University of Bergen
Department of Informatics

A new parallel adaptive heuristic for combinatorial optimization problems

Author: Håvar Andre Melheim Salbu

Supervisor: Ahmad Hemmati

Co-Supervisor: Ramin Hasibi

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

February 2023

Abstract

In recent years single process computing performance has improved very little, and we are seeing an increasing shift in focus onto parallel computing. Therefore, to achieve more computational efficiency moving forward, additional research is needed in parallel computing. Within combinatorial optimization, parallel solution methods is a promising research area we believe deserves more attention in the literature. We draw inspiration from existing work that has addressed the implementation of a parallel adaptive large neighbourhood search (ALNS) algorithm on the GPU for solving the distance-constrained capacitated vehicle routing problem (DCVRP). In this paper we propose a parallel adaptive heuristic for pickup and delivery problems, but the algorithm is generalizable to a wide range of combinatorial optimization problems.

Acknowledgement

I would like to thank my supervisor, Ahmad Hemmati, and co-supervisor Ramin Hasibi for their dedication, excellent support, guidance, and literature recommendations. Furthermore, I would like to thank my family and friends for their encouragement and support throughout this challenging journey.

Håvar A. Melheim Salbu

17. February 2023

Table of contents

List of abbreviations and acronyms

AGES	-	Adaptive guided ejection search
ALNS	-	Adaptive large neighbourhood search
COPs	-	Combinatorial optimization problems
CPU	-	Central processing unit
CUDA	-	Compute unified device architecture
DCVRP	-	Distance-constrained capacitated vehicle routing problem
EMA	-	Exponential moving average
GA	-	Genetic algorithm
GES	-	Guided ejection search
GPU	-	Graphical processing unit
LAHC	-	Late acceptance hill climbing
LNS	-	Large neighbourhood search
LS	-	Local search
NP	-	Nondeterministic polynomial time
OpenMP	-	Open Multi-Processing
PALNS	-	Perceptive adaptive large neighbourhood search
PDPTW	-	Pickup and delivery problem with time windows
PSO	-	Particle swarm optimization
SA	-	Simulated annealing
TSPPD	-	Traveling salesperson problem with pickup and deliveries
URS	-	Uniform random selection
VRPs	-	Vehicle routing problems

List of figures

List of tables

List of formulas and algorithms

Chapter 1

Introduction

Here we give introductory remarks, stating our hypothesis and providing motivation for the selected topic. We then introduce the problem that we have focused on, namely the pickup and delivery problem with time windows and perform simple complexity analysis on the problem. We introduce standard concepts of approximation-based algorithms for combinatorial optimization problems (COPs) and provide an outline for the rest of the thesis.

1.1 Hypothesis

We hypothesize that a novel adaptation to the ALNS framework with conditional branching based on immediate feedback from each step of a search leading down to separate independent ALNS selection components can outperform regular ALNS in the presence of large sample sizes within each segment, we can effectively achieve large sample sizes by utilizing parallel computing. These concepts will be explained in later sections.

1.2 Motivation

Today, companies are working to reduce costs and climate emissions to meet requirements set forth by governing bodies around the world. For example, logistics companies are working to improve capacity utilization and reduce travel distances and cost. Manufacturing companies are working to improve resource utilization and productivity and cut waste. Power companies are working to produce electricity

cheaply and efficiently from renewable energy sources. Optimization is at the core of achieving all these goals, and in many areas, meeting one of these objectives has a positive spill over effect on the other. This is especially true in the field of logistics where improved capacity utilization and reduced travel distance for a distribution route is positive both in terms of cost and for cutting greenhouse gas emissions, killing two birds with one stone. Seeing how optimization is at the centre of addressing all these issues we were motivated to create a new type of algorithm for finding good approximations to real world complex optimization problems in a reasonable amount of time. We propose a new implementation to the well-known state-of-the-art ALNS framework in combination with the simulated annealing metaheuristic (SA) and some components from tabu search. Our proposed algorithm will be explained in chapter 3.

1.3 Description of the pickup and delivery problem with time windows

The pickup and delivery problem with time windows (PDPTW) is a well-known and well-studied NP-Hard logistics problem in combinatorial optimization (Savelsbergh & Solomon, 1995). Informally the problem can be described as follows: We are given a set of calls where something is to be picked up and delivered. Each call has a size, a pickup node, and a delivery node as well as time windows for the pickup and the delivery. There are different variations of the problem. Some use a decentralized heterogeneous fleet of vehicles with individual call compatibility restrictions, load capacities, an earliest possible departure time from origin nodes, travel times and travel costs between nodes, as well as individual handling time and toll costs (common in ship routing) at the loading/unloading nodes of each request. Some problem formulations use a homogeneous vehicle fleet, and some use a centralised depot location where all vehicles must start and end their route. Generally, the goal is to assign the vehicles to the set of calls in the cheapest way possible such that all calls are handled, and the problem constraints satisfied. In some formulations there is also an objective of minimizing the total number of vehicles required to serve the requests.

Vehicles that arrive at a node to service a call before its time window opens must wait until the window opens. Similarly, if a vehicle arrives late, then such solution is infeasible. Moreover, in the standard version of the problem, a vehicle is only required to arrive to service a call before its time-window closes. It is not required that the upper time window is respected when the loading/unloading time is accounted for. Many formulations of the problem also use an imaginary "outsource" vehicle. Calls may then be outsourced at spot charter rates which are usually much higher than handling the calls internally. For a complete mathematical formulation of the multi vehicle PDPTW in the context of industrial and tramp ship routing and scheduling, the reader is referred to Hemmati et al. (2014).

1.4 Solution representation and complexity analysis

A solution to the pickup and delivery problem with time windows can be represented as an array of a sequence of numbers where zero has a special meaning as a separator separating two vehicles. Calls from one up to $\boldsymbol{n}$ may then be placed between the separators, with each call occurring twice. The first occurrence denoting the pickup of the call, and the second occurrence denoting the delivery of the call. A solution may be read from left to right such that all calls appearing before the first zero belongs to the first vehicle, all calls between the first and the second zero belongs to the second vehicle and so on. We make a distinction between a valid solution and a feasible one. For a solution to be valid the only requirements are that there are exactly $\boldsymbol{k}$ number of zeros where $\boldsymbol{k}$ is the number of vehicles, and that each call occurs twice with both occurrences placed between the same separators i.e., each call is picked up and delivered by the same vehicle. For a solution to be feasible, it must be valid, and all the problem constraints must be respected.

Assume an instance of the PDPTW with seven pickup/delivery requests and three vehicles and an imaginary outsource vehicle. Then there are $\mathbf{4^7 = 16384}$ possible ways of distributing the calls among the four vehicles. This number does not include the number of ways to arrange the calls within each vehicle. If all calls are placed in one vehicle, there are $\frac{\mathbf{14!}}{\mathbf{2^7}}$ which is approximately 681 million arrangements of the pickup and delivery nodes. Although depending on problem formulation, many of these

solutions may be infeasible, and a state-of-the-art algorithm would not need to consider all of them, it illustrates that for an exact approach, the computational effort required quickly gets out of hand in the size of the problem.

To the best of our knowledge previous work has not established a mathematical formula for determining the number of valid solutions to the PDPTW with $\boldsymbol{n}$ calls and $\boldsymbol{k}$ vehicles. Therefore, we present such formula here in this paper. But before doing so, let us make a few observations on some simpler cases that will make the final formula easier to grasp. If we ignore the imaginary vehicle for a moment and consider a case with only a single vehicle and $\boldsymbol{n}$ calls to be picked up and delivered, then the formula $\boldsymbol{a(n) = \frac{(2n)!}{2^n}}$ (https://oeis.org/A000680) gives all possible solutions to such an instance where $\boldsymbol{(2n)!}$ gives all possible arrangements for $\boldsymbol{2n}$ unique elements and dividing by $\boldsymbol{2^n}$ removes identical solutions since a call always gets picked up before delivery.

By introducing the imaginary vehicle to the equation, the formula becomes slightly more complex, namely $\boldsymbol{a(n) = \sum_{i=0}^{n} \binom{n}{i} \frac{(2i)!}{2^i}}$. This happens to give a sequence identical to the number of strings of n symbols in the Stockhausen problem to which the above formula was contributed (https://oeis.org/A008269). The first few entries of the sequence are: $1, 2, 9, 112, 2921, 126966, 8204497, 735944084$ corresponding to the number of valid solutions for $\boldsymbol{n}$ calls, one vehicle and the imaginary vehicle. Adding additional vehicles is when the formula gets more complicated. It is not simply a matter of multiplying the above formula with $\boldsymbol{k}$ number of vehicles as this only covers the situation where calls are placed in the imaginary and/or one of the $\boldsymbol{k}$ vehicles. It does not account for calls being distributed among multiple vehicles (and) the imaginary vehicle. We present the complete formula for the number of valid solutions with $\boldsymbol{n}$ calls and $\boldsymbol{k}$ vehicles as follows:

$$a(n,k) = 1 + k\sum_{i=0}^{n}\left(\binom{n}{i} * \frac{(2i)!}{2^i}\right) - k +$$

$$\binom{k}{2} * \sum_{i=2}^{n}\binom{n}{i} * \sum_{t_1=1}^{i-1}\binom{i}{t_1} * \left(\frac{(2(i-t_1))!}{2^{i-t_1}} * \frac{(2*t_1)!}{2^{t_1}}\right) +$$

$$\binom{k}{3} * \sum_{i=3}^{n}\binom{n}{i} * \sum_{t_1=2}^{i-1}\sum_{t_2=1}^{t_1-1}\binom{i}{t_1} * \binom{t_1}{t_2} * \left(\frac{(2(i-t_1))!}{2^{i-t_1}} * \frac{(2(t_1-t_2))!}{2^{t_1-t_2}} * \frac{(2t_2)!}{2^{t_2}}\right) +$$

$$: \quad : \quad : \quad : \quad : \quad : \quad : \quad : \quad : \quad +$$

$$\binom{k}{k} * \sum_{i=k}^{n}\binom{n}{i} * \sum_{t_1=k-1}^{i-1}\sum_{t_2=k-2}^{t_1-1}\cdots\sum_{t_{k-1}=1}^{t_{k-2}-1}\binom{i}{t_1} * \binom{t_1}{t_2} * \ldots * \binom{t_{k-2}}{t_{k-1}} * \left(\frac{(2(i-t_1))!}{2^{i-t_1}} * \frac{(2(t_1-t_2))!}{2^{t_1-t_2}} * \ldots * \frac{(2t_{k-1})!}{2^{t_{k-1}}}\right)$$

Formula 1: Number of valid solutions for the multi vehicle PDPTW with outsourcing.

1.5 Standard concepts of approximation-based solution methods for combinatorial optimization problems

For many real-world optimization problems an exact solution is not necessary, and a solution relatively close to the optimum may be sufficient. The solution quality required generally depends on the real-world context in which the solution is to be used. In the context of required computational effort approximate solutions are generally easier to find and the computational effort needed usually increases the closer one gets to an optimal solution. We now step away from exact methods for solving the problem at hand and instead focus on methods that yield good approximate solutions in a reasonable time span. The first thing that comes to mind in this endeavour is the random search heuristic. Random search is the simplest form of heuristic. It randomly removes a set of elements from a solution and randomly inserts them somewhere else. Thorough testing has been conducted on random search and we have found that even for many small problem instance formulations of the PDPTW it frequently fails to find feasible solutions and in 10,000 iterations it rarely finds the optimal solution for small instances. For most problem instances of moderate size, it fails to find feasible solutions within 10,000 iterations and for most larger instances it fails to find any feasible solutions at all. However, by restricting this operator to randomly select only from feasible new solutions it becomes a powerful diversification operator as we shall see later.

For solving large complex combinatorial optimization problems, a standalone simple heuristic is not enough. Therefore, we introduce the concept of a metaheuristic for which a definition follows: "A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms" (Sörensen & Glover, 2013, p. 4). Most metaheuristics abstracts away from the structure and specific nature of the underlying optimization problem being solved. As it does not need to know anything about the underlying problem, it acts on generalizable feedback common to all optimization problem to make its decisions, most commonly the objective values which are the values returned by the function that we are trying to minimize or maximize subject to certain constraints on the variables. The second part of the definition means that the nature and origin of the selected metaheuristic may give guidance in deciding on the types of lower-level perturbative heuristics we should use.

A metaheuristic often makes use of a connected hyperheuristic component for which we borrow the following definition: "Hyperheuristics represent a novel search methodology that is motivated by the goal of automating the process of selecting or combining simpler heuristics in order to solve hard computational search problems." (Burke et al, 2009, p. 177). In simple terms a hyper-heuristic is a heuristic for selecting or generating low-level (often problem specific) heuristics.

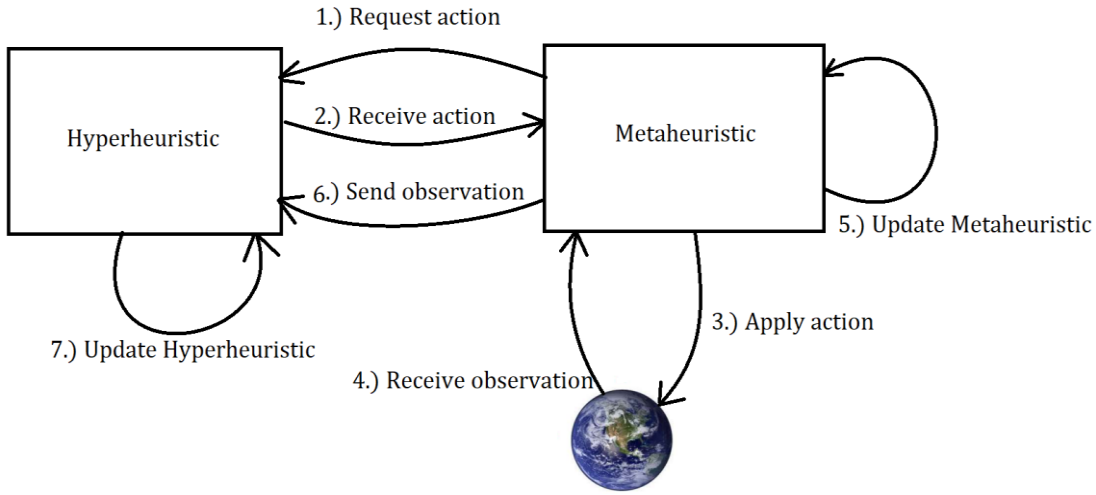


Figure 1.1: A simplified view of a step in an optimization algorithm with a metaheuristic and a hyperheuristic.

1.6 Thesis Outline

The outline for the rest of the thesis is the following:

Chapter 2 – Overview of the literature gives all the required definitions and theoretical background for the rest of the thesis.

Chapter 3 -A parallel perceptive ALNS presents the proposed algorithm and gives insight into an implementation in two distinct compute environments, namely the Open Multi-Processing (OpenMP) and the compute unified device architecture (CUDA) environments.

Chapter 4 – Experimental setup contains all the required details on how the testing was conducted, including the nature and origin of the benchmark instances used, details on hardware and the parameters used for the algorithms.

Chapter 5 – Results Presents the results from running our algorithm on many distinct PDPTW instances multiple times. The performance of our algorithm is measured using ALNS as baseline.

Chapter 6 – Concluding remarks and future work Provides a summary of what we have covered in the thesis and a conclusion is presented with respect to our initial hypothesis. Finally, we present possible future research directions.

Chapter 2

Overview of the literature

In this chapter important topics in the literature will be reviewed. A brief overview of a selection of different types of metaheuristics and hyperheuristics often used for solving pickup and delivery problems is given. This will be helpful in getting a better understanding of the many different paths one may follow when designing an approximation-based optimization algorithm. The algorithms that are fundamental to the theory presented in this paper will be explained in greater detail.

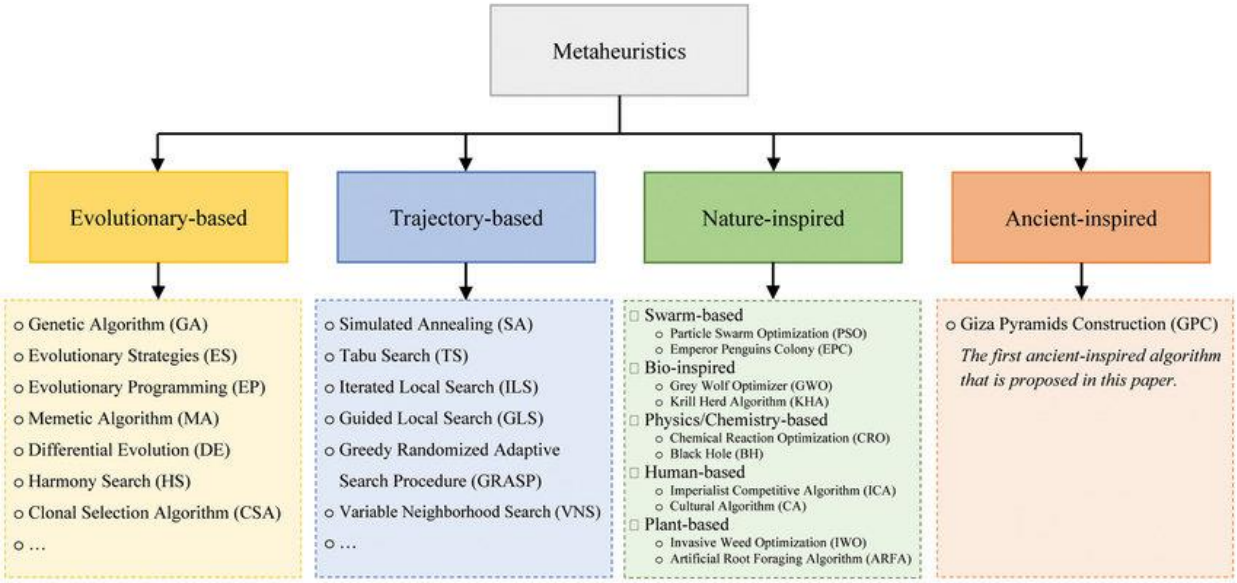


Figure 2.1: Some of the most widely used metaheuristics classified by their origin. Figure borrowed from Harifi et al (2021).

2.1 A Review of exact methods and matheuristics

The best-known exact methods to date for the PDPTW are state-of-the-art hybrids of branch and bound and column generation methods with cuts and relaxations that can reduce the search space to a degree. For example, Ropke & Cordeau (2009) developed a branch and cut and price algorithm giving exact solutions to small and some moderately sized PDPTW problems. Later Baldacci et al (2011) proposed an exact method based on set-partitioning–like integer formulation able to solve many PDPTW instances with around hundred requests. Most problem instances of moderate to large size remain computationally intractable for exact methods (Ropke & Cordeau, 2009). Another research direction that has received attention in the literature are the so called matheuristics combining elements from exact methods with elements from approximation techniques. For example, Parragh & Schmid (2013) proposed a hybrid of column generation and large neighbourhood search for the dial-a-ride problem where LNS was used to generate routes and set covering used to recombine them. Homsi et al (2018) combined unified hybrid genetic search with set partitioning outperforming all heuristics at the time and producing near optimal solutions to the PDPTW in industrial and tramp ship routing.

2.2 A review of some well-known metaheuristics

There are many different types of metaheuristics originating from different fields. Some are evolutionary based, some trajectory-based, some nature inspired, meanwhile others have a more exotic place of origin. We will now review some of the most widely used metaheuristics. The simplest form of metaheuristic is pure local search. in which a solution that is worse than the incumbent solution always gets rejected. More formally, assume for a minimization problem that we have a solution $\boldsymbol{l}$ and a new solution $\boldsymbol{l'}$ and $\boldsymbol{f(l)}$ gives the objective value of solution $\boldsymbol{l}$. Then $\boldsymbol{l'}$ is accepted if and only if $\boldsymbol{f(l') < f(l)}$. The disadvantage of this metaheuristic is that if it falls into a locally optimal solution, it may not be able to escape from it. Many of the more advanced metaheuristics have a mechanism for sometimes accepting a worse solution with the expectation that this will help in escaping local optimums and lead to an overall better result in the long run, an example is Tabu Search (Glover, 1989).

In tabu search a modified local search explores the neighbourhood of an incumbent solution and accepts a worse solution if no improvements or acceptable solutions are available. A tabu list keeps track of solutions that have recently been observed. The word "tabu" gives a good indication that solutions currently on the list cannot be accepted by the metaheuristic, and indeed this is the case. Members of the list usually have an expiration time such that a solution observed **n** iterations ago is removed from the list and may again be accepted if encountered. Medium term memory structures are sometimes used to intensify a search around promising areas of a search space and longer-term memory structures are used as an escape mechanism if a search becomes stuck in locally optimal areas. In this case the algorithm will jump back to some previous solution in some area of a search space that has not been thoroughly explored (Glover, 1989).

Genetic algorithm (GA) is another well-known metaheuristic developed by John Holland and his colleagues in the 1960's and 70's and is based on evolutionary theory inspired by Charles Darwin. In GA a set of initial feasible solutions are randomly generated and treated as the "initial population." A fitness score is given to each member of the population. The algorithm then selects parents for the next generation where those with high fitness score have a greater chance of being a parent compared to those with low fitness. The next generation consists of three types of individuals, elites, crossovers, and mutations. Elites are the members of the current generation with the highest fitness, they pass on to the next generation unchanged. Crossovers are created by combining vectors of pairs of parents and mutations are introduced by making random changes to a parent. In constrained optimization problems, there should be some mechanism ensuring that only feasible changes are made. The part of the population not selected as parents dies in the current generation. This process repeats until some stopping criterion is met at which point the individual with the highest fitness constitutes the best solution found (Holland, 1975).

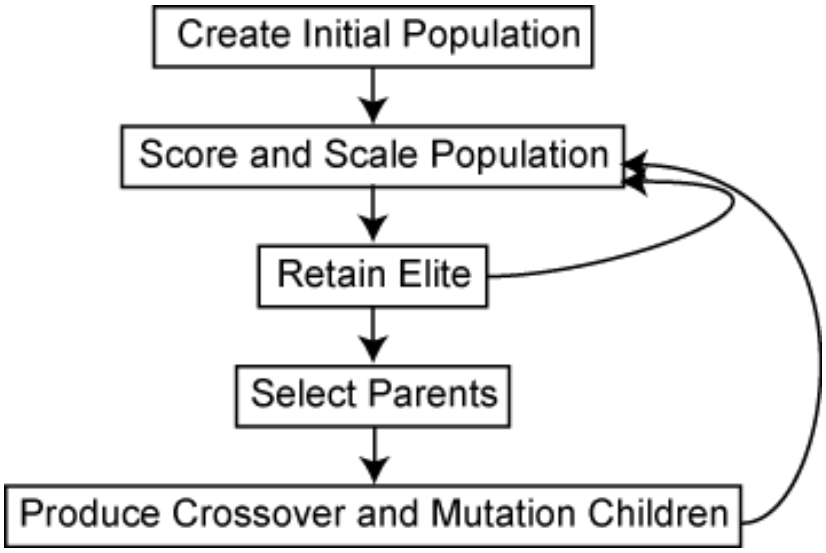


Figure 2.2: A flow chart of a step (generation) in the genetic algorithm. Figure from https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html

Another very interesting population-based metaheuristic is the nature inspired particle swarm optimization algorithm (PSO). This metaheuristic also uses a randomly generated initial population (solutions) as with GA. The fundamental idea of PSO is that each particle in the swarm attempts to avoid collision with other particles while exploring its own neighbourhood. Furthermore, each particle is drawn towards its locally best-found solution as well as the best-known solution of the entire swarm. Directional velocities which are weighted averages of directional vectors towards each particle's best-known solution and the swarms globally best-known solution are calculated. The weighting as well as the acceleration of the particles are implementation dependant. In each step of the PSO algorithm each particle moves a certain distance determined by its current velocity, acceleration, collision avoidance, the particle's best-known location and the swarm's globally best-known location. Over time the expectation is that the swarm will be drawn towards a near optimal solution (Kennedy & Eberhart, 1995).

2.3 The Simulated Annealing Metaheuristic

We will now discuss our metaheuristic of choice in greater detail. The simulated annealing metaheuristic was proposed by Kirkpatric et al (1983) and later by Cerny (1985). The SA metaheuristic is a simulation of the natural process of annealing. When substances are slowly cooled their atomic structure gradually seeks a state of equilibrium causing the underlying substance to become more stable and resilient compared to when cooling is done too rapidly. As the temperature decreases slowly over time, the ability of the atoms to move around in the structure becomes more and more restrictive. To illustrate a connection between finding approximate solutions to combinatorial optimization problems and the process of annealing, we quote:

> *There is a deep and useful connection between statistical mechanics (the behaviour of systems with many degrees of freedom in thermal equilibrium at a finite temperature) and multivariate or combinatorial optimization (finding the minimum of a given function depending on many parameters). A detailed analogy with annealing in solids provides a framework for optimization of the properties of very large and complex systems. This connection to statistical mechanics exposes new information and provides an unfamiliar perspective on traditional optimization problems and methods.* (Kirkpatric et al, 1983, p. 671)

At the core of the simulated annealing metaheuristic is the acceptance criterion for a new solution. Let $\boldsymbol{\zeta}$ be the current objective value of some solution to a minimization problem. Then for any $\boldsymbol{\Delta\zeta \leq 0}$ we accept, but a change in objective $\boldsymbol{\Delta\zeta > 0}$ may also be accepted probabilistically by using the Boltzmann probability function: $\boldsymbol{p(\Delta\zeta) = e^{-\frac{-\Delta\zeta}{T}}}$, where p is the probability of acceptance and $\boldsymbol{T}$ is the current temperature of the system. A uniform random number $\boldsymbol{r}$ in the range $\boldsymbol{(0, 1)}$ is generated and if $\boldsymbol{r < p(\Delta\zeta)}$ the new solution is retained. This is also known as the metropolis criterion.

There are many ways of selecting a temperature schedule. In our implementation we are using a warmup phase where all feasible solutions are accepted. During this phase we calculate the mean of all $\boldsymbol{\Delta\zeta > 0}$ and use this to calculate a starting temperature $\boldsymbol{T_0}$ and an end temperature $\boldsymbol{T_f}$. $\boldsymbol{T_0}$ then becomes $\boldsymbol{T_0 = \frac{-k_0 * mean(\Delta\zeta > 0)}{\ln(P_0)}}$ and $\boldsymbol{T_f = \frac{-k_f * mean(\Delta\zeta > 0)}{\ln(P_f)}}$ where $\boldsymbol{P_0}$ and $\boldsymbol{P_f}$ are the desired start and end probabilities, and $\boldsymbol{k_0, k_f}$ are arbitrary adjustment coefficients. The cooling schedule becomes $\boldsymbol{(\frac{T_f}{T_0})^{\frac{1}{S}}}$ where $\boldsymbol{S}$ is a fixed number of steps to be taken.

The ability of SA to sometimes accept non improving solutions is an important feature of this metaheuristic allowing it to escape a local optimum. But there is another important feature of the SA metaheuristic disguised in this ability: "A second and more important feature is that a sort of adaptive divide-and-conquer occurs" (Kirkpatric et al, 1983, p. 673). In the early stages when the temperature is high SA explores very large areas of a search space via a rough search method, thereby in a sense dividing it. As the temperature decreases SA has been statistically shown to intensify around promising areas of a search space, in a sense "conquering" it. In our implementation, additional measures have been taken to ensure intensification in promising areas which will be explained in later sections.

