FAKULTÄT FÜR !NFORMATIK

Parallel Variable Neighbourhood Search for the Car Sequencing Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Informatik

eingereicht von

Markus Knausz
Matrikelnummer 9926567

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.Ass. Mag.rer.soc.oec. Dipl.-Ing. Matthias Prandtstetter

Wien, 27.10.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

Variable Neighborhood Search (VNS) is a relatively new metaheuristic for solving hard combinatorial optimisation problems. One such optimisation problem is the Car Sequencing Problem (CarSP), where a sequence of cars along the assembly line with minimum production costs has to be found. Although VNS is a successful metaheuristic, it takes a long time until a suitable solution is found for real-world instances of CarSP. Two approaches should be investigated in more detail: Firstly, the efficiency of neighborhoods, i.e. the relation of computation time and the solution improvement, should be used for identifying efficient neighborhood orderings on the fly. Secondly, the high potential of parallelisation techniques should be exploited. Within this thesis both approaches are combined. Computational tests showed that a substantial reduction of the computation time is possible. Further, the tests revealed that no "perfect" neighborhood ordering can be identified which implies that such a parallel self-adaptive approach is valuable and necessary for obtaining good solution qualities.

Kurzfassung

Variable Nachbarschaftssuche (VNS) ist eine relative neue Metaheuristik zum Lösen von schwierigen kombinatorischen Optimierungsproblemen. Ein solches Optimierungsproblem ist das Car Sequencing Problem (CarSP), wo eine Anordnung von Autos am Fließband mit minimalen Produktionskosten gefunden werden muss. Obwohl VNS eine erfolgreiche Metaheuristik ist, dauert es für praxisnahe Instanzen von CarSP eine lange Zeit bis eine brauchbare Lösung gefunden wird. Zwei Ansätze sollen ausführlicher untersucht werden: Erstens sollte die Effizienz von Nachbarschaften, d.h. das Verhältnis von Berechnungszeit und Lösungsverbesserung, während der Programmausführung verwendet werden, um effiziente Nachbarschaftsreihenfolgen zu bestimmen. Zweitens sollte das hohes Parallelisierungspotential ausgenutzt werden. Im Rahmen dieser Diplomarbeit werden beide Ansätze kombiniert. Die Tests zeigten, dass eine beträchtliche Reduktion der Berechnungszeit möglich ist. Weiters haben die Tests gezeigt, dass keine ,,perfekte" Nachbarschaftsreihenfolge identifiziert werden kann was bedeutet, dass ein paralleler selbst-adaptiver Ansatz nützlich und wichtig ist, um gute Lösungsqualitäten zu erhalten.

Danksagung

An dieser Stelle möchte ich mich, sowohl bei jenen bedanken, die mir durch ihre Unterstützung das Schreiben dieser Diplomarbeit erst ermöglicht haben, als auch bei jenen, die entscheidend zum Gelingen dieser Arbeit beigetragen haben.

Günther Raidl und Matthias Prandtstetter danke ich für die ausgezeichnete Betreuung, Motivation und für die vielen wertvollen Hinweise zum wissenschaftlichen Arbeiten.

Weiters gilt mein besonderer Dank Elfriede Hofer für das zeitaufwändige Korrekturlesen der Arbeit.

Nicht zuletzt möchte ich mich bei meiner Familie bedanken, wobei mein besonderer Dank meiner Mutter gilt, die mir ein sorgloses Studium ermöglicht hat.

Contents

1 Introduction

Approximately 100 years ago Ford Motor Company introduced the Model T. It was the first car mass produced on assembly lines. The advantage of this new production method was a massive cost reduction, but with the disadvantage that only a single, standardised product could be produced on the same assembly line. However, requirements have changed dramatically. Nowadays the automotive industry has to offer a high product variety, where the customer can choose among a multitude of components. But due to the economic pressure, the automotive industry still has to produce the cars cost-effectively. To meet both antithetical demands an additional planning and optimisation process is necessary. One important planning step is the daily sequence planning where an arrangement of cars on the assembly line with minimum production costs has to be found.

One model for solving the sequence problem is the Car Sequencing Problem. Thereby, complex requirements are modelled by rules of the form l_c/m_c, meaning no more than l_c cars are allowed to require component c in a consecutive interval of length m_c. Although only a part of the overall optimisation process is considered and a simplified model is used, exact algorithms are not able to find an optimal solution for real life instances within suitable time. Thus metaheuristics are applied which sacrifice guarantees on solution quality for speeding up the solution process. However, even for metaheuristics it takes a long time until a reasonable solution is provided. If taking into account several production days or considering the whole planning process to further reduce the production costs, things still get worse. A promising approach to overcome this performance problem is a parallelisation of metaheuristics.

Another important aspect for solving the Car Sequencing Problem is the performance of the used microprocessors. In the past decades microprocessors were undergoing a rapid performance improvement. One reason was a significant raise of the clock speed. However, the clock speed cannot be increased arbitrarily, because increasing the clock speed leads to a disproportionately high power consumption. Furthermore, signal speeds set boundaries to the clock speed. In addition to the increase of the clock frequency, a lot of architectural improvements were implemented. These architectural enhancements were mainly done by implicit parallelisation techniques like pipelining, superscalar execution and so forth. However, these implicit parallelisation techniques reach their limits, due to data dependencies between instructions. To realise a further performance advancement of processors, multiple independent cores are combined to a single processor, called multi-core processor. The disadvantage is that the parallelisation has to be done explicitly by the programmer. Therefore, if a program is not written to use multiple processors, it does not profit from the additional cores. Furthermore, if in the future performance improvements are mainly done by adding additional cores, a sequential algorithm cannot benefit from a new processor. Therefore parallel metaheuristics are currently not only important to speedup the search process, but also essential to benefit from future processors.

A relatively new and successful metaheuristic which has only a few parameters to be tuned is *Variable Neighbourhood Search* (VNS). One crucial point in the performance of a VNS approach is using appropriate neighbourhoods and deciding in which order to use them. Currently most frequently used is to order neighbourhoods by increasing complexity and using afterwards this static neighbourhood-order during the whole program execution. However, this approach is not optimal. The key question is what the efficiency of the used neighbourhood is, i.e. the relation of solution improvement and time needed for an improvement. In this thesis, I present a parallel VNS approach which tries to find an efficient usage of neighbourhoods to speedup the optimisation process. Thereby the additional computing power is used to determine the most efficient neighbourhood repeatedly.

Thesis Overview

The next chapter gives a detailed description of the Car Sequencing Problem. Furthermore, a formal definition of the Car Sequencing Problem as proposed by Renault for the ROADEF challenge is given and complexity aspects are discussed. Chapter 3 is devoted to previous work relevant for this thesis. In Chapter 4, I first discuss Variable Neighbourhood Search in detail. Afterwards an introduction to the topic of parallel computing is given. Thereby, the basics of parallel computers and their programming models are discussed. Furthermore different parallel performance measures are presented. Finally, I present a classification of parallelisation strategies for Variable Neighbourhood Search. In chapter 5, I first give a formal description of an efficient usage of neighbourhoods. Based on this theoretical formulation, I propose a special, for parallelisation adapted sequential Variable Neighbourhood Search (VNS) and the corresponding new parallel VNS approach. In Chapter 7 and 8, I present the test results and discuss potential improvements.

2 The Car Sequencing Problem

This chapter discusses the Car Sequencing Problem (CarSP). First an overview of all necessary planning steps of an assembly line is presented. Then a description of sequence planning with its constraints is given. Afterwards a formal definition of the CarSP as defined by the car manufacture Renault for the ROADEF challenge 2005, is presented.

2.1 Production Planning

One of the challenges in automotive industry is to offer a wide variety of individual cars at low cost. These different equipped cars are very similar to each other, but differ in colour and installed components. Although assembly lines were originally developed to produce a standardised product very cost-effectively, most car manufactures produce a wide range of different cars along the same assembly line. Therefore, an additional planning phase is necessary for economic reasons. According to [5] the following planning steps are needed:

- *Initial configuration.* In this step all needed resources and the detailed configuration of the assembly line are determined, depending primarily on future sales of the cars.
- *Reconfiguration.* There may be changes in the demand for a special equipped car or there may be technological developments. In this case the initial configuration of the assembly line must be reconfigured.
- *Master scheduling.* In this planning step the overall ordered cars are assigned to smaller planning periods, mostly on a daily basis. The assignment depends on the configuration of the cars and the order time of the customers.
- *Sequence planning.* After the assignment of cars to a single day, an arrangement of the cars must be found such that all constraints are satisfied and/or the costs are minimised.
- *Resequencing.* In the production process there may be unforeseen disturbances such as a machine breakdown. In this case a rearrangement of the predefined sequence may lead to better results.

Although dependencies exist between the above mentioned planning steps the individual planning steps are mostly performed independently of each other. As CarSP mainly addresses the daily sequencing of cars, the next section discusses sequence planning in more detail.

2.2 Sequence Planning

In automotive industry an assembly line is composed of three stages: body shop, paint shop and assembly shop. First of all, the chassis of the automobile is produced in the body shop. Then the vehicle is painted with the desired colour in the paint shop. Finally, in the assembly shop, each vehicle gets the different components such as an air condition, a sound system and the engine. Because the requirements in the body shop are similar to the requirements in the assembly shop, the main focus lies on the paint shop and the assembly shop.

The assembly line itself consists of several working stations defining individual requirements which must be fulfilled as good as possible. Moreover, the determined sequence of the last day must also be considered, because assembly lines operate mostly 24h a day. The goal of sequence planning is to find an arrangement of the cars along the assembly line, such that all constraints are satisfied and at the same time the costs are minimised.

2.2.1 Paint Shop Constraints

In the paint shop costs arise primarily when colours are changed. In this case the injectors must be cleaned, which is a very cost intensive process. Therefore, cars with the same colour should be grouped together to minimise the production costs. However, it is not possible to cluster all cars with the same colour, because the colour in the injector agglutinates and therefore the injectors must be cleaned after a predefined number of cars. Moreover, after cleaning the injector the colour must be changed, as reusing the same colour would lead to an imprecise workmanship.

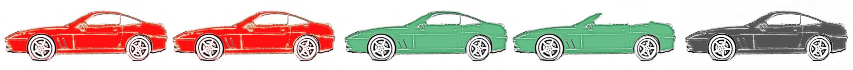

Fig. 2.1: Cars with same colour grouped together

2.2.2 Assembly Shop Constraints

In the assembly shop the components of the cars are installed. According to [6] there are two sorts of requirements, which must be fulfilled in the assembly shop.

Levelling Part Usage

On the assembly line different equipped cars are produced. These cars need different components and material in the production. Because only limited space is available in front of the assembly line, all components must be delivered when they are needed. On the other hand a large number of components is needed due to the wide variety of cars. An aggravating circumstance in this context is that the components are delivered by the suppliers just in time, i.e. all components are delivered when they are needed in production. As components are mostly delivered in complete cargo carries, a delivery of a single part can be very costly. Therefore, the goal is to smooth the component usage.

Smooth Workload

In the assembly shop different work stations exist. There the components are installed by the workers. Each component needs a different amount of time to be installed. This leads to variations in the workload of the stations along the assembly line. But, if the workers get too much workload, they make mistakes, which lead to additional costs for the company. Or, even worse, the production must be stopped until the component is installed. If, on the other side, the employees have not much to do, this will obviously lead to higher production costs. Therefore the goal is to smooth the workload on the different stations to avoid an over- and underloading of the stations.

2.2.3 Modelling the Real World Problem

To solve the sequence problem with the help of computers, models of the real world are needed. In literature three basic models for solving the sequencing problem can be found [7]:

- *Level Scheduling.* Level Scheduling tries to level part usage by defining "ideal" production rates. The goal is to find a sequence such that the gap between actual and ideal production rate is minimised.
- *Mixed-Model Sequencing.* This approach tries to avoid work overloads by a detailed modelling of the assembly line. Thereby installation times of components, the work flow and characteristics of the assembly line are taken into account.
- *Car Sequencing.* In contrast to Mixed-Model Sequencing, Car Sequencing tries to avoid work overloads in an implicit manner by defining capacity constraints for the working stations.

2.3 Car Sequencing Models

CarSP was proposed by Parello et al. [26] in 1986 for the automotive industry. The idea behind CarSP can be explained by the following example [12]:

Suppose that on an average 60% of all cars require component c. Furthermore, assume that the installation of component c takes x minutes. During these x minutes on an average 5 cars pass the work station, where component c is installed. Because 60% of all cars require component c, we can expect that 3 cars will need an installation of component c. Therefore 3 work stations are needed for the installation. Thus, the constraint for component c can be formulated as 3/5, meaning that no more than 3 cars are allowed to require component c in a consecutive interval of length 5.

CarSP was first formulated as a constraint satisfaction problem (CSP) [26]. The goal is to find a sequence of cars such that all constraints l_c/m_c are satisfied. Recently CarSP was also formulated as a combinatorial optimisation problem (COP) [17]. In contrast to the CSP formulation of CarSP the optimisation approach provides also a solution if constraints are violated. To transform the CSP, an objective function is introduced where a cost factor is added if a constraint is violated.

2.3.1 Counting the Violations

In literature different approaches for counting violations of constraints can be found. For an overview of different approaches see [7]. Currently the "sliding-window" approach is most frequently used.

"Sliding Window" Approach

The "sliding window" approach examines all possible "sliding windows" (subsequences of cars) with length m_c. Thereby one violation is counted, if the number of components of the investigated "sliding window" is bigger than the allowed l_c components [16, 17]. But this approach has the following two weak points [14]:

- Because a single car is covered by several "sliding windows", the scheduling of a single car can lead to more than one constraint violation in respect to one component. However, this double-counting does not match reality.
- The same subsequence of cars scheduled at the beginning or at the end of the assembly line might cause less violations than the same subsequence of cars scheduled in the middle of the assembly line. The reason is that "sliding windows" prior to the beginning and beyond the end are not considered.

Advanced "Sliding Window" Approach

ROADEF introduced an advanced "sliding window" approach. If a constraint is violated, the number of components exceeding the maximum allowed number of components l_c is counted. Furthermore, the previous production day and the next day (assuming no components occur) are also taken into account. Hence, only the problem of counting more than one violation caused by a single component persists.

Counting Components Causing a Violation

Due to the lack of the "sliding window" technique another approach for counting violations of the defined constraints was proposed in [14]. Instead of counting the number of violated "sliding windows", this approach counts the components which lead to the violation of a constraint. Like the "sliding window" approach all possible subsequences are examined. The difference is that the violation is only counted if the first position of the subsequence contains component c.

2.3.2 Renault Car Sequencing Problem

The French carmaker Renault proposed a CarSP for the ROADEF'05 challenge [34], which we will denote as Renault CarSP. An interesting property of Renault CarSP is that also constraints defined by the paint shop and the last production day are included. Furthermore, the defined instances for the ROADEF'05 challenge include up to 1300 cars. Therefore, Renault CarSP seems to be a quite realistic model of the real world problem. Renault CarSP uses the advanced "sliding window" approach as described in the previous subsection. Although it is not perfect, we will use the same counting method to allow a comparison of our results with the results of the ROADEF'05 challenge.

Formal Definition

The work within this thesis is based on Renault CarSP. Therefore, here a formal definition is given. The formalisation is based on [28] and will be used throughout the whole document.

The Renault CarSP is defined by a tupel (X, f), where

- X is a set of all possible arrangements of desired cars on the assembly line
- $\mathrm{f} : X \to \mathbb{R}^+$ is a cost function, which assigns each $x \in X$ a positive objective value

The goal is to find a solution $x \in X$, so that the function $\mathrm{f}(x)$ is minimised. Formal we can write:

$$\min_{x \in X} \mathrm{f}(x) \tag{2.1}$$

To define possible solutions x, a set of components C and a set of colours $F \subseteq C$ are given. K is a set of possible configurations and a configuration $k \in K$ has certain installed components and exactly one colour. The set of configurations can be defined as $K = \{k : k \subseteq C, |k \cap F| = 1\}$ with the restriction $k_i \neq k_j$ for all k_i, k_j with $i \neq j$. Furthermore, there is defined a demand δ_k for each configuration $k \in K$. The demand δ_k defines the number of cars which must be produced for each configuration $k \in K$ and each car χ_i is represented by a configuration $k \in K$.

