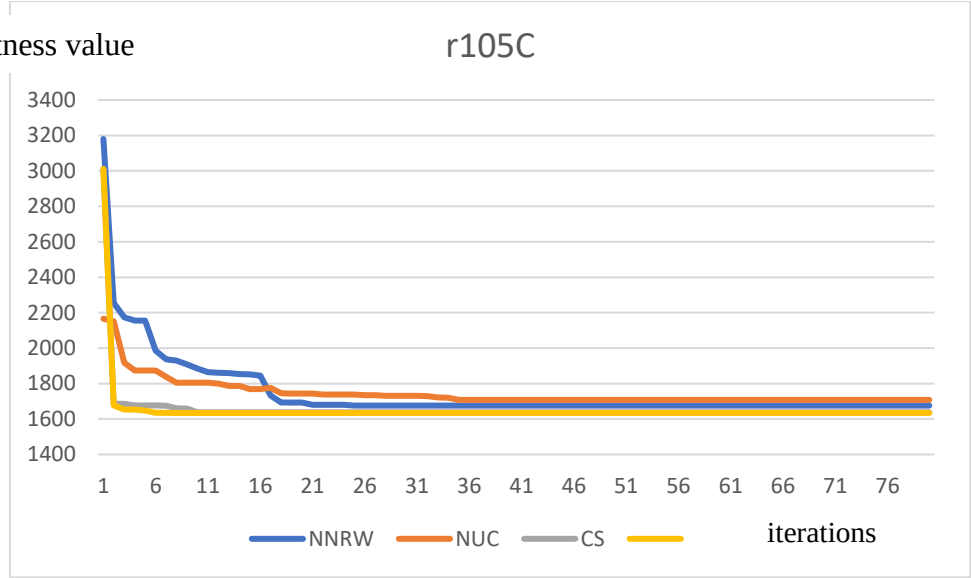

fitness value
r104A
3000
2800
2600
2400
2200
2000
1800
1600
1400
1200
1000
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations


fitness value
r104B
3000
2800
2600
2400
2200
2000
1800
1600
1400
1200
1000
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations


fitness value
r104C
3000
2800
2600
2400
2200
2000
1800
1600
1400
1200
1000
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
EABC
iterations


fitness value
r105A
3200
3000
2800
2600
2400
2200
2000
1800
1600
1400
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
iterations


fitness value
r105B
3400
3200
3000
2800
2600
2400
2200
2000
1800
1600
1400
1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76
NNRW
NUC
CS
iterations



B. Code of CPLEX program for mathematical model

```
int          NumCus = ...;
int          NumCusLine=...;
int          NumVeh = ...;
range Cus = 1..(NumCus+1); //Cus+Depot
range CusLine=1..(NumCusLine+1);//CusLine+Depot
range CusBack=(NumCusLine+2)..(NumCus+1);//Cusback
range Veh = 1..NumVeh;
int       VehCapa[Veh] = ...;
int       CusDemand[Cus] = ...;// CusDemand[Depot] = 0
float Dis[Cus,Cus] = ...;
dvar boolean      X[Cus,Cus,Veh];   // Customer Selection
dvar int+         Y[Cus,Cus];
float e[Cus]=...;
float l[Cus]=...;
float w[Cus]=...;
//dvar boolean    Y[Veh];      // Veh Selection
dexpr int   VehCapaUse[k in Veh] = sum(i,j in Cus)
CusDemand[j]*X[i][j][k];
dvar float+ t[Cus,Veh];
```

```
      dexpr float       OBJ = sum(i,j in Cus, k in Veh)
(Dis[i][j]*X[i][j][k]);
      //-----------------------Main-------------------------------
      minimize  OBJ;
      subject to {
            con1:
            forall( j in Cus : j != 1 )
                  sum(i in Cus, k in Veh) X[i][j][k] == 1;
            con2:
            forall( i in Cus : i != 1 )
                  sum(j in Cus, k in Veh) X[i][j][k] == 1;
            con3:
            forall(s in Cus, k in Veh) //: s != 1 may not use
                  sum(i in Cus) X[i][s][k] - sum(j in Cus) X[s][j][k]
== 0;
            con4:
            forall(j in CusLine: j != 1)
                  sum(i in CusLine) Y[i][j]== sum(l in Cus)
Y[j][l]+CusDemand[j];
            con5:
            forall(j in CusBack)//:j != 1  may not use
            sum(l in CusBack) (Y[j][l])+Y[j][1]== CusDemand[j]+sum(i
in Cus:i!=1) Y[i][j];
            con6:
            forall(i in CusLine, j in CusBack)
                  Y[i][j]==0;
            con7:
            forall(i in CusLine)
                  Y[i][1]==0;
            con8:
            forall(i in Cus)
                  Y[i][i]==0;
            con9:
                  sum(i in CusBack) Y[i][1] == sum(i in CusBack)
CusDemand[i];
            con10:
                  sum(j in CusLine:j!=1) Y[1][j] == sum(j in
CusLine:j!=1) CusDemand[j];
```

```
            con11:
            forall(i in CusBack,j in CusLine:j!=1, k in Veh)
                  X[i][j][k]==0;
            con12:
            forall(i in Cus,j in Cus:i!=j)
                  Y[i][j]<=sum(k in Veh) X[i][j][k]*VehCapa[k];
            con13:
            forall(k in Veh,j in Cus:j!=1)
                  Dis[1][j] - t[j][k]<=10000000*(1-X[1][j][k]);
            con14:
            forall(k in Veh, i in Cus:i!=1)
                  t[i][k]+w[i]+Dis[i][1]-t[1][k] <=10000000*(1-
X[i][1][k]);
            con15:
            forall(k in Veh,i in Cus:i!=1, j in Cus:j!=1)
                  t[i][k]+w[i]+Dis[i][j]-t[j][k] <=10000000*(1-
X[i][j][k]);
            con16_1:
            forall(k in Veh, i in Cus:i!=1)
                  e[i]<=t[i][k];
            con16_2:
            forall(k in Veh, i in Cus:i!=1)
                  t[i][k]<=l[i];

            con17_1:
            forall(k in Veh)
                  0<=t[1][k];     //T min
            con17_2:
            forall(k in Veh)
                  t[1][k]<=l[1];  //T max
      }
```