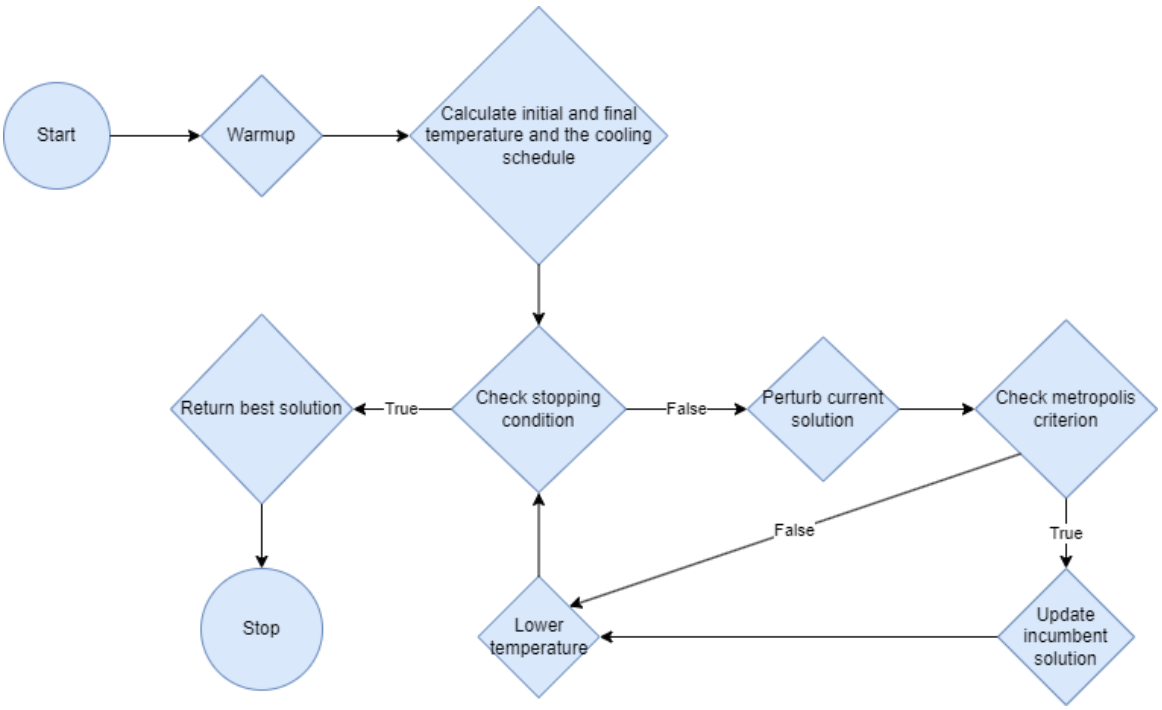


Figure 2.3: Flowchart of the simulated annealing metaheuristic.

2.4 Hyperheuristics

Most metaheuristics are pure high-level frameworks that reacts to high-level generalizable feedback. Most of them do not provide a mechanism for acting directly on the low-level solution domain. Genetic programming is an example of an exception to this rule as it provides guidelines for perturbing solutions by mutations and crossovers, hence may be called a hybrid. Metaheuristics that do not themselves provide guidelines for acting on the solution domain do need a hyperheuristic: A heuristic for selecting/combining lower-level heuristics. As well as a non-empty set of lower-level heuristics to select from.

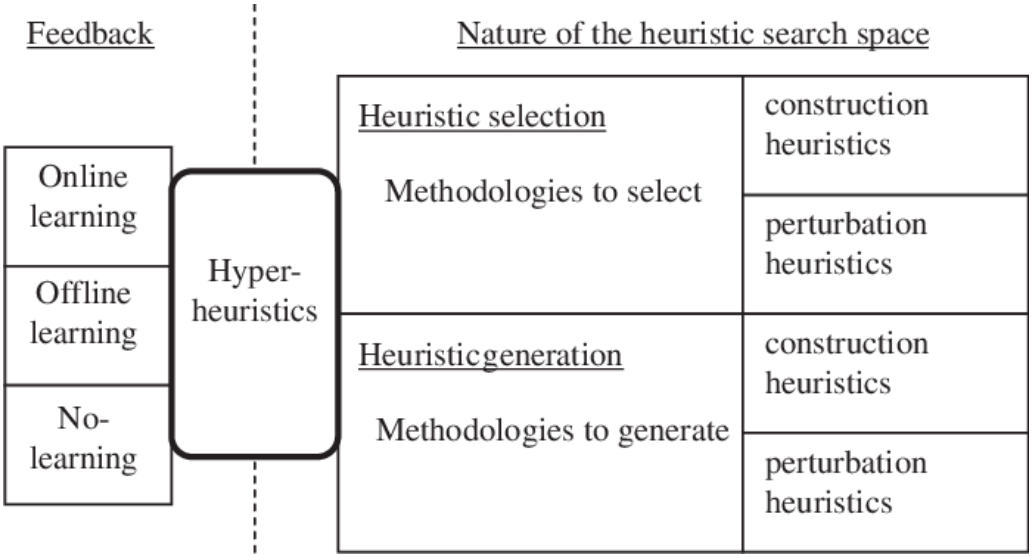


Figure 2.4: A classification of hyper-heuristic approaches according to the origin of feedback and the nature of the heuristic search space. Borrowed from Burke et al (2019).

Hyperheuristics can be divided into two main categories, generation based, and selection based hyperheuristics. Generation based hyperheuristics seek to create new types of heuristics by combining small incomplete heuristic building blocks into complete heuristics using machine learning techniques, where perhaps the most popular approach is genetic algorithms as explained in Burke et al (2009). Such an approach may require constructs to be in place ensuring that only feasible heuristics are created. Selection hyperheuristics on the other hand selects from a predefined set of complete standalone heuristics. We will focus on selection hyperheuristics in this paper.

Beyond the classification of selection/generation is a further classification in the types of lower-level heuristics that is used, construction heuristics and perturbation heuristics. Construction heuristics are lower-level heuristics that builds a solution from bottom up gradually throughout the length of a search, solutions here need not be complete from one iteration of a search onto the next. Perturbation heuristics on the other hand are ones that perturbs a complete solution, always leaving a complete solution after each iteration of a search.

Hyperheuristics are divided into three categories based on their feedback mechanism. Online learning, offline learning, and no learning. Hyperheuristics that adapt to situations that arise during a search of a solution space belongs to the online learning group. These hyperheuristics use reward functionality, statistical observation and/or machine learning to modify the order of how and when heuristics are called during a search, an example is the selection and update component of the adaptive large neighbourhood search framework. Offline learning is the class of hyperheuristics that can be trained ahead of time and are able to carry this training over to new problem instances, for example deep reinforcement learning hyperheuristic which mixes online and offline learning. Finally, in the no learning class are the ones that cannot be trained ahead of time and that cannot adapt to events during a search.

The simplest form of selection hyperheuristic is uniform random selection (URS) belonging to the no-learning feedback class. URS, as the name suggests always randomly selects from a pool of heuristics with maximum entropy. This hyperheuristic works well if there are no exploitable patterns in the order in which heuristics are called. I.e., every heuristic is equally likely to work in every situation, this is seldom the case as explained in Drake et.al (2020) referring to a study by Fisher & Thompson (1963). Additional members of this group are the heuristic selection component of variable neighbourhood search and large neighbourhood search. They use predefined sequences or probabilities for heuristics, and these do not adapt to results during a search (Burke et al, 2019).

2.5 Adaptive large neighbourhood search

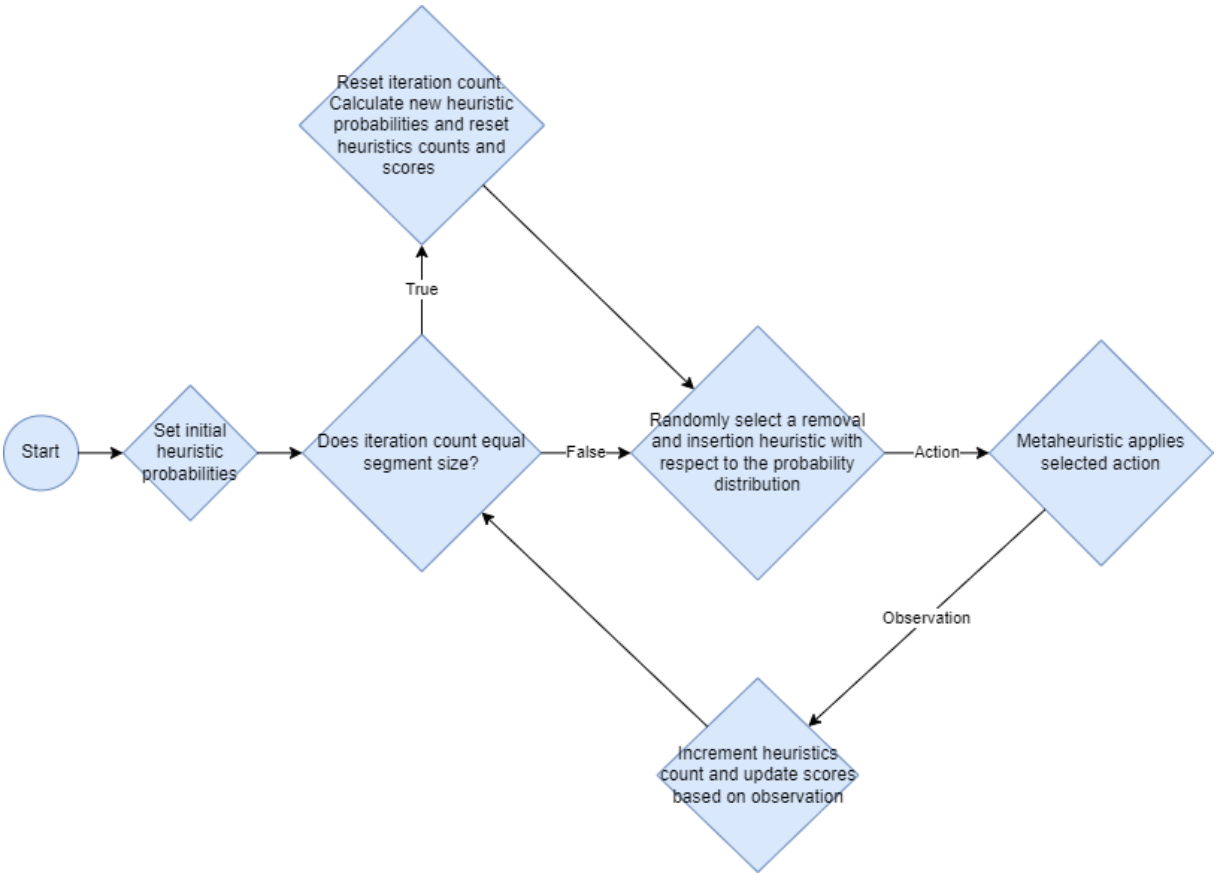


Figure 2.5: Flowchart of the ALNS heuristic selection and update mechanism.

Adaptive large neighbourhood search (Pisinger & Ropke, 2006 & 2007) is an optimization framework extending the LNS framework providing it with an adaptive heuristic selection layer. LNS selection uses predefined fixed probabilities in selecting from a set of low-level heuristics in each step of a search. The heuristic selection mechanism of ALNS on the other hand usually starts out at maximum entropy, and adaptively changes based on observations and reward functionality throughout the course of a search, thus this component belongs to the online learning classification.

Algorithm 1: Adaptive Large Neighbourhood Search

Input: An optimization problem.
Output: A solution to the optimization problem.
1 ***Construct a feasible solution*** x; ***set*** $x^* := x$
2 ***Repeat:***
3 Choose heuristic pair **j** consisting of destroy neighbourhood N^- and repair neighbourhood N^+ according to **Algorithm 2.**
4 Generate a new solution x' from x using the heuristics corresponding to the chosen destroy and repair neighbourhoods
5 if x' can be accepted, then set $x := x'$
6 Update count τ_j and score π_j of heuristic pair j
7 **if** end of segment; **update** neighbourhood probabilities according to **Algorithm 3**
8 if $f(x) < f(x^*)$ ***set*** $x^* := x$
9 ***Until stop criteria is met***
10 ***Return*** x^*

(Pisinger & Ropke, 2007).

The ALNS pseudocode describes a high-level problem independent optimization framework that can be applied to mostly any optimization problem. It extends an acceptance criterion and defines a selection mechanism controlling the use of lower level, mostly problem specific heuristics, referred to as destroy and repair neighbourhoods, N^- ***and*** N^+ in Pisinger & Ropke (2007). From now on we will refer to them as removal and insertion heuristics or operators. Starting with a fixed probability for each of the pair of heuristics that can be selected (usually uniform), a pair is selected probabilistically at each stage of a search using the following "roulette wheel" procedure where $p(i)$ is the probability of selecting heuristic pair i:

Algorithm 2: ALNS heuristic selection mechanism

Input: A set of heuristic pairs with probabilities to select from.
Output: The selected heuristic pair.
1 ***Initialization of variables:*** $t \leftarrow 0.0 \quad i \leftarrow -1$
2 ***Assign random value to r:*** $r \leftarrow [0,1)$
3 ***while*** $(t \leq r)$ ***do:***
4 $i \leftarrow i + 1$;
5 $t \leftarrow t + p(i)$;
6 ***end***
7 ***return*** i

We observe that for long segments and large sets of heuristics, it is beneficial in terms of performance to calculate accumulative probabilities of $\boldsymbol{p(i)}$ such that one may perform binary search to find the heuristic pair to use instead of potentially having to iterate from zero to $\mathbf{n}$. Reducing the selection complexity from $\boldsymbol{O(n)}$ to $\boldsymbol{O(log(n))}$.

The selected pair of remove and insert heuristics is applied on the solution domain and the resulting new solution may either be accepted or rejected based on acceptance criterion used. The result of the action taken is passed on to the ALNS's update mechanism which updates the appropriate score according to predefined rules from a reward scheme. This process continues up until the number of iterations reaches the end of a segment, where the probability of each operator is recalculated according to the following algorithm where $\mathbf{rf}$ is the reaction factor, meaning the weight to be applied to the new probabilities calculated from the counts and scores of the current segment over the previous probabilities. $\boldsymbol{p(i)}$ is the probability of selecting heuristic pair $\boldsymbol{i}$. $\boldsymbol{s(i)}$ is the score of heuristic pair $\boldsymbol{i}$, and $\boldsymbol{c(i)}$ is the count of heuristic pair $\boldsymbol{i}$.

Algorithm 3: ALNS update mechanism

Input: A set of scores, counts and probabilities for selecting each heuristic pair.
Output: New heuristic selection probabilities.

1 *for all* operator pairs $\boldsymbol{i \in Op}$ *do:*
2 $\quad \boldsymbol{p(i) = p(i) * (1 - rf) + rf * \frac{s(i)}{c(i)}}$
3 *end for*
4 *for all* operator pairs $\boldsymbol{i \in Op}$ *do:*
5 $\quad \boldsymbol{p(i) = \frac{p(i)}{\sum_{j=0}^{n} p(j)}}$
6 *end for*
7 ***Reset counts and scores:*** *for all* operator pairs $\boldsymbol{i \in Op}$ *do:*
8 $\quad \boldsymbol{c(i) \leftarrow 0}$
9 $\quad \boldsymbol{s(i) \leftarrow 0}$
10 *end for*

To prevent some probabilities from tending to zero it is common to put a lower bound on $\boldsymbol{p(i)}$. For further details on ALNS, the reader is referred to Pisinger & Ropke (2010).

2.6 Motivation for guided ejection search

The motivation for this heuristic is the observation that for the purpose of route elimination over travel cost minimization in the context of VRPs with time windows, there are great difficulties in arriving at a representative and responsive objective function required for metaheuristic algorithms to be effective in pursuing the desired objective. For example, Bent & Van Hentenryck (2006) argues that the standard objective function of minimizing the total cost may in many cases drive the search towards solutions with low travel costs and many short routes making it difficult to reach solutions with fewer but longer routes and higher travel costs, i.e., routes that makes better use of the time windows and hence are able to service more calls. They propose a two-stage algorithm for the dual objective PDPTW where the first stage focuses on minimizing the total number of routes. Here they use the simulated annealing metaheuristic with a simple pair relocation heuristic and a modified objective function consisting of three objective components with weights o1 $\gg$ *o*2 $\gg$ *o*3. The first component is the number of routes, second is the sum of the negative squares of the length of each route which favours longer routes and shorter routes over requests evenly distributed among all the routes, the final component is the sum of the travel cost of all the routes. In the second stage they use LNS with the objective function of total travel cost for minimizing the travel cost. This approach gave many new best results on the Li & Lim benchmark instances at the time of publishing.

Ropke & Pisinger (2006) came up with another two-stage approach for minimizing the number of routes. Starting from a feasible schedule, a route is removed, and the requests placed in a request bank where they are given an arbitrary high cost such that they get moved back into the remaining routes whenever possible. Guided ejection search was not known at the time, and a simple LNS is used for attempting to move the calls in the request bank back into the remaining routes. If LNS succeeds in emptying the request bank another route gets cleared becoming the new request bank. This process runs for a predefined time limit and the best feasible solution with the fewest number of routes becomes the starting point for the second phase of the search where minimizing the travel cost is the objective. The approach was able to find quite a few new best solutions to the Li & Lim benchmark instances at the time but became obsolete after the discovery of guided ejection search.

2.7 Guided ejection search

Guided ejection search (GES) is a search heuristic originally proposed by Nagata & Tojo (2009) for the job shop scheduling problem where a set of jobs $j \in J$ are to be processed on a subset $m \subseteq M$ machines. Job j consists of t steps that must be completed in a predefined order with no overlap. Each machine may only work on one job at a time and for each step t in a job there is an associated processing time on the required machine m. The objective is to find the job schedule that minimizes the total makespan. GES was later proposed for the pickup and delivery problem with time windows with dual objectives by Nagata & Kobayashi (2010) where it was designed solely for the purpose of route elimination, and it represents the current state-of-the-art in this regard. The GES algorithm proposed by Nagata & Kobayashi (2010) for the dual objective PDPTW can be described as follows:

Algorithm 4: Guided Ejection Search for the PDPTW

Input: A feasible pickup and delivery schedule.

Output: A feasible pickup and delivery schedule with number of routes minimized.

until stopping condition:

    Select a route and unassign all requests and place them in a request queue

    while remining attempts and unassigned requests ***do:***

        Select an unassigned request

        Check for all possible rearrangements of at most k (p, d) pairs in the remaining routes, **if** the unassigned request can be inserted, **then** insert the request

        else perturb the routes by several rearranging moves.

    end while

end

(Nagata & Kobayashi, 2010)

The above algorithm may be considered a brute force heuristic within a limited neighbourhood of size k determining if the insertion of some unassigned request pair is possible within that neighbourhood. Indeed, the computational effort required grows in the size of k and the same is true for how certain one can be regarding the quality of the solution obtained.

Curtois et al (2018) proposed an adaptive GES in combination with local search and LNS with late acceptance hill climbing (LAHC, Burke & Bykov 2008) which is similar to simulated annealing except that it replaces the probability-based acceptance of worse solutions by a time-based deterministic one. The outlined algorithmic framework combining AGES, LS and LNS resulted in a robust and effective two-stage algorithm for the dual objective pickup and delivery problem with time windows where AGES was used for the route elimination phase and LS and LNS used for the objective of total travel cost minimization. They found new best solutions for a large majority of the Li & Lim PDPTW benchmark instances.

The AGES algorithm outlined in Curtois et al (2018) differs from the original GES proposed by Nagata & Kobayashi (2010) in several ways. Firstly, instead of keeping the value of $\boldsymbol{k}$ explained earlier fixed, they increase its value in steps, starting at zero, meaning that no pairs are rearranged, and an attempt is made at simply inserting an unassigned request at an available position in one of the routes. If unsuccessful $\boldsymbol{k}$ gets increased to the value of one and an attempt is made at inserting the unassigned request by all possible rearrangements of each single $(\boldsymbol{p}, \boldsymbol{d})$ pair, one at a time. Here a loop iterates over all $(\boldsymbol{p}, \boldsymbol{d})$ pairs in the routes and if the removal of one of the $(\boldsymbol{p}, \boldsymbol{d})$ pairs allow the insertion of the unassigned request, then that pair allowing the insertion is removed, and it is checked whether that removed pair can be inserted somewhere else in one of the routes. If this is unsuccessful for all the $(\boldsymbol{p}, \boldsymbol{d})$ pairs, then the value of $\boldsymbol{k}$ gets increased to two which is the maximum value for $\boldsymbol{k}$ used in the implementation by Curtois et al (2018). As an example of a successful application of AGES with $\boldsymbol{k} = \boldsymbol{2}$ assume $\boldsymbol{q1}$ is the unassigned request and the pairs $\boldsymbol{e1}$ *and* $\boldsymbol{e2}$ gets ejected from the routes. If $\boldsymbol{q1}$ can be inserted in the route of $\boldsymbol{e1}$ by the removal of $\boldsymbol{e1}$, and $\boldsymbol{e1}$ can be inserted in the route of $\boldsymbol{e2}$ by the removal of $\boldsymbol{e2}$ and finally $\boldsymbol{e2}$ can be inserted back into the route where $\boldsymbol{e1}$ was replaced by $\boldsymbol{q1}$, then such a series of events represents a successful application of AGES. Curtois et al (2018) also uses an adaptive heuristic for selecting the pair(s) to eject based on how difficult it has been to insert in the past. For the stopping criterion of their AGES, they use an adaptive mechanism where if GES is unsuccessful at reducing the size of the unassigned queue for a while and there are more calls in the queue than a given threshold, GES will stop early. Otherwise, it is allowed to perform a maximum of one million iterations without improvement in their implementation.

2.8 Exploring parallel metaheuristics

Population-based metaheuristics lends themselves more easily to parallelization than their counterpart, single point-based metaheuristics. Particle swarm as an example, intuitively one may use a thread to represent a particle in a particle swarm. Venter & Sobieszczanski-Sobieski (2006) reports good solution quality with a parallel synchronous implementation and similar solution quality with an asynchronous implementation of PSO with lower runtimes. Their synchronous version assigns each particle to individual threads, where each particle takes a step according to the PSO algorithm described earlier. Synchronization and reduction follow each step to ensure updated information on particle locality and the best objective found so far. This process repeats **n** number of times. Their asynchronous version uses a shared memory locations for each particles data and for the global best solution. Although this could lead to race conditions in updating a global best, they determined through empirical observations that the negative effects on solution quality are low compared to the gain in efficiency. In Chang et al (2005), instead of parallelizing particles inside a swarm another PSO parallelization strategy is proposed where groups of particle swarms run in parallel on individual threads or groups of threads communicating with one another and exchanging information between swarms. Their experiments are promising.

Another metaheuristic that is commonly parallelized in the literature is the genetic algorithm. We see multiple approaches for parallelization. The simplest form is to parallelize the calculation of fitness values (highly parallelizable), the creation of crossovers and mutations (highly parallelizable) followed by calculation of new fitness values (highly parallelizable). The selection of new offspring and elites requires sorting the new fitness values and careful selection (Abbasi, 2020). Other more sophisticated parallelization exists that uses multiple independent "chromosome pools" where strong candidates from one pool sometimes spill over into other pools replacing weak candidates as proposed in Chipperfield et al (1996) in the section on migration GAs. Another interesting approach discussed in the same paper is diffusion GA where a single population is used but mating is restricted to only occur between chromosomes in close proximity on the population surface. The authors concludes that parallel GAs with some restrictions enforcing more local selection and reproduction are superior to GAs with unconstrained global selection and reproduction.

2.9 A parallel ALNS

A parallel-ALNS framework for the traveling salesperson problem with pickup and deliveries (TSPPD) is presented in Ropke, S (2009). The TSPPD solved is without capacity, precedence, and time restrictions. The problem is identical to the unconstrained single vehicle pickup and delivery problem without outsourcing. As this problem was explained in detail in the introduction, we conclude from introductory remarks that there are a total of $\frac{(2n)!}{2^n}$ feasible arrangements of the pickup and deliveries. The paper proposes a parallelization strategy where a large pool of threads works on a single solution and share the same probabilities of selecting from destroy and repair heuristics. The presented framework uses the simulated annealing's metropolis criterion for acceptance/rejection. In each iteration of a search, every thread gets its own copy of a shared incumbent solution and randomly selects a destroy and repair neighbourhood using ALNS such that the difference in results between threads that select the same operator pair comes from elements of randomness within the individual operators. Once all threads have finished their perturbations, the best new solution is passed on and accepted/rejected according to the acceptance criterion. Finally, the ALNS weights are adjusted before the next iteration begins.

The single-depot distance-constrained capacitated homogenous vehicle routing problem or standard DCVRP is a problem in the family of vehicle routing problems. Informally there is a depot with a set of vehicles that has a maximum carrying capacity as well as a maximum travel distance from departure to arrival at the depot. Furthermore, there is a set of customers with demand of a quantity of goods from the depot, and undirected edges defining travel cost and travel distance between each customer and between each customer and the depot. Every customer is visited exactly once, and the objective is to assign the vehicles in a way that minimizes the total cost while serving all the customer demands and respecting problem constraints such as time windows and vehicle load capacity. For further information on the DCVRP the reader is referred to Toth & Vigo (2014).

A parallel ALNS implementation for the standard DCVRP running on the graphical processing unit (GPU) relevant to our research is proposed by Bach et al (2019). The authors debate the single solution design versus the multiple solution design, and arguments are given that a single solution approach is going to suffer from frequent synchronization between thread blocks, which is costly in terms of time on the GPU. Furthermore, it is argued that such an approach will lead to stalls where blocks that finish early remains idle until the most demanding thread block completes its perturbations leading to lower GPU utilization. The authors conclude that a multiple solution design is the most suited approach where instead of each thread copying from a single "master" solution every iteration, every thread block works on independent solutions throughout the entire course of a search.

The authors are executing exactly enough blocks to meet the occupancy limit of their device and with an assignment of 512 threads to each block it should yield at most 56 concurrent blocks in total for the Titan GPU they are using with 28672 active threads. This gives a type of 2-level hierarchy of parallelism where both a moderate number of solutions are being worked on in parallel, and operations on individual solutions are done in parallel (Bach et al, 2019).

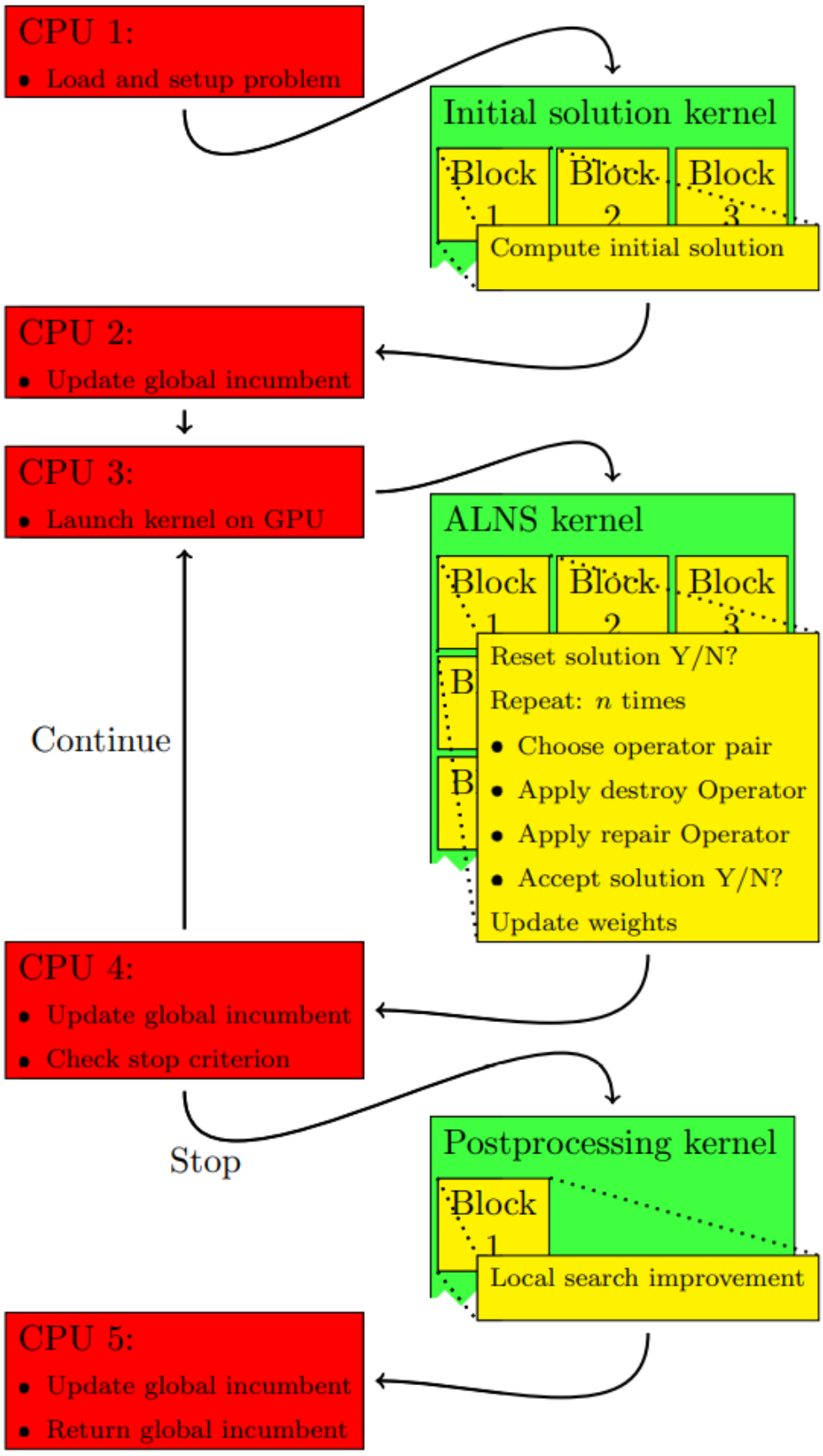


Figure 2.6: General design of GPU-ALNS borrowed from Bach et al (2019).

