Masters Degree in Informatics Engineering
Dissertation
Final Report

Branch-and-Bound for the Hypervolume Subset Selection Problem

June, 2017

Ricardo Jorge Pires Gomes
rjgomes@student.dei.uc.pt

Adviser
Prof. Dr. Luís Paquete

FCTUC DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Masters Degree in Informatics Engineering
Dissertation
Final Report

Branch-and-Bound for the Hypervolume Subset Selection Problem

June, 2017

Ricardo Jorge Pires Gomes
rjgomes@student.dei.uc.pt

Adviser
Prof. Dr. Luís Paquete

Jury
Prof. Dr. Jorge Sá Silva
Prof. Dr. Nuno Lourenço

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

The main focus of this thesis is the design and analysis of a branch-and-bound algorithm for the hypervolume subset selection problem for an arbitrary number of objectives. This problem arises in selection procedures of heuristic algorithms for multiobjective optimisation, in which the goal is to select a small subset of good compromise solutions. The branch-and-bound approach discussed in this thesis combines several notions of bounds and a branching strategy. In particular, four bounding functions and a dynamic variable ordering for the branching strategy are proposed. Moreover, a parallel version of the branch-and-bound algorithm is presented, which integrates a thread pool to explore, concurrently, the nodes of the search tree. The branch-and-bound algorithm is compared with a state-of-the-art solution approach based on an integer programming formulation. The experimental results indicate that our branch-and-bound approach performs faster for a wide range of instances.

Keywords – Branch-and-Bound Algorithm, Hypervolume Subset Selection Problem, Multiobjective Optimisation, Integer Programming

Resumo

O foco principal desta tese é a análise e síntese de um algoritmo de *branch-and-bound* para o problema de seleção do subconjunto que maximiza o indicador de hipervolume para um número arbitrário de objetivos. Este problema surge nos procedimentos de seleção em heurísticas para otimização multiobjetivo, no qual se pretende selecionar um pequeno subconjunto de soluções de compromisso. A abordagem de *branch-and-bound* discutida nesta tese combina várias noções de limites e uma estratégia de *branching*. Em particular, quatro funções de limite e um ordenamento dinâmico de variáveis para a estratégia de *branching* são propostos. Uma versão paralela do algoritmo de *branch-and-bound* é também apresentada, que integra uma *pool* de *threads* que explora os nós da árvore de procura de forma concorrentemente. O algoritmo de *branch-and-bound* é comparado com uma abordagem baseada na formulação de programação inteira. Os resultados experimentais obtidos numa grande quantidade de instâncias deste problema indicam que a nossa abordagem tem melhor desempenho.

Palavras Chave – Algoritmo de *Branch-and-Bound*, Problema de Seleção do Subconjunto que maximiza o Indicador de Hipervolume, Otimização Multiobjectivo, Programação Inteira

Acknowledgements

First of all, I would like to thank my family, especially my parents, for their continued support. I would like to thank my adviser, Prof. Luís Paquete, for all the guidance and attention given to this thesis. I would like to thank Andreia Guerreiro for all the help and assistance given throughout this thesis, in particular, for the integration of the calculation of hypervolume contributions in linear time for the three-dimensional case. I thank Tobias Kuhn for providing the code to generate the integer programming models. To my friends, thank you for all the encouragement and motivation. Finally, I would like to thank everyone who helped me directly or indirectly in achieving this goal.

Contents

1 Introduction **1**

2 Preliminaries **5**
2.1 Hypervolume Subset Selection Problem 5
2.2 Branch-and-Bound 7

3 State-of-the-art **11**
3.1 Complexity of the Hypervolume Subset Selection Problem 11
3.2 Two-Dimensional Hypervolume Subset Selection Problem 12
3.3 Three-Dimensional Hypervolume Subset Selection Problem 12
3.3.1 Subset Enumeration 12
3.3.2 Integer Programming Model 13
3.3.3 Heuristics 14

4 The Branch-and-Bound Algorithm **17**
4.1 Overview 17
4.2 Branching Strategy 17
4.3 Bounding Functions 19
4.3.1 Calculation of ub_1 19
4.3.2 Calculation of ub_2 22
4.3.3 Calculation of ub_3 23
4.3.4 Calculation of ub_4 23
4.4 Parallel Branch-and-Bound 26

5 Experimental Analysis **31**
5.1 Methodology 31
5.2 Computational Results 33

6 Conclusion **39**
6.1 Future Work 39

Bibliography **41**

Appendix A **45**

A.1 Results 45

List of Figures

1.1 Illustration of different aircraft designs 2

2.1 (a) Two-dimensional example of the hypervolume indicator; (b) three-dimensional example of the hypervolume indicator 6
2.2 (a) Two-dimensional example of the hypervolume contribution (grey region); (b) two-dimensional example of all exclusive hypervolume contributions (grey regions) 7
2.3 Typical enumeration tree in a B&B algorithm 8

3.1 Two-dimensional example of the decomposition of the hypervolume in rectangles 14

4.1 Two-dimensional example for $k = 3$ of (a) the calculation of ub_1 and (b) with the exclusive hypervolume contributions of points b and e removed . 21
4.2 (a) Input points generated by Eq. (4.3) for $n = 9$; (b) subset that covers the most rectangles for $k = 3$ 25

5.1 Three-dimensional examples of nondominated points datasets: (a) Cliff, (b) Concave, (c) Convex and (d) Linear fronts 32
5.2 Average running time in seconds taken by both approaches for three-dimensional fronts with fixed $n = 50$ 35
5.3 Average running time in seconds taken by both versions of the B&B algorithm for three-dimensional fronts with fixed $k = n/2$ 36
5.4 Average number of nodes taken by the B&B algorithm for three-dimensional fronts with fixed $k = n/2$ 37
5.5 Average running time taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Cliff fronts with fixed $n = 200$ 37
5.6 Average running time taken by the B&B algorithm for four-dimensional fronts with fixed $n = 50$ 38

List of Tables

5.1 Machine specifications, compiler and solver flags 33

A.1 Average running time in seconds taken by both approaches for three-dimensional Cliff fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ 45

A.2 Average running time in seconds taken by both approaches for three-dimensional Convex fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ 45

A.3 Average running time in seconds taken by both approaches for three-dimensional Concave fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ 46

A.4 Average running time in seconds taken by both approaches for three-dimensional Linear fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ 46

A.5 Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Cliff fronts with fixed $k = n/2$ 46

A.6 Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Convex fronts with fixed $k = n/2$ 47

A.7 Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Concave fronts with fixed $k = n/2$ 47

A.8 Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Linear fronts with fixed $k = n/2$ 47

A.9 Average number of nodes and running time in seconds taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Cliff fronts with fixed $n = 200$ and $k \in \{10, 20, \ldots, n-10\}$ 48

A.10 Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Cliff fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ 48

A.11 Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Convex fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ 49

A.12 Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Concave fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ . . . 49
A.13 Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Linear fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$ . . . 49
A.14 Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Cliff fronts with fixed $k = n/2$ . . . 50
A.15 Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Convex fronts with fixed $k = n/2$ . . . 50
A.16 Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Concave fronts with fixed $k = n/2$ . . . 50
A.17 Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Linear fronts with fixed $k = n/2$ . . . 51

Acronyms

HSSP Hypervolume Subset Selection Problem

DP Dynamic Programming

B&B Branch-and-Bound

IP Integer Programming

MO Multiobjective Optimisation

LP Linear Programming

NP Nondeterministic Polynomial

OS Operating System

CPU Central Processing Unit

Chapter 1

Introduction

The goal of optimisation is to find the global optimum of a given problem from the set of feasible solutions. Most real-life optimisation problems can be naturally formalized with more than one objective. In such problems, there is usually no single solution that simultaneously optimises all objectives due to their conflicting nature.

For example, in aircraft design optimisation problems, which involve many design variables (e.g. aerodynamics, propulsion, controls), a large number of conflicting objectives need to be combined. Consider that we are interested in the design of fast aircraft with large payload capacities. We might decide to maximize both the payload capacity and the speed of the aircraft. However, in order for the aircraft to accommodate larger payload capacities, it must be designed larger, which will increase its weight and will, as a result, slow down the aircraft. Therefore, there is usually no best design that simultaneously has the largest payload capacity and is the fastest, since the objectives payload capacity and speed conflict with each other. This is usually a critical design issue for the takeoff phase of aircraft, since the takeoff speed is highly dependent on the aircraft weight in the sense that the heavier the weight, the greater the speed needed. Still, we might be able to design aircraft with stronger propulsion engines, so that we can maintain the speed for the larger designs. However, if we consider now a third objective, for example, the fuel consumption or even the cost of the aircraft, the new design with the stronger propulsion engines would be worse in these objectives.

Figure 1.1 illustrates the trade-off between the objectives top speed and maximum payload capacity. We can choose the Boeing aircraft, which has the greatest payload capacity but in contrast it is not very fast. We can also choose the Fighter Jet, which trades payload capacity for maximum speed. We can even choose the Airbus and Concorde aircraft that are also optimal designs but not the Private Jet since the Concorde has both greater payload capacity and speed. Except for the Private Jet, all other aircraft are optimal solutions, that is, no aircraft is better than the others both in terms of top speed and maximum payload capacity.

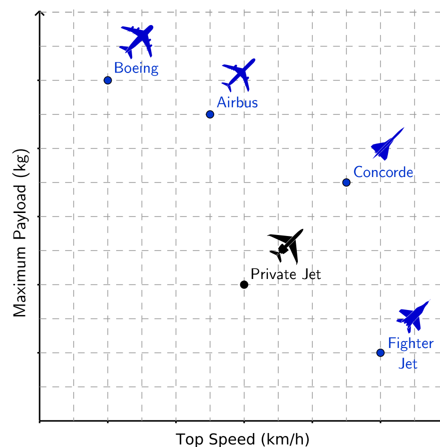

Figure 1.1: Illustration of different aircraft designs

When dealing with optimisation problems with multiple objectives, such as the one described above, the solution techniques should find the complete set of Pareto-optimal solutions; a solution is Pareto-optimal if there is no other feasible solution that is at least as good with respect to all objectives and strictly better with respect to at least one objective. Finding the complete set of Pareto-optimal solutions may be intractable (Figueira et al., 2017), and usually heuristic approaches are the methods of choice to tackle problems with multiple objectives. Heuristic approaches often select a small subset of solutions, based on a certain quality measure. This is known as the Subset Selection problem and can be defined in several different manners, depending on the quality measure that is chosen. Such measures are used to assess the quality of a set of solutions in terms of their distribution and their closeness to the Pareto-optimal set in the objective space, among other properties (Zitzler et al., 2003). One of these measures is the Hypervolume Indicator, which maps the region dominated by an approximation set of solutions into a scalar value. The hypervolume indicator is one of the most popular quality measures, due to several interesting properties (Zitzler et al., 2007). Unfortunately, the hypervolume indicator becomes computationally demanding for an arbitrary number of objectives.

The problem of selecting a subset with a given cardinality from a larger set of candidate solutions that maximizes the hypervolume indicator is known as the Hypervolume Subset Selection Problem (HSSP). This problem arises in selection procedures of heuristic methods, in which the objective is to select a small representative subset of solutions. For two objectives, the HSSP can be solved efficiently (in polynomial time), using the principles of dynamic programming (DP) (Bader, 2009; Bringmann et al., 2014; Kuhn et al., 2016). For more than two objectives, the HSSP is known to require exponential amount of time (Bringmann et al., 2017). Approximation methods, such as greedy and local search, have been largely applied to approximate the optimal value of the HSSP for

more than two objectives (Guerreiro et al., 2015; Basseur et al., 2016; Bringmann et al., 2017).

The main contribution of this thesis is the design and analysis of a branch-and-bound (B&B) algorithm to solve the HSSP for an arbitrary number of objectives. We devise four upper bounds and introduce a dynamic variable ordering for the B&B algorithm. Furthermore, we present a parallel version of the B&B algorithm. Finally, we conduct an in-depth experimental analysis of our approach and compare it with a solution approach based on the Integer Programming (IP) model proposed in the literature (Kuhn, 2015).

The thesis is structured as follows. In Chapter 2, the necessary definitions, notation and principals of the B&B approach are introduced. In Chapter 3, the literature related to the HSSP is reviewed. In Chapter 4, the proposed B&B algorithm is presented. In Chapter 5, an analysis on the B&B algorithm is performed on a wide range of instances of the problem. In Chapter 6, final conclusions are drawn. Additionally, some ideas for future work are presented.

