universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

„Variable Neighborhood Search for the Multi-Level Capacitated Lotsizing Problem“

Verfasserin

Sylvia-Maria Pichler

angestrebter akademischer Grad

Magistra der Sozial- und Wirtschaftswissenschaften
(Mag. rer. soc. oec.)

Wien, im Februar 2011

Studienkennzahl lt. Studienblatt:	157
Studienrichtung lt. Studienblatt:	Internationale Betriebswirtschaft
Betreuer:	Privatdozent Dr. Christian Almeder

Abstract

The Multi-Level Capacitated Lotsizing Problem (MLCLSP) depicts the important decision in production planning of determining adequate lot sizes from final products onward, to subassemblies, parts and raw materials, all the while assuming limited capacities of the resources employed for manufacture. It is an *NP*-hard problem where exact methods fail in solving larger – one could say realistic – problem instances. Sequential approaches that tackle the problem item by item and postpone capacity considerations dominate current practice; approximate solution methods abound throughout the literature. Local search and metaheuristics based on it constitute a class of approximate methods well-equipped to take on the challenge of eventually replacing the trial-and-error process that impedes manufacturing companies in establishing feasible and cost-minimal production plans.

This thesis presents a study of local search based procedures for solving the MLCLSP. Eight different neighborhood structures, resulting from manipulations of the setup variables, are devised and evaluated. Fundamental options when designing an iterative improvement algorithm, such as best-improvement versus first-improvement step functions or the inclusion of infeasible solutions during the search are explored and compared.

Although only the *Switch* move, which alters the value of a single setup value, is convincing as a stand-alone neighborhood structure, the other neighborhoods can in any case be employed for the perturbation of solutions during the shaking step of a Variable Neighborhood Search (VNS). The implementation of this metaheuristic, shaped by the findings from testing the basic local search variants, led to mixed results. The procedure designed to tackle the MLCLSP cannot outperform the compared heuristics. Neither does it produce results that are terribly off – the average gap to the best known solutions settles around four percent over all problem classes tested. Nonetheless, the impression is supported that the VNS procedure is a robust method leading to good, sometimes even very good solutions at a regular basis that is amenable to further adjustments and thus eventually becoming a serious competitor for existing methods dealing with multi-level capacitated lotsizing decisions.

Zusammenfassung

Das dynamische mehrstufige kapazitierte Losgrößenproblem (MLCLSP) behandelt im Rahmen der Produktionsplanung die wichtige Entscheidung über die optimalen Losgrößen, angefangen bei Endprodukten über Komponenten bis hin zu Rohstoffen, bei gleichzeitiger Berücksichtigung beschränkter Kapazitäten der zur Produktion benötigten Ressourcen. Da es sich um ein *NP*-schweres Problem handelt, stoßen exakte Lösungsverfahren an ihre Grenzen, sobald die Problemdimensionen ein größeres – man könnte durchaus sagen realistisches – Ausmaß erreichen. In der Praxis dominieren deshalb Methoden, die die Losgrößen der einzelnen Produkte sequenziell festlegen und überdies etwaige Kapazitätsbeschränkungen im Nachhinein, falls überhaupt, berücksichtigen. In der Literatur finden sich zahlreiche approximative Ansätze zur Lösung dieses komplexen betriebswirtschaftlichen Problems. Lokale Suche und auf ihr basierende Metaheuristiken stellen vielversprechende Werkzeuge dar, um die Defizite der aktuell eingesetzten *Trial-and-Error* Ansätze zu beheben und letzten Endes zulässige sowie kostenoptimale Produktionspläne zu erstellen.

Die in dieser Diplomarbeit vorgestellte Studie beschäftigt sich mit lokalen Suchverfahren für das MLCLSP. Acht Nachbarschaftsstrukturen, die sich aus einer Veränderung der Rüstvariablen ergeben, werden präsentiert und evaluiert. Grundlegende Optionen bei der Gestaltung eines iterativen Verbesserungsverfahrens, wie beispielsweise unterschiedliche Schrittfunktionen oder die temporäre Berücksichtigung unzulässiger Lösungen, werden getestet und verglichen.

Obwohl nur die *Switch* Nachbarschaft, die durch das Ändern einer einzigen Rüstvariable definiert wird, wirklich überzeugende Resultate liefert, können die übrigen Nachbarschaftsstrukturen durchaus als Perturbationsmechanismen im Rahmen einer Variablen Nachbarschaftssuche (VNS) zum Einsatz kommen. Die Implementierung dieser Metaheuristik, geprägt von den Ergebnissen der einfachen lokalen Suchverfahren, kann allerdings nicht vollkommen überzeugen. Die entwickelte VNS Variante kann die Lösungsgüte anderer zum Vergleich herangezogener Lösungsverfahren nicht erreichen und benötigt relativ lange Laufzeiten. Andererseits sind die Ergebnisse mit einer durchschnittlichen Abweichung zur besten bekannten Lösung von etwa vier Prozent über sämtliche untersuchte Problemklassen weit entfernt von einem Totalversagen. Es überwiegt der Eindruck, dass es sich um eine robuste Methode handelt, die in der Lage ist,

Lösungen von hoher, teils sehr hoher Qualität nicht nur in Ausnahmefällen zu liefern. Etwaige Nachjustierungen könnten das Verfahren durchaus zu einem ernstzunehmenden Konkurrenten für bereits existierende Lösungsmethoden für das MLCLSP machen.

Contents

List of Figures

List of Tables

Abbreviations

ACO	Ant Colony Optimization
BOM	Bill of Material
CLSD	Capacitated Lotsizing Problem with Sequence Dependent Setups
CLSP	Capacitated Lotsizing Problem
CLSPL	Capacitated Lotsizing Problem with Linked Lot Sizes
CRP	Capacity Requirements Planning
CSLP	Continuous Setup Lotsizing Problem
DLSP	Discrete Lotsizing and Scheduling Problem
EA	Evolutionary Algorithm
ELSP	Economic Lot Scheduling Problem
EOQ	Economic Order Quantity
GA	Genetic Algorithm
GLSP	General Lotsizing and Scheduling Problem
GRASP	Greedy Randomized Adaptive Search Procedure
ILS	Iterated Local Search
LP	Linear Program
MIP	Mixed-Integer Program
MLCLSP	Multi-Level Capacitated Lotsizing Problem
MLDLSP	Multi-Level Discrete Lotsizing and Scheduling Problem
MLULSP	Multi-Level Uncapacitated Lotsizing Problem
MLPLSP	Multi-Level Proportional Lotsizing and Scheduling Problem
MPS	Master Production Schedule / Scheduling
MRP	Material Requirements Planning
MRP II	Manufacturing Resource Planning
PLSP	Proportional Lotsizing and Scheduling Problem
PPC	Production Planning and Control
SA	Simulated Annealing
SLULSP	Single-Level Uncapacitated Lotsizing Problem
TS	Tabu Search
VNS	Variable Neighborhood Search
W-W	Wagner-Within Problem

Author's note:

Lotsizing, lot sizing, or lot-sizing? All three variants can be found throughout the vast literature on the topic. We will stick to the first one throughout this thesis.

1. Introduction

Lotsizing is a crucial part of the mid- to short-term production planning process. Given the demand for its products, how many units of which item should a company produce at a time, and when exactly should it do so? By deciding carefully on this, any manufacturing company can ensure a smooth production run and keep the resulting costs at a low or even minimum level. This is achieved through balancing inventory holding costs against costs incurred when setting up a production facility. The former arise when lot sizes are chosen large enough to not only cover immediate, but also future demands, which entails longer production runs and therefore less frequent setups. A more flexible policy with smaller lot sizes avoids carrying large inventories and the associated costs, but involves more resources spent on the frequent change of setups. The economic relevance of lotsizing is therefore grounded in the potential cost savings stemming from a trade-off between setup costs on the one hand, and inventory costs on the other hand.

The Multi-Level Capacitated Lotsizing Problem (MLCLSP) is one of the more realistic mathematical lotsizing models, due to its assumption of finite resource capacities and general multi-level product structures, rendering it however one of the more complex models. Thus, from an operations research viewpoint, the MLCLSP is interesting and challenging because it belongs to the hard-to-solve class of *NP*-hard problems. Simply put, this means that the traditional arsenal of exact mathematical methods will soon be at a loss the larger the problem instances become, as computational times rise exponentially with the size of the instance. A real-world manufacturing company might be faced with a bill of material involving hundreds of components, parts, and final products. Solving instances of such big dimensions to proven optimality in acceptable time is impossible. In other words, there is a reason why current practice relies on the use of commercial production planning and control software with simplified calculations, accepting however at the same time some severe weaknesses due to the sequential consideration of items and the ex-post or non-consideration of capacity restrictions. This leaves an enormous potential for improvement and an incentive for research as to practically relevant solution methods that deliver solid and feasible near-optimal production plans.

The application of specialized heuristics and metaheuristics constitutes an alternative way to tackle such complex problems, sacrificing proven optimality for sufficiently good solutions reached with much less effort. This thesis employs a Variable Neighborhood

Search (VNS) in solving the MLCLSP. As local search is one of the core components of this metaheuristic, a great deal of this work is dedicated to it. First of all, eight different neighborhoods that are based on the manipulation of the binary setup variables are devised, ranging from a simple single switch to a complex double swap move. The various move mechanisms are evaluated based on their success in finding better solutions, and on how long this takes on average. While testing the neighborhoods within a basic local search procedure on a wide range of problem instances, several fundamental design options, such as best-improvement versus first-improvement, or random neighbor samples versus systematic exhaustive neighbor checking, were weighed against each other. Sequential combinations of two or more neighborhoods have been examined also, the insights gained from these preliminary test runs shaping the implementation of the VNS. The main question that this thesis hopes to answer is what works well for solving the MLCLSP, and what does not in terms of local search methods.

From a metaheuristic viewpoint, another contribution of the thesis is finding out whether this specific metaheuristic, Variable Neighborhood Search, works well for the MLCLSP, as previously – as far as the author knows – it has been applied mostly to other production related problems, in particular scheduling, but not to this exact lotsizing problem.[1]

The thesis is structured as follows. Theoretical background on lotsizing is provided in chapter 2, including a classification and precise descriptions of lotsizing models, in particular the Multi-Level Capacitated Lotsizing Problem, as well as mathematical formulations. The second theoretical pillar, local search, will be dealt with thoroughly in chapter 3, where a detailed description of the basic procedure and explanations of important ideas and concepts are provided. Metaheuristic extensions, particularly Variable Neighborhood Search, are described in detail, before a literature review with an emphasis on metaheuristic approaches to lotsizing problems is presented in chapter 4. Chapter 5 introduces eight different local search neighborhoods for the MLCLSP and summarizes the results from various first test runs, before the implementation details and the results of the VNS are finally given in chapter 6. Chapter 7 concludes.

[1] Hindi et al. (2003) employed a VNS for the *single-level* CLSP, as well as Almada-Lobo and James (2008) and Almada-Lobo et al. (2008) for the single-level CLSD. Several other scheduling examples can be found in Hansen et al. (2008).

2. Lotsizing

2.1 *How Many Parts to Make at Once?*

Ford W. Harris answered the title question of his article with the up until today well-known Economic Order Quantity, which marks the beginning of growing interest in and research on the lotsizing decision. While Harris stated in 1913 regarding his EOQ formula that *"...it may be well to say that the method given is not rigorously accurate, for many minor factors have purposely been left out of consideration"* (cf. Harris, 1913, p.950), over the course of the last century a variety of models and techniques dealing with this crucial decision for any manufacturing company has been put forth to account for the variety of possible production environments and constellations a company can face. According to Jans and Degraeve (2005) p.3, *"The power of production planning theory comes from the ability to solve more and more complex industrial problems. Whereas the early models where* [sic] *usually more compact, capturing the main trade-off, the extensions focus more and more on incorporating relevant industrial concerns."* Lotsizing models differ in their underlying assumptions and in the details they incorporate, and for this reason we should give a specific description of the problem this thesis deals with – the Multi-Level Capacitated Lotsizing Problem. We will however first take a look at the lotsizing decision in general.

2.1.1 The Lotsizing Decision in Production Planning

If we imagine a manufacturing company and consider its production-related activities, we can detect several hierarchical levels of decisions that have to be made. We will follow Bahl et al. (1987) and Pochet (2001) for a brief and simple classification. Strategic decisions have a long-term scope and address questions such as what to offer on the market (product mix), where to build plants and warehouses (location), or whether to acquire new equipment (investment). Tactical decisions cover problems with a medium-range impact, such as the design of facilities (layout), contracts with suppliers, and adequate workforce levels. Like strategic choices, tactical decisions rely on aggregate data – demand for product families rather than single products and capacities of entire production lines rather than particular machines, which goes hand in hand with and is justified by the aggregation of *time* (cf. Lang, 2010). A planning horizon of several years for strategic choices and of several months to one year for tactical considerations makes it impossible to use detailed

information that will be first of all available, and much less accurate. Inputs for such decisions are therefore aggregated forecasts with a smaller margin of error. As a third level, operational production planning is concerned with the short-term implementation and execution of plans to reach the goals previously settled on at higher levels. Establishing sequences of operations for each machine and determining exact start and end times of operations are for example carried out at this level. Operational decisions use detailed information and a fine time grid.

Lotsizing problems can arise at several points in medium- to short-term production planning. Determining the production quantities for end products in the course of Master Production Scheduling usually covers a time span of several weeks and is based on forecasted demand. The lot sizes of end products affect directly the demand for the components from which they are assembled. In the course of the subsequent Material Requirements Planning, lots for subassemblies and parts as well as orders for raw materials can be coupled, which thus gives rise to lotsizing problems farther down the product structure. As we will see shortly, lotsizing decisions can also be integrated with sequencing and scheduling decisions. The time span considered in such a case is very short, and the resulting production plan usually covers about a week. Figure 2.1 provides a quick overview of production planning decisions with an emphasis on lotsizing.

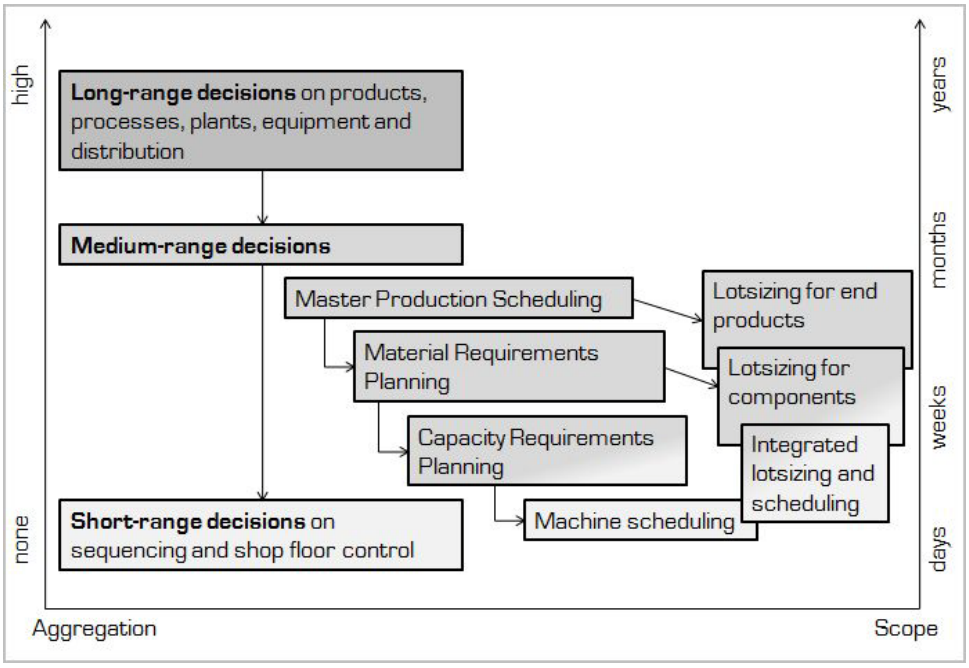

Figure 2.1 – Lotsizing decisions in production planning.
Based on illustrations in Bahl et al. (1987), Tempelmeier (1997) and Lang (2010).

After having seen when and where lotsizing problems occur, we turn our attention to the nature of the problem. What exactly is a lotsizing problem? The reader is asked to take a look at Figure 2.2 below.

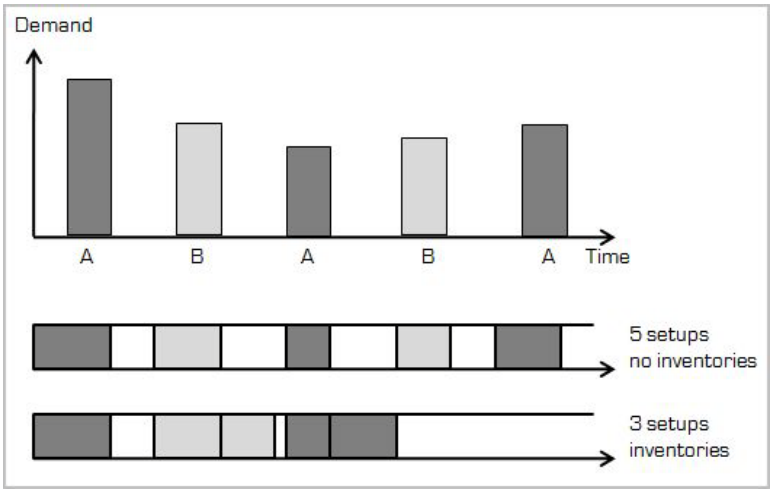

Figure 2.2 – The lotsizing problem.
Adapted from an illustration in Stadtler (2001).

Starting from known demands – be they forecasted for finished goods or calculated for components – the company's goal is to satisfy them and provide the required quantities on time. One way to achieve this is a lot-for-lot policy, which entails in our simple example five setup activities each time a new lot begins on the single resource considered here. Holding the items in inventory is not necessary, as consumption follows production immediately. Another policy would be grouping lots of the same item together, and thus reduce the number of setups. This means however that the items produced to cover future demands must be stored somewhere until they are needed. Both setup activities and inventories incur costs, and lotsizing is concerned with the trade-off between the former and the latter. Choosing small lot sizes entails frequent adjustments to the resources and therefore high setup costs, whereas longer production runs of the same item result in a high proportion of the total costs being caused by holding stock. Figure 2.3 shows a graphical representation of the cost curves depending on the size of a lot or an order respectively, as originally designated in this famous illustration from Harris (1913), depicting the cost trade-off for a single item with a constant demand rate. The optimal lot size q^* minimizes the total costs, composed of the setup costs, Ds/q, and the inventory costs, $qh/2$, where D stands for the total demand over the considered period, and s and h represent the cost per setup and per unit of the item held in stock respectively. The derivative of the total costs

with respect to the lot size yields the well-known *Economic Order Quantity,* $q^* = \sqrt{\frac{2Ds}{h}}$. As we will see shortly, the application of this formula is however restricted to lotsizing problems with quite special assumptions.

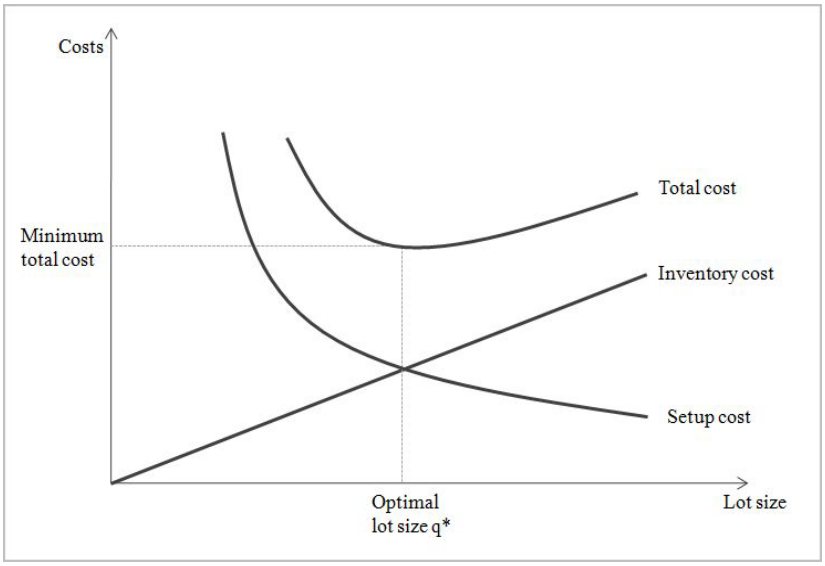

Figure 2.3 – The trade-off between setup and inventory costs.
Adapted from illustrations in Harris (1913).

Apart from the explicit monetary factor, choosing adequate lot sizes affects the practicability of production plans and the ease and efficiency of production runs. *"Formally, lotsizing can be described as the assignment of production resources to production activities, such that agreed upon quantities of products are available for meeting external demand. Lotsizing is important, since it can be used to propose feasible production plans at minimal costs"*, Kuik et al. (1993) write on page 62.

Feasible production plans and minimal costs are our cues to take a look at the handling of lotsizing decisions in practice. Regular commercial production planning and control (PPC) software implements a successive execution of the respective planning steps depicted in Figure 2.1 – master production scheduling, material requirements planning, capacity requirements planning, short-term scheduling (cf. Tempelmeier, 1997). In the Anglo-Saxon world, this approach corresponds to a *closed-loop MRP* (Material Requirements Planning) system, which developed from the item-focused MRP calculations of the 1950s and 1960s by adding resource-focused capacity planning steps. Integration of further, more strategic

aspects of production management, led to *MRP II* (Manufacturing Resource Planning) systems (cf. Toomey, 1996).

The planning starts with lotsizing decisions for end products and proceeds downstream in the product structure. Production and purchase orders for components are calculated level by level based on the master production schedule (MPS), the bill of material (BOM), inventories, lead times, and mostly simple lotsizing rules, such as the EOQ formula or the Silver/Meal and Groff heuristics (cf. Helber, 1995; Tempelmeier, 1997).[2] This sequential item-by-item decomposition of the lotsizing problem can however not guarantee that the resulting costs are minimal, like a simultaneous determination of the lot sizes for all items would. A second serious flaw of commercial PPC systems is the assumption of infinite capacities when planning the production quantities, which means that the schedules might be infeasible. The ex-post consideration of limited capacities in the course of Capacity Requirements Planning (CRP) attempts at fixing infeasibility by shifting lots, and in doing so might in turn neglect precedence relations between the items. Each step in the sequential MRP approach might therefore have to be carried out repeatedly until an acceptable – feasible – production plan emerges, which *"is essentially a trial-and-error process with no guarantee of success"* (cf. Bahl and Ritzman, 1984, p.389). The practical outcome of infeasible production plans is congestion in front of overloaded machines, resulting in long lead times, large amounts of work-in-process, and ultimately low service levels.

The advantage of such PPC systems might be the fast and easy handling of an otherwise very complex problem, but given the crucial importance of choosing adequate lot sizes because of the resulting impact on total costs, sub-optimal and moreover infeasible production plans just will not do in the long run. MRP and MRP II can be regarded as information – rather than optimization – systems with an enormous potential for improvement consequently. For this reason, researchers have come up with a multitude of mathematical models with an explicit optimization goal.

2.1.2 Different Models with Different Assumptions

The Economic Order Quantity of Harris and the MLCLSP treated in this thesis both deal with lotsizing, yet they could not lie further apart in their underlying assumptions. Production environments take on many forms in practice, and so do lotsizing models in

[2] The mentioned heuristics were introduced in Silver and Meal (1973) and Groff (1979) respectively.

theory. Even within one manufacturing company, it can be perfectly sound to employ several different models. Classifications can be found in Bahl et al. (1987), Karimi et al. (2003), and particularly Lang (2010), all of which build the foundation of this section. How do lotsizing models differ then?

Let *static* versus *dynamic demand* be our first classification criterion. Static or stationary demand is assumed to be constant over time, whereas dynamic demand varies with time. The latter requires therefore a contemplation of discrete periods, also called time buckets, usually over a finite planning horizon. Static demand on the other hand can be modeled in a continuous manner over an infinite time horizon. Demand can further be classified according to its *deterministic* or *stochastic* nature.

A quite distinctive characteristic of lotsizing models is whether they assume unlimited capacities of resources (termed *uncapacitated* lotsizing), or limited capacities (*capacitated* lotsizing). We have seen before that a fundamental flaw of current practice is the disregard for capacity considerations during the lotsizing decision and the ensuing sub-optimality of production plans. Yet uncapacitated models are important and persist because *"the ability to solve* [such problems] *has transfer value to more complex problems. The problem also may be sufficiently close to some real manufacturing environments, which have flexible resources (workforce and equipment) and considerable capacity slack."* (cf. Bahl et al., 1987, p.330)

If capacity is however scarce and modeled as such, the time spent for setup activities becomes an issue, as it takes away from capacity for production. Setup times *"represent the capacity lost due to cleaning, preheating, machine adjustments, calibration, inspection, test runs, change in tooling, etc., when the production for a new item starts"*, Jans and Degraeve (2005) specify on page 8. Setup times can be included explicitly in a model; due to the resulting increased complexity in such a case, they are however oftentimes indirectly incorporated via the setup costs. The latter consequently consist of actual expenses for additional material, personnel, etc., and opportunity costs of the lost capacity. Explicit setup times should be seen as an extension of the respective models, rather than a separate class of models.

Assuming limited capacities usually goes hand in hand with planning for *multiple items* that compete for time on the resources and are thus interrelated, although uncapacitated

variants for multi-level lotsizing exist; they constitute one of the problem classes in Bahl et al. (1987) – MLUR (Multiple-Level, Unconstrained-Resources), also known as MLULSP (Multi-Level Uncapacitated Lotsizing Problem). Assuming unrestricted availability of resources in a single-level environment lifts the interdependency between items, resulting in lotsizing problems for *single items* that can be dealt with sequentially.

If multiple items are considered, they can either be from a *single level* of the product structure, i.e. multiple independent final products are considered, or they can be on different levels, i.e. parent-component relationships between the items are present. In such *multi-level*, also called *multi-stage* production systems, end products are assembled from intermediate products (sub-assemblies), which might in turn require raw materials or parts for manufacture. The output of one stage is thus the input for the next stage. This gives rise to two different forms of demand: external or independent demand for end products triggered by the market, and internal or dependent demand for components triggered by scheduled production on superior levels. Some sub-assemblies and parts can however also have external demand, e.g. if they are sold as spare parts. The precedence relations and the exact number of units of any given predecessor required for production of its immediate successor(s) define the product structure, and are captured in the bill of material. Product structures can be categorized as serial, convergent (also: assembly), divergent (disassembly), and general structures. Apart from the items on the first and last levels, which do not have any predecessors / successors at all, each item in a serial structure has exactly one immediate predecessor and one immediate successor. Assembly systems are characterized by each item having possibly several predecessors and exactly one successor, while the opposite is true for divergent structures. General structures do not have any limitations on the number of predecessors and successors.[3] In addition to the possible interdependency between items through limited capacities in case of a capacitated model, production of the individual items is linked through the input-output relationships in multi-level models. Such models are more complex and thus harder to solve than single-level and uncapacitated ones.

Regarding the time structure used for a model, we differentiated above between continuous and discrete time horizons. Discrete periods can further be categorized as macro or micro periods, better known as *large* or *big buckets*, and *small buckets* respectively. While a large

[3] Illustrations of a general and an assembly product structure are provided in chapter 6.1.2 for the description of the test instances used.

bucket model assumes time periods with a length of about a week and an overall planning horizon of usually up to six months, the length of the individual periods in small bucket models corresponds to days, shifts or hours dividing a much shorter overall planning horizon (cf. Drexl and Kimms, 1997). Interpreting each period as such a small time slot, it is a logical conclusion to assume that only one, at most two items can be produced per period. A direct consequence of this is that an unambiguous order of the items to be produced is established. Small bucket models realize thus an integration of lotsizing and scheduling decisions. Large bucket models on the other hand have the distinguishing mark of allowing production of several items per period, without any sequencing decisions being made. They can however be extended in this direction. Imagine the item to be produced first in a period to be the very item that was produced last in the preceding period. The standard large bucket model enforces a setup in each of the respective periods. Skipping the second one and preserving the setup state of the resource is referred to as *setup carryover*. Building in this possibility establishes a partial sequence – the first and last items per period have to be determined. The assumption that setup activities depend on the previous state of the resource, i.e. the item that was produced before influences the necessary adjustments to the machine, leads to the inclusion of *sequence-dependent setup costs and times*. Once present, they inevitably establish a complete sequence of items.

Further differences of lotsizing models result from various extensions to the standard versions. Some models formulate so-called soft constraints, where violations cause penalties rather than infeasibility: instead of fixed capacities in every period, they include overtime decisions; instead of insisting that all demand must be met on time, they allow backlogging, to name two examples. A comprehensive review of extensions can be found in Jans and Degraeve (2005).

With the help of the just mentioned criteria, we can classify some of the well-known lotsizing models. Harris's EOQ formula can be applied to lotsizing problems for a single item with constant demand, where capacities are not an issue and cost parameters are static. The Economic Lot Scheduling Problem (ELSP), see for example Elmaghraby (1978), can be seen as an extension to multiple items sharing a single resource with limited capacities. Wagner and Within (1958) considered lotsizing for a single item with unlimited capacities over a finite planning horizon divided into discrete periods. Demand and costs are accordingly time-varying. This single-level uncapacitated lotsizing problem

(abbreviated to SLULSP) is therefore sometimes referred to as Wagner-Within (W-W) problem.

The Capacitated Lotsizing Problem (CLSP) is the extension of the Wagner-Within problem to multiple items and consequently limited capacities. It is the standard large bucket model; sequencing decisions are not included. The Capacitated Lotsizing Problem with Linked Lot Sizes (CLSPL), see Suerie and Stadtler (2003), extends the CLSP with the possibility of setup carryover, while the Capacitated Lotsizing Problem with Sequence Dependent Setup Costs (CLSD) by Haase (1996) extends the CLSP according to its name. The General Lotsizing and Scheduling Problem (GLSP) of Fleischmann and Meyr (1997) uses a two-fold time structure, where each macro-period is divided into several micro-periods of variable length. A complete sequence of items is established.

Small bucket models for dynamic capacitated lotsizing in a single-level system are the Discrete Lotsizing and Scheduling Problem (DLSP), the Continuous Setup Lotsizing Problem (CSLP), and the Proportional Lotsizing and Scheduling Problem (PLSP). Prominent references are Salomon et al. (1991) for the DLSP, Karmarkar and Schrage (1985) for the CSLP, and Drexl and Haase (1995) for the PLSP. The DLSP allows production of only one item in each period. The production is furthermore assumed to be "all or nothing": the total capacity available per period is used for production of the scheduled item. The CSLP is equivalent to the DLSP without the "all or nothing" requirement, which can lead to periods with some slack capacity. The PLSP goes one step further then and allows production of a second item to avoid too much idle time on the resource. Multi-level versions do also exist – as presented for example in Kimms (1996a) or Jordan and Koppelmann (1998).