C. Code of C# program for NN heuristic

```
//int[] seqCus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25};
//int[] seqCus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50
};
```

```
int[] seqCus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
95, 96, 97, 98, 99, 100 };
            int[,] vehicle = new int[numVehicle + 1, NumCusForIndex];
            double alpha = 1.0;
            // set u
            int[] vehicleTemp = new int[NumCusForIndex];
            int countIndexVehicle = 1;
            int u = seqCus[0];
            vehicleTemp[countIndexVehicle] = u;
            seqCus = seqCus.Except(new int[] { u }).ToArray();
            double TimeArriveAtU = dis[0, u];
            if (TimeArriveAtU < Node[u].earlytime)
            {
                TimeArriveAtU = Node[u].earlytime;
            }
            double TotalCapLine = 0;
            double TotalCapBack = 0;
            if (Node[u].type == true)
            {
                TotalCapLine = Node[u].demand;
            }
            else
            {
                TotalCapBack = Node[u].demand;
            }
            int k = 1; // num of vehicle use
            while (seqCus.Length != 0)
            {
                // build temp seq
                int[] seqCusTemp = new int[seqCus.Length];
                for (int i = 0; i < seqCus.Length; i++)
                {
                    seqCusTemp[i] = seqCus[i];
                }
                // find the proper v for adding route after u
                int BestV = 0; // v that is properly
                double TimeAvirreAtBestV = 0;
                double MinCost = 10000;
                bool ExistV = false;
                while (seqCusTemp.Length != 0)
                {
                   if (Node[u].type == true)  // u-line
                    {
                        int v = seqCusTemp[0];
                        seqCusTemp = seqCusTemp.Except(new int[] { v
                                  }).ToArray();
                        // check cap fesibility
                        if (Node[v].type == true) // v-line
                        {
                            if (TotalCapLine + Node[v].demand <= cap)
                            {
                                double TimeArriveAtV = TimeArriveAtU +
                                  Node[u].servicetime + dis[u, v];
                                double WaittingTime = Math.Max(0,
                                  Node[v].earlytime - TimeArriveAtV);
```

```
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v];
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                    }
                }// end if time

            }// end if cap
        }
        else // v-back
        {
            if (TotalCapBack + Node[v].demand <= cap)
            {
                double TimeArriveAtV = TimeArriveAtU +
                        Node[u].servicetime + dis[u, v];
                double WaittingTime = Math.Max(0,
                        Node[v].earlytime - TimeArriveAtV);
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v];
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;

                    }
                }// end if time

            }// end if cap
        }
    }
    else  // u-back
    {
        int v = seqCusTemp[0]; ;
        while (Node[v].type == true)
        {
            seqCusTemp = seqCusTemp.Except(new int[] { v
                  }).ToArray();
            if (seqCusTemp.Length == 0)
            {
                v = 0;
                break;
```

```
                }
                v = seqCusTemp[0];
            }
            if (v == 0)
            {
                break;
            }
            seqCusTemp = seqCusTemp.Except(new int[] { v
                }).ToArray();
            if (TotalCapBack + Node[v].demand <= cap)
            {
                double TimeArriveAtV = TimeArriveAtU +
                                Node[u].servicetime + dis[u, v];
                double WaittingTime = Math.Max(0, Node[v].earlytime
                                    - TimeArriveAtV);
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v];
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                    }
                }// end if time
            }// end if cap
        }// end else
    }// end
    // update veh temp
    if (ExistV == true)
    {
        countIndexVehicle++;
        vehicleTemp[countIndexVehicle] = BestV;
        seqCus = seqCus.Except(new int[] { BestV }).ToArray();
        // set new u
        u = BestV;
        TimeArriveAtU = TimeAvirreAtBestV;
        if (Node[u].type == true)
        {
            TotalCapLine = TotalCapLine + Node[BestV].demand;
        }
        else
        {
            TotalCapBack = TotalCapBack + Node[BestV].demand;
        }
    }
    else //can't add any more
    {
        //update to real veh
        for (int i = 1; i < vehicleTemp.Length; i++)
        {
            vehicle[k, i] = vehicleTemp[i];
            vehicleTemp[i] = 0;
```

```
            }
            //start new vehicle
            k++;
            //set u
            countIndexVehicle = 1;
            u = seqCus[0];
            vehicleTemp[countIndexVehicle] = u;
            seqCus = seqCus.Except(new int[] { u }).ToArray();
            TimeArriveAtU = dis[0, u];
            if (TimeArriveAtU < Node[u].earlytime)
            {
                TimeArriveAtU = Node[u].earlytime;
            }
            if (Node[u].type == true)
            {
                TotalCapLine = Node[u].demand;
                TotalCapBack = 0;
            }
            else
            {
                TotalCapBack = Node[u].demand;
                TotalCapLine = 0;
            }
        }
    } // end while seq
    //add last veh temp to real veh
    if (seqCus.Length == 0)
    {
        //update to real veh
        for (int i = 1; i < vehicleTemp.Length; i++)
        {
            vehicle[k, i] = vehicleTemp[i];
            vehicleTemp[i] = 0;
        }
    }
    // build index
    int[] index = new int[NumCusForIndex];
    int count3 = 1;
    for (int i = 1; i <= k; i++)
    {
        int count1 = 1;
        while (vehicle[i, count1] != 0)
        {
            index[count3] = vehicle[i, count1];
            count1++;
            count3++;
        }
        count3++;
    }
    // updated time arrive and capacity
    double[] capForindex = new double[index.Length];
    double[] timeArrive = new double[index.Length];
    double temp2 = 0;
    int count15 = 1;
    int count16 = 0;
    for (int m = 1; m <= numVehicle; m++)
    {
        temp2 = 0;
        while (index[count15] != 0)
```

```
                {
                    temp2 = temp2 + Node[index[count15]].demand;
                    if (index[count15 - 1] == 0)
                    {
                        if (Node[index[count15]].earlytime == 0)
                            //Setup arrive timeArrive for first customer
                        {
                            timeArrive[count15] = dis[index[count15 - 1],
                                                      index[count15]];
                        }
                        else
                        {
                            timeArrive[count15] =
                                        Node[index[count15]].earlytime;
                        }
                    }
                    else
                    {
                        double wait = Math.Max(Node[index[count15]].earlytime -
        (timeArrive[count15 - 1] + Node[index[count15 - 1]].servicetime +
        dis[index[count15 - 1], index[count15]]), 0);
                        if (wait != 0.0)
                        {
                            timeArrive[count15] =
                                  Node[index[count15]].earlytime;
                        }
                        else
                        {
                            timeArrive[count15] = timeArrive[count15 - 1] +
Node[index[count15 - 1]].servicetime + dis[index[count15 - 1], index[count15]];
                        }
                    }
                    count15++;
                }
                if (index[count15] == 0)
                {
                    timeArrive[count15] = 0.0;
                }
                for (int n = count16; n < count15; n++)
                {
                    capForindex[n] = temp2;
                }
                count16 = count15;
                count15++; // skip depot
            }
            //backhaul
            for (int i = 1; i <= numVehicle; i++)
            {
                int count2 = 2;
                while (index[count2] != 0)
                {
                    int n1 = index[count2 - 1];
                    int n2 = index[count2];
                    if (Node[n1].type == false && Node[n2].type == true)
                    {
                        Console.WriteLine("\n -*-*-*-*-*-*Infeasible-
                                          Backhauls*-*-*-*-*-*- ");
                    }
                    count2++;
```

```
            }
            count2 = count2 + 2;
        }
         //NN end here
```

