David Fernandes Semedo

Licenciado em Engenharia Informática

A Public Transport Bus Assignment Problem: Parallel Metaheuristics Assessment

Dissertação para obtenção do Grau de
Mestre em Engenharia Informática

Orientador:	Prof. Doutor Pedro Barahona, Prof. Catedrático, Universidade Nova de Lisboa
Co-orientador:	Prof. Doutor Pedro Medeiros, Prof. Associado, Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutor José Augusto Legatheaux Martins
Arguente:	Prof. Doutor Salvador Luís Bettencourt Pinto de Abreu
Vogal:	Prof. Doutor Pedro Manuel Corrêa Calvente de Barahona

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2015

A Public Transport Bus Assignment Problem: Parallel Metaheuristics Assessment

Aos meus pais.

Acknowledgements

This research was partly supported by project "RtP - Restrict to Plan", funded by FEDER (*Fundo Europeu de Desenvolvimento Regional*), through programme COMPETE - POFC (*Operacional Factores de Competitividade*) with reference 34091.

First and foremost, I would like to express my genuine gratitude to my advisor professor Pedro Barahona for the continuous support of my thesis, for his guidance, patience and expertise. His general optimism, enthusiasm and availability to discuss and support new ideas helped and encouraged me to push my work always a little further. Additionally, he was able to provide me financial support allowing me to keep focused on my work, which I believe otherwise would not have been possible.

Additionally, I would like to give a very special thanks to my co-advisor professor Pedro Medeiros for his utmost predisposition for any doubt on parallel architectures and implementations aspects, whose ideas and knowledge made possible the results achieved.

Second, I would like to thank Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa not only for providing me all the resources needed along the course but also for his excellent environment, in which I was able to forge valuable friendships. Moreover, I thank my department, Departamento de Informática, and its associated research center NOVA-LINCS, for the excellent professors which whom I had the opportunity to learn from, specially for its close relation with the students, and for supporting my attendance at the INFORUM 2015 conference.

My sincere thanks to Marco Correia, for all his contributions to my work, in the development of the CaSPER LS solver and modelling of the vehicle scheduling problem, and for is insightful suggestions regarding implementation and modelling aspects.

To professor Jorge Cruz for his precious suggestions on the modelling of the problem.

To all my colleagues, for sharing both stressful and relaxed moments, for providing me good moments and for giving me motivation even without knowing it. A special thanks to my colleague Alexandre Garcia, for sharing with me some of the most stressful moments during the course, for our wonderful projects, and for our discussions related with our works, which always helped clarifying and confirming ideas.

To my high school friends, Carlos Crespo, Diana Coimbra, Iolanda Fortes and João

Mirão, for their friendship and for the moments we spent together (including all the silly ones ☺) which always helped me relax and regain strengths. Carlos and Mirão, thank you for our CS:GO nights, which I am sure ☺ also contributed to this work.

To Olga, Pedro, Ana, Zé, Lurdes, Beatriz and Nuno for all their presence and support, and for helping me more than they can imagine.

To all my family, specially to my grandparents, who were always there to help me, despite not having the chance of seeing me very often. A huge thanks to my parents which despite all the difficulties, they always believed in me and gave me infinite support. Concretely, to my father which was always interested in my work and for being my inspiration, and to my mother for simply being my mother, for all is guidance through my life and for being ever-present even when it was difficult for her. To my warrior little sister which whom I shared marvellous moments, doing our things, and for inspiring me everyday. Words are not enough to express my gratitude.

The most deepest thanks to my beloved girlfriend, Marta Rolho, for these wonderful six years, full of obstacles for both sides, which we surpassed together, and with whom I learned the meaning of the world love. Without all the support, encouragement and friendship she provided me I would not have finished this thesis.

Abstract

Combinatorial Optimization Problems occur in a wide variety of contexts and generally are NP-*hard* problems. At a corporate level solving this problems is of great importance since they contribute to the optimization of operational costs. In this thesis we propose to solve the Public Transport Bus Assignment problem considering an heterogeneous fleet and line exchanges, a variant of the Multi-Depot Vehicle Scheduling Problem in which additional constraints are enforced to model a real life scenario.

The number of constraints involved and the large number of variables makes impracticable solving to optimality using complete search techniques. Therefore, we explore metaheuristics, that sacrifice optimality to produce solutions in feasible time. More concretely, we focus on the development of algorithms based on a sophisticated metaheuristic, Ant-Colony Optimization (ACO), which is based on a stochastic learning mechanism.

For complex problems with a considerable number of constraints, sophisticated metaheuristics may fail to produce quality solutions in a reasonable amount of time. Thus, we developed parallel shared-memory (SM) synchronous ACO algorithms, however, synchronism originates the *straggler problem*. Therefore, we proposed three SM asynchronous algorithms that break the original algorithm semantics and differ on the degree of concurrency allowed while manipulating the learned information.

Our results show that our sequential ACO algorithms produced better solutions than a Restarts metaheuristic, the ACO algorithms were able to learn and better solutions were achieved by increasing the amount of cooperation (number of search agents). Regarding parallel algorithms, our asynchronous ACO algorithms outperformed synchronous ones in terms of speedup and solution quality, achieving speedups of $\approx$ 17.6x. The cooperation scheme imposed by asynchronism also achieved a better learning rate than the original one.

Keywords: Metaheuristics, Parallelism, Vehicle Scheduling Problem, Ant-Colony Optimization, Local Search, Asynchronism, Cooperative Search.

RESUMO

Problemas de Optimização Combinatória estão presentes em diversos contextos e são regra geral NP-*hard*. É importante resolver estes problemas a nível empresarial de forma a optimizar os custos operacionais. Nesta dissertação iremos resolver o problema de Alocação de Autocarros num Serviço de Transportes Públicos considerando uma frota heterogénea e mudanças de linha, uma variante do problema de escalonamento de veículos com múltiplos depósitos, com restrições adicionais para modelar um cenário real.

O grande número de restrições e de variáveis tornam impraticável a utilização de técnicas que garantam a otimalidade das soluções. Como tal, exploram-se metaheurísticas, métodos que sacrificam otimalidade, de forma a produzir soluções em tempo útil. Focamo-nos no desenvolvimento de algoritmos baseados numa metaheurística sofisticada, Ant-Colony Optimization (ACO), que tem por base um mecanismo de aprendizagem estocástico.

Para problemas complexos com um elevado número de restrições, metaheurísticas sofisticadas poderão não conseguir produzir boas soluções em tempo útil. Como tal, desenvolveram-se algoritmos de ACO paralelos síncronos em memória partilhada (MP), no entanto, o síncronismo causa *straggler problem*. Assim, propomos três algoritmos assíncronos em MP, que alteram a semântica do algoritmo original e diferem no grau de concorrência permitido na manipulação da informação recolhida.

Os nossos resultados demonstram que os nossos algoritmos sequenciais de ACO produzem melhores soluções que a metaheurística de Restarts, que são capazes de aprender obtendo-se melhores soluções aumentando o nível de cooperação (número de agentes). Em relação aos algoritmos paralelos, os algoritmos assíncronos de ACO apresentaram melhores resultados em termos de speedup e qualidade da solução do que os síncronos, com speedups na ordem de $\approx$ 17.6x. Com o esquema de cooperação imposto pelo assíncronismo também se obteve uma melhor taxa de aprendizagem em relação ao original.

Palavras-chave: Meta-Heurísticas, Paralelismo, Problema de Alocação de Veículos, Ant-Colony, Pesquisa Local, Assíncronismo, Pesquisa Cooperativa.

CONTENTS

List of Figures

LIST OF TABLES

List of Algorithms

GLOSSARY

ACO Ant-Colony Optimization.

ACS Ant Colony System.

API Application Programming Interface.

AS Ant System.

ASSync Synchronous Ant System.

AsyncBC Asynchronous Blocking Column Ant System.

AsyncBM Asynchronous Blocking Matrix Ant System.

AsyncLF Asynchronous Lock-Free Ant System.

CBLS Constraint-Based Local Search.

COMA Cache-Only Memory Architecture.

COP Combinatorial Optimization Problem.

CP Constraint-Programming.

CPU Central Processing Unit.

CSOP Constraint Satisfaction and Optimization Problem.

CSP Constraint Satisfaction Problem.

GPU Graphical Processing Unit.

L1 Level 1.

L2 Level 2.

LS Local Search.

MIMD Multiple Instruction Stream Multiple Data Stream.

MISD Multiple Instruction Stream Single Data Stream.

MP MultiProcessor.

MS Multi-start.

MW Multiple-Walk.

NUMA Non-Unified Memory Architecture.

ParallelPTBAP-OptLib Parallel PTBA Problem Optimization Library.

PC Personal Computer.

PTBA Public Transport Bus Assignment.

QAP Quadratic Assignment Problem.

RAM Random-Access Memory.

SIMD Single Instruction Stream Multiple Data Stream.

SISD Single Instruction Stream Single Data Stream.

S-metaheuristics Single-solution based Metaheuristics.

SM Shared-Memory.

SQL Structured Query Language.

SW Single-Walk.

TSP Travelling Salesman Problem.

UMA Unified Memory Architecture.

VSP Vehicle Scheduling Problem.

CHAPTER 1

INTRODUCTION

This chapter introduces the context and the motivation surrounding this work in section 1.1. Then the objectives of our work will be presented in section 1.2 regarding the development of an algorithm to solve the Public Transport Bus Assignment problem and the exploration of parallelism.

After presenting the objectives, we present in section 1.3 the problem which this work tackles. More concretely, we formulate and describe the vehicle scheduling problem and enumerate the parallel metaheuristic hypothesis.

Lastly, the main contributions are specified and the document organization will be described, in sections 1.4 and 1.5, respectively.

1.1 Context and Motivation

Combinatorial Optimization Problems (COPs) occur in a wide variety of contexts, including science, business, industry, among others. These problems are very demanding in terms of computational complexity (in general *NP-Hard* [31]), which makes human solving highly impractical. Nevertheless, at a corporate level, solving these problems is of great importance, since it allows corporations to optimize their operational resources/-costs. Many efficient decision support tools which automate the solving process, emerged, allowing companies to maximize their profits by applying an efficient resources/costs management.

In general, COPs can be solved with two different methods: complete search methods or incomplete/approximative methods. Even though complete search methods guarantee an optimal solution, they do not scale. Approximative methods, in particular metaheuristics [8, 72], do not guarantee an optimal solution but, by sacrificing completeness, are able to obtain near-optimal solutions, in a reasonable amount of time, even for large

instances. However, as the problem instances become very large, metaheuristic methods performance tends to degrade.

Solving large instances of COPs in reasonable time is very important, not only at a corporate level, but also in several fields of science like physics, biology and chemistry (e.g. sequencing and assembling of DNA chains [11]), contributing for progress.

The performance of the methods for solving COPs depend not only on how suited the method is to a specific problem (i.e. the method characteristics vs. the problem characteristics), but also on the hardware being used and how it is exploited.

Nowadays, due to technological limits, the processing power of Central Processing Units (CPUs) does not increase by increasing the clock speed, but instead, by increasing the number of transistors and the number of cores on the CPU (multi-core CPUs). Multiprocessors systems follow the same trend by using two or more processors (single or multi-core processors) in order to achieve superior processing power. Moore's law [58], which states that the number of transistors in a CPU would double approximately every two years, is still valid and it is worth exploring parallelism to increase applications performance.

Methods for solving COPs can exploit this degree of parallelism (from multi-core and multiprocessors systems) in order to increase their effectiveness in terms of computation time, solution quality or both. By reducing computation time, larger instances can be addressed. Since real life COPs tend to be very complex, involving a large number of variables, it is important to use all the potential parallelism in order to design solving methods capable of tackling these type of problems.

This thesis addresses a real life problem that Public Transport Services companies face, the Public Transport Bus Assignment (PTBA) problem , a Vehicle Scheduling Problem (VSP) variant with additional constraints, which comprises a large number of variables, resulting in a very complex and CPU time-consuming problem. The problem is described in detail in section 1.3.1.

Due to the promising features of Constraint-Based Local Search [76], which will be highlighted in section 2.2, we developed a sequential Constraint-Based Local Search (CBLS) solver: CaSPER LS [68]. CBLS solvers allow one to model complex COPs in a declarative manner, therefore simplifying the process of developing effective metaheuristic algorithms which not only attempt to maximize/minimize a given objective function but also attempt to minimize the number of constraints violations.

1.2 Objectives

The main goal of this thesis is to develop an effective algorithm for solving the PTBA problem, i.e., an algorithm capable of solving real life instances in a reasonable amount of time, while producing quality solutions. The variant addressed in this thesis captures several details that occur in a real life setting that are left out from the original VSP and common variants addressed in the literature.

Initially our purpose is to develop a Local Search (LS) algorithm which a search agent can use to build complete solutions, satisfying all the constraints. This algorithm is an essential component to develop more intelligent metaheuristic-based algorithms.

Based on this algorithm we intend to develop an intelligent algorithm capable of escaping local optima and effectively explore the search space. The Ant-Colony Optimization (ACO) metaheuristic [22] will be used to achieve this goal.

Current multi-core/multiprocessor systems offer a high degree of parallelism. In this thesis, different parallel strategies targeting *shared-memory* (SM) architectures for different local search metaheuristics will be analysed in order to develop improved, efficient and intelligent search techniques that not only take full advantage of the available hardware, but also of the problem characteristics. More concretely, it is our objective to perform the following tasks:

Parallel strategies - Analyse and develop improved parallel strategies for the metaheuristics used for the sequential implementations, namely the Ant-Colony Optimization metaheuristic. We will analyse the possibility the asynchronous approach since it mitigates the straggler problem, which will be presented on section 2.6. Concretely we intend to design ant test asynchronous strategies for the ACO metaheuristic;

Cooperation between processes - Study how can different search processes cooperate in order to reach higher quality solutions and assess if cooperation improves solutions quality.

The effectiveness of resulting techniques will be experimentally evaluated and tested against a real life instance with a considerable number of departures.

The best algorithm will be incorporated on a decision support tool that is being developed in the context of the research project "RtP - Restrict to Plan" under development by the research centre NOVA LINCS at Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa. This project aims to develop a state-of-the-art decision support tool for solving the three main problems of a public transport bus service: producing a time-table of departures for all the bus lines, assigning vehicles to each departure and finally assigning drivers to each $\langle departure, vehicle \rangle$ assignment.

1.3 Problem Description

Complete search techniques like Constraint Programming (CP) [64], Integer Programming [59], tree search methods [65] (e.g. A*) and others, guarantee optimality. However, they all require exponential computational time complexity, making it impossible to obtain a solution in a reasonable amount of time.

By relaxing the optimality requirement, one can use approximative methods that are able to achieve near-optimal solutions in a reasonable amount of time. Metaheuristics [32] are approximative methods that have been extensively applied to COPs [1]. As discussed

in section 2.1.1, the effectiveness of metaheuristics is in part directly related to its capability of escaping local optima in order to effectively explore the search space. The most simple metaheuristics (e.g. Restarts and Multi-Starts, discussed in section 2.1.2.1) may fail to achieve this objective, therefore, more sophisticated metaheuristics should be considered. Even with sophisticated metaheuristics bad results may be observed since they may not be suited for a given problem. Therefore, it is important to analyse each metaheuristic characteristics and compare with the problem characteristics in order choose adequately.

Nevertheless, sophisticated metaheuristics demand higher computation times. For complex problems with large search spaces and with a considerable number of constraints, as the one we address in this work, they may also fail to find quality solutions in a reasonable amount of time.

Metaheuristics performance can be improved by exploring parallelism offered by parallel architectures. Taking advantage of all the hardware resources is crucial in order to design algorithms that are able to address complex problems and to scale (in terms of instances size). Additionally, parallel computing may improve not only the performance but also the solution quality. In some parallel strategies, search agents execute a given algorithm in order to build a solution to the problem. Unlike the sequential scenario case, parallel algorithms introduce an interesting natural behaviour: cooperation between search agents. Search agents may exchange knowledge collected during the search in order to achieve a better exploration of the search space by indirectly guiding subsequent searches towards promising regions.

1.3.1 Public Transport Bus Assignment Problem

The Public Transport Bus Assignment (PTBA) problem is a combinatorial optimization problem arising in transit services companies, which consists of producing a schedule for each vehicle such that each trip is covered by one and only one vehicle and the operational costs are minimized, whilst satisfying a set of constraints. The PTBA problem is a variant of the more general Vehicle scheduling problem [30] in which several depots are considered and line exchanges are allowed. Additionally, an heterogeneous fleet of vehicles is considered (i.e. there are several types of vehicles each with different characteristics and constraints).

The problem can be formulated as follows:

> Let L be a set of lines to be covered, P be a set of terminals and T be a set of n time-tabled departures $T_1, \ldots, T_n$, such that for each $j \in [1, n]$, s_j and e_j denote the starting and ending time, respectively, it_j, $ft_j \in P$ denote the initial and final terminal, respectively, and $l_j \in L$ the corresponding line of departure j. Let D be a set of m depots $D_1, \ldots, D_m$ where each depot D_k has a total capacity of r_k (with $r_k \leq n$) vehicles.
>
> Let $\tau_{i,j}$ be the travel time for a vehicle to go from the arrival terminal of departure T_i to the terminal of departure T_j, where $i, j \in [1, n]$. An ordered pair of trips

(T_i, T_j) is said to be *compatible* iff the same vehicle can cover trips T_i and T_j in the sequence, i.e, the following inequality is verified:

$$e_i + \epsilon_{min} + \gamma_{i,j} \leq s_j \tag{1.1}$$

where ϵ_{min} is the minimum time between an arrival and a departure, in order to allow preparation of the vehicle and/or perform a driver exchange.

Let $term_i$ be a terminal such that $term_i \in P$. At every moment in time, only $maxNumVehiclesAtTerminal_i$ vehicles can be parked at $term_i$ and the duration in which a vehicle can be parked at a $term_i$ cannot exceed $maxDurAtTerminal_i$. A vehicle service is interrupted when a vehicle is idle at a terminal or at a depot (and already performed some work) for more than $maxIdleTimeInService$ minutes.

A vehicle *scheduling* S_i, performed by a vehicle v_i stationed at depot D_k, consists of a sequence of tasks. Each task consists in a tuple $\langle v\_id, il, fl, smin, emin, type \rangle$ where:

- v_id denotes the vehicle identifier;
- il and fl are the initial and final locations, respectively, where $il, fl \in P \cup D$;
- $smin$ and $emin$ are the starting and the ending minute, respectively. For departure tasks , $smin = s_j$ and $emin = e_j$ for a given departure T_j;
- $type \in \{InService, Wait, VoidIn, VoidOut, LineExchange, Maintenance, ServiceInterrupt\}$ and corresponds to the type of the task.
  $InService$ tasks correspond to trips with passengers, $Wait$ tasks correspond to a waiting state in which the vehicle is idle, $VoidIn/VoidOut$ tasks correspond to trips performed without passengers from/to a depot, $LineExchange$ corresponds to the situation where a vehicle performs a line exchange, $Maintenance$ corresponds to a maintenance task where a vehicle performs maintenance and $ServiceInterrupt$ correspond to a service interruption.

Vehicle schedules cannot have more than $maxNumInterrupts$ service interrupts.

Each type of vehicle has the following constraints associated:

- Maximum daily working time;
- Maximum time between two maintenance tasks;
- Minimum time of a maintenance task;
- Set of lines in which the vehicle is allowed to perform departures.

Additionally, for each type of vehicle, the following parameters are considered

- Investment cost (€/day) of the vehicle, measured by its daily depreciation;

- Working cost (€/day) of the vehicle (i.e. fuel and maintenance costs).

Given the specification above, the problem consists in finding an assignment of vehicles to departures such that:

i) each departure is covered by exactly one vehicle;

ii) each vehicle is assigned to a sequence of pairwise *compatible* departures, starting from and returning to its depot at the end of the *schedule*;

iii) the number of vehicles in a depot D_k and used in the solution does not exceed r_k, for $k \in [1, \ldots, m]$;

iv) each vehicle waits ϵ_{min} minutes when arrives at a terminal before performing the next departure;

v) the operational costs of the assignment are minimized.

We consider that each PTBA problem instance corresponds to a period of one day, where the services are the routes specified in a fixed timetable for that day. Furthermore, line exchanges are taken into account (vehicles are not bound to a single service line) although each vehicle is still bound to only one depot.

In our formulation, a vehicle travels along service lines starting and ending in the assigned depot. No interruptions occur during a trip between two terminals. Sporadically, a vehicle can return to his corresponding depot either for maintenance or in case no more services are allocated to that vehicle.

Vehicles can stay at a terminus $term_i$ waiting for a departure (idle time), to avoid travelling to the depot, if a spot on $term_i$ is available and if the waiting time does not exceed $maxDurAtTerminal_i$. When the waiting time exceeds $maxIdleTimeInService$ the service is interrupted which means that the vehicle is not wasting working time.

Objective Function. The objective function F to be minimized takes into account the investment cost of each vehicle measured by its daily depreciation (€/day) (including the cost of maintenance episodes) and the running cost of the total distance travelled by each vehicle. Each of the costs referred change according to the vehicle type.

Let Nb be the total number of vehicles used (i.e. the total number of schedules S_i in the solution). The function F of a solution sol is defined as follows:

$$F(sol) = \sum_{i=1}^{Nb} (distTravelled(v_i) * distTypeCost(v_i) + dailyTypeCost(v_i)) \qquad (1.2)$$

where $distTravelled(v_i)$ is the total distance travelled by the vehicle v_i in the solution sol, $distTypeCost(v_i)$ and $dailyTypeCost(v_i)$ is the running cost and the daily depreciation, respectively, of vehicles from the type of vehicle v_i. The unused vehicles slots in each depot D_k do not contribute to the total cost.

For $m \geq 2$ this optimization problem is classified as an NP-Hard problem [5, 47], therefore, unless $P = NP$, cannot be solved in polynomial time. The proof is omitted in this document and can be seen in [5]. When $m = 1$ the original problem can be solved in polynomial time [47].

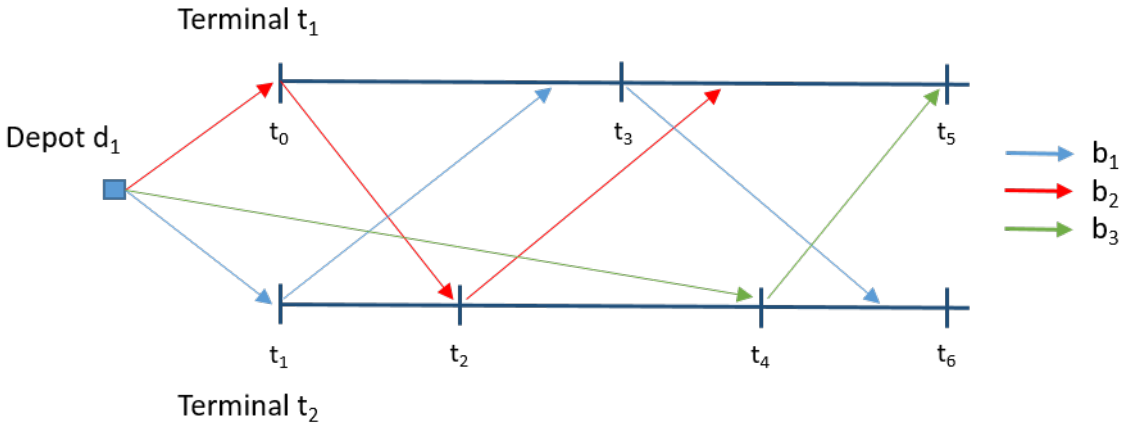


Figure 1.1: Partial Assignment Example of the PTBA problem.

Partial Assignment example. Figure 1.1 illustrates a partial assignment example for one service line $l1$ with n departures. The depots are omitted and all vehicles belong to the depot d_1. In this example we have:

- $D = \{d_1\}$ with $r_1 = \infty$;
- $T = \{T_0, T_1, T_2, T_3, T_4, \ldots, T_n\}$, with starting times $t_0, t_0, t_1, t_2, t_3, \ldots, t_n$ and ending times $e_0, e_1, e_2, e_3, e_4, e_5, \ldots, e_n$;
- Let $B = \{b_1, b_2, b_3, \ldots\}$ the set of vehicles used;
- $Solution = \{s_1, s_2, s_3, \ldots\}$, where:
  - $s_1 = \{\langle b_1, d_1, t_0, -, s_0, VoidIn\rangle, \langle b_1, t_0, t_1, s_0, e_0, InService\rangle, \langle b_1, t_1, t_1, s_0, s_2, Wait\rangle, \langle b_1, t_1, t_0, s_2, e_2, InService\rangle, \langle b_1, t_0, d_1, e_2, -, VoidOut\rangle, \ldots\}$
  - $s_2 = \{\langle b_2, d_1, t_1, -, s_1, VoidIn\rangle, \langle b_1, t_1, t_0, s_2, e_2, InService\rangle, \langle b_2, t_0, t_1, e_2/s_3, e_3, InService\rangle, \langle b_2, t_1, d_1, e_3, -, VoidOut\rangle, \ldots\}$
  - $s_3 = \{\langle b_3, d_1, t_1, -, s_4, VoidIn\rangle, \langle b_3, t_1, t_0, s_4, e_4, InService\rangle, \langle b_3, t_0, d_1, e_4, -, VoidOut\rangle, \ldots\}$

  are the *schedules* for vehicles b_1, b_2 and b_3, respectively;
- $F = \sum_{i=1}^{|B|}(distTravelled(b_i) * distTypeCost(b_i) + dailyTypeCost(b_i))$[1].

[1]To keep the presentation simple the real F value and the real costs are omitted.

1.3.2 Parallel Metaheuristics Hypothesis

Parallel computing is often used to decrease computation time. In search, parallel schemes may be developed from sequential algorithms in order to improve performance. The success of a parallelization schemes is closely related to the natural intrinsic parallelism of each algorithm and not all algorithms are suited for being parallelized.

In humans, knowledge gathering and learning from observations occurs sequentially. We observe the world with our senses (Sight, Hearing, Touch, etc.) and then we combine the obtained information in some manner in order to learn. This learning process is iterative. When learning a given concept, several iterations may be required. Just like humans, intelligent metaheuristics, i.e., in general metaheuristic that are not greedy and attempt to guide the search towards promising regions of the search space by gathering data from previous searches or by exploring problems characteristics, are also iterative.

The degree of metaheuristics parallelization is restrained by this fact since a certain degree of iterative processing, not parallelizable, must exist. Nevertheless, parallelism can still be explored and many parallel schemes hypothesis exist. In particular and as described in section 1.2, the following two major hypothesis exist:

Parallel strategies - Improved parallel strategies may be developed using parallel computing. Several parallelization strategies for metaheuristics have already been proposed and should be analysed to evaluate if they can possibly lead to performance gains within our algorithms.

ACO is a sophisticated metaheuristic which uses knowledge from previous search experiences in order to guide subsequent search processes. Furthermore, ACO can be naturally paralellized. Proposed ACO parallel strategies should be analysed and if possible new strategies may be derived for the purpose of developing a robust and scalable search algorithm for the PTBA problem. A common characteristic of the major parallel strategies for ACO is the fact that they are synchronous. As it will be discussed in section 3.4 synchronism causes the straggler problem [13]. Asynchronous strategies mitigate this problem but usually increase algorithms complexity and/or change the original algorithm semantics. Therefore, the asynchronism hypothesis should be analysed taken into account this tradeoff. The developed strategies must be tested in order to assess their effectiveness.

Cooperation between processes - Search agents executing in parallel can work together and exchange knowledge gathered during the search process. Cooperation schemes can help in guiding different search processes towards promising regions of the search space, and help metaheuristics escaping local optima. It is therefore important to analyse the issues in cooperative schemes: *what* information should be shared, *when* and *where* (between which processes) should the information exchange occurs.

1.4 Main Contributions

The PTBA problem variant was not yet addressed in the literature. This thesis will contribute with a set of approximative algorithms for solving the PTBA problem using Constraint-Based Local Search and Metaheuristics. Different metaheuristics, with different characteristics and degrees of sophistication will be analysed and implemented.

A detailed study of parallel mechanisms and strategies applied to metaheuristics suitable for solving the PTBA problem, as well as under what assumptions they prove to be, or not effective will be performed. More concretely, we will contribute with an analysis of:

i) How can sequential metaheuristics be extended by exploring the available resources in SM architectures (*multi-core* and *multiprocessors*) in order to achieve better performance in terms of computation time and solution quality. Regarding ACO, we will develop asynchronous parallel strategies for ACO and analyse their respective performance;

ii) How can cooperation between processes be achieved and contribute towards an effective and intelligent exploration of problem's search space;

To support this analysis, we will implement a parallel optimization library for the PTBA problem with implementations of the resulting algorithms from our analysis. This library aims at being efficient, compositional and extensive, such that new algorithms can be easily added by reusing existing ones. Additionally, even though we evaluate our algorithms in a SM architecture, our library will be designed such that developed algorithms can also target distributed architectures.

We will experimentally evaluate the resulting algorithms and strategies and assess the results obtained, namely the solution quality and speedups achieved with parallel strategies.

Our contribution on the analysis and design of parallel strategies for ACO, concretely synchronous and asynchronous strategies, was published at INFORUM 2015 [67]: *Asynchronous Parallel Ant-Colony Optimization Strategies: Application to the Multi-Depot Vehicle Scheduling Problem with Line Exchanges* .

1.5 Document Organization

The remainder of this document is organized as follows:

Chapter 2 - This chapter details the background on the topics covered by this thesis in particular Local Search Techniques and Metaheuristics, emphasizing the ACO metaheuristic, and Parallel Computing and Concurrency. The chapter finishes with a discussion of state of the art work in Parallel Metaheuristic Algorithms, also emphasising ACO, and in solutions to the PTBA problem;

Chapter 3 - This chapter is the core of this work. First, the developed model, for the PTBA problem will be presented, followed by a discussion of all the decisions made and

also describing how the model incorporates the associated constraints. Based on the developed model sequential and parallel versions of a first randomized algorithm will be presented. The chapter finishes presenting ACO sequential model and algorithm for the PTBA problem and discussing the derived model characteristics. Additionally, parallel synchronous and asynchronous strategies for ACO are proposed and analysed;

Chapter 4 - Aspects related with the implementation are covered in this chapter. The chapter starts by presenting and discussing the tools and the software model of the library as well as the associated algorithms, highlighting the properties obtained with our model. Problem's input data abstractions which allow one to obtain data from different sources (e.g. text files, remote database, etc...) are shown. An overview of the CaSPER LS solver is given as well as which features of the solver are used on our implementation. The remaining part of the chapter discusses implementation details and decisions of both sequential and parallel algorithms.

Chapter 5 - In order to assess the effectiveness of the proposed algorithms, each algorithm is experimentally evaluated. In the first part of the chapter our experimental setup is presented as well as the instance of the problem which our experiments will use. The solutions produced by our algorithms are analysed in terms of relevant characteristics and *quality of service* aspects. The influence of different parametrizations and aspects on the solutions obtained will also be evaluated. The last part of this chapter focus on presenting and discussing results of the performance and solution quality for both sequential and parallel algorithms.

Chapter 6 - This chapter finishes our work giving a summary of the proposed algorithms and strategies, and also of the main results obtained. The overall conclusions of this dissertation are drawn and perspectives of future work will be presented.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter we describe the background material necessary to the understanding of this work and we also present a brief overview of the related work of our problem, described in section 1.3.

We start by discussing in section 2.1, Combinatorial Optimization Problems, Local Search and Metaheuristics techniques used to solve this type of problems and we define the underlying base concepts. Metaheuristics and Constraint-Based Local Search approaches will be described in sections 2.1.2 and 2.2, respectively (emphasizing relevant metaheuristics for this thesis). The Ant-Colony Optimization Metaheuristic and relevant ACO versions will be described and discussed in section 2.3.

In section 2.4 we introduce Parallel Computing and Concurrency aspects. Multi-core and NUMA architectures are described in sections 2.4.1 and 2.4.2, respectively. Concurrency control mechanisms are introduced in section 2.4.4. In section 2.5, an analysis of the related work in Parallel Metaheuristics applied to combinatorial optimization problems, will be presented and discussed. An overview of different approaches and architectures for Parallel Local Search discussing each one advantages/disadvantages and complexity issues will also be presented. Section 2.6 presents the state of the art in Parallel Ant-Colony Optimization (considering techniques using CPUs).

Lastly, the state of the art in algorithms and techniques to solve the VSP problem will be discussed in section 2.7.

The purpose of this chapter is to introduce the main concepts underlying our work and to provide an initial analysis highlighting contributions to our algorithms of the current state of the art on the topics we address.

2.1 Local Search Techniques and Metaheuristics

Complete search methods guarantee *optimality*, in the sense that if several solutions exist for the problem, the method finds the best one. However, to provide such guarantee it is necessary to explore the whole search space or, by using an heuristic to guide the search, to explore a subset of the search space, as long as the heuristic guarantees optimality[65].

In general COPs are NP-*hard* and cannot be solved to optimality within polynomial bounded computation time [31].

Consequently, it turns out that for complex (in terms of instances size or problem formulation) COPs, the search space is huge and complete methods cannot solve these problems in reasonable computation time.

Approximative algorithms (e.g. Metaheuristics), despite not providing optimality, can be used to find near-optimal solutions in reasonable computation time. This algorithms can be classified as either *constructive* or *reparative* algorithms.

Constructive algorithms build solutions incrementally, by adding components to a partial solution until a complete solution is obtained. Due to their incremental nature, these type of algorithms tend to be very greedy, as in general at each iteration, the best component is selected. Consequently, this methods potentially tend to provide inferior quality solutions when compared to reparative algorithms.

Reparative or *Perturbative* algorithms start from an initial solution and attempt to progressively improve the solution quality, by applying a *move* from the old solution to the new one.

2.1.1 Combinatorial Optimization Problems and Local Search

In this section we will start by introducing a variety of concepts regarding COPs and Local Search (LS), mentioned throughout this thesis.

In general, a Combinatorial Optimization Problem is specified by a set of problem instances [1]. Each instance is a pair $\langle \mathscr{S}, F \rangle$ where $\mathscr{S}$ is the set of feasible solutions and f the cost function with the mapping $F : \mathscr{S} \mapsto \mathbb{R}$. The COP can be either a minimization or a maximization problem[1]. For a minimization problem, the objective is to find a globally optimal solution, i.e., an $i^* \in \mathscr{S}$ such that $F(i^*) \leq f(i)$ for all $i \in \mathscr{S}$.

Local Search algorithms consist in performing "local changes" to a given solution s in order to improve the solution quality, therefore moving from solution to solution in the space of candidate solutions. This process terminates according to a given criteria (e.g. time elapsed, lower/upper bound on solution quality, among others).