Now, the set of all possible solutions X, called a solution space, can formally be described as:

$$X = \{(\chi_1, \ldots, \chi_n) : \{1, \ldots, n\} \to K \wedge |\{\chi_i : \chi_i = k\}| = \delta_k \ , \forall k \in K\} \tag{2.2}$$

Then the objective value of a solution $x \in X$ is calculated by the following function:

$$\mathrm{f}(x) = \sum_{i=1}^{n} \mathrm{costs}(i) \tag{2.3}$$

$$\mathrm{costs}(i) = \mathrm{change}(i) + \sum_{c \in C \setminus F} \mathrm{viol}(i, c) + \mathrm{hard}(i) \tag{2.4}$$

$$\mathrm{change}(i) = \begin{cases} \gamma_f \cdot \max\limits_{f \in F}(a_{f\chi_i} - e_{f1}) & \text{if } i = 1 \\ \gamma_f \cdot \max\limits_{f \in F}(a_{f\chi_i} - a_{f\chi_{i-1}}) & \text{otherwise} \end{cases} \tag{2.5}$$

$$\mathrm{viol}(i, c) = \begin{cases} \gamma_c \cdot \max(0, (\sum\limits_{j=i-m_c+1}^{i} a_{c\chi_j}) - l_c) & \text{if } i \geq m_c \\ \gamma_c \cdot \max(0, (\sum\limits_{j=1}^{i} a_{c\chi_j} + \sum\limits_{j=1}^{m_c-i} e_{cj}) - l_c) & \text{otherwise} \end{cases} \tag{2.6}$$

$$\mathrm{hard}(i) = \begin{cases} \gamma_{hard} \cdot \max\limits_{f \in F}(0, (\sum\limits_{j=i-m_f+1}^{i} a_{f\chi_j}) - l_f) & \text{if } i \geq m_f \\ \gamma_{hard} \cdot \max\limits_{f \in F}(0, (\sum\limits_{j=1}^{i} a_{f\chi_j} + \sum\limits_{j=1}^{m_f-i} e_{fj}) - l_f) & \text{otherwise} \end{cases} \tag{2.7}$$

Equation (2.5) shows the calculation of colour changes. For this purpose a cost factor $\gamma_f \in \{1, 10^3, 10^6\}$ is defined. Furthermore, a component vector $a_k^T = (a_{0k}, \ldots, a_{|C|k})$ exists. If configuration $k \in K$ contains component $c \in C$, the corresponding entry a_{ck} is set to 1, otherwise a_{ck} is set to 0. Remember that the set of components C also includes the set of colours F, thus a_{fk} is set to 1 if configuration $k \in K$ has colour $f \in F$.

Equation (2.6) shows the calculation of violations occurring at position i for component $c \in C$. For this purpose a cost factor $\gamma_c \in \{1, 10^3, 10^6\} \setminus \{\gamma_f\}$ for each component $c \in C$ exists. The constraints for component $c \in C$ are defined by values $m_c, l_c \in \mathbb{N}$, meaning that no more than l_c cars are allowed in a consecutive sequence of m_c cars.

To take the last production day into account the determined sequence of the last day is stored by constants $e_{ci} \in \{0,1\}$ for all $c \in C$ and $i = 1, \ldots, m_c - 1$. Thereby e_{ci} is set to 1 if the i-th last car requires component $c \in C$, otherwise e_{ci} is set to 0.

Finally, Equation (2.7) shows the calculation of the hard constraints defined by the paint shop. Thus a constant $s \in \mathbb{N}$ exists, meaning that maximal s cars having the same colour $f \in F$ are allowed to be scheduled in a consecutive interval. For calculation it is transformed to a rule l_f/m_f, where l_f is equal to s and m_f is equal to $s+1$ for all colors $f \in F$. As solutions $x \in X$ can be found violating the defined hard constraints, an additional penalisation constant $\gamma_{hard} \in \mathbb{N}$ is defined. This constant $\gamma_{hard} \in \mathbb{N}$ should ensure that in principle not allowed solutions are discarded.

2.4 Computational Complexity

It can be shown that CarSP belongs to the class of $\mathcal{N}P$-hard problems [23]. Under the assumption that $\mathcal{N}P \neq \mathcal{P}$, no polynomial time algorithm exists. Therefore, in worst case exponential time is needed to search through the whole search space X. Thus, exact approaches such as Integer Linear Programming and Constraint Programming are not capable of solving real-life instances [13].

An indicator of the complexity of an instance of CarSP is the size of the search space size(X), which can be calculated by:

$$\text{size}(X) = \frac{n!}{\prod\limits_{k \in K} \delta_k!} \tag{2.8}$$

Furthermore, the utilisation rates utilRate(c) of the components $c \in C \setminus F$ can be used as an indicator of the complexity of CarSP instances [32]. The utilisation rate is defined as the ratio of the whole number of cars requiring component c and the maximum number of cars that can be sequenced without violating the defined constraints by the component. Formal we can write:

$$\text{utilRate}(c) = \frac{\left(\sum_{k \in K} \delta_k \cdot a_{kc}\right) \cdot m_c}{n \cdot l_c} \;, \forall c \in C \setminus F \tag{2.9}$$

If the utilisation rate utilRate(c) of a component c is greater than 1, a rule l_c/m_c will be violated in any case, i.e. the capacity of the station installing component c will be exceeded in any case. Therefore, only with utilisation rates lower than 1 it might be possible to find a sequence without violating any constraints.

3 Related Work

This chapter presents the most relevant related work. Section 3.1 is devoted to different approaches that have already been made for CarSP. In Section 3.2 different parallelisation strategies for VNS are presented. Finally, in Section 3.3 the topic of VNS with dynamic neighbourhood-ordering is presented.

3.1 Car Sequencing Problem

In literature a great deal of work on topics related to the CarSP can be found. For this reason only a summary of the different approaches is presented here. For a state of the art overview the reader is refered to [34].

3.1.1 Exact Approaches

On the side of exact approaches, those based on Constraint Programming (CP) [10] and Integer Linear Programming (ILP) [11] have to be mentioned. Exact algorithms find a provable, optimal solution for a given instance. To solve the CarSP with CP or ILP a suitable model has to be defined. For this purpose the constraints defined by the CarSP must be mapped to a CP/ILP-model. In addition, constraints must be defined, so that exactly one car is assigned to each position and all cars are used in a solution. If the CarSP is defined as a combinatorial optimisation problem, soft-constraints can be violated, but a cost-factor for each violation is added to an optimisation function. Then a solution is a sequence of cars, where all hard-constraints are satisfied and the optimisation function is minimised [27, 28].

3.1.2 Heuristic Approaches

Contrarily to exact approaches, heuristics sacrifice guaranteed solution quality to speeding up the solution process.

Greedy Heuristics

The idea behind greedy heuristics is to start from an empty solution and add solution components until the solution is complete. For CarSP that means that we start with an empty sequence and add the next car to the existing sequence step by step. Which car is selected next, is determined by the heuristic function. For CarSP the heuristic function should select a car in this way that as few as possible new constraint violations are introduced. A study of six different greedy heuristics for the CarSP can be found in [17].

Local Search Based Approaches

The basic idea of Local Search (LS) based approaches is to improve the current solution iteratively. First an initial solution is generated randomly or a greedy heuristic is used. Afterwards the solution is improved by searching after better solutions in a neighbourhood. A neighbourhood defines a set of solutions which we can be derived from a given solution. Neighbourhoods are often generated by performing small changes in the current solution, for example by swapping two cars. A Local Search based approach can be found in [30, 17]. Thereby, different neighbourhoods are used. These neighbourhoods are selected randomly and afterwards a single iteration is performed in this neighbourhood.

Ant Colony Optimisation

Ant Colony Optimisation (ACO) is a probabilistic algorithm inspired from natural ants. ACO tries to find a minimum cost path in a given graph. Therefore, the CarSP must be transformed into a graph. Similar to natural ants, artificial ants lay pheromone trails on the path they use. The path chosen by an individual ant is based on a probabilistic decision, depending on the pheromone concentration laid on the graph by the whole ant colony. An example of an ACO for the CarSP can be found in [33].

Genetic Algorithms

A genetic algorithm is a population based method, i.e. there is a pool of candidate solutions which is considered by the algorithm. A subset of these solutions called parents is selected to be combined for generating new offspring. To avoid pure random search, better solutions are selected with a higher probability. To bring in new genes mutation is performed, where parts of the solution are randomly disturbed. These steps—selection, crossover and mutation—are repeated until some stopping criterion is met. A genetic algorithm for CarSP can be found in [8, 35].

3.1.3 Hybrid Approaches

Hybrid approaches try to combine the advantages of heuristic and exact algorithms. Heuristic approaches have an advantage in run time, whereas exact approaches are able to provide guarantees on solution quality. In [27, 28] a Variable Neighbourhood Search was combined with an ILP based approach. Thereby, neighbourhood structures examined via ILP techniques are included in the classical VNS.

3.2 Parallel Variable Neighbourhood Search

Different parallelisation strategies for VNS can be found in literature. Here, we present a short overview of these approaches. For a detailed description of possible parallelisation strategies see Section 4.3.

3.2.1 Direct Parallelisation

García-López et al. [15] proposed two simple parallelisation strategies for VNS. The first approach simply parallelises the Local Search by decomposition of the search space of a single neighbourhood. The second presented approach is a classical multistart approach, which is performed in parallel.

3.2.2 Centralised Parallelisation

In literature two more advanced parallelisation strategies for VNS can be found. The key features are cooperation between the different processes and a central coordination. García-López et al. [15] proposed a synchronous, cooperative parallel VNS. In this approach a Shaking and a Local Search are performed in parallel. After all Local Searches have finished, the best value is determined. With this best solution the calculation is continued.

Crainic et al. [9] presented an asynchronous, cooperative parallel VNS. Instead of determining the best solution after each step (Shaking + Local search), the cooperation is done in an asynchronous fashion. At the beginning different VNS are performed in parallel by the slaves. If a slave can achieve no further improvement in a neighbourhood, it sends the current solution to the master. The master determines the best until yet received result and sends it back to the slave. Now the slave continues with this solution and with the current neighbourhood.

3.2.3 Decentralised Parallelisation

Sevkli and Aydin [31] proposed two decentralised parallelisation strategies for VNS. The first approach uses a unidirective ring-topology, where the outcome of one node is passed to the next node in the ring. No further selection rules exist, i.e. the current solution is replaced by the result of the predecessor node. Finally, they proposed another decentralised approach using a so-called mesh-topology. Thereby, each process receives two solutions from two other processes. Afterwards, each process continues the calculation with the best of all locally known solutions.

3.3 Variable Neighbourhood Search with Dynamic Neighbourhood-Ordering

The research in the field of VNS with dynamic neighbourhood-ordering is still limited. There are only two approaches which try to rearrange the neighbourhood-order dynamically during the VNS. The first approach in this direction is from Puchinger and Raidl [29]. The order of neighbourhoods is determined dynamically based on relaxation values computed for all neighbourhoods. In his work, Puchinger examines neighbourhoods using ILP based-methods. The relaxations are used as an improvement indicator. During the VNS the neighbourhood-order is recalculated after each improvement of the best neighbourhood.

A more general approach is from Hu and Raidl [22]. The presented algorithm uses information from past behaviour to determine a new order of the neighbourhoods. The neighbourhood-order depends on the execution time and on the success of the neighbourhoods (current solution can be improved). In contrast to the approach from Puchinger the neighbourhoods are not rearranged after each improvement. The rearrangement is only performed if a neighbourhood gets better/worse than the existing best/worst neighbourhood. Between the adaption of the neighbourhood-ordering a static order is used.

4 The Basics of Parallel Variable Neighbourhood Search

Many practical instances of CarSP are too large to be solved with exact algorithms. Therefore, heuristics, which give up finding the optimal solution for an improvement of run time, are applied. But even with heuristics it may take a long time until a suitable solution is provided. Parallelisation of (meta)heuristics seems to be a promising approach to overcome this issue.

4.1 Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is a metaheuristic developed by Hansen and Mladenović and is based on the idea of changing the neighbourhood structure systematically [20, 21]. Metaheuristics try to guide underlying heuristics such that moderate computing times are used and (near) optimal solutions are provided. The main idea of VNS is to explore different neighbourhood structures systematically, with the goal to escape local minima.

4.1.1 Basic Local Search

Starting from an initial solution, Local Search improves the current solution x iteratively by searching for better solutions x' in the neighbourhood $\mathcal{N}(x)$ of solution x. Under a neighbourhood $\mathcal{N}(x)$ one understands a set of solutions similar to x. The definition of neighbourhoods is often done implicitly based on the application of one or more specific moves to a current solution x. Identifying the next solution x' to be used within LS can be done according to different functions. Using a best improvement

Algorithm 1 Basic Local Search

```
Input: initial solution $x$
1: repeat
2:    choose $x' \in \mathcal{N}(x)$
3:    if $f(x') < f(x)$ then
4:       $x \leftarrow x'$
5:    end if
6: until termination condition
```

strategy, among all candidate solutions in $\mathcal{N}(x)$ that with the greatest improvement is selected. In contrast, a next improvement strategy returns the first found solution improving the current solution x. Algorithm 1 presents the pseudocode of Basic Local Search. Mostly Local Search is applied until no further improvement can be achieved. In this case a local minimum with respect to the used neighbourhood structure has been reached.

4.1.2 Variable Neighbourhood Descent

Based on the idea that a local minimum with respect to $\mathcal{N}_i$ needs not necessarily be a local optimum w.r.t. $\mathcal{N}_i$, $i \neq j$, *Variable Neighbourhood Descent* (VND) changes systematically between different predefined neighbourhood structures. See Figure 4.1 for an illustration of this idea. At point x it is not possible to escape from the local minimum by using the neighbourhood $\mathcal{N}_1(x)$. But when examining neighbourhood $\mathcal{N}_2(x)$, it is possible to escape from the local minimum and reach solution x' with $\mathrm{f}(x') < \mathrm{f}(x)$.

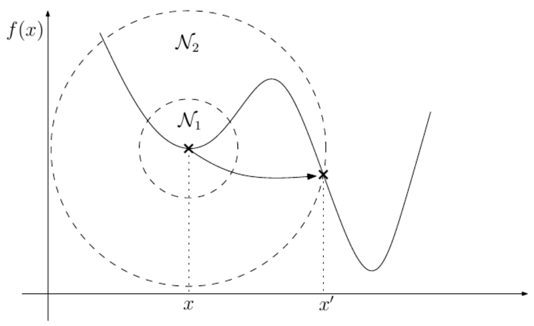

Fig. 4.1: Basic idea of VND

The pseudocode of VND is presented in Algorithm 2. VND starts with an initial solution x and a given neighbourhood-order $(\mathcal{N}_1, \ldots, \mathcal{N}_{max})$. In the first step a Local Search using neighbourhood structure $\mathcal{N}_1$ is performed until no further improvement can be achieved. If a local optimum has been reached, VND continues by examining neighbourhood structures $\mathcal{N}_2, \ldots, \mathcal{N}_{max}$. If an improvement can be achieved in any of these neighbourhoods, VND swaps back to the first neighbourhood structure. Otherwise it will terminate. In this case a local minimum has been reached with respect to all neighbourhood structures.

The basic principle of VND is simple and only a few parameters have to be adjusted, but nevertheless the following questions should be kept in mind [21]:

1. What is the complexity of the used neighbourhoods? Can the selected neighbourhoods provide a solution in admissible time?

Algorithm 2 Variable Neighbourhood Descent (VND)

```
Input: initial solution $x$ and a neighbourhood-order $(\mathcal{N}_1, \dots, \mathcal{N}_{max})$
 1: $k \leftarrow 1$
 2: repeat
 3:     choose $x' \in \mathcal{N}_k$
 4:     if $\mathrm{f}(x') < \mathrm{f}(x)$ then
 5:         $x \leftarrow x'$
 6:         $k \leftarrow 1$
 7:     else
 8:         $k \leftarrow k + 1$
 9:     end if
10: until $l = l_{max}$
```

2. Are the used neighbourhoods adequate to explore the region containing x completely? Do appropriate neighbourhoods to escape from narrow valleys exist?

3. What is the best neighbourhood order $(\mathcal{N}_1, \dots, \mathcal{N}_{max})$?

The third question mainly addresses the efficiency of the used neighbourhoods, i.e. the relation of computation time and the solution improvement of a neighbourhood. Currently most frequently used is the order of neighbourhoods by increasing complexity, often corresponding to the size $|\mathcal{N}(x)|$ of the neighbourhood. However, sometimes this rule of thumb cannot be used, e.g. when the times necessary for examining the neighbourhoods are similar to each other or when the size of a neighbourhood cannot even be determined deterministically. Furthermore, this rule of thumb can be misleading in the case when a neighbourhood with a greater complexity has a better efficiency. Finally, a static neighbourhood-order during the whole run of VND will probably not be ideal [29, 22].

4.1.3 Shaking

Although the usage of different neighbourhood structures reduces the probability to be trapped in a local minimum, it is still possible. Based on the fact that local minima are relatively close to each other, a random Shaking is performed, where only a few variables are changed randomly. The goal of Shaking is to alter only as many positions as necessary to escape from the valley containing the local minimum [21].