The MLCLSP finally is a model that combines the assumptions of *dynamic demand*, to be met for *multiple items* on *multiple levels* of the product structure, to be produced on *multiple resources* with *limited capacities*. Expanding the CLSP, it is a *large bucket* model as well. It can be seen as a simultaneous execution of the MPS, MRP and CRP steps described previously. The MLCLSP captures a very common manufacturing environment and is thus a very realistic model, the satisfactory solution of which could help overcome the deficiencies of commercial PPC software.

Figure 2.4 on the next page provides an overview of the above mentioned lotsizing models. It further has the purpose of demonstrating the exact problem we are interested in.

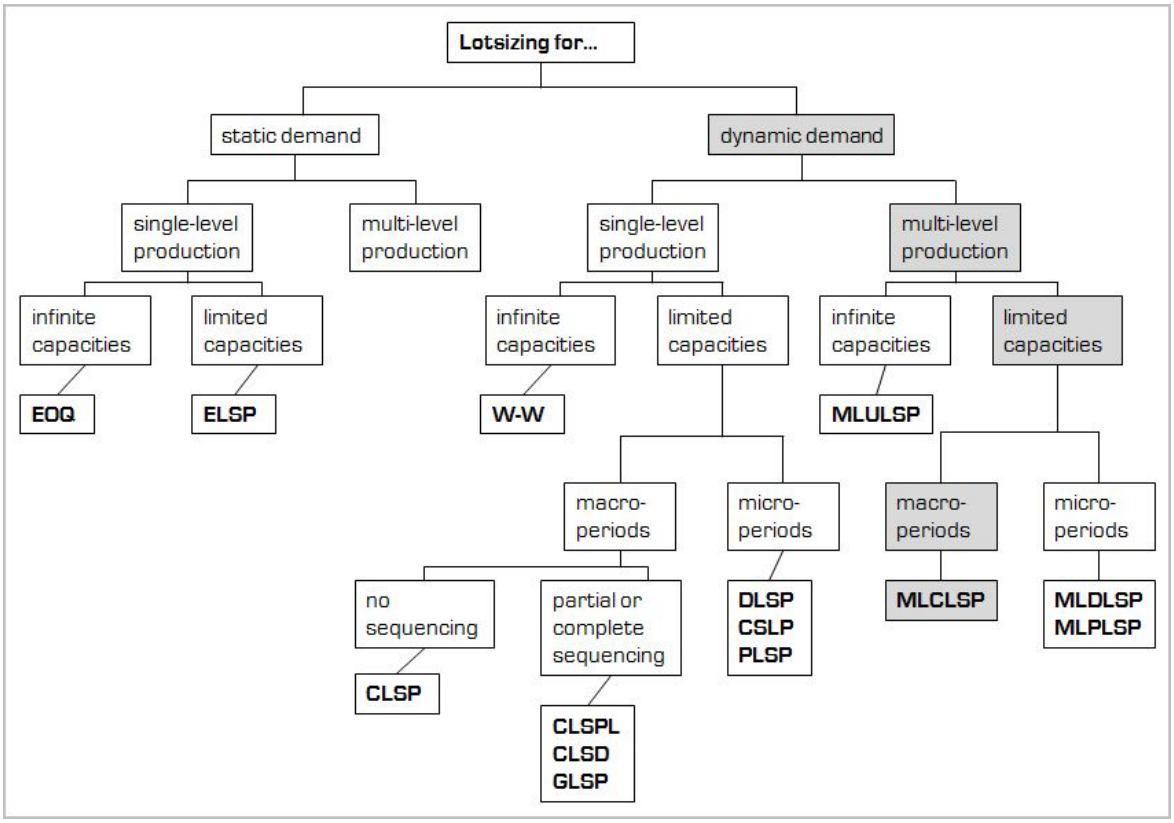

Figure 2.4 – Classification of lotsizing models.
Based on illustrations in Bahl et al. (1987) and Stadtler (2001).

2.2 Mathematical Formulations

2.2.1 The Capacitated Lotsizing Problem

Not every formulation of the CLSP published throughout the years is exactly the same. Minor, sometimes even major differences characterize the literature (cf. Lang, 2010). We will stick to Drexl and Kimms (1997), Pochet (2001), Jans (2002), and Karimi et al. (2003) for this exposition.

To be absolutely clear, let us again specifically state the assumptions the CLSP models:

- The planning horizon is finite and divided into big time buckets.
- The lots are determined for multiple items without any technological interrelationships. The model thus considers a single production level.

- The external demand for the items is dynamic and deterministic and has to be satisfied immediately and completely. The standard model allows no backordering (delaying fulfillment) or lost sales (no fulfillment).

- There is a single resource with limited capacity that is shared by all items. Overtime decisions are not considered in the standard version.

- For every item that gets produced in a given period, one setup takes place. The current setup state of the resource has no influence on setting it up for the next item – the setups are sequence-independent, postponing the scheduling decision. The setup state of the resource at the end of a period does not extend to the beginning of the subsequent period. This means that there is no possibility of a setup carryover, and not even a partial sequence is established.

- Setup times are not included as such. Instead of modeling them explicitly, they are hidden in the setup costs, represented as opportunity costs accounting for the time lost to actual production.

- Variable production costs, holding costs and setup costs can either be modeled to vary with time, or they can be modeled static throughout the planning horizon. Since the total amount to be produced of each item is predetermined by the sum of its external demand, *time-invariant* production costs and the respective term in the objective function can be omitted from the model. The objective of the CLSP is to minimize all costs incurred throughout the planning horizon.

The formulation as a mixed-integer program requires the following notation:

- *Indices*

  i item index; $i = 1,...,N$

  N number of items

  t period index; $t = 1,...,T$

  T number of periods

- *Parameters*

  c_i capacity in time units needed to produce one unit of item i

  C_t available capacity in time units in period t

  D_{it} external demand for item i in period t

hc_{it} holding cost for one unit of item i in period t

I_{i0} number of units of item i in inventory at the beginning of period 1

M_{it} upper bound on the production quantity of item i in period t

pc_{it} variable production cost for one unit of item i in period t

sc_{it} setup cost for item i in period t

- *Variables*

I_{it} number of units of item i in inventory at the end of period t

x_{it} number of units of item i produced in period t

$y_{it} = \begin{cases} 1 & \text{if a setup for item } i \text{ takes place in period } t \\ 0 & \text{otherwise} \end{cases}$

Model **CLSP**:

Minimize $$\sum_{i=1}^{N}\sum_{t=1}^{T}(sc_{it}y_{it} + pc_{it}x_{it} + hc_{it}I_{it}) \qquad (1)$$

subject to

$$I_{it-1} + x_{it} = D_{it} + I_{it} \qquad \forall\, i,\, t \qquad (2)$$

$$\sum_{i=1}^{N} c_i x_{it} \le C_t \qquad \forall\, t \qquad (3)$$

$$x_{it} \le M_{it} y_{it} \qquad \forall\, i,\, t \qquad (4)$$

$$I_{it}, x_{it} \ge 0 \qquad \forall\, i,\, t \qquad (5)$$

$$y_{it} \in \{0,1\} \qquad \forall\, i,\, t \qquad (6)$$

The *objective function* (1) minimizes the sum of all costs incurred by setting up the resource, producing the items, and holding them in inventory. Constraints (2) state that the demand for an item can either be satisfied by production in the respective period or by falling back on previously built up inventory of that item – I_{it-1} is the inventory of item i at the beginning of period t. Any amount exceeding the demand is in turn stored for future use – I_{it} is defined as the inventory at the end of a period. Constraints (2) are called *inventory* or *flow balance equations*. The inequalities (3) are the *capacity constraints*; they ensure that the level of production does not exceed the available capacity per period. The *setup constraints* (4) enforce a setup for item i in period t if it gets produced, i.e. y_{it} must take on a value of 1 if x_{it} is non-zero. In other words, production of an item is only possible if the resource has been set up for that item. If no setup takes place ($y_{it} = 0$), no production

can take place (x_{it} = 0). The upper bound on the production, M_{it}, is either given by the remaining unfulfilled demand, or by the available capacity. Mathematically speaking, $M_{it} = \min\left\{ C_t / c_i, \sum_{k=t}^{T} D_{ik} \right\}$. Non-negativity (5) and binary (6) requirements for the decision variables complete the MIP.

The above formulation involves *NT* binary variables, *2NT* continuous variables, and *(5N+1)T* constraints. The CLSP belongs to the class of *NP*-hard problems as shown by Florian et al. (1980) for the single item case, and Bitran and Yanasse (1982) for multiple items.

2.2.2 Extensions

Including *setup times* only indirectly via the setup costs entails two drawbacks regarding optimality and feasibility. Since setups reduce the available capacity – the CLSP is on top of that a model with several setups per period[4] – the calculated lot sizes might in reality not fit in the time buckets, rendering the respective solutions infeasible. Secondly, the opportunity cost part of the setup costs cannot be measured exactly as it depends heavily on the capacity utilization and remains thus an educated guess, rendering the solutions possibly suboptimal (cf. Pochet, 2001). An explicit consideration affects only the capacity constraints. Substituting (3) in the CLSP model above with

$$\sum_{i=1}^{N} (c_i x_{it} + st_i y_{it}) \le C_t \qquad \forall\, t \qquad (7)$$

yields the corresponding formulation. st_i represents the setup time for item *i* and is incurred whenever a setup takes place. Although the model dimensions stay the same, including setup times explicitly comes with the disadvantage of increased complexity – simply finding a *feasible* solution to the CLSP with setup times is an *NP*-complete problem, as shown by Maes et al. (1991).[5]

Considering *overtime* amounts to a relaxation of the capacity constraints. Normal capacity per period can be extended by using a second source, albeit at a higher price. Compared to

[4] Each period expands on the other hand over a relatively large time interval, so that setup times might only take a small fraction of the available capacity and the complicating explicit inclusion might be unwarranted.
[5] *NP*-complete is a classification for decision problems that lie in *NP*. The optimization versions of such problems are *NP*-hard.

the standard CLSP model, the objective function (1) and the capacity constraints (3) change into

$$\min \quad \sum_{i=1}^{N}\sum_{t=1}^{T}(sc_{it}y_{it} + pc_{it}x_{it} + hc_{it}I_{it}) + \sum_{t=1}^{T} oc_t O_t \qquad (8)$$

$$\sum_{i=1}^{N} c_i x_{it} \le C_t + O_t \qquad \forall\, t \qquad (9)$$

where O_t is an additional decision variable standing for the amount of overtime consumed in period t. The associated cost per time unit consumed in period t is oc_t. Like inventories and production quantities, overtime is defined to be non-negative:

$$O_t \ge 0 \qquad \forall\, t \qquad (10)$$

Backordering or *backlogging* offers the possibility to delay the fulfillment of external demands to a later period, incurring however a penalty cost that is meant to reflect the loss of consumer goodwill. Allowing backorders requires changes in the objective function and the inventory balance equations. With the additional variables B_{it}, the amount of back-orders in units of item i at the end of period t, and an additional cost parameter bc_{it} per unit of an item, we can formulate

$$\min \quad \sum_{i=1}^{N}\sum_{t=1}^{T}(sc_{it}y_{it} + pc_{it}x_{it} + hc_{it}I_{it} + bc_{it}B_{it}) \qquad (11)$$

$$I_{it-1} + x_{it} + B_{it} = D_{it} + B_{it-1} + I_{it} \qquad \forall\, i,\, t \qquad (12)$$

$$B_{it} \ge 0 \qquad \forall\, i,\, t \qquad (13)$$

The new inventory balance equations (12) make clear that there are now three ways – drawing on stock, producing, backordering – to either satisfy current demand, satisfy previous demand, or to build up new inventory. The required domain for the additional variables (13) together with constraints (3) – (6) of the standard formulation completes the backorder model. Allowing backordering can have a significant impact on feasibility in *"highly capacitated environments as well as in many real-life situations"* (cf. Quadt and Kuhn, 2008, p.63).

The CLSP does not provide the possibility of *setup carryover* and enforces a setup twice, even if the first item in a period and the last item in the preceding period are identical. To preserve the setup state of the resource, a partial sequence specifying the first and last items to be produced in each period has to be established, which comes at the price of additional binary decision variables for memorizing the setup state. A model with setup carryover can however affect the feasibility of solutions if setup times are positive, and is in some, if not most, production environments a rather straightforward step towards actual circumstances. We introduce new binary variables

$$\gamma_{it} = \begin{cases} 1 & \text{if the resource is set up for item } i \text{ at the end of period } t \\ 0 & \text{otherwise} \end{cases}$$

and a new parameter γ_{i0} specifying the initial setup state of the resource to formulate the following constraints, based on the formulation in Quadt and Kuhn (2008):

$$x_{it} \leq M_{it}(y_{it} + \gamma_{it-1}) \qquad \forall\, i,t \qquad (14)$$

$$\sum_{i=1}^{N} \gamma_{it} = 1 \qquad \forall\, t \qquad (15)$$

$$\gamma_{it} \leq y_{it} + \gamma_{it-1} \qquad \forall\, i,\, t \qquad (16)$$

$$\gamma_{it-1} + \gamma_{it} + y_{jt} \leq 2 + y_{it} \qquad \forall\, i,\, t;\, j=1,\ldots,N;\, j \neq i \qquad (17)$$

$$\gamma_{it} \in \{0,1\} \qquad \forall\, i,\, t \qquad (18)$$

Constraints (14) replace the setup constraints (4) of the standard model to allow production of an item even if no setup for it takes place in the respective period, which is not necessary if the resource is already set up for it at the beginning of that period – γ_{it-1} is in this case equal to 1, letting the left side of the equation take on a positive value as well. Conditions (15) ensure that the resource is set up for exactly one item at the end of each period. (16) states that the resource must have been set up either during the period ($y_{it} = 1$) or at the beginning of the period ($\gamma_{it-1} = 1$) if it is set up for item i at the end of that period ($\gamma_{it} = 1$). Equations (17) handle the case where the resource is set up for an item at the beginning and at the end of a period, but a different item j was produced in-between. This requires a re-setting of the resource back to item i (y_{it} on the right side must be equal to 1). Adding the variable domain for γ_{it} (18) and (1) – (3), (5) and (6) of the standard formulation yields the complete model for the CLSP with setup carryover.

Establishing this partial sequence involves additional NT binary variables and additional $(N^2+N+1)T$ constraints. Although the CLSP can also be modeled to include sequence-dependent setup costs and times and thus establish complete sequences, the models tend to become rather large and complicated, since a range of additional binary variables is required. We will take a look at an integration of scheduling and lotsizing when we come to small bucket models in section 2.2.4.

2.2.3 The Multi-Level Capacitated Lotsizing Problem

Technically, the MLCLSP could also be regarded as an extension of the CLSP, but being the topic of this thesis a separate exposition is warranted.

The MLCLSP differs from the CLSP in the following assumptions:

- Lot sizes and inventories are determined for multiple items, ranging from raw materials to end products. The items are connected through a general multi-level product structure, i.e. there are precedence relations between them that have to be considered. In addition to the external demand, internal demand arises and has to be satisfied immediately and completely as well.

- There are multiple non-identical resources with limited capacities. Each resource can be shared by several items. Each item has a given assignment to the resources required for its production.

- Especially when dealing with a multi-level structure, lead times can become relevant. According to Billington et al. (1983) p.1127, *"the average lead time for an item is composed of setup and variable production time, time to move the item to the next stage and waiting time."* The latter can be caused by congestion in front of resources, which needs to be considered in capacity-ignoring MRP systems, but is not an issue in capacitated optimization models. Waiting times can however also be caused by technical restrictions, such as needing to wait for items to cool down or for paint to dry before further processing. In the multi-level environment with its internal demand, deterministic minimum lead times for components, often set to one period, are included in the model (cf. Pochet and Wolsey, 2006; Lang, 2010; Buschkühl et al., 2010).

- Other than that, the CLSP assumptions stay valid. The MLCLSP is a big bucket finite time horizon model with deterministic and dynamic demand and limited capacities. All costs can either vary with time or be constant; extensions with overtime or backorders for end products are possible. Sequences of the lots are not established.

As with the single-level model, we can thus observe that formulations throughout the literature differ slightly. The formulation presented below is based on Günther and Tempelmeier (2007) and Stadtler (1996) and includes setup times, overtime and positive lead times. All costs are assumed to be constant throughout the planning horizon – they are only dependent on items and resources. We use the following adapted and additional notation:

- *Indices and sets*
  - i, j item indices; $i, j = 1,\ldots,N$
  - m resource index; $m = 1,\ldots,M$
  - M number of resources
  - S_i set of immediate successors of item i
- *Parameters*
  - a_{ij} number of units of item i needed to produce one unit of item $j \in S_i$
  - c_{im} capacity in time units needed to produce one unit of item i on resource m
  - C_{mt} available capacity in time units of resource m in period t
  - h_i holding cost for one unit of item i
  - l_i minimum lead time in periods for item i
  - oc_m overtime cost for one time unit of resource m
  - sc_i setup cost for item i
  - st_{im} setup time in time units for item i on resource m
- *Variables*
  - O_{mt} overtime in time units on resource m in period t

Model **MLCLSP**:

Minimize $$\sum_{i=1}^{N}\sum_{t=1}^{T}(sc_i y_{it} + h_i I_{it}) + \sum_{m=1}^{M}\sum_{t=1}^{T} oc_m O_{mt} \qquad (19)$$

subject to

$$I_{it-1} + x_{it-l_i} = D_{it} + \sum_{j \in S_i} a_{ij} x_{jt} + I_{it} \qquad \forall\, i,\, t \qquad (20)$$

$$\sum_{i=1}^{N}(c_{im} x_{it} + st_{im} y_{it}) \le C_{mt} + O_{mt} \qquad \forall\, m,\, t \qquad (21)$$

$$x_{it} \le M_{it} y_{it} \qquad \forall\, i,\, t \qquad (22)$$

$$I_{it}, x_{it} \ge 0 \qquad \forall\, i,\, t \qquad (23)$$

$$O_{mt} \ge 0 \qquad \forall\, m,\, t \qquad (24)$$

$$y_{it} \in \{0,1\} \qquad \forall\, i,\, t \qquad (25)$$

The objective function (19) minimizes the sum of all costs incurred during the whole planning horizon. Constraints (20) are the inventory balance equations reflecting the multi-level nature of the problem: in addition to the independent external demand, dependent demand has now to be satisfied as well. The internal demand is calculated with the help of the corresponding technology coefficients a_{ij} that relate item i to its successors in the bill of material. Note that for $l_i > 0$, item i becomes available only after its lead time has passed, and therefore the respective production quantities at hand in period t were already produced in period t-l_i. The capacity constraints (21) and the setup constraints (22), as well as the variable domains (23) – (25) are not affected by the presence of the multi-level product structure and should not require further explanation at this point.

The above formulation contains NT binary and $(2N+M)T$ continuous variables; the solution space is defined by $(5N+2M)T$ constraints. As the MLCLSP can be reduced to the single-level model (by setting the gozinto factors a_{ij} to zero), it is at least as difficult to solve as the CLSP with setup times. The MLCLSP is therefore an *NP*-hard problem.

2.2.4 Small bucket models

The fundamental difference to big bucket models is the assumption of much smaller periods corresponding to shifts or days, and the resulting limit on the number of items that can be produced in each period. Furthermore, setup states can be carried over between

adjacent periods. Otherwise, small bucket models have a finite time horizon, deterministic and dynamic demand for multiple independent items that has to be satisfied immediately and completely from built up inventories or through production on a single resource with limited capacity – just like the CLSP. Mixed-integer programs can be formulated with an additional set of binary setup variables and a corresponding parameter; all other indices, parameters and variables being identical to the ones used for the CLSP. Costs are assumed to be time-invariant. Foundation for the ensuing descriptions is Drexl and Kimms (1997).

Additional notation for small bucket models:

- *Parameters*

  $z_{i0} = \begin{cases} 1 & \text{if the resource is set up for item } i \text{ at the beginning of the first period} \\ 0 & \text{otherwise} \end{cases}$

- *Variables*

  $z_{it} = \begin{cases} 1 & \text{if the resource is set up for item } i \text{ in period } t \\ 0 & \text{otherwise} \end{cases}$

Model **DLSP**:

Minimize $$\sum_{i=1}^{N}\sum_{t=1}^{T}(sc_i y_{it} + hc_i I_{it}) \tag{26}$$

subject to

$$I_{it-1} + x_{it} = D_{it} + I_{it} \qquad \forall\ i,\ t \tag{27}$$

$$c_i x_{it} = C_t z_{it} \qquad \forall\ i,\ t \tag{28}$$

$$\sum_{i=1}^{N} z_{it} \le 1 \qquad \forall\ t \tag{29}$$

$$y_{it} \ge z_{it} - z_{it-1} \qquad \forall\ i,\ t \tag{30}$$

$$I_{it}, x_{it}, y_{it} \ge 0 \qquad \forall\ i,\ t \tag{31}$$

$$z_{it} \in \{0,1\} \qquad \forall\ i,\ t \tag{32}$$

The objective function (26) and inventory balance equations (27) do not differ from models previously presented. The "all or nothing" assumption becomes evident in constraints (28). If the resource is set up for a particular item ($z_{it} = 1$), as much as capacity permits is produced of that item. Constraints (29) ensure that the resource is set up for at most one item per period, implementing thus the typical small bucket condition of limiting production to one item per micro-period. Equations (30) trigger the setup costs: if the

resource is set up for item i in a given period ($z_{it} = 1$), and the setup state was not carried over from the last period ($z_{it-1} = 0$), y_{it} on the left side is forced to take on a value of one. Regarding the variable domains (31) and (32), it should be noted that the y_{it} setup variables need not be explicitly declared to be binary, as the minimization objective deters them from taking on a higher value than equations (30) require – they will therefore either be zero or one. The DLSP comprises then NT binary and $3NT$ continuous variables, restricted by $(7N+1)T$ constraints. The solution to the DLSP translates into a complete sequence of lots and thus incorporates the next step in production planning.

The CSLP relaxes the "all or nothing" requirement and allows production to take on any value within capacity limits. This can be achieved by simply replacing the equalities (28) of the DLSP with the following inequalities:

$$c_i x_{it} \leq C_t z_{it} \qquad \forall\, i,\, t \qquad (33)$$

In contrast to the DLSP, idle periods between production of the same item do not trigger setup costs twice. This is because z_{it} must be set to zero in idle periods and later re-set to one because of the "all or nothing" condition, whereas in the CSLP the resource can be set up for an item without that item being automatically produced.

The PLSP extends the CSLP by allowing production of two items per period. A slightly different interpretation of the setup variables z_{it} as the setup state of the resource at the end of a period permits the formulation of

$$c_i x_{it} \leq C_t (z_{it-1} + z_{it}) \qquad \forall\, i,\, t \qquad (34)$$

which replace constraints (33) in the CSLP and constraints (28) in the DLSP respectively. They state that production of an item is possible if the machine is set up for it either at the beginning ($z_{it-1} = 1$) or at the end ($z_{it} = 1$) of a period. Since either both setup variables can be equal to one or a second item can be produced in the respective period, separate capacity constraints are needed.

$$\sum_{i=1}^{N} c_i x_{it} \leq C_t \qquad \forall\, t \qquad (35)$$

Together with equations (26), (27), (29) – (32) and (34), they form the complete MIP for the PLSP. Regarding the complexity of these small bucket problems, they are also *NP*-hard (cf. Zhu and Wilhelm, 2006).

2.3 How to Solve Lotsizing Problems

We have seen that most lotsizing problems have a high complexity, rendering them hard to solve. Apart from (rather special) single-item uncapacitated instances, which can be solved exactly with the EOQ formula in case of static demand and the Dynamic Programming recursion of Wagner and Within in case of dynamic demand, most lotsizing problems of – practically relevant – larger size can only be solved with the help of approximate methods. This includes exact methods, such as Branch & Bound, stopped prematurely. Reformulations of the original problem, the addition of valid inequalities reducing the solution space, decomposition into smaller sub-problems and relaxation of constraints, as well as specialized problem-specific heuristics have all been employed in trying to find satisfactory solutions to complex lotsizing problems. Important work tackling the MLCLSP includes for instance Billington et al. (1983), who presented a first MIP formulation for the multi-level case, and a follow-up from 1986 considering however only a single constrained resource, where they introduced a method combining Branch & Bound with a Lagrangean heuristic, followed by a smoothing procedure to restore feasibility. Lagrangean Relaxation of both inventory and capacity constraints was employed by Tempelmeier and Derstroff (1996), in order to decompose the MLCLSP into several SLULSP sub-problems for the computation of lower bounds. Upper bounds are calculated through a level-by-level MRP explosion followed by a heuristic scheduling procedure to obtain feasible solutions. Katok et al. (1998) proposed a method that fixes the binary setup variables by modifying the technological gozinto factors and setup cost coefficients accordingly and repeatedly solving the resulting LP relaxations with updated coefficients. Stadtler (2003) employed a time-oriented decomposition approach, by solving the Simple Plant Location reformulation of the problem in a rolling horizon fashion.

Reviews of other solution approaches can be found in Karimi et al. (2003) for the CLSP, in Buschkühl et al. (2010) for the MLCLSP, as well as in Jans and Degraeve (2007). Due to the vast amount of articles on the topic, the literature review presented in this thesis

concentrates on the use of single-point metaheuristics in solving lotsizing problems. However, before listing the particular metaheuristic approaches, it seems appropriate to first describe them, their underlying principles, and their origin: local search.

3. Local Search

3.1 Why Use Local Search?

Local search belongs to the class of approximate methods, as opposed to exact methods such as Branch & Bound algorithms or Dynamic Programming. The fundamental difference between the two problem-solving approaches is that the latter is guaranteed to find the optimal solution for an optimization problem[6], while the former might return solutions of high quality, or even an optimum, but without any certainty. This begs the question of why one would then resort to local search algorithms in the first place.

The major drawback of exact methods lies in their inability to deliver the optimal solutions to combinatorial optimization problems of higher complexity and bigger dimensions within acceptable time. Expressing this in a more formal way with the help of complexity theory, most combinatorial optimization problems, such as the prominent Traveling Salesman Problem or precisely the MLCLSP, are *NP*-hard.[7] This means that they can be solved within polynomial time bounded by the size of the instance on a *nondeterministic* Turing machine. As such ideal machines do not exist to date, we are left with deterministically operating machines, on which the computational time to solve said problems to optimality rises exponentially with respect to the size of the instance. Small instances or special subclasses of a given problem still can be solved efficiently with exact algorithms, but when it comes to larger problem dimensions, computational times become prohibitive and render such an approach futile (cf. Stützle and Hoos, 2004).

One way to deal with *NP*-hardness is to abandon the quest for optimality and concentrate on finding high-quality or *near*-optimal solutions within reasonable time limits, which brings us back to the notion of approximate algorithms in general and local search in particular. The main justification for the use of local search methods lies in the need for quick answers to oftentimes giant-sized real-world problems, and the difficulties of exact algorithms to provide them.

There are several further arguments in favor of local search methods and by extension metaheuristics. The alleged lack of elegance and sophistication, which are typical of exact mathematical approaches, can in turn be regarded as a refreshingly different simplicity,

[6] Or to determine with certainty that there does not exist a solution to a given problem instance.
[7] *NP* stands for nondeterministic polynomial.

which allows ideas and efforts of practitioners without a PhD in mathematics. To quote Stützle and Hoos (2004) p.2 on the pros of stochastic local search (SLS) methods[8],

"SLS is closely related to a very natural approach in human problem solving. Many SLS methods are surprisingly simple, and the respective algorithms are rather easy to understand, communicate and implement. Yet, these algorithms can often solve computationally hard problems very effectively and robustly."

To summarize so far, one can resort to local search methods when faced with an intractable problem, where exact algorithms require too much effort: this applies to both the computational effort when running, as well as the intellectual effort when designing a procedure. Situations where one is not familiar enough with either the specific problem at hand or the detailed workings of an exact approach can be overcome with the employment of a natural heuristic approach such as local search.

Another interesting aspect in favor of local search methods is brought up when questioning the necessity of optimality. How useful is an optimal solution to a problem that does not really exist? *"Every solution we create is, to be precise, a solution only to the model that we postulate as being a useful representation of some real-world setting that we want to capture"*, Michalewicz and Fogel (2004) remind us on page 31. With all the necessary simplifications and omissions of mathematical models, and a possibly high degree of uncertainty and inaccuracy regarding input data that can be subject to frequent changes, especially in short-term production planning for instance, could it suffice to get a good answer close to the optimum? – Several authors besides Michalewicz and Fogel (e.g. Talbi, 2009; Pérez Brito et al., 2005; Fink and Rothlauf, 2006) state that instead of trying to solve exactly a model that only approximates the real world, it might be worth a try to choose a more exact and realistic representation and solve it approximately.

After these introductory remarks on the usefulness of local search, a detailed description of what local search exactly is and how it works is in order. We will pick up on advantages – and disadvantages – of local search methods in the metaheuristics subchapter.

[8] Most algorithms implemented in this thesis comprise randomized choices and constitute stochastic local search methods.

3.2 How Local Search Works

The main idea behind any local search method is to try and improve a solution to a given problem instance by slightly modifying said solution repeatedly. The procedure starts from a given initial solution and moves through the search space by replacing the current solution at each iteration with a solution from its neighborhood, which is the set of all solutions that can be obtained by a pre-defined more or less small transformation of some solution attributes. This concept of vicinity between visited solutions and the resulting *"locality of moves"* (cf. DiGaspero, 2003, p.14) gave rise to the term local search.

A few formal and precise definitions of local search related terms are now in order. From the plethora of available descriptions mainly those presented in Vaessens et al. (1998), Stützle (1998), Osman and Kelly (1996), Talbi (2009), DiGaspero (2003), and Varrentrapp (2005) form the basis for the following exposition.

Local search aims at finding solutions to (numerical) problems. The MLCLSP for example is an optimization problem Π, which comprises a whole set of concrete problem instances $\pi^{(n)}$, where $n \in \mathbb{N}^+$ denotes the size. The size of an instance is determined by the number of decision variables, also called solution components. A solution x to the instance is an assignment of a concrete value to each variable from its variable domain. Constraints among the variables further restrict the possible assignment of values to the solution components. A feasible solution is a solution that satisfies all problem constraints.

Each instance can be specified by a pair (S, f). S represents the solution space, defined as the set of all solutions. If S is a finite set of alternative solutions, Π is called a combinatorial optimization problem. f stands for a cost function that relates every element x of S to a real number. In other words, every solution has an associated objective function value $f(x)$ that is measurable and therefore comparable. A minimization problem can then be formulated as

$$\min \{ f(x) \mid x \in X, X \subseteq S \}$$

where X designates the set of all feasible solutions. The goal is to find an optimal solution $x^* \in X$ with an objective function value for which the property

$$f(x^*) \leq f(x) \quad \forall\, x \in X$$

holds. x^* then constitutes a global optimum (minimum in our case) of the instance π.