Chapter 2

Preliminaries

In this chapter, some relevant definitions for this thesis are introduced. In Section 2.1, some concepts, definitions and notation used in this thesis are given and in Section 2.2, the B&B paradigm is presented.

2.1 Hypervolume Subset Selection Problem

Multiobjective optimisation (MO) involves finding solutions that optimise multiple objective functions. Without loss of generality, it is assumed maximization for all objectives. More formally, a MO problem can be defined as:

$$\begin{aligned} \text{maximize} \quad & (f_1(x), \ldots, f_d(x)) \\ \text{subject to} \quad & x \in \mathbb{X} \end{aligned} \tag{2.1}$$

where $d > 1$ is the number of objective functions and $\mathbb{X}$ is the set of feasible solutions. Due to the conflicting nature of the objectives, there is usually no single optimal solution to a MO problem, but multiple optimal solutions.

Definition 2.1.1. Dominance: Let $p, q \in \mathbb{R}^d$ be two points in objective space. Point p dominates point q, denoted by $p \succeq q$, if $p_i \geq q_i$ for all $i = 1, \ldots, d$ and $p \neq q$. If neither point dominates the other (i.e. $p \not\succeq q$ and $q \not\succeq p$), then p and q are nondominated points.

Definition 2.1.2. Pareto-optimality: A solution $x \in \mathbb{X}$ is Pareto-optimal if there is no other solution $x' \in \mathbb{X}$ for which it holds that $f(x') \succeq f(x)$.

Definition 2.1.3. Hypervolume Indicator: Let S be a set of nondominated points in objective space $\mathbb{R}^d$. Without loss of generality, assuming that the reference point r is 0^d, the hypervolume indicator of S is given as (Bringmann and Friedrich, 2010):

$$H(\mathrm{S}) = \mathrm{VOL}\left(\bigcup_{(s_1,\ldots,s_d) \in \mathrm{S}} [0, s_1] \times \ldots \times [0, s_d] \right) \tag{2.2}$$

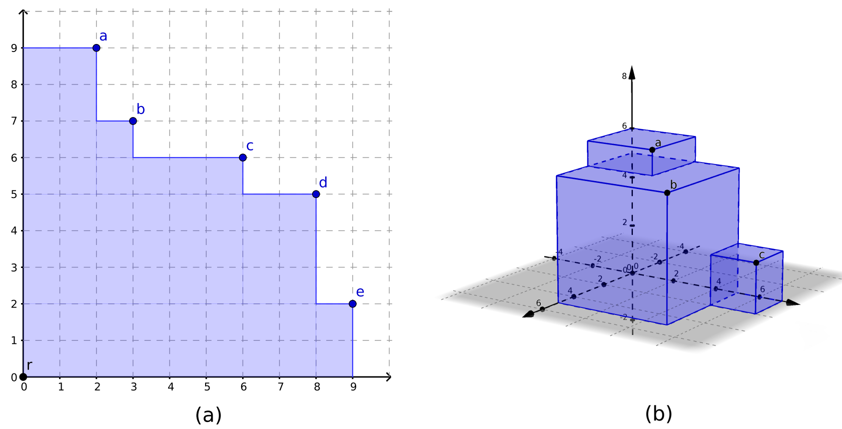

Figure 2.1: (a) Two-dimensional example of the hypervolume indicator; (b) three-dimensional example of the hypervolume indicator

where VOL(·) is the Lebesgue measure. In the two-dimensional case, $H(S)$ is the area of the union of a set of boxes, each of which bounded from above by each point in S and bounded below by the reference point. In the three-dimensional case, we take the volume instead of the area. Figure 2.1 shows (a) the hypervolume indicator in the two-dimensional case and (b) the hypervolume indicator in the three-dimensional case, which are illustrated by the blue regions.

Definition 2.1.4. Hypervolume Contribution: Given a point $p \in \mathbb{R}^d$ and a point set $S \subset \mathbb{R}^d$, the hypervolume contribution of p to S is:

$$H(p, S) = H(S \cup \{p\}) - H(S)$$

The hypervolume contribution of a point p with respect to a set S corresponds to the increase of hypervolume, once point p is added to S. In Figure 2.2 (a), the hypervolume contribution of point b to $S = \{c, d\}$ is illustrated by the area of the grey region.

Definition 2.1.5. Exclusive Hypervolume Contribution: Given a point $p \in \mathbb{R}^d$ and a point set $S \subset \mathbb{R}^d$, in which $p \subseteq S$, the exclusive hypervolume contribution of p in S is:

$$H_x(p, S) = H(S) - H(S \setminus \{p\})$$

The exclusive hypervolume contribution of a point p in a set S corresponds to the hypervolume that is not dominated by any other point of set S, except by the point p. In Figure 2.2 (b), the exclusive hypervolume contributions of each point in $S = \{a, b, c, d, e\}$ are illustrated by the grey regions.

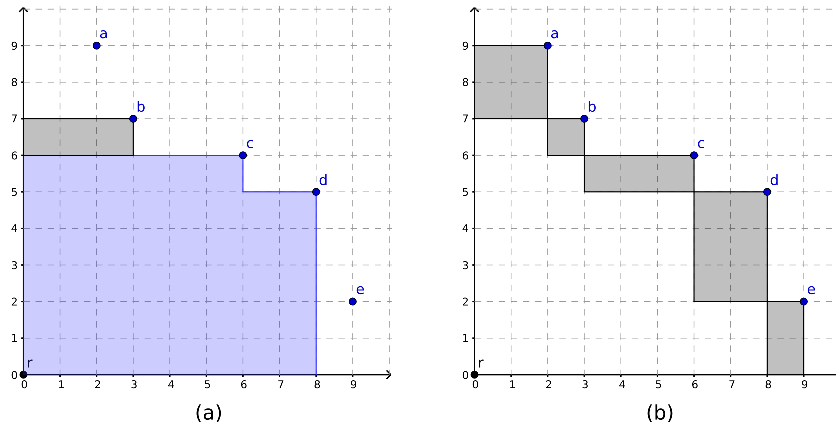

Figure 2.2: (a) Two-dimensional example of the hypervolume contribution (grey region); (b) two-dimensional example of all exclusive hypervolume contributions (grey regions)

Definition 2.1.6. Hypervolume Subset Selection Problem: Given a set $S \subset \mathbb{R}^d$ of n nondominated points and an integer $k \in \{1, 2, \ldots, n\}$, the HSSP is defined as finding a subset $A \subseteq S$ of size k that maximizes the hypervolume indicator among all subsets of size k. Formally, it is stated as follows:

$$H(A) = \max_{\substack{B \subseteq S \\ |B| = k}} H(B)$$

2.2 Branch-and-Bound

B&B is a generic algorithm paradigm that combines enumeration techniques and bounding functions to solve optimisation problems (Clausen, 1999). In a B&B algorithm, a large problem is divided recursively into a few smaller subproblems. This is known as the branching strategy. Due to the exponentially increasing number of potential subproblems, a pure enumeration strategy may be infeasible or too slow in practice. In order to improve upon this inefficiency, bounds are estimated for each smaller subproblem, which may allow some of them to be ignored.

The branching implicitly enumerates all feasible solutions, by successively partitioning the feasible set into smaller sets. The branching is similar to an exhaustive search and may be visualized as a construction of a search tree whose interior nodes are represented by partial solutions and leaf nodes by complete solutions. Figure 2.3 illustrates an example of such enumeration tree, in which x_i is a decision variable of a given optimisation problem. Each node of the search tree may be seen as a different optimisation subproblem, which is based on the original problem (the problem at the root node), but with some of the decision variables already fixed.

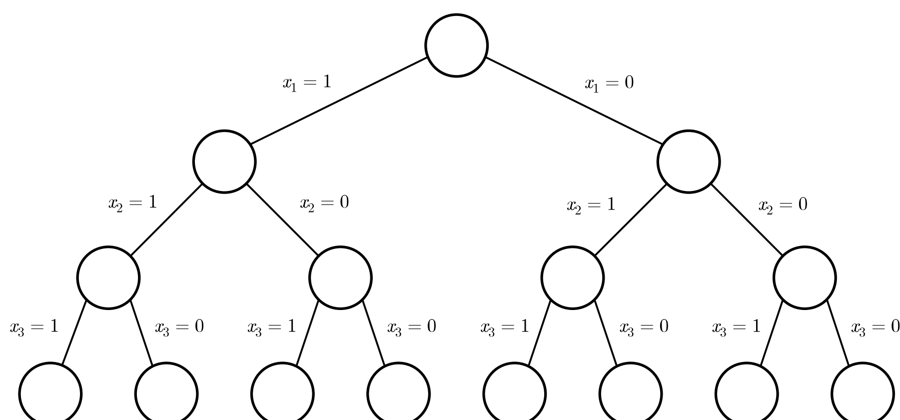

Figure 2.3: Typical enumeration tree in a B&B algorithm

Due to the exponential growth of the number of nodes of the tree, a mere exhaustive search can quickly become computationally expensive. In order to improve the performance of a B&B algorithm, bounding functions are used to discard or prune certain branches of the search tree that do not lead to an optimum. This may be done by estimating bounds on partial solutions to a problem, which can be of two types: lower and upper bounds. In a maximization problem, the value of the best solution found during the search process represents a lower bound and its solution is designated as the *incumbent*. An upper bound on a partial solution is an optimistic value that must always be greater than or equal to the value of the best solution that can be constructed using the partial solution. Before the enumeration process takes place on a particular node, its partial solution is checked against the lower and upper bounds. If the partial solution has an upper bound that is less than or equal to the lower bound, the branching on that node can be stopped. In such cases, regardless of the path that is chosen in the branching of that node, it will never lead to a new optimal solution. In order to tighten the bounding window, that is, the gap between the lower bound and the upper bound, the *incumbent* and its objective value are updated every time a new best solution is found.

The B&B method is used in problems that are by nature very hard to solve. In such problems, efficient polynomial algorithms are probably not available or are not known and thus, an implicit enumeration of all the possible solutions is a possible way of finding the best solution. The performance of a B&B algorithm, greatly depends on how effectively the search tree is pruned by the bounds. Bounds that vastly prune the search tree are sufficient for a B&B algorithm to perform well. However, since the upper bound calculation is performed at each node of the search tree, any increase in the computation time of the upper bound is reflected in the overall computation time of the algorithm. It

is advantageous that the bounds prune the search tree as early as possible so that large portions of the tree do not require to be searched. Upper bound calculation efficiency and pruning effectiveness plays a critical role in a B&B algorithm. However, since upper bounds need to be checked against lower bounds, for the actual pruning to be verified, good lower bounds are equally needed. Initial lower bounds can be obtained at the root node of the search tree using heuristic methods. Heuristics are methods to find solutions to a problem, typically in a short amount of time, when optimality is not a requirement. These methods can take advantage of some properties of the problems to gain knowledge on how to construct a good and sometimes even optimal solution.

Chapter 3

State-of-the-art

In this chapter, the literature related to the HSSP is reviewed. In Section 3.1, the complexity of the HSSP is analysed. Moreover, a brief introduction to computational complexity analysis based on the book of Garey and Johnson (1990) is presented. In Section 3.2, three known HSSP algorithms for the two-dimensional case are discussed. Finally, in Section 3.3, three different approaches that were proposed for the three-dimensional HSSP are presented.

3.1 Complexity of the Hypervolume Subset Selection Problem

Computational complexity analysis is concerned with measuring and classifying problems according to the resources (e.g. time, memory) they need to be solved. Computational complexity theory describes complexity in terms of well-defined complexity classes. Complexity classes group problems that require similar resources. The Polynomial class is the set of all decision problems that are solvable in polynomial time, whereas the Nondeterministic Polynomial (NP) class is the set of all decision problems that can be verified in polynomial time but may not be solvable in polynomial time. The NP-hard class is the set of all problems that are not solvable in polynomial time and the NP-complete class is the set of all problems that are both in the NP and NP-hard classes. A NP problem is said to be complete, if it is possible to reduce it to any other NP problem in polynomial time. While decision problems only require a yes or no answer, counting problems need to count answers and are at least as hard as the corresponding decision problems. For counting problems the complexity class #P is the analog of the NP class.

It has been shown in Bringmann and Friedrich (2008) that the calculation of the hypervolume indicator is #P-hard. In addition, the fastest known algorithm has a time complexity of $O(n^{d/3} \text{ polylog } n)$ (Chan, 2013). Since the HSSP is at least as hard as the calculation of the hypervolume indicator, both results suggest that the HSSP must be an hard problem, for an arbitrary dimension. In fact, it has been conjectured that the HSSP is NP-hard for three and more dimensions (Bader, 2009). Only very recently, Bringmann

et al. (2017) have shown that in fact this is true. In the two-dimensional case the HSSP can be solved efficiently, in polynomial time.