4.1.4 General VNS Scheme

General Variable Neighbourhood Search scheme combines VND and Shaking into a general framework, on the one hand to escape from local minima and on the other hand to escape from the valleys containing them. Given are a neighbourhood-order

$(\mathcal{N}_1, \ldots, \mathcal{N}_{max})$ and an initial solution x. First a shaking step is performed and afterwards VND is applied to the current solution until no further improvement with respect to all defined neighbourhoods can be achieved. These two steps are subsequently iterated whereas the neighbourhood used for the shaking phase is enlarged systematically if no improvement could be achieved during the last local search phase. For the pseudo code see Algorithm 3.

Algorithm 3 General Variable Neighbourhood Search scheme

```
Input: initial solution $x$ and a neighbourhood-order $(\mathcal{N}_1, \ldots, \mathcal{N}_{max})$
1: repeat
2:   $l \leftarrow 1$
3:   repeat
4:     $x' \leftarrow$ Shaking$(l, x)$ {choose a random solution $x' \in \mathcal{N}_l$}
5:     $x' \leftarrow VND(x')$
6:     if $\mathrm{f}(x') < \mathrm{f}(x)$ then
7:       $x \leftarrow x'$
8:       $l \leftarrow 1$
9:     else
10:      $l \leftarrow l + 1$
11:    end if
12:  until $l = l_{max}$
13: until stopping conditions are met
```

4.2 Parallel Computing

The primary objective of parallel computing is to reduce the wall clock time for solving a given problem. Of course, the additional computing power can also be used to solve more complex problems in the same time [1]. At first glance this objective seems to be clear, but on closer inspection the following different detailed reasons for parallelisation can be identified [3]:

- *Absolute Speed.* The goal is to solve a given problem faster than it is possible with the best algorithm running on the fastest computer.
- *Relative Speed and Cost.* Often it is only important to improve the wall clock time subject to constraints such as cost or power consumption. The idea is to perform the computation with a large amount of relatively inexpensive but slow processors faster than using a cost-equivalent sequential computer.
- *Scalable Computing.* The objective is to develop a flexible design which offers the possibility to increase the performance by adding additional processors.

4.2.1 Parallel Architectures and Their Programming Models

The main idea of parallel computing is to divide computation into smaller computation parts which can be executed simultaneously on multiple processors. As in general the different computation parts are not independent from each other, a coordinated cooperation is necessary. The coordinated cooperation is done by information exchange and synchronisation. To understand how the information exchange and synchronisation can be done, it is important to have a look on the underlying parallel computer. According to [36] a *parallel computer*

> "[...] is either a single computer with multiple internal processors or multiple computers interconnected to form a coherent high-performance computing platform."

A frequently used classification of parallel computers is based on the memory organisation which has a significant influence on the parallel programming model.

Multiprocessor System - Shared Memory

The common attribute of shared memory multiprocessor systems is that all processors have access to a shared memory. Hence, the information exchange can be done by using global variables which all processes can access. Therefore the programming model is called *shared memory programming*. To avoid concurrent access to a shared variable, which can cause data inconsistency, mutual exclusion must be performed. Mutual exclusion algorithms guarantee that only one process has access to a common resource by entering critical sections of a program. Synchronisation between processes to achieve a certain sequence of actions is also done by using the shared memory. In this architecture the parallelisation must be done explicitly, whereas the communication is done implicitly.

Multicomputer - Distributed Memory

In a distributed memory multicomputer system each processor has only access to its own local memory. In this architecture the information exchange is done by sending messages explicitly to other processes. The underlying programming model is therefore called *message-passing programming*. In a message-passing system no mutual exclusion is necessary because the parallelisation is done implicitly. However, in contrast to shared memory programming the communication must be done explicitly.

4.2.2 Measuring Performance

For evaluating parallel algorithms different performance metrics exist. Nevertheless, all of them are based on the execution time of a program.

Execution Time

In sequential algorithms the number of iterations or evaluations of a program is often used as a performance metric. This metric is prefered by many researchers because implementation details, used software and hardware do not falsify the results and therefore a meaningful comparison is possible. But this approach is not always suitable. For example, when comparing the effectiveness of different neighbourhoods, the time to find a solution is an important factor and therefore cannot be neglected. Nevertheless, also the number of evaluations should be reported to allow a comparison between different algorithms [1].

The CPU time is also widely used as a performance metric for sequential programs. It is the time a program is running on the CPU and excludes all system overhead activities. Because the goal of parallelisation is the reduction of the wall clock time for solving a problem, all overheads introduced by the parallelisation must be included. Therefore, the *parallel execution time* T_p of a parallel program running on p processors is defined as the wall clock time from starting a parallel program until all processes have finished the execution. The *sequential execution time* T_s of a sequential program is defined as the wall clock time between start and end of the program [1, 18].

Basically two stopping conditions for metaheuristics exist. First, an a priori defined effort, e.g. a given amount of available wall clock computation time or a predefined number of available iterations, can be used as a stopping condition and secondly a metaheuristic can be terminated, if a predefined solution quality has been reached. When using an a priori defined effort, no meaningful conclusion about the speedup is possible, because we cannot conclude about time improvements based on the achieved solution quality. Only when using a predefined solution quality as a stopping condition, statements about the speedup of the execution time are possible. Furthermore, when stochastic algorithms are used, multiple independent runs must be performed to obtain meaningful values for the execution times [1].

Costs

The costs of a parallel program running on p processors are defined as:

$$C_p = T_p \cdot p \tag{4.1}$$

The costs are a measure of the overall time spend by all processors running in a parallel program.

Parallel Overhead

Theoretically, if using p processors, we can expect a p-fold faster parallel execution time T_p compared to the sequential execution time T_s. But in a typical parallel program this will not be the case, because of the additional overhead introduced by the parallelisation. According to [18] the main sources therefore are:

- *Interprocess Interaction.* The time needed to exchange information between processes.
- *Idling.* In a parallel program processes may run out of doing useful work and get idle. Reasons therefore are:
  - *Serial components.* Serial components cannot be parallelised and therefore they can only be performed by a single processor.
  - *Load Imbalance.* Load imbalance occurs when not each process has the same amount of work to do and therefore some processes must wait.
  - *Synchronisation.* Since in a parallel program the computation is distributed on more than one process, it might occur that a process must wait for the termination of an other process.
- *Excess Computation.* In most cases it is not possible or at least difficult to parallelise a given high developed, complex sequential algorithm. Therefore a poorer sequential algorithm must often be used as a starting point of the parallelisation. An additional frequent reason for excess of computation is that in a parallel algorithm some computations are performed multiple times to avoid additional interprocess interactions and idling.

All of these overheads can be summed up to a *total overhead*:

$$T_o = C_p - T_s = p \cdot T_p - T_s \tag{4.2}$$

Speedup

The most common performance metric for parallel programs is the speedup and it is defined as:

$$S_p = \frac{T_s}{T_p} \tag{4.3}$$

The speedup defines how much faster a parallel program runs on p processors in comparison to the sequential program. In an ideal parallelisation we can achieve a speedup of $S_p = p$ which is called *linear speedup*. But in most cases only a *sublinear speedup* with $S_p < p$ can be achieved due to the parallelisation overhead. A *super linear speedup* can be caused by hardware effects, e.g. when in the parallel program all data fit into the cache. But as any parallel algorithm can be simulated on a single processor, a super linear speedup suggests that the original sequential algorithm was not optimal [36].

Determining the Sequential Execution Time

When using the speedup metric, different assumptions about the sequential execution time T_s are possible [3]. The most unambiguous definition is *absolute speedup*. Absolute speedup compares the fastest sequential program on the fastest sequential computer with the parallel program running on p processors of a parallel computer. Two practical problems arise when using absolute speedup. The first problem is that it is mostly not possible to have access to the fastest sequential computer. The second problem is finding the best sequential algorithm, especially when algorithms have parameters which can be tuned. Consequently all known algorithms must be run on different computers to find out the best parameters which cause the shortest sequential execution time. Because of these problems most researchers use relative speedup [1].

In *relative speedup* the time for running the parallel program on one processor is compared with the parallel program running on p processors. Consider that the relative speedup can only be used if the parallel algorithm running on one processor is faster than other sequential algorithms. That means that the "poorer" parallel algorithm must be faster than the sequential algorithm, which was used as a starting point of the parallelisation. Otherwise this would lead to misleading statements about the achieved speedup.

Finally, *speedup* compares the time of the fastest sequential program for solving a problem with the parallel program running on p processors. Here, also the best known sequential algorithm is used but the usage of the best sequential computer is disregarded.

Efficiency

Efficiency is the ratio of the sequential execution time T_s to the costs of the parallel program $C_p = p \cdot T_p$. If substituting T_p by T_p from Equation (4.2), we get the following equation for the efficiency:

$$E = \frac{T_s}{T_p \cdot p} = \frac{1}{1 + \frac{T_o}{T_s}} \tag{4.4}$$

Efficiency is the fraction of time during which a parallel algorithm is doing meaningful work. Only in an ideal system with a parallel overhead $T_o = 0$ an efficiency of 100% can be achieved.

4.2.3 Maximum Speedup

Theoretically, if adding additional processors, an improvement of the speedup can be expected. Assuming a constant serial fraction f of computation which cannot be parallelised, then $f \cdot T_s$ of the time can only be done by a single processor. If the rest of the computation can be parallelised perfectly, the computation time $(1 - f)T_s$ can be divided by p processors. Then the upper limit for the possible speedup, excluding all other parallelisation overheads, is given by:

$$S_p = \frac{T_s}{f \cdot T_s + \frac{(1-f)T_s}{p}} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f} \tag{4.5}$$

Equation (4.5) is also known as the *Amdahl's law* [2]. The main assumption of Amdahl's law is that a fixed part of serial work must be done in a parallel algorithm. In the year 1967 Amdahl argued against massive parallelism based on his formulated law.

It took 21 years until Gustafson presented massive parallel algorithms running on a 1024-processor system with a speedup above 1000 [19]. To achieve this speedup the problem size is increased to minimise the serial fraction f. If, as in many cases possible, the serial fraction decreases, while the problem size is increased, the assumption of a fixed serial fraction can be discarded. Therefore, contrary to a fixed serial fraction f as in the Amdahl's law, we can assume that the parallel execution time is constant. If the parallel execution time is expressed by $f \cdot T_s + (1-f)T_s$ and for mathematically simplification it is assumed to be 1, we get

$$S_p = \frac{f \cdot T_s + (1-f)T_s \cdot p}{f \cdot T_s + (1-f)T_s} = f \cdot T_s + (1-f)T_s \cdot p = p + (1-p)f \tag{4.6}$$

for the possible speedup. Equation (4.6) is known as the *Gustafson's law* or *scaled speedup* [19]. In this case the possible speedup is not bounded.

4.2.4 Scalability

The conclusion from Amdahl's law is that, when using a constant problem size, the efficiency goes down when increasing the number of processors. And from Gustafson's law we can conclude that, when increasing the problem size and letting the number of processors constant, the efficiency improves. Both observations can be described by *scalability.* Scalability is the ability of a parallel algorithm to use additional processors efficiently. The main question of scalability is how much must we increase the problem size so that we can use the additional processors at the same efficiency. An analytical metric is the isoefficiency metric of scalability [18].

Isoefficiency Metric of Scalability

In sequential algorithms the problem size is mainly defined as the input size of the problem. A drawback of this definition is that depending on the used algorithm a different number of operations will be performed for the same problem. Therefore, the *problem size* W is commonly defined as the number of basic operations of the best sequential algorithm to solve a given problem. Assuming that a basic operation takes unit time and all other constants are normalised, the problem size W is equal to the sequential execution time T_s. Now, the total overhead T_o can be defined by an overhead function $T_o(W, p)$. If replacing the sequential execution time T_s by the problem size W and the total overhead T_o by the overhead function $T_o(W, p)$ in Equation (4.4) the efficiency can be rewritten as:

$$E = \frac{1}{1 + \frac{T_o(W,p)}{W}} \tag{4.7}$$

To maintain a fixed efficiency $T_o(W, p)/W$ must be held constant. If transforming Equation (4.7), we get for the problem size

$$W = KT_o(W, p) \tag{4.8}$$

where $K = \frac{E}{1-E}$ is a constant depending on the efficiency to be maintained. The expression (4.8) is called the *isoefficiency function*. The isoefficiency function determines the problem size W which is needed to sustain a fixed efficiency when using p processors. High values of the isoefficiency function indicate poor scalability, whereas low values indicate high scalability. For unscalable systems the isoefficiency function does not exist, because the efficiency cannot be kept constant when p increases, no matter how big the problem size W is.

Degree of Concurrency

The *degree of concurrency* $C(W)$ is the number of tasks that can be performed simultaneously. If the number of processors p exceeds the degree of concurrency $C(W)$, the additional processors cannot be used efficiently and the efficiency goes down. In this case $p - C(W)$ processors cannot be employed. As in a scalable system the efficiency can be kept constant, also the degree of concurrency $C(W)$ can be increased by increasing the problem size W. If a fixed degree of concurrency $C(W)$ independent of the problem size W is used, the parallel algorithm is in any case unscalable. The reason is that the efficiency cannot be held constant when the number of processors exceeds $C(W)$.

4.3 Parallel Variable Neighbourhood Search

In this section a classification of parallelisation strategies for VNS based on various characteristics is presented. This classification is based on a classification of parallel metaheuristics presented in [9], which is extended by additional characteristics.

4.3.1 Direct versus Indirect

In a direct parallelisation the sequential VNS is parallelised by running the original VNS on multiple processors. Examples of direct approaches are parallelising the Local Search by decomposition of the search space or running a multistart VNS in parallel. Because a sequential VNS is optimised to run efficiently on a sequential computer, it might be better to develop a "poorer" sequential algorithm for parallelisation. Therefore, in an indirect parallelisation a new algorithm based on the principles of the classical VNS which can be parallelised efficiently is developed first. Although, in general additional computation is introduced, an increased performance compared to a direct approach can be expected. An obvious approach is exploring different neighbourhoods in parallel.

4.3.2 Single-Trajectory versus Multi-Trajectory

When using a single-trajectory strategy only a single trajectory of the search space is walked through. A possible strategy is a decomposition of the search space such that each processor searches through a part of the whole search space. After each process has finished, the best solution is determined and the search is continued with it. Another approach is searching through different neighbourhoods and selecting the best found solution afterwards. In a multi-trajectory parallelisation strategy multiple search trajectories are searched through in parallel. Figure 4.2 illustrates the main difference between single-trajectory and multi-trajectory strategy. Furthermore multi-trajectory strategies can be classified whether information among the different search trajectories is exchanged or not.

$$\begin{tikzcd}
 & x_0 \arrow[dl] \arrow[d] \arrow[dr] & \\
x_{1_1} \arrow[dr] & x_{1_2} \arrow[d] & x_{1_3} \arrow[dl] \\
 & x_1 &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
x_{0_1} \arrow[d] & x_{0_2} \arrow[d] & x_{0_3} \arrow[d] \\
x_{1_1} \arrow[d] & x_{1_2} \arrow[d] & x_{1_3} \arrow[d] \\
x_{2_1} & x_{2_2} & x_{2_3}
\end{tikzcd}$$

(b)

Fig. 4.2: Example of single-trajectory (a) and mulit-trajectory (b) search

Independent versus Cooperative

In an independent search strategy no information is exchanged and all different search trajectories are performed completely independent of each other. A straightforward approach is to perform multiple, independent runs of VNS and collect the results at the end. Although, an independent search strategy can be parallelised perfectly, the disadvantage of considering worse solutions or neighbourhoods, which do not contribute to the solution quality, exists. Hence, in a cooperative strategy information among different search trajectories is exchanged to adapt the future search process. Thus, a more detailed investigation of promising neighbourhoods or solutions, which should lead to a speedup of the search process, is possible.

4.3.3 Low-Level versus High-Level

When using a low-level parallelisation strategy, we basically try to parallelise low-level computation such as loops or computation tasks of a single iteration such as single moves of a neighbourhood. Contrarily, in a high-level parallelisation strategy the focus lies on parallelising single parts of VNS. For example, the Local Search can be replicated and run with different neighbourhoods in parallel.

4.3.4 Central versus Distributed

The basic characteristic of central approaches is the existence of a central point where all pieces of information flow together. As in general also the whole information is considered, decisions on global information are possible. A common approach is a master-slave model, where mostly one master process generates work and allocates it to the slaves. Thereby, the master makes the control decisions of the algorithm and the slaves execute the assigned work. In contrast to a central parallelisation approach, control decisions are mostly distributed over all involved processors in a distributed approach. Thus, no central control point exists and almost always only a part of the global information is considered for future decisions. If a process considers only neighbour processes for future decisions, a possible decrease of the synchronisation overhead can be expected. Furthermore, the disadvantage of a single point of failure or the problem that a single point gets a performance bottleneck do not arise. An example for a distributed approach is to use generated solutions of neighbour processes and decide on basis of this local information with which solution to continue.