D. Code of C# program for INN heuristic

```
//INN start here
//int[] seqCus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25};
//int[] seqCus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50
};
int[] seqCus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
95, 96, 97, 98, 99, 100 };
            int[,] vehicle = new int[numVehicle + 1, NumCusForIndex];
            double alpha = 0.4;
            double beta = 0.3;
            double gramma = 0.3;
            // set u
            int[] vehicleTemp = new int[NumCusForIndex];
            int countIndexVehicle = 1;
            int u = seqCus[0];
            vehicleTemp[countIndexVehicle] = u;
            seqCus = seqCus.Except(new int[] { u }).ToArray();
            double TimeArriveAtU = dis[0, u];
            if (TimeArriveAtU < Node[u].earlytime)
            {
                TimeArriveAtU = Node[u].earlytime;
            }
            double TotalCapLine = 0;
            double TotalCapBack = 0;
            if (Node[u].type == true)
            {
                TotalCapLine = Node[u].demand;
            }
            else
            {
                TotalCapBack = Node[u].demand;
            }
            int k = 1; // num of vehicle use
            while (seqCus.Length != 0)
            {
                // build temp seq
                int[] seqCusTemp = new int[seqCus.Length];
                for (int i = 0; i < seqCus.Length; i++)
                {
                    seqCusTemp[i] = seqCus[i];
                }

                // find the proper v for adding route after u
                int BestV = 0; // v that is properly
                double TimeAvirreAtBestV = 0;
```

```
double MinCost = 10000;
bool ExistV = false;
while (seqCusTemp.Length != 0)
{
   if (Node[u].type == true)  // u-line
    {
        int v = seqCusTemp[0];
        seqCusTemp = seqCusTemp.Except(new int[] { v
            }).ToArray();
        // check cap fesibility
        if (Node[v].type == true) // v-line
        {
            if (TotalCapLine + Node[v].demand <= cap)
            {
                double TimeArriveAtV = TimeArriveAtU +
                  Node[u].servicetime + dis[u, v];
                double WaittingTime = Math.Max(0,
                  Node[v].earlytime - TimeArriveAtV);
                double urgent = Node[v].lasttime -
                  TimeArriveAtV;
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v] + beta *
                          WaittingTime+gramma*urgent;
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                    }
                }// end if time

            }// end if cap
        }
        else // v-back
        {

            if (TotalCapBack + Node[v].demand <= cap)
            {
                double TimeArriveAtV = TimeArriveAtU +
                  Node[u].servicetime + dis[u, v];
                double WaittingTime = Math.Max(0,
                  Node[v].earlytime - TimeArriveAtV);
                double urgent = Node[v].lasttime -
                  TimeArriveAtV;
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
```

```
                    double cost = alpha * dis[u, v] + beta *
                         WaittingTime+gamma * urgent;
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                    }
                }// end if time

            }// end if cap

        }
    }
    else  // u-back
    {
        int v = seqCusTemp[0]; ;
        while (Node[v].type == true)
        {
            seqCusTemp = seqCusTemp.Except(new int[] { v
                    }).ToArray();
            if (seqCusTemp.Length == 0)
            {
                v = 0;
                break;
            }
            v = seqCusTemp[0];
        }
        if (v == 0)
        {
            break;
        }
        seqCusTemp = seqCusTemp.Except(new int[] { v
                        }).ToArray();
        if (TotalCapBack + Node[v].demand <= cap)
        {
            double TimeArriveAtV = TimeArriveAtU +
                        Node[u].servicetime + dis[u, v];
            double WaittingTime = Math.Max(0, Node[v].earlytime
                        - TimeArriveAtV);
            double urgent = Node[v].lasttime - TimeArriveAtV;
            if (WaittingTime != 0)
            {
                TimeArriveAtV = Node[v].earlytime;
            }
            //check time fesibility
            if (TimeArriveAtV <= Node[v].lasttime)
            {
                double cost = alpha * dis[u, v] + beta *
                        WaittingTime+gamma * urgent;
                if (cost < MinCost)
                {
                    ExistV = true;
                    MinCost = cost;
                    BestV = v;
                    TimeAvirreAtBestV = TimeArriveAtV;
                }
            }// end if time
```

```
                }// end if cap

            }// end else

        }// end

        // update veh temp
        if (ExistV == true)
        {
            countIndexVehicle++;
            vehicleTemp[countIndexVehicle] = BestV;
            seqCus = seqCus.Except(new int[] { BestV }).ToArray();
            // set new u
            u = BestV;
            TimeArriveAtU = TimeAvirreAtBestV;
            if (Node[u].type == true)
            {
                TotalCapLine = TotalCapLine + Node[BestV].demand;
            }
            else
            {
                TotalCapBack = TotalCapBack + Node[BestV].demand;
            }
        }
        else //can't add any more
        {
            //update to real veh
            for (int i = 1; i < vehicleTemp.Length; i++)
            {
                vehicle[k, i] = vehicleTemp[i];
                vehicleTemp[i] = 0;
            }
            //start new vehicle
            k++;
            //set u
            countIndexVehicle = 1;
            u = seqCus[0];
            vehicleTemp[countIndexVehicle] = u;
            seqCus = seqCus.Except(new int[] { u }).ToArray();
            TimeArriveAtU = dis[0, u];
            if (TimeArriveAtU < Node[u].earlytime)
            {
                TimeArriveAtU = Node[u].earlytime;
            }
            if (Node[u].type == true)
            {
                TotalCapLine = Node[u].demand;
                TotalCapBack = 0;
            }
            else
            {
                TotalCapBack = Node[u].demand;
                TotalCapLine = 0;
            }
        }
    } // end while seq
    //add last veh temp to real veh
    if (seqCus.Length == 0)
```

```
            {
                //update to real veh
                for (int i = 1; i < vehicleTemp.Length; i++)
                {
                    vehicle[k, i] = vehicleTemp[i];
                    vehicleTemp[i] = 0;
                }
            }
            // build index
            int[] index = new int[NumCusForIndex];
            int count3 = 1;
            for (int i = 1; i <= k; i++)
            {
                int count1 = 1;
                while (vehicle[i, count1] != 0)
                {
                    index[count3] = vehicle[i, count1];
                    count1++;
                    count3++;
                }
                count3++;
            }
            // updated time arrive and capacity
            double[] capForindex = new double[index.Length];
            double[] timeArrive = new double[index.Length];
            double temp2 = 0;
            int count15 = 1;
            int count16 = 0;
            for (int m = 1; m <= numVehicle; m++)
            {
                temp2 = 0;
                while (index[count15] != 0)
                {
                    temp2 = temp2 + Node[index[count15]].demand;
                    if (index[count15 - 1] == 0)
                    {
                        if (Node[index[count15]].earlytime == 0)
                            //Setup arrive timeArrive for first customer
                        {
                            timeArrive[count15] = dis[index[count15 - 1],
                                    index[count15]];
                        }
                        else
                        {
                            timeArrive[count15] =
                                    Node[index[count15]].earlytime;
                        }
                    }
                    else
                    {
                        double wait = Math.Max(Node[index[count15]].earlytime -
(timeArrive[count15 - 1] + Node[index[count15 - 1]].servicetime +
dis[index[count15 - 1], index[count15]]), 0);
                        if (wait != 0.0)
                        {
                            timeArrive[count15] =
                                    Node[index[count15]].earlytime;
                        }
                        else
```

```
                        {
                            timeArrive[count15] = timeArrive[count15 - 1] +
Node[index[count15 - 1]].servicetime + dis[index[count15 - 1], index[count15]];
                        }
                    }
                    count15++;
                }
                if (index[count15] == 0)
                {
                    timeArrive[count15] = 0.0;
                }
                for (int n = count16; n < count15; n++)
                {
                    capForindex[n] = temp2;
                }
                count16 = count15;
                count15++; // skip depot
            }
            //backhaul
            for (int i = 1; i <= numVehicle; i++)
            {
                int count2 = 2;
                while (index[count2] != 0)
                {
                    int n1 = index[count2 - 1];
                    int n2 = index[count2];
                    if (Node[n1].type == false && Node[n2].type == true)
                    {
                        Console.WriteLine("\n -*-*-*-*-*-*Infeasible-
                                          Backhauls*-*-*-*-*-*- ");
                    }
                    count2++;
                }
                count2 = count2 + 2;
            }
             //INN end here
```