As will be detailed in section 2.2, with the constraint-based local search approach, problems can be modelled as Constraint Satisfaction Problems (CSPs). CSPs[65, p. 202] are

[1] Every maximization/minimization problem can be turned into a minimization/maximization problem, respectively, by reversing the sign of the objective function.

defined as a triple $\langle X, D, C \rangle$ where:

i) X = $\{X_1, \ldots, X_n\}$ is a set of variables;
ii) D = $\{D_1, \ldots, D_n\}$ is a set of domains of the corresponding variables;
iii) C = $\{C_1, \ldots, C_m\}$ is a set of constraints.

A CSP is solved by finding values $x_i \in D_i$ for each variable X_i that satisfy all the constraints in C.

COPs can be modelled as Constraint Satisfaction and Optimization Problems (CSOPs). These last problems are defined as a CSP problem with an additional objective function F defined over the variables, to be optimized. The solution space is defined by a set of constraints. CSOPs have the following canonical form:

$$\begin{aligned} \textbf{Minimize:} \quad & F(\vec{x}) \\ \textbf{Subject to:} \quad & C_1(\vec{x}), \\ & C_2(\vec{x}), \\ & \ldots \\ & C_m(\vec{x}). \end{aligned}$$

where:

- $\vec{x}$ is a vector of **n** discrete decision variables;
- F is an objective function $F : \mathcal{D}^n \mapsto \mathbb{R}$;
- $C_i(\vec{x})$ are the **m** constraints that define the solution space of the problem.

A ***solution*** to a *CSOP* problem $\mathcal{P}$ is an assignment of values to the variables in $\vec{x}$ and a ***feasible solution*** is a solution that satisfies all the constraints C_i. A ***partial solution*** is a *solution* in which some variables in $\vec{x}$ do not have any value assigned.

Let $\hat{\mathcal{L}_p}$ be the set of all the *feasible solutions*. An ***Optimal Solution*** (or a *global optimum*) is a *feasible solution* that minimizes the objective function F:

$$F^* = \arg\min \left\{ F(s) \mid s \in \hat{\mathcal{L}_p} \right\}$$

And the set of optimal solutions to $\mathcal{P}$, denoted by $\mathcal{L}_p^*$, is defined as:

$$\mathcal{L}_p^* = \left\{ s \in \hat{\mathcal{L}_p} \mid F(s) = F^* \right\}$$

The ***Search Space*** of a *CSOP* problem $\mathcal{P}$, is the set of solutions that satisfy some subset of the constraints $C_i(\vec{x})$ of the problem and is denoted by $\mathcal{L}_p^{'}$.

Therefore, the *Search Space* is the set of acceptable solutions (assignments that satisfy a subset of $C_i(\vec{x})$) and $\hat{\mathcal{L}_p} \subseteq \mathcal{L}_p^{'} \subseteq \mathcal{D}^n$.

The moves from a solution to another are defined by the *Neighbourhood*, the *Transition Graph* and *Local Optimality*.

A ***Neighbourhood*** of a *CSOP* problem $\mathcal{P}$ is a pair $\langle \mathcal{L}_p^{'}, \mathcal{N} \rangle$ where $\mathcal{L}_p^{'}$ is a search space and $\mathcal{N}$ is a mapping $\mathcal{N} : \mathcal{L}_p^{'} \mapsto 2^{\mathcal{L}_p^{'}}$ that defines for each solution s, the set of adjacent solutions $\mathcal{N}(s) \subseteq \mathcal{L}_p^{'}$. Furthermore, when the relation $s \in \mathcal{N}(j) \Leftrightarrow j \in \mathcal{N}(s)$ holds, the *Neighbourhood* is symmetric.

A ***Transition Graph*** $G(\mathcal{L}_p^{'}, \mathcal{N})$ of a problem $\mathcal{P}$, is the graph associated to the *Neighbourhood* $\langle \mathcal{L}_p^{'}, \mathcal{N} \rangle$, where the nodes are solutions in $\mathcal{L}_p^{'}$ and an arc $a \rightarrow b$ exists if $b \in \mathcal{N}(a)$.

A solution s^+ is ***Locally Optimal*** (or a *local optimum*) if none of its neighbours have a smaller cost (measured by F). Formally:

$$F(s^+) \ \leq \min_{i \in \mathcal{N}(s^+)} f(i).$$

LS algorithms must attempt to escape local optima. Therefore, they must select a solution from $\mathcal{N}(s)$ by taking this into account.

A ***Selection Rule*** $S(\mathcal{M}, s)$ is a function $S : 2^{\hat{\mathcal{L}_p}} \times \hat{\mathcal{L}_p} \mapsto \hat{\mathcal{L}_p}$ that picks an element $s^{'}$ from $\mathcal{M}$, where $\mathcal{M} = \mathcal{N}(s)$, according to some strategy and decides to accept it or to select the current solution s instead.

LS *reparative* or *perturbative* algorithms [39] start from an initial solution and iteratively attempt to improve the solution by examining the current solution neighbourhood and by applying a given move operator (e.g. changing the value of a variable or swapping the values of two variables), evolves the current solution to a new one.

Formally, a *reparative local search* algorithm is a path:

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$$

in the transition graph $G(\mathcal{L}_p^{'}, \mathcal{N})$ for $\mathcal{P}$ such that:

$$s_{i+1} = S(\mathcal{N}(s_i), s_i)$$

The sequence $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$ will eventually lead to a local optimum (or even a global optimum). Heuristics are used to guide the search towards local optima. Metaheuristics are used to escape from local optima.

The neighbourhood definition $\mathcal{N}(s) \subseteq \mathcal{L}_p^{'}$ is crucial in order to achieve high-quality solutions and depending on how each COP is modelled, should be defined in a problem specific way in order to take into account the problem search space structure.

The search space elements [65] of a COP are illustrated in figure 2.1, which represents a one-dimensional state-space for a generic maximization COP[2]. The following additional elements are defined:

plateau - a region of the search space where the evaluation function f is flat;

shoulder - A *plateau* from which progress is possible.

[2]A solution to a COP can be any state itself (or in particular, the goal state) or the path taken by the algorithm from the initial state to the goal state.

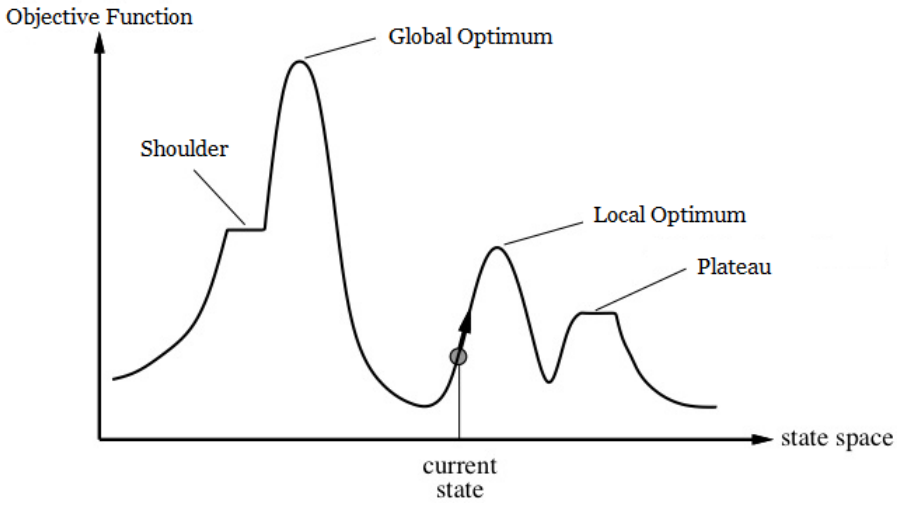


Figure 2.1: Search Space Structure. Adapted from [65, p. 121]

When in a given iteration, the move operator cannot evolve the old solution to a better one, the LS algorithm is said to be stuck in a *local optimum*. This situation is also illustrated in figure 2.1. The LS algorithm is progressing towards a local optimum and, when it reaches the local optimum, the algorithm will be stuck since none of the neighbours improve the solution quality.

In order to escape *local optima*, specially in problems with complex search spaces (e.g. large number of local optima), additional mechanisms must be used. This mechanisms, referred as Metaheuristics, will be discussed in section 2.1.2.

Constructive algorithms [39] build solutions incrementally by completing partial solutions. This type of algorithms implement two criteria: the order of the assignments, i.e. to which variable should a value be assigned next, and an heuristic to choose the value to assign. Additionally, constructive algorithms can in general be interpreted as *constructive local search methods*, however, it should be noted that solutions generated by this type of algorithm are not guaranteed to be locally optimal with respect to a simple neighbourhood $\langle \mathcal{L}'_p, \mathcal{N} \rangle$.

2.1.2 Metaheuristics

Metaheuristics [32] are defined as methods that orchestrate an interaction between local improvement procedures (e.g. LS algorithms) and higher level strategies in order to develop search strategies capable of escaping local optima and guiding the search towards promising regions of the search space. This type of heuristics (heuristics that manipulate a single solution) are categorized as *single-solution based metaheuristics*[72] (S-metaheuristics). This type of metaheuristics manipulate and transform a single solution during the search. On the other hand, in *population based metaheuristics* (P-metaheuristics) a population of solutions are manipulated.

As NP-*hard* problems, like most COP, tend to have a very large number of *local optima*

it is necessary to use mechanisms like metaheuristics in order to avoid the underlying search process being stuck.

A large number of different metaheuristics have been developed [37]. For this thesis, for reasons that will be explained shortly, the relevant metaheuristics are Multi-Starts and Restarts, and Ant-Colony Optimization, which will be described in sections 2.1.2.1 and 2.3, respectively.

2.1.2.1 Multi-Starts and Restarts Metaheuristics

The Multi-start (MS) and Restarts methods are closely related. Multi-start methods in general aim at helping search procedures exploring different regions of the search space, by applying a search procedure (or multiple search procedures) to multiple random initial solutions [54], for *reparative* LS algorithms, or by executing a randomized *constructive* procedure, building multiple solutions. Randomization is needed such that constructive search agents can construct different solutions.

Restarts are a simple and generic technique that consists in restarting a search process using some stopping criteria (e.g. stop after a given number of iterations, stop after a given number of iterations without solution improvements, among others).

A MS metaheuristic is in some sense a restart metaheuristic in which the initial solutions are not totally random.

Diversification in generating initial solutions or constructing distinct solutions, achieved with randomization, is an essential aspect in order provide Multi-Start methods the ability to overcome local optimality [54].

In [53] the authors show that Multi-start methods with memory, i.e., that use elements of previous solutions to generate each new solution, contribute to overcome local optimality since they guide the diversification process.

A Multi-start method with memory approach is used in [25] to solve the Maximum Diversity Problem. The algorithm has two phases: solution construction and solution improvement. In the first phase an initial solution is constructed using the Tabu Search Metaheuristic, proposed by Fred Glover [34, 35], and in the second phase a local search algorithm is used to post-process that solution, attempting to improve the solution quality. At each iteration the first phase algorithm penalizes elements that appeared in previous solutions and rewards elements which previously appeared in high quality solutions. Additionally, the authors compared a MS algorithm without memory with a MS algorithm with memory and concluded that the version with memory is more effective (in terms of solution quality).

Brønmo et al. [9] presented a MS local search method for a short-term ship scheduling problem. This method is also composed by an initial constructive algorithm that generates different initial solutions and a local search algorithm that attempts to improve those solutions. The authors state that the first phase is the most important part of the method and the initial solutions generated should reveal a combination of high-quality and

diversity, in order to the local search algorithm perform the search close to the optimum. They compare the MS method with a complete search method that uses mixed integer programming and, the MS method proved to be robust providing optimal or near-optimal solutions for real-life instances within a reasonable response time.

In both works the post-processing phase that aims to improve initial generated/constructed solutions consists in a simple and fast scheme. This is due to the fact that the objective here is to maximize the portion of the search space explored, in order to increase the possibility of reaching a global optimum. This maximization is accomplished by generating/constructing as much as possible (high-quality) solutions. Subsequently, a local search over those solutions may be applied if possible.

The MS method can be naturally parallelized due to the fact that the method consists in starting a search algorithm several times.

2.2 Constraint-Based Local Search

General purpose languages lack expressivity and do not provide good abstractions to the development of local search algorithms, leading to very complex implementations, lacking modularity and compositionality.

CBLS [76] consists in a paradigm that incorporates Constraint Programming in local search methods.

With this paradigm it is possible to model complex COPs using constraints, inheriting all the expressivity offered by the different types of constraints (e.g. Numerical, Combinatorial, Logical, amongst others), originating a CSOP (introduced in section 2.1.1).

In CBLS constraints are used to describe and control local search algorithms and the number of constraints violations is included in the objective function to be posteriorly minimized. Additionally, CBLS provides a clean separation of concerns between the model and the search procedure. With this separation of concerns, implementation of generic and reusable search algorithms is straightforward. This also allows the programmer to design local search algorithms by specifying the model, in a declarative way, and the search, that will use the specified model structure to effectively explore the neighbourhoods.

Michel and Hentenryck [57] proposed a CBLS architecture that follows the principles described above and which was materialized in the COMET [76] programming language. The COMET[3] language is an object-oriented programming language that offers a wide variety of modelling and search abstractions and high-level control structures.

We developed CaSPER LS [68] a CBLS library, implemented in the *C++* language, which captures the principles of compositionality, reuse and extensibility at the core of CBLS and offers high-level abstractions, providing a suitable environment to address COPs with local search, modelled as CSPs. CaSPER LS will be described in section 4.1.2.

Listing 2.1 shows an example of a model and a search component for the well-known N-queens problem using CaSPER LS. It can be seen that the features of CaSPER LS and CBLS allow one to easily specify the problem model in a declarative manner and develop

a local search component based on the model specification, which is abstracted by the constraints, since the search component only focus on minimizing the number of constraint violations.

Listing 2.1: N-Queens model and search using CaSPER LS.

```
#include <casper/ls/ls.h>
#include <iostream>

using namespace Casper::LS;
using namespace std;

int main(int argc, char* argv[]){
  Solver s;
  ConstraintSystem<int> cs(s);
  int N = 10;
  Range<int> Size(0,N-1);
  IntVarArray queens(s,RandomPermutation(Size));

  //Model definition
  IntVar i(s,Size);
  cs.post(alldifferent(
      aggregate<int>({inRange(i,Size)}, queens[i] + (i+1) )));
  cs.post(alldifferent(
      aggregate<int>({inRange(i,Size)}, queens[i] - (i+1) )));

  //Search
  int it =  0;
  IntVar p(s, Size);
  IntVar v(s, Size);
  while( cs.getViolations() > 0 && it < maxIt){
    selectMaxExpr<int>({inRange(p, Size)}, cs.violations(queens[p]));
    selectMinExpr<int>({inRange(v, Size)}, cs.swapDelta(queens[p],queens[v]));
      swap(queens[p],queens[v]);
    it++;
  }

  cout << "Solution: " << queens << endl;
  return 0;
}
```

As will be discussed in sections 3 and 4, our proposed PTBA model will be designed using CBLS ideas and will be implemented using CaSPER LS features.

2.3 Ant-Colony Optimization Metaheuristic

The Ant-Colony Optimization (ACO) metaheuristic [22, 23] is a P-metaheuristic which was originally proposed by Dorigo [22] and is inspired on the behaviour of real ants

[3]The COMET programming language was discontinued.

which use pheromones as a communication medium. In particular, ants initially leave their nest (or colony) and travel along a random path in order to find food. When a given ant finds food, it returns to his colony leaving a pheromone trail over the path taken. As time goes by, subsequent ants will start following the pheromone trails instead of wandering randomly. Additionally, over time the deposited pheromone trails evaporate and become less appealing. As ants travel along a certain path, the pheromone trails on that path will be reinforced, therefore, longer paths will have more time to evaporate the pheromone trails. Since shorter paths will be used more frequently, the pheromone density will increase over time, making the path more appealing.

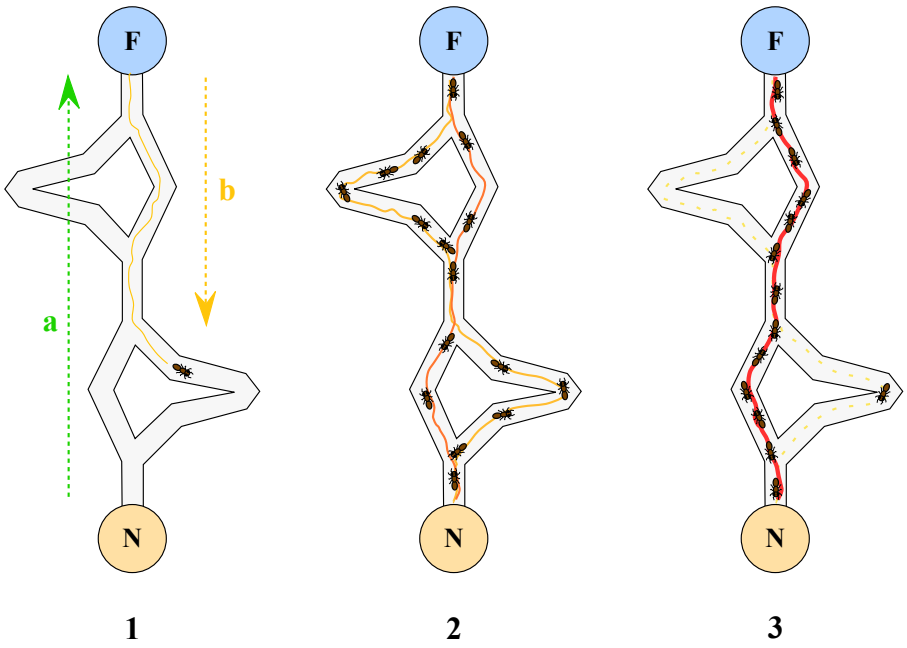


Figure 2.2: ACO ants convergence. [24]

This process is illustrated in figure 2.2. In step 1 an ant finds a food source (F) using a path a and returns by a path b depositing pheromone trail. At step 2 ants leave the nest and follow one of the 4 possible paths, however, due to pheromone evaporation and reinforcement, the shortest path becomes more appealing over time. Finally at step 3, approximately all ants converge to the same path, the shortest one.

Artificial ants are modelled as agents that communicate indirectly within the colony. Each agent performs a stochastic solution construction that probabilistically builds a solution by complementing a partial solution at each iteration. Concretely, to complement partial solutions, at each iteration heuristic information and pheromone trails are taken into account. Pheromone trails are updated at run-time reflecting the agents acquired search experience [32].

The purpose of the stochastic component is to allow the agents to build a wide variety of solutions, leading to a better (in depth) exploration of the search space, than greedy procedures. Agents search experience influences future iterations which makes

the ACO similar to a reinforcement learning procedure, i.e., a learning by interaction procedures [70].

Algorithm 1 Ant-Colony Optimization.[32]

```
1: procedure ANT-COLONY OPTIMIZATION(problem) return best solution found
2:     Initialization
3:     while termination condition not met do
4:         ConstructAntSolutions
5:         ApplyLocalSearch          % Optional
6:         UpdatePheromoneTrails
       return best solution
```

Algorithm 1 describes the main steps of ACO metaheuristic. In *Initialization* step pheromone variables are initialized. In the *ConstructAntSolutions* a set of N agents perform a constructive search procedure guided by heuristic information and by the pheromone trails, constructing the solutions. At the end of this step, a local search procedure may be applied (e.g. Steepest Ascent Hill-Climbing [65]) in order to improve the obtained solutions. Lastly, in the *UpdatePheromoneTrails* step the pheromone trails are updated such that good solutions will be more desirable. In order to avoid a fast convergence of all the agents towards sub-optimal solutions, pheromone evaporation is applied.

In this thesis we explore the ACO metaheuristic since it not only is robust, but is also a constructive procedure, which as will be discussed in section 3.1, is more suitable to solve the PTBA problem. Additionally, we intend to explore parallel synchronous and asynchronous versions of ACO, in particular, how parallel cooperative schemes can build pheromone trails in an effective manner.

In the next sections we describe three ACO algorithms: Ant System [21] (AS), the first ACO algorithm proposed, the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [69] ($\mathcal{MM}$AS) and Ant Colony System [20] (ACS), two more sophisticated and robust ACO algorithms [69] which both consist in extensions to the AS algorithm.

Several other algorithms have been proposed like Rank-Based Ant System [10], Hyper-Cube Framework for ACO [6], among others. In this thesis we focus only on AS, $\mathcal{MM}$AS and ACS due to the following reasons:

AS - as it will be discussed in section 3.4, despite of being inferior in terms of robustness, the AS algorithm allows for a better exploitation of parallel resources;

$\mathcal{MM}$AS and ACS - both algorithms are superior extensions of the AS algorithm. As will also be described in section 3.4 these algorithms introduce several additional mechanisms that affect negatively the exploitation of parallel resources. Despite of this fact, this algorithms achieve in general better results than AS in a sequential setup and should be analysed.

2.3.1 Ant System

The Ant System algorithm [21] was the first ACO algorithm proposed and was applied to the Travelling Salesman Problem (TSP).

The pheromone trails model holds information of the desirability of each solution component c_{ij}. The definition of a component depends on the problem being solved and on how one models the problem. As an example, in the TSP problem, a component c_{ij} refers to visiting a city j after visiting a city i. The AS algorithm assigns a desirability τ_{ij} (the pheromone value) to each possible component c_{ij} of the model of the problem.

We will now describe how each step of the algorithm 1 is implemented in the AS algorithm.

2.3.1.1 Initialization

In the initialization phase pheromone trails are initialized. The value τ_0 to which each pheromone trail variable should be initialized depends on the problem being solved. However the following aspects must be taken into account.

The pheromone trails should be initialized with a value slightly higher than the expected amount of pheromone deposited by the ants in one iterations [23]. This is due to the fact that if the value τ_0 is too low, the search will be quickly biased by the first solutions constructed, which will lead to a poor exploration of the search space. On the other hand, if the value τ_0 is too high, many iterations will be lost until pheromone evaporation decreases pheromone trails values such that added pheromone from constructed ants starts to bias the search.

2.3.1.2 Ant Solutions Construction

As in the generic algorithm, at each iteration N ants construct a solution using a constructive search procedure. At each construction step t each agent k uses a probability distribution to select the solution component c_{ij} to be added to a partial solution s_p.

The probability distribution used in the AS is defined as follows:

$$p_k(c_{ij}|s_p^k) = \begin{cases} \dfrac{\left[\tau_{ij}\right]^{\alpha} \cdot \left[\eta(c_{ij})\right]^{\beta}}{\sum_{l \in \mathcal{N}(s_p^k)} \left[\tau_{lj}\right]^{\alpha} \cdot \left[\eta(c_{lj})\right]^{\beta}} & , \; j \in \mathcal{N}(s_p^k) \\ 0 & , \; otherwise \end{cases} \tag{2.1}$$

where $\mathcal{N}(s_p^k) \subseteq \mathcal{L}_p^{'}$ denotes the feasible neighbourhood of the partial solution s_p of ant k, $\eta(\cdot)$ is a function that assigns to each c_{ij} in which $j \in \mathcal{N}(s_p^k)$ an heuristic value that evaluates the quality of this assignment in the context of the problem. The α and β parameters are used to control the influence of pheromone values and heuristic information, respectively, on the algorithm behaviour.

The neighbourhood $\mathcal{N}(s_p^k)$ of a partial solution also depends on the problem being solved and must be defined according to the problem model.

2.3.1.3 Pheromone Trails Evaporation and Update

In the end of an iteration pheromone trails are evaporated and updated. Pheromone evaporation is implemented as follows:

$$\tau_{ij} \leftarrow (1 - \rho) * \tau_{ij} \tag{2.2}$$

where $0 < \rho \leq 1$ is a parameter that denotes the rate of evaporation. If a solution component is not used by an agent, its corresponding pheromone value τ_{ij} decreases exponentially in the number of iterations, allowing the algorithm to avoid bad solution components, since its desirability is reduced.

After performing the pheromone evaporation step, the pheromone values are updated with learned information from each agent. The update is defined as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum \Delta\tau_{ij}^{k} \tag{2.3}$$

where $\Delta\tau_{ij}^{k}$ denotes the amount of pheromone deposited by agent k on the solution components that belong the solution s^k obtained:

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{\lambda}{F(s^k)} & , \; c_{ij} \in s^k \\ 0 & , \; otherwise \end{cases} \tag{2.4}$$

where λ is a parameter used to control the amount of pheromone deposited. With this expression, and assuming a minimization problem, the amount of pheromone deposited by each agent depends on the quality of the solutions obtained. Therefore, agents that obtain better solutions will deposit a larger amount of pheromone, favouring good solution components.

2.3.2 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

The $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [69] extends the AS algorithm by introducing additional mechanisms which by assuming that concentrating the search around the best solutions found during the search is beneficial and leads to better results, attempt to perform a stronger exploration of the search history.

Based on this assumption, the following main modifications are performed:

- stronger exploitation of the best solutions found by allowing only the iteration-best (the agent which produced the best solution of the current iteration) or the best-so-far agent are allowed to deposit pheromone;
- allowing only the iteration-best or the best-so-far agent to deposit pheromone may lead to a stagnation situation due to an excessive growth of pheromone values from components on those solutions. Besides, is likely that this solutions are suboptimal. In order to overcome this problem, lower and upper bounds, τ_{min} and τ_{max}, respectively, are imposed to the domain of pheromone values;

- pheromone trail values are initialized to the current upper bound τ_{max}. This increases diversification in the initial search iterations;
- each time the algorithm approaches stagnation or when improvements do not occur, pheromone trails are reinitialized (e.g. after a given number of iterations without improvements).

The ant solution construction phase remains exactly the same. The modifications described just above will now be presented in more detail.

2.3.2.1 New Pheromone Trails Update Mechanism

In the AS algorithm all the agents deposit pheromone. The update is performed according to equation 2.3. In $\mathcal{MM}$AS, only the iteration-best or the best-so-far agent perform the update. The update then is performed as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best} \tag{2.5}$$

where $\Delta\tau_{ij}^{best}$ is defined as in equation 2.4 but without depending on the agent k and being based on the best solution of an iteration s^{ib} or on the global best solution s^{gb}, i.e., the denominator is $F(s^{best})$ instead of $F(s^k)$.

2.3.2.2 Pheromone Trail Limits

The domain of pheromone trails values is restricted to the interval $[\tau_{min}, \tau_{max}]$, i.e., the following restriction is imposed:

$$0 \leq \tau_{min} \leq \tau_{ij} \leq \tau_{max} \leq 1 \tag{2.6}$$

It is shown by the authors that the maximum possible pheromone trail value is asymptotically bounded by $1/(\rho f(s^{opt}))$ where $f(s^{opt})$ is the objective function value of the optimal solution. Therefore, τ_{max} must be defined using an estimative of this value: $1/(\rho f(s^{best}))$. Therefore, each time the best solution is improved, τ_{max} should be updated.

The lower bound of pheromone values is defined as $\tau_{min} = \tau_{max} * a$ where a is a parameter.

2.3.2.3 Pheromone Trails Initialization and Reinitialization

Initially pheromone trails are initialized to an estimate of τ_{max}. Additionally, the rate of evaporation ρ is set to a small value in order to achieve a slow increase across all the pheromone trail values, such that diversification is achieved.

As already described, when the algorithm detects stagnation pheromone trails are reinitialized. Stagnation may be measured by some statistics or if after a given number of iterations no improvements occur.

2.3.3 Ant Colony System

The Ant Colony System [20] (ACS) also extends the AS algorithm by introducing three additional mechanisms. Like the $\mathcal{MM}$AS, ACS attempts to perform a stronger exploration of the search history.

ACS employs a different selection rule and introduces local updates. The main modifications introduced by ACS will be described in the next sections.

2.3.3.1 New Ant Solutions Construction

At each iteration step, each agent k uses the following selecting rule, called *pseudorandom proportional* rule, to select the next component c_{ij} of a partial solution s_p^k:

$$c_{ij} = \begin{cases} argmax_{l \in \mathcal{N}(s_p^k)} \left[\tau_{ij} \cdot [\eta(clj)]^\beta \right] & , \ q \leq q_0 \\ J & , \ otherwise \end{cases} \tag{2.7}$$

where q is a random variable distributed in $[0, 1]$, q_0 is a parameter with domain $[0, 1]$ and J is a random variable selected according to the probability distribution defined by equation 2.1 with $\alpha = 0$.

The parameter q_0 allows one to achieve intensification around the best-so-far solution (increasing q_0) or to achieve diversification (decreasing q_0).

2.3.3.2 Global and Local Pheromone Trail Update

ACS applies a global update mechanism (like $\mathcal{MM}$AS) and a local update mechanism in which agents are allowed to perform updates during solutions construction.

The global update mechanism is the same as in $\mathcal{MM}$AS, defined by equation 2.5. The update is performed using the best-so-far agent solution. Unlike AS and $\mathcal{MM}$AS, evaporation is only performed on the solution components of the best-so-far solution.

Each time an agent visits a solution component c_{ij} a local update on the pheromone value τ_{ij} is performed as follows:

$$\tau_{ij} \leftarrow (1 - \xi)\tau_{ij} + \xi\tau_0 \tag{2.8}$$

where ξ (domain $[0, 1]$) and τ_0 are parameters. The purpose of local updates is to decrease the pheromone trail of visited solution components such that they become less desirable to other agents.

2.4 Parallel Computing and Concurrency

Parallel computing consists in solving a computational problem using multiple compute resources. Nowadays every ordinary personal computer has a multi-core CPU and a Graphical Processing Unit(GPU). In order to take full advantage of these resources parallel computing must be used.

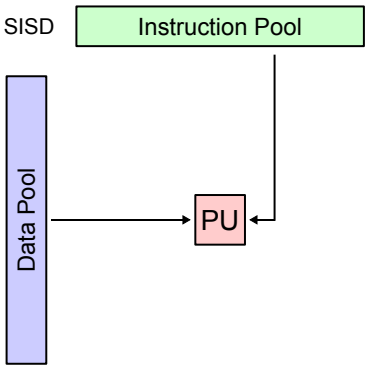


Figure 2.3: SISD [79]

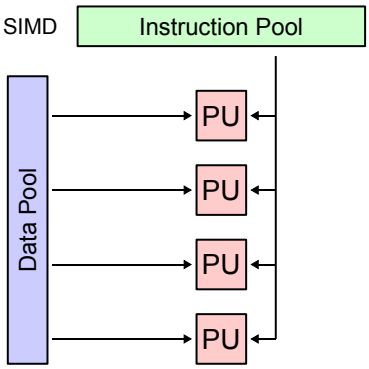


Figure 2.4: SIMD [79]

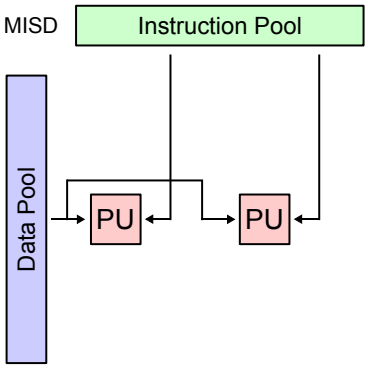


Figure 2.5: MISD [79]

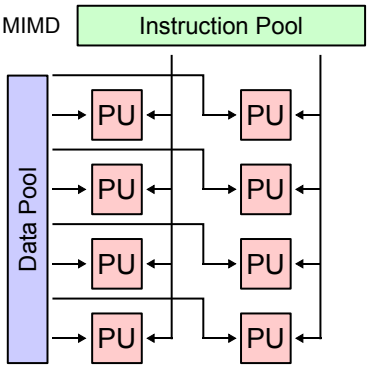


Figure 2.6: MIMD [79]

Multi-processor computer architectures are classified according to the Flynn's taxonomy [26] as follows:

- Single Instruction Stream Single Data Stream (SISD) - A serial computer (e.g. single core computer). Illustrated in figure 2.3;
- Single Instruction Stream Multiple Data Stream (SIMD) - Multiple processors execute the same instruction on different data streams (e.g. GPUs). Illustrated in figure 2.4;
- Multiple Instruction Stream Single Data Stream (MISD) - Multiple instruction streams are executed on a single data stream. Illustrated in figure 2.5;
- Multiple Instruction Stream Multiple Data Stream (MIMD) - Multiple processors execute different instructions on different data streams (e.g. multi-core computers, multiprocessors, computer clusters, among others). Illustrated in figure 2.6.

This thesis focus on MIMD, which is the multi-core and multiprocessor computers architecture.

The effectiveness of a parallel implementation in terms of computation time is measured by the speedup and efficiency achieved [44]. The speedup $S(N)$ of a parallel implementation in relation to a sequential implementation is defined as:

$$S(N) = \frac{T(1)}{T(N)} \tag{2.9}$$

where $T(N)$ is the time it takes to execute the program using N processors. The speedup metric denotes the gain of parallelizing a given program.

Efficiency $E(N)$ of a parallel program denotes the usage ratio of all the available processors and is defined as follows:

$$E(N) = \frac{S(N)}{N} \tag{2.10}$$

In theory, the speedup cannot exceed N [44]. However in practice, the speedup can be greater than N. This phenomenon is called *superlinear speedup*.
One situation where superlinear speedup can be achieved is when the amount of data required to solve a problem does not fit into the processor cache, degrading the performance of the parallel implementation. However, when using N processors and assuming each processor has an assigned cache, the data is partitioned across the N processors fitting in the processors caches.

The Amdahl's law [4] states that the speedup achieved by paralellizing a single program with N processors where a fraction f of the code can be parallelized is:

$$S(N) = \frac{1}{(1-f) + \frac{f}{N}} \tag{2.11}$$

As discussed by Hill and Marty in [38] the Amdahl's law assumes that the fraction f of a program's execution time is infinitely parallelizable without overhead, which as they stated is a very simplistic assumption.

Parallel computing essentially targets *shared-memory* (SM) and *distributed* architectures. In SM architectures processors share the address space and concurrent accesses to this memory must be synchronized. In distributed architectures, each processor has its own memory and all the processors are interconnected through a network. Due to the reasons presented in section 2.5, in this thesis we target SM architectures. Within SM architectures, the following memory architectures exist [28]:

- Uniform Memory Access (UMA) - all processors access physical memory equally, sharing the same data bus;
- Non-Uniform Memory Access (NUMA) - despite the fact that all processors can access the whole physical memory using real addresses, each processor have a part of physical memory attached. Accessing attached memory is faster than accessing foreign memory zones;

- Cache-only memory architecture (COMA) - similar to NUMA. However, the local memory (memory attached to a processor) is used as a cache. Therefore, when a processor accesses foreign (non local) memory, data is migrated to its local memory.