4.3.5 Synchronous versus Asynchronous

In synchronous computation a process waits at predefined points until all other processes have finished computation, thereby becoming synchronised. In a parallel program process synchronisation is done to guarantee that the same sequence of actions is performed as in the sequential algorithm. A drawback of synchronous computation is that additional synchronisation overhead is introduced. Asynchronous computation omits synchronisation to avoid synchronisation overhead. However, developing an asynchronous algorithm is tricky, because any possible sequence of actions should lead to the desired results. An example of an asynchronous approach is using the so far best found solution in the case a local minimum has been achieved.

5 Advanced Parallel Variable Neighbourhood Search

In this Chapter we present a new parallelisation strategy for VNS called *Parallel Efficiency Guided VNS* (PEGVNS). This approach exploits observed efficiencies of neighbourhoods to guide the search process and is based on a special for parallelisation adapted VNS called *Time Restricted Randomised VNS* (TRRVNS), which uses a random neighbourhood selection, a Time Restricted Local Search and a next improvement strategy with randomised examination order.

5.1 Efficiency of Neighbourhoods

When applying VND, the neighbourhoods are mostly ordered by increasing complexity and afterwards this static neighbourhood-order is applied during the whole search process. But as stated in Section 3, this is in general not optimal. If giving up this standard practice, any possible neighbourhood could be selected in each iteration. Such a sequence of neighbourhoods to be used is denoted as a neighbourhood-usage. In this section we give a formal definition of an efficient neighbourhood-usage. This formulation is based on the efficiency of a neighbourhood evaluation presented in [4] which is extended to an efficient neighbourhood-usage. Table 5.1 shows the notations used for the formal definition.

Table 5.1: Notations used for describing an efficient neighbourhood-usage

x^i	...	solution in i-th iteration
x^i_k	...	solution in i-th iteration generated by $\mathcal{N}_k$
$\mathcal{N}^i$	...	neighbourhood structure used in the i-th iteration
$\mathcal{N}^i_k$	...	neighbourhood structure $\mathcal{N}_k$ used in the i-th iteration
e	...	overall number of performed iterations until termination
$(\mathcal{N}^1, \ldots, \mathcal{N}^e)$	...	neighbourhood-usage
$(x^0, \ldots, x^e)$	...	search trajectory
$\mathfrak{N}$	...	a set of neighbourhoods
$\mu_{k,j}$	...	move j in $\mathcal{N}_k$
$\mathcal{M}_k \subseteq \mathcal{N}_k$	...	a subset of moves $\mu_{k,j}$
$\mathcal{S}(\mathcal{N}_k(x^i))$	...	step function to be used in Local Search

For this formulation, we assume that the search trajectory also contains solutions x^i where $x^i = x^{i-1}$, i.e. the search trajectory contains solutions x^i where no improved solution is returned by $\mathcal{S}(\mathcal{N}^i(x^{i-1}))$. Furthermore, we assume a minimisation problem and a deterministic improvement strategy $\mathcal{S}$, i.e. applying $\mathcal{S}$ on a solution x^i will always return the same solution x^{i+1}.

5.1.1 Improvement per Iteration

The so-called value *improvement per iteration* $\mathcal{I}$ is the improvement of the objective value during one iteration and is defined as:

$$\mathcal{I}(\mathcal{N}^i, x^{i-1}) = \mathrm{f}(\mathcal{S}(\mathcal{N}^i(x^{i-1}))) - \mathrm{f}(x^{i-1}) \tag{5.1}$$

When using a best improvement strategy, the maximum value of $\mathcal{I}$ can always be achieved, because the whole neighbourhood is examined. Whereas, using a next improvement strategy, mostly only an improvement less than the theoretically possible improvement will be achieved during one local search iteration. A further influence on $\mathcal{I}$ has the chosen neighbourhood structure $\mathcal{N}^i$. With appropriate and larger neighbourhoods we can expect larger improvements. Furthermore, the current solution x^{i-1}, which depends on the previous applied neighbourhood structures $(\mathcal{N}^0, \ldots, \mathcal{N}^{i-1})$, has a major influence. Thus, when maximising $\mathcal{I}$, a best improvement strategy with large neighbourhoods is expected to be most promising.

5.1.2 Time per Iteration

The value *time per iteration* $\mathcal{T}$ is the time for examining a neighbourhood until a solution is returned, according to the step function $\mathcal{S}$, and it is defined as:

$$\mathcal{T}(\mathcal{N}_k^i, x^{i-1}) = \sum_{\mu_{k,j} \in \mathcal{M}_k^{\mathcal{S}}} \mathcal{T}(\mu_{k,j}, x^{i-1}) \tag{5.2}$$

Here $\mathcal{T}(\mu_{k,j}, x^{i-1})$ is the time for evaluating a single move and depends on the chosen neighbourhood structure $\mathcal{N}_k^i$ as well as on implementation details and on the used hardware. The improvement strategy determines the size of the investigated set of moves $\mathcal{M}_k^{\mathcal{S}}$. When using a best improvement strategy, $\mathcal{T}$ will usually be higher compared with a next improvement strategy. To minimise $\mathcal{T}$, a next improvement strategy with neighbourhoods where moves can be evaluated quickly should be used.

5.1.3 Efficient Neighbourhood-Usage

If we can choose among a set of neighbourhoods $\mathfrak{N}$, the best neighbourhood is neither the neighbourhood with maximum $\mathcal{I}$ nor the neighbourhood with minimum $\mathcal{T}$. Both points considered in isolation are not appropriate to make statements about the quality of a neighbourhood, only the relation between them is meaningful. Thereby, a tradeoff between $\mathcal{I}$ and $\mathcal{T}$ must be found. On the one side evaluating a larger subset of moves increases the chance of identifying moves with larger improvements, but on the other side it increases $\mathcal{T}$. Therefore, we define efficiency of a neighbourhood in i-th generation using neighbourhood structure $\mathcal{N}^i$ and solution x^{i-1} as:

$$\mathcal{E}((\mathcal{N}^i), x^{i-1}) = \frac{\mathcal{I}(\mathcal{N}^i, x^{i-1})}{\mathcal{T}(\mathcal{N}^i, x^{i-1})}, \text{ with } i > 0 \tag{5.3}$$

Because a selection of the most efficient neighbourhood in the i-th iteration must not necessarily lead to an optimal neighbourhood-usage, we are interested in the efficiency of a neighbourhood-usage for a given amount of computation time $tTime$. Therefore, we first define the efficiency of a neighbourhood-usage between iteration j and i as:

$$\mathcal{E}((\mathcal{N}^j, \ldots, \mathcal{N}^i), x^j) = \frac{\sum_{l=j}^{i} \mathcal{I}(\mathcal{N}^l, x^{l-1})}{\sum_{l=j}^{i} \mathcal{T}(\mathcal{N}^l, x^{l-1})}, \text{ with } 0 > j > i \tag{5.4}$$

Given an initial solution x^0, we can define an optimal neighbourhood-usage as:

$$\max_{\mathcal{N}^1,\ldots,\mathcal{N}^e \in \mathfrak{N}} \mathcal{E}((\mathcal{N}^1, \ldots, \mathcal{N}^e), x^0), \text{ with} \tag{5.5}$$

$$\sum_{l=1}^{e-1} \mathcal{T}(\mathcal{N}^l, x^{l-1}) <= tTime \tag{5.6}$$

$$\sum_{l=1}^{e} \mathcal{T}(\mathcal{N}^l, x^{l-1}) > tTime \tag{5.7}$$

$$e \geq 1,\ tTime > 0$$

Equation (5.5) states that we are searching for a neighbourhood-usage $(\mathcal{N}^1, \ldots, \mathcal{N}^e)$, such that for a given amount of time $tTime$ the best efficiency can be achieved. Consider that the overall number of performed iterations e depends on $tTime$, the initial solution x^0 and on the chosen neighbourhood structures. Assuming that all neighbourhood-usages have the same execution time, this formulation is equivalent of finding a neighbourhood-usage which provides a solution with the minimum objective value in the given amount of time.

When e_{min} is the minimum number of iterations of all possible neighbourhood-usages within the given amount of time $tTime$, then at least $|\mathfrak{N}|^{e_{min}}$ possible neighbourhood-usages exist. Thus, exploring all of them is in practise impossible.

5.2 Underlying Sequential VNS

The objective of Parallel Efficiency Guided VNS (PEGVNS) is to find an efficient neighbourhood-usage $(\mathcal{N}^1, \ldots, \mathcal{N}^e)$. Thereby, the current solution is improved by diverse neighbourhoods in parallel and afterwards the solution of the most efficient neighbourhood is used for the future search.

Within this section the so-called *Time Restricted Randomised VNS* (TRRVNS) which builds the basis for PEGVNS is presented. TRRVNS enables an efficient parallelisation and supports finding an efficient neighbourhood-usage which might also prove advantageous for TRRVNS.

5.2.1 Time Restricted Local Search within a Parallel VNS-approach

The only difference between *Time Restricted Local Search* (TRLS) and Basic LS is that the search is stopped after a predefined amount of available computation time t. The stop of Local Search is mainly required to reduce the synchronisation overhead and to allow a changing to a more efficient neighbourhood. Consider to make possible a TRLS, also a time restricted neighbourhood evaluation has to be used. Within this work a next improvement strategy with a randomised examination order, denoted as RandExam, is used. Therefore, we use RandExam as described in Algorithm 5 for $\mathcal{S}(\mathcal{N}_k(x^i))$. Algorithm 4 shows the pseudocode of TRLS.

Algorithm 4 Time Restricted Local Search (TRLS)

```
Input: initial solution $x$, a neighbourhood structure $\mathcal{N}_k$ and a maximum allowed
    execution time $t$
1: repeat
2:    $x' = \text{RandExam}(\mathcal{N}_k, x, t - execTime)$ {stop if remaining time has elapsed}
3:    if $\text{f}(x') < \text{f}(x)$ then
4:       $x \leftarrow x'$
5:    end if
6: until $execTime > t$ or or no improvement is achieved
```

Reducing Parallelisation Overhead

From the viewpoint of an efficient parallelisation the goal is using a sequential algorithm which causes a low parallelisation overhead. Without loss of generality, we assume here that a single iteration of LS is performed with two neighbourhood structures in parallel. Assume we improve a given solution x^{i-1} with $\mathcal{N}^i_j$ and $\mathcal{N}^i_k$ in parallel and obtain two solution improvements x^i_j and x^i_k. Because the values of $\mathcal{T}$ of different neighbourhoods are mostly not equal to each other, the problem that in some cases $\mathcal{E}((\mathcal{N}_j^i), x^{i-1}) > \mathcal{E}((\mathcal{N}_k^i), x^{i-1})$, while $\mathrm{f}(x_j^i) > \mathrm{f}(x_k^i)$, arises. If choosing the solution generated by the neighbourhood with the higher efficiency, the solution with the worse objective value is chosen. Therefore, the additional computation does not contribute to a finding of an efficient neighbourhood-usage. This causes an excess of computation, resulting in an additional parallelisation overhead. However, choosing the solution with the best objective value leads to an inefficient neighbourhood-usage, which causes also an excess of computation. Although we want to use the most efficient neighbourhood in each iteration, we will always use the solution generated by the neighbourhood with maximum $\mathcal{I}$. A usage of TRLS can avoid such effects.

An additional argument supporting a TRLS is that the synchronisation overhead is minimised. Otherwise, all processes must wait until the process using the neighbourhood with the maximum value of $\mathcal{T}$ has finished. Therefore each TRLS running in parallel should use the same execution time t. However, even if all processes stop at the same point of time, exchanging the new solution causes a basic synchronisation overhead. Therefore, the number of necessary synchronisations should be reduced. Moreover, the communication overhead decreases when the number of synchronisations is decreased. This implies that the execution time t of TRLS should be as large as possible and thus several iterations of TRLS should be performed, whereas the number of performed iterations depends on the used neighbourhood. However, running TRLS too long leads to an inefficient neighbourhood-usage.

Finding an efficient Neighbourhood-Usage

Assume we apply on an initial solution x^0 a complete LS with a single neighbourhood $\mathcal{N}_j$ until a local minimum has been achieved. Let us assume that the local minimum is reached in iteration i. In this case the efficiency of the neighbourhood structure will be $\mathcal{E}((\mathcal{N}_j^i), x^{i-1}) = 0$, but often a neighbourhood $\mathcal{N}_k^i$ exists where $\mathcal{E}((\mathcal{N}_k^i), x^{i-1}) > 0$ with $j \neq k$. Therefore, it is in any case better to abort the last iteration of LS and to use neighbourhood $\mathcal{N}_k^i$ instead where the solution can be improved. Figure 5.1 illustrates the idea of skipping the last iteration of LS to increase efficiency.

$$\begin{tikzcd}
\mathcal{N}_1 \arrow[r] \arrow[d, dotted, no head] & \mathcal{N}_1 \arrow[r] \arrow[d, dotted, no head] & \mathcal{N}_1 \arrow[r] & \mathcal{N}_2 \arrow[r] \arrow[dl, dotted, no head] & \mathcal{N}_1 \arrow[r] & \mathcal{N}_2 \arrow[dll, dotted, no head] \\
\mathcal{N}_1 \arrow[r] & \mathcal{N}_1 \arrow[r] & \mathcal{N}_2 \arrow[r] & \mathcal{N}_2 & &
\end{tikzcd}$$

Fig. 5.1: Rescheduling neighbourhoods to increase efficiency

However, a forecast, when this last iteration occurs, is not possible. But if choosing appropriate values for t, TRLS aborts when the time for evaluating a neighbourhood increases. This can be a hint that we get near a local minimum or we are in this last iteration where no improvement will occur. Then a switching to another more efficient neighbourhood is possible.

Moreover, if running TRLS with a neighbourhood there may already be an earlier point of time when the efficiency of the used neighbourhood is worse than another neighbourhood. Thus, stopping TRLS from time to time gives the opportunity to investigate a more efficient neighbourhood and therefore to increase the efficiency of the neighbourhood-usage. Therefore, from the viewpoint of finding an efficient neighbourhood-usage, TRLS should stop after a small amount of time t, to allow a changing to a more efficient neighbourhood.

5.2.2 Random Neighbourhood Selection

The basic idea of a randomised VNS (RandVNS) is to choose the neighbourhoods randomly instead of applying a statically determined order. Computational experiments carried out in [29] have delivered better results for RandVNS than for VNS using a static neighbourhood-order.

The superiority of a RandVNS over a VNS using a static neighbourhood-order might be traced back to interfering effects of VND when using a static neighbourhood-order. We have already argued that the efficiency goes down during LS and therefore another neighbourhood might get more efficient. But in VND using a static neighbourhood-order in any case the first neighbourhood is used until a local minimum is achieved, even if a more efficient neighbourhood exists. Finally, we get into a local minimum, resulting in an efficiency of $\mathcal{E}(\mathcal{N}_1, x^{i-1}) = 0$. But as afterwards only one iteration with another neighbourhood $\mathcal{N}_j$, with $j \neq 1$, is performed, the efficiency of the first neighbourhood cannot be boosted enough. The reason is that most moves will not lead to improvements, because only in a small area changes have occurred. Therefore, the efficiency of the first neighbourhood may still be less efficient compared to other neighbourhoods. By using a random neighbourhood selection, we can overcome this drawback. Thus, it would be better to switch to the most efficient neighbourhood and continue TRLS until another neighbourhood gets more efficient. Because we do not know which neighbourhood is most efficient, we choose a random neighbourhood. When running more neighbourhoods in parallel we are able to choose the solution generated from the most efficient neighbourhood.

5.2.3 Next Improvement with Randomised Examination Order

For evaluating a neighbourhood, different strategies, which have a major influence on the efficiency are possible. When using a best improvement strategy, always the maximum value for $\mathcal{I}$ can be achieved. However, we always get the maximum value for $\mathcal{T}$. Assuming all moves have the same execution time. Then in a next improvement strategy the optimal efficiency can be achieved if the first applied move leads to the best solution. In this optimal case a next improvement strategy has in any case a higher efficiency than a best improvement strategy.

To increase efficiency of a next improvement strategy, it is better to begin applying moves where improvements can quickly be found and where the improvement potential is the highest. When using always the same predefined examination order, the neighbourhood is always examined in the same order which might reduce the improvement potential. Suppose, we divide a given neighbourhood $\mathcal{N}_k$ into two neighbourhoods $\mathcal{N}_k = \mathcal{N}_{k1} \cup \mathcal{N}_{k2}$ with $\mathcal{N}_k = \mathcal{N}_{k1} \cap \mathcal{N}_{k2} = \emptyset$. Then in a specific examination order these neighbourhoods are always applied in the same order $(\mathcal{N}_{k1}, \mathcal{N}_{k2})$. As in a specific examination order the neighbourhood $\mathcal{N}_{k2}$ is not evaluated until a local minimum has been achieved in $\mathcal{N}_{k1}$, the same problems as in VND using a static neighbourhood-order occur. To overcome this drawback a randomised examination order should be chosen to diversify the search [4].

