I N A U G U R A L – D I S S E R T A T I O N

zur

Erlangung der Doktorwürde

der

Naturwissenschaftlich–Mathematischen Gesamtfakultät

der

R U P R E C H T – K A R L S – U N I V E R S I T Ä T

H E I D E L B E R G

vorgelegt von

Tan Khoa Vo, M.Eng.

aus Ninh Thuan, Vietnam

Tag der mündlichen Prüfung

5. Oktober 2011

Exact and Heuristic Solutions to the Bandwidth Minimization Problem

Gutachter

Prof. Dr. Gerhard Reinelt

Prof. Dr. Klaus Ambos-Spies

Abstract

The *bandwidth minimization problem* is a classical combinatorial optimization problem studied since about 1960. It is formulated as follows. Given a connected graph $G = (V, E)$ with n vertices, the task is to find a permutation π of the vertices (also called a labeling), i.e., a bijection between V and $\{1, 2, \ldots, n\}$, such that the maximum difference $|\pi(u) - \pi(v)|$, for $uv \in E$, is minimized. This problem is $\mathcal{NP}$-hard, even for binary trees. Applications of the bandwidth problem can be found in many areas: solving systems of linear equations, data storage, electronic circuit design, and recently in compression of topological information from digital road networks.

In this dissertation we report our contributions of both heuristic and exact methods for the bandwidth problem. On the heuristic side, we start by modifying a heuristic method which exploits properties of the graph. Next we propose an approximate objective function for the bandwidth problem. It is very sensitive to alterations in a permutation and can thus be used efficiently in global optimization heuristic methods. A simulated annealing method using the approximate objective function is reported. We also present an application of the bandwidth problem to the compression of topological information of digital road networks.

For exact methods, which are our main focus, we formulate the concept of a *partial permutation*, i.e., a bijection between $S \subset V$ and $L \subset \{1, 2, \ldots, n\}$. Based on this concept, we introduce new constraints for the bandwidth problem and apply them efficiently in a branch-and-bound algorithm. We analyze the relation between certain partial permutations and show that some partial permutations are *dominated* by others. Therefore, they can be eliminated in the branch-and-bound tree and this reduces the search space and running time. Furthermore, we enhance the use of partial permutations in branch-and-bound algorithms with a *2-labeling* scheme, supported by the *dominance* rule. Instead of extending the partial permutation one-by-one, our scheme uses two vertices simultaneously.

We evaluate our algorithms on a popular benchmark suite which comprises 113 instances with less than 1,000 vertices each. In many cases our work improves on the best known lower bound in the literature. Moreover, our exact algorithms are capable of computing lower bounds for much larger instances. We perform computational experiments on a second suite of 36 instances with more than 1,000 vertices each, whose best known lower bound so far is the generic theoretical one. We can improve this bound for some instances in this suite, the largest such instance having about 15,600 vertices. Finally, we parallelize our branch-and-bound algorithms and run the solver on a parallel cluster with 256 processors, improving the lower bound for some instances in the first benchmark suite even further.

Zusammenfassung

Das *Bandwidth Minimization Problem* ist ein klassisches kombinatorisches Optimierungsproblem, das bereits seit etwa 1960 untersucht wird: Zu einem gegebenen zusammenhängenden Graphen $G = (V, E)$ der Ordnung n ist eine Permutation π der Knoten (auch *Labeling* genannt) gesucht, d.h. eine Bijektion von V auf die Menge $\{1, 2, \ldots, n\}$, für welche die maximale Differenz $|\pi(u) - \pi(v)|$ über alle $uv \in E$ minimal ist. Das Problem ist bereits auf Binärbäumen $\mathcal{NP}$-schwer. Anwendungen des Problems finden sich beispielsweise beim Lösen großer linearer Gleichungssysteme, in der Datenspeicherung, beim Entwurf von Schaltkreisen und jüngst auch bei der Komprimierung topologischer Daten von digitalen Straßennetzen.

In der vorliegenden Arbeit stellen wir unsere Beiträge zur näherungsweisen sowie zur exakten Lösung des Problems vor. Zunächst verbessern wir durch geeignete Modifikationen eine bestehende Heuristik, die sich Eigenschaften des Graphen zunutze macht. Anschließend schlagen wir eine approximative Zielfunktion vor, welche sensibler auf Veränderungen in den Permutationen reagiert und sich dadurch vor allem für globale Heuristiken eignet. Wir beschreiben einen Simulated-Annealing Algorithmus, der diese Approximation verwendet. Wir präsentieren ferner eine Anwendung des Problems auf die Komprimierung topologischer Daten von digitalen Straßennetzen.

Unser Hauptaugenmerk liegt jedoch auf exakten Lösungsverfahren. Für diese führen wir den Begriff der *Teilpermutation* ein; dies ist eine Bijektion zwischen $S \subset V$ und $L \subset \{1, 2, \ldots, n\}$. Darauf basierend entwickeln wir neue gültige Nebenbedingungen, welche wir effizient in einen Branch-and-Bound Algorithmus einbinden. Wir analysieren die Beziehung bestimmter Teilpermutationen zueinander und beweisen, dass einige Teilpermutationen durch andere *dominiert* werden. Dominierte Teilpermutationen können dabei im Branch-and-Bound-Suchbaum vernachlässigt werden, wodurch sich die Laufzeit reduzieren lässt. Des Weiteren präsentieren wir mit dem sogenannten *2-Labeling-Schema* eine verbesserte Methode zur Verwendung von Teilpermutationen in Branch-and-Bound Algorithmen. Hierbei nutzen wir die obige Dominanzbeziehung und vergrößern die Teilpermutation in jedem Iterationsschritt um zwei statt nur um einen Knoten.

Wir testen unsere Algorithmen mit einer gängigen Sammlung von 113 Benchmark-Instanzen mit jeweils weniger als 1000 Knoten. In vielen Fällen können wir hierbei die besten bekannten unteren Schranken aus der Literatur verbessern. Darüberhinaus können wir mit unseren Methoden untere Schranken für weitaus größere Instanzen bestimmen. Wir belegen dies anhand von Rechenexperimenten auf einer zweiten Benchmark-Sammlung. Diese umfasst 36 Instanzen mit jeweils mehr als 1000 Knoten. Die beste bekannte untere Schranke jeder dieser Instanzen ist durch eine allgemeine Abschätzungsformel gegeben. Die größte Instanz, für welche wir eine

erstmalige Verbesserung dieser unteren Schranke erzielen können, besitzt 15600 Knoten. Schließlich parallelisieren wir unseren Branch-and-Bound Algorithmus und testen ihn auf einem Cluster mit 256 Prozessoren. Hierdurch können wir die unteren Schranken für einige Instanzen mit weniger als 1000 Knoten noch weiter verbessern.

Contents

1 Introduction 1

1.1 Contributions 2

1.2 Outline 3

1.3 Acknowledgments 4

2 Preliminaries 5

2.1 Graph theory 5

2.2 Complexity theory 6

2.3 Combinatorial optimization 7

2.4 Parallel computing 8

3 The Bandwidth Minimization Problem 11

3.1 Literature review 11

3.2 Lower bounds 16

3.3 The canonical IP formulation 17

3.4 Partial permutation 17

3.5 Branch-and-bound approaches 25

4 Heuristics and applications 32

4.1 The iGPS heuristic 32

4.2 An approximate objective function 35

4.3 A simulated annealing method 38

4.4 Compression of topological information 40

5 Exact methods 44

5.1 The new constraints 44

5.2 The dominance relation 54

5.3 The 2-labeling scheme 61

6 Implementation and Parallelization 68

6.1 A solver for the bandwidth problem 68

6.2 The parallel solver 72

7 Computational results 77

7.1 Introduction 77

7.2 Performance evaluation 81

7.3 The lower bounds 86

7.4 Parallel computation 95

7.5 The upper bounds 103

8 Conclusions and Discussion 108

A Detailed computational results 111

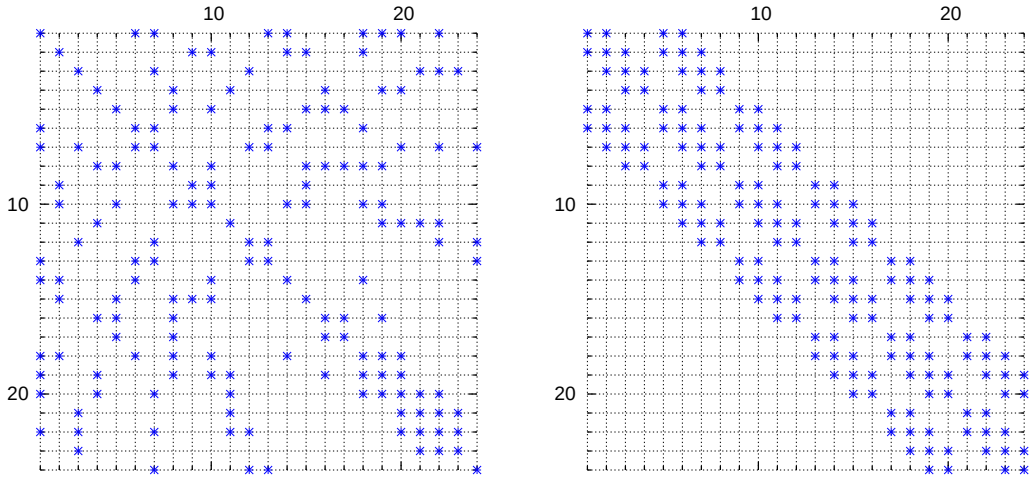
Symbols and Notation 128

References 129

Index 135

1 Introduction

The *bandwidth minimization problem* is a classical combinatorial optimization problem which has been studied since about 1960. It is formulated as follows. Given a connected graph $G = (V, E)$ with n vertices, the task is to find a permutation π of the vertices (also called a labeling), i.e., a bijection between V and $\{1, 2, \ldots, n\}$, such that maximum difference $|\pi(u) - \pi(v)|$, for $uv \in E$, is minimized. The name itself originates from an equivalent matrix problem. Given an (n, n)-matrix M the bandwidth problem consists of finding a simultaneous permutation of the rows and columns of M such that the maximum distance of nonzero elements to the main diagonal is as small as possible, i.e., to obtain a permuted matrix of minimum bandwidth. Figure 1.1 shows an example with the original matrix[1] on the left and the permuted matrix of optimal bandwidth on the right. The equivalence between the two formulations can be observed when forming the adjacency matrix of the graph. Then the minimum bandwidth of this matrix corresponds to an optimal permutation of the vertices of the graph.



(a) A sparse matrix (b) A matrix of minimum bandwidth

Figure 1.1: Matrix bandwidth minimization.

Applications of the bandwidth problem can be found in many areas. The primary application is in solving systems of linear equations [7], because the running time for

[1] The matrix was introduced in [23].

Gaussian elimination reduces from $O(n^3)$ to $O(nb^2)$ if the matrix is permuted to have bandwidth b. Since matrices in practice are often very sparse, bandwidth minimization can lead to substantial running time reduction. Further applications [32] [13] include storage of data, electronic circuit design, and information retrieval. Recently, we have applied the bandwidth problem to the compression of topological information of digital road networks [54].

The problem is $\mathcal{NP}$-hard [42], surprisingly even for binary trees [20]. In addition, the popular branch-and-cut method, which is based on a direct integer linear formulation and its relaxation (the original integer formulation whose integrality constraints are removed), seems not to be useful with the bandwidth minimization problem. This makes the problem even harder to solve.

Because of its important applications, the bandwidth problem has been the subject of extensive research. The reported approaches usually divide into two categories: exact and heuristic methods. *Exact methods* aim to improve the lower bound and find the provably optimal solution. In contrast, *heuristic methods* try to find a good solution (upper bound) in a reasonable time, without being able to guarantee its optimality. There is a large body of literature on heuristic methods for the problem, but only a few exact algorithms are known. Methods for solving the bandwidth problem are often evaluated on a popular benchmark suite which comprises 113 instances with less than 1,000 vertices each. For many instances in this suite, the gap between the best known lower bound and upper bound is still huge.

As we shall review in Chapter 3, recent heuristic methods have reported upper bounds of nearly equivalent solution quality for instances in the popular suite. It seems that the computation of upper bounds has reached a mature state, but this could not be proven so far due to the lack of good lower bounds. Therefore, powerful methods for computing lower bounds, in particular for large instances, are needed. This is the main motivation of our research.

1.1 Contributions

In this dissertation we report on our contributions of both heuristic and exact methods for the bandwidth problem. On the heuristic side, we start by modifying a heuristic method which exploits properties of the graph. Next we propose an approximate objective function for the bandwidth problem, which is very sensitive to alterations in a permutation and can thus be used efficiently in global optimization heuristic methods. A simulated annealing method using the approximate objective function is reported. We also present an application of the bandwidth problem to the compression of topological information of digital road networks.

For exact methods, which are our main focus, we formulate the concept of a *partial*

permutation, i.e., a bijection between $S \subset V$ and $L \subset \{1, 2, \ldots, n\}$. Based on this concept, we introduce new constraints for the bandwidth problem and apply them efficiently in a branch-and-bound algorithm. We analyze the relation between certain partial permutations and show that some partial permutations are *dominated* by others. Therefore, they can be eliminated in the branch-and-bound search tree and this reduces the running time. Furthermore, we enhance the use of partial permutations in branch-and-bound algorithms with a *2-labeling* scheme, supported by the *dominance* rule. Instead of extending the partial permutation one-by-one, our scheme uses two vertices simultaneously.

We evaluate our algorithms on 113 instances from the popular benchmark suite. Our work can solve more instances to optimality, and in many cases obtains a better lower bound, compared with the best known results in the literature. Moreover, our exact algorithms are capable of computing the lower bound for much larger instances. We perform computational experiments on a second suite of 36 instances with more than 1,000 vertices each. Here, the best known lower bound so far is the generic theoretical one. We can improve this bound for some instances in this suite, the largest such instance having about 15,600 vertices.

Finally, we parallelize our branch-and-bound algorithms and run the solver on a parallel cluster [26] with 256 processors. The same parallel framework as in our previous work [55] is used, but the new solver is more efficient due to the introduction of new constraints and the dominance rule. The lower bounds for some instances in the first benchmark suite are improved even further.

1.2 Outline

The dissertation is organized as follows. Chapter 1 gives a high level overview. Chapter 2 summarizes the basic concepts used throughout this dissertation from graph theory, complexity theory, combinatorial optimization, and parallel computing. Chapter 3 reviews previous methods which have made improvements to the problem over time. We introduce some new definitions and describe the previous exact algorithms in detail, so that they can be used as a basis for our own later improvements.

In Chapter 4, we report the improvement of a heuristic method which exploits properties of the graph. An approximate objective function named *Sigma* is introduced, followed by a description of a simulated annealing method which applies the *Sigma* function. The chapter is finished with an application of the bandwidth problem to the compression of topological information of digital road networks.

Our contributions to exact methods are introduced in Chapter 5. Here the new constraints for the bandwidth problem are presented, followed by the analysis of the

dominance relation between certain partial permutations. We also present a new branching scheme named *2-labeling*. The implementation of the branch-and-bound algorithm and its parallelization are described in Chapter 6.

We report our computational results in Chapter 7 and close the dissertation with our conclusions and discussion in the last chapter. The full computational results are given in Appendix A.

1.3 Acknowledgments

This dissertation has been completed with contributions and support, directly or indirectly, from many people. I would like to take this opportunity to express my gratitude to them.

First of all, I would like to thank my supervisor Prof. Gerhard Reinelt for the direction and support during my doctoral work. I appreciate his straight and useful comments to the dissertation.

I acknowledge my colleague Marcus Oswald for initiating the idea of dominant subproblems and many fruitful discussions in the beginning phase of my doctoral work. I thank Prof. Christoph Schnörr for his lectures which inspired me to develop the approximate objective function Sigma. Thanks to Martin Pfeifle for the interesting collaboration project on the compression of digital road networks. I am also grateful to the work of Prof. Alberto Caprara and Prof. Salazar-González which provides a good basis for approaching the bandwidth problem.

I appreciate colleagues who helped proofread parts of this dissertation and provided useful comments: Cara Cocking, Thorsten Bonato, and Stefan Wiesberg. Thanks to Christian Kirches for a nice dissertation template.

I acknowledge many former and current members of the research group Discrete Optimization, the Institute of Computer Science, and the IWR Heidelberg for a friendly working environment as well as pleasant social events. Thanks to the administrative support from Georgios Nikolis, Catherine Proux-Wieland, and Karin Tenschert in the research group, and Stefan Friedel from IWR Helics.

I would also like to thank Dr. Stefan Reineck for the encouragement during my doctoral work and many practical lessons.

Finally, thanks to family members and relatives for encouraging and keeping me in touch with home: Luong Thi Trinh, Vo Tan Khanh, Vo Thi Luong Tran, Vo Tan Khoi, Vo Tan Khue, and Le Trinh Thong.

2 Preliminaries

In this chapter we review the basic concepts that will be used throughout this dissertation. In particular, they are from graph theory, complexity theory, combinatorial optimization, and parallel computing. Concepts with a limited scope will be introduced later at the appropriate place.

2.1 Graph theory

An *undirected graph* $G = (V, E)$ consists of a finite set V of *vertices* and a finite set E of edges, each edge being an unordered pair of distinct vertices. A vertex v is *incident* with an edge e if $v \in e$. Two vertices incident with an edge are its *endvertices*. An edge $\{v, w\}$ is written as vw, and $vw = wv$. In this case v and w are said to be *adjacent* and are called *neighbors*. The *adjacency matrix* $A = (a_{ij})_{|V|,|V|}$ is defined by setting $a_{ij} = 1$ if vertex i is adjacent to vertex j and 0 otherwise.

A *path* is a set of edges $\{v_0v_1, v_1v_2, \ldots, v_{k-1}v_k\}$ where the v_i are all distinct for $i < k$, and is denoted $P = v_0 \ldots v_k$. The number of edges of a path is called its *length*. The *distance* in G of two vertices v and w, denoted $d(v, w)$, is the length of the shortest path between v and w in G. A graph is said to be *connected* if there is a path between every pair of its vertices. The greatest distance between any two vertices in G is the *diameter* of G, denoted by $d(G)$.

If $P = v_0 \ldots v_k$ is a path and $v_0 = v_k$, then P is called a *cycle*. A cycle in a graph G visiting all vertices in G is called a *Hamiltonian cycle*. A *forest* is an edge set in a graph which does not contain any cycle. A connected forest containing all vertices is called a *tree*.

We denote by $N_k(v)$, $k \leqslant d(G)$, the set of vertices at distance at most k from v, not including v. In line with that definition, $N_1(v)$ is the set of neighbors of v. The *degree* of a vertex is the number of its neighbors, denoted by $|N_1(v)|$. Given a set $F \subset V$, $N_1(F) = \bigcup_{v \in F} N_1(v)$.

Two edges with the same endvertices are called *parallel*. Graphs without parallel edges are called *simple*. In the scope of this dissertation, we always refer to undirected, simple, and connected graphs, unless otherwise stated.

Further subjects in graph theory can be found in Diestel [14].

2.2 Complexity theory

Complexity theory allows us to know the relative efficiency of an algorithm as well as the difficulty of a problem. We only review concepts which will be used in our work. A comprehensive guide to complexity theory can be found in Garey and Johnson [21].

2.2.1 Running time of algorithms

The *running time* is defined by the number of elementary steps an algorithm needs to solve a problem, usually depending on the size of the input to the algorithm. Because the running time might be different for several instances of the same input size, we are interested in the worst-case measure. The *time complexity function* for an algorithm is defined by giving, for each possible input size, the maximum running time required by the algorithm to solve a problem instance of that size.

A function $r(n)$ is said to be $O(f(n))$ if there exists a constant $c > 0$ such that $r(n) \leqslant c \times f(n)$ for all values of $n \geqslant 0$. We also say the running time of an algorithm is $O(f(n))$ if its time complexity function is $O(f(n))$. A *polynomial time algorithm* is one whose time complexity function can be bounded by a polynomial, e.g., $O(n \log(n))$, $O(n^2)$. Any algorithm whose time complexity function cannot be bounded by a polynomial, such as $O(2^n)$, is called an *exponential time algorithm*. The difference in running time between a polynomial time algorithm and an exponential time algorithm becomes huge when the problem size is large, for example a few seconds versus years.

2.2.2 Complexity classes

Classes of problems in complexity theory are based on decision problems. A *decision problem* has only two possible solutions, either "yes" or "no". For example, the question whether a graph $G = (V, E)$ has a Hamiltonian cycle is a decision problem.

The complexity class $\mathcal{P}$ consists of decision problems which can be solved by a polynomial time algorithm. A decision problem is said to be in class $\mathcal{NP}$ if the solution of a yes instance can be verified by a polynomial time algorithm. Referring to the question whether a graph has a Hamiltonian cycle again, whenever the answer to a problem instance is "yes", a set of edges can be produced by the algorithm and the statement that it is a Hamiltonian cycle can be verified in polynomial time.

Since a polynomial time algorithm for any decision problem in $\mathcal{P}$ can be used for the verification, $\mathcal{P} \subseteq \mathcal{NP}$. It is suspected that $\mathcal{P} \neq \mathcal{NP}$, but whether this is true is not known.

Assuming that $\mathcal{P} \neq \mathcal{NP}$, there are some problems in $\mathcal{NP}$ but not in $\mathcal{P}$, and they are the "harder" problems. The satisfiability problem is the "hardest" problem in $\mathcal{NP}$, as proved by Cook [8]. Some other problems have been shown to be equivalently "hard" as the satisfiability problem, and these "hardest" problems belong to the class of $\mathcal{NP}$-*complete* problems. If an $\mathcal{NP}$-complete problem can be solved in polynomial time, so can all the problems in $\mathcal{NP}$. If some problem in $\mathcal{NP}$ cannot be solved in polynomial time, then neither can all $\mathcal{NP}$-complete problems. We can also say that if a decision problem π is $\mathcal{NP}$-complete, then $\pi \in \mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.

There are other types of problems in complexity theory. An *optimization problem* aims at finding among possible solutions the best solution with respect to some specified sense. A decision problem can be derived from the optimization problem. For example, if the optimization problem asks for a structure of minimum "cost", it can be associated with a decision problem asking whether there exists a structure whose cost is no more than a certain value.

A *search problem* can be considered as a more general type of a decision problem. Given a problem instance I, an algorithm for solving this search problem returns some solution belonging to the solution set of I if this is a yes instance and otherwise returns "no". Any decision problem can be formulated as a search problem by defining the solution set for each yes instance of the search problem with only "yes".

The complexity class $\mathcal{NP}$-*hard* refers to optimization problems and search problems whose corresponding decision problem is $\mathcal{NP}$-complete. These problems are hard to be solved. The bandwidth minimization problem is $\mathcal{NP}$-hard [42] [20].

2.3 Combinatorial optimization

2.3.1 Integer Programming

An *Integer Programming problem* (IP) can be stated as follows. Given a matrix $A \in \mathbb{Z}^{m \times n}$, a vector $b \in \mathbb{Z}^m$, and a vector $c \in \mathbb{Z}^n$, find a vector $x \in \mathbb{Z}^n$ such that $Ax \leqslant b$ and $c^T x$ is minimum. A short form of an IP is $\min\{c^T x : Ax \leqslant b, x \in \mathbb{Z}^n\}$.

The linear function $c^T x$ is called the *objective function* of the IP. A *feasible solution* of an IP is a vector $x \in \mathbb{Z}^n$ with $Ax \leqslant b$. A feasible solution x^* is called an *optimal solution* if $c^T x^* \leqslant c^T x$ for all feasible solution x. Integer programming is $\mathcal{NP}$-hard.

If the integrality constraints of an integer program are removed, the IP becomes a *Linear Programming problem* (LP), the so-called *LP relaxation* of the IP. Linear programming problems can be solved efficiently in practice using methods such as the simplex algorithm [10]. As a result, LP relaxation is used in many techniques for solving IP problems, among them *branch-and-cut* [28] is a powerful method. How-

ever, using LP relaxation does not seem to be useful with the known IP formulations of the bandwidth minimization problem, as will be explained later. Therefore, we do not discuss further these topics, but rather focus on the branch-and-bound method.

2.3.2 Branch-and-bound

Branch-and-bound is an exact solution technique for solving integer problems. It uses a divide-and-conquer strategy to partition the original problem into smaller problems, the so-called *subproblems*, and then recursively solves each subproblem. The hierarchy of the problems looks like a tree, the so-called *branch-and-bound tree*. In this tree, the *root vertex* is associated with the original problem and each vertex is associated with a subproblem.

A branch-and-bound algorithm requires two basic procedures. The first one is the partitioning of a problem into smaller subproblems, called *branching*, and then adding them to the list of active subproblems, the so-called *subproblem queue*. The second procedure is to compute the bounds of each subproblem, which is called *bounding*. A branch-and-bound algorithm is illustrated in Algorithm 2.1, we consider a minimization problem for explaining the bounding procedure.

During execution, the algorithm maintains the best feasible solution found, which provides a *global upper bound* on the original problem. While the subproblem queue is not empty, a subproblem is selected from it for processing. The selection method is called the *search strategy*. For each processed subproblem X, if we can prove that X has no feasible solution at all or that its feasible solution cannot be better than the best feasible solution, then there is no need to continue and subdivide X. In that case we say X is *fathomed*. If the feasible solution of the subproblem is better than the best feasible solution, the global upper bound and the corresponding best feasible solution are updated. If a subproblem X cannot be fathomed, it must be split into smaller subproblems $X_1, \ldots, X_q$ such that $\bigcup_{i=1,\ldots,q} X_i = X$. These subproblems $X_1, \ldots, X_q$ are then added to the queue.

The branch-and-bound algorithm terminates when there is no subproblem left in the queue. The current best feasible solution is the optimal solution.

Further topics in combinatorial optimization can be found in the comprehensive textbook of Korte and Vygen [31].

2.4 Parallel computing

In parallel computing, programs are executed on parallel computers. A *parallel computer* consists of processors that are able to work together to solve a problem.

Algorithm 2.1: A general branch-and-bound algorithm.

Initialize the subproblem queue with the original problem.
while the subproblem queue is not empty **do**
Select a subproblem X and remove it from the queue. Process X and determine:
(1) If X has no solution then it is fathomed.
(2) If the lower bound of X is greater than the global lower bound then X is fathomed.
(3) If X has a feasible solution better than the current best solution then the global upper bound is updated with the new feasible solution and X is fathomed.
(4) If none of the three cases above applies, X is split into smaller subproblems which are then added to the queue.
end while
The best feasible solution is the optimal solution.

Parallel computers can be characterized by the communication mechanism between processes. In the *shared-memory* model, processors share a common memory and the communication between processes is performed via shared variables. In the *message-passing* model, processes (usually on different processors) communicate with each other by sending and receiving messages.

A *cluster* is a parallel computer consisting of *computer nodes* that are physically interconnected via a network. Each computer node can be either a personal computer (PC) or a shared-memory parallel computer. A node may have one or more processors. Processes in a cluster communicate with the others using message-passing protocols.

Many message-passing protocols have been developed, two among the most popular are MPI [24] [38] and PVM [22]. MPICH [39] is a good implementation of MPI and is freely available.

In this dissertation, we limit *one processor to one process*, unless otherwise stated.

The goal of a *parallel system*, including the parallel computer and the parallel algorithm, is to achieve higher computing power. Therefore one would be interested in knowing how stronger it is compared with the single-processor system. The *speedup* factor is such a metric.

Definition 2.1
Speedup is defined as:

$$S_p = \frac{t_s}{t_p}, \tag{2.1}$$

where t_s is the execution time by the single-processor system and t_p is the execution time by the parallel system having p processors. △

Another important metric to evaluate the performance of a parallel system is *scalability*, which tells whether the system performance is still "good" at larger scales (or numbers of used processors). The system's scalability is relatively expressed by the *efficiency* factor.

Definition 2.2
Efficiency is defined as:

$$E_p = \frac{S_p}{p}, \tag{2.2}$$

where S_p is the speedup of the parallel system having p processors. △

A parallel algorithm is called *scalable* if it can maintain well the performance of the parallel system at different scales and sizes of problem instances.

Further details in parallel computing and parallel systems can be found in Hwang and Xu [27]. Another introduction to parallel computing is given in Foster [19].

3 The Bandwidth Minimization Problem

We begin the chapter with a review of previous approaches which have made improvements to the bandwidth problem over time. The canonical IP formulation of the bandwidth minimization problem and the known lower bounds are presented. We then introduce the so-called *partial permutation* concept. Previous exact algorithms are described in detail based on this concept so that they can be used as a basis for our own later improvements.

In the remaining chapters of this dissertation, we always assume the graph of interest is an undirected graph $G = (V, E)$, $|V| = n$ and $|E| = m$.

3.1 Literature review

The developed algorithms for the bandwidth problem usually divide into two groups: heuristic and exact methods. We begin with the heuristic approaches. Since there is a large body of literature on the heuristic side, in this section we refer to typical methods only. On the other hand, we will review all known exact methods before going into detail in the next section.

3.1.1 Heuristic methods

One of the first published methods is the *reverse CM algorithm* formulated by Cuthill and McKee [9]. It uses breadth-first search to construct ordered levels of vertices, so-called level structures, and labels the vertices according to these level structures. The *GPS algorithm* by Gibbs, Poole, and Stockmeyer [23] also uses level structures, but introduces additional steps to find endpoints of a pseudo-diameter of certain graphs, and then minimizes the number of vertices on all levels before labeling. Thus the GPS algorithm achieves solution qualities comparable to the CM algorithm, however in shorter time. It should be noted that the GPS algorithm works particularly well on certain graphs of some special patterns.

Esposito, Malucelli, and Tarricone [17] proposed the *Wonder Bandwidth Reduction Algorithm* (WBRA) which is a variant of the CM method achieving better qualities.

CM, GPS and WBRA are able to compute fairly good solutions, but do not give a guarantee on their quality. In some cases, even for small problems, the quality is only moderate. There are also approximation algorithms giving solutions within a polylogarithmic multiplicative factor of the optimum by Feier [18] and Blum, Konjevod, Ravi, and Vempala [1]. The results of Blum et al. also imply the lower bound $\gamma(G)$ for the bandwidth problem, whose usage can be found later in the text.

Martí, Laguna, Glover, and Campos [37] proposed a *tabu search method* (TS). It uses a candidate list strategy to avoid the expensive computation of the objective function. The solution quality obtained with tabu search is better than with previous algorithms. However, as it is usually the case for such approaches, it is considerably more time consuming than pure construction heuristics like the GPS algorithm. Further studying metaheuristics methods, Piñana, Plana, Campos, and Martí [43] developed a *greedy randomized adaptive search procedure* (GRASP) coupled with a *path relinking strategy*. It uses GPS in the constructive phase, and then applies tabu search and path relinking to improve the results. Computations showed that GRASP achieves slightly better results than TS, but is slower in speed. In line with these meta-heuristic methods, Campos, Piñana, and Martí [4] proposed a method using *scatter search combined with tabu search* (SS_TS). The results are somewhat better than those of GRASP.

Lim, Rodrigues, and Xiao [34] introduced the *node-shift heuristic* (NS). Here weights of vertices are computed according to their adjacent vertices, then the vertices are labeled in a non-decreasing order of their weights. The procedure incorporates a local hill climbing search and is repeated until no more label changes are found. Lim et al. also introduced a *genetic algorithm* (GA), having an initial population phase based on a level structure like GPS and using local hill climbing search. GA uses a middle-point crossover operator for generating new solutions. NS obtains better upper bounds than GA and also improves the results of GRASP to a small degree.

Rodriguez-Tello, Hao, and Torres-Jimenez [47, 48] proposed an evaluation function named *alpha*. The important meaning of *alpha* is that it takes into consideration all the label differences, instead of only the maximal label difference like the original objective function. This evaluation function is applied to a simulated annealing method (SA-σ). They reported both the worst-case and best-case results, where the average results are slightly worse than those of GRASP and TS on small graphs up to 199 vertices, but slightly better on large graphs having more than 200 vertices.

Safro, Ron, and Brandt [50] later used the *multigrid* and *multilevel* methods for the bandwidth problem. According to their report, the upper bounds on the popular benchmark suite (113 instances with less than 1,000 vertices each) is comparable to those of Lim et al. [34], but the average running time is at least 28 times faster. They also reported the upper bounds on another benchmark suite of 51 very large instances, whose number of vertices ranging from a few hundreds up to about 217,000

vertices. Their upper bounds show an improvement of at least 23% compared to the best known results on the second benchmark suite.

3.1.2 Exact methods

It turns out that known exact methods for the bandwidth minimization problem are based on the same concepts. They will be formally described later, but here we try to give a rough overview.

Given an undirected graph $G = (V, E)$ and a permutation π of its vertices, we define the *bandwidth* of G under π is the maximum difference $|\pi(v) - \pi(u)|$ for all $uv \in E$. If there exists a permutation π such that the bandwidth of G under π is φ, we say G has a bandwidth φ.

The bandwidth minimization problem is approached by solving a sequence of search problems. For each φ in the range from 1 to $n - 1$, a search problem, denoted by BANDWIDTH (BW), asks whether G has a bandwidth φ. The optimal solution is the smallest φ for which the answer is "yes".

The algorithms for solving BW are also based on the same concept, the so-called *partial permutation*, which means that a set of k vertices, $k < n$, is assigned to consecutive labels from $1...k$ and/or $n - k + 1...n$. The assigned vertices impose restrictions on label domains for the remaining unassigned, or *free*, vertices. A *label domain* defines the possible labels that a free vertex can be assigned to, without violating the bandwidth constraint $|\pi(v) - \pi(u)| \leqslant \varphi$ for all $uv \in E$. The extendability of a partial permutation, telling whether it can be extended to a full permutation π such that the bandwidth of G under π is φ, is tested by checking if all free vertices can be assigned to the remaining labels in accordance with their label domains. If so, the partial permutation is continuously extended. This procedure is repeated until a partial permutation becomes full, having all vertices assigned. In that case a permutation for the "yes" answer has been found. If none of partial permutations can be extended to such a full permutation, it can be concluded that G does not have a bandwidth φ.

The first known exact method, which uses dynamic programming, was given by Saxe [51]. He showed that BW can be solved in $O(f(\varphi)n^{\varphi+1})$, where $f(\varphi)$ depends only on φ. His work was then improved by Gurari and Sudborough [25], in which BW can be solved in time $O(n^{\varphi})$ using $O(n^{\varphi})$ memory space. Gurari and Sudborough tested the extendability of a partial permutation π^L, where L is the set of assigned vertices and $|L| = k$, by three conditions. The first one is obvious, $|\pi(v) - \pi(u)| \leqslant \varphi$ for $u, v \in L$ and $uv \in E$. Second, there are at most φ assigned vertices in π^L that are adjacent to a free vertex. If this condition did not hold, at least one edge connecting an assigned vertex to a free vertex would have a label difference greater than φ. The third condition states that, the vertex assigned to label $k - i + 1, 1 < i < \varphi$, is

adjacent to at most $\varphi - i + 1$ free vertices. Again the idea is to keep the bandwidth constraint between two *adjacent* vertices, one in the assigned set and one is still free.

Later Del Corso and Manzini [12] proposed two branch-and-bound algorithms. The first one uses a depth first search strategy within a so-called *iterative deepening framework* (MB_ID), while the second one uses a so-called *perimeter search approach* (MB_PS). Both MB_ID and MB_PS start with a trivial lower bound and test if the graph has a bandwidth of this value. If it is not the case, then the lower bound is increased by 1 and the procedure continues. The algorithm stops when an optimal solution has been found or when a CPU time limit is exceeded. MB_ID extends the partial permutations by assigning labels in the direction from 1 to n. MB_PS first creates a set of all possible partial permutations having d vertices being assigned to labels $n - d + 1, ..., n$. It then uses the same procedures as in MB_ID to extends these partial permutations by assigning remaining vertices to labels $1, ..., n - d$. To tighten the label domains, both algorithms also used the bandwidth constraint between *adjacent* vertices like the algorithm of Gurari and Sudborough. However, the bounds on the free vertices, set by the assigned vertices, are numerically computed and then used for the extendability test. Experimental results showed that these algorithms are able to treat small graphs having up to 100 vertices, with a solution quality better than that of WBRA.

A remarkable work with respect to exact methods was done by Caprara and Salazar-González [6]. The authors proposed two lower bounds named $\gamma(G)$ and $\alpha(G)$ which can be computed in $0(nm)$ time. For each partial permutation, the bandwidth constraint is applied in a stronger way, in that an assigned vertex has effect on the label domains of all free vertices, not only on those of its neighbors. Therefore, their approach works more efficiently on sparse graphs than the procedures by Del Corso and Manzini. In addition, they have developed efficient procedures for the extendability test. Furthermore, they proposed a constraint to tighten the label domains of free vertices based on the bounds of vertices closer to the assigned set. By using the label domains as integer variables, they are able to perform the bounding and testing procedures numerically. The work of Caprara and Salazar-González therefore has set an advanced step in solving the bandwidth problem.