3.2 Two-Dimensional Hypervolume Subset Selection Problem

Three DP algorithms were proposed in the literature for the two-dimensional HSSP. Bader (2009) introduced an algorithm with $O(n^2k)$ time complexity that is based on the idea that the hypervolume contribution of a point to a set only depends on its two adjacent neighbors. Later, Bringmann et al. (2014) proposed an algorithm with $O(n(k + \log n))$ time complexity. The algorithm calculates at each iteration ℓ, the maximal hypervolume indicator achievable with at most ℓ points by computing the upper envelope of n linear functions in linear time using a *convex hull trick*. Very recently, Kuhn et al. (2016) proposed a different approach with $O(k(n-k) + n \log n)$ time complexity, by reducing the HSSP to a *k-link shortest path* formulation that can be solved with DP. To the best of our knowledge, this is currently the fastest algorithm available for the two-dimensional HSSP.

3.3 Three-Dimensional Hypervolume Subset Selection Problem

This thesis focus on improvements to the multidimensional HSSP ($d \geq 3$), specially the three-dimensional case, for which some new results have been obtained recently. In particular, the first IP formulation of the HSSP and a new method to update hypervolume contributions in linear time have been proposed. In the following sections, three different approaches for the three-dimensional HSSP are presented. These include a subset enumeration, the IP formulation and several heuristic methods.

3.3.1 Subset Enumeration

In order to solve the HSSP, it is sufficient to enumerate all subsets of size k and select the subset that has the maximal hypervolume indicator. Clearly, this approach involves computing the hypervolume indicator for each of the $\binom{n}{k}$ possible subsets, which is considered to be computationally too expensive (Bader and Zitzler, 2011). However, Bringmann and Friedrich (2010) showed that it is possible to avoid the calculation of $\binom{n}{k}$ conventional hypervolumes. The algorithm solves the equivalent problem of determining a subset of $n-k$ points with minimal hypervolume contribution with respect to the original set in $O(n^{d/2} \log n + n^{n-k})$ time complexity. The algorithm is based on the idea that it is possible to maintain the contribution volumes of every set of $n-k$ points during the hypervolume calculation. This allows the direct computation of the contribution of every set of $n-k$ points, which gives an additive term of $\binom{n}{k}$ in the running time of the algorithm instead of a multiplicative factor.

Very recently, at the time of the writing of this thesis, Bringmann et al. (2017) proposed a DP algorithm with $n^{O(\sqrt{k})}$ time complexity, which improved upon the known time complexity bound for the three-dimensional case.

3.3.2 Integer Programming Model

An IP model was proposed in Kuhn (2015) to solve the three-dimensional HSSP. The model is a generalization of the two-dimensional IP formulation proposed in Kuhn et al. (2016), which is based on the decomposition of the region dominated by the hypervolume indicator into hyperrectangles or boxes. Figure 3.1 illustrates an example of such decomposition, where A_{ij} is a rectangle. This decomposition process involves a preprocessing step, which provides the hyperrectangles to generate the IP model. In the two-dimensional case, this decomposition generates a set of rectangles and requires $O(n^2)$ time. As for the three-dimensional case, the decomposition needs to generate a set of three-dimensional boxes, which increases the time complexity by a multiplicative factor of n. This may be a bottleneck of the model, both in terms of time and memory, since an equal number of constraints need to be generated and inserted in the model. The following IP formulation models the three-dimensional HSSP (Kuhn, 2015):

$$\text{maximize} \quad \sum_{A_{ij\ell} \in \mathcal{A}} w_{ij\ell} \cdot x_{ij\ell} \tag{3.1}$$

$$\text{subject to} \quad \sum_{r=1}^{n} x_{h(y^r)} = k \tag{3.2}$$

$$x_{ij\ell} \leq \sum_{\substack{y^r \in N: \\ (i,j,\ell)^T \leqq h(y^r)}} x_{h(y^r)} \qquad \forall A_{ij\ell} \in \mathcal{A} \setminus \{A_{h(y^s)} : y^s \in N\} \tag{3.3}$$

$$x_{ij\ell} \in \{0,1\} \qquad \forall A_{ij\ell} \in \mathcal{A}$$

where $N = \{y^1, \dots, y^n\}$ is the initial set of points, $\mathcal{A}$ is a set of boxes of the dominated region, the weight $w_{ij\ell}$ corresponds to the volume of the box $A_{ij\ell}$ and h is a function that maps a point to the indexes of a box A that is not dominated by any other point. Constraint (3.3) ensures that the boxes covered by the chosen points are also selected. Constraint (3.2) ensures that exactly k points are selected and the objective function (3.1) calculates the hypervolume indicator of a current selection, which has to be maximized. The model can be easily changed to work for a fixed dimension different than three.

In terms of time complexity, the IP problem is NP-hard (Garey and Johnson, 1990). However, IP solvers often use a technique called Linear Programming (LP) relaxation that transforms an IP problem into a related LP problem that can be solved in polynomial time. This technique consists on relaxing the domain of the problem variables to gain additional information about the solution of the original IP problem. Still, this solution may not be optimal, and more importantly, it may not even be feasible, that is, it can

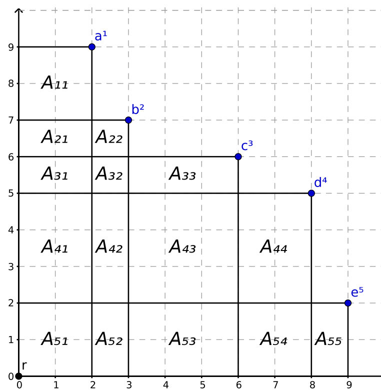

Figure 3.1: Two-dimensional example of the decomposition of the hypervolume in rectangles

violate some constraints of the original IP model. In order to take advantage of the relaxation technique, IP solvers use the valid LP problems only to provide bounds of the original IP problem. This allows the solvers to reduce the solution space without loosing the optimal solution. In the experiments reported in Kuhn (2015), the CPLEX IP solver was used as well as a B&B scheme that was adjusted specially for the structure of the three-dimensional HSSP. The B&B scheme was able to solve problem instances up to 100 points, within 5 minutes for different values of k. However, it still required some seconds even for instances with 50 points, due to the preprocessing step. Instances of 200 points were also tested but the construction took two hours only for the initial CPLEX model and had a size of 2 Gigabytes, which led to the tests being aborted. The B&B scheme was also slightly faster than the CPLEX solver. Nonetheless, even though the results were considered satisfactory, it was concluded that future research was needed to overcome the huge amount of time required to generate the model.

3.3.3 Heuristics

Both approaches discussed in Section 3.3.1 and Section 3.3.2 rely on some kind of exhaustive enumeration mechanism. These approaches, capable of finding the global optimum are known as exact methods. Heuristic methods, on the other hand, might not find the best solution but they usually have the advantage of being able to compute a reasonable approximation in a short amount of time. Common heuristic approaches for the HSSP often rely on the hypervolume contributions to make a choice, about what point to select or eliminate next (Guerreiro et al., 2015; Basseur et al., 2016).

The approach proposed in Guerreiro et al. (2015) starts with an empty subset and at each of the k iterations, the point that has the largest hypervolume contribution is added to the subset, followed by an efficient update in linear time of all the hypervolume contributions of the points that are not in the subset. The time complexity of this

algorithm was shown to be $O(n(k + \log \text{n}))$. Also, this algorithm is able to return at least the guaranteed theoretical approximation of $1 - 1/e \simeq 0.63$, but empirically the approximation quality was shown to stay within 0.89 of the optimal values.

Very recently, Basseur et al. (2016) explored the approximation quality and the computational cost of four heuristic algorithms. The following three greedy procedures and local search heuristics were tested: greedy forward selection, greedy backward elimination, greedy sequential insertion and a first-improvement hill-climbing local search. The greedy forward selection is the same algorithm proposed in Guerreiro et al. (2015). The backward elimination algorithm starts with the original set of points as the subset, and at each of the $n - k$ iterations the point with the least hypervolume contribution is removed. The greedy sequential insertion algorithm starts with a subset of k elements randomly chosen from the original set. Iteratively, a remaining point from the original set is selected at random and added to the subset, and the point with the least hypervolume contribution is removed. The algorithm stops when all points from the original set have been considered exactly once for integrating in the subset solution. The first-improvement hill-climbing local search starts with a subset of k points chosen from the original set, provided by some initialization process (e.g. greedy). At each step, the algorithm searches for pairs of points to be swapped, one being selected while the other not, in order to improve the hypervolume value of the obtained subset. The local search procedure stops when no swap can increase the hypervolume indicator of the subset.

According to the experimental results described by the authors, the local search returned the better approximations among all heuristics, especially when combined with the greedy procedures. With regard to computational cost, the greedy sequential insertion and the greedy forward selection were the less computationally demanding. Except for the greedy forward selection, it is not known whether any of the other heuristics tested is able to return a guaranteed approximation to the optimal. As such, the local search heuristic combined with the greedy forward selection as the initialization technique should be preferred over other heuristics. Alternatively, if faster running times are a requirement, then the greedy forward selection alone is the most adequate heuristic.

Chapter 4

The Branch-and-Bound Algorithm

In this chapter, the main contributions of this thesis are introduced. In Section 4.1, an overview of the B&B algorithm is provided. In Section 4.2, the branching strategy is explained and in Section 4.3 the calculation of four bounding functions is presented. Then, in Section 4.4 a parallel version of the B&B algorithm is discussed.

4.1 Overview

The following terminology describes two different sets of nondominated points that are used to represent a node of the search tree:

Included point set – The set of points that were explicitly accepted in the branching. This set represents a current subset solution.

Unassigned point set – The set of points that still need to be branched. This set keeps the points of which a subset needs to be chosen in order to obtain a complete solution.

Algorithm 1 shows the pseudocode of our B&B approach for the HSSP, where S is the included point set, P is the unassigned point set, k is the subset size of the HSSP, s_{best} is the *incumbent*, ub_i is an upper bound for $i = 1, 2, 3$ and lb is the hypervolume indicator of the *incumbent* solution. The following sections of this chapter will explain Algorithm 1 in more detail. Note that, the hypervolume contribution should be understood with respect to a point $p \in$ P to set S (see Definition 2.1.4) and the exclusive hypervolume contribution should be understood with respect to a point $p \in$ P in set S $\cup$ P (see Definition 2.1.5).

4.2 Branching Strategy

Starting at the root node, the unassigned point set is the initial set of points, the included point set is an empty subset and $lb = -\infty$. The branching mechanism performs a depth-first-search on the unassigned point set. Each node has two branching options:

Algorithm 1 B&B algorithm for the HSSP

```
 1: function branch-and-bound(S, P)
 2:     if |S| = k then
 3:         if H(S) > lb then
 4:             lb ← H(S)
 5:             s_best ← S
 6:         return
 7:     if |P| = 0 then
 8:         return
 9:     if |P| + |S| < k then
10:         return
11:     if min(ub_1, ub_2, ub_3) ≤ lb then
12:         return
13:     p ← argmax_{q∈P}{H(q, S)}
14:     branch-and-bound(S ∪ {p}, P \ {p})
15:     branch-and-bound(S, P \ {p})
```

accept (line 14, Algorithm 1) and ignore (line 15, Algorithm 1) a chosen point p of the unassigned point set. Therefore, at each node the search develops at most two children nodes, similar to the construction of a binary search tree. The branching is stopped for a node, if one of the following four cases is satisfied:

1. The included point set is complete (line 2, Algorithm 1)
2. The unassigned point set is empty (line 7, Algorithm 1)
3. The unassigned point set has not enough points to reach a complete solution (line 9, Algorithm 1)
4. The node is pruned by the bounds (line 11, Algorithm 1)

In such cases, no nodes are developed and the search proceeds by backtracking the current branch until a new node is found that still needs to be branched or no such node exists. For the case when the included point set is complete, the best solution and its hypervolume indicator are updated if it can improve the *incumbent* (lines 3 to 5, Algorithm 1).

Whenever a point is accepted, the hypervolume indicator of the included point set can be updated incrementally by adding the hypervolume contribution of the accepted point. This allows the hypervolume indicator of the included point set to be retrieved in constant time at any point in the algorithm. Whenever a point is ignored, the included point set and its hypervolume indicator remain unchanged.