Local search operates in a search space, which we will label S_π. Its elements are called candidate solutions, search states or search positions. The search space does not have to be identical to the solution space of an instance. This depends on the chosen representation of solutions, which can either take a direct or indirect form of either partial or complete solutions. To give a concrete example, the MLCLSP contains both continuous decision variables for the lot sizes and inventories, and binary variables for the setups. In a direct representation, local search could either be conducted on complete solutions with both types of variables, or the search could be restricted to the binary setup matrices only, determining the production quantities separately. Kimms (1999) provides an example for an indirect representation: the population for his Genetic Algorithm consists of two-dimensional matrices containing rules for selecting setup states of machines.[9] Regardless of the chosen representation, every element of the search space corresponds to an element of the solution space, and thus maintains its comparability. In order to have a valid representation of the problem, the search space must furthermore contain at least one global optimum.

This so-called encoding of solutions is a fundamental issue when designing a local search procedure, as it shapes the viable move mechanisms that in turn establish the neighborhood(s). As stated above, local search is based on the idea of finding improvements by looking at solutions that are only slightly different from the current search state. This slight difference is brought about by elementary transformations called moves. To continue with our concrete example, when working only with the binary setup matrix of the MLCLSP, changing a single entry from one to zero constitutes a possible move. Chapter 5 of this thesis will show further options for manipulating the setup matrix. If the continuous variables are considered, one could define a move as shifting a partial or complete lot of an item to an earlier period. Once the move mechanism is decided upon, we have implicitly defined the neighbors of search state s. The neighborhood of s is then the set of all solutions that can be reached in one iteration of the local search algorithm by applying the selected move to it. Different move operators generate different neighborhoods, and there are usually several possibilities for perturbing a search state in a given search space.

[9] Further examples of indirect representations for lotsizing problems can be found in Jans and Degraeve (2007).

Getting formal again with the help of Stützle (1998), a neighborhood structure or function $N : S_\pi \rightarrow 2^{S\pi}$ assigns to each $s \in S_\pi$ a subset of the search space, $N(s) \subseteq S_\pi$, called neighborhood of s. Every $s' \in N(s)$ is a neighbor of s.

Figure 3.1 illustrates an example neighborhood. The area inside the circle marks the neighborhood of solution s and contains alternative solutions that are "close" to s, for instance neighbor s'. The shaded areas indicate the feasible regions $S_\pi' \subseteq S_\pi$, which the search might not necessarily be restricted to.

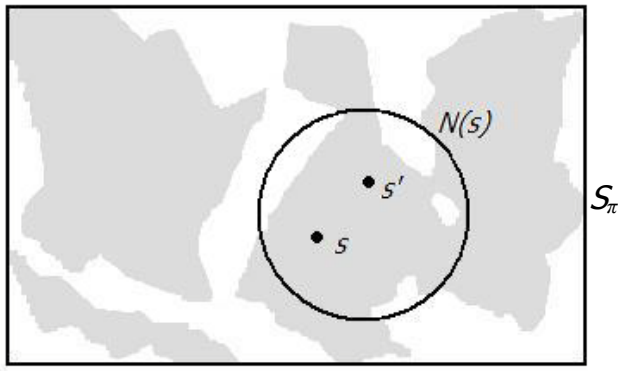

Figure 3.1 – A neighborhood in the search space.
Adapted from illustrations in Michalewicz and Fogel (2004).

If infeasible solutions are permitted during the search, one way to deal with them is through the evaluation function, which is another important component of local search. Local search is looking for improving neighbors after all, and to that end the quality of alternative search states has to be evaluated and compared. Following Stützle and Hoos (2004), we specify that the evaluation function $g(s)$ relates every $s \in S_\pi$ to a real number that preserves the ranking of solutions established by the objective function value. In other words, a global optimum should still be recognizable as such. A natural and evident option for $g(s)$ would be the use of the objective function $f(x)$ of the problem. In some cases however, calculating the objective function value for quite a few neighbors at each iteration might be computationally so expensive that some other evaluation criterion turns out more efficient. In some cases, e.g. the Propositional Satisfiability Problem (SAT), the original objective function is not suited for a clear distinction between two suboptimal neighbors[10], therefore a more telling surrogate must be formulated. It should be noted however that for many neighborhoods an incremental evaluation is viable. This means that

[10] They both have FALSE (0) as objective function value.

the objective function value can be updated rather quickly and does not have to be calculated from scratch for each neighbor. In any case, the chosen evaluation function should help guide the search towards good solutions, where "good solution" not only suggests an objective function value as low (or high) as possible, but also implies feasibility.

The final answer of any local search algorithm should be a *feasible* solution, which does not automatically exclude all infeasible solutions *during* the search process. Considering them can be a smart choice when confronted with a problem where feasible search states are hard to encounter. Considering them can furthermore be recommended in case the feasible regions of the search space are disjoint – as depicted in Figure 3.1 for example. Crossing over to another feasible area might only be achieved through temporarily visiting infeasible search positions. Finally, considering them offers a possibility to reach global optima, which are usually located on the boundary between feasibility and infeasibility, from the other side. Fred Glover devised an approach he called strategic oscillation that captures the advantage *"that moving outside of a boundary and returning from different directions uncovers opportunities for improvement that are not readily attainable when the search is more narrowly confined."* (cf. Glover, 1989, p.198)

There are thus several motives for bringing infeasibility into play, but in the end only a feasible solution is of use. An infeasible solution violates one or more problem constraints, such that the degree of infeasibility is usually measurable. Relaxing a constraint and making use of the evaluation function, one can favor feasible solutions by adding a weighted penalty term to it. Such a penalizing approach raises the critical question of choosing the right weight: setting a value too low, one might end up with an infeasible final solution, a higher value may however deter infeasibility from the start. Talbi (2009) lists three options regarding the weight coefficient: static, dynamic, and adaptive. The first strategy employs a constant weight throughout the whole search, while in the second approach the penalties are decreased or increased depending solely on the time passed. An adaptive strategy utilizes information on the search so far and adjusts the penalties accordingly; having encountered many infeasible solutions for example, the weights are increased and thus feasible solutions are now more likely to be selected. Chapter 5 of this thesis compares a static and adaptive strategy for the MLCLSP. Besides dismissing and

penalizing infeasible solutions, repairing them is a third approach one could choose and should at least be mentioned here.

One important concept that still needs an explanation is that of local optima. Once again focusing on a minimization problem, a local minimum with respect to a neighborhood structure N is defined as a candidate solution $s \in S_\pi$' for which

$$g(s) \leq g(s') \quad \forall\, s' \in N(s)$$

holds. In words, a local optimum is a solution that has no improving neighbors within a given neighborhood. Different neighborhood structures can lead to different local optima, which will come up again later as this is a property that Variable Neighborhood Search exploits. Every global optimum is also a local optimum, but this does not hold the other way round. A local minimum can be far from optimality, which Figure 3.2 helps demonstrate. Something else one can see in the illustration is that different starting points in a local search algorithm can lead to different local minima.

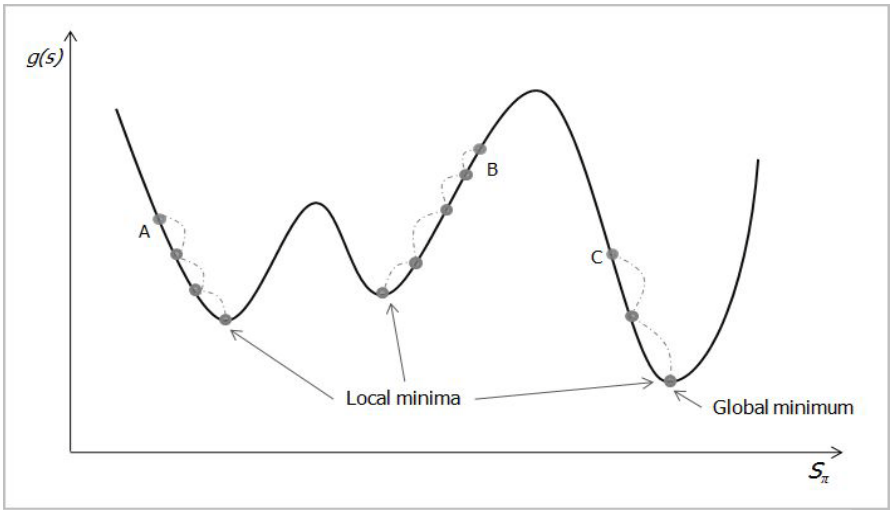

Figure 3.2 – Local minima in a search landscape.
Based on an illustration in Pérez Brito et al. (2005).

So how exactly do we get from point A to the local minimum? After having defined the most important terms in local search, it is time to take a look at the procedure itself.

The basic local search algorithm is called iterative improvement. It starts at an arbitrary initial solution s_0 (for instance point A, B or C in Figure 3.2) and stops at the first local optimum it encounters. Regarding the initial solution, it is mostly either constructed using

a greedy heuristic or generated at random and needed as input for the procedure. Having decided on a neighborhood structure, neighbors of s_0 are generated by applying the move mechanism and are then evaluated. An improving neighbor s_1 is chosen and becomes the tentative new solution, called incumbent s^*, by replacing s_0. The procedure is then repeated, such that at each iteration the search progresses from search state s_i to search state $s_{i+1} \in N(s_i)$ with a lower objective function value, drawing a trajectory in the search space. Once the whole neighborhood of the currently best solution s^* contains no improving neighbors, the search stops and a locally optimal solution is returned as the output of the procedure.

This basic procedure allows some latitude in design, most importantly regarding the questions of neighborhood generation and selection of the new incumbent. How many neighbors should we look at, and which one should we move to?

```
procedure ITERATIVE IMPROVEMENT
 INPUT
   initial solution s0 ∈ Sπ
   neighborhood structure N(s)
 INITIALIZATION
1  s* := s0
 LOOP
2  while (∃ an unchecked neighbor of s*)
3     {generate neighbor s' (s*, N(s))
4      evaluate neighbor (s')
5      if (g(s') < g(s*))
6       set s* := s' }
7 RETURN s*
```

Figure 3.3 – Iterative improvement procedure with first improvement pivoting in pseudo-code

Figure 3.3 shows an algorithmic outline of iterative improvement for a minimization problem that employs a so-called first improvement pivoting rule. One neighbor at a time is generated and evaluated, and the first neighbor that improves the objective function is picked as the new incumbent. A best improvement approach on the other hand first evaluates all neighbors and then chooses the one that realizes the greatest reduction in the objective function value to replace the current solution – a technique that is also known as

steepest descent.[11] A third option would be random improvement where s_{i+1} is chosen randomly from the set of improving neighbors. It is apparent that there exists a trade-off between the computational effort needed to examine the neighborhood and the achievable progress per search step. A first improvement pivoting rule finds the next point in the search trajectory rather quickly – when the search comes closer to a local optimum, finding an improving neighbor might however take longer – but it might miss out on better moves available, and a higher number of iterations is needed to realize significant progress. The time needed to examine all neighbors at each iteration within a best improvement procedure is certainly not negligible, sometimes even prohibitively high, but the improvement per iteration is likely to be higher and it might take fewer steps to reach a solution of good quality.

In general, we can conclude that a small neighborhood can be examined quickly, but the search might end in a solution of poor quality. Large neighborhoods might lead to better local optima, involving however a higher computational cost when scanning them for improvements. The size of a neighborhood depends on the size of a problem instance on the one hand, and on the other hand on the neighborhood structure used. As the problem dimensions are an external condition, the paramount importance of choosing an adequate neighborhood structure prompted the research of this thesis and cannot be emphasized enough.

The run-time of an algorithm will always be a crucial issue, and attempts to drive it down without a substantial loss in quality are still desirable. Apart from using a first improvement step function to speed up the search, one could reduce the computational effort by examining only a subset of the neighborhood, preferably one that contains the most promising neighbors. In case such an ideal subset cannot be determined easily, a random sample of neighboring solutions constitutes an attractive alternative. Adding a stochastic element to the search process can be fairly beneficial, in particular for multi-start iterative improvement versions with more than one run, and the enhancement of search diversification, which forms an integral part of any sophisticated local search algorithm.[12] Another way to add some randomness to an otherwise deterministic local search procedure is to establish the order in which the neighbors are examined by chance, as opposed to going a systematic route with a fixed sequence at every iteration.

[11] For maximization problems, these basic procedures are known as hill climbing methods.
[12] Diversification and its counterpart intensification will be discussed in section 3.3.

One aspect that has not been mentioned yet is the termination condition for a local search procedure. Iterative improvement naturally comes to a halt as soon as a local optimum is reached, but stopping prematurely is sometimes desired or required. Typical termination criteria include the total number of iterations, crossing a boundary on the objective function value, or simply a limit on the run-time.

Local search sounds promising so far – a fast and easy problem solving tool – but when asking the reader to take another look at Figure 3.2 on page 31, the fatal flaw of it becomes instantly evident. A local search procedure such as iterative improvement gets caught in a *local* optimum, which might be a solution that is actually quite suboptimal when compared with the *global* optimum. Depending mainly and strongly on the starting point for the search and the neighborhood structure used, solutions of high quality can of course be reached even with a simple local search algorithm, but given the multitude of locally optimal points in the search landscape of most problems, mechanisms to go beyond local optima and improve the method itself have emerged and prevailed.

3.3 How Local Search Can Be Improved

There are several ways to remedy the fundamental weakness of a simple iterative improvement procedure and continue the search beyond its natural halt, namely an arbitrary local optimum. The three prevalent strategies are to iterate iterative improvement, to allow non-improving moves, and to change the search landscape of the problem (cf. Talbi, 2009).

According to Aarts and Lenstra (1997) p.13, *"A straightforward extension of local search would be to run a simple local search algorithm a number of times using different start solutions and to keep the best solution found as the final solution."* Although restarting the procedure from several, mostly randomly generated points in the search space might be easy to execute, it lacks a sophisticated element that directs the search more efficiently and that amounts to more than pure chance (cf. Varrentrapp, 2005; Stützle and Hoos, 2004). GRASP (standing for Greedy Randomized Adaptive Search Procedure) is an example for a multi-start method with more punch.

Allowing non-improving steps sounds like a strange idea for an improvement procedure, but with the visual of a search landscape in mind it is a quite obvious way to escape from a local optimum. In order to avoid immediately returning back there through an improving move, anti-cycling mechanisms become necessary. Tabu Search and Simulated Annealing are two very prominent examples for this kind of escape strategy.

The third possibility, changing the search landscape of the problem, can for instance refer to changing the evaluation function throughout the search, as is done in Guided Local Search. Another way to change the landscape is to use different neighborhood structures during the search, an idea that leads to the – for this thesis most relevant – Variable Neighborhood Search. Before describing some of the just mentioned methods more detailed in sections 3.3.1 and 3.3.2, some explanations and general remarks on metaheuristics seem appropriate.

The history of local search (cf. Stützle, 1998, and Frank, 1997) dates back to the late 1950s and 1960s when improvement heuristics for the Traveling Salesman Problem were devised – Croes (1958) and Lin (1965) with the famous *k-opt* procedure. At the time, heuristics in general were rather looked down on by the research community and were considered as *"quick and dirty approaches"* (cf. Osman, 1995, p.95). In the 1970s, computational complexity theory initiated a rethinking, and helped by the technological advance and the growing interdisciplinarity between computer science and operations research, more attention and effort was dedicated to the development and improvement of the heuristic approach. The mid-1980s saw the appearance of successful heuristics that revived and took local search one step further – Simulated Annealing (1983) and Tabu Search (1986) – and with them appeared a new label for these higher-level, general-purpose search methods: *metaheuristics*. From the 1990s on, more and more innovative metaheuristic strategies were invented, tested, and finally became a well established and valued problem solving approach.

Although our textual transition from iterative improvement and its main flaw to metaheuristics might suggest so, simply being an extension of local search is not what defines a metaheuristic. Actually, there is no formal, single definition. So before we ponder the term metaheuristic, let us be thorough and first define the term *heuristic* in the words of Foulds (1983) p.929 as *"a method which, on the basis of experience or judgement, seems likely to yield a good solution to a problem but which cannot be guaranteed to produce an*

optimum." As was mentioned earlier in the introduction to this chapter, heuristics or approximate algorithms constitute a second broad class of problem-solving methods beside the class of exact algorithms, and the employment of the former is justified when the latter fail. The family of heuristics comprises several subclasses, with constructive heuristics and local search (also called local improvement or neighborhood search) heuristics usually being distinguished first and foremost (cf. Voss, 2000; Blum and Roli, 2003). A more detailed classification is presented in Silver (2004), who lists seven types of heuristics; Foulds (1983) names four. These so-called "rules of thumb" offer moderate computational effort and an easy understanding, and with this comes a certain flexibility and practicality. Still, the effort needed to create a heuristic tailored to a specific problem cannot be overlooked, and as Johnson (2008) p.164 writes, *"One of the aims of computer science is to reduce the amount of individual effort which needs to be put into the solution of particular problems. Whilst computational efficiency is usually taken as synonymous with the amount of computational effort required to solve a problem on the computer, it is often the complexity of setting up the computer to solve the problem in the first place which provides the greatest challenge."* This is exactly what metaheuristics aim to alleviate by offering widely applicable, reusable algorithmic frameworks for a heuristic problem-solving approach.

This brings us closer to the definition of a metaheuristic. In the words of the man who coined the term, *"A meta-heuristic refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality."* (cf. Glover and Laguna, 1997, p.17) The meta-prefix implies an upper level concerned with concepts of a more strategic nature that guide and improve a search process. At the lower level operates a subordinate heuristic, or just elements of a heuristic, that are shaped by the somewhat less myopic and more sophisticated rules the metaheuristic imposes. Citing the expert again, *"The heuristics guided by such a meta-strategy may be high level procedures or may embody nothing more than a description of available moves for transforming one solution into another, together with an associated evaluation rule."* (Ibid.) Using a local search procedure at the core is a common denominator for a range of metaheuristics and explains why the terms 'local search algorithms' and 'metaheuristics' are sometimes regarded as equivalent (cf. Vaessens et al., 1998; Stützle, 1998). We can in any case underline that a metaheuristic algorithm itself is

not problem-specific, although it *"may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy"* (cf. Blum and Roli, 2003, pp.270-271, on the *"fundamental properties which characterize metaheuristics"*).

Let us take a look at another definition of a metaheuristic as *"an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search spaces using learning strategies to structure information in order to find efficiently near-optimal solutions"*, courtesy of Osman and Kelly (1996) p.3. This definition names two important concepts typical for any elaborate metaheuristic: exploration and exploitation, also called diversification and intensification respectively. The latter refers to concentrating the search on promising areas of the search space, capitalizing on information about good solutions gathered so far and on the empirical insight that for many problems good local optima tend to be relatively close to another and form clusters (cf. Varrentrapp, 2005; Battiti et al., 2007). Encountering an even better local optimum or a global one in the proximity of a high-quality solution is therefore not unlikely. Intensification can also refer to identifying common components of these so-called elite solutions and induce an incorporation of them into currently active solutions (cf. Glover and Laguna, 1997; Gendreau and Potvin, 2005b). One could for instance fix or 'freeze' some of the decision variables to the common value found in most or all good solutions visited, leaving only the rest admissible for moves. Nonetheless, the tricky aspect of intensification is that assessing the absolute quality of local optima during the search is difficult, so that a supposedly good solution can turn out mediocre in the end. For this reason, the overall search should not only be focused on a small region of the search space and avoid the risk of wasting too much time in possibly the wrong place and of a stagnation of the search progress. The counterpart to intensification is then to venture into regions of the search space previously unexplored and diversify the search in the hope of finding other attractive areas that might contain a global optimum. Diversification involves perturbing a solution more or less profoundly, at least more than in a normal search step, and bringing about a change that cannot be easily undone. On the other hand, landing at completely unrelated random points with every 'kick' amounts to a simple random restart procedure. Diversification mechanisms in most metaheuristics are however more elaborate than pure randomization and accidentally discovering optima. Determining the adequate strength of the perturbation is thus the tricky aspect of diversification. A common characteristic of well-developed metaheuristics is then not only to devise

functional and efficient intensification and diversification strategies, but also to try and achieve a clever balance of the two contrasting concepts.

We noted above that metaheuristics go beyond plain randomization in their strategic guidance, or as Stützle (1998) p.23 puts it, "*the main difference to pure random search is that in these algorithms randomness is not used blindly but in an intelligent, biased form.*" Stochastic elements of this kind found their way into most modern metaheuristics and can thus be seen as another frequent common feature, in contrast to the purely deterministic iterative improvement default procedure for example (cf. Blum and Roli, 2003; Gendreau and Potvin, 2005a; and Stützle and Hoos, 2004, with their book on stochastic local search).

Apart from the fundamental characteristics and some ingredients common to all metaheuristics, there are naturally differences also, which allows for a classification of the various methods. As with heuristics in general, one can distinguish between constructive and improvement metaheuristics (cf. Gendreau and Potvin, 2005a). Osman (2004) specifies the three categories *guided construction, guided neighborhood-search, and guided population metaheuristics*, which includes the more common and relevant distinction between single-solution (also called single-point or trajectory) methods and population-based methods. While the reader certainly has an idea about the former by now, the latter call for a sentence or two. Instead of moving from one solution to the next, they maintain a pool of solutions. This so-called population is iteratively manipulated, for instance by imitating biological processes such as natural selection or the behavior of ants. Examples for population-based metaheuristics are thus the group of evolutionary algorithms, with their most prominent member being Genetic Algorithms (cf. Holland, 1975), and Ant Colony Optimization (cf. Dorigo, 1992). The previously mentioned methods Tabu Search, Simulated Annealing, and Variable Neighborhood Search in contrast operate on a single solution per iteration.

Metaheuristics can further be classified according to whether they use memories throughout the search or not. This refers to the decision to not only let the current search state direct the search, but to additionally benefit from information and experience gathered during the search and to learn from search history. Tabu Search is well-known for its explicit use of short-term and long-term memories; pheromone trails in Ant Colony Optimization and the population of a Genetic Algorithm serve the same purpose more

indirectly. At the other end, Simulated Annealing, Variable Neighborhood Search, and basic local search are memory-less methods. For further expositions on the classification of metaheuristics the reader is referred to Glover and Laguna (1997) and Stützle (1998).

We should not completely omit disadvantages of metaheuristics. Apart from the inherent shortcoming of not being able to guarantee an optimal solution, a proof of optimality, or even a measure of how far from optimality the final answer is, the theoretical and mathematical assessment of the performance of metaheuristics is difficult, leaving the important questions of why and for what kind of problems a given metaheuristic works well mostly unanswered. Missing universal guidelines for testing and comparing methods complicate the predicament (cf. Voss, 2000; Pirlot, 1996; Osman, 1995). Another problem with some metaheuristics is the gap between the intentional simplicity of use of these methods and the actual effort needed for a successful implementation in practice. Designed to be flexible, metaheuristics are not rigid schemes and leave the user with choices to be made and parameters to be tuned. Additionally, profound problem-specific knowledge might still be necessary when fleshing out a metaheuristic framework (cf. Johnson, 2008; Hertz and Widmer, 2003; Gendreau and Potvin, 2005a). 'Easy to use' might therefore be a commendable intention only for some metaheuristics.

Let us conclude this section on a high note and stress the undeniable success of metaheuristics as problem-solving tools, evident in countless publications on the topic – growing numbers of theoretical and empirical analyses as well as reports of successful applications to a wide range of problems. Quoting Osman and Kelly (1996) p.15, *"Meta-heuristics have proven their power in obtaining high quality solutions to many real world complex problems, and we expect this number to continue to grow in the future. We encourage and advocate their usages due to simplicity, robustness and ease of modification. However, the design of a good meta-heuristic remains an art."*

Let us take a look at the artwork of Pierre Hansen and Nenad Mladenović then.

3.3.1 Variable Neighborhood Search

Earlier on we defined a local minimum as a solution with no improving neighbors in the whole neighborhood. A different neighborhood structure might however contain an improving neighbor for this very solution, from which it follows that a local optimum is optimal only for the neighborhood the local search procedure employs. We defined a

global minimum as a solution with no improving neighbors in the whole search space, from which it follows that a global optimum is optimal for any neighborhood the local search procedure might employ. We also mentioned previously that empirical investigation revealed that for many problems local minima tend to form clusters.

Looking for a way to escape from local minima, Mladenović and Hansen (1997) exploited these three facts and devised a single-point multi-level metaheuristic – according to the classification in Vaessens et al. (1998) – called Variable Neighborhood Search (VNS), which comes in several shapes. What all variants have in common is that they systematically change the neighborhood structure throughout the search and thus need more than one neighborhood as input. We will start with a description of the variant implemented for this thesis. The following section is also based on Hansen and Mladenović (2001; 2003a; 2003b) and Hansen et al. (2003; 2008).

Basic VNS is built on top of an arbitrary local search procedure, such as regular iterative improvement, which corresponds to the intensification phase of the search. For diversification purposes and in order to escape from a local optimum encountered by the local search, a so-called shaking step is used. In this step, the incumbent is perturbed through a random move within the current neighborhood in order to obtain a new starting point for the local search step. The general dilemma of any diversification strategy applies to VNS procedures as well: a perturbation too small might not lead to the desired different basin of attraction, while perturbations too strong might soon resemble the less desired random multi-start procedure. Figure 3.4 on the next page shows the details of the basic VNS procedure and how it deals with the issue.

Beside at least two neighborhood structures, the basic VNS needs an initial solution to start the search process, and a stopping condition to end the search process as input. Usually a limit on the total time passed, on the total number of iterations, or on the number of iterations without any improvement is used as termination criterion. The main body of the basic VNS consists of three steps. Starting with the first neighborhood, a random neighbor *s'* of the incumbent *s** is generated in the shaking step. In the next step, a local search procedure is executed from *s'*, delivering a locally optimal solution *s''*. In the third step, the decision whether to move from the incumbent to the new local optimum is made, depending on the quality of *s''* and on the acceptance criterion specified by the user. In

case of accepting only improving solutions, no move is made if the local search step led back to the same local optimum, i.e. $s'' = s^*$, or if the new local optimum is not better than the current, i.e. $g(s'') \geq g(s^*)$[13]. The procedure continues with a shaking step in the next neighborhood of the unchanged incumbent. Only if s'' is better than s^*, the move is executed and the procedure continues with a shaking step in the first neighborhood of the new incumbent. In case the whole inner loop from k_1 to k_{max} brings no improvement on the incumbent, the process is iterated, i.e. a new outer loop is started beginning with the first neighborhood, until the stopping condition at some point finally evaluates to true.

```
procedure BASIC VNS
 INPUT
   initial solution s0 ∈ Sπ
   neighborhood structures Nk(s) where k = 1,…,kmax
   stopping condition
 INITIALIZATION
1  s* := s0
 OUTER LOOP
2  while (stopping condition = FALSE)
3     {k := 1
 INNER LOOP
4     while (k ≤ kmax)
5        {a. SHAKING: generate random s' ∈ Nk(s*)
6         b. LOCAL SEARCH: apply local search procedure to s'
7                          and obtain local minimum s''
8         c. MOVE OR NOT: if (g(s'') < g(s*))
9                           s* := s'' and k := 1
10                         else
11                          k := k+1 }}
12 RETURN s*
```

Figure 3.4 – The basic VNS algorithm in pseudo-code.
k_{max} is a user-specified parameter (number of neighborhoods).
Adapted from Hansen and Mladenović (2003b).

[13] for a minimization problem

Figure 3.5 below shows part of a basic VNS procedure graphically. Following the search process from left to right, we just found a new best solution in local minimum *s''* through a local search descent from a randomly generated neighbor *s'*. The local search from the neighbors of N_1 and N_2 generated in the next shaking steps leads back to the same local minimum; the neighbor from N_3 and the following local search however make it possible to escape from the valley and reach a different local minimum. As the latter is better than the incumbent, we move there and start the process anew.

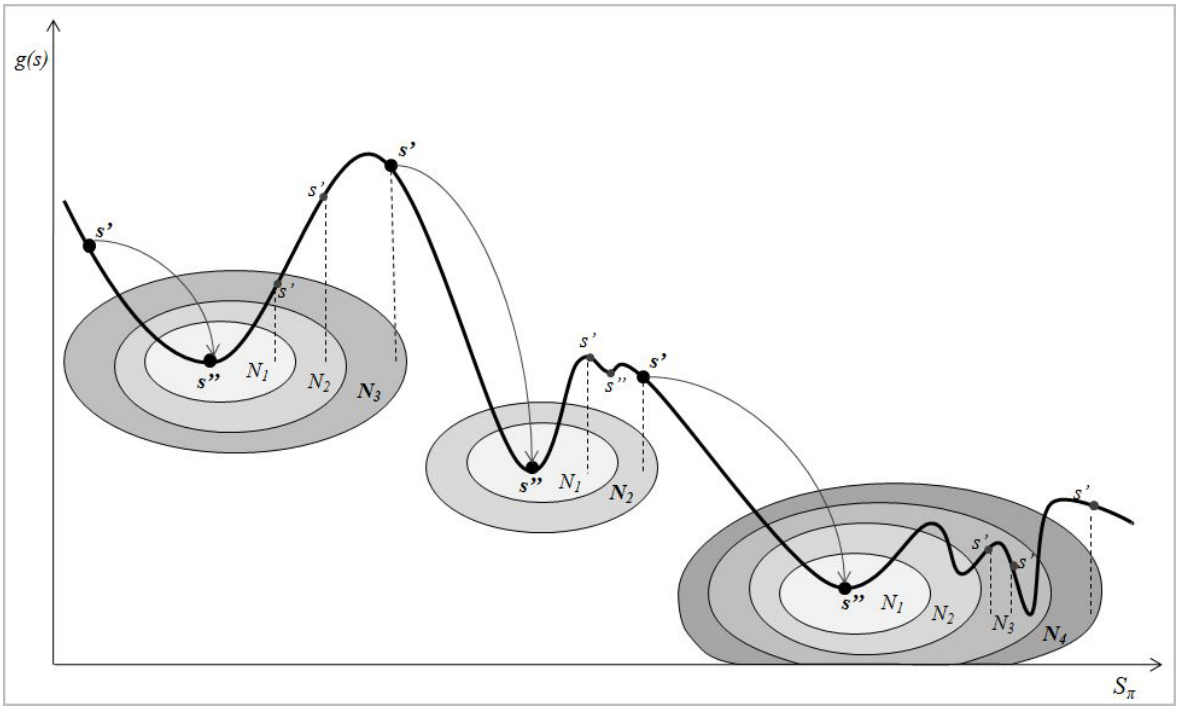

Figure 3.5 – The basic VNS algorithm in an illustration.
Based on an illustration in Hansen et al. (2008).