Algorithm 5 presents the pseudocode of the next improvement strategy with a randomised examination order used within this work. Thereby, in each iteration a random permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ is newly created. Afterwards moves are selected by using the random values $\pi(0), \dots, \pi(n)$ for the parameters $a_1, \dots, a_i$ defining a move. If the selected move is included in the corresponding neighbourhood structure, it is applied to the current solution, otherwise the next move is chosen.

Algorithm 5 RandExam

```
Input: a neighbourhood structure $\mathcal{N}_k$, a solution $x$ and a maximum allowed execution
    time $t$
 1: $\pi \leftarrow$ randShuffle($\pi$)
 2: for $a_1 = \pi(0)$ to $\pi(n)$ do
 3:    ...
 4:    for $a_i = \pi(0)$ to $\pi(n)$ do
 5:       if $execTime > t$ then
 6:          return $x$
 7:       else
 8:          $j \leftarrow$ Index($a_1, \dots, a_i$) {move $\mu_{k,j}$ is defined by parameters $a_1, \dots, a_i$}
 9:          if $\mu_{k,j} \in \mathcal{N}_k$ then
10:             $x' = \mu_{k,j}(x)$
11:             if f($x'$) < f($x$) then
12:                return $x'$
13:             end if
14:          end if
15:       end if
16:    end for
17:    ...
18: end for
19: return $x$
```

5.2.4 Time Restricted Randomised VNS

Combining the above ideas regarding neighbourhood ordering and considerations on Local Search we present a new variant of VNS called Time Restricted Randomised VNS (TRRVNS). This algorithm is based upon General VNS scheme described in Algorithm 3 and uses TRLS, a next improvement strategy with a randomised examination order and a random neighbourhood selection. Algorithm 6 shows the pseudocode of TRRVNS.

At the beginning it is easy to find better solutions quickly. Therefore, we can choose a small value for time t, to avoid evaluating a neighbourhood which is near a local minimum or in the last iteration. With time it gets harder to find better solutions with small values of time t, because only a part of the large neighbourhoods can be evaluated while small neighbourhoods do not lead to improvements any more. Therefore, we increase the value of t by the function adaption(t) when the number of neighbourhood examinations without improvements exceeds a given parameter *maxRetries*. This leads in general to a more detailed search within neighbourhoods. Furthermore if *maxRetries* is exceeded, a Shaking is performed.

5.3 Parallel Efficiency Guided VNS

When parallelising TRRVNS a new variant of VNS evolves called Parallel Efficiency Guided VNS (PEGVNS). The basic idea of Parallel Efficiency Guided VNS (PEGVNS) is to improve a given solution x' by examining more than one neighbourhood of x' at the same time. Afterwards the solution generated by the most efficient neighbourhood is selected and the search is continued with this solution. In a first step, a set of neighbourhood structures is chosen randomly to be applied on the current best solution using TRLS as local improvement strategy. In the case TRLS terminates prematurely due to a local minimum, a further neighbourhood structure is selected randomly and the search is continued for the remaining time. This is done to utilise a processor for the whole assignend CPU time. Finally, we can choose the solution with the minimum objective value in each iteration which is also the solution generated by the most efficient neighbourhood. Because all neighbourhoods are chosen randomly, it is possible choosing a single neighbourhood twice. But as a next improvement strategy with a randomised examination order is used, it is likely that even in this case two different solutions can be generated. Algorithm 7 shows the pseudocode of PEGVNS.

Algorithm 6 Time Restricted Randomised VNS (TRRVNS)

```
1:  $x \leftarrow$ initialize()
2:  $x' \leftarrow x$
3:  $l \leftarrow 1$
4:  $retry \leftarrow 0$
5:  **repeat**
6:     **if** $retry = maxRetries$ **then**
7:        **if** $\mathrm{f}(x') \geq \mathrm{f}(x)$ **then**
8:           $x' \leftarrow x$
9:           $l \leftarrow l + 1$
10:       **else**
11:          $x \leftarrow x'$
12:          $l \leftarrow 1$
13:       **end if**
14:       $x' \leftarrow$ Shaking($x', l$)
15:       $retry \leftarrow 0$
16:       $t \leftarrow$ adaption($t$) {for example $t \leftarrow t \cdot 2$}
17:    **end if**
18:    $\mathcal{N}_k \leftarrow$ selectRandNeighbourhood()
19:    $x'' \leftarrow$ TRLS($\mathcal{N}_k, x', t$)
20:    **if** $\mathrm{f}(x') \leq \mathrm{f}(x'')$ **then**
21:       $retry \leftarrow retry + 1$
22:    **else**
23:       $x' \leftarrow x''$
24:       $retry \leftarrow 0$
25:    **end if**
26: **until** stopping condition is met
```

Possible Speedup

The only possibility of PEGVNS to achieve a speedup compared to a sequential VNS can be caused by a more efficient neighbourhood-usage, which implies that the efficiency of different neighbourhood-usages must not be equal when applying PEGVNS. Otherwise the speedup S_p is always equal to one, independent of the number of processors used.

For the speedup metric it is necessary to compare the parallel execution time T_p of PEGVNS with the sequential execution time T_s of a sequential VNS. For this purpose the considered programs stop at a predefined objective value $tObj$. In this case the efficiency of a neighbourhood-usage $\mathcal{E}((\mathcal{N}^1, \ldots, \mathcal{N}^e), x^0)$ holds the following restrictions:

$$\begin{aligned} \mathrm{f}(x^e) &< tObj \\ \mathrm{f}(x^{e-1}) &>= tObj \end{aligned}$$

Then the maximum speedup of PEGVNS including only the excess of computation introduced by running TRLS with different neighbourhoods in parallel can be written as:

$$S_p = \frac{T_s}{T_p} <= \frac{\mathcal{E}_\mathrm{p}((\mathcal{N}^{1_p}, \ldots, \mathcal{N}^{e_p}), x^0)}{\mathcal{E}_\mathrm{s}((\mathcal{N}^{1_s}, \ldots, \mathcal{N}^{e_s}), x^0)} = \frac{\sum_{l=1_s}^{e_s} \mathcal{T}(\mathcal{N}^l, x^{l-1})}{\sum_{l=1_p}^{e_p} \mathcal{T}(\mathcal{N}^l, x^{l-1})} \tag{5.8}$$

Here $\mathcal{E}_p$ is the efficiency of the generated neighbourhood-usage of PEGVNS and $\mathcal{E}_s$ is the efficiency of the neighbourhood-usage of the to be compared sequential VNS. Equation (5.8) states that the real speedup including all other parallelisation overheads is bounded by the efficiency of the neighbourhood-usage used by PEGVNS. For example, a linear speedup can only be achieved if the generated neighbourhood-usage is at least p-fold so efficient when running PEGVNS on p processors. However, even a super linear speedup of PEGVNS is possible. But when comparing PEGVNS running on p processors with PEGVNS simulated on a sequential computer, this super linear speedup disappears.

Algorithm 7 Parallel Efficiency Guided VNS (PEGVNS)

```
1: $x \leftarrow$ initialize()
2: $x' \leftarrow x$
3: $l \leftarrow 1$
4: $retry \leftarrow 0$
5: **repeat**
6:    **if** $retry = maxRetries$ **then**
7:       $retry \leftarrow 0$
8:       **if** $\mathrm{f}(x') \geq \mathrm{f}(x)$ **then**
9:          $x' \leftarrow x$
10:         $l \leftarrow l + 1$
11:      **else**
12:         $l \leftarrow 1$
13:         $x \leftarrow x'$
14:      **end if**
15:      $x' \leftarrow$ Shaking$(x', l)$
16:      $t \leftarrow$ adaption$(t)$
17:   **end if**
18:   **do in parallel**
19:      $x_i \leftarrow x'$
20:      **repeat**
21:         $\mathcal{N}_k \leftarrow$ selectRandNeighbourhood()
22:         $x_i \leftarrow$ TRLS$(\mathcal{N}_k, x_i, t - execTime)$
23:      **until** $execTime > t$
24:   **end do in parallel**
25:   **if** $\mathrm{f}(x') \leq \mathrm{f}(x_i)$, $\forall x_i$ **then**
26:      $retry \leftarrow retry + 1$
27:   **end if**
28:   **for all** $x_i$ **do**
29:      **if** $\mathrm{f}(x_i) < \mathrm{f}(x')$ **then**
30:         $x' \leftarrow x_i$
31:      **end if**
32:   **end for**
33: **until** stopping condition is met
```

6 Neighbourhood Structures

One crucial point in developing a VNS approach is the appropriate design of neighbourhoods used with VND and VNS, respectively. For the formal description of the neighbourhoods the notations of Table 6.1 are used. These notations and all defined moves are based upon [17] and the implementations of neighbourhood Swapping and Shifting are based upon [27, 28].

Table 6.1: Notations used for describing neighbourhood structures

$(\chi_1, \ldots, \chi_n)$	...	configuration vector of current solution x
χ_i	...	car at position i
π_i	...	subsequence of x of length $\geq\ 0$
(χ_i)	...	subsequence of x of length $= 1$
$\cdot$	...	concatenates two subseuqences
$(\chi_i, \chi_{i+1}, \ldots, \chi_{i+k})$	...	subsequence of x of length $= k$

Table 6 lists the five used neighbourhood structures, their size and the time for evaluating the whole neighbourhood. All neighbourhood structures use an incremental

Table 6.2: Used Neighbourhood Structure

short	neighbourhood	size	complete examination
$\mathcal{N}_{SW}$	Swapping	$O(n^2)$	$O(n \cdot \|K\| \cdot \sum_{c \in C \setminus F} m_c)$
$\mathcal{N}_{SH}$	Shifting	$O(n^2)$	$O(n \cdot \|K\| \cdot \sum_{c \in C \setminus F} m_c)$
$\mathcal{N}_{INV}$	Inverting	$O(n^2)$	$O(n^2 \cdot \sum_{c \in C \setminus F} m_c)$
$\mathcal{N}_{BS}$	Block Swapping	$O(n^3)$	$O(n^3 \cdot \sum_{c \in C \setminus F} m_c)$
$\mathcal{N}_{BSH}$	Block Shifting	$O(n^3)$	$O(n^3 \cdot \sum_{c \in C \setminus F} m_c)$

evaluation of the objective value. For this purpose an array of size $n \cdot (|C|-1)$ is used, where for each component c and each sliding window of length m_c the number of occurrences for the component within the sliding window is stored. Then the number of violations of a component c within a sliding window caused by a single configuration exchange can be computed in constant time. Furthermore, all colour changes are computed incrementally by comparing the colour of the new configuration with the colour of the old configuration. For the maximum colour block size we use a second array of size n, where each entry points to the beginning of the colour block. Thus, the recalculations of violations caused by exceeding the maximum colour block size can be done in constant time.

6.1 Swapping

A swap move (SW) exchanges the positions of the configurations χ_i and χ_j of the current solution x. Formal a swap move can be defined as:

$$\begin{aligned}\mathrm{SW}((\pi_1 \cdot (\chi_i) \cdot \pi_2 \cdot (\chi_j) \cdot \pi_3), i, j) = \\ (\pi_1 \cdot (\chi_j) \cdot \pi_2 \cdot (\chi_i) \cdot \pi_3)\end{aligned} \tag{6.1}$$

To recalculate the objective value efficiently, only the configurations $(\chi_i, \ldots, \chi_{i+m_c-1})$ and $(\chi_j, \ldots, \chi_{j+m_c-1})$ must be considered. Thereby, for each component $c \in C \setminus F$ and for each sliding window the changes in violations have to be recalculated. Thus, an evaluation of a single swap move is possible in time $O(\sum_{c \in C \setminus F} m_c)$.

Then the neighbourhood $\mathcal{N}_{SW}$ consists of all solutions which can be derived by applying a permitted single swap move SW on the current solution:

$$\mathcal{N}_{SW}(x) = \{x' : x' = \mathrm{SW}(x, i, j) \land 1 \leq i < j \leq n, \forall i, j \in \mathbb{N}\} \tag{6.2}$$

Because the maximum size of the neighbourhood Swapping is bounded by $\frac{n^2-n}{2}$, the size of the neighbourhood is in $O(n^2)$. To further enhance the recalculation of the objective value for each position i the changes in violations for the whole subsequence $(\chi_i, \ldots, \chi_{i+m_c-1})$ are cached. Because a configuration at position i can be replaced by at most $|K|$ configurations, a complete examination of $\mathcal{N}_{SW}$ is possible in time $O(n \cdot |K| \cdot \sum_{c \in C \setminus F} m_c)$.

6.2 Shifting

By the operator shifting (SH) a configuration at position j is inserted at another position i and all configurations between position i and j are shifted backward or forward. Formal we can write:

$$\begin{aligned}\mathrm{SH}((\pi_1 \cdot (\chi_i) \cdot \pi_2 \cdot (\chi_j) \cdot \pi_3), i, j) = \\ (\pi_1 \cdot (\chi_j) \cdot (\chi_i) \cdot \pi_2 \cdot \pi_3), \text{ with } j > i\end{aligned} \tag{6.3}$$

or

$$\begin{aligned}\mathrm{SH}((\pi_1 \cdot (\chi_i) \cdot \pi_2 \cdot (\chi_j) \cdot \pi_3), i, j) = \\ (\pi_1 \cdot \pi_2 \cdot (\chi_j) \cdot (\chi_i) \cdot \pi_3), \text{ with } j > i\end{aligned} \tag{6.4}$$

The incremental computation is done similar to the recalculation of the objective value of a swap move. Thus a single shift move SH can be evaluated in time $O(\sum_{c \in C \setminus F} m_c)$.

Then the neighbourhood $\mathcal{N}_{SH}$ can be defined as:

$$\mathcal{N}_{SH}(x) = \{x' : x' = \text{SH}(x,i,j) \wedge 1 \leq i < j-1 \leq n-1, \forall i,j \in \mathbb{N}\} \tag{6.5}$$

The neighbourhood Shifting contains at most $n^2 - n$ solutions, thus the size of this neighbourhood is in $O(n^2)$. Because the neighbourhood Shifting uses the same caching strategy as used for the neighbourhood Swapping, a complete examination of the neighbourhood is possible in time $O(n \cdot |K| \cdot \sum_{c \in C \setminus F} m_c)$.

6.3 Inverting

An inversion move (INV) inverts the configurations between the positions i and j. Thereby, the configuration χ_i is exchanged with configuration χ_j, the configuration χ_{i+1} is exchanged with configuration χ_{j-1} and so on. In a more formal manner an inversion move can be written as:

$$\begin{aligned}\text{INV}((\pi_1 \cdot (\chi_i, \chi_{i+1}, \ldots, \chi_{j-1}, \chi_j) \cdot \pi_2), i, j) = \\ (\pi_1 \cdot (\chi_j, \chi_{j-1}, \ldots, \chi_{i+1}, \chi_i) \cdot \pi_2)\end{aligned} \tag{6.6}$$

Because the configurations $(\chi_i, \ldots, \chi_j)$ are inverted, there may be changes in violations in the the whole subsequence $(\chi_i, \ldots, \chi_{j+m_c-1})$. If for simplicity disregarding the previous production day, the violations for the subsequence $(\chi_{i+m_c-1}, \ldots, \chi_j)$ before applying the inversion move can be calculated by:

$$\sum_{p=i+m_c-1}^{j} \max(0, (\sum_{q=p}^{p+m_c-1} a_{c\chi_q}) - l_c)) \tag{6.7}$$

Assuming a configuration at position l from the original solution x gets the position l' in the new solution $x' = \text{INV}(x,i,j)$, the violations for the subsequence $(\chi_{j'}, \ldots, \ldots, \chi_{i'+m_c-1})$ must be considered. The objective value in this interval can be recalculated by:

$$\sum_{p=j'}^{i'+m_c-1} \max(0, (\sum_{q=p}^{p+m_c-1} a_{c\chi_q}) - l_c)) \tag{6.8}$$

Because $a_{c\chi_{k'}} = a_{c\chi_k}$ with $j' \leq k \leq i' + m_c - 1, \forall k \in \mathbb{N}$, Eq. (6.7) and Eq. (6.8) are equivalent. Therefore, for a recalculation of the objective value, only the configurations $(\chi_i, \ldots, \chi_{i+m_c-1})$ and $(\chi_{j+1}, \ldots, \chi_{j+m_c-1})$ must be considered. Thus, an evaluation of an inversion move is possible in time $O(\sum_{c \in C \setminus F} m_c)$.