Our experiments target the NUMA architecture.

2.4.1 Multi-Core Architectures

One *core* refers to a piece of hardware that executes a stream of machine instructions. In Multi-core architectures a computer CPU has more than one core where each core can be *hyperthreaded*, i.e., is able to execute more than one stream of machine instructions in parallel [41].

In single-core architectures (only one core) the core reads machine instructions from memory, decodes and executes them. Only one stream of instructions can be processed at once. Consequently, to execute more than one thread the operating system needs to perform a context switch and exchange the thread that is being executed. The program instructions and data are stored in main memory, which is much more slower than the processors. To reduce the latency of memory accesses a level 2 (L2) cache is placed between the core and main memory. The L2 cache is faster than main memory but is also much smaller (only has capacity for a few megabytes). Since the L2 cache is still not as fast as the core circuitry an additional level 1 (L1) is placed between the core and the L2 cache. Typically the L1 cache is almost as fast as the processors core but is even smaller than the L2 cache.

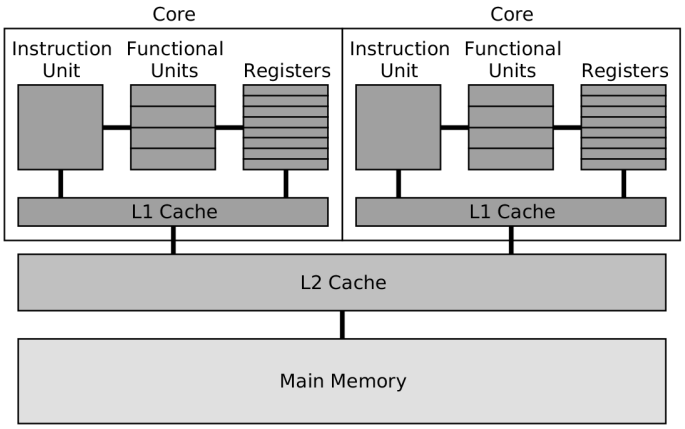


Figure 2.7: Multi-Core computer architecture. [41]

In multi-core architectures, as illustrated in figure 2.7, processors have more than one core (typically 2 or more). Each core has an L1 cache and all cores in a CPU share the L2 cache. Having processors with more than one core allows the operating system to execute applications truly in parallel.

Hyperthreading can be achieved by replicating instruction units, allowing the execution of multiple threads without context switches. However, the multiple threads will only

run at full speed if each one uses different registers or any other functional units. When multiple threads use the same registers or functional units the access to those devices is interleaved.

2.4.2 NUMA Architectures

UMA architectures have limited scalability in the number of processors due to contention in accessing physical memory (the data bus is shared among all the processors).

In the Non-Uniform Memory Architecture (NUMA), processes access physical memory in a non uniform way. A chunk of memory is assigned to each processor (local memory), referred as a *node*. Nodes are connected through a high bandwidth low latency interconnection network. Figure 2.8 illustrates this architecture.

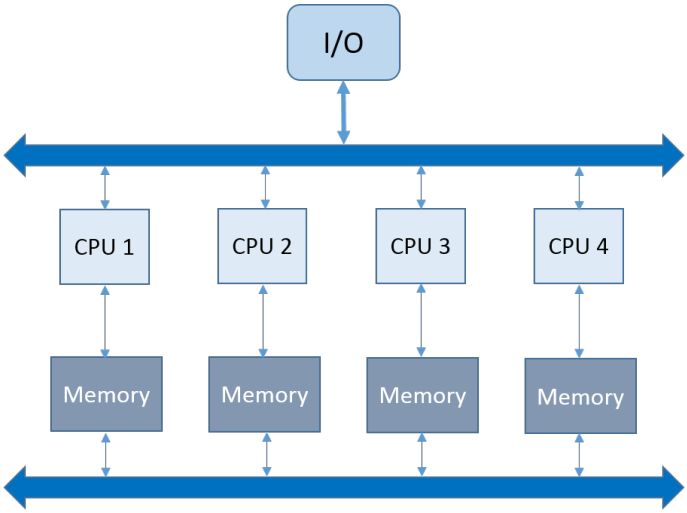


Figure 2.8: NUMA Architecture.

Accesses to local memory are faster than accesses to non-local memory. This scheme is achieved using distributed shared-memory implemented as a distributed virtual memory scheme. Since physical memory is distributed across processors, accesses to local memory do not interfere with each other.

All modern computer architectures have caches, causing the cache coherence problem. This occurs when two or more processors have data from the same real memory address and one of the processors modifies its local copy of the data. The Cache-Coherent NUMA [45] architecture implements a mechanism that ensures cache coherence, using the global shared bus.

2.4.2.1 NUMA-Aware Applications

Applications must be aware of the NUMA architecture since it has a great influence in memory access performance.

The following aspects must be taken into account:

- Processes/Threads must be placed on processors close to the memory location of the data that will be accessed;
- Memory allocations within a process must occur on the corresponding NUMA node;
- the OS scheduler should be NUMA-aware;
- child processes should be dispatched on the same processor as the parent process.

The Linux operating system (OS) offers NUMA support since kernel 2.5, implementing a NUMA-aware scheduler.

In order to develop NUMA-Aware applications, the Linux OS offers a NUMA Application Programming Interface [50] (API) consisting of a set of system calls, allowing programmers to control threads binding and memory allocation policies.

Additionally, the GCC[4] compiler implements the NUMA policy library [49] (*libnuma*) which abstracts OS NUMA API's, providing the same level of control.

2.4.3 Communication within processes

In SM architectures processes/threads communicate implicitly through shared memory. A *thread* is referred as a lightweight process which exist within a process. The memory address space of a thread is the same as the process from which the thread was created, therefore, when a process spawns several threads, these threads share the address space.

Communication within threads can be achieved through writes and reads to specific addresses (meaningful for the application) on the shared address space with minimal overhead. The only overhead imposed is the duration of a read/write operation on secondary memory (RAM). However, when multiple processes communicate with each other via the same memory regions, accesses must be synchronized using concurrency control mechanisms. These mechanisms will be presented in the next section.

2.4.4 Concurrency Control Mechanisms

Several concurrency control mechanisms have been proposed [66]. From the mechanisms available we use locks to enforce mutual exclusion. Mutual exclusion is a concurrency control requirement which consists of assuring that only one process executes a *critical section*, a portion of the code that cannot be executed concurrently, at each moment.

In our algorithms we use three different types of locks: *mutex*, *Spin locks* and *Read-Writer* locks.

Mutex locks (mutual exclusion locks) can be implemented with busy-waiting mechanisms [66], and can be used to achieve mutual exclusion.

A *Spin lock* consists of a lock in which threads wait in a loop ("spin") until the lock is free and can be acquired. Since threads are active (in a loop) they are not preempted, i.e, their execution is not stopped therefore they remain on the processor. This technique avoids

[4]https://gcc.gnu.org/

the overhead of performing a context-switch (the process of storing or restoring a process/thread execution state). However, this type of locks only improves the performance if the threads are blocked for short periods.

Reader-Writer [15, 66] locks relax the constraints of mutual exclusion by allowing concurrent accesses for read-only operations. Write operations require exclusive access to the critical section. This allows one to maximize the throughput in terms of read-only operations. An appropriate situation in which this type of locks are suitable is when the number of writes is less than the number of reads.

However, despite of this advantages, this type of lock can lead to write operations starvation if contention is high. If at every moment of the execution at least one single read-only operation is executing, new arriving threads performing read-only operations will be allowed to enter the critical section and write operations will starve. To overcome this problem, *write-preferred* operations were introduced [15, 66]. *Write-preferred* locks do not grant access to new read-only operations if atleast one writer operation is queued and waiting for the lock. This scheme reduces concurrency in the presence of write operations, however, as already described, if write operations are sporadic, this type of lock is still advantageous.

An additional mechanism can be used to complement locks, *condition-variables*. A condition-variable is a mechanisms that allow threads to suspend execution while waiting for a given condition to be verified (e.g. a given data structure is not empty). The execution is then resumed by signaling on the condition, by other threads. As will be discussed in section 4.4.2, conditional variables are used to implement thread-safe data structures like pools.

2.5 Parallel Metaheuristics Algorithms

Although metaheuristics improve the effectiveness of search algorithms for solving COPs by guiding the search process towards promising regions of the search space and also by reducing the overall time of the search process, for many COPs, the search process remains time-consuming. This is mainly due to the following characteristics of COPs:

- Complex objective functions whose computation is time-consuming;
- Complex models involving a large number of constraints;
- Large search spaces.

In general COPs, especially real life COP instances, have a combination of the characteristics highlighted above.

Parallel computing can be used to address this problems and several parallel metaheuristics and algorithms have been proposed to solve COPs, including CBLS strategies [2, 3, 19, 63]. More concretely, parallel computing can be advantageous due to the following aspects [52]:

Computation time - Parallelization can reduce the search time;

Solution quality - Parallel cooperation mechanisms between processes can lead to a more effective search since information exchanges influence the behaviour of each search algorithm process;

Robustness - A parallel scheme may be more robust in the sense that it can handle a wider spectrum of COP instances in an effective manner;

Scaling - Parallelization should help solving large-scale problems since the aggregate computational power is superior.

Verhoeven and Aarts in [78] proposed two distinct approaches for LS parallel algorithms: *single-walk* (SW) and *multiple-walk* (MW).

In SW algorithms only a single walk in the search space is carried out. This approach involves partitioning a solution or elements of a solution over different processors. Within SW algorithms, the authors distinguish between *single-step* and *multi-step* parallelism. Either algorithms evaluate the neighbourhood in parallel, however single-step algorithms perform a single LS step whereas multi-step algorithms perform multiple LS steps. Figure 2.9 illustrates the difference between the two versions.

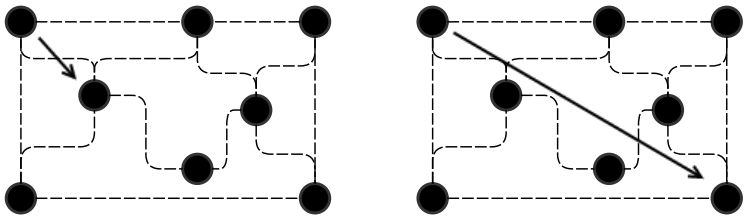


Figure 2.9: Single-step parallelism (left) and multi-step parallelism (right). Dotted edges denote connected neighbours, vertices denote solutions and the solid arc denotes a LS step. In single-step parallelism only one move is performed whereas in multi-step parallelism multiple moves are performed (distant neighbours are reached).

In MW algorithms, several walks through the search space are performed simultaneously. Within MW algorithms the authors distinguish between algorithms that perform multiple independent walks or interacting walks (i.e. processes exchange information during the search).

Finally the authors classify both SW and MW as *synchronous* or *asynchronous*. In synchronous algorithms one or more algorithm steps are performed simultaneously by all processors whereas in asynchronous algorithms no synchronization occurs at each algorithm step.

Parallel metaheuristics can be implemented in a SM architecture or in a distributed architecture. Metaheuristics implemented in distributed architectures, as discussed in [51], are limited by communication latencies hence, the multiple interacting walks approach is not suitable.

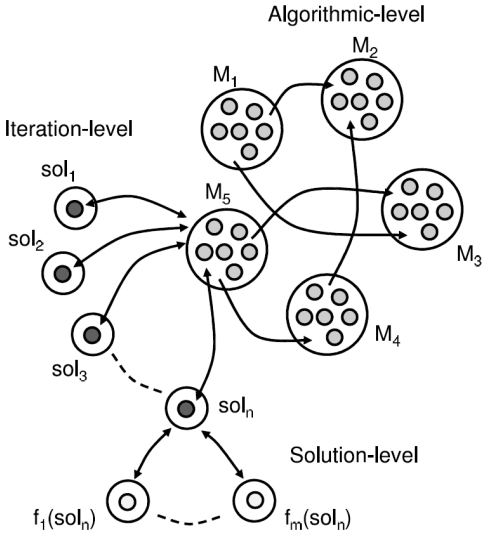


Figure 2.10: Parallel metaheuristic models. Adapted from [51]

Regarding parallel models, the following three major parallel models in the design of metaheuristics [72], illustrated in figure 2.10, have been proposed:

Solution-level Parallel Model - Parallel evaluation of a single solution (SW parallelism). Suitable when the evaluative function is CPU time-consuming and can be decomposed in several partial functions;

Iteration-level Parallel Model - Each iteration of the metaheuristic is parallelized (SW parallelism). This model does not change the original metaheuristic. This model is suitable when the problem being solved has large neighbourhoods ;

Algorithmic-level Parallel Model - Multiple metaheuristic executions (the same metaheuristic or different metaheuristics in each execution) with the same or different parameters, starting from the same or different initial solutions (MW paralellism).

Multi-Starts (discussed in section 2.1.2.1) can be trivially mapped as a multiple independent walk parallel algorithm by executing a search algorithm in several processors.

As it will be discussed in section 2.6, the common ACO metaheuristic parallelization strategy consists of a dependent multiple-walk approach and explores algorithmic-level parallelism in which agents cooperate. In section 3.4 we propose additional strategies wherein one of the characteristics of this strategies consists of introducing a different cooperation scheme between search agents.

2.5.1 Cooperative Parallel Search

Cooperation between parallel processes makes it possible to achieve intelligent behaviour on search algorithms, in the sense that several processes cooperate in order to solve a given problem.
In general, each step of a given metaheuristic depends on the previous steps, making it

difficult to achieve a large degree of parallelism. One simple approach consists in designing a multiple independent walk algorithm, where in each processor a given metaheuristic (or the same) is executed, in order to achieve a large degree of parallelism. This approach can be extended to a cooperative search and possibly lead to a more effective search algorithm.

Toulouse et al. [74] described the *systemic behaviour* (i.e. interactions triggered by the occurrence of other interactions) of cooperation between processes and analysed how cooperation impacts the search. The authors show that correlated *effective interactions* (interactions that change the search path) lead to different search patterns, comparing with the pattern which would have been observed by using the original metaheuristic. This means that solutions that would not be explored, will possibly be explored, and different solutions can be achieved. However, this does not imply that the search will converge to promising regions. Based on this observation we can conclude that cooperation can be used to explore *different* areas of the search space. Therefore, by extending existing metaheuristics with a cooperative scheme, or by using metaheuristics that already implement one, and adding a mechanism to control when should shared knowledge be used, it should give to the used metaheuristic the ability to overcome local optima.

In order to design effective cooperative search algorithms certain issues must be addressed. Toulouse et al. [73] identified and analysed the main issues of cooperative algorithms.

One main issue is the *convergence to similar search paths*. The dissemination of information must be controlled in order to avoid the situation where all parallel search processes converge to the same region of the search space. To avoid convergence to similar search space regions, access by a given parallel process to shared knowledge, i.e. specific search information which all processes gather, must be constrained.

It is possible to achieve collective intensification (increase exploitation of a search space area), by relaxing access to shared knowledge, or diversification by tightening the access.

The authors established the *feedback loop* phenomenon which describes an interaction of shared (global) and non-shared information. This is based on the fact that shared information at iteration it_j depends on the information accumulated by all processes in their non-shared information, from iteration it_{j-1}. On the other hand, the non-shared information of each process between iterations it_j and it_{j+1} is partially determined by the shared information, therefore, both sources of information (shared and non-shared) influence one another.

Furthermore, the authors highlighted the fact that the feedback loop has a great impact in the search algorithm and analysed this impact in synchronous and asynchronous cooperative algorithms. They observed that asynchronous information exchanges yields better results than synchronous due to the fact that synchronous algorithms develop self-organized search strategies that concentrate on the synchronization events. On the other hand, in asynchronous algorithms the events that trigger the interactions amongst processes are the improvements on the objective function value or end of a solution construction phase. Based on this observation we can conclude that asynchronous algorithms

should perform better than synchronous due to its superior capacity in adapting the exploration of the search space according to each problem structure and to the fact that communication is performed based on events of the space structure rather than in the synchronization logic.

Caniou and Codognet [12] developed a multi-start multiple-independent walk CBLS algorithm where different processes cooperate in order to achieve better solutions. The algorithm is asynchronous and terminates as soon as one process finds a solution. A master-worker architecture is used where the master coordinates the gathered knowledge. In order to compare the effectiveness of the cooperation, the developed algorithm is compared with an equivalent algorithm but without cooperation.

In a first attempt the authors developed one version of the algorithm where the shared information consists in the current cost of each process. At each iteration c, each processor checks if the other processes have a lower cost and chooses to restart according to a given probability p. The experimental results shown that this algorithm could not outperform the version without cooperation, which can be explained by the fact that the cost of a solution is not a reliable information since it is just an heuristic value.

In a second attempt the authors developed another version of the algorithm in which the shared information also contains the number of iterations at which the cost was computed. With this information, a process may decide to restart if the shared cost value is better for a better number of iterations, also depending on the probability p. The experimental results also shown that this algorithm does not outperform the version without cooperation.

From this observations and based on the fact that the algorithm that does not make restarts is always superior, we can conclude that simply restarting a process is not effective. One observation that also supports this conclusion is the fact that processes decide or not to restart at each c iterations but, different processes spend different amount of time in each c iterations and follow different search paths. Therefore, if a restart is made, the new initial solution must be built based on gathered information such that the process will search in a region closer to the region where the best information was obtained.

Additionally, we conclude that sharing the cost of obtained solutions, the iteration in which the correspondent solution was obtained, or both, does not give sufficient information such that search agents can direct the search towards promising region of the search space. It turns out that this type of knowledge is related to the metaheuristic used and not to the problem characteristics. Therefore, shared knowledge must be much richer and should give more insight about the problem search space.

As will be detailed in section 2.6, we use the ACO metaheuristic to implement cooperative parallel search, in which the shared knowledge is directly related to the problem being solved. The mechanism used to achieve intensification and diversification, by restricting access to shared knowledge, will also be explained.

2.6 Parallel Ant-Colony Optimization

Pedemonte et al.[60] recently proposed a taxonomy to classify parallel ACO algorithms which is illustrated in table 2.1.

The *master-slave* model is based on a master process that manages global information and controls a group of slave processes that perform tasks related to the ACO search space exploration. Regarding the *granularity* (i.e. the amount of work delegated to each slave) the model includes three distinguished categories: Coarse-grain, Medium grain and Fine-grain.

In Coarse-grain, tasks delegated to slaves may correspond to one or more ants where each builds full solutions and report the results to the master. In Medium-grain slave processes solve subproblems independently and report partial solutions to the master, which is in charge of building the complete solution. Finally, in Fine-grain, slave processes execute small tasks and communicate frequently with the master.

		#Colonies	
		one	*many*
Cooperation	*yes*	cellular	multicolony
	no	master-slave (coarse/medium/fine)	parallel independent runs

Table 2.1: Parallel ACO taxonomy categories.

The *cellular* model is based on a single colony that is structured in small neighbours, each with one pheromone matrix and uses techniques from cellular evolutionary algorithms. This model will not be explored in this thesis. *Parallel independent runs* models consist in simply executing one independent sequential ACO algorithm in each process, without any communication. Finally, in the *multicolony* model several colonies, each running in one processor, explore the search space using different pheromone matrices. This model takes into consideration cooperation within processes that exchange informations periodically.

The authors in [60], based on the overview of related works, concluded that the coarse-grain master-slave and multicolony models are the most promising models for achieving high computational efficiency. The developed algorithms in this thesis use the coarse-grain master-slave model. Either way, the library developed will allow one to easily extend our implementations in order to use the multicolony model using the algorithms already implemented.

In [71] a parallel ACO algorithm which uses a LS phase using Tabu Search is proposed in order to solve the Quadratic Assignment Problem (QAP). To represent the solution, a permutation of integers is used. An interesting initialization scheme is described which consists of assigning an initial random solution to each ant, a LS procedure is applied over all solutions and the best solution is selected. Posteriorly, the pheromones are initialized based on the objective function value of the best solution.

This algorithm has a master-slave coarse grained model with communication in which

the master implements a central memory that stores the global knowledge acquired during the search and each slave implement the search (one ant). At each iteration the master broadcasts the pheromone matrix to all slaves. A diversification phase is also applied by the master when after $n/2$ iterations (n is the number of locations in QAP) the best solution is not improved. In the next iteration slaves start from the solutions generated in this phase. The authors show that the cooperation scheme implemented (centralized pheromone matrix) and parallelization complementarily lead to an improvement of the effectiveness of the algorithm in achieving high quality solutions.

In [17] the authors propose an SM ACO algorithm that uses the *coarse grain master-slave* model and compare SM architectures with distributed architectures. The SM version of the proposed algorithm outperformed the distributed version. The main reason is the communication overhead imposed by the network.

Delisle et al. [16] presented an SM parallel implementation using OpenMP of ACO in order to solve an industrial scheduling problem. As the authors stated, the ACO algorithm follows exactly a *fork-join* scheme. Additionally, despite the fact that with SM no explicit communications are needed, therefore omitting communications overheads, a synchronization barrier that occurs when each ant finishes the search and the pheromone matrix needs to be updated, causes the straggler problem, which is defined above, and prevents the algorithm from achieving better performance.

Cipar et al. [13] defined the *straggler problem* as the situation in which a small number of threads (the stragglers) take longer than the others to execute a given iteration. Since all the threads will eventually be synchronized, all threads will proceed at the speed of the slowest one. As stated in [13] one of the solutions for the straggler problem is to make the algorithm asynchronous. Despite of potentially making the algorithm more complex in order to maintain its semantics and properties, it eliminates completely the straggler problem, and allows the algorithms to use all the available parallelism.

In the context of ACO and in a distributed architecture, Kotsis et al.[43] proposed a *partially asynchronous* parallelization scheme on the AS algorithm to reduce the straggler problem caused by the ACO sychronization barrier. The scheme consists in creating temporary sub-colonies in computer nodes which live for a certain number of iterations. Despite reducing the amount of communication performed the straggler problem still occurs within each sub-colony. After *it* iterations the results are reported to the master node. With this scheme, not only the amount of communication performed is reduced but also the total execution time of a thread in a sub-colony is bounded by the slower thread of the sub-colony instead of bounded by the slower thread of the whole colony. Therefore, if one thread in a colony takes considerable more time than the others from his sub-colony, there will be no interference with other threads belonging to other sub-colonies. However, the straggler problem still exists within the belonging sub-colony.

In [75] the authors propose two ACO asynchronous models based on the *cunning* Ant System (cAS) algorithm. The difference in the two models consists in modelling the pheromone matrix as a critical section (AP-cAS) or not (RAP-cAS). The RAP-cAS model

achieved the most promising results (in terms of speedup) comparing with (AP-cAS) and a synchronous model. In order to use the cAS algorithm, the problem model must target a *reparative* search scheme, which as will be discussed in section 2.7, is not suited for our problem. Additionally, asynchronism breaks the semantics of the original ACO algorithm and the learning capabilities of the proposed algorithms was not verified.

In this work we propose three asynchronous ACO algorithms based on the AS algorithm, that break the semantics of the original metaheuristic in order to achieve better performance, and differ on the degree of concurrency allowed while manipulating the learned information. Different cooperation schemes are achieved with each model. The proposed algorithms will be presented in section 3.4 and analysed in section 5.3.

2.7 Public Transport Bus Assignment problem

According to the terminology in the literature the most similar problem to the PTBA problem is the multi-depot vehicle assignment problem with heterogeneous fleet and line exchanges. However, as our variant models a real life scenario, as stated in section 1.3.1, additional constraints are considered related to vehicle maintenances, waiting times on terminals to load passengers or exchange drivers, among others.

To the best of our knowledge, our variant of the the MD-VSP was not yet addressed.

Real life instances of the PTBA problem, either in urban and inter-urban contexts, tend to be large. Complete search is not able to solve these instances in reasonable time due to the combinatorial involved. In the literature several heuristic methods based on variable fixing, i.e., reducing the problem size by fixing some variables were proposed [27, 33]. These methods still have a complete search phase that is applied over a reduced problem (in the number of variables) which restricts the ability for this algorithms to scale, in terms of solving larger instances.

A different approach based on Iterated LS is used by Laurent and Hao[46]. The MD-VSP problem is considered with an homogeneous fleet and no line exchanges. The algorithm starts by generating an initial solution and then iterates over that solution applying a perturbation mechanism followed by a LS phase. In each iteration, after each LS phase an acceptance criterion is applied that verifies if the obtained solution is admissible (i.e. does not violate any constraint).

Two simple neighbourhoods are considered:

- Shift Neighbourhood - transfer the trip of a vehicle to another vehicle;
- Swap Neighbourhood - Swapping the two vehicles of two trips (vehicles must be different.

These neighbourhoods are very naive since they lack any consideration for constraints like maximum travelling time, void trips, among others. Due to this fact the authors consider an additional neighbourhood which is named *block-moves* neighbourhood. This neighbourhood is based on ejection chains (originally introduced in [36]) which are based

in selecting a set of elements to undergo a change of state, leading to an identification of a collection of other sets with the property that atleast one of the elements must be *ejected* from the current state. The *block-move* applied by the authors consists in shifting a given number of consecutive trips of a given vehicle v to another vehicle v'.

Naturally this move often leads to constraint violations and consequently, a repair mechanism is applied. This last neighbourhood is more sophisticated and captures the notion of a sequence of services, however, the notion of line exchanges is not captured. Therefore, this move is not suitable for our case.

Another LS approach is used in [77], where the Multi-Depot VSP with line exchanges is considered. The problem is formulated using a CBLS approach. Hard constraints, that enforce the consistency of each solution, are always satisfied. On the other hand, soft constraints (used for line changes, vehicle transfer operations, amongst others) are incorporated in the objective function. The neighbourhood used is the same as the previous work, the shift neighbourhood. The authors show that by considering line exchanges it is possible to achieve solutions with smaller operational costs (around 9.91% reduction).

In both works analysed above, we verify a great effort in assuring that hard constraints are not violated. A study in how much LS iterations are performed and whose resultant solutions are discarded is not present in either works, however, we can infer that a significant number of iterations are discarded due to the used neighbourhoods. This justifies why solution repair schemes are used. Based on this observation we believe that constructive approaches should be more adequate and yield better results, therefore, our developed algorithms will follow the constructive approach.

CHAPTER 3

PTBA PROBLEM ALGORITHMS

In this chapter we start by discussing our problem model based on the formulation presented on section 1.3.1. First, in section 3.1.1 the base sequential constructive algorithm will be presented, followed by a discussion of how hard constraints are enforced, such that an admissible solution is produced. Our sequential algorithm is designed so that different heuristics may be used to decide which vehicles should be assigned to a given departure. In order to achieve different behaviours, in particular randomization, we use different heuristics. Our developed heuristics will be discussed in section 3.1.2. Based on this sequential algorithm, we developed a parallel probabilistic Multi-Start Metaheuristic. This parallel metaheuristic and its associated details will be presented in section 3.2.

Then, in section 3.3 we discuss how the PTBA problem can be modelled with the ACO metaheuristic and we present the developed algorithms based on the AS and $\mathcal{MM}$AS algorithms. Lastly, section 3.4 presents and discusses parallel synchronous and asynchronous strategies and the underlying cooperation scheme for the developed ACO algorithms.

3.1 Problem Model and Base Algorithm

In the PTBA problem the objective is to produce a schedule for a set of vehicles such that each time-tabled departure $T_i \in T$ is covered, whilst minimizing the operational costs and satisfying a set of constraints.

Our variant of the problem takes into account a significant number of constraints. As already discussed in section 2.7, even on more general variants of the problem, it is difficult to define a move operator that perturbs a given solution into a new solution, while satisfying all the hard constraints. This is even worse with our variant of the problem since we take into account additional constraints.

One approach would be to apply a simple and fast operator by sacrificing solution feasibility. This would allow one to design reparative LS algorithms, however, as also

was discussed, this would require the definition of repair schemes such that infeasible solutions become feasible. Depending on the type of move operator used, repair schemes may be too complex, what would lead to a strategy that spends a significant part of the time (based on the fact that the procedure for generating new solutions is faster than the repair scheme) repairing solutions instead of exploring the search space.

The constraint which enforces that vehicle schedules in which *InService* tasks are *compatible* trips, is the one which has greater chances of becoming violated, when a given LS move operator is applied. This is due to the fact that this constraint implicitly implies a sequence of compatible trips within a schedule for each vehicle. When performing modifications on a vehicle sequence of compatible trips, one must ensure that in the end, all trips of all vehicles are still compatible within the respective service.

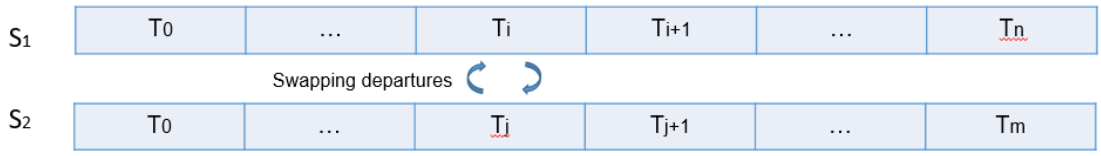


Figure 3.1: Trips dependency example.

Figure 3.1 exemplifies this problem. Two schedules S_1 and S_2 are considered, with n and m associated tasks, respectively. A move operator, which consists of swapping vehicles assigned to trips from the two schedules, is applied. Despite of the operator simplicity, there are no guarantees that task T_j and T_{i+1} are compatible (the same happens to T_i and T_{j+1}).

To overcome this problem, we follow a constructive approach in which a partial solution is extended at each iteration, until a final solution is obtained. The constructive approach allow us to incrementally build solutions without violating hard constraints by allowing only admissible components to be added to the current partial solution.

We apply this approach by assigning only vehicles to departures in which the assignment does not lead to constraint violations. To model this behaviour, a set of admissible vehicles is built for each departure and then one vehicle is chosen according to some heuristic. This approach assumes that departures are processed in a given order. Our model is therefore *departure oriented*, in the sense that local decisions are made when processing a given departure.

In the following section we introduce our algorithm which materializes the described approach.

3.1.1 Sequential Algorithm

The approach described above is materialized as follows.

Let $G = (S, A)$ be a directed graph where each vertex $s \in S$ corresponds to a state of the problem and each arc $a \in A$ corresponds to an admissible assignment of a vehicle to a departure in a given state. A state is composed by:

i) Next departure to assign a vehicle (T_j);
ii) For each depot D_k, the set VD_k of vehicles in the depot k and their arrival times;
iii) For each terminal P_k, the set VP_k of vehicles in the terminal k and their arrival time;
iv) Distance travelled by each vehicle;
v) Total waiting time of each vehicle;
vi) Total working time of each vehicle;
vii) Time (working time) of last maintenance task of each vehicle;
viii) The total number of service interruptions of each vehicle;
ix) Attended departures.

Our algorithm constructs a solution by performing a walk in G, using a given strategy to select which vehicle to assign to a departure in each state. Figure 3.2 illustrates the problem graph. A solution is found when a vehicle is assigned to all the m departures.

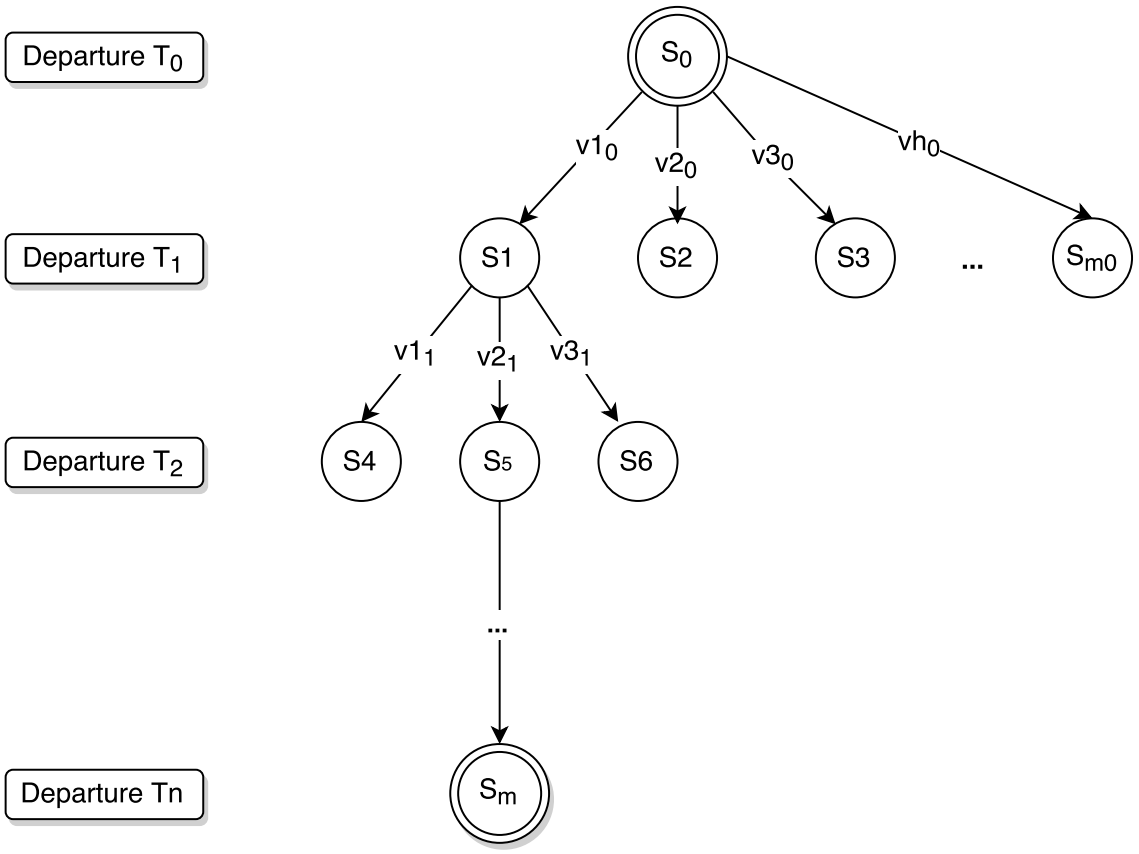


Figure 3.2: Example of a possible problem graph.

Let $VD = \bigcup VD_k$ and $VP = \bigcup VP_k$. At each state S_i, when processing departure T_j, a set of admissible vehicles $AdmV_i$ is computed based on the sets VD_k for each depot D_k and on the sets VP_k for each terminal P_k. The set $AdmV_i$, which corresponds to the

neighbourhood $\mathcal{N}(s_p)$ of a partial solution s_p, is equal to $AdmVD_i \cup AdmVP_i$, where $AdmVD_i \subseteq VD$ and $AdmVP_i \subseteq VP$ are the sets of admissible vehicles from depots and terminals, respectively. Each element vh_i of $AdmV_i$ denotes a vehicle that when assigned to departure T_j satisfies all the hard constraints described in section 1.3.1. The successors of each vertex are the assignments of elements of $AdmV_i$ to the current departure. Based on a given strategy, as it will be discussed in section 3.1.2, one element of this set is chosen and assigned to T_i.