Bach et al, (2019) describes a GPU-ALNS algorithm where initial loading and setup of host data structures is done by the central processing unit (CPU). A GPU kernel initializes the device data structure and generates initial feasible solutions within the problem instance of each block. Control is returned to the CPU where the initial global incumbent solution is determined. Then the ALNS kernel is launched, where each block decides whether to reset its incumbent solution or not. This decision is based on whether data made available from the last global synchronization shows that the block's solution has likely converged or is futile, if so then a new initial solution is generated by the initial solution kernel. The standard ALNS with a segment of size $\boldsymbol{n}$, where $\boldsymbol{n = 100}$ follows. Each block performs an ALNS where an operator pair is selected probabilistically, the destroy operator is applied, followed by the repair operator, and the new solution either accepted or rejected based on an acceptance criterion, (here the metropolis acceptance criterion). After $\boldsymbol{n}$ iterations the weights are updated based on the counts and scores recorded in the segment. A global synchronization happens where the CPU updates the global incumbent to the best solution found so far and the stopping criterion is checked. If the stopping criterion is not met, then the ALNS kernel is called again and runs for another $\boldsymbol{n}$ iterations. Once the stopping criterion is met a postprocessing kernel is launched that performs a local search on the global incumbent returning once no more local improvements can be found. The CPU finally updates and returns the global incumbent to the caller (Bach et al, 2019).

A linked list data structure is proposed instead of an array representation to represent a routing schedule. The authors point out that removal and insertion of nodes in an array requires $\boldsymbol{n}$ operations $\boldsymbol{O(n)}$ where $\boldsymbol{n}$ is the size of the array, since the elements of the array will have to be shifted left/right to accommodate the removal of a node and reinsertion somewhere else. A linked list on the other hand can accommodate removal and insertions in constant time $\boldsymbol{O(1)}$. It is just a matter of updating the address/index of the preceding node's next node index and the succeeding node's previous node index. Although deciding on the location of a particular node $\boldsymbol{i}$ in a vehicle $\boldsymbol{j}$ requires linear time in a linked list as opposed to constant time in an array, the authors argue that such knowledge is seldom used and that the iterations of the nodes tend to begin at the end nodes of a vehicle.

Bach et al (2019) uses a standard set of destroy and repair heuristics. Destroy heuristics include worst removal, random removal, related removal, historical node-pair removal, and cluster removal. Insertion heuristics include greedy insert and regret-2. Parallelization is used where appropriate, and the authors describe areas where parallelization is more suitable than others. Parallelization can be useful in for example determining the best insertion location of a delivery point, one may use multithreading to examine the cost of insertion on multiple insertion points in parallel and do a parallel reduction to find the cheapest option, another example is determining the most expensive delivery location in a current solution, one would calculate the savings obtained by removing each node in a solution in parallel and do a reduction where one ends up with the node yielding the greatest saving when removed. The authors explain operations that are less suited for parallelism such as the selection of a set of random nodes in a current solution, and the updating of nodes within the same vehicle in the linked list solution representation since that requires updating the addresses of the preceding and succeeding nodes which may cause faults if two neighbouring nodes are removed or inserted simultaneously by different threads.

The article reports good GPU capacity utilization and competitive results on many DCVRP instances both in terms of solution quality and execution speed. For more details the reader is referred to Bach et al (2019).

Chapter 3

A Parallel Perceptive ALNS

In this chapter we explain our proposed algorithm in detail and present two implementations of the algorithm for the pickup and delivery problem with time windows. One running on the CPU using the OpenMP parallelization framework and one developed for the compute unified device architecture (CUDA) running on an Nvidia GPU. We present justification for why we believe such an extension to the ALNS framework is useful and discuss motivation for our design, and present challenges and solutions to these challenges.

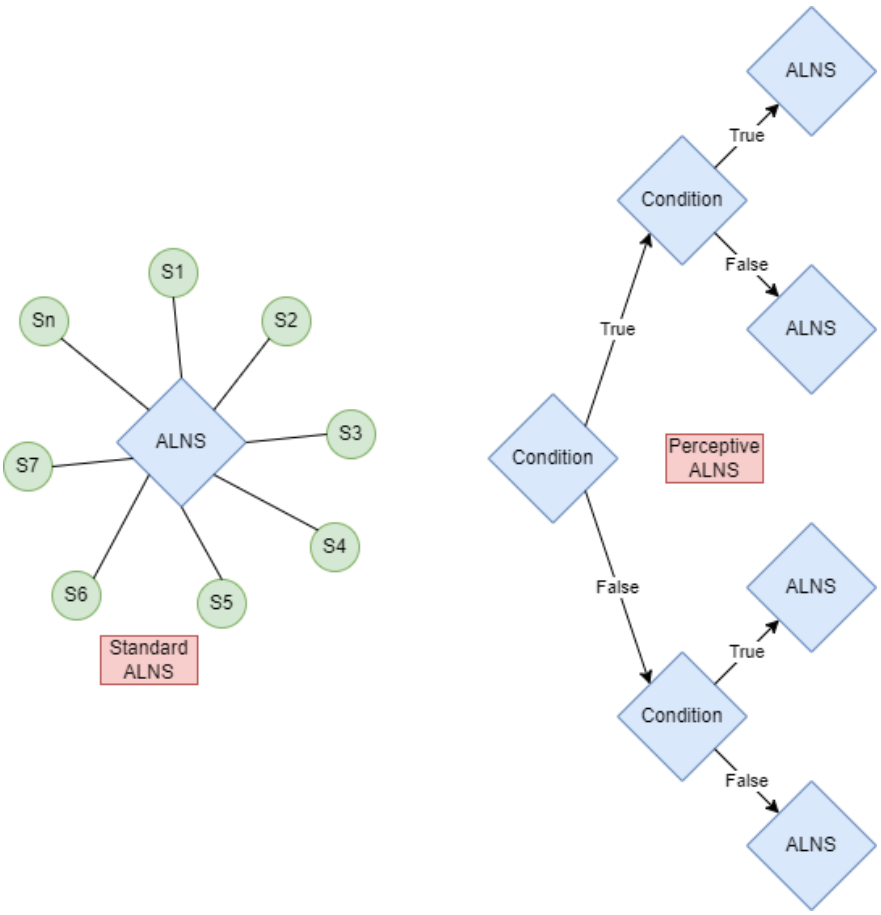


Figure 3.1: Graphical illustration of the key difference between ALNS and the proposed perceptive ALNS.

3.1 Introduction to PALNS

As previously discussed, the ALNS selection/update mechanism is a state-of-the-art hyperheuristic component for selecting from a pool of lower-level heuristics by adapting the likelihood of each heuristic being called every $\boldsymbol{k}$ step throughout the length of a search, and it has received a lot of praise from many researchers in the community since its inception in Pisinger & Ropke (2006 & 2007). However, there are some key potential disadvantages that should be discussed. Suppose there is some intensifying heuristic that works well after some other heuristic is used and an improved objective value not previously seen is found, or some diversifying heuristic works well after some intensifying heuristic fails to find an improved solution as it could be argued that the likelihood of being in a local optimum is higher in such a case. There may also be some deterministic heuristic that is redundant to call two times in a row if the objective does not change the first time. ALNS does not provide a way to exploit such successful patterns in the order in which the lower-level heuristics are used and does not provide a way to immediately act on results from a single perturbation. ALNS randomly selects a heuristic pair from a probability distribution and must wait until the current segment finishes before it is able to modify these likelihoods.

To address these challenges, we propose a modification to the standard ALNS, we call it "perceptive" ALNS. The change as illustrated in figure 3.1 is to have multiple conditional branches all leading down to separate independent ALNS selection components. It should be noted that the number of samples required to saturate such nested ALNS structures grows exponentially in the depth of the conditional tree. Therefore, it is recommended to stay conservative in the number of features used. This is also our main motivation for using parallel computing as it allows us to generate independent samples across independent threads working on unique instances of the same problem in each step of a search. As an example, assume that we have a pool of $\mathbf{12}$ different heuristic pairs that we can select from and we want to consider the previous heuristic pair used, whether that action is successful defined as a binary that is true if it leads to an improvement in objective, and a binary denoting whether the new solution has been previously encountered. PALNS has $\mathbf{12 * 12 * 2 * 2 = 576}$ potential targets as opposed to standard ALNS which only has $\mathbf{12}$ targets. This clearly illustrates that the potential benefit of PALNS comes at a great cost.

Algorithm 5: Perceptive Adaptive Large Neighbourhood Search

Input: An optimization problem.

Output: A solution to the optimization problem.

1 Construct a feasible solution x; **set** $x^* := x$

2 **Repeat**

3 Choose heuristic pair j consisting of destroy neighbourhood N^- and repair neighbourhood N^+ by identifying the appropriate ALNS branch based on the most recent observation(s) and applying **Algorithm 2** to the selected ALNS branch

4 Generate a new solution x' from x using the heuristics corresponding to the chosen destroy and repair neighbourhoods

5 if x' can be accepted, then set $x := x'$

6 Update count τ_j and score π_j of heuristic pair j through the selected branch

7 **if** end of a segment **update** neighbourhood probabilities **for each** of the ALNS branches according to **Algorithm 3**

8 **if** $f(x) < f(x^*)$ **set** $x^* := x$

9 **Until stop criteria is met**

10 **Return** x^*

3.2 Data representation of the PDPTW

The OpenMP version of the outlined algorithm has been implemented purely in the C programming language. Both implementations use approximately the same data representation of the PDPTW. Our design makes heavy use of custom data structures to make the framework easier to work with and maintain. For the implementation of the PDPTW we do a separation at the topmost level where we separate the constant data from the variable data. The constant data includes all the data that is read from an external problem formulation as well as auxiliary clustering and relatedness metrics proposed by Shaw (1998) derived from this data during the initialization phase. The variable data includes all the problem instances and the auxiliary data structures that they require. A large data structure of arrays is used to represent the constant input data. These are the number of nodes, number of vehicles and number of calls. The calls data includes pickup and delivery nodes, cost of not transporting, size of shipment, and the lower- and upper-time windows for pickup and delivery. The vehicles data contains each vehicles origin node, starting time and capacity. Compatibility data is a binary matrix of each call and each vehicle, and it has the value true if a call is compatible with the vehicle and false otherwise. The cost matrix holds the cost of traveling from node $\boldsymbol{x}$ to node $\boldsymbol{y}$ with vehicle $\boldsymbol{i}$. The distance matrix holds the amount of time it takes to travel from node $\boldsymbol{x}$ to node $\boldsymbol{y}$ with vehicle $\boldsymbol{i}$. The wait times holds the service times for vehicle $\boldsymbol{i}$ at the pickup and delivery of call $\boldsymbol{j}$. And the toll cost holds information on the toll cost incurred when vehicle $\boldsymbol{i}$ picks up or delivers call $\boldsymbol{j}$.

The auxiliary clustering and relatedness metrics constant data includes a calls difficulty array which measures a calls relative difficulty compared to the other calls. A vehicle cluster array defines a fuzzy clustering of nearby calls around vehicle $\boldsymbol{i's}$ starting position. Transition clustering array defines a fuzzy clustering around a call's delivery node on calls by the proximity of their pickup node to the delivery node and remoteness of their delivery node to the pickup node. Similarity clustering defines a fuzzy clustering around each call to all other calls by a closeness in time windows, pickup and delivery locations, shipment size and vehicle compatibilities. Adjacency clustering defines a fuzzy clustering around each pair of pickup and delivery nodes such that calls with pickup node closest to $\boldsymbol{i}$ and delivery node closest to $\boldsymbol{j}$ has the greatest adjacency to pair $(\boldsymbol{i},\boldsymbol{j})$.

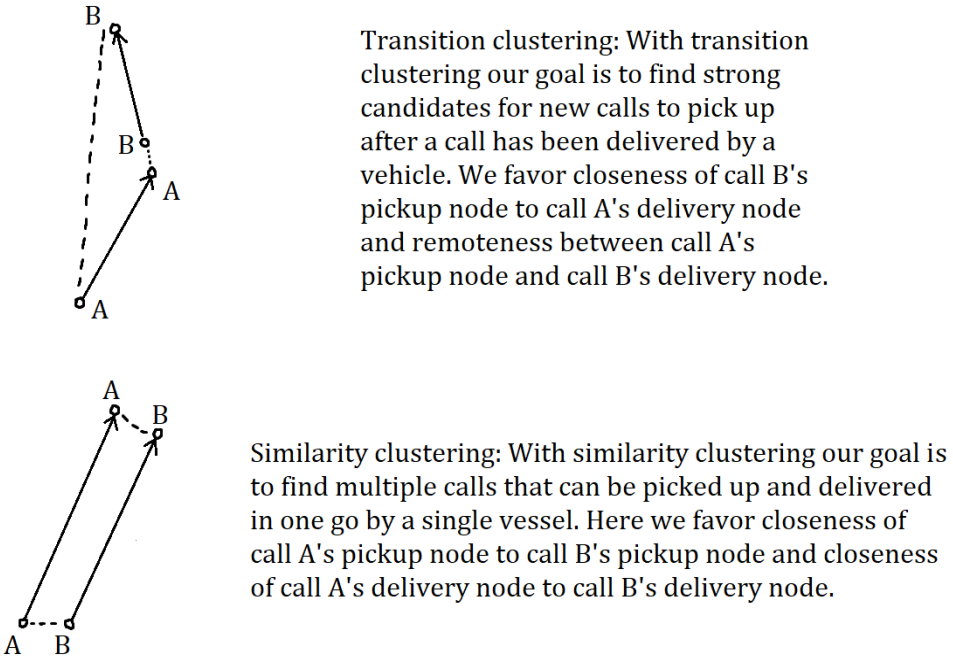


Figure 3.2: The general idea behind clustering of related calls.

The global variable data is shared among all threads. These include an array which is a hash of all seen objective values, the size of this array needs to be at least the cost of any reasonable objective value. A calls counter array counts the number of times call ***i*** has been perturbed, and a calls placement counter array counts the number of times call ***i*** has been inserted in vehicle ***j***. For increased efficiency, arrays are flattened to one dimension and accessed through pointer indexing functions. This data structure is shown in A1.

We now consider the implementation of the problem. An independent problem structure A2 is allocated for all threads since all threads are working on independent copies of the same problem and all perturbations on the underlying problem domain happens using this structure. The problem structure consist of several parts and the instance structure A3 is at the core of each problem as it holds a solution representation from one step to the next, this structure is made up of several components, the structure A4 is a wrapper for an array of A5 and holds information on the number of allocated A5 structures as well as a count of recorded possible insertion points in a current solution for some call ***i.*** This is useful in operations such as random insert and beam search where one needs to identify all possible insertion locations for a call in a current solution. In every instance there is an array of vehicle structures A6,

one for each vehicle as described in some external problem formulation and one additional for an imaginary outsource vehicle. Each vehicle structure holds information on the cost of the vehicles travel plan, the feasibility of the travel plan, the length of the travel plan and a first and a last index into an array of nodes A7 holding the vehicles first and last node.

Indeed, we opted to go for a doubly linked list for the solution representation in a similar fashion to the one proposed in Bach et al (2019). There are several reasons for this, first and foremost, it allows us to remove and insert calls in constant time, and although we are not able to determine in constant time which call is at a particular position in a vehicle, such knowledge is rarely used. We are however able to determine in constant time what vehicle a particular call belongs to, as we are maintaining a hash of this as we move along where each call number acts as an index in the hash giving back the vehicle index of that call or a $-\mathbf{1}$ if the call is not currently in the solution. We are also able to determine in constant time what node is representing a particular pickup/delivery of a call since the linked list is allocated as a contiguous array where the actual sequence of structures within the array never changes. The first array element holds the pickup node of the first call, the second element holds the delivery node of the first call, the third element holds the pickup node of the second call and so on. The order of the pickup and deliveries is instead controlled by indexing into the array by a previous node index and a next node index within each node structure.

The nodes from index $\boldsymbol{2n}$ to index $\boldsymbol{2n + 2(m + 1)}$ where $\boldsymbol{n}$ is the number of calls and $\boldsymbol{m}$ is the number of vehicles holds artificial start and end nodes for each vehicle including the imaginary vehicle. Each vehicle in the array of vehicles updates their first node index and last node index as required and keeps track of the number of calls assigned to it. The array of beam instances is just an array of regular PDPTW instances used within the beam search operator, and the beam solutions are just a lightweight temporary storage for the beam instances. The tour deviation structure A8 gives the total cost of a call in a current solution. There is an array of such structure within each PDPTW problem where the size of the array matches the total number of calls in the problem. A9 and A10 defines key value pairs used throughout the algorithm, they are particularly useful when sorting vehicles/calls with respect to certain metrics.

3.3 The general implementation of PALNS for the PDPTW

The design of our implementation follows carefully from the model shown in figure 1.1, separating the hyperheuristic and the metaheuristic frameworks. For the parallelization it follows in the footsteps of Bach et al (2019) in the implementation of a parallel perceptive ALNS. It is desired however to have more cooperation between threads compared to what is proposed in their paper, and to separate the size of the neighbourhood from the pairs of destroy and remove heuristics and instead take the number of calls to be perturbed by each pair as a parameter controlled by a secondary ALNS which will adaptively change the neighbourhood size probabilities of each heuristic pair over time. Assume the neighbourhood sizes used are **{0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50}** where the numbers are percentages of the total number of calls in some problem instance. Then for example, if the neighbourhood **0.4** is selected for some heuristic pair the agent will select between **35%** and **40%** of the total number of calls and pass that number as parameter to the heuristic pair.

Each thread has its own local PALNS agent and a local SA metaheuristic. The local PALNS agents needs to keep separate counts and scores of heuristics used through every conditional branch. In our PALNS implementation we are using the same perceptive branches mentioned in section 3.1. Previous action, improvement/not improvement, and seen/unseen objective value. Therefore each agent has to remember the previous action that it took, whether or not that action lead to an improvement and whether or not the resulting solution was unseen, it can be useful for it to keep track of how many times it has applied each different pair of heuristics, the number of iterations since a change in objective value, since an unseen objective value, since a previously seen objective, since an improvement was made and since a deterioration occurred.

A global PALNS agent is used such that the data from the local PALNS agents may be accumulated during the end of each segment and global operator probabilities for each branch calculated based on all recorded counts and scores through that branch, it is also interesting to keep track of how many times each heuristic has been used globally. The global PALNS agent may hold useful data such as the number of accumulative steps since a new global best solution was found, the total number of unique solutions found, the total number of wasted actions which are actions where the objective value does not change, the total number of times a new best solution has been found and the total number of times a previously seen solution was encountered. Furthermore, constant data common to all local agents such as function pointers into the pool of heuristic pairs one can select from, the rewards given for different types of observations, the weighting of locally calculated probabilities over globally calculated probabilities, reaction factor and segment size can naturally be held by the global agent.

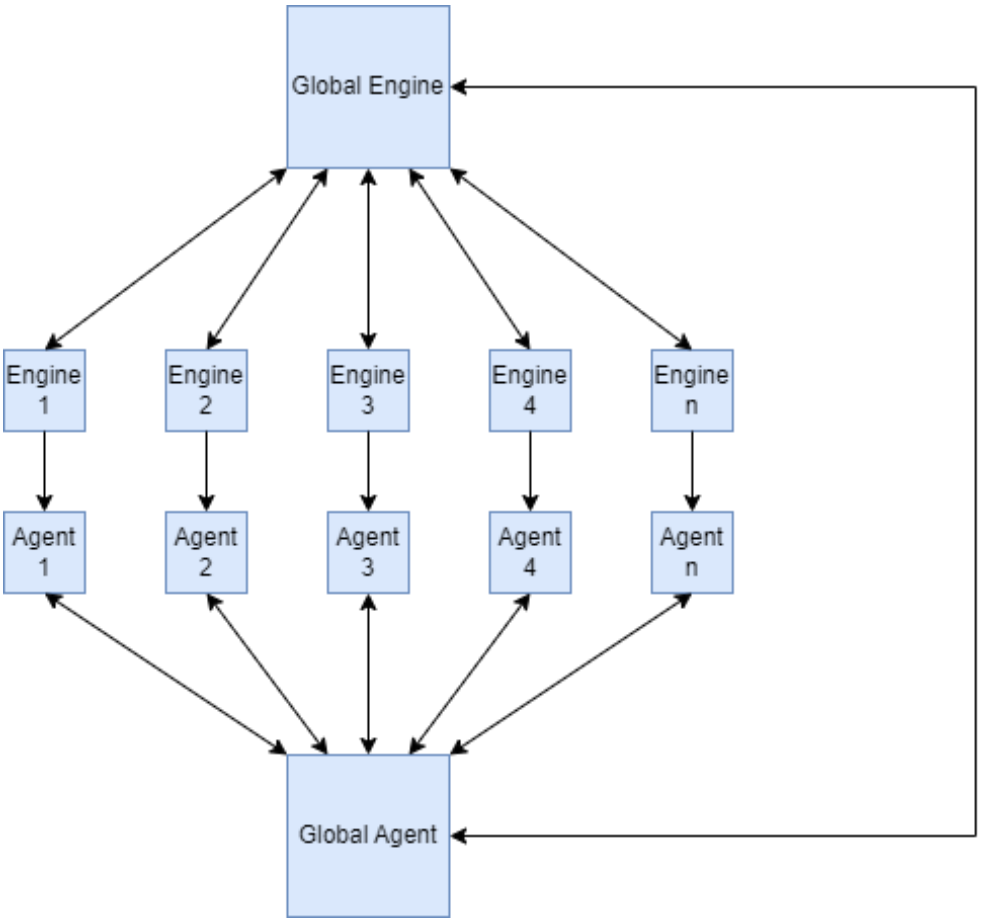


Figure 3.3: A flowchart of our overall PALNS - SA design philosophy

Each local SA engine has an independent temperature variable and controls the general "flow" of each individual problem instance. They are responsible for perturbing a local temporary solution, checking the metropolis acceptance criterion, and updating the incumbent solution in accordance with the acceptance criterion and keeping their best-known solution stored, in a similar fashion to a standard simulated annealing metaheuristic. Each engine communicates with its local PALNS agent, requesting what heuristic pair to apply to the problem instance and sends an observation back to the agent describing the result of the perturbation. The engines hold their own iteration counter and operates independently up until they reach the end of a segment at which point, they stall until every local engine has synchronized and information have been accumulated and shared among them before starting the next segment.

A global SA engine is at the top of all the local SA engines holding a reference to the underlying data of the problem being solved, a count of and a reference to all the local agents, and an array of key value structures that is used for sorting engines based on certain metrics that can be used for intelligent information sharing among them. Furthermore, constant data common to all SA engines is stored in the global engine, these include the number of warmup steps to be taken, the size of a segment, the start temperature, the end temperature, the cooling schedule and the targeted start and end probability of acceptance for an average deterioration calculated during warmup. Variables such as the global minimum objective found, the global maximum deterioration, the global minimum deterioration, an exponential moving average (EMA) of global improvements and an EMA of global deteriorations where the period used is the warmup size are held by the global agent. Figure 3.3 shows our general design philosophy.

3.4 Developing an OpenMP version of PALNS for the general PDPTW

For our OpenMP version of PALNS for the PDPTW we use a set of five removal heuristics and three insertion heuristics as well as two standalone heuristics. In total creating a pool of 17 heuristics to select from since every removal heuristic can be combined with one of the insertion heuristics and there are 15 such combinations. The tables below introduce the heuristics used.

Name	
Remove random	Removes calls randomly
Remove least frequent	Removes calls that have been removed least frequently
Remove similar	Selects a seed randomly and removes calls similar to the selected seed
Remove most expensive and candidates	Removes an expensive call and calls that can replace it at a low cost
Remove most expensive tour deviation	Removes some of the most expensive calls in a current solution

Table 3.1 List of destroy heuristics used in the OpenMP version:

Name	
Insert Beam Search	Insert each element in the best position using beam search
Insert by variance	Sorts the insertion order based on variance and inserts each element in the best possible position using beam search
Insert random	Inserts each element in a random feasible position

Table 3.2 List of repair heuristics used in the OpenMP version:

Name	
Insert single best	Steps through every call temporarily removing it and determines the cheapest new position for that call before reinserting it at its original position. Maintains a record of the best change seen and at the end executes that single change on the problem instance.
Rearrange all vehicles	Removes all the elements from each vehicle and inserts them back into the same vehicle using insert beam search rearrange.

Table 3.3 List of standalone heuristics used in the OpenMP version:

In this section, the insertion heuristics are explained in greater detail. The beam search operator also called the k-regret operator is a computationally expensive operator that requires two external parameters, a search width, and a beam width. Beam search has the advantage over greedy search in that it can look past local optimums so long as the "ridges are not too wide", the beam width and search width has something to say in this regard. It differs from greedy search in that it maintains a pool of promising solutions throughout the insertion process. This pool can contain up to beam width many solutions. The search width tells the beam search operator how many solutions it may keep from the expansion of each of the solutions in the pool when determining the best positions of the next call in the queue of calls to be inserted. In the beginning there is only one solution in the pool, namely the starting solution with some calls removed. At the beginning of the second round there can be at most search width many solutions in the pool. Once more calls get inserted, the number of solutions may exceed the capacity of the pool (beam width). Then the solutions are sorted, and the best ones remain in the pool. Wang et al (2017) reports good results with a stochastic beam search where some worse solutions can be placed in the pool probabilistically over better solutions for the integrated vehicle dispatching and container allocation problem. Although we do not use this implementation, we are able to achieve good solution diversification by keeping the search width significantly lower than the beam width imposing restrictions on the number of branches allowed per solution in the pool.

Insert by variance calculates a variance for each of the calls that have been removed from the solution. For each call, it determines the costs of all possible insertion points and sorts the insertion points and calculates a variance using a given sampling size. Finally, the calls are sorted based on the calculated variances and the calls with the highest variance should be first in the sorted array and inserted first since there are fewer good opportunities for these calls. The sorted calls may then be dispatched to for example greedy insertion or beam search. Insert random determines all feasible insertion points for each call in the queue and randomly selects and inserts the calls in a feasible position.

In the OpenMP version of PALNS for PDPTW the design philosophy is simple. Each thread represents an independent PALNS instance and synchronization happens at the end of each segment where information is shared, and search probabilities updated.

3.5 Developing a CUDA version of PALNS for the general PDPTW

We will start by giving a brief simplified introduction to an Nvidia GPU and do a simple comparison between the CPU and GPU. All information in this introduction is derived from the CUDA programming guide, and for the reader interested in an in-depth introduction to the CUDA programming model, we refer to the CUDA programming guide.

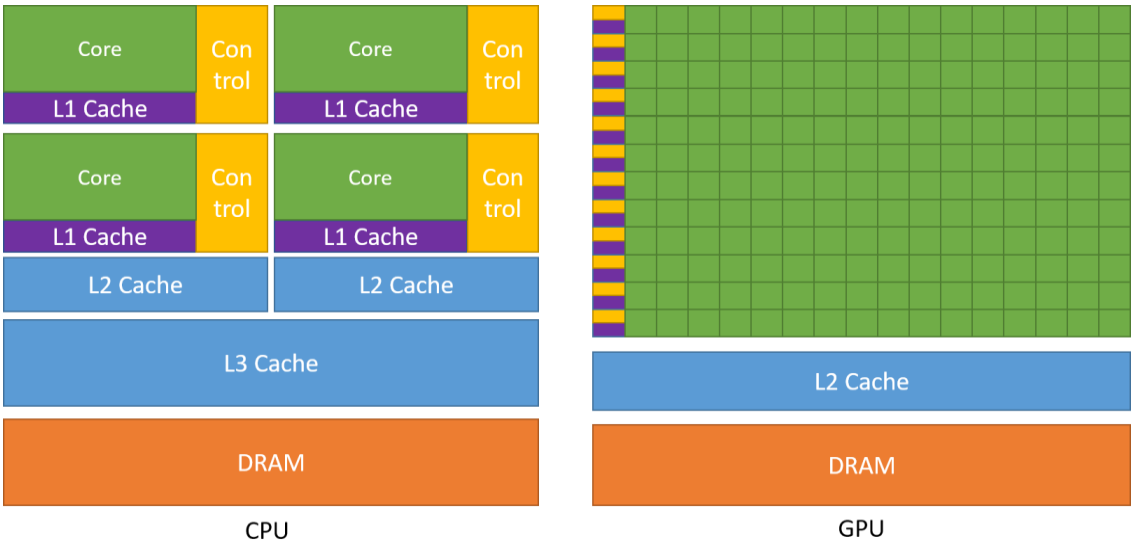


Figure: 3.4: A simplified comparison of the CPU and GPU architecture characteristics, figure borrowed from the official CUDA programming guide.