The hypervolume contributions of the points in unassigned point set with respect to the included point set can be used as the variable ordering: choose the point from the unassigned point set that has the largest hypervolume contribution (line 13, Algorithm 1). In other words, the search chooses at each node the point of the unassigned point

set that can increase the most hypervolume in the included point set. This allows the branching to be dynamic, by successively selecting a point from the unassigned point set that is heuristically a good candidate for a current partial solution. This is an advantage over choosing a random point or even using a fixed order point set, if the branching leads to the good solutions earlier. As a consequence, it may allow better lower bounds to be found earlier, which therefore increases the pruning effectiveness of upper bounds. In terms of time complexity, the calculation of the hypervolume contributions can be done for $d = 3$ in linear time on the total number of points of the unassigned point set using the method proposed in Guerreiro et al. (2015). In higher dimensions, the equation given in Definition 2.1.4 can be used, which calls the hypervolume indicator to calculate the hypervolume contributions. In this case, using the fastest known algorithm to calculate the hypervolume indicator leads to $O(n^{d/3+1} \log n)$ time complexity.

The depth-first-search traversal approach was chosen over a breath-first-search, in order for the algorithm to arrive at complete solutions earlier. This combined with the dynamic variable ordering allows the construction of an initial good lower bound at the first branch of the algorithm. In addition, the first node to be branched is the one that accepts the point of the unassigned point set, since it increases the hypervolume indicator of the included point set.

4.3 Bounding Functions

As stated in Section 2.2, initial lower bounds can be obtained using heuristic methods. However, this will not be the case for the proposed algorithm. The dynamic variable ordering for the branching can quickly obtain an initial lower bound at the first complete branch of the search tree without the need of extra initialization techniques or heuristics. In fact, the first complete branch explored by the B&B algorithm corresponds to the greedy algorithm proposed in Guerreiro et al. (2015). The dynamic variable ordering allows not only a fast calculation of an initial lower bound, but also provides a flexible mechanism for the algorithm to improve the *incumbent* much quicker during execution. In the following subsections of this section, four upper bounds are proposed for the B&B algorithm.

4.3.1 Calculation of ub_1

The hypervolume indicator of the union of the included point set with the unassigned point set represents an upper bound of a current partial solution. Formally, ub_1 can be defined as follows:

$$ub_1 = H(\mathrm{S} \cup \mathrm{P}) \tag{4.1}$$

The hypervolume indicator is strictly monotonic with respect to dominance (Bader,

2009) and therefore by adding points from the unassigned point set to the included point set, the value of the hypervolume indicator never decreases. Note that the input consists only of nondominated points. Therefore, if all points of the unassigned point set are accepted, this upper bound returns the maximum possible hypervolume that it is possible to achieve in a current node.

Several improvements can be made to speedup the calculation and tighten the value of ub_1. A possible improvement is to cache the calculation of ub_1 for the branch that accepts points. This is possible because, when a point is accepted, it is moved from the unassigned point set to the included point set and, therefore, the value of ub_1 remains unchanged. Another improvement is to subtract the smallest $|\mathrm{S} \cup \mathrm{P}| - k$ exclusive hypervolume contributions of the unassigned point set in the union of the included point set with the unassigned point set, to the value of ub_1. Let $\mathrm{P} = \{q_1, \ldots, q_{|\mathrm{P}|}\}$. Assume without loss of generality that

$$i \leq j \iff H_x(q_i, \mathrm{S} \cup \mathrm{P}) \leq H_x(q_j, \mathrm{S} \cup \mathrm{P})$$

The improved ub_1' is computed as follows

$$ub_1' = ub_1 - \sum_{i=1}^{|\mathrm{S} \cup \mathrm{P}| - k} H_x(q_i, \mathrm{S} \cup \mathrm{P}) \tag{4.2}$$

Due to line 9 of Algorithm 1, $|\mathrm{S} \cup \mathrm{P}| - k \geq 0$, that is, the total number of points of $\mathrm{S} \cup \mathrm{P}$ is always greater than or equal to k. Therefore, we can subtract the smallest $|\mathrm{S} \cup \mathrm{P}| - k$ exclusive hypervolume contributions to the value of ub_1, since the subset at most will have k points. Figure 4.1 shows the effect of this improvement, where $\mathrm{S} = \{a\}$, $\mathrm{P} = \{b, c, e, f\}$ and the exclusive hypervolume contributions of the points in P with respect to $\mathrm{S} \cup \mathrm{P}$ are $\{1, 4, 2, 3\}$ respectively. Figure 4.1 (a) shows the calculation of ub_1 using Eq. (4.1) and (b) the calculation of ub_1' using Eq. (4.2), which are illustrated by the blue region plus the grey region.

The calculation of the exclusive hypervolume contributions also provides means to incrementally update the value of ub_1. The objective is to reuse the computation of the exclusive hypervolume contributions to update the value of ub_1 for the new branching nodes, without requiring to recalculate the full hypervolume indicator at each node. As it was explained above, for the case when a point is accepted, the value of ub_1 does not change. As for the case when a point is rejected, the value of ub_1 decreases exactly by the value of the exclusive hypervolume contribution of the rejected point. In the following we prove this fact.

Proposition 4.3.1.

$$H(\mathrm{S} \cup \mathrm{P} \setminus \{p\}) = ub_1 - H_x(p, \mathrm{S} \cup \mathrm{P})$$

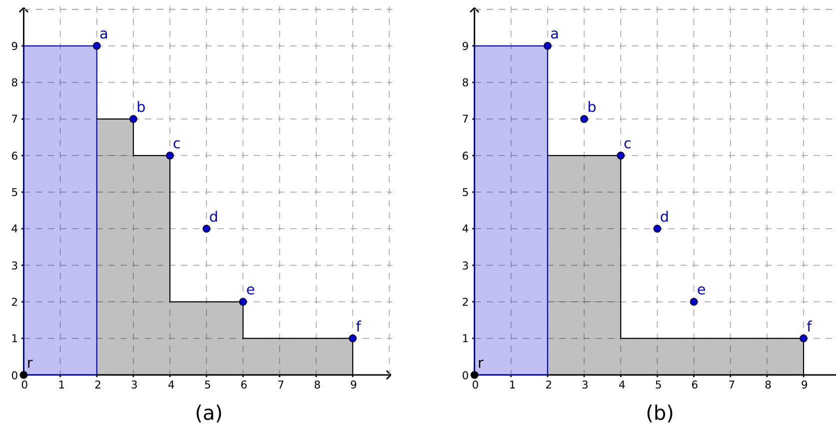

Figure 4.1: Two-dimensional example for $k = 3$ of (a) the calculation of ub_1 and (b) with the exclusive hypervolume contributions of points b and e removed

Proof. From Definition 2.1.5 and the definition of ub_1 (see Eq. (4.1)) we obtain

$$\begin{aligned} H(\mathrm{S} \cup \mathrm{P} \setminus \{p\}) &= ub_1 - H(\mathrm{S} \cup \mathrm{P}) + H(\mathrm{S} \cup \mathrm{P} \setminus \{p\}) \\ &= H(\mathrm{S} \cup \mathrm{P} \setminus \{p\}) \end{aligned}$$

In order to incrementally update the value of ub_1, we perform as follows: At the root node set ub_1 to the hypervolume indicator of the initial set of points; then, when a point is rejected in the branch, subtract from ub_1 the exclusive hypervolume contribution of the rejected point. Note that for the branch that accepts points, nothing needs to be done.

The complexity of the calculation of ub_1 is bounded by the computation of the hypervolume indicator. Currently, the fastest known algorithm for calculating the hypervolume indicator has a time complexity of $O(n^{d/3} \text{ polylog } n)$. For example, in $d = 3$ the calculation of ub_1 has linearithmic time complexity on the total number of points of $\mathrm{S} \cup \mathrm{P}$. For the incremental version, the complexity of the calculation of ub'_1 is bounded by the computation of all exclusive hypervolume contributions and by the selection of the smallest $|\mathrm{S} \cup \mathrm{P}| - k$ exclusive hypervolume contributions. Computing all exclusive hypervolume contributions for $d \leq 3$ can be done in $O(n)$ and for $d = 4$ in $O(n^2)$ (Guerreiro and Fonseca, 2017). The similar problem of selecting the n-th largest or smallest element can be solved in the worst-case in linear time using Selection Algorithms (Musser, 1997). The problem of selecting the smallest $|\mathrm{S} \cup \mathrm{P}| - k$ elements can also be solved on average in linear time on the total number of elements using the same Selection Algorithms. For $d > 4$ calculating all exclusive hypervolume contributions can be computationally expensive and as a result, the algorithm falls back to the exclusive contributions at the root node. This is possible because the exclusive hypervolume contributions at the root node are always less than or equal to the exclusive hypervolume contributions for

any other node of the search tree. This results in a less tightening of the upper bound than that of the actual exclusive hypervolume contributions. Fortunately, the exclusive hypervolume contributions at the root only need to be calculated once. However, and as a consequence of falling back to the exclusive hypervolume contributions at the root node, the incremental calculation of ub_1 is disabled for $d > 4$ in our implementation.

4.3.2 Calculation of ub_2

The hypervolume indicator of the included point set plus the largest $k - |\mathrm{S}|$ hypervolume contributions of the unassigned point set with respect to the included point set represents an upper bound of a partial solution. Let $\mathrm{P} = \{q_1, \ldots, q_{|\mathrm{P}|}\}$. Assume without loss of generality that

$$i \leq j \iff H(q_i, \mathrm{S}) \geq H(q_j, \mathrm{S})$$

The bound ub_2 is computed as follows

$$ub_2 = H(\mathrm{S}) + \sum_{i=1}^{k-|\mathrm{S}|} H(q_i, \mathrm{S})$$

Let W be a set of n nondominated points, from which we want to choose k. Let $\mathrm{J} = \{j_1, \ldots, j_k\}$ be the set of k points in W that maximizes the hypervolume indicator. Let $\mathrm{Z} = \{z_1, \ldots, z_k\}$ be the set of k points in W, whose sum of the individual hypervolume contributions of its elements is the largest possible. We want to prove that:

$$H(\mathrm{J}) \leq \sum_{i=1}^{k} H(\{z_i\})$$

note that:

$$H(\mathrm{J}) \leq \sum_{i=1}^{k} H(\{j_i\})$$

Since Z was constructed so as to obtain the largest possible sum, we have that:

$$\sum_{i=1}^{k} H(\{j_i\}) \leq \sum_{i=1}^{k} H(\{z_i\})$$

Therefore,

$$H(\mathrm{J}) \leq \sum_{i=1}^{k} H(\{j_i\}) \leq \sum_{i=1}^{k} H(\{z_i\})$$

Furthermore, if the volumes of the largest $k - |\mathrm{S}|$ contributions do not intersect with each other there is no need to branch the current node, even if this upper bound returns a value greater than the lower bound. Note also that if this happens an optimal solution is found. This is known as the fathom node technique and, in this particular case, by bounds. Sometimes, in addition to the computation of an upper bound, a

feasible complete solution that is the best possible expansion of the current node is also constructed. In such cases, the branching for the node can be stopped.

The complexity of the calculation of ub_2 is bounded by the selection of the largest $k-|\mathrm{S}|$ hypervolume contributions. Similarly to the selection of the smallest $|\mathrm{S}\cup\mathrm{P}|-k$ exclusive hypervolume contributions for ub'_1, the selection of the largest $k-|\mathrm{S}|$ elements can also be solved with Selection Algorithms in linear time on average on the total number of elements (Musser, 1997). Note that the calculation of the hypervolume contributions is performed by the branching part for the variable ordering, and thus, its computation is not considered in the time complexity of the calculation of ub_2. In order to verify if the fathom node technique described above can be applied, the hypervolume indicator of the union of the included point set with the points of the unassigned point set that have the largest $k-|\mathrm{S}|$ hypervolume contributions must be equal to ub_2. If it is less than ub_2, then there is certainly intersections between some of the contributions.

4.3.3 Calculation of ub_3

As explained in Subsection 3.3.3, if the remaining points of the unassigned point set are selected using the greedy algorithm proposed in Guerreiro et al. (2015), the hypervolume indicator of the solution returned by it has a guaranteed approximation of $1-1/e$. Therefore, the multiplication of $1/(1-1/e)$ by the hypervolume indicator of the solution returned by the greedy algorithm represents an upper bound of a current partial solution. More formally, ub_3 can be defined as follows:

$$ub_3 = \mathrm{greedy}(\mathrm{P}, \mathrm{S}, k) \times \frac{1}{1-1/e}$$

where greedy(P, S, k) is a procedure that returns the hypervolume indicator of S after choosing $k-|\mathrm{S}|$ points from P for integrating in S, using the algorithm proposed in Guerreiro et al. (2015).