E. Code of C# program for NUC heuristic

```
//NUC start here
//int[] seqCus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25};
//int[] seqCus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50
};
int[] seqCus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
95, 96, 97, 98, 99, 100 };
            int[,] vehicle = new int[numVehicle + 1, NumCusForIndex];
            int NumCadidate = 3;
            double alpha = 0.4;
            double beta = 0.3;
            // set u
            int[] vehicleTemp = new int[NumCusForIndex];
```

```
int countIndexVehicle = 1;
int u = seqCus[0];
vehicleTemp[countIndexVehicle] = u;
seqCus = seqCus.Except(new int[] { u }).ToArray();
double TimeArriveAtU = dis[0, u];
if (TimeArriveAtU < Node[u].earlytime)
{
    TimeArriveAtU = Node[u].earlytime;
}
double TotalCapLine = 0;
double TotalCapBack = 0;
if (Node[u].type == true)
{
    TotalCapLine = Node[u].demand;
}
else
{
    TotalCapBack = Node[u].demand;
}
int k = 1; // num of vehicle use
while (seqCus.Length != 0)
{
    // build temp seq
    int[] seqCusTemp = new int[seqCus.Length];
    for (int i = 0; i < seqCus.Length; i++)
    {
        seqCusTemp[i] = seqCus[i];
    }

    // find the proper v for adding route after u
    int count1 = 0; // count for cadidate
    int BestV = 0; // v that is properly

    double TimeAvirreAtBestV = 0;
    double MinCost = 10000;
    bool ExistV = false;
    while (seqCusTemp.Length != 0 && count1 < NumCadidate)
    {
       if (Node[u].type == true)  // u-line
        {
            int v = seqCusTemp[0];
            seqCusTemp = seqCusTemp.Except(new int[] { v
                      }).ToArray();
            // check cap fesibility
            if (Node[v].type == true) // v-line
            {
                if (TotalCapLine + Node[v].demand <= cap)
                {
                    double TimeArriveAtV = TimeArriveAtU +
                      Node[u].servicetime + dis[u, v];
                    double WaittingTime = Math.Max(0,
                      Node[v].earlytime - TimeArriveAtV);
                    double urgent = Node[v].lasttime -
                      TimeArriveAtV;
                    if (WaittingTime != 0)
                    {
                        TimeArriveAtV = Node[v].earlytime;
                    }
                    //check time fesibility
```

```
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v] + beta *
                   WaittingTime+gramma*urgent;
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                        count1++; // count v be cadidate
                    }
                }// end if time

            }// end if cap
        }
        else // v-back
        {

            if (TotalCapBack + Node[v].demand <= cap)
            {
                double TimeArriveAtV = TimeArriveAtU +
                  Node[u].servicetime + dis[u, v];
                double WaittingTime = Math.Max(0,
                  Node[v].earlytime - TimeArriveAtV);
                double urgent = Node[v].lasttime -
                  TimeArriveAtV;
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v] + beta *
                         WaittingTime;
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                        count1++; // count v be cadidate
                    }
                }// end if time

            }// end if cap

        }
    }
    else  // u-back
    {
        int v = seqCusTemp[0]; ;
        while (Node[v].type == true)
        {
            seqCusTemp = seqCusTemp.Except(new int[] { v
            }).ToArray();
            if (seqCusTemp.Length == 0)
            {
```

```
                    v = 0;
                    break;
                }
                v = seqCusTemp[0];
            }
            if (v == 0)
            {
                break;
            }
            seqCusTemp = seqCusTemp.Except(new int[] { v
                }).ToArray();
            if (TotalCapBack + Node[v].demand <= cap)
            {
                double TimeArriveAtV = TimeArriveAtU +
                Node[u].servicetime + dis[u, v];
                double WaittingTime = Math.Max(0, Node[v].earlytime
                      - TimeArriveAtV);
                double urgent = Node[v].lasttime - TimeArriveAtV;
                if (WaittingTime != 0)
                {
                    TimeArriveAtV = Node[v].earlytime;
                }
                //check time fesibility
                if (TimeArriveAtV <= Node[v].lasttime)
                {
                    double cost = alpha * dis[u, v] + beta *
                                WaittingTime;
                    if (cost < MinCost)
                    {
                        ExistV = true;
                        MinCost = cost;
                        BestV = v;
                        TimeAvirreAtBestV = TimeArriveAtV;
                        count1++; // count v be cadidate
                    }
                }// end if time
            }// end if cap

        }// end else

    }// end cadidate

    // update veh temp
    if (ExistV == true)
    {
        countIndexVehicle++;
        vehicleTemp[countIndexVehicle] = BestV;
        seqCus = seqCus.Except(new int[] { BestV }).ToArray();
        // set new u
        u = BestV;
        TimeArriveAtU = TimeAvirreAtBestV;
        if (Node[u].type == true)
        {
            TotalCapLine = TotalCapLine + Node[BestV].demand;
        }
        else
        {
            TotalCapBack = TotalCapBack + Node[BestV].demand;
```

```
            }


        }
        else //can't add any more
        {
            //update to real veh
            for (int i = 1; i < vehicleTemp.Length; i++)
            {
                vehicle[k, i] = vehicleTemp[i];
                vehicleTemp[i] = 0;
            }
            //start new vehicle
            k++;
            //set u
            countIndexVehicle = 1;
            u = seqCus[0];
            vehicleTemp[countIndexVehicle] = u;
            seqCus = seqCus.Except(new int[] { u }).ToArray();
            TimeArriveAtU = dis[0, u];
            if (TimeArriveAtU < Node[u].earlytime)
            {
                TimeArriveAtU = Node[u].earlytime;
            }
            if (Node[u].type == true)
            {
                TotalCapLine = Node[u].demand;
                TotalCapBack = 0;
            }
            else
            {
                TotalCapBack = Node[u].demand;
                TotalCapLine = 0;
            }
        }

    } // end while seq

    //add last veh temp to real veh
    if (seqCus.Length == 0)
    {
        //update to real veh
        for (int i = 1; i < vehicleTemp.Length; i++)
        {
            vehicle[k, i] = vehicleTemp[i];
            vehicleTemp[i] = 0;
        }
    }


    // build index
    int[] index = new int[NumCusForIndex];
    int count3 = 1;
    for (int i = 1; i <= k; i++)
    {
        int count1 = 1;
        while (vehicle[i, count1] != 0)
        {
```

```
                    index[count3] = vehicle[i, count1];
                    count1++;
                    count3++;
                }
                count3++;
            }

            // updated time arrive and capacity
            double[] capForindex = new double[index.Length];
            double[] timeArrive = new double[index.Length];
            double temp2 = 0;
            int count15 = 1;
            int count16 = 0;
            for (int m = 1; m <= numVehicle; m++)
            {
                temp2 = 0;
                while (index[count15] != 0)
                {
                    temp2 = temp2 + Node[index[count15]].demand;
                    if (index[count15 - 1] == 0)
                    {
                        if (Node[index[count15]].earlytime == 0)
                     //Setup arrive timeArrive for first customer
                        {
                            timeArrive[count15] = dis[index[count15 - 1],
                            index[count15]];
                        }
                        else
                        {
                            timeArrive[count15] =
                                  Node[index[count15]].earlytime;
                        }
                    }
                    else
                    {
                        double wait = Math.Max(Node[index[count15]].earlytime -
(timeArrive[count15 - 1] + Node[index[count15 - 1]].servicetime +
dis[index[count15 - 1], index[count15]]), 0);
                        if (wait != 0.0)
                        {
                            timeArrive[count15] =
                                  Node[index[count15]].earlytime;
                        }
                        else
                        {
                            timeArrive[count15] = timeArrive[count15 - 1] +
Node[index[count15 - 1]].servicetime + dis[index[count15 - 1], index[count15]];
                        }
                    }
                    count15++;
                }
                if (index[count15] == 0)
                {
                    timeArrive[count15] = 0.0;
                }
                for (int n = count16; n < count15; n++)
                {
                    capForindex[n] = temp2;
                }
```

```
            count16 = count15;
            count15++; // skip depot
        }
        //backhaul
        for (int i = 1; i <= numVehicle; i++)
        {
            int count2 = 2;
            while (index[count2] != 0)
            {
                int n1 = index[count2 - 1];
                int n2 = index[count2];
                if (Node[n1].type == false && Node[n2].type == true)
                {
                    Console.WriteLine("\n -*-*-*-*-*-*Infeasible-
                        Backhauls*-*-*-*-*-*- ");
                }
                count2++;
            }
            count2 = count2 + 2;
        }
         //NUC end here
```