Then the neighbourhood $\mathcal{N}_{INV}$ consists of all solutions which can be derived by applying a single permitted inversion move on the current solution:

$$\mathcal{N}_{INV}(x) = \{x' : x' = \text{INV}(x, i, j) \wedge 1 \leq i < j - 2 \leq n - 1, \forall i, j \in \mathbb{N}\} \tag{6.9}$$

In the neighbourhood $\mathcal{N}_{INV}$ only inversion moves of length $|j-i|+1 \geq 4$ are included, because all solutions generated by inversion moves with $|j - i| + 1 < 4$ are already included in the neighbourhood Swapping. The size of the neighbourhood is in $O(n^2)$ and a complete examination of the neighbourhood can be achieved in time $O(n^2 \cdot \sum_{c \in C \setminus F} m_c)$.

6.4 Block Swapping

A block swap move (BSW) exchanges the two blocks starting at position i and j with equal length $l - 1$:

$$\begin{aligned} \text{BSW}((\pi_1 \cdot (\chi_i, \ldots, \chi_{i+l}) \cdot \pi_2 \cdot (\chi_j, \ldots, \chi_{j+l}) \cdot \pi_3), i, i+l, j) = \\ (\pi_1 \cdot (\chi_j, \ldots, \chi_{j+l}) \cdot \pi_2 \cdot (\chi_i, \ldots, \chi_{i+l}) \cdot \pi_3) \end{aligned} \tag{6.10}$$

For the recalculation of the objective value only changes in violations for configurations $(\chi_j, \ldots, \chi_{j+m_c-1})$, $(\chi_{j+l}, \ldots, \chi_{j+l+m_c-1})$, $(\chi_i, \ldots, \chi_{i+m_c-1})$, and $(\chi_{i+l}, \ldots, \chi_{i+l+m_c-1})$ can occur. Furthermore, for the last three intervals a possible deviation of the stored number of occurrences of components within a sliding window has to be taken into account. But as the recalculation has to be done only once and the calculation takes maximum $\sum_{c \in C \setminus F} m_c$ steps, a complete evaluation of a block move is possible in time $O(\sum_{c \in C \setminus F} m_c)$.

Then the neighbourhood $\mathcal{N}_{BSW}(x)$ can be defined as:

$$\begin{aligned} \mathcal{N}_{BSW}(x) = \{x' : x' = \text{BS}(x, i, i+l, j) \wedge \\ 1 \leq i < i+l < j < j+l \leq n, \forall i, j, l \in \mathbb{N}\} \end{aligned} \tag{6.11}$$

The size of the neighbourhood is in $O(n^3)$ and a complete examination of the neighbourhood can be achieved in $O(n^3 \cdot \sum_{c \in C \setminus F} m_c)$.

6.5 Block Shifting

By a block shift move (BSH) the block $(\chi_j, \ldots, \chi_{j+l})$ is shifted directly between cars χ_i and χ_{i-1} or χ_i and χ_{i+1}, respectively. A block shift move BSH can also be seen as exchanging two consecutive blocks with arbitrary lengths. Formal we can write:

$$\begin{aligned}\mathrm{BSH}((\pi_1 \cdot (\chi_i, \ldots, \chi_{j-1}) \cdot (\chi_j, \ldots, \chi_{j+l}) \cdot \pi_3), i, j, j+l) = \\ (\pi_1 \cdot (\chi_j, \ldots, \chi_{j+l}) \cdot (\chi_i, \ldots, \chi_{j-1}) \cdot \pi_3)\end{aligned} \tag{6.12}$$

The recalculation of the objective value is done analog to BSW. Thus, an evaluation is possible in time $O(\sum_{c \in C \setminus F} m_c)$.

Now we can define the neighbourhood $\mathcal{N}_{BSH}$ as:

$$\begin{aligned}\mathcal{N}_{BSH}(x) = \{x' : x' = \mathrm{BSH}(x, i, j, j+l) \wedge 1 \leq i < j-1 < j < j+l \leq n \\ \wedge\, j-1-i \neq l,\ \forall i, j, l \in \mathbb{N}\}\end{aligned} \tag{6.13}$$

In the neighbourhood Block Shifting all block shifting moves with equal block length are excluded, because they are already included in the neighbourhood Block Swapping. The size of the neighbourhood is in $O(n^3)$ and a complete examination of the neighbourhood can be achieved in $O(n^3 \cdot \sum_{c \in C \setminus F} m_c)$.

7 Tests and Results

This chapter presents the results of the computational experiments carried out in this thesis. All tests were performed on two Dual Core AMD Opteron(tm) 270 Processors with 2 GHz and 8 GB RAM in total. Everything was implemented in C++ using EAlib 2.0, which is implemented in C++ too. EAlib 2.0 is a generic library for metaheuristics which was developed at the Institute of Computer Graphics and Algorithms at the Vienna University of Technology. The parallel programs were implemented with MPI [24] using a master-slave approach.

All algorithms use a randomly generated initial solution. Thereby, first a basic solution is generated by placing cars with configuration 1 in the first positions of the assembly line, in the subsequent positions cars with configuration 2 and so forth. Afterwards n random swap moves are applied on this basic solution. For Shaking(x, l), instead of applying a random move in neighbourhood $\mathcal{N}_l$, l random swap moves are applied on the current solution. If not stated otherwise, all algorithms use a next improvement strategy with a randomised examination order as described in Algorithm 5.

All programs were tested on instances defined by Renault for the ROADEF Challenge 2005 and on newly created instances, which are also based upon instances of the ROADEF Challenge 2005. For the comparison of the results we implemented a *Parallel Multi Neighbourhood-Order VNS* (PMNOVNS). The basic idea of PMNOVNS is to run General VNS as described in Algorithm 3 with all possible static neighbourhood-orders in parallel. To save computation time, PMNOVNS uses the approach depicted in Figure 7.1 for investigating all possible neighbourhood-orders. First an initial solution x^0 is created randomly. Afterwards, for each neighbourhood a General VNS scheme is started, where the different neighbourhoods are used for the first neighbourhood $\mathcal{N}_1$ of the neighbourhood-order. If in one of the performed General VNS schemes a local minimum occurs, several new General VNS schemes are created. Thereby, all yet unused neighbourhoods of the neighbourhood-order are used for $\mathcal{N}_2$.

At whole three algorithms—PMNOVNS, TRRVNS and PEGVNS—with different settings were tested. In the case of PMNOVNS at least 5 and possible up to 120 General VNS schemes were performed with different neighbourhood-orders in parallel. In the case of PEGVNS the program was run with 2, 4, 8 and 16 processes in parallel. Therefore the different algorithms were only tested with 10 instances and for each instance 10 independent runs were performed.

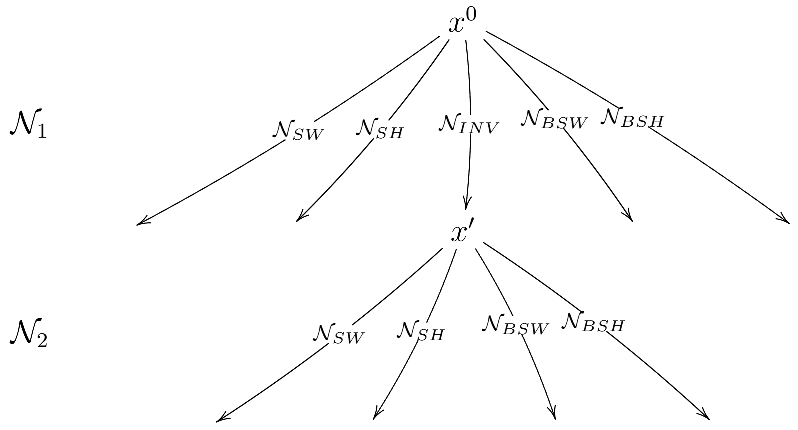

Fig. 7.1: Visualisation of PMNOVNS

7.1 Used Test Instances

In this section we present the used test instances in detail. Altogether 10 instances, including instances with 918 up to 1992 cars, were used for computational experiments. All used test instances are based upon test instances published by ROADEF and Renault for the ROADEF Challenge 2005 [25]. For the ROADEF challenge 2005 three test sets—A, B and X—were published. Because set X was used for the final evaluation process, we used set X as the basis of our test instances. It consists of 19 test instances, including test instances with 65 up to 1319 cars, 5 to 20 colours and 5 to 26 components. To allow a better comparison of the possible speedup of PEGVNS, we have chosen the five instances of set X with the largest number of cars n. Furthermore we created five new test instances by duplicating a single instance of set X and combining them to a new test instance. This can be seen as a possible attempt to consider more than one production day in the optimisation process. Table 7.1 presents the used instances for the computational experiments. In the first column the short name of the used instance, which is used within this thesis, is given. Instances (1)-(5) are the same instances as defined in set X, whereas instances (6)-(10) are the newly created test instances. For the new test instances in the second column the number of combined test instances is given.

7.2 Performance Measurement

For the performance evaluation we use a predefined objective value *tObj* as the stopping condition of the algorithms, i.e. an algorithm stops if $\mathrm{f}(x) < tObj$. Table 7.2 shows the used values of *tObj* for each instance. Furthermore in column two the best obtained result during the ROADEF 2005 challenge (R-best) is given. For the newly created instances (6)-(10) the hypothetical value for R-best is given in parentheses, which is derived by multiplying the value of R-best of the original instance

Table 7.1: Used test instances for computational experiments

short	dup.	original instance	n	$\|C\|$
(1)		023_EP_RAF_ENP_S49_J2	1260	12
(2)		024_EP_RAF_ENP_S49_J2	1319	18
(3)		025_EP_ENP_RAF_S49_J1	996	20
(4)		039_CH1_EP_RAF_ENP_S49_J1	1247	12
(5)		039_CH3_EP_RAF_ENP_S49_J1	1073	12
(6)	2 x	025_EP_ENP_RAF_S49_J1	1992	20
(7)	2 x	034_VP_EP_RAF_ENP_S51_J1_J2_J3	1842	8
(8)	2 x	048_CH1_EP_RAF_ENP_S50_J4	1038	22
(9)	2 x	048_CH2_EP_RAF_ENP_S49_J5	918	20
(10)	4 x	028_CH1_EP_ENP_RAF_S50_J4	1300	20

with the number of considered instances in the new instance. Column three shows the theoretical placement in the ROADEF 2005 competition of an algorithm achieving a result *tObj*. Again for the new instances we give the hypothetical rank in parentheses. Because at whole 18 teams participated in the final evaluation procedure, the last position would be position 19.

Table 7.2: Stopping conditions for instances

short	R-best	rank	*tObj*
(1)	192,466	15	235,000
(2)	337,006	10	500,000
(3)	160,408	15	220,000
(4)	69,239	12	74,000
(5)	231,030	15	240,000
(6)	(320,815)	(1)	300,000
(7)	(111,990)	(16)	173,000
(8)	(394,011)	(16)	425,000
(9)	(62,155,832)	(18)	64,205,000
(10)	(145,365,982)	(18)	198,000,000

In Section 4.2 we have argued that the primary goal of parallelisation is a reduction of the wall clock time and therefore the primary performance measure should be based upon the execution time T_s of a program. Nonetheless, we will use the CPU time as the basic performance metric and not the execution time T_s. The main reason is to allow a comparison of the efficiency of neighbourhood-usages generated by the different algorithms. By the usage of the CPU time as a performance measure, all other overheads which would falsify a comparison are excluded. Thus, it is possible to compare the generated neighbourhood-usages and draw conclusion about the appearance of an efficient neighbourhood-usage. Another reason is that PEGVNS was explicitly designed such that the parallel overheads idling and interprocess interaction are minimised. Thus, the dominant component of the parallel overhead will be the excess of computation.

7.3 Comparison of Randomised and Specific Examination Order

In order to evaluate the performance of a randomised examination order, we compare PMNOVNS$_{spec}$ using a specific examination order and PMNOVNS$_{rand}$ using a randomised examination order with each other. For this purpose we run PMNOVNS$_{rand}$ using the randomised examination order described in Algorithm 5, where in each iteration a new random permutation is used. Because a good specific examination for PMNOVNS$_{spec}$ order is not known, we use a specific examination similar to Algorithm 5, but without performing the function randShuffle. Consider that for PMNOVNS$_{spec}$ the same initial permutation is used during the whole program execution, whereas for PMNOVNS$_{rand}$ the random permutation is newly generated in each iteration.

Table 7.3 presents the results of this comparison. Listed are the average CPU times, standard deviations and median CPU times of both algorithms. Thereby for the CPU time of PMNOVNS the CPU time of a single considered General VNS scheme, which first reached the predefined objective value, is used. Furthermore we performed a one-sided Wilcoxon rank sum test (U-test) to see how significant the differences in the results between PMNOVNS$_{spec}$ and PMNOVNS$_{rand}$ are. Column U-test shows the error probabilities for the hypotheses that the smaller mean CPU time (depicted in bold) is less than the larger mean CPU time. Finally, in the last row the average CPU times over all instances are listed.

When comparing the mean and median CPU times of both algorithms, a relative clear difference can be observed. In nine of ten instances the mean CPU time of PMNOVNS$_{rand}$ is better compared with PMNOVNS$_{spec}$ and when comparing the median CPU times, PMNOVNS$_{rand}$ is still better in seven instances. Also when comparing the average CPU time over all performed runs, a clear difference can be observed. However, only for four out of the ten instances significantly better results can be obtained.

Therefore we investigate the influence of a random examination order in more detail. For this purpose we inspect the CPU times of LS$_{spec}$ using a specific examination and LS$_{rand}$ using a randomised examination of the performed algorithms PMNOVNS$_{spec}$ and PMNOVNS$_{rand}$. Thereby we determined the CPU times of LS$_{\mathrm{rand}}$ and LS$_{\mathrm{spec}}$ from creating the initial solution x^0 until a defined stopping condition $tObj$ is achieved. Table 7.4 presents the results of the average CPU times in seconds and standard deviation of LS$_{rand}$ and LS$_{spec}$. In column three the stopping condition $tObj$ for each neighbour-

Table 7.3: Comparison of PMNOVNS$_{spec}$ and PMNOVNS$_{rand}$. Listed are mean CPU times in seconds, standard deviations in parentheses, median CPU times and the error probabilities of a Wilcoxon rank sum test (U-test).

	PMNOVNS$_{spec}$			PMNOVNS$_{rand}$			
inst.	mean	(σ)	median	mean	(σ)	median	U-test
(1)	**115.4**	(69.6)	92.7	126.0	(74.3)	119.0	0.40
(2)	459.1	(243.8)	428.3	**301.2**	(135.6)	294.3	0.08
(3)	527.3	(218.1)	422.6	**411.4**	(63.3)	423.5	0.30
(4)	326.5	(90.3)	340.9	**143.6**	(180.9)	76.1	< 0.01
(5)	821.5	(698.1)	587.8	**666.6**	(542.8)	646.1	0.46
(6)	991.3	(218.8)	940.5	**499.4**	(161.8)	461.9	< 0.01
(7)	407.0	(399.6)	256.1	**112.3**	(113.2)	73.5	< 0.01
(8)	187.9	(66.9)	184.2	**143.9**	(116.3)	108.6	0.09
(9)	743.0	(401.9)	771.7	**498.2**	(299.8)	497.9	0.11
(10)	1792.0	(245.2)	1883.2	**391.6**	(161.9)	405.5	< 0.01
overall	637.1			**329.4**			

hood and for each instance is listed. Furthermore we performed an one-sided Wilcoxon rank sum test (U-test) to see how significant the results are. Column U-test shows the error probabilities for the hypotheses that the mean CPU time of LS$_{rand}$ is less than the mean CPU time of LS$_{spec}$.

Table 7.4: Comparing efficiency of LS$_{spec}$ and LS$_{rand}$. Listed are average CPU times in seconds, standard deviations in parentheses and the error probabilities of a Wilcoxon rank sum test (U-test).

			LS$_{spec}$		LS$_{rand}$		
inst.	$\mathcal{N}$	*tObj*	mean	(σ)	mean	(σ)	U-test
(1)	$\mathcal{N}_{SW}$	$44 \cdot 10^6$	51.15	(13.88)	**1.18**	(0.39)	< 0.01
	$\mathcal{N}_{SH}$	$37 \cdot 10^5$	50.49	(35.33)	**6.47**	(1.03)	< 0.01
	$\mathcal{N}_{INV}$	$255 \cdot 10^3$	35.82	(9.02)	**14.32**	(5.00)	< 0.01
	$\mathcal{N}_{BSW}$	$362 \cdot 10^6$	10.92	(12.54)	**0.16**	(0.20)	< 0.01
	$\mathcal{N}_{BSH}$	$333 \cdot 10^6$	19.05	(26.05)	**0.40**	(0.27)	0.07
(5)	$\mathcal{N}_{SW}$	$40 \cdot 10^4$	240.44	(25.00)	**5.19**	(1.07)	< 0.01
	$\mathcal{N}_{SH}$	$35 \cdot 10^4$	261.15	(75.29)	**40.70**	(10.53)	< 0.01
	$\mathcal{N}_{INV}$	$25 \cdot 10^4$	303.58	(134.48)	**48.12**	(12.96)	< 0.01
	$\mathcal{N}_{BSW}$	$130 \cdot 10^6$	101.84	(136.05)	**0.16**	(0.08)	< 0.01
	$\mathcal{N}_{BSH}$	$160 \cdot 10^6$	96.44	(92.87)	**0.21**	(0.06)	< 0.01

Now we can detect a considerable difference between LS$_{rand}$ and LS$_{spec}$. For all considered comparisons the mean CPU time is clearly better and for nine out of ten comparisons a significantly better result can be obtained. If comparing these results with the results presented in Table 7.3, we notice that the randomised examination order has a major influence on the performance of LS at the beginning of program execution. However, this dominant influence disappears and the difference between a randomised examination and a specific examination gets smaller after a longer runtime.