Caprara and Salazar-González employed their procedures in two branch-and-bound algorithms, the *LeftToRight* extends the partial permutations from left-to-right direction only, which means that vertices are assigned to consecutive labels $1, 2, ..., k$, while the *BothWay* can extend from the other direction with labels $n, n - 1, ..., n - q + 1$ as well. The algorithms start searching from a good lower bound $b_{low} = \max\{\gamma(G), \alpha(G)\}$ instead of a trivial bounds, and going upwards until either an optimal solution is found or the time limit is exceeded. LeftToRight and BothWay are able to solve 24 out of 30 problem instances with less than 200 vertices to optimality, while MB_ID and MB_PS can achieve this in only 10 cases. For instances not solved to optimality, better lower bounds are obtained in shorter time.

Martí, Campos, and Piñana [36] proposed an improvement of LeftToRight, namely *BB*, in which a single branch-and-bound tree is used instead of a series of branch-and-bounds. The reason is that they tested values φ downward, starting from the initial upper bound. Clearly, partial permutations which failed the extendability test with a large φ cannot be "extendable" with smaller φ. GRASP is used to compute the first upper bound and also the initial solution. They also proposed a new constraint to tighten label domains of free vertices. By adding this constraint BB achieves better lower bounds than did BothWay and LeftToRight.

3.1.3 Computational experiments

Benchmark instances from the library *Matrix Market* organized by Boisvert, Pozo, Remington, Barrett, and Dongarra [2] are often used to perform computational experiments. Many methods have been evaluated on a *popular benchmark suite* which was introduced in Martí et al. [37] and consists of 113 instances with less than 1,000 vertices. To date, best known lower bounds for these instances are reported in Martí et al. [36]. The best known upper bounds are obtained by taking the best values from different methods reported in Martí et al. [37, 36], Piñana et al. [43], Campos et al. [4, 5], and Lim et al. [34]. These are the best known results obtained with a non-parallel computer for instances in the popular suite (also referred to as *the first suite*).

In addition to the popular suite, some heuristic methods also used another one for their computational experiments. The *second benchmark suite* contains 51 very large instances with size ranging from a few hundreds up to 217,000 vertices. The instances are taken from *The University of Florida Sparse Matrix Collection* which is maintained by Davis and Hu [11]. The best known upper bounds for instances in this suite are obtained by Safro, Ron, and Brandt [50]. We do not know if there exists any work reporting lower bounds for instances with more than 1,000 vertices from this suite.

On the popular benchmark suite, the gap between best known lower and upper bound is still huge for many instances. For the reader to have an idea of how large these gaps are, some hard instances are listed in Table 3.1, along with their best known lower bounds (LB) and upper bounds (UB).

In our previous work [55], we have parallelized the algorithm *BothWay* (developed by Caprara and Salazar-González [6]) and ran the solver on a parallel cluster [26] using 508 processors (127 computer nodes with 4 processors each). We also used the first benchmark suite, but performed computational experiments on only 33 small instances with less than 200 vertices. Compared with the best known results, we found optimal solutions for 2 more instances which are listed in Table 3.1: `impcol_b` and `bcsstk22`. In addition, lower bounds for 3 other instances are improved.

Table 3.1: Best known bounds for some hard instances.

Instance	Vertices	Edges	LB	UB	Gap(%)
impcol_b	59	281	19	20	5.26%
bcsstk22	110	254	9	10	11.11%
west0132	132	404	25	33	32.00%
gre___185	185	650	17	21	23.53%
str___600	363	3244	101	132	30.69%
gre___512	512	1680	30	36	20.00%
nos7	729	1944	43	65	51.16%
bp___1600	822	4809	199	300	50.75%
west0989	989	3500	123	210	70.73%

Table 3.1 shows the best known lower and upper bounds from the literature for some hard instances using a non-parallel computer. The huge gaps indicate that the bandwidth minimization problem belongs to the class of difficult problems. As reviewed in the previous section, many heuristic methods have reported upper bounds of nearly equivalent solution quality for instances in the popular benchmark suite. We wonder if the computation of upper bounds has reached a mature state, but this can only be answered with stronger lower bounds. Therefore, powerful methods for computing lower bounds are still needed.

3.2 Lower bounds

Caprara and Salazar-González [6] defined the lower bound $\gamma(G)$ and also derived from the result of Blum et al. [1] another lower bound $\alpha(G)$. The bounds are defined as follows:

$$\alpha(G) = \max_{v \in V} \max_{h \in \{1,\ldots,d(v,V)\}} \left\lceil \frac{|N_h(v)|}{2h} \right\rceil \tag{3.1}$$

$$\gamma(G) = \min_{v \in V} \max_{h \in \{1,\ldots,d(v,V)\}} \left\lceil \frac{|N_h(v)|}{h} \right\rceil \tag{3.2}$$

where $N_h(v)$ is the set of vertices whose distance from v is at most h, *not* including v itself. $d(v, V)$ is the maximum distance from v to any vertex in V.

Both of these bounds above can be efficiently computed in time $0(nm)$.

3.3 The canonical IP formulation

Given an undirected graph $G = (V, E)$, $|V| = n$, the canonical IP formulation of the bandwidth minimization problem is defined as follows:

$$\begin{array}{lcll} \min & \Phi & & \\ & \Phi & \geqslant & \pi(u) - \pi(v), \text{ for all } uv \in E, \\ & \Phi & \geqslant & \pi(v) - \pi(u), \text{ for all } uv \in E, \\ & \pi & \in & \Pi, \end{array} \tag{3.3}$$

where Π is the set of all permutations π of $\{1, ..., n\}$. As stated in Caprara and Salazar-González [6], a permutation π can be mathematically described by the constraints:

$$\begin{array}{lcll} \displaystyle\sum_{u \in V} \pi(u) & = & \dfrac{n(n+1)}{2}, & \\ \displaystyle\sum_{u \in S} \pi(u) & \geqslant & \dfrac{|S|(|S|+1)}{2}, & \text{for all } S \subset V,\ S \neq \emptyset, \\ \pi \text{ integer.} & & & \end{array} \tag{3.4}$$

We observe that the LP relaxation of formulation (3.3), i.e., the integrality constraints and $\pi \in \Pi$ are removed from a combination of (3.3) and (3.4), is of no use at all. The reason is that its optimal objective function value is zero, with solution values $\pi(u) = n(n+1)/2$ for all $u \in V$. We also notice that the objective function in (3.3) is not a linear function of π.

For the reason outlined above, the branch-and-cut method, which is very efficient in solving integer problems, does not seem to be useful with the bandwidth minimization problem. One often uses the classical branch-and-bound method, trying to define the branching and bounding procedures in such a way that the bandwidth constraint is applied as efficiently as possible. This is the approach used in previous exact methods as well as in our improvement. They will be described in detail shortly.

3.4 Partial permutation

In this section we introduce the *partial permutation* concept and describe how it is used to find solutions for the bandwidth minimization problem.

3.4.1 Definitions

Definition 3.1
Given a graph $G = (V, E)$, a permutation *π of V is a bijection between V and $\{1, 2, \ldots, n\}$. $\pi(v)$ is called* label *of v under π.* △

Definition 3.2
The bandwidth *of a graph G under permutation π is the maximum of label differences between any pair of adjacent vertices. That is: $\Phi_\pi(G) = \max |\pi(v) - \pi(u)|$ for all $uv \in E$.* △

We say that a graph G has a bandwidth φ if there exists a permutation π such that the bandwidth of G under π is φ.

Since it is hard to be solved directly from the known IP formulation (3.3), the bandwidth minimization problem can be approached by a sequence of search problems. For each φ from 1 to $n - 1$, the search problem asks whether the graph G has a bandwidth φ. When the search sequence is finished, the optimal solution is the smallest φ for which the answer is yes. Such a search problem is formally defined as follows.

Definition 3.3
BANDWIDTH (BW)
Instance: An undirected graph $G = (V, E)$ and an integer φ, $1 \leqslant \varphi < n$.
Question: Does G have a bandwidth φ?
Task: If the answer is yes, find a permutation π such that the bandwidth of G under π is φ and return true, otherwise return false. △

In the context of BW, we call φ the *search parameter*. BW can be solved by exhaustively testing all possible permutations to check whether G has a bandwidth φ. This can be done more efficiently in a branch-and-bound algorithm, where vertices are successively assigned first to label 1, then label 2, and so on. After each assignment, which is equivalent to a branch-and-bound subproblem, labels of assigned vertices can be used in combination with the *basic bandwidth constraint*, i.e., $|\pi(v) - \pi(u)| \leqslant \varphi$ for all $uv \in E$, to reduce the set of possible labels that an unassigned or *free* vertex can be assigned to. If there is no possible label remaining for any vertex, we know that the assignment cannot be extended to a (full) permutation such that the bandwidth of G under it is φ. This enables the elimination of subproblems in the search tree and reduces the running time of the algorithm.

The idea of the branch-and-bound algorithm for solving BW is based on the sequence of assigning some vertices to consecutive labels. Such a sequence is called a *partial permutation*, and is formally defined as follows.

Definition 3.4
A partial permutation on the left, denoted as π^L, is a one-to-one mapping from a set of k vertices $L = \{v_1, v_2, ..., v_k\}$ to consecutive labels $1, 2, ..., k$ with $\pi^L(v_1) = 1$, $\pi^L(v_2) = 2$,..., $\pi^L(v_k) = k$. △

A permutation can also be viewed as a *linear layout* of n vertices which assigns them to n distinct labels $1, 2, ..., n$ from left to right. In Definition (3.4), the vertices are assigned to labels $1, 2, ..., k$ on the left side of the layout. This can be done on both sides. In addition to k vertices on the left, we can assign q other vertices to labels $n, n-1, ..., n-q+1$ on the right side.

Definition 3.5
A partial permutation on both sides, denoted as $\pi^{L,R}$, is a one-to-one mapping from a set of k vertices $L = \{v_1, v_2, ..., v_k\}$ to consecutive labels on the left with $\pi^{L,R}(v_1) = 1$, $\pi^{L,R}(v_2) = 2$,..., $\pi^{L,R}(v_k) = k$, and another set of q vertices $R = \{v_{n-q+1}, v_{n-q+2}, ..., v_n\}$ to consecutive labels on the right with $\pi^{L,R}(v_{n-q+1}) = n-q+1, \pi^{L,R}(v_{n-q+2}) = n-q+2, ..., \pi^{L,R}(v_n) = n$. △

For convenience, we write *left partial permutation* for a partial permutation on the left, and *both-sided partial permutation* for a partial permutation on both sides. A partial permutation in general refers to either types. In a partial permutation, the set of possible labels that a free vertex can be assigned to without violating the bandwidth constraint is called the *label domain* of that vertex. We call the set of assigned vertices *assigned set* and the set of unassigned vertices *free set*. We denote the free set F, i.e., $F = V \setminus L$ in π^L and $F = V \setminus \{L \cup R\}$ in $\pi^{L,R}$. In a partial permutation L is called the *left set* and R the *right set*.

In a branch-and-bound algorithm, BW is solved by checking all possible partial permutations whether they can be extended to a *feasible permutation*, i.e., a permutation π such that the bandwidth of G under π is φ. Recall that in each partial permutation label domains of free vertices can be reduced, or *tightened*, by the basic bandwidth constraint using labels of the assigned vertices. If some vertex cannot be assigned to any remaining label due to its label domain, a *violation* has been found. In that case further extensions would result in new partial permutations that violate the bandwidth constraint, so it is sufficient to stop here. If no such violation is found, $\pi^{L,R}$ is continuously extended. The algorithm stops if none of the partial permutations can be extended fully, in that case G does not have a bandwidth φ. Otherwise, there will be at least one partial permutation that can be extended to the size of n vertices, meaning that a feasible permutation has been found.

The label domains of free vertices can be tightened not only by the basic bandwidth constraint, but also through the other constraints derived from the graph properties. The stronger the constraints and the more efficient to compute the bounds, the fewer number of partial permutations have to be processed and the faster the running time.

The following exact methods focus on finding new constraints which can tighten the bounds as strongly and efficiently as possible. We first describe how to test the extendability of a partial permutation.

3.4.2 Extendability problems

We begin with the extendability problem for left partial permutations. The information about the undirected graph $G = (V, E)$ is implied in the problem instance.

Definition 3.6
Left Partial Permutation Extendability (LPPE)
Instance: A left partial permutation π^L, $|L| = k$, and an integer φ.
Question: Can π^L be extended to a permutation π such that the bandwidth of G under π is φ? △

In the two branch and bounds proposed by Del Corso and Manzini [12], LPPE is implicitly used with the constraint:

$$\pi^L(v) - \pi^L(u) \leqslant \varphi, \quad u \in L,\, v \in F,\, uv \in E \tag{3.5}$$

LPPE together with (3.5) basically check whether π^L can be extended to a feasible permutation in such a way that the basic bandwidth constraint is satisfied between any two adjacent vertices: one in the assigned set and the other is still free.

Proposition 3.1
The LPPE problem with constraint (3.5) can be solved in time $O(m)$. △

Proof First, compute $d_v = min\{\pi^L(u) + \varphi : u \in L \cap N_1(v)\}$ for all v ∈ F. If $N_1(v) \cap L = \emptyset$ then set $d_v = n$. Second, sort values d_v non-decreasingly. Third, for each d_v let $S = \{s : d_s \leqslant d_v\}$. If $|S| + k > d_v$ then at least one vertex will be assigned to a label outside its domain, which clearly shows that π^L will violate the bandwidth constraint after some further extensions.

The running time for computing d_v is $O(m)$, sorting values d_v with bucket sort is $O(n)$, and testing the values d_v in the third step is $O(n)$, thus yields a total running time $O(m)$. □

Note that we describe the procedure for solving LPPE with bucket sort because this is how it is used in the original work of Caprara and Salazar-González [6]. In fact one can apply any efficient sorting technique here.

Constraint (3.5) works well for dense graphs, in which a vertex is adjacent to many others, thus an assigned vertex can restrict the label domains of many free vertices. However for sparse graphs it is not the case because the constraint cannot bound to

further vertices which are more than one distance unit away from labeled vertices. Caprara and Salazar-González [6] resolved this problem for sparse graphs with a stronger bounding. It is formulated in constraint (3.6). For each assigned vertex, the bandwidth constraint is applied to any free vertex, not only to adjacent ones as in constraint (3.5).

$$\pi^L(v) - \pi^L(u) \leqslant \varphi d(u,v), \quad u \in L,\, v \in F \tag{3.6}$$

Proposition 3.2
The LPPE problem with constraint (3.6) can be solved in time $O(m)$. △

Proof Define l_v as the largest possible label in π^L that a vertex v can be assigned to, i.e.,

$$l_v = \min\{h\varphi + \pi^L(u) : u \in L \cap N_h(v)\}, \tag{3.7}$$

where $v \in F$ is at distance h from $u \in L$, $h = d(u,v)$. We call l_v the *max label* of v in π^L.

The label domain of an unassigned vertex v in π^L is defined by the range $[k+1, l_v]$. A partial permutation π^L can be extended to a feasible permutation only if $\pi^L(v) \leqslant l_v$ for all $v \in F$. This condition is tested as follows. First, compute l_v for all $v \in F$. Second, sort values l_v non-descendingly. Third, for each l_v, let $S = \{l_s : l_s \leqslant l_v\}$. If $|S| + k > l_v$ then at least one vertex will be assigned to a label outside of its domain, which means that π^L cannot be extended to any feasible permutation.

The time for computing l_v is $O(m)$, sorting values l_v with bucket sort is $O(n)$, and testing values l_v in the third step is $O(n)$, thus the total running time is $O(m)$. □

We now describe how to test the extendability of a both-sided partial permutation. The problem is formulated as follows.

Definition 3.7
Both-sided Partial Permutation Extendability (BPPE)
Instance: A both-sided partial permutation $\pi^{L,R}$, $|L| = k$, $|R| = q$, and an integer φ.
Question: Can $\pi^{L,R}$ be extended to a permutation π such that the bandwidth of G under π is φ? △

In their work, Caprara and Salazar-González [6] used the BPPE problem and applied the bandwidth constraint from both (left and right) assigned sets to the free set. The constraint is formulated in (3.8).

$$\begin{aligned} \pi^{L,R}(v) - \pi^{L,R}(u) &\leqslant \varphi d(u,v), \quad u \in L,\, v \in F \\ \pi^{L,R}(w) - \pi^{L,R}(v) &\leqslant \varphi d(w,v), \quad w \in R,\, v \in F \end{aligned} \tag{3.8}$$

Proposition 3.3
The BPPE problem with constraint (3.8) can be solved in time $O(m + n^2)$. △

Proof In a both-sided partial permutation $\pi^{L,R}$, we keep the definition of l_v in (3.7), i.e., the largest possible label that vertex v can be assigned to. Define f_v as the smallest possible label in $\pi^{L,R}$ that a vertex $v \in F$ can be labeled to, i.e.,

$$f_v = \max\{\pi^{L,R}(w) - h\varphi : w \in R \cap N_h(v)\}, \tag{3.9}$$

where $v \in F$ is at distance h from $w \in R$, $h = d(w, v)$. We call f_v the *min label* of v in π^L.

Notice that by definition values l_v are determined only by the left set and similarly values f_v are determined by the right set. The partial permutation $\pi^{L,R}$ can be extended to a feasible permutation only if $f_v \leqslant \pi^{L,R}(v) \leqslant l_v$ for all $v \in F$. In other words, the range $[f_v, l_v]$ defines the label domain of vertex v. The extendability test for a both-sided partial permutation thus comprises of three steps.

The first step checks whether $\pi^{L,R}(v) \leqslant l_v$ for all $v \in F$. The running time of this step is $O(n)$. To start with, free vertices are sorted non-decreasingly according to their l_v with bucket sort in $O(n)$. Notice that more than one vertex may have the same l_v. Then, for each l_v we check whether all free vertices s having l_s not greater than l_v can be assigned to label range $[k+1, l_v]$. If not then violation has been found. The pseudo code is described in Algorithm 3.1.

Algorithm 3.1: Testing values l_v.

Input: $\pi^{L,R}$, $|L| = k$.
Output: True if $\pi^{L,R}(v) \leqslant l_v$ for all $v \in F$ and false otherwise.

```
sort vertices v ∈ F non-decreasingly according to l_v;
put sorted vertices into list F_l;
numSmallerEqual = 0;
nextLeftLabel = k + 1;
{start from the smallest l_v to the largest}
for each value l_v of vertices in F_l do
    S = {s ∈ F_l : l_s = l_v};
    numSmallerEqual += |S|;
    if l_v - (numSmallerEqual − 1) < nextLeftLabel then
        return false;
    end if
end for
return true;
```

The same test is performed in the second step, checking whether $f_v \leqslant \pi^{L,R}(v)$ for all $v \in F$. The running time of this step is also $O(n)$. It is done by first sorting values f_v non-decreasingly. Starting with the largest f_v, for each f_v we check if all free vertices whose f_v not smaller than f_v can be assigned to label range $[f_v, n - q]$, if not then violation has been found, as outlined in Algorithm 3.2.

Algorithm 3.2: Testing values f_v.

Input: $\pi^{L,R}$, $|R| = q$.
Output: True if $f_v \leqslant \pi^{L,R}(v)$ for all $v \in F$ and false otherwise.

```
sort vertices v ∈ F non-decreasingly according to f_v;
put sorted vertices into list F_f;
numLargerEqual = 0;
nextRightLabel = n − q;
{start from the largest f_v down to the smallest}
for each value f_v of vertices in F_f do
  S = {s ∈ F_f : f_s = f_v};
  numLargerEqual += |S|;
  if f_v + (numLargerEqual − 1) > nextRightLabel then
    return false;
  end if
end for
return true;
```

Finally, after testing separately f_v and l_v for all $v \in F$ against the remaining labels, one needs to test the main condition $f_v \leqslant \pi^{L,R}(v) \leqslant l_v$. For each label l in the range $[k+1, n-q]$, inserting all vertices w having $f_w \leqslant l$ into a *heap*, then removing w with smallest l_w from the heap and assigning w to l. Violation is found in two cases:

1. The heap is empty, meaning that there is no vertex w whose min label f_w less than or equal to l.
2. The currently smallest l_w is smaller than l. This means that vertex w cannot be assigned to any remaining label starting from l upwards.

The pseudo code of the final test is described in Algorithm 3.3, except for line 1 and line 15 are for an upper bound heuristic which will be described shortly. The heap is actually an array of integers. It is initialized with a set of vertices w having $f_w \leqslant$ *nextLeftLabel* at the first iteration of the for loop. For each increased value of l, only vertices w with $f_w = l$ are inserted to the heap. Then all vertices w in the heap are scanned to find the vertex with the smallest l_w.

The running time for sorting free vertices is $O(n)$. Actually the sorted list can be reused from the second test. The complexity for testing all remaining labels with the for loop is $O(n^2)$. Thus the running time of the final test is $O(n^2)$.

Combining all three tests and adding $O(m)$ for computing f_v and l_v, the running time for testing the extendability of a both-sided partial permutation is $O(m+n^2)$.□

It should be remarked that these three extendability tests including the implementation for bucket sort have been developed by Caprara and Salazar-González [6]. They work efficiently and we continue using them in our implementations.

Algorithm 3.3: Testing both values l_v and f_v.

Input: $\pi^{L,R}$, $|L| = k$, $|R| = q$.
Output: True if $f_v \leqslant \pi^{L,R}(v) \leqslant l_v$ for all $v \in F$ and false otherwise.

1: $\pi^h(u) = \pi^{L,R}(u) : u \in \{L \cup R\}$; {heuristic for upper bound}
2: $nextLeftLabel = k + 1$;
3: $nextRightLabel = n - q$;
4: sort vertices $v \in F$ non-decreasingly according to f_v;
5: put sorted vertices into list F_f;
6: $heap = \emptyset$;
7: **for** each label l in $nextLeftLabel, ..., nextRightLabel$ **do**
8: $W = \{w \in F_f : f_w \leqslant l\}$; $heap = heap \cup W$; $F_f = F_f \setminus W$;
9: **if** $heap = \emptyset$ **then**
10: **return** false;
11: **else**
12: select vertex w having smallest l_w in $heap$;
13: $heap = heap \setminus \{w\}$;
14: **if** $l_w < l$ **then**
15: **return** false;
16: **else**
17: $\pi^h_w = l$; {heuristic for upper bound}
18: **end if**
19: **end if**
20: **end for**
21: **return** true;

3.4.3 The rounding heuristic

In Algorithm 3.3, line 1 and line 17 are marked with a comment about a heuristic for upper bound. If the graph G does have a bandwidth φ, some partial permutation will be extended to a feasible permutation. If this is the case, for some partial permutations $\pi^{L,R}$, especially ones with a large assigned set, the domain labels of free vertices become so "narrow" that a simple "rounding" which fills the remaining labels with free vertices may result in a feasible permutation.

Line 1 of Algorithm 3.3 initializes a heuristic permutation π^h with the assigned set of $\pi^{L,R}$. The assignment of the "most possible" vertex w, which has the smallest l_w in the heap, to the current label l is done in line 17. When the *for* loop on line 7 is finished, π^h has become a full and valid permutation. The bandwidth of G under π^h is computed and if it is not greater than φ, the algorithm will accept π^h as a new upper bound of the original bandwidth minimization problem.

This turns out to be a useful heuristic in practice. In some cases it can find a feasible permutation quickly because the algorithm does not have to wait until all vertices are assigned to obtain a feasible permutation.

It could be more appropriate to introduce this heuristic in the branch-and-bound section, but we leave it here for the reader's convenience of referring to the pseudo code. The other reason is that it is embedded in the extendability test. The heuristic was also developed by Caprara and Salazar-González [6].

3.5 Branch-and-bound approaches

In this section, we review the basic branch-and-bound algorithm for solving BW. Two constraints for the BW problem from previous works are then presented.

3.5.1 The basic branch-and-bound algorithm

Having the procedures for testing the extendability of a partial permutation available, we can now describe the basic branch-and-bound algorithm for solving BW which was used in Caprara and Salazar-González [6]. Its pseudo code is outlined in Algorithm 3.4.

The main function *bandwidth*(G, φ) creates an empty partial permutation $\pi^{L,R}$ at the root vertex of the branch-and-bound tree, initializes label domains of the vertices, and then passes $\pi^{L,R}$ to the function *labeling*. Here, the partial permutation is extended by having one more vertex assigned. The label domains of its free vertices are tightened with the basic bandwidth constraint, and then its extendability is tested.

Depending on the side of the extension, either left or right, the label domains are tightened by the function *updateBasicBounds* using the label of the newly assigned vertex. Its pseudo code is outlined in Algorithm 3.5, basically applying the definitions of f_v and l_v in (3.9) and (3.7).

The partial permutations are recursively extended until either all vertices are assigned, i.e., a feasible permutation has been found, or all extensions stop in the middle because of failing the extendability test.

The correctness of the algorithm is proved shortly.

Proposition 3.4
BW is correctly solved by Algorithm 3.4. That means, if the graph $G = (V, E)$ has a bandwidth φ, at least one permutation π such that the bandwidth of G under π is φ has been found and the algorithm returns true, otherwise it returns false. △

Proof Proposition 3.4 is proved for the case the partial permutations are extended in both directions, where variable *direction* is set to *BothSides* in Algorithm 3.4. The proof for the case of extending partial permutations only from left to right is included.

The branch-and-bound algorithm starts with a root vertex having an *empty* partial permutation, i.e., both left and right sets are empty. The free set F is initialized with all vertices, whose min labels f_v are set to 1 and max labels l_v are set to n.

For each partial permutation $\pi^{L,R}$, where $|L| = k$ and $|R| = q$, two labels are considered for extending the partial permutation: $k+1$ on the left and $n-q$ on the right. It is obvious that only vertices whose $f_v \leqslant k+1$ can be assigned to the left side, and similarly $l_v \geqslant n-q$ can be assigned to the right side. Aiming at a small branch-and-bound tree, the algorithm chooses to extend on the side producing a smaller number of partial permutations. The extendability of each newly extended partial permutation π^e is tested. If no violation is found, π^e is recursively extended.

If G does really have a bandwidth φ, there will be at least one course of the search that leads to a feasible permutation because no violation can be found in all partial permutations of that search course. Otherwise, all courses stop in the middle because of failing the extendability test. □

Depth-first search strategy is used in Algorithm 3.4 for ease of reading. In practice other strategies can also be used. This is only the core algorithm, where f_v and l_v are calculated using the basic definitions (3.9) and (3.7). These values can be tightened more strongly by valid constraints for the BW problem, thus violation of partial permutations can be detected early and the branch-and-bound tree has fewer number of subproblems. This reduces the search space and speeds up the running time significantly.

Algorithm 3.4: A branch-and-bound algorithm for solving BANDWITH.

Input: An undirected graph $G = (V, E)$ and an integer φ.
Output: True if G has a bandwidth φ and false otherwise.
bool **bandwidth**(G, φ)
begin
  direction = BothSides;
  initialize $\pi^{L,R}$: $L = \emptyset$; $R = \emptyset$; $F = V$;
  for each $v \in V$ **do**
    $f_v = 1$; $l_v = n$;
  end for
  return labeling($\pi^{L,R}$, φ);
end
{Recursively extend $\pi^{L,R}$ until either all vertices are assigned}
{or all extensions stop in the middle because of failing the extendability test}
bool **labeling**($\pi^{L,R}$, φ)
begin
  if $|L| + |R| = |V|$ **then**
    return true;
  end if
  $k = |L|$; $q = |R|$;
  $nextLeftLabel = k + 1$; $nextRightLabel = n - q$;
  $leftCandidates = \{v \in F : f_v \leqslant nextLeftLabel\}$;
  $rightCandidates = \{v \in F : l_v \geqslant nextRightLabel\}$;
  if *direction* = LeftToRight **or** $|leftCandidates| \leqslant |rightCandidates|$ **then**
    for each $v \in leftCandidates$ **do**
      $\pi^e = \pi^{L,R}$; $\pi^e(v) = nextLeftLabel$;
      return testAndExtend(π^e, v, φ, LEFT);
    end for
  else
    for each $v \in rightCandidates$ **do**
      $\pi^e = \pi^{L,R}$; $\pi^e(v) = nextRightLabel$;
      return testAndExtend(π^e, v, φ; RIGHT);
    end for
  end if
end
{Update bounds and test $\pi^{L,R}$. Extend further if no violation found}
bool **testAndExtend**($\pi^{L,R}$, v, φ, *side*)
begin
  updateBasicBounds($\pi^{L,R}$, v, φ, *side*);
  if extendabilityTest($\pi^{L,R}$, φ) = false **then**
    return false;
  else
    return labeling($\pi^{L,R}$, φ);
  end if
end

Algorithm 3.5: Update basic bounds.

void **updateBasicBounds**($\pi^{L,R}$, v, φ, *side*)
begin
  if *side* = LEFT **then**
    $\forall u \in F$: update $l_u = \min\{l_u, \pi^{L,R}(v) + \varphi d(v,u)\}$;
  else
    $\forall u \in F$: update $f_u = \max\{f_u, \pi^{L,R}(v) - \varphi d(v,u)\}$;
  end if
end

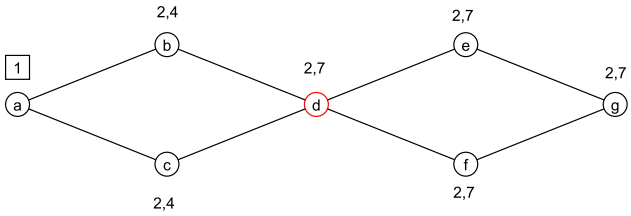


Figure 3.1: An example partial permutation.

3.5.2 Constraint and illustration convention

In many sections of the dissertation, we will describe how label domains of free vertices in a partial permutation are tightened. The tightening is based on *constraints*, which apply the basic bandwidth constraint to specific properties of the graph.

We normally use example graphs to illustrate the idea in which a partial permutation is used. In such a graph, a vertex is drawn as a circle with the vertex number (or vertex identifier) inside. A vertex having a square box next to it is already assigned to a label, whose value is shown in the square box. For each unassigned vertex v, its min label f_v and max label l_v are shown above the vertex as a pair f_v, l_v.

Consider the partial permutation $\pi^{L,R}$ in Figure 3.1. Here we have a graph $G = (V, E)$ where $V = \{a, b, c, d, e, f, g\}$. There is a box next to vertex a, indicating that a has been already assigned to label 1, i.e., $L = \{a\}$, $\pi^{L,R}(a) = 1$, and $R = \emptyset$. The free set $F = \{b, c, d, e, f, g\}$. There is a pair $2, 7$ above vertex d, it means $f_d = 2$ and $l_d = 7$. Therefore $[2, 7]$ defines the label domain of d. In other words, the valid possible labels that d can be assigned to in $\pi^{L,R}$ are $\{2, 3, 4, 5, 6, 7\}$.

Two graphs are often used to illustrate the idea of each constraint: one with "loose" bounds and the other showing bounds tightened with that constraint. The search parameter φ is always 3 in these cases, unless otherwise stated.

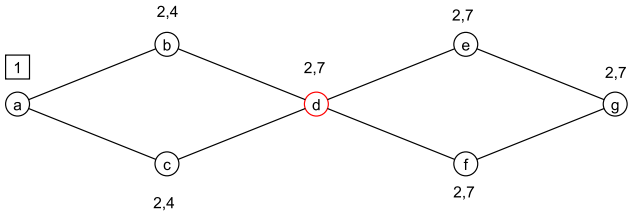


Figure 3.2: Max labels set by the definition.

3.5.3 The first constraint

Recall that l_v is defined as the largest possible label in a permutation $\pi^{L,R}$ that a vertex v can be assigned to. For the basic bandwidth constraint to be satisfied, the definition also implies that $l_v = \min\{h\varphi + l_w : w \in N_h(v)\}$, letting $l_w = \pi_w^{L,R}$ for $w \in L$. Since each vertex can be assigned to only one distinct label, l_v can be tightened more strongly.

For a free vertex v and the set $N_1(v)$ of its neighbors, if there are more than one vertex $w \in N_1(v)$ having the same l_w, for example $l_w = l$, by definition the max label l_v is still weakly constrained to $l + \varphi$. In fact, for all of vertices w such that $l_w = l$, each of them can be assigned to only one label and a stronger l_v should have the value constrained from the smallest l_w.

This constraint was developed by Caprara and Salazar-González [6] and presented as IP formulations in a minimization problem. In the context of the BW problem, it can be formulated as in (3.10).

$$l_v \leqslant \min\Big\{(l_w - (|S| - 1) + \varphi) : w \in N_1(v), S = \{t \in N_1(v) : l_t \leqslant l_w\}\Big\} \tag{3.10}$$

We consider the example in Figure 3.2, which has been explained in the illustration convention section. Here, the max label of d is originally 7 by its definition. We have $N_1(d) = \{b, c, e, f\}$, where $l_b = 4$, $l_c = 4$, $l_e = 7$, and $l_f = 7$. In $N_1(d)$, two max labels are 4 and 7. The constraint is first evaluated with max label 7, for which there are four vertices w whose $f_w \leqslant 7$, and $l_d^1 = 7 - (4 - 1) + 3 = 7$. The second evaluation with max label 4 would result in $l_d^2 = 4 - (2 - 1) + 3 = 6$. At the end, l_d is tightened to 6 by taking the minimum of l_d^1 and l_d^2 , as shown in Figure 3.3.

The min label is similarly tightened as in (3.11).

$$f_v \geqslant \max\Big\{(f_w + (|S| - 1) - \varphi) : w \in N_1(v), S = \{t \in N_1(v) : f_t \geqslant f_w\}\Big\} \tag{3.11}$$

Notice that in [6], the authors did not use the full set $N_1(v)$, but only the set of vertices which are free and one distance unit closer to the assigned set. In the

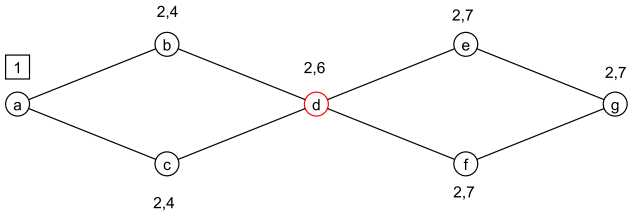


Figure 3.3: Max labels are tightened by the first constraint.

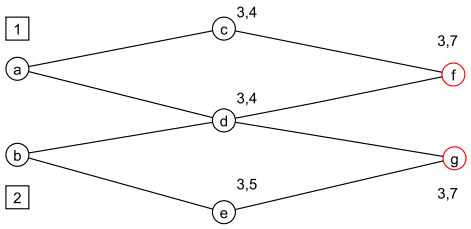


Figure 3.4: A graph arranged as layers away from the left set.

example above, that set is $\{b, c\}$. They reported that using $N_1(v)$ requires longer evaluation time while the results are the same on their instances.

3.5.4 The second constraint

Given a left partial permutation π^L, Martí, Campos, and Piñana [36] proposed a constraint to tighten the min labels by arranging free vertices into layers N_h^L, where N_h^L is the set of vertices whose shortest distance to any vertex $u \in L$ is h.

For each N_h^L, let $N^L = \{N_h^L \cup N_{h-1}^L \cup \ldots \cup L\}$. If the largest max label of vertices $v \in N^L$, namely l_v^c, is equal to $|N^L|$, then vertices in N^L will certainly use all possible labels in the range $[1, l_v^c]$, which is the union of all domain labels in this set. Therefore, vertices of the next layers must be labeled above that range, and their min labels are tightened to $f_v^c + 1$.

The constraint is best described with an example. We consider the partial permutation π^L in Figure 3.4. The left set L consists of two vertices a and b, assigned to labels 1 and 2 respectively. The remaining free vertices are arranged into two layers. The first layer $N_1^L = \{c, d, e\}$, having $l_c = 4$, $l_d = 4$, and $l_e = 5$. The next layer $N_2^L = \{f, g\}$ having $l_f = 7$ and $l_g = 7$. All free vertices v has the same $f_v = 3$. This bound is trivial because with $|L| = 2$ obviously the next label for any free vertex must be 3.