This greedy algorithm is only available for $d \leq 3$. For $d = 3$ the time complexity of this upper bound function is bounded by the time complexity of the greedy algorithm, which is $O(n(k+\log n))$. In higher dimensions, ub_3 can be computed naïvely by calculating the hypervolume contributions using the standard hypervolume indicator. However, at the best-case, this leads to an algorithm with $O(kn^{d/3+1}$ polylog $n)$ time complexity, which can be computationally expensive.

4.3.4 Calculation of ub_4

The following upper bound is only available in the two-dimensional case and for this reason it was not implemented for Algorithm 1.

Consider the decomposition step described in Subsection 3.3.2 that partitions the region dominated by the hypervolume indicator into hyperrectangles. Let h be the

number of hyperrectangles already selected so far in the search and let h_{max} be the maximum number of hyperrectangles needed to solve a particular HSSP for a given n and k. The hypervolume indicator of the included point set plus the largest $h_{max} - h$ hypervolumes of the hyperrectangles of the unassigned point set represents an upper bound of a current partial solution.

A lookup table can be built that has the maximum number of hyperrectangles that are needed to solve the HSSP for any given n and k. In order to use the lookup table, the maximum number of hyperrectangles for each value of n and k must be calculated and stored in advance. For the two-dimensional case it is possible to obtain the exact number of maximum rectangles by running any of the exact algorithms described in Section 3.2 for a given n and k with a prepared input in the form of:

$$\{(x,y) : y = -x + n + 1,\ x = 1,\dots,n\} \tag{4.3}$$

Eq. (4.3) generates a set of points in such a way that the area of each rectangle is exactly 1. Doing so, ensures that the hypervolume indicator of the solution returned by the exact algorithm is exactly equal to the number of rectangles that are dominated by the points of the solution. Figure 4.2 (a) illustrates an example of an input of points generated by Eq. (4.3), in which the rectangles generated by the decomposition of the hypervolume indicator have exactly 1 of area. Figure 4.2 (b) shows the hypervolume indicator (blue region) of the best subset, which is equal to the number of dominated rectangles.

The process above generates a matrix with all the maximum numbers of rectangles up to a given n for each value of k. The disadvantage of this method is that it is computationally demanding to run the algorithm for each value of k up to a desired n, even if the lookup table only needs to be filled once. For example, if $n = 100$, the exact algorithm would need to be executed $n(n+1)/2 = 5050$ times, which can be computationally expensive.

For the two-dimensional case, our results suggest a new relation between the maximum number of rectangles and the maximum number of edges of any Turán Graph, a complete multipartite graph whose partite sets differ by at most one vertex. A complete graph is a graph in which every pair of graph vertices is connected by exactly one edge and a multipartite or s-partite graph is a graph whose graph vertices can be partitioned into s different independent sets, a subset of vertices of a graph G such that no two vertices in the subset represent an edge of G. A r-clique in a graph G is a complete subgraph of G with exactly r vertices. Turán's theorem gives an upper bound on the maximum number of edges of any Turán Graph with v vertices without a r-clique as (Aigner, 1995):

$$T(v,r) = \frac{(r-2)v^2}{2(r-1)} \tag{4.4}$$

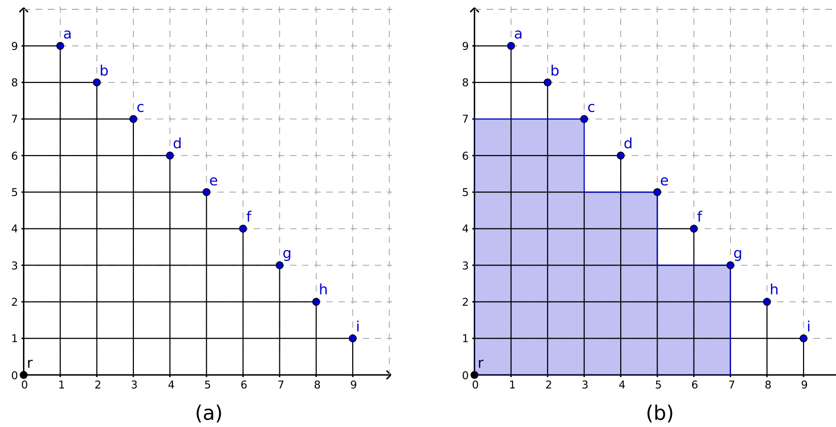

Figure 4.2: (a) Input points generated by Eq. (4.3) for $n = 9$; (b) subset that covers the most rectangles for $k = 3$

Based on simulations up to 100 points, the number of rectangles returned by Eq. (4.4), where $v = n+1$ and $r = k+2$, is always greater than or equal to the exact maximum number of rectangles. Thus, it can be used to obtain an upper bound on the maximum number of rectangles, in constant time, up to $n = 100$. In fact it is able to correctly match 1545 table entries in 5050 total, up to $n = 100$ and approximates with an average error of approximately 2.19 rectangles for all entries.

Since Eq. (4.4) gives an upper bound and not the exact maximum number of edges of any Turán Graph, there is still a trade-off between the optimality achieved using the exact algorithm to fill the lookup table, with the speed of using the constant time approximation of Eq. (4.4). Also, it is not known whether the exact maximum number of edges of any Turán Graph is exactly equal to the maximum number of rectangles needed to solve the two-dimensional HSSP.

Another result in the two-dimensional case is that for $k > n/2$ the maximum number of rectangles is:

$$\frac{n(n+1)}{2} - n + k \tag{4.5}$$

since removing one point only involves removing at most one rectangle from the maximum possible number of rectangles when $n = k$. Therefore, Eq. (4.5) can be used to obtain in constant time the maximum number of rectangles for a given n if $k > n/2$.

For three and more dimensions it is not possible to apply the same techniques proposed for the two-dimensional case to calculate the maximum number of rectangles. In two dimensions, regardless of the points configuration that is chosen for the input, the number of the maximum rectangles will always be equal for the same values of n and k. This does not happen for three and more dimensions and for this reason further research is needed to find a way to obtain the maximum number of hyperrectangles.

4.4 Parallel Branch-and-Bound

In a B&B algorithm, the order in which the nodes of the search tree are expanded in the branching, as well as the computation of the bounds are independent of the final result, that is, it will always terminate with the global optimal solution. This attribute makes B&B methods the ideal algorithms for parallel computation. The goal of parallel computation is to speed up the execution time of an algorithm. In a B&B algorithm this may be done by exploring the nodes of the search tree concurrently, preferably in such a way that the nodes are evenly distributed among processing units. This may allow the global optimum to be found earlier, which consequently helps the bounding functions to prune even more nodes in the search tree.

In order to take advantage of systems with multiple processing units, a thread pool was implemented for the parallel version of the B&B algorithm. A thread pool maintains a group of threads, each of which waits on tasks to be given. The main advantage of a thread pool over creating a thread for each new task is that thread creation and destruction overhead is negated. The benefits of a thread pool become more clear in a B&B algorithm, as the tasks are essentially the nodes of the search tree, which can grow exponentially. There is still a small overhead in the locking mechanism of the thread pool that is required to synchronize the access to a queue of tasks.

Algorithms 2, 3 and 4 show the pseudocode of the parallel version of the B&B algorithm, where:

- Q is the queue of tasks
- t is the number of pool threads
- $workers$ is the current number of working threads
- $stop$ is a boolean variable that triggers the termination of the threads
- m_1 and m_2 are mutexes
- lock(m) is a procedure that acquires a lock on a mutex m
- unlock(m) is a procedure that releases a lock on a mutex m
- wait(m) is a procedure that blocks a current thread on a condition variable until it is notified by some other thread
- notify_one() is a procedure that unblocks one idle thread waiting on a conditional variable
- notify_all() is a procedure that unblocks all idle threads waiting on a conditional variable

A mutex is used to block multiple threads from entering critical sections simultaneously. For example, critical sections were added to prevent the invalidation of two or

Algorithm 2 Parallel B&B algorithm for the HSSP

```
1:  function branch-and-bound(S, P)
2:      if |S| = k then
3:          lock(m_2)
4:          if H(S) > lb then
5:              lb ← H(S)
6:              s_best ← S
7:          unlock(m_2)
8:          return
9:      if |P| = 0 then
10:         return
11:     if |P| + |S| < k then
12:         return
13:     if min(ub_1, ub_2, ub_3) ≤ lb then
14:         return
15:     p ← argmax_{q∈P}{H(q, S)}
16:     if workers < t then
17:         schedule(S ∪ {p}, P \ {p})
18:     else
19:         branch-and-bound(S ∪ {p}, P \ {p})
20:     if workers < t then
21:         schedule(S, P \ {p})
22:     else
23:         branch-and-bound(S, P \ {p})
```

more best solutions found simultaneously (lines 3 to 7, Algorithm 2) and to synchronize any access to the queue of tasks (lines 3 to 6, Algorithm 3) and (lines 5 to 22, Algorithm 4).

The parallel version of the B&B starts by setting $workers = 0$ and $stop =$ false and creates t worker threads, which run Algorithm 4. These threads will have both the roles of producing and consuming tasks and immediately go into idle state by waiting on a monitor (line 17, Algorithm 4). Then, the root node is scheduled and placed in the queue of tasks. In order to schedule a task, a snapshot of the node is created (line 2, Algorithm 3). The snapshot procedure takes a full copy of the state of the node, which requires the included point set and the unassigned point set to be copied. After scheduling a task, we signal one of the threads (line 5, Algorithm 3). One thread will eventually wake and will retrieve the task from the queue (line 21, Algorithm 4). The thread then starts working on the task by calling Algorithm 2 (line 23, Algorithm 4) and it may decide if it wants to schedule the node's new branches. A thread decides to schedule a new task only if the number of workers is less than the total number of pool threads (lines 16 and 20, Algorithm 2), that is, there exists an idle thread waiting in the pool. If the thread decides to schedule the node for the branch that rejects a point, it still has to backtrack the current node. Whenever worker threads completely finishes a task, they go again

Algorithm 3

```
1: function schedule(S, P)
2:     task ← (S,P)
3:     lock(m_1)
4:     Q.push(task)
5:     notify_one()
6:     unlock(m_1)
```

Algorithm 4

```
1: function work()
2:     dismiss ← true
3:     while true do
4:         idle ← true
5:         lock(m_1)
6:         if dismiss then
7:             dismiss ← false
8:         else
9:             if Q.empty() = false then
10:                idle ← false
11:            else
12:                workers ← workers − 1
13:                if workers = 0 then
14:                    stop ← true
15:                    notify_all()
16:        while stop = false and Q.empty() do
17:            wait(m_1)
18:        if stop then
19:            return
20:        workers ← workers + idle
21:        task ← Q.pop()
22:        unlock(m_1)
23:        branch-and-bound(task.S, task.P)
```

into idle state. The termination of the algorithm is triggered by one of the worker threads when it finishes a task, there are no other worker threads and the queue of tasks is empty (lines 13 to 15, Algorithm 4).

Several features were added to the standard thread pool in order to stabilize and improve it. We allow worker threads to schedule multiple tasks, and not just one, in multiple occasions. No lock is acquired to check if the number of worker threads is less than the total number of pool threads. Therefore, multiple worker threads may detect simultaneously that there is an idle thread, and all can schedule tasks to the queue, due to the monitor being released upon threads going into idle state (line 17, Algorithm 4). Also, from the moment that one worker thread finishes a task to the moment that it is ready to work, other worker threads may be fast enough to schedule both of their children

nodes. Furthermore, backtracking the current node is a relatively fast operation, which may allow even more tasks to be scheduled if nodes are found during the backtracking process. The tasks can then be stored in a priority queue, so that when the thread is ready to work, it can choose a task based on some priority. For the priority queue, a max-heap structure is used, in which the hypervolume indicator of the included point set is the compare method. A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node. The goal of the priority queue is to not allow the threads to work first on branches that have small chance of improving the $incumbent$ solution. In a way, the pool threads are competing to see which one can generate the best subsets. Furthermore, task scheduling is disabled until batches of tasks are emptied from the priority queue. This can greatly reduce the number of tasks scheduled and, consequently, the memory footprint, without penalizing in the performance, since threads can still work. Still, the parallel version will need at least t times more memory for the run time, in comparison with the sequential version of the B&B (Algorithm 1).