The algorithm pseudo-code is outlined in Algorithm 2. Initially all the departures $T_j \in T$ are sorted in increasing order by their departure minute s_j. Then, for each T_j, the set $AdmV_i$ of *admissible* vehicles, departing either from a depot ($AdmVD_i$ set) or from a terminal ($AdmVP_i$ set), is computed. Subsequently, a vehicle from $AdmV_i$ is chosen according to a given heuristic function hF and assigned to departure T_j. The different heuristic functions will be described in section 3.1.2. When assigning tasks, we assume that a regular service day starts at 5:00 AM and our unit of time is minutes. To simplify calculations, we also assume that 5:00 AM corresponds to minute 0.

It is also worth noting that initially, all vehicles are on their respective depot. When vehicles terminate their scheduling, they are sent to the depot.

Algorithm 2 PTBA Algorithm

```
1:  procedure PTBA_ALGORITHM return set of schedules for each bus
2:      sorted_deps ← set of departures T sorted by departure time
3:      for all departure t in sorted_deps do
4:          AdmVD_i ← select admissible buses from each depot
5:          AdmVP_i ← select admissible buses from each terminal
6:          AdmV_i ← AdmVD_i ∪ AdmVP_i
7:
8:          % Select a bus using a given strategy
9:          choosen_bus ← SelectBus(hF, AdmV_i)
10:         t.assign(choosen_bus)
```

3.1.1.1 Admissible Vehicles set construction

Let $duration(l_1, l_2)$ denote the duration of a trip from location l_1 to location l_2.

When computing the set $AdmVD_i$ the following decisions are made:

- vehicles that are not allowed on the bus line of departure T_j are excluded;
- vehicles that already performed $maxNumInterrupts$ services are excluded;
- vehicles in a depot that were already used are preferred. This contributes to the maximization of vehicles reuse;

- by default, even if $|AdmVD_i| > 1$, only one vehicle is selected. This reduces the cardinality of $AdmVD_i$ and improves performance, however, good candidates may be ignored. Therefore additionally, a number of candidates $NumVDCandidates$ can be specified in order to take into consideration $NumVDCandidates$ instead of only one. When $NumVDCandidates$ is specified, the vehicles to be added to the set $AdmVD_i$ will be selected based on the arrival time, in increasing order.

When computing the set $AdmVP_i$ the following decisions are made:

- vehicles that are not allowed on the bus line of departure T_j are excluded;
- if no line exchanges are allowed, vehicles in a different bus line are excluded. Additionally, in this situation, we also exclude vehicles that are on the opposite terminal of the same bus line;
- vehicles arriving at their current terminal $term_j \in P$ at minute ma, awaiting for more than $maxDurAtTerminal_j$ minutes, are sent to the corresponding depot d_k with arrival:

  $$ma + duration(term_j, d_k) \tag{3.1}$$

  This originates a *VoidOut* task in which *smin* corresponds to ma and *emin* corresponds to $smin + duration(term_j, d_k)$. Since the departures are sorted by departure time in increasing order, when a vehicle that exceeds the $maxDurAtTerminal_j$ is sent to the depot while processing a departure T_j, we know that the vehicle was not selected for previous departures T_l where $l < j$ and, if it exceeds the $maxDurAtTerminal_j$ at T_j it will also exceed it for all T_l such that $l > j$. This reasoning is valid also for vehicles that are sent to maintenance.

From both origins (depots and terminals) only vehicles whose last trip is compatible with departure T_j are considered, i.e., vehicles are considered if the following condition is verified:

$$arrival\_minute + duration(loc_j, it_j) \leq s_i \tag{3.2}$$

were $arrival\_minute$ is the minute of arrival of the vehicle to the current location loc_j and $duration(loc_j, it_j)$ is the travelling time from location loc_j to location it_j.

As already described, each type of vehicle has a corresponding maximum working time. Our algorithm attempts to exhaust as much working time as possible, without violating the limit imposed. In order to ensure that no violations occur, when a vehicle is chosen as admissible, it needs to have not only sufficient working time to perform the trip T_j but also to travel to the depot.

This criteria is defined as follows. Vehicles in terminals can wait such that a trip to the depot is avoided (we describe all the possible wait situations in section 3.1.1.2). Let $waitingTime$ be the total waiting time of a vehicle stationed at terminal $term_i$ or in a

different terminal, of departure T_j. Assuming that the vehicle was chosen, *waitingTime* is defined as:

$$waitingTime = s_j - (arrival\_minute + duration(loc_j, it_j)) \tag{3.3}$$

Then *waitingTime* is normalized as:

$$waitingTime = \begin{cases} waitingTime & , waiting\_time \leq MaxWaitingTimeOnService \\ 0 & , otherwise \end{cases} \tag{3.4}$$

where *MaxWaitingTimeOnService* is the maximum waiting time in which a vehicle is considered to be in service (working). This normalization models the situation in which a service interrupt occurs and the vehicle does not spend any working time while waiting (e.g. the driver turned off the vehicle).

Based on the definition of *waitingTime*, the vehicle working time after performing departure T_j and travelling to its corresponding depot *workTimeAfter* is defined as:

$$\begin{aligned} workTimeAfter = CurrentWorkingTime(v) \; + \; duration(loc_j, it_j) + waitingTime \\ + (e_j - s_j) + \epsilon_{min} + duration(ft_j, depot\_loc) \end{aligned} \tag{3.5}$$

where $CurrentWorkingTime(v)$ denotes the current working time value of vehicle v.

Vehicles are considered if the following condition, with respect to vehicles working time, is verified:

$$workTimeAfter \leq MaxDailyWorkingTime(v) \tag{3.6}$$

where $MaxDailyWorkingTime(v)$ denotes the maximum daily working time of vehicles whose type corresponds to the type of vehicle v.

Lastly, vehicles from both origins are sent to maintenance while building the admissible sets. The difference between a vehicle being in a depot or in a terminal is that in a terminal vehicles must travel first to the corresponding depot and only then perform maintenance.

As with the working time criteria, the algorithm decides to send a vehicle v to maintenance if after performing the trip T_j, travelling to the vehicle depot exceeds the maximum time between maintenance tasks, i.e:

$$workTimeAfter - LastMaintenance(v) > MaintenanceLimit(v) \tag{3.7}$$

where $LastMaintenance(v)$ corresponds to the time of last maintenance task of vehicle v and $MaintenanceLimit(v)$ the maximum time between maintenance tasks of vehicles whose type corresponds to the type of vehicle v. This originates a task of type *Maintenance* with $smin = arrival\_time$ and $emin = smin + MaintenanceDuration(v)$, where $MaintenanceDuration(v)$ denotes the duration of a maintenance episode of vehicles whose type correspond to the type of vehicle v.

When the vehicle is not at its depot it must travel first to the depot and only then maintenance can be performed. In this situation a *VoidOut* task is originated with $smin = arrival\_time$ and $emin = duration(loc_j, d_v)$, where d_v denotes the depot of a vehicle v.

3.1.1.2 Vehicle Assignment to departure

After choosing a vehicle v for trip T_j based on a given heuristic, the vehicle is assigned.

When assigning a vehicle, the following situations regarding the current location loc_j of v may occur:

Vehicle on Depot - when loc_j corresponds to a depot, v must travel to the departure terminal it_j. A task of type *VoidIn* is created with $smin = e_j - duration(loc_j, it_j)$ and $emin = e_j$. Since we assume that 5:00 AM corresponds to minute 0, a special case may occur when $e_j - duration(loc_j, it_j) < 0$, meaning that the task must start at a given point in time before 5:00 AM. In this situation, the $smin_j$ is irrelevant[1];

Line Exchange - When $loc_j \neq it_j$ then v must travel to it_j and only then can perform the trip T_j. In this situation the following two tasks are generated:

- A *LineExchange* task in which a vehicle travels in void without passengers from its current terminal loc_j to the departure T_j terminal it_j, with $smin = arrival\_minute$ and $emin = duration(loc_j, it_j)$;
- A *Wait* task is created if $emin_{LineExchange} < s_j$ where $emin_{LineExchange}$ is the ending minute of the line exchange task. This means that if the vehicle arrives the departure terminal task sooner, it must wait the remaining time to reach s_j. The starting minute *smin* of wait task is set to $smin = emin_{LineExchange}$ and the ending minute is set to $emin = s_j$;

Vehicle already on terminal it_j - If the vehicle is already at the departure terminal, i.e. $loc_j = it_j$, then the vehicle must wait until minute s_j. A *Wait* task is originated with $smin = arrival\_minute$ and $emin = s_j$.

Once the vehicle is at the current departure terminal a *InService* task is originated with $smin = s_j$ and $emin = e_j - s_j$. After performing the trip T_j the vehicle arrival time is updated as follows:

$$arrival\_time = e_j + \epsilon_{min} \tag{3.8}$$

The minimum time between an arrival and a departure constraint is enforced by making the vehicle available only after arriving the terminal and waiting the respective minimum time ϵmin.

In order to enforce the maximum number of vehicles in a terminal constraint, after the vehicle v performing the departure, the algorithm must decide to send it or not to its

[1] When interpreting the solution one must take this case into consideration.

depot. This decision is based on the following rule:

$$sendToDepot = \begin{cases} true & , numVehiclesAtTerm_j < maxNumVehiclesAtTerminal_j \\ false & , otherwise \end{cases} \quad (3.9)$$

where $sendToDepot$ is a boolean variable which indicates if the vehicle must be sent to depot ($true$) or not ($false$), and $numVehiclesAtTerm_j$ denotes the number of vehicles stationed at terminal it_j. When the decision is to send the vehicle to the depot, a $VoidOut$ task is originated with $smin = arrival\_time$ (the updated arrival time) and $emin = duration(ft_j, d_v)$, where d_v denotes the depot of vehicle v.

As discussed in section 1.3.1, when a vehicle is idle for more than $maxIdleTimeInService$, on a depot (if already departed) or in a terminal, a service interrupt occurs. In this case, a $ServiceInterrupt$ task is created with $smin = arrival\_time$ (before the update) and the number of service interruptions of vehicle v is incremented by 1. The ending minute $emin$ of this task is irrelevant.

3.1.2 Vehicle Selection Heuristics

In this section we will describe a set of heuristics used to select a vehicle, in the context of assigning a vehicle to a departure T_j, from the set of vehicles $AdmV_i$.

We propose three heuristics:

- Random
- Probabilistic based on vehicle utility
- Vehicle Utility selection - Greedy

In the next sections we introduce each of the heuristics mentioned above.

3.1.2.1 Random

The random heuristic is also a simple and straightforward heuristic which consists in selecting a integer number $r \in [1, |AdmV_i|]$ using a uniform distribution of discrete integer values in the same range as r.

The random heuristic is defined as follows:

$$hF_{random} = selectFromUniformDistribution(\ [1, |AdmV_i|]\ ) \quad (3.10)$$

where $selectFromUniformDistribution(interval)$ is a function that produces random integer values based on a pseudo random number generator function over the range $interval$. Produced values are distributed according to the following discrete probability function:

$$P(val|lb, ub) = \frac{1}{ub - lb + 1} \quad (3.11)$$

where $val \in [lb, ub]$ denotes a given value, lb and ub denote the lower and upper bound of the desired range of values (in this case $lb = 1$ and $ub = |AdmV_i|$).

Since with this heuristic vehicles are chosen randomly, no attempt to minimize the objective function is made. Consequently, this heuristic will produce bad quality solutions. However, it introduces randomization in the algorithm, which is essential to implement multi-start based metaheuristics. This heuristic can be used as comparison criteria for more sophisticated methods (either heuristics and/or metaheuristics) since bad solutions are expected.

3.1.2.2 Probabilistic based on vehicle utility

Both heuristics previously described are very naive and are not able to produce high quality solutions. The greedy heuristic blindly selects always the best option at each departure assignment which consists in optimizing each local choice but not the complete solution produced. The random heuristic does not even attempts to minimize the objective function but introduces randomization.

We developed an additional probabilistic heuristic which introduces randomization and attempts to minimize the objective function of the problem.

An utility value is assigned to each vehicle. The utility value depends on the current state S_i of the algorithm and on the trip T_j being processed.

The objective is to select a vehicle with a given probability, which is computed based on its utility value. The rationale is that vehicles with greater utility values are more likely to be selected. Additionally, it should minimize both terms of the objective function.

In order to take into account several factors, where some are more important than others, we model our expression using a linear combination of these factors. The utility $\eta(v)$ of a vehicle v on a location loc_v performing a departure T_j starting in terminal it_j, is a function $\eta : \mathbb{R}_{\geq 0} \mapsto]0, 1]$ defined as follows:

$$\eta(v) = e^{\frac{-\left[dist(loc_v, it_j) * distTypeCost(v) * k_1 + dTerm(loc_v, it_j) * k_2 + newVehicle(v) * k_3\right]}{k_4}} \tag{3.12}$$

where $dist(t_1, t_2)$ is the distance from t_1 to t_2, $dTerm(t_1, t_2)$ is a boolean function that returns 1 if t_1 is a terminal and $t_1 = t_2$ or 0 otherwise, $newVehicle(v)$ is a boolean function that returns 1 if v is a new vehicle (is its first trip) or 0 otherwise, and k_w, where $w \in [1, 2, 3, 4]$, are weights that can be parametrized.

Each of these terms are described as follows:

- $dist(loc_v, it_j) * distTypeCost(v)$ - Denotes the incremental cost (in terms of distance travelled) in € of performing the current departure with vehicle v. This term is aligned with the objective function distance travelled factor and can be used to minimize it;

- $dTerm(loc_v, it_j)$ - Captures the occurrence of a line exchanges. When $dTerm(loc_j, it_j) = 1$ a line exchange occurs, therefore, this term can be used to control the influence of line exchanges in the vehicle utility;

- $newVehicle(v)$ - Captures the situation in which a new vehicle (a vehicle that has not performed any departure yet) is needed to cover the departure. This term is also aligned with the objective function total number of vehicles factor and can be used to minimize it.

The utility expression is dimensionless due to the introduction of weights. The weights are also used to control the influence of each of the terms described above in the definition of a vehicle utility.

In order to ensure correctness, terms must be normalized to the $[0, 1]$ range. Terms are normalized as follows:

- Let $P_i = dTerm(loc_v, it_j) * k_2 + newVehicle(v) * k_3$. Given that weights k_2 and k_3 are defined on the range $[0, 1]$ and that $dTerm$ and $newVehicle$ have domain $\{0, 1\}$, normalization is implemented as:

$$P_{inorm} = \frac{P_i}{k_2 + k_3} \tag{3.13}$$

- Unlike the previous case, the normalization of the increment in cost the term cannot be static, since the domain of values of the term changes for each departure assignment. Despite the fact that the lower bound of the term can be set to 0, there may be situations in which the lower bound is not 0 (i.e. there are no vehicle in the departure terminal). Additionally the more costly assignment, which corresponds to the upper bound, also changes for each departure assignment. Therefore, in an assignment of a departure T_j, the set $AdmV_i$ is iterated in order to find the lower and upper bounds, Δ_{min} and Δ_{max}, respectively. Let $\Delta_{vnorm} = dist(loc_v, it_j) * distTypeCost(v)$. This term is normalized as follows:

$$\Delta_v = \frac{\Delta_v - \Delta min}{\Delta_{max} - \Delta_{min}} \tag{3.14}$$

Weights are used to control the influence of each term in the definition of a vehicle utility. Different parametrizations can be used not only to minimize the objective function but to achieve good real solutions to the problem. Despite the fact that real solutions are hard to define, some general guidelines may be straightforward like keeping the number of line exchanges small. Either way, these parameters should be choose according to transport services companies needs.

The negative exponential is used such that utility of a vehicle decreases exponentially. The rationale is that bad choices will get a very bad utility value. Figure 3.3 shows a plot of the function $y = e^{-x}$ with domain $]0, 1]$. By interpreting y as the utility of a vehicle and x as the value of the linear combination of the terms explained above, it can be seen that as the x value increases, the utility of a vehicle decreases exponentially.

One trivial decision is when a vehicle v is located on a terminal it_j of a departure T_j. It is expected that in this situation the vehicle utility is maximal. Our utility expression $\eta(v)$

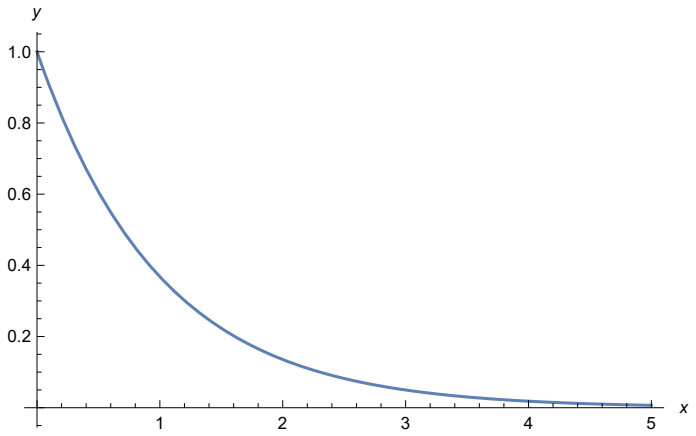


Figure 3.3: Plot of the function $y = e^{-x}$ in which the behaviour of the function can be observed

evaluates to 1 in this case (the upper bound value) since the value of the linear combination is 0 and $e^0 = 1$. Additionally, by using the exponential, values are implicitly normalized between 0 and 1.

Since e^{-x} converges quickly to 0 as x increases, the weight k_4 can be used to reduce the value of the linear combination such that utility values can get spread across the range $[0, 1]$ instead of laying close to 0.

The probabilistic heuristic works as follows:

1. a set $AdmV_i\_utilities$ is created where each element $u_v \in AdmV_i\_utilities$ corresponds the utility value $\eta(v)$ of a vehicle v;

2. the set $AdmV_i\_utilities$ is truncated, originating the set $NAdmV_i\_utilities$, such that only the N best elements are considered;

3. the probability of each element $u_v \in NAdmV_i\_utilities$, is computed based on a probability distribution over the utility values defined as follows:

$$P(v) = \frac{u_v}{\sum_{l=1}^{|NAdmV_i\_utilities|} u_l} \tag{3.15}$$

4. a vehicle v is selected with probability $P(v)$ by obtaining a pseudo-random number r from a uniform distribution over the range $[0, 1]$ and selecting, using the cumulative probability of all vehicles until r is reached.

The truncation step is introduced to avoid losing good decisions due to the probability distribution used. Figure 3.4 illustrates this problem. As we can see, a very good choice ($v1$) with utility 1 has a probability of 0.25 of being selected while the remaining 100 decisions, each with utility 0.04 and probability 0.008, have a total probability of 0.75. Therefore it is

likely that a bad decision will be selected instead of the good one. By reducing the number of elements to 11, the very good choice gets a probability of ≈ 0.71 and the remaining 10 decisions get a probability of ≈ 0.029 each, and a total of ≈ 0.29. Now, it is more likely that the selected vehicle will be the best one.

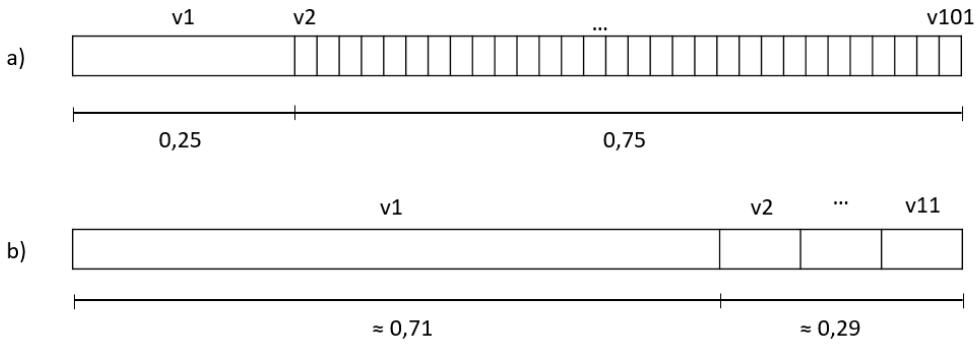


Figure 3.4: Illustration of the probabilities issue by considering a different number of vehicles while selecting a vehicle to assign to the current departure. In a) a vehicle $v1$ with utility 1 has 0.25 probability of being selected. The remaining vehicles from $v2$ to $v101$ have a total probability of 0.75, each with probability 0.04. In b) only 11 vehicles are considered. A vehicle $v1$ with utility 1 has an associated probability of ≈ 0.71 and the remaining vehicles have a probability of ≈ 0.29.

The number of elements N to consider must be chosen such that good decisions are not lost (assigned a low probability value) and such that a considerable degree of diversity is achieved, by allowing the algorithm to also explore bad decisions sporadically.

3.1.2.3 Vehicle Utility selection - Greedy

The probabilistic heuristic classifies for each departure assignment, the vehicles on the set $AdmV_i$ of the current state with a utility value. Then, using a probability distribution over the utilities, one vehicle is chosen. A straightforward heuristic can be achieved by enforcing a deterministic behaviour on the probabilistic vehicle utility presented on the previous section.

The greedy heuristic is defined as follows:

$$hF_{greedy} = \arg\max_{va} \ \eta(va), \ \ \forall\, va \in AdmV_i \tag{3.16}$$

This heuristic selects the vehicle with greater value of utility. When draws occur, the first vehicle (based on the order of iteration of the set) is chosen.

This heuristic is deterministic, therefore, no randomization is achieved. However, this heuristic is useful to provide an upper bound on the solutions quality, and establish meaningful comparisons.

3.2 Sequential and Parallel Probabilistic Restarts Metaheuristic

With the introduction of randomization in the algorithm described in section 3.1.1 by using the Random or the Probabilistic heuristics, a restarts metaheuristic can be developed.

The restarts metaheuristic consists in performing N distinct starts of our proposed algorithm, yielding N solutions. The best solution found so far is stored. The algorithm pseudo-code is outlined in algorithm 3.

Algorithm 3 Restarts Metaheuristic

```
1:  procedure RESTARTS-METAHEURISTIC return the best schedule found
2:      bestSolutionFound ← empty solution
3:      bestSolutionFoundValue ← +∞
4:      i = 0
5:      while i < m do
6:          currentSolution ← PTBA_ALGORITHM()
7:          tempSolutionValue ← currentSolution.objectiveFunctionValue()
8:          if tempSolutionValue < bestSolutionFoundValue then
9:              bestSolutionFound ← currentSolution
10:             bestSolutionFoundValue ← tempSolutionValue
11:
12:         i ← i + 1
```

Despite of its simplicity, by performing several restarts and by exploring the randomization offered by the heuristics (specially from the probabilistic heuristic) this metaheuristic is able to achieve a good degree of diversity. Additionally, as discussed in section 2.1.2.1, in order to maximize the portion of the search space explored, as much as possible solutions must be produced. Therefore, the value m must be as large as possible.

The value m is restricted by the computation time needed to produce m solutions. Generally speaking, in a sequential setting, the computation time needed to execute the algorithm is $m * sol_t$, where sol_t denotes the time needed to produce a single solution. As m increases, the computation time is expected to increase linearly. By exploring parallelism, larger values of m may become feasible.

Since each algorithm execution (restart) is independent, a straightforward parallelization strategy consists of performing p executions in parallel. This strategy follows the *multiple-walk* approach with multiple independent walks, i.e., executions are completely independent and no interaction occurs.

Theoretically, by performing p executions in parallel, p solutions can be produced in sol_t units of time. Therefore, in the same period of time, the parallel algorithm is able to produce p times more solutions than the sequential algorithm.

3.3 Ant-Colony algorithms

The sequential algorithm presented in section 3.1.1 is characterized by a somewhat greedy behaviour. The developed parallel restarts metaheuristic, presented in section 3.2 inherits

this behaviour since it is based on the sequential algorithm. This is due to the fact that, even with the probabilistic heuristic, decisions are made locally (for each departure assignment) and each start executes independently, disregarding any valuable information obtained during previous executions. Consequently, the metaheuristic performs a poor exploration of the search space. The search is not guided properly towards promising regions, harming not only the degree of diversity achieved but also intensification on promising regions.

As discussed in section 2.3, the ACO metaheuristic performs a more sophisticated exploration of the search space through a stochastic learning mechanism. We develop a model of the PTBA problem for the ACO metaheuristic and two algorithms, one based on the AS and one based on the $\mathcal{MAX}$–$\mathcal{MIN}$ algorithm. The model and the developed algorithms will be presented in the following sections.

3.3.1 PTBA Ant-Colony Model and Ant-System algorithm

In order to develop an algorithm for the PTBA problem based on the ACO metaheuristic the following components must be defined:

Agent Algorithm - Each agent (an ant) must execute an algorithm that follows the constructive approach, in order to construct an admissible solution.

Pheromone matrix model - the pheromone matrix holds information about the desirability τ_{ij} of each solution component c_{ij}. A solution component must be defined in the context of the problem.

Each agent will use our sequential algorithm to construct solutions since it meets all the requirements.

In our model each component c_{ij} added to a solution corresponds to an assignment of a vehicle i to a departure T_j. Let $L = P \cup D$ be the set of possible locations from where a vehicle can be picked when assigning a vehicle to a given departure (depots and terminals). Our model corresponds to a $|L| \times |T|$ matrix, i.e., we assign a pheromone value τ_{ij} to each pair $\langle l_i, T_j \rangle$ that denotes the desirability of assigning vehicles that are in a location $l_i \in L$ to a departure $T_j \in T$. The pheromone model is represented by the following matrix:

$$\begin{array}{c} \\ l_1 \\ l_2 \\ \vdots \\ l_{|L|} \end{array} \begin{array}{c} \begin{array}{cccc} T_1 & T_2 & \dots & T_{|T|} \end{array} \\ \begin{pmatrix} \tau_{11} & \tau_{12} & \dots & \tau_{1|T|} \\ \tau_{21} & \tau_{22} & \dots & \tau_{2|T|} \\ \vdots & \vdots & \ddots & \vdots \\ \tau_{|L|1} & \tau_{|L|2} & \dots & \tau_{|L||T|} \end{pmatrix} \end{array} \tag{3.17}$$

where $l_1, \dots, l_{|L|}$ correspond to the possible locations L.

Our pheromone model is actually based on a relaxation. Each c_{ij} is based on a given vehicle i, however, in our model we do not take into account the desirability of assigning a specific vehicle i to a given departure. Instead, we model the desirability of assigning a

vehicle from a specific location l to a departure. This is due to the fact that the number of vehicles may be very large. Since at each assignment, from the probability distribution expression 2.1, it follows that for each departure, the probability of each vehicle must be computed. Consequently, this impacts negatively the performance of the metaheuristic.

Another observation is that by taken into account specific vehicles instead of locations, the pheromone matrix would not only become very sparse but also with misleading desirability values. Considering a situation in which for the first departure T_0 the optimal decision is to assign a vehicle of type 1 from location l. If we have 100 vehicles of type 1 in a location l and agents perform this assignment, when the pheromone values are updated the amount of pheromone deposited will be spread across the vehicles pheromone values. With our model all the agents would update the same matrix entry, emphasizing the assignment.

3.3.2 $\mathcal{MAX}$–$\mathcal{MIN}$ Algorithm

By extending our AS algorithm, we developed a ACO metaheuristic based on the $\mathcal{MAX}$–$\mathcal{MIN}$ algorithm. As discussed in section 2.3.2, the $\mathcal{MM}$AS algorithm is more sophisticated and more capable of avoiding premature convergence to a local optimal solutions and perform a stronger exploration of the search history.

The following extensions were added to our AS algorithm:

- in [69] the authors show that better results may be obtained by using s^{ib}, therefore, the pheromone update is now performed based only on the agent that produced the best solution s^{ib} of the current iteration;
- the lower bound τ_{min} is initialized with τ_{max}/a where a is a parameter and the upper bound τ_{max} is initialized with an arbitrary constant;
- a pheromone reinitialization mechanism is implemented using a fixed number IT of iterations to infer stagnation.

3.4 Parallel Synchronous and Asynchronous ACO Algorithms

The Ant-Colony based algorithms presented in section 3.3.1 and 3.3.2 can be naturally parallelized by using the multiple-walk approach. Additionally, cooperation may be achieved by using the stochastic learning component of the ACO metaheuristic, originating a multiple interacting walks search algorithm. We developed both synchronous and asynchronous algorithms based on the AS targeting a shared-memory architecture. For the $\mathcal{MM}$AS algorithm only a synchronous parallel algorithm is developed since as it will be explained, this algorithm is not suited for an asynchronous scheme. These algorithms will be presented and discussed in the following sections.

3.4.1 Synchronous ACO Algorithms

A parallel synchronous scheme keeps the original semantics of the ACO metaheuristic. More concretely, at each iteration, N agents construct solutions which are then used to update the pheromone matrix. In the AS algorithm the update is performed by using all the solutions constructed and in the $\mathcal{MM}$AS algorithm only the best solution constructed is used.

Given that at each iteration, N agents build a solution independently, i.e., despite the fact they all access the pheromone matrix to perform decisions, their progress is completely independent from others agents. This observation is valid for the AS and $\mathcal{MM}$AS algorithms, however, for the ACS algorithm this is not true since each agent performs local updates to the pheromone matrix while building solutions. Nevertheless, in our work we will not develop a parallel algorithm based on ACS.

3.4.1.1 Ant System Parallel Synchronous Algorithm

The Ant System Parallel Synchronous ACO algorithm is based on the general ACO algorithm 1, which for the sake of clarity is reproduced in this section in algorithm 4, in which a parallel region is introduced on line 4 (*ConstructAntSolutions*).

Algorithm 4 Parallel Ant-Colony Optimization.

```
1: procedure PARALLEL ANT-COLONY OPTIMIZATION(problem) return best solution found
2:     Initialization
3:     while termination condition not met do
4:         ConstructAntSolutions          % Performed in parallel
5:         UpdatePheromoneTrails
       return best solution
```

This parallel region achieves the behaviour illustrated in figure 3.5. Agents are given a task which consists of executing a given algorithm in order to construct a solution. Each agent will perform its own task in parallel, reporting the solution constructed (or failure, if no solution could be constructed). The algorithm is synchronous since an iteration ends only when all the agents finish their task.

The AS mechanisms of update and pheromone evaporation are performed outside the parallel region at the end of an iteration.

Theoretically, with this algorithm, almost linear speedup may be achieved. Even theoretically, and based on the Amdahl's Law defined on equation 2.11 of section 2.4, the speedup would never be linear since the update and evaporation mechanisms are still performed sequentially.

Despite of its simplicity, one interesting aspect of this parallelization strategy is that it introduces cooperation within agents. This cooperation, despite of being performed indirectly, contributes to a better search exploration and exploitation of the search space in a reduced amount of time. The pheromone trails, are defined over the problem search space,

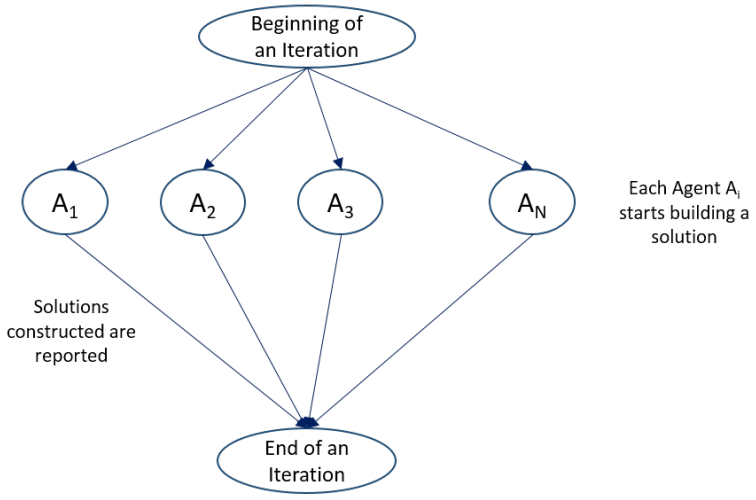


Figure 3.5: Illustration of the parallel synchronous algorithm behaviour on each iteration.

therefore, parallel search agents cooperate in completing and evolving the pheromone trails matrix which gathers knowledge obtained during the search, by constructing solutions in parallel and reporting the components of those solutions. This approach differs from the works analysed in section 2.5.1 since the shared knowledge of ACO gives much more insight about the problem characteristics.

This implicit cooperation mechanism when following a synchronous scheme, has a predictable behaviour, i.e., the influence of the cooperation mechanism on the search algorithm is the same as in a sequential setting. As it will be discussed in section 3.4.2, when following an asynchronous approach different interesting search behaviours are achieved since now agents influence each other during the solution construction phase.

However, the synchronization barrier, at the end of each iteration, causes the straggler problem. This problem can be mitigated by following an asynchronous scheme, as it will also be discussed in section 3.4.2.

3.4.1.2 $\mathcal{MAX}$–$\mathcal{MIN}$ Parallel Synchronous Algorithm

Since the $\mathcal{MM}$AS is an extension of the AS algorithm, it can be parallelized by following the same approach: adding a parallel region on line 4 of ACO algorithm 1.

Parallel schemes for the $\mathcal{MM}$AS must be synchronous. This is due to the fact that at the end of an iteration, only the best solution is used to perform the update.

If we applied the exactly same approach as in the AS algorithm the computation of the best solution would be performed sequentially and would have a complexity of $O(N)$ in the worst case. This approach clearly would have a negative impact on the maximum possible speedup achieved. However, if at any moment agents know the objective function value of the best solution found on the current iteration, they can compare this value with the objective function value of the solution just constructed and update the iteration best solution, if that is the case.

In order to support this mechanism the following set of operations must be atomic:

- read the current iteration best solution value (read);
- compare this value with the value of the solution produced (comparison);
- update the iteration best solution value if a better solution is found (write).

With this mechanism, the task of finding the best solution value of an iteration has a time complexity of $O(1)$ since the search is distributed by each agent. A small overhead is imposed from performing the set of operations described above as an atomic operation. This overhead should not be very significant since all the operations are simple and can be executed very quickly on the hardware.

3.4.2 Asynchronous ACO Algorithms

The first step in order to develop an asynchronous parallel scheme consists in choosing a base ACO algorithm.