For an Nvidia GPU All instructions are executed by warps, which are currently groups of 32 threads. Each CUDA warp executes in lock steps meaning that all threads in a warp executes the same instruction at the same time or are "masked out". Since every thread in a warp shares the same program counter, any diverging conditional branch within a warp must be serialized. A streaming multi-processor (SM) is responsible for scheduling the warps for execution and an SM can schedule multiple warps simultaneously depending on GPU model and intended shared memory usage. There are quite a few total numbers of SMs depending on the GPU model used. The single core frequency of the GPU is significantly lower than that of a comparable CPU, meaning that if there is only a single task or the number of tasks to be done are less than or equal to the number of CPU cores, then the CPU should always win the race

in terms of execution speed. The CPU is optimized for serial tasks and hides latency by prefetching of data and low memory and operational latency compared to the GPU. The GPU is optimized for parallel tasks and hides latency by rapid context switching between warps, high throughput, and memory bandwidth.

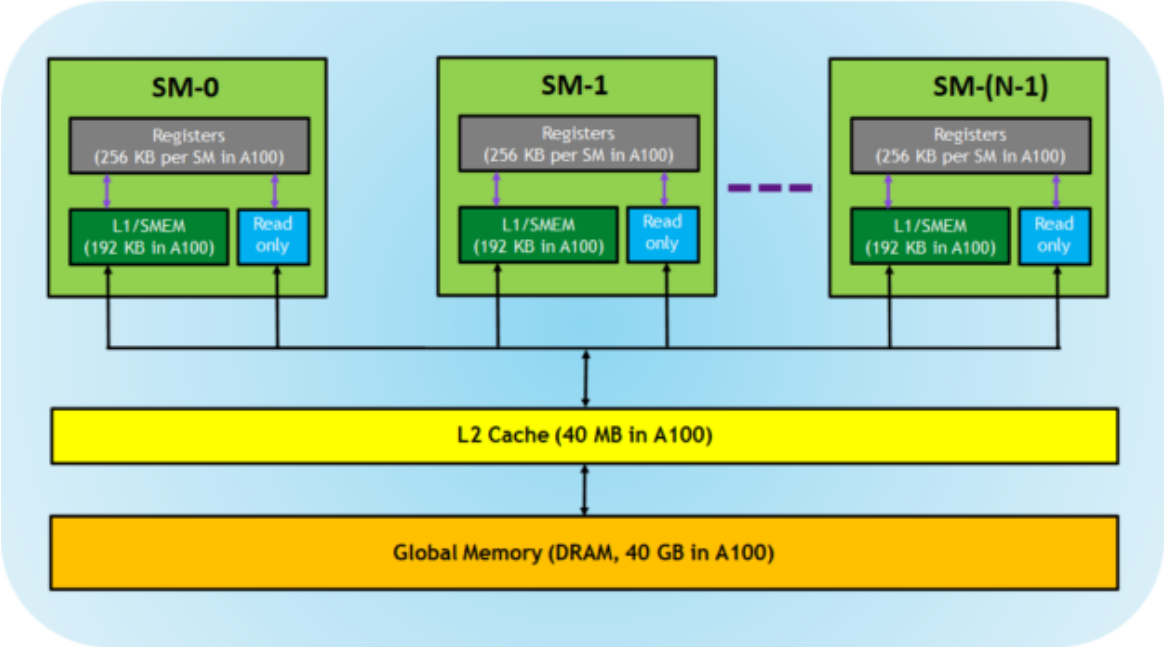


Figure 3.5: Overview of memory hierarchy in a GPU borrowed from Gupta (2020).

Each SM have their own L1 cache which is essentially shared memory that can be allocated to scheduled blocks, which in most models can contain at most 32 warps or 1024 threads. There is also a small constant read only cache per SM and a large register pool which can be used by each thread scheduled by an SM, the registers are private to each thread that occupies them. There is a larger L2 cache on the order of several megabytes that is shared among SMs. Outside L2 cache sits the global memory on the order of several gigabytes, it has the highest latency but also the highest capacity.

To summarize, for a comparable CPU and GPU there is a significantly smaller amount of cache per CUDA core compared to per CPU core. A single CPU core is more powerful than a single GPU core. The global memory latency is higher for the GPU compared to the CPU; the memory bandwidth is higher for the GPU than the CPU. There is no automated prefetching mechanism in hardware for the GPU, instead this

must be explicitly incorporated in software by moving data into registers or shared memory ahead of time.

When designing a CUDA version of PALNS for the PDPTW it is natural to have one block represent a problem instance. There are two main options to consider, a large block size with many warps could be used imposing restrictions on the number of concurrent instances one would be able to run but at the same time it would allow the usage of more heavyweight parallelizable heuristics without the penalty of greatly increased computational time. The second option is to assign exactly one warp per block and use a simpler set of heuristics but with a vastly greater number of concurrent solution instances. After careful considerations the second option was selected. This is mainly because we wanted to investigate how many active agents with a simple set of heuristics can be used to effectively explore large areas of an underlying search space compares to a fewer number of search agents but with a more powerful set of heuristics in our CPU implementation. Thus naturally, our focus in comparing the two implementations is on solution quality over speed of execution.

To achieve a reasonably efficient CUDA implementation of PALNS, an efficient parallel reduction mechanism for accumulating the data from the many parallel agents at the end of each segment is essential. At the end of each segment, every local agent calculates new local probabilities using the kernel A11 with a grid size equal to the number of parallel agents and a block size of 1024. There is a block for every parallel agent and each thread within each block independently updates the heuristic probabilities within an ALNS branch as shown in figure 3.1 belonging to that agent.

The next step is a reduction where all local heuristic usage counts as well as every count/score through every branch are accumulated and transferred to their corresponding location in the global agent, before all the local counts/scores are reset to zero. The kernel A12 is at the core of this step, and it is called with a grid size equal to the total number of count/score targets and a block size of 1024. Essentially the goal is to have each block be responsible for one PALNS end point and to use the threads within each block to sum all counts and scores from all the agents in logarithmic time via a reduction to a shared memory location. The thread with the lowest rank within

each block then updates the corresponding end point within the global agent with the total sum of counts and scores at that end point. Finally, once all the counts and scores have been accumulated in the global agent, global probabilities are updated by A13 with a grid size of one and a block size of 1024 in a similar fashion to the one explained for the local updates.

In our CUDA version of PALNS for the PDPTW we use a slightly different set of heuristics. We use 5 destroy heuristics and 3 insertion heuristics and no standalone heuristics yielding a total of 15 heuristic pairs to select from. The tables below show the heuristics used:

Name	
Remove random	Removes calls randomly
Remove least frequent	Removes calls that have been removed least frequently
Remove similar	Selects a seed randomly and removes calls similar to the selected seed
Remove most expensive and candidates	Removes an expensive call and calls that can replace it at a low cost
Remove most expensive tour deviation	Removes some of the most expensive calls in a current solution

Table 3.4: List of destroy heuristics used in the CUDA version:

Name	
Insert greedy stochastic	Insert each element at the one of the best possible positions in the order
Insert by variance	Sorts the insertion order based on variance and inserts each element in the best possible position using greedy insert
Insert random	Inserts each element in a random feasible position

Table 3.5: List of repair heuristics used in the CUDA version:

Chapter 4

Experimental setup

In this chapter we explain how the testing of our algorithms was conducted, including the nature and origin of the benchmark instances used, details on hardware and the parameters used for the algorithms.

4.1 Benchmarks

In our experiments we use two distinct sets of benchmarks, namely the benchmarks from Hematti al (2014) and the well-known and well-studied benchmarks from Li & Lim (2001). These two benchmark sets differ not only in the format of their external problem formulations but also in the type of PDPTW they represent. The differences are illustrated by the following table:

Li & Lim (2001)	**Hematti et al (2014)**
Homogeneous vehicle fleet, all properties of the vehicles are identical.	Heterogeneous vehicle fleet where some vehicles may be incompatible with certain requests. Each vehicle's capacity, travel speed and travel cost between pairs of nodes, and loading times and toll costs at request locations may differ.
Vehicles are centralised at a depot.	Decentralised vehicle fleet.
Each vehicle must start and end at the depot.	Each vehicle starts at individual starting nodes and starting times and may end on any delivery node.
The travel costs match the travel time.	The travel costs may differ from the travel time.
There are no toll costs at handling locations.	There may be toll costs at handling locations.
Requests may not be outsourced	Requests may be outsourced at spot charter rates.
Dual objective: Minimizing the number of vehicles required has priority over minimizing the sum of distances of all the routes.	Single objective: To minimize the total cost consisting of the sum of travel costs, toll costs and outsourcing costs.

Table 4.1: Comparison of the Li & Lim (2001) and the Hematti et al (2014) sets of PDPTW benchmark instances.

The Hematti et al (2014) benchmark instances originate from an instance generator they developed for the PDPTW in the context of ship routing and scheduling. Their goal was to generate realistic problems with regards to the real-world ship routing and scheduling situation. They have a variety of input settings to their generator ranging from port and ship types to market conditions. These instances can be found at: https://home.himolde.no/hvattum/benchmarks/. The Li & Lim (2001) benchmark instances are based on work by Solomon (1987) who created several benchmarking datasets for the DCVRP. In some datasets pure uniform randomness were used to generate the location of the request nodes while others used uniform randomness within clusters to generate the locations. Furthermore, the benchmark instances differ in schedule time horizons and vehicle capacities. Li & Lim (2001) generated PDPTW instances from these Solomon (1987) instances by randomly pairing customer locations within routes in solutions they obtained from their heuristic approach for DCVRP. These instances can be found at https://www.sintef.no/projectweb/top/pdptw/

4.2 Extending our OpenMP PALNS framework to the Li & Lim (2001) benchmark instances

To apply our OpenMP PALNS framework to the Li & Lim (2001) benchmark instances some modifications were required. In particular, the code responsible for loading a problem from a text file had to be modified to match the new format. The similarity of the problems allowed us to reuse most of our PDPTW data structures outlined in section 3.2. The main difference in the two implementations is at the higher level. Our first attempt was a single stage PALNS approach using an artificial route activation cost initially distributed evenly over the requests in the route, such that the requests in a route with few requests had a higher cost compared to the requests in a route with more requests driving the search into solutions with fewer but longer routes. During the search, we gradually rolled back the cost from the requests onto the route itself allowing the search to focus on rearrangements of requests within the individual routes. However, as previously referenced work has established, confirmed by our empirical observations, such an approach is not competitive with the state-of-the-art guided ejection search (Nagata & Kobayashi, 2010) for the objective of route elimination. Therefore, we use a two-stage approach with AGES for route elimination in stage one, followed by the regular PALNS for route cost minimization in stage two.

Our implementation of AGES is similar to that of Curtois et al (2018). In particular, the previously explained value of $\boldsymbol{k}$ is increased in steps starting at zero with a maximum value of two. Unlike Curtois et al (2018) we find that sorting the requests to be ejected heuristically based on previous difficulty of insertion leads to unfavourable results in terms of repetitive behaviour, therefore we use uniform randomness to determine the order in which requests are ejected. For improved efficiency we use a memoization technique where for the case of $\boldsymbol{k = 1}$ we store whether the ejection of pair $\boldsymbol{(i, j)}$ allowed the insertion of the unassigned request currently being processed (and obviously not the reinsertion of the ejected pair somewhere else). This is useful as when considering the case of $\boldsymbol{k = 2}$ we may skip combinations of two pairs placed in different routes that did not allow the insertion of the unassigned request since they cannot possibly allow the insertion when removed together. Hence, for considering two pairs for ejection we may require without loss of efficacy and with some gain in efficiency that either at least one of the two pairs must have made space for the unassigned request, or both ejected pairs must be from the same route. Suppose a value of $\boldsymbol{1}$ is given to a pair that allows the insertion of the unassigned request, a value of $\boldsymbol{-1}$ is given to a pair that both does not allow the insertion of the unassigned request and is not movable to some new route and a value of $\boldsymbol{0}$ is given otherwise. Let $\boldsymbol{q1}$ be the selected unassigned request and let $\boldsymbol{e1}$ and $\boldsymbol{e2}$ be the ejected pairs, then the following algorithm shows the flow of an insertion attempt with $\boldsymbol{k = 2}$:

Algorithm 6: GES insertion for $\boldsymbol{k = 2}$

```
1   if route(e1) equals route(e2) do
2       if insertion of q1 in route(e1) is successful
3           Try insertion of e1 and e2 in route(q1)    if success return
4           if f(e2) = 0 Try insert e1 in route(q1) and e2 in some other route
            if success return      else if insert e1 in route(q1) success then f(e2) = -1
5           if f(e1) = 0 Try insert e2 in route(q1) and e1 in some other route
            if success return      else if insert e2 in route(q1) success then f(e1) = -1
6           if f(e1) = 0 and f(e2) = 0 Try insert both e1 and e2 in some route, if success return
            else if insert e1 fail then f(e1) = -1, if insert e2 fail then f(e2) = -1
7       end if
8   else do twice
8       if f(e1) = 1 then insert q1 in route(e1)
9           Try insert e1 and e2 in route(e2)                                  if success return
10          Try insert e1 in route(e2) and e2 in route(e1)                     if success return
11          if f(e2) = 0 Try insert e1 in route(e2) and e2 in some other route  if success return
12      Swap e1 and e2
12  return failed attempt
```

We use the perturbation moves of swapping requests from different routes and attempting the insertion of a request from one route into some other route if GES insertion is unsuccessful in accordance with the original GES by Nagata & Kobayashi (2010). We find however that these simple perturbative moves do not always work effectively, and that sometimes more drastic measures are required to move GES out of a local optimum. We therefore propose to sometimes remove some random or expensive calls from the routes and move them back into the request bank after prolonged periods of non-improving GES. Our termination criterion for phase one is adaptive, and either one of a hundred thousand iterations without improvement or twenty thousand iterations with no cleared routes and more than three requests in the request bank. For the second stage the best feasible solution found by GES, meaning the final solution where a request bank was cleared is used as the starting solution for PALNS - SA.

4.3 Hardware and algorithmic parameters

All our OpenMP benchmark results comes from an Intel i9-7900x CPU. This CPU has 10 cores and 20 threads and may run a total of 20 simultaneous agents. For all our OpenMP experiments on the Hemmati et al (2014) benchmark instances a beam width of 30 and a search width of 6 is used for the beam search heuristic. All the CUDA benchmark results comes from an Nvidia GeForce GTX 1080 Ti and we use 512 agents for all our experiments with this implementation. We perform ten thousand iterations for all our runs on the Hemmati et al (2014) benchmark instances, to get the true number of iterations, this is multiplied by the number of parallel agents used, hence for the OpenMP implementation the actual number of iterations is two hundred thousand and for the CUDA implementation 5.12 million. A total of one hundred segments and a segment size of one hundred is used for the OpenMP experiments and one thousand segments and a segment size of ten is used for the CUDA experiments resulting in a sample size of 2000 and 5120 in each segment respectively. The following reward scheme is used in all our experiments:

$$R = \begin{cases} \mathbf{3.0} : \textit{New Global Best Solution} \\ \mathbf{2.0} : \textit{Unseen Improved Solution} \\ \mathbf{1.0} : \textit{Unseen Solution XOR Improvement} \end{cases}$$

Furthermore, the unseen solution reward may be increased up to $\mathbf{3.0}$ depending on how long since an unseen objective was found, and the improvement reward is reduced based on how many times the obtained objective value has been seen previously according to the following formula: $\frac{\boldsymbol{Reward_{Improvement}}}{(\mathbf{1.0} + \boldsymbol{times\_seen})}$. The expectation is that these changes to the reward scheme will help in escaping locally optimal areas. A reaction factor of 0.1 is used in all our experiments, both for the pool of heuristics and the neighbourhood sizes. Local weight is 0.1 for the CUDA implementation and 0.0 for the OpenMP implementations. The PALNS branching scheme is the same as the one discussed in section 3.1. Every $\boldsymbol{n}$ segment we replace the worst performing agent's solution by the best solution of the best performing agent to prevent agents from putting too much time into less promising areas of a search space.

For the Li & Lim (2001) experiments we perform an AGES in stage one with a maximum of one hundred thousand iterations without sizable improvement according to the adaptive scheme we outlined in section 4.2. After every 500 application of AGES 25% of the calls, either selected randomly or by their expensiveness in the current solution gets removed from the routes and moved into the request bank to prevent AGES from stalling. As explained earlier the best feasible solution found by AGES becomes the starting solution for all PALNS - SA parallel agents in stage two. A segment size of 125 and a total of 800 segments are used for stage 2 resulting in a total of 100,000 iterations. Thus, the total number of iterations performed in stage two is 2 million as there are 20 parallel agents. We use all the previous destroy heuristics except for *Remove most expensive and candidates* which is replaced by a heuristic that removes calls that are similar to one of the most expensive calls in a current solution. We also replace the *Insert by variance* heuristic by another beam search heuristic, hence we use two beam search heuristics, one shallow beam search with a search width of 5 and a beam width of 16 and a deep beam search with a search width of 3 and a beam width of 27. Finally, the standalone *Rearrange all vehicles* is replaced by the GES perturbative swap and insert heuristics. All problem instances are run 5 times and we consider a broad number of interesting metrics such as how the optimality/best known gaps widen for the implementations when the number of calls and number of vehicles increases. We compare standard ALNS to PALNS for different instance sizes and the average objectives at different stages of a search between ALNS and PALNS.

Chapter 5

Results

Here our main results on the benchmark datasets described in the previous chapter are presented. In the first few sections, we focus on the primary benchmark results on the Hemmati et al (2014) instances and in the final sections we step through some of the Li & Lim (2001) benchmark results. We also provide some figures illustrating solutions to some of the Li & Lim benchmark instances where the goal is to review the difference between the three main categories of Li & Lim instances, namely clustered request locations (lc*), uniformly random request locations (lr*) and the mixed category of clustered and random request locations (lcr*).

#C	#V	OpenMP - PALNS (2023)		OpenMP - ALNS (2023)		Hemmati et al (2014)	
		Average Best Gap %	Seconds	Average Best Gap %	Seconds	Average Best Gap %	Seconds
7	3	0.0	0.773	0.0	0.784	0.0	1.59
10	3	0.0	1.645	0.0	1.706	0.0	2.54
15	4	0.0	7.235	0.0	7.849	0.0	5.28
18	5	0.0	11.606	0.0	12.491	0.0	7.54
22	6	0.0	6.817	0.0	7.198	0.0	11.50
23	13	0.0	10.030	0.0	10.152	0.0	14.95
30	6	0.0	13.313	0.0	13.017	0.123	22.61
35	7	0.0	21.987	0.0	20.330	0.243	32.34
60	13	0.040	37.789	0.062	33.349	0.770	120.89
80	20	0.138	60.511	0.218	51.868	1.501	248.08
100	30	0.140	132.231	0.245	118.921	1.685	466.46
130	40	0.493	245.600	0.575	222.822	1.823	1016.78

Table 5.1: Average results for the Hemmati et al (2014) short sea shipping instances with mixed cargo sizes.

#C	#V	OpenMP - PALNS (2023)		OpenMP - ALNS (2023)		Hemmati et al (2014)	
		Average Best Gap %	Seconds	Average Best Gap %	Seconds	Average Best Gap %	Seconds
8	3	0.0	0.888	0.0	0.901	0.0	1.84
11	4	0.0	3.477	0.0	3.648	0.0	2.85
13	5	0.0	4.128	0.0	4.387	0.0	4.10
16	6	0.0	2.091	0.0	2.229	0.0	6.14
17	13	0.0	9.080	0.0	9.360	0.0	8.59
20	6	0.0	5.397	0.0	5.869	0.0	10.71
25	7	0.0	16.543	0.0	16.017	0.0	16.47
35	13	0.0	35.794	0.0	31.643	0.0	35.91
50	20	0.005	34.886	0.015	32.047	0.166	83.08
70	30	0.086	55.630	0.135	51.132	0.584	210.66
90	40	0.139	106.613	0.131	97.872	0.970	418.57
100	50	0.167	166.241	0.171	155.201	0.604	587.52

Table 5.2: Average results for the Hemmati et al (2014) short sea shipping instances with full load cargoes.

		OpenMP - PALNS (2023)		OpenMP - ALNS (2023)		Hemmati et al (2014)	
#C	#V	Average Best Gap %	Seconds	Average Best Gap %	Seconds	Average Best Gap %	Seconds
7	3	0.0	0.784	0.0	0.808	0.0	1.64
10	3	0.0	1.264	0.0	1.319	0.0	2.40
15	4	0.0	2.170	0.0	2.206	0.0	5.07
18	5	0.0	9.684	0.0	10.582	0.0	7.58
22	6	0.0	5.766	0.0	6.091	0.0	11.29
23	13	0.0	7.406	0.0	7.388	0.0	14.59
30	6	0.0	14.159	0.0	14.059	0.240	22.06
35	7	0.0	24.955	0.0	22.454	0.094	30.89
60	13	0.023	27.162	0.046	23.045	1.276	115.32
80	20	0.128	55.612	0.185	47.253	1.230	255.76
100	30	0.485	108.240	0.531	96.755	2.102	479.91
130	40	1.152	209.398	1.380	187.255	3.556	1048.17

Table 5.3: Average results for the Hemmati et al (2014) deep sea shipping instances with mixed cargo sizes.

		OpenMP - PALNS (2023)		OpenMP - ALNS (2023)		Hemmati et al (2014)	
#C	#V	Average Best Gap %	Seconds	Average Best Gap %	Seconds	Average Best Gap %	Seconds
8	3	0.0	0.781	0.0	0.786	0.0	1.75
11	4	0.0	1.273	0.0	1.269	0.0	2.74
13	5	0.0	4.495	0.0	4.754	0.0	3.79
16	6	0.0	7.119	0.0	7.575	0.0	5.91
17	13	0.0	4.582	0.0	4.723	0.0	8.00
20	6	0.0	6.465	0.0	6.712	0.0	9.82
25	7	0.0	10.974	0.0	10.919	0.0	15.87
35	13	0.0	19.858	0.0	17.584	0.002	35.29
50	20	0.0	34.319	0.001	30.510	0.137	84.81
70	30	0.020	54.012	0.023	50.449	0.298	212.74
90	40	0.036	103.766	0.065	98.207	0.497	420.68
100	50	0.060	144.777	0.072	139.198	0.463	591.11

Table 5.4: Average results for the Hemmati et al (2014) deep sea shipping instances with full load cargoes.

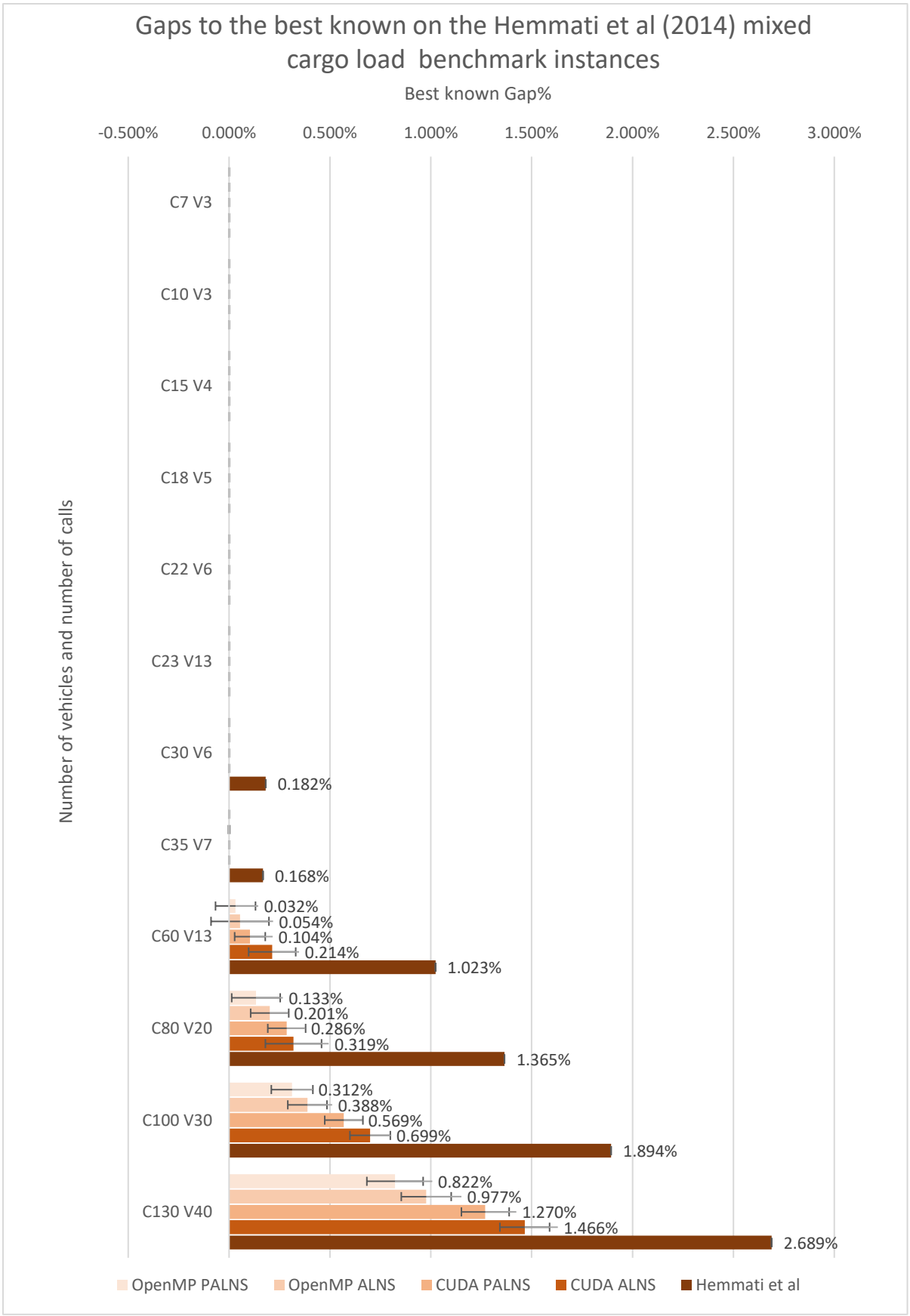


Figure 5.1: Results demonstrating the superiority of our PALNS implementation over ALNS on mixed load cargoes in the presence of large sample sizes.