Note that the random neighbor from N_1 does lead to a new local minimum this time, but to no improvement in the objective function value. VNS continues therefore with a shaking step in the next neighborhood. The drawing around the next incumbent illustrates why repeating the outer loop makes sense: suppose that a first random neighbor from N_3 progresses to a new but worse local minimum and the search continues with the last neighborhood N_4. If we cannot realize any improvement from there on, VNS returns to N_1 and generates a new random neighbor of the incumbent. Maybe this time the shaking step in N_3 will produce a neighbor that makes it possible to reach the better local (maybe even global) optimum.

It is important to note that the local search phase itself is completely independent of the surrounding VNS mechanisms. There are no rules regarding the design of it; the neighborhood employed does not have to be one of the neighborhoods used in the shaking step.[14] It is also possible to use more than one neighborhood for the local search step. Regarding the shaking step, we can observe that each time a *random* neighbor from the respective neighborhood is generated, which aims to prevent cycling between already visited solutions. Note that the random neighbor also does not have to fulfill any requirements regarding solution quality, i.e. its objective function value is irrelevant.

The basic VNS combines a deterministic local search procedure with a stochastic mechanism to overcome local optima. It is a first-improvement method in the sense that it moves to the first better local optimum it encounters. One could very well tweak the original procedure, such that it first conducts all k_{max} shaking and local search steps, and only afterwards jumps to the best of all local optima reached. The basic VNS is by default a pure descent method, which can however be easily changed into a descent-ascent method with an adjustment in the acceptance decision. Also conceivable are modifications in the shaking step – for example generating several neighbors from a respective neighborhood and choosing the best as starting point for the local search. Changing the default procedure might add complexity and parameters – the almost complete lack of which being in fact however a main advantage of VNS.

If the local search step of the basic VNS is omitted, one obtains a variant called *Reduced VNS (RVNS)*. The reader is asked to recall or take another look at the pseudo-code for the Basic VNS: step b of the inner loop (code lines 6-7) is simply left out. Step c then compares the objective function values of the generated random neighbor *s'* (instead of the local optimum *s''*) and the incumbent *s** and makes a move or not. As the local search step can be very time-consuming, this shortened VNS variant is typically very fast and designed for problem instances of very large dimensions.

We mentioned above that the local search phase in the basic VNS might employ several neighborhoods as well. *Variable Neighborhood Descent (VND)* is such a multi-level local search procedure. It is purely deterministic in contrast to the basic VNS and RVNS. After finding a local minimum in the first neighborhood, the search continues with a best-

[14] To be more precise, it should not be one of the neighborhoods used for the shaking step. We want to escape from the local optimum reached with this neighborhood structure, and therefore need to perturb the solution more profoundly.

improvement exploration of the second neighborhood. Should the latter contain a solution better than the incumbent, one descending step is made in this direction before returning to the first neighborhood. If the local minimum of the first neighborhood is also a local minimum of the second neighborhood, i.e. no improving neighbor can be found, the next neighborhood is explored.

The stopping condition for the VND algorithm is that of no improvement: once no direction of descent exists in any neighborhood, the incumbent is a local minimum with respect to all neighborhoods – a fact that increases its chances of being a very good local or maybe even global optimum. As the rest of the procedure is identical to the basic VNS, Figure 3.6 only shows the distinguishing inner loop.

```
procedure VND
  …
 INNER LOOP
4      while (k ≤ k_max)
5          {a. EXPLORATION: generate all s' ∈ N_k(s*)
6                           set s_min' := arg min g(s')
7           b. MOVE OR NOT: if (g(s_min') < g(s*))
8                            s* := s_min' and k := 1
9                           else
10                           k := k+1 }
  …
```

Figure 3.6 – The inner loop of the Variable Neighborhood Descent algorithm in pseudo-code

Combining the basic VNS with VND in the local search step yields the *General VNS (GVNS)* variant, which reportedly is responsible for the most successful applications of the VNS family (cf. Hansen et al., 2008). *Variable Neighborhood Decomposition Search (VNDS)* and *Skewed Variable Neighborhood Search (SVNS)* are further extensions of the basic scheme. VNDS is addressed at solving very large instances more efficiently by restricting the local search step to only part of the whole search space. A subset of solution attributes is fixed, reducing the neighborhood of a given search state to solutions that modify the free variables only. In other words, the problem is decomposed into sub-problems, which are iteratively solved through the respective local search phase. SVNS employs an advanced criterion in the acceptance decision of a new found local optimum.

While improving solutions are still always accepted in the move-or-not step, SVNS also considers worsening moves on condition that they lead sufficiently far away from the incumbent. The underlying idea is to diversify the search in a more clever way than a simple random multi-start once a large region of the search space has been explored and the search needs to progress to regions farther away. To this end, SVNS needs an additional parameter and a function measuring the distance between two solutions.

The success of a VNS implementation is to a large extent dependent on the neighborhood structures used. Determining which neighborhoods are suitable for the problem at hand is thus a first crucial step that might require preliminary testing – a crucial first step that this thesis is devoted to. Another natural question that arises concerns the adequate order of the neighborhoods. Again, there is no universal rule, leaving the user with several options. Neighborhoods can be ordered according to their increasing size, i.e. the number of neighboring solutions it contains. This often coincides with a neighborhood sequence of increasing strength of the perturbation.[15] A prominent example would be k-opt neighborhoods in a sequence with increasing k, resulting in so-called nested neighborhoods and a very sensible *"intensification first, minimal diversification only if needed"* strategy (cf. Battiti et al., 2007, p.9). A third and simple option is to determine the active neighborhood at random, either by establishing a fixed sequence beforehand or by dynamically selecting neighborhoods during the search. A dynamic sequence does not have to be stochastic however; as a fourth option, the active neighborhood can be selected according to previous successes and failures in finding better solutions for example – Raidl and Hu (2006) proposed such a self-adaptive neighborhood-ordering for VND. This means of course that some form of search memory must be incorporated into the otherwise memory-less VNS.

According to the creators, Variable Neighborhood Search fulfills a list of *"desirable properties"* any metaheuristic should possess.[16] The VNS method is derived from the *simple and clear principle* of systematic neighborhood change during the search. The procedures are *coherent* and can be formulated in *precise mathematical terms*. They are *efficient* in the sense that a majority of problem instances can be solved satisfyingly, i.e. to optimality or near-optimality, which is furthermore accomplished *effectively*, i.e. in

[15] Several neighborhoods presented in chapter 5 are however examples of neighborhoods of smaller size and stronger perturbation than the default neighborhood *Switch*.
[16] The list was however not established by a third party, but by Hansen and Mladenović (2001) themselves.

acceptable computational time. Efficiency and effectiveness over a range of problem instances, as opposed to some small particular set, give VNS the required *robustness*. A very distinguishing quality of VNS can be seen in the employment of as few parameters as possible – the number of neighborhoods largely remains the only one to be specified. This minimalism leads to procedures that are easy to understand and use, or in other words, a high degree of *user-friendliness*, which constitutes an important asset in the market for metaheuristics.

All these words might capture the positive traits and advantages of the VNS family, but nothing says excellence more loud and clear than a long list of successful applications to numerous and diverse optimization problems. This long list will not be reproduced at this point due to lack of available pages; the most current one can be found in Hansen et al. (2008). We will however take a closer look at applications to production-related problems in chapter 4.

3.3.2 Brief Descriptions of Other Metaheuristics

As we will encounter various other metaheuristics as well when discussing ways to solve the MLCLSP, brief descriptions of some of the methods seem appropriate.[17] As previously mentioned, population-based metaheuristics constitute a second broad class of methods with a quite different underlying philosophy. The scope of this section is restricted to metaheuristics that extend local search and are thus similar and related to Variable Neighborhood Search.

Simulated Annealing (SA) by Kirkpatrick, Gelatt, and Vecchi (1983) is a memory-less trajectory method. The name of this metaheuristic stems from the intended simulation of the physical process of cooling liquid metal slowly into a solid state, which is known as annealing. It is a probabilistic local search algorithm that overcomes local optima by also accepting non-improving solutions. Starting from a given initial solution, each iteration consists of generating a random neighbor of the incumbent and subjecting it to an acceptance criterion. While improving moves will always be executed, solutions of inferior quality can be accepted depending on the dynamic probability function $p = e^{-\Delta f(s)/T_k}$, where $\Delta f(s)$ stands for the difference in the objective function value of the incumbent and

[17] This section concentrates on the basic or "default" versions of the algorithms. The references point to seminal papers.

its neighbor, $f(s') - f(s^*)$, and T_k represents the *temperature* parameter at time interval k. A high temperature and/or only a slight deterioration in solution quality increase p and thus the chances of acceptance, whereas a low T value or a significantly worse neighbor will – probably – leave the incumbent unchanged. As the temperature is supposed to decrease throughout the search, every SA procedure needs a *cooling schedule*, which specifies a starting value for T, the length of time interval k for which T remains unchanged, and the rate at which the temperature is lowered. Regarding the strategy behind SA, the high temperature at the beginning of the search leads to almost all moves being accepted, which enables the algorithm to explore the search space – the diversification mechanism of SA. Through the gradually lowered temperature the search process intensifies, as the chances for inferior solutions to replace the incumbent go down, until finally only improving moves are made. The procedure is usually ended once improvement stalls. What is briefly described here is a simple SA default procedure; the reader can imagine the range of possibilities for modification of the cooling schedule.

Tabu Search (TS) by Fred Glover (1986) is a deterministic trajectory method making extensive use of search memories. Similar to SA, it prevents getting trapped in local optima through allowing non-improving moves. Starting from a given initial solution, TS performs a best-improvement local search: at each iteration it chooses the best neighbor to become the new incumbent. In case no improving neighbors exist, the best neighbor is simply the least non-improving one. Allowing this kind of move can lead to cycling, i.e. returning to the old incumbent in one of the next iterations, and while Simulated Annealing counters this with randomization, Tabu Search forbids such a move – it is *tabu*, incidentally explaining the name of this metaheuristic. Rather than complete solutions, only attributes of forbidden moves – reversals of the most recent moves made – are stored in a *tabu list*, the size of which depends on the *tabu tenure*, which specifies for how many iterations a certain move is excluded from consideration. A very basic version would handle the list in a "first in, first out" mode with a static tenure: insert the attributes of the most recent move and delete those of the least recent move, keeping the tabu tenure equal for all moves and thus the size of the list constant during the search. The length of the tabu tenure is in any case crucial for the behavior of the algorithm: setting the value too low might leave too much room for possible cycling and impede escaping a basin of attraction, whereas a value too high might limit the move options so severely that the procedure frequently misses out on promising neighbors. The latter concern can however be remedied

through the use of *aspiration criteria*, which are used to revoke the tabu status of solutions. A straightforward example would be the selection of a neighbor – despite involving a move that is on the tabu list – with a better objective function value than the best solution encountered so far. Tabu Search usually operates with a single yet dynamically changing neighborhood structure. At each iteration only a (different) subset of the neighborhood is considered: the so-called *admissible neighbors*, consisting of the solutions that can be reached without a move that is tabu plus the solutions passing the aspiration level, from which the best one is chosen. Tabu Search ends as soon as a termination condition specified by the user is reached.

Tabu lists and aspiration criteria are manifestations of the *short-term memory* of TS, but the real power of this metaheuristic unfolds once *intermediate and long-term memories* complement the procedure. In addition to the *recency* dimension of the search history, recording how often a move was executed (*frequency*), memorizing common solution attributes of high-quality solutions or maintaining a pool of elite solutions (*quality*), and keeping track of solution components that prove to be crucial to the overall quality of a solution (*influence*) are examples for memory structures that equip TS with advanced intensification and diversification mechanisms. The Tabu Search framework is suited very well for enhancements and experimentation – which might however come at the price of further parameters. It is one of the most widely applied metaheuristics – with considerable success (cf. Stützle, 1998; Blum and Roli, 2003; Gendreau and Potvin, 2005b).

The *Greedy Randomized Adaptive Search Procedure (GRASP)* proposed by Feo and Resende (1989; 1995) is a primarily constructive method enhanced with a local search phase. Emphasizing the latter, one could say that GRASP is a memory-less trajectory method that escapes local optima through generating new starting solutions and iterating local improvement. In any case, GRASP is composed of a construction phase followed by a local search phase, both of which are iterated for a predetermined number of iterations. In the first phase, a solution is constructed by adding one solution component at a time in a *greedy* manner. To this end, the most beneficial alternatives at this point for insertion into the partial solution are ranked and stored on the *restricted candidate list*. The degree of restriction, that is the number of elements on this list, is a user-made decision with considerable impact on the success of the procedure: keeping a list that is too long might turn GRASP into a plain local search with random restarts, while a short list might produce

too few and moreover too similar starting points for the local search step. The latter concern also makes it imperative that the selection of the component from the list to be added next is *randomized* – GRASP stochastically chooses one of the best and not necessarily the best option every time, in order to construct a greater number of more diverse solutions than a deterministic version ever could. The ranking of the solution components is done according to a heuristic (greedy) value that changes throughout the construction: every time an element has been added to the solution the values are updated and thus reflect the previous choices. As opposed to static values, this approach makes the procedure *adaptive*. Once a complete solution has been constructed, a local *search* step – which can range from plain iterative improvement to an elaborate Tabu Search – is carried out and takes GRASP to a local optimum. The final answer of the algorithm is the best local optimum encountered during its run. The goal of the construction phase in GRASP is to generate a sufficient number of sufficiently dissimilar solutions for a thorough exploration of the search space; the purpose of the local search phase in contrast is intensification from hopefully already high-quality solutions – they are constructed in a greedy manner after all. The algorithm could even be executed without the local search phase.

Iterated Local Search (ILS) – see the chapter in the *Handbook of Metaheuristics* by Lourenço et al. (2003) or a shorter description in Blum and Roli (2003) – is a very general metaheuristic framework that can be used as a common heading for methods that perform local search in the search space of local optima. As global optima are local optima as well, condensing the search progress to a sequence of locally optimal solutions – in contrast to a sequence of simply solutions in regular local search – is a straightforward ambition. The search space S_π for an ILS algorithm is reduced to a much smaller subset S_π^* consisting only of local optima. This simple idea is somewhat hindered by the lack of a corresponding neighborhood structure for S_π^* – local optima are not known in advance; otherwise the search would be trivial. One way to identify them is of course through a local search procedure. Starting from a given initial solution or more precisely the first local optimum detected with local search, Iterated Local Search repeatedly carries out the following three steps until a stopping condition is met: *perturbation – local search – acceptance decision*. This sounds familiar; the inner loop of the basic VNS consists of the same three steps. The *perturbation* involves however usually a modification within a single neighborhood, but is in general not specified explicitly in the deliberately vague ILS framework – the nature of

the perturbation is neither described nor prescribed. Regarding the strength of it, the same considerations mentioned previously apply. The perturbation should be sufficiently strong to reach a new region of the search space, or the following local search step would be in vain, leading back to the very local optimum one wants to escape from. The incumbent must thus be altered stronger than within the local search phase. The perturbation should on the other hand not change the incumbent so severely that it amounts to a random restart. The goal is to keep some solution components of the high-quality solutions – local optima after all – and run the local search from an already good starting point, which may additionally reduce the computational effort it requires. The strength of the perturbation directly influences the level of exploration ("big jumps") and exploitation ("small jumps"). The generation of the perturbed solution is usually done at random in order to avoid cycling. All new found local optima are faced with an *acceptance decision*: objective function values of the incumbent, i.e. a previously encountered local optimum, and the new local optimum are compared and the decision whether to move or not is made. The ILS framework does not specify the acceptance criterion itself. One can choose between several options, which influence the direction of the algorithm as well: to name two extremes, always accepting the new local optimum diversifies the search, whereas accepting only improving solutions would be an intensifying strategy.

The two-level architecture characteristic of metaheuristics is quite evident in the ILS structure: the local search step – or possibly some other problem-specific heuristic – is treated as a black-box, which is embedded in the higher-level steps (perturbation and acceptance decision) of the metaheuristic. The metaheuristic itself also performs a search – with the output of the underlying heuristic – but it operates completely unaffected by problem-specific aspects.

Iterated Local Search methods are not always labeled as such. They appear in the literature as Iterated Descent (Baum, 1986), Large-step Optimization or Markov Chains (Martin et al., 1991; Lourenço, 1995), or Chained Local Optimization (Martin and Otto, 1996), which combines Simulated Annealing with local search by applying the latter to the perturbed random neighbor before the acceptance decision of the former. Although it is derived from completely different premises, the likeness of Variable Neighborhood Search to ILS (or vice versa) is undeniable. It is arguable to identify VNS as a well-defined ILS procedure

where the crucial perturbation step is laid out explicitly (cf. Hansen and Mladenović, 2003b).[18]

Which metaheuristic is the best one? Let us conclude this chapter by mentioning the *"No Free Lunch Theorems"* (cf. Wolpert and Macready, 1997), which state that no optimization method performs better than all other methods for all problems. Method A is superior to method B for some problem? – There exists another problem where B outperforms A. No metaheuristic can therefore claim the title "best metaheuristic", which might explain the ongoing *"unification or convergence phenomenon"* (according to Gendreau and Potvin, 2005a, p.198) and the growing number of hybrids that pick the best elements of existing methods and combine them to create a new method. Maybe not the best – but better would be nice.

[18] They seem however rather displeased that VNS gets *"squeezed into the ILS framework."*

4. Metaheuristics for Lotsizing Problems

Although employing metaheuristics does not seem to be the primary solution approach to lotsizing problems, over the years applications accumulated, warranting separate literature reviews – Jans and Degraeve (2007) reviewed metaheuristics in general; Aytug et al. (2003) as well as Goren et al. (2008) focused on Genetic Algorithms alone. Still, pure pristine implementations of a single metaheuristic are rare; most papers combine several techniques and incorporate metaheuristic methods or elements into their overall solution approach. The most widely applied metaheuristics in this area are Simulated Annealing, Tabu Search, and Genetic Algorithms.

4.1 Single-Level Lotsizing

The standard CLSP has been tackled by Hindi (1996), who combined a Tabu Search on the setup pattern with solving a reformulation of the problem as an uncapacitated transshipment problem. The same reformulation was used in Hindi et al. (2003), where they employed a Variable Neighborhood Search for the CLSP with setup times. The various neighborhoods resulted from switching off k setups, where k was set to range between one and seven. A smoothing heuristic executed before the VNS and Langrangean Relaxation for the computation of lower bounds completed their approach and gave satisfactory results. Gopalakrishnan et al. (2001) also devised several move mechanisms, combined them however into a single neighborhood in their Tabu Search implementation to solve the CLSP with setup carryover. Hung et al. (2003) tested three different Tabu Search versions: a regular best improvement procedure, a best improvement procedure where the neighbors are ranked on a candidate list according to a simplex tableau-based heuristic value, and finally a first improvement version with said candidate list. Özdamar and Bozyel (2000) and the follow-up Özdamar et al. (2002) use the lot sizes for generating neighbors. The former implemented a stand-alone Simulated Annealing procedure, compared to a stand-alone Genetic Algorithm, while the latter presented a hybrid metaheuristic incorporating probabilistic Tabu Search steps in a Genetic Algorithm.

Table 4.1 – Applications of single-point metaheuristics to single-level lotsizing problems *(continued on next page)*

Author(s)	Model	Specific Features	Meta-heuristic	Solution Representation	Neighborhood(s)	Notes
Gaafar et al. (2009)	SLULSP	BO, batch ordering	SA	string of T digits with values 0,1,2	16 perturbations of string	Comparison with GA and modified Silver/Meal
Hindi (1996)	CLSP		TS	setup variables	change single y_{it}	Combination with reformulation as transshipment problem
Hindi et al. (2003)	CLSP	ST	VNS	setup variables	switch off k setup variables; (k=1,..,7)	Smoothing heuristic followed by VNS. Transshipment reformulation as in Hindi (1996).
Pedroso and Kubo (2005)	CLSP	BO, multiple machines	TS	setup variables y_{imt}	deleting setup or shifting setup to different machine in same period	Construction of solutions and restart with relax-and-fix heuristic
Özdamar and Bozyel (2000)	CLSP	ST, OT, no setup costs	SA	production quantities	shifting random feasible amount of one item between two consecutive periods	SA tested with three different initial solutions
Özdamar et al. (2002)	CLSP	ST, OT, no setup costs	Hybrid GA, TS and SA	production quantities	as in Özdamar and Bozyel (2000)	GA with parallel populations, once in a while TS with SA acceptance criterion activated
Özdamar and Birbil (1998)	Capacitated Lotsizing and Loading Problem	OT, parallel facilities	Hybrid GA, TS and SA	production quantities	as in Özdamar and Bozyel (2000)	Same hybrid as in Özdamar et al. (2002)
Hung et al. (2003)	CLSP	ST, BO, multiple resources	TS	setup variables	change single y_{it}	Three versions: best improvement, best improvement with candidate list according to ranking heuristic based on simplex tableau, and first improvement with candidate list
Gopalakrishnan et al. (2001)	CLSP	setup carryover	TS	sequences of lots / production quantities	three moves altering sequence, two moves shifting lots backward or forward in time	All five moves yield one neighborhood. Also contains TS version results for CLSP without setup carryover
Karimi et al. (2006)	CLSP	BO, setup carryover	TS	setup variables	change single setup state	Evaluation of move via solving network flow problem

Table 4.1 cont. *– Abbreviations of features: ST…setup times, ST sd…sequence-dependent setup times, OT…overtime, BO…backorders. Abbreviations of metaheuristics: SA…Simulated Annealing, TS…Tabu Search, VNS…Variable Neighborhood Search, GA…Genetic Algorithm, TA…threshold accepting, LS…local search.*

Author(s)	***Model***	***Specific Features***	***Meta-heuristic***	***Solution Representation***	***Neighborhood(s)***	***Notes***
Almada-Lobo and James (2008)	CLSD	ST	TS, VNS	sequences of lots (jobs)	three moves altering sequence	General VNS; short-term memory TS with alternating neighborhoods
Almada-Lobo et al. (2008)	CLSD	real-world case in glass container industry	VNS	sequences of lots (jobs)	six moves altering sequence	Starts with RVNS, then switches to basic VNS
Laguna (1999)	CLSD	ST, OT, no setup costs	TS	sequences of lots (jobs) per period	four moves altering sequence	All four moves yield one neighborhood.
Hindi (1995)	CSLP	single item	TS	setup state variables z_{it}	change single z_{it}	Move evaluation via solving transshipment problem
Brüggemann and Jahnke (2000)	DLSP	batch availability	SA	setup variables	eliminating setup periods, then exchanging two production periods, then inserting setup periods anew	Two phases: first phase for finding feasible solutions, second phase for optimizing; see also multi-level version
Fleischmann and Meyr (1997)	GLSP	conservation of setup state after idle periods	TA	setup variables	three moves: insert, delete, swap	Deterministic change between the three moves based on search history. Move evaluation with three different greedy heuristics to compute lot sizes and inventories
Meyr (2000)	GLSP	ST sd	TA, SA	setup variables	as in Fleischmann and Meyr (1997)	Move evaluation via solving network flow problem
Meyr (2002)	GLSP	ST sd, parallel machines	TA, SA	setup variables	as in Fleischmann and Meyr (1997)	Move evaluation via solving network flow problem
de Araujo et al. (2008)	GLSP	real-world case in foundries	LS, SA	vector of items in first period	change entry in vector, i.e. schedule different item	Three LS variants: descent heuristic, diminishing neighborhoods (change Z=10,..,1 entries for given number of iterations), and SA

Two articles, Almada-Lobo and James (2008) as well as Almada-Lobo et al. (2008) employ a VNS for solving the CLSP with sequence-dependent setups. The former presents a General VNS with three neighborhoods resulting from manipulating the sequence of lots in a period. The second paper deals with a real-world case from the glass container industry and implements a Reduced VNS followed by a Basic VNS with six different neighborhoods, again altering the sequence of jobs.

Another real-world case can be found in de Araujo et al. (2008), who modeled the problem as a GLSP and solved it with three different local search approaches: a basic descent method, a "diminishing neighborhood" method, and Simulated Annealing. The diminishing neighborhoods resulted from the stepwise reduction of the number of elementary changes to the sequence of lots. Several other applications of point-based metaheuristics for single-level lotsizing can be looked up in Table 4.1 on pages 54 and 55.

Population-based metaheuristics have of course been employed also. Applications of Evolutionary Algorithms include Hung et al. (1999), as well as Süer et al. (2008) for the CLSP, a Genetic Algorithm for the SLULSP with batch ordering by Gaafar (2006), and a hybrid GA for the CLSD by Miller et al. (1999).

4.2 Multi-Level Lotsizing

Several articles apply more than one metaheuristic to the MLCLSP and present comparisons. A prominent example is Kuik et al. (1993), who tested Simulated Annealing and Tabu Search against LP-based heuristics for an assembly product structure with a single capacitated resource. Although SA and TS fare better, the authors are in favor of a compromise between metaheuristic and LP-based heuristic, as this also provides a much desired lower bound on the problem. Helber (1995) compared Simulated Annealing, Tabu Search, a Genetic Algorithm, and an Evolutionary Algorithm with a decomposition approach. While the GA performed poorly, SA and TS produced high quality results, needing however more computational time than decomposition. Shin (2007) implemented both Tabu Search and Simulated Annealing, with the former faring slightly better than the latter. Hung and Chien (2000) compared Simulated Annealing, Tabu Search and a Genetic Algorithm for the MLCLSP with multiple demand classes. Their findings point to TS and

SA outperforming the GA. Barbarosoglu and Özdamar (2000) concentrated on Simulated Annealing for the MLCLSP and tested several neighborhood transition schemes, allowing in turn infeasible solutions where both the inventory and capacity constraints are violated, partially infeasible solutions where just one of the two constraints is violated, and strictly feasible solutions to be visited during the search. The same authors presented a Lagrangean heuristic enhanced with Simulated Annealing in Özdamar and Barbarosoglu (2000). An extensive study of stochastic local search procedures, in particular Simulated Annealing, is presented in Jeunet and Jonard (2005), tackling the MLULSP however. Berretta et al. (2005) enhanced their procedure consisting of a smoothing, improvement and perturbation step with elements of Tabu Search and Simulated Annealing, which could improve the obtained solution quality. Chen and Chu (2003) embedded a local search based on the simplex tableau for the linear relaxation of the problem into their Lagrangean heuristic. Almeder (2010) enhanced his ACO procedure with two different local search steps, one altering the production quantities, the other manipulating the setup matrix. Some other applications of point-based metaheuristics to multi-level lotsizing problems are mentioned in Table 4.2 on pages 58 and 59.

Population-based methods dealing with the MLULSP are implemented in Dellaert and Jeunet (2000), as well as in Dellaert et al. (2000) and Prasad and Krishnaiah Chetty (2001), each presenting Genetic Algorithms; Homberger and Gehring (2009) and Pitakaso et al. (2007) with Ant Colony Optimization. The latter also tackled the MLCLSP in a related article from 2006. Both Xie and Dong (2002) and Fakhrzad and Khademi Zare (2009) employed Genetic Algorithms for the MLCLSP; Caserta et al. (2010) present a metaheuristic called corridor method for the MLCLSP with setup carryover. Kimms (1999) tackled the multi-level PLSP with a Genetic Algorithm.

Studying Tables 4.1 and 4.2, we can observe that most authors rely on straightforward, intuitive move mechanisms to generate neighboring solutions. Changing the value of a single setup variable seems to be a popular – and effective – move. The following chapter takes a closer look and also investigates several other possible neighborhoods that can be constructed through manipulating the setup variables.

Table 4.2 – Applications of single-point metaheuristics to multi-level lotsizing problems *(continued on next page)*

Author(s)	Model	Specific Features	Meta-heuristic	Solution Representation	Neighborhood(s)	Notes
Tang (2004)	MLULSP	assembly system	SA	setup variables	change single y_{it}, adapt setups of successor/predecessors	Experiments on parameters of SA (cooling schedule)
Jeunet and Jonard (2005)	MLULSP		SA	setup variables	1) change single y_{it} 2) change single y_{it} on next level in product structure in next iteration 3) extension of switch to all predecessors 4) extension of switch to immediate predecessors 5) insert or delete setup for single item and all its predecessors	Neighborhoods tested separately – in summary 280 stochastic LS versions tested; they are outperformed by problem-specific heuristics in terms of solution quality and computational times, except for large instances, where carefully calibrated search procedures were still slower, but yielded better results than problem-specific heuristics.
Kuik et al. (1993)	MLCLSP	assembly system with single bottleneck; no ST	SA, TS	setup variables	switch off one y_{it}	Move evaluation with greedy heuristic; comparison with LP-based heuristics
Salomon et al. (1993)	MLULSP/ MLCLSP	assembly system with single bottleneck; no ST	SA, TS	setup variables	change single y_{it} and five other, not specified "elaborate" moves	Elaborate moves are not better than simple ones; parameter settings require "a lot of trial-and-error"
Helber (1995)	MLCLSP		SA, TS, GA, EA	setup variables	change single y_{it}	Comparison of different solution methods for the MLCLSP
Shin (2007)	MLCLSP		SA, TS	setup variables	change single y_{it}	TS slightly better than SA
Chen and Chu (2003)	MLCLSP		LS	setup variables	exchange values of two setup variables	LS embedded in Langrangean Relaxation procedure; LS uses simplex method on the LP relaxation
Barbarosoglu and Özdamar (2000)	MLCLSP		SA	production quantities	shifting randomly decided quantity	Analysis of different neighborhood transition schemes – allowing (partial) infeasibility or not
Özdamar and Barbarosoglu (2000)	MLCLSP		SA	production quantities	shifting randomly decided quantity	SA embedded in Lagrangean heuristic

Table 4.2 cont. *– For abbreviations used see table 4.1*

Author(s)	Model	Specific Features	Meta-heuristic	Solution Representation	Neighborhood(s)	Notes
Almeder (2010)	MLCLSP	OT	ACO	production quantities /setup variables	shifting partial or complete lot of one item / release 10% of y_{it} variables and recalculate MIP	Two LS procedures embedded in ACO algorithm
Berretta et al. (2005)	MLCLSP	Echelon stock formulation	Hybrid TS and SA	production quantities	shifting partial or complete lots	Incorporation of TS and SA elements improved solution quality
Hung and Chien (2000)	MLCLSP	BO, multiple demand classes	SA, TS, GA	setup variables	change single y_{it}	TS and SA better than GA
Özdamar and Barbarosoglu (1999)	Multi-Level Capacitated Lotsizing and Loading Problem	OT, parallel facilities	SA and hybrid GA/SA	production quantities	shifting lot of one item	See also single-level version
Brüggemann and Jahnke (1994)	MLDLSP	two-stage system, batch availability	SA	setup variables	eliminating setup periods, then exchanging two production periods, then inserting setup periods anew	See also single-level version
Kimms (1996b)	MLPLSP	single machine	TS	indirect representation via demand arcs	redirecting one arc	Comparison with randomized regret-based procedure

5. Neighborhoods for the Multi-Level Capacitated Lotsizing Problem

As we have seen in chapter 3, move mechanisms and the resulting neighborhoods form an essential part of any local search based technique. With that in mind, the starting point for the research of this thesis was to devise a variety of possible neighborhoods for the MLCLSP, and to see what works well and what does not. Based on the insights gained from these experiments, the decision to implement a Variable Neighborhood Search was made later along the way. The output and results of this chapter can of course be used for other metaheuristic applications as well.