The $\mathcal{MM}$AS and the *ACS* algorithms include mechanisms to avoid premature convergence and consequently may achieve better results than AS, in a sequential setup. However, these mechanisms affect negatively the exploitation of parallel resources. The most prohibitive aspect is that either in $\mathcal{MM}$AS and *ACS* algorithm instead of updating the learned information with all the N agents solutions, only the best solution is used to perform the update. This property naturally implies that the algorithm has to wait for all the agents to finish the current iteration in order to proceed.

The issue with the *ACS* algorithm is that agents perform local updates while building solutions, i.e., each time a solution component c_{ij} is added to a partial solution s_p, the pheromone matrix cell τ_{ij} is updated. In a parallel setup, this would cause a high degree of concurrency due to the high number of accesses to the pheromone matrix. In a synchronous algorithm, at each iteration, N agents would be concurrently accessing the matrix, however, in a asynchronous algorithm this number can potentially be larger.

Therefore, we conclude that AS is more suitable to design an asynchronous parallel strategy.

3.4.3 Base Algorithm

Our algorithm follows the *coarse-grain master-slave* model in which a master executes the update and pheromone evaporation mechanisms and controls the tasks assigned to the slaves.

The main modification of our asynchronous model with respect to the synchronous parallel ACO algorithm, also illustrated in figure 3.6 , is the following:

> The master generates a set of $P \times N$ tasks, where P is the total number of iterations. Each task belongs to a given iteration $p \in P$ and a given search agent $n \in N$. These tasks are executed asynchronously and the results are sent to the master.

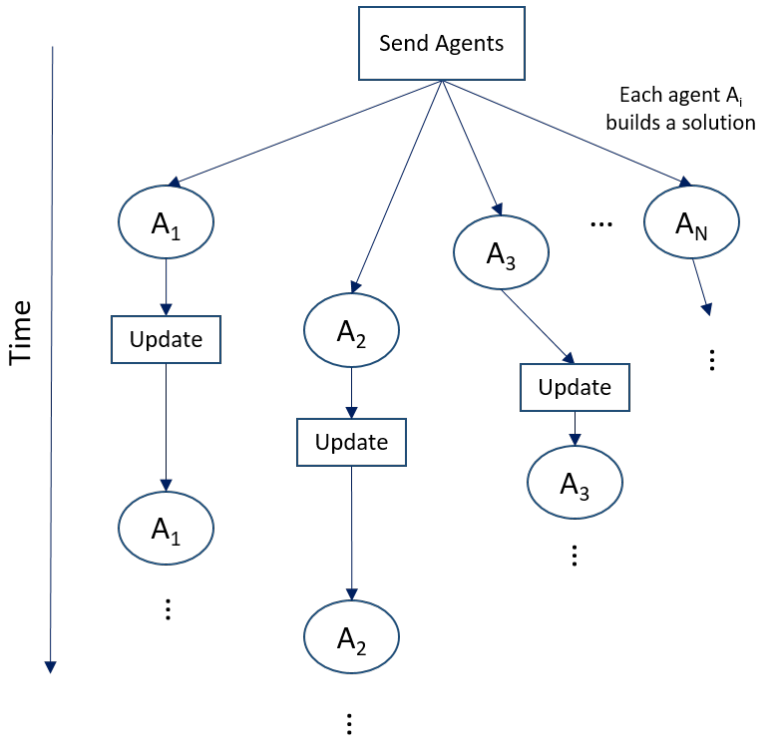


Figure 3.6: ACO behaviour after introducing asynchronism.

It is worth nothing that the concept of an iteration, with the semantics of an iteration of the original ACO algorithm, is now merely virtual. This is due to the fact that as soon as an agent finishes, it starts a new task immediately, instead of waiting for the remaining agents.

Additionally, as illustrated in figure 3.7, it may now occur that a search agent that is constructing a solution views an update to the matrix. This is the situation that breaks the original semantics of the ACO algorithm and can potentially affect negatively the learning capabilities of the algorithm. In order to approximate the behaviour of the algorithm, namely the learning capabilities of ACO, the stochastic learning component must be reviewed.

The following modifications were made:

Pheromone Update - When an agent k finishes the search, the master is notified and performs a partial update, using equation 2.3, based on the only generated solution;

Pheromone Evaporation - It is not possible to maintain the evaporation mechanism semantics of the original algorithm, i.e., at each iteration, all the N agents use values of the pheromone matrix that have been evaporated in the previous iteration. We can attempt to approximate this behaviour by noticing that in the original AS algorithm the total number of evaporations performed is equal to P.

Since now the agents execute asynchronously agents from an iteration p_1 with $p_1 > p_2$ may finish first than agents from an iteration p_2 and we do not know in

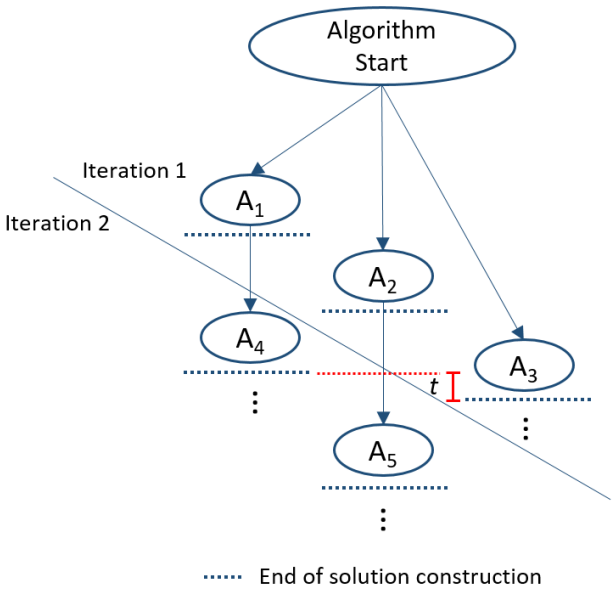


Figure 3.7: Illustration of the parallel asynchronous algorithm behaviour. We can see that an agent from a virtual iteration 2 finishes first (t units of time) than an agent from an iteration 1.

each iteration how many agents will finish first, since it depends on the real-time decisions made by the operating system scheduler.

In order to achieve an approximate behaviour, an evaporation step is performed after $N/2$ search agents from an iteration p, finish their execution. This assumes a pessimist scenario where half of the search agents of each virtual iteration may be late. However, this is not a problem since the total number of evaporations steps is the same as in the original algorithm.

With this strategy, it is expected that evaporation steps may be performed sooner than they would be on the original algorithm, which is also not a problem since there is an attempt to perform evaporation steps after an average of x agents have finished, although this is not guaranteed.

Additionally, even if some agents take a huge amount of time compared with the remaining ones, they will not affect our algorithm as long as $N/2$ agents of each virtual iteration execute adequately. If this is verified, evaporation steps will be well distributed across the search process duration.

Ideally an evaporation step would be performed every time x search agents finish. We believe that our proposed scheme is a good approximation of this behaviour and its contribution to the algorithm learning capabilities is evaluated in section 5.3.1.

Despite of the new algorithm conceptual simplicity, it introduces concurrency on the pheromone matrix. The master will be regularly performing updates while agents will

be reading the matrix contents. To handle this behaviour we developed three different concurrency control mechanisms which will be described in the next section.

3.4.4 Concurrency Models.

We propose three different concurrency control models, that differ on the degree of concurrency allowed while manipulating the pheromone matrix. Each model affects the original algorithm semantics in a different manner, therefore, the ACO learning capabilities for each model must be assessed. Additionally, different cooperation behaviours are achieved with each model.

Blocking Matrix (AsyncBM) - In this model when a master is performing updates the whole matrix is blocked and none of the agents can proceed until the update is performed. The same happens when an agent wants to access the matrix. With this model if an update is performed, all the agents will receive the complete update of all the solutions components;

Blocking Column (AsyncBC)- When an agent is choosing a component c_{ij} to a partial solution, it will read an entire column of the pheromone matrix, i.e., selecting from which location i should a vehicle be picked to perform the departure j. In this model, instead of blocking the entire matrix, only the column that is being updated/read is blocked. This ensures that if updates/reads are going to be concurrently performed, each agent will decide on each assignment based either on the new or the old information, but not on a mix of both;

Locking-Free (AsyncLF)- In the limit, we can allow all the concurrent updates and a search agent can possibly use both information before an update and information after an update, on the same decision. Conceptually this achieves better diversification and in practice maximizes concurrency. This model assumes that reads and writes operations are atomic.

From the three models proposed, as the degree of concurrency allowed increases, the more the original semantics of the original algorithm are lost. Now, agents may see updates sooner since as soon as an agent finishes, the solution components from its solution will have their correspondent pheromone values augmented.

This can be beneficial in some situations and bad in others. By seeing updates sooner, each agent search starts being influenced sooner from previous searches. If the first agents to finish contributes only with bad solutions, it means that bad solution components may have higher desirability values than good ones. This situation can be solved by using the heuristic. From the AS probability distribution (equation 2.1) we know that two aspects are taken into account: the pheromone value of each component and the heuristic value. In order to overcome the situation just described, the parameter β should have a value that allows the heuristic aspect to have some real significance in the final probability value.

Together with the evaporation mechanism, it is expected that bad solution components will have their pheromone values decreased over time.

Hereupon, this is beneficial since the algorithm should perform a slow start, i.e. none of the solution components will have high pheromone values, what means that even (locally) bad solution components will be considered. This increases the algorithm capability to increase diversity, which as stated in section 2.1.2.1, is crucial in order to achieve high-quality solutions.

CHAPTER 4

IMPLEMENTATION - PARALLEL PTBA PROBLEM OPTIMIZATION LIBRARY

This chapter covers all the implementation related aspects and tools used and the materialization of each algorithm will be presented in detail.

First, we will present in section 4.1, the tools and technologies used to implement the Parallel PTBA Problem Optimization Library (ParallelPTBAP-OptLib) and the solver application for the PTBA problem. Since the library aims to be efficient, we will discuss how the tools chosen contribute to this objective. Furthermore, as already stated, in our implementations, namely in the base algorithm implementation of each agent, we use the features offered by CaSPER LS solver. This solver has been previously developed by us and a short description of its main components and features will be presented in section 4.1.2.

The ParallelPTBAP-OptLib software model and architecture will be presented in section 4.2. The software model was designed in order to be extensible and favouring reuse, such that new algorithms may be developed using our library with small effort. We will discuss in this last section how this is achieved. The base implementation of the library (base components) will be discussed in section 4.3.

In section 4.4 we will present and discuss the implementation of all the algorithms described in chapter 3.

4.1 Tools and Technologies

The ParallelPTBAP-OptLib library was implemented using the C++ language and the standard C++14. The C++ language is a general-purpose multi-paradigm programming language that is efficient and flexible, and supports several paradigms such as object-oriented programming, generic programming, functional programming[1], among others.

C++ is a compiled, strongly-typed unsafe language with a rich library that offers efficient data structures and algorithms.

Our choice of the C++ language is based on the fact that it is efficient, rich in functionality and as will be described below, most parallel frameworks and standards are based on it. Additionally, the CaSPER LS solver is also deployed as a C++ library.

As it will be discussed in section 4.2, our implementations relies on the mechanisms provided by the object-oriented paradigm and on generic programming, for which C++ provides great support.

Our library provides a set of configuration options in a XML file. We use the *pugixml* [42] XML processing C++ library for processing this file. This library consists of a DOM-like interface with rich traversal/modification capabilities.

To develop the entire library we use the Qt Framework [62] (version 5.4). This framework provides a rich environment for developing C++ applications and offers a wide variety of features:

Qt Creator - consists in a Integrated Development Environment which improves productivity by offering several features that eases software development;

Cross-Platform - the Qt framework provides cross-platform support. This requirement is essential such that solvers developed with our library can be deployed on different platforms;

Command Line Arguments Parsing - provides a mean for handling the command line options in a clean way;

Database Drivers Plugins - plugins which provide APIs to communicate with different databases are provided. These plugins provide an additional level of abstraction since databases can be changed without being necessary to change the solver code.

We used the open-source GCC 5.0 compiler [61]. Since we will use features from C++14 standard, which is recent, only GCC 5.0 or later versions support this standard.

Due to the amount of data to be processed when producing the results, after executing the algorithms, we used Python to automate the process. Concretely, we used the *matplotlib* [40] and the *Pandas* [56] libraries which provide great support for processing and analysing data. The *matplotlib* library provides 2D plotting features while the *Pandas* library provides data structures for analysing data. With *Pandas* data can be treated as a spreadsheet table in which operations like the ones available on the Structured Query Language (SQL) can be applied.

4.1.1 Shared-Memory Multi-Threading Tools

The C++14 standard offers built-in support for multi-threading applications with a wide variety of features like:

[1]The functional paradigm is supported since the introduction of the C++11 stantard.

Threading - cross-platform thread support;

Locks - Different lock implementations: mutexes, shared mutexes (Reader-Writer lock), among others;

Atomics - support for atomic operations on primitive data types or complex objects.

OpenMP [7] is an API that supports multi-platform shared-memory parallel programming in C/C++. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications. With OpenMP, low-level details are abstracted to the programmer, thus easing the development of parallel applications while assuring scalability and portability within different platforms. OpenMP programs follow a Fork–join [55] model, in which a master thread launches N slaves who execute some task in parallel and join back into the master thread, which continues its execution. Programmers annotate source code with OpenMP directives to develop parallel applications.

4.1.2 CaSPER LS

The CaSPER LS [68] is a declarative, efficient and extensible Constraint-Based Local Search solver, developed in C++. The solver is integrated in *CaSPER* [14] (Constraint Solving Programming Environment for Research), a programming environment for the development and integration of Constraint Solvers, developed at CENTRIA[2] which includes solvers for Finite Domains, Finite Sets and others.

The CaSPER LS implements and provides the following concepts:

Variable - Represents a decision variable with a given finite domain and an assigned value belonging to the domain. The domain is an interval of the form $[a, b]$ where a and b denote the lower and upper bound, respectively, of the domain. Each variable has a type associated that corresponds to the type of the values of the domain;

Expression - Models a set of relations (an empty set for constants or variables) between constants, variables or other expressions. Expressions can be formed compositionally by applying arithmetic, functional or logical operators. A value of an expression is obtained by recursively computing the value of each relation in the expression;

Constraint - Specifies a relation between one or more expressions. Each constraint has a total number of violations that is an expression. The following types of constraints are supported: arithmetic, logical and the global constraint All-Different. Like expressions, new constraints can be formed compositionally by applying arithmetic, functional or logical operators;

[2]CENTRIA (Centre for Artificial Intelligence) is a research centre from the Faculty of Science and Technology of the New University of Lisbon, which was recently merged with another research unit, originating the NOVA-LINCS centre.

Constraint System - Describes a conjunction of constraints. The total number of violations of a Constraint System is the sum of all the violations of each individual constraint in the Constraint System;

Objective Function - An objective function can be defined as an expression or the user can provide a custom implementation;

Solver - Represents an abstract entity where information of all the variables can be obtained (ex. statistics).

In our implementations we use these concepts to implement a Constraint Manager and the problem objective function, as will be described in more detail in section 4.3.

4.2 Software Model

A complete application for solving the PTBA problem was implemented. This application is based on the ParallelPTBAP-OptLib library. The library architecture, as depicted in figure 4.1, is composed by two layers:

Data Layer - contains all the components that store, provide and obtain data about the problem. More concretely this layer contains a component which is responsible for abstracting the origin of the input data and a component for storing the data of each PTBA problem instance;

Core Layer - contains the components that are used to solve the problem. This layer comprises all the algorithms and tools developed and all the PTBA problem specific components. In particular, this layer contains the following metaheuristic related components:

- Metaheuristic Algorithms
- Metrics
- Utilities (thread pool, solutions pool, among others)

This layer also contains the following problem related components:

- Agent Algorithm
- Problem Representation
- Constraint Manager
- Heuristics
- Objective Functions

Both layers will now be described in more detail in the next sections.

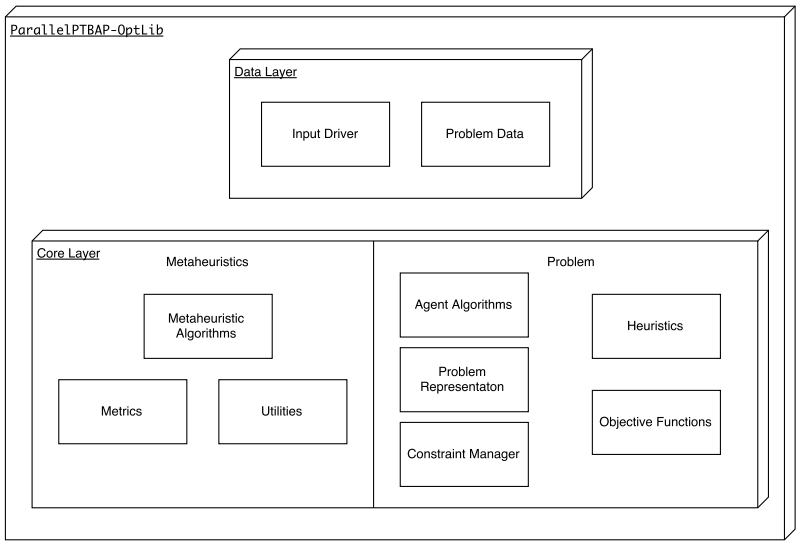


Figure 4.1: Application Architecture.

4.2.1 Data Layer

In order to allow users to obtain input data from different sources without needing to perform changes to existent developed solutions using our library, the data input source is abstracted, for the algorithms implementation.

To achieve this, we implemented an intermediate class `ProblemData` which different data input loaders classes can fill. This intermediate class specification is known by the internal classes. Therefore, driver classes, which are responsible for creating and populating an object of class `ProblemData`, can be implemented such that different data sources can be accessed.

The `ProblemData` class abstracts all the data loading details (and sources) to the rest of the library classes.

Listing 4.1: ProblemData Class

```
1  /**
2   * ProblemData Class - This class contains data about a given instance
3   * of the problem and is used to abstract different data sources
4   *
5   */
6  class ProblemData {
7
8      RouteData getRoute(routeIdentifier);
9      VehicleTypeData  getVehicleType(vehicleTypeIdentifier);
10     int getNumberNodes();
11     int getNbVehicles(vehicleTypeIdentifier, nodeIdentifier);
12     bool isAllowedVehicle(vehicleTypeIdentifier, routeIdentifier);
13     int getDistance(nodeIdentifierA,nodeIdentifierB);
14     int getDuration(nodeIdentifierA, nodeIdentifierB);
15     int getMaxIdleTimeInService();
16     int getMaxDurAtNode(nodeIdentifier);
```

```
17      int getMaxNumberVehiclesInNode(nodeIdentifier);
18      int getMinTimeBeforeDeparture();
19  }
```

Listing 4.1 shows the `ProblemData` class. This class provides access to the problem routes, to the different vehicle types and its associated parameters, to information from different nodes (terminals and depots), to the $maxNumVehiclesAtTerminal_i$ values and to $maxDurAtTerminal_i$ of each terminal $term_i \in P$.
The values $maxIdleTimeInService$ and ϵ_{min} are also provided.

Listing 4.2: RouteData and VehicleTypeData Class

```
1   /**
2    * RouteData Class -This class stores information of a given Route.
3    */
4   class RouteData {
5     typedef tuple<int,int,int> Duration;
6
7       int getTi();
8       int getTf();
9       int getBusLine();
10      string getDirection();
11      unsigned int getNumberDurations();
12      Duration getDuration(unsigned int i);
13  }
14
15   /**
16   * VehicleTypeData Class -This class stores information of a given Vehicle Type
17   */
18  class VehicleTypeData {
19      int getMaxWorkingTime();
20      int getMinMaintenanceTime();
21      int getMaxTimeBetweenTwoMaintenanceTasks();
22      double getCostPerDay();
23      double getCostPerKm();
24  }
```

Each route information is stored on objects of class `RouteData` and information from each vehicle type on objects of class `VehicleTypeData`. The `RouteData` class holds information of the terminal endpoints, the bus line $l \in L$, the direction (ascendant or descendant) and the durations. The durations are defined as a tuple $\langle lb, ub, dur \rangle$ where lb and ub are the lower and upper bound, respectively, of the period of time in which a vehicle takes dur minutes to perform a service on this route. Objects from class `VehicleTypeData` store information of each vehicle type like the maximum daily working time, the duration of a maintenance tasks, the maximum time between two maintenance tasks, the daily cost and the cost per kilometre.

4.2.2 Core Layer

As stated previously, the core layer comprises all the algorithms and tools to develop PTBA solving applications.

This layer was designed in order to provide good extensibility and reuse. More concretely, metaheuristics and algorithms were designed and implemented such that our library can be used to develop solutions deployed on different architectures. In this thesis our implementations target a shared-memory architecture, however, through compositionality, the library allows programmers to easily adapt the existing algorithms and/or metaheuristics to develop applications on different architectures (e.g. distributed architectures).

In figure 4.2 a partial class diagram of the core components of this layer is illustrated.

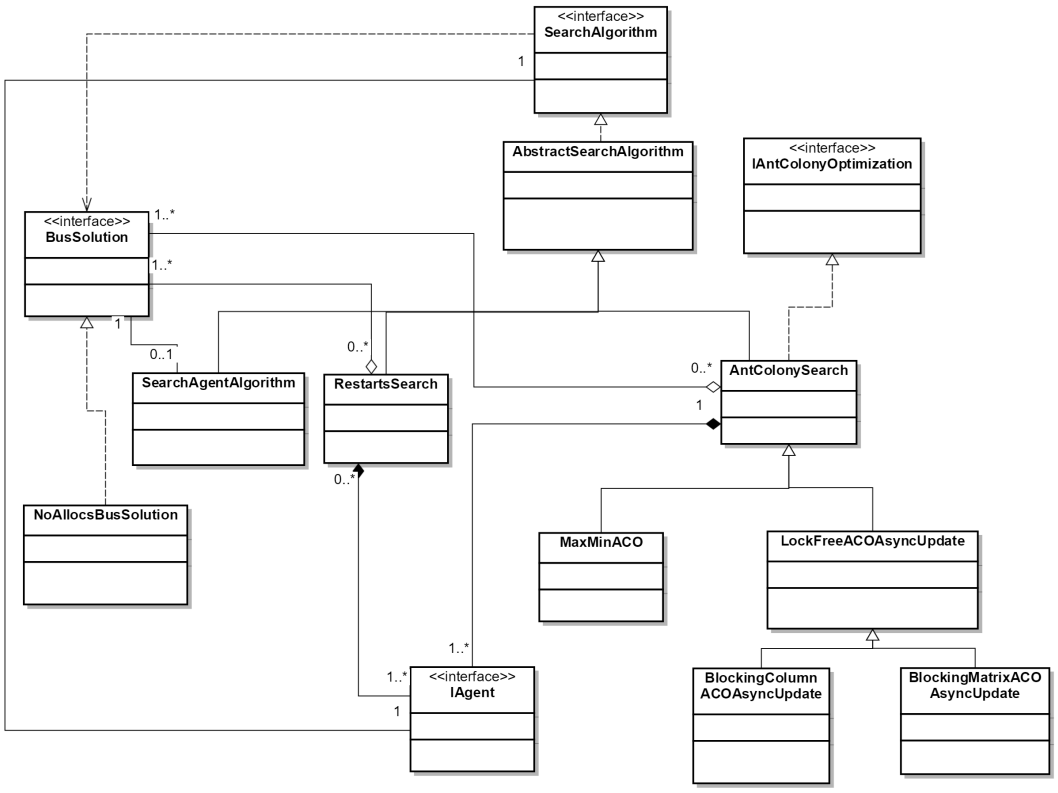


Figure 4.2: Partial Class Diagram of Metaheuristics and Algorithms

The agents Base Algorithm described in section 3.1.1 is implemented by the `SearchAgentAlgorithm` and the Restarts metaheuristic described in section 2.1.2.1 is implemented by the `RestartsSearch` class. The ACO metaheuristics mapping to classes are:

ACO Ant System - class `AntColonySearch`;

$\mathcal{MM}$**AS** - `MaxMinACO`;

Asynchronous Blocking Matrix - `BlockingMatrixACOAsyncUpdate`;

Asynchronous Blocking Column - `BlockingColumnACOAsyncUpdate`;

Asynchrinous Lock-Free - `LockFreeACOAsyncUpdate`.

For all classes, a parameter p is passed which corresponds to the number of agents that execute in parallel. The sequential algorithm is obtained with $p = 1$.

It can be seen that all algorithms and metaheuristics implement the `SearchAlgorithm` interface and extend the `AbstractSearchAlgorithm` class. Both `RestartsSearch` and `AntColonySearch` class use agents (classes that implement the `IAgent` interface) to perform the search. In our library, an agent executes a given search algorithm (an object whose class implements the `SearchAlgorithm` interface). This is the key aspect that contributes to a compositional model. For the ACO algorithms and for the Restarts metaheuristic, agents use an object of the class `SearchAgentAlgorithm` to perform the search and build solutions. However, different algorithms may be implemented by exploring this feature.

For example, despite the fact that we do not implement a Multicolony ACO metaheuristic, it can be easily implemented by defining the search algorithm of an agent to one of the ACO algorithms. Combining this procedure with a restarts metaheuristic, i.e., by using the restarts metaheuristics implementation in which the corresponding agents execute an ACO algorithm, we obtain a multicolony implementation without cooperation between colonies.

The interface `BusSolution` defines a set of operations related with a problem solution. Classes that implement this interface, like the class `NoAllocsBusSolution`, are responsible for defining the problem solution representation. Different problem representations can be provided since the interface `BusSolution` abstracts the real implementation. Each algorithm that implements the interface `SearchAlgorithm` use objects of type `BusSolution`. Namely, the class `SearchAgentAlgorithm`, uses the operations provided while performing the search and building a solution.

The pheromone model of our problem is implemented by the `ACOPheromone` class. Figure 4.3 presents the partial class diagram of the pheromone model classes. This class implements the interface `IACOPheromone` which defines a set of operations to manipulate the pheromone matrix. The synchronous parallel and sequential ACO, AS and $\mathcal{MM}$AS algorithms, use the implementation provided by class `ACOPheromone`. Our developed concurrency models for parallel asynchronous ACO optimization are implemented by extending the class `ACOPheromone` and class `LockFreeACOAsyncUpdate`, for the Asynchronous Blocking Matrix and Asynchronous Blocking column algorithms, and by overriding the implemented operations.

4.3 Base Library Problem Components Implementation Details

Apart from the algorithms and metaheuristics, we developed a set of components that are specific to the PTBA problem. This components belong to the core layer of the library and

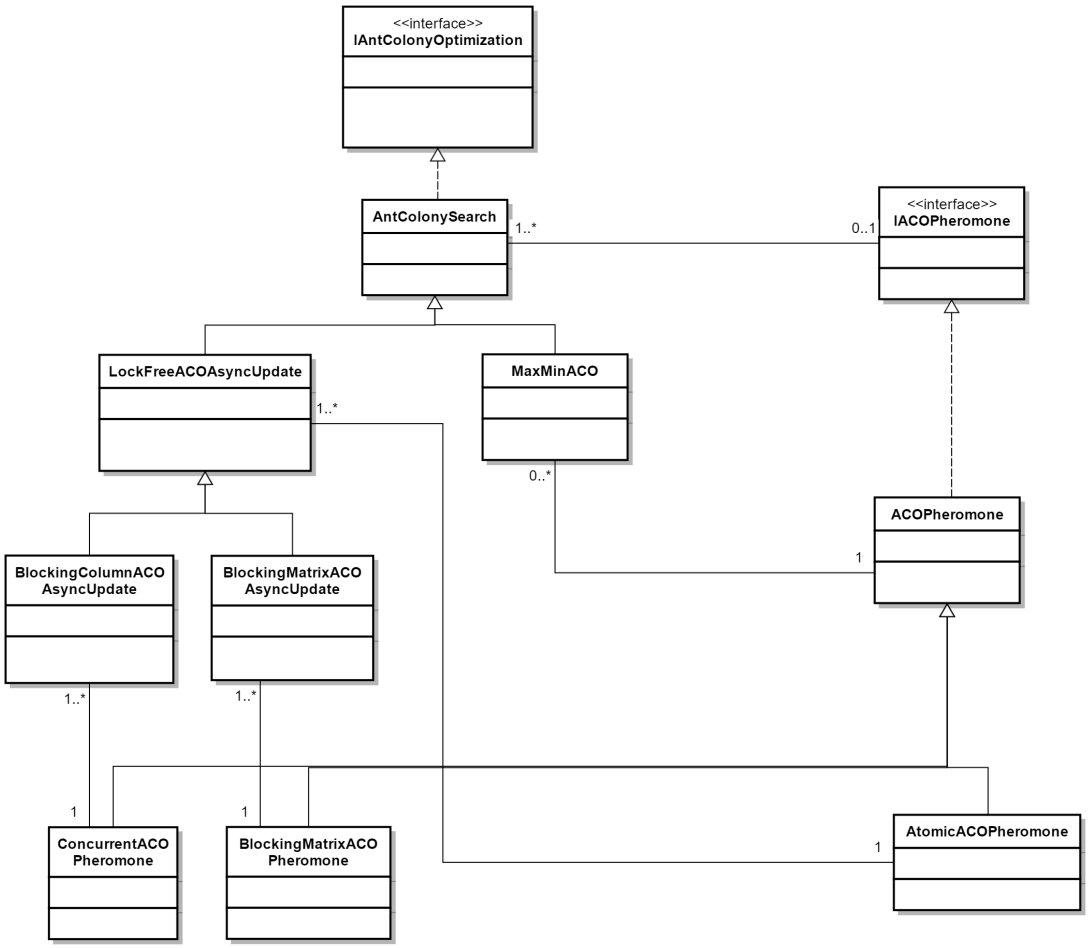


Figure 4.3: Partial Class Diagram of Pheromone models classes.

are used by the agent search algorithm, which is responsible for constructing solutions.

4.3.1 Objective Functions

The objective function, defined by equation 1.2 on section 1.3.1 is implemented using CaSPER LS features, namely Expressions and Objective Functions.

In CaSPER, it is possible to define a custom objective function by implementing a class that extends the class `Function`. The implementation of the objective function of our problem can be seen in listing 4.3.

The first part of the objective function of equation 1.2 is implemented by a CaSPER LS `Function` on the `HeterogeneousFleetBusSystemObjectiveFunction`. This class has a method (`updateSumFormula`) to update the current value which receives a CaSPER LS expression (class `Expr`) and a weight. This method is called each time a vehicle travels, with the expression consisting of the total kilometres travelled and weight consisting of the cost per kilometre of the type of vehicle that travelled.

Class `GlobalBusSystemObjectiveFunction` implements the second part of the

equation and uses the implementation of the first part described just above. A method for updating the function (`updateNumVehicles`) is available. This method receives a list of pairs $\langle nV, dCostV \rangle$ where nV denotes the number of vehicles of a given type and $dCostV$ the daily depreciation of vehicles from that type.

Listing 4.3: Objectives Function Definition.

```
class HeterogeneousFleetBusSystemObjectiveFunction : Function<double>{
  void updateSumFormula(Expr<double> e, double weight){...}
  double getValue() {...}
};

class GlobalBusSystemObjectiveFunction : Function<double> {
  GlobalBusSystemObjectiveFunction( Function<double>& fleetObjFn){...}

  void updateNumVehicles(list<pair< int,int>>& ws){...}
  double getValue() {
    double sum = 0;
    sum += busFleet.getValue();
    for(auto & w: weights){
      sum += w.first*w.second;
    }
    return sum;
  }
  vector<Expr<double>> exprs;
  Function<double>& busFleet;
  list<pair< int,int>> weights;
};
```

By implementing the total distance travelled by all vehicles and the vehicles used cost factors with two different classes we allow programmers to implement different objective functions while reusing one of our implementations.

4.3.2 Heuristic Functions

We implemented all the heuristic functions presented on section 3.1.2. It is desirable that despite of the fact that different heuristics may be used, they all consist in a strategy to select a vehicle from a set of vehicles with different properties. Therefore, we provide an abstraction for algorithms such that they can use transparently a given heuristic. All the heuristic implementations must implement the interface `BusSelectingHeuristic` which can be seen in listing 4.4. The method `selectBus` receives the following information:

- Information about the current problem instance. Argument `bs`;
- set of tuples $\langle onDepot, indexOnAdmissibleSet,$ $arrivalObject \rangle$ where $onDepot$ is a boolean variable used to indicate if the current vehicle is currently on a depot (true) or on a terminal (false), $indexOnAdmissibleSet$

denotes the index on the set $AdmVD_i$ or $AdmVP_i$ if the vehicle is on a depot or on a terminal, respectively, and *arrivalObject* is an object from class `BusArrival` which stores information about the arrival like the minute of arrival, the location, among others. Argument `hypothesis` which corresponds to the set $AdmVD_i$.;

- the identifier of the terminal of the departure being processed.
  Argument `terminal_index`;

- information about the departure being processed like starting minute, terminal endpoints, bus line identifier, among others. This information is stored on objects of class `Departure`. Argument `d`;

- the current partial solution. Argument `solution`.

Listing 4.4: `BusSelectingHeuristic` interface. Used to abstract different heuristics from the algorithms.

```
1 class BusSelectingHeuristic {
2     virtual tuple<bool,int, BusArrival>
3       selectBus(BusSystem* bs, vector<tuple<bool,int, BusArrival>>& hypothesis,
4         int terminal_index, Departure& d,BusSolution * solution) = 0;
5 };
```

It is also worth noting that for the probabilistic heuristic based on vehicle utility (section 3.1.2.2), we implemented the utility function on the class `SelectProbabilistic` which in turn implements the `BusSelectingHeuristic` interface. The ACO probability distribution, which is based on the pheromone model and on an heuristic function (the vehicle utility), extends this class, and the vehicle utility function implementation is reused. However, new heuristics may be developed with a different vehicle utility specification by overriding the method where it is implemented.

4.3.3 Constraint Manager

The constraint manager is implemented using the CaSPER LS features. We use the Constraint System concept, provided by the CaSPER LS library, to achieve a modular implementation of the problem constraints. It is possible to build constraint systems which are defined by other sub constraint systems. Using this feature, we developed a global constraint system which is defined by two additional constraint systems: one for vehicle associated constraints and one for route associated constraints.

This scheme is illustrated in figure 4.4. For each route, a constraint system is defined. Despite the fact that we define a constraint system for routes, our search agent algorithm does not enforce any constraint on it. This is due to the fact that route constraints must never be violated (hard-constraints) and, assignments that violate hard constraints are not taken into account. However, in order to allow programmers that use our library to define

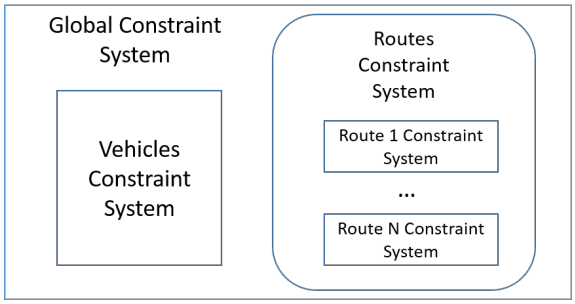


Figure 4.4: Constraint Systems definition. A global constraint system is defined based on sub constraint systems.

constraints over the routes, or in a situation where hard constraints may be violated, we introduced this constraint system.