Chapter 5

Experimental Analysis

In this chapter, an analysis on the B&B algorithms introduced in Chapter 4 is performed. In Section 5.1, the methodology for the design of the experimental tests is explained and in Section 5.2, the B&B algorithm is assessed and compared with a solver that solves the IP model described in Section 3.3.2.

5.1 Methodology

Four different types of nondominated points instances were generated: Cliff, Concave, Convex and Linear fronts (Emmerich and Fonseca, 2011; Lacour et al., 2017). Figure 5.1 illustrates examples of each front. The parameter n of the datasets varies between 10 and 100 points, depending on the type of the test. Three main tests were conducted. In the first test, we compared the performance between the B&B algorithm and an IP solver. In this test, we used three-dimensional instances with fixed $n = 50$ and $k \in \{1, 2, \ldots, n\}$. We were not able to test instances with $n > 50$, due to the excessive amount of time taken by the IP solver. In the second test, we analysed how the running time of the B&B algorithm grows by changing $n = \{10, 20, \ldots, 100\}$ for a fixed $k = n/2$. The parameter k was fixed to $n/2$, since the number of subset combinations is the maximum. In the third test we replicated the second test, but using the parallel version of the B&B algorithm with $t = 2$. We did not test an higher number of threads, since the machine used only had two cores.

In all tests, for each triple of front, n and k, we generated 10 different instances. In the first test we recorded the running time required to solve each instance. Then, we computed the average running time for each triple. Note that, for testing the IP formulation, the running time used to generate the model was not considered. In the second and the third tests, both running time and the number of nodes of the search tree were recorded. We computed the average running time and number of nodes for each pair of front type and n. In the third test, we also calculated the speedup and the nodes factor of the parallel version of the B&B algorithm. The speedup is defined as the time taken by parallel version divided by the time taken by the sequential version and the

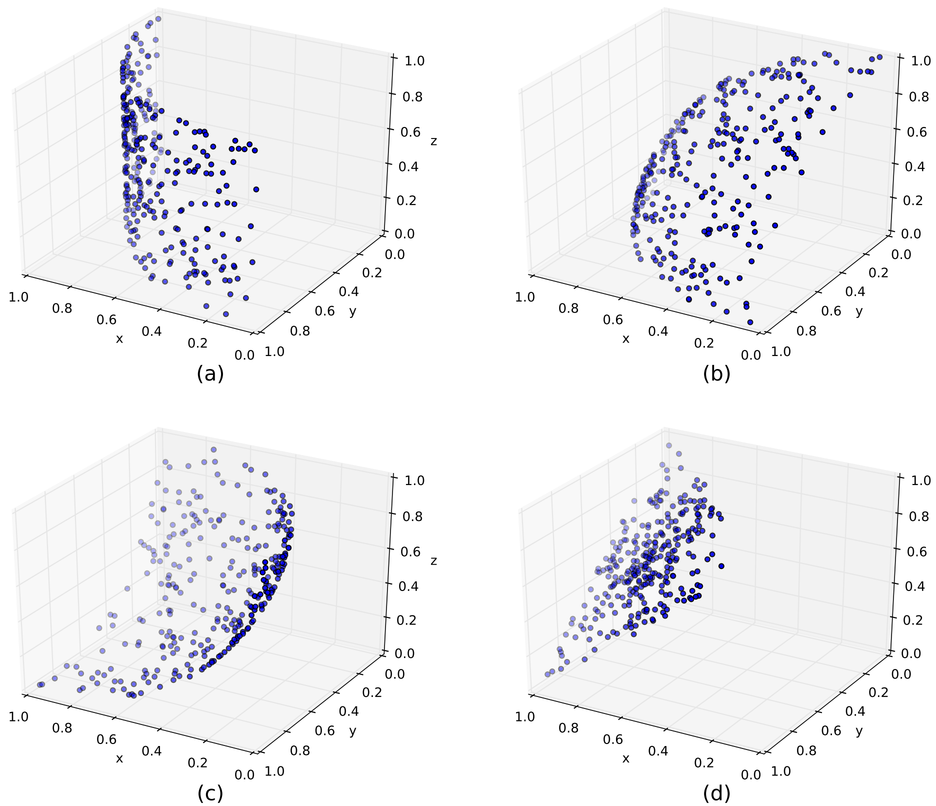

Figure 5.1: Three-dimensional examples of nondominated points datasets: (a) Cliff, (b) Concave, (c) Convex and (d) Linear fronts

nodes factor is defined as the number of nodes taken by sequential version divided by the number of nodes taken by the parallel version.

All experiments were conducted on a machine with a Linux Operating System (OS), equipped with an Intel i5-2410M processor, running at 2.30 GHz and 4GB RAM. The source code for the B&B algorithm was written in C++ and compiled with GCC 5.4.0. The IP models were executed using the GNU Linear Programming Kit (GLPK) 4.57 solver. More information on the machine, compiler and solver can be viewed in Table 5.1.

Even though the tests were run in order to assess the performance of the B&B algorithm, they were also used for testing the correctness of the code. Several datasets were generated for the experimental tests and their best solutions were obtained using the IP model, which were used to compare with the solutions returned by the B&B algorithm. Furthermore, several regression tests were built during the implementation of the algorithm, which helped validating its components and thereby reducing code mistakes. In the regression tests we preferred the use of canonical methods, such as the hypervolume indicator, to validate, for example, the incremental calculation of ub_1.

C++ was chosen over C due to the various containers and multithreading support provided by the Standard Template Library (STL). The problem of obtaining the smallest

OS	Ubuntu 16.04, 64-bit
Memory	4 GB, 1333 MHz
CPU	Intel i5-2410M Dual-Core, 2.30 GHz
Compiler	-std=c++11 -O3 -flto -march=native -pthread
Solver	--dual --tmlim 1000

Table 5.1: Machine specifications, compiler and solver flags

$|S \cup P| - k$ exclusive hypervolume contributions, required for the calculation of ub'_1 and the problem of selecting the largest $k - |S|$ hypervolume contributions, required for the calculation of ub_2, can both be solved using selection algorithms. The standard library of C++ provides the *nth_element* selection algorithm, which solves the problem of selecting the largest n-th element in linear time on average. Selecting the largest $k - |S|$ or the smallest $|S \cup P| - k$ elements can also be solved using the same algorithm, since it leaves all smaller elements before the n-th element and the larger ones after. For the hypervolume calculation in $d > 3$ we used the algorithm proposed in Fonseca et al. (2006), which is available at `http://lopez-ibanez.eu/hypervolume`. To calculate the hypervolume contributions for $d = 3$ and to compute all exclusive contributions for $d \leq 4$ it is used the algorithms proposed in Guerreiro et al. (2015) and Guerreiro and Fonseca (2017), respectively. The latest version of both these algorithms can be obtained at `http://github.com/apguerreiro/HVC`. The B&B algorithm is available at `http://github.com/rgoomes/hssp`.

5.2 Computational Results

In Figure 5.2, we present the results for the first test with the running time in a logarithmic scale. The goal of this test was to compare both approaches and study the influence of k in the B&B algorithm. The results show that the B&B algorithm outperformed the GLPK solver in all tests. It is clear that the Cliff front is the easiest and the Linear front is the hardest for both approaches. The GLPK solver required at least 40 seconds, whereas the B&B algorithm was able to solve all instances under 1 second. The GLPK solver was somewhat very unstable for smaller values of k and reached several times the time limit of 1000 seconds. The B&B algorithm tends to spend more time when k is close to $n/2$ in the harder fronts. The detailed results of this test for some values of k can be seen in Appendix Tables A.1. A.2, A.3 and A.4.

In Figures 5.3 and 5.4, we present the results for the second and third tests with the running time and number of nodes in a logarithmic scale. The purpose of these tests was to study the influence of n in the B&B algorithm. Since for $k = n/2$ the number of possible subset combinations is the maximum, we can see how far the algorithm is able solve instances in reasonable time with the proposed machine configuration. In the results we show that the algorithm can to solve Cliff instances up to $n = 100$ rather quickly.

Regarding Convex fronts, the B&B algorithm was able solve instances up to $n = 90$, under 3 minutes on average. As for Concave and Linear fronts we were not able to test instances for $n > 80$. The detailed results of these tests can be seen in Appendix Tables A.5. A.6, A.7 and A.8. Since the number of nodes for the Cliff front with $n = 100$ was very low, we tried to see how far the B&B algorithm was able to solve Cliff instances. In this test we used the parallel version of the algorithm to achieve maximum performance. The B&B algorithm was able to solve Cliff instances up to $n = 200$. Figure 5.5 illustrates the average running time taken by the B&B algorithm for this test and the detailed results for some values of k can be seen in Appendix Table A.9.

We also tested some four-dimensional instances. In Figure 5.6, we present these results with the running time in a logarithmic scale. Clearly, the running times are an order of magnitude higher in comparison to the three-dimensional results for the same n. This is due to the increase of the time complexity in the calculation of the hypervolume contributions and exclusive hypervolume contributions in four dimensions. The detailed results of this test for some values of k can be seen in Appendix Tables A.10, A.11, A.12 and A.13.

Regarding the results obtained with the parallel version of the B&B algorithm, we can observe that for the tests that completed under 1 millisecond in the sequential version, worsened in the parallel version. This is expected in so small running times, due to thread creation, destruction and locking overhead. As n increases the parallel version starts to completely outperform the sequential version, reaching 1.8x speedup for large values of n. In the parallel version of the algorithm, even though two threads are working in parallel trying to find the best solution, the number of nodes remained fairly the same for most cases. The detailed results of this test can be seen in Appendix Tables A.14, A.15, A.16 and A.17.

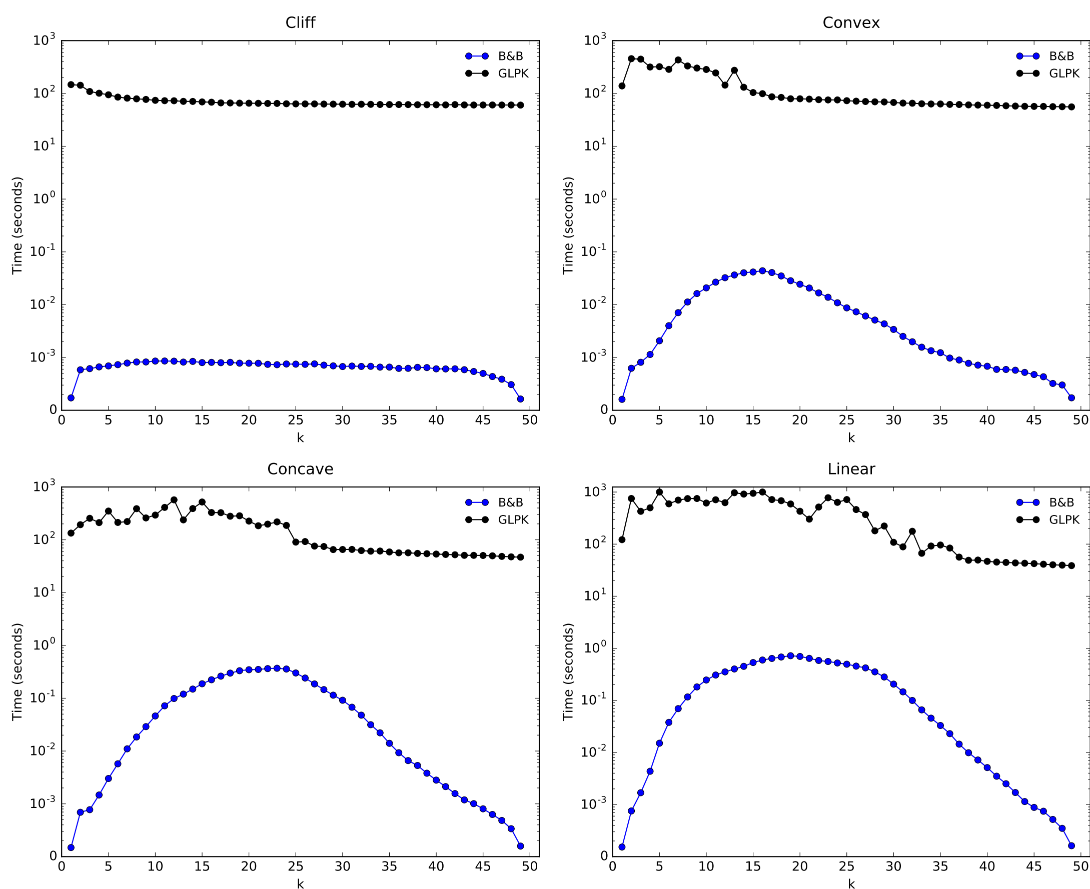

Figure 5.2: Average running time in seconds taken by both approaches for three-dimensional fronts with fixed $n = 50$