F. Code of C# program for NNRW heuristic

```
//NNRW start here
//int[] seqCus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25};
//int[] seqCus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50
};
int[] seqCus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
95, 96, 97, 98, 99, 100 };
double alpha = 0.4;
double beta = 0.3;
double gamma = 0.3;
int[] index = new int[NumCusForIndex];
            int CountRunIndex = 0;
            double timeArriveV = 0;
            double totalCap = 0;
            while (Cus.Length != 0)
            {
                int u = index[CountRunIndex];
                double timeArriveU = timeArriveV;
                int[] FeasibleCustomer = new int[Cus.Length];
                double[] UrgentTimeFeasibleCus = new double[Cus.Length];
                double[] WaitingTimeFeasibleCus = new double[Cus.Length];
                int count = 0;

                for (int i = 0; i < Cus.Length; i++)
                {
                    int v = Cus[i];
                    bool feasible = true;
                    double waitingTime = 0;
```

```
            // -*-*-check feasibility*-*-*
            //---backhauls fesibility---
            if (Node[u].type == false)
            {
                if (Node[v].type == true)
                {
                    feasible = false;
                }
            }
            //---time window fesibility---
            if (feasible == true)
            {
                timeArriveV = timeArriveU + Node[u].servicetime +
                    dis[u, v];
                if (timeArriveV < Node[v].earlytime)
                {
                    waitingTime = Node[v].earlytime - timeArriveV;
                    timeArriveV = Node[v].earlytime;
                }
                if (timeArriveV > Node[v].lasttime)
                {
                    feasible = false;
                }
            }
            //---capacity---
            if (feasible == true)
            {
                if (totalCap + Node[v].demand > cap)
                {
                    feasible = false;
                }
            }
            //*-*-We can use this v*-*-*-Add to Array
            if (feasible == true)
            {
                FeasibleCustomer[count] = v;
                UrgentTimeFeasibleCus[count] = Node[v].lasttime -
                    timeArriveV;
                WaitingTimeFeasibleCus[count] = waitingTime;
                count++;
            }

        }// end for to fine feasivle cus

        // construction of wheel
        count = 0;
        FeasibleCustomer = FeasibleCustomer.Except(new int[] { 0
                           }).ToArray();
        if (FeasibleCustomer.Length != 0)
        {
            Random random = new Random(seedRand);
            double[] prob = new double[FeasibleCustomer.Length];
            double[] cumuprob = new double[FeasibleCustomer.Length];
            double[] cost = new double[FeasibleCustomer.Length]; ;
            double totalDist = 0;
            for (int i = 0; i < FeasibleCustomer.Length; i++)
            {
                cost[i] = alpha * dis[u, FeasibleCustomer[i]] + beta *
WaitingTimeFeasibleCus[i] + gramma * UrgentTimeFeasibleCus[i];
```

```
                totalDist = totalDist + (1 / (cost[i]));
            }
            // compute prob
            for (int i = 0; i < FeasibleCustomer.Length; i++)
            {
                prob[i] = (1 / cost[i]) / totalDist;
            }
            // compute q = cumulative prob
            cumuprob[0] = prob[0];
            for (int i = 1; i < FeasibleCustomer.Length; i++)
            {
                cumuprob[i] = cumuprob[i - 1] + prob[i];
            }
            double r1 = random.NextDouble();
            count = 0;
            bool found = true;
            while (found == true)
            {
                if (r1 <= cumuprob[count])
                {
                    break;
                }
                count++;
            }
            // run index
            CountRunIndex++;
            //input new cus to index
            index[CountRunIndex] = FeasibleCustomer[count];
            // del assinged cus from Cus
            Cus = Cus.Except(new int[] { index[CountRunIndex]
             }).ToArray();
            // update time and cap
            timeArriveV = timeArriveU + Node[u].servicetime + dis[u,
             index[CountRunIndex]];
            if (timeArriveV < Node[index[CountRunIndex]].earlytime)
            {
                timeArriveV = Node[index[CountRunIndex]].earlytime;
            }
            totalCap = totalCap + Node[index[CountRunIndex]].demand;
        }
        else // means can't add anymore -> new veh
        {
            CountRunIndex++;
            // set initial
            index[CountRunIndex] = 0;
            timeArriveV = 0;
            totalCap = 0;

        }


    } // end while
      //NNRW end here
```

Code of C# program for CS heuristic

```
double alpha = 0.4;
double beta = 0.3;
double gramma = 0.3;
int NumOfSolution = 15;
```

```
int[] index = new int[NumCusForIndex];
int[,] Sol = new int[NumOfSolution, NumCusForIndex];
double[] objFeasible = new double[NumOfSolution];
//int[] Cus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25 };
//int[] Cus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50};
int[] Cus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99, 100 };
//wheel for initial Sol.
            for (int j = 0; j < NumOfSolution; j++)
            {
                    index = ContByRouletteWheel(alpha, beta, gramma, Cus,
                     NumCusForIndex, cap, seedRand, numVehicle, Node, dis);
                    double obj = 0;
                    for (int l = 0; l < index.Length; l++)
                    {
                        if (l != index.Length - 1)
                        {
                            obj = obj + dis[index[l], index[l + 1]];
                        }
                    }
                     //objFeasible[j] = obj;
                    Console.Write("   " + obj);
                    for (int i = 0; i < index.Length; i++)
                    {
                        Sol[j, i] = index[i];
                    }
                    seedRand++;
            }// end for numofSol

            //Cuckoo Search*************************************************
            double BestKnownOld = 100001;
            double BestKnownNew = 100000;
            int[] NotImprove = new int[NumOfSolution];
            int[] BestSolOld = new int[NumCusForIndex];
            int[] BestSolNew = new int[NumCusForIndex];
            while (BestKnownNew < BestKnownOld)
            {
                //check best known solution
                int MinIndex = -1;
                double MinValue = 100000;
                for (int i = 0; i < NumOfSolution; i++)
                {
                    double obj = 0; // compute all fitness
                    for (int l = 0; l < NumCusForIndex; l++)
                    {
                        if (l != NumCusForIndex - 1)
                        {
                            obj = obj + dis[Sol[i, l], Sol[i, l + 1]];
                        }
                    }
                    objFeasible[i] = obj;
                    if (obj < MinValue)
```

```
                {
                    MinValue = obj;
                    MinIndex = i;
                }
            }
            // updated best sol
            if (MinValue < BestKnownNew)
            {
                BestKnownOld = BestKnownNew;
                BestKnownNew = MinValue;
                for (int j = 0; j < NumCusForIndex; j++)
                {
                    BestSolOld[j] = BestSolNew[j];
                    BestSolNew[j] = Sol[MinIndex, j];
                }
            }

            // randomly lay cuckoo egg
            int[] nest = new int[15] { 0,1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14 };
            Random rnd = new Random(seedRand);

            for (int i = 0; i < 15; i++) //num cuckoo=15
            {
                int RandChoosePosition = rnd.Next(0, nest.Length);
                int NestRChoosen = nest[RandChoosePosition];


                int[] index3 = new int[NumCusForIndex];
                for (int j = 0; j < NumCusForIndex; j++)
                {
                    index3[j] = Sol[NestRChoosen, j];
                }

                index3=CuckooEgg(index3, cap, numVehicle,Node, dis);
                 //build cuckoo egg

                for (int j = 0; j < NumCusForIndex; j++)
                 // replace cukoo egg to ole egg
                {
                    Sol[i, j] = index3[j];
                }

            }

            // abandon worst nest Pc=0.25 of 15 is 3 nest
                // find all obj
                double[] TeamObj = new double[NumOfSolution];
                for (int j = 0; j < NumOfSolution; j++)
                {
                     for (int k = 0; k < NumCusForIndex; k++)
                        {
                            if (k != NumCusForIndex - 1)
                                {
                                    TeamObj[j] = TeamObj[j] + dis[Sol[j,
                                     k], Sol[j, k + 1]];
                                }
                        }
                }
```

```
                    //find max obj/ worst sol
                    for (int j = 0; j < 3; j++)
                    {
                        // find best
                        double m = TeamObj.Max();
                        int maxIndex = Array.IndexOf(TeamObj, m);
                        TeamObj[maxIndex] = -1;
                        //bulid new nest
                        int[] indexTemp = new int[NumCusForIndex];
                        indexTemp = ContByRouletteWheel(alpha, beta, gramma,
Cus, NumCusForIndex, cap, seedRand, numVehicle, Node, dis);
                        seedRand++;
                        for (int i = 0; i < indexTemp.Length; i++)
                        {
                            Sol[maxIndex, i] = indexTemp[i];
                        }
                    }

            }//end while
```