5.1 Eight Neighborhoods

As it consists of both continuous and binary decision variables, the MLCLSP lends itself to more than one possible search space and solution representation in a local search procedure. As the literature review has shown, various articles employ shifting actual lot sizes as move mechanisms, i.e. changing the continuous x_{it} variables. This thesis concentrates on the binary setup variables y_{it}: the search space is given by the set of all binary setup matrices. Each solution is represented by exactly one setup matrix, the manipulation of which gives its neighbors. Once the binary variables are fixed, i.e. their value is constant and known, only a linear program remains, which can be solved rather quickly by any optimization software, e.g. CPLEX, which was used in the implementation for this thesis. This not only determines the values of the continuous variables, but more importantly the total costs of a solution, which will be used as the evaluation criterion throughout the search.

- *Neighborhood 1:* ***Switch***

A single setup variable is switched, either from one to zero or from zero to one. This move constitutes the simplest, most intuitive, and therefore – as the literature review demonstrated – most widely used manipulation of the matrix. The resulting change in the total number of setups is either +1 or -1. As every y_{it} can be flipped, the total number of neighbors for each solution amounts to $N \times T$, where N and T are borrowed from the mathematical notation of the MLCLSP in chapter 2.2 and stand for the number of products and the number of periods respectively.

- *Neighborhood 2:* ***Swap***

The notion of shifting the production of a certain product either forward or backward can be captured by a move that swaps two setup variables of one product with diametrical values. One y_{it} is switched, while simultaneously a second variable with the same index i is switched in the opposite direction. What is not possible within this neighborhood is changing both variables from one to zero, or both from zero to one. Thus, the number of setups stays the same when applying the *Swap* move. The maximum number of neighbors is given by $N \times ((T \times (T-1)) / 2)$.

- *Neighborhood 3:* ***Double Swap***

Being an extension of the *Swap* move, this neighborhood is based on the idea that if the production of one product is shifted forward (backward), capacities in the target period might become an issue. To counteract this, the production of a second product in the target period is cancelled and, if possible, shifted backward (forward) to the period in which the first item was originally scheduled for production. In other words, the production of two different products in two different periods is swapped.

What is meant by "shifted if possible" in the previous paragraph? This is best explained with the help of Figure 5.1, where setup matrices for a problem with six products and periods each are depicted. In the upper half we see the "normal" case as envisioned when creating the move, where the production of item 2 is shifted backward from period 4 to period 2, while the production of item 5 is shifted in the opposite direction. If the setup matrix of the incumbent, however, happens to look like the one in the bottom half of Figure 5.1, the backward shift for item 2 still takes place, production of item 5 in the target period is still adjusted, but the production of item 5 in period 2 remains unchanged. The reason for this seemingly complicated rule is simply because otherwise the capacities in period 2 would most likely be exceeded. Furthermore, the concern that a perfect constellation like the one in the upper half of Figure 5.1 might not occur too often – equivalent to finding not enough neighbors – is reduced.

In conclusion, for both affected y_{it} variables of the second product to be switched, they must also have diametrical values, as is the case for the first product. Otherwise, only the setup variable of the second product in the target period will be flipped. The change in the

number of setups can therefore be either 0, +1 or -1. The maximum number of neighbors amounts to *N x (N-1) x ((T x (T-1)) / 2)*.

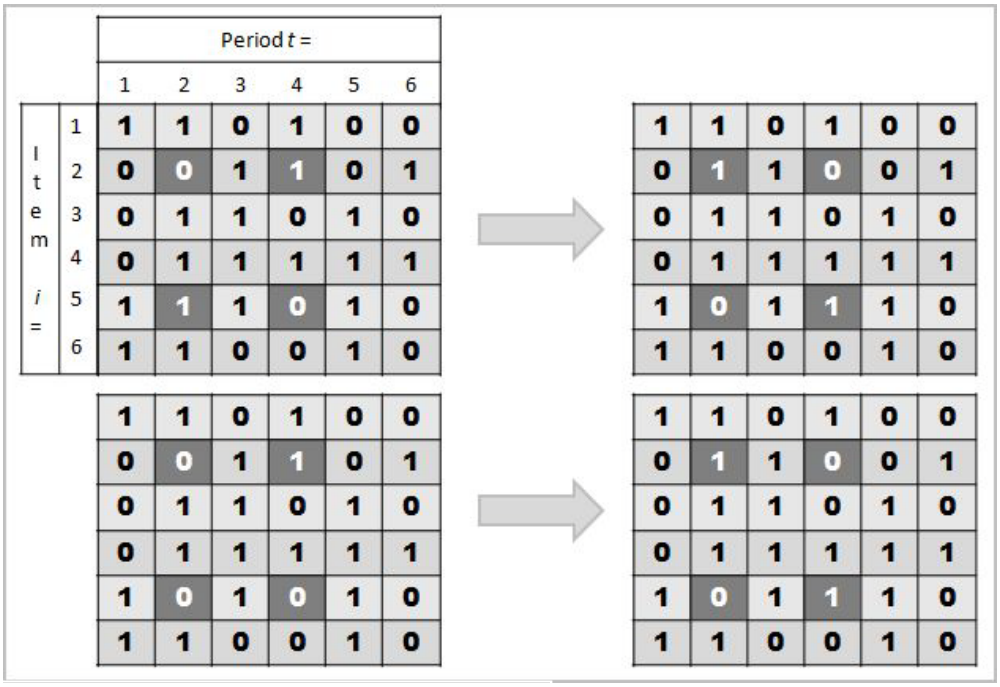

Figure 5.1 – The Double Swap neighborhood

- *Neighborhood 4:* ***Column Swap***

As the name of this neighborhood already gives away, the move mechanism consists of two columns of the setup matrix being swapped. In economic terms, this means that the whole setup pattern of two periods is exchanged. The total number of setups stays the same, and the total number of neighbors amounts to *(T x (T-1)) / 2*.

- *Neighborhood 5:* ***Switch 5***

This neighborhood was created in the hope of accelerating the search process of the *Switch* move, after having seen in first experiments with the latter that it works quite well in the sense of always finding better neighbors in runs with many iterations. The idea was to skip some iterations by changing not one setup variable at a time, but five. Thus, the move consists of simultaneously switching five y_{it} variables of the matrix to their opposite values, the change in the number of setups falls between +5 and -5, and the total number of neighbors amounts to $\binom{N\ x\ T}{5}$.

- *Neighborhood 6:* ***Insert 1***

The main idea for this neighborhood was to devise a move that will increase the number of setups. So, if we insert a setup variable with a value of one to replace any variable of the matrix, the total number of setups will certainly not go down. Technically, the move works as depicted in Figure 5.2. A variable with a value of 1 is inserted in any period for a given product, here in period 3 for product 1. The setup variables from this period onward are then shifted forward, and as the dimensions of the matrix must remain constant of course, the last value is pushed out of the matrix, in other words deleted. The resulting changes can be seen on the right-hand side of Figure 5.2. It should be noted that a move inserting 1 in the last period is not permitted, as this would generate a neighbor that a) can already be obtained through the *Switch* neighborhood, or b) has the same configuration as the incumbent.

The total number of setups either goes up by one, or remains unchanged in case the last variable of the row has a value of 1. The total number of neighbors amounts to *N x (T-1)*.

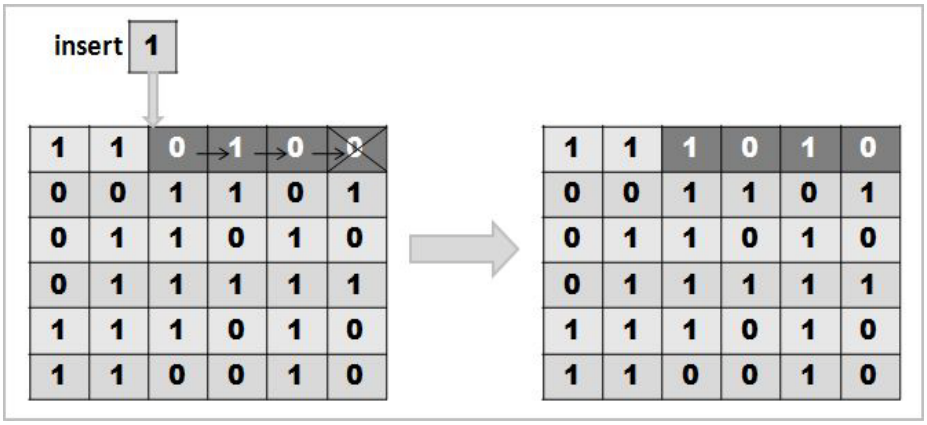

Figure 5.2 – The Insert 1 neighborhood

- *Neighborhood 7:* ***Delete***

With this move mechanism a second neighborhood likely to increase the number of setups was devised. If one setup variable of the matrix is deleted and a variable with a value of one is inserted to replace it, the number of setups will not be reduced. This will sound familiar to the attentive reader, but there is of course a difference to the *Insert 1* mechanism.

While a move with *Insert 1* always deletes the setup of the last period for the chosen item and the insertion can occur anywhere between periods 1 to *T-1*, *Delete* works the other way round. The value of any period between 2 and *T* can be deleted, while 1 is always inserted in the first period. Figure 5.3 should help in making the difference clear. The setup variable of item 1 in period 4 is deleted, causing the values up to that period to be shifted forward. The free spot in period 1 is then filled with a variable with a value of 1.

Period 1 is excluded from deletion for the same reason that the last period is not considered in neighborhood *Insert 1*. The two admittedly similar moves *Insert 1* and *Delete* can produce the same neighbor when the former inserts 1 in the first period and the latter deletes the last period. The change in number of setups for *Delete* is either +1 or 0, and the number of neighbors generated by this mechanism for each solution is *N x (T-1)*.

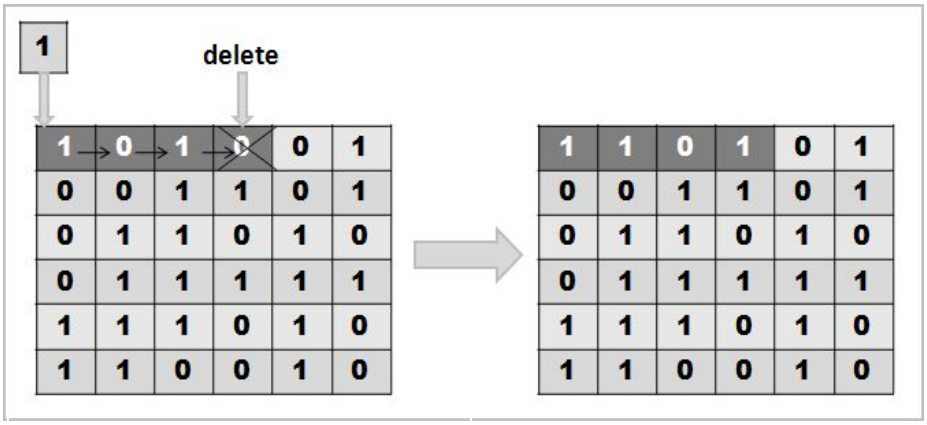

Figure 5.3 – The Delete neighborhood

- *Neighborhood 8:* ***Insert 0***

This is yet another neighborhood beside *Insert 1* and *Delete* where a setup variable is added to the matrix and another one is deleted. The difference with the *Insert 0* move is, as the reader will probably have already guessed from the name of the neighborhood, that this time *zero* is inserted anywhere between periods 2 and *T*. The setup variables of the affected product up to the chosen period are shifted backward, causing the first value of the row to be cut off. An insertion in the first period is again not permitted, as this would lead to either no new neighbor at all or one already present in the *Switch* neighborhood. The total number of setups will go down by one in case the first value of the row is 1, otherwise it will remain unchanged. The size of this neighborhood is the same as that of the *Insert 1* and *Delete* neighborhoods, *N x (T-1)*.

What should be noted for all of the neighborhoods is that unnecessary setups are adjusted automatically after the call to CPLEX. An unnecessary setup happens in the case where a setup for a given item in a given period occurs, but the item is not produced in that period. Mathematically speaking, y_{it} is 1 and x_{it} is 0. The total number of setups could therefore go down in any case.

5.2 Basic Local Search

5.2.1 Implementation Details

To test all of the above neighborhoods, a simple local search procedure was implemented in C++. Simple is used here to distinguish this approach from the other more sophisticated ones that are mentioned throughout the thesis, which are based on and extend local search steps. Simple refers to the classic iterative improvement algorithm as explained in chapter 3.2. Despite employing the basic form, there are, however, quite a few aspects where multiple options regarding the fundamental design of the procedure exist, as well as some parameters that need to be determined beforehand.

The maximum amount of computational time allowed per program run was set to 500 seconds independent of problem size, which was expected to be constraining for the large instances, but nonetheless expected to be permitting valuable insights regarding the individual neighborhoods. The maximum number of iterations allowed per run was set to a value so high that it would not prematurely terminate the search process, which would thus effectively only stop either when a local optimum was found, or when the time was up.

For the initial solution from where the local search starts, all setup variables are simply set to 1, and the corresponding objective function value is calculated by CPLEX. As mentioned previously, there exist more sophisticated construction heuristics, but for the purpose of testing the individual neighborhood mechanisms this approach provides rather quickly a feasible and oftentimes even quite good starting solution.

To avoid possible cycling between solutions, sideways moves, i.e. selecting a neighbor with the same objective function value as the incumbent, are not allowed.

The following fundamental aspects of the algorithm were not decided upon beforehand, but rather became an issue after the first rounds of testing, leading to modifications of the procedure along the way. In each case, both options were implemented and compared, providing valuable insights, which were then also used in the design of the VNS.

- *Exhaustive vs. random sample*

This represents a choice that directly affects the number of neighbors to be examined and in turn the computational effort of the algorithm. In the exhaustive search mode, the whole neighborhood is scanned. Another possibility would be limiting the number of neighbors checked at every iteration to some fixed value. Which neighbors are included in the subset is determined by randomly generated product and period indices in this implementation, leading to a desired stochastic element in the search. Instead of beginning every iteration at the left top of the setup matrix with the first product in the first period, and then scanning row after row for a better neighbor, jumping to a random point might lead to faster improvements.

- *Semi-random vs. fully random*

Fully random refers to every product and period index that is needed for a given move mechanism to be determined at random. If we look at neighborhood *Swap* for example, we have one product for which two periods are swapped, giving us three indices in total that characterize one move in this neighborhood. For a *Double Swap* move we need two period and two product indices, while for *Column Swap* only two periods have to be selected for swapping. In the semi-random mode only the starting point for the move is determined at random, and several neighbors are checked by letting the remaining indices run through from beginning to end. The pseudo-code sketch for the *Swap* neighborhood in Figure 5.4 will hopefully make the difference between exhaustive, semi-random and fully random modes clearer.

In the first version implemented, neighborhoods *Swap*, *Double Swap*, and *Column Swap* were given semi-random sample generation. There were concerns that in fully random mode not enough neighbors per iteration would be found, given that for example for a *Double Swap* one needs diametrical values of the y_{it} variables not only in the periods of the first product that are swapped, but also between the first and second product. It was hoped

that a compromise between exhaustive and totally random search would generate sufficient neighbors at each iteration.

```
// exhaustive mode
   for (product index  i = 1, 2,…, N)
      for (period index t  = 1, 2,…, T)
         switch setup variable y_it
         for (period index u = 1, 2,…, T)
            if (setup variable y_iu has same value as already switched y_it)
               switch y_iu
               evaluate neighbor
               if (neighbor better than incumbent)
                   break and start new iteration
               else undo changes and generate next neighbor

// semi-random mode
   while (maximum number of random neighbors not reached)
      generate random product index i and random period index t
      switch setup variable y_it
      for (period index u = 1, 2,…, T)
         if (setup variable y_iu has same value as already switched y_it)
            switch y_iu
            …
            …

// fully random mode
   while (maximum number of random neighbors not reached)
      generate random product index i and random period index t
      switch setup variable y_it
      generate random period index u
      if (setup variable y_iu has same value as already switched y_it)
         switch y_iu
         …
         …
```

Figure 5.4 – Three different implementations of the Swap neighborhood in pseudo-code

- *Best-improvement vs. first-improvement*

When selecting the neighbor to be visited next, one can employ a best-improvement strategy, which entails generating and evaluating all neighbors and then picking the one that reduces the objective function value the most; in other words the best neighbor. The second approach discussed in chapter 3.2 would be to generate and evaluate neighbor after neighbor and immediately jump to the first one that is better than the incumbent, ignoring the rest and a possibly even larger improvement. The first-improvement pivoting rule, however, might lead to faster descents.

- *Handling of backorders*

As we have seen in chapter 2, allowing items to be backordered is an extension of the standard MLCLSP model. In case they are not part of the problem originally, i.e. the inventory constraints are not relaxed, solutions that still rely on them in order to satisfy total demand are in fact infeasible solutions. As the final output of any algorithm should preferably be feasible, one can of course not allow backorders at all, or one can allow backorders temporarily and somehow make sure that the final solution will have none. The former approach is slightly relaxed and implemented using so-called *instant elimination*: the model formulation implemented contains the respective backorder variables, i.e. backorders are still allowed or rather possible, but the respective cost parameters are set so high from the beginning that visiting solutions with backorders is practically prohibited. The second approach, an adaptive strategy, is handled in the following way: every time twenty solutions containing backorders are visited during the search, backorder costs are doubled, thereby *gradually eliminating* the infeasibility. Another way to look at this is interpreting the backorder costs in the objective function as a penalty term for infeasible neighbors, making them less attractive. The reason for including backorders in the implemented model, and in doing so extending the search space, is based on the notion discussed in chapter 3.2 that high quality solutions are often on the edge of infeasibility and can be reached from both sides of that line.

Table 5.1 – Problem class dimensions of the MLCLSP test instances used for evaluation of the local search procedures

Problem Class	*Periods*	*Products*	*Resources*	*Setup Times*	*Classification*	*Number of instances used*
A	4	10	3	no	very small	2
B	4	10	3	yes	very small	2
A^+	24	10	3	no	small	3
B^+	24	10	3	yes	small	4
C	16	40	6	no	medium	5
C^m	16	40	9	no	medium	3
D	16	40	6	yes	medium	5
D^+	48	40	6	yes	large	2
E	16	100	10	no	large	3
E^+	48	100	10	no	large	1

All tests were run on a personal computer with an Intel Core Duo2 2.00 GHz processor, 2046 MB RAM and Microsoft Windows Vista, using CPLEX 11.0. Input data for all program runs were test instances from the problem libraries of Tempelmeier and Derstroff (1996), and the extended libraries from Stadtler (2003). 30 instances in total from all problem classes were chosen at random for testing the local search algorithms. Each instance was solved exactly once for every specific mode and constellation of parameters. Table 5.1 provides an overview over the dimensions of the individual problem classes.

5.2.2 Computational Results

5.2.2.1 Tuning

Tuning refers to the testing of the different local search options outlined above, and comparing the results before finally deciding on which version to use for further implementations, especially of course the Variable Neighborhood Search.

- *Exhaustive vs. random sample*

During first runs of the program, it soon became obvious that the exhaustive mode takes up too much time: neighborhoods *Swap* and *Double Swap* reached the limit of 500 seconds even for the small problem instances, and for neighborhood *Switch* time became an issue from the medium instances onward. Limiting the amount of neighbors per iteration to a random sample naturally increased the speed of the algorithm, but the question remained how this affected the solution quality. Table 5.2 gives the rather surprising answer. For comparison, all exhaustive and random sample program runs with all neighborhoods were considered, and average computational times, average improvement of the initial solution, as well as the average number of iterations were calculated. The random sample mode is not only faster, but generates on average better final solutions as well, probably due to the higher number of iterations carried out during the search. Below the overall average results in Table 5.2, some individual examples help further demonstrate the difference. While the first example shows an expected outcome where exhaustive mode is slower but better, the second small instances example proves that faster *and* better is possible, too. As already mentioned above, the stochastic nature of the sample mode might be responsible for this superiority. The larger the problem instances become, fewer and fewer iterations can be carried out in the exhaustive mode, which explains the low improvement rates. The

computational time of the random sample mode naturally also rises with problem size, but enough iterations fit in the 500 seconds time limit, so that good solutions can still be found.

Conclusively, even for small instances the exhaustive mode does not guarantee a better result, much less for the medium and most of all the large instances, so the decision to concentrate on the random sample mode was easy.

Table 5.2 – Comparison of the local search results for the exhaustive and random sample modes

	Exhaustive mode	***Random sample mode***
Average computational time	*244 sec.*	***137 sec.***
Average improvement	*18,37%*	***21,45%***
Average number of iterations	*33*	***58***
Small instances example – Swap		
Iterations	***19***	*10*
Computational time	*536 sec.*	***89 sec.***
Improvement	***9,74%***	*7,60%*
Small instances example –Switch		
Iterations	***50***	***50***
Computational time	*172 sec.*	***72 sec.***
Improvement	*53,99%*	***54,72%***
Medium instances example – Switch		
Iterations	*16*	***38***
Computational time	*515 sec.*	***222 sec.***
Improvement	*26,56%*	***30,17%***
Large instances example –Double Swap		
Iterations	*2*	***9***
Computational time	*589 sec.*	***557 sec.***
Improvement	*4,72%*	***25,51%***

Regarding the size of the random sample, where we are again confronted with a trade-off between computational time and solution quality, the results point to 50% of the setup variables for small instances, 30% for medium instances, and 10% for large instances respectively being a reasonable amount of neighbors to examine at each iteration.

- *Semi-random vs. fully random*

Table 5.3 compares the performances of *Swap*, *Double Swap*, and *Column Swap* in the fully random and the semi-random modes as described previously. Regarding the *Swap* move, the average improvement rate for the fully random mode was exceptionally bad, so for this neighborhood the semi-random mode was kept for further implementations. In regards to *Double Swap* and *Column Swap*, it should be noted that the semi-random version was coupled with the best-improvement approach (see results below), while the

fully random mode was tested together with the first-improvement rule, which makes it more difficult to ascertain the effect of one modification only. Although the improvement rate in the *Double Swap* neighborhood dropped significantly, the prohibitively high computational times of the first version – reaching the time limit in random sample mode already for small instances – tipped the scales in favor of the second version, i.e. first-improvement with fully random neighbor generation. Neighborhood *Column Swap* was also switched to the fully random mode.

Table 5.3 – Comparison of the local search results for the fully random and semi-random modes

	Average computational time	***Average improvement***	***Average number of iterations***
Swap			
Fully random	***20 sec.***	*0,77%*	*4*
Semi-random	*67 sec.*	***8,48%***	***38***
Double Swap			
Fully random	***38 sec.***	*10,48%*	***60***
Semi-random	*369 sec.*	***20,13%***	*8*
Column Swap			
Fully random	***3 sec.***	*9,90%*	***19***
Semi-random	*150 sec.*	***12,08%***	*12*

- *Best-improvement vs. first-improvement*

As was implied before, testing started with a best-improvement program version, which quickly turned out to be too time-consuming. The switch to the first-improvement strategy brought about the expected and desired reduction in computational effort, as Table 5.4 shows. Even better news, however, is that the solution quality did not suffer at all, staying nearly constant at 20,64%. An explanation for this might come from the fact that on average more iterations per program run are carried out during a first-improvement search, similar to what could be observed in the exhaustive vs. random mode comparison.

Table 5.4 – Comparison of the local search results for the best-improvement and first-improvement modes

	Average computational time	***Average improvement***	***Average number of iterations***
Best-improvement	*197 sec.*	***20,65%***	*21*
First-improvement	***127 sec.***	*20,64%*	***85***

- *Handling of backorders*

When allowing backorders in any case and setting the respective costs to 10.000 monetary units, eight of the thirty random instances turned out to be affected. Their solutions were re-calculated using instant elimination as well as gradual elimination. For the former, backorder costs were set to 100.000 from the beginning, while in the latter version they were doubled to 20.000 after twenty solutions containing backorders had been visited. For the comparison between the two, the neighborhood combinations[19] yielding the best result with instant elimination were put up against the same neighborhood combination with gradual elimination, and vice versa. In addition to that, both versions were tested with neighborhood *Switch*, giving in total 18 objective function values for each variant that could be compared. Gradual elimination of backorders led to better results in about 72% (13 out of 18 times), so it was maintained for the Variable Neighborhood Search implementation.

Based on the above results, a first-improvement local search with fully random samples (except for the *Swap* neighborhood with semi-random samples), allowing backorders and employing a gradual elimination should they occur, can be recommended. We will next take a look which of the neighborhoods themselves can be recommended.

5.2.2.2 Comparison of Neighborhoods

For the comparison of the individual neighborhoods several figures are shown in Table 5.5 on page 75. Only one run per instance with the same random sample size and first-improvement mode was considered for each neighborhood. The first row of the table gives the number of successful attempts to improve the initial solution, expressed in percentages of the thirty instances in total. Neighborhoods *Switch* and *Insert 0* always found better neighbors, with the *Switch 5* move coming in third place. Rows 2 to 5 reveal that the nonetheless very high overall success rate of 90% is due to failing at most of the very small instances, which is not surprising given the rather big moves of *Switch 5*. The results for the rest of the neighborhoods are rather discouraging. The *Swap* mechanism could improve the initial solution in only half of the instances; the others have even lower rates. They do however exhibit a tendency to work best for medium-sized instances. Given the nature of the move mechanisms of *Swap*, *Double Swap*, *Column Swap*, *Insert 1*, and *Delete*, an

[19] Combinations of neighborhoods are dealt with in detail later in chapter 5.3.

initial solution with all setup variables set to 1 bears a high risk of not finding any better neighbors, simply because there might be no neighbors at all. Only removing unnecessary setups makes it possible to start from such a solution with these neighborhoods. Starting from a different solution has been tried in the course of testing neighborhood combinations, for which the results are presented later.

The average improvement rates in percentages of the initial solution reflect the failure of this group of neighborhoods to find better neighbors. With just around 8%, neighborhoods *Swap* and *Double Swap* fare best, but in comparison with average improvements of over 30% for *Switch* and *Switch 5*, this result pales. The *Insert 0* move yields a rate approximately half as good as the *Switch* move. The slightly lower improvements in the very small problem class are due to the optimal solution being nearly found just with the initial setup configuration in some of the instances. Slightly surprising is the very low rate of neighborhood *Double Swap* for the small instances because of the rather constant and not too disappointing improvements – considering that this very move had the fewest successful attempts of all neighborhoods – in every other problem class.[20]

The average computational time is another crucial measure when evaluating any heuristic procedure. Neighborhoods *Switch* and *Switch 5*, which yield the highest improvements, also take the longest in doing so. The hope that *Switch 5*, with its altering of not one but five setup variables at a time, would result in an accelerated version of the simple *Switch* move was crushed, as the former instead takes longer than the latter. Both settle around the 500 seconds time limit for the large instances, but for the smaller problem classes especially *Switch* rather quickly finds a local optimum. The *Insert 0* move reaches half the improvement rate of *Switch* also in half the time. Neighborhoods *Swap*, *Double Swap*, *Column Swap*, *Insert 1*, and *Delete* all show lower average running times in general because half of the time or less they could not find better neighbors at all, resulting in very short local searches. The *Swap* move takes the longest of the group, a consequence of keeping the semi-random mode for this neighborhood, which however also led to the highest improvement rate among the group.

[20] It should be noted here that for *Double Swap* and *Column Swap* the numbers differ from those of the fully vs. semi-random comparison above, as additional results, e.g. from different sample sizes, were considered there.

Table 5.5 – Percentage of successful attempts to improve the initial solution, average improvement of the initial solution, and average computational time in seconds for the individual neighborhoods. *Average improvement rates are given in percent of the initial solution.*

Neighborhood	*Switch*	*Swap*	*Double Swap*	*Column Swap*	*Switch 5*	*Insert 1*	*Delete*	*Insert 0*
Successful attempts	***100%***	*50%*	*37%*	*43%*	*90%*	*43%*	*40%*	***100%***
Very small instances	***100%***	*50%*	*25%*	*25%*	*25%*	*25%*	*25%*	***100%***
Small instances	***100%***	*43%*	*14%*	*43%*	***100%***	*43%*	*43%*	***100%***
Medium instances	***100%***	*62%*	*54%*	*54%*	***100%***	*62%*	*54%*	***100%***
Large instances	***100%***	*33%*	*33%*	*33%*	***100%***	*17%*	*17%*	***100%***
Average improvement	***34,83%***	*8,48%*	*8,37%*	*3,86%*	*30,70%*	*2,68%*	*0,75%*	*18,86%*
Very small instances	***16,72%***	*0,75%*	*9,30%*	*0,75%*	*8,09%*	*0,01%*	*0,01%*	*11,42%*
Small instances	***42,14%***	*15,56%*	*0,56%*	*11,59%*	*39,10%*	*2,19%*	*1,87%*	*19,93%*
Medium instances	***34,82%***	*9,11%*	*12,00%*	*2,40%*	*31,32%*	*5,01%*	*0,73%*	*19,30%*
Large instances	***38,38%***	*4,02%*	*8,99%*	*0,06%*	*34,66%*	*0,00%*	*0,00%*	*21,64%*
Average computational time	*138,90*	*66,33*	*20,17*	***3,43***	*156,43*	*13,87*	*8,70*	*69,47*
Very small instances	***0,00***	***0,00***	*0,50*	***0,00***	***0,00***	***0,00***	***0,00***	***0,00***
Small instances	*28,86*	*31,57*	***0,57***	*2,00*	*11,43*	*1,86*	*2,00*	*4,14*
Medium instances	*78,92*	*85,46*	*8,08*	***3,38***	*117,08*	*19,31*	*7,08*	*27,85*
Large instances	*489,83*	*109,67*	*82,33*	***7,50***	*515,17*	*25,33*	*25,83*	*282,17*

Table 5.6 – Average number of iterations, average improvement of the initial solution per iteration, and average computational time per iteration in seconds for the individual neighborhoods. *Average improvement rates per iteration are given in percent of the initial solution.*

Neighborhood	*Switch*	*Swap*	*Double Swap*	*Column Swap*	*Switch 5*	*Insert 1*	*Delete*	*Insert 0*
Average number of iterations	***195,43***	*37,87*	*34,17*	*7,77*	*85,43*	*9,30*	*2,53*	*53,37*
Very small instances	***6,50***	*1,50*	*5,00*	*1,75*	*1,75*	*1,25*	*1,25*	*2,00*
Small instances	***98,71***	*39,00*	*1,71*	*18,71*	*29,29*	*4,14*	*4,14*	*17,57*
Medium instances	***174,38***	*44,85*	*34,00*	*6,31*	*86,46*	*18,15*	*2,54*	*49,54*
Large instances	***479,83***	*45,67*	*91,83*	*2,17*	*204,50*	*1,50*	*1,50*	*137,67*
Average improvement per iteration	*0,18*	*0,22*	*0,25*	***0,50***	*0,36*	*0,29*	*0,30*	*0,35*
Very small instances	*2,57*	*0,50*	*1,86*	*0,43*	*4,62*	*0,01*	*0,01*	***5,71***
Small instances	*0,42*	*0,40*	*0,33*	*0,62*	***1,34***	*0,53*	*0,45*	*1,13*
Medium instances	*0,20*	*0,20*	*0,35*	*0,38*	*0,36*	*0,28*	*0,29*	***0,39***
Large instances	*0,08*	*0,09*	*0,10*	*0,03*	***0,17***	*0,00*	*0,00*	*0,16*
Average computational time per iteration	*0,71*	*1,75*	*0,59*	***0,44***	*1,83*	*1,49*	*3,43*	*1,30*
Very small instances	***0,00***	***0,00***	*0,10*	***0,00***	***0,00***	***0,00***	***0,00***	***0,00***
Small instances	*0,29*	*0,81*	*0,33*	***0,11***	*0,39*	*0,45*	*0,48*	*0,24*
Medium instances	*0,45*	*1,91*	***0,24***	*0,54*	*1,35*	*1,06*	*2,79*	*0,56*
Large instances	*1,02*	*2,40*	***0,90***	*3,46*	*2,52*	*16,89*	*17,22*	2,05

The comparison of the neighborhoods is continued in Table 5.6 on page 76, which shows the average number of iterations per run, as well as average improvement rates and computational times in seconds per iteration. The average number of iterations for *Swap*, *Double Swap*, *Column Swap*, *Insert 1*, and *Delete* again reflects all the instances where no real search could be carried out, resulting in many runs with only one iteration. Regarding neighborhoods *Column Swap*, *Insert 1*, and *Delete*, the instances they actually could tackle also resulted in runs with not too many iterations, explaining why they fared so badly in the average improvement department. *Swap* and *Double Swap* reached their slightly higher improvement rates with more iterations. It appears that with the multiple switch of *Switch 5* some iterations with the single switch of *Switch* can be skipped as intended, but as we have seen above, at the price of slightly less improvement in longer computational times. In general and as expected, the larger the problem instances become, the more iterations are carried out, at least in the well-functioning neighborhoods.