An example of constraint system creation, that creates the constraint system illustrated on figure 4.4 using CaSPER LS, is shown on listing 4.5.

Listing 4.5: Example of Constraint Systems definition.

```
1  Solver solver;
2  ConstraintSystem<int> globalCS(solver);
3  ConstraintSystem<int> busesConstraintSystem(solver);
4
5  for(unsigned int i = 0 ; i < numberRoutes; i++){
6    ConstraintSystem<int> routeCS(solver);
7    globalCS.post(routeCS); //Add route i Constraint System
8  }
9
10 //Add vehicles Constraint System
11 globalCS.post(busesConstraintSystem);
```

The vehicles constraint system will contain all the maintenance and maximum daily working time constraints for all the vehicles used. This constraints are enforced as is shown in listing 4.6 for each vehicle. The function `postMaintenanceConstraint` receives as parameter the vehicle identifier (`bus_id`), a CaSPER LS variable which stores the working time minute of the last maintenance (`v`), and the maximum time between maintenance tasks value (`limit`). The function `getBusWorkingTimeExpression(id)` returns an object of type `Expr` which denotes an expression whose value is the total working time of vehicle with identifier `id`.

Listing 4.6: Vehicle Maintenance and Maximum daily working time creation.

```
1  void ConstraintManager::postMaintenanceConstraint(int bus_id, Var<int> v,
2                                                    int limit){
3    Expr<int> busExpr = this->getBusWorkingTimeExpression(bus_id);
4    Constraint<int> maintenanceConstraint = busExpr - v <= limit;
5    busesConstraintSystem.post(c);
6  }
7
```

```
8  void ConstraintManager::makeBusExpressions(int busid, int maxWorkingTime, ...){
9    ...
10   Constraint<int> maxWorkingTimeConstraint = busExpr <= maxWorkingTime;
11   busesConstraintSystem.post(c);
12   ...
13 }
```

The maximum daily working time constraint is enforced in the function `makeBusExpressions`. This function also receives as parameter the vehicle identifier (`bus_id`) and the maximum daily working time of the vehicle (`maxWorkingTime`).

Through the operator overloading mechanism of the C++ language, CaSPER LS creates a tree of expressions from the operators used in the expression `busExpr - v <= limit`, for the maintenance constraints case. Since the operator with less priority is a relational operator, an object of type `Constraint`, which denotes a constraint, is created. The same procedure is applied for the maximum daily working time constraints.

4.4 Parallel Algorithms Implementation

Parallel strategies [18, 60] can target either *shared-memory* (SM) or *distributed* architectures, either using CPUs, GPUs or both. Distributed architectures offer better scalability than SM ones, however, communication within nodes is more expensive due to the necessity of transferring data between disjoint address spaces. ACO agents rely on a high-level of communication in order to report their findings and improve the learned information about the search space. More concretely, the following steps imply communication:

ACO parameters and problem data - parameters like α, β, ρ, among others, and data about the problem instance, must be sent to the agent so that a solution can be constructed;

Reporting a constructed solution - Each solution constructed must be entirely sent back to the colony such that it may be used to perform updates. Additionally, in order to maximize the parallel agents usage, the objective function of the constructed solution may be computed on by the agent, therefore, the objective function value must also be sent;

Accessing the pheromone matrix - At each departure assignment, each agent in the worst case needs to read an entire column of the pheromone matrix. On the other hand, the master will be concurrently modifying entries of the matrix and the modifications must be sent to the agents in order to ensure that all the agents and the master have the same version of the matrix.

Hence, in a distributed architecture, the overhead introduced on the algorithms due to communications will be significant. Based on this observation, we target a SM architecture

in which communications can be performed by sharing data structures among threads, and the only overhead introduced is the one imposed by concurrency control mechanisms.

As previously stated, we use generic programming features on our algorithms which is a programming paradigm for developing efficient and reusable software libraries. The base idea of this paradigm is to find commonality among similar implementations of an algorithm and abstract possible differences across concrete implementations. This approach provides reusability, since written algorithms can be used with different concrete types, and is efficient because type instantiation is performed (in the case of C++ language) at compile time. Thus, concrete algorithm implementations will be as efficient as they would be if code was duplicated and the generic programming paradigm was not used. In C++, generic programming is performed using *templates*. Classes are implemented based on *templates* instead of concrete types and, when an object of the class is created, the programmer specifies a concrete type for each *template* of the class. Consequently, the class is instantiated with the specified concrete types, at *compile time*. In the context of the C++ language this approach is called *meta-programming*.

In the following sections we will discuss how we use generic programming features and we will describe the implementation of our proposed parallel algorithms.

4.4.1 Parallel Restarts

The parallel restarts metaheuristic, described in section 3.2, is implemented by the class `RestartsSearch`. This class, extends the class `AbstractSearchAlgorithm` which in turn implements the `SearchAlgorithm` interface.

To maximize reusability, the `RestartsSearch` is parametrized by four templates. Listing 4.7 shows the class templates definition and an example of object creation. The first template `AgentConcreteType` is instantiated with a concrete agent type, the template `Heuristic` is instantiated with the heuristic that should be used by the agent when choosing a vehicle to assign to a given departure, the template `ConcreteBusSolutionType` is instantiated with a concrete solution representation type and finally the template `AgentManagerType` is instantiated with a concrete ant manager type. An agent manager is a class which is used to keep track of a set of agents and store relevant information like the best solution achieved. In section 4.4.4 we will discuss in more detail the purpose of this class.

Listing 4.7: Template parametrization of Restarts Metaheuristic and object creation.

```
1 //Class definition
2 template<class AgentConcreteType,class Heuristic,class ConcreteBusSolutionType,
3   template<class, class> class AgentManagerType >
4 class RestartsSearch: public AbstractSearchAlgorithm {
5
6   static_assert(std::is_base_of<IAntAgent, AgentConcreteType>::value,
7       "AgentConcreteType must derive from IAntColonyOptimization");
8   static_assert(std::is_base_of<BusSelectingHeuristic, Heuristic>::value,
```

```
9           "Heuristic must derive from BusSelectingHeuristic");
10      static_assert(std::is_base_of<BusSolution, ConcreteBusSolutionType>::value,
11          "ConcreteBusSolutionType must derive from BusSolution");
12  ...
13  }
14
15  //Object creation example
16  RestartsSearch<VehicleRestartsAgentAllocAware,BusSelectingHeuristic,
17    NoAllocsBusSolution,PTBAAntManager> * restarts =
18      new RestartsSearch<VehicleRestartsAgentAllocAware,BusSelectingHeuristic,
19        NoAllocsBusSolution,PTBAAgentManager>(...);
```

In the object creation example, we instantiate the four templates with classes that we already developed and that belong to the library, however, programmers may instantiate the templates with other classes that also make part of our library or even with custom classes, as long as the implementation obeys a set of rules. For example, if a programmer wants to develop a custom heuristic class and instantiate an algorithm with that class, the custom class must implement the `BusSelectingHeuristic` interface. We added a static assertion (checked at compile time) on the type of each custom class that is used to instantiate a template using the C++ *static_assert* and the *is_base_of* functions. For example, for the `heuristic` template, we verify that the concrete type provided is a base of the `BusSelectingHeuristic` type.

This metaheuristic consists in performing N executions in parallel yielding N solutions. We implemented this metaheuristic by using OpenMP and adding a *pragma* on the loop in which the N executions are performed. This implementation is shown on listing 4.8.

Given that N iterations will be performed (each iteration corresponds to one agent execution), we added a *parallel for pragma* available on OpenMP which automatically paralellizes the loop. Since we do not specify any type of task scheduling policy, OpenMP uses the *static* scheduling by default which consists in dividing the loop into equal-sized chunks. When N is not divisible by the number of threads multiplied by the chunk size, OpenMP attempts to produce almost equal chunks.

Listing 4.8: Parallelization of the Restarts Metaheuristic

```
1  double tmpVal;
2  #pragma omp parallel for num_threads(numThreads) if (parallelism)
3  for(int its = 0 ; its < N ; ++its){
4    ...
5    BusSolution* newSol = agent->search();
6    tmpVal = newSol->getGlobalObjectiveFunction();
7    ...
8  #pragma omp critical
9    {
10     if(tmpVal < globalBestSolVal){
11       ...
12       globalBestSolution =  newSol;
13             globalBestSolutionValue = tmpVal;
```

```
14              ...
15          }
16          ...
17      }
18  }
```

The option *num_threads* is used to dynamically specify the number of threads to be launched in each execution of the metaheuristic and the *if* statement is used to enable or disable parallelism, i.e., the variable *parallelism* is a boolean flag which is `true` when the user wants to use parallelism or `false` otherwise.

During executions, we keep track of the best solution found so far (global best). Each agent computes the objective function of the solution produced and executes a given critical section. A critical section is implemented with OpenMP by creating a block of code (which defines a new scope) and by adding a *critical pragma*. Since concurrency occurs when checking if the a new solution is better than the currently global best, and also when performing the update of the global best solution when it is the case, it is necessary to define this block of code as a critical section.

4.4.2 Parallel Synchronous ACO Algorithms

The parallel synchronous AS and $\mathcal{MM}$AS algorithms implementations also use OpenMP. The AS algorithm is implemented on class `AntColonySearch` and the $\mathcal{MM}$AS on class `MaxMinACO`. Both classes have the same templates as the restart metaheuristics class.

Since the $\mathcal{MM}$AS is an extension of the AS algorithm, both implementations are very similar. Therefore, class `MaxMinACO` extends the `AntColonySearch` and overrides only some methods to incorporate the $\mathcal{MM}$AS additional mechanisms.

In both algorithms, at each iteration a set of N agents are executed in parallel. Therefore, we added a *parallel for pragma* on a loop in which at each iteration an agent execute and constructs a solution. The added *pragma* is the same as in the parallel restarts metaheuristic. The parallel synchronous region code is shown on listing 4.9, where the total number of iterations is assumed to be *numIterations*.

Listing 4.9: Parallelization of the AS and $\mathcal{MM}$AS algorithms

```
1 for(int its = 0 ; its < numIterations ; ++its){
2 #pragma omp parallel for num_threads(numThreads) if (parallelism)
3   for(int ant_idx = 0 ; ant_idx < numAnts ; ++ant_idx){
4     //Perform search and build a solution
5     //Compute the objective function of the constructed solution
6     //Update the global best solution found so far (critical section)
7   }
8 ...
```

In both algorithms we also keep track of the best solution found so far. The procedure of updating the global best solution when a better solution is found is the same as in the parallel restarts metaheuristic, implementing it as a critical section.

In section 3.4.1.2 we described a mechanism to find the best solution of an iteration for the $\mathcal{MM}$AS algorithm with $O(1)$ time complexity, based on the fact that each agent computes the objective function value of its constructed solution, and compares the obtained value with the current iteration best value, performing an update when it is the case. Therefore, for the $\mathcal{MM}$AS algorithm, we implemented the three operations of the mechanism just described (read, comparison, update) as a critical section. In fact, we did not created an additional critical section but we replaced the critical section in which the global best solution was updated. At the end of a parallel region, we know the best solution found on the current iteration and it only takes one comparison and one write to update the global best solution, instead of N as in the AS algorithm.

4.4.3 Parallel Asynchronous ACO algorithm

The base parallel asynchronous ACO algorithm introduced in section 3.4.3 is implemented on class `LockFreeACOAsyncUpdate`. This class is also parametrized by the same templates as in parallel restart metaheuristic and parallel synchronous ACO algorithm classes.

We modified the AS algorithm and we incorporated additional mechanisms and data structures to achieve an asynchronous behaviour. The base asynchronous algorithm design is based on the thread pool, avoiding consecutive thread creation and destruction overhead, in which the results are placed on a pool of solutions, implemented using condition variables and one *mutex*. The overall scheme corresponds to a 1-producer/N-consumers pattern and is illustrated on figure 4.5.

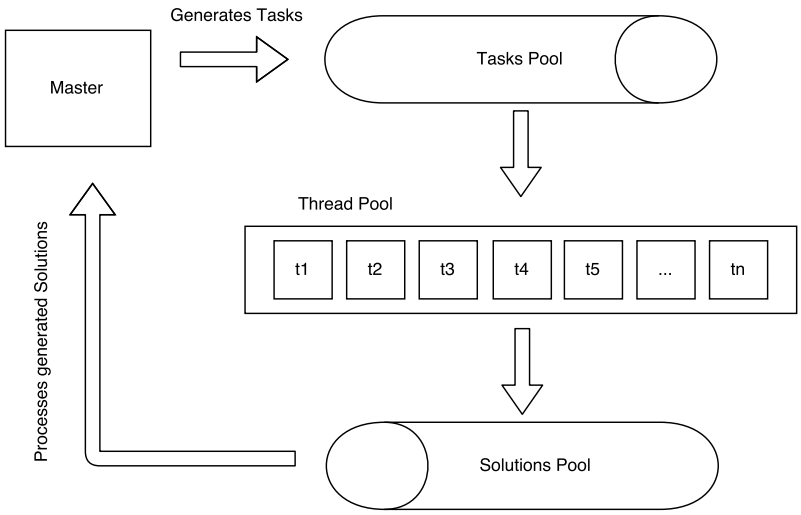


Figure 4.5: Parallel Asynchronous ACO algorithm architecture.

Tasks are placed on a pool of tasks implemented with a lock-free queue, from which threads in the thread pool will consume. When a thread finishes processing a task, i.e., builds a solution, the solution is placed on a solutions pool and the master consumes it by performing the pheromone update and evaporation steps when it is the case. The solution pool is implemented using one *mutex* and C++ conditions variables.

Each task is implemented through a *lambda* expression (an anonymous function). When a thread receives a task, the expression is executed. The solutions pool has objects of type `AntSolutionInfo` which store the solution produced, the objective function value of the correspondent solution, the identifier of the agent and the virtual iteration in which it was produced.

The proposed concurrency models are implemented as follows:

Blocking Matrix - Each time the master or one of the threads wants to access the pheromone matrix they have to acquire a lock. This lock is implemented as a *Spin lock*. With this type of lock threads wait in a loop, performing a check at each iteration that verifies if the lock is available and, since threads remain active in a loop, the chances of a thread being re-scheduled and performing a context-switch which is time-consuming, are reduced. The *Spin lock* implementation is shown on listing 4.10. This model is implemented on class `BlockingMatrixACOPheromone`;

Blocking Column - In this model we have an array of *reader-writer* locks with $|T|$ elements, where each element is the lock of a column of the pheromone matrix. The *reader-writer* locks are implemented on C++, on the type *std::shared_lock*. This model is implemented on class `BlockingColumnACOAsyncPheromone`;

Lock-Free - The lock-free model is implemented by using C++ *atomics*, ensuring that read and write operations are atomic. This model is implemented on class `AtomicACOPheromone`.

Listing 4.10: Spin lock Implementation

```
1  class SpinLock {
2      std::atomic_flag locked = ATOMIC_FLAG_INIT ;
3  public:
4      void lock() {
5          while (locked.test_and_set(std::memory_order_acquire)) { ; }
6      }
7      void unlock() {
8          locked.clear(std::memory_order_release);
9      }
10 };
```

The classes `BlockingColumnACOAsyncUpdate` and `BlockingMatrixACOAsyncUpdate` extend the class `LockFreeACOAsyncUpdate`. Hence, the base algorithm implementation is exactly the same. The only difference is that the respective model, i.e., `BlockingColumnACOAsyncPheromone` for class `BlockingColumnACOAsyncUpdate` and `BlockingMatrixACOPheromone` for class `BlockingMatrixACOAsyncUpdate`, is passed by argument to the constructor of class `LockFreeACOAsyncUpdate`. The constructor receives a pointer to an object of the interface `IACOPheromone` and the concrete implementation is abstracted.

Listing 4.11: Asynchronous Parallel ACO Master Implementation.

```
for(int processed = 0; processed < totalExecutions;++processed){
  //block until the solution pool is not empty.
  //Get the produced solution (object from class AntSolutionInfo) from the pool
  //Perform partial update
  //Perform evaporation if the condition is verified
  //Update global best solution
  ...
}
```

Listing 4.11 shows the procedure executed by the master, assuming that the number of tasks created is *totalExecutions*. At each step, the master attempts to get an object from the solutions pool and blocks while the pool is empty. When an agent puts a solution object on the solutions pool the master processes it by performing a partial update, by performing evaporation if half the threads of the iteration of the solution being processed already finished, and by updating the global best solution if a better solution is found. This procedure is repeated until *totalExecutions* tasks are processed.

Listing 4.12: Asynchronous Parallel ACO Slave *lambda* expression mplementation.

```
[this,ant_id,virtual_iteration] {
  //Create dynamically a new solution object (BusSolution)
  //Perform search and build a solution
  //Compute the objective function of the constructed solution
  //Create an AntSolutionInfo object and push it into the solution pool
}
```

Each agent executes a given *lambda* expression which consists in the operations shown on listing 4.12. The values inside the square brackets are the variables that can be accessed on the scope of the *lambda* expression. The first operation consists in dynamically creating a new object of type `BusSolution` which is the object that the agent will complete during search. Then, the search is performed and a solution is built. The task of the agent terminates by computing the objective function value of the produced solution and by adding a new solution object (`AntSolutionInfo`) to the solutions pool.

4.4.4 Parallel Implementations NUMA Details

Our parallel algorithms attempt to be NUMA-Aware as possible, therefore, we followed the guidelines presented on section 2.4.2.1. When memory is allocated, it will be placed on the NUMA node in which the allocation was performed.

This is the reason why each agent creates and allocates memory for each solution object. Additionally, as the solution is constructed, memory allocations from created objects are performed by the agent. This ensures that the created objects data will be close to the NUMA node of the agent (probably on the same node), thus reducing the overhead of accessing those memory areas.

We do not control in which NUMA node each thread will be executed and all the agents, will execute in arbitrary threads and will access the pheromone matrix. The pheromone matrix will be created by the master thread and its data will reside on memory from the master NUMA node. Consequently, some threads which execute in different NUMA nodes will get a slight performance hit.

One possible solution would be to replicate the pheromone matrix in each thread. Even despising the memory waste due to redundancy, what would severely degrade the performance would be the necessity of assuring that all the threads have the same matrix with the same values, which rules out this approach.

Another possibility is to use the interleaved memory policy supported by NUMA architectures. With this approach, when allocating a chunk of memory, the chunk will be interleaved across all the NUMA nodes. With this technique all accesses should be spread out evenly over all nodes. We believe that interleaving memory of the pheromone matrix will not lead to significant performance gains since there will be always accesses to memory of non-local nodes.

CHAPTER 5

EVALUATION

In this chapter we analyse and evaluate experimentally our model of the problem and the proposed algorithms, in order to assess their effectiveness. The performance of our proposed parallel algorithms, namely the ACO parallel algorithms, will also be evaluated.

The chapter starts by describing our experimental setup in section 5.1 where the machines used to perform the tests are described as the instance of the problem taken into account.

In section 5.2 we analyse, assuming a sequential setup, the algorithms and the problem model proposed. More concretely, we start by analysing the solutions produced by our algorithms and then we evaluate the restarts algorithm. Experiments performed in order to evaluate how each parameter of the model affects the solutions produced will be presented. Our ACO AS and $\mathcal{MM}$AS algorithms will also be analysed in this section.

Section 5.3 evaluates the Parallel Synchronous and Asynchronous ACO algorithms in terms of performance and solution quality.

Finally, in section 5.4 we provide a summary of the conclusions drawn from the experiments performed.

5.1 Experimental Setup

Our experiments were performed on two different machines with different architectures:

Personal-Computer (PC) - Intel(R) Quad-Core (4 cores) i7-4700MQ CPU @ 2.40GHz supporting Hyperthreading(R) and 32GB RAM;

MultiProcessor (MP) - 4-node NUMA machine with 4 AMD(R) Opteron 6272 processors @ 2.1 GHz, each with 16 cores, and with 64 GB RAM (16 GB for each NUMA node).

For experiments that do not require high computational power we use the PC machine. However, for experiments in which we evaluate the scalability of our solutions as the search depth increases and as the number of threads increases we use the MP machine. This last experiments require a higher number of cores.

For the sake of clarity, we introduce the following abbreviations of the algorithms that will be analysed:

- Parallel Synchronous AS ACO algorithm (ASSync)
- Parallel Asynchronous Blocking Matrix ACO algorithm (AsyncBM)
- Parallel Asynchronous Blocking Column ACO algorithm (AsyncBC)
- Parallel Asynchronous Lock-Free ACO algorithm (AsyncLF)

5.1.1 Problem Instance

The instance of the public transport bus assignment problem addressed corresponds to a subset of 6 CARRIS[1] bus lines with ascending and descending directions, with a total of 732 departures and 12 terminal locations. Additionally, 3 vehicle categories and 2 depots are considered. We decided to use CARRIS routes since the schedule of each route corresponds to a real life schedule.

Table 5.1 lists the capacities of both depots considered for each of the three types of vehicles taken into account.

Table 5.1: Capacities r_k of each depot D_k for each type of vehicle.

	Type 1	Type 2	Type 3
D_1	20	10	20
D_2			

Each vehicle type has different characteristics, therefore, different values for each constraint exist. Additionally, each type has different costs. These parameters can be seen in table 5.2.

The global and terminal specific parameters values used in our experiments can be seen in table 5.3. It is worth noting that both terminal specific parameters ($maxDurAtTerminal_i$ and $maxNumVehiclesAtTerminal_i$) have the same value for all the 12 terminals.

Table 5.4 lists all the 6 routes with ascending and descending directions and its associated number of departures and duration.

Since we have 2 depots and 12 terminals, we have a total of 14 locations. We built two 14×14 matrices: one for distances ($distance(loc_i, loc_j)$) and one for the travelling times

[1] A portuguese bus service transports company.

Table 5.2: Characteristics of each vehicle type. Max time between maintenance tasks is denote by *MaxTimeBMaint* and maintenance durarion is denoted by *MaintDur*.

	Cost km (€)	Daily Cost (€)	MaxWorkingTime (minutes)	MaxTimeBMaint (minutes)	MaintDur (minutes)
Type 1	3	30	960	720	30
Type 2	5	40	1080		
Type 3	10	50			

Table 5.3: List of all the global parameters and its associated values.

Parameter	Value
$maxIdleTimeInService$	50
$maxNumInterrupts$	4
ϵ_{min}	3
$maxDurAtTerminal_i$	120
$maxNumVehiclesAtTerminal_i$	100
$NumVDCandidates$	1

($duration(loc_i, loc_j)$), between locations. Based on the GPS coordinates of each location, we obtained the distance (km) between each pair of locations of the shortest valid path (e.g. respecting traffic signs). The durations were obtained using the maximum allowed speed at each section of the path. It is worth noting that we set the $maxNumVehiclesAtTerminal_i$ to 100, which given the fact that the total number of vehicles is 100, it means that the maximum number of vehicles at a terminal is not constrained. Therefore, we assume a situation in which all the vehicles, based on the idle situations in a solution produced, can be idle at a terminal.

Unless stated otherwise, when an agent is selecting a vehicle using the probabilistic heuristic, the 10 best options are considered. Additionally, the weights of the probabilistic heuristic, defined on equation 3.12 in section 3.1.2.2, are defined as is shown on table 5.5.

Due to the stochastic nature of our algorithms and in order to achieve more robust results, 5 runs are performed for each configuration at each experiment.

Table 5.4: List of all the 12 routes taken into account and its associated characteristics. The descendant direction is denoted by DESC and the ascendant direction by ASC.

Name	Direction	#Departures	Duration (minutes)
702	DESC	72	22
702	ASC	71	22
732	DESC	44	48
732	ASC	45	48
748	DESC	70	48
748	ASC	69	48
746	DESC	66	36
746	ASC	65	36
746 (711)	DESC	55	48
746 (711)	ASC	56	44
744	DESC	59	58
744	ASC	60	57

Table 5.5: Default weight values for the probabilistic heuristic based on vehicle utility.

Weight	Value
k_1	10
k_2	0.5
k_3	0.5
k_4	1

5.2 Problem Model and Algorithms Analysis

This section focus on evaluating the effectiveness of our model, implemented by the agents algorithm described in section 3.1, as well as the solutions produced. We perform experiments using the restarts and ACO AS and $\mathcal{MM}$AS algorithms assuming a sequential setting.

Since this section does not focus on evaluating the parallel performance in terms of speedups and scalability of our proposed parallel algorithms, we perform the experiments on the PC machine.

5.2.1 Analysis of Solutions

A solution of the PTBA problem consists of a set of assignments of vehicles to departures. Our solver application essentially outputs the solution in two different formats: vehicle oriented and route oriented.

In the vehicle oriented solution the application outputs each *sheduling* S_i performed by a vehicle v_i where S_i contains all the tasks performed by the vehicle. On the other hand, the route oriented format outputs for each route, a set of pairs $\langle T_j, v_i \rangle$, where T_j denotes a departure and v_i the vehicle assigned to that departure.

Listing 5.1: Example of a schedule produced by our application for a given vehicle.

```
Bus id: 28
Category: 1
Assigned Depot -> Index: 0 Location: 12
Total time in service: 444 minutes
Total distance travelled: 74 km
Total Time Waiting in a terminal: 59 minutes
Schedule:
    <28,12,2,-,152,VoidIn>
    <28,2,3,152,200,InService>
    <28,3,3,203,207,Wait>
    <28,3,2,207,255,InService>
    <28,2,2,258,265,Wait>
    <28,2,3,265,313,InService>
    <28,3,12,316,326,VoidOut>
    <28,12,12,326,326,ServiceInterrupt>
    <28,12,0,755,766,VoidIn>
    <28,0,1,766,788,InService>
    <28,1,1,791,793,Wait>
    <28,1,0,793,815,InService>
    <28,0,0,818,826,Wait>
    <28,0,1,826,848,InService>
    <28,1,11,851,852,LineExchange>
    <28,11,11,852,858,Wait>
    <28,11,10,858,915,InService>
    <28,10,10,918,950,Wait>
    <28,10,11,950,1008,InService>
    <28,11,12,1011,1020,VoidOut>
```

An example of a produced schedule for a given vehicle can be seen in listing 5.1. We can check that vehicle with identifier 28 of type 1 worked 444 minutes, travelled a total of 74 km and the total amount of idle time was 59 minutes.

Regarding the tasks of the vehicle, each line consists in a tuple, as defined in section 1.3.1. We can see that the first task is a `VoidIn` task (trip performed without passengers from the depot to a terminal), then a set of tasks of type *InService* and *Wait* are performed. It should be noted that the *Wait* tasks does not start immediately after the previous task but only after ϵ_{min} minutes (3 minutes in this case), which denotes the

minute at which the vehicle is available to perform other services. The vehicle has a service interruption (task of type *ServiceInterrupt*) since it was idle on the depot for more than 50 minutes. We can also see that a line exchange (task of type *LineExchange*) was performed. At the end of the schedule (which is the last task of the day) the vehicle returns to the depot, which corresponds to a *VoidOut* task.

The instance of the problem in which we perform our tests has a total of 732 departures which is a reasonably large instance. Each produced solution consists in several schedules each with several tasks which in turn are presented as text. For this particular instance, each solution is presented on a file with 2500 lines of text. Hence, the task of analysing a complete solution in terms of *quality of service* is not easy.

We attempt to reduce this problem by generating a plot using *python* which shows what services each vehicle does over time. This plot is shown on figure 5.1. The X-axis denotes the time (minutes) and the Y-axis denotes, for each value, a given vehicle. It is noteworthy that the Y-axis scale has no meaning, we simply plot each vehicle services on a vertical line. As discussed in section 3.1.1 the minute 0 corresponds to 5:00 AM. This plot is subject to interpretation which may be subjective in some cases. For a more precise and formal analysis of a solution, one must look at the generated solution instead.

For each vertical line, we draw a rectangle for each departure T_j. Each rectangle starts at minute s_j and ends at minute e_j (the length of the rectangle is $e_j - s_j$). The places on a vertical line in which no rectangle exists correspond to tasks that are not services. To distinguish vehicles, we painted each vehicle line with a random color, from a set of colors.

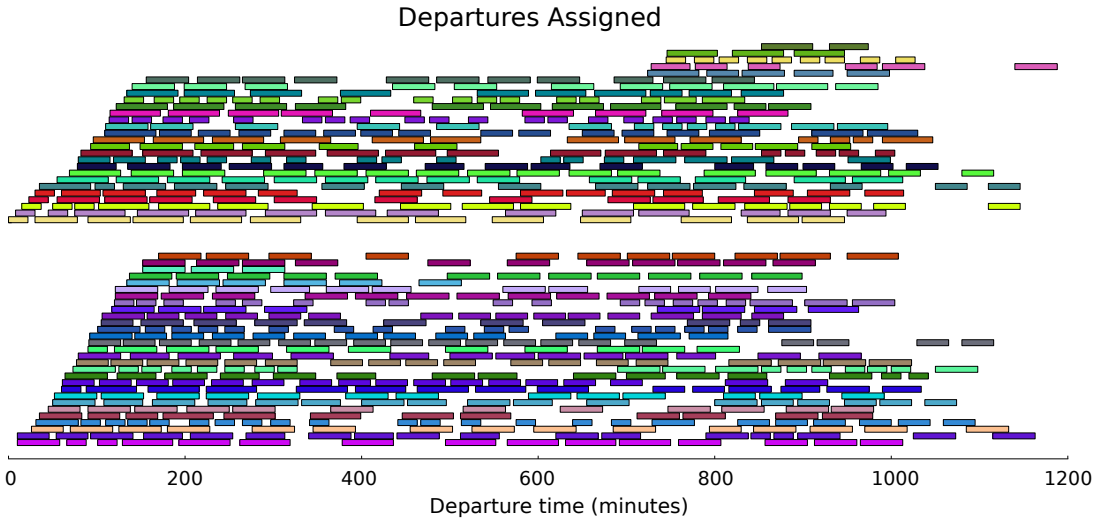


Figure 5.1: Vehicles service tasks over time. Each vertical line denotes a given vehicle. In the plot two clusters of vehicles, where each cluster corresponds to vehicles from one of the two depots, can be identified.

Two clusters of vehicles are identifiable where each cluster corresponds to vehicles

from a given depot: vehicles on the upper cluster belong to depot 2 and vehicles on the lower cluster belong to depot 1.

By interpreting the plot one first observation, based on the fact that the problem instance is from a city scene where on rush hours the number of departures is higher, specially in the morning, a great number of vehicles are picked. Outside rush hours (lunch time and mid afternoon) the number of vehicles needed is less and it happens that a considerable number of vehicles become idle, performing only some sporadic services. There are even some cases in which a vehicle performs services only in the morning and this is clearly a situation in which a vehicle work capability (in terms of hours of service) is not maximized.

We can also see that around minute 800 ($\approx$ 06:20 PM) the algorithm started picking new vehicles from depot 2. This can be seen in the top right corner of the plot, on the upper cluster. This situation occurs due to the fact that vehicles that started working early (e.g. in the morning) exceed their working time without performing maintenance (720 minutes) and need to return to the depot, thus forcing the algorithm to pick new vehicles. Furthermore, we can see that our algorithm performs reasonably well when distributing the services across the available vehicles. The majority of the vehicles picked, perform services across all the day.

It is also clear that the algorithm attempts to reuse vehicles already picked since on the solution that we are analysing only 56 vehicles of a total of 100 were used.

5.2.2 Restarts Algorithm

In section 3.2 we presented our proposed parallel probabilistic restarts metaheuristic. In this section we will use this metaheuristic in a sequential setting (only one execution at a time). The restarts algorithm consists of using the restarts metaheuristic so that at each execution an agent builds a solution using the agents algorithm described in section 3.1.1.

More concretely, we are interested in evaluating the effectiveness of each of the three heuristics proposed in section 3.1.2: Vehicle Utility selection - greedy (h_1), Random (h_2) and Probabilistic based on vehicle utility (h_3).

It is worth noting that the heuristic h_1 (greedy) is deterministic, therefore we do not use the Restarts algorithm for this heuristic. Instead, we only perform one execution. However, as a baseline, we show for each configuration the results achieved with heuristic h_1.

We executed the algorithm with different configurations, more concretely, with 100, 250, 500 and 1000 iterations, using each heuristic for each number of iterations. The results are shown on table 5.6.

We can see that the greedy algorithm (h_1) produces a solution which costs 26509 €. The random heuristic (h_2) presented the worse results by producing solutions with an average cost of 43380 € with 1000 iterations, which was its best configuration. From all the configurations, the best solution produced by h_2 had a cost of 42839 €. The mean of the costs of the solutions produced by h_2 is almost the same for all the configurations,

Table 5.6: Results in terms of solution quality and execution time of the sequential Restarts algorithm for the three heuristics proposed, while varying the search depth in the number of restarts (#RS). The values of the mean μ, standard deviation σ, minimum min and maximum max values are shown. We duplicate the results of the greedy algorithm to facilitate comparisons.

#RS	Heuristic	Objective Function (€)				Execution Time (s)			
		μ	σ	min	max	μ	σ	min	max
100	h_1	26509.00	0	26509.00	26509.00	0.5556	0.0114	0.5360	0.5650
	h_2	45398.60	883.31	44471.00	46764.00	4.3064	0.0358	4.2680	4.3600
	h_3	26269.00	347.41	**26048.00**	26876.00	4.3188	0.04989	4.2660	4.3850
250	h_1	26509.00	0	26509.00	26509.00	0.5556	0.0114	0.5360	0.5650
	h_2	43622.60	438.19	43183.00	44298.00	7.8806	0.0540	7.7850	7.9110
	h_3	25935.00	114.67	**25738.00**	26032.00	7.6924	0.08777	7.5380	7.7570
500	h_1	26509.00	0	26509.00	26509.00	0.5556	0.0114	0.5360	0.5650
	h_2	43596.8	501.84	42839.00	44022.00	13.5282	0.1790	13.3090	13.7280
	h_3	25827	206.29	**25511.00**	25980.00	13.3556	0.1289	13.2400	13.5470
1000	h_1	26509.00	0	26509.00	26509.00	0.5556	0.0114	0.5360	0.5650
	h_2	43380.40	299.66	42952.00	43791.00	25.0150	0.2887	24.5910	25.3550
	h_3	25685.60	181.20	**25444.00**	25871.00	24.5382	0.3198	24.9840	24.1650

except with 100 restarts in which the mean was slightly higher. Standard deviations were considerable high for h_2 in all the configurations. These mean and standard deviations value are somewhat expected since at each decision, agents select one vehicle of the set $AdmV_i$ randomly, i.e., the selection is not based on any type of information of the problem. This heuristic is very poor and it is expected that bad solutions will be constructed, however, this heuristic gives us an upper bound of the solutions quality which can be used to compare our proposed heuristics and algorithms.