G. Code of C# program for EABC heuristic

```
int NumOnlookerBee = 25;
int NumLimit = 20;
int seedRand = 0;
double alpha = 0.4;
double beta = 0.3;
double gramma = 0.3;
int NumOfSolution = 25;
int[] index = new int[NumCusForIndex];
int[,] Sol = new int[NumOfSolution,NumCusForIndex];
double[] objFeasible=new double[NumOfSolution];
//int[] Cus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25 };
//int[] Cus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50};
int[] Cus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99, 100 };

for (int j = 0; j < NumOfSolution; j++)
            {
                bool CheckDuplicated = true;
                while (CheckDuplicated == true)
                {
                    index = ContByRouletteWheel(alpha, beta, gramma, Cus,
                     NumCusForIndex, cap, seedRand, numVehicle, Node, dis);
                    double obj = 0;
                    for (int l = 0; l < index.Length; l++)
                    {
                        if (l != index.Length - 1)
                        {
```

```
                obj = obj + dis[index[l], index[l + 1]];
            }
        }

        // check duplication
        bool ExistRepeat = false;
        for (int i = 0; i < objFeasible.Length;i++ )
        {
            if (obj == objFeasible[i])
            {
                seedRand++; // at least one duplication
                ExistRepeat = true;
                break;
               // break for to repeat while loop check duplication
            }
        }
        if (ExistRepeat == false)// if no-duplication
        {
            CheckDuplicated = false;
        }
        if (CheckDuplicated == false)
         // collect sol if not duplicated
        {
            //objFeasible[j] = obj;
            Console.Write("  " + obj);
            for (int i = 0; i < index.Length; i++)
            {
                Sol[j, i] = index[i];
            }
            seedRand++;
            break;
        }
    }// end while check duplicated
}// end for numofSol

//EABC************************************************
double BestKnownOld = 100001;
double BestKnownNew = 100000;
int[] NotImprove = new int[NumOfSolution];
int[] BestSolOld = new int[NumCusForIndex];
int[] BestSolNew = new int[NumCusForIndex];
while (BestKnownNew < BestKnownOld)
{
    //check best known solution
    int MinIndex = -1;
    double MinValue = 100000;
    for (int i = 0; i < NumOfSolution; i++)
    {
        double obj = 0; // compute all fitness
        for (int l = 0; l < NumCusForIndex; l++)
        {
            if (l != NumCusForIndex - 1)
            {
                obj = obj + dis[Sol[i, l], Sol[i, l + 1]];
            }
        }
        objFeasible[i] = obj;
        if (obj < MinValue)
        {
```

```
            MinValue = obj;
            MinIndex = i;
        }
    }
    // updated best sol
    if (MinValue < BestKnownNew)
    {
        BestKnownOld = BestKnownNew;
        BestKnownNew = MinValue;
        for (int j = 0; j < NumCusForIndex; j++)
        {
            BestSolOld[j] = BestSolNew[j];
            BestSolNew[j] = Sol[MinIndex, j];
        }
    }
    // woker bee improve food source
    int[] index3 = new int[NumCusForIndex];
    for (int i = 0; i < NumOfSolution; i++)
    {
        for (int j = 0; j < NumCusForIndex; j++)
        {
            index3[j] = Sol[i, j];
        }
        double objBegin = 0; // compute all fitness
        for (int l = 0; l < NumCusForIndex; l++)
        {
            if (l != NumCusForIndex - 1)
            {
                objBegin = objBegin + dis[Sol[i, l], Sol[i, l +
                                1]];
            }
        }
        Apply neighborhood search here
        for (int j = 0; j < NumCusForIndex; j++)
        {
            Sol[i, j] = index3[j];
        }
        double objEnd = 0; // compute all fitness
        for (int l = 0; l < NumCusForIndex; l++)
        {
            if (l != NumCusForIndex - 1)
            {
                objEnd = objEnd + dis[Sol[i, l], Sol[i, l + 1]];
            }
        }
        if (objBegin == objEnd) // not improve?
        {
            NotImprove[i] = NotImprove[i] + 1;
        }
        int[] index4 = new int[NumCusForIndex];
        if (NotImprove[i] >= NumLimit)// reach limit
        {
            //build new
            seedRand++;
            Console.Write("new-");
            index4 = ContByRouletteWheel(alpha, beta, gramma, Cus,
         NumCusForIndex, cap, seedRand, numVehicle, Node, dis);
            for (int j = 0; j < NumCusForIndex; j++)
            {
```

```
                Sol[i, j] = index4[j];
            }
            NotImprove[i] = 0;
        }

    }


    // roulet wheel by onlooker bees
    int[] index2 = new int[NumCusForIndex];
    double[] probSol = new double[NumOfSolution];
    double[] cumuprobSol = new double[NumOfSolution];
    double TotalFitness = 0;
    for (int i = 0; i < NumOfSolution; i++)
    {
        TotalFitness = TotalFitness + (1 / (objFeasible[i]));
    }
    for (int i = 0; i < NumOfSolution; i++)
    {
        probSol[i] = (1/objFeasible[i])/TotalFitness;
    }
    cumuprobSol[0] = probSol[0];
    for (int i = 1; i < NumOfSolution; i++)
    {
        cumuprobSol[i] = cumuprobSol[i - 1] + probSol[i];
    }
    Random randomSol = new Random(1);
    //try to find repeat sol.******************test
    Console.WriteLine("round " + countIteration);
    for (int i = 0; i < NumOnlookerBee; i++)
    {
        double r1 = randomSol.NextDouble();
        int count = 0;
        bool found = true;
        // onlooker choose food source
        while (found == true)
        {
            if (r1 <= cumuprobSol[count])
            {
                break;
            }
            count++;
        }
        Console.Write(" "+count);  // **********************test
        for (int j = 0; j < NumCusForIndex; j++)
        {
            index2[j] = Sol[count, j];
        }
        //improve food source
        Apply neighborhood search here
        for (int j = 0; j < NumCusForIndex; j++)
        {
            Sol[count, j] = index2[j];
        }
    } //end for select and improv

}//end
```