In layer N_1^L, since $l_e = 5$ and is equal to $|L| + |N_1^L|$, three vertices $\{c, d, e\}$ will

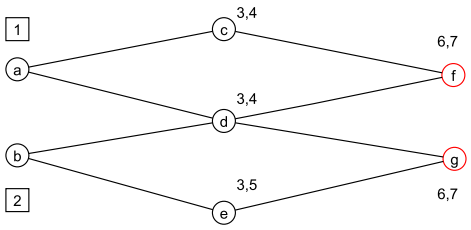


Figure 3.5: Min labels are tightened by the second constraint.

certainly be assigned to the labels $\{3, 4, 5\}$, regardless of what label for each vertex. This leaves no label in the range $[3, 5]$ available to other vertices. Therefore, vertices in layer $N_2^L = \{f, g\}$ must be assigned above that range and their min labels are tightened from 3 to 6, as shown in Figure 3.5.

4 Heuristics and applications

This chapter presents our contributions of heuristic methods for the bandwidth problem and one of its applications. We begin with a modification of a heuristic method which exploits properties of the graph. An approximate objective function named *Sigma* is then introduced. Next, we describe a simulated annealing method applying Sigma. We conclude the chapter with an application of the bandwidth problem to the compression of topological information of digital road networks.

For convenience, in this chapter we refer to the bandwidth of the graph of interest under a valid permutation π as the bandwidth of π.

4.1 The iGPS heuristic

The *GPS heuristic* was proposed by Gibbs, Poole, and Stockmeyer [23]. It is based on the CM algorithm developed by Cuthill and McKee [9]. We first review the GPS heuristic and then introduce our modified version named *iGPS*.

4.1.1 The GPS heuristic

Given an undirected graph, the GPS heuristic constructs level structures which are composed of ordered levels of vertices. A permutation can be obtained from each level structure by consecutively assigning labels to vertices level by level. The bandwidth of this permutation depends on the structure of the levels and the label assignment procedure. The concepts are described as follows.

A *level structure* is a partition of V into sets L_0, L_1, ..., L_k called *levels* such that if a vertex u is adjacent to a vertex v then u and v must belong either to the same level or two adjacent levels. That means, if $uv \in E$, $u \in L_i$ and $v \in L_j$ then $|i - j| \leqslant 1$.

A *level structure* $L_v(G)$ *rooted at* v satisfies $L_0 = \{v\}$ and, for every $i \geqslant 1$ L_i is the set of vertices which are at distance i from v. Recall that $N_k(v)$ is the set of vertices at distance at most k from v not including itself. We can say $N_k(v) = L_1 \cup \ldots \cup L_k$. Since the sets $N_k(v)$ are required for computing lower bounds in exact methods, we can make use of their availability for the GPS heuristic.

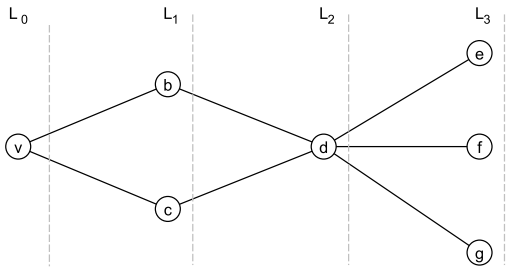


Figure 4.1: A level structure rooted at v.

Figure 4.1 illustrates a level structure rooted at v, where $L_0 = \{v\}$, $L_1 = \{b, c\}$, $L_2 = \{d\}$, and $L_3 = \{e, f, g\}$.

We define the *width of level* L_i as the number of vertices contained in L_i. The *width of a level structure* is the maximum width among its levels and its *depth* is the number of its levels. In the example in Figure 4.1, $L_v(G)$ has width 3 and depth 4.

The GPS heuristic comprises of three steps.

(1) Find endpoints of a pseudo diameter: This step attempts to locate two vertices with nearly maximal distance. The algorithm selects a vertex v of minimal degree and generates a level structure rooted at v and supposedly has depth k. Then it chooses a vertex u in the last level of $L_v(G)$ such that $L_u(G)$ has the same depth k and the smallest width. If there exists any u in the last level of $L_v(G)$ such that the depth of $L_u(G)$ is greater than k then the procedure is started over with u replacing v. This step returns u and v, assumed to be two endpoints of the pseudo diameter.

(2) Reduce level width: The algorithm generates two level structures rooted at u and v and then combines them into a new level structure which usually has smaller width. The reason is that with the vertex numbering procedure introduced in step (3), permutations of smaller bandwidths can usually be obtained from level structures of smaller width.

(3) Vertex numbering: A permutation is obtained by assigning labels to vertices in the combined level structure consecutively from 1 to n level by level. In the first level, label 1 is assigned to the first endpoint of the pseudo diameter v. Next labels are given first to unassigned vertices which are adjacent to an assigned vertex. If that priority is the same, to vertices with a smaller degree. This procedure is repeated until all vertices in the level are assigned. Starting at the second level, the priority are given first to vertices adjacent to ones in the previous level and then to vertices adjacent to assigned ones in the same level. In case of having the same priority, vertices of smaller degree are selected first.

Regarding the complexity, at first vertices of minimal degree need to be determined in $O(n)$. Given a vertex v, the running time for finding a pseudo diameter is $O(nm)$. The procedure for combining two level structures in step (2) requires time $O(m)$ and so does the one for assigning labels in step (3). Thus the overall complexity of the algorithm is $O(nm)$.

In practice, usually the other end point of the pseudo diameter can be found immediately in the last level of $L_v(G)$ without starting over. As a result, this step can be finished in $O(m)$. In this case the running time of the GPS algorithm is only $O(m)$, explaining why it is a very fast heuristic.

4.1.2 The improvement

In our previous work [54], the improvement uses the fact that the set $N_h(v)$, which contains levels of a level structure rooted at v, has to be computed for each vertex v to compute the lower bounds $\alpha(G)$ and $\gamma(G)$ as described in section 3.2. Therefore, we used level structures with exact minimal width, instead of near minimal width as in the original algorithm. Having the list of vertices generating level structures of minimal width, we apply steps (2) and (3) of the original algorithm and select the permutation having the smallest bandwidth.

In exact methods based on the partial permutation concept, we have seen that assigned vertices have effect on the label domains of free vertices as well as the bandwidth of the full permutation. This also means that deciding what vertex to assign to label 1 is an important step to obtain a permutation having a good bandwidth.

In fact, more candidates can be exploited from the list of level structures generated during the computation of theoretical lower bounds. Consider vertices whose rooted level structure has one of the following properties:

(i) Minimal width

(ii) Longest depth

(iii) Smallest sum of the widths of two adjacent levels.

The rationale of the third property is that the bandwidth of a permutation, which is constructed from a level structure using the GPS heuristic, is not larger than the sum of the widths of two adjacent levels. In addition, two vertices that constitute the lower bounds are also considered:

(i) vertex v_α that creates lower bound $\alpha(G)$:

$$v_\alpha = \left\{ v \in V : \max_{h \in \{1,\ldots,d(v,V)\}} \left\lceil \frac{|N_h(v)|}{2h} \right\rceil \text{ is largest } \right\}$$

(ii) and vertex v_γ that creates lower bound $\gamma(G)$:

$$v_\gamma = \left\{ v \in V : \max_{h \in \{1,\ldots,d(v,V)\}} \left\lceil \frac{|N_h(v)|}{h} \right\rceil \text{ is smallest } \right\}$$

While computing the lower bounds using level structures rooted at each vertex, the algorithm records five vertices satisfying each of the five properties above. For each vertex the iGPS heuristic applies three steps of the original GPS. This includes finding the other endpoint of the pseudo diameter, combining two level structures into a new one, and applying the vertex numbering procedure to the combined level structure. Finally, the permutation with smallest bandwidth is selected.

Now we consider the running time. Since each level structure $L_v(G)$ is generated in $O(m)$, all of them are done in $O(nm)$. The selection of the five mentioned vertices is embedded during the generation of level structures. For each selected vertex the running time to obtain a permutation is the same as in GPS. Therefore, the complexity of the iGPS heuristic is still $O(nm)$. It cannot be reduced to $O(m)$ in practical cases like the original heuristic due to the time for generating all level structures. However, if the lower bound computation must be done first as in the cases of exact algorithms then iGPS can make use of that information and needs only $O(m)$ to finish.

The solution quality of iGPS is slightly better than that of the original version. It is somewhat slower, however. A comparison of the two versions on the two benchmark suites is given in Chapter 7.

4.2 An approximate objective function

The original objective function of the bandwidth minimization problem evaluates the bandwidth value of the graph under valid permutations of the vertices. It is defined as $\Phi_\pi = \max\{|\pi(u) - \pi(v)| : uv \in E\}$ for a permutation π. We notice that the number of possible objective function values is only n, which is very small compared with the huge number $n!$ of possible permutations. Therefore, neighbor permutations obtained by changing labels of the current permutation π are very likely to have the same bandwidth as π. As a result, it is not so efficient to use Φ_π in a global optimization heuristic such as the simulated annealing method.

We propose an approximate objective function for the bandwidth minimization problem. It is named *Sigma*, denoted as σ, and composed of two parts. The integral part preserves the original bandwidth value and the fractional part is very likely to change if the labels in the current permutation are permuted. With this property Sigma can be used efficiently in global optimization heuristic methods.

4.2.1 The approximate objective function *Sigma*

Sigma is derived from the original objective function by smoothing the *max* function to a *sum* function. We were introduced to the concept of the smoothing function [46, pp. 27] in a lecture on digital image processing [52] in which the function is used for solving a clustering problem. It is described as follows.

Suppose that we have a set S of integers. $S = \{x_1, x_2, ..., x_u\}$ and $x_{\max} = \max\{x : x \in S\}$. With $k \in Z$ large enough, we have:

$$\sum_{i=1}^{u} b^{kx_i} \approx b^{kx_{\max}} \tag{4.1}$$

Since $kx = \log_b(b^{kx})$, from (4.1) the maximum element in S can be approximated in terms of a sum function:

$$x_{\max} = \tfrac{1}{k}kx_{\max} = \tfrac{1}{k}\log_b(b^{kx_{\max}}) \approx \tfrac{1}{k}\log_b(\sum_{i=1}^{u} b^{(kx_i)}) \tag{4.2}$$

Recall that the bandwidth of a graph under a permutation π is the maximum of all label differences of two adjacent vertices. Each label difference is an integer in the range $[1, \Phi_\pi]$, where $\Phi_\pi = \max\{|\pi(u) - \pi(v)| : uv \in E\}$. The function Sigma is derived as follows.

Given a permutation π, denote $C = b^k$ and n_d the number of edges uv having label difference $|\pi(u) - \pi(v)| = d$, from (4.2) we have:

$$\begin{aligned}
\max_{uv\in E}|\pi(u)-\pi(v)| &= \max\{1, 2, \ldots, \Phi_\pi\} \\
&\approx \frac{1}{k}\log_b(n_1 b^k + n_2 b^{2k} + \ldots + b^{\Phi_\pi k}) \\
&= \frac{1}{k}\log_b(n_1 C + n_2 C^2 + \ldots + C^{\Phi_\pi}) \\
&= \frac{1}{k}\log_b C^{\Phi_\pi}(\frac{n_1}{C^{\Phi_\pi-1}} + \frac{n_2}{C^{\Phi_\pi-2}} + \ldots + \frac{n_{\Phi_\pi-1}}{C} + 1) \\
&= \frac{1}{k}\log_b C^{\Phi_\pi} + \frac{1}{k}\log_b(\sum_{i=1}^{\Phi_\pi-1}\frac{n_i}{C^{\Phi_\pi-i}} + 1) \\
&= \frac{1}{k}\log_b b^{k\Phi_\pi} + \frac{1}{k}\log_b(\sum_{i=1}^{\Phi_\pi-1}\frac{n_i}{C^{\Phi_\pi-i}} + 1) \\
&= \Phi_\pi + \frac{1}{k}\log_b(\sum_{i=1}^{\Phi_\pi-1}\frac{n_i}{C^{\Phi_\pi-i}} + 1) = \sigma'(\pi)
\end{aligned} \tag{4.3}$$

The function $\sigma'(\pi)$ in (4.3) is a real approximation of the original objective function of the bandwidth minimization problem. The deviation from the exact value, expressed by the fractional part, takes into consideration all label differences less

than Φ_π. To make the use of such a function in a simulated annealing method even more effectively, we introduce a slightly changed version in (4.4). The function is named Sigma and its fractional part considers all label differences, including those equal to Φ_π.

$$\sigma(\pi) \quad = \Phi_\pi + \frac{1}{k}\log_b\left(\sum_{i=1}^{\Phi_\pi}\frac{n_i}{C^{\Phi_\pi - i}}\right) \tag{4.4}$$

where k and b are constants, $C = b^k$, and $n_d = |D|, D = \{uv \in E : |\pi(u) - \pi(v)| = d\}$.

The approximate objective function is named *Sigma* because the term is also used as the notations for some relevant mathematical functions. First, this is an approximation of the original objective function in terms of a *deviation* from the original bandwidth. Here, the deviation is expressed by the fractional part. We usually denote the lower case σ as the deviation in statistics. Second, the upper case sigma $\sum$ is often used to denote a *sum*, which is the result of the smoothing idea in our case.

The fractional part of Sigma in (4.4) takes into account all label differences. This makes it very likely to change when the permutation π changes and enables the function Sigma to work effectively in global optimization heuristic methods.

4.2.2 Computing *Sigma*

The function *Sigma* in (4.4) can be computed in $O(\Phi_\pi)$ as described in Algorithm 4.1. The value of b is chosen to be the Euler's constant for the programming convenience, since basic C/C++ libraries already support the power and log functions of this base. The value of constant k is chosen by experiment and $k = 20$ makes *Sigma* the most sensitive for our implementation.

Algorithm 4.1: Computing Sigma.

Input: A permutation π.
Output: $\sigma(\pi)$.

```
b = e; {Euler's constant}
k = 20;
C = b^k;
denom = 1;
sigmaFraction = n_{Φ_π};
for i = Φ_π − 1 downto 1 do
    denom = denom/C;
    sigmaFraction = sigmaFraction + n_i × denom;
end for
sigmaFraction = (1/k) × log_b (sigmaFraction);
sigma = Φ_π + sigmaFraction;
return sigma;
```

4.3 A simulated annealing method

To improve the solution obtained by iGPS and also to test the function *Sigma*, we apply it in a simulated annealing (SA) method. We first review the SA algorithm and then describe our implementation.

4.3.1 The SA algorithm

SA is a heuristic method which provides some means of escaping local minima. A reference text can be found in [49, pp. 115]. We briefly describe SA as follows.

The SA algorithm starts with an initial solution and goes through many iterations to reach an acceptable solution. The iterations are controlled by a *temperature* T, which is initialized with a high value and decreased over iterations according to the so-called *cooling model*. At each iteration, the algorithm considers some neighbor of the current solution. The neighbor is accepted as a new solution if it is better than the current one, or in case it is not, with a probability specified by an *acceptance probability function*. In this way the algorithm can escape from local minima. The algorithm stops when T reaches a specified value or some stopping condition is met.

We use *Sigma* as the objective function in our implementation and call it SA-σ to differentiate from other SA methods. The initial solution is a permutation obtained from the result of iGPS. SA-σ is described in Algorithm 4.2. Here, the function *sigma* realizes the Algorithm 4.1. Searching for neighbors of a solution is handled by the function *neighbor*. The acceptance probability function is $e^{\frac{-\Delta\sigma}{T}}$.

Algorithm 4.2: The SA-σ algorithm.

Input: An initial permutation π_{iGPS} obtained from the iGPS heuristic.
Output: A permutation π, expected to have a smaller bandwidth.

```
T = T_start;
π_c = π_iGPS;
while T > T_end and runningTime() < timeLimit do
    σ_c = sigma(π_c);
    π_n = neighbor(π_c);
    σ_n = sigma(π_n);
    Δσ = σ_n − σ_c;
    if σ_n < σ_c or e^(−Δσ/T) > random() then
        π = π_n;
    end if
    T = T * r; {r is the cooling rate}
end while
return π
```

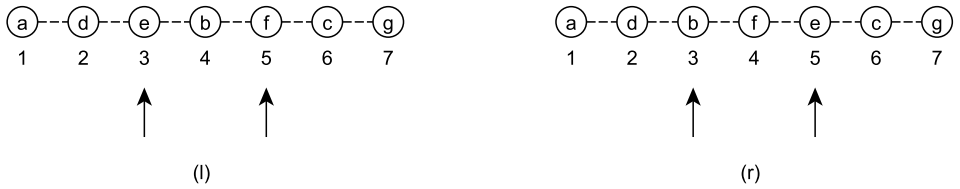


Figure 4.2: A permutation and its rotation.

4.3.2 The neighbor functions

Neighbor candidates can be searched for by two methods. The first one is *swap*, which randomly chooses two labels and exchanges two vertices assigned to the labels. The second method is called *rotation*. It also picks two labels randomly, but performs a round-shift of the vertices between two labels. The rotation method has been introduced in Rodriguez-Tello, Hao, and Torres-Jimenez [48].

Figure 4.2 illustrates an example rotation for a permutation of 7 vertices. The linear layout is depicted as a dashed line with integer labels below it. Each assigned vertex is depicted as a circle on the line at its label. The original permutation is shown on the left (l). When a *rotation(3,5)* is performed, three vertices e, b, and f are round-shifted, as shown in the permutation on the right (r).

As presented in Rodriguez-Tello et al. [48], the rotation function can be expressed as as a product of swap operations. If we define *swap(i,j)* as the exchange of two vertices at label i and label j, then *rotation(i,j)* where $i < j$ can be defined as in (4.5):

$$\text{rotation(i,j)} = \text{swap(i,j)} \times \text{swap(i,j-1)} \times \text{swap(i,j-2)} \times \ldots \times \text{swap(i,i+1)} \quad (4.5)$$

Clearly, a rotation is a combination of swaps. In [48], the authors did confirm the dominance in term of quality of the *rotation* method over the *swap* method. It is a bit different in our case.

We would like to test our methods on the same instances used in previous works with which our results are compared. The reason is that with the same matrix instance, different conversions to graphs may result in different bandwidth values. Such a case can be seen in Chapter 7. Therefore, for instances in the popular suite we use the files prepared in Martí, Campos, and Piñana et al. [37, 43, 4, 36]. In these files, we observe that the graph properties are preserved the same as in the original instance but vertices have been randomly reordered. Our SA-σ which uses *swap* obtains better results than the one with *rotation* on these instances. However, on instances taken directly from the library *Matrix Market rotation* is the better method. Given the instance files, we use *swap* for this benchmark suite.

For the second benchmark suite introduced in Safro, Ron, and Brandt [50], we used instances taken directly from the collection [11]. Here SA-σ with *swap* produces comparable results to the one with *rotation*. Intuitively, *rotation* is likely to create more diversity and expected to produce better quality than *swap*. Probably on very large instances, the number of swaps in each *rotation* operation should be limited to make it more efficient.

4.3.3 The SA-σ implementation

For the implementation of the SA-σ algorithm, we used the parallel simulated annealing library *parSA* developed by Kliewer and Klohs [29, 30]. This looks to be a well-designed framework in C++, supporting MPI [38] and a variety of cooling and acceptance models. We initially wanted to configure and run our algorithm in the parallel mode. Since our time for heuristic methods is limited, we only run SA-σ in the non-parallel mode using the simplest cooling model, the so-called geometric schedule. This is equivalent to Algorithm 4.2.

Concerning the parameters for SA-σ, we used the values reported in [48]. In particular, we set the starting temperature $T_{start} = 1.0 \times 10^{-2}$ and the cooling rate $r = 0.92$. The stop temperature is chosen small enough so that the algorithm can fully use the specified running time. More specifically, the time limit is set to 180 seconds for small instances having less than 200 vertices and 30 minutes for larger instances. The computational results will be reported in Chapter 7.

Since SA is an experimental method, we will need many configurations and tests to select the best parameters. We may even need a set of parameters specific to each benchmark suite. As exact methods are our main focus in this dissertation, we could not spend too much time on this. However, with the availability of the SA framework there is space for improvement in future work.

4.4 Compression of topological information

In this section, we describe an application of the bandwidth problem to the compression of topological information of digital road networks. It is the result of a collaboration project [54] on new approaches for the compression of digital map databases. Here we only focus on the application of the bandwidth problem, the other approaches can be found in the original work. We use again the data and computational results in [54] for the illustration purpose.

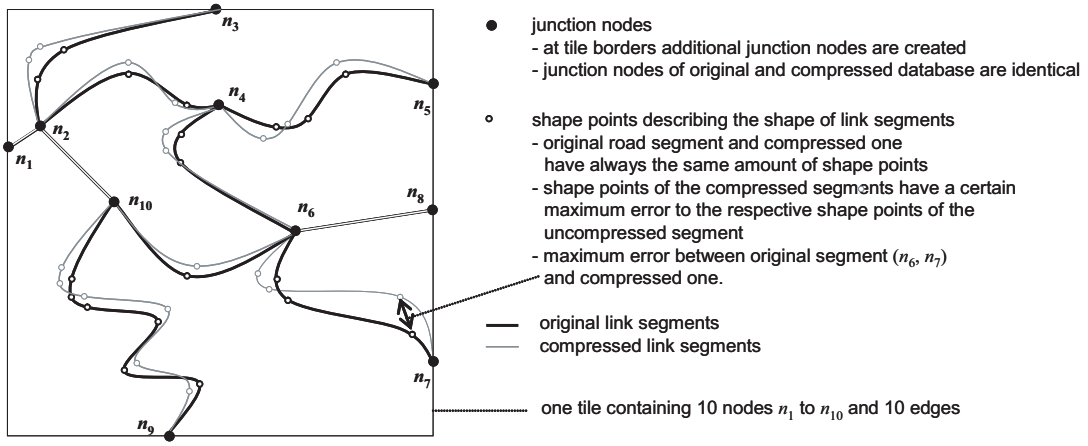


Figure 4.3: Road network within one tile.

4.4.1 Topological information

In digital map databases, topological information of road networks is used for route calculation and is stored in the routing building block of the databases, along with additional geometric information on the positions of vertices and the shapes of links. For having efficient loading units, the road network is often partitioned into small cells called *tiles*, each containing a subgraph of the complete road network. Figure 4.3 illustrates an example tile of a road network. We will only use its topological information, i.e., the junction nodes (subgraph vertices) and the links (edges).

The topological information of road networks is often encoded using adjacency lists since the corresponding graphs are sparse. The graph is considered to be undirected and additional information for each link such as direction is stored separately.

Given an arbitrary vertex ordering which is equivalent to a permutation, traditional approaches would store the graph as follows. The first data entries are the number of vertices n and the maximum vertex degree d_{max}. Then, for each vertex they store its degree and incident edges encoded by the list of its neighbors (sorted according to their labels in the permutation). Notice that an edge is stored only once at the vertex with the smaller vertex number. The vertex degree requires $\lceil \log_2 (d_{max} + 1) \rceil$ bits, while each entry in the list of its neighbors needs $\lceil \log_2 (n + 1) \rceil$ bits.

Table 4.1 shows the topological information, the values, and the corresponding numbers of bits required for storing the graph in Figure 4.3. Here we use 2 bytes for the number of vertices per tile and 1 byte for the maximum vertex degree.

Table 4.1: Storage of the topological information.

Data	Values *(Number of bits)*	Remarks
n	10*(16)*	
$d_{\max}$	4*(8)*	
vertex 1	1*(3)* 2*(4)*	degree 1, one edge: (n_1, n_2)
vertex 2	3*(3)* 3*(4)* 4*(4)* 10*(4)*	three edges: $(n_2, n_3), (n_2, n_4), (n_2, n_{10})$
vertex 3	0*(3)*	no edge: (n_2, n_3) is already included
..	..	..

In the approach described above, the edge information can be stored more compactly using the *delta information*, or the difference of two vertex numbers, instead of the absolute vertex numbers. (Delta encoding is used in many applications such as video codecs or image processing.) This can be improved even further. If the maximum delta is reduced, we can use a smaller number of bits for encoding an edge.

In Figure 4.3, the delta value of the edge (n_2, n_{10}) is 8. If the vertex numbers of n_{10} and n_7 are exchanged, the maximum delta in the graph is decreased from 8 to 5. Therefore, each edge can be encoded using only 3 bits instead of 4 bits. A vertex ordering with a reduced maximum delta can be obtained from the solution of the bandwidth problem.

4.4.2 A compression approach

The traditional approaches require $\lceil \log_2 (n+1) \rceil$ bits for storing a single edge within a tile containing n vertices. Using the solution of the bandwidth problem, the maximum delta can be reduced to a value Φ_u. As a result, the edge can be stored with only $\lceil \log_2 \Phi_u + 1) \rceil$ bits. Interestingly, in this application an optimal gap is sufficient and we do not need to find the optimal bandwidth of the graph. Since binary encoding is used, a bandwidth of 63 produces the same compression ratio as a bandwidth of 32. If the lower bound for the bandwidth of a graph is 2^{k-1} and the upper bound is $2^k - 1$ then the optimal solution for encoding the edges requires k bits.

The upper bound Φ_u of the solution gap and the corresponding vertex ordering can be obtained using a heuristic for the bandwidth problem. Since a road network database is huge and may contain up to a hundred thousand tiles, fast heuristics such as GPS and iGPS are required. In [54], we used the first improvement of the GPS algorithm to obtain the upper bound for the bandwidth of each tile. The lower bound was computed by taking the maximum of $\alpha(G)$ in (3.1) and $\gamma(G)$ in (3.2). It is only for knowing whether the solution gap is optimal and not necessarily required for the edge encoding.

4.4.3 Experimental evaluation and conclusion

The evaluation in [54] is based on real-world European road network data. A data set *DB* consisting of 10,000 arbitrarily chosen tiles is used. Each tile contains between 163 and 296 vertices. The experiments were performed on a 3 GHz PC having 2 GB RAM running Linux. The algorithms were coded in C/C++.

Table 4.2: Evaluation of the topological compression approach.

n	m	$d_{\max}$	Φ_l	Φ_u	*time*	S_t	S_c	R_c (%)
163	212	6	7	11	0.05	2209	1369	38.03
221	301	7	9	15	0.10	3095	1899	38.64
296	397	7	12	21	0.19	4485	2905	35.23

The results of the topological compression for some arbitrarily chosen tiles of *DB* are shown in Table 4.2. For each tile, n is the number of vertices, m is the number of edges, d_{max} is the maximum vertex degree, Φ_l is the lower bound, and Φ_u is the upper bound for its bandwidth. The running times are in seconds.

S_t is the number of bits required for storing the graph using the traditional approach and S_c is the number used by the compression approach. As shown in Table 4.1, $S_t = 3 \times 8 + n \times \lceil \log_2 (d_{max} + 1) \rceil + m \times \lceil \log_2 (n + 1) \rceil$. Since one additional byte is needed for encoding the value Φ_u, we have $S_c = 4 \times 8 + n \times \lceil \log_2 (d_{max} + 1) \rceil + m \times \lceil \log_2 (\Phi_u + 1) \rceil$. The compression ratio is $R_c = (S_t - S_c)/S_t$.

We observe noteworthy compression ratios in Table 4.2. Since Φ_u is less than 16 in most of the tests, an edge can be encoded with only 4 bits instead of 8 or more bits. For the 10,000 tiles, an average compression ratio of 37.98% is achieved.

In summary, a technique for compression of topological information from digital road networks using the solutions of the bandwidth problem has been described. It allows encoding each edge with a small number of bits. We hope that the technique can inspire similar applications of the bandwidth problem where an reordering of vertices may lead to an efficient storage or computation.

5 Exact methods

This chapter presents our contributions of exact methods for the bandwidth problem. They are based on the partial permutation concept introduced in Chapter 3. First, new constraints for the bandwidth problem are introduced. In the next section, we analyze the *dominance* relation between certain partial permutations and describe how it can be used in a branch-and-bound algorithm with a hash-table. The chapter is closed with a new branching scheme named *2-labeling*.

Notice that in the examples, partial permutations and constraints are illustrated with the convention described in Section 3.5.2.

5.1 The new constraints

The new constraints are derived by applying the basic bandwidth constraint to specific properties of the graph. Since they tighten label domains of the free vertices, the size of the branch-and-bound tree can be reduced and violated partial permutations can be detected early. This speeds up the running time significantly.

5.1.1 The fitting constraint

The bandwidth of a graph under a permutation is φ only if any pair of adjacent vertices are not assigned to two labels more than a distance φ away. If the degree of a vertex v is larger than φ, v cannot be assigned to a label which is completely on the left (or on the right) of its neighbors, instead it must be fitted between them. We use this fact as a way to tighten the label domains, and call it the *fitting constraint*.

Tightening max labels

Consider the example in Figure 5.1. The graph has 7 vertices, vertex a has been assigned to label 1, and max labels of all free vertices are set by the definition. In addition, the max label of v has been tightened by the first constraint (3.10). For simplicity, we only focus on the max labels for now and ignore the min labels, whose values are all 2.

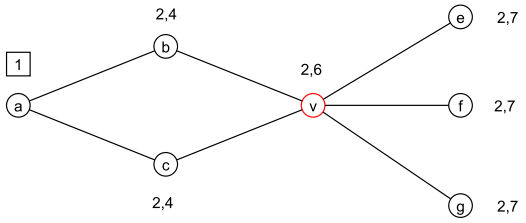


Figure 5.1: Max labels already tightened by the first constraint.

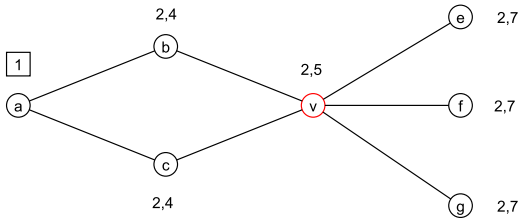


Figure 5.2: Max labels are tightened further by the fitting constraint.

Since vertex v has degree 5 and the search parameter φ is 3, v cannot be assigned on the right of more than three neighbors. In other words, the label of v must be smaller than those of at least *two* adjacent vertices. The constraint can be formulated as follows:

$$\begin{aligned} &\text{if } (|N_1(v)| > \varphi) \text{ then} \\ &\quad l_v \leqslant l_w^m - (|N_1(v)| - \varphi) : l_w^m = \max\{l_w : w \in N_1(v)\} \end{aligned} \tag{5.1}$$

Notice that w does not have to be a free vertex. Applying the fitting constraint to the current example, because v has 5 adjacent vertices, we have $l_v \leqslant 7 - (5 - 3)$. Therefore, the max label of v is now tightened from 6 to 5, as shown in Figure 5.2.

If we apply the same idea to the list of vertices which are more than one distance unit away from v, the fitting constraint can be generalized as follows:

$$\begin{aligned} &\text{if } (|N_h(v)| > h\varphi) \text{ then} \\ &\quad l_v \leqslant l_w^m - (|N_h(v)| - h\varphi) : l_w^m = \max\{l_w : w \in N_h(v)\} \end{aligned} \tag{5.2}$$

We denote by *excess-range* the value of $(|N_h(v)| - h\varphi)$. Notice that the fitting constraint is applicable only if the excess-range is positive. In many cases, the max labels l_w of free vertices are the same. Since these vertices must be assigned to unique labels in a permutation, we can exploit this fact to get even stronger bounds with the fitting constraint in a way similar to that of the first constraint. First, max labels of $w \in N_h(v)$ are sorted non-decreasingly. Given this sorted list, the algorithm starts

with the largest l_w and re-evaluated l_v according to (5.2), and then the value of the excess-range is reduced with an amount equal to the number of vertices having that max label. This procedure is repeated until the excess-range is not positive anymore, and through that loop the smallest l_v, or the strongest, is selected.

Since layers $N_h(v)$ are fixed for each vertex v, the excess-range should be evaluated only once for each φ. Before solving each BW problem, the algorithm checks and records vertices v which have positive excess-ranges, i.e., $|N_h(v)| - h\varphi > 0$, and for each vertex the layer h_{heu} which makes the excess-range largest. For all later computations, the fitting constraint (5.2) is applied only to recorded vertices having positive excess-ranges and directly with their layers h_{heu}. This makes the procedure more efficient.

Finally, to save the cost of the sorting procedure, only free vertices will be used. Therefore, we use the computation as in (5.3).

$$\begin{aligned} &\text{if } (|N_h^F(v)| > h\varphi) \text{ then} \\ &\quad l_v \leqslant l_w^m - (|N_h^F(v)| - h\varphi) : l_w^m = \max\{l_w : w \in N_h^F(v)\}, \end{aligned} \tag{5.3}$$

where $N_h^F(v) = N_h(v) \cap F$.

The layer h_{heu} computed in advance may not always generate the best excess-range when (5.3) is used. However, it is still more efficient than re-evaluating the excess-range before applying the constraint for every partial permutation.

Tightening min labels

The same idea is used to tighten the min labels f_v. We continue using the example in Figure 5.2. Here, the max label of v has been tightened, but its min label remains unchanged at 2.

Since $|N_1(v)| = 5$ and the search parameter $\varphi = 3$, v cannot be assigned to the left of more than three of its adjacent vertices, which means that its label must be larger than those of at least *two* adjacent vertices. Because all of its adjacent vertices have min label 2, the min label of v is tightened from 2 to 4, as shown in Figure 5.3. We notice that the fitting constraint has tightened the label domain of v quite strongly in this case, from $[2, 6]$ to $[4, 5]$.

The fitting constraint for min labels is formulated as follows:

$$\begin{aligned} &\text{if } (|N_1(v)| > \varphi) \text{ then} \\ &\quad f_v \geqslant f_w^m + (|N_1(v)| - \varphi) : f_w^m = \min\{f_w : w \in N_1(v)\} \end{aligned} \tag{5.4}$$

The generalized version of the fitting constraint, in which free vertices are more than

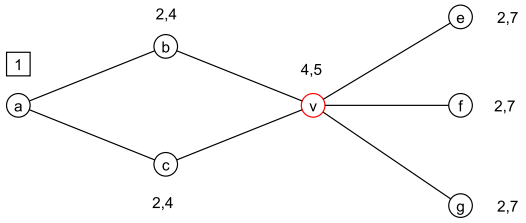


Figure 5.3: Min labels tightened by the fitting constraint.

one distance unit away from a vertex v, is formulated in (5.5).

$$\begin{aligned} &\text{if } (|N_h^F(v)| > h\varphi) \text{ then} \\ &\quad f_v \geqslant f_w^m + (|N_h^F(v)| - h\varphi) : f_w^m = \min\{f_w : w \in N_h^F(v)\}, \end{aligned} \tag{5.5}$$

where $N_h^F(v) = N_h(v) \cap F$.

We use the same procedures for improving the efficiency as in the case of tightening max labels. The fitting constraint is applied only to vertices v having positive excess-ranges and for each vertex the layer h_{heu} with which the excess-range is largest. These values are computed once in advance before solving each BW problem. The evaluation of f_v is also based on the sorted list of f_w, for $w \in N_h^F(v)$, to utilize the fact that each vertex can be assigned only to one label. With these procedures the constraint can tighten f_v more strongly.

5.1.2 The pulling constraint

This is not a completely new constraint, in fact it is a generalization of the first constraint developed by Caprara and Salazar-González [6] in which only adjacent vertices are considered in tightening the label domain of a free vertex v. The authors did mention an extension of their constraint which uses a set of free vertices whose shortest distance to an assigned vertex is closer than that of v. However they did not use it in the end because on their benchmark instances, stronger bounds did not compensate for the computational cost and this resulted in longer running time.

We propose a generalization of this constraint in a way similar to that of the fitting constraint, so that it can be used efficiently. The name *pulling* is given to this constraint because it tends to "pull" free vertices closer to the assigned set. The pulling constraint for max labels is formulated as follows.

$$l_v \leqslant \min\Big\{(l_w - (|S| - 1) + h\varphi) : w \in N_h(v), S = \{t \in N_h(v) : l_t \leqslant l_w\}\Big\} \tag{5.6}$$

The idea of constraint (5.6) is the same as that of the first constraint. Since each

vertex w in the set $N_h(v)$ can be assigned to one label only, the label of v must be constrained from the smallest label of a vertex in $N_h(v)$. This can be done by first sorting values l_w non-decreasingly. Then, for each value l_w, let $S = \{t \in N_h(v) : l_t \leqslant l_w\}$. The smallest label of a vertex $t \in S$ is $l_w - (|S| - 1)$, keeping v from being assigned to a label larger than $l_w - (|S| - 1) + h\varphi$ since v is at a distance at most h from w.