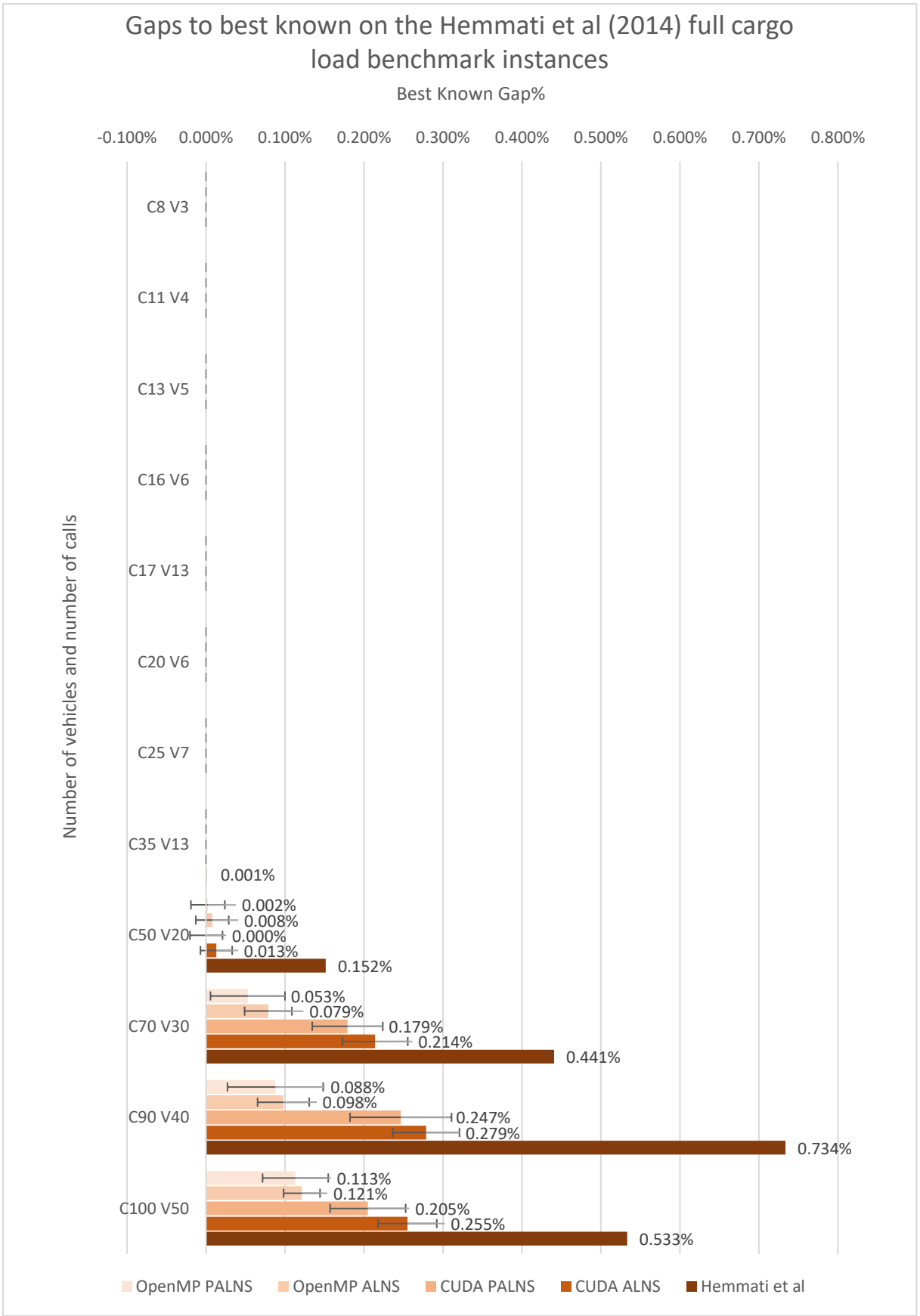


Figure 5.2: Results demonstrating the superiority of our PALNS implementation over ALNS on full load cargoes in the presence of large sample sizes.

The above plots demonstrate the superiority of PALNS over ALNS on all the Hemmati et al (2014) PDPTW benchmark instances. The Hemmati et al (2014) gaps are based on the reported results in their scientific paper from 2014 where they ran 25,000 iterations of ALNS on each instance ten times. We see that our OpenMP implementation with a more advanced set of heuristics is able to outperform our CUDA implementation with a simpler set of heuristics but many more simultaneous agents and overall number of iterations on all instances except for the C50V20 instances where CUDA PALNS outperforms our OpenMP implementation. The error bars are derived from the average standard deviation of the 5 runs of each instance.

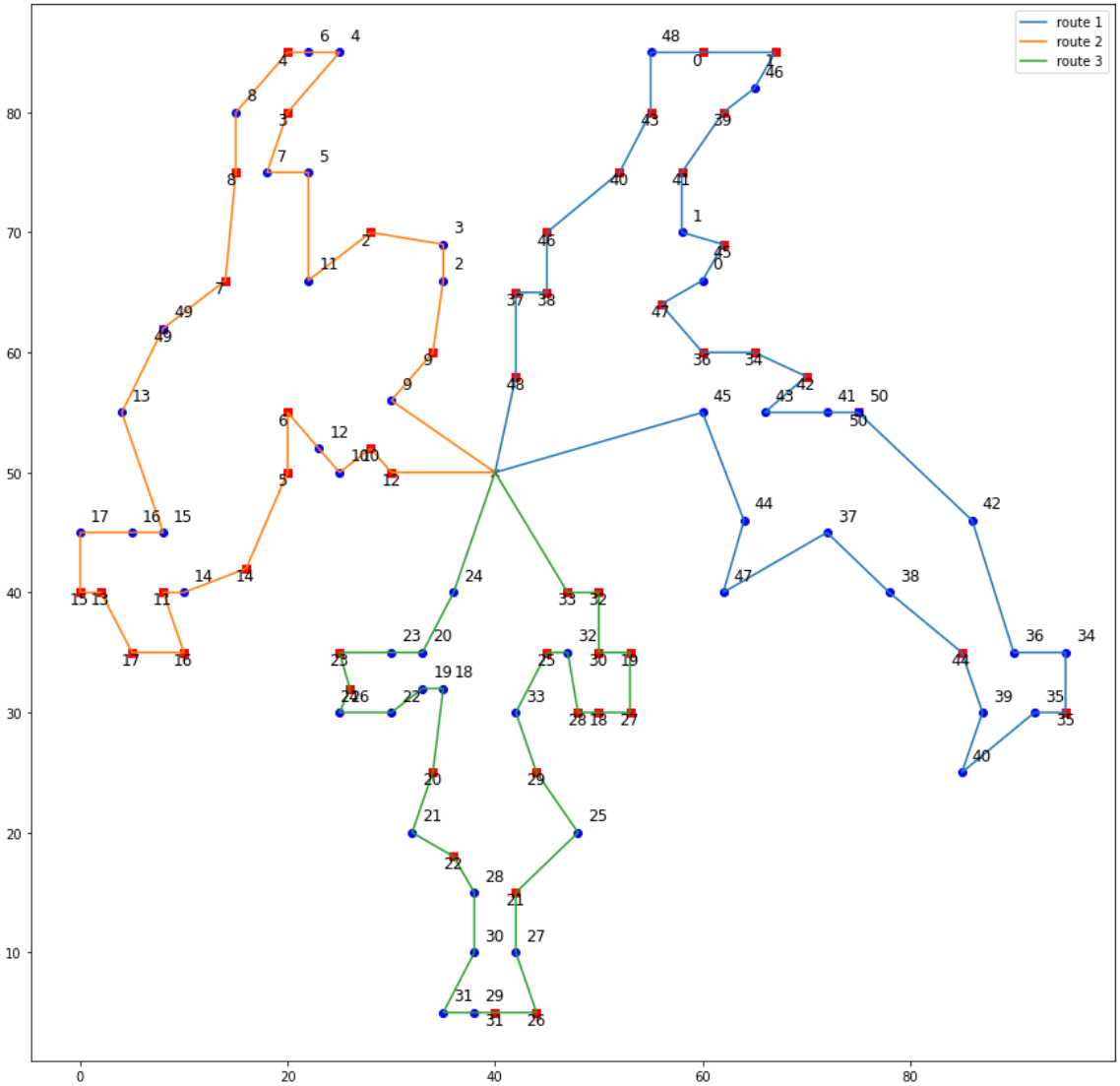


Figure 5.3: This figure shows the optimal route schedule at a total cost of **590.60** for the Li & Lim **lc204** benchmark instance where there are a total of 50 pickups and 50 deliveries. The red squares are the pickups, and the blue circles are the deliveries, and the depot is the purple cross in the centre. The identification number at the pickup matches the number at the delivery. Source: The author.

Instance	Best Known		PALNS (2023)					Li & Lim (2001)		
	td	nv	td^{best}	nv^{best}	td^{avg}	nv^{avg}	ct	td	nv	ct
lc101	828.94	10	828.94	10	828.94	10	38	828.94	10	33
lc102	828.94	10	828.94	10	828.94	10	38	828.94	10	71
lc103	1035.35	9	1035.35	**9**	1035.35	9	49	827.86	10	191
lc104	860.01	9	**860.01**	9	860.01	9	82	861.95	9	1254
lc105	828.94	10	828.94	10	828.94	10	29	828.94	10	47
lc106	828.94	10	828.94	10	828.94	10	33	828.94	10	43
lc107	828.94	10	828.94	10	828.94	10	34	828.94	10	54
lc108	826.44	10	826.44	10	826.44	10	45	826.44	10	82
lc109	1000.6	9	1000.6	**9**	1000.6	9	55	827.82	10	255
lc201	591.56	3	591.56	3	591.56	3	33	591.56	3	27
lc202	591.56	3	591.56	3	591.56	3	65	591.56	3	94
lc203	591.17	3	591.17	3	591.17	3	112	591.17	3	145
lc204	590.6	3	**590.6**	3	590.6	3	330	591.17	3	746
lc205	588.88	3	588.88	3	588.88	3	68	588.88	3	190
lc206	588.49	3	588.49	3	588.49	3	85	588.49	3	88
lc207	588.29	3	588.29	3	588.29	3	99	588.29	3	102
lc208	588.32	3	588.32	3	588.32	3	105	588.32	3	178
lr101	1650.80	19	1650.80	19	1650.80	19	45	1650.80	19	87
lr102	1487.57	17	1487.57	17	1487.57	17	60	1487.57	17	1168
lr103	1292.68	13	1292.68	13	1292.68	13	53	1292.68	13	169
lr104	1013.39	9	1013.39	9	1013.39	9	51	1013.39	9	459
lr105	1377.11	14	1377.11	14	1377.11	14	46	1377.11	14	69
lr106	1252.62	12	1252.62	12	1252.62	12	45	1252.62	12	87
lr107	1111.31	10	1111.31	10	1111.31	10	49	1111.31	10	287
lr108	968.97	9	968.97	9	968.97	9	51	968.97	9	415
lr109	1208.96	11	**1208.96**	11	1208.96	11	45	1239.96	11	348
lr110	1159.35	10	1159.35	10	1159.35	10	44	1159.35	10	547
lr111	1108.9	10	1108.9	10	1108.9	10	52	1108.9	10	179
lr112	1003.77	9	1003.77	9	1003.77	9	54	1003.77	9	638
lr201	1253.23	4	**1253.23**	4	1253.23	4	57	1263.84	4	193
lr202	1197.67	3	1197.67	3	1197.67	3	76	1197.67	3	885
lr203	949.4	3	949.4	3	949.4	3	181	949.4	3	1950
lr204	849.05	2	849.05	2	849.05	2	420	849.05	2	2655
lr205	1054.02	3	1054.02	3	1054.02	3	92	1054.02	3	585
lr206	931.63	3	931.63	3	931.63	3	157	931.63	3	747
lr207	903.06	2	903.06	2	903.06	2	356	903.06	2	1594
lr208	734.85	2	734.85	2	734.85	2	809	734.85	2	3572
lr209	930.59	3	**930.59**	3	930.59	3	112	937.05	3	2773
lr210	964.22	3	964.22	3	964.22	3	146	964.22	3	1482
lr211	911.52	2	**911.52**	2	911.52	2	190	927.8	2	4204
lrc101	1708.8	14	1708.8	14	1708.8	14	37	1708.8	14	119
lrc102	1558.07	12	**1558.07**	**12**	1558.07	12	40	1563.55	13	152
lrc103	1258.74	11	1258.74	11	1258.74	11	47	1258.74	11	175
lrc104	1128.4	10	1128.4	10	1128.4	10	57	1128.4	10	202
lrc105	1637.62	13	1637.62	13	1637.62	13	41	1637.62	13	179
lrc106	1424.73	11	**1424.73**	11	1424.73	11	39	1425.53	11	459
lrc107	1230.14	11	**1230.14**	11	1230.14	11	46	1230.15	11	154
lrc108	1147.43	10	**1147.43**	10	1147.43	10	47	1147.97	10	650
lrc201	1406.94	4	**1406.94**	4	1406.94	4	55	1468.96	4	266
lrc202	1374.27	3	1374.27	3	1374.27	3	67	1374.27	3	987
lrc203	1089.07	3	1089.07	3	1089.07	3	114	1089.07	3	1605
lrc204	818.66	3	**818.66**	3	818.66	3	292	827.78	3	3634
lrc205	1302.2	4	1302.2	4	1302.2	4	69	1302.2	4	639
lrc206	1159.03	3	**1159.03**	3	1159.03	3	76	1162.91	3	445
lrc207	1062.05	3	**1062.05**	3	1062.05	3	96	1424.6	3	607
lrc208	852.76	3	852.76	3	852.76	3	150	852.76	3	4106

Table 5.5: Li & Lim PDP-100 benchmark results. (td) total distance, (nv) number of vehicles, (ct) compute time used in seconds.

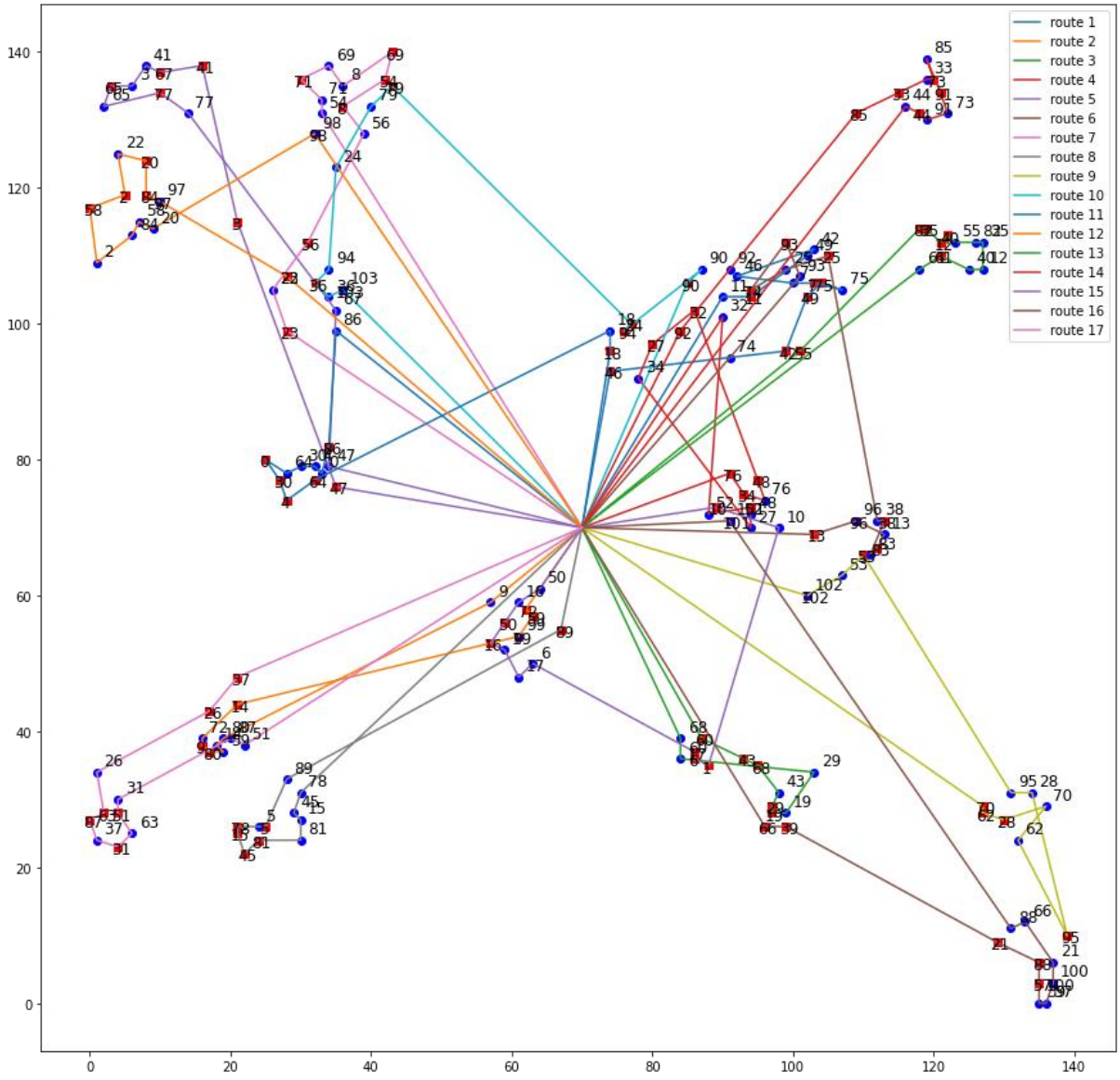


Figure 5.4: Example of the Li & Lim clustered request location benchmark instances. The figure shows the best-known route schedule at a total cost of **2942.13** for the Li & Lim **lc1_2_10** benchmark instance where there are a total of 100 pickups and 100 deliveries. The red squares are the pickups, and the blue circles are the deliveries, and the depot is the purple cross in the centre. The identification number at the pickup matches the number at the delivery. Source: The author.

Instance	Best Known		PALNS (2023)			ALNS (2023)			Curtois et al (2018)		
	td	nv	td	nv	ct	td	nv	ct	td	nv	ct
lc1_2_1	2704.57	20	2704.57	20	245	2704.57	20	305	2704.57	20	900
lc1_2_2	2764.56	19	2764.56	19	826	2764.56	19	795	2764.56	19	900
lc1_2_3	3127.78	17	**3127.78**	**17**	297	3127.78	17	471	3128.61	17	900
lc1_2_4	2693.41	17	2693.41	17	2022	2693.41	17	1917	2693.41	17	900
lc1_2_5	2702.05	20	2702.05	20	372	2702.05	20	491	2702.05	20	900
lc1_2_6	2701.04	20	2701.04	20	401	2701.04	20	547	2701.04	20	900
lc1_2_7	2701.04	20	2701.04	20	422	2701.04	20	602	2701.04	20	900
lc1_2_8	3354.27	19	**3379.97**	**19**	293	3404.95	19	300	3397.65	19	900
lc1_2_9	2724.24	18	2724.24	18	1008	2724.24	18	1040	2724.24	18	900
lc1_2_10	2942.13	17	**2942.13**	**17**	261	2942.13	17	307	2947.00	17	900
lc2_2_1	1931.44	6	1931.44	6	491	1931.44	6	492	1931.44	6	900
lc2_2_2	1881.40	6	1881.40	6	796	1881.40	6	855	1881.40	6	900
lc2_2_3	1844.33	6	1844.33	6	1044	1844.33	6	1136	1844.33	6	900
lc2_2_4	1767.12	6	1767.12	6	3465	1767.12	6	3625	1767.12	6	900
lc2_2_5	1891.21	6	1891.21	6	604	1891.21	6	651	1891.21	6	900
lc2_2_6	1857.78	6	1857.78	6	925	1857.78	6	977	1857.78	6	900
lc2_2_7	1850.13	6	1850.13	6	777	1850.13	6	829	1850.13	6	900
lc2_2_8	1824.34	6	1824.34	6	1427	1824.34	6	1488	1824.34	6	900
lc2_2_9	1854.21	6	1854.21	6	1137	1854.21	6	1211	1854.21	6	900
lc2_2_10	1817.45	6	1817.45	6	2327	1817.45	6	2612	1817.45	6	900
lr1_2_1	4819.12	20	4819.12	20	644	4819.12	20	672	4819.12	20	900
lr1_2_2	4621.21	17	4621.21	17	654	4621.21	17	730	4621.21	17	900
lr1_2_3	4402.38	14	4402.38	14	387	4402.38	14	424	4402.38	14	900
lr1_2_4	3027.06	10	3031.20	10	843	**3027.06**	**10**	1020	3044.69	10	900
lr1_2_5	4760.18	16	4760.18	16	569	4760.18	16	568	4760.18	16	900
lr1_2_6	4800.94	13	4800.94	13	198	4800.94	13	222	4800.94	13	900
lr1_2_7	3543.36	12	**3543.36**	**12**	333	3543.36	12	594	3550.61	12	900
lr1_2_8	2759.32	9	**2759.32**	**9**	1026	2759.32	9	1092	2814.32	9	900
lr1_2_9	5050.75	13	5050.75	13	647	5050.75	13	557	5050.75	13	900
lr1_2_10	3664.08	11	**3664.08**	**11**	250	3672.99	11	302	3748.06	11	900
lr2_2_1	4073.10	5	4073.10	5	554	4073.10	5	606	4073.10	5	900
lr2_2_2	3796.00	4	3796.00	4	518	3796.00	4	744	3796.00	4	900
lr2_2_3	3098.36	4	3098.36	4	1964	3098.36	4	2669	3098.36	4	900
lr2_2_4	2486.00	3	**2486.00**	**3**	6845	2486.00	3	4416	2491.73	3	900
lr2_2_5	3438.39	4	3438.39	4	1793	3438.39	4	1064	3438.39	4	900
lr2_2_6	4457.95	3	4535.56	3	658	**4468.60**	**3**	666	4639.85	3	900
lr2_2_7	3098.35	3	**3098.35**	**3**	1320	3098.35	3	1531	3201.68	3	900
lr2_2_8	2449.36	2	2510.84	2	1856	**2460.49**	**2**	2662	2586.42	2	900
lr2_2_9	3922.11	3	**3922.11**	**3**	726	3922.11	3	470	3927.13	3	900
lr2_2_10	3254.83	3	**3254.83**	**3**	919	3254.83	3	1155	3274.96	3	900
lrc1_2_1	3606.06	19	3606.06	19	636	3606.06	19	664	3606.06	19	900
lrc1_2_2	3671.02	15	**3671.02**	**15**	296	3671.02	15	503	3683.24	15	900
lrc1_2_3	3154.92	13	3154.92	13	431	3154.92	13	734	3154.92	13	900
lrc1_2_4	2631.82	10	2631.82	10	867	2631.82	10	958	2631.82	10	900
lrc1_2_5	3715.81	16	3715.81	16	449	3715.81	16	494	3715.81	16	900
lrc1_2_6	3572.16	16	3572.16	16	451	3572.16	16	656	3572.16	16	900
lrc1_2_7	3666.34	14	3697.38	14	257	3697.71	14	250	3697.71	14	900
lrc1_2_8	3145.74	13	**3145.74**	**13**	266	3145.74	13	530	3202.16	13	900
lrc1_2_9	3157.34	13	3164.58	13	264	3174.36	13	474	**3157.34**	**13**	900
lrc1_2_10	2928.90	12	2935.37	12	385	**2931.34**	**12**	1222	2948.60	12	900
lrc2_2_1	3595.18	6	3595.18	6	386	3595.18	6	383	3595.18	6	900
lrc2_2_2	3158.25	5	3184.23	5	557	**3158.25**	**5**	687	3342.00	5	900
lrc2_2_3	2881.99	4	2946.19	4	1025	**2886.49**	**4**	1084	2948.56	4	900
lrc2_2_4	2835.40	3	**2849.09**	**3**	2791	2849.43	3	3171	2908.50	3	900
lrc2_2_5	2776.93	5	2776.93	5	1316	2776.93	5	1406	2776.93	5	900
lrc2_2_6	2707.96	5	2707.96	5	925	2707.96	5	2144	2707.96	5	900
lrc2_2_7	3010.68	4	**3010.68**	**4**	634	3015.49	4	704	3057.23	4	900
lrc2_2_8	2399.89	4	**2399.89**	**4**	1308	2399.89	4	1590	2400.19	4	900
lrc2_2_9	2208.49	4	2208.49	4	2187	2208.49	4	1866	2208.49	4	900
lrc2_2_10	2437.88	3	2506.86	3	1950	2437.88	3	2196	2664.99	3	900

Table 5.6: Li & Lim PDP-200 benchmark results. (td) total distance (nv) number of vehicles, (ct) compute time used in seconds.

The results of table 5.6 shows that our approach is highly competitive on the Li & Lim 200-task instances performing equally well or outperforming Curtois et al (2018) on all instances except for one.

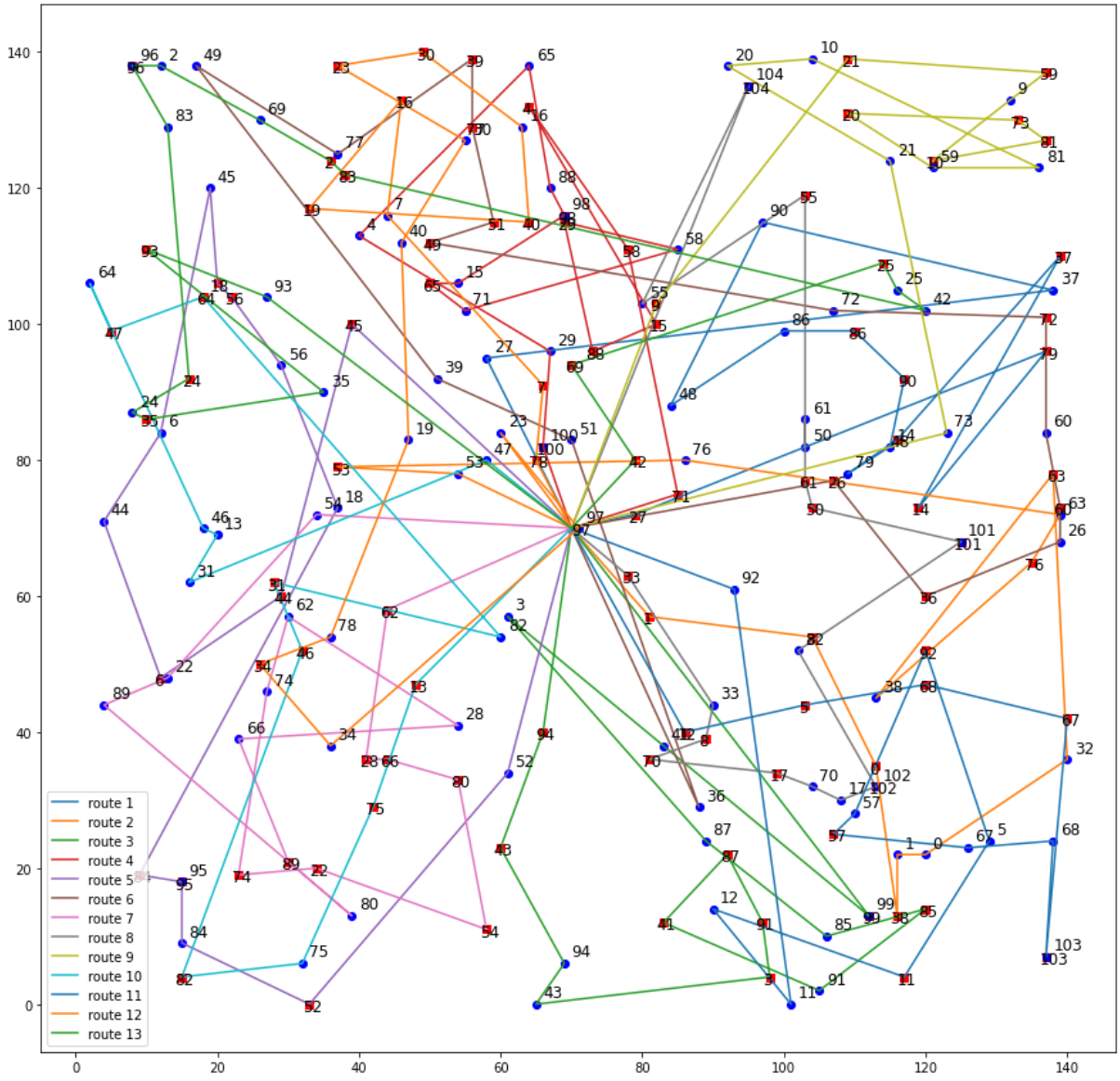
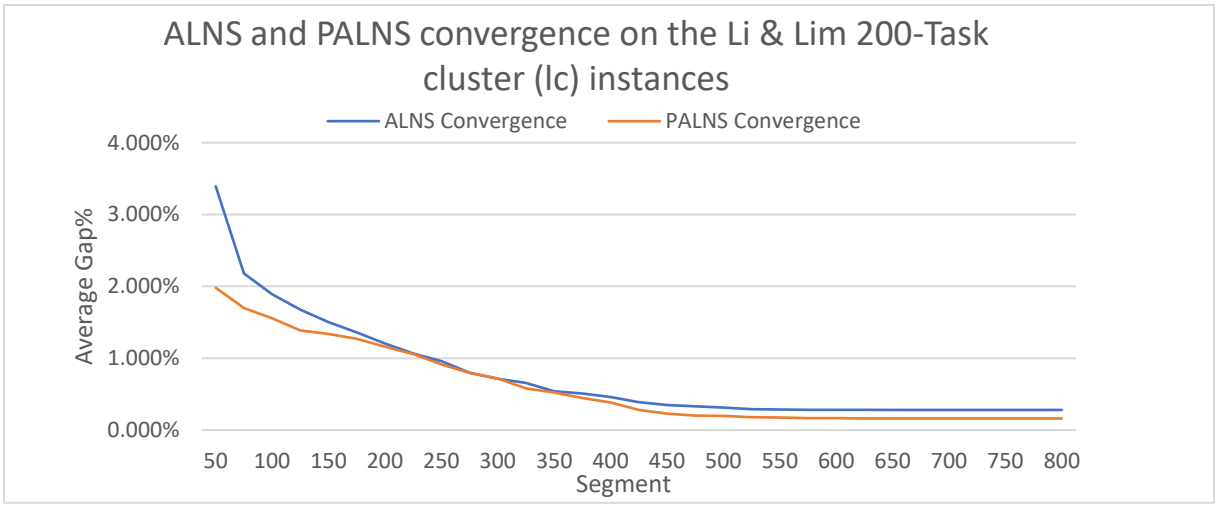
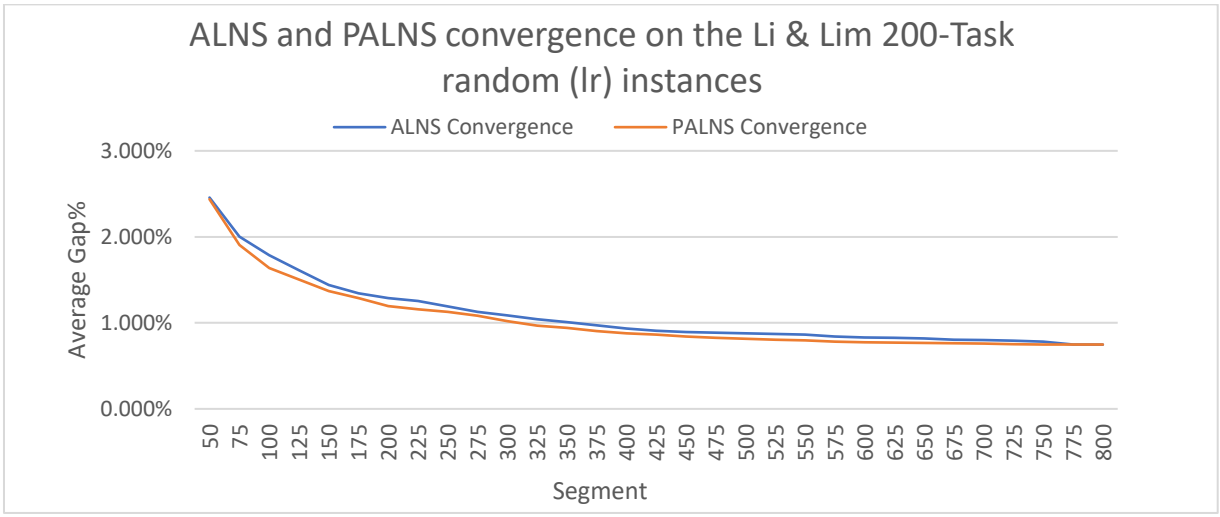


Figure 5.5: Example of the Li & Lim uniformly random request location benchmark instances. The figure shows the best-known route schedule at a total cost of **5050.75** for the Li & Lim **LR1_2_9** benchmark instance where there are a total of 100 pickups and 100 deliveries. The red squares are the pickups, and the blue circles are the deliveries, and the depot is the purple cross in the centre. The identification number at the pickup matches the number at the delivery. Source: The author.