H. Code of C# program for EGB heuristic

```
int NumLimit = 5;
int seedRand = 0;
double alpha = 0.4;
double beta = 0.3;
double gramma = 0.3;
int NumOfSolution = 48;
int[] index = new int[NumCusForIndex];
int[,] Sol = new int[NumOfSolution,NumCusForIndex];
double[] objFeasible=new double[NumOfSolution];
//int[] Cus = new int[25] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25 };
//int[] Cus = new int[50] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50};
int[] Cus = new int[100] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99, 100 };
for (int j = 0; j < NumOfSolution; j++)
            {
                bool CheckDuplicated = true;
                while (CheckDuplicated == true)
                {
                    index = ContByRouletteWheel(alpha, beta, gramma, Cus,
                     NumCusForIndex, cap, seedRand, numVehicle, Node, dis);
                    double obj = 0;
                    for (int l = 0; l < index.Length; l++)
                    {
                        if (l != index.Length - 1)
                        {
                            obj = obj + dis[index[l], index[l + 1]];
                        }
                    }

                    // check duplication
                    bool ExistRepeat = false;
                    for (int i = 0; i < objFeasible.Length;i++ )
                    {
                        if (obj == objFeasible[i])
                        {
                            seedRand++; // at least one duplication
                            ExistRepeat = true;
                            break;
                           // break for to repeat while loop check duplication
                        }
                    }
                    if (ExistRepeat == false)// if no-duplication
                    {
                        CheckDuplicated = false;
                    }
                    if (CheckDuplicated == false)
                     // collect sol if not duplicated
                    {
                        //objFeasible[j] = obj;
                        Console.Write("  " + obj);
```

```
            for (int i = 0; i < index.Length; i++)
            {
                Sol[j, i] = index[i];
            }
            seedRand++;
            break;
        }
    }// end while check duplicated
}// end for numofSol

  //*************Divided Team***********************************
int numberMembers=8;
int numberTeams = 6;
int count4Sol = 0;
int[,,] TEAM = new int[numberTeams,numberMembers, NumCusForIndex];
for (int i = 0; i < numberTeams; i++)
{
    for (int j = 0; j < numberMembers; j++)
    {
        for (int k = 0; k < NumCusForIndex;k++ )
        {
            TEAM[i, j, k] = Sol[count4Sol, k];
        }
        count4Sol++;
    }
}
//*************End-Divided Team*****************************

double previousBest = 1000000; // previous Best
double presentBest = 999999;  // presentBest
int NumSeason = 1;
while (presentBest<previousBest )
{
    Console.WriteLine("\n\n =||=||=|| Season " + NumSeason + "
                                ||=||=||=||=");
    previousBest = presentBest;
    double[] Score = new double[numberTeams]; //test
    //a season divided 2 part
    for (int m = 0; m < 2; m++)
    {
        //*************-Training-**********************************
        for (int i = 0; i < numberTeams; i++)
        {
            for (int j = 0; j < numberMembers; j++)
            {
                int[] indexTemp = new int[NumCusForIndex];
                // copy to index Temp
                for (int k = 0; k < NumCusForIndex; k++)
                {
                    indexTemp[k] = TEAM[i, j, k];
                }
                // improve sol
                indexTemp = Training(indexTemp, cap, numVehicle,
                             Node, dis);
                // updated
                for (int k = 0; k < NumCusForIndex; k++)
                {
                    TEAM[i, j, k] = indexTemp[k];
                }
```

```
    }
}
//*************End-Training-*************************

//*************Re-Arrange sol and choose captain*******
//Simplify Sol
for (int i = 0; i < numberTeams; i++)
{
    for (int j = 0; j < numberMembers; j++)
    {
        int[] indexTemp = new int[NumCusForIndex];
        for (int k = 0; k < NumCusForIndex; k++)
        {
            indexTemp[k] = TEAM[i, j, k];
        }
        indexTemp = SimplifyForm(indexTemp, numVehicle);
        // updated
        for (int k = 0; k < NumCusForIndex; k++)
        {
            TEAM[i, j, k] = indexTemp[k];
        }
    }
}
// Arrange Sol in team (Best obj is captain *index=0*)
TEAM = ArrangePlayers(TEAM, numberTeams, numberMembers,
       NumCusForIndex, dis);

//print test Arrange player in each team
Console.WriteLine("\n\n ===> After Training :=");
for (int i = 0; i < numberTeams; i++)
{
    Console.WriteLine("\n TEAM :" + i);
    for (int j = 0; j < numberMembers; j++)
    {
        double TeamObj = 0;
        for (int k = 0; k < NumCusForIndex; k++)
        {
            if (k != NumCusForIndex - 1)
            {
                TeamObj = TeamObj + dis[TEAM[i, j, k],
                      TEAM[i, j, k + 1]];
            }
        }
        Console.Write(" " + TeamObj);
    }
}

//*************End--Re-Arrange sol and choose captain***

//************-Custom Trianing-***********************
int CountForVehicle = 0;
Random random = new Random(seedRand);
bool check = false;
int[] IndexCap = new int[NumCusForIndex];
int[] IndexPlay = new int[NumCusForIndex];
int[] NewIndex = new int[NumCusForIndex];
for (int l = 0; l < numberTeams; l++)
{
    for (int j = 1; j < numberMembers; j++)
```

```
                        {
                            int countLimit = 0;
                            while (countLimit < NumLimit)
                            {
                                for (int i = 0; i < index.Length; i++)
                                {
                                    IndexCap[i] = TEAM[l, 0, i];
                                    IndexPlay[i] = TEAM[l, j, i];
                                }
                                // fine the number of vehicles
                                double NumOfCapVehicle =
                                   CountNumOfVehicles(IndexCap);
//random to select the number of vehicles that will be duplicated from captain
                                int Min =
       Convert.ToInt32(Math.Floor(NumOfCapVehicle * 40 / 100));
                                int Max =
       Convert.ToInt32(Math.Floor(NumOfCapVehicle * 80 / 100));

                                CountForVehicle = random.Next(Min, Max + 1);
                                //plus 1 because it is a form
                                //Train by captain
                                NewIndex = CustomTraining(IndexCap, IndexPlay,
cap, CountForVehicle, numVehicle, Node, dis);
                                check = NewBetterThanOld(NewIndex, IndexPlay,
dis);
                                if (check == true)
                                {
                                    for (int i = 0; i < index.Length; i++)
                                    {
                                        TEAM[l, j, i] = NewIndex[i];
                                    }
                                    break;
                                }
                                else
                                {
                                    countLimit++;
                                }
                            }//end while
                        }//end for j
                    }//end for l

                // Arrange Sol again*** in team (Best obj is captain *index=0*)
                    TEAM = ArrangePlayers(TEAM, numberTeams, numberMembers,
                           NumCusForIndex, dis);

                    //print test
                    Console.WriteLine("\n\n ===> After Custom train := ");
                    for (int i = 0; i < numberTeams; i++)
                    {
                        Console.WriteLine("\n TEAM :" + i);
                        for (int j = 0; j < numberMembers; j++)
                        {
                            double TeamObj = 0;
                            for (int k = 0; k < NumCusForIndex; k++)
                            {
                                if (k != NumCusForIndex - 1)
                                {
                                    TeamObj = TeamObj + dis[TEAM[i, j, k],
                                          TEAM[i, j, k + 1]];
```

```
            }
        }
        Console.Write(" " + TeamObj);
    }
}

//************-End-Custom****************************

//************-Start Match-************************
//compute obj
double[,] ObjAll = new double[numberTeams, numberMembers];
for (int i = 0; i < numberTeams; i++)
{
    Console.Write(" \n ");
    for (int j = 0; j < numberMembers; j++)
    {
        double TempObj = 0;
        for (int l = 0; l < NumCusForIndex; l++)
        {
            if (l != NumCusForIndex - 1)
            {
                TempObj = TempObj + dis[TEAM[i, j, l],
                                        TEAM[i, j, l + 1]];
            }
        }
        ObjAll[i, j] = TempObj;
        //Console.Write(" " + TempObj);
    }
}
// competition
for (int i = 0; i < numberTeams; i++) // Team A
{
    for (int ii = i + 1; ii < numberTeams; ii++) //Team B
    {
        int ScoreA = 0;
        int ScoreB = 0;
        for (int k = 0; k < numberMembers; k++) //Member
        {
            if (ObjAll[i, k] < ObjAll[ii, k])
            {
                ScoreA++;
            }
            else if (ObjAll[i, k] > ObjAll[ii, k])
            {
                ScoreB++;
            }
            else
            {
                ScoreA++;
                ScoreB++;
            }
        }
        if (ScoreA > ScoreB)
        {
            Score[i] = Score[i] + 3;
        }
        else if (ScoreA < ScoreB)
        {
            Score[ii] = Score[ii] + 3;
```

```
                        }
                        else
                        {
                            Score[i]++;
                            Score[ii]++;
                        }
                    }
                }

                //************-End Match-**************************

                //************-Exchange-**************************
                // find best score
                int[] OrderedTeamByScore = new int[numberTeams];
                for (int i = 0; i < numberTeams; i++)
                {
                    double ScoreMax = Score.Max();
                    int MaxScoreIndex = Array.IndexOf(Score, ScoreMax);
                    OrderedTeamByScore[i] = MaxScoreIndex;
                    Score[MaxScoreIndex] = -1; // never choosing again
                }
                //exchange
                for (int i = 0; i < numberTeams / 2; i++)
                {
                    int Temp = 0;
                    for (int j = 0; j < NumCusForIndex; j++)
                    {
                        Temp = TEAM[OrderedTeamByScore[i], numberMembers -
1 - i, j];
                        TEAM[OrderedTeamByScore[i], numberMembers - 1 - i,
j] = TEAM[OrderedTeamByScore[numberTeams - 1 - i], i, j];
                        TEAM[OrderedTeamByScore[numberTeams - 1 - i], i, j]