Similarly, the pulling constraint for min labels is formulated in (5.7).

$$f_v \geqslant \max\Big\{(f_w + (|S| - 1) - h\varphi) : w \in N_h(v), S = \{t \in N_h(v) : f_t \geqslant f_w\}\Big\} \quad (5.7)$$

In the generalized version of the pulling constraint, we consider the set of free vertices whose distance is at most h from v. The choice of an effective layer h is heuristically selected by finding a layer h_m such that $(l_w - (|S| - 1) + h_m\varphi)$ is smallest, or $(|S| - h_m\varphi)$ is largest. This coincidently happens to be the same layer h_{heu} described in the section on the fitting constraint. Therefore, the pulling constraint can be applied efficiently. The reason is that it can reuse the sorted lists of min and max labels of vertices in $N_{h_{heu}}(v)$, which has been prepared for the computation of the fitting constraint.

Finally, to save the cost of the sorting procedure, we consider only free vertices as in the case of the fitting constraint. Because both fitting and pulling constraints use the same set of free vertices, we need to sort only once. The version of the pulling constraint used is formulated in (5.8) and (5.9).

$$l_v \leqslant \min\Big\{(l_w - (|S| - 1) + h\varphi) : w \in N_h^F(v), S = \{t \in N_h^F(v) : l_t \leqslant l_w\}\Big\}, \quad (5.8)$$

and,

$$f_v \geqslant \max\Big\{(f_w + (|S| - 1) - h\varphi) : w \in N_h^F(v), S = \{t \in N_h^F(v) : f_t \geqslant f_w\}\Big\}, \quad (5.9)$$

where $N_h^F(v) = N_h(v) \cap F$.

5.1.3 The density cut constraint

We review the second constraint by Martí, Campos, and Piñana [36] which is detailed in Section 3.5.4. Given a left partial permutation π^L, the min labels can be tightened by arranging free vertices into layers N_h^L, where N_h^L is the set of vertices whose shortest distance to any vertex $u \in L$ is h. For each layer N_h^L, let $N^L = \{N_h^L \cup N_{h-1}^L \cup \ldots \cup L\}$ and $l_v^c = \max\{l_v : v \in N^L\}$. If $l_v^c = |N^L|$ then we know that vertices in N^L will be assigned to all labels in the range $[1, l_v^c]$, regardless of the assignment order. Therefore, vertices of the remaining layers must be assigned to labels above that range, and their min labels are tightened to $f_v^c + 1$.

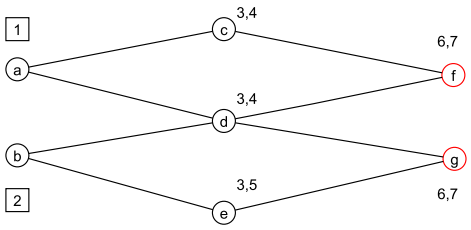


Figure 5.4: Min labels are tightened by the second constraint.

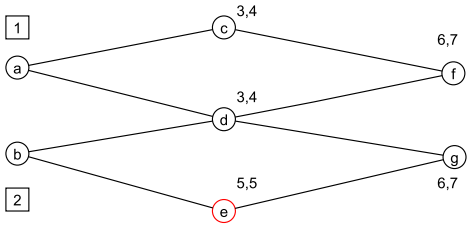


Figure 5.5: Further tightening by the density cut constraint.

Figure 5.4 shows again the example illustrating the second constraint. Here, $L = \{a, b\}$ and $N_1^L = \{c, d, e\}$. Due to $l_v^c = 5$ for $N^L = \{N_1^L \cup L\}$, min labels of vertices in $N_2^L = \{f, g\}$ have been tightened from 3 to 6. On the linear layout, label 5 separates c, d, e to the left side and f, g to the right side. We call such a separation a *cut*. Notice that the label domain of e is unchanged and $f_e = 3$.

To use the second constraint, layers N_h^L must be generated and a cut can be formed if the size of N^L is equal to the largest max label of vertices in this set. The cuts can be found more naturally by sorting vertices according to their max labels and finding label ranges which are fully used by some vertices due to their label domains. This is the idea of the new constraint called *density cut*.

Assume that $\pi^{L,R}$ is a partial permutation whose left set has just been extended and $|L| = k$. Since this tightens max labels of the free vertices, the density cut constraint can be used to strengthen the min labels as follows. First we sort free vertices non-decreasingly according to their max labels. Starting with the smallest l_s upward, for each max label l_s let $S = \{v \in F : l_v \leqslant l_s\}$. If $k + |S| = l_s$, obviously all vertices $v \in S$ must be assigned to labels in the range $[k + 1, l_s]$, regardless of the assignment. Therefore, remaining vertices can only be assigned to a label not less than $l_s + 1$. In this case, a new cut has been found at l_s.

Consider the example in Figure 5.4 again, after sorting free vertices according to their max labels we have three values l_s: 4, 5, and 7. We find a cut immediately at the first max label $l_s = 4$ because $S = \{c, d\}$ and $|L| + |S| = 4$. Since vertex e must be assigned to a label above that range, its min label is tightened to 5 instead

of 3 as in the second constraint. When the next max label $l_s = 5$ is considered, another cut is found, tightening the min labels of f and g to 6. Figure 5.5 show the partial permutation with stronger bounds. Compared with the second constraint, the density cut constraint generates one more cut and tightens the min label of one more vertex.

Computing the density cut constraint requires a sorting procedure. However, in practice it often leads to stronger bounds than does the second constraint since more cuts are generated. The pseudo code for applying the constraint is outlined in Algorithm 5.1.

Algorithm 5.1: The density cut algorithm.

Input: $\pi^{L,R}$, $|L| = k$.
Output: tightened f_v if possible.

```
Sort v ∈ F non-decreasingly according to l_v;
minLabelRange = k + 1;
numSmallerEqual = k;
for each max label l_s in the sorted list do
    let S = {v : l_v = l_s};
    for v ∈ S do
        if f_v < minLabelRange then
            f_v = minLabelRange;
        end if
    end for
    numSmallerEqual += |S|;
    if numSmallerEqual = l_s then
        minLabelRange = numSmallerEqual + 1;
    end if
end for
```

A similar procedure can be used to tighten max labels of a partial permutation $\pi^{L,R}$ whose right side has just been extended. Assume that $|R| = q$. First we sort the min labels of free vertices non-ascendingly. Starting with the largest value downward, for each min label f_s let $S = \{v \in F : f_v \geqslant f_s\}$. If $f_s + (|S| - 1) = n - q$, all vertices $v \in S$ must be assigned to labels in the range $[f_s, n - q]$ and a new cut has been found at f_s. In this case, max labels of the remaining free vertices $v \in \{F \setminus S\}$ will be tightened to $f_s - 1$.

5.1.4 The density near-cut constraint

In certain cases the density cut constraint cannot be used, simply because of the values of the max labels. Figure 5.6 shows such an example. Here, we have a graph of 9 vertices. Vertex a is assigned to label 1 and the remaining free vertices have

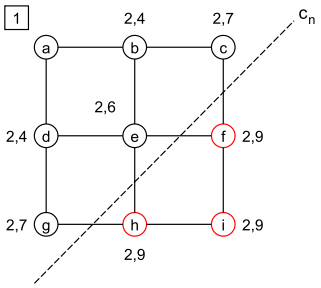


Figure 5.6: A density cut cannot be formed.

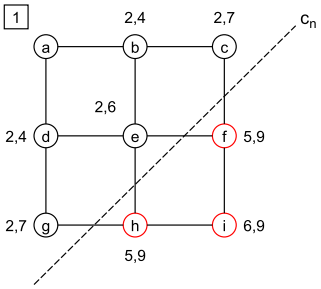


Figure 5.7: Tightening by the density near-cut constraint.

label domains as shown. After sorting vertices according to their max labels, we have max labels 4, 6, 7, and 9. Let us consider $f_s = 7$ and the set $S = \{b, d, e, c, g\}$, where $f_v \leqslant f_s : v \in S$. Since $k + |S| = 1 + 5 < f_s$, a density cut cannot be formed. Similar cases are observed for other max labels.

We notice that the gap between the cut f_s and $k + |S|$ is only 1. Therefore, a cut can be created if a free vertex not in S is assigned to a label not greater than f_s. We call that a *near-cut*, graphically illustrated by the dashed line $c_n = 7$ on the graph.

In the example in Figure 5.6, if vertex f is assigned to a label not greater than 7, a cut will be formed. In this case, the smallest label such that i can be assigned to is 8 because the smaller labels has been fully used by f and $v \in S$. Since i is adjacent to f, it constrains the min label of f to 5. Therefore, f_f can be tightened to 5 instead of 2, as shown in Figure 5.7. Vertices h and i are tightened similarly. It is even stronger for i. Since i has two adjacent vertices on the right of the near-cut, one of them must be assigned to label 9, which constrains f_i to 6.

In the implementation, near-cuts c_n are located by finding label ranges which still lack one vertex to form a density cut. For each vertex v above the label range, i.e., those whose max labels are larger than c_n, the number of its adjacent vertices which are also outside the range are counted, and the new min label of v is constrained to:

$f_v = \max\{f_v, c_n + |S| - \varphi\}$, where $S = \{w : w \in N_1(v) \text{ and } l_w > c_n\}$.

The pseudo code for the density near-cut constraint is outlined in Algorithm 5.2. Notice that the *near-cuts* are tried in a descending order; if a vertex v has been tightened with a large near-cut, all the later tightenings can hardly be better because the later near-cuts are always smaller. Therefore if a vertex has ever been tightened, it should not be tried again.

Algorithm 5.2: The density near-cut algorithm.

Input: $\pi^{L,R}$.
Output: tightened f_v if possible.

```
1:  for all v ∈ F: tightened[v] = false;
2:  find all near-cuts c_n and put into set C_n in a descending order;
3:  for each near cut c_n ∈ c_n do
4:    for v ∈ F do
5:      if tightened[v] = true then
6:        continue;
7:      end if
8:      S = {w : w ∈ N_1(v) and l_w > c_n};
9:      if |S| > 0 then
10:       newMinLabel = min{n, c_n + |S| − φ};
11:       if f_v < newMinLabel then
12:         f_v = newMinLabel;
13:         tightened[v] = true;
14:       end if
15:     end if
16:   end for
17: end for
```

The density near-cut constraint can be considered as a supplementary version of the density cut constraint and both should be used together to utilize the sorted list and the procedure for detecting cuts and near-cuts. Having free vertices sorted according to their max labels, for each label range the condition of a real cut is tested. If a cut is found, max labels of the remaining vertices are tightened. Otherwise, the label range is checked if it lacks only one element to form a cut. If so, a near-cut is found and recorded in the list N_c. After the tightening procedure of the density cut constraint is finished, max labels are tightened using near-cuts in a descending order. In Algorithm 5.2, this starts at line 3.

Like in the case of the density cut constraint, a procedure similar to what described above can be used to strengthen the max labels l_v when the partial permutation is extended on the right side.

5.1.5 Branch-and-bound algorithms with constraints

The new constraints can easily be used in a branch-and-bound algorithm for BW like Algorithm 3.4. For efficiency, these constraints are hierarchically applied, one by one in the order of their computational complexity. Extendability of the input partial permutation is tested after applying a group of constraints using the same data. In particular, extendability test is called after updating basic bounds using the definitions of l_v and f_v, after applying the pulling constraint and the fitting constraint, and finally after applying the density cut constraint and the density near-cut constraint.

The pseudo code is outlined in Algorithm 5.3. Compared with Algorithm 3.4, only the function *testAndExtend* is changed to include the new constraints, other functions remain the same.

Algorithm 5.3: Applying constraints in a branch-and-bound algorithm.

```
bool testAndExtend(π^{L,R}, v, φ, side)
begin
  updateBasicBounds(π^{L,R}, v, φ, side);
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  end if
  apply pulling constraint and fitting constraint on side;
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  end if
  apply density cut constraint and density near-cut constraint on side;
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  end if
  return labeling(π^{L,R}, φ);
end
```

5.1.6 Tightening bounds from the side not being extended

In the work of Caprara and Salazar-González [6] and the later improvement by Martí, Campos, and Piñana [36], the constraints are applied only once on the side where the partial permutation is extended. With the introduction of the density cut and the density near-cut constraints, we observe that when a partial permutation is extended on the left, which can only update max labels l_v by definition, its min labels f_v can also be tightened. Therefore, after applying the density cut and the density near-cut constraints we check if there is any update made to the min labels.

If this is the case then the pulling constraint and the fitting constraint are applied again for tightening bounds, but on the other side of the partial permutation.

This procedure is applied similarly when a partial permutation is extended on the right side. The general pseudo code is outlined in Algorithm 5.4.

Algorithm 5.4: Tightening bounds from the side not being extended.

```
bool testAndExtend(π^{L,R}, v, φ, side)
begin
  updateBasicBounds(π^{L,R}, v, φ, side);
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  end if
  apply pulling constraint and fitting constraint on side;
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  end if
  apply density cut constraint and density near-cut constraint on side;
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  end if
  if density cut or density near-cut made any update on side then
    if side = LEFT then
      otherSide = RIGHT;
    else
      otherSide = LEFT;
    end if
    apply pulling constraint and fitting constraint on otherSide;
    if extendabilityTest(π^{L,R}, φ) = false then
      return false;
    end if
  end if
  return labeling(π^{L,R}, φ);
end
```

5.2 The dominance relation

In this section we analyze the relation between certain partial permutations and show that some partial permutations are dominated by others, and can therefore be eliminated in a branch-and-bound search tree. The data structure and algorithms to employ this relation in a branch-and-bound algorithm are then presented.

5.2.1 The dominance concept

In the BW problem for a graph G, all possible permutations must be tested to prove that G does not have a bandwidth φ. On the other hand, it is sufficient to conclude that G has such a bandwidth if one can find a feasible permutation π such that $|\pi(i) - \pi(j)| \leqslant \varphi$ for all $ij \in E$. Among two partial permutations having the same assigned set, in certain cases it is guaranteed that if a partial permutation can be extended to a feasible permutation then so can the other. This leads to an interesting property of the BW problem: the dominance relation between partial permutations having the same assigned set. It is defined as follows.

Definition 5.1
π_1^L and π_2^L are two left partial permutations having the same assigned set L. If the fact that π_2^L can be extended to a feasible permutation also guarantees that π_1^L can be extended to a feasible permutation, then π_1^L is said to dominate *π_2^L.* △

In many places in this section we illustrate the dominance relation between two partial permutations. Figure 5.8 shows such an example. Here, two left partial permutations π_1^L and π_2^L have the same assigned set $L = \{a, b, c, d\}$. The linear layout is depicted by the horizontal dashed line with labels $1, 2, \ldots$ drawn below the line. Vertices are depicted as circles on this line, each with its vertex number above the circle. Edges are curved lines connecting adjacent vertices. Consider π_1^L, we have $\pi_1^L(a) = 4$, $\pi_1^L(b) = 3$, $\pi_1^L(c) = 2$, and $\pi_1^L(d) = 1$. There is a vertical dashed line separating the assigned set L and the free set F. (In Figure 5.8, this separator is on the right of vertex a.) We do not show the vertex numbers of free vertices because it is not necessary.

In this section, we always assume that a partial permutation already satisfies the basic bandwidth constraint on vertices in the assigned set. For left partial permutations, this means that $|\pi^L(i) - \pi^L(j)| \leqslant \varphi$, $ij \in E$ for all $i, j \in L$. For both-sided partial permutations, $|\pi^L(i) - \pi^L(j)| \leqslant \varphi$, $ij \in E$ for all $i, j \in L, R$. This condition can be met in the outlined branch-and-bound algorithms, where violated partial permutations have been eliminated.

Proposition 5.1
π_1^L and π_2^L are two left partial permutations having the same assigned set L. If $\pi_1^L(v) \geqslant \pi_2^L(v)$ for all $v \in \{L \cap N_1(F)\}$ then π_1^L dominates *π_2^L.* △

Proposition 5.1 defines the *dominance rule* for two left partial permutations having the same left set. It states that the dominance relation is determined by the set of assigned vertices $v \in L$ which are adjacent to a free vertex. The idea is illustrated in Figure 5.8, in which partial permutation π_1^L dominates π_2^L. Here, such a set $L \cap N_1(F) = \{a, b, c\}$.

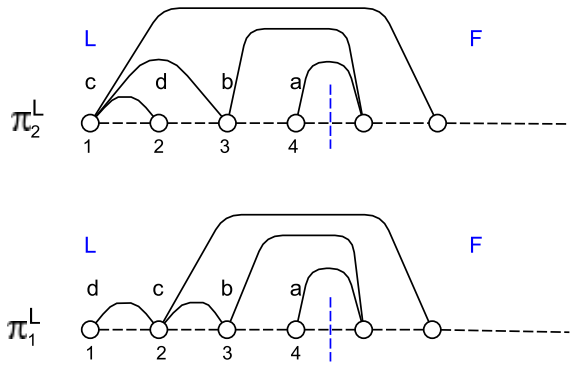


Figure 5.8: The dominance relation between two partial permutations.

Proof Assuming that the partial permutation π_2^L can be extended to a feasible permutation π such that $\pi(v) = \pi_2^L(v) : v \in L$, it follows that π_1^L can also be extended to a feasible permutation π_1 such that:

$$\pi_1(v) = \begin{cases} \pi_1^L(v) & \text{if } v \in L, \\ \pi(v) & \text{if } v \in F. \end{cases}$$

We need to show that for any pair of adjacent vertices i and j the difference of their labels in π_1 is not larger than φ. Both vertices can be either in the same set L or F, or one in L and the other in F.

First, by the assumption we have $|\pi_1^L(i) - \pi_1^L(j)| \leqslant \varphi$ for all $i, j \in L$.

Since π is a feasible permutation itself, $|\pi_1(i) - \pi_1(j)| = |\pi(i) - \pi(j)| \leqslant \varphi$ for all $i, j \in F$.

If $\pi_1^L(v) \geqslant \pi_2^L(v) : v \in N_1(F) \cap L$, for $j \in L$ and $i \in F$ we have $\pi_1(i) - \pi_1(j) \leqslant \pi_1(i) - \pi_2^L(j) = \pi(i) - \pi(j) \leqslant \varphi$. Since the bandwidth constraint is satisfied in all cases, π_1 is a feasible permutation. □

The dominance relation between two both-sided partial permutations is described as follows.

Definition 5.2
$\pi_1^{L,R}$ and $\pi_2^{L,R}$ are two both-sided partial permutations having the same assigned sets L and R. If the fact that $\pi_2^{L,R}$ can be extended to a feasible permutation also guarantees that $\pi_1^{L,R}$ can be extended to a feasible permutation, then $\pi_1^{L,R}$ is said to dominate *$\pi_2^{L,R}$.* △

Proposition 5.2
$\pi_1^{L,R}$ and $\pi_2^{L,R}$ are two both-sided partial permutations having the same assigned sets

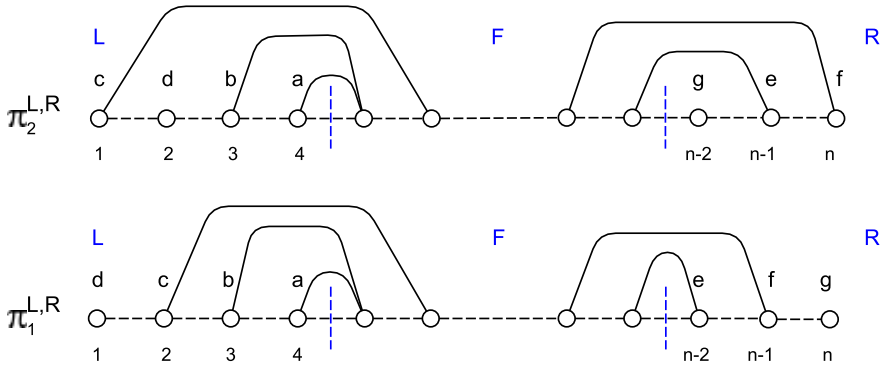


Figure 5.9: The dominance relation between both-sided partial permutations.

L and R. If $\pi_1^{L,R}(v) \geqslant \pi_2^{L,R}(v) : v \in \{L \cap N_1(F)\}$ *and* $\pi_1^{L,R}(v) \leqslant \pi_2^{L,R}(v) : v \in \{R \cap N_1(F)\}$ *then* $\pi_1^{L,R}$ dominates $\pi_2^{L,R}$. △

Extending Proposition 5.1, Proposition 5.2 defines the *dominance rule* for two both-sided partial permutations having the same assigned set. It also states that the set of assigned vertices which are adjacent to a free vertex determines the dominance relation between the two partial permutations. The idea is illustrated in Figure 5.9, in which $\pi_1^{L,R}$ dominates $\pi_2^{L,R}$. For simplicity, we only draw edges connecting an assigned vertex and a free one, not between assigned vertices. Thus two vertices d and g stand on their own.

Consider the example in Figure 5.9, the sets of interest are $\{a, b, c\}$ on the left and $\{e, f\}$ on the right. We observe that in $\pi_1^{L,R}$, labels of vertices in these sets are either equal to or closer to the remaining free labels than those of the same vertices in $\pi_2^{L,R}$. As a result, the label difference between a vertex $v \in L \cup R$ and a vertex $u \in F$ in $\pi_1^{L,R}$ is not larger than that in $\pi_2^{L,R}$. Therefore, if $\pi_2^{L,R}$ can be extended to a feasible permutation then so can $\pi_1^{L,R}$. The formal proof follows.

Proof Proposition 5.2 is proved similarly to Proposition 5.1. If $\pi_2^{L,R}$ can be extended to a feasible permutation π such that $\pi(v) = \pi_2^{L,R}(v) : v \in \{L \cup R\}$, it follows that $\pi_1^{L,R}$ can also be extended to a feasible permutation π_1 such that:

$$\pi_1(v) = \begin{cases} \pi_1^{L,R}(v) & \text{if } v \in L, \\ \pi_1^{L,R}(v) & \text{if } v \in R, \\ \pi(v) & \text{if } v \in F. \end{cases}$$

We need to show again that in π_1 the bandwidth constraint between two adjacent vertices i and j is always satisfied. The two vertices can either be in the same set, assigned or free, or one vertex is assigned and the other is free.

First, by the assumption in the branch-and-bound algorithm, for all $i, j \in \{L \cup R\}$ we have $|\pi_1(i) - \pi_1(j)| \leqslant \varphi$. Since π is a feasible permutation, $|\pi_1(i) - \pi_1(j)| = |\pi(i) - \pi(j)| \leqslant \varphi$ for all $i, j \in F$.

If $\pi_1^{L,R}(v) \geqslant \pi_2^{L,R}(v) : v \in \{L \cap N_1(F)\}$, for all $i \in F, j \in L$ we have $\pi_1(i) - \pi_1(j) \leqslant \pi_1(i) - \pi_2^{L,R}(j) = \pi(i) - \pi(j) \leqslant \varphi$.

Similar result holds for the right side. If $\pi_1^{L,R}(v) \leqslant \pi_2^{L,R}(v) : v \in \{R \cap N_1(F)\}$, for all $i \in F, j \in R$ we have $\pi_1(j) - \pi_1(i) \leqslant \pi_2^{L,R}(j) - \pi_1(i) = \pi(j) - \pi(i) \leqslant \varphi$.

Since in π_1 the bandwidth constraint is satisfied between any two adjacent vertices, π_1 is a feasible permutation. □

5.2.2 The hash-table

Employing the dominant concept in a branch-and-bound algorithm requires the following information. First, given an assigned set one needs to know the *most dominating* partial permutation, i.e., the one which dominates others in the group of partial permutations having this assigned set. Second, how this permutation can be accessed later for analyzing its dominance relation with others. We describe the necessary data structure as follows.

A unique set of vertices needs to be mapped to a unique integer. We do that by encoding each vertex $v \in 1 \ldots n$ as an integer $p_v = 2^{v-1}$. A set L of vertices is encoded into a unique integer $p_L = \sum_{v \in L} p_v$. We can also imagine p_L as the value of an unsigned integer composed of n bits, in which the vertex numbers of vertices $v \in L$ define the positions of 1-bits. For example, $p_L = 5$ for $L = \{1, 3\}$.

For each group of partial permutations $\pi^{L,R}$ having the same left set L and right set R, we keep track of the most dominating one by maintaining its *pointer* (address in the memory) in a cell of a two-dimensional array called *hash-table*. The cell is addressed in the array by two indices: one is the hashed value of the integer encoding the left set and the other is that of the right set. Let S_l be the predefined size of the dimension for the left set and similarly S_r for the right set, the hashing function for a left set L is defined as follows:

$$h_l(L) = p_L \bmod S_l = \sum_{v \in L} 2^v \bmod S_l, \tag{5.10}$$

and for a right set R:

$$h_r(R) = p_R \bmod S_r = \sum_{v \in R} 2^v \bmod S_r \tag{5.11}$$

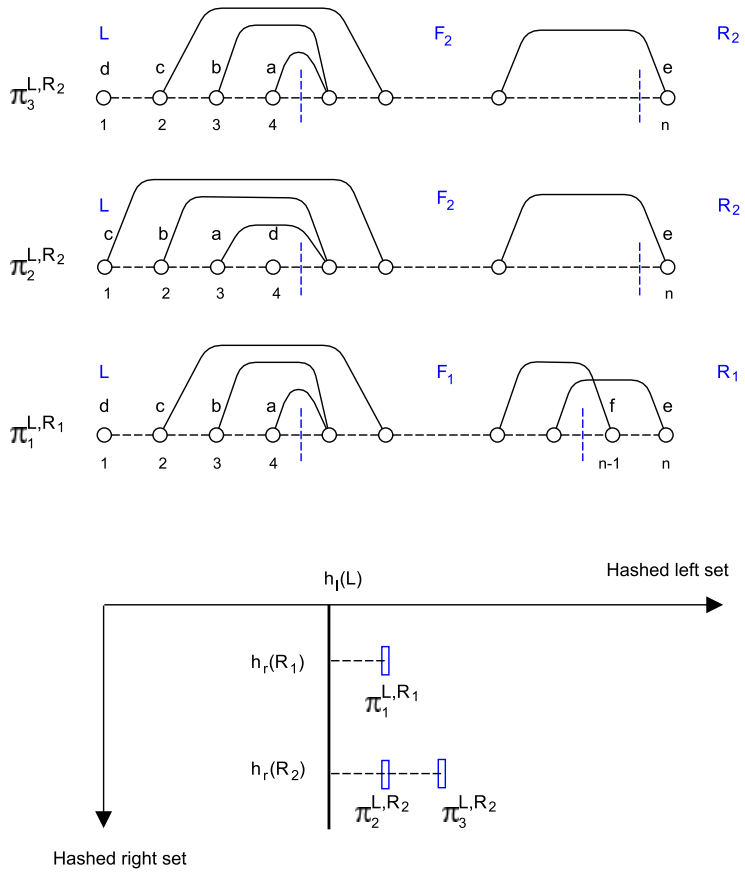


Figure 5.10: An instance of the hash-table.

The hash-table is initially constructed with only $|S_l|$ pointers to cells. Only until the first partial permutation having a hashed left set as $h(L)$ is inserted into the hash-table that a column containing S_r cells is created at $h_l(L)$. In this way computer memory can be saved until it is actually needed. In the hash-table, a cell at position $(h_l(L), h_r(R))$, if ever created, contains a *list* of pointers of partial permutations whose hashed value of the left set is $h_l(L)$ and of the right set is $h_r(R)$.

In our implementation, we set $S_l = 49,999$ and $S_r = 999$. They are experimental numbers and should be tuned for an efficient operation of the hash-table according to the size of instances. S_l is larger than S_r because we choose the left side to be the main side in our case. Recalling that at the branching step in the branch-and-bound algorithm, if sizes of two candidate sets for both sides are equal then the left side is selected for extension. It also means that the label of the first assigned vertex is always 1. Therefore, more space for hashed left sets will be more efficient.

Figure 5.10 shows how the pointers of three partial permutations π_1^{L,R_1}, π_2^{L,R_2}, and π_3^{L,R_2} are stored in the hash-table. Since they all have the same left set, the cells containing their pointers are located at left set index $h_l(L)$. π_2^{L,R_2} and π_3^{L,R_2}

have the same right set R_2, thus their pointers are stored in the same list at cell $(h_l(L), h_r(R_2))$. The pointer of π_1^{L,R_1} stands alone. In the picture, the pointers are depicted as empty rectangles.

5.2.3 Algorithms

With the dominance concept and the data structure outlined above, we can now describe the algorithm for using them in a branch-and-bound algorithm for BW.

During execution, each newly generated subproblem in a branch-and-bound tree, or a new partial permutation $\pi^{L,R}$, is checked against the hash-table. How the new subproblem is processed depends on the result of this check.

First, the assigned set of $\pi^{L,R}$ is checked if it has ever been tracked in the hash-table If not, the result is *Non-existent* and the pointer of $\pi^{L,R}$ is inserted into the hash-table. The branch-and-bound algorithm proceeds as usual. Otherwise, we check the dominance relation between $\pi^{L,R}$ and the reference partial permutation $\pi_m^{L,R}$ stored in the hash-table. With our storing scheme $\pi_m^{L,R}$ is the partial permutation that dominates the others which also have the same assigned set until that time.

The dominance relation between $\pi^{L,R}$ and $\pi_m^{L,R}$ is determined according to the dominance rule in Proposition 5.2. For each assigned vertex which is adjacent to a free vertex, the algorithm verifies if its label in $\pi^{L,R}$ is equal or closer to the remaining free labels compared with its label in $\pi_m^{L,R}$. If at least one label is closer and the remaining are the same, the result is *Dominating*, meaning that $\pi^{L,R}$ dominates $\pi_m^{L,R}$. If at least one is further away and the others are the same, the result is *Dominated*. If some label is closer and some is further, an *Inconsistent* case is found. Otherwise, it is *Undecided*.

Depending on the dominance relation result between $\pi^{L,R}$ and $\pi_m^{L,R}$, the branch-and-bound algorithm processes $\pi^{L,R}$ as follows:

1. *Dominated*: $\pi^{L,R}$ is eliminated because it can never lead to a better permutation than can $\pi_m^{L,R}$.
2. *Dominating*: $\pi^{L,R}$ replaces the reference in the hash-table.
3. *Undecided*, or *Inconsistent*: The pointer of $\pi^{L,R}$ is inserted into the hash-table.

The modification of function *testAndExtend*$(\pi^{L,R}, v, \varphi, side)$ to include the dominance concept is outlined in Algorithm 5.5. Similar to the way the constraints are applied, the dominance relation is also processed hierarchically in two steps.

In the first step, the new partial permutation is only checked if it is dominated by the reference, if so it is fathomed. This is done by function *lightProcessDominance*, outlined in Algorithm 5.6, in which neither updating nor inserting to the hash-table is allowed.

Only after $\pi^{L,R}$ passes all the extendability tests, including the bound tightening procedures with constraints, that it can be processed by the function *processDominance*, outlined in Algorithm 5.7. Here, $\pi^{L,R}$ is inserted into the hash-table if its assigned set has not been there, or if an inconsistency is found. If $\pi^{L,R}$ dominates the reference partial permutation then it replaces the reference in the hash-table. The function *checkDominance*($\pi^{L,R}$, $\pi_m^{L,R}$) determines the dominance relation between $\pi^{L,R}$ and $\pi_m^{L,R}$ as described above.

Algorithm 5.5: The dominance relation in a branch-and-bound algorithms.

```
bool testAndExtend (π^{L,R}, v, φ, side)
begin
  updateBasicBounds(π^{L,R}, v, φ, side);
  {only check hash-table and fathom π^{L,R} if it is dominated }
  if lightProcessDominance(π^{L,R}) = Dominated then
    return false;
  end if
  if extendabilityTest(π^{L,R}, φ) = false then
    return false;
  else
    {apply appropriate constraints, each followed by a feasibility test}
    apply constraints;
    if extendabilityTest(π^{L,R}, φ) = false then
      return false;
    end if
    {check π^{L,R} and update to hash-table if necessary}
    if processDominance(π^{L,R}) = Dominated then
      return false;
    else
      return labeling(π^{L,R}, φ);
    end if
  end if
end
```

5.3 The 2-labeling scheme

In this section we present a new branching scheme named *2-labeling* for branch-and-bound algorithms for the BW problem. We first introduce the concept and then describe the algorithm.

Algorithm 5.6: Light process of the dominance relation.

Input: A partial permutation $\pi^{L,R}$.
Output: Dominance relation between $\pi^{L,R}$ and the reference.
integer **lightProcessDominance** ($\pi^{L,R}$)
begin
  if (L, R) does not exist in hash-table **then**
    return Non-existent;
  end if
  {get the most dominating partial permutation having L, R}
  $\pi_m^{L,R}$ = HashTable($h_l(L)$, $h_r(R)$);
  {only check the dominance relation, no update}
  dominance = checkDominance($\pi^{L,R}$, $\pi_m^{L,R}$);
  return dominance;
end

Algorithm 5.7: Processing the dominance relation.

Input: A partial permutation $\pi^{L,R}$.
Output: Dominance relation between $\pi^{L,R}$ and the reference. Hash-table is updated if necessary.
integer **processDominance** ($\pi^{L,R}$)
begin
  if (L, R) does not exist in hash-table **then**
    insert pointer of $\pi^{L,R}$ into HashTable($h_l(L)$, $h_r(R)$);
    return Non-existent;
  end if
  {get the most dominating partial permutation having (L, R)}
  $\pi_m^{L,R}$ = HashTable($h_l(L)$, $h_r(R)$);
  dominance = checkDominance($\pi^{L,R}$, $\pi_m^{L,R}$);
  if dominance = Inconsistent **then**
    insert pointer of $\pi^{L,R}$ into HashTable($h_l(L)$, $h_r(R)$);
    return Inconsistent;
  end if
  if dominance = Dominating **then**
    {update $\pi^{L,R}$ as the most dominating in the hash-table}
    replaceHashTable($h_l(L)$, $h_r(R)$, $\pi^{L,R}$);
  end if
  return dominance;
end

5.3.1 The 2-labeling concept

In previous branch-and-bound algorithms which are based on the partial permutation concept, partial permutations are extended with one vertex at the branching step as outlined in Algorithm 3.4. In principle the extension can be done with two or more vertices. This is the idea of the *multi-labeling* scheme, or *s-labeling* where s is the number of vertices used to extend a partial permutation at each branching step. We discuss the advantage as well as the drawback of this scheme, and report our realization for a *2-labeling* scheme for the BW problem.

In the traditional one-by-one extension called *single-labeling*, if an extendability test was not used for fathoming subproblems, the complexity for solving BW would be $O(n!)$.

If the extendability test was not used in the case of the *2-labeling* scheme, there would be $n(n-1)/2$ possibilities for the first two labels, multiplying $(n-2)(n-3)/2$ for the next two labels, and so on. Of course, for each pair of vertices one has to try twice to decide the vertex for the smaller label, so in the worst-case the complexity will end up being the same as that of the single-labeling scheme. However, if there is a way to determine in one step the labels of the two vertices used for extending, the complexity would be reduced to approximately $O(\frac{n!}{2^{n/2}})$. With the introduction of the dominance rule in Section 5.2, the idea becomes feasible. We may not reach that ideal improvement, but we can expect a reduced number of subproblems in the branch-and-bound tree. This turns out to be the case for many real instances.

The drawback of the 2-labeling scheme is the rapid increase of memory usage. Imagine that the branch-and-bound tree grows with a width of n subproblems in the single-labeling scheme, it becomes $n(n-1)/2$ in case of 2-labeling scheme. Therefore, to use 2-labeling in a branch-and-bound algorithm one needs strong dominance rules in combination with a compact data structure.

To employ the 2-labeling scheme in a branch-and-bound algorithm, we need to state the dominance rule in Section 5.2 again, but in a slightly different context. We start with the extension of a partial permutation on the left side.

Proposition 5.3
$\pi^{L,R}$ is a both-sided partial permutation, $|L| = k$, $|R| = q$, and two vertices $v, w \in F$ will be assigned to the next labels on the left $k+1, k+2$. Let $\pi_1^{L',R}$ and $\pi_2^{L',R}$ be two extended partial permutations, $\pi_1^{L',R}(v) = k+2, \pi_1^{L',R}(w) = k+1$ and $\pi_2^{L',R}(v) = k+1, \pi_2^{L',R}(w) = k+2$, where $L' = L \cup \{v, w\}$ and $\pi_1^{L',R}(u) = \pi_2^{L',R}(u) = \pi^{L,R}(u)$ for all $u \in L \cup R$. If v is adjacent to a free vertex and w is not, then $\pi_2^{L',R}$ can be eliminated in the branch-and-bound tree. △

Proof Since $\pi_1^{L',R}(v) \geqslant \pi_2^{L',R}(v)$ for all $v \in \{L' \cap N_1(F')\}$ and $\pi_1^{L',R}(v) = \pi_2^{L',R}(v)$ for all $v \in \{R \cap N_1(F')\}$, $\pi_1^{L',R}$ dominates $\pi_2^{L',R}$ according to Proposition 5.2. Therefore,

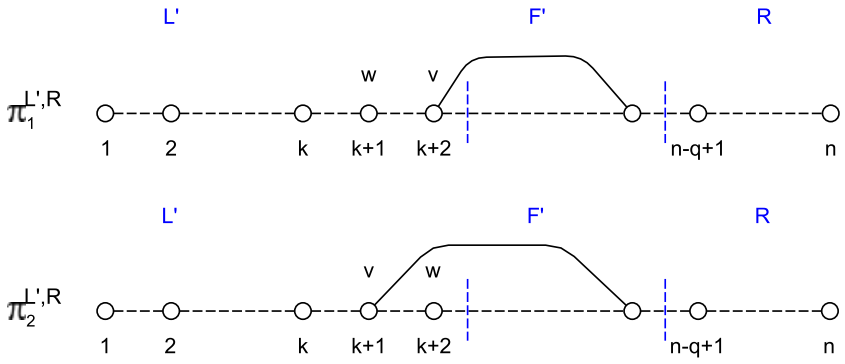


Figure 5.11: Extending partial permutations with the 2-labeling scheme.