7.4 Best Obtained Neighbourhood-Orders for General VNS Scheme

This section presents the best obtained neighbourhood-orders for General VNS scheme which were determined in each run of PMNOVNS$_{rand}$. Figure 7.2 summarises these results in a tree diagram. The nodes of the tree diagram represent the number of occurrences of each neighbourhood-order. The edges indicate the used neighbourhoods in the neighbourhood-order. Because five neighbourhoods are used within this work, 120 different neighbourhood-orders are possible. In the root node the overall number of determined neighbourhood-orders is represented, which corresponds to the 100 performed runs. Consider that not all neighbourhoods of a neighbourhood-order must be used. For example in five runs only neighbourhood $\mathcal{N}_{SW}$ was necessary to achieve $tObj$.

According to the rule of thumb, neighbourhoods are ordered by increasing size. However, this rule of thumb cannot be used, because neighbourhoods with equal size exist. If considering the time needed for a complete examination of a neighbourhood, the decision $\mathcal{N}_1 = \mathcal{N}_{SW}$ would be the result. However, from the results presented in Figure 7.2 we can see for neighbourhood $\mathcal{N}_1$ of the neighbourhood-order a clear dominance of $\mathcal{N}_{INV}$. In 90 out of the 100 runs $\mathcal{N}_1 = \mathcal{N}_{INV}$ and only in 10 runs $\mathcal{N}_{SW}$ is most qualified.

As the dominant neighbourhood-orders use $\mathcal{N}_1 = \mathcal{N}_{INV}$, we inspect the results of them in more detail. Here we can see an advantage of $\mathcal{N}_2 = \mathcal{N}_{SW}$, which is the best in 28 runs. However, no clear best neighbourhood can be determined for $\mathcal{N}_2$. Surprisingly, even $\mathcal{N}_{BSH}$ is most qualified in 11% of all runs. When looking at neighbourhood $\mathcal{N}_3$, no best neighbourhood can be determined. Thus, we can conclude that the first used neighbourhood $\mathcal{N}_1$ has a dominant influence on the performance of General VNS scheme. For all other neighbourhoods $\mathcal{N}_2, \ldots, \mathcal{N}_{max}$ the influence on the performance drops rapidly. Obviously the last neighbourhoods of a neighbourhood-order are used so rarely that stochastic factors dominate.

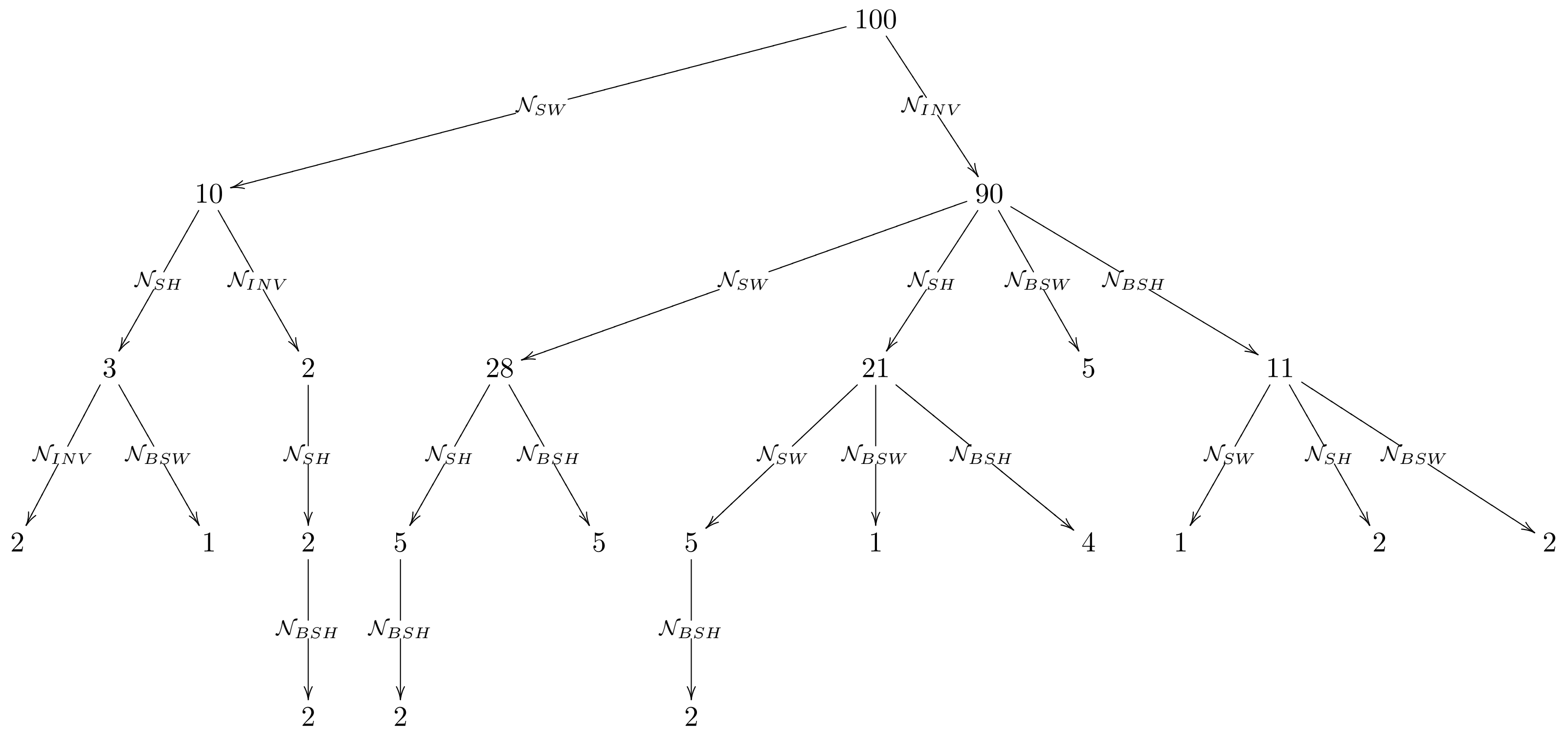

Fig. 7.2: Best obtained static neighbourhood-orders for General VNS scheme

7.5 Experimental Results of TRRVNS

TRRVNS was designed especially for an efficient parallelisation. Because no concrete, best neighbourhood-order for General VNS scheme can be identified, we abandon a comparison with a sequential General VNS scheme. Thus, we will compare TRRVNS with PMNOVNS$_{rand}$. For the parameters of TRRVNS we used $maxRetries = 5$, the initial value of t was set to 0.25 and for the function adaption(t) we used $t = t \cdot 2$. This parameters were determined in preliminary tests.

In Table 7.5 the average CPU times, standard deviations and median CPU times are listed. To determine significant differences, we performed a one-sided Wilcoxon rank sum test (U-test). Column U-test lists the error probabilities for the hypotheses that the smaller mean CPU time (depicted in bold) is less than the larger mean CPU time. Finally, in the last row the average CPU times over all instances are listed.

Table 7.5: Comparison of TRRVNS and PMNOVNS$_{rand}$. Listed are average CPU times in seconds, standard deviations in parentheses and median CPU times as well as the error probabilities of a Wilcoxon rank sum test (U-test).

	PMNOVNS$_{rand}$			TRRVNS			
inst.	mean	(σ)	median	mean	(σ)	median	U-test
(1)	**126.0**	(74.3)	119.0	231.3	(319.6)	110.4	0.37
(2)	301.2	(135.6)	294.3	**128.9**	(159.5)	48.9	< 0.01
(3)	411.4	(63.3)	423.5	**47.5**	(11.0)	48.7	< 0.01
(4)	**143.6**	(180.9)	76.1	1221.1	(840.1)	1443.5	< 0.01
(5)	666.6	(542.8)	646.1	**300.2**	(199.2)	207.8	0.11
(6)	499.4	(161.8)	461.9	**122.6**	(24.6)	121.9	< 0.01
(7)	**112.3**	(113.2)	73.5	481.7	(212.6)	411.8	< 0.01
(8)	143.9	(116.3)	108.6	**109.0**	(32.3)	101.8	0.44
(9)	**498.2**	(299.8)	497.9	621.8	(455.4)	480.1	0.37
(10)	391.6	(161.9)	405.5	**123.1**	(71.9)	104.6	< 0.01
overall	**329.4**			338.7			

When looking at the results presented in Table 7.5, no clear difference between both algorithms is observable. Only in six of ten instances the mean CPU time of TRRVNS is better compared with PMNOVNS$_{rand}$. When comparing the median CPU times, TRRVNS is better in eight instances, however, for some instances the difference is only marginal. On the other hand, if looking at the average CPU times over all performed runs, PMNOVNS$_{rand}$ is slightly better. Furthermore, in four instances significantly better results can be obtained by TRRVNS, whereas PMNOVNS$_{rand}$ can deliver in two of the ten instances significantly better results.

However, an interesting result is that no performance benefit of PMNOVNS$_{rand}$ is detectable, although PMNOVNS$_{rand}$ performs all possible neighbourhood-orders in parallel. Thus, even when using the best neighbourhood-order in each run, General VNS scheme cannot outperform TRRVNS. Therefore, if a best neighbourhood-order is not known, it might be better to use TRRVNS instead of General VNS scheme. Furthermore, we can recognise for some instances substantial differences of the mean CPU times. Thus, we can conclude that both approaches have strengths and weaknesses. If it is possible to exploit the strengths of both algorithms without taking over their weaknesses, a considerable performance improvement of VNS can be obtained.

7.6 Experimental Results of PEGVNS

Finally, here we present the results of PEGVNS and compare them with the already presented results of the other algorithms. To investigate the performance of PEGVNS we run it with different number of processes p, denoted as PEGVNS$_p$. Consider that the master, who coordinates the computation and determines the best solution in each iteration, is not included in p. In total we run PEGVNS with 2, 4, 8 and 16 processes. In addition to the performance investigation, we will also analyse the generated neighbourhood-usage of PEGVNS in detail. For the parameter settings of PEGVNS$_p$ we used the same setting as for TRRVNS, except we set parameter $maxRetries = 3$. These parameters were determined in preliminary tests.

Tab 7.6 presents the overall results. For each algorithm the average CPU times over 10 runs and standard deviations are listed. Furthermore, in the second last last column the average CPU times over all instances are listed. For PEGVNS the CPU time of a single process is used as an estimation of the parallel execution time T_p. Finally, the last column represents the costs of PEGVNS$_p$.

At first, we investigate the maximum possible speedup considering only the excess of computation and excluding all other parallelisation overheads. To do this, we compare the average CPU times over all instances of PEGVNS$_p$ with TRRVNS. Figure 7.3 shows the maximum possible speedup for different number of processes. Here we can see that for $p > 4$ the possible speedup goes down abruptly. Moreover, even the speedup for $p \leq 4$ is not optimal. For example PEGVNS$_2$ can only achieve a maximum speedup of 1.58, whereas the optimal speedup would be 2. In addition this value gets still worse when taking into account the whole parallelisation overhead. As we can see from Table 7.6, PEGVNS$_{16}$ can generate the most efficient neighbourhood-usages when comparing the CPU time over all instances. Although efficient neighbourhood-usages exist, the algorithms PEGVNS$_p$ with $p \leq 8$ are not able to find an efficient enough neighbourhood-usage, so that a linear speedup can be achieved. Thus, we can conclude that a reduction of the CPU time with increasing size of processes p is possible, but from the viewpoint of an efficient parallelisation not optimal.

Table 7.6: Comparison of PMNOVNS, TRRVNS and PEGVNS with different settings. Listed are average CPU times in seconds and the standard deviations in parentheses. For the parallel algorithm PEGVNS the CPU time of a single process is given. In the second last column the average CPU times over all runs and in the last column the costs of PEGVNS are listed.

inst.	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	oa	costs
PMNOVNS$_{spec}$	115.4 *(69.6)*	459.1 *(243.8)*	527.3 *(218.1)*	326.5 *(90.3)*	821.5 *(698.1)*	991.3 *(218.8)*	407.0 *(399.6)*	187.9 *(65.9)*	743.0 *(401.9)*	1792.0 *(245.2)*	637.1	–
PMNOVNS$_{rand}$	126.0 *(74.3)*	301.2 *(135.6)*	411.4 *(63.3)*	143.6 *(180.9)*	666.6 *(542.8)*	499.4 *(161.8)*	112.3 *(113.2)*	143.9 *(116.3)*	498.2 *(299.8)*	391.6 *(161.9)*	329.4	–
TRRVNS	231.3 *(319.6)*	128.9 *(159.5)*	47.5 *(11.0)*	1221.1 *(840.1)*	300.2 *(199.2)*	122.6 *(24.6)*	481.7 *(212.6)*	109.0 *(32.3)*	621.8 *(455.4)*	123.1 *(71.9)*	338.7	–
PEGVNS$_2$	195.1 *(200.2)*	131.0 *(130.4)*	25.5 *(4.7)*	946.7 *(1175.9)*	158.3 *(69.7)*	71.1 *(19.5)*	273.7 *(138.6)*	61.5 *(9.0)*	199.2 *(102.2)*	82.0 *(36.6)*	214.4	428.8
PEGVNS$_4$	115.2 *(69.3)*	35.6 *(31.4)*	19.1 *(7.5)*	234.6 *(205.7)*	108.2 *(83.9)*	39.5 *(3.9)*	226.7 *(222.0)*	40.4 *(15.9)*	209.0 *(206.0)*	46.7 *(21.4)*	107.5	429.9
PEGVNS$_8$	62.4 *(54.5)*	15.1 *(3.3)*	13.0 *(1.3)*	274.1 *(241.3)*	84.3 *(47.8)*	30.7 *(2.4)*	113.5 *(53.4)*	30.9 *(6.7)*	97.6 *(64.7)*	38.6 *(30.6)*	76.0	608.1
PEGVNS$_{16}$	39.4 *(39.4)*	12.8 *(2.3)*	10.3 *(0.9)*	108.5 *(90.0)*	54.5 *(54.0)*	25.4 *(1.5)*	59.6 *(24.0)*	17.9 *(2.1)*	50.7 *(15.2)*	46.7 *(69.0)*	42.6	681.0

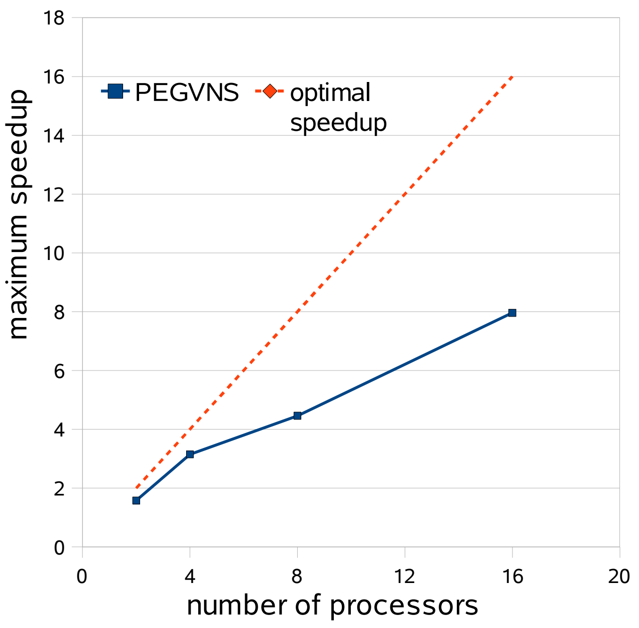

Fig. 7.3: Maximum possible speedup of PEGVNS$_p$ compared with TRRVNS considering only the excess of computation

In each iteration of PEGVNS only the solution generated by the most efficient neighbourhood is used for the future search. Thus in each iteration only a single neighbourhood can contribute to a solution improvement. To analyse the appearance of the generated neighbourhood-usages of PEGVNS, we determined the relative frequency of each neighbourhood. The relative frequency of a neighbourhood is the overall number of contributions to a solution improvement of a neighbourhood divided by the total number of solution improvements of all neighbourhoods. Note that we only considered the first randomly selected neighbourhood of each process for the relative frequency. This is acceptable, because an additional neighbourhood is only used in rare situations to utilize the processor for the remaining time in the case of a local minimum. Figure 7.4 shows the average, relative frequencies over 10 runs for PEGVNS$_{16}$. PEGVNS$_{16}$ was selected, because it generated the best neighbourhood-usages.