= Temp;
                    }
                }
                //************-End Exchange-**********************

// Arrange Sol again after exchange players*** in team (Best obj is captain
*index=0*)
                TEAM = ArrangePlayers(TEAM, numberTeams, numberMembers,
                        NumCusForIndex, dis);

                //*************************-END-*****************
            NumSeason++;
        }// end while

//*************************************************************************
public static int[] SimplifyForm(int[] index, int numVehicle)
        {
            int[,] Vehicle = new int[numVehicle, index.Length];
            int count1 = 1;
            for (int k = 0; k < numVehicle; k++) // seperated vehicles
            {
                int count2 = 1;
                while (index[count1] != 0)
                {
                    Vehicle[k, count2] = index[count1];
                    count2++;
                    count1++;
```

```
                }
                count1++; // skip depot
            }
            // re-arrange
            int count3=1;
            int[] index2 = new int[index.Length];
            for (int i = 0; i < numVehicle; i++)
            {
                if (Vehicle[i, 1] != 0)
                {
                    int count4=1;
                    while (Vehicle[i, count4] != 0)
                    {
                        index2[count3] = Vehicle[i, count4];
                        count4++;
                        count3++;
                    }
                    index2[count3] = 0;
                    count3++;
                }
            }
                return index2;
        }

//*************************************************************************
public static int[, ,] ArrangePlayers(int[, ,] TEAM, int numberTeams, int
numberMembers, int NumCusForIndex, double[,] dis)
        {
            for (int i = 0; i < numberTeams; i++)
            {
                double[] TeamObj = new double[numberMembers];
                for (int j = 0; j < numberMembers; j++)
                {
                    for (int k = 0; k < NumCusForIndex; k++)
                    {
                        if (k != NumCusForIndex - 1)
                        {
                            TeamObj[j] = TeamObj[j] + dis[TEAM[i, j, k],
                                TEAM[i, j, k + 1]];
                        }
                    }
                }
                int[,] TempTeam = new int[numberMembers, NumCusForIndex];
                for (int j = 0; j < numberMembers; j++)
                {
                    // find best player
                    double m = TeamObj.Min();
                    int minIndex = Array.IndexOf(TeamObj, m);
                    TeamObj[minIndex] = 10000000;
                    //record new ordered
                    for (int k = 0; k < NumCusForIndex; k++)
                    {
                        TempTeam[j, k] = TEAM[i, minIndex, k];
                    }
                }
                //updated
                for (int j = 0; j < numberMembers; j++)
                {
                    for (int k = 0; k < NumCusForIndex; k++)
```

```
                    {
                        TEAM[i, j, k] = TempTeam[j, k];
                    }
                }
            }

            return TEAM;
        }

//*************************************************************************
public static int[] CustomTraining(int[] IndexCap, int[] IndexPlay, int cap,
int CountForVehicle, int numVehicle, Data[] Node, double[,] dis)
        {
            //copy some part of captain
            int[] NewIndex = new int[IndexCap.Length];
            int[] DelIndex = new int[IndexCap.Length];
            int count4Run = 1, count4Index = 1;

            while (CountForVehicle != 0)
            {
                while (IndexCap[count4Run] != 0)
                {
                    NewIndex[count4Index] = IndexCap[count4Run];
                    DelIndex[count4Index] = IndexCap[count4Run];
                    count4Run++;
                    count4Index++;
                }
                NewIndex[count4Index] = 0;
                count4Index++;
                count4Run++;
                CountForVehicle--;
            }
            //delete depot to get deleting seq
            DelIndex = DelIndex.Except(new int[] { 0 }).ToArray();
            //copy player to tempPlayer
            int[] TempPlayer = new int[IndexCap.Length];
            for (int i = 0; i < IndexCap.Length; i++)
            {
                TempPlayer[i] = IndexPlay[i];
            }
            //del duplicate customer from tempPlayer
            TempPlayer = TempPlayer.Except(new int[] { 0 }).ToArray();
            for (int i = 0; i < DelIndex.Length; i++)
            {
                TempPlayer = TempPlayer.Except(new int[] { DelIndex[i]
                                    }).ToArray();
            }
            // devide the rest of customer --> 2vars
            int countLine = 0, countBack = 0;
            int[] LineTempPlayer = new int[TempPlayer.Length];
            int[] BackTempPlayer = new int[TempPlayer.Length];
            for (int i = 0; i < TempPlayer.Length; i++)
            {
                if (Node[TempPlayer[i]].type == true)
                {
                    LineTempPlayer[countLine] = TempPlayer[i];
                    countLine++;
                }
                else
```

```
                {
                    BackTempPlayer[countBack] = TempPlayer[i];
                    countBack++;
                }
            }
            //Build fesible vehicle
            LineTempPlayer = LineTempPlayer.Except(new int[] { 0 }).ToArray();
            BackTempPlayer = BackTempPlayer.Except(new int[] { 0 }).ToArray();
            while (LineTempPlayer.Length != 0 || BackTempPlayer.Length != 0)
            {
                int[] TempRoute = new int[TempPlayer.Length];
                double timeLeave = 0;
                double capacityCount = 0;
                int count4TempRoute = 1;
                if (LineTempPlayer.Length != 0)
                {
                    TempRoute[0] = LineTempPlayer[0];
                    capacityCount = Node[TempRoute[0]].demand;
                    LineTempPlayer = LineTempPlayer.Except(new int[] {
                                LineTempPlayer[0] }).ToArray();
                    timeLeave = Math.Max(dis[0, TempRoute[0]],
        Node[TempRoute[0]].earlytime) + Node[TempRoute[0]].servicetime;
                }
                else
                {
                    TempRoute[0] = BackTempPlayer[0];
                    capacityCount = Node[TempRoute[0]].demand;
                    BackTempPlayer = BackTempPlayer.Except(new int[] {
                                        BackTempPlayer[0] }).ToArray();
                    timeLeave = Math.Max(dis[0, TempRoute[0]],
        Node[TempRoute[0]].earlytime) + Node[TempRoute[0]].servicetime;
                }
                //linehual
                for (int i = 0; i < LineTempPlayer.Length; i++)
                {
                    if (capacityCount + Node[LineTempPlayer[i]].demand <= cap)
                    {
                        if (timeLeave + dis[TempRoute[count4TempRoute - 1],
                            LineTempPlayer[i]] <=
Node[LineTempPlayer[i]].lasttime)
                        {
                            TempRoute[count4TempRoute] = LineTempPlayer[i];
                            capacityCount = capacityCount +
                                   Node[LineTempPlayer[i]].demand;
                            timeLeave = Math.Max(timeLeave +
                         dis[TempRoute[count4TempRoute - 1], LineTempPlayer[i]],
Node[LineTempPlayer[i]].earlytime)+ Node[LineTempPlayer[i]].servicetime;
                            LineTempPlayer = LineTempPlayer.Except(new int[] {
LineTempPlayer[i] }).ToArray();
                            count4TempRoute++;
                        }
                        else
                        {
                            break;
                        }
                    }
                    else
                    {
                        break;
```

```
                    }
                }
                // backhual
                for (int i = 0; i < BackTempPlayer.Length; i++)
                {
                    if (capacityCount + Node[BackTempPlayer[i]].demand <= cap)
                    {
                        if (timeLeave + dis[TempRoute[count4TempRoute - 1],
BackTempPlayer[i]] <= Node[BackTempPlayer[i]].lasttime)
                        {
                            TempRoute[count4TempRoute] = BackTempPlayer[i];
                            capacityCount = capacityCount +
Node[BackTempPlayer[i]].demand;
                            timeLeave = Math.Max(timeLeave +
dis[TempRoute[count4TempRoute - 1], BackTempPlayer[i]],
Node[BackTempPlayer[i]].earlytime) + Node[BackTempPlayer[i]].servicetime;
                            BackTempPlayer = BackTempPlayer.Except(new int[] {
BackTempPlayer[i] }).ToArray();
                            count4TempRoute++;
                        }
                        else
                        {
                            break;
                        }
                    }
                    else
                    {
                        break;
                    }
                }
                //Add new route to NewIndex
                int countTemp = 0;
                while (countTemp<TempRoute.Length && TempRoute[countTemp] != 0)
                {
                    NewIndex[count4Index] = TempRoute[countTemp];
                    countTemp++;
                    count4Index++;
                }
                NewIndex[count4Index] = 0;
                count4Index++;
            } //end while

            //Improve solution before return
            NewIndex = Training(NewIndex, cap, numVehicle, Node, dis);
            //Simplify Sol
            NewIndex = SimplifyForm(NewIndex, numVehicle);
            return NewIndex;
        }
```

VITA

Mr. Tanawat Worawattawechai was born in June 28, 1987, in Uttaradit. He received a bachelor degree in Mathematics from Department of Mathematics, Faculty of science, ChiangMai University, Thailand 2009, and a master degree in Applied Mathematics and Computational Science from Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University, Thailand 2012. He has been financially supported by the Development and Promotion of Science and Technology talents project (DPST).

PUBLICATION

Worawattawechai, T., B. Intiyot, and C. Jeenanunta, *Heuristic Approach to Vehicle Routing Problem with Backhauls and Time Windows : International Conference on Applied Statistics 2016, Phuket, Thailand, July 13 – 15, 2016. Proceedings*, W. Panichkitkosolkul and P. Srisuradetchai, Editors. Thammasat University, Pathum Thani. p. 121-128.

Worawattawechai, T., B. Intiyot, and C. Jeenanunta, *Cuckoo search algorithm for the vehicle routing problem with backhauls and time windows*. Panyapiwat Journal, 2016. **8**: p.136-149.

Worawattawechai, T., B. Intiyot, and C. Jeenanunta, *An artificial bee colony algorithm for the vehicle routing problem with backhauls and time windows*. Songklanakarin Journal of Science and Technology, [in press].