When looking at the other figures in the middle and at the bottom of Table 5.6, the *Column Swap* neighborhood turns out the definite winner with the highest average improvement per iteration in the shortest average computational time per iteration. One could therefore conclude that it would not hurt much to try this neighborhood in any local search procedure, just not as a stand-alone, as the previous results of Table 5.5 imply. This could also be said for the *Double Swap* move. Another neighborhood that fares rather well is *Insert 0*, while *Switch 5* is convincing in the improvement category, but takes up too much time for a single iteration. The *Switch* neighborhood, top of the class so far, shows the lowest average improvement per iteration, but at least in an acceptable computational time.

Of the 30 instances in total, *Switch* yielded the best result 21 times, followed by *Swap* and *Switch 5* with three times each, where the former always won in the small problems class, and the latter in the medium-sized class. *Insert 0* equaled the result of *Switch* in two of the very small instances, and *Double Swap* came out on top once also for a class B problem. Neighborhoods *Column Swap*, *Insert 1*, and *Delete* never yielded the best solution, which does not mean that they always led to the worst results though. Tables 5.7 and 5.8 on the following pages offer another comparative look at the individual neighborhoods.

Table 5.7 – Percentage of instances for which the individual neighborhoods yielded a better final solution than the other neighborhoods

Neighborhood	***Switch***	***Swap***	***Double Swap***	***Column Swap***	***Switch 5***	***Insert 1***	***Delete***	***Insert 0***
	yielded a better result in … % of the 30 instances							
than…								
Switch	***---***	*10*	*3*	*0*	*10*	*0*	*0*	*0*
Swap	***90***	***---***	*17*	*10*	***73***	*3*	*3*	***73***
Double Swap	***97***	***37***	***---***	*23*	***80***	*20*	*13*	***73***
Column Swap	***100***	***40***	***27***	***---***	***83***	*7*	*3*	***93***
Switch 5	***90***	*20*	*10*	*7*	***---***	*0*	*0*	*17*
Insert 1	***100***	***43***	***33***	***43***	***90***	***---***	*0*	***97***
Delete	***100***	***43***	***37***	***43***	***90***	***23***	***---***	***100***
Insert 0	***93***	*27*	*27*	*7*	***83***	*3*	*0*	***---***

Table 5.7 shows the percentage of all instances for which the local search with any given neighborhood ended in a better result than with its competitors.[21] For example, *Switch* always came out on top of *Column Swap*, *Insert 1*, and *Delete*, but in 10% of the instances led to a worse solution than both *Swap* and *Switch 5*. The *Delete* move, exhibiting the worst performance so far, could still beat the *Double Swap* in 13% of the instances, but generally the table corroborates the findings so far, namely the superiority of neighborhoods *Switch*, *Switch 5*, and *Insert 0* in that order.

Table 5.8 gives details on the relative quickness of the individual neighborhoods. Here the *Column Swap* and the *Double Swap* moves dominate, but no single neighborhood was faster than the others *every* time. Although *Switch* has a more than twice as high average computational time than *Swap*, the former still reached its local optimum sooner than the latter in 60% of the instances. Even the slowest neighborhood *Switch 5* tops the *Swap* move, rendering it more or less the loser of this comparison.

[21] The complementary values not always adding up to 100% is due to two neighborhoods yielding the same result, oftentimes the unchanged initial solution, for one instance.

Table 5.8 – Percentage of instances for which the individual neighborhoods were faster than the other neighborhoods

Neighborhood	***Switch***	***Swap***	***Double Swap***	***Column Swap***	***Switch 5***	***Insert 1***	***Delete***	***Insert 0***
	was faster in … % of the 30 instances							
than…								
Switch	***---***	*27*	***91***	***92***	*41*	***85***	***92***	***80***
Swap	***60***	***---***	***67***	***79***	***53***	***80***	***79***	***60***
Double Swap	*9*	*17*	***---***	***67***	*9*	*40*	*38*	*18*
Column Swap	*0*	*7*	*33*	***---***	*0*	*8*	*0*	*8*
Switch 5	***56***	*33*	***91***	***92***	***---***	*77*	***92***	***85***
Insert 1	*8*	*7*	***60***	***67***	*15*	***---***	***33***	*8*
Delete	*0*	*7*	***63***	***67***	*0*	*25*	***---***	*8*
Insert 0	*7*	*27*	***73***	***85***	*11*	*77*	***83***	***---***

It is of course also interesting to know *how much* better and faster than their competitors the individual neighborhoods were. Appendix A provides further tables with some answers for the interested reader.

5.3 Sequential Local Search

When devising several neighborhoods, not only their stand-alone behavior is interesting, but also the possibilities of joint applications. While in the previous chapter the question was what worked well and what did not, we are now concerned with what works well *together* and what does not.

For the examination of sequential combinations of neighborhoods, the local search procedure did not have to be changed, as the neighborhoods were simply searched after one another. For example, we start with neighborhood *Insert 0*, the so found local optimum becomes the initial solution for the second neighborhood chosen, let us say *Double Swap*, again the algorithm finds the local optimum, which will be the input for the third neighborhood, and so on. The sequence can also contain any given neighborhood more

than once. Combinations of up to four neighborhoods were tested, but due to the immense number of possible sequences with eight different neighborhoods, a concentration on certain combinations was inevitable.

Having learned from the single neighborhood runs that in about half the time some neighborhoods will not find better or any neighbors at all and leave the initial solution unchanged, turning the attention to combinations starting with a reliable neighborhood seemed rational. Consequently, all pairings starting with the *Switch* neighborhood were examined thoroughly. Putting *Insert 0* and *Switch 5* up front also seemed like a good idea, while at the same time leaving out the *Switch* move completely did not, so only two more pairs – *Insert 0/Switch* and *Switch 5/Switch* – made the list. Regarding combinations with three and four neighborhoods, several of those were tested, but as no real trend towards a clear winner manifested after the first runs, pursuing this direction seemed futile. Nonetheless, insights could still be gained whether sequential local search runs with more than one neighborhood are advisable or not.

For this comparison only 21 of the 30 instances were used, which should explain any deviations of the figures for the *Switch* neighborhood from the previous results. Stand-alone *Switch* was included to be able to measure the impact of adding a second neighborhood to it, instead of just comparing the combinations among themselves.

The maximum amount of time allowed for computation for the sequential runs was increased to 600 seconds split equally among the neighborhoods used, in order to account for the time needed for additional program-related necessities, such as initializations of data structures or writing of statistics.

Table 5.9 on page 82 presents a comparative look at the selected pairings, giving the usual figures for the average number of iterations, average improvement and computational time in total and per iteration, as well as the percentage of successful attempts of the second neighborhood to further improve the solution. Regarding the last criterion, employing *Switch* after another neighborhood – *Insert 0* and *Switch 5* in this case – always leads to further improvements of the solution. This reinforces the finding of the single neighborhood runs, where the *Switch* move also had a 100% success rate – implying that no matter whether one starts from a quickly constructed initial solution or an already improved intermediate solution, the single switch mechanism still produces better

neighbors. The starting point for *Switch* does matter however in terms of improvement and computational time, where the former is slightly higher when applying *Insert 0* or *Switch 5* first, while at the same time the latter naturally rises also. Interestingly, whereas stand-alone *Switch 5* yielded a slightly higher average improvement than *Insert 0*, this is now reversed, with combination *Insert 0/Switch* being better – and faster, as could be expected, too – than *Switch 5/Switch*.
Regarding the pairings with neighborhood *Switch* at the front, it was hoped that starting from an already advanced setup configuration with – simply put – enough zeros in the matrix would give the setup-increasing neighborhoods *Delete* and *Insert 1* as well as the three swap mechanisms the chance to show their potential, as they might have been impeded by the all-set-to-1 initial solution in the single runs. Alas, with the exception of the *Swap* move, they still disappoint. *Column Swap* managed to further improve the objective function in only one of the instances, reaching a success rate of just 5%, while *Insert 1* could at least find a better neighbor in about half the time, with *Double Swap* and *Delete* lying in-between. The result is that the average improvement stays nearly constant at about 38%, which can be reached with *Switch* alone in less time.

With respect to *Insert 0* and *Switch 5*, Table 5.9 shows that beginning the search with either one of these two and using *Switch* afterwards leads to a better result than the other way round. Neighborhood *Swap* could increase its successful attempts from 50% in stand-alone runs to very good 95% when applied after *Switch*. The consequence is the highest average improvement of all combinations, but this comes at a price – the highest computational time by far. Considering the magnitude of the difference in average improvement to stand-alone *Switch* – not quite one percent – the more than doubled runtime of 255 seconds becomes rather unacceptable.

It should also be noted explicitly that compared to all pairs starting with *Switch*, the stand-alone *Switch* neighborhood yields the highest average improvement per iteration – in the lowest average computational time per iteration.

As for the single neighborhoods, appendix A provides additional tables depicting in detail who beat whom how often and by what margins.

Table 5.9 – Results of sequential local search with two neighborhoods. *Successful attempts are given as percentage of the 21 test instances used. Average improvement rates are in percent of the initial solution.*

Neighborhood combination	Successful attempts of second neighborhood to further improve solution	Average improvement of initial solution	Average computational time	Average number of iterations	Average improvement per iteration	Average computational time per iteration
Switch/Swap	*95%*	***38,85%***	*255,43 sec.*	***185***	*0,210%*	*1,382 sec.*
Switch/Double Swap	*29%*	*38,07%*	*130,24 sec.*	*174*	*0,219%*	*0,748 sec.*
Switch/Column Swap	*5%*	*37,99%*	***104,52 sec.***	*172*	*0,221%*	***0,608 sec.***
Switch/Switch 5	*19%*	*38,09%*	*140,86 sec.*	*174*	*0,218%*	*0,808 sec.*
Switch/Insert 1	*52%*	*38,13%*	*148,19 sec.*	*179*	*0,213%*	*0,828 sec.*
Switch/Delete	*33%*	*38,10%*	*150,10 sec.*	*179*	*0,213%*	*0,839 sec.*
Switch/Insert 0	*19%*	*38,02%*	*140,71 sec.*	*172*	*0,220%*	*0,816 sec.*
Insert 0/Switch	***100%***	*38,46%*	*153,81 sec.*	*164*	*0,234%*	*0,935 sec.*
Switch 5/Switch	***100%***	*38,30%*	*208,33 sec.*	*138*	***0,278%***	*1,510 sec.*
Switch	**---**	*37,97%*	*100,76 sec.*	*171*	*0,222%*	*0,589 sec.*

It was mentioned earlier that the instances were not only solved with the pairings of Table 5.9, but also with combinations of up to four neighborhoods. Average improvements and computational times aside, it is now time to take a look which neighborhood / neighborhood combination came up with the best absolute result, i.e. the lowest objective function value, how often.

Table 5.10 shows the winners in terms of the best absolute result. It corroborates the findings of the neighborhood pairings presented above, namely a dominance of combinations *Switch/Swap* and *Insert 0/Switch* with stand-alone *Switch* being able to keep up with them. Regarding the other combinations, the table makes it clear that there is no real trend towards one being the best of them all. One can try any combination of the eight neighborhoods and get lucky, but in practice one can of course not always employ all of them to reach the best possible result. As the top three of the table however account for about two thirds of the best results, a concentration on them is warranted and recommended.

Table 5.10 – Number of times a given neighborhood / neighborhood combination yielded the best solution to an instance

Neighborhood(s)	*Number of instances*	*Cumulative percentage of instances*
Switch/Swap	*8*	*26,7%*
Switch	*6*	*46,7%*
Insert 0/Switch	*5*	*63,4%*
Switch 5/Switch	*3*	*73,4%*
Swap/Switch/Double Swap/Switch	*2*	*80,1%*
Double Swap	*1*	*83,4%*
Column Swap/Switch/Swap	*1*	*87,7%*
Switch/Delete/Switch	*1*	*91%*
Switch/Swap/Switch/Double Swap	*1*	*94,3%*
Column Swap/Double Swap/Swap/Switch	*1*	*97,6%*
Switch/Column Swap/Double Swap/Switch	*1*	*100%*

5.4 Solution Quality of the Local Search Algorithms

Up to now, the results presented only compared the neighborhoods to themselves. As there are known solutions to most of the instances of the Tempelmeier and Stadtler problem libraries, one can of course compare the results of all the local search implementations so far with them. From the solutions reported by Tempelmeier and Derstroff (1996), Stadtler (2003), Pitakaso et al. (2006) and Almeder (2010), the lowest objective function value for the respective instance is used as a benchmark termed *Best solution*.

Table 5.11 – Comparison of the local search results with external results. *Only 25 of the 30 random instances from all problem classes are considered.*

	Best combination	**Switch**	**Switch/Swap**	**Insert 0/Switch**
Average gap to best solution	*3,03%*	*6,47%*	***4,48%***	*5,98%*
Subset gaps				
Very small instances	*2,94%*	***3,14%***	***3,14%***	*3,63%*
Small instances	*3,99%*	*9,32%*	***4,90%***	*7,31%*
Medium instances	*2,07%*	*5,86%*	***3,94%***	*5,51%*
Large instances	*5,63%*	***6,73%***	*8,99%*	*8,92%*
Maximum gap to best solution	*14,11%*	*17,41%*	***14,79%***	*16,77%*
Number of instances where…				
gap < 0%	*3*	*2*	*3*	*1*
0% <= gap < 5%	*14*	*8*	*12*	*9*
5% <= gap < 20%	*8*	*15*	*10*	*15*
gap >= 20%	*0*	*0*	*0*	*0*
Average gap to lower bound	*18,05%*	*22,19%*	***19,80%***	*21,59%*
Average computational time	*216 sec.*	***139 sec.***	*265 sec.*	*161 sec.*

For this comparison with external solutions, only backorder-free, i.e. feasible results of the local search runs were considered. Furthermore, the instances of problem class D had to be re-calculated, as the input data turned out to be incorrect after the first runs, which did not

matter for the internal comparison of the neighborhoods because they all had the same – albeit wrong – numbers. For five of the thirty instances no comparable result was available, leaving 25 instances in total to which Table 5.11 refers.

In addition to the best results reached with any of the combinations tested, results for the three contestants most likely to yield a best result – stand-alone *Switch*, *Switch/Swap*, and *Insert 0/Switch* – are also compared with the lower bound and the best known solution for any given instance. The overall average gap to the latter amounts to a surprisingly low 3,03% for the best combination, while it naturally rises when considering only the results of *Switch, Switch/Swap*, and *Insert 0/Switch*, but still lies within an acceptable range. A look at the average gaps for the respective subsets of the instances reveals that the simple local search fares worst in the large problem class. If one bears in mind however that the computational time was limited to just ten minutes, this is then a) not surprising, and b) actually quite a good result under the circumstances. Rather surprising are the average gaps in the small problem class, especially the very high 9,32% for *Switch*, which are mostly due to the one A+ instance that brought about the maximum gap for all four constellations, bringing down the average. While the approximately 3% average gap for the very small instances is rather mediocre, considering that these problems can be solved to optimality with CPLEX in no time, the results for the medium-sized problem class are encouraging. The average gaps to the lower bounds, as calculated by Tempelmeier and Stadtler respectively, range from about 18% for the best combination to 22% for the *Switch* neighborhood.

In the bottom half of Table 5.11 the reader can find some absolute numbers showing how often the gap to the best known solution fell into a certain range. Each constellation was able to improve at least once a (previously) best result, and – as the maximum gap already gave away – never yielded a result with a gap higher than 20%. For the best combination and *Switch/Swap*, the majority of instances fall into the "good" category of less than 5%, while *Switch* and *Insert 0/Switch* mostly reach mediocre results with a gap between five and twenty percent.

The last row of the table gives the average computational time for each constellation. We can see that it boils down to a trade-off between solution quality and computational speed once again. Neighborhood *Switch* is the fastest method, but also the one with the largest average gap. Using combination *Insert 0/Switch*, one can reduce the gap a little in slightly

more time. A two percent reduction of the gap when employing *Switch/Swap* costs about 120 seconds of computational time. As we must discard the use of a "best combination" in the Variable Neighborhood Search implementation because we do not know in advance which one it will be, the average runtime is just given for the sake of completeness.

Table A.7 in appendix A provides the objective function values for the thirty selected instances as calculated with the respective neighborhood combinations, as well as with the VNS procedure presented in the next chapter, and the best solutions used for comparison.

These are the results of the test runs so far, employing simple local search with a single neighborhood and with sequential combinations of neighborhoods. Some have proven that they can carry a local search step in a more sophisticated procedure, while others have disappointed. Nonetheless, knowing what does not work is also important and helpful when taking it all to the next level – in the case of this thesis an implementation of Variable Neighborhood Search.

6. Variable Neighborhood Search for the Multi-Level Capacitated Lotsizing Problem

6.1 Implementation Details

After testing the rather plain version of sequential runs of various neighborhoods, the natural next step was to resort to the more elaborate Variable Neighborhood Search. The approach chosen for this thesis is the basic VNS as explained in chapter 3.3.1. There are, however, some important details regarding the implementation that require further explanation. The reader might remember Figure 3.4 on page 41, which showed an algorithmic sketch of the basic VNS procedure. Figure 6.1 on the next page contains the now fleshed out, concrete procedure implemented to deal with the MLCLSP.

All neighborhoods devised and presented in the previous chapter are kept and used for the shaking step. Regardless of how well or badly they fared as neighborhoods carrying a complete local search procedure alone, they could in any case be used for the perturbation of the current solution. An additional neighborhood was included, extending the *Switch* move to ten setup variables being changed to their opposing values. The neighborhood is called *Switch 10* accordingly. In total, nine different neighborhood structures are responsible for generating random neighbors in the shaking step. As the neighborhoods are not nested, no special importance was placed on the order, resulting in the predetermined deterministic sequence as laid out in Figure 6.1.

The insights gained from the testing of the individual neighborhoods had however a strong influence on the design of the local search step of the inner loop. It is set to a procedure with solely one neighborhood, *Switch*, which proved to be a reliable and rather fast neighborhood structure. A first-improvement step function on a random sample of neighbors with varying size dependent on the size of the problem instance tackled is employed. 50%, 30%, and 10% of the setup variables are the limits for very small / small, medium, and large instances respectively. The gradual elimination of possible backorders is kept as well. In order to avoid the total run time to be taken up by the local search step, a time limit on the latter is introduced and set to 300 seconds for large instances, and 200 seconds for all other instances.

```
procedure BASIC VNS Version 1.0 for MLCLSP
INPUT
neighborhoods: N_k where k = 1(Switch), 2(Swap), 3(Double Swap),
                            4(Insert 1), 5(Column Swap), 6(Delete),
                            7(Insert 0), 8(Switch 5), 9(Switch 10)
stopping condition: max. time (300/600/1200 sec.)
                    max. iterations without improvement (50)
INITIALIZATION of all data structures
1  start time
2  generate initial solution: set all y_it to 1, solve with CPLEX
                              run local search with neighborhood Insert 0
                              - first-improvement
                              - random sample (50%/30%/10%)
                              - gradual elimination of backorders
                              - time limit (200/300 sec.)
OUTER LOOP
3  while (stopping condition = FALSE)
4     {set k := 1
INNER LOOP
5     while (k ≤ 9)
6        {a. SHAKING: generate random neighbor from N_k
                      - limit on objective function value of neighbor
                        set to 5*objective function value of incumbent
                      - attempts to generate admissible neighbor limited
7         b. LOCAL SEARCH: with neighborhood Switch
                           - first-improvement
                           - random sample (50%/30%/10%)
                           - gradual elimination of backorders
                           - time limit (200/300 sec.)
8         c. MOVE OR NOT: if local optimum better than incumbent
9                         set local optimum as new incumbent and break;
10                        else
11                         set k:=k+1 }}
12 RETURN best solution encountered during search
```

Figure 6.1 – The basic VNS procedure for the MLCLSP in pseudo-code

The total amount of time allowed for a single run of the program is set to 300 seconds for very small / small instances, to 600 seconds for medium-sized instances, and to 1200

seconds for large instances. The second stopping condition employed is the maximum number of iterations without improvement of the outer loop, which is set to 50.

Regarding the initial solution, the solution obtained through setting all setup variables to one and solving the remaining problem with CPLEX is further improved before the actual VNS procedure starts. A local search step with neighborhood *Insert 0* is executed for this purpose.

The shaking step has some limitations as well. The first limit is on the objective function value of the random neighbor, which is restricted to a value at most five times as high as the objective function value of the incumbent. Secondly, in order to avoid an endless generation of non-admissible neighbors, the number of attempts is restricted to the size of the random samples used in the local search steps. In case no admissible random neighbor can be found, the following local search step is skipped and the shaking step is continued with the next neighborhood k+1.

The procedure as depicted in Figure 6.1 was neither the first – nor the last – VNS version devised and tested. While the details and results for the successor version 2.0 will be presented shortly in chapter 6.2.3, we now want to take a quick look at how we arrived at version 1.0, i.e. what other design options there were.

6.1.1 Tuning

The first version tested – again with the representative sample of 30 instances from all problem classes – simply used neighborhood *Switch* in the local search step and led to disappointing results insofar as the results from the plain local search implementations could not be improved substantially. The supposedly more clever VNS scheme did not pay off. As it could be observed that perturbations leading to a considerable increase in the objective function value very rarely led to new best solutions through the subsequent local search step, a limit on the objective function value of the randomly generated neighbors in the shaking step was introduced in order to avoid too much time spent on local search steps with no gain. Plotting no limit at all against a limit of five times and also ten times the objective function value of the incumbent revealed that the middle course produced the best results. To summarize, putting a limit on this value seemed to be a good idea, and setting this limit to the solution quality times five of the currently best solution seemed to be a good measure.

Still, the results obtained could not be considered a breakthrough. The decision to improve the initial solution before the first shaking phase of the VNS loop with a separate local search step using the *Insert 0* neighborhood, which turned out to be a good compromise between solution quality and computational time, especially if placed before neighborhood *Switch* as shown in chapter 5, could indeed boost the performance of the algorithm, albeit only slightly and mainly from medium-sized problems onward.

Substituting neighborhood *Switch* for *Insert 0* when improving the initial solution barely affected the results, but substituting *Insert 0* for *Switch* in the local search step of the inner loop led to a dramatic decline in solution quality. The decision to keep *Switch* as the principal neighborhood for the repeating inner loop was thus straightforward and easy.

Finally, the parameters stipulating the time limits and the size of the random sample used for neighbor generation were also varied and compared. While the maximum amount of total run time allowed had some leeway and could simply be increased up to the values specified above, the size of the random sample was decreased with considerations of saving time in mind, but proved to be just right at the values of 50% for very small and small instances, 30% for medium-sized instances and 10% for large instances, as the results of chapter 5 already indicated. Table 6.1 presents a brief summary of the final parameter values used for the individual problem classes.

Table 6.1 – Parameters of the VNS procedure and the local search step for problem classes A+, B, C, E

Problem Class	***VNS Parameters***		***Local Search Parameters***	
	Max. Time	***Max. Iterations without Improvement***	***Max. Time***	***Size of Random Sample***
A+	*300 seconds*	*50*	*200 seconds*	*50% of setup variables*
B	*300 seconds*	*50*	*200 seconds*	*50% of setup variables*
C	*600 seconds*	*50*	*200 seconds*	*30% of setup variables*
E	*1200 seconds*	*50*	*300 seconds*	*10% of setup variables*

6.1.2 A Few More Words on the Test Instances

While for the tuning of the VNS procedure the same 30 instances as for tuning of the local search procedures were used, the extensive testing of the final VNS version was carried out

with four complete classes of test instances. 600 instances of class B, representing very small problems, 120 small instances of class A+, 600 medium-sized instances of class C, and 150 large instances of class E summed up to a total of 1470 test instances to assess the potential of the VNS procedure designed for the MLCLSP. As this study also presents results for different subsets of the individual classes, let us take a quick look at these subsets.

In classes A+, B and C, the product structure is either general or an assembly system with four end products in the former two classes, and six end products in the latter class. The assembly structures contain one and two end products respectively. Class E consists solely of instances with a general product structure involving 15 end products and 85 parts. The assignment of the resources to the individual items can either be cyclic or non-cyclic. The former term means that operations / items on different levels in the product structure require the same resource for processing. This is the proper time to append graphical representations of two product structure examples to enhance understanding.

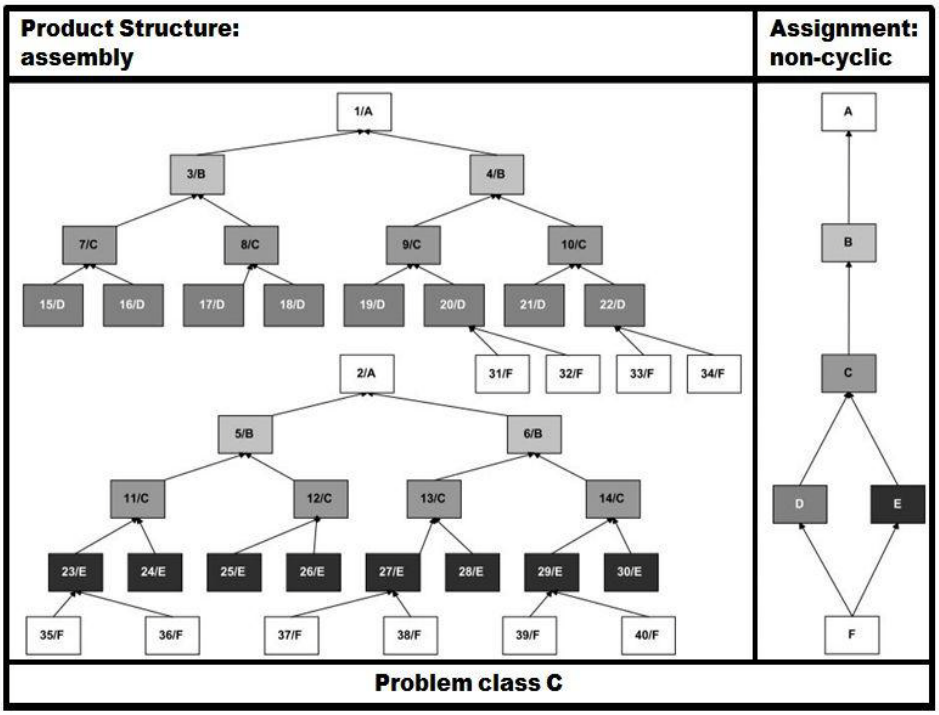

Figure 6.2 – Assembly product structure with non-cyclic assignment to resources. *Capital letters represent the resources, while items are numbered from 1 to 40. Source: Stadtler and Sürie (2000).*

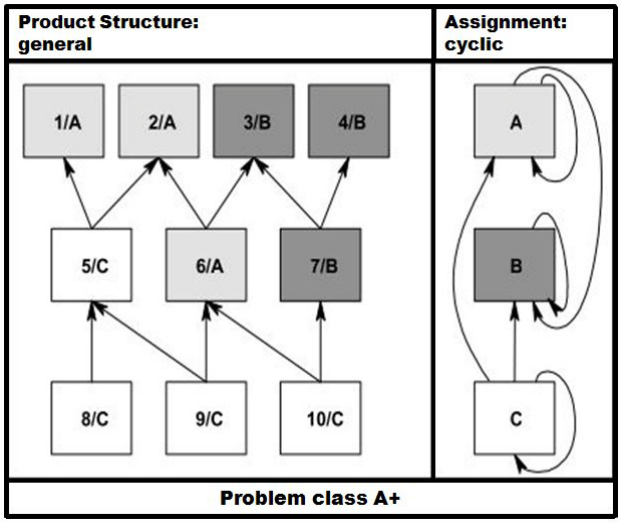

Figure 6.3 – General product structure with cyclic assignment to resources. *Capital letters represent the resources, while items are numbered from 1 to 10. Source: Stadtler and Sürie (2000).*

Other subsets of the problem classes result from different variations in the external demand series and possible seasonality, as well as from five different resource utilization profiles used. Table 6.2 provides an overview and a quick reminder of the problem class dimensions for easier reference.

Table 6.2 – Product structures and resource assignments for problem classes A+, B, C, E.
N, T and M stand for the number of items, periods and resources respectively.