A more interesting heuristic is the probabilistic heuristic h_3. We can see that the best solution values, for each configuration, were obtained with h_3. One observation that we can make is that the average quality of the solutions produced decreases as the number of restarts increases. Additionally, the cost of the best solutions obtained for each configuration also decreases as the number of restarts increases. This is based on the fact that to all the possible decisions, a probability value different than 0 is assigned, hence, if we performed infinite executions, it is certain that all the possible solution components would be used. By increasing the number of restarts we also increase the chances of selecting different solution components at each decision, which directly translates in a better exploration of the search space. The standard deviations obtained with h_3 for all the

configurations are also lower than the ones obtained with the h_2 which is an indication that h_3 is more consistent in terms of the quality of the solutions produced.

The heuristic h_3 is superior to h_1 since despite the fact that both use information about the problem, it is not as greedy as h_1. In the heuristic h_1, what the heuristic is doing is minimizing locally the cost of the objective function, at each decision. However, due to its deterministic behaviour, in which the vehicle selected for each departure is always the same, it gets stuck at a sub-optimal solution with objective function value 26509 €.

Both h_1 and h_3 provide, to domain experts, a mean of controlling the type of solutions produced in terms of *quality of service*. Namely, preferences like using more vehicles or less vehicles (even if that implies some extra cost) and allow or disallow line exchanges can be specified through the weights in the definition of a vehicle utility. However, since h_3 produces better results we conclude that it is superior, when comparing to h_1 and also h_2.

In terms of execution time, we can see that h_1 takes ≈ 0.556 s to produce a solution. It should be noted that h_1 executes faster than the other heuristics due to the fact that only one solution is constructed. On the other hand the number of solutions constructed in heuristics h_2 and h_3 is equal to the number of restarts.

As the number of restarts increases, the execution times increase. This is expected since the number of solutions constructed is equal to the number of restarts performed, which means that more computations are performed. Both heuristics h_2 and h_3 execution times are similar in all the configurations. This is an indication that the normalization and truncation steps performed on heuristic h_3 do not introduce a significant overhead on the overall execution time of the algorithm.

In general, since the amount of work (computations) performed for each configuration is practically the same, the execution times do not oscillate much. This is confirmed by the fact that the standard deviations of the execution time for all the configurations are very close to 0.

5.2.3 Model Parameters Influence Analysis

On our variant of the vehicle scheduling problem we take into account line exchanges. Theoretically this would allow us to spare vehicles and maximize the use of the already picked ones. Additionally our proposed definition of a vehicle utility takes into account parameters which in principle allow one to define how important are line exchanges and how important is the minimization of the number of vehicles (even if it implies additional costs).

In the following two sections we evaluate both aspects in order to confirm if the desired behaviour is indeed verified on the solutions produced.

5.2.3.1 Line Exchanges

In order to analyse the influence of line exchanges in the solutions produced we performed an experiment in which the restarts algorithm is used with the probabilistic heuristic. The

number of total restarts was set to 1000.

We perform the experiment with two configurations: one in which line exchanges are allowed and one in which line exchanges are forbidden. The results of these experiments can be seen in table 5.7. In this table we can see the average quality, the average number of vehicles and the average distance performed in void by all the vehicles on the solutions produced in both configurations.

Table 5.7: Analysis of the influence of line exchanges in the solutions produced. The results of the average objective function value of the solutions produced, the average number of vehicles used in each of those solutions and the average total distance in void are shown.

Line Exchanges	Objective Function (€)	#Vehicles	Distance in Void (km)
With	25606.60	58.40	3921.00
Without	26087.80	63.00	3480.60

A first observation is that by allowing line exchanges, better solutions are achieved. As we expected the average number of vehicles when line exchanges are allowed is smaller (58.4) than when no line exchanges are considered (63). However, with line exchanges, the average distance performed in void is greater (3921) than without line exchanges (3480.6).

This results confirm our intuition about the advantages of considering line exchanges. It is expected that with line exchanges vehicles perform greater distances in void due to the fact that they must travel from terminal to terminal, while when no line exchanges exist this does not happen. However, since the number of vehicles used is smaller with line exchanges, the final cost is also smaller. The decrease on the final cost is not only due to the daily cost of a vehicle but also due to the fact that when more vehicles are used, the chances of using more expensive vehicles (in terms of cost/km) increase.

On this experiment the solutions used less vehicles with line exchanges since when building the set $AdmV_i$ for a given departure, the algorithm does not only take into account vehicles from depots but also from terminals. By selecting vehicles from terminals it is likely that their use will be maximized since the vehicle will perform a service instead of being idle.

5.2.3.2 Vehicle Utility Parametrization

As stated before, the definition of a vehicle utility takes into account weights that can be used to control the influence of each term. Since the objective when solving this problem is to minimize the objective function (the operational costs) we believe that the term which denotes the incremental cost to the objective function is essential and must be set such that it is more important than the remaining terms. Therefore, we fix the weight of this term k_1 to 10, as presented in section 5.1.1. The experiments that we will perform on this

section will be based on experimenting different values for the weights k_2 and k_3 in order to evaluate how each of them influences the solutions produced.

These weights can be used in principle to penalize line exchanges (k_2) and to obtain solutions in which less vehicles are used (k_3). These are two aspects which transport service companies may want to control.

Table 5.8: Influence of the parameters k_2 and k_3 on the solutions produced. We fixed the number of restarts to 1000. The values of the mean μ, the standard deviation σ and minimum *min* of the average values of the objective function of the solutions produced are shown. The average number of vehicles (column #Vehicles) and the number of line exchanges (column #LineExchanges) of the solutions produced are also shown.

k_2	k_3	Objective Function (€)			#Vehicles	#LineExchanges
		μ	σ	*min*		
0	0	25671.20	97.34	25563.00	57.60	316.00
	0.5	—	—	—	—	—
	1	25127.60	106.59	25007.00	55.20	311.80
0.5	0	—	—	—	—	—
	0.5	25611.60	203.79	25415.00	57.20	308.80
	1	25260.40	117.17	25082.00	58.40	316.80
1	0	25344.80	151.08	25101.00	58.40	222.60
	0.5	25525.60	144.90	25334.00	58.20	232.60
	1	—	—	—	—	—

As in the experiment described in the previous section, we executed the restarts algorithm using 1000 restarts. From equation 3.12 we know that in the configurations in which one of the weights is 0, we only need to test one value. The part of the utility expression of the line exchanges and new vehicles terms can be simplified as follows:

$$\frac{t_2 \times k_2 + t_3 \times k_3}{k_2 + k_3} \tag{5.1}$$

where t_2 denotes the line exchanges term and t_3 denotes the new vehicles term. When one weight is 0, the correspondent term is cancelled. For instance, if $k_2 = 0$ we have that:

$$\frac{t_3 \times k_3}{k_3} = t_3 \tag{5.2}$$

Hence, we only need to test one value in this situations.

Additionally, when $k_2 = k_3 = val$ and $val \in]0, 1]$ the influence on the expression is the same with any of the values and we only need to take into account one. Following the previous example, we have that if $k_2 = k_3$ then we can substitute both weights by k. The

expression now simplifies as follows:

$$\frac{t_2 \times k + t_3 \times k}{k + k} = \frac{k \times (t_2 + t_3)}{2k} = \frac{t_2 + t_3}{2} \tag{5.3}$$

which again, is independent of the weight values.

For each needed combination, we computed the average value of the objective function values of the solutions produced and we counted the number of vehicles and the number of line exchanges performed. Table 5.8 shows the result of this experiment.

When $k_2 = k_3 = 0$ the correspondent terms of the vehicle utility are cancelled and the utility is defined only by the travel cost. Hence, with this configuration, the objective function is directly minimized since the utility of a vehicle is aligned with it. The average value of the objective function was 25671.2 € with a small standard deviation of 97.34. The best solution produced was the worse from the best solutions achieved on all the configurations.

When $k_2 = 0$ the line exchanges term is cancelled and the utility expression has only a term that attempts to control the number of vehicles on the solution and the travel cost term. When $k_2 = 0$ and $k_3 = 1$ the produced solutions used 55.2 vehicles on average which is the lower average number of vehicles used on all the configurations. This confirms that the term has in fact influenced the algorithm towards solutions with less vehicles. The average number of line exchanges was 311.8 which is a value somewhat high when compared with the remaining configurations. This is expected since the desire of reducing line exchanges is not reflected on the utility expression. It is worth noting that the average quality of the solutions produced improved significantly (25127.6 €). In fact, it was the best average objective function value of all the configurations.

When $k_3 = 0$ the new vehicles term is cancelled and the utility expression has only a term that attempts to control the number of line exchanges and a term for the travel cost. Hence, with $k_2 = 1$ the average number of line exchanges has in fact been reduced. This configuration yielded the lowest average number of line exchanges, which also confirms that the term has influenced the algorithm towards the minimization of the number of line exchanges. Regarding the number of vehicles, we observed that it increased, which is expected since $k_3 = 0$.

In the configuration in which both weights have the same value (e.g. $k_2 = k_3 = 0.5$) both terms are considered in the vehicle utility and attempt to influence the algorithm. We can see that the average number of line exchanges suffers a slight decrease (302.6), with respect to the configurations with $k_2 = 0$ and $k_3 = 0$ or $k_3 = 1$, and the average number of vehicles was not affected. The best solution achieved is also 360 € more expensive, compared to the best solution achieved on the experiment.

The configuration in which $k_2 = 0.5$ and $k_3 = 1$ is interesting since despite the fact that it has the highest values for the average number of vehicles and line exchanges, it achieved a solution whose objective function value is very close to the value of the best solution of this experiment. However with these values the parameters seem to overlap and are not able to influence the algorithm. When we swap these weights ($k_2 = 1$ and

$k_3 = 0.5$) we observe a significant decrease on the average number of line exchanges but the average quality of the solutions was worse.

It is noteworthy to say that the lower average number of line exchanges was 222.6 which is a high value. This high value is due to the fact that even with $k_2 = 1$ and $k_3 = 0$, the weight k_1 is equal to 10. Therefore, the algorithm always prefers to perform more line exchanges instead of performing services with more expensive vehicles (in terms of travel cost).

5.2.4 Ant System and $\mathcal{MM}$AS ACO algorithms

We developed a model for the ACO metaheuristic for our problem which was presented in section 3.3.1. Based on this model, we proposed an AS and a $\mathcal{MM}$AS ACO algorithm. On this section we will evaluate the model and the algorithms developed in terms of effectiveness and performance on a sequential setting.

In this thesis our main algorithm uses the AS ACO algorithm. As discussed in section 2.3, in the literature the $\mathcal{MM}$AS is pointed as a superior algorithm comparing to the AS algorithm for the TSP problem in terms of solution quality. Therefore, we implemented the $\mathcal{MM}$AS in order to compare the results achieved in terms of solution quality with our main algorithm based on the AS ACO algorithm. Hence, the experiments will focus on evaluating the ACO model with the AS. However, experiments which evaluate the learning capabilities of the stochastic learning component will be based on both algorithms.

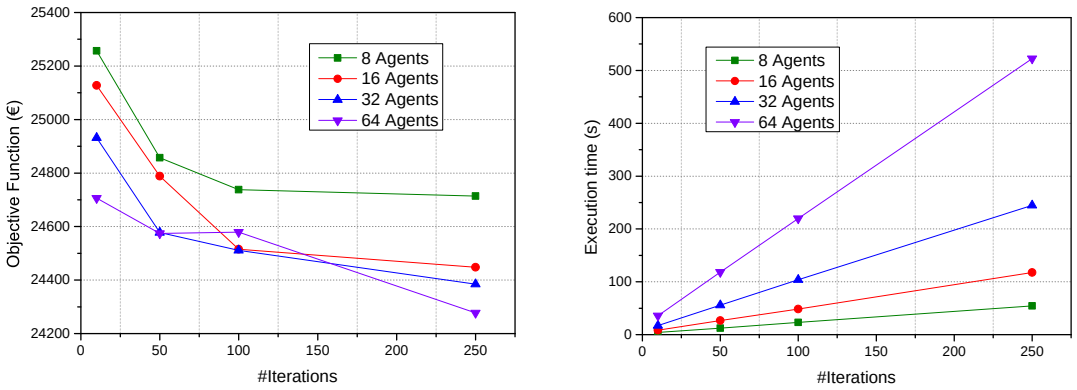
To simplify the analysis, unless stated otherwise, the values of the ACO parameters are defined as shown in table 5.9.

Table 5.9: Default values for each ACO parameter.

Parameter	Value
α	1
β	2
ρ	0.25

Our first experiment aims at evaluating the quality of the solutions produced while varying the number of iterations of the algorithm and the number of agents N launched at each iteration. We executed the AS algorithm with 10, 50, 100 and 250 iterations and, for each number of iterations we set the number of agents to 8, 16, 32 and 64. The results are shown on the plots from figure 5.2.

On the left (figure 5.2a) we can see an average of the objective function values obtained when executing the AS algorithm with different numbers of iterations. On the right (figure 5.2b) we can see a similar plot but with the execution time shown on the Y-axis.



(a) Solution quality for the different configurations (b) Execution times for the different configurations.

Figure 5.2: Analysis of the solution quality (on the left) and execution times (on the right) by varying the number of iterations and the number of agents on each execution.

A first observation is that the quality of the solutions increases as the number of iterations increases. On the other hand, as expected, performing more iterations also takes more time. The execution time also increases when more agents are used.

Regarding the effect of using a different number of agents, it is noticeable that better solutions are obtained when more agents are used. This result is also expected since the number of agents corresponds to the number of solutions constructed at each iteration, therefore, by constructing more solutions the algorithm is allowed to explore more deeper the search space (in terms of the number of solution components explored). Additionally, for example with 64 agents in an execution with 250 iterations, at iteration 240 there are 64 agents that build their solutions using the information gathered from previous executions. More concretely, the pheromone matrix will be in a state in which 240×64 solutions were used to update it. In short, the amount of cooperation is higher which contributes to the construction of better solutions, hence, increasing the number of agents and the number of iterations leads to a better exploration of the search space by the algorithm which in turn leads to better solutions.

Our second experiment intends to evaluate the influence of the parameters α and β on the quality of the solutions produced. By the equation 2.1 from section 2.1, we know that these parameters allow one to control the influence of the information gathered by the agents in previous iterations (α) and the influence of the vehicle utility (β) when selecting a given vehicle from the set $AdmV_i$, by a given agent. Thereby, we fix the number of iterations to 250 and the number of agents at each iteration to 8. Then we execute the algorithm while varying the parameters α and β within values of the set $\{0, 1, 2\}$ and testing all the combinations.

The results of this experiment are shown on table 5.10. In the table we show the average,

the standard deviation and the minimum objective function values of all the combinations of α and β.

Table 5.10: Solution quality of the solutions achieved in function of the parameters α and β. We fixed the number of iterations to 250 and the number of agents to 8. The values of the mean μ, the standard deviation σ and minimum min are shown.

α	β	Objective Function (€)		
		μ	σ	min
0	0	46571.60	584.69	45644.00
	1	25966.20	57.80	25870.00
	2	25100.00	112.75	24928.00
1	0	33521.60	394.09	32939.00
	1	24618.40	73.81	24531.00
	2	24648.00	148.96	24396.00
2	0	38437.80	747.05	37706.00
	1	24923.00	100.91	24836.00
	2	24740.80	334.88	24405.00

With $\alpha = \beta = 0$ the AS algorithm behaves as the restarts metaheuristic using the random heuristic since all the decisions will always get the same probability ($1/|AdmVi|$) and, as expected, the results are very bad. When $\alpha = 0$ and $\beta = 1$ the AS algorithm behaves as the restarts algorithm using the probabilistic heuristic based on the vehicle utility. We somehow confirm this based on the fact that the average of the objective function values of the solutions produced was 25966.2 € while the restarts algorithm had an average of 25685 €, which are both very close. With $\alpha = 0$ and $\beta = 2$ the probabilities of each element of the set $AdmV_i$ are computed based on the square of the utility values. Since the utility values are defined over the set $]0, 1]$, taking the square means that the values will be reduced. This last configuration yielded better results in terms of average quality of the solutions (25100 €) than with $\beta = 1$.

By setting the β value to 0 the AS does not take into account the utility of vehicles and makes decisions based only on learned information. As it was expected, the average quality of the solutions produced, with $\alpha = 1$ and $\alpha = 2$, is very poor compared with the restarts algorithm. This is observed since if agents select solution components without any measure of their quality (e.g. utility), the entries that will be updated are not necessarily good ones.

From the configurations which neither α or β are 0, the configuration which achieved the best average objective function values was with $\alpha = \beta = 1$. However, the best solution was obtained with $\alpha = 1$ and $\beta = 2$ (24396 €).

The ACO metaheuristic uses an additional parameter ρ, as defined in equation 2.2, to control the rate of evaporation. In order to understand the influence of this parameter on the quality of the solutions produced we performed an experiment in which we try different values for ρ. Namely, the values 0, 0.25, 0.5, 0.75 and 1 were tested. The number of iterations was set to 250 and the number of agents executed at each iteration was set to 8. The α and β parameters were set to the default values of this section, shown in table 5.9.

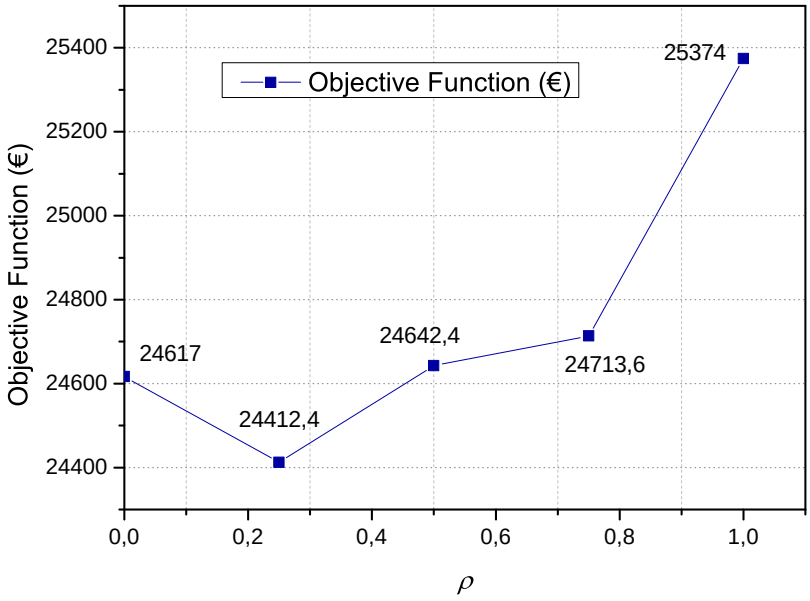


Figure 5.3: Influence of the evaporation rate ρ parameter on the solution quality of the AS algorithm.

Figure 5.3 shows the results of this experiment in which the Y-axis corresponds to the average quality of the solutions obtained with each ρ value. As we can see, this parameter has some influence on the quality of the solutions achieved. With $\rho = 0$ no evaporation occurs and with $\rho = 1$ the values on the pheromone matrix are completely evaporated (set to 0). For $\rho = 1$ it is clear from the results that always forgetting everything after an iteration is a bad decision since in this way, the information gathered by the agents in previous iterations is discarded. In other words, learned information only subsists for one iteration.

The configuration that yielded the best solutions, in average, was the configuration with $\rho = 0.25$. In general the average quality of the solutions decreases as the value ρ increases (increase in the level of forgetfulness) which is an indication that the rate of evaporation must be small ($0 < \rho < 0.5$).

Agents cooperate in gathering knowledge about the search space by constructing solutions using a randomized strategy and those solutions are then used to quantify the

quality of each component of the search space. The experiments presented so far were based on the AS algorithm in which the final solutions quality was analysed but, we did not analyse the behaviour of the stochastic learning mechanism and its influence on the quality of the solutions produced. Our next experiment attempts to evaluate if the stochastic learning mechanism does in fact learn.

We executed the AS and the $\mathcal{MM}$AS algorithms using 250 iterations and 8 agents. For $\mathcal{MM}$AS the parameter a was set to $a = 1$. For each iteration we computed the mean of the objective function value of the 8×5 solutions produced (8 agents and 5 runs) and the obtained values were plotted. Then, we applied a linear regression to the data. The result is shown on figure 5.4.

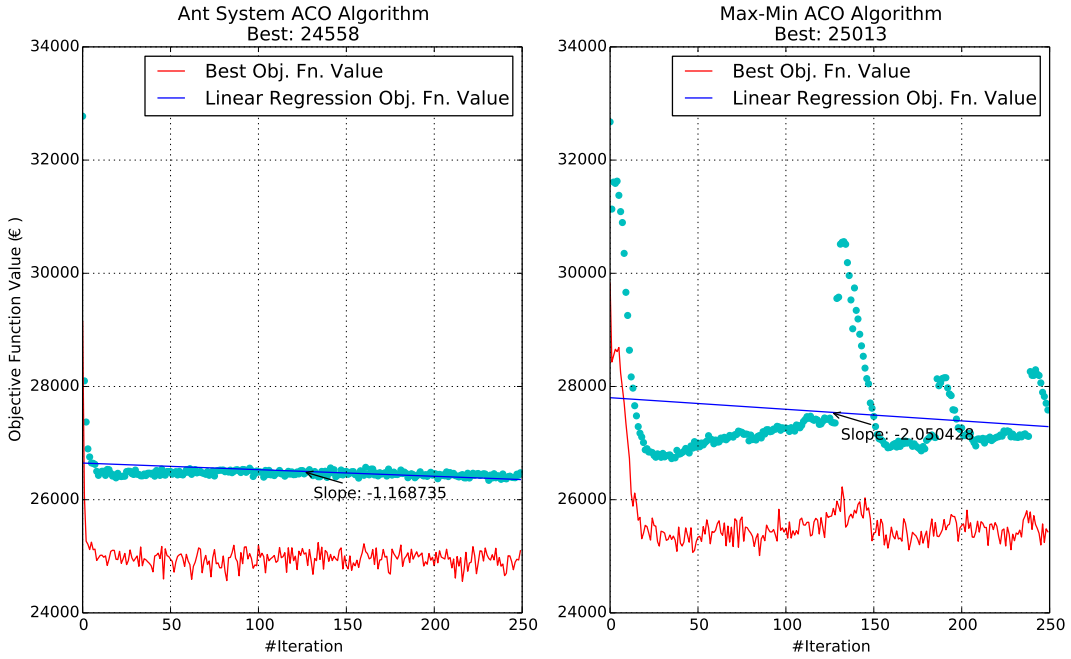


Figure 5.4: Analysis of the learning capabilities of the AS and $\mathcal{MM}$AS algorithms. A linear regression line was computed based on the average values of the objective function of the solutions produced at each iteration, where the slopes indicate that the average quality of the solutions improved as the iterations increased. The number of iterations was set to 250 and the number of agents to 8.

The left plot shows the results with the AS algorithm and the right plot the results with the $\mathcal{MM}$AS algorithm. The cyan dots are the mean values of the objective function values of the solutions produced at each iteration, and the blue line corresponds to the linear regression line. The red line denotes the objective function value of the best solution obtained at each iteration. The slopes of the linear regression lines are also shown.

For both algorithms the slopes of the linear regression lines are negative which indicates the the average quality of the solutions produced improved as the number of iterations increased, i.e., in fact there is some learning.

The AS algorithm was more consistent in the sense that the average quality of the

solutions produced at each iteration is similar (small standard deviation). On the other hand, this not the case with the $\mathcal{MM}$AS algorithm. The $\mathcal{MM}$AS has a pheromone reinitialization mechanism which resets all the pheromone values after a given number of iterations without improvement (in our test we reset pheromones after 100 iterations without improvements). This justifies that, for example around iteration 150, the algorithm stagnates and pheromones are reinitialized. This reinitialization scheme also affects the slope of the linear regression, however, we believe this is a fair comparison because we want to analyse the behaviour of the algorithms on 250 iterations and not only on some subset of the iterations in which no pheromone reinitialization occurred.

We can also observe that the AS algorithm produced better solutions comparing to the $\mathcal{MM}$AS. The best solution of the AS algorithm costs 24558 € while the best solution of the $\mathcal{MM}$AS algorithm costs 25013 €. The AS algorithm also converges faster (in about 20 iterations) but then stagnates. On the other hand the $\mathcal{MM}$AS mechanism of pheromone reinitialization avoids this stagnation. The reason why $\mathcal{MM}$AS performs worse than the AS algorithm may be due to a bad parametrization. This algorithm is more complex and its parametrization is also harder. Further experiments should be performed with different parameters to determine if in fact the extra mechanisms of the $\mathcal{MM}$AS algorithm are effective or not for solving our problem.

As we observed, the AS algorithm stagnates after a few iterations. This is not bad since it keeps searching around areas of the search space which lead to solutions with similar objective functions values. This may be a clue that a local optima exists near these zones of the search space. This stagnation could be suppressed with the introduction of the reinitialization mechanism of the $\mathcal{MM}$AS on the AS algorithm, but without any other changes.

Based on the experiments performed we can conclude that the stochastic learning component in fact improves the effectiveness of the search algorithm and allows ACO to achieve better solutions. Additionally, both AS and $\mathcal{MM}$AS achieved better solutions when comparing with the restarts algorithm, which is an indication that they are in fact superior algorithms.

5.3 Parallel Synchronous and Asynchronous ACO

In this section we evaluate and compare our proposed parallel synchronous and asynchronous ACO algorithms in terms of performance and solution quality.

Unlike the experiments described on the last sections, experiments described on the following sections are performed on the MP machine described in section 5.1.

In the previous section we concluded that the ACO algorithms, namely the AS and the $\mathcal{MM}$AS algorithms, are superior than the restarts algorithm. Therefore from now on, and until the end of this chapter, we do not perform any comparisons between the restarts algorithm and our ACO algorithms (both on a sequential or on a parallel setting).

It is worth noting that in order to preserve coherence between the results reported on our paper [67] published at INFORUM 2015 in which we proposed our parallel asynchronous parallel strategies for ACO, the experiments in this section are based on a slightly different definition of a vehicle utility and no limit to the number of service interruptions is considered.

The vehicle utility $\eta(v)$ of a vehicle v located on a location loc_v, performing a departure Tj starting on terminal it_j, which was used to produce the results of this section, is a function $\eta : \mathbb{R}_{\geq 0} \mapsto]0, 1]$ defined as follows:

$$\eta(v) = e^{\frac{-\left[dist(loc_v, it_j) * distTypeCost(v) * k_1 + dTerm(loc_v, it_j) * k_2\right]}{k_4}} \tag{5.4}$$

Each term of this definition of utility has exactly the same meaning as the ones defined in the vehicle utility definition from section 3.1.2.2. The difference between the two definitions is that the definition proposed in this thesis takes into account an additional term $newVehicle(v)$.

5.3.1 Learning Capability Analysis

In order to assess our algorithms learning capability we performed an experiment in which the number of iterations P is 250 and the number of search agents N is 16. The number of threads used is irrelevant since we are not analysing the speedup. The procedure of this experiment is the same as the procedure described in section 5.2.4 for assessing the influence of the stochastic component on the quality of the solutions produced.

For each iteration $p \in [1, 250]$ of the search, we computed the mean of the objective function of all the 16×5 generated solutions from the 16 agents and 5 runs. We plotted the mean values for each p iteration and we applied a linear regression on the data. The results can be seen in Figure 5.5. In this figure four plots are shown, one for the ASSync and one for each of our concurrency models. The slopes of the linear regression lines (blue lines) are shown and can be interpreted as a measure of the learning rate. The red line shows the best solution obtained during each iteration p. The best solution objective value achieved is shown in the plot title.

We observe that our proposed asynchronous strategies are able to learn as the number of iterations increases, i.e., the quality of the solutions produced improved along the iterations. It is worth noting that both the AsyncLF and AsyncBC algorithms outperformed ASSync in terms of learning intensity and solution quality. The AsyncBM algorithm was not superior in terms of solution quality but was very close to ASSync. The AsyncLF algorithm not only presents the highest learning intensity (slope of ~ -0.22) but also achieved the best solution in the first 100 iterations. However, it takes longer to start achieving solutions with objective function value below 23800 € than the other algorithms. Only after iteration 50 AsyncLF achieves solutions with objective function values clearly

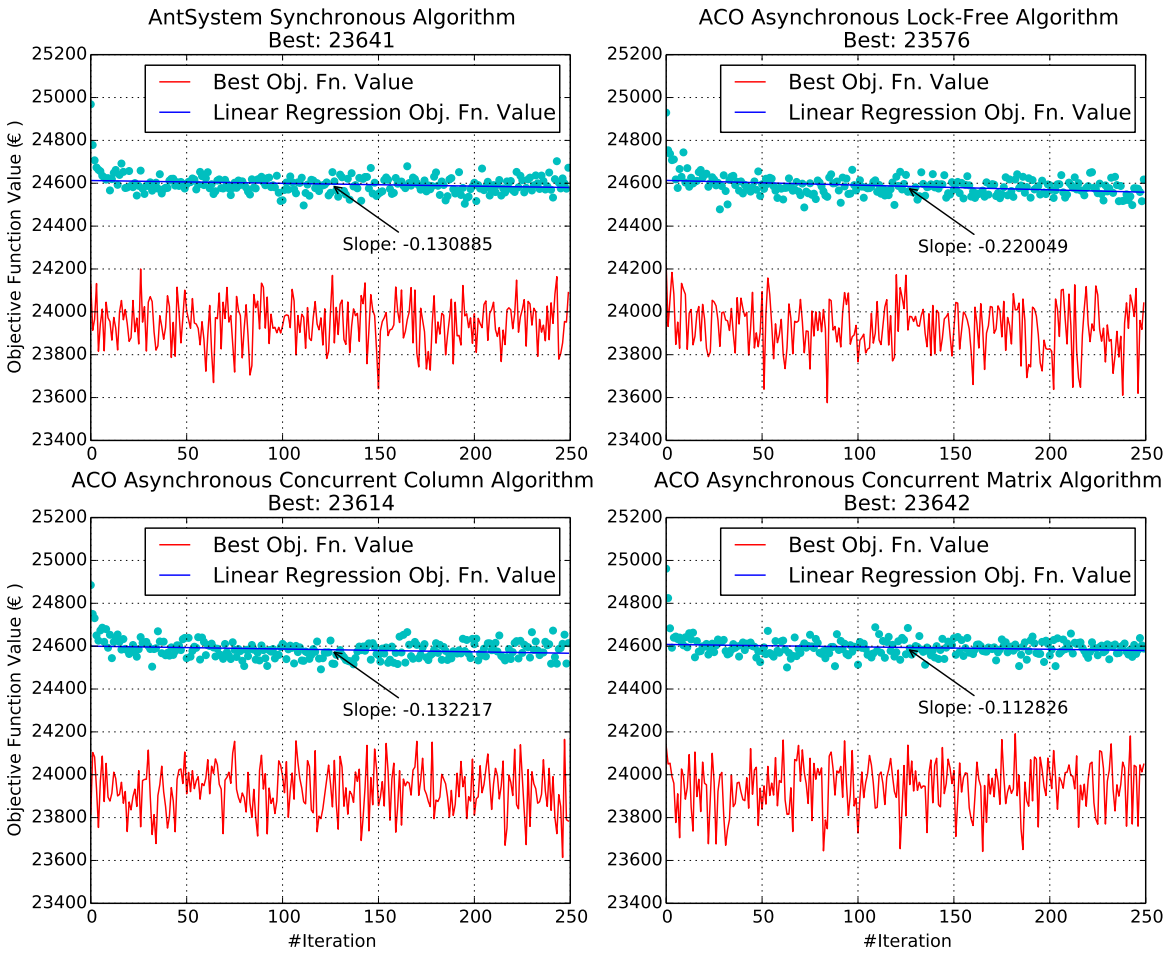


Figure 5.5: Learning Capability Analysis of our parallel synchronous AS and asynchronous AsyncBM, AsyncBC and AsyncLF algorithms, with 250 iterations and 16 agents.

below 23800 €. The AsyncBC and AsyncBM algorithms start achieving quality solutions in less iterations and should be preferred if one wants a good solution quickly.

The ability of achieving good diversity while generating solutions is crucial since it corresponds directly to a better exploration of the search space. In an asynchronous setting the cooperation mechanism achieved is different, hence it has different characteristics. A first modification is that as the degree of concurrency increases, search agents that are executing receive updates sooner from agents that have already finished (agents cooperation is performed early and more often). This allows the executing agents to perform *better* decisions sooner, i.e., agents start exploring promising regions sooner which increases the chances of achieving quality solutions.

An additional aspect is that in a synchronous setting, at the end of each iteration N agents contribute with N solutions, each with $|T|$ departures, which are then used to update the pheromone matrix. This *batch* update, which consists in only one cooperation

phase, may guide the search towards a sub-optimal (e.g. local optimal) solution since some of the desirability components may not be reinforced so frequently. In an asynchronous setting, where the *batch* update does not occur, agents from posterior iterations will use information from partial updates (all the components of one solution or even only one component from a solution) and therefore have a greater change to explore unvisited search space areas. This means that the algorithm will achieve better diversity which directly translates in a more effective search strategy.

5.3.2 Performance Analysis

In our performance experiments we target the following algorithms: sequential AS, ASSync and our three proposed parallel asynchronous algorithms.

The first experiment was performed using 64 threads. With this experiment we intend to analyse how the speedups are affected when the search goes deeper (in number of iterations). We run each algorithm using $p \in [10, 50, 100, 250, 500]$ iterations. This experiment was performed with $8, 16, 32$ and 64 search agents. Figure 5.6 shows the result of the experiment. We observe that in all cases both AsyncLF and ASyncBC algorithms achieves better speedups than ASSync. The AsyncBM algorithm proved to be unstable yielding better speedups than ASSync with 8 and 16 agents but not with 32 and 64.

With 8, 16 and 32 search agents there are threads that were launched but are not working since the number of threads is greater than the number of tasks to be executed. With 64 agents this situation does not occur and each thread gets atleast one agent from each iteration. This explains why speedups are slightly better in general with 64 agents (except for AsyncBM). We observe that AsyncLF achieves almost always better speedups than the other asynchronous strategies. As we expected, and due to the fact that the whole matrix is blocked, the AsyncBM algorithm is the worse asynchronous algorithm. In general and as we expected, we verify that as more concurrency is allowed, the better are the speedups achieved.