$\pi_2^{L',R}$ can be eliminated in the branch-and-bound tree. □

The idea of Proposition 5.3 is illustrated in Figure 5.11, in which the extended partial permutation $\pi_1^{L',R}$ dominates $\pi_2^{L',R}$. The illustration convention is almost the same as in the previous section, except that we only show the vertex numbers of vertices v, w used for the extension and edges connecting them to free vertices.

The 2-labeling scheme is used similarly for extending partial permutations on the right side. The dominance rule is stated in Proposition 5.4.

Proposition 5.4
$\pi^{L,R}$ is a both-sided partial permutation, $|L| = k$, $|R| = q$, and two vertices $v, w \in F$ will be assigned to the next labels on the right $n-q-1, n-q$. Let $\pi_1^{L,R'}$ and $\pi_2^{L,R'}$ be two extended partial permutations, $\pi_1^{L,R'}(v) = n-q-1, \pi_1^{L,R'}(w) = n-q$ and $\pi_2^{L,R'}(v) = n-q, \pi_2^{L,R'}(w) = n-q-1$, where $R' = R + \{v, w\}$ and $\pi_1^{L,R'}(u) = \pi_2^{L,R'}(u) = \pi^{L,R}(u)$ for all $u \in L \cup R$. If v is adjacent to a free vertex and w is not, then $\pi_2^{L,R'}$ can be fathomed in the branch-and-bound tree. △

Proof Proposition 5.4 is proved similarly to Proposition 5.3. □

5.3.2 The 2-labeling algorithm

A branch-and-bound algorithm using the 2-labeling scheme is basically the same as those using the single-labeling scheme, except for the branching procedure.

After the candidate set and the side for extending a partial permutation have been determined at the branching step, *unique pairs* of vertices from the candidate set are selected for being assigned to the next two labels. For each pair, if a vertex v is adjacent to any of the remaining free vertices and the other is not then v is assigned to the label closer to the free set, according to the rule in Propositions 5.3 or

Propositions 5.4. In this case only one new partial permutation is created. Otherwise two partial permutations are generated, each includes an order of these two vertices. Of course if the candidate set has only one vertex, it will be assigned to the next label as in the single-labeling scheme. The pseudo code is outlined in function *Labeling* in Algorithm 5.8.

The procedures for initializing the branch-and-bound tree's root vertex, updating basic bounds, applying constraints, and testing extendability are the same as those in the single-labeling scheme. For the reader's convenience, they are outlined again in Algorithm 5.9. One can also use the hash-table with the dominance rule, but for the focus on the 2-labeling scheme we do not include it here. There is a small change in the pseudo code. The function *testAndExtend*($\pi^{L,R}, U, \varphi$, *side*) now accepts a set U of newly assigned vertices in $\pi^{L,R}$ instead of only one vertex as in the single-labeling scheme.

Algorithm 5.8: Branching with the *2-labeling* scheme.

```
bool labeling(π^{L,R}, φ)
begin
  if |L| + |R| = |V| then
    return true;
  end if
  k = |L|; q = |R|;
  leftCandidates = {v ∈ F : f_v ⩽ nextLeftLabel};
  rightCandidates = {v ∈ F : l_v ⩾ nextRightLabel};
  if direction = LeftToRight or |leftCandidates| ⩽ |rightCandidates| then
    if |leftCandidates| = 1 then
      π^e = π^{L,R}; π^e(v) = k + 1 : v ∈ leftCandidates;
      return testAndExtend(π^e, {v}, φ, LEFT);
    else
      for each unique pair {v, w} ∈ leftCandidates do
        if v ∩ N_1(F) ≠ ∅ and w ∩ N_1(F) = ∅ then
          π^e = π^{L,R}; π^e(w) = k + 1; π^e(v) = k + 2;
          return testAndExtend(π^e, {v, w}, φ, LEFT);
        else
          π^{e_1} = π^{L,R}; π^{e_1}(v) = k + 2; π^{e_1}(w) = k + 1;
          testAndExtend(π^{e_1}, {v, w}, φ, LEFT);
          π^{e_2} = π^{L,R}; π^{e_2}(v) = k + 1; π^{e_2}(w) = k + 2;
          return testAndExtend(π^{e_2}, {v, w}, φ, LEFT);
        end if
      end for
    end if
  else
    if |rightCandidates| = 1 then
      π^e = π^{L,R}; π^e(v) = n − q : v ∈ rightCandidates;
      return testAndExtend(π^e, {v}, φ, LEFT);
    else
      for each unique pair {v, w} ∈ rightCandidates do
        if v ∩ N_1(F) ≠ ∅ and w ∩ N_1(F) = ∅ then
          π^e = π^{L,R}; π^e(w) = n − q; π^e(v) = n − q − 1;
          return testAndExtend(π^e, {v, w}, φ, RIGHT);
        else
          π^{e_1} = π^{L,R}; π^{e_1}(v) = n − q; π^{e_1}(w) = n − q − 1;
          testAndExtend(π^{e_1}, {v, w}, φ, RIGHT);
          π^{e_2} = π^{L,R}; π^{e_2}(v) = n − q − 1; π^{e_2}(w) = n − q;
          return testAndExtend(π^{e_2}, {v, w}, φ, RIGHT);
        end if
      end for
    end if
  end if
end
```

Algorithm 5.9: The 2-labeling scheme in a branch-and-bound algorithm.

Input: An undirected graph $G = (V, E)$ and an integer φ.
Output: True if G has a bandwidth φ and false otherwise.
bool **bandwidth**(G, φ)
begin
  direction = BothSides;
  initialize $\pi^{L,R}$: $L = \emptyset$; $R = \emptyset$; $F = V$;
  for each $v \in V$ **do**
    $f_v = 1$; $l_v = n$;
  end for
  return labeling($\pi^{L,R}$, φ);
end
{testing extendability of $\pi^{L,R}$. Extend further if no violation found}
bool **testAndExtend**($\pi^{L,R}, U, \varphi$, *side*)
begin
  for each newly assigned vertex $v \in U$ **do**
    updateBasicBounds($\pi^{L,R}, v, \varphi$, *side*);
  end for
  if extendabilityTest($\pi^{L,R}, \varphi$) = false **then**
    return false;
  end if
  apply *pulling* constraint and *fitting* constraint on *side*;
  if extendabilityTest($\pi^{L,R}, \varphi$) = false **then**
    return false;
  end if
  apply *density cut* constraint and *density near-cut* constraint on *side*;
  if extendabilityTest($\pi^{L,R}, \varphi$) = false **then**
    return false;
  end if
  return labeling($\pi^{L,R}, \varphi$);
end
{Update bounds l_v and f_v by definitions }
void **updateBasicBounds**($\pi^{L,R}$, v, φ, *side*)
begin
  if *side* = LEFT **then**
    $\forall u \in F$: update $l_u = \min\{l_u, \pi^{L,R}(v) + \varphi d(v, u)\}$;
  else
    $\forall u \in F$: update $f_u = \max\{f_u, \pi^{L,R}(v) - \varphi d(v, u)\}$;
  end if
end

6 Implementation and Parallelization

This chapter describes the implementation and the parallelization of our exact algorithms. First, technical details of the solver for the bandwidth problem are outlined. We then discuss the choice of a parallel branch-and-bound framework for our solver and report the realization of our algorithms based on the chosen framework.

Throughout the chapter we use the term *node* to refer to a subproblem generated by the branch-and-bound algorithm.

6.1 A solver for the bandwidth problem

We begin with a review of the branch-and-bound algorithm specific to BW. The choice of a search strategy is then discussed. We conclude the section with the description about a scheme searching for the optimal bandwidth.

6.1.1 The algorithm review

We have described in Chapter 3 that the bandwidth minimization problem can be approached by solving a sequence of search problems. For each φ from 1 to $n-1$, the search problem BW asks whether the graph G has a bandwidth φ. When the search sequence is finished, the optimal solution is the smallest φ for which the answer is "yes". The BW problem can be solved by a branch-and-bound algorithm.

The execution of the branch-and-bound algorithm for BW, whose algorithms are outlined in Chapter 3 and Chapter 5, can be illustrated as follows. For simplicity, we use the traditional single-labeling scheme which is detailed in Algorithm 3.4. The partial permutations are always extended on the left side. One can imagine a similar execution for other configurations such as applying the constraints, using the 2-labeling scheme, and/or exploiting the dominance rule with the hash-table.

An example branch-and-bound tree is illustrated in Figure 6.1. The search tree is initialized with a root node having an empty permutation, meaning that none of the vertices is assigned. From the root node, a number of n subproblems are generated

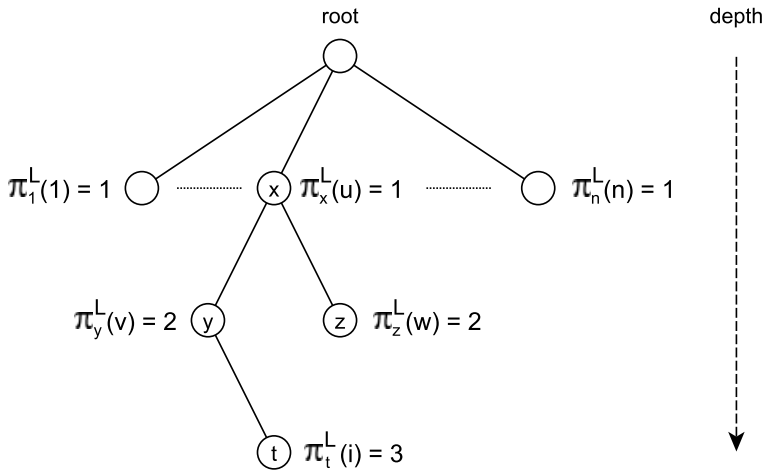


Figure 6.1: A branch-and-bound search tree.

and added to the queue, each is associated with a left partial permutation π^L in which each vertex is assigned to label 1. In the tree, these nodes are of depth 1.

A node, or a subproblem associated with a partial permutation, is selected from the queue and *processed* by applying the constraints and testing the extendability of the partial permutation. If the node fails this test then it is *fathomed*. If no violation is detected, this node is *split* into *child* nodes, each extending the partial permutation by assigning a candidate vertex to the next label on the left. In the tree in Figure 6.1, we see that node x is split into child nodes y and z, whose partial permutations have two assigned vertices. We consider the partial permutation of node y. Here, vertex u is assigned to label 1 and the new vertex v is assigned to label 2.

This procedure is repeated until a decision is reached. In the case the graph G has a bandwidth φ, there is one path on the tree that constitutes a feasible permutation. That path has a length of exactly n and so does the height of the search tree. The search is finished here because the answer has been found. If G does not have a bandwidth φ, all paths in the search tree will be traversed and none of them will have a length of n.

The differences of the branch-and-bound algorithm for BW from the relaxation-based method can be summarized as follows. First, if the graph G has a bandwidth φ then a feasible permutation should be returned as quickly as possible. On the other hand, the full search space needs to be examined to prove that the answer is "no". In the case of the relaxation-based method, the optimal solution is always sought throughout the full search space.

The second difference is that we do not fathom nodes using bounds obtained by solving LP relaxations. Instead we use the constraints of the bandwidth problem

and the extendability test. BW is a search problem and we can derive neither lower bound nor upper bound of a node, or the associated partial permutation, during the search. This may be an issue for the deployment of some search strategies. Except for these two differences, the characteristics of a branch-and-bound algorithm for BW are the same as those of the relaxation-based method. Therefore, we can realize the algorithms for BW using standard branch-and-bound frameworks.

6.1.2 The search strategy

The order in which a node is selected for processing among other candidates in the queue is defined by the search strategy. It has a large influence on the performance of the solver in both non-parallel and parallel modes. A detailed survey of search strategies for branch-and-bound methods can be found in Linderoth and Savelsbergh [35].

The basic strategies are breadth-first, best-first, and depth-first. The *breath-first* approach processes and expands all nodes at each depth in the search tree before proceeding to nodes in the next depth. Since it requires much memory while cannot find feasible solutions efficiently, this approach is rarely used in branch-and-bound algorithms. The *best-first* strategy selects the best node, i.e., the subproblem with the smallest lower bound in the queue. This reduces the unnecessary processing of nodes whose lower bounds are larger than the optimal value. However, the search tree tends to expand on different branches and thus requires more memory.

In contrast to best-first search, nodes at the largest depth in the search tree are favored in the *depth-first* approach. This requires less memory than the best-first approach since the number of unprocessed nodes stored in the search tree is small. In the case of parallelization, it has another advantage of saving communication overhead since branching can be done locally. In addition, feasible solutions (often associated with nodes stored at large depths in the tree) can be found quickly with depth-first search. The main disadvantage of this method is that it may wasting time processing nodes whose lower bounds are larger than the optimal value. In the case of the BW problem, if G has a bandwidth φ and the current partial permutation is being extended on the path of a feasible solution then it can be extended to a feasible permutation quickly. However it is hard to get back to a "feasible" path if the search is on a "wrong" path.

The *hybrid* search strategy tries to combine the advantages of both best-first and depth-first searches. An example implementation can be found in Xu et al. [56, 57]. Here, the search is initialized with the node having the smallest lower bound. The search continues expanding the tree using the depth-first strategy until either the current node is fathomed or the depth of the current subtree exceeds a predefined value. The search then resumes to process the best node which is still unprocessed in

the local queue. In this way the algorithm can use memory efficiently while reduces the risk of processing unnecessary nodes and getting "stuck" in "wrong" paths.

Given its advantages the hybrid search strategy is favored for our branch-and-bound algorithms. The problem is that we do not have any lower bound computation for a node. We did try to apply the function *Sigma* (4.4) to compute the lower bound of a partial permutation using only labels of the assigned vertices. However, the solver ran out of memory on some large instances of more than 800 vertices. Therefore, in the end we apply the hybrid search strategy for our branch-and-bound algorithms but skip the use of lower bounds.

6.1.3 Searching for the optimal bandwidth

The optimal solution for the bandwidth minimization problem is obtained by solving a sequence of BW problems, each for a value of φ from 1 to $n-1$. Usually this can be done efficiently by a binary search, but it is not the case for the bandwidth problem. When the search parameter φ is increased by 1, the search tree for BW gets much larger, almost exponentially. The reason is that the constraints for BW are based on the fact that label differences of adjacent vertices in a partial permutation are allowed to be at most φ. When this value is increased, the label domains become larger. Therefore, more partial permutations can pass the extendability test and more candidate nodes are generated. This can be seen more clearly by observing the search space of BW when φ is increased, which will be reported in Section 7.2.2.

For the reason above, in searching for the optimal bandwidth it is more efficient to start with a lower bound and increase φ until a feasible permutation is found for that φ, rather than using binary search. The scheme can be described as follows.

The search is started with the iGPS heuristic, followed by the simulated annealing method SA-σ. After running these two heuristics, we have an initial upper bound. We first improve the upper bound by solving BW for smaller φ in a predefined time limit for upper bounds. If a feasible permutation is found, the upper bound is updated, φ is decreased by 1, and the search continues. Each search is initialized by arranging vertices in an order according to their labels in the last found feasible permutation (i.e., the one with the smallest upper bound). The upper bound improvement procedure stops if the upper bound is equal to the lower bound, i.e., optimality is found. It also stops if no feasible permutation can be found in a search within the time limit. In that case we switch to improve the lower bound.

The initial lower bound is computed by taking the maximum of $\alpha(G)$ in (3.1) and $\gamma(G)$ in (3.2). The search parameter φ is always set to the current lower bound. For each φ, the solver for BW is applied. If a feasible permutation is found, it becomes the optimal solution and the solver stops here. Otherwise, if the solver concludes that G does not have a bandwidth φ then the lower bound is increased by 1. The

upper bound solution becomes the optimal solution if the lower bound is equal to the upper bound. This procedure is continued until either optimality is found or the total running time exceeds a global time limit. In that case the solver stops and accepts a gap between lower and upper bound for the problem instance.

During the improvement of either the upper bound or the lower bound, we always apply the rounding heuristic embedded in the extendability test (detailed in Section 3.4.3) to find new upper bounds. In addition, initializing the search with an initial solution helps to find feasible permutations more quickly, since vertices have been arranged in an order which is near to the feasible solution being searched. For example, if we initialize the solver for BW with a solution obtained by iGPS, the upper bounds for instances `impcol_b` (59 nodes) and `west0156` (156 nodes) can be found in a few seconds while the program without iGPS takes longer than 1 hour.

For solving each BW problem, we use the branch-and-bound algorithms outlined in the previous chapters. They are realized based on a branch-and-bound framework which can run in both parallel or non-parallel modes, as described in the next section.

6.2 The parallel solver

For the time and resource constraints, we want to develop the parallel solver based on a good open-source framework rather than starting all the implementation from scratch. In this section we first review popular parallel frameworks and discuss the choice of a framework suitable for our purposes. We then describe in detail the realization of our algorithms based on the chosen framework.

6.2.1 Parallel branch-and-bound frameworks

The parallel framework must support the hardware and the software configuration of the parallel cluster at the IWR computing center [26] which will be used for our benchmarks. It consists of 156 computer nodes, each equipped with two Dual Core AMD 2.8 GHz processors with 8 GB RAM memory running Debian 4.0. The cluster uses Myricom 10G for the network layer with MPICH-MX.

The criteria for choosing the parallel framework are summarized as follows. It should be well designed for Linux cluster systems using MPI, supporting branch-and-bound for the current need and branch-and-cut for future use, in addition to having a continuous support from the framework developers.

Many parallel frameworks have been introduced. The popular ones which are open-source and still supported are SYMPHONY [45], PEBBL [15, 16], and ALPS [56, 57]. Both SYMPHONY and ALPS are COIN-OR projects [3]. SYMPHONY is developed by Ralphs, Güzelsoy, and Mahajan for solving MILP programs. It works in

the PVM environment. The framework employs a *master-worker* scheme for work distribution and load balancing, using a central node pool controlled by the master. This enables effective load balancing but somewhat limits the system's scalability. The framework also supports the *differencing* scheme for storing data compactly. One of SYMPHONY's applications is the capacitated vehicle routing problem [44].

PEBBL (formerly PICO) is a parallel framework for realizing branch-and-bound algorithms. It is developed by Eckstein, Phillips, and Hart [15, 16]. In PEBBL processors are grouped into *clusters*, each consists of a hub processor and some worker processors. Note that "cluster" in PEBBL is a software concept and different from the Linux cluster which is a real parallel computer. Work distribution within a cluster is controlled by the hub. Load balancing can be dynamically handled within a cluster or between clusters. PEBBL does not use a "global" master for controlling hubs. One of its important features is *checkpointing* which saves the system's internal state to disk every given interval and computation can be restarted from the checkpoint.

ALPS is designed by Xu, Ralphs, Ladányi, and Saltzman [56, 57] as a framework for implementing parallel algorithms based on tree search. Its architecture was influenced by PEBBL. Developed by the same authors, the CHiPPS framework [57] with layers built on top of ALPS supports the implementation of relaxation-based branch-and-bound as well as branch-and-cut algorithms. ALPS works in the MPICH environment and uses a *master-hub-worker* scheme for solving the bottle-neck problem for the master in load balancing. Developed later, ALPS uses good designs from its predecessors and is able to avoid their limitations.

We chose ALPS as the framework to realize our parallel solver because of its good design and the MPICH-based working environment. In addition, the support of CHiPPS for branch-and-cut methods and CLP (the open-source COIN-OR LP solver) may be helpful for us later to approach either the bandwidth problem or another. More details about ALPS will be described in the next section.

6.2.2 Implementation

The parallel solver for BW is based on the ALPS framework and implements the heuristics and exact algorithms described previously in Chapter 4 and Chapter 5. It is named *HEBAS*, short for Heuristic and Exact BAndwidth Solver.

The design of Hebas can be explained with the class diagram in Figure 6.2 which follows the architecture of ALPS. There are two types of relationships between classes depicted by arrows in the diagram: the first one with an empty triangle indicates an *inheritance* relationship, and the second one with a diamond indicates a *containing* relationship. For example, the class `BwNodeDesc` inherits from the class `AlpsNodeDesc` and contains a pointer to an object of the class `PartialPermutation`.

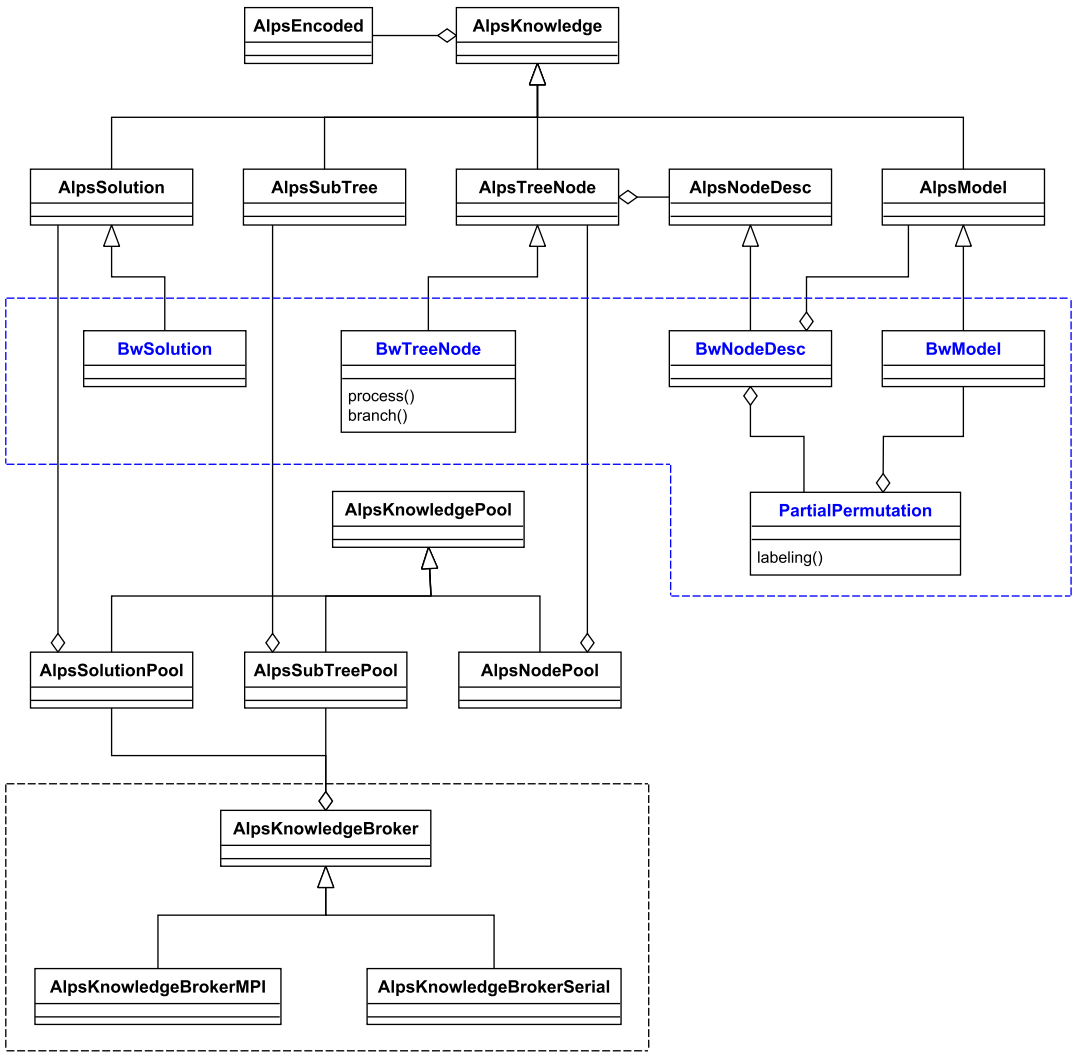


Figure 6.2: The class diagram of the parallel solver.

Information in an ALPS-based program is denoted as *knowledge* and derived from the class `AlpsKnowledge`. There are four types of knowledge: model, solution, tree node, and subtree. A *subtree* contains a hierarchy of nodes in part of the search tree. During the search, processes in ALPS use subtrees instead of nodes to share work between each other. By doing that ALPS can store and transfer a group of nodes efficiently, and thus fewer communications are required. This knowledge is already realized in the original class AlpsSubTree.

The other three knowledges are prototyped in ALPS and need to be realized for each specific problem. A *model* describes the problem data. It is realized in the class `BwModel` to describe the graph instance and properties of the bandwidth problems. A *solution* describes a feasible solution and is realized in the class `BwSolution` to characterize a permutation. A *tree node* describes the data and methods for a branch-and-bound tree node. The data of a node is realized in the class `BwNodeDesc`. Its associated methods are realized in the class `BwTreeNode`. Here, the member function *process* performs bounding procedures on the node and determine its status for further processing. This can be either fathomed, solution found, or waiting for being split. The function *branch* splits the current node into child nodes.

The fundamental data of the BW problem, the partial permutation, is implemented in the class `PartialPermutation`. For a partial permutation, this class keeps label domains of all vertices and provides supporting functions such as the extendability test or the analysis of the dominance relation with another partial permutation. The member function *labeling* updates basic bounds of the partial permutation and applies constraints along with extendability tests. It returns one of the following results: the partial permutation has become a feasible permutation, the extendability test is failed, or no violation is detected.

Searching is driven by the functions *process* and *branch* of the class `BwTreeNode` and *labeling* of the class `PartialPermutation`. Three of them constitute all the procedures in the function *labeling* of Algorithm 3.4 and Algorithm 5.8. In particular, the solver selects a node from the queue and call *process* to determine the status of the associated partial permutation. This function gets the answer from the function *labeling*. Depending on the answer, *process* fathoms the node if a feasible permutation has been found or violation is detected, or marks the node for branching. In the latter case, the function *branch* is called to extend the partial permutation and add child nodes to the queue.

Knowledge in ALPS is stored in repositories called *knowledge pools*. They do not communicate directly with each other, but through agents called *knowledge brokers*. Each process has a knowledge broker which is responsible for sending, receiving, and routing any kind of knowledge. Communication protocols can be supported via the implementation of these broker objects. ALPS supports two protocols: single-process (non-parallel) and MPICH on Linux clusters.

ALPS uses a master-hub-worker scheme to overcome the bottle-neck problem at the master process when scaling up the parallel system. In this scheme, the master still functions as the central controller, but does not manage the workers directly. Instead an intermediate management level with hubs is inserted in between. Depending on how many workers are allocated, an appropriate number of hubs will be used. The master initializes the data and communication protocol, establishes the hubs, and passes them with initial search nodes. The hubs allocate jobs to the workers and control the search of its portion.

The execution of a parallel search program in ALPS can be divided into three phases: ramp-up, primary, and ramp-down. In the *ramp-up* phase, work is initialized and distributed to all workers. Searching for the solution is conducted in the *primary* phase. The final phase is *ramp-down*. It executes the termination procedures and reports the results.

A very important task of the parallel solver is *load balancing*. It attempts to allocate work to all workers during the ramp-up phase and balance the workload between them during the search. In ALPS, this task is called *static* load balancing in the ramp-up phase and *dynamic* load balancing in the primary phase.

Two static load balancing schemes are supported in ALPS. In the *spiral* scheme, the master generates child nodes from the root node and distributes them to the hubs. The hub in turn generates child nodes and allocates them to its workers. The static balancing procedure stops when all knowledge pools have received at least one node. In the *two-level root initialization* scheme, the master generates a predefined number of child nodes and distributes them to the hubs, which in turn generates another predefined number of child nodes for the workers. We simply use the spiral static load balancing and let the master and the hubs define the necessary number of child nodes based on the sizes of the candidate sets.

During the search, due to the bounding and branching procedures some workers may have little work left while the others are overloaded. *Dynamic* load balancing is thus important to ensures that all workers have work to do and the work is *useful*. ALPS supports two dynamic load balancing schemes. The *intra-cluster* dynamic load balancing scheme controls the workload between workers within a cluster. The request for donating or receiving work can be initiated by either the hub or a worker. The second scheme called *inter-cluster* dynamic load balancing is used for balancing the workload between clusters. In the primary phase, the hubs periodically report their workload status to the master. Based on these information, the master work with the hubs to reallocate and balance the load between their portions. The two dynamic schemes work cooperatively for an effective load balancing of the whole system. We enable both schemes in our ALPS-based solver.

We perform an evaluation of the system performance and report about it in Section 7.4, in particular Section 7.4.1 and Section 7.4.2.

7 Computational results

This chapter presents our computational results for the bandwidth problem. The first section introduces the benchmark suites, the hardware, the notation used in the chapter, and the solver configurations. We then report an evaluation of the solver's performance over different configurations of constraints and running modes.

In the next sections we report our computational results and compare them with the best known results from the literature. First, lower bounds for small and large instances in the popular benchmark suite and for very large instances in the second suite are presented. The results of the parallel computation are then introduced. The last section reports the results of our heuristic methods on both benchmark suites.

For the reader's convenience of reading and focusing on the topic, in this chapter we only present selective results which represent the trends of the solver and the statistics. The detailed computational results can be found in Appendix A.

7.1 Introduction

7.1.1 Benchmark suites and comparison results

We performed computational experiments on two benchmark suites. The popular suite has 113 instances which are divided into two sets. The first set consists of 33 *small instances* with less than 200 vertices each. The second set consists of 80 *large instances* whose number of vertices is larger than 200 and smaller than 1000. These instances were originally matrices in the library *Matrix Market* [2]. They were converted to graphs by Martí et al. [37] and used in different publications since then. The best known lower bounds for instances in this suite are reported in Martí et al. [36]. The best known upper bounds are obtained by taking the best values from different methods in Martí et al. [37, 36], Piñana et al. [43], Campos et al. [4, 5], and Lim et al. [34]. For convenience, in this chapter we refer to the work in [37, 43, 4, 36] as *MCP* and the work in [34] as *LRX*.

Our experiments for instances in the popular suite are performed directly on the available graph files introduced in [37]. One reason is that the original matrix of some instances contains *zero-edges*, i.e., edges whose weight is zero in the matrix file.

Depending on whether these zero-edges are included in the graph or not, the graph properties may be different and so is the bandwidth. We will see such an example in Section 7.3.3. Another reason is that the corresponding graph of some matrix instances is disconnected and usually the authors [37] chose the largest component as the final graph. In addition, vertices are randomly rearranged in their graph files. This changes the initial bandwidth upper bound and thus affects the solution quality of heuristic methods. Therefore, we think it is reasonable to use the same instance data for comparing our results and their results.

The second benchmark suite is based on the work of Safro, Ron, and Brandt [50]. This suite consists of 51 very large instances and the largest one has about $217,000$ vertices. The instances are taken from *The University of Florida Sparse Matrix Collection* which is maintained by Davis and Hu [11].

We notice that some instances having less than 1000 vertices are already included in the first suite. For this reason and the ones mentioned in the case of the first suite, for evaluating our exact algorithms we select a set of 36 instances whose corresponding graph is connected and has more than 1000 vertices. On the other hand, our heuristics will be experimented on all 51 instances. The heuristic results will be compared with the best known upper bounds obtained by Safro, Ron, and Brandt [50]. We will refer to their work in this chapter as *SRB*.

When we refer to *very large instances*, for evaluating the exact Hebas solver we mean the set of 36 connected instances with more than 1,000 vertices each. For heuristic methods we mean the set of all 51 instances used in the reference work [50]. The instance files are downloaded directly from the collection [11].

7.1.2 Hardware

In non-parallel modes, the experiments are performed on PC computers running Debian 4.1 for computing upper bounds for all instances and lower bounds for instances in the first suite. Each has 4 dual-cores 2.5 GHz and the entire physical memory is 2.5 GB RAM. Since the PC has 8 processors, to save time we run 8 non-parallel jobs simultaneously, so on the average each job uses a memory of 0.3 GB. Notice that this is not parallel computation, here each job processes a different instance.

We use a stronger computer to compute the lower bounds for very large instances in the second suite. It also runs Debian 4.1 and has 4 dual-cores 2.5 GHz, but is equipped with a physical memory of 16 GB RAM. We run 4 single jobs at a time, so in this case each job can use up to 4 GB of memory on the average.

The parallel computation were carried out on the cluster of the Interdisciplinary Center for Scientific Computing (IWR) in Heidelberg [26]. This cluster consists of 156 computer nodes, each is equipped with two Dual Core AMD 2.8 GHz processors

with 8 GB RAM memory running Debian 4.0. The cluster uses Myricom 10G for the network layer with MPICH-MX.

The software programs are written in C++ and built with compiler `gcc` version 4.1.2 for both non-parallel and parallel versions.

7.1.3 Notation

For each instance in the report tables, n is the number of vertices of the graph and m is the number of edges. Notice that m is usually different from the number of value entries of the original matrix. Φ_t is the initial lower bound for a graph. Φ_l is the lower bound and Φ_u is the upper bound obtained by a solution method.

Φ_t is computed by taking the maximum of two lower bounds $\gamma(G)$ and $\alpha(G)$ which have been described in section 3.2. Since Φ_t is fixed and can be easily computed for all graphs, we choose it as the standard value to evaluate the lower bound and the upper bound obtained by a certain method.

We denote by *LB Improvement*, abbreviated as *imp*, the amount that a lower bound has been improved from the initial lower bound. It is computed according to formula (7.1). The larger the LB improvement, the stronger the lower bound obtained by an exact method.

$$imp\ (\%) = \frac{100 \times (\Phi_l - \Phi_t)}{\Phi_t} \tag{7.1}$$

Similarly to the evaluation of the lower bound, we denote by *UB Deviation*, abbreviated as *dev*, the amount an upper bound deviates from the initial lower bound. It is computed as in (7.2). The smaller the UB deviation, the better the obtained upper bound.

$$dev\ (\%) = \frac{100 \times (\Phi_u - \Phi_t)}{\Phi_t} \tag{7.2}$$

In addition, we use the factor *gap* to know whether an instance has been solved to optimality and if not, how large is the gap between the lower bound and upper bound. It is computed as in (7.3).

$$gap\ (\%) = \frac{100 \times (\Phi_u - \Phi_l)}{\Phi_l} \tag{7.3}$$

7.1.4 Solver configurations

To distinguish different configurations of constraints and running modes used for the Hebas solver, we assign a code to each configuration. Actually, for the final com-

putation we will apply the strongest configurations for the single-labeling scheme, the 2-labeling scheme, and their parallel versions. The other configurations are only used to evaluate the effect of the constraints, the dominance rule, the 2-labeling scheme, as well as the system's performance.

The configuration whose code start with 1., such as Hebas 1.1, uses the single-labeling scheme. Those whose code starts with 2. refer to the 2-labeling scheme. A parallel configuration has a code begins with the numbers of its non-parallel mode and ends with a letter "p". For example, Hebas 1.4p is the parallel version of Hebas 1.4 which uses the single-labeling scheme and applies all the constraints as well as the dominance rule with the hash-table. The configurations are listed in Table 7.1.

Table 7.1: Hebas configurations.