In Figure 7.4 we can see that the relative frequencies of $\mathcal{N}_{INV}$ and $\mathcal{N}_{SW}$ are the highest, which accords with the best found neighbourhood-orders of PMNOVNS$_{rand}$ (see Figure 7.2). However, in comparison with PMNOVNS$_{rand}$ and TRRVNS a clear difference can be determined. Because in TRRVNS all neighbourhoods are chosen randomly, the relative frequencies of all neighbourhoods are approximately equal. For PMNOVNS$_{rand}$ a direct comparison is not possible. But because in PMNOVNS$_{rand}$ only in cases of a local minimum another neighbourhood is used and then only a

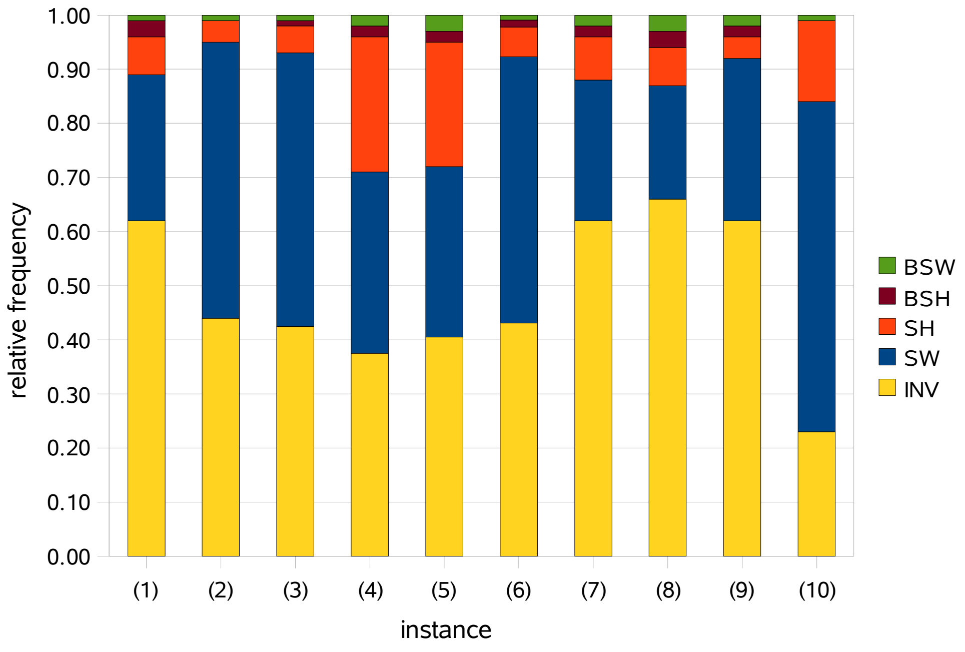

Fig. 7.4: Relative frequencies of neighbourhoods $\mathcal{N}_{INV}$ (a), $\mathcal{N}_{SW}$ (b), $\mathcal{N}_{SH}$ (c), $\mathcal{N}_{BSH}$ (d) and $\mathcal{N}_{BSW}$ (e) for instances (1)-(10) of PEGVNS$_{16}$.

single iteration is performed, the first neighbourhood will mostly be used. Taking further into account that for PEGVNS during a single iteration one neighbourhood is in general applied several times, then the relative frequency of a single neighbourhood of PMNOVNS$_{rand}$ is approximately 1. All other neighbourhoods have a relative frequency of approximately 0.

Because the generated neighbourhood-usage of PEGVNS$_{16}$ delivers the best results, we can conclude that neither a concentration on a single neighbourhood, nor a pure random neighbourhood selection causes the best results. Rather, a random selection of neighbourhoods with a boosted selection of promising neighbourhoods and a reduced selection of unpromising neighbourhoods is most efficient. Figure 7.5 presents an example of this effect. In this figure the relative frequencies of neighbourhoods for PEGVNS$_p$ are depicted for instance (1). In Table 7.6 we can see that with increasing number of processes p, the CPU time decreases and therefore the efficiency of the generated neighbourhood-usage increases. When comparing this with the corresponding graphs in Fig 7.5 we can see that the relative frequency of the promising neighbourhood $\mathcal{N}_{INV}$ increases, whereas the relative frequency of poor neighbourhoods decreases.

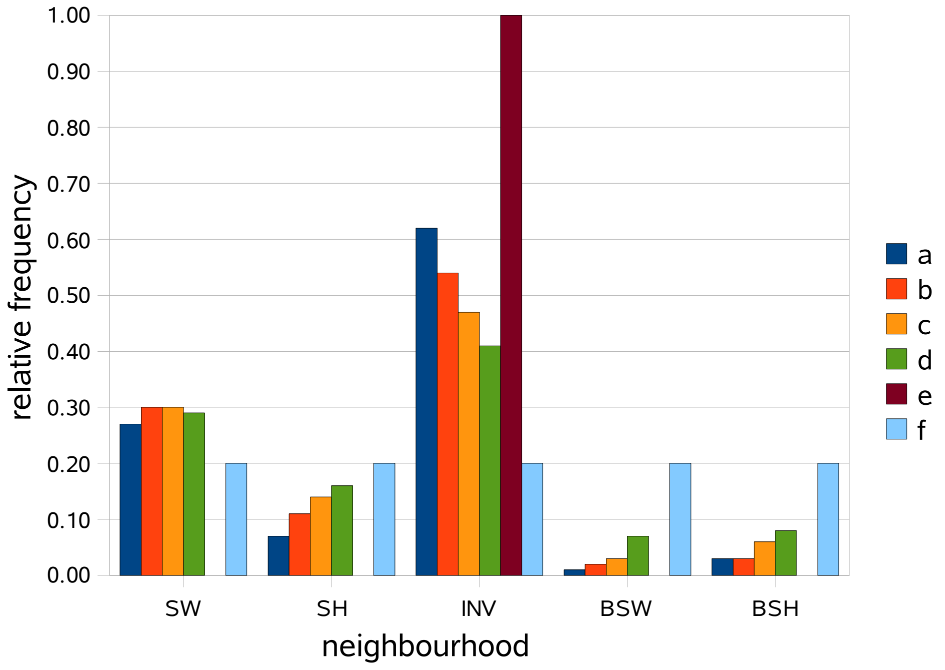

Fig. 7.5: Comparing relative frequencies of neighbourhoods $\mathcal{N}_{SW}$, $\mathcal{N}_{SH}$, $\mathcal{N}_{INV}$, $\mathcal{N}_{BSW}$ and $\mathcal{N}_{BSH}$ generated by algorithms PEGVNS$_{16}$ (a), PEGVNS$_{8}$ (b), PEGVNS$_{4}$ (c) and PEGVNS$_{2}$ (d) for instance (1). For comparison the approximation of the relative frequencies of PMNOVNS$_{rand}$ (e) and TRRVNS (f) are depicted.

8 Conclusions and Future Work

8.1 Conclusions

This thesis proposed a new parallel Variable Neighbourhood Search (VNS) approach called Parallel Efficiency Guided VNS (PEGVNS) for solving the Car Sequencing Problem. The main idea of PEGVNS for speeding up computations is based on using the additional computing power of a parallel computer to find an efficient neighbourhood-usage. Starting point of this work was the question in which order neighbourhoods should be applied in VNS. This question is related to the efficiency of a neighbourhood examination, i.e. the relation of computation time for examining a neighbourhood and solution improvement of a neighbourhood. For this purpose, we both adapted metrics proposed in other works and introduced a new measure for evaluating the efficiency of a neighbourhood as well as an efficient neighbourhood-usage. Keeping this theoretical foundation in mind, we evolved PEGVNS which supports finding efficient neighbourhood-usages.

Experimental results revealed that an application of PEGVNS to the Car Sequencing Problem can substantially reduce the computation time compared with a single processor variant of PEGVNS and compared with a conventional implementation of a General VNS scheme. Furthermore, it can be observed that no general optimal neighbourhood-usage exists. Nevertheless, some neighbourhoods are selected more often than others according to the results produced by PEGVNS. In addition the tests showed that no single neighbourhood dominates the others and complex neighbourhoods also contribute to the final solution. Furthermore, the results revealed that beside an efficient neighbourhood-usage, evaluating a single neighbourhood with different randomised examination orders in parallel is a further reason for the reduction of the computation time of PEGVNS.

8.2 Future Work

The experimental results presented within this work showed that PEGVNS can only achieve a sublinear speedup. The main reason for this lies in the fact that although a multitude of solutions is computed during the parallel search phase, the further iteration of PEGVNS only relies on the best last so far found solution whereas all other solutions are disregarded. Therefore, we achieve an efficiency of $\max_{\mathcal{N}_k \in \mathfrak{N}}(\mathcal{E}((\mathcal{N}_k), x^{i-1}))$ for iteration i. But ideally, a parallel VNS approach combines all generated solution improvements and continues its computation with this solution. In this optimal case an efficiency of $\sum_{\mathcal{N}_k \in \mathfrak{N}}(\mathcal{E}((\mathcal{N}_k), x^{i-1}))$ can be achieved in each iteration. However, a merging of different solutions is tricky. To avoid a time-consuming repair function, the new solution has still be valid and of course should be better than the original solutions.

Because Car Sequencing Problem considers "sliding windows" with length m_c, the violations at position i depend only on the configurations $(\chi_{i-m_c+1}, \ldots, \chi_i)$. Therefore, a promising approach would be a partitioning of neighbourhoods into several sub-neighbourhoods. For example, we could divide $\mathcal{N}_k$ into $\mathcal{N}_{k1}$ and $\mathcal{N}_{k2}$ where only configurations $(\chi_1, \ldots, \chi_{\alpha_1})$ and $(\chi_{\alpha_2}, \ldots, \chi_n)$ are allowed to be changed, with $\alpha_2 - \alpha_1 \geq \max_{c \in C}(m_c)$, respectively. Furthermore, we get a sub-neighbourhood $\mathcal{N}_{k3} = (\mathcal{N}_k \setminus \mathcal{N}_{k1}) \setminus \mathcal{N}_{k2}$. With this neighbourhood partitioning it is possible to merge a solution x_{k1} generated by $\mathcal{N}_{k1}$ with a solution x_{k2} generated by $\mathcal{N}_{k2}$ without introducing new violations and resulting in a permitted solution. However, for solution generated by $\mathcal{N}_{k3}$ no solution merge is possible.

Bibliography

[1] E. Alba and G. Luque. Measuring the performance of parallel metaheuristics. In E. Alba, editor, *Parallel Metaheuristics: A New Class of Algorithms*, pages 43–62. Wiley-Interscience, 2005.

[2] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *American Federation of Information Processing Societies Conference Proceedings*, volume 30, pages 483–485, Atlantic City, USA, 1967.

[3] R. S. Barr and B. L. Hickman. Reporting computational experiments with parallel algorithms: Issues, measures, and experts' opinions. *Operations Research Society of America Journal on Computing*, 5(1):2–18, 1993.

[4] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization.* Operations research/Computer Science Interfaces. Springer Verlag, 2008.

[5] N. Boysen, M. Fliedner, and A. Scholl. Production planning of mixed-model assembly lines: Overview and extensions. Jena Research Papers in Business and Economics 06/2007, Friedrich-Schiller-University Jena, Department of Economics and Business Administration, 2007. Available online at `http://ideas.repec.org/p/jen/jenjbe/2007-06.html`, last visited on September 13, 2008.

[6] N. Boysen, M. Fliedner, and A. Scholl. Sequencing mixed-model assembly lines to minimize part inventory cost. *OR Sprectrum*, 30(3):611–633, 2008.

[7] N. Boysen, M. Fliedner, and A. Scholl. Sequencing mixed-model assembly lines: Survey, classification and model critique. *European Journal of Operational Research*, 192(2):349–373, 2009.

[8] J. Cheng, Y. Lu, G. Puskorius, S. Bergeon, and J. Xiao. Vehicle sequencing based on evolutionary computation. In *Proceedings of 1999 Congress on Evolutionary Computation, Washington D. C.*, volume 2, pages 1207–1214, 1999.

[9] T. Crainic, P. Gendreau, P. Hansen, and N. Mladenović. Cooperative parallel variable neighbourhood search for the p-median. *Journal of Heuristics*, 10(3):293–314, 2004.

[10] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *Proceedings of the European Conference on Artificial Intelligence (ECAI-88)*, pages 290–295, 1988.

[11] A. Drexl and A. Kimms. Sequencing JIT mixed-model assembly lines under station-load and part-usage constraints. *Management Science*, 47(3):480–491, 2001.

[12] A. Drexl, A. Kimms, and L. Matthießen. Algorithms for the car sequencing and the level scheduling problem. *Journal of Scheduling*, 9(2):153–176, 2006.

[13] B. Estellon, F. Gardi, and K. Nouioua. Two local search approaches for solving real-life car sequencing problems. *European Journal of Operational Research*, 191(3):928–944, 2008.

[14] M. Fliedner and N. Boysen. Solving the car sequencing problem via branch & bound. *European Journal of Operational Research*, 191(3):1023–1042, 2008.

[15] F. García-López, B. Melián-Batista, J. A. Moreno-Pérez, and J. M. Moreno-Vega. The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics*, 8(3):375–388, 2002.

[16] I. P. Gent and T. Walsh. CSPlib: A benchmark library for constraints. In J. Joxan, editor, *Principles and Practice of Constraint Programming – CP'99*, volume 1713 of *LNCS*, pages 480–481, 1999.

[17] J. Gottlieb, M. Puchta, and C. Solnon. A study of greedy, local search, and ant colony optimization approaches for car sequencing problems. In G. R. Raidl et al., editors, *Applications of Evolutionary Computing*, volume 2611 of *LNCS*, pages 246–257, 2003.

[18] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing: Second Edition.* Addison-Wesley, USA, 2003.

[19] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[20] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In S. Voss, S. Martello, I. Osman, and C. Roucairol, editors, *Metaheuristics, Advances and Trends in Local Search Paradigms for Optimization*, page 433–458. Kluwer, 1999.

[21] P. Hansen and N. Mladenović. A tutorial on variable neighborhood search. Technical Report G-2003-46, Les Cahiers du GERAD, HEC Montréal and GERAD, Canada, 2003.

[22] B. Hu and G. R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. In C. Cotta, A. J. Fernandez, and J. E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics*, Malaga, Spain, 2006.

[23] T. Kis. On the complexity of the car sequencing problem. *Operations Research Letters*, 32(4):331–335, 2004.

[24] MPI Forum. MPI documents. Available online at `http://www.mpi-forum.org/docs/docs.html`, last visited on September 26, 2008.

[25] A. Nguyen. Challenge ROADEF'2005: Car sequencing problem. Available online at `http://www.prism.uvsq.fr/~vdc/ROADEF/CHALLENGES/2005/`, last visited on September 13, 2008.

[26] B. Parello, W. Kabat, and L. Wos. Job-shop scheduling using automated reasoning: a case study of the car sequencing problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.

[27] M. Prandtstetter. Exact and heuristic methods for solving the car sequencing problem. Master's thesis, Vienna University of Technology, Austria, 2005.

[28] M. Prandtstetter and G. R. Raidl. An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *European Journal of Operational Research*, 191(3):1004–1022, 2008.

[29] J. Puchinger and G. R. Raidl. Bringing order into the neighborhoods: relaxation guided variable neighborhood search. *Journal of Heuristics*, 14(5):457–472, 2008.

[30] M. Puchta and J. Gottlieb. Solving car sequencing problems by local optimization. In *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002*, pages 132–142, London, UK, 2002. Springer.

[31] M. Sevkli and M. E. Aydin. Parallel variable neighbourhood search algorithms for job shop scheduling problems. *Institute of Mathematics and its Applications Journal of Management Mathematics*, 18(2):117–133, 2007.

[32] B. M. Smith. Succeed-first or fail-first: A case study in variable and value ordering. In *third Conference on the Practical Applications of Constraint Technology PACT'97*, page 321–330, 1997.

[33] C. Solnon. Solving permutation constraint satisfaction problems with artificial ants. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 118–122, Amsterdam, Netherlands, 2000.

[34] C. Solnon, V. Cung, A. Nguyen, and C. Artigues. The car sequencing problem: overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *European Journal of Operational Research*, 191(3):912–927, 2008.

[35] T. Warwick and E. P. K. Tsang. Tackling car sequencing problems using a generic genetic algorithm strategy. *Evolutionary Computation*, 3(3):267–298, 1995.

[36] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers: international edition, second edition.* Prentice Hall, USA, 2005.