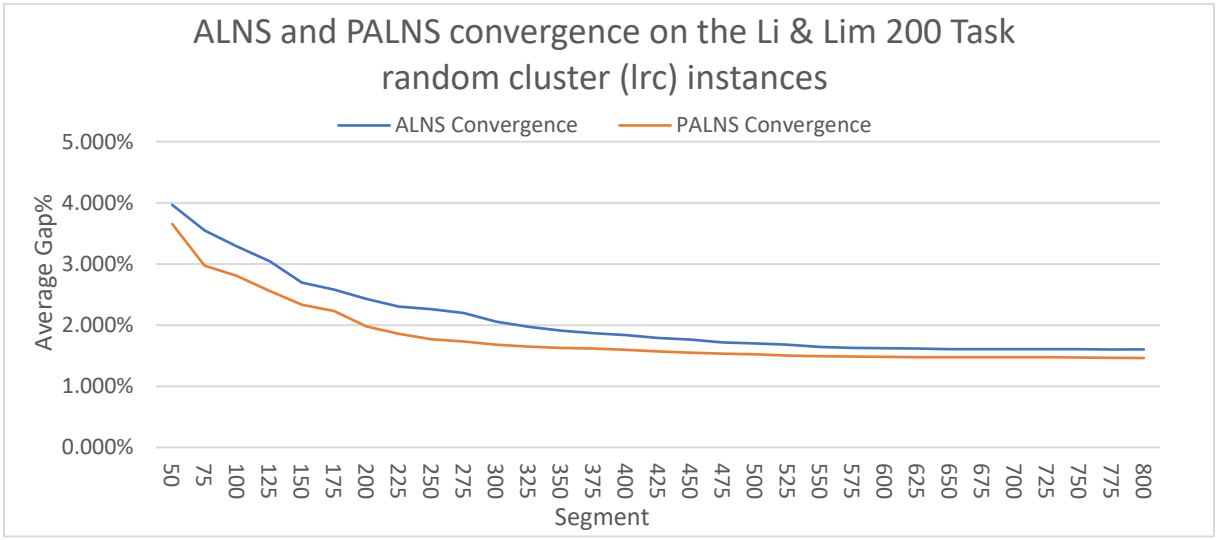


Figure 5.6: plots showing the convergence on a 25-segment interval starting at segment number 50 on the 200-task Li & Lim benchmark instance types for our ALNS and PALNS implementations. It clearly demonstrates the superior rate of convergence for our PALNS approach in the average case.

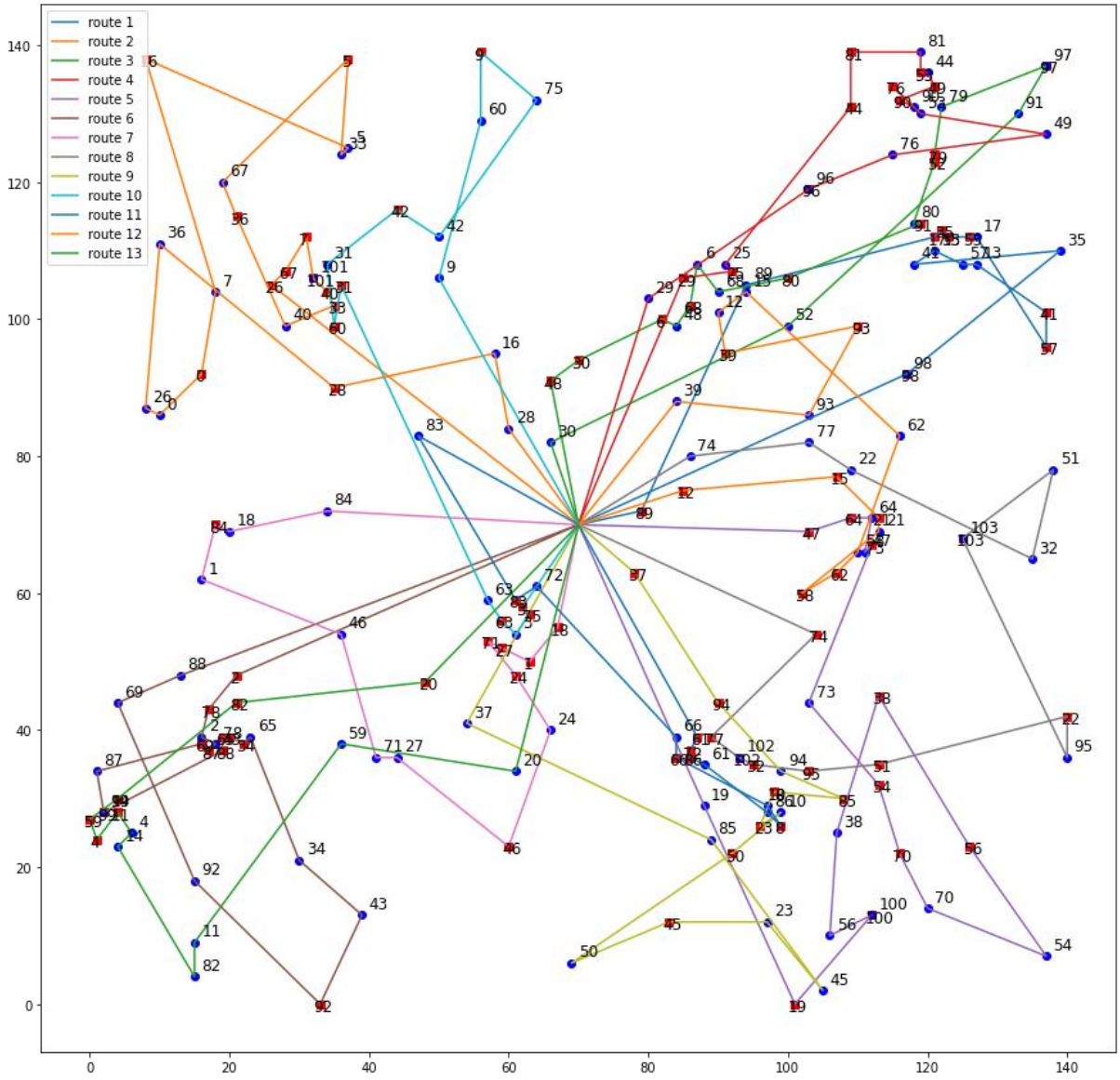


Figure 5.7: Example of the Li & Lim mixed request location benchmark instances. The figure shows the best-known route schedule at a total cost of **3145.74** for the Li & Lim **LRC1_2_8** benchmark instance where there are a total of 100 pickups and 100 deliveries. The red squares are the pickups, and the blue circles are the deliveries, and the depot is the purple cross in the centre. The identification number at the pickup matches the number at the delivery. Source: The author.

Chapter 6

Concluding remarks and future work

In this paper we have studied the pickup and delivery problem in great depth. We have investigated guided ejection search for the dual objective PDPTW. We have seen how the adaptive large neighbourhood search framework can be augmented by adding a perceptive layer in front, allowing us to branch into separate independent ALNS components. We have discussed the costs associated with this augmentation and have seen that the number of end nodes increases exponentially in the depth of the conditional tree. By utilizing parallel computing where we ran 100s of concurrent problem instances on the GPU we were able to generate a much larger number of samples than what is required to saturate a standard ALNS and reducing the segment size was not a good option as synchronization between threads is expensive. Therefore, instead of letting all this additional information made available go to waste we conclude based on empirical observations that it can be made use of through conditional perceptive branching outperforming regular ALNS in such circumstances.

Li & Lim (2001) emphasizes the importance of having a tabu structure in place to avoid the situation of constantly returning to some previously explored area of a search space. We believe it becomes even more important when hundreds of agents are searching in parallel to be able to mark off explored parts of a search space such that the individual agents are "pushed away from one another" and directed into unexplored territory. That is why we used a hash of all seen objective values which gave us an efficient way for threads to communicate the areas that had been explored. Although there is a chance that multiple solutions share the same objective value, we consider such events to be so rare that it is not worth the additional memory cost of hashing entire solutions which would constitute long sequences of numbers.

Future work may investigate the effectiveness of PALNS on different problem classes within logistics and other areas. One could investigate the effectiveness of a deeper PALNS on distributed compute networks with thousands of compute units. Another research topic is investigating the difficulty ALNS seems to have when navigating tightly constrained problems with many large infeasible regions, one option that may be considered is allowing a certain degree of infeasibility to occur at a high penalty. A single parallel guided ejection search unlike our many GES approach may be investigated as it is likely that one will see good speedup since determining whether the ejection of different pickup and delivery pairs allows the insertion of a request from the request bank may be done in parallel with very little to no need for communication. Finally, one should investigate data representation and data flow to bring about more efficient optimization algorithms, particularly for large problems with heterogeneous constraints, since heuristics relying on random access patterns in combination with limited cache sizes is a leading cause of computational inefficiency in this area.

Bibliography:

- [1] Abbasi, Mahdi. Rafiee, Milad. Khosravi, Mohammad R. Jolfaei, Alireza. Menon, Varun G. Koushyar, Javad Mokhtari. (2020) An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems. Journal of Cloud Computing 9, 6. https://doi.org/10.1186/s13677-020-0157-4

- [2] Lukas Bach, Geir Hasle, Christian Schulz. (2019). Adaptive Large Neighborhood Search on the Graphics Processing Unit. European Journal of Operational Research, Volume 275, Issue 1, Pages 53-66.

- [3] Roberto Baldacci, Enrico Bartolini, Aristide Mingozzi, (2011) An Exact Algorithm for the Pickup and Delivery Problem with Time Windows. Operations Research 59(2):414-426. https://doi.org/10.1287/opre.1100.0881

- [4] Bent, Russell & Van Hentenryck, Pascal. (2006). A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. Computers & OR. 33. 875-893.

- [5] Burke, E.K & Bykov, Yuri. (2008). A late acceptance strategy in hill-climbing for examination timetabling problems.

- [6] Burke, E.K., Hyde, M.R., Kendall, G., Ochoa, G., Ozcan, E., Woodward, J.R. (2009). Exploring Hyper-heuristic Methodologies with Genetic Programming. In: Mumford, C.L., Jain, L.C. (eds) Computational Intelligence. Intelligent Systems Reference Library, vol 1. Springer, Berlin, Heidelberg.

- [7] Burke, E. K., Hyde, M. R., Kendall, G., Ochoa, G., Özcan, E., & Woodward, J. R. (2019). A classification of hyper-heuristic approaches: Revisited. In Handbook of metaheuristics (453-477). Cham: Springer Publishing Company.

- [8] Cerny, V. (1985) Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm. Journal of Optimization Theory and Applications, 45, 4151.

- [9] Chang, Jui-Fang & Chu, Shu-Chuan & Roddick, John & Pan, Jeng-Shyang. (2005). A Parallel Particle Swarm Optimization Algorithm with Communication Strategies. Journal of Information Science and Engineering. 21. 809-818.

- [10] Chipperfield, Andrew & Fleming, Peter. (1996). Parallel genetic algorithms. Parallel and Distributed Computing Handbook. 1118-1143.

- [11] Curtois, T., Silva, D.L., Qu, Y., & Laesanklang, W. (2018). Large neighbourhood search with adaptive guided ejection search for the pickup and delivery problem with time windows. EURO Journal on Transportation and Logistics, 7, 151-192.

- [12] John H. Drake, Ahmed Kheiri, Ender Özcan, Edmund K. Burke (2020), Recent advances in selection hyper-heuristics, European Journal of Operational Research, Volume 285, Issue 2, Pages 405-428.

- [13] Fisher, H., and Thompson, G.L. (1963) Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules. Prentice-Hall, Englewood Cliffs, 225-251.

- [14] Gendreau, Michel. Potvin, Jean-Yves. (2010). *Handbook of Metaheuristics.* Springer New York, NY. https://doi.org/10.1007/978-1-4419-1665-5

- [15] Fred Glover, (1989) Tabu Search—Part I. ORSA Journal on Computing 1(3):190-206. https://doi.org/10.1287/ijoc.1.3.190

- [16] Harifi, Sasan & Mohammadzadeh, Javad & Khalilian, Madjid & Ebrahimnejad, Sadoullah. (2021). Giza Pyramids Construction: an ancient-inspired metaheuristic algorithm for optimization. Evolutionary Intelligence. 14. 10.1007/s12065-020-00451-3.

- [17] Hemmati, A. Hvattum, L.M. Fagerholt, K. & Norstad, I (2014) Benchmark Suite for Industrial and Tramp Ship Routing and Scheduling Problems, INFOR: Information Systems and Operational Research, 52:1, 28-38, https://doi.org/10.3138/infor.52.1.28

- [18] Hemmati, A. and Hvattum, L.M. (2017), Evaluating the importance of randomization in adaptive large neighborhood search. Intl. Trans. in Op. Res., 24: 929-942. https://doi.org/10.1111/itor.12273

- [19] Holland, J. H. (1975). Adaptation in natural and artificial systems. Michigan: University of Michigan Press.

- [20] Homsi, Gabriel & Martinelli, Rafael & Vidal, Thibaut & Fagerholt, Kjetil. (2018). Industrial and Tramp Ship Routing Problems: Closing the Gap for Real-Scale Instances.

- [21] J. Kennedy and R. Eberhart (1995), "Particle swarm optimization," Proceedings of ICNN'95 International Conference on Neural Networks, pp. 1942-1948

- [22] Kallestad, Jakob Vigerust. (2021). Developing an Intelligent Hyperheuristic for Combinatorial Optimization Problems using Deep Reinforcement Learning. The University of Bergen

- [23] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. Science, 220(4598), 671–680. http://www.jstor.org/stable/1690046

- [24] Li, Haibing & Lim, Andrew. (2001). A Metaheuristic for the Pickup and Delivery Problem with Time Windows. International Journal on Artificial Intelligence Tools. 12. 160-167. 10.1109/ICTAI.2001.974461.

- [25] Nagata, Yuichi & Tojo, Satoshi. (2009). Guided Ejection Search for the Job Shop Scheduling Problem. 168-179. 10.1007/978-3-642-01009-5_15.

- [26] Nagata, Yuichi & Kobayashi, Shigenobu. (2010). Guided Ejection Search for the Pickup and Delivery Problem with Time Windows. 202-213. 10.1007/978-3-642-12139-5_18.

- [27] Parragh, Sophie & Schmid, Verena. (2013). Hybrid column generation and large neighorhood search for the dial-a-ride problem. Computers & operations research. 40. 490-497. 10.1016/j.cor.2012.08.004.

- [28] David Pisinger, Stefan Ropke (2007), A general heuristic for vehicle routing problems, Computers & Operations Research, Volume 34, Issue 8, Pages 2403-2435, ISSN 0305-0548, https://doi.org/10.1016/j.cor.2005.09.012.

- [29] Pisinger, D., & Ropke, S. (2010). Large Neighborhood Search. In M. Gendreau (Ed.), Handbook of Metaheuristics (2 ed., pp. 399-420). Springer.

- [30] Ropke, S., & Pisinger, D. (2006). An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. Transportation Science, 40(4), 455–472. http://www.jstor.org/stable/25769321

- [31] Ropke, S. (2009). PALNS - A software framework for parallel large neighborhood search. In 8th Metaheuristic International Conference CDROM

- [32] Ropke, Stefan & Cordeau, Jean-François. (2009). Branch and Cut and Price for the Pickup and Delivery Problem with Time Windows. Transportation Science. 43. 267-286. 10.1287/trsc.1090.0272.

- [33] M.W.P. Savelsbergh, M. Solomon (1995), “The General Pickup and Delivery Problem”, Transportation Science, 29, 17-29.

- [34] Shaw, P. (1998). Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In: Maher, M., Puget, JF. (eds) Principles and Practice of Constraint Programming — CP98. CP 1998. Lecture Notes in Computer Science, vol 1520. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-49481-2_30

- [35] Marius M. Solomon, (1987) Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. Operations Research 35(2):254-265. https://doi.org/10.1287/opre.35.2.254

- [36] K. Sorensen and F. Glover. (2013) Metaheuristics. In S. Gass and M. Fu, editors, Encyclopaedia of OR/MS, 3rd ed., pages 960–970,

- [37] Toth, P., & Vigo, D. (2014). Vehicle routing: Problems, methods, and applications. Monographs on discrete mathematics and applications (2nd). Philadelphia: SIAM.

- [38] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. (2006). Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations. Journal of Aerospace Computing, Information, and Communication. 3:3, 123-137

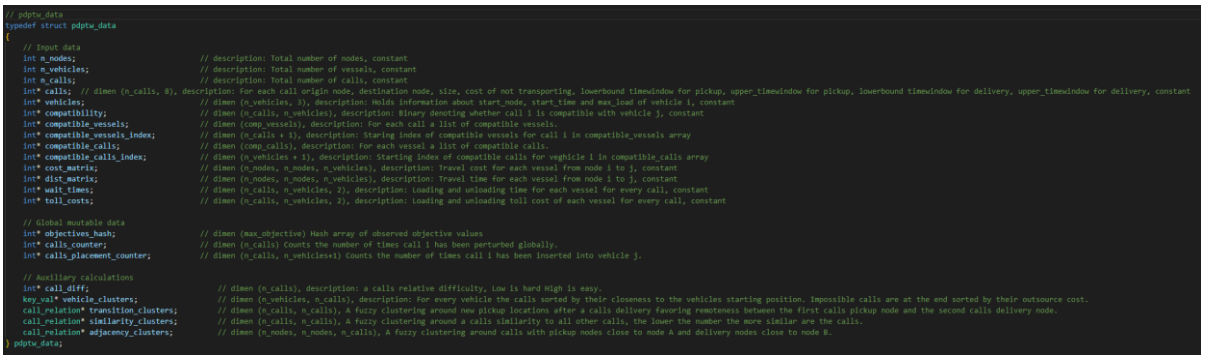
- [39] Yuan Wang, Xinjia Jiang, Loo Hay Lee, Ek Peng Chew, Kok Choon Tan, (2017). Tree based searching approaches for integrated vehicle dispatching and container allocation in a transshipment hub, Expert Systems with Applications, Volume 74, Pages 139-150, ISSN 0957-4174, https://doi.org/10.1016/j.eswa.2017.01.003.

- [40] The CUDA programming guide v11.8.0, November 9, 2022, https://docs.nvidia.com/cuda/cuda-c-programming-guide/

- [41] Gupta, Pradeep, (2020). CUDA Refresher: The CUDA Programming Model. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Appendix A. Code segments

A1: PDPTW data structure



```
// pdptw_data
typedef struct pdptw_data
{
    // Input data
    int n_nodes;                    // description: Total number of nodes, constant
    int n_vehicles;                 // description: Total number of vessels, constant
    int n_calls;                    // description: Total number of calls, constant
    int* calls;  // dimen (n_calls, 8), description: For each call origin node, destination node, size, cost of not transporting, lowerbound timewindow for pickup, upper_timewindow for pickup, lowerbound timewindow for delivery, upper_timewindow for delivery, constant
    int* vehicles;                  // dimen (n_vehicles, 3), description: Holds information about start_node, start_time and max_load of vehicle i, constant
    int* compatibility;             // dimen (n_calls, n_vehicles), description: Binary denoting whether call i is compatible with vehicle j, constant
    int* compatible_vessels;        // dimen (comp_vessels), description: For each call a list of compatible vessels.
    int* compatible_vessels_index;  // dimen (n_calls + 1), description: Staring index of compatible vessels for call i in compatible_vessels array
    int* compatible_calls;          // dimen (comp_calls), description: For each vessel a list of compatible calls.
    int* compatible_calls_index;    // dimen (n_vehicles + 1), description: Starting index of compatible calls for veghicle i in compatible_calls array
    int* cost_matrix;               // dimen (n_nodes, n_nodes, n_vehicles), description: Travel cost for each vessel from node i to j, constant
    int* dist_matrix;               // dimen (n_nodes, n_nodes, n_vehicles), description: Travel time for each vessel from node i to j, constant
    int* wait_times;                // dimen (n_calls, n_vehicles, 2), description: Loading and unloading time for each vessel for every call, constant
    int* toll_costs;                // dimen (n_calls, n_vehicles, 2), description: Loading and unloading toll cost of each vessel for every call, constant

    // Global mutable data
    int* objectives_hash;           // dimen (max_objective) Hash array of observed objective values
    int* calls_counter;             // dimen (n_calls) Counts the number of times call i has been perturbed globally.
    int* calls_placement_counter;   // dimen (n_calls, n_vehicles+1) Counts the number of times call i has been inserted into vehicle j.

    // Auxillary calculations
    int* call_diff;                  // dimen (n_calls), description: a calls relative difficulty, Low is hard High is easy.
    key_val* vehicle_clusters;       // dimen (n_vehicles, n_calls), description: For every vehicle the calls sorted by their closeness to the vehicles starting position. Impossible calls are at the end sorted by their outsource cost.
    call_relation* transition_clusters;  // dimen (n_calls, n_calls), A fuzzy clustering around new pickup locations after a calls delivery favoring remoteness between the first calls pickup node and the second calls delivery node.
    call_relation* similarity_clusters;  // dimen (n_calls, n_calls), A fuzzy clustering around a calls similarity to all other calls, the lower the number the more similar are the calls.
    call_relation* adjacency_clusters;   // dimen (n_nodes, n_nodes, n_calls), A fuzzy clustering around calls with pickup nodes close to node A and delivery nodes close to node B.
} pdptw_data;
```

Data structure used to hold the input data from an external problem specification as well as initial auxiliary calculations and certain global mutable data.

A2: PDPTW problem structure

```
// Structure representing a perturbative pdptw optimization problem instance.
typedef struct pdptw_problem
{
    pdptw_instance* instance;         // A pointer to the 'primary' instance structure
    int* r_calls;                     // dimen(n_calls+1) Used to hold the calls that is to be removed from the solution
    int* r_vehicles;                  // dimen(n_vehicles+2) used to hold the vehicles that will be perturbed in this iteration
    int* hash_table;                  // dimen(max(n_calls,n_vehicles) A hash array used for utility functionality
    pdptw_instance** beam_instances;  // dimen(n_local_threads) Auxiliary instance structs used in beam search
    deep_solution** beam_solutions;   // dimen(BEAM_WIDTH*SEARCH_WIDTH+BEAM_WIDTH) Auxiliary solution structs used in beam search
    key_val* metrics;                 // dimen(max(data->n_calls, data->n_vehicles+1)) Utility key_val array used to sort calls based on some metric
    tour_deviation* tds;              // dimen(data->n_calls) tour_deviation array used to find most expensive calls in current solution
} pdptw_problem;
```

Data structure used to hold a copy of a perturbative problem. Each thread has an independent version of a problem and all perturbations on some underlying problem domain happens through this structure.

A3: PDPTW instance structure

```
// This struct represents an instance of the pdptw.
typedef struct pdptw_instance
{
    int infeasible_count;  // Number of infeasible vehicles
    int sum_costs;         // Total cost of the instance
    int* call_loc;         // dimen(n_calls): Vehicle location of call i
    vessel* vessels;       // dimen(n_vehicles+1) vessel structures
    node* nodes;           // dimen(2 * data->n_calls + 2 * (data->n_vehicles + 1)) node structures
    best_changes* bcs;     // A best_changes struct
} pdptw_instance, deep_solution;
```

Data structure at the core of each problem structure representing a complete travel plan from one state of the search onto the next.

A4: Wrapper structure for PD insertion structures

```
// Wrapper for an array of best_change structures
typedef struct best_changes
{
    int count;          // The number of changes recorded in the current configuration
    int limit;          // The currently allocated number of best_change structs
    best_change* bc;    // dimen(limit): Array of best_change structures
} best_changes;
```

A wrapper structure holding an array of structures representing insertion locations in a schedule of a current solution for some pickup and delivery request pair (i, j).

A5: Insertion structure for PD pair

```
// Used by check_all_position_changes function
typedef struct best_change
{
    int i_idx;              // The pickup node is to be inserted after this node in the vessel
    int j_idx;              // The delivery node is to be inserted before this node in the vessel
    int vessel;             // The vessel where the call is to be inserted
    int insertion_cost;     // The insertion cost
} best_change;
```

Structure representing an insertion location in a schedule of a current solution for some pickup and delivery request pair (i, j) in a vehicle, and the cost of this insertion.

A6: Vehicle structure

```
// This structure is used to represent a vessel belonging to some particular pdptw instance.
typedef struct vessel
{
    int first_node_idx;         // The index of the vehicles first node
    int last_node_idx;          // The index of the vehicles last node
    unsigned int cost;          // The cost of the vehicles travel plan
    bool infeasible;            // Whether the vehicles travel plan is feasible
    int num_calls;              // Number of calls (x2) in the vehicles travel plan
} vessel;
```

Vehicle structure used to represent a vehicle. Holds the first and the last index of the vehicle into an array of nodes as well as the cost of the vehicles travel plan, whether the travel plan is infeasible and the number of requests in the travel plan.