Code	**Constraints and running mode**
0.9	Basic branch-and-bound algorithm (Algorithm 3.4) and the first constraint (3.10) (3.11)
1.0	Hebas 0.9 + fitting constraint using only adjacent vertices (5.1) (5.4)
1.1	Hebas 1.0 + density cut constraint (Algorithm 5.1) + density near-cut constraint (Algorithm 5.2)
1.2	Hebas 1.1 + tightening at the non-extended side (Algorithm 5.4)
1.3	Hebas 1.2 and only if at least 70% of vertices have positive excess-ranges: + generalized fitting constraint (5.3) (5.5) + generalized pulling constraint (5.8) (5.9)
1.3.1	Hebas 1.2 + (always) generalized fitting constraint (5.3) (5.5) + (always) generalized pulling constraint (5.8) (5.9)
1.4	Hebas 1.3 + dominance relation the with hash-table (Algorithm 5.5)
1.4.1	Hebas 1.3.1 + dominance relation with the hash-table (Algorithm 5.5)
2.3	Hebas 1.3 in 2-labeling mode (Algorithm 5.8)
2.4	Hebas 1.4 in 2-labeling mode
1.4p	Parallel version of Hebas 1.4
2.3p	Parallel version of Hebas 2.3
2.4p	Parallel version of Hebas 2.4

The configurations 1.3 and 1.3.1 may need a bit more explanation. As described in section 5.1.1, before solving each BW problem for a given φ we check and record vertices v having positive excess-ranges, i.e., $|N_h(v)| - h\varphi > 0$, and for each vertex

the layer h_{heu} such that the excess-range is largest. For some small instances the generalized fitting constraint is not effective. The number of subproblems is smaller but the solver requires longer running time. We will see such an example in section 7.2.

Therefore, in configuration 1.3 the generalized fitting constraint and the generalized pulling constraint are applied only if the number of vertices having positive excess-ranges is at least 70% of n. On the other hand, these constraints are always applied in configuration 1.3.1. This is used for computing the lower bound for very large instances.

7.2 Performance evaluation

We evaluate the performance of the bandwidth solver over different configurations by observing the number of subproblems generated during the search and the running time required to solve each single BW problem. The graphs are selected among small instances in the popular suite and the search parameter φ is intentionally chosen to be smaller than the optimal value. Therefore, there is no feasible solution for every BW problem in this section and the total number of generated subproblems characterizes the *full search space*.

As mentioned previously, we only introduce representative data here. The full data can be found in Appendix A, where the numbers of subproblems are reported in Table A.7 and the running times are reported in Table A.8. In these two tables, *BW(φ)* denotes the BW problem for the given φ. The running times are in seconds. A value marked with "_" indicates that the solver is not able to solve the corresponding BW problem in a reasonable time.

For convenience, we briefly summarize the constraint configurations as follows. We start with the first constraint developed by Caprara and Salazar-González [6] in Hebas 0.9. The new constraints are then gradually added from Hebas 1.0 until Hebas 1.3. The dominance relation is implemented with the hash-table in Hebas 1.4. The single-labeling scheme is used from Hebas 0.9 to Hebas 1.4, while Hebas 2.3 and Hebas 2.4 employ the 2-labeling scheme.

7.2.1 The development of the solver

A view of the solver's development can be seen with the numbers from instance `bcsstk22`. It is depicted with the chart in Figure 7.1. This instance has 110 vertices and the optimal bandwidth is 10. The chosen search parameter is $\varphi = 9$.

We can observe that while the lower bound $\varphi = 9$ could not be solved by Hebas 0.9, it becomes possible by Hebas 1.0 with 7.88×10^8 subproblems in 37,020 seconds.

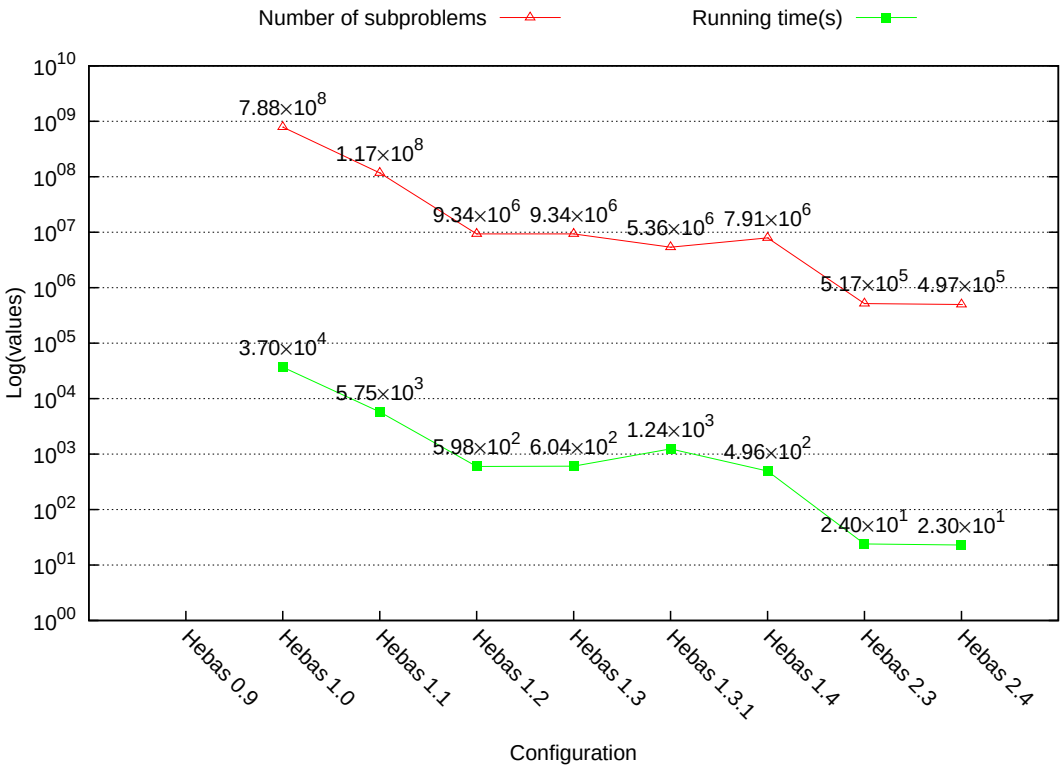


Figure 7.1: Solving instance `bcsstk22` for $\varphi = 9$.

These numbers are gradually reduced in later configurations. With Hebas 2.4, the problem can be solved with 4.97×10^5 subproblems in only 23 seconds.

There are also a few instances that the new constraints have negative effect, i.e., Hebas obtains worse lower bounds when these constraints are applied. These instances will be mentioned explicitly in the computational reports.

We notice the case of Hebas 1.3.1 (the generalized pulling and fitting constraints are always applied) in Figure 7.1. The number of subproblems is reduced about a half, but the running time is almost doubled when compared with Hebas 1.3 in which these constraints are not applied. This issue happens to some instances like `bcsstk22`. To address it, for small and large instances we use the heuristic that these constraints are applied only if at least 70% of the vertices having positive excess-ranges. For very large instances we need the constraints to be as strong as possible. This allows the branch-and-bound tree to generate fewer subproblems and require less memory. Thus Hebas 1.3.1 will be used.

7.2.2 The search space of the BW problem

Figure 7.2 depicts the numbers of subproblems generated by Hebas 0.9 for solving some graph instances with different φ whose values are shown on the horizontal axis of the chart. For each instance, a set of increasing values φ are chosen to observe

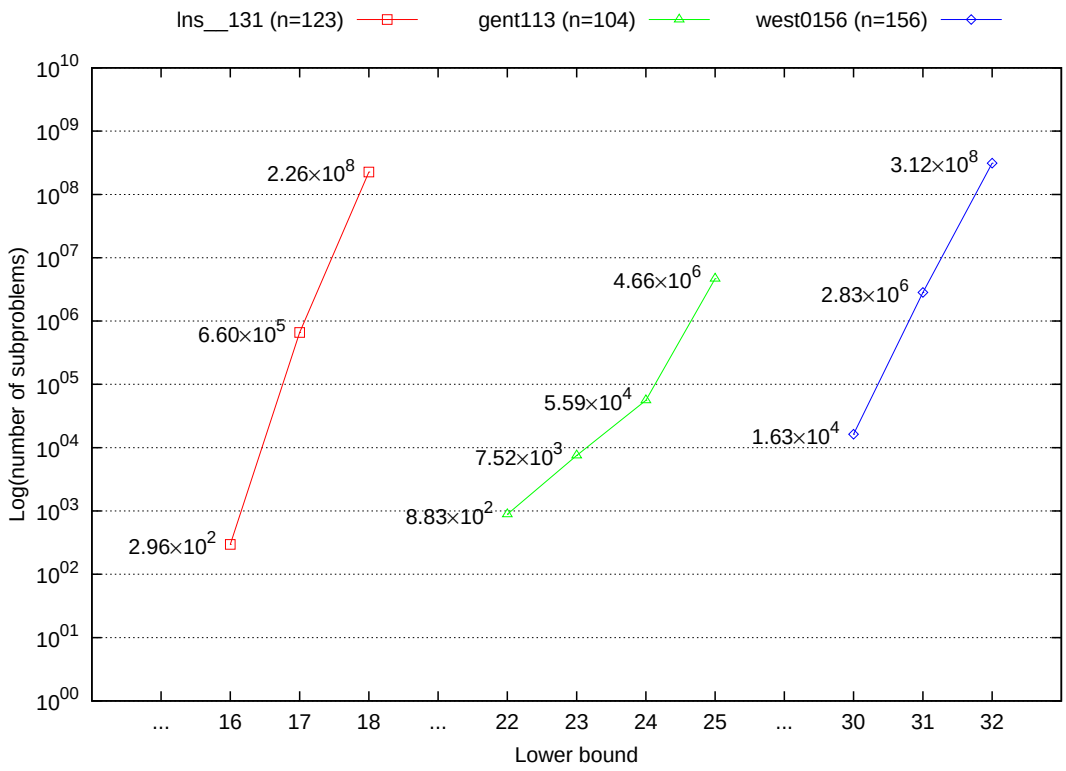


Figure 7.2: Number of subproblems generated by Hebas 0.9.

how the search space expands when φ is increased only by 1.

The numbers of subproblems for instance `gent113` show a typical expansion of the search space of BW. Starting with $\varphi = 22$ only 883 subproblems are generated. This number is increased to 7.52×10^3 for solving $\varphi = 23$. The significant change starts at $\varphi = 24$ where Hebas 0.9 generates 5.59×10^4 subproblems. This goes up to 4.66×10^6 for solving $\varphi = 25$. In other words, the increase is approximately equivalent to the size of the graph. We have similar observation with the other instances `lns__131` and `west0156`.

This quick increase of the number of subproblems reminds us the algorithm by Saxe [51] whose running time is $O(f(\varphi)n^{\varphi+1})$ and the improvement to $O(n^{\varphi})$ by Gurari and Sudborough [25]. In brief, the search space of the bandwidth solver gets much larger if φ is increased only by 1. This makes it difficult to improve the lower bound if the solver does not have a constraint which is useful for the graph.

The number of subproblems shows only one dimension of the solver's performance. It also depends on the running time for solving the problem, or in other words the efficiency of the solver. If a constraint is strong but requires too much time for a certain instance, it may be possible that the problem can be solved with fewer subproblems but the total running time is higher. We have observed this situation previously with instance `bcsstk22` in Figure 7.1. When the generalized pulling and fitting constraints are always applied in Hebas 1.3.1, the problem can be solved with

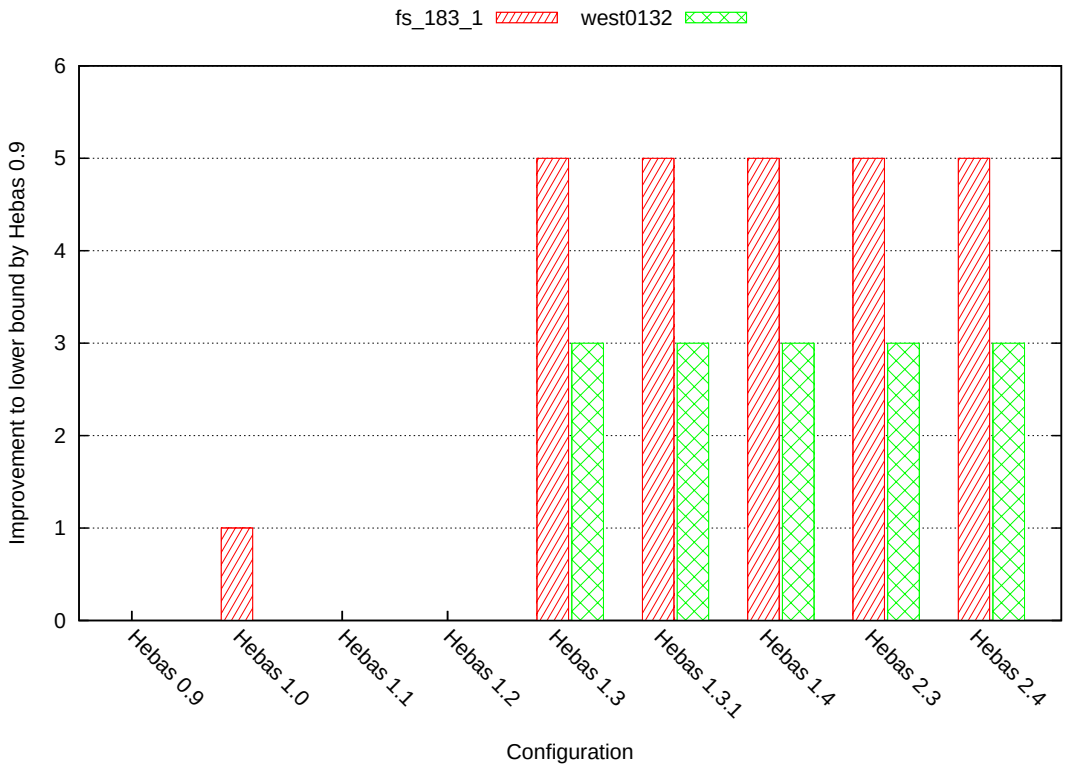


Figure 7.3: Lower bound improvement compared to Hebas 0.9.

approximately a half of the number of subproblems but in almost doubled running time, compared to the case of without using these constraints.

7.2.3 The effect of the constraints

If the solver has a constraint which is useful for the graph, the problem can be quickly solved because label domains can be effectively tightened by that constraint. This can be demonstrated by the chart in Figure 7.3. The vertical axis shows the improvement made by later Hebas configurations compared with the lower bound obtained by Hebas 0.9. The detailed numbers of subproblems and the running times can be found in Table A.7 and Table A.8, respectively.

Consider instance `fs_183_1`(183 vertices). The lower bound obtained by Hebas 0.9 is 52, i.e., Hebas 0.9 is able to solve $\varphi = 51$ but cannot make a conclusion for $\varphi = 52$ in a reasonable time. This becomes possible with Hebas 1.0. Here BW for $\varphi = 52$ can be solved with approximately 6.42×10^8 subproblems in about 9 hours. Since the constraints added in Hebas 1.1 and Hebas 1.2 are not useful for this instance, they only make the running time longer and even $\varphi = 52$ cannot be solved anymore.

However, when the generalized pulling and fitting constraints are applied as in Hebas 1.3 the problem can be solved for $\varphi = 52$ up to $\varphi = 56$. Only 184 subproblems are generated for $\varphi = 52$, i.e., exactly one root subproblem and the other 183

subproblems of the first depth of the branch-and-bound tree. In other words, all these 183 subproblems are immediately solved without branching. The running time is only a few milliseconds and the lower bound is improved from 52 to 57.

We have the same observation with instance `west0132`. The lower bound obtained by Hebas 0.9 is 25. This remains unchanged from Hebas 1.0 to Hebas 1.2. When the generalized pulling and fitting constraints are applied in Hebas 1.3, the lower bound is improved from 25 to 28.

7.2.4 The effect of the dominance rule

The dominance concept is first applied in the branch-and-bound algorithm with the hash-table in Hebas 1.4 which still uses the single-labeling scheme. It is then applied in Hebas 2.3 and Hebas 2.4 within the 2-labeling scheme. To begin with, we examine the effect of the dominance rule by looking at the numbers of subproblems in Table 7.2.

Table 7.2: The effect of the dominance rule: Numbers of subproblems.

Instance	n	m	BW(φ)	**Hebas 1.3**	**Hebas 1.4**	**Hebas 2.3**	**Hebas 2.4**
bcsstk22	110	254	9	9,340,176	7,907,288	516,781	497,142
impcol_b	59	281	19	676,113,799	676,040,314	12,934,039	12,934,039
impcol_c	137	352	26	864,365	864,325	60,759	60,749
lns_131	123	275	18	225,320,091	205,923,411	34,416,427	33,684,015
fs_183_1	183	701	52	184	184	16,654	16,654
gre_185	185	650	17	186	186	17,021	17,021
lund_a	147	1,151	18	148	148	10,732	10,732
west0167	167	489	30	168	168	13,862	13,862
			31	_	_	31,860,757	31,860,757

The instances in Table 7.2 can be divided into three groups. The first group consisting of instances `bcsstk22`, `impcol_b`, `impcol_c`, and `lns__131` shows a clear improvement made by the dominance concept. In ideal cases such as instance `bcsstk22`, the solver with the hash-table in Hebas 1.4 reduces a noteworthy number of subproblems compared with the solver without it in Hebas 1.3. The improvement is made even further by the 2-labeling scheme.

In contrast to the first group, the second one with instances `fs_183_1`, `gre__185`, and `lund_a` shows a significant increase on the number of subproblems when the 2-labeling scheme is used. When instance `fs_183_1` is solved for $\varphi = 52$, only 184 subproblems are generated by Hebas 1.3. However, Hebas 2.3 and Hebas 2.4 generate up to 16,654 subproblems, equal to one root subproblem and exactly $n(n-1)/2$

subproblems where $n = 183$ as mentioned in Section 5.3. The width of the search tree becomes much wider compared with that of the solver using the single-labeling scheme, and this causes the solver to run out of memory when the 2-labeling scheme is used for solving large instances. The issue can only be solved with stronger dominance rules or by using computers with large memory capacity such as a parallel cluster.

The second group, however, does not draw a complete picture of the 2-labeling scheme. Even though approximately $n(n-1)/2$ subproblems are generated at each branching step, the dominance concept can still be helpful for large φ. This can be seen with instances in the last group. Consider instance `west0167`. The solver with the 2-labeling scheme Hebas 2.3 generates many more subproblems than does the single-labeling scheme for solving $\varphi = 30$. However, it can solve the larger $\varphi = 31$ in about 7,500 seconds while the solver using the single-labeling scheme cannot make this in a reasonable time.

Now we look at the running times of the solver using the dominance concept over different configurations in Table A.8. Compared with Hebas 1.3, Hebas 1.4 does speed up the running time for solving some instances. In fact this will be the case for most large instances. Hebas 2.3 is clearly the best configuration for small instances with respect to the lower bound improvement as well as the running time.

Between Hebas 2.3 and Hebas 2.4 (the solvers using the 2-labeling scheme without and with the hash-table) we observe that for some instances, in particular `west0156`, `west0167`, and `will199`, Hebas 2.4 generates the same number of subproblems but requires much larger running time. Since the hash-table can make no further effect to the elimination of subproblems than the 2-labeling branching procedure (which already exploits the dominance rule), its processing procedures only makes the running time longer on these instances. Therefore, for the final computation in the non-parallel mode we use Hebas 1.4 as the configuration for the single-labeling scheme. For the case of the 2-labeling scheme Hebas 2.3 will be used.

We also observe that with the *current* dominance rule, the hash-table and/or the 2-labeling scheme can help the solver to reduce the running time only linearly (in cases of improvement). Therefore, the task of finding stronger lower bounds still relies on the constraints. On the other hand, the hash-table within the single-labeling scheme can help the solver find feasible permutations more quickly because dominated partial permutations are eliminated in the branch-and-bound tree.

7.3 The lower bounds

In this section we report the lower bounds computed for instances in both benchmark suites. As mentioned previously, only selective instances which represents typical

characteristics of the solver are shown. In addition, the new optimal solutions and the statistics on the results for each suite are reported.

7.3.1 Lower bounds for small instances

For small instances the running time of the exact solver is limited to 4 hours. Table 7.3 reports the computational results for 12 instances whose lower bounds are different from those of MCP. We obtain the same lower bounds for the other 21 instances. The full results for 33 instances can be found in Table A.1 in Appendix A.

For each instance in Table 7.3, two values under *MCP* show their lower bound and the corresponding improvement from the initial lower bound. Values under *VR08* are the results from our previous work [55] which was obtained by the parallel computation at a large scale with 508 processors. There are four values under *Hebas*: Φ_l of *1.4* and *2.3* report the lower bounds obtained by Hebas 1.4 and Hebas 2.3 respectively, *Best* takes the maximum of these two, and *imp* shows the lower bound improvement of this best value.

We format the values Φ_l of VR08, Hebas 1.4 and Hebas 2.3 for ease of comparison. For all of them, an italic value Φ_l means that the computed lower bound is worse than that of MCP. A bold value Φ_l by VR08 and Hebas 1.4 indicates an improvement compared with MCP. The value Φ_l by Hebas 2.3 is in bold type if it gives better lower bound than both Hebas 1.4 and MCP.

In our previous work [55], when Hebas 0.9 was run on a large-scale parallel cluster with 508 processors it could improve the lower bounds for 5 instances (including 2 new optimal solutions) compared with MCP. The non-parallel Hebas 1.4 can improve the lower bounds for 3 other instances. In particular, the lower bound for instance `fs_183_1` is increased from 52 to 57, for `west0132` from 25 to 28, and for `impcol_c` from 26 to 27. The reason is that the generalized pulling and fitting constraints are useful for these instances. On the other hand, the lower bounds for instances `west0156` and `will199` computed by Hebas 1.4 are worse than those of MCP partly because of these constraints.

The 2-labeling scheme works effectively on small instances. Hebas 2.3 continues to improve the lower bound for instance `west0167` from 31 to 32. In addition, it increases the lower bound for two instances `west0156` and `will199`. Therefore, compared with MCP `will199` is the only instance that Hebas 2.4 has a worse lower bound. It obtains either equally good or better lower bounds for all other instances. The lower bounds for instances `lund_a` and `lund_b` are not improved here, but we will see that in the parallel mode.

Hebas 1.4 also finds two more optimal solutions compared with the best known results. They are listed in Table 7.4. Actually these optimal solutions have been

Table 7.3: Lower bound for selective small instances.

Instance				**MCP**		**VR08**		**Hebas**			
				Best		Best		1.4	2.3	Best	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	Φ_l	*imp*
bcsstk22	110	254	8	9	12.50	**10**	25.00	**10**	10	10	25.00
fs_183_1	183	701	52	52	0.00	52	0.00	**57**	57	57	9.62
gre___115	115	267	16	20	25.00	**21**	31.25	20	20	20	25.00
gre___185	185	650	16	17	6.25	**18**	12.50	**18**	18	18	12.50
impcol_b	59	281	15	19	26.67	**20**	33.33	**20**	20	20	33.33
impcol_c	137	352	21	26	23.81	26	23.81	**27**	27	27	28.57
lund_a	147	1151	17	19	11.76	19	11.76	19	19	19	11.76
lund_b	147	1147	17	19	11.76	19	11.76	19	19	19	11.76
west0132	132	404	23	25	8.70	25	8.70	**28**	28	28	21.74
west0156	156	371	27	34	25.93	34	25.93	*33*	34	34	25.93
west0167	167	489	28	31	10.71	31	10.71	31	**32**	32	14.29
will199	199	660	44	57	29.55	**59**	34.09	*53*	*55*	*55*	25.00

obtained in [55], but this is the first time they have been proven by a non-parallel algorithm.

Table 7.4: New optimal solutions for small instances.

Instance				**MCP & LRX**		**Hebas**	
Name	n	m	Φ_t	Φ_l	Φ_u	Φ_l	Φ_u
bcsstk22	110	254	8	9	10	10	10
impcol_b	59	281	15	19	20	20	20

The statistics on the lower bounds for small instances are reported in Table 7.5. The first three rows depict the average of improvement of lower bound from the initial value, as well as the largest and the smallest. With the new constraints and the dominance rule, the lower bound improvement made by the new solver in the non-parallel mode is even better than that of the previous solver which used a large-scale parallel cluster [55]. Hebas 1.4 is able to find two new optimal solutions and Hebas 2.3 makes the best lower bound improvement with an average of 15.79%. Compared with MCP, Hebas 2.3 obtains better lower bounds for 7 instances and worse result for 1 instance. This only worse case will be improved again in the parallel mode.

7.3.2 Lower bounds for large instances

The running time of the solver is also limited to 4 hours for large instances. Table 7.6 shows the results for selective instances. The detailed results can be found

Table 7.5: Statistics on lower bounds for small instances.

	MCP	**VR08**	**Hebas 1.4**	**Hebas 2.3**
LB Improvement (%)				
Average	14.22	15.32	15.44	15.79
Largest	30.00	34.09	33.33	33.33
Smallest > 0	1.61	1.61	1.61	1.61
Optimal solutions				
Number	18	20	20	20
Versus MCP				
Number of better		5	6	7
Number of worse		0	2	1

in Table A.3 in Appendix A.

The meaning of the values in Table 7.6 is the same as in Table 7.3. We refer to the results of MCP only since parallel computation was not performed for large instances in [55]. We also format the values Φ_l for the convenience of comparison. An italic value Φ_l means that the computed lower bound is worse than that of MCP. A bold value Φ_l by Hebas 1.4 indicates a lower bound improvement compared with MCP. The value Φ_l by Hebas 2.3 is in bold type if it obtains better lower bound than both Hebas 1.4 and MCP.

We begin to experience the memory issue with the 2-labeling scheme on some large instances. They are marked with a "_" at the value Φ_l. This happens when running the solver on a computer with moderate memory (as described in Section 7.1.2). The issue may not exist any more on computers with much larger memory capacity such as a parallel cluster.

Based on the results obtained by the solver, the instances are divided into four groups in Table 7.6. The first group consists of instances which have been solved to optimality for the first time. The dominance rule with the hash-table works effectively if the graph has a bandwidth φ for which BW is being solved. It reduces the search space and thus speeds up the time to reach a feasible solution, which is the optimal solution in these cases.

The second group includes three instances that Hebas 1.4 obtains worse lower bounds than MCP. They are `dwt__419`, `str__600`, and `west0655`. Actually, these are the only three instances for which Hebas 1.4 yields worse results than MCP. It obtains either equally good or better lower bounds for all other large instances. On the other hand, Hebas 2.3 using the 2-labeling scheme works well on these instances. In particular, the lower bound for `dwt__419` is increased from 22 to 23 and that of instance `str__600` is improved to 101, equally good as those of MCP. The lower bound for instance `west0381` is improved from 118 to 120 which is even better than the result of MCP. By combining the results of Hebas 1.4 and Hebas 2.3, Hebas in

Table 7.6: Lower bounds for selective large instances.

Instance				**MCP**		**Hebas**			
				Best		1.4	2.3	Best	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	Φ_l	Φ_l	*imp*
fs_760_1	760	3518	35	36	2.86	**37**	37	37	5.71
nos6	675	1290	15	15	0.00	**16**	16	16	6.67
dwt___419	419	1572	21	23	9.52	*22*	23	23	9.52
str___600	363	3244	79	101	27.85	*98*	101	101	27.85
west0381	381	2150	100	119	19.00	*118*	**120**	120	20.00
mbeacxc	487	41686	243	248	2.06	**249**	**250**	250	2.88
west0655	655	2841	100	109	9.00	**124**	**125**	125	25.00
bp___1600	822	4809	192	199	3.65	**201**	_	201	4.69
dwt___918	918	3233	27	27	0.00	**29**	_	29	7.41
west0989	989	3500	121	123	1.65	**165**	_	165	36.36

the non-parallel mode already obtains either equally good or better lower bounds for large instances than does MCP.

The third group consists of instances on which both single-labeling and 2-labeling schemes work effectively. Consider instance `west0655`. The lower bound is improved from 109 by MCP to 124 by Hebas 1.4 and Hebas 2.3 makes an even better improvement to 125. Similar patterns can be observed with instance `mbeacxc`.

The last group lists instances that work effectively with the single-labeling scheme but have memory problem with the 2-labeling scheme on computers having moderate memory. Due to the large number of subproblems generated by the 2-labeling scheme, Hebas 2.3 cannot reach a good lower bound as does the solver using the single-labeling scheme. However, only the solver using the single-labeling scheme already makes noticeable improvement. Consider the second largest instance in this set `west0989` with 989 vertices. The lower bound is improved from 123 by MCP to 165 by Hebas 1.4. In other words, a lower bound improvement from 1.65% to 36.36% has been made.

The new optimal solutions for large instances are reported in Table 7.7, showing the initial lower bounds, the best known results, and the bounds computed by Hebas.

Figure 7.4 illustrates the matrix versions of different bandwidths of the largest instance in the first suite: `dwt__992` ($n = 992$). Figure 7.4(a) shows the original matrix downloaded from the library *Matrix Market* whose bandwidth is 513. The instance is converted to a graph by MCP whose matrix version is shown in Figure 7.4(b). Since they randomly reorder vertices of the graph, the bandwidth is changed to 977.

Table 7.7: New optimal solutions for large instances.

Instance				MCP & LRX		Hebas	
Name	n	m	Φ_t	Φ_l	Φ_u	Φ_l	Φ_u
bcspwr05	443	590	25	25	28	26	26
dwt___245	245	608	21	21	22	21	21
fs_760_1	760	3518	35	36	38	37	37
nos6	675	1290	15	15	16	16	16

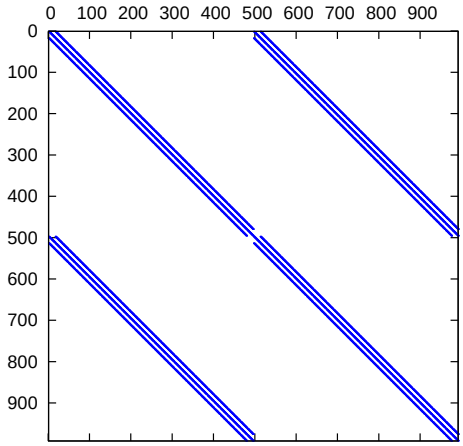
We can solve both versions of this instance to optimality. The optimal bandwidth is 35 and a matrix of optimal bandwidth is depicted in Figure 7.4(c).

Finally, the statistics on the lower bounds for the set of 80 large instances are reported in Table 7.8. The average lower bound improvement of 5.75% by MCP is first increased by Hebas 1.4 to 9.28%. When combining the results of Hebas 1.4 and Hebas 2.3, the average is improved to 9.49%. As already mentioned, there are four new optimal solutions and this increases the total number of optimal solutions from 8 to 12. For this set of 80 large instances, Hebas 1.4 obtains better lower bounds than MCP in 52 cases and only 3 worse cases are observed. For some instances such as `west0989` the improvement is very strong. Hebas, combining Hebas 1.4 and Hebas 2.3, increases the number of better cases to 54 and leaves no instance whose lower bound is worse than that of MCP.

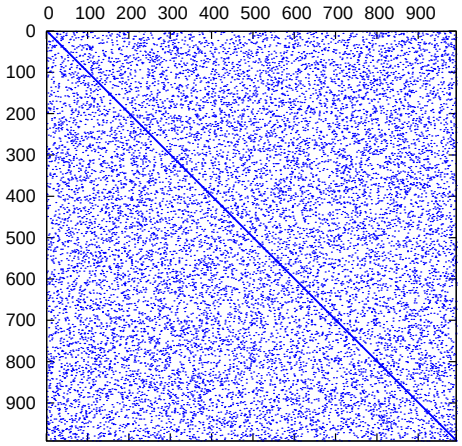
Table 7.8: Statistics on lower bounds for large instances.

	MCP	Hebas 1.4	Hebas
LB Improvement (%)			
Average	5.75	9.28	9.49
Largest	30.43	36.36	36.36
Smallest > 0	0.53	1.08	1.08
Optimal solutions			
Number	8	12	12
Versus MCP			
Number of better		52	54
Number of worse		3	0

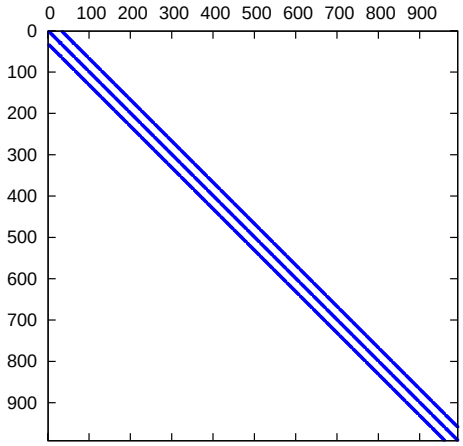
We have seen that Hebas with the new constraints and the dominance rule works effectively on instances with less than 1,000 vertices. This is not all, the solver can improve the lower bound for instances of much larger size, as reported in the next section.



(a) The original matrix ($\Phi_\pi = 513$)



(b) Randomized by MCP ($\Phi_\pi = 977$)



(c) Optimal bandwidth ($\Phi_\pi = 35$)

Figure 7.4: Instance `dwt__992` ($n = 992$)

7.3.3 Lower bounds for very large instances

We first introduce the necessary changes made to the solver so that it can work on very large instances. The results on lower bounds are then reported.

Implementation changes

The implementation of the exact solver needs some changes to enable Hebas to work on very large instances. First, solving them requires much computer's memory so we want to reduce the number of generated subproblems as much as possible. Thus the solver uses the strongest configuration of the constraints within the single-labeling scheme. In addition, it employs the dominance rule implemented with the hash-table. The generalized pulling and fitting constraints are always applied, i.e., the configuration of Hebas 1.4.1 is used, instead of Hebas 1.4 as in the case for

instances in the first suite. Second, previous extendability test procedures employ *bucket sort* which has complexity $O(n)$ and is relatively fast. The problem is that in our implementation it needs $O(n^2)$ memory. This would cause the solver to run out of memory for very large instances. Therefore, we switch to use the sorting function of the Standard Template Library (STL) [53]. It implements *introsort* [41] whose complexity is $O(n \log(n))$. This sorting procedure is relatively slower than bucket sort but requires less memory.

The third change that significantly affects the solver's performance is the computation of the distance between two vertices. Recall that the work of Caprara and Salazar-González [6] is effective on sparse graphs because it applies the basic bandwidth constraint not only to adjacent vertices but also to those more than one distance unit away from each other. In Hebas, the generalized pulling and fitting constraints also apply that idea and their effectiveness has been proven with the lower bound improvement for large instances. In the normal implementation of Hebas for instances in the first suite, the distances between vertices are calculated in advance and recorded in a distance matrix. However, this is impossible for very large instances because the memory required for such a matrix is $O(n^2)$. Consider a graph having 16,000 vertices. Only the distance matrix already needs 256×10^6 memory slots, equivalent to 1024×10^6 bytes or approximately 1 GB because an integer variable in C++ uses 4 bytes. Since they cannot be recorded, the distances between a vertex and remaining vertices have to be recalculated every time the label domain of that vertex is tightened.

These three changes cause the new solver to be slower for solving a small instance compared with the normal implementation mentioned in the previous sections, but it can now work on very large instances. For this set of very large instances, Hebas means the solver using the configuration of Hebas 1.4.1 including the three changes described above.

The conversion from a matrix to a graph needs to be noticed as well. In a matrix file of the *Matrix Market* format, the information about an edge in the equivalent graph is described by one line containing two vertices and the weight of that edge. Usually the weight is different from zero, but it is not the case for some instances. Depending on whether *zero-edges*, i.e., edges whose weight is zero in the matrix file, are included in the converted graph or not, the graph properties will be different in two cases and so are the bandwidths. Since the lower bound for an instance might be helpful to evaluate the upper bounds obtained by a heuristic which may accept or not the zero-edges, we apply two ways of conversion and reported the results in both cases.

The lower bounds

We use the computer with large memory capacity as described in Section 7.1.2 to compute lower bounds for instances in this suite. The solver is initialized with the solution first obtained by iGPS and improved further by SA-σ. The maximal running time is 12 hours including 30 minutes for the SA method. The detailed results can be found in Table A.5 in Appendix A. Here we only show selective instances.

The results for some very large instances are shown in Table 7.9. The meaning of values in the table is explained as follows. For each instance, the first three values describe the basic information of the matrix: its name, the number of columns (which is the same as the number of rows), and the number of edge entries (which might include zero-edges or duplicated edges). The next four values under *Without zero-edges* describe the information about the graph converted from the original matrix if zero-edges are skipped and the lower bound computed by Hebas. Similar to the report for small and large instances, m is the number of edges, Φ_t is the initial lower bound, Φ_l is the lower bound obtained by Hebas, and *imp* is the improvement from the initial lower bound. The last four values under *With zero-edges* show the same information for the converted graph which includes zero-edges.

We also format the values in Table 7.9 so that the report contains more information. For each instance, an italic value m indicates that the matrix contains zero-edges and the converted graphs in two cases will have different numbers of edges. For example, the converted graph of instance `add32` has 7,444 edges if zero-edges are skipped, but the number of edges is 9,462 edges if zero-edges are included. This makes the graph properties different and naturally leads to different bandwidths. A bold value Φ_l means that Hebas is able to work on that instance and improve the lower bound. The problem that Hebas runs out of memory on the mentioned computer is marked by an italic value Φ_l.