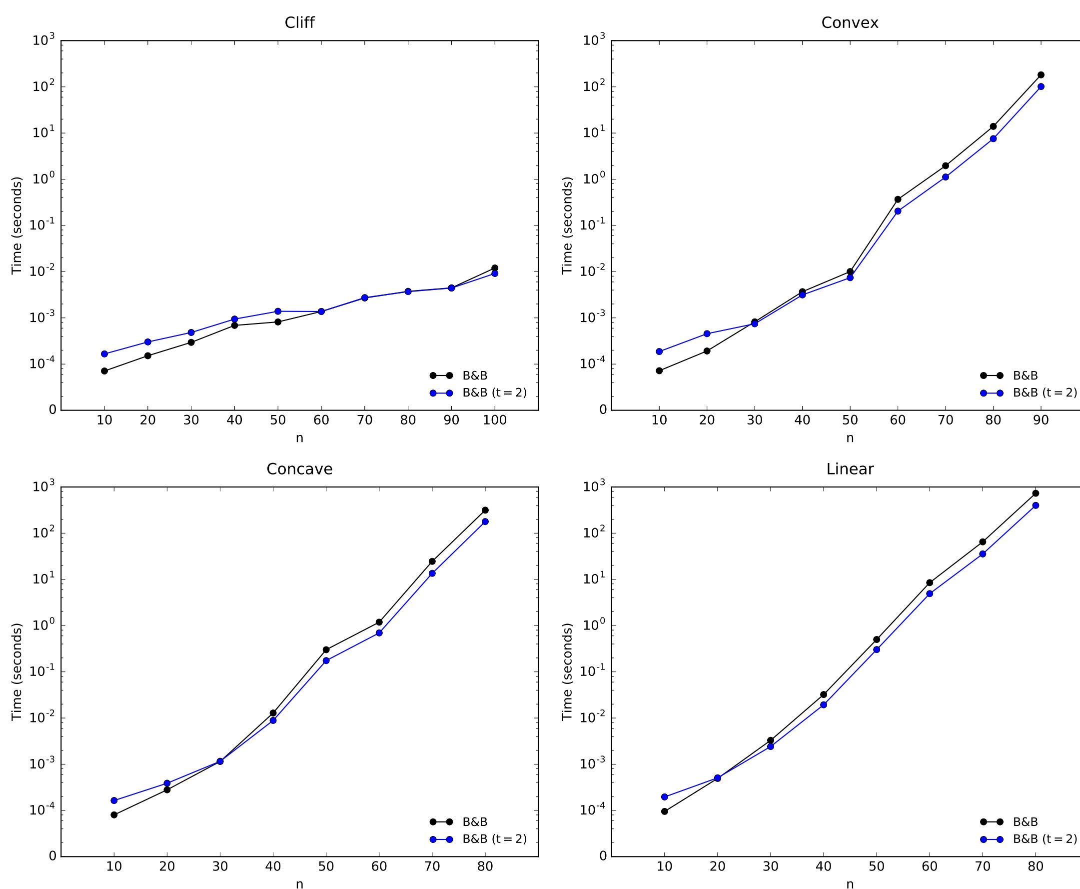

Figure 5.3: Average running time in seconds taken by both versions of the B&B algorithm for three-dimensional fronts with fixed $k = n/2$

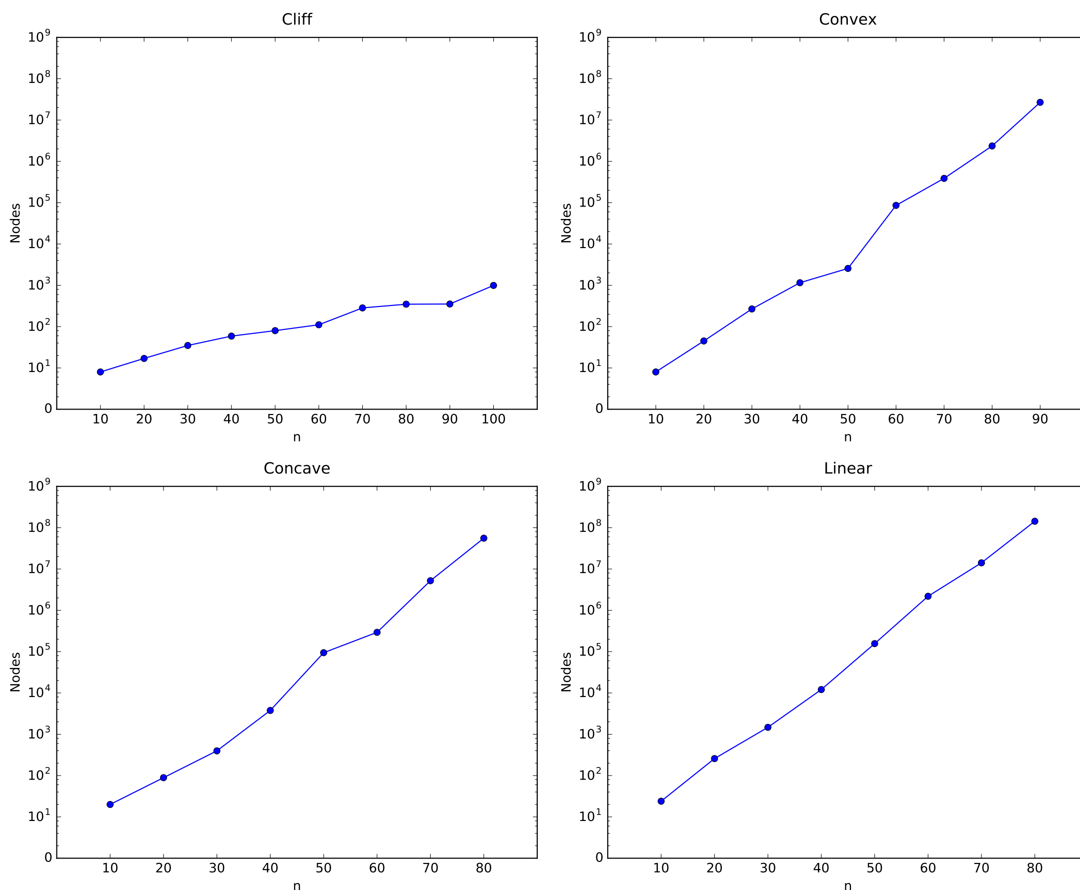

Figure 5.4: Average number of nodes taken by the B&B algorithm for three-dimensional fronts with fixed $k = n/2$

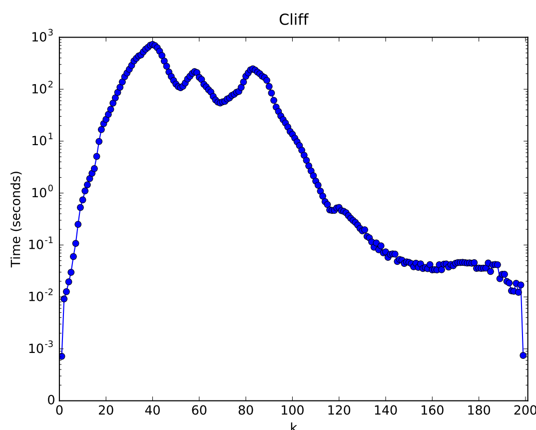

Figure 5.5: Average running time taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Cliff fronts with fixed $n = 200$

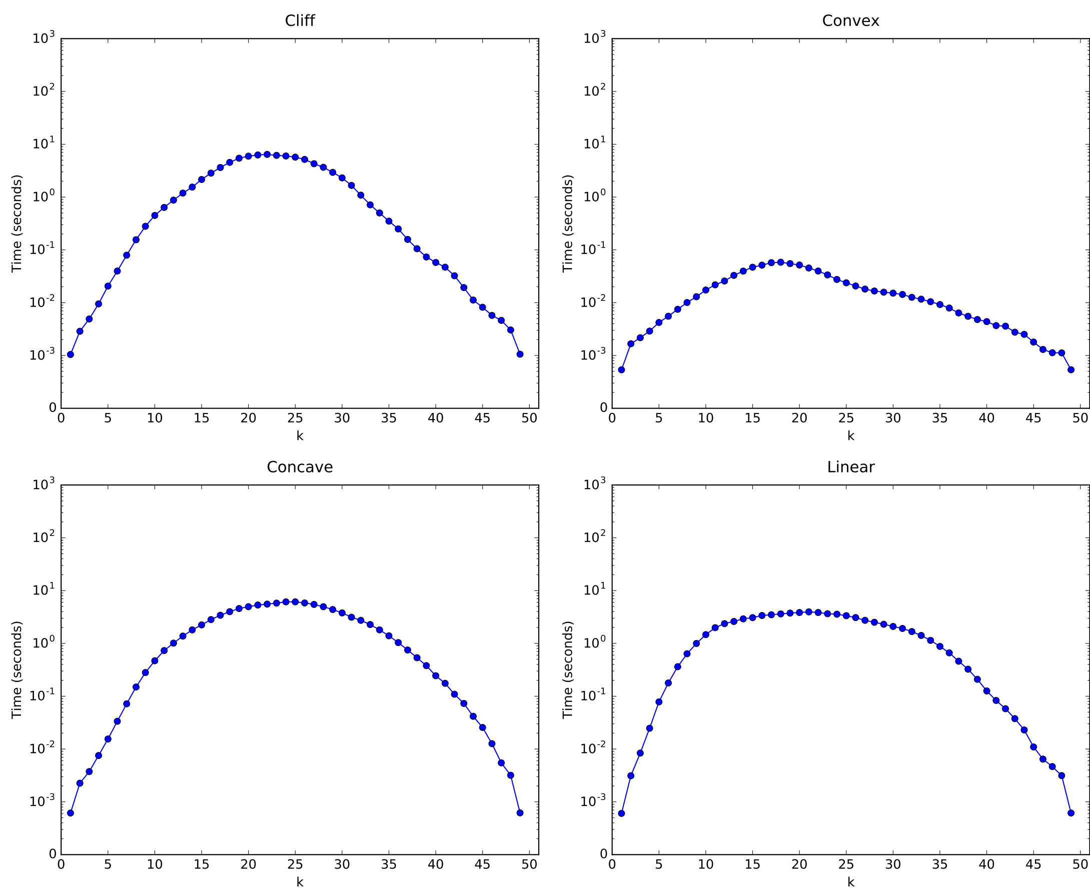

Figure 5.6: Average running time taken by the B&B algorithm for four-dimensional fronts with fixed $n = 50$

Chapter 6

Conclusion

In this thesis, we investigated how a B&B algorithm could be developed to solve the HSSP. We introduced several notions of bounds and a dynamic variable ordering, which uses the hypervolume contributions to heuristically choose points for the branching mechanism. Moreover, a parallel version of the B&B was proposed in order to take advantage of multiple processing units.

We performed an in-depth experimental analysis of our approach and compared with an IP solver. The experimental results show that B&B algorithms have some potential to solve the HSSP. One drawback of the IP approach is the inability of being able to solve three-dimensional instances with $n \geq 200$, due to both memory and time constraints in constructing the models. The B&B algorithm not only was able to solve some instances with $n = 200$, but also outperformed the GLPK solver in all tests. The parallel version of the B&B algorithm with $t = 2$ achieved an impressive speedup of 1.8x in some tests as compared to the sequential version.

6.1 Future Work

As future research directions for this work, it would be interesting to investigate other notions of upper bounds to further improve the performance of the B&B algorithm. A good starting point would be to extend the calculation of ub_4 for $d > 2$. Another possibility could be tighten the proposed upper bounds, similar to the idea of subtracting the smallest $|S \cup P| - k$ exclusive hypervolume contributions to ub_1. This can be accomplished in the calculation of ub_2: the idea is to find a set U of $k - |S|$ points in the unassigned point set, such that the hypervolume indicator of the union of included point set with the points of set U is larger than the hypervolume indicator of the points considered in ub_2. Then, the goal is to use the hypervolume contributions of the points of set U in the calculation of ub_2, instead of the largest $k - |S|$ hypervolume contributions of the unassigned point set. This would come as an extension in the calculation of ub_2. Still, there are two problems in this idea: it is possible that there is another set U' such that the hypervolume indicator

of the union of U' with the included point set is greater than the value of the *incumbent*, that is, there exists a new best solution in the current branch. In such cases, if we pruned with a valid set U, the new best solution would be lost. This can be fixed by generating the c greatest sums combinations of size $k - |S|$ of the hypervolume contributions. Then we iterate over the c sets in decreasing order of their sums, and if a best solution is found we stop. Larger values of c result in a more powerful extension, but on the other hand it is much more computationally expensive. This takes us to the second problem: it is very demanding to calculate c hypervolume indicators, let alone generating the c candidates.