A7: Node structure

```
// This structure is used to represent a node in a vessels travel plan.
// Holds index of the previous and the next node in that travel plan.
typedef struct node node;
struct node
{
    int call;                   // What call the node belongs to
    int leave_time;             // The leave time from the previous node
    int lpat;                   // The latest possible arrival time to this node that will still satisfy the requirement of all later nodes
    int node_type;              // 0 Pickup / 1 Delivery node
    int weight;                 // The weight of the vehicle at the previous node
    int prev_node_idx;          // The index of the previous node
    int next_node_idx;          // The index of the next node
};
```

A node in a travel plan holds key information such as accumulated time, vehicle load and latest possible arrival time and index of previous/next node in the travel plan.

A8: Tour deviation structure

```
// This structure represents a tour_deviation
typedef struct tour_deviation
{
    int call;                           // The call whos tour deviation is being calculated
    int pickup_cost_deviation;          // The pickup cost deviation in the current solution
    int delivery_cost_deviation;        // The delivery cost deviation in the current solution
    int cost_deviation;                 // The cost deviation in the current solution
    int pickup_distance_deviation;      // The pickup distance deviation in the current solution
    int delivery_distance_deviation;    // The delivery distance deviation in the current solution
    int distance_deviation;             // The distance deviation in the current solution
}tour_deviation;
```

A tour deviation for a call represents the cost of that call in the current solution.

A9: Key value structure

```
// This structure is used to represents a key value pair.
typedef struct key_val
{
    int key;
    int value;
} key_val;
```

key value pair structure is used for comparison and sorting purposes.

A10: Call relation structure

```
// This structure is used to represent a relation between calls
typedef struct call_relation
{
    int key;
    float value;
} call_relation;
```

The call relation structure defines a relationship value between pairs of calls.

A11: CUDA PALNS local update kernel

```
__global__ void update_local_probabilities(global_agent* global) {
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    adaptive_agent* agent = global->local_agents[bid];

    for (int j = tid; j < 2 * 2 * global->n_actions; j += blockDim.x) {
        float sum_probs_action = 0.0f;
        float sum_probs_neighbourhood = 0.0f;
        action* a = &agent->operator_probabilities[j * global->n_actions];
        neighbourhood* n = &agent->neighbourhood_probabilities[j * global->n_neighbourhoods];
        for (int i = 0; i < max(global->n_actions, global->n_neighbourhoods); i++) {
            if (i < global->n_actions) {
                if (a[i].count == 0) {
                    sum_probs_action += a[i].probability;
                }
                else {
                    a[i].probability = a[i].probability * (1.0f - global->reaction_factor_action) + global->reaction_factor_action * (a[i].score / (float)a[i].count);
                    sum_probs_action += a[i].probability;
                }
            }
            if (i < global->n_neighbourhoods) {
                if (n[i].count == 0) {
                    sum_probs_neighbourhood += n[i].probability;
                }
                else {
                    n[i].probability = n[i].probability * (1.0f - global->reaction_factor_neighbourhood) + global->reaction_factor_neighbourhood * (n[i].score / (float)n[i].count);
                    sum_probs_neighbourhood += n[i].probability;
                }
            }
        }

        for (int i = 0; i < max(global->n_actions, global->n_neighbourhoods); i++) {
            if (i < global->n_actions) {
                a[i].probability = max(a[i].probability, sum_probs_action / (float)pow(global->n_actions, 3.0f)) / sum_probs_action;
            }
            if (i < global->n_neighbourhoods) {
                n[i].probability = max(n[i].probability, sum_probs_neighbourhood / (float)pow(global->n_neighbourhoods, 3.0f)) / sum_probs_neighbourhood;
            }
        }
    }
}
```

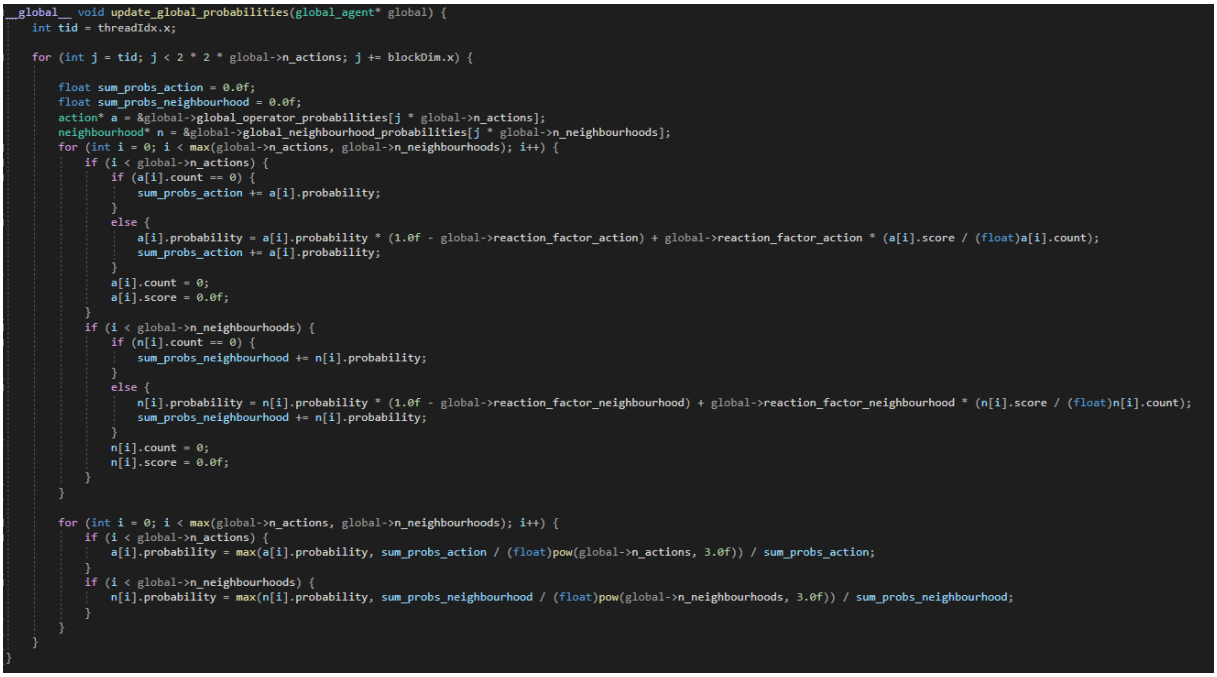
The above CUDA Kernel is responsible for updating the local heuristic selection probabilities of each local PALNS agent.

A12: CUDA PALNS reduction and transmission kernel

```cuda
__global__ void reduce_and_transmit_scores(global_agent* global) {
    __shared__ int op_counts[1024];
    __shared__ float op_scores[1024];
    __shared__ int nb_counts[1024];
    __shared__ float nb_scores[1024];

    __shared__ int action_cs[1024];
    __shared__ int neighbourhood_cs[1024];

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    if (tid < global->n_agents) {
        if (bid < 2 * 2 * global->n_actions * global->n_actions) {
            op_counts[tid] = global->local_agents[tid]->operator_probabilities[bid].count;
            op_scores[tid] = global->local_agents[tid]->operator_probabilities[bid].score;
            global->local_agents[tid]->operator_probabilities[bid].count = 0;
            global->local_agents[tid]->operator_probabilities[bid].score = 0.0f;
        }
        if (bid < 2 * 2 * global->n_actions * global->n_neighbourhoods) {
            nb_counts[tid] = global->local_agents[tid]->neighbourhood_probabilities[bid].count;
            nb_scores[tid] = global->local_agents[tid]->neighbourhood_probabilities[bid].score;
            global->local_agents[tid]->neighbourhood_probabilities[bid].count = 0;
            global->local_agents[tid]->neighbourhood_probabilities[bid].score = 0.0f;
        }

        if (bid < global->n_actions) {
            action_cs[tid] = global->local_agents[tid]->action_counts[bid];
        }
        if (bid < global->n_neighbourhoods) {
            neighbourhood_cs[tid] = global->local_agents[tid]->neighbourhood_counts[bid];
        }
    }
    else {
        op_counts[tid] = 0;
        op_scores[tid] = 0.0f;
        nb_counts[tid] = 0;
        nb_scores[tid] = 0.0f;
        action_cs[tid] = 0;
        neighbourhood_cs[tid] = 0;
    }
    __syncthreads();
    for (int i = 2; i <= 1024; i *= 2) {
        if (tid % i == 0) {
            op_counts[tid] += op_counts[tid + i / 2];
            op_scores[tid] += op_scores[tid + i / 2];
            nb_counts[tid] += nb_counts[tid + i / 2];
            nb_scores[tid] += nb_scores[tid + i / 2];
            action_cs[tid] += action_cs[tid + i / 2];
            neighbourhood_cs[tid] += neighbourhood_cs[tid + i / 2];
        }
        __syncthreads();
    }
    if (tid == 0) {
        if (bid < 2 * 2 * global->n_actions * global->n_actions) {
            global->global_operator_probabilities[bid].count = op_counts[0];
            global->global_operator_probabilities[bid].score = op_scores[0];
        }
        if (bid < 2 * 2 * global->n_actions * global->n_neighbourhoods) {
            global->global_neighbourhood_probabilities[bid].count = nb_counts[0];
            global->global_neighbourhood_probabilities[bid].score = nb_scores[0];
        }
        if (bid < global->n_actions) {
            global->accumulative_action_counts[bid] = action_cs[0];
        }
        if (bid < global->n_neighbourhoods) {
            global->accumulative_neighbourhood_counts[bid] = neighbourhood_cs[0];
        }
    }
}
```

This kernel is responsible for accumulating and transferring all the counts and scores of the local agents recorded in a segment to the global agent in the CUDA implementation of PALNS for the PDPTW.

A13: CUDA PALNS global update kernel



```
__global__ void update_global_probabilities(global_agent* global) {
    int tid = threadIdx.x;

    for (int j = tid; j < 2 * 2 * global->n_actions; j += blockDim.x) {

        float sum_probs_action = 0.0f;
        float sum_probs_neighbourhood = 0.0f;
        action* a = &global->global_operator_probabilities[j * global->n_actions];
        neighbourhood* n = &global->global_neighbourhood_probabilities[j * global->n_neighbourhoods];
        for (int i = 0; i < max(global->n_actions, global->n_neighbourhoods); i++) {
            if (i < global->n_actions) {
                if (a[i].count == 0) {
                    sum_probs_action += a[i].probability;
                }
                else {
                    a[i].probability = a[i].probability * (1.0f - global->reaction_factor_action) + global->reaction_factor_action * (a[i].score / (float)a[i].count);
                    sum_probs_action += a[i].probability;
                }
                a[i].count = 0;
                a[i].score = 0.0f;
            }
            if (i < global->n_neighbourhoods) {
                if (n[i].count == 0) {
                    sum_probs_neighbourhood += n[i].probability;
                }
                else {
                    n[i].probability = n[i].probability * (1.0f - global->reaction_factor_neighbourhood) + global->reaction_factor_neighbourhood * (n[i].score / (float)n[i].count);
                    sum_probs_neighbourhood += n[i].probability;
                }
                n[i].count = 0;
                n[i].score = 0.0f;
            }
        }

        for (int i = 0; i < max(global->n_actions, global->n_neighbourhoods); i++) {
            if (i < global->n_actions) {
                a[i].probability = max(a[i].probability, sum_probs_action / (float)pow(global->n_actions, 3.0f)) / sum_probs_action;
            }
            if (i < global->n_neighbourhoods) {
                n[i].probability = max(n[i].probability, sum_probs_neighbourhood / (float)pow(global->n_neighbourhoods, 3.0f)) / sum_probs_neighbourhood;
            }
        }
    }
}
```

The above CUDA Kernel is responsible for updating the global heuristic selection probabilities of the global PALNS agent.

Appendix B. Additional result tables

B1: OpenMP Results for all the Hemmati et al (2014) deep sea shipping with full cargo load benchmark instances

Instance		Best Known	OpenMP - PALNS (2023)			OpenMP - ALNS (2023)			Hemmati et al (2014)
			Min Gap%	Average Gap%	Ct^{Avg}	Min Gap%	Average Gap%	Ct^{Avg}	Min Gap%
C8V3	#1	9584863	0.0	0.0	0.720	0.0	0.0	0.723	0.0
C8V3	#2	9369654	0.0	0.0	0.652	0.0	0.0	0.655	0.0
C8V3	#3	4596681	0.0	0.0	1.011	0.0	0.0	1.007	0.0
C8V3	#4	6899730	0.0	0.0	0.788	0.0	0.0	0.807	0.0
C8V3	#5	6815253	0.0	0.0	0.735	0.0	0.0	0.73	0.0
C11V4	#1	34854819	0.0	0.0	1.448	0.0	0.0	1.466	0.0
C11V4	#2	25454434	0.0	0.0	1.445	0.0	0.0	1.421	0.0
C11V4	#3	29627143	0.0	0.0	1.165	0.0	0.0	1.140	0.0
C11V4	#4	33111680	0.0	0.0	1.112	0.0	0.0	1.120	0.0
C11V4	#5	28175914	0.0	0.0	1.194	0.0	0.0	1.196	0.0
C13V5	#1	11629005	0.0	0.0	3.693	0.0	0.0	3.919	0.0
C13V5	#2	11820655	0.0	0.0	4.219	0.0	0.0	4.301	0.0
C13V5	#3	9992593	0.0	0.0	5.463	0.0	0.0	5.871	0.0
C13V5	#4	12819619	0.0	0.0	2.831	0.0	0.0	2.894	0.0
C13V5	#5	10534892	0.0	0.0	6.268	0.0	0.0	6.782	0.0
C16V6	#1	51127590	0.0	0.0	6.229	0.0	0.0	6.508	0.0
C16V6	#2	44342796	0.0	0.0	9.797	0.0	0.0	10.518	0.0
C16V6	#3	45391842	0.0	0.0	4.815	0.0	0.0	5.037	0.0
C16V6	#4	39687114	0.0	0.0	8.414	0.0	0.0	9.021	0.0
C16V6	#5	42855603	0.0	0.0	6.341	0.0	0.0	6.790	0.0
C17V13	#1	17316720	0.0	0.0	4.329	0.0	0.0	4.484	0.0
C17V13	#2	12194861	0.0	0.0	4.958	0.0	0.0	5.189	0.0
C17V13	#3	12091554	0.0	0.0	3.982	0.0	0.0	4.187	0.0
C17V13	#4	12847653	0.0	0.0	3.849	0.0	0.0	3.909	0.0
C17V13	#5	13213406	0.0	0.0	5.792	0.0	0.0	5.842	0.0
C20V6	#1	16406738	0.0	0.0	6.503	0.0	0.0	6.749	0.0
C20V6	#2	16079401	0.0	0.0	7.057	0.0	0.0	7.496	0.0
C20V6	#3	17342200	0.0	0.0	5.759	0.0	0.0	6.002	0.0
C20V6	#4	16529748	0.0	0.0	6.760	0.0	0.0	6.971	0.0
C20V6	#5	17449378	0.0	0.0	6.248	0.0	0.0	6.341	0.0
C25V7	#1	22773158	0.0	0.0	11.129	0.0	0.0	10.626	0.0
C25V7	#2	20206329	0.0	0.0	8.948	0.0	0.0	9.270	0.0
C25V7	#3	19108952	0.0	0.0	11.412	0.0	0.0	12.071	0.0
C25V7	#4	22668675	0.0	0.0	12.518	0.0	0.0	12.193	0.0
C25V7	#5	23036603	0.0	0.0	10.863	0.0	0.0	10.433	0.0
C35V13	#1	86951609	0.0	0.0	22.683	0.0	0.0	20.137	0.0
C35V13	#2	83422071	0.0	0.0	17.664	0.0	0.0	15.322	0.0
C35V13	#3	83898591	0.0	0.0	17.191	0.0	0.0	14.723	0.0
C35V13	#4	91970481	0.0	0.0	20.987	0.0	0.0	19.377	0.0
C35V13	#5	91123040	**0.0**	0.0	20.765	0.0	0.0	18.357	0.00781
C50V20	#1	41310946	**0.0**	0.0	35.447	0.0	0.001	31.276	0.21053
C50V20	#2	37784994	**0.0**	0.0	35.842	0.0	0.00005	33.064	0.23046
C50V20	#3	39841724	**0.0**	0.0	31.526	0.003	0.104	26.793	0.18821
C50V20	#4	43941098	**0.0**	0.0	34.671	0.0	0.0	31.367	0.00000
C50V20	#5	41947437	**0.0**	0.0	34.109	0.0	0.0	30.048	0.05826
C70V30	#1	142679953	**0.0**	0.00004	50.149	0.0	0.0001	45.308	0.17061
C70V30	#2	135031988	0.042	0.0446	58.864	**0.006**	0.034	56.171	0.54117
C70V30	#3	162848836	0.0002	0.001	53.267	**0.0**	0.002	50.417	0.03380
C70V30	#4	155855123	**0.0**	0.035	50.599	0.053	0.058	46.185	0.43817
C70V30	#5	156557723	0.056	0.098	57.184	**0.055**	0.084	54.163	0.30541
C90V40	#1	190627186	**0.051**	0.071	102.281	0.058	0.142	96.740	0.54672
C90V40	#2	189770977	**0.0009**	0.010	99.846	0.003	0.01	96.179	0.71814
C90V40	#3	211038412	0.0433	0.059	99.843	**0.039**	0.073	90.770	0.52535
C90V40	#4	210449287	**0.047**	0.068	111.312	0.063	0.093	106.200	0.28283
C90V40	#5	197804917	**0.038**	0.112	105.546	0.159	0.204	101.144	0.41299
C100V50	#1	205826535	**0.034**	0.059	136.812	0.049	0.077	127.978	0.61765
C100V50	#2	207809147	0.078	0.104	141.632	**0.073**	0.118	136.426	0.35085
C100V50	#3	217000928	**0.108**	0.115	165.816	0.140	0.181	159.470	0.65807
C100V50	#4	220879632	**0.068**	0.076	143.255	0.070	0.080	142.075	0.16658
C100V50	#5	223265017	**0.011**	0.018	136.372	0.027	0.035	130.042	0.51935

Table B1: Results on the Hemmati et al (2014) deep sea shipping with full cargo load instances

B2: OpenMP Best results for all the Hemmati et al (2014) deep sea shipping with mixed cargo load benchmark instances

Instance	Best Known	OpenMP - PALNS (2023)			OpenMP - ALNS (2023)			Hemmati et al (2014)
		Min Gap %	Average Gap %	Ct^{Avg}	Min Gap %	Average Gap %	Ct^{Avg}	Min Gap %
C7V3 #1	5233464	0.0	0.0	0.909	0.0	0.0	0.939	0.0
C7V3 #2	6053699	0.0	0.0	0.676	0.0	0.0	0.699	0.0
C7V3 #3	5888949	0.0	0.0	0.748	0.0	0.0	0.761	0.0
C7V3 #4	6510656	0.0	0.0	0.749	0.0	0.0	0.784	0.0
C7V3 #5	7220458	0.0	0.0	0.837	0.0	0.0	0.858	0.0
C10V3 #1	7986248	0.0	0.0	1.288	0.0	0.0	1.367	0.0
C10V3 #2	7754484	0.0	0.0	1.375	0.0	0.0	1.429	0.0
C10V3 #3	9499357	0.0	0.0	0.952	0.0	0.0	0.985	0.0
C10V3 #4	8617192	0.0	0.0	1.129	0.0	0.0	1.163	0.0
C10V3 #5	8653992	0.0	0.0	1.578	0.0	0.0	1.649	0.0
C15V4 #1	13467090	0.0	0.0	1.974	0.0	0.0	1.948	0.0
C15V4 #2	12457251	0.0	0.0	2.286	0.0	0.0	2.320	0.0
C15V4 #3	12567396	0.0	0.0	2.077	0.0	0.0	2.079	0.0
C15V4 #4	11764241	0.0	0.0	1.671	0.0	0.0	1.747	0.0
C15V4 #5	10833640	0.0	0.0	2.841	0.0	0.0	2.936	0.0
C18V5 #1	43054055	0.0	0.0	8.132	0.0	0.0	8.538	0.0
C18V5 #2	25068287	0.0	0.0	12.279	0.0	0.0	13.413	0.0
C18V5 #3	29211238	0.0	0.0	8.767	0.0	0.0	9.757	0.0
C18V5 #4	32281904	0.0	0.0	9.117	0.0	0.0	10.338	0.0
C18V5 #5	40718028	0.0	0.0	10.124	0.0	0.0	10.865	0.0
C22V6 #1	41176718	0.0	0.0	5.943	0.0	0.0	6.396	0.0
C22V6 #2	37236363	0.0	0.0	6.005	0.0	0.0	6.105	0.0
C22V6 #3	38215238	0.0	0.0	4.804	0.0	0.0	5.194	0.0
C22V6 #4	34129809	0.0	0.0	5.274	0.0	0.0	5.896	0.0
C22V6 #5	46379332	0.0	0.0	6.805	0.0	0.0	6.864	0.0
C23V13 #1	41002992	0.0	0.0	8.445	0.0	0.0	8.211	0.0
C23V13 #2	28014147	0.0	0.0	6.298	0.0	0.0	6.748	0.0
C23V13 #3	29090422	0.0	0.0	7.390	0.0	0.0	7.492	0.0
C23V13 #4	33685274	0.0	0.0	7.420	0.0	0.0	7.035	0.0
C23V13 #5	38664843	0.0	0.0	7.474	0.0	0.0	7.454	0.0
C30V6 #1	19227093	0.0	0.0	14.805	0.0	0.0	14.217	0.0
C30V6 #2	16784810	**0.0**	**0.0**	15.710	0.0	0.0	15.073	0.66175
C30V6 #3	21183928	**0.0**	**0.0**	12.475	0.0	0.0	13.078	0.53815
C30V6 #4	21076728	0.0	0.0	15.888	0.0	0.0	15.501	0.0
C30V6 #5	24490671	0.0	0.0	11.917	0.0	0.0	12.424	0.0
C35V7 #1	65082675	0.0	0.0	26.338	0.0	0.0	23.147	0.0
C35V7 #2	54810586	**0.0**	**0.0**	27.417	0.0	0.0	23.777	0.05214
C35V7 #3	56182502	**0.0**	**0.0**	22.123	0.0	0.0	20.805	0.13452
C35V7 #4	61354812	**0.0**	**0.0**	29.734	0.0	0.0	26.212	0.21985
C35V7 #5	63904705	**0.0**	**0.0**	19.162	0.0	0.0	18.330	0.06139
C60V13 #1	80649895	**0.0**	**0.104**	25.668	0.000	0.140	22.010	1.29690
C60V13 #2	74881109	**0.095**	**0.242**	27.629	0.095	0.444	24.000	1.31510
C60V13 #3	91766747	**0.0**	**0.009**	24.640	0.0	0.138	19.918	0.88477
C60V13 #4	89702352	**0.0**	**0.0**	30.243	0.0	0.111	26.527	1.50846
C60V13 #5	88486544	**0.0225**	**0.211**	27.629	0.135	0.237	22.773	1.37665
C80V20 #1	70718084	**0.288**	0.443	51.713	0.305	**0.440**	43.190	1.80696
C80V20 #2	73558165	**0.175**	**0.370**	56.893	0.318	0.472	50.142	1.23300
C80V20 #3	78250612	**0.003**	**0.011**	57.345	0.003	0.015	46.571	0.84580
C80V20 #4	75962439	**0.130**	**0.20**	55.288	0.186	0.449	47.848	0.65009
C80V20 #5	74162521	**0.045**	0.164	56.821	0.111	**0.140**	48.512	1.61271
C100V30 #1	150481912	**0.745**	**0.960**	112.758	0.803	0.971	100.537	2.12407
C100V30 #2	150826322	**0.478**	**0.533**	115.101	0.619	0.663	103.023	2.39899
C100V30 #3	151027805	**0.274**	**0.386**	111.447	0.277	0.410	103.970	0.86054
C100V30 #4	151193009	0.825	0.90	93.323	**0.747**	**0.858**	81.947	3.70926
C100V30 #5	159789021	**0.104**	**0.149**	108.570	0.210	0.330	94.298	1.41639
C130V40 #1	232582224	**1.029**	**1.235**	225.317	1.505	1.554	195.711	3.04106
C130V40 #2	228036360	**1.077**	**1.278**	228.726	1.106	1.393	206.423	2.92569
C130V40 #3	235657072	**0.737**	**1.127**	209.235	1.282	1.451	191.695	3.51395
C130V40 #4	220357686	**1.758**	2.08	201.527	1.885	**2.040**	186.063	3.63937
C130V40 #5	235381937	**1.157**	**1.307**	182.186	1.122	1.334	156.383	4.65875

Table B2: Results on the Hemmati et al (2014) deep sea shipping with mixed cargo load instances

B3: OpenMP Best results for all the Hemmati et al (2014) short sea shipping with full cargo load benchmark instances

Instance	Best Known	OpenMP - PALNS (2023)			OpenMP - ALNS (2023)			Hemmati et al (2014)
		Min Gap %	Average Gap %	Ct^{Avg}	Min Gap %	Average Gap %	Ct^{Avg}	Gap %
C8V3 #1	1391997	0.0	0.0	0.854	0.0	0.0	0.891	0.0
C8V3 #2	1246273	0.0	0.0	0.960	0.0	0.0	0.958	0.0
C8V3 #3	1698102	0.0	0.0	1.005	0.0	0.0	1.026	0.0
C8V3 #4	1777637	0.0	0.0	0.726	0.0	0.0	0.738	0.0
C8V3 #5	1636788	0.0	0.0	0.896	0.0	0.0	0.892	0.0
C11V4 #1	1052463	0.0	0.0	3.499	0.0	0.0	3.672	0.0
C11V4 #2	1067139	0.0	0.0	3.690	0.0	0.0	3.887	0.0
C11V4 #3	1212388	0.0	0.0	3.893	0.0	0.0	4.061	0.0
C11V4 #4	1185465	0.0	0.0	4.115	0.0	0.0	4.364	0.0
C11V4 #5	1310285	0.0	0.0	2.189	0.0	0.0	2.255	0.0
C13V5 #1	2034184	0.0	0.0	4.128	0.0	0.0	4.333	0.0
C13V5 #2	2043253	0.0	0.0	5.298	0.0	0.0	5.717	0.0
C13V5 #3	2378283	0.0	0.0	4.894	0.0	0.0	5.121	0.0
C13V5 #4	2707215	0.0	0.0	3.717	0.0	0.0	4.064	0.0
C13V5 #5	3011648	0.0	0.0	2.605	0.0	0.0	2.698	0.0
C16V6 #1	3577005	0.0	0.0	1.914	0.0	0.0	1.997	0.0
C16V6 #2	3560203	0.0	0.0	1.872	0.0	0.0	2.028	0.0
C16V6 #3	4081013	0.0	0.0	2.404	0.0	0.0	2.573	0.0
C16V6 #4	3667080	0.0	0.0	2.092	0.0	0.0	2.227	0.0
C16V6 #5	3438493	0.0	0.0	2.174	0.0	0.0	2.318	0.0
C17V13 #1	2265731	0.0	0.0	8.219	0.0	0.0	9.170	0.0
C17V13 #2	3154165	0.0	0.0	8.317	0.0	0.0	8.077	0.0
C17V13 #3	2699378	0.0	0.0	9.539	0.0	0.0	9.394	0.0
C17V13 #4	2806231	0.0	0.0	12.903	0.0	0.0	13.374	0.0
C17V13 #5	2910814	0.0	0.0	6.4218	0.0	0.0	6.787	0.0
C20V6 #1	2973381	0.0	0.0	6.746	0.0	0.0	7.619	0.0
C20V6 #2	3206514	0.0	0.0	3.894	0.0	0.0	4.208	0.0
C20V6 #3	3197445	0.0	0.0	5.883	0.0	0.0	6.275	0.0
C20V6 #4	3342130	0.0	0.0	4.976	0.0	0.0	5.379	0.0
C20V6 #5	3156378	0.0	0.0	5.485	0.0	0.0	5.866	0.0
C25V7 #1	3833588	0.0	0.0	13.478	0.0	0.0	14.038	0.0
C25V7 #2	3673666	0.0	0.0	15.778	0.0	0.0	15.576	0.0
C25V7 #3	4238213	0.0	0.0	17.664	0.0	0.0	16.409	0.0
C25V7 #4	4260762	0.0	0.0	15.802	0.0	0.0	15.309	0.0
C25V7 #5	4069693	0.0	0.0	19.992	0.0	0.0	18.753	0.0
C35V13 #1	2986667	0.0	0.0	38.140	0.0	0.0	32.740	0.0
C35V13 #2	3002973	**0.0**	0.0	38.433	0.0	0.0	33.849	0.00003
C35V13 #3	3084339	0.0	0.0	36.203	0.0	0.0	31.662	0.0
C35V13 #4	3952461	0.0	0.0	30.025	0.0	0.0	27.018	0.0
C35V13 #5	3293086	0.0	0.0	36.169	0.0	0.0	32.944	0.0
C50V20 #1	7258266	**0.0**	**0.0**	34.166	0.0	0.0	30.624	0.09501
C50V20 #2	7452465	**0.024**	**0.051**	34.794	0.024	0.075	32.012	0.24180
C50V20 #3	6922293	**0.0**	0.083	35.240	0.0	**0.076**	33.044	0.23079
C50V20 #4	8933846	**0.0**	**0.0**	33.739	0.0	0.0	30.437	0.15084
C50V20 #5	7322307	**0.0**	**0.027**	36.491	0.053	0.064	34.120	0.11021
C70V30 #1	10051856	**0.024**	**0.069**	61.051	0.035	0.088	56.075	0.36587
C70V30 #2	10455468	**0.097**	**0.142**	54.448	0.113	0.164	50.700	0.45437
C70V30 #3	10172541	**0.206**	**0.402**	48.407	0.488	0.713	44.465	1.37651
C70V30 #4	10854036	0.103	**0.145**	54.785	**0.038**	0.223	49.950	0.52055
C70V30 #5	10886838	**0.0**	**0.001**	59.457	0.0	0.008	54.470	0.20022
C90V40 #1	13361947	**0.210**	**0.291**	109.286	0.226	0.329	96.872	0.79368
C90V40 #2	13828112	**0.122**	**0.204**	96.659	0.139	0.301	91.175	1.09926
C90V40 #3	12627125	0.198	**0.273**	111.792	**0.139**	0.307	103.094	1.10114
C90V40 #4	14406428	0.077	**0.092**	115.842	**0.030**	0.152	104.108	0.69315
C90V40 #5	13560830	**0.088**	**0.144**	99.486	0.124	0.203	94.113	1.16349
C100V50 #1	13800823	**0.138**	**0.172**	159.834	0.149	0.221	143.464	0.66517
C100V50 #2	14644836	0.152	**0.214**	158.498	**0.137**	0.247	149.696	0.49948
C100V50 #3	13135505	**0.165**	**0.253**	173.770	0.175	0.270	159.585	0.53802
C100V50 #4	14841840	**0.241**	**0.259**	173.300	0.243	0.267	162.020	0.63174
C100V50 #5	14009874	**0.136**	**0.175**	165.804	0.150	0.253	161.242	0.68667