For the second experiment the number of iterations was fixed to 1000. This experiment aims to evaluate the scalability of our algorithms as the number of threads increases. We run each algorithm using $th \in [2, 4, 8, 16, 32, 64]$ threads and, for each number of threads th we measured the speedup achieved. Like in the previous experiment, we tested with $4, 8, 16$ and 32 search agents. Figure 5.7 shows the result of the experiment in which the Y-axis corresponds to the speedup achieved and the X-axis to the number of iterations of each execution (logarithmic scale).

We observe a speedup increase as the number of threads increases. In all configurations, the overhead of concurrency mechanisms is only observed when more than 8 threads are used. Like in the previous experiment, the best results were achieved with 64 search agents per iteration. Both results indicate that as we increase the amount of work in each iteration, better speedups are achieved, i.e., parallel resources exploitation is more effective.

Despite of the overhead introduced by read/write locks, the AsyncBC performance is

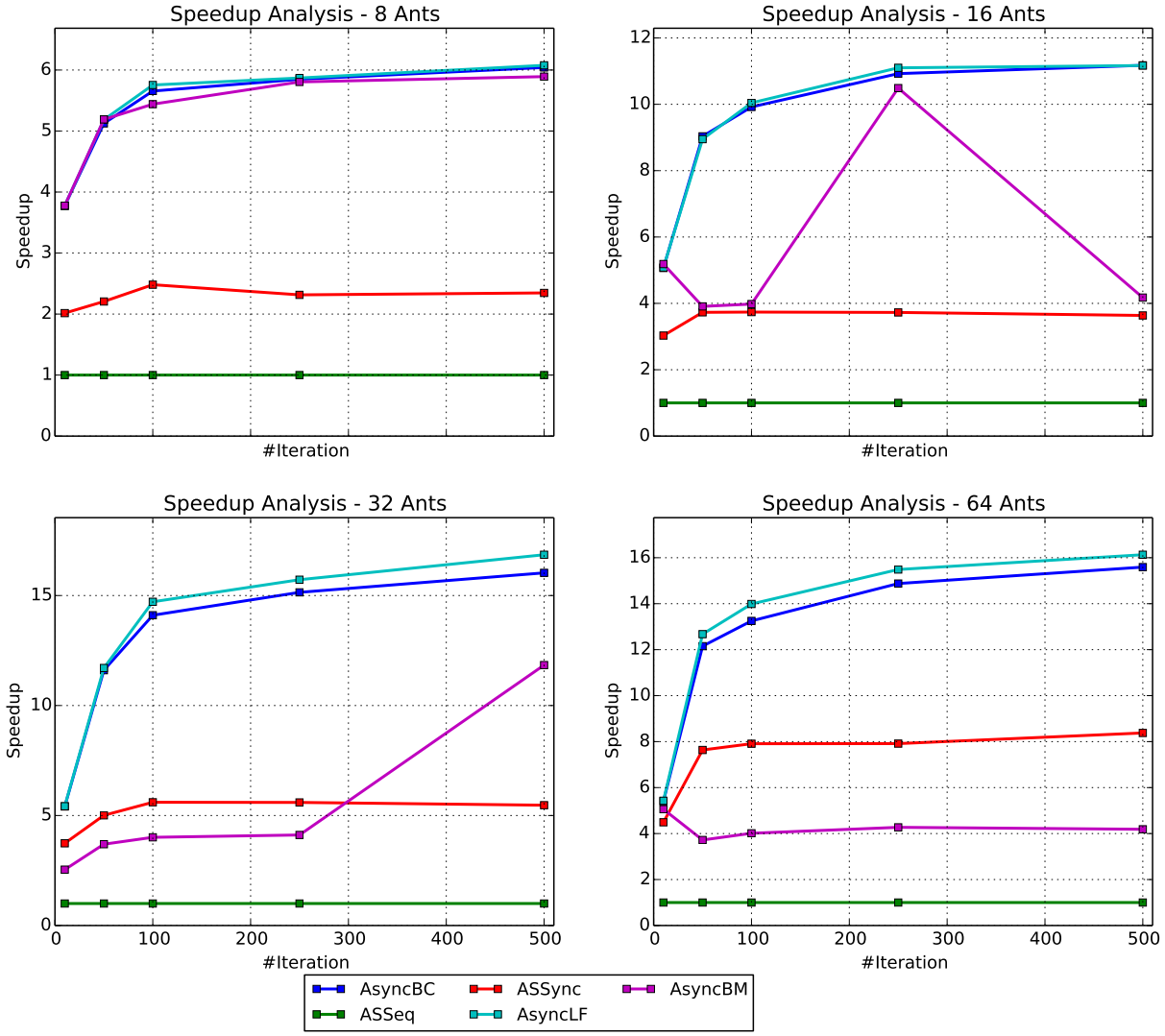


Figure 5.6: Speedup Results with 64 threads.

similar to the AsyncLF algorithm, with the last being slightly better. We observe that both AsyncBC and AsyncLF algorithms are able to scale in terms of workload and number of threads. Furthermore, both algorithms scale almost logarithmically with 32 and 64 agents as the number of threads increases, suggesting that better results would be achieved on a machine with more cores and using more threads.

The AsyncBM algorithm is always worse than the other asynchronous models and sometimes even worse than ASSync, revealing that blocking the entire matrix has a significant negative impact on the performance. The maximum speedup achieved was $\approx$ 17.6x with 64 threads and 64 agents on a 64 core machine, what gives an efficiency of $\approx$ 28%. Even for the best configuration (64 agents) efficiency decreases as the number of threads increases. A strict comparison with other proposed algorithms is not possible to

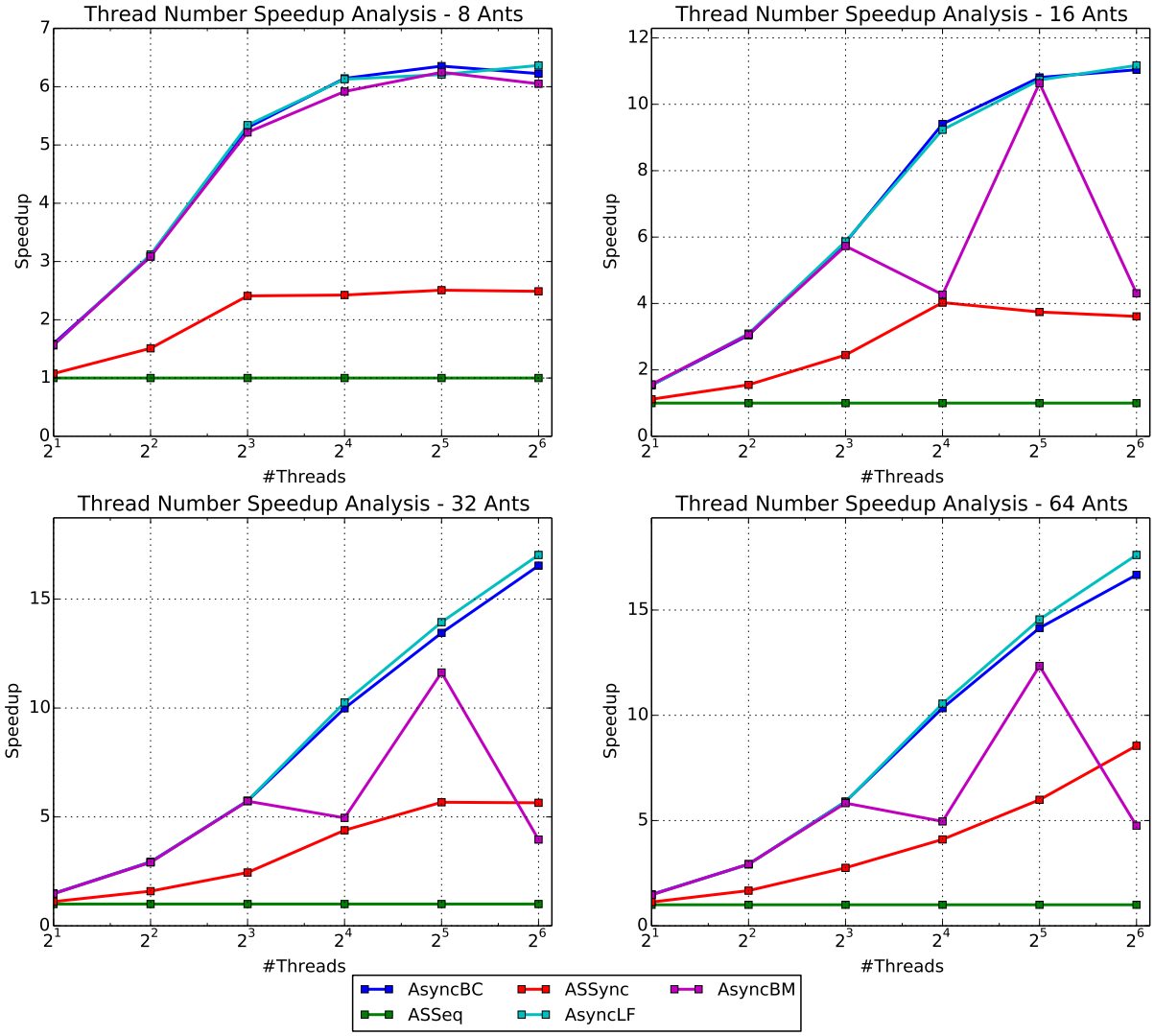


Figure 5.7: Speedup Results with 1000 iterations.

stablish since no ACO parallel algorithm was developed and applied to the variant of vehicle scheduling problem that we address in this thesis.

5.3.3 Parallel Issues

It is worth noting that in our first implementation of both parallel synchronous and asynchronous ACO algorithms we were only able to achieve speedups smaller than 5.5x with CPU cores usage being very low. We performed some experiments and we found out that concurrent memory allocations have a great impact on the performance (*malloc*(3) contention [29, 48]). Therefore, we attempted to minimize the number of memory allocations on parallel regions in our parallel implementations, however, the remaining allocations still affect the performance.

To reduce the number of allocations (calls to the *malloc(3)* system call) we redesigned our parallel algorithms such that needed allocations are performed in the beginning of the execution and outside a parallel region.

We know that if the number of agents is N then for each iteration we need to allocate memory for N solutions. Additionally, we know that each solution will have atleast $|T|$ (number of departures) tasks. In fact this is a very naive approximation since before performing a departure a vehicle usually has additional tasks (*Wait*, *LineExchange*, *VoidIn* and *ServiceInterrupt*) and the same happens after performing a departure (*VoidOut* or *Maintenance* tasks). Therefore, we allocate memory for $N + 1$ solutions in which for each solution we allocate memory for $3 \times |T|$ tasks.

It may happen that $3 \times |T|$ is not enough. When this situation occurs, we allocate memory (now on a parallel region) for one more task. This will introduce a small overhead on a execution of an agent but this overhead will be amortized in the following agents executions.

We allocate memory for $N + 1$ tasks which to correspond to N solutions, one for each agent, and one more for storing the global best solution.

It is worth noting that this scheme introduces some overhead due to the allocations performed. When the search depth is small (small number of iterations) the parallel algorithm is expected to perform worse than the sequential algorithm. Naturally, the overhead also increases as the number of agents N increases.

We also apply the mechanism just described on the parallel restarts algorithm described in section 4.4.1, however, instead of allocating N solutions (one for each agent) we allocate a solution for each thread.

5.4 Summary

To sum up, we will present in this section the results of our experiments.

First, it should be noted that it is not possible to compare our results in terms of solution quality or speedup achieved since, to the best of our knowledge, neither an algorithm for our variant of the VSP neither a parallel ACO algorithm applied to our variant as been addressed or proposed.

We concluded that our restarts algorithm developed produces better solutions as the number of restarts performed increases. From the heuristics proposed, we concluded that the probabilistic heuristic based on the vehicle utility is superior.

Regarding the experiments performed related with our problem model parameters influence and effectiveness we concluded that:

- By allowing line exchanges the algorithm is able to produce solutions that use a less number of vehicles and which are also cheaper;
- The weights which are part of the definition of a vehicle utility are able to effectively influence the solutions produced;

We verified that it is worth using more agents on the AS algorithm since it leads to solutions with better quality. In other words, increasing the amount of cooperation (more agents) leads to better solutions. However, as the number of agents increases the execution time also increases. The α and β parameters play an important role on the ACO metaheuristic. We concluded that a good combination of values for this parameters is with $\alpha = 1$ and $\beta = 2$. With this combination the agents of the AS algorithm base their decisions more on the learned information than on heuristic information about the problem. Our experiments also indicate the evaporation rate value ρ must be low ($0 < \rho < 0.5$).

The ACO metaheuristic is based on a stochastic learning mechanism which aims at gathering information about the search and using that information to influence agents decisions and guide the search towards promising regions. Our experiments which intended to evaluate the learning capability of the AS and $\mathcal{MM}$AS algorithms confirmed that in fact the algorithms are able to learn and the quality of the solutions improves over the iterations. Another conclusion of this experiment is that our implementation of the AS performed better than the $\mathcal{MM}$AS one, in terms of solution quality.

We confirmed that both our ACO algorithms produce better solutions than the restarts algorithm, therefore, to achieve good solutions one should prefer the ACO algorithms.

Regarding parallel synchronous and asynchronous ACO algorithms, our experiments show that our proposed asynchronous algorithms not only are able to learn but also achieved better solutions than ASSync. The AsyncBC and AsyncLF algorithms outperform ASSync in terms of speedups achieved and scalability, with the AsyncLF algorithm yielding the best results.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this final chapter an overall summary of this work will be outlined and the main conclusions are drawn in section 6.1. The most relevant results and contributions will also be discussed in this section.

To conclude, section 6.2 discusses future work ideas to extend and complement this work and enhance its contributions.

6.1 Conclusions and Contributions

The purpose of this work was to develop an algorithm to solve the Public Transport Bus Assignment optimization problem formulated in the introductory chapter, which is a variant of the general VSP problem not addressed yet in the literature, capable of tackling real life instances and producing quality solutions in a reasonable amount of time by exploring parallelism.

Due to the complexity of the problem, we use approximative algorithms which sacrifice optimality in order to produce solutions in a reasonable amount of time. Combining an heuristic algorithm with a sophisticated metaheuristic (Ant-Colony Optimization) allowed us to develop an intelligent approximative algorithm capable of effectively exploring the search space, through cooperation.

It happens that for complex problems, metaheuristics, namely ACO, may also fail to produce quality solutions in a reasonable amount of time. Therefore, we explored and proposed new parallel strategies for this metaheuristic, targeting *shared-memory* architectures, to take advantage of the available computational resources on current *multi-core* and *multiprocessor* machines.

First we developed a model of the problem and implemented a probabilistic heuristic algorithm which is based on a proposed definition of a vehicle utility, and constructively builds a solution while assuring that no constraints are violated. Based on this algorithm,

we developed a restarts metaheuristic and two ACO algorithms: one based on the Ant System algorithm and one based on the $\mathcal{MAX}$–$\mathcal{MIN}$ algorithm.

Parallel versions of the restarts and ACO algorithms were developed. Regarding ACO, in order to mitigate the straggler problem, we proposed three different parallel asynchronous strategies which differ in the degree of concurrency allowed while accessing the learned information of the search history.

Our results shown that both ACO algorithms produced better solutions than the restarts algorithm, with the Ant System algorithm yielding the best results. The implicit cooperation mechanism of ACO, in which agents cooperate in the task of classifying components of the search space, yields in fact better solutions. Results have shown that our ACO model allowed the metaheuristic to learn from the collected knowledge and improve the quality of the solutions over the iterations. Furthermore, we concluded that increasing the amount of cooperation (agents per iteration) allows the algorithm to produce better results.

Regarding our parallel algorithms, namely the parallel synchronous and asynchronous ACO algorithms, we concluded that our asynchronous strategies, namely the ASyncBC and AsyncLF algorithms, outperformed in terms of performance and solution quality the synchronous ones, achieving speedups of $\approx$ 17.6x with our best asynchronous algorithm (AsyncLF). It is worth noting that asynchronism changes the original cooperation scheme of ACO and, based on our results, we concluded that with asynchronism cooperation is more effective (in terms of learning rate) and better solutions are achieved.

We contributed with approximative algorithms for solving the PTBA problem, based on the restarts and on the ACO metaheuristic.

Regarding parallel metaheuristics, we contributed with a parallel version of the restarts and parallel synchronous versions of the AS and $\mathcal{MMAS}$ algorithms. We also contributed with three parallel asynchronous algorithms, in which two proved to be superior than the synchronous ones. Our proposal of asynchronous strategies for the AS algorithm was submitted and accepted on the INFORUM 2015 conference [67].

An extensible Parallel Optimization Library for the PTBA problem (ParallelPTBAP-OptLib) was developed. This library provides all the algorithms implemented in this work and offers abstractions that allow programmers to easily develop applications to solve this problem or similar variants.

6.2 Future Work

In future work we intend to develop a linear relaxation of our problem and solve it using a complete search method (where optimality is guaranteed) in order to obtain a lower bound of the solution quality for a given instance. This will allow us to measure the error of our approximative algorithms. Additional experiments, taking into account all the parameters, must be performed in order to determine a good global parametrization for the algorithms.

In order to improve the performance of our parallel synchronous and asynchronous ACO algorithms different *malloc* implementations (e.g. *TCMalloc* or *jemalloc*) should be experimented, instead of the *glib* implementation, since we observed that heap contention is not allowing the algorithm to fully utilize the CPU.

The ACO algorithms can be improved by extending the pheromone trails model. Instead of having a row for each location, the set of all possible vehicles can be considered. This model significantly increases the pheromone matrix dimensions and, the computations performed by each search agent become more expensive. Introducing a second level of parallelism, through GPUs, to compute the ACO probability distribution values for each vehicle at each decision point would be advantageous. This pheromone trails model allows for a more complete classification of the search space components.

BIBLIOGRAPHY

[1] E. Aarts and J. Lenstra, eds. *Local Search in Combinatorial Optimization*. Discrete Mathematics and Optimization. Wiley, Chichester, UK, 1997. ISBN: 0-471-94822-5.

[2] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms*. NJ, USA: John Wiley & Sons, Aug. 2005. ISBN: 0-471-67806-6. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471678066.html.

[3] E. Alba, G. Luque, and S. Nesmachnow. "Parallel metaheuristics: recent advances and new trends." English. In: *Int. Trans. Oper. Res.* 20.1 (2013), pp. 1–48. ISSN: 0969-6016; 1475-3995/e. DOI: 10.1111/j.1475-3995.2012.00862.x.

[4] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[5] A. A. Bertossi, P. Carraresi, and G. Gallo. "On some matching problems arising in vehicle scheduling models." In: *Networks* 17.3 (1987), pp. 271–281. URL: http://dblp.uni-trier.de/db/journals/networks/networks17.html#BertossiCG87.

[6] C. Blum and M. Dorigo. "The Hyper-cube Framework for Ant Colony Optimization". In: *Trans. Sys. Man Cyber. Part B* 34.2 (Apr. 2004), pp. 1161–1172. ISSN: 1083-4419. DOI: 10.1109/TSMCB.2003.821450. URL: http://dx.doi.org/10.1109/TSMCB.2003.821450.

[7] O. A. R. Board. *The OpenMP® API specification for parallel programming*. 2014. URL: http://openmp.org/ (visited on 01/21/2015).

[8] I. Boussaïd, J. Lepagnot, and P. Siarry. "A survey on optimization metaheuristics". In: *Information Sciences* 237.0 (2013). Prediction, Control and Diagnosis using Advanced Neural Computations, pp. 82 –117. ISSN: 0020-0255. DOI: http://dx.doi.org/10.1016/j.ins.2013.02.041. URL: http://www.sciencedirect.com/science/article/pii/S0020025513001588.

[9] G. Brønmo, M. Christiansen, K. Fagerholt, and B. Nygreen. "A multi-start local search heuristic for ship scheduling - a computational study." In: *Computers & OR* 34.3 (Dec. 20, 2006), pp. 900–917. URL: http://dblp.uni-trier.de/db/journals/cor/cor34.html#BronmoCFN07.

[10] B. Bullnheimer, R. F. Hartl, and C. Strauß. "A New Rank Based Version of the Ant System - A Computational Study". In: *Central European Journal for Operations Research and Economics* 7 (1997), pp. 25–38.

[11] J. Błażewicz, P. Formanowicz, and M. Kasprzak. "Selected combinatorial problems of computational biology". In: *European Journal of Operational Research* 161.3 (2005), pp. 585 –597. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2003.10.054. URL: http://www.sciencedirect.com/science/article/pii/S0377221704002218.

[12] Y. Caniou and P. Codognet. "Communication in Parallel Algorithms for Constraint-Based Local Search". In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. 2011, pp. 1961–1970. DOI: 10.1109/IPDPS.2011.357.

[13] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. "Solving the Straggler Problem with Bounded Staleness". In: *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. Santa Ana Pueblo, New Mexcio: USENIX Association, 2013, pp. 22–22. URL: http://dl.acm.org/citation.cfm?id=2490483.2490505.

[14] M. Correia, P. Baharona, and F. Azevedo. "CaSPER: A Programming Environment for Development and Integration of Constraint Solvers". In:

[15] P. J. Courtois, F. Heymans, and D. L. Parnas. "Concurrent control with "readers" and "writers"". In: *Commun. ACM* 14.10 (Oct. 1971), pp. 667–668. ISSN: 0001-0782. DOI: 10.1145/362759.362813. URL: http://doi.acm.org/10.1145/362759.362813.

[16] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné. "Parallel implementation of an ant colony optimization metaheuristic with OpenMP". In: *International Conference on Parallel Architectures and Compilation Techniques*. 2001.

[17] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price. "Comparing Parallelization of an ACO: Message Passing vs. Shared Memory". In: *Proceedings of the Second International Conference on Hybrid Metaheuristics*. HM'05. Barcelona, Spain: Springer-Verlag, 2005, pp. 1–11.

[18] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. "Parallel Ant Colony Optimization on Graphics Processing Units". In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 52 –61.

[19] D. Diaz, S. Abreu, and P. Codognet. "Parallel Constraint-Based Local Search on the Cell/BE Multicore Architecture". English. In: *Intelligent Distributed Computing IV*. Ed. by M. Essaaidi, M. Malgeri, and C. Badica. Vol. 315. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2010, pp. 265–274. ISBN: 978-3-642-15210-8. DOI: 10.1007/978-3-642-15211-5_28. URL: http://dx.doi.org/10.1007/978-3-642-15211-5_28.

[20] M. Dorigo and L. M. Gambardella. "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem". In: *Trans. Evol. Comp* 1.1 (Apr. 1997), pp. 53–66. ISSN: 1089-778X. DOI: 10.1109/4235.585892. URL: http://dx.doi.org/10.1109/4235.585892.

[21] M. Dorigo, V. Maniezzo, and A. Colorni. "Ant System: Optimization by a Colony of Cooperating Agents". In: *Trans. Sys. Man Cyber. Part B* 26.1 (Feb. 1996), pp. 29–41. ISSN: 1083-4419. DOI: 10.1109/3477.484436. URL: http://dx.doi.org/10.1109/3477.484436.

[22] M. Dorigo. "Optimization, Learning and Natural Algorithms". PhD thesis. Italy: Politecnico di Milano, 1992.

[23] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004. ISBN: 0262042193.

[24] J. Dréo. *Shortest path find by an ant colony*. 2006. URL: https://commons.wikimedia.org/wiki/File:Aco_branches.svg (visited on 08/10/2015).

[25] A. Duarte and R. Martí. "Tabu search and {GRASP} for the maximum diversity problem". In: *European Journal of Operational Research* 178.1 (2007), pp. 71 –84. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2006.01.021. URL: http://www.sciencedirect.com/science/article/pii/S0377221706000634.

[26] R. Duncan. "A survey of parallel computer architectures". In: *Computer* 23.2 (1990), pp. 5–16. ISSN: 0018-9162. DOI: 10.1109/2.44900.

[27] B. Dávid and M. Krész. "Application Oriented Variable Fixing Methods for the Multiple Depot Vehicle Scheduling Problem." In: *Acta Cybern.* 21.1 (2013), pp. 53–73. URL: http://dblp.uni-trier.de/db/journals/actaC/actaC21.html#DavidK13.

[28] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005. ISBN: 0471467405.

[29] J. Evans. *A Scalable Concurrent malloc(3) Implementation for FreeBSD*. 2006.

[30] C. A. Floudas and P. M. Pardalos, eds. *Encyclopedia of Optimization, Second Edition*. Springer, 2009. ISBN: 978-0-387-74758-3. URL: http://dblp.uni-trier.de/db/reference/opt/opt2009.html.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[32] M. Gendreau and J.-Y. Potvin. *Handbook of Metaheuristics*. 2nd. Springer Publishing Company, Incorporated, 2010. ISBN: 1441916636, 9781441916631.

[33] V. Gintner, V. Gintner, I. Steinzen, I. Steinzen, L. Suhl, and L. Suhl. "A variable fixing heuristic for the multiple-depot integrated vehicle and crew scheduling problem". In: *Proceedings Of 10th Meeting of the European Working Group of Transportation (EWGT) Poznan, Poland* (2005).

[34] F. Glover. "Tabu Search—Part I". In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206. DOI: 10.1287/ijoc.1.3.190. eprint: http://dx.doi.org/10.1287/ijoc.1.3.190. URL: http://dx.doi.org/10.1287/ijoc.1.3.190.

[35] F. Glover. "Tabu Search—Part II". In: *ORSA Journal on Computing* 2.1 (1990), pp. 4–32. DOI: 10.1287/ijoc.2.1.4. eprint: http://dx.doi.org/10.1287/ijoc.2.1.4. URL: http://dx.doi.org/10.1287/ijoc.2.1.4.

[36] F. Glover. "Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems". In: *Discrete Appl. Math.* 65.1-3 (Mar. 1996), pp. 223–253. ISSN: 0166-218X. DOI: 10.1016/0166-218X(94)00037-E. URL: http://dx.doi.org/10.1016/0166-218X(94)00037-E.

[37] T. F. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics (Chapman & Hall/Crc Computer & Information Science Series)*. Chapman & Hall/CRC, 2007. ISBN: 1584885505.

[38] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (July 2008), pp. 33–38. ISSN: 0018-9162. DOI: 10.1109/MC.2008.209. URL: http://dx.doi.org/10.1109/MC.2008.209.

[39] H. Hoos and T. Sttzle. *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608729.

[40] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[41] A. Kaminsky. *BIG CPU, BIG DATA - Solving the World's Toughest Computational Problems with Parallel Computing*. Department of Computer Science B. Thomas Golisano College of Computing and Information Sciences Rochester Institute of Technology, 2015.

[42] A. Kapoulkine. *pugixml XML processing C++ Library*. URL: http://pugixml.org/ (visited on 09/05/2015).

[43] G. Kotsis, B. Bullnheimer, and C. Strauss. *Parallelization Strategies for the ANT System*. Technical Report. 1997.

[44] V. Kumar. *Introduction to Parallel Computing*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201648652.

[45] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server". In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pp. 241–251. ISSN: 0163-5964. DOI: 10.1145/384286.264206. URL: http://doi.acm.org/10.1145/384286.264206.

[46] B. Laurent and J.-K. Hao. "Iterated Local Search for the Multiple Depot Vehicle Scheduling Problem". In: *Comput. Ind. Eng.* 57.1 (Aug. 2009), pp. 277–286. ISSN: 0360-8352. DOI: 10.1016/j.cie.2008.11.028. URL: http://dx.doi.org/10.1016/j.cie.2008.11.028.

[47] J. K. Lenstra and A. H. G. R. Kan. "Complexity of vehicle routing and scheduling problems." In: *Networks* 11.2 (1981), pp. 221–227. URL: http://dblp.uni-trier.de/db/journals/networks/networks11.html#LenstraK81.

[48] C. Lever and D. Boreham. "Malloc() Performance in a Multithreaded Linux Environment". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '00. San Diego, California: USENIX Association, 2000, pp. 56–56. URL: http://dl.acm.org/citation.cfm?id=1267724.1267780.

[49] *Linux Man page: numa(3) - NUMA policy library*. URL: http://linux.die.net/man/3/numa (visited on 08/11/2015).

[50] *Linux Programmer's Manual: numa(7) - overview of Non-Uniform Memory Architecture*. 2012. URL: http://man7.org/linux/man-pages/man7/numa.7.html (visited on 08/11/2015).

[51] T. V. Luong, N. Melab, and E.-G. Talbi. "GPU Computing for Parallel Local Search Metaheuristic Algorithms". In: *Computers, IEEE Transactions on* 62.1 (2013), pp. 173–185. ISSN: 0018-9340. DOI: 10.1109/TC.2011.206.

[52] T. V. LUONG. "Parallel Metaheuristics on GPU". PhD thesis. France: LIFL Lille 1 University - INRIA Lille Nord Europe, Dec. 2011.

[53] R. Martí, Moreno-Vega, and A. Duarte. "Advanced Multi-start Methods". In: *Handbook of Metaheuristics*. Ed. by M. Gendreau and J.-Y. Potvin. Vol. 146. International Series in Operations Research & Management Science. Springer US, 2010, pp. 265–281. DOI: 10.1007/978-1-4419-1665-5_9. URL: http://dx.doi.org/10.1007/978-1-4419-1665-5_9.

[54] R. Martí, M. G. Resende, and C. C. Ribeiro. "Multi-start methods for combinatorial optimization". In: *European Journal of Operational Research* 226.1 (2013), pp. 1 –8. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2012.10.012. URL: http://www.sciencedirect.com/science/article/pii/S0377221712007394.

[55] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439, 9780124159938.

[56] W. McKinney. *pandas: a Foundational Python Library for Data Analysis and Statistics*.

[57] L. Michel and P. V. Hentenryck. "A Constraint-based Architecture for Local Search". In: *SIGPLAN Not.* 37.11 (Nov. 2002), pp. 83–100. ISSN: 0362-1340. DOI: 10.1145/583854.582430. URL: http://doi.acm.org/10.1145/583854.582430.

[58] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965).

[59] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1982. ISBN: 0-13-152462-3.

[60] M. Pedemonte, S. Nesmachnow, and H. Cancela. "A Survey on Parallel Ant Colony Optimization". In: *Appl. Soft Comput.* 11.8 (Dec. 2011), pp. 5181–5197. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2011.05.042. URL: http://dx.doi.org/10.1016/j.asoc.2011.05.042.

[61] G. project. *GCC 5 compiler*. URL: https://gcc.gnu.org/gcc-5/ (visited on 09/05/2015).

[62] *QT C++ Application Development Framework*. URL: http://www.qt.io/ (visited on 09/05/2015).

[63] M. G. C. Resende, P. M. Pardalos, and S. D. Eksioglu. "Parallel Metaheuristics for Combinatorial Optimization". In: *INTERNATIONAL SCHOOL ON ADVANCED ALGORITHMIC TECHNIQUES FOR PARALLEL COMPUTATION WITH APPLICATIONS*. Kluwer Academic, 1999, pp. 179–206.

[64] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006. ISBN: 0444527265.

[65] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594.

[66] M. L. Scott. "Shared-Memory Synchronization". In: Morgan & Claypool, 2013.

[67] D. Semedo, P. Barahona, and P. Medeiros. "Asynchronous Parallel Ant-Colony Optimization Strategies: Application to the Multi-Depot Vehicle Scheduling Problem with Line Exchanges". In: *INForum 2015 - Actas do 7º Simpósio de Informática*. Sept. 2015.

[68] D. F. Semedo, P. Barahona, and M. Correia. "CaSPER LS : A Constraint-Based Local Search Solver for CaSPER". In: (2014).

[69] T. Stützle and H. H. Hoos. "MAX-MIN Ant System". In: *Future Gener. Comput. Syst.* 16.9 (June 2000), pp. 889–914. ISSN: 0167-739X. URL: http://dl.acm.org/citation.cfm?id=348599.348603.

[70] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[71] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard. "Parallel Ant Colonies for the Quadratic Assignment Problem". In: *Future Gener. Comput. Syst.* 17.4 (Jan. 2001), pp. 441–449. ISSN: 0167-739X. DOI: 10.1016/S0167-739X(99)00124-7. URL: http://dx.doi.org/10.1016/S0167-739X(99)00124-7.

[72] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009. ISBN: 0470278587, 9780470278581.

[73] M. Toulouse, T. G. Crainic, and M. Gendreau. "Communication Issues in Designing Cooperative Multi-Thread Parallel Searches". In: *Meta-Heuristics: Theory and Applications*. Kluwer Academic Publishers, 1995, pp. 501–522.

[74] M. Toulouse, T. G. Crainic, and B. Sansó. "Systemic behavior of cooperative search algorithms". In: *Parallel Computing* 30.1 (2004), pp. 57 –79. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2002.07.001. URL: http://www.sciencedirect.com/science/article/pii/S0167819103001145.

[75] S. Tsutsui and N. Fujimoto. "Parallel Ant Colony Optimization Algorithm on a Multi-core Processor". In: *Proceedings of the 7th International Conference on Swarm Intelligence*. ANTS'10. Brussels, Belgium: Springer-Verlag, 2010, pp. 488–495. ISBN: 3-642-15460-3.

[76] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005. ISBN: 0262220776.

[77] K. Vanitchakornpong, N. Indra-Payoong, A. Sumalee, and P. Raothanachonkun. "Constrained Local Search Method for Bus Fleet Scheduling Problem with Multi-depot with Line Change." In: *EvoWorkshops*. Ed. by M. Giacobini, A. Brabazon, S. Cagnoni, G. D. Caro, R. Drechsler, A. Ekárt, A. Esparcia-Alcázar, M. Farooq, A. Fink, J. McCormack, M. O'Neill, J. Romero, F. Rothlauf, G. Squillero, S. Uyar, and S. Yang. Vol. 4974. Lecture Notes in Computer Science. Springer, Apr. 15, 2008, pp. 679–688. ISBN: 978-3-540-78760-0. URL: http://dblp.uni-trier.de/db/conf/evoW/evoW2008.html#VanitchakornpongISR08.

[78] M. Verhoeven and E. Aarts. "Parallel local search". English. In: *Journal of Heuristics* 1.1 (1995), pp. 43–65. ISSN: 1381-1231. DOI: 10.1007/BF02430365. URL: http://dx.doi.org/10.1007/BF02430365.

[79] Wikipedia. *Flynn's taxonomy*. 2014. URL: http://en.wikipedia.org/wiki/Flynn%27s_taxonomy (visited on 01/15/2015).