It may be possible to improve the performance of the B&B algorithm, by exploring other types of variable ordering, possibly using heuristics methods. In other tests, not present in this thesis, it was observed that the dynamic variable ordering allowed the upper bounds to prune much earlier.

It would also be interesting to evaluate how the parallel version of the B&B algorithm scales with more CPU cores, since it supports $t > 2$.

Finally, it would be interesting to compare our approach to that of Bringmann et al. (2017), which was proposed only in June, 2017. To the best of our knowledge there is no code available at the time of the writing of this thesis.

Bibliography

Aigner, M. (1995). Turán's Graph Theorem. *The American Mathematical Monthly*, 102:808–816.

Bader, J. and Zitzler, E. (2011). HypE: An Algorithm for Fast Hypervolume-Based Many-Objective Optimization. *Evolutionary Computation*, 19(1):45–76.

Bader, J. M. (2009). *Hypervolume-Based Search for Multiobjective Optimization: Theory and Methods*. PhD thesis, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zürich, Zürich, Switzerland.

Basseur, M., Derbel, B., Goëffon, A., and Liefooghe, A. (2016). Experiments on Greedy and Local Search Heuristics for d–dimensional Hypervolume Subset Selection. In *Proceedings of the 2016 Genetic and Evolutionary Computation Conference (GECCO 2016)*, pages 541–548, Denver, Colorado, USA. ACM Press.

Bringmann, K., Cabello, S., and Emmerich, M. T. M. (2017). Maximum Volume Subset Selection for Anchored Boxes. In *33rd International Symposium on Computational Geometry (SoCG 2017)*, pages 22:1–22:15, Schloss Dagstuhl Leibniz-Zentrum für Informatik, Germany. Dagstuhl Publishing.

Bringmann, K. and Friedrich, T. (2008). Approximating the volume of unions and intersections of high-dimensional geometric objects. *Computational Geometry-Theory and Applications*, 43(6-7):601–610.

Bringmann, K. and Friedrich, T. (2010). An efficient algorithm for computing hypervolume contributions. *Evolutionary Computation*, 18(3):383–402.

Bringmann, K., Friedrich, T., and Klitzke, P. (2014). Two-dimensional Subset Selection for Hypervolume and Epsilon-Indicator. In *Proceedings of the 2014 Genetic and Evolutionary Computation Conference (GECCO 2014)*, pages 589–596, Vancouver, Canada. ACM Press.

Chan, T. M. (2013). Klee's Measure Problem Made Easy. In *Proc. 54th IEEE Symposium*

on Foundations of Computer Science (FOCS), pages 410–419, Los Alamitos, CA, USA. IEEE Computer Society.

Clausen, J. (1999). Branch and Bound Algorithms – Principles and Examples. Technical report, University of Copenhagen.

Emmerich, M. T. M. and Fonseca, C. M. (2011). Computing Hypervolume Contributions in Low Dimensions: Asymptotically Optimal Algorithm and Complexity Results. In *Proceedings of the 6th International Conference on Evolutionary Multi-criterion Optimization*, EMO'11, pages 121–135, Berlin, Heidelberg.

Figueira, J., Fonseca, C., Halffmann, P., Klamroth, K., Paquete, L., Ruzika, S., Schulze, B., Stiglmayr, M., and Willems, D. (2017). Easy to say they're hard, but hard to see they're easy - Toward a categorization of tractable multiobjective combinatorial optimization problems. *Journal of Multi-Criteria Decision Analysis*, pages 82–98.

Fonseca, C. M., Paquete, L., and López-Ibáñez, M. (2006). An Improved Dimension-Sweep Algorithm for the Hypervolume Indicator. In *Congress on Evolutionary Computation (CEC 2006)*, pages 1157–1163, Vancouver, BC, Canada. IEEE Press.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Guerreiro, A. P. and Fonseca, C. M. (2017). Computing and Updating Hypervolume Contributions in Up to Four Dimensions. *CISUC Technical Report TR-2017-001, University of Coimbra*.

Guerreiro, A. P., Fonseca, C. M., and Paquete, L. (2015). Greedy Hypervolume Subset Selection in the Three-Objective Case. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference (GECCO 2015)*, pages 671–678, Madrid, Spain. ACM Press.

Kuhn, T. (2015). *Representative Systems and Decision Support for Multicriteria Optimization Problems*. PhD thesis, University of Kaiserslautern, Germany.

Kuhn, T., Fonseca, C. M., Paquete, L., Ruzika, S., Duarte, M. M., and Figueira, J. R. (2016). Hypervolume Subset Selection in Two Dimensions: Formulations and Algorithms. *Evolutionary Computation*, 24(3):411–425.

Lacour, R., Klamroth, K., and Fonseca, C. M. (2017). A Box Decomposition Algorithm to Compute the Hypervolume Indicator. *Computers & Operations Research*, 79:347–360.

Musser, D. R. (1997). Introspective Sorting and Selection Algorithms. *Software–Practice and Experience*, 27(8):983–993.

Zitzler, E., Brockhoff, D., and Thiele, L. (2007). The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicator Via Weighted Integratio. In *Evolutionary Multi-Criterion Optimization (EMO 2007)*, volume 4403, pages 862–876, Japan.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2003). Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.

Appendix A

A.1 Results

k	B&B time (s)	GLPK time (s)
5	0.00069	94.33
10	0.00085	74.01
15	0.00080	69.12
20	0.00077	65.31
25	0.00074	63.43
30	0.00067	62.50
35	0.00065	61.61
40	0.00061	61.01
45	0.00049	60.47

Table A.1: Average running time in seconds taken by both approaches for three-dimensional Cliff fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$

k	B&B time (s)	GLPK time (s)
5	0.00207	321.34
10	0.02082	284.32
15	0.04157	103.95
20	0.02437	79.23
25	0.00870	72.90
30	0.00339	67.49
35	0.00123	63.16
40	0.00068	59.78
45	0.00047	56.82

Table A.2: Average running time in seconds taken by both approaches for three-dimensional Convex fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$

k	B&B time (s)	GLPK time (s)
5	0.00301	349.96
10	0.04612	293.56
15	0.18777	519.69
20	0.34609	226.12
25	0.30069	90.42
30	0.09152	65.48
35	0.01398	58.84
40	0.00281	53.62
45	0.00080	50.42

Table A.3: Average running time in seconds taken by both approaches for three-dimensional Concave fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$

k	B&B time (s)	GLPK time (s)
5	0.01505	> 1000.0
10	0.24624	614.61
15	0.53242	945.26
20	0.69758	427.04
25	0.49434	716.76
30	0.20516	108.24
35	0.03294	96.09
40	0.00512	46.69
45	0.00088	42.21

Table A.4: Average running time in seconds taken by both approaches for three-dimensional Linear fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$

n	nodes	time (s)
10	8	0.00007
20	17	0.00015
30	35	0.00029
40	59	0.00068
50	80	0.00081
60	111	0.00137
70	284	0.00269
80	349	0.00375
90	353	0.00446
100	994	0.01201

Table A.5: Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Cliff fronts with fixed $k = n/2$

n	nodes	time (s)
10	8	0.00007
20	45	0.00019
30	268	0.00082
40	1156	0.00368
50	2563	0.01005
60	86040	0.36744
70	387419	1.96258
80	2360963	13.9379
90	26847400	181.888

Table A.6: Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Convex fronts with fixed $k = n/2$

n	nodes	time (s)
10	20	0.00008
20	89	0.00027
30	397	0.00114
40	3776	0.01283
50	94274	0.30057
60	293394	1.18611
70	5203940	24.5828
80	55707157	314.770

Table A.7: Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Concave fronts with fixed $k = n/2$

n	nodes	time (s)
10	24	0.00009
20	258	0.00049
30	1473	0.00328
40	12086	0.03220
50	156484	0.50036
60	2189155	8.49631
70	14116063	64.9449
80	143227000	730.769

Table A.8: Average number of nodes and running time in seconds taken by the B&B algorithm for three-dimensional Linear fronts with fixed $k = n/2$

k	nodes	time (s)
10	104582	0.73983
20	3128224	26.2814
30	24175021	242.166
40	65490504	729.500
50	10104990	125.684
60	12152907	169.520
70	3631928	57.0669
80	10054723	177.803
90	6311427	113.053
100	705266	13.5499
110	86814	1.69998
120	23254	0.52865
130	8016	0.18734
140	2265	0.07376
150	889	0.04630
160	528	0.03323
170	332	0.04442
180	279	0.03595
190	234	0.02723

Table A.9: Average number of nodes and running time in seconds taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Cliff fronts with fixed $n = 200$ and $k \in \{10, 20, \ldots, n-10\}$

k	nodes	time (s)
5	754	0.02007
10	16135	0.46389
15	65037	2.20299
20	146097	6.01635
25	119339	5.73123
30	42872	2.31995
35	5891	0.35415
40	876	0.05835
45	99	0.00825

Table A.10: Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Cliff fronts with fixed $n = 50$ and $k = \{5, 10, \ldots, n-5\}$

k	nodes	time (s)
5	124	0.00421
10	628	0.01736
15	1583	0.04690
20	1545	0.05166
25	665	0.02377
30	388	0.01516
35	217	0.00921
40	89	0.00437
45	25	0.00179

Table A.11: Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Convex fronts with fixed $n = 50$ and $k = \{5, 10, \dots, n-5\}$

k	nodes	time (s)
5	658	0.01548
10	18586	0.46935
15	72488	2.24225
20	135327	4.95303
25	145804	6.12419
30	81215	3.78192
35	27360	1.39401
40	4508	0.24423
45	424	0.02560

Table A.12: Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Concave fronts with fixed $n = 50$ and $k = \{5, 10, \dots, n-5\}$

k	nodes	time (s)
5	5524	0.07806
10	70727	1.46563
15	111302	3.07715
20	116191	3.84518
25	86786	3.34785
30	47716	2.09330
35	17794	0.87711
40	2297	0.12578
45	177	0.01094

Table A.13: Average number of nodes and running time in seconds taken by the B&B algorithm for four-dimensional Linear fronts with fixed $n = 50$ and $k = \{5, 10, \dots, n-5\}$

n	nodes	time (s)	speedup	nodes factor
10	8	0.00016	0.427	1.0
20	17	0.00030	0.503	1.0
30	35	0.00048	0.611	1.0
40	60	0.00094	0.73	0.983
50	80	0.00138	0.588	1.0
60	111	0.00137	1.002	1.0
70	286	0.00274	0.983	0.993
80	349	0.00370	1.013	1.0
90	353	0.00442	1.009	1.0
100	994	0.00913	1.315	1.0

Table A.14: Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Cliff fronts with fixed $k = n/2$

n	nodes	time (s)	speedup	nodes factor
10	8	0.00018	0.384	1.0
20	45	0.00045	0.422	1.0
30	268	0.00074	1.102	1.0
40	1155	0.00314	1.173	1.001
50	2547	0.00741	1.356	1.006
60	85920	0.20422	1.799	1.001
70	388647	1.11846	1.755	0.997
80	2370824	7.55555	1.845	0.996
90	27182815	101.299	1.796	0.988

Table A.15: Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Convex fronts with fixed $k = n/2$

n	nodes	time (s)	speedup	nodes factor
10	20	0.00016	0.489	1.0
20	89	0.00038	0.72	1.0
30	397	0.00115	0.992	1.0
40	3790	0.00886	1.449	0.996
50	93641	0.17470	1.72	1.007
60	297921	0.69490	1.707	0.985
70	5169548	13.5292	1.817	1.007
80	55817416	178.154	1.767	0.998

Table A.16: Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Concave fronts with fixed $k = n/2$

n	nodes	time (s)	speedup	nodes factor
10	24	0.00019	0.485	1.0
20	257	0.00050	0.971	1.004
30	1475	0.00241	1.36	0.999
40	11817	0.01932	1.667	1.023
50	156151	0.30284	1.652	1.002
60	2160777	4.91369	1.729	1.013
70	14103181	35.5611	1.826	1.001
80	143009494	399.534	1.829	1.002

Table A.17: Average number of nodes, running time in seconds, speedup and nodes factor taken by the parallel version of the B&B algorithm with $t = 2$ for three-dimensional Linear fronts with fixed $k = n/2$