Problem Class	*N*	*T*	*M*	*Setup Times*	*Product Structure*	*Resource Assignment*
A+	*10*	*24*	*3*	*No*	*General / Assembly*	*Cyclic*
B	*10*	*4*	*3*	*Yes*	*General / Assembly*	*Cyclic / Non-cyclic*
C	*40*	*16*	*6*	*No*	*General / Assembly*	*Cyclic / Non-cyclic*
E	*100*	*16*	*10*	*No*	*General*	*Cyclic / Non-cyclic*

6.2 Computational Results

The final version of the VNS procedure was tested on a personal computer with a Pentium D 3.2 GHz processor with 4 GB RAM and SUSE Linux 10.1. As with the simple local search test runs, each instance was solved exactly once with each respective VNS version.

6.2.1 Individual Problem Classes

Problem class B. We start with the results for the smallest of the test instances, presented in Table 6.3. For this class, optimal solutions calculated with CPLEX are known and used for evaluation of the VNS. The average gap to the optimum amounts to 0,463%, a value sufficiently low that is due to more than three quarters of the 600 instances being solved to optimality with the VNS procedure. Only 2,50% of the instances were solved unsatisfactorily, i.e. with a gap to the optimum larger than five percent, with a maximum gap of about twelve percent. Taking a look at other *heuristic* methods for this problem class, Almeder (2010) and Pitakaso et al. (2006) report mean average percentage deviations to the optimum of less than 0,1%, while the algorithm of Tempelmeier and Derstroff (1996) produces a mean deviation of 1,30%. This reveals that the VNS procedure lies in-between and yields merely acceptable results.

Table 6.3 – Results of VNS version 1.0 for problem class B

Problem Class	**B**	
Average gap to CPLEX optimum	***0,463%***	
Median gap to CPLEX optimum	***0,00%***	
Maximum gap to CPLEX optimum	***11,71%***	
	Number of instances	***Percentage of instances***
gap = 0%	***463 of 600***	***77,17%***
0% < gap < 5%	*122 of 600*	*20,33%*
5% <= gap < 20%	*15 of 600*	*2,50%*
gap >= 20%	*0 of 600*	*0%*
Average objective function value		
VNS	***10.998***	
CPLEX optimum	***10.922***	
Average computational time	***65 seconds***	
Average number of iterations	***58,41***	
Average number of iterations without improvement	***55,67***	
In percent	***95,30%***	

The VNS procedure took about a minute of computational time, in which on average just under 60 iterations of the outer loop were carried out. Interestingly, 95,30% of these iterations did not bring any improvement on the objective function value. In light of the moderate gaps, this is probably due to very good initial solutions – remember that a local

search step with neighborhood *Insert 0* is placed before the start of the actual VNS scheme. Also remember that as a stopping condition 50 consecutive iterations without improvement is implemented; reducing this parameter to a lower value could therefore also reduce the average run time of the procedure in this problem class.

The results for very small instances are not overwhelmingly good, but might nonetheless be considered a good start in the right direction. Further adjustments and a little tweaking of the VNS procedure might very well enable it to catch up with other heuristics.

Problem class A+. Table 6.4 on the next page shows the results for the 120 small instances. For this class, the solutions obtained with the VNS are compared to the best available solutions from Stadtler (2003) and Almeder (2010). Not a single one of the best known solutions could be topped; the solutions to the majority of problems were however not far off with a gap of under five percent. Thirty percent of the instances could not be solved convincingly, leading to an overall gap of 4,37% to the best known heuristic solutions. Nonetheless, the maximum gap of about thirteen percent is an encouraging bright spot of the results.

A comparison with a lower bound on the solutions, as appended to the test instances themselves and thus obtained, is provided further down the table. An average gap of about 25% is rather disappointing, with none of the results showing a gap less than five percent and five of the 120 instances being considerably off with a gap of over 50%. Of course, the best known solution also shows a relatively large average gap of twenty percent to the lower bound, which puts the performance of the VNS in a more favorable light again.

The average run time in this class amounts to 322 seconds[22], in which a quite small number of on average twelve iterations took place. A quarter of the iterations were on top of that in vain.

Although the A+ problems are classified here as small, five minutes for solving them competitively might simply be an insufficient amount of time, i.e. a parameter adjustment increasing the 300 seconds of time allowed should definitely boost the performance of the VNS procedure. This direction has been explored; subchapter 6.2.3 will inform the reader how much the gaps could be reduced with extra time.

[22] The allowed run times in all problem classes can be exceeded because the stopping conditions are checked only when a new outer loop starts.

Table 6.4 – Results of VNS version 1.0 for problem class A+. *Best solution refers to the best available solution from either Stadtler (2003) or Almeder (2010).*

Problem Class	**A+**	
Average gap to best solution	***4,37%***	
Median gap to best solution	***4,21%***	
Maximum gap to best solution	***13,33%***	
	Number of instances	***Percentage of instances***
gap < 0%	*0 of 120*	*0%*
0% <= gap < 5%	***84 of 120***	***70,00%***
5% <= gap < 20%	*36 of 120*	*30,00%*
gap >= 20%	*0 of 120*	*0%*
Average gap to lower bound	***25,21%***	
Median gap to lower bound	***24,53%***	
Maximum gap to lower bound	***64,19%***	
Average gap of best solution to LB	*19,89%*	
	Number of instances	***Percentage of instances***
0% <= gap < 5%	*0 of 120*	*0%*
5% <= gap < 20%	*44 of 120*	*36,67%*
20% <= gap < 50%	***71 of 120***	***59,17%***
gap >= 50%	*5 of 120*	*4,17%*
Average objective function value		
VNS	***152.421***	
Best solution	***146.402***	
Lower bound	***123.252***	
Average computational time	***322 seconds***	
Average number of iterations	***12,39***	
Average number of iterations without improvement	***3,12***	
In percent	***25,15%***	

Problem class C. This class of medium-sized problems can be considered the high point of the performance of the VNS procedure, in that it finally produced some results one can live with. Reaching an encouraging overall gap of 3,24% to the best known solutions, for this class either from Tempelmeier and Derstroff (1996), Pitakaso et al. (2006) or Almeder (2010), nearly 80% of the 600 instances were solved within acceptable limits, that is within a five percent gap. Three and a half percent of the problems now have a new best solution. Slightly more than twenty percent of the results are not so pleasant and show gaps falling between 5% and 24,53%. Regarding the comparison with the lower bound, about half of the instances fall in the five-to-twenty percent interval, leading to an average gap of 18%. Only thirteen percent of the results fall in an acceptable range.

Clocking in at 685 seconds on average, the average number of iterations still goes down the larger the problem instances become. About six iterations – where only about four are successful – in a ten minute program run can either indicate a large number of local search steps carried out in vain from poor random neighbors, requiring an adjustment of the shaking step, or it can mean that the local search step itself needs an adjustment with respect to the possibly excessive computational time it requires. In light of the already high solution quality reached in comparison to other heuristic solutions for this problem class, this should then be taken as an encouraging challenge rather than a knock-down.

Table 6.5 – Results of VNS version 1.0 for problem class C. *Best solution refers to the best available solution from either Tempelmeier and Derstroff (1996), Pitakaso et al. (2006) or Almeder (2010).*

Problem Class	**C**	
Average gap to best solution	***3,24%***	
Median gap to best solution	***3,19%***	
Maximum gap to best solution	***24,53%***	
	Number of instances	***Percentage of instances***
gap < 0%	*21 of 600*	*3,50%*
gap = 0%	*58 of 600*	*9,67%*
0% < gap < 5%	***394 of 600***	***65,67%***
5% <= gap < 20%	*126 of 600*	*21,00%*
20% <= gap < 50%	*1 of 600*	*0,17%*
gap >= 50%	*0 of 600*	*0%*
Average gap to lower bound	***18,01%***	
Median gap to lower bound	***16,37%***	
Maximum gap to lower bound	***78,51%***	
Average gap of best solution to LB	*14,33%*	
	Number of instances	***Percentage of instances***
gap = 0%	*24 of 600*	*4,00%*
0% < gap < 5%	*54 of 600*	*9,00%*
5% <= gap < 20%	***296 of 600***	***49,33%***
20% <= gap < 50%	*218 of 600*	*36,33%*
gap >= 50%	*8 of 600*	*1,33%*
Average objective function value		
VNS	***209.457***	
Best solution	***201.619***	
Lower bound	***174.125***	
Average computational time	***685 seconds***	
Average number of iterations	***6,29***	
Average number of iterations without improvement	***1,86***	
In percent	***29,50%***	

Problem class E. The VNS solutions are compared to the best solutions reported by either Tempelmeier and Derstroff (1996), Stadtler (2003), Pitakaso et al. (2006), or Almeder (2010). The 150 large instances had to be solved in 20 minutes each, and in retrospect, the time allowed was probably set too low for the VNS to be competitive. The highest average gap of all problem classes to the best known solutions with an average value of about eight percent is the result. A majority of seventy percent of the instances solved fall in the not-so-good gap interval of five-to-twenty percent. Not quite thirty percent, or 41 of the 150 instances, were solved to satisfaction. The fact that the solution to one instance could be improved is no real consolation; the fact that the solution to only four instances was way off and that the maximum gap of 25,18% is not much higher than in problem class C just might be.

Table 6.6 – Results of VNS version 1.0 for problem class E. *Best solution refers to the best available solution from either Tempelmeier and Derstroff (1996), Stadtler (2003), Pitakaso et al. (2006) or Almeder (2010).*

Problem Class	**E**	
Average gap to best solution	***7,95%***	
Median gap to best solution	***8,12%***	
Maximum gap to best solution	***25,18%***	
	Number of instances	***Percentage of instances***
gap < 0%	*1 of 150*	*0,67%*
0% <= gap < 5%	*40 of 150*	*26,67%*
5% <= gap < 20%	***105 of 150***	***70,00%***
20% <= gap < 50%	*4 of 150*	*2,67%*
gap >= 50%	*0 of 150*	*0%*
Average gap to lower bound	***19,58%***	
Median gap to lower bound	***17,94%***	
Maximum gap to lower bound	***60,45%***	
Average gap of best solution to LB	*10,63%*	
	Number of instances	***Percentage of instances***
0% <= gap < 5%	*21 of 150*	*14,00%*
5% <= gap < 20%	*62 of 150*	*41,33%*
20% <= gap < 50%	***64 of 150***	***42,67%***
gap >= 50%	*3 of 150*	*2,00%*
Average objective function value		
VNS	***2.301.198***	
Best solution	***2.090.843***	
Lower bound	***1.838.879***	
Average computational time	***1491 seconds***	
Average number of iterations	***4,60***	
Average number of iterations without improvement	***1,23***	
In percent	***26,81%***	

The lower bound gaps show a familiar pattern with values to be expected after having seen the results of the other problem classes; the difference to the average gap of the best solution to the lower bound is obviously more pronounced as in the other classes.

Although the average number of iterations is at a minimum, reaching not even five, the percentage of iterations without improvement stays roughly the same. Nonetheless, this value is something one should strive to improve for all problem classes.

The results for the large instances are far from being a complete disaster. Unfortunately, they are however also far from being something to write home about.

Roughly the same could of course be said for all problem classes, to summarize the results so far. The VNS procedure designed for this thesis cannot beat or even reproduce the results of the compared heuristic algorithms already employed to solve the test problems considered here. While the results for medium-sized instances are satisfactory, the results for the very small and small instances are merely tolerable, and the results for the large problem class E are rather disappointing.

To emphasize the positive, regardless of problem classes and instance sizes, the VNS procedure is on average only lagging behind four percent to the best known solutions. The percentages of instances with a gap higher than 20% to the best known solutions are also sufficiently low, so that one cannot deny a certain robustness of the procedure. Given any multi-level capacitated lotsizing instance of the Tempelmeier and Derstroff / Stadtler problem libraries, the algorithm presented here has a high chance of delivering a *good* solution – maybe not excellent, but neither terribly bad.

6.2.2 Subsets of the Problem Classes

A further indication of the robustness of the VNS procedure can be found when looking at the results for the various subsets present in all problem classes. The average gaps to the best known solutions and the percentage of instances where the VNS yielded a best solution itself, i.e. instances where the gap is less than or equal to zero, are compared in Table 6.7 on the next page.

Instances with an assembly structure show a slightly lower gap than the general product structure problems. The VNS procedure seems to handle the less complex assembly structures a little better, which then again lacks any real element of surprise.

The results seem to indicate a slight preference of the VNS for problems with cyclic assignment of the items to the resources. Consider however the results for problem class B: although the average gap to the best solution is lower for the cyclic case, more instances could be solved to optimality in case of non-cyclic resource assignment.

Table 6.7 – Results of VNS version 1.0 for subsets of the problem classes

	Average gap to best solution				***Percentage of instances of subset where gap to best solution <= 0***			
Problem Class	**A+**	**B**	**C**	**E**	**A+**	**B**	**C**	**E**
Product Structure								
General	*4,93%*	*0,75%*	***3,19%***	*7,95%*	*0%*	*73,00%*	***14,33%***	*0,67%*
Assembly	***3,82%***	***0,18%***	*3,29%*	***---***	*0%*	***81,33%***	*12,00%*	***---***
Resource Assignment								
Non-cyclic	***---***	*0,57%*	***3,16%***	*9,33%*	***---***	***80,67%***	***13,33%***	*0%*
Cyclic	*4,37%*	***0,35%***	*3,32%*	***6,58%***	*0%*	*73,67%*	*13,00%*	***1,33%***
Resource Utilization								
High	*4,70%*	*0,65%*	***2,32%***	*8,50%*	*0%*	*77,50%*	***18,33%***	***3,00%***
Medium	*4,37%*	*0,40%*	*3,36%*	*7,24%*	*0%*	*75,00%*	*15,00%*	*0%*
Low	*4,16%*	***0,14%***	*3,82%*	*7,28%*	*0%*	***85,00%***	*15,00%*	*0%*
Mix 1	***3,30%***	*0,37%*	*3,06%*	***6,43%***	*0%*	*78,33%*	*11,67%*	*0%*
Mix 2	*5,33%*	*0,76%*	*3,63%*	*10,32%*	*0%*	*70,00%*	*5,83%*	*0%*
Demand Variation								
Slight	***4,36%***	***0,42%***	*3,45%*	*7,98%*	*0%*	*77,50%*	***19,00%***	*0%*
Medium	***---***	*0,47%*	*3,30%*	*8,04%*	***---***	*73,50%*	*10,00%*	*0%*
Strong	*4,38%*	*0,50%*	***2,96%***	***7,85%***	*0%*	***80,50%***	*10,50%*	***2,00%***
Seasonality of Demand								
Zero	***3,93%***	***---***	***---***	***---***	*0%*	***---***	***---***	***---***
Slight	*5,15%*	***---***	***---***	***---***	*0%*	***---***	***---***	***---***
Strong	*4,04%*	***---***	***---***	***---***	*0%*	***---***	***---***	***---***

The resource utilization does not seem to have a huge effect on the VNS procedure. The clearest bias that can be detected is for utilization profile “Mix 2” – the resources used late in the production process have a low (50%) utilization, which gradually increases to high (90%) for resources needed earlier – to be solved least efficiently. The reversed profile “Mix 1” with increasing degree of utilization yields better results, although class C and E show the highest percentage of best solutions by the VNS procedure in case of highly restrained resources.

The average gaps are distributed fairly even when it comes to variation in demand, although we face again contradicting results, this time in problem class C. Varying seasonality of the external demand is only present in class A+, where the pattern with no seasonality at all fared best.

There do exist some inclinations to certain subsets which the VNS seems to deal with more efficiently than others. The differences are however not very pronounced, in that no subset could be singled out as a complete failure, strengthening the impression of a robust solution method.

6.2.3 Adjusted VNS Versions

A second, moderately adjusted version was devised and tested on problem classes A+ and C, after some doubts about the quality of the originally tested procedure emerged. Version 2.0 differs from version 1.0 in that it lifts the time limit on the local search step and scans the *Switch* neighborhood in an exhaustive, yet randomized manner. This means that the whole neighborhood is considered instead of just a sample, but the order in which the neighbors are generated is still at random instead of systematically starting with item 1 in period 1 every time. The adjusted shaking step leaves out three neighborhoods; *Switch* because it is used in the local search step anyway, and *Insert 1* and *Delete* because these two came up with the worst results in the simple local search tests. Furthermore, the limit on the objective function value of the perturbed solution is lowered. Table 6.8 offers a quick overview of the changes.

Table 6.8 – Differences in the local search and shaking steps between VNS version 1.0 and version 2.0

	Version 1.0	*Version 2.0*
Local search step		
Mode	*random sample*	*randomized exhaustive*
Time Limit	*200 seconds*	*none*
Shaking Step		
Neighborhoods	*all*	*all except Switch, Insert 1, Delete*
Limit on objective function value	*5 times that of incumbent*	*3 times that of incumbent*

The results could not be improved. A comparison is nonetheless interesting and therefore provided in Table 6.9. The average gaps to the best available solutions and the lower bounds are slightly higher in version 2.0. The maximum gaps to the best solution, a strong point of version 1.0, are significantly higher in the new version, even if the percentages of instances with such extremely bad results stay in the range of the old version. Note that version 2.0 produced slightly more best solutions in problem class C than version 1.0, but other than that, the percentages falling in the specified gap intervals tend to be shifted farther down, i.e. in the wrong direction. The increased computational time of version 2.0 is due to the exhaustive mode in the local search step.

What version 2.0 accomplished is a reduction of iterations without improvement, which we complained about earlier. If this reduction comes however at the price of a lower overall solution quality with a more erratic behavior, we can either conclude that seeing the words "without improvement" somehow make an alarm go off in spite of this number not being so important after all, or we can try to figure out exactly which one of the changes brought about the desired decrease, which is however beyond our scope for now.

The differences between the two versions proved to be statistically significant indeed, especially for problem class C. The statistically inclined reader will be happy to find the results of the Wilcoxon signed-rank test in appendix A.

Rather than being disappointed that the new version did not bring significant improvements on the solution quality, the author chooses to regard these results as a confirmation that the old version is not so bad after all.

Table 6.9 – Comparison of the results of VNS version 1.0 and version 2.0 for problem classes A+ and C

	Problem Class A+		Problem Class C	
	Version 1.0	***Version 2.0***	***Version 1.0***	***Version 2.0***
Average gap to best solution	***4,37%***	*4,82%*	***3,24%***	*3,47%*
Median gap to best solution	*4,21%*	***4,14%***	***3,19%***	*3,30%*
Maximum gap to best solution	***13,33%***	*43,50%*	***24,53%***	*61,02%*
Average gap to lower bound	***25,21%***	*25,78%*	***18,01%***	*18,27%*
Median gap to lower bound	***24,53%***	*25,12%*	*16,37%*	***16,21%***
Maximum gap to lower bound	***64,19%***	*85,42%*	***78,51%***	*85,15%*
Percentage of instances that…				
gap < 0,00	*0%*	*0%*	*3,50%*	*4,50%*
gap = 0,00	*0%*	*0%*	*9,67%*	*10,00%*
0,00 < gap < 0,05	*70,00%*	*64,17%*	*65,67%*	*59,67%*
0,05 <= gap < 0,2	*30,00%*	*35,00%*	*21,00%*	*25,50%*
0,2 <= gap < 0,5	*0%*	*0,83%*	*0,17%*	*0,17%*
gap >= 0,5	*0%*	*0%*	*0%*	*0,17%*
Average objective function value	***152.421***	*153.204*	***209.457***	*210.650*
Average computational time	***322 seconds***	*329 seconds*	***685 seconds***	*801 seconds*
Average number of iterations	***12,39***	*9,23*	***6,29***	*3,03*
Average number of iterations without improvement	*3,12*	***1,87***	*1,86*	***0,48***
In percent	*25,15%*	***20,23%***	*29,50%*	***15,76%***

A few pages back, we expressed some concerns that the total run time granted to the VNS procedure might be set too low for it to be competitive. Problem class A+ and hundred randomly selected instances of problem class C were therefore submitted to a further test with considerably longer computational times. The five minutes for small instances were extended to half an hour, while medium-sized instances now had a full hour to be solved. Obviously, the gaps were expected to decrease. Table 6.10 on the next page shows how much they indeed did.

Alas, despite the generously credited time, the average gaps to the best known solutions in both problem classes are still positive. The VNS procedure uses however the extra time productively and reaches quite competitive results in terms of solution quality. Particularly class A+ offers now convincing results. The results for class C were not so disappointing in the first place; with extended time the number of instances where the VNS yielded a best solution could be increased from 9 to 21 of 100. A small flaw is the maximum gap to the best known solution, at 15,20% higher than the original value.

Table 6.10 – Results of VNS version 1.0 with extended run-time for problem classes A+ and C

	Problem Class A+ (120 instances)		Problem Class C (100 instances)	
Version 1.0	***5 minutes***	***30 minutes***	***10 minutes***	***60 minutes***
Average gap to best solution	*4,37%*	***1,99%***	*3,41%*	***1,85%***
Median gap to best solution	*4,21%*	***1,87%***	*3,27%*	***1,62%***
Maximum gap to best solution	*13,33%*	***6,37%***	***14,69%***	*15,20%*
Average gap to lower bound	*25,21%*	***22,33%***	*18,90%*	***17,07%***
Median gap to lower bound	*24,53%*	***22,37%***	*18,07%*	***15,62%***
Maximum gap to lower bound	*64,19%*	***56,31%***	*59,33%*	***56,97%***
Percentage of instances that…				
gap < 0,00	*0%*	***7,50%***	*1,00%*	***13,00%***
gap = 0,00	*0%*	*0%*	*8,00%*	*8,00%*
0,00 < gap < 0,05	*70,00%*	*90,00%*	*70,00%*	*74,00%*
0,05 <= gap < 0,2	*30,00%*	*2,50%*	*21,00%*	*5,00%*
gap >= 0,2	*0%*	*0%*	*0%*	*0%*
Average objective function value	*152.421*	***149.048***	*225.939*	***221.347***
Average computational time	*322 seconds*	*1823 seconds*	*689 seconds*	*3692 seconds*
Average number of iterations	*12,39*	*51,42*	*6,29*	*28,09*
Average number of iterations without improvement	*3,12*	*32,28*	*1,59*	*15,39*
In percent	***25,15%***	*62,8%*	***25,28%***	*54,8%*

Unfortunately, the author again cannot overlook the average number of iterations without improvement. The values are unexpectedly high, more than twice as high as originally, with a single iteration in both classes now more often than not bringing no improvement.

In light of the small gaps however, the results are taken as a demonstration of the potential of Variable Neighborhood Search for the MLCLSP. The procedure is robust and able to deliver high quality solutions. Now, if we could only overlook somehow the time it needs to do so…

7. Conclusion

Local search and metaheuristics based on or incorporating it are powerful weapons to tackle hard optimization problems where exact methods fail. The Multi-Level Capacitated Lotsizing Problem falls into this category and approximate methods of all kinds dominate the area. Finding efficient ways to solve this specific problem is not only of theoretic interest, but also motivated by the shortcomings of current practice and the consequent opportunity for handling a central decision in production planning much better.

The study presented in this thesis explored different options for designing efficient local search procedures to tackle the MLCLSP. Eight neighborhoods based on manipulating the binary matrix of the setup variables were devised and evaluated. The most commonly used neighborhood throughout the literature, changing a single value of the matrix to its diametrical value, seems to be the most commonly used neighborhood for a reason – the simple *Switch* move is a reliable and rather fast mechanism for exploring the search space. As an alternative, the *Insert 1* neighborhood and a move switching five setup variables at once proved to be able to carry a local search procedure more or less successfully. The other neighborhoods failed to leave much of a positive impression.

When pondering other fundamental design issues, this study comes out in favor of a first-improvement step function and a restriction on the number of neighbors to be examined at each iteration – a random sample with varying size depending on the size of the problem instance to be tackled. Both options are preferable due to faster computational times without solution quality suffering much. Permitting infeasible solutions during the search, in case of the MLCLSP by relaxing the inventory balance equations with the possibility of backordering items, and gradually eliminating infeasibility with an adaptive penalty can also be recommended.

The implementation of the simple local search procedures led to surprisingly good results with respect to solution quality when compared with best known solutions to the test instances used. This raised great expectations that the more sophisticated Variable Neighborhood Search procedure would be able to keep up with or even partially outperform the other heuristic methods taken as reference point. Alas, the results of the VNS procedure are mixed. For neither of the four problem classes that it took on is the achieved solution quality extraordinarily bad. Neither is it extraordinarily good evidently.

The average gaps to the best known solutions range from an acceptable 0,46% for the very small instances to a rather disappointing 8% for the large instances. The VNS procedure did its best job when faced with medium-sized problems. Even if it certainly could not outperform its competitors, the impression of a robust method delivering good, sometimes even very good solutions on a more or less regular basis prevails.

Although Variable Neighborhood Search is a metaheuristic with very few parameters, this does not rule out numerous possibilities for tuning and development. This does especially apply to the – of course actually VNS-independent – local search step, which could in any case use a further reduction in computational time. Employing mechanisms to obtain samples of promising solutions instead of plain random samples, in the spirit of candidate lists, might also be well worth a try. The shaking step is not immune to enhancements as well; the attempts to select promising random neighbors by limiting objective function values go in this direction. A next step would be for example to reconsider the order of the neighborhoods used for perturbation. In this respect, this work then cannot be considered finished. This thesis thus ends not with the observation that the Variable Neighborhood Search approach can be discarded and a notion of failure. It ends with a feeling of discovering a potential future competitor for methods trying to solve the Multi-Level Capacitated Lotsizing Problem.

References

Aarts, E.H.L., Lenstra, J.K. (1997) *Local Search in Combinatorial Optimization*. John Wiley and Sons, Chichester.

Almada-Lobo, B., James, R.J.W. (2008) Neighbourhood search meta-heuristics for capacitated lot-sizing with sequence-dependent setups. International Journal of Production Research, iFirst, 1(18)

Almada-Lobo, B., Oliveira, J.F., Carravilla, M.A. (2008) Production planning and scheduling in the glass container industry: A VNS approach. International Journal of Production Economics 114, 363-375

Almeder, C. (2010) A hybrid optimization approach for multi-level capacitated lot-sizing problems. European Journal of Operational Research 200(2), 599-606

Aytug, H., Khouja, M., Vergara, F.E. (2003) Use of genetic algorithms to solve production and operations management problems: a review. International Journal of Production Research 41(17), 3955-4009

Bahl, H.C., Ritzman, L.P. (1984) An Integrated Model for Master Scheduling, Lot Sizing and Capacity Requirements Planning. Journal of the Operational Research Society 35(5), 389-399

Bahl, H.C., Ritzman, L.P., Gupta, J.N.D. (1987) Determining Lot Sizes and Resource Requirements: A Review. Operations Research 35(3), 329-345

Barbarosoglu, G., Özdamar, L. (2000) Analysis of solution space-dependent performance of simulated annealing: the case of the multi-level capacitated lot sizing problem. Computers & Operations Research 27, 895-903

Battiti, R., Brunato, M., Mascia, F. (2007) Reactive Search and Intelligent Optimization. Technical Report DIT-07-049, Version 1.02, Università di Trento.

Baum, E.B. (1986) Iterated descent: A better algorithm for local search in combinatorial optimization problems. Technical Report, Caltech, Pasadena.

Berretta, R., França, P.M., Armentano, V.A. (2005) Metaheuristic Approaches for the Multilevel Resource-Constrained Lot-Sizing Problem with Setup and Lead Times. Asia-Pacific Journal of Operational Research 22(2), 261-286

Billington, P.J., McClain, J.O., Thomas, L.J. (1983) Mathematical Programming Approaches to Capacity-Constrained MRP Systems: Review, Formulation and Problem Reduction. Management Science 29(10), 1126-1141

Billington, P.J., McClain, J.O., Thomas, L.J. (1986) Heuristics for Multilevel Lot-Sizing with a Bottleneck. Management Science 32(8), 989-1006

Bitran, G.R., Yanasse, H.H. (1982) Computational Complexity of the Capacitated Lot Size Problem. Management Science 28(10), 1174-1186

Blum, C., Roli, A. (2003) Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Computing Surveys 35(3), 268-308

Brüggemann, W., Jahnke, H. (1994) DLSP for two-stage multi-item batch production. International Journal of Production Research 32(4), 755-768

Brüggemann, W., Jahnke, H. (2000) The discrete lot-sizing and scheduling problem: Complexity and modifications for batch availability. European Journal of Operational Research 124, 511-528

Buschkühl, L., Sahling, F., Helber, S., Tempelmeier, H. (2010) Dynamic capacitated lot-sizing problems: a classification and review of solution approaches. OR Spectrum 32(2), 231-261

Caserta, M., Ramirez, A., Voß, S. (2010) A Math-Heuristic for the Multi-Level Capacitated Lot Sizing Problem with Carryover. In: *Applications of Evolutionary Computation*. C. Di Chio et al. (Eds.), Lecture Notes in Computer Science 6025, Springer, Berlin.

Chen, H., Chu, C. (2003) A Lagrangian Relaxation Approach for Supply Chain Planning with Order/Setup Costs And Capacity Constraints. Journal of Systems Science and Systems Engineering 12(1), 98-110

Croes, G.A. (1958) A Method for Solving Traveling-Salesman Problems. Operations Research 6(6), 791-812

de Araujo, S.A., Arenales, M.N., Clark, A.R. (2008) Lot sizing and furnace scheduling in small foundries. Computers & Operations Research 35, 916-932

Dellaert, N., Jeunet, J. (2000) Solving large unconstrained multilevel lot-sizing problems using a hybrid genetic algorithm. International Journal of Production Research 38(5), 1083-1099

Dellaert, N., Jeunet, J., Jonard, N. (2000) A genetic algorithm to solve the general multi-level lot-sizing problem with time-varying costs. International Journal of Production Economics 68, 241-257

DiGaspero, L. (2003) Local Search Techniques for Scheduling Problems: Algorithms and Software Tools. Dissertation, University of Udine.

Dorigo, M. (1992) Optimization, Learning, and Natural Algorithms. Dissertation, Politecnico di Milano.

Drexl, A., Haase, K. (1995) Proportional lotsizing and scheduling. International Journal of Production Economics 40, 73-87

Drexl, A., Kimms, A. (1997) Lot sizing and scheduling – Survey and extensions. European Journal of Operational Research 99, 221-235

Elmaghraby, S.E. (1978) The Economic Lot Scheduling Problem (ELSP): Review and Extensions. Management Science 24(6), 587-598

Fakhrzad, M.B., Khademi Zare, H. (2009) Combination of genetic algorithm with Lagrange multipliers for lot-size determination in multi-stage production scheduling problems. Expert Systems with Applications 36(6), 10180-10187

Feo, T.A., Resende, M.G.C. (1989) A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters 8(2), 67-71

Feo, T.A., Resende, M.G.C. (1995) Greedy Randomized Adaptive Search Procedures. Journal of Global Optimization 6, 109-134

Fink, A., Rothlauf, F. (2006) Heuristische Optimierungsverfahren in der Wirtschaftsinformatik. Working Papers in Information Systems 1, University of Mannheim.