Table B3: Results on the Hemmati et al (2014) short sea shipping with full cargo load instances

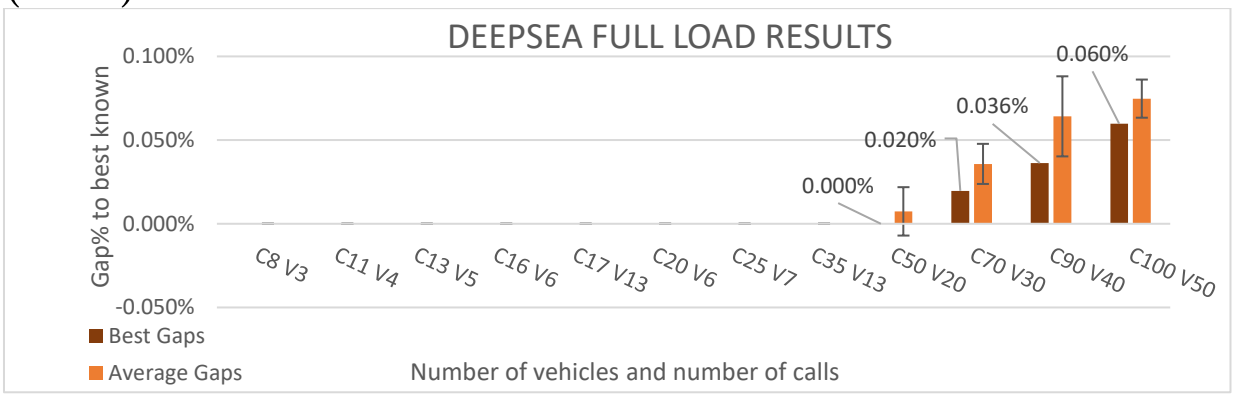
B4: OpenMP Best results for all the Hemmati et al (2014) short sea shipping with mixed cargo load benchmark instances

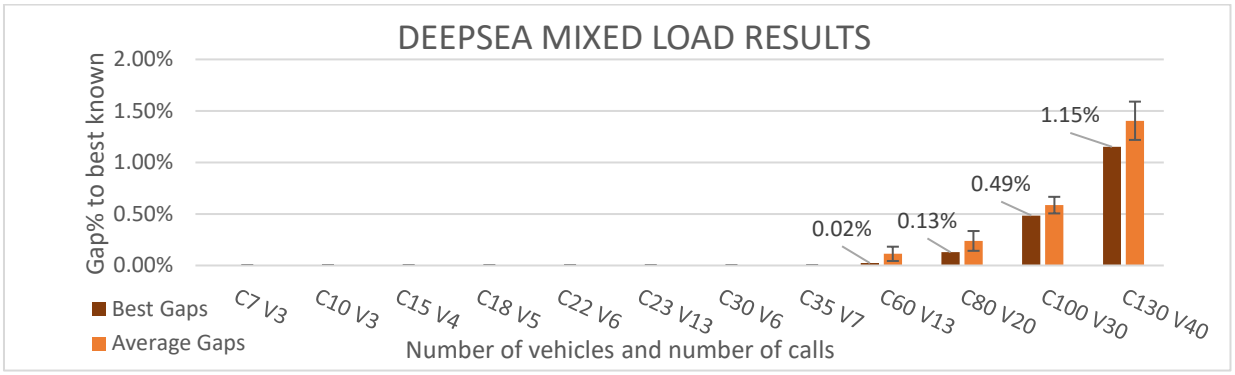
Instance	Best Known	OpenMP - PALNS (2023)			OpenMP - ALNS (2023)			Hemmati et al (2014)
		Min Gap %	Average Gap %	Ct^{Avg}	Min Gap %	Average Gap %	Ct^{Avg}	Gap %
C7V3 #1	1476444	0.0	0.0	0.681	0.0	0.0	0.702	0.0
C7V3 #2	1134176	0.0	0.0	0.753	0.0	0.0	0.751	0.0
C7V3 #3	1196466	0.0	0.0	0.811	0.0	0.0	0.810	0.0
C7V3 #4	1256139	0.0	0.0	0.857	0.0	0.0	0.886	0.0
C7V3 #5	1160394	0.0	0.0	0.764	0.0	0.0	0.770	0.0
C10V3 #1	2083965	0.0	0.0	1.501	0.0	0.0	1.556	0.0
C10V3 #2	2012364	0.0	0.0	1.501	0.0	0.0	1.880	0.0
C10V3 #3	1986779	0.0	0.0	1.729	0.0	0.0	1.832	0.0
C10V3 #4	2125461	0.0	0.0	1.938	0.0	0.0	2.013	0.0
C10V3 #5	2162453	0.0	0.0	1.237	0.0	0.0	1.252	0.0
C15V4 #1	1959153	0.0	0.0	8.509	0.0	0.0	9.434	0.0
C15V4 #2	2560004	0.0	0.0	7.157	0.0	0.0	7.557	0.0
C15V4 #3	2582912	0.0	0.0	6.549	0.0	0.0	7.007	0.0
C15V4 #4	2265396	0.0	0.0	8.04	0.0	0.0	8.782	0.0
C15V4 #5	2230861	0.0	0.0	5.919	0.0	0.0	6.466	0.0
C18V5 #1	2374420	0.0	0.0	9.314	0.0	0.0	10.080	0.0
C18V5 #2	2987358	0.0	0.0	14.685	0.0	0.0	16.769	0.0
C18V5 #3	2301308	0.0	0.0	9.981	0.0	0.0	10.892	0.0
C18V5 #4	2400016	0.0	0.0	11.118	0.0	0.0	11.598	0.0
C18V5 #5	2813167	0.0	0.0	12.932	0.0	0.0	13.115	0.0
C22V6 #1	3928483	0.0	0.0	4.856	0.0	0.0	5.001	0.0
C22V6 #2	3683436	0.0	0.0	6.419	0.0	0.0	6.708	0.0
C22V6 #3	3264770	0.0	0.0	7.941	0.0	0.0	8.512	0.0
C22V6 #4	3228262	0.0	0.0	8.474	0.0	0.0	9.154	0.0
C22V6 #5	3770560	0.0	0.0	6.397	0.0	0.0	6.616	0.0
C23V13 #1	2276832	0.0	0.0	9.094	0.0	0.0	9.507	0.0
C23V13 #2	2255870	0.0	0.0	8.464	0.0	0.0	8.890	0.0
C23V13 #3	2362503	0.0	0.0	10.047	0.0	0.0	9.884	0.0
C23V13 #4	2250110	0.0	0.0	13.838	0.0	0.0	13.459	0.0
C23V13 #5	2325941	0.0	0.0	8.708	0.0	0.0	9.021	0.0
C30V6 #1	4958542	0.0	0.0	11.251	0.0	0.0	11.202	0.0
C30V6 #2	4549708	**0.0**	0.0	14.256	0.0	0.0	13.331	0.4096
C30V6 #3	4098111	**0.0**	0.0	15.03	0.0	0.0	14.901	0.1993
C30V6 #4	4449449	**0.0**	0.0	15.678	0.0	0.0	15.023	0.0082
C30V6 #5	4528514	0.0	0.0	10.349	0.0	0.0	10.628	0.0
C35V7 #1	4893734	**0.0**	0.0	19.382	0.0	0.016	18.244	0.9853
C35V7 #2	4533265	**0.0**	0.0	23.964	0.0	0.0	22.038	0.2286
C35V7 #3	4433847	0.0	0.0	22.226	0.0	0.0	20.687	0.0
C35V7 #4	4580935	0.0	0.0	25.313	0.0	0.0	22.759	0.0
C35V7 #5	5511661	0.0	0.0	19.047	0.0	0.0	17.924	0.0
C60V13 #1	8133385	**0.0**	0.019	39.765	0.0	**0.007**	35.400	0.8382
C60V13 #2	7971476	**0.164**	**0.217**	39.684	0.265	0.285	34.098	1.0488
C60V13 #3	7604198	**0.0**	**0.097**	40.770	0.0	0.118	36.189	0.6206
C60V13 #4	8505125	**0.038**	0.306	38.450	0.045	**0.271**	34.419	1.0274
C60V13 #5	8921750	**0.0**	**0.159**	30.275	0.0	0.253	26.642	0.3162
C80V20 #1	10289573	**0.051**	**0.200**	60.244	0.120	0.283	51.491	0.8367
C80V20 #2	10240618	0.441	0.620	67.077	**0.360**	**0.518**	57.088	1.4117
C80V20 #3	9606530	**0.0**	**0.078**	67.474	0.075	0.135	57.893	1.6067
C80V20 #4	11302476	**0.179**	**0.375**	54.793	0.363	0.447	47.731	2.5516
C80V20 #5	10862563	**0.021**	0.230	52.967	0.172	**0.215**	45.135	1.0976
C100V30 #1	12626988	**0.003**	**0.130**	143.837	0.131	0.255	130.054	1.7018
C100V30 #2	12774864	**0.153**	**0.445**	124.288	0.504	0.621	105.177	2.1648
C100V30 #3	11935332	**0.154**	**0.261**	136.124	0.267	0.373	124.744	1.2666
C100V30 #4	13605352	**0.170**	0.313	136.097	0.199	**0.290**	124.984	1.3527
C100V30 #5	13240648	0.218	0.331	120.811	**0.124**	**0.284**	109.645	1.9410
C130V40 #1	16316051	**0.334**	**0.429**	263.548	0.407	0.478	240.106	1.2596
C130V40 #2	16260579	0.750	**0.847**	228.458	**0.701**	0.900	209.634	2.7074
C130V40 #3	15537963	**0.540**	**0.784**	252.348	0.734	0.822	229.759	2.0438
C130V40 #4	17011065	**0.420**	**0.591**	248.563	0.631	0.740	226.096	1.7033
C130V40 #5	18273893	0.422	**0.546**	235.083	**0.400**	0.548	208.515	1.3996

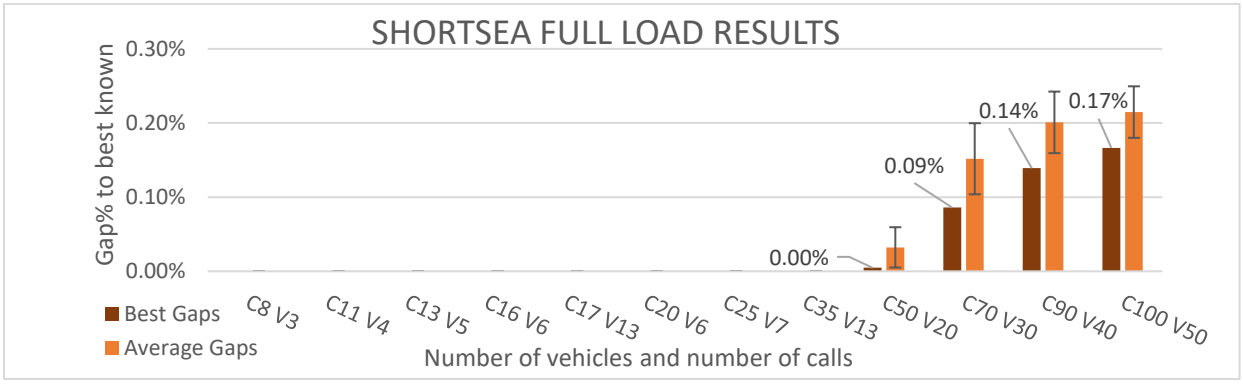
Table B4: Results on the Hemmati et al (2014) short sea shipping with mixed cargo load instances

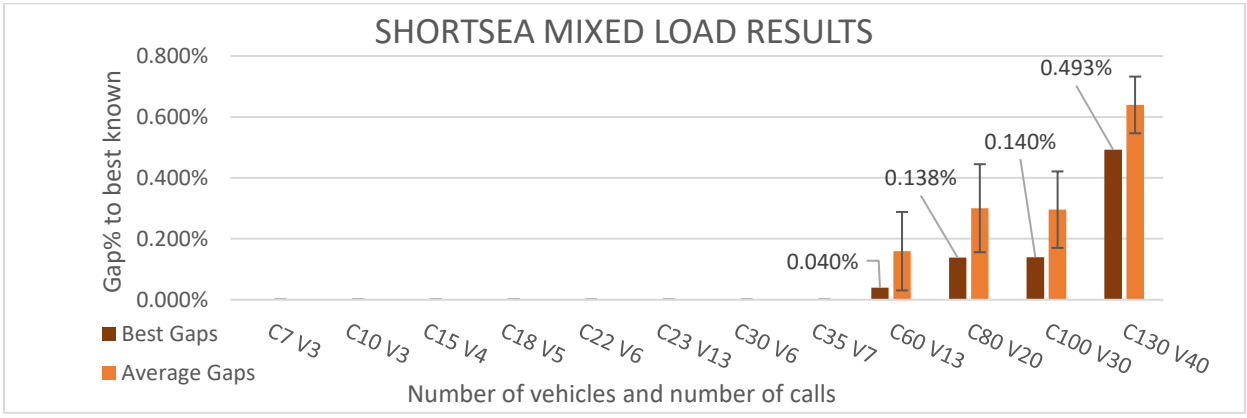
Appendix C. Additional result bar plots

C1: OpenMP results of PALNS on the Hemmati et al (2014) PDPTW benchmark instances

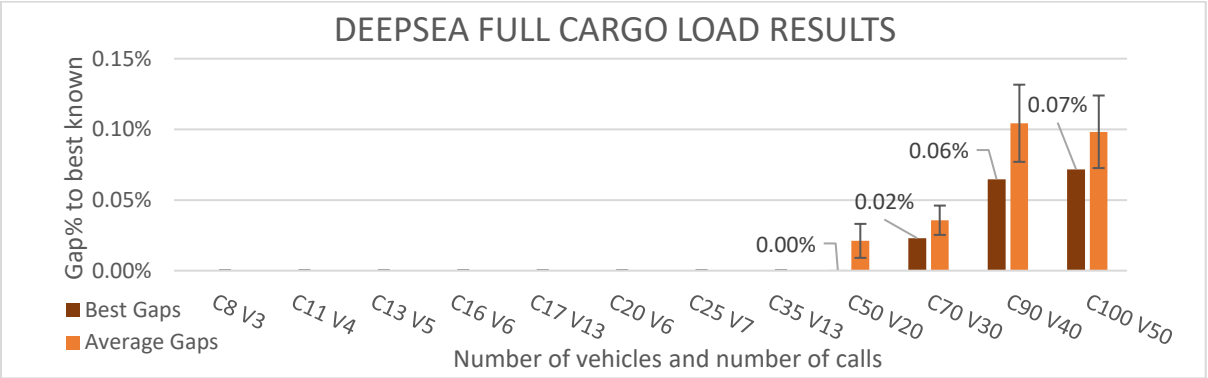


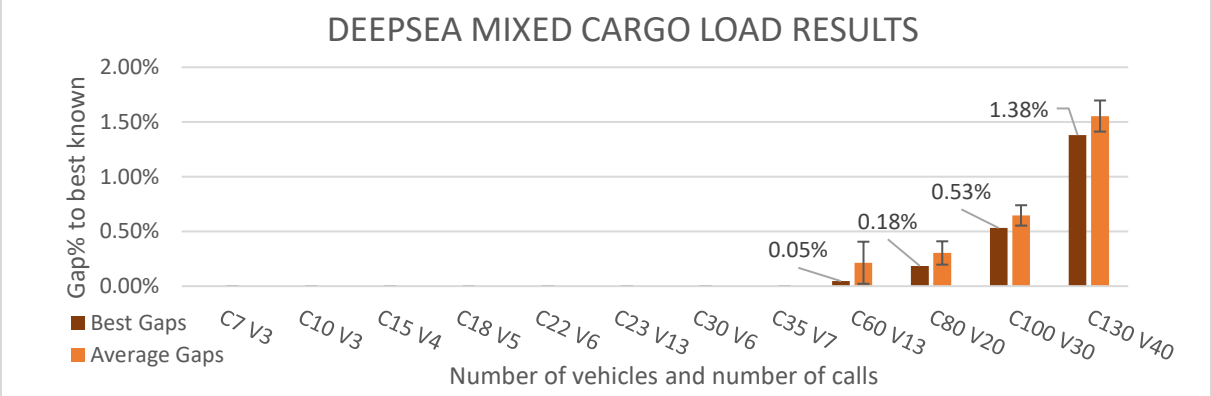


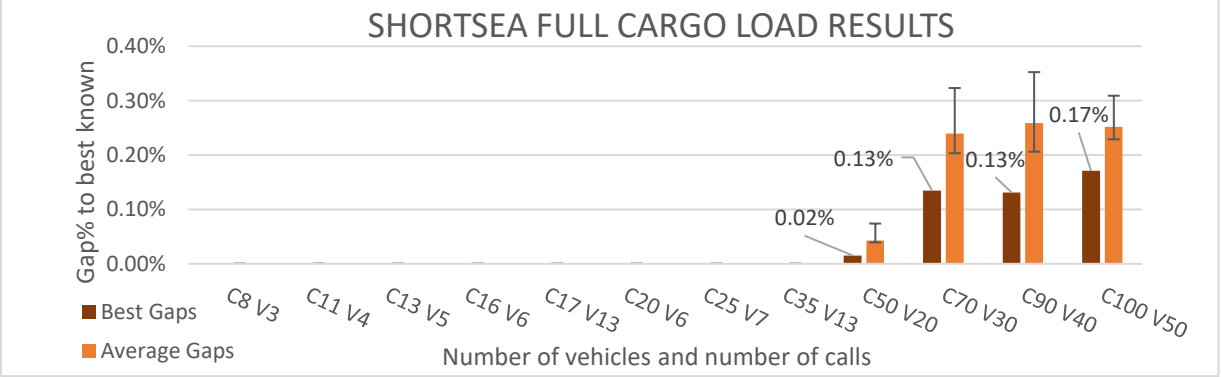


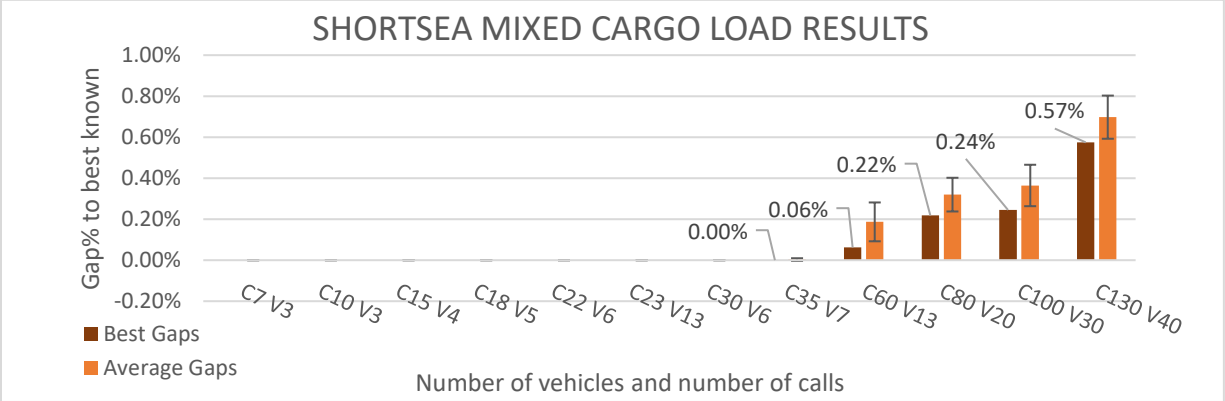


C2: OpenMP results of ALNS on the Hemmati et al (2014) PDPTW benchmark instances

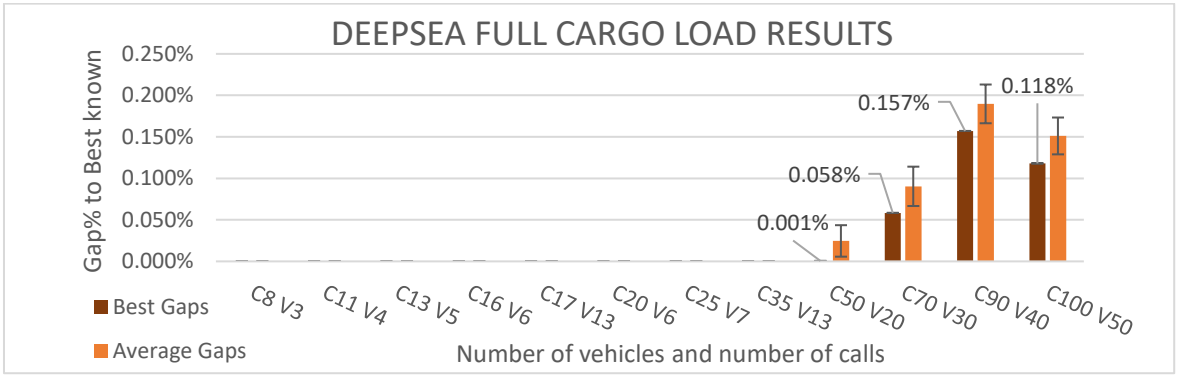


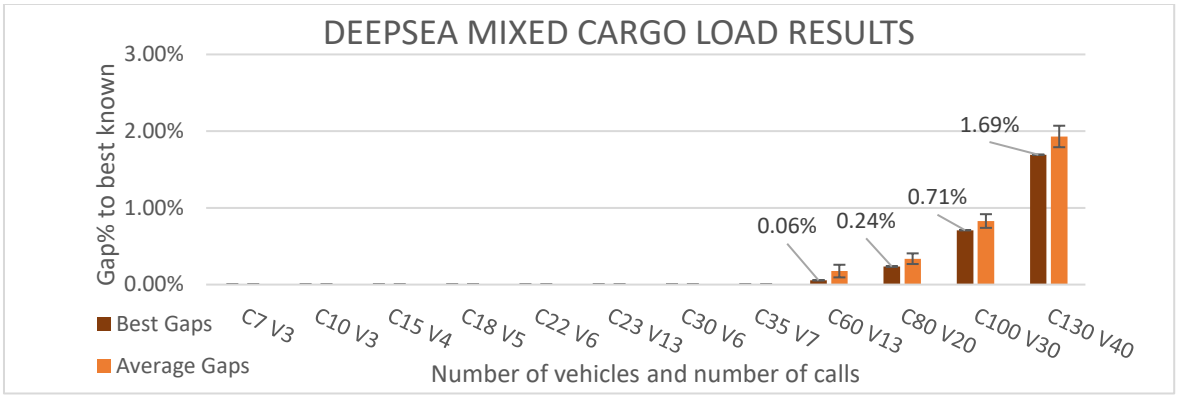


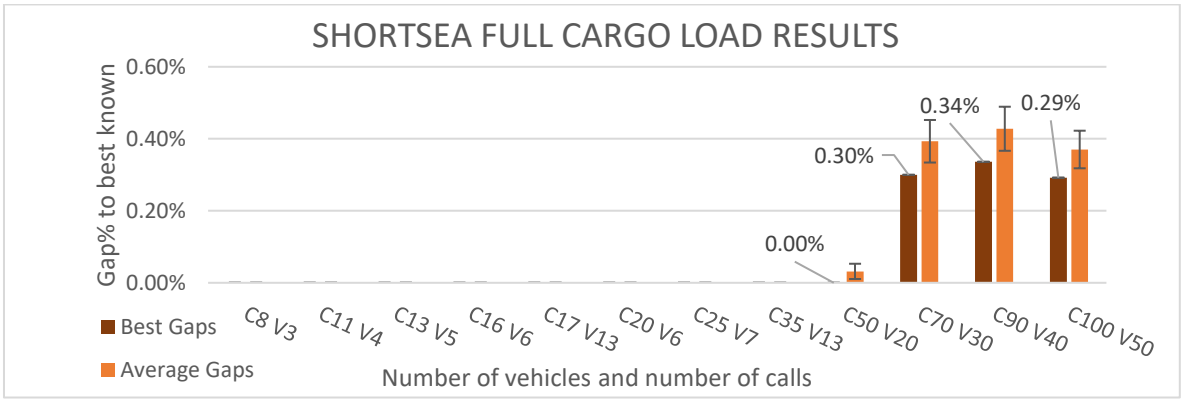


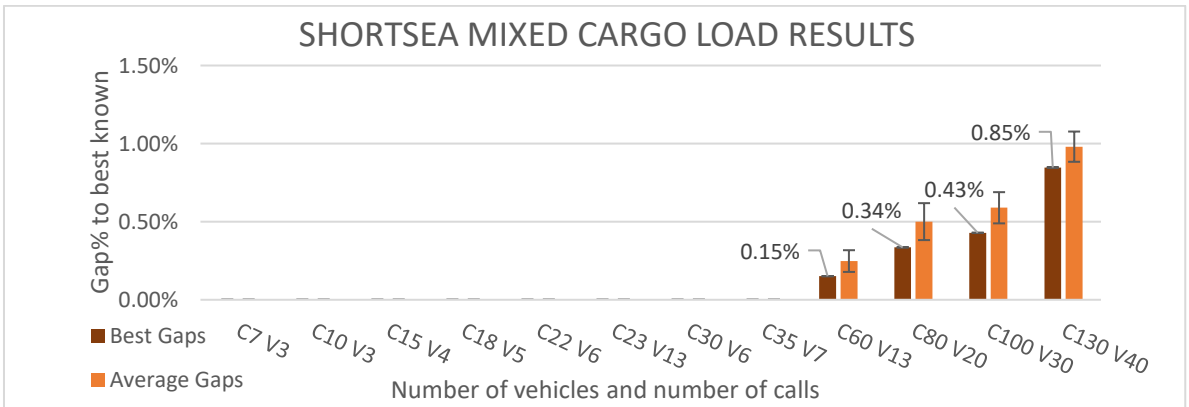


C3: CUDA results of PALNS on the Hemmati et al (2014) PDPTW benchmark instances









C4: CUDA results of ALNS on the Hemmati et al (2014) PDPTW benchmark instances