The selected instances in Table 7.9 are divided into three groups. The first group consists of instances `add20`, `add32`, and `msc10848`. Their original matrices contain zero-edges; therefore, the numbers of edges in the converted graphs are different. Instance `add32` is a special case. The work SRB reported an upper bound of only 21 while the upper bound of their reference work is 669. If the graph takes zero-edges then the initial lower bound is already 276 and this makes the SRB's upper bound 21 invalid. However if zero-edges are skipped then the graph becomes more sparse and the initial lower bound is only 19. That is the reason why we think reporting the lower bounds in both cases might be helpful.

The second group consists of instances that Hebas runs out of memory when improving the lower bound. It starts with instance `gupta` having 16,783 vertices up to the largest instance `pwtk` with 217,918 vertices. Notice that the initial lower bound for these instances is already pretty large. This makes the search space for solving

larger φ becomes huge as already discussed in Section 7.2.2. Hebas stops at the initial lower bounds for these instances.

The last group includes instances that Hebas is able to process and improve the lower bound. Here we present five instances whose sizes are in the range from three thousands to more than fifteen thousands. In particular, `bcsstk24` with 3,562 vertices, `bcspwr10` with 5,300 vertices, `shuttle_eddy` with 10,429 vertices, `vibrobox` with 12,328 vertices, and `barth5` with 15,606 vertices. The largest instance in this group is `barth5`. It has the lower bound improved from 163 to 180, about 10,43% from the initial lower bound.

It should be noticed that if the running time is limited to 12 hours, the lower bound for instance `barth5` is only 178 and `shuttle_eddy` is only 100. In the case of including zero-edges, in 12 hours Hebas can only improve the lower bound for instance `add20` to 392 and `vibrobox` to 1,573. However the improvement continues. Since stronger lower bounds might be helpful to evaluate heuristic methods and also to show that the solver is not yet "saturated" for larger φ in these cases, we let the solver run longer. Therefore, the final lower bounds reported in Table 7.9 as well as in Table A.5 in Appendix A for these four instances (`add20`, `barth5`, `shuttle_eddy`, and `vibrobox`) are computed in 48 hours.

For instances in this set, usually Hebas does not run out of memory and is able to improve the lower bound for an instance if the initial bound is smaller than 500. There are some exceptions. Instance `msc10848` with 10,848 vertices has the lower bound increased from 790 to 792 in both cases. Another very large instance is `vibrobox`. It has 12,328 vertices and the lower bound is improved from 1,360 to 1,476 or from 1,497 to 1,609, depending on whether zero-edges are skipped or not.

The statistics on the lower bound computation for 36 very large instances are reported in Table 7.10. Hebas can improve the lower bounds for 21 instances. The largest such instance is `barth5` with 15,606 vertices. The smallest instance in this set that Hebas cannot improve the lower bound is `add32` with 4,960 vertices. We can find no optimal solution for instances in this set.

We look at the lower bound improvement before finishing the section. In the case zero-edges (ZE) are skipped the average improvement is 4.38%. This average value is 4.51% if the graph includes zero-edges. The largest improvement is found at instance `shuttle_eddy` (10,429 vertices) with 31.65% in both cases.

7.4 Parallel computation

To improve the lower bound further for some benchmark instances, we run the parallel versions of Hebas on the parallel cluster of the Interdisciplinary Center for

Table 7.9: Lower bounds for very large instances.

Matrix			Without zero-edges				With zero-edges			
			Graph		**Hebas**		**Graph**		**Hebas**	
Instance	n	*entries*	m	Φ_t	Φ_l	*imp*	m	Φ_t	Φ_l	*imp*
add20	2,395	17,319	*5,378*	323	**328**	1.55	*7,462*	369	**396**	7.32
add32	4,960	23,884	*7,444*	19	19	0.00	*9,462*	276	*276*	0.00
msc10848	10,848	620,313	*609,464*	790	**792**	0.25	*609,465*	790	**792**	0.25
gupta3	16,783	4,670,105	4,653,322	7,336	*7,336*	0.00	4,653,322	7,336	*7,336*	0.00
ct20stif	52,329	1,375,396	*1,273,983*	1,436	*1,436*	0.00	*1,323,067*	1,436	*1,436*	0.00
twotone	120,750	1,224,224	*1,012,970*	7,727	*7,727*	0.00	*1,029,429*	7,732	*7,732*	0.00
pwtk	217,918	5,926,171	*5,653,257*	814	*814*	0.00	*5,708,253*	814	*814*	0.00
bcsstk24	3,562	81,736	78,174	151	**158**	4.64	78,174	151	**158**	4.64
bcspwr10	5,300	13,571	8,271	132	**135**	2.27	8,271	132	**135**	2.27
shuttle_eddy	10,429	57,014	46,585	79	**104**	31.65	46,585	79	**104**	31.65
vibrobox	12,328	177,578	*144,686*	1,360	**1,476**	8.53	*165,250*	1,497	**1,609**	7.48
barth5	15,606	61,484	45,878	163	**180**	10.43	45,878	163	**180**	10.43

Table 7.10: Statistics on lower bounds for very large instances.

	Without ZE	**With ZE**	**Instance**
LB Improvement (%)			
Average	4.38	4.51	
Largest	31.65	31.65	shuttle_eddy
Smallest > 0	0.25	0.25	
Optimal solutions			
Number	0	0	
Improvement			
Number	21	21	
Largest size	15,606	15,606	barth5
No improvement			
Number	15	15	
Smallest size	4,960	4,960	add32

Scientific Computing (IWR) in Heidelberg [26]. The hardware has been detailed in Section 7.1.2.

We use a cluster of 256 processors (64 computer nodes with 4 processors each) and perform computational experiments on all 113 instances in the popular suite. The running time is limited to 1 hour and the solver is initialized with the solution of the iGPS heuristic. For the time constraint, we do not include the second suite consisting of very large instances in this parallel computation.

This section consists of three parts. The first part describes how parameters of the parallel solver are chosen to make it work efficiently. The performance evaluation of the parallel system is then given. The final part reports the results on the lower bounds for instances in the popular suite.

7.4.1 Tuning system parameters

Since ALPS [56, 57] introduces the intermediate layer of hubs for an effective dynamic load balancing, we need to know how many hubs should be used so that Hebas can work efficiently. In addition, we need to determine whether the 2-labeling scheme will be used with (Hebas 2.4) or without (Hebas 2.3) the hash-table.

The decision is based on the running times required to solve a set of single BW problems. The instances and the search parameter φ are chosen in such a way that they can be solved quickly by the parallel system but in much longer time by the non-parallel solver. Similarly to the performance evaluation in Section 7.2, the value φ is intentionally smaller than the optimal bandwidth and each BW problem characterizes the full search space.

The four problems satisfying our selection criteria are shown in Table 7.11 along with the running times required to solve them in different configurations. Since Hebas 2.3 is the most efficient for small instances in the non-parallel mode and can solve these four BW problems in a reasonable time, it results will be used to compare with those of the parallel version.

The solver's configurations are listed in the first row of Table 7.11. These include Hebas 2.3, its parallel version Hebas 2.3p, and Hebas 2.4p (the parallel version using the 2-labeling scheme with the hash-table). The second row shows the number of used processors (abbreviated *proc* in this section). The running times are in seconds.

Table 7.11: The running times.

				Hebas **2.3**	Hebas **2.3p**			Hebas **2.4p**
Instance	n	m	BW(φ)	1 proc	64 proc	128 proc	256 proc	256 proc
gre___115	115	267	20	30,942	709	367	231	208
impcol_b	59	281	19	172	13	9	9	7
west0156	156	371	33	22,199	509	255	161	137
west0167	167	489	31	7,558	75	46	39	29

From the results in Table 7.11, it can be observed that Hebas 2.4p is relatively faster than Hebas 2.3p. The reason is that even requires more computational time, Hebas 2.4 with the hash-table generates fewer number of subproblems than Hebas 2.3 for these instances. We have commented in Section 7.2 that fewer number of subproblems but more computational work may result in larger running time. It may not be the case in parallel modes because here fewer number of subproblems means less communication overhead and less work on load balancing.

We now consider the number of hubs that should be configured for our cluster of 256 processors. Hebas 2.4p is experimented with three different hub configurations: 1-hub, 8-hubs, and 16-hubs. The corresponding running times are given in Table 7.12.

Table 7.12: The running times with different hub-configurations.

Instance	BW(φ)	1 hub	8 hubs	16 hubs
gre___115	20	207	529	546
impcol_b	19	7	17	25
west0156	33	137	245	355
west0167	31	29	49	72

Surprisingly, Hebas 2.4p requires the least running time when only one hub is configured. This can be understood that our current configuration for the dynamic load

balancing only creates more work for the parallel solver and further improvement on the dynamic load balancing needs to be considered. In any case, we need to choose the most suitable configuration for the solver. Therefore, all the following parallel computations will be done using 1 hub.

7.4.2 The performance of the parallel solver

The performance of the parallel system is evaluated by measuring the speedup ratios and the efficiency factors which have been described in Section 2.4. The speed up ratios are reported in Table 7.13. It should be noted that the hardware in two cases are not precisely the same. The PC in non-parallel mode has CPU of 2.5 GHz, while the computer nodes in the parallel cluster has CPU of 2.8 GHz. Here we only want to know approximately how faster the parallel system is.

Being computed with the running times in Table 7.11, the speedup ratios are reported in Table 7.13. On the cluster of 256 processors, except for the case of instance `impcol_b` whose running time is too short, the running time for solving these instances is about 133 to 193 times faster compared with the non-parallel cases. Notice that the speedup ratio may be larger than the number of used processors, since the CPU of computer nodes in the cluster is stronger than that of the non-parallel computer.

Table 7.13: The speedup ratios S_p.

Instance	BW(φ)	64 proc	128 proc	256 proc
gre___115	20	43.64	84.31	133.95
impcol_b	19	13.23	19.11	19.11
west0156	33	43.61	87.05	137.88
west0167	31	100.77	164.30	193.79

The scalability of the parallel system is computed at three scales: 64, 128, and 256 processors. The results are reported in Table 7.14. We observe that the efficiency of the system decreases when the system is scaled up from 64 to 128 processors, and then to 256 processors. Probably the dynamic load balancing of the system needs to be improved. On the other hand, with this parallel cluster a speedup factor of approximately 133 to 193 is still helpful to improve the lower bounds further for some instances.

7.4.3 Results on the lower bound

It is clear now that using one hub is more suitable for our parallel cluster and the 2-labeling scheme will be used with the hash-table. We performed computational

Table 7.14: The efficiency E_p.

Instance	BW(φ)	64 proc	128 proc	256 proc
gre___115	20	0.68	0.66	0.52
impcol_b	19	0.21	0.15	0.07
west0156	33	0.68	0.68	0.54
west0167	31	1.57	1.28	0.76

experiments on instances in the first suite using the strongest configuration of the single-labeling and 2-labeling schemes, i.e., Hebas 1.4p and Hebas 2.4p.

The lower bounds for selective small instances are reported in Table 7.15. The detailed results can be found in Table A.1 in Appendix A. Here *Hebas* means the best results of both single-labeling and 2-labeling schemes in the non parallel mode, and *Hebasp* means the best results of both schemes in parallel mode. The meaning of the values is exactly the same as in the similar table in the non-parallel mode. For ease of following the lower bound improvement we show again the results of MCP and the best results obtained in the non-parallel mode.

There is no new lower bound for small instances obtained by Hebas 1.4p compared with the non-parallel solver using the single-labeling scheme. The lower bounds of two instances `west0156` and `will199` continue being worse than those of MCP. However, Hebas 2.4p does show its efficiency for small instances. In particular, it can improve the lower bound for instances `gre__115` from 20 to 21 and for `lund_a` and `lund_b` from 19 to 20. There is no instance whose lower bound is worse than the reference any more; the lower bound for instance `will199` is increased to 57 and Hebas 2.3 already proved the lower bound 34 for instance `west0156`.

The statistics on the lower bounds for 33 small instances computed by the parallel solvers are given in Table 7.16. The average improvement of lower bound by MCP is 14.22%. It is increased to 15.79% by the solver in the non-parallel mode and then to 16.48% by the parallel solver. The number of optimal solutions remains 20. Comparing with MCP, Hebas 1.4p using the single-labeling scheme still has worse lower bounds in two cases. However, Hebas 2.4p leaves no worse results and the number of better cases is increased to 10.

We continue the parallel computation for 80 large instances. Similar to the non-parallel case, only the results of selective instances are introduced in Table 7.17 and the detailed results can be found in Table A.3 in Appendix A. For ease of following the lower bound improvement, the same set of selective instances reported in the non-parallel mode is presented again, except for the ones which have been solved to optimality.

The first group shows the only three instances that Hebas 1.4 in non-parallel mode obtained worse lower bounds than MCP. These are reduced to two cases since

Table 7.15: Parallel computation on selective small instances.

Graph				**MCP**		**Hebas**		**Hebasp**			
				Best		Best		1.4p	2.4p	Best	
Instance	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	Φ_l	*imp*
bcsstk22	110	254	8	9	12.50	10	25.00	10	10	10	25.00
fs_183_1	183	701	52	52	0.00	57	9.62	57	57	57	9.62
gre___115	115	267	16	20	25.00	20	25.00	20	**21**	21	31.25
gre___185	185	650	16	17	6.25	18	12.50	18	18	18	12.50
impcol_b	59	281	15	19	26.67	20	33.33	20	20	20	33.33
impcol_c	137	352	21	26	23.81	27	28.57	27	27	27	28.57
lund_a	147	1151	17	19	11.76	19	11.76	19	**20**	20	17.65
lund_b	147	1147	17	19	11.76	19	11.76	19	**20**	20	17.65
west0132	132	404	23	25	8.70	28	21.74	28	28	28	21.74
west0156	156	371	27	34	25.93	34	25.93	*33*	34	34	25.93
west0167	167	489	28	31	10.71	32	14.29	31	32	32	14.29
will199	199	660	44	57	29.55	*55*	25.00	*55*	57	57	29.55

Table 7.16: Statistics of parallel computation results for small instances.

	MCP	**Hebas**	**1.4p**	**Hebasp**
LB Improvement (%)				
Average	14.22	15.79	15.57	16.48
Largest	30.00	33.33	33.33	33.33
Smallest > 0	1.61	1.61	1.61	1.61
Optimal solutions				
Number	16	20	20	20
Versus MCP				
Number of better		7	6	10
Number of worse		1	2	0

Table 7.17: Parallel computation on selective large instances.

Graph				MCP		Hebas		Hebasp			
				Best		Best		1.4p	2.4p	Best	
Instance	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	Φ_l	*imp*
dwt___419	419	1572	21	23	9.52	23	9.52	*22*	23	23	9.52
str___600	363	3244	79	101	27.85	101	27.85	**103**	102	103	30.38
west0381	381	2150	100	119	19.00	**120**	20.00	*118*	**121**	121	21.00
mbeacxc	487	41686	243	248	2.06	**250**	2.88	249	**251**	251	3.29
west0655	655	2841	100	109	9.00	**125**	25.00	124	125	125	25.00
bp___1600	822	4809	192	199	3.65	**201**	4.69	**203**	_	203	5.73
dwt___918	918	3233	27	27	0.00	**29**	7.41	29	_	29	7.41
west0989	989	3500	121	123	1.65	**165**	36.36	165	_	165	36.36

Hebas 1.4p can improve the lower bound for instance `str__600` to 103 compared with 101 of MCP. Recall that Hebas 2.3 (the 2-labeling scheme) already obtains either equally good or better results for these three instances in the non-parallel mode. Hebas 2.3p continues improving the lower bound for instance `west0381` from 119 by MCP to 121.

We also observe further improvement by the parallel solver in the second group. The last group consists of instances that the non-parallel solver using the 2-labeling scheme runs out of memory. Hebas 2.4p does not encounter this memory problem on the parallel cluster. However, we still leave a dash "_" here because Hebas 2.4p cannot prove the lower bound obtained by Hebas 1.4 in the specified time (one hour).

The statistics on the lower bounds for large instances obtained by the parallel computation are given in Table 7.18. Recall that the average improvement of lower bound achieved by MCP is 5.75% and is improved solely by the non-parallel Hebas solver to 9.49%. Hebas 1.4p can improve this value to 9.53%. The best results in the parallel mode improve it further to 9.68%. No new optimal solution has been found.

We notice that even the parallel computation can improve the results of the non-parallel solver, the change is pretty small. This has been already explained in Section 7.2.2 about the problem's search space; if the solver does not have a constraint useful for the graph then the search space of BW expands very quickly when the search parameter φ is increased. Since the speedup ratios of this parallel system are limited by approximately 256, the small improvement for large instances is understandable.

Compared with the results of MCP, Hebas 1.4p obtains better lower bounds for 54 instances and has worse results in two cases. When these are combined with the

Table 7.18: Statistics of parallel computation results for large instances.

	MCP	Hebas	1.4p	Hebasp
LB Improvement (%)				
Average	5.75	9.49	9.53	9.68
Largest	30.43	36.36	37.19	37.19
Smallest > 0	0.53	1.08	1.08	1.08
Optimal solutions				
Number	5	12	12	12
Versus MCP				
Number of better		54	54	55
Number of worse		0	2	0

results of Hebas 2.4p, there is no more worse result and better lower bounds are observed for 55 instances among the total of 80 large instances.

7.5 The upper bounds

In this section we report the results of our heuristic methods for instances in two benchmark suites and compare them with the best known results from the literature.

The experimented heuristics include the original GPS heuristic implemented by us, its modified version iGPS, the simulated annealing heuristic SA-σ which uses the approximate objective function Sigma, and the exact solver Hebas. For each instance, the best lower bound obtained by Hebas is used for computing the factor *gap* between it and the upper bounds obtained by these heuristics. Notice that SA-σ is initialized with a feasible solution obtained by iGPS and Hebas also starts with an initial upper bound provided by SA-σ.

Recall that the best known upper bounds for small and large instances are obtained by taking the best value from many individual methods in MCP [37, 43, 4, 36] and LRX [34]. We use the lower bounds from MCP [36] for computing the factor *gap* for these upper bounds. For very large instances, the best known upper bounds are obtained from SRB [50] and their *gaps* are computed using the best lower bounds obtained by Hebas.

7.5.1 Heuristic results for small instances

For small instances SA-σ is set to run at most 3 minutes and the neighbor function is the *swap* method. Since the amount of data is large, only the statistics of the results are included in this section in Table 7.19. The detailed results can be found in Table A.2 in Appendix A.

Table 7.19: Statistics of heuristic results for small instances.

	MCP & LRX	**GPS**	**iGPS**	**SA-σ**	**Hebas**
UB Deviation (%)					
Average	21.76	62.63	54.60	27.81	22.16
Largest	50.00	131.82	109.09	75.00	52.27
Smallest > 0	1.61	12.50	12.50	4.84	1.61
Optimal solutions					
Number	18	4	4	13	20
Gap (%)					
Average	6.63	39.24	32.40	9.38	4.75
Largest	32.00	105.26	71.93	33.33	21.43
Smallest > 0	3.85	10.00	7.69	3.17	3.85

We begin with the GPS heuristic which is one of the early published methods. Its average deviation from the initial lower bound is 62.63% and the largest deviation is 131.82%. iGPS improves the solution quality a bit further in that the average deviation is reduced to 54.60%. The two heuristics can find optimal solutions for 4 among 33 instances in this set. Both GPS and iGPS are very fast; they finish every instance in this set in far less than 1 second. SA-σ continues improving the upper bounds and reduces the average deviation to 27.81%. This is still worse than the best known values by MCP and LRX whose average deviation is 21.76%.

Since we really want to reduce the gaps between lower and upper bounds for small instances, in cases that Hebas in the non-parallel mode does not have good upper bounds as those of MCP and LRX (such as instances `gre__115`, `fs_183_1`, and `will199`) we try to obtain them with the parallel solver. Hebas can reduce the average deviation to 22.16%.

The known number of small instances which can be solved to optimality is increased by Hebas from 18 to 20. Now we look at the gaps between lower and upper bounds to see how far the bandwidth problem has been solved. The best known average gap by MCP and LRX is 6.63% and has been improved by Hebas to 4.75%.

7.5.2 Heuristic results for large instances

For large instances SA-σ is set to run at most 30 minutes and the neighbor function is also the *swap* method. The statistics of the upper bounds are reported in Table 7.20. The detailed results can be found in Table A.4 in Appendix A.

For 80 large instances with less than 1000 vertices the solution quality of GPS begins to decrease. Its results are worse than those on the small instances. In particular, the average deviation given by GPS is 92.50% and the average gap to Hebas lower bounds is 75.55%. iGPS improves the solution quality of the original heuristic about 14.50%;

Table 7.20: Statistics of heuristic results for large instances.

	MCP & LRX	**GPS**	**iGPS**	**SA-σ**	**Hebas**
UB Deviation (%)					
Average	29.87	92.50	78.00	37.53	35.76
Largest	85.71	227.78	171.43	171.43	171.43
Smallest > 0	3.67	6.67	6.67	4.35	4.00
Optimal solutions					
Number	8	3	3	7	12
Gap (%)					
Average	22.55	75.55	62.16	25.05	23.31
Largest	70.73	218.92	137.50	137.50	137.50
Smallest > 0	2.01	7.69	7.69	1.59	1.59

it reduces the average deviation to 78.00%. The average gap is reduced by iGPS to 62.16%. GPS again finishes all these instances in less than 1 second. The average running time by iGPS for these instances is 6.83 seconds.

SA-σ can reduce the average deviation to 37.53%. Hebas continues to improve the upper bound for some instances and reduces the average deviation to 35.76%. The upper bounds from MCP and LRX are still the best known results for large instances with an average UB deviation of 29.87%.

There are two reasons that the results of Hebas for these instances are worse than the best known upper bounds. First, the solution quality of SA-σ whose output is used as the initial solution for Hebas is not as good as the best known results. Second, for some large instances the initial upper bounds are pretty large and cannot be improved further by the exact solver Hebas. Recall that the search space of BW is huge if the search parameter φ is large. This makes it hard for Hebas to search for a feasible permutation.

Both GPS and iGPS can find optimal solutions for 3 instances in this set. The best known results from MCP and LRX reported 8 cases of optimality. SA-σ can obtain the optimal solution for 7 instances since it is initialized with the solution by iGPS. Hebas with strong exact algorithms can solve 12 instances to optimality. Now we consider the gaps between lower and upper bounds. The average gap of best known results by MCP and LRX is 22.55%. Even having stronger lower bounds, Hebas can only reduce the average gap to 23.31%.

We observe that even though more optimal solutions are found and the lower bounds for many instances have been improved, the gaps between the lower and upper bounds are still huge for many large instances. The bandwidth problem is challenging and more improvements are still needed.

7.5.3 Heuristic results for very large instances

We experimented our heuristics on 51 very large instances whose matrix files are downloaded directly from the collection [11]. As we have mentioned in the report on lower bounds in Section 7.3.3, the SRB work must have excluded zero-edges in the matrix files since otherwise their upper bounds for some instances will be invalid. Therefore, our heuristics also skip zero-edges when converting these matrices to graphs for computing the bandwidth.

Only GPS, iGPS, and SA-σ are applied for instances this suite. Hebas cannot be used for improving upper bounds because the instances' size and the initial upper bounds are too large. The upper bound for each disconnected instance is obtained by taking the largest bandwidth among all of its connected components. For instances in this suite, SA-σ is set to run 30 minutes and the neighborhood function is the *rotation* method. We observe that SA-σ with *rotation* gives equivalent solution quality as the one with *swap*.

The detailed computational results can be found in Table A.6 in Appendix A. The running times are reported in seconds. The statistics of the upper bounds obtained by our heuristics and SRB are given in Table 7.21. The first two columns show the results of two configurations in SRB work. The upper bounds by SRB-M_5 are reported with the running time and those by SRB-M_{100} are their best results.

Table 7.21: Statistics of heuristic results for very large instances.

	SRB-M_5	SRB-M_{200}	GPS	iGPS	SA-σ
UB Deviation (%)					
Average	54.26	32.57	100.63	90.13	80.36
Largest	188.58	118.49	285.10	284.05	283.14
Smallest	8.85	4.14	9.68	9.68	9.68
Optimal solutions					
Number	0	0	0	0	0
Time (seconds)					
Average	230.21	_	4.31	53.41	1800
Gap (%)					
Average	48.93	27.80	93.44	83.39	74.14
Largest	188.58	118.49	285.10	284.05	283.14
Smallest	6.96	3.80	8.00	8.00	8.00

The GPS heuristic gives an average deviation of 100.63%. This value is improved by iGPS to 90.13% and SA-σ reduces it to 80.36%. The results by SRB-M_5 (the first configuration in SRB) yield an average deviation of 54.26%. This is improved further by their finer configuration SRB-M_{200} to 32.57%. Clearly SRB has the best upper bounds for instances in this suite.

The running times are worth noticing. The average running time of SRB-M_5 is 230.21 seconds. The average running time of GPS is only 4.31 seconds and that value of iGPS is 53.41 seconds. (The mentioned running time of iGPS does not include the time for computing initial lower bounds whose average value is 1,100 seconds.) This confirms again the fastness of the GSP heuristic which might be useful for obtaining initial solutions for some solution methods. For exact methods which requires initial lower bounds, iGPS is a better choice because it can make use of the data structures generated during the lower bound computation to obtain a better initial solution.

No optimal solution has been observed for instances in this suite. Obviously we need stronger lower bound as well as upper bound methods. Concerning the gaps between lower and upper bounds, SA-σ can achieve an average gap of 74.14%. The reported upper bounds by SRB-M_{200} are clearly the best; their average gap is 27.80%.

We observe that for very large sparse graphs, an SA method which uses simple neighbor functions without considering the graph properties is not effective anymore. Neighbor functions which create more diversity are required for improving the solution quality of the SA method. Since the graph is very sparse and usually consists of subgraphs connecting with each other through a few edges, divide-and-conquer strategies might be useful because they can exploit the graph properties.

Another remark is that heuristics exploiting graph properties such as GPS can provide initial solutions in very short time. In addition, we observe that the vertex numbering procedure of GPS and iGPS, i.e., the steps that assign labels to vertices in a level structure, is somewhat similar to the procedure extending a partial permutation. One may consider improving the solution quality of GPS and iGPS further by using a numbering procedure which takes into account the constraints for the bandwidth problem.

8 Conclusions and Discussion

In this dissertation we have studied the bandwidth minimization problem from many angles: heuristics, applications, and exact methods. We focused on improving the lower bound for benchmark instances using exact methods.

We presented an application of the bandwidth problem to the compression of topological information of digital road networks. On an evaluation set of real-world European road network data, a noteworthy compression ratio of topological information was achieved by reordering the graphs' vertex numbers using the solution of the bandwidth problem. The compression technique may inspire similar applications where a vertex reordering makes the storage of some data more efficient.

The GPS heuristic was adapted to exploit the data structure available during the lower bound computation of exact algorithms. Our version, named iGPS, is somewhat slower but gives better solution quality than the original one. We observe that the vertex numbering procedure of GPS is quite similar to the extension of a partial permutation. Thus one may consider improving its solution quality by using a vertex numbering step which takes into account the constraints for the bandwidth problem.

We introduced an approximate objective function named Sigma. It is composed of two parts. The integral part preserves the original bandwidth value. The fractional part considers all label differences in a permutation; thus it is very likely to change when the permutation is altered. We applied Sigma to a simulated annealing implementation named SA-σ and it is able to improve the upper bounds obtained by iGPS. The current solution quality of SA-σ is not as good as that of the best known results, but there is room for improvement with the availability of the SA framework.

While a method for solving the bandwidth problem using IP formulations is not yet available, the problem can be efficiently approached using branch-and-bound methods. It turns out that known exact methods are based on the same concepts: the partial permutation, the search problem BW, and the extendability problems. We reviewed previous exact algorithms based on these concepts and used them as a basis for our improvement. The further development includes: new constraints for the bandwidth problem, the analysis of the dominance relation between certain partial permutations which allows eliminating dominated partial permutations, and a new branching scheme named 2-labeling which exploits the dominance rule.

The new constraints are the main engine to improve the lower bound for most instances. Using label domains as integer intervals in a partial permutation allows them to be numerically tightened by constraints for the bandwidth problem. The constraints can be derived by applying the basic bandwidth constraint to specific properties of the graph, like the fitting constraint or the pulling constraint. They are strong constraints and can tighten label domains effectively. Other constraints, such as the density cut and the density near-cut constraints, can be derived by analyzing the distribution of label domains on the linear layout and finding label ranges which are fully used. These constraints also enable label domains to be tightened on the side where the partial permutation is not being extended.

We observe that the search space of the BW problem expands quickly when the search parameter φ is increased by only 1. This makes it difficult to improve the lower bound for a graph instance if the solver does not have a constraint which is useful for that graph.

Among two partial permutations having the same assigned set, in some cases it is guaranteed that if a partial permutation can be extended to a feasible permutation then so can the other. This leads to an interesting property of the BW problem: the dominance relation between partial permutations having the same assigned set. We explained the dominance concept and showed that dominated partial permutations can be eliminated. We also described how to use the dominance rule in a branch-and-bound algorithm with a hash-table. This implementation usually speeds up the running time.

Furthermore, the dominance rule can be applied at the branching step. In the 2-labeling scheme two vertices are used simultaneously to extend partial permutations instead of one-by-one. On small instances (less than 200 vertices), this scheme speeds up the running time in most cases. The main disadvantage of this scheme is its memory usage because the width of its branch-and-bound tree is larger than that of the normal single-labeling scheme. Therefore, it has memory problem on some large instances (between 200 and 1,000 vertices). On computers having much larger memory capacity such as a parallel cluster, the memory issue may not happen anymore. Despite that issue, the 2-labeling scheme still contributes to the improvement of the lower bound for some large instances.

Notice that the current dominance rule is derived by analyzing labels of assigned vertices which are adjacent to a free vertex. In other words, these rules are based on certain sets of adjacent vertices like the constraints for the bandwidth problem in the 1980s. We think that it can be strengthened further, like the way the constraints have been improved, by considering sets of vertices which are more than one distance unit away from each other in addition to the adjacent ones. The first applications are obvious: being used in a branch-and-bound algorithm either directly with the hash-table and/or the 2-labeling scheme.

Parallelizing the branch-and-bound algorithm and then running the solver on a powerful cluster of 256 processors also help to improve the lower bound for some instances. However, parallelization can only reduce the running time within a linear speedup factor while the BW problem's search space increases rapidly for solving larger φ. As a result, the lower bound for many instances can only be improved to a small degree, or even not improved at all, when compared with the results of the non-parallel solver. On the other hand, a parallel cluster can be seen as a "special" computer with high computing power and large memory capacity, which is useful for solving very large instances or for methods requiring much memory like the 2-labeling scheme.

The new constraints and the dominance rule have shown their effectiveness. On the popular benchmark suite consisting of 113 instances with less than 1,000 vertices each, we are able to solve more instances to optimality and obtain better lower bounds for many instances, compared with the best known results from the literature. Since the branch-and-bound algorithms are stronger with the new constraints and the dominance rule, the solver can work on instances of much larger size. On the second suite consisting of 36 instances with more than 1,000 vertices, the largest instance whose lower bound can be improved by our solver has about 15,600 vertices.

In summary, the bandwidth minimization problem has practical applications since its solutions can make the storage or computation of some data more efficient. The problem is hard and still challenging: the gap between the lower and upper bound is still large for many instances in the popular benchmark suite. We have introduced some additional tools for approaching the problem. The approximate objective function Sigma can be useful for some heuristics. For exact methods, our contributions include the formulation of the partial permutation concept, the new constraints, and the analysis of the dominance relation between partial permutations having the same assigned set. The exact methods for the bandwidth problem presented in this dissertation, in combination with Caprara and Salazar-González's work [6], provide a framework and some efficient tools for further research.

Appendix A

Detailed computational results

Table A.1: Lower bounds for small instances.

Instance				**MCP**		**VR08**		**Hebas**				**Parallel Hebas**			
				Best known		Best		1.4		2.3		1.4p		2.4p	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*
arc130	130	715	62	63	1.61	63	1.61	63	1.61	63	1.61	63	1.61	63	1.61
ash85	85	219	8	9	12.50	9	12.50	9	12.50	9	12.50	9	12.50	9	12.50
bcspwr01	39	46	4	5	25.00	5	25.00	5	25.00	5	25.00	5	25.00	5	25.00
bcspwr02	49	59	6	7	16.67	7	16.67	7	16.67	7	16.67	7	16.67	7	16.67
bcspwr03	118	179	9	10	11.11	10	11.11	10	11.11	10	11.11	10	11.11	10	11.11
bcsstk01	48	176	15	16	6.67	16	6.67	16	6.67	16	6.67	16	6.67	16	6.67
bcsstk04	132	1758	32	37	15.63	37	15.63	37	15.63	37	15.63	37	15.63	37	15.63
bcsstk05	153	1135	16	20	25.00	20	25.00	20	25.00	20	25.00	20	25.00	20	25.00
bcsstk22	110	254	8	9	12.50	**10**	25.00	**10**	25.00	10	25.00	10	25.00	10	25.00
can___144	144	576	12	13	8.33	13	8.33	13	8.33	13	8.33	13	8.33	13	8.33
can___161	161	608	16	18	12.50	18	12.50	18	12.50	18	12.50	18	12.50	18	12.50
curtis54	54	124	8	10	25.00	10	25.00	10	25.00	10	25.00	10	25.00	10	25.00
dwt___234	117	162	10	11	10.00	11	10.00	11	10.00	11	10.00	11	10.00	11	10.00
fs_183_1	183	701	52	52	0.00	52	0.00	**57**	9.62	57	9.62	57	9.62	57	9.62
gent113	104	549	20	26	30.00	26	30.00	26	30.00	26	30.00	26	30.00	26	30.00
gre___115	115	267	16	20	25.00	**21**	31.25	20	25.00	20	25.00	20	25.00	**21**	31.25
gre___185	185	650	16	17	6.25	**18**	12.50	**18**	12.50	18	12.50	18	12.50	18	12.50
ibm32	32	90	9	11	22.22	11	22.22	11	22.22	11	22.22	11	22.22	11	22.22
impcol_b	59	281	15	19	26.67	**20**	33.33	**20**	33.33	20	33.33	20	33.33	20	33.33
impcol_c	137	352	21	26	23.81	26	23.81	**27**	28.57	27	28.57	27	28.57	27	28.57
lns___131	123	275	15	19	26.67	19	26.67	19	26.67	19	26.67	19	26.67	19	26.67
lund_a	147	1151	17	19	11.76	19	11.76	19	11.76	19	11.76	19	11.76	**20**	17.65
lund_b	147	1147	17	19	11.76	19	11.76	19	11.76	19	11.76	19	11.76	**20**	17.65
mcca	168	1662	32	32	0.00	32	0.00	32	0.00	32	0.00	32	0.00	32	0.00
nos1	158	312	3	3	0.00	3	0.00	3	0.00	3	0.00	3	0.00	3	0.00
nos4	100	247	9	10	11.11	10	11.11	10	11.11	10	11.11	10	11.11	10	11.11

Continued on the next page

Table A.1: Lower bounds for small instances.

Instance				**MCP**		**VR08**		**Hebas**				**Parallel Hebas**			
				Best known		Best		1.4		2.3		1.4p		2.4p	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*	Φ_l	*imp*
pores_1	30	103	6	7	16.67	7	16.67	7	16.67	7	16.67	7	16.67	7	16.67
steam3	80	424	7	7	0.00	7	0.00	7	0.00	7	0.00	7	0.00	7	0.00
west0132	132	404	23	25	8.70	25	8.70	**28**	21.74	28	21.74	28	21.74	28	21.74
west0156	156	371	27	34	25.93	34	25.93	*33*	22.22	34	25.93	*33*	22.22	34	25.93
west0167	167	489	28	31	10.71	31	10.71	31	10.71	**32**	14.29	31	10.71	32	14.29
will199	199	660	44	57	29.55	**59**	34.09	*53*	20.45	*55*	25.00	*55*	25.00	57	29.55
will57	57	127	6	6	0.00	6	0.00	6	0.00	6	0.00	6	0.00	6	0.00
Average					**14.22**		**15.32**		**15.44**		**15.79**		**15.57**		**16.48**

Table A.2: Upper bounds for small instances.