Fleischmann, B., Meyr, H. (1997) The general lotsizing and scheduling problem. OR Spektrum 19, 11-21

Florian, M., Lenstra, J.K., Rinnooy Kan, A.H.G. (1980) Deterministic Production Planning: Algorithms and Complexity. Management Science 26(7), 669-679

Foulds, L.R. (1983) The Heuristic Problem-Solving Approach. Journal of the Operational Research Society 34(10), 927-934

Frank, J. D. (1997) Local Search for NP-Hard Problems. Dissertation, University of California.

Gaafar, L.K. (2006) Applying genetic algorithms to dynamic lot sizing with batch ordering. Computers & Industrial Engineering 51, 433-444

Gaafar, L.K., Nassef, A.O., Aly, A.I. (2009) Fixed-quantity dynamic lot sizing using simulated annealing. International Journal of Advanced Manufacturing Technology 41(1-2), 122-131

Gendreau, M., Potvin, J.-Y. (2005a) Metaheuristics in Combinatorial Optimization. Annals of Operations Research 140(1), 189-213

Gendreau, M., Potvin, J.-Y. (2005b) Tabu Search. In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Edmund K. Burke and Graham Kendall (Eds.), Springer, New York.

Glover, F. (1986) Future Paths for Integer Programming and Links to Artificial Intelligence. Computers and Operations Research 13(5), 533-549

Glover, F. (1989) Tabu Search – Part I. ORSA Journal on Computing 1(3), 190-206

Glover, F., Laguna, M. (1997) *Tabu Search*. Kluwer Academic Publishers, Norwell.

Gopalakrishnan, M., Ding, K., Bourjolly, J.-M., Mohan, S. (2001) A Tabu-Search Heuristic for the Capacitated Lot-Sizing Problem with Set-up Carryover. Management Science 47(6), 851-863

Goren, H.G., Tunali, S., Jans, R. (2008) A review of applications of genetic algorithms in lot sizing. Journal of Intelligent Manufacturing 21(4), 575-590

Groff, G.K. (1979) A lot sizing rule for time-phased component demand. Production and Inventory Management 20(1), 47-53

Günther, H.-O., Tempelmeier, H. (2007) *Produktion und Logistik*. 7. Auflage, Springer, Berlin.

Haase, K. (1996) Capacitated lot-sizing with sequence dependent setup costs. OR Spektrum 18(1), 51-59

Hansen, P., Mladenović, N. (2001) Variable neighborhood search: Principles and applications. European Journal of Operational Research 130, 449-467

Hansen, P., Mladenović, N. (2003a) A Tutorial on Variable Neighborhood Search. Les Cahiers du GERAD, G-2003-46

Hansen, P., Mladenović, N. (2003b) Variable Neighborhood Search. In: *Handbook of Metaheuristics*. Fred Glover and Gary A. Kochenberger (Eds.), Kluwer Academic Publishers, Norwell.

Hansen, P., Mladenović, N., Moreno Pérez, J.A. (2003) Búsqueda de Entorno Variable. Revista Iberoamericana de Inteligencia Artificial 19, 77-92

Hansen, P., Mladenović, N., Moreno Pérez, J.A. (2008) Variable Neighborhood Search: Methods and Applications. Les Cahiers du GERAD, G-2008-39

Harris, F.W. (1913) How Many Parts to Make at Once. Factory, The Magazine of Management 10(2), 135-136/152, reprinted in Operations Research 38(6), 947-950

Helber, S. (1995) Lot sizing in capacitated production planning and control systems. OR Spektrum 17, 5-18

Hertz, A., Widmer, M. (2003) Guidelines for the use of meta-heuristics in combinatorial optimization. European Journal of Operational Research 151, 247-252

Hindi, K.S. (1995) Solving the single-item, capacitated dynamic lot-sizing problem with startup and reservation costs by tabu search. Computers & Industrial Engineering 28(4), 701-707

Hindi, K.S. (1996) Solving the CLSP by a Tabu Search Heuristic. Journal of the Operational Research Society 47(1), 151-161

Hindi, K.S., Fleszar, K., Charalambous, C. (2003) An effective heuristic for the CLSP with set-up times. Journal of the Operational Research Society 54, 490-498

Holland, J.H. (1975) *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor.

Homberger, J., Gehring, H. (2009) An Ant Colony Optimization Approach for the Multi-Level Unconstrained Lot-Sizing Problem. Proceedings of the 42nd Hawaii International Conference on System Sciences

Hung, Y.-F., Chien, K.-L. (2000) A multi-class multi-level capacitated lot sizing model. Journal of the Operational Research Society 51(11), 1309-1318

Hung, Y.-F., Shih, C.-C., Chen, C.-P. (1999) Evolutionary algorithms for production planning with setup decisions. Journal of the Operational Research Society 50(8), 857-866

Hung, Y.-F., Chen, C.-P., Shih, C.-C., Hung, M.-H. (2003) Using tabu search with ranking candidate list to solve production planning problems with setups. Computers & Industrial Engineering 45, 615-634

Jans, R. (2002) Capacitated Lot Sizing Problems: New Applications, Formulations and Algorithms. Dissertation, Katholieke Universiteit Leuven

Jans, R., Degraeve Z. (2005) Modeling Industrial Lot Sizing Problems: A Review. ERIM Report Series Research in Management, ERS-2005-049-LIS

Jans, R., Degraeve, Z. (2007) Meta-heuristics for dynamic lot sizing: A review and comparison of solution approaches. European Journal of Operational Research 177, 1855-1875

Jeunet, J., Jonard, N. (2005) Single-point stochastic search algorithms for the multi-level lot-sizing problem. Computers & Operations Research 32, 985-1006

Johnson, C.G. (2008) A design framework for metaheuristics. Artificial Intelligence Review 29(2), 163-178

Jordan, C., Koppelmann, J. (1998) Multi-Level Lotsizing and Scheduling by Batch Sequencing. Journal of the Operational Research Society 49(11), 1212-1218

Karimi, B., Fatemi Ghomi, S.M.T., Wilson, J.M. (2003) The capacitated lot sizing problem: a review of models and algorithms. Omega 31, 365-378

Karimi, B., Fatemi Ghomi, S.M.T., Wilson, J.M. (2006) A tabu search heuristic for solving the CLSP with backlogging and set-up carry-over. Journal of the Operational Research Society 57(2), 140-147

Karmarkar, U.S., Schrage, L. (1985) The Deterministic Dynamic Product Cycling Problem. Operations Research 33(2), 326-345

Katok, E., Lewis, H.S., Harrison, T.P. (1998) Lot Sizing in General Assembly Systems with Setup Costs, Setup Times, and Multiple Constrained Resources. Management Science 44(6), 859-877

Kimms, A. (1996a) Multi-level, single-machine lot sizing and scheduling (with initial inventory). European Journal of Operational Research 89, 86-99

Kimms, A. (1996b) Competitive methods for multi-level lot sizing and scheduling: tabu search and randomized regrets. International Journal of Production Research 34(8), 2279-2298

Kimms, A. (1999) A genetic algorithm for multi-level, multi-machine lot sizing and scheduling. Computers & Operations Research 26, 829-848

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P. (1983) Optimization by Simulated Annealing. Science 220(4598), 671-680

Kuik, R., Salomon, M., van Wassenhove, L.N., Maes, J. (1993) Linear Programming, Simulated Annealing and Tabu Search Heuristics for Lotsizing in Bottleneck Assembly Systems. IIE Transactions 25(1), 62-72

Laguna, M. (1999) A heuristic for production scheduling and inventory control in the presence of sequence-dependent setup times. IIE Transactions 31(2), 125-134

Lang, J.C. (2010) Production and Inventory Management with Substitutions. Lecture Notes in Economics and Mathematical Systems 636, Springer, Berlin.

Lin, S. (1965) Computer Solutions for the Traveling Salesman Problem. Bell Systems Technology Journal 44, 2245-2269

Lourenço, H.R. (1995) Job-shop scheduling: Computational study of local search and large-step optimization methods. European Journal of Operational Research 83, 347-364

Lourenço, H.R., Martin, O.C., Stützle, T. (2003) Iterated Local Search. In: *Handbook of Metaheuristics*. Fred Glover and Gary A. Kochenberger (Eds.), Kluwer Academic Publishers, Norwell.

Maes, J., McClain, J.O., van Wassenhove, L.N. (1991) Multilevel capacitated lotsizing complexity and LP-based heuristics. European Journal of Operational Research 53, 131-148

Martin, O.C., Otto, S.W. (1996) Combining Simulated Annealing with Local Search Heuristics. Annals of Operations Research 63, 57-75

Martin, O.C., Otto, S.W., Felten, E.W. (1991) Large-Step Markov Chains for the Traveling Salesman Problem. Complex Systems 5(3), 299-326

Meyr, H. (2000) Simultaneous lotsizing and scheduling by combining local search with dual reoptimization. European Journal of Operational Research 120, 311-326

Meyr, H. (2002) Simultaneous lotsizing and scheduling on parallel machines. European Journal of Operational Research 139, 277-292

Michalewicz, Z., Fogel, D.B. (2004) *How to Solve It: Modern Heuristics*. 2nd Edition, Springer, Berlin.

Miller, D.M., Chen, H.-C., Matson, J., Liu, Q. (1999) A Hybrid Genetic Algorithm for the Single Machine Scheduling Problem. Journal of Heuristics 5(4), 437-454

Mladenović, N., Hansen, P. (1997) Variable Neighborhood Search. Computers and Operations Research 24(11), 1097-1100

Osman, I.H. (1995) An introduction to meta-heuristics. In: *Operational Research Tutorial Papers*. M. Lawrence and C. Wilsdon (Eds.), Operational Research Society Press, Birmingham.

Osman, I.H. (2004) Metaheuristics: Models, Design and Analysis. Proceedings of the Fifth Asia-Pacific Industrial Engineering and Management Systems Conference, Vol.1, pp.1.2-1 – 1.2-16

Osman, I.H., Kelly, J.P. (1996) Meta-Heuristics: An Overview. In: *Meta-Heuristics: Theory & Applications*. Ibrahim H. Osman and James P. Kelly (Eds.), Kluwer Academic Publishers, Boston.

Özdamar, L., Barbarosoglu, G. (1999) Hybrid heuristics for the multi-stage capacitated lot sizing and loading problem. Journal of the Operational Research Society 50(8), 810-825

Özdamar, L., Barbarosoglu, G. (2000) An integrated Lagrangean relaxation-simulated annealing approach to the multi-level multi-item capacitated lot sizing problem. International Journal of Production Economics 68, 319-331

Özdamar, L., Birbil, S.I. (1998) Hybrid heuristics for the capacitated lot sizing and loading problem with setup times and overtime decisions. European Journal of Operational Research 110, 525-547

Özdamar, L., Birbil, S.I., Portmann, M.-C. (2002) Technical note: New results for the capacitated lot sizing problem with overtime decisions and setup times. Production Planning & Control 13(1), 2-10

Özdamar, L., Bozyel, M.A. (2000) The capacitated lot sizing problem with overtime decisions and setup times. IIE Transactions 32(11), 1043-1057

Pedroso, J.P., Kubo, M. (2005) Hybrid Tabu Search for Lot Sizing Problems. In: *Hybrid Metaheuristics*. M.J. Blesa et al. (Eds.), Lecture Notes in Computer Science 3636, Springer, Berlin.

Pérez Brito, D., Moreno Pérez, J.A., García González, C.G. (2005) Búsqueda por entornos variables: desarrollo y aplicaciones en localización. In: *Avances en Localización de Servicios y sus Aplicaciones.* Blas Pelegrín Pelegrín (Ed.), Servicio de Publicaciones de la Universidad de Murcia, Murcia.

Pirlot, M. (1996) General local search methods. European Journal of Operational Research 92, 493-511

Pitakaso, R., Almeder, C., Doerner, K.F., Hartl, R.F. (2007) A MAX-MIN Ant System for Unconstrained Multi-Level Lot-Sizing Problems. Computers & Operations Research 34, 2533-2552

Pitakaso, R., Almeder, C., Doerner, K.F., Hartl, R.F. (2006) Combining population-based and exact methods for multi-level capacitated lot-sizing problems. International Journal of Production Research 44(22), 4755-4771

Pochet, Y. (2001) Mathematical Programming Models and Formulations for Deterministic Production Planning Problems. Computational Combinatorial Optimization, Springer Lecture Notes in Computer Science Vol. 2241, 57-111

Pochet, Y., Wolsey, L.A. (2006) *Production Planning by Mixed Integer Programming*. Springer, New York.

Prasad, P.S.S., Krishnaiah Chetty, O.V. (2001) Multilevel Lot Sizing with a Genetic Algorithm Under Fixed and Rolling Horizons. International Journal of Advanced Manufacturing Technology 18, 520-527

Quadt, D., Kuhn, H. (2008) Capacitated lot-sizing with extensions: a review. 4OR 6(1), 61-83

Raidl, G.R., Hu, B. (2006) Variable Neighborhood Descent with Self-Adaptive Neighborhood-Ordering. Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics

Salomon, M., Kroon, L.G., Kuik, R., van Wassenhove, L.N. (1991) Some Extensions of the Discrete Lotsizing and Scheduling Problem. Management Science 37(7), 801-812

Salomon, M., Kuik, R., van Wassenhove, L.N. (1993) Statistical search methods for lotsizing problems. Annals of Operations Research 41, 453-468

Shin, H.J. (2007) A Hybrid Search Heuristic for Supply Chain Planning with a Multi-Level Multi-Item Capacitated Lot Sizing Model. IEEE Conference on Industrial Engineering and Engineering Management, 1584-1588

Silver, E.A. (2004) An overview of heuristic solution methods. Journal of the Operational Research Society 55, 936-956

Silver, E.A., Meal, H.C. (1973) A heuristic for selecting lot size quantities for the case of a deterministic time-varying demand and discrete opportunities for replenishment. Production and Inventory Management 14(2), 64-74

Stadtler, H. (1996) Mixed integer programming model formulations for dynamic multi-item multi-level capacitated lotsizing. European Journal of Operational Research 94, 561-581

Stadtler, H. (2001) Losgrößenplanung: Von der Grundlagenforschung zur Anwendung im Supply Chain Management. Presentation slides, Technical University Darmstadt.

Stadtler, H. (2003) Multilevel Lot Sizing with Setup Times and Multiple Constrained Resources: Internally Rolling Schedules with Lot-Sizing Windows. Operations Research 51(3), 487-502

Stadtler, H., Sürie, C. (2000) Description of MLCLSP Test Instances. Technical University Darmstadt.

Stützle, T. (1998) Local Search Algorithms for Combinatorial Problems – Analysis, Improvements, and New Applications. Dissertation, Technical University Darmstadt.

Stützle, T., Hoos, H. (2004) *Stochastic Local Search: Foundations and Applications.* Morgan Kaufmann, San Francisco.

Süer, G.A., Badurdeen, F., Dissanayake, N. (2008) Capacitated lot sizing by using multi-chromosome crossover strategy. Journal of Intelligent Manufacturing 19(3), 273-282

Suerie, C., Stadtler, H. (2003) The Capacitated Lot-Sizing Problem with Linked Lot Sizes. Management Science 49(8), 1039-1054

Talbi, E.-G. (2009) *Metaheuristics: From Design to Implementation.* John Wiley and Sons, Hoboken.

Tang, O. (2004) Simulated annealing in lot sizing problems. International Journal of Production Economics 88, 173-181

Tempelmeier, H. (1997) Resource-constrained materials requirements planning – MRP rc. Production Planning & Control 8(5), 451-461

Tempelmeier, H., Derstroff, M. (1996) A Lagrangean-Based Heuristic for Dynamic Multilevel Multiitem Constrained Lotsizing with Setup Times. Management Science 42(5), 738-757

Toomey, J.W. (1996) *MRP II: Planning for Manufacturing Excellence*. Kluwer Academic Publishers, Norwell.

Vaessens, R.J.M., Aarts, E.H.L., Lenstra, J.K. (1998) A Local Search Template. Computers and Operations Research 25(11), 969-979

Varrentrapp, K.E. (2005) A Practical Framework for Adaptive Metaheuristics. Dissertation, Technical University Darmstadt.

Voss, S. (2000) Meta-heuristics: The State of the Art. Local Search for Planning and Scheduling, Springer Lecture Notes in Computer Science Vol. 2148, 1-23

Wagner, H.M., Within, T.M. (1958) Dynamic Version of the Economic Lot Size Model. Management Science 5(1), 89-96

Wolpert, D.H., Macready, W.G. (1997) No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1), 67-82

Xie, J., Dong, J. (2002) Heuristic Genetic Algorithms for General Capacitated Lot-Sizing Problems. Computers and Mathematics with Applications 44, 263-276

Zhu, X., Wilhelm, W.E. (2006) Scheduling and lot sizing with sequence-dependent setup: A literature review. IIE Transactions 38(11), 987-1007

Appendix

A. Additional Results

Table A.1 – Differences in average improvements of individual neighborhoods

Neighborhood	***Switch***	***Swap***	***Double Swap***	***Column Swap***	***Switch 5***	***Insert 1***	***Delete***	***Insert 0***
	yielded on average a… times better result							
than…								
Switch	***---***	***---***	***---***	***---***	***---***	***---***	***---***	***---***
Swap	***4,07***	***---***	***---***	***---***	***3,52***	***---***	***---***	***2,20***
Double Swap	***4,19***	***1,03***	***---***	***---***	***3,62***	***---***	***---***	***2,27***
Column Swap	***9,41***	***2,31***	***2,25***	***---***	***8,14***	***---***	***---***	***5,09***
Switch 5	***1,16***	***---***	***---***	***---***	***---***	***---***	***---***	***---***
Insert 1	***12,89***	***3,16***	***3,07***	***1,37***	***11,14***	***---***	***---***	***6,97***
Delete	***46,05***	***11,31***	***10,99***	***4,89***	***39,81***	***3,57***	***---***	***24,89***
Insert 0	***1,85***	***---***	***---***	***---***	***1,60***	***---***	***---***	***---***

Table A.2 – Differences in average computational times of individual neighborhoods

Neighborhood	***Switch***	***Swap***	***Double Swap***	***Column Swap***	***Switch 5***	***Insert 1***	***Delete***	***Insert 0***
	was on average … times faster							
than…								
Switch	***---***	***2,07***	***2,76***	***40,28***	***---***	***6,04***	***12,20***	***2,01***
Swap	***---***	***---***	***1,33***	***19,41***	***---***	***2,91***	***5,88***	***---***
Double Swap	***---***	***---***	***---***	***14,61***	***---***	***2,19***	***4,43***	***---***
Column Swap	***---***	***---***	***---***	***---***	***---***	***---***	***---***	***---***
Switch 5	***1,25***	***2,59***	***3,44***	***50,23***	***---***	***7,53***	***15,22***	***2,50***
Insert 1	***---***	***---***	***---***	***6,67***	***---***	***---***	***2,02***	***---***
Delete	***---***	***---***	***---***	***3,30***	***---***	***---***	***---***	***---***
Insert 0	***---***	***1,03***	***1,37***	***20,08***	***---***	***3,01***	***6,08***	***---***

Table A.3 – Percentage of instances for which a given neighborhood combination yielded a better final solution than the other neighborhood combinations

Neighborhood Combination	**Switch/ Swap**	**Switch/ Double Swap**	**Switch/ Column Swap**	**Switch/ Switch 5**	**Switch/ Insert 1**	**Switch/ Delete**	**Switch/ Insert 0**	**Insert0/ Switch**	**Switch5/ Switch**
	yielded a better result in … % of the 21 instances								
than…									
Switch/Swap	---	*5*	*0*	*10*	*10*	*10*	*5*	*33*	*43*
Switch/Double Swap	*86*	---	*0*	*14*	*33*	*24*	*10*	*43*	*71*
Switch/Column Swap	*95*	*29*	---	*19*	*38*	*29*	*14*	*48*	*71*
Switch/Switch 5	*86*	*14*	*5*	---	*38*	*24*	*5*	*43*	*67*
Switch/Insert 1	*86*	*24*	*5*	*10*	---	*10*	*5*	*38*	*67*
Switch/ Delete	*86*	*24*	*5*	*10*	*29*	---	*5*	*43*	*67*
Switch/Insert 0	*90*	*38*	*5*	*19*	*38*	*24*	---	*43*	*71*
Insert 0/Switch	*67*	*57*	*52*	*52*	*62*	*52*	*52*	---	*62*
Switch 5/Switch	*57*	*29*	*29*	*33*	*33*	*33*	*29*	*38*	---

Table A.4 – Percentage of instances for which a given neighborhood combination was faster than the other neighborhood combinations

Neighborhood Combination	**Switch/ Swap**	**Switch/ Double Swap**	**Switch/ Column Swap**	**Switch/ Switch 5**	**Switch/ Insert 1**	**Switch/ Delete**	**Switch/ Insert 0**	**Insert0/ Switch**	**Switch5/ Switch**
	were faster in … % of the 21 instances								
than…									
Switch/Swap	---	*90*	*95*	*86*	*81*	*90*	*81*	*95*	*76*
Switch/Double Swap	*10*	---	*43*	*10*	*10*	*10*	*10*	*33*	*14*
Switch/Column Swap	*0*	*33*	---	*10*	*5*	*10*	*5*	*19*	*5*
Switch/Switch 5	*10*	*90*	*81*	---	*29*	*33*	*38*	*38*	*29*
Switch/Insert 1	*14*	*90*	*90*	*52*	---	*52*	*48*	*48*	*33*
Switch/ Delete	*5*	*90*	*86*	*29*	*19*	---	*19*	*43*	*24*
Switch/Insert 0	*5*	*90*	*90*	*29*	*29*	*43*	---	*38*	*24*
Insert 0/Switch	*0*	*62*	*67*	*57*	*48*	*52*	*57*	---	*19*
Switch 5/Switch	*19*	*86*	*86*	*62*	*57*	*67*	*67*	*71*	---

Table A.5 – Differences in average improvements of neighborhood combinations

Neighborhood Combination	**Switch/ Swap**	**Switch/ Double Swap**	**Switch/ Column Swap**	**Switch/ Switch 5**	**Switch/ Insert 1**	**Switch/ Delete**	**Switch/ Insert 0**	**Insert0/ Switch**	**Switch5/ Switch**
	yielded a … times better result								
than…									
Switch/Swap	---	---	---	---	---	---	---	---	---
Switch/Double Swap	*1,020*	---	---	*1,001*	*1,002*	*1,001*	---	*1,010*	*1,006*
Switch/Column Swap	*1,023*	*1,002*	---	*1,003*	*1,004*	*1,003*	*1,001*	*1,012*	*1,008*
Switch/Switch 5	*1,020*	---	---	---	*1,001*	*1,000*	---	*1,010*	*1,006*
Switch/Insert 1	*1,019*	---	---	---	---	---	---	*1,009*	*1,004*
Switch/ Delete	*1,020*	---	---	---	*1,001*	---	---	*1,009*	*1,005*
Switch/Insert 0	*1,022*	*1,001*	---	*1,002*	*1,003*	*1,002*	---	*1,012*	*1,007*
Insert 0/Switch	*1,010*	---	---	---	---	---	---	---	---
Switch 5/Switch	*1,014*	---	---	---	---	---	---	*1,004*	---

Table A.6 – Differences in average computational times of neighborhood combinations

Neighborhood Combination	**Switch/ Swap**	**Switch/ Double Swap**	**Switch/ Column Swap**	**Switch/ Switch 5**	**Switch/ Insert 1**	**Switch/ Delete**	**Switch/ Insert 0**	**Insert0/ Switch**	**Switch5/ Switch**
	were on average … times faster								
than…									
Switch/Swap	**---**	*1,961*	*2,444*	*1,813*	*1,724*	*1,702*	*1,815*	*1,661*	*1,226*
Switch/Double Swap	---	**---**	*1,246*	---	---	---	---	---	---
Switch/Column Swap	---	---	**---**	---	---	---	---	---	---
Switch/Switch 5	---	*1,082*	*1,348*	**---**	---	---	*1,001*	---	---
Switch/Insert 1	---	*1,138*	*1,418*	*1,052*	**---**	---	*1,053*	---	---
Switch/ Delete	---	*1,152*	*1,436*	*1,066*	*1,013*	**---**	*1,067*	---	---
Switch/Insert 0	---	*1,080*	*1,346*	---	---	---	**---**	---	---
Insert 0/Switch	---	*1,181*	*1,472*	*1,092*	*1,038*	*1,025*	*1,093*	**---**	---
Switch 5/Switch	---	*1,600*	*1,993*	*1,479*	*1,406*	*1,388*	*1,481*	*1,354*	**---**

Table A.7 – Objective function values for the thirty test instances

[T] denotes an instance from Tempelmeier and Derstroff (1996); [S] denotes an instance from Stadtler (2003). The capital letter next to [T] or [S] in the Instance column specifies the problem class. The column "Best known solution" refers to the best solution reported by either Tempelmeier and Derstroff (1996), Stadtler (2003), Pitakaso et al. (2006), or Almeder (2010). The last column gives the optimum as calculated with CPLEX / the lower bound as reported in the test instance libraries.

	Switch	*Switch / Swap*	*Insert 0 / Switch*	*Best Local Search Result*	*VNS version 1.0*	*Best known solution*	*Optimum / Lower Bound*
Instance							
[T] A G507145	16.790,39	16.790,39	16.891,89	16.790,39	16.765,40	15.326,22	15.326,22
[T] A K001532	28.566,00	28.566,00	28.566,00	28.566,00	28.566,00	28.566,00	28.566,00
[S] A+ G502132	615.005,40	617.155,34	612.865,65	612.865,65	611.449,00	606.926,97	548.535,00
[S] A+ K502431	113.614,12	108.744,73	109.216,59	108.744,73	110.440,82	104.838,93	85.456,00
[S] A+ G502541	183.653,80	179.558,95	182.653,13	178.495,94	164.931,95	156.421,54	122.361,00
[T] B G514111	2.883,00	2.883,00	2.883,00	2.883,00	2.883,00	2.883,00	2.883,00
[T] B K527441	14.125,78	14.125,78	14.305,22	14.017,72	13.886,70	13.713,67	13.713,67
[S] B+ G512441	135.108,25	127.725,86	132.915,95	127.725,86	127.432,64	121.765,00	107.750,00
[S] B+ K522340	125.989,50	123.335,00	131.442,00	123.335,00	122.396,00	119.052,00	113.972,00
[S] B+ G542132	147.234,00	132.157,86	138.712,42	131.697,63	131.686,53	131.508,00	95.537,00
[S] B+ K512142	160.959,57	154.689,33	153.307,51	147.864,23	144.360,73	147.138,00	112.383,00
[T] C G001342	474.980,00	473.164,00	488.713,00	467.512,00	469.327,00	457.057,00	445.656,50
[T] C K801252	46.620,24	46.213,57	44.693,28	44.693,28	44.309,50	42.070,57	37.571,92
[S] C K501321	110.827,50	108.649,00	112.409,25	108.649,00	107.307,00	108.762,00	95.289,00
[S] C G501430	506.009,79	497.060,01	508.658,14	497.060,01	505.705,46	481.503,00	436.807,00
[S] C G501532	676.245,47	666.813,06	647.079,75	647.079,75	636.289,90	610.207,00	538.208,00
[S] C^m K501422	128.073,04	125.063,71	131.345,88	125.063,71	123.690,63	137.917,00	100.288,00
[S] C^m G501131	684.464,09	675.099,22	708.512,41	641.183,39	626.835,37	686.961,00	471.208,00
[S] C^m K502532	666.318,73	667.443,66	665.874,81	665.874,81	666.226,87	###	###
[T] D G029341	478.022,00	477.914,00	479.860,16	471.870,00	468.879,00	455.609,58	441.416,64
[T] D K815551	44.696,51	44.527,34	44.824,89	41.285,93	42.132,10	40.711,89	35.942,50
[S] D G042541	634.788,00	582.848,35	612.155,06	582.848,35	###	551.594,00	529.749,00
[S] D G012432	463.296,83	455.081,19	454.861,27	454.861,27	###	447.528,00	407.802,00
[S] D G512531	607.776,17	601.729,26	604.743,81	601.729,26	###	542.650,00	462.844,00
[S] D+ G542442	4.632.147,9	4.702.909,8	4.680.226,8	4.632.147,9	4.583.172,9	###	###
[S] D+ G012332	1.333.329,0	1.389.477,0	1.355.107,0	1.333.329,0	1.324.372,0	###	###

[S] E G502440	2.604.081,1	2.659.609,5	2.727.752,9	2.564.284,3	2.523.155,3	###	###
[S] E G501230	4.380.769,0	4.529.246,6	4.536.933,3	4.380.769,0	4.374.622,8	4.146.330,4	3.303.435,9
[T] E G019251	1.693.122,9	1.707.821,4	1.702.663,4	1.658.478,4	1.683.774,5	1.570.583,5	1.384.120,4
[S] E+ G501120	7.228.937,0	7.373.352,7	7.448.557,5	7.201.110,7	7.093.405,7	###	###

Table A.8 – Results of the Wilcoxon signed-rank test for VNS versions 1.0 and 2.0

The null hypothesis is that there is no difference in the mean of the results of version 1.0 and version 2.0.

	A+ (5 minutes)	**A+ (30 minutes)**	**C (10 minutes)**	**C (60 minutes)**
Summary of ranks				
Positive ranks *(v2 < v1)*	*57*	*53*	*248*	*34*
Negative ranks *(v1 < v2)*	*63*	*67*	*276*	*56*
Zero difference	*0*	*0*	*76*	*10*
Z	*1,087*	*2,092*	*2,190*	*2,670*
N	*120*	*120*	*524*	*90*
Two-tailed test ***(H_1 : v1 ≠ v2)***				
P	*0,277*	*0,036*	*0,029*	*0,008*
Statistical significance	*no*	***beyond 5% level***	***beyond 5% level***	***beyond 1% level***
One-tailed test ***(H_1 : v1 < v2)***				
P	*0,139*	*0,018*	*0,014*	*0,004*
Statistical significance	*no*	***beyond 5% level***	***beyond 5% level***	***beyond 1% level***

B. About the Author

Personal Information

Name:	Sylvia-Maria Pichler
Date of Birth:	September 29, 1978
Place of Birth:	Linz/Danube
Nationality:	Austrian

Education

1985-1989	Elementary school
1989-1997	Akademisches Gymnasium, Linz/Danube
1997-2010	University of Vienna, study program International Business Administration

Employment

1996-2000	Holiday internships at Swietelsky Baugesellschaft m.b.H., 4020 Linz, Edlbacherstraße 10
2007-2008	Study assistant at the Chair of Production and Operations Management, Department of Business Administration, University of Vienna, 1210 Vienna, Brünner Straße 72