Instance			**MCP & LRX**				**Our results**									
			Best known				Hebas	GPS		iGPS		SA-σ		Hebas		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	*gap*	Φ_l	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	*gap*
arc130	130	62	63	63	1.61	0.00	63	101	62.90	101	62.90	65	4.84	63	1.61	0.00
ash85	85	8	9	9	12.50	0.00	9	10	25.00	10	25.00	10	25.00	9	12.50	0.00
bcspwr01	39	4	5	5	25.00	0.00	5	7	75.00	6	50.00	5	25.00	5	25.00	0.00
bcspwr02	49	6	7	7	16.67	0.00	7	11	83.33	11	83.33	7	16.67	7	16.67	0.00
bcspwr03	118	9	10	10	11.11	0.00	10	16	77.78	16	77.78	10	11.11	10	11.11	0.00
bcsstk01	48	15	16	16	6.67	0.00	16	25	66.67	23	53.33	20	33.33	16	6.67	0.00
bcsstk04	132	32	37	37	15.63	0.00	37	46	43.75	43	34.38	41	28.13	37	15.63	0.00
bcsstk05	153	16	20	20	25.00	0.00	20	25	56.25	24	50.00	22	37.50	20	25.00	0.00
bcsstk22	110	8	9	10	25.00	11.11	10	12	50.00	12	50.00	11	37.50	10	25.00	0.00
can___144	144	12	13	13	8.33	0.00	13	18	50.00	14	16.67	13	8.33	13	8.33	0.00
can___161	161	16	18	18	12.50	0.00	18	18	12.50	18	12.50	18	12.50	18	12.50	0.00
curtis54	54	8	10	10	25.00	0.00	10	13	62.50	13	62.50	10	25.00	10	25.00	0.00
dwt___234	117	10	11	11	10.00	0.00	11	14	40.00	13	30.00	11	10.00	11	10.00	0.00
fs_183_1	183	52	52	60	15.38	15.38	57	117	125.00	98	88.46	64	23.08	61	17.31	7.02
gent113	104	20	26	27	35.00	3.85	26	33	65.00	33	65.00	28	40.00	27	35.00	3.85
gre___115	115	16	20	23	43.75	15.00	21	33	106.25	28	75.00	28	75.00	23	43.75	9.52
gre___185	185	16	17	21	31.25	23.53	18	21	31.25	21	31.25	21	31.25	21	31.25	16.67
ibm32	32	9	11	11	22.22	0.00	11	13	44.44	13	44.44	11	22.22	11	22.22	0.00
impcol_b	59	15	19	20	33.33	5.26	20	29	93.33	29	93.33	25	66.67	20	33.33	0.00
impcol_c	137	21	26	30	42.86	15.38	27	46	119.05	41	95.24	31	47.62	31	47.62	14.81
lns___131	123	15	19	20	33.33	5.26	19	30	100.00	30	100.00	20	33.33	20	33.33	5.26
lund_a	147	17	19	23	35.29	21.05	20	23	35.29	23	35.29	23	35.29	23	35.29	15.00
lund_b	147	17	19	23	35.29	21.05	20	23	35.29	23	35.29	23	35.29	23	35.29	15.00
mcca	168	32	32	37	15.63	15.63	32	53	65.63	53	65.63	37	15.63	37	15.63	15.63
nos1	158	3	3	3	0.00	0.00	3	3	0.00	3	0.00	3	0.00	3	0.00	0.00
nos4	100	9	10	10	11.11	0.00	10	11	22.22	11	22.22	10	11.11	10	11.11	0.00

Continued on the next page

Table A.2: Upper bounds for small instances.

Instance			**MCP & LRX**				**Our results**									
			Best known				Hebas	GPS		iGPS		SA-σ		Hebas		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	*gap*	Φ_l	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	*gap*
pores_1	30	6	7	7	16.67	0.00	7	7	16.67	7	16.67	7	16.67	7	16.67	0.00
steam3	80	7	7	7	0.00	0.00	7	7	0.00	7	0.00	7	0.00	7	0.00	0.00
west0132	132	23	25	33	43.48	32.00	28	47	104.35	47	104.35	37	60.87	34	47.83	21.43
west0156	156	27	34	37	37.04	8.82	34	55	103.70	55	103.70	40	48.15	37	37.04	8.82
west0167	167	28	31	34	21.43	9.68	32	64	128.57	49	75.00	34	21.43	34	21.43	6.25
will199	199	44	57	66	50.00	15.79	57	102	131.82	92	109.09	70	59.09	67	52.27	17.54
will57	57	6	6	6	0.00	0.00	6	8	33.33	8	33.33	6	0.00	6	0.00	0.00
Average					**21.76**	**6.63**			**62.63**		**54.60**		**27.81**		**22.16**	**4.75**

Table A.3: Lower bounds for large instances.

Instance				**MCP**		**Hebas**					**Parallel Hebas**				
				Best known		1.4		2.3	Best		1.4p		2.4p	Best	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	*imp*
494_bus	494	586	24	25	4.17	**26**	8.33	26	26	8.33	26	8.33	26	26	8.33
662_bus	662	906	36	36	0.00	**37**	2.78	37	37	2.78	37	2.78	37	37	2.78
685_bus	685	1,282	29	30	3.45	**31**	6.90	31	31	6.90	31	6.90	31	31	6.90
ash292	292	958	16	16	0.00	**17**	6.25	17	17	6.25	17	6.25	17	17	6.25
bcspwr04	274	669	23	23	0.00	23	0.00	23	23	0.00	23	0.00	23	23	0.00
bcspwr05	443	590	25	25	0.00	**26**	4.00	26	26	4.00	26	4.00	26	26	4.00
bcsstk06	420	3,720	32	38	18.75	**39**	21.88	39	39	21.88	39	21.88	39	39	21.88
bcsstk19	817	3,018	12	13	8.33	13	8.33	_	13	8.33	13	8.33	_	13	8.33
bcsstk20	467	1,295	7	8	14.29	8	14.29	8	8	14.29	8	14.29	8	8	14.29
bcsstm07	420	3,416	32	37	15.63	**39**	21.88	39	39	21.88	39	21.88	39	39	21.88
bp_____0	822	3,260	174	174	0.00	174	0.00	*174*	174	0.00	174	0.00	174	174	0.00
bp____200	822	3,788	186	186	0.00	**188**	1.08	_	188	1.08	188	1.08	_	188	1.08
bp____400	822	4,015	188	188	0.00	**193**	2.66	_	193	2.66	**195**	3.72	_	195	3.72
bp____600	822	4,157	189	190	0.53	**195**	3.17	_	195	3.17	**197**	4.23	_	197	4.23
bp____800	822	4,518	190	197	3.68	**200**	5.26	_	200	5.26	**201**	5.79	_	201	5.79
bp___1000	822	4,635	191	197	3.14	**200**	4.71	_	200	4.71	**202**	5.76	_	202	5.76
bp___1200	822	4,698	193	197	2.07	**201**	4.15	_	201	4.15	**202**	4.66	_	202	4.66
bp___1400	822	4,760	193	199	3.11	**201**	4.15	_	201	4.15	**203**	5.18	_	203	5.18
bp___1600	822	4,809	192	199	3.65	**201**	4.69	_	201	4.69	**203**	5.73	_	203	5.73
can___292	292	1,124	33	34	3.03	**35**	6.06	35	35	6.06	35	6.06	35	35	6.06
can___445	445	1,682	42	46	9.52	46	9.52	46	46	9.52	46	9.52	46	46	9.52
can___715	715	2,975	52	54	3.85	**56**	7.69	56	56	7.69	56	7.69	56	56	7.69
can___838	838	4,586	75	75	0.00	**81**	8.00	_	81	8.00	81	8.00		81	8.00
dwt___209	209	767	20	21	5.00	**22**	10.00	22	22	10.00	22	10.00	22	22	10.00
dwt___221	221	704	11	12	9.09	12	9.09	12	12	9.09	12	9.09	12	12	9.09
dwt___245	245	608	21	21	0.00	21	0.00	21	21	0.00	21	0.00	21	21	0.00

Continued on the next page

Table A.3: Lower bounds for large instances.

Instance				**MCP**		**Hebas**					**Parallel Hebas**				
				Best known		1.4		2.3	Best		1.4p		2.4p	Best	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	*imp*
dwt___310	310	1,069	11	11	0.00	11	0.00	11	11	0.00	11	0.00	11	11	0.00
dwt___361	361	1,296	14	14	0.00	14	0.00	14	14	0.00	14	0.00	14	14	0.00
dwt___419	419	1,572	21	23	9.52	*22*	4.76	23	23	9.52	*22*	4.76	23	23	9.52
dwt___503	503	2,762	29	29	0.00	**34**	17.24	34	34	17.24	34	17.24	34	34	17.24
dwt___592	592	2,256	22	22	0.00	22	0.00	_	22	0.00	22	0.00	_	22	0.00
dwt___878	878	3,285	23	23	0.00	23	0.00	_	23	0.00	23	0.00	_	23	0.00
dwt___918	918	3,233	27	27	0.00	**29**	7.41	_	29	7.41	29	7.41	_	29	7.41
dwt___992	992	7,876	35	35	0.00	35	0.00	_	35	0.00	35	0.00	_	35	0.00
fs_541_1	541	2,466	270	270	0.00	270	0.00	_	270	0.00	270	0.00	_	270	0.00
fs_680_1	680	1,464	17	17	0.00	17	0.00	17	17	0.00	17	0.00	17	17	0.00
fs_760_1	760	3,518	35	36	2.86	**37**	5.71	37	37	5.71	37	5.71	37	37	5.71
gr_30_30	900	3,422	31	31	0.00	31	0.00	_	31	0.00	31	0.00	_	31	0.00
gre___343	343	1,092	23	23	0.00	**25**	8.70	25	25	8.70	25	8.70	25	25	8.70
gre___512	512	1,680	30	30	0.00	**31**	3.33	31	31	3.33	31	3.33	31	31	3.33
gre_216a	216	660	17	17	0.00	**18**	5.88	18	18	5.88	18	5.88	18	18	5.88
hor___131	434	2,138	42	46	9.52	**47**	11.90	47	47	11.90	47	11.90	47	47	11.90
impcol_a	206	557	23	30	30.43	**31**	34.78	31	31	34.78	31	34.78	31	31	34.78
impcol_d	425	1,267	34	36	5.88	36	5.88	36	36	5.88	36	5.88	36	36	5.88
impcol_e	225	1,187	34	34	0.00	**36**	5.88	36	36	5.88	36	5.88	36	36	5.88
jagmesh1	936	2,664	22	24	9.09	**25**	13.64	_	25	13.64	25	13.64	_	25	13.64
jpwh_991	983	2,678	79	82	3.80	**84**	6.33	_	84	6.33	84	6.33	_	84	6.33
lns___511	503	1,425	30	33	10.00	**35**	16.67	35	35	16.67	35	16.67	35	35	16.67
mbeacxc	487	41,686	243	248	2.06	**249**	2.47	**250**	250	2.88	249	2.47	**251**	251	3.29
mbeaflw	487	41,686	243	246	1.23	**249**	2.47	**250**	250	2.88	249	2.47	**251**	251	3.29
mbeause	492	36,209	245	249	1.63	**251**	2.45	**252**	252	2.86	252	2.86	252	252	2.86
mcfe	731	15,086	112	112	0.00	**121**	8.04	121	121	8.04	121	8.04	121	121	8.04
nnc261	261	794	20	22	10.00	**23**	15.00	23	23	15.00	23	15.00	23	23	15.00

Continued on the next page

Table A.3: Lower bounds for large instances.

Instance				**MCP**		**Hebas**					**Parallel Hebas**				
				Best known		1.4		2.3	Best		1.4p		2.4p	Best	
Name	n	m	Φ_t	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	*imp*	Φ_l	*imp*	Φ_l	Φ_l	*imp*
nnc666	666	2,148	32	33	3.13	**35**	9.38	35	35	9.38	35	9.38	35	35	9.38
nos2	638	1,272	3	3	0.00	3	0.00	3	3	0.00	3	0.00	3	3	0.00
nos3	960	7,442	43	43	0.00	43	0.00	*43*	43	0.00	43	0.00	43	43	0.00
nos5	468	2,352	51	53	3.92	**57**	11.76	57	57	11.76	57	11.76	57	57	11.76
nos6	675	1,290	15	15	0.00	**16**	6.67	16	16	6.67	16	6.67	16	16	6.67
nos7	729	1,944	38	43	13.16	**46**	21.05	46	46	21.05	46	21.05	46	46	21.05
orsirr_2	886	2,542	54	62	14.81	**69**	27.78	_	69	27.78	69	27.78	69	69	27.78
plat362	362	2,712	28	29	3.57	**30**	7.14	30	30	7.14	30	7.14	30	30	7.14
plskz362	362	880	14	15	7.14	15	7.14	15	15	7.14	15	7.14	15	15	7.14
pores_3	456	1,769	12	13	8.33	13	8.33	13	13	8.33	13	8.33	13	13	8.33
saylr1	238	445	10	12	20.00	12	20.00	12	12	20.00	12	20.00	12	12	20.00
saylr3	681	1,373	31	35	12.90	**37**	19.35	37	37	19.35	37	19.35	37	37	19.35
sherman1	681	1,373	31	35	12.90	**37**	19.35	37	37	19.35	37	19.35	37	37	19.35
sherman4	546	1,341	21	21	0.00	**22**	4.76	22	22	4.76	22	4.76	22	22	4.76
shl_____0	663	1,682	211	211	0.00	211	0.00	*211*	211	0.00	211	0.00	211	211	0.00
shl___200	663	1,720	220	220	0.00	220	0.00	*220*	220	0.00	220	0.00	220	220	0.00
shl___400	663	1,709	213	213	0.00	213	0.00	*213*	213	0.00	213	0.00	213	213	0.00
steam1	240	1,761	26	32	23.08	32	23.08	**33**	33	26.92	33	26.92	33	33	26.92
steam2	600	6,580	48	54	12.50	**55**	14.58	55	55	14.58	55	14.58	55	55	14.58
str_____0	363	2,446	70	87	24.29	**88**	25.71	88	88	25.71	88	25.71	**89**	89	27.14
str___200	363	3,049	72	90	25.00	**91**	26.39	91	91	26.39	**93**	29.17	92	93	29.17
str___600	363	3,244	79	101	27.85	*98*	24.05	**101**	101	27.85	**103**	30.38	102	103	30.38
west0381	381	2,150	100	119	19.00	*118*	18.00	**120**	120	20.00	*118*	18.00	**121**	121	21.00
west0479	479	1,889	76	84	10.53	**91**	19.74	91	91	19.74	91	19.74	91	91	19.74
west0497	497	1,715	69	69	0.00	**79**	14.49	79	79	14.49	79	14.49	79	79	14.49
west0655	655	2,841	100	109	9.00	**124**	24.00	**125**	125	25.00	124	24.00	125	125	25.00
west0989	989	3,500	121	123	1.65	**165**	36.36	_	165	36.36	165	36.36	_	165	36.36
Average					**5.75**		**9.28**			**9.49**		**9.53**			**9.68**

Table A.4: Upper bounds for large instances.

Instance			**MCP & LRX**				**Our results**									
			Best known				Hebas	GPS		iGPS		SA-σ		Hebas		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	*gap*	Φ_l	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	*gap*
494_bus	494	24	25	31	29.17	24.00	26	62	158.33	53	120.83	30	25.00	30	25.00	15.38
662_bus	662	36	36	40	11.11	11.11	37	118	227.78	70	94.44	41	13.89	41	13.89	10.81
685_bus	685	29	30	34	17.24	13.33	31	77	165.52	53	82.76	42	44.83	42	44.83	35.48
ash292	292	16	16	20	25.00	25.00	17	28	75.00	24	50.00	20	25.00	19	18.75	11.76
bcspwr04	274	23	23	25	8.70	8.70	23	38	65.22	38	65.22	24	4.35	24	4.35	4.35
bcspwr05	443	25	25	28	12.00	12.00	26	66	164.00	43	72.00	28	12.00	26	4.00	0.00
bcsstk06	420	32	38	46	43.75	21.05	39	50	56.25	49	53.13	45	40.63	45	40.63	15.38
bcsstk19	817	12	13	14	16.67	7.69	13	18	50.00	18	50.00	15	25.00	15	25.00	15.38
bcsstk20	467	7	8	13	85.71	62.50	8	20	185.71	19	171.43	19	171.43	19	171.43	137.50
bcsstm07	420	32	37	46	43.75	24.32	39	56	75.00	51	59.38	45	40.63	45	40.63	15.38
bp_____0	822	174	174	239	37.36	37.36	174	416	139.08	403	131.61	241	38.51	241	38.51	38.51
bp____200	822	186	186	268	44.09	44.09	188	446	139.78	435	133.87	272	46.24	272	46.24	44.68
bp____400	822	188	188	276	46.81	46.81	195	530	181.91	461	145.21	281	49.47	281	49.47	44.10
bp____600	822	189	190	280	48.15	47.37	197	541	186.24	463	144.97	290	53.44	290	53.44	47.21
bp____800	822	190	197	292	53.68	48.22	201	526	176.84	432	127.37	299	57.37	299	57.37	48.76
bp___1000	822	191	197	290	51.83	47.21	202	535	180.10	436	128.27	306	60.21	306	60.21	51.49
bp___1200	822	193	197	295	52.85	49.75	202	472	144.56	470	143.52	307	59.07	307	59.07	51.98
bp___1400	822	193	199	297	53.89	49.25	203	545	182.38	470	143.52	309	60.10	309	60.10	52.22
bp___1600	822	192	199	300	56.25	50.75	203	561	192.19	471	145.31	310	61.46	310	61.46	52.71
can___292	292	33	34	39	18.18	14.71	35	56	69.70	54	63.64	44	33.33	41	24.24	17.14
can___445	445	42	46	54	28.57	17.39	46	74	76.19	73	73.81	70	66.67	70	66.67	52.17
can___715	715	52	54	72	38.46	33.33	56	103	98.08	99	90.38	78	50.00	78	50.00	39.29
can___838	838	75	75	88	17.33	17.33	81	109	45.33	105	40.00	92	22.67	92	22.67	13.58
dwt___209	209	20	21	23	15.00	9.52	22	37	85.00	30	50.00	27	35.00	23	15.00	4.55
dwt___221	221	11	12	13	18.18	8.33	12	16	45.45	15	36.36	14	27.27	13	18.18	8.33
dwt___245	245	21	21	22	4.76	4.76	21	38	80.95	38	80.95	21	0.00	21	0.00	0.00

Continued on the next page

Table A.4: Upper bounds for large instances.

Instance			**MCP & LRX**				**Our results**									
			Best known				Hebas	GPS		iGPS		SA-σ		Hebas		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	*gap*	Φ_l	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	*gap*
dwt___310	310	11	11	12	9.09	9.09	11	15	36.36	14	27.27	13	18.18	12	9.09	9.09
dwt___361	361	14	14	14	0.00	0.00	14	14	0.00	14	0.00	14	0.00	14	0.00	0.00
dwt___419	419	21	23	26	23.81	13.04	23	33	57.14	32	52.38	28	33.33	28	33.33	21.74
dwt___503	503	29	29	42	44.83	44.83	34	56	93.10	52	79.31	51	75.86	51	75.86	50.00
dwt___592	592	22	22	30	36.36	36.36	22	42	90.91	42	90.91	35	59.09	35	59.09	59.09
dwt___878	878	23	23	26	13.04	13.04	23	27	17.39	27	17.39	26	13.04	26	13.04	13.04
dwt___918	918	27	27	33	22.22	22.22	29	48	77.78	48	77.78	35	29.63	35	29.63	20.69
dwt___992	992	35	35	35	0.00	0.00	35	40	14.29	40	14.29	38	8.57	35	0.00	0.00
fs_541_1	541	270	270	270	0.00	0.00	270	529	95.93	529	95.93	270	0.00	270	0.00	0.00
fs_680_1	680	17	17	17	0.00	0.00	17	20	17.65	20	17.65	17	0.00	17	0.00	0.00
fs_760_1	760	35	36	38	8.57	5.56	37	41	17.14	41	17.14	39	11.43	37	5.71	0.00
gr_30_30	900	31	31	31	0.00	0.00	31	49	58.06	49	58.06	48	54.84	31	0.00	0.00
gre___343	343	23	23	28	21.74	21.74	25	28	21.74	28	21.74	28	21.74	28	21.74	12.00
gre___512	512	30	30	36	20.00	20.00	31	36	20.00	36	20.00	36	20.00	36	20.00	16.13
gre_216a	216	17	17	21	23.53	23.53	18	21	23.53	21	23.53	21	23.53	21	23.53	16.67
hor___131	434	42	46	56	33.33	21.74	47	78	85.71	65	54.76	65	54.76	65	54.76	38.30
impcol_a	206	23	30	32	39.13	6.67	31	48	108.70	48	108.70	33	43.48	33	43.48	6.45
impcol_d	425	34	36	40	17.65	11.11	36	65	91.18	51	50.00	46	35.29	46	35.29	27.78
impcol_e	225	34	34	42	23.53	23.53	36	62	82.35	62	82.35	43	26.47	43	26.47	19.44
jagmesh1	936	22	24	27	22.73	12.50	25	27	22.73	27	22.73	27	22.73	27	22.73	8.00
jpwh_991	983	79	82	90	13.92	9.76	84	147	86.08	141	78.48	97	22.78	97	22.78	15.48
lns___511	503	30	33	45	50.00	36.36	35	57	90.00	57	90.00	47	56.67	47	56.67	34.29
mbeacxc	487	243	248	262	7.82	5.65	251	481	97.94	397	63.37	273	12.35	273	12.35	8.76
mbeaflw	487	243	246	262	7.82	6.50	251	481	97.94	397	63.37	273	12.35	273	12.35	8.76
mbeause	492	245	249	254	3.67	2.01	252	472	92.65	472	92.65	256	4.49	256	4.49	1.59
mcfe	731	112	112	127	13.39	13.39	121	181	61.61	178	58.93	135	20.54	135	20.54	11.57
nnc261	261	20	22	24	20.00	9.09	23	38	90.00	38	90.00	24	20.00	24	20.00	4.35

Continued on the next page

Table A.4: Upper bounds for large instances.

Instance			**MCP & LRX**				**Our results**									
			Best known				Hebas	GPS		iGPS		SA-σ		Hebas		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	*gap*	Φ_l	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	Φ_u	*dev*	*gap*
nnc666	666	32	33	41	28.13	24.24	35	64	100.00	64	100.00	44	37.50	44	37.50	25.71
nos2	638	3	3	3	0.00	0.00	3	3	0.00	3	0.00	3	0.00	3	0.00	0.00
nos3	960	43	43	43	0.00	0.00	43	71	65.12	52	20.93	48	11.63	43	0.00	0.00
nos5	468	51	53	64	25.49	20.75	57	70	37.25	70	37.25	67	31.37	67	31.37	17.54
nos6	675	15	15	16	6.67	6.67	16	16	6.67	16	6.67	16	6.67	16	6.67	0.00
nos7	729	38	43	65	71.05	51.16	46	65	71.05	65	71.05	65	71.05	65	71.05	41.30
orsirr_2	886	54	62	87	61.11	40.32	69	113	109.26	111	105.56	93	72.22	93	72.22	34.78
plat362	362	28	29	34	21.43	17.24	30	39	39.29	38	35.71	37	32.14	37	32.14	23.33
plskz362	362	14	15	18	28.57	20.00	15	24	71.43	23	64.29	21	50.00	21	50.00	40.00
pores_3	456	12	13	13	8.33	0.00	13	14	16.67	14	16.67	13	8.33	13	8.33	0.00
saylr1	238	10	12	14	40.00	16.67	12	14	40.00	14	40.00	14	40.00	14	40.00	16.67
saylr3	681	31	35	47	51.61	34.29	37	57	83.87	54	74.19	53	70.97	53	70.97	43.24
sherman1	681	31	35	47	51.61	34.29	37	57	83.87	54	74.19	53	70.97	53	70.97	43.24
sherman4	546	21	21	27	28.57	28.57	22	27	28.57	27	28.57	27	28.57	27	28.57	22.73
shl_____0	663	211	211	232	9.95	9.95	211	412	95.26	412	95.26	228	8.06	228	8.06	8.06
shl___200	663	220	220	238	8.18	8.18	220	426	93.64	426	93.64	236	7.27	236	7.27	7.27
shl___400	663	213	213	235	10.33	10.33	213	425	99.53	425	99.53	235	10.33	235	10.33	10.33
steam1	240	26	32	44	69.23	37.50	33	50	92.31	50	92.31	47	80.77	47	80.77	42.42
steam2	600	48	54	63	31.25	16.67	55	67	39.58	67	39.58	67	39.58	67	39.58	21.82
str_____0	363	70	87	118	68.57	35.63	89	182	160.00	176	151.43	120	71.43	120	71.43	34.83
str___200	363	72	90	126	75.00	40.00	93	214	197.22	185	156.94	130	80.56	130	80.56	39.78
str___600	363	79	101	132	67.09	30.69	103	250	216.46	202	155.70	137	73.42	137	73.42	33.01
west0381	381	100	119	153	53.00	28.57	121	252	152.00	231	131.00	156	56.00	156	56.00	28.93
west0479	479	76	84	122	60.53	45.24	91	210	176.32	194	155.26	122	60.53	122	60.53	34.07
west0497	497	69	69	86	24.64	24.64	79	142	105.80	142	105.80	87	26.09	87	26.09	10.13
west0655	655	100	109	161	61.00	47.71	125	253	153.00	253	153.00	159	59.00	159	59.00	27.20
west0989	989	121	123	210	73.55	70.73	165	328	171.07	325	168.60	210	73.55	210	73.55	27.27
Average					**29.87**	**22.55**			**92.50**		**78.00**		**37.53**		**35.76**	**23.31**

Table A.5: Lower bounds for very large instances.

Instance			**Without zero-edges**				**With zero-edges**			
			Graph		**Hebas**		**Graph**		**Hebas**	
Name	n	*entries*	m	Φ_t	Φ_l	*imp*	m	Φ_t	Φ_l	*imp*
1138_bus	1,138	2.60×10^3	1,458	44	**47**	6.82	1,458	44	**47**	6.82
3dtube	45,330	1.63×10^6	1,584,144	1,398	*1,398*	0.00	1,584,144	1,398	*1,398*	0.00
add20	2,395	1.73×10^4	*5,378*	323	**328**	1.55	*7,462*	369	**396**	7.32
add32	4,960	2.39×10^4	*7,444*	19	19	0.00	*9,462*	276	*276*	0.00
barth	6,691	2.64×10^4	19,748	101	**103**	1.98	19,748	101	**103**	1.98
barth4	6,019	2.35×10^4	17,473	94	**100**	6.38	17,473	94	**100**	6.38
barth5	15,606	6.15×10^4	45,878	163	**180**	10.43	45,878	163	**180**	10.43
bcspwr08	1,624	3.84×10^3	2,213	53	**60**	13.21	2,213	53	**60**	13.21
bcspwr09	1,723	4.12×10^3	2,394	55	**61**	10.91	2,394	55	**61**	10.91
bcspwr10	5,300	1.36×10^4	8,271	132	**135**	2.27	8,271	132	**135**	2.27
bcsstk13	2,003	4.29×10^4	40,940	207	**237**	14.49	40,940	207	**237**	14.49
bcsstk24	3,562	8.17×10^4	78,174	151	**158**	4.64	78,174	151	**158**	4.64
bcsstk30	28,924	1.04×10^6	1,007,284	1,038	*1,038*	0.00	1,007,284	1,038	*1,038*	0.00
bcsstk32	44,609	1.03×10^6	985,046	968	*968*	0.00	985,046	968	*968*	0.00
bcsstk33	8,738	3.00×10^5	291,583	458	458	0.00	291,583	458	458	0.00
bcsstk35	30,237	7.40×10^5	709,963	677	*677*	0.00	709,963	677	*677*	0.00
bcsstk36	23,052	5.83×10^5	560,044	606	*606*	0.00	560,044	606	*606*	0.00
bcsstk37	25,503	5.83×10^5	557,737	586	*586*	0.00	557,737	586	*586*	0.00
bcsstk38	8,032	1.82×10^5	173,714	328	**334**	1.83	173,714	328	**334**	1.83
can_1054	1,054	6.63×10^3	5,571	77	**79**	2.60	5,571	77	**79**	2.60
can_1072	1,072	6.76×10^3	5,686	99	**106**	7.07	5,686	99	**106**	7.07
ct20stif	52,329	1.38×10^6	*1,273,983*	1,436	*1,436*	0.00	*1,323,067*	1,436	*1,436*	0.00
dwt_1007	1,007	4.79×10^3	3,784	23	**24**	4.35	3,784	23	**24**	4.35
dwt_2680	2,680	1.39×10^4	11,173	44	**46**	4.55	11,173	44	**46**	4.55
finan512	74,752	3.36×10^5	261,120	925	*925*	0.00	261,120	925	*925*	0.00
gupta3	16,783	4.67×10^6	4,653,322	7,336	*7,336*	0.00	4,653,322	7,336	*7,336*	0.00

Continued on the next page

Table A.5: Lower bounds for very large instances.

Instance			**Without zero-edges**				**With zero-edges**			
			Graph		**Hebas**		**Graph**		**Hebas**	
Name	n	*entries*	m	Φ_t	Φ_l	*imp*	m	Φ_t	Φ_l	*imp*
jagmesh9	1,349	5.23×10^3	3,876	26	**28**	7.69	3,876	26	**28**	7.69
memplus	17,758	1.26×10^5	*41,534*	2,865	*2,865*	0.00	*54,196*	2,953	*2,953*	0.00
msc10848	10,848	6.20×10^5	*609,464*	790	**792**	0.25	*609,465*	790	**792**	0.25
msc23052	23,052	5.89×10^5	*559,817*	606	*606*	0.00	*565,881*	606	*606*	0.00
nasa1824	1,824	2.05×10^4	18,692	118	**129**	9.32	18,692	118	**129**	9.32
nasa4704	4,704	5.47×10^4	50,026	171	**183**	7.02	50,026	171	**183**	7.02
pwtk	217,918	5.93×10^6	*5,653,257*	814	*814*	0.00	*5,708,253*	814	*814*	0.00
shuttle_eddy	10,429	5.70×10^4	46,585	79	**104**	31.65	46,585	79	**104**	31.65
twotone	120,750	1.22×10^6	*1,012,970*	7,727	*7,727*	0.00	*1,029,429*	7,732	*7,732*	0.00
vibrobox	12,328	1.78×10^5	*144,686*	1,360	**1,476**	8.53	*165,250*	1,497	**1,609**	7.48
Average						**4.38**				**4.51**

Table A.6: Upper bounds for very large instances.

Instance			**Hebas**	**SRB-M_5**		**SRB-M_{200}**			**GPS**			**iGPS**			**SA-σ**		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	Φ_u	*dev*	*gap*	Φ_u	*time*	*dev*	Φ_u	*time*	*dev*	Φ_u	*dev*	*gap*
1138_bus	1,138	44	47	65	47.73	55	25.00	17.02	122	1	177.27	122	1	177.27	80	81.82	70.21
3dtube	45,330	1,398	1,398	2,078	48.64	1,891	35.26	35.26	2,372	20	69.67	2,357	83	68.60	2,357	68.60	68.60
685_bus	685	29	31	40	37.93	35	20.69	12.90	72	1	148.28	53	1	82.76	43	48.28	38.71
add20	2,395	323	328	427	32.20	356	10.22	8.54	889	1	175.23	751	1	132.51	526	62.85	60.37
add32	4,960	19	19	21	10.53	21	10.53	10.53	37	1	94.74	35	1	84.21	32	68.42	68.42
barth	6,691	101	103	152	50.50	128	26.73	24.27	170	1	68.32	167	1	65.35	165	63.37	60.19
barth4	6,019	94	100	128	36.17	118	25.53	18.00	238	1	153.19	171	1	81.91	166	76.60	66.00
barth5	15,606	163	180	241	47.85	211	29.45	17.22	296	1	81.60	296	1	81.60	293	79.75	62.78
bcspwr08	1,624	53	60	83	56.60	70	32.08	16.67	108	1	103.77	108	1	103.77	101	90.57	68.33
bcspwr09	1,723	55	61	84	52.73	71	29.09	16.39	118	1	114.55	118	1	114.55	85	54.55	39.34
bcspwr10	5,300	132	135	196	48.48	150	13.64	11.11	260	1	96.97	260	1	96.97	222	68.18	64.44
bcsstk12	1,473	51	51	67	31.37	63	23.53	23.53	76	1	49.02	61	1	19.61	61	19.61	19.61
bcsstk13	2,003	207	237	377	82.13	328	58.45	38.40	427	1	106.28	368	1	77.78	364	75.85	53.59
bcsstk24	3,562	151	158	180	19.21	180	19.21	13.92	227	2	50.33	227	1	50.33	226	49.67	43.04
bcsstk29	13,992	451	451	570	26.39	528	17.07	17.07	874	1	93.79	737	3	63.41	729	61.64	61.64
bcsstk30	28,924	1,038	1,038	1,256	21.00	1,081	4.14	4.14	2,509	21	141.71	2,509	34	141.71	2,492	140.08	140.08
bcsstk31	35,558	670	670	1,259	87.91	862	28.66	28.66	1,089	1	62.54	1,089	8	62.54	1,083	61.64	61.64
bcsstk32	44,609	968	968	2,269	134.40	1,661	71.59	71.59	2,600	3	168.60	2,313	13	138.95	2,303	137.91	137.91
bcsstk33	8,738	458	458	582	27.07	514	12.23	12.23	514	1	12.23	514	5	12.23	514	12.23	12.23
bcsstk35	30,237	677	677	1,218	79.91	971	43.43	43.43	1,764	8	160.56	1,444	13	113.29	1,432	111.52	111.52
bcsstk36	23,052	606	606	1,030	69.97	839	38.45	38.45	1,254	1	106.93	1,240	3	104.62	1,240	104.62	104.62
bcsstk37	25,503	586	586	1,030	75.77	811	38.40	38.40	1,364	2	132.76	1,362	9	132.42	1,353	130.89	130.89
bcsstk38	8,032	328	334	429	30.79	368	12.20	10.18	540	1	64.63	540	1	64.63	532	62.20	59.28
bcsstm13	2,003	93	93	107	15.05	103	10.75	10.75	141	1	51.61	138	1	48.39	130	39.78	39.78
blckhole	2,132	93	93	121	30.11	101	8.60	8.60	102	2	9.68	102	1	9.68	102	9.68	9.68
can_1054	1,054	77	79	90	16.88	82	6.49	3.80	112	1	45.45	109	1	41.56	93	20.78	17.72
can_1072	1,072	99	106	129	30.30	118	19.19	11.32	178	1	79.80	156	1	57.58	134	35.35	26.42

Continued on the next page

Table A.6: Upper bounds for very large instances.

Instance				**Hebas**	**SRB-M_5**		**SRB-M_{200}**		**GPS**			**iGPS**			**SA-σ**		
Name	n	Φ_t	Φ_l	Φ_u	*dev*	Φ_u	*dev*	*gap*	Φ_u	*time*	*dev*	Φ_u	*time*	*dev*	Φ_u	*dev*	*gap*
can_445	445	42	46	60	42.86	56	33.33	21.74	74	1	76.19	74	1	76.19	72	71.43	56.52
can_838	838	75	81	98	30.67	90	20.00	11.11	141	1	88.00	105	1	40.00	98	30.67	20.99
ct20stif	52,329	1,436	1,436	4,144	188.58	2,550	77.58	77.58	3,108	1	116.43	3,108	9	116.43	3,095	115.53	115.53
dwt_1007	1,007	23	24	29	26.09	29	26.09	20.83	33	1	43.48	33	1	43.48	33	43.48	37.50
dwt_2680	2,680	44	46	64	45.45	56	27.27	21.74	69	1	56.82	69	1	56.82	69	56.82	50.00
dwt_918	918	27	29	36	33.33	34	25.93	17.24	49	1	81.48	49	1	81.48	40	48.15	37.93
ex27	974	113	115	123	8.85	122	7.96	6.09	181	1	60.18	179	1	58.41	151	33.63	31.30
finan512	74,752	925	925	1,212	31.03	1,119	20.97	20.97	1,209	1	30.70	1,209	6	30.70	1,208	30.59	30.59
gearbox	153,746	1,866	1,866	4,265	128.56	4,077	118.49	118.49	4,209	2	125.56	4,050	52	117.04	4,045	116.77	116.77
gupta3	16,783	7,336	7,336	8,775	19.62	8,274	12.79	12.79	9,329	57	27.17	9,013	1,996	22.86	8,794	19.87	19.87
jagmesh1	936	22	25	33	50.00	30	36.36	20.00	27	1	22.73	27	1	22.73	27	22.73	8.00
jagmesh9	1,349	26	28	40	53.85	40	53.85	42.86	40	1	53.85	40	1	53.85	40	53.85	42.86
memplus	17,758	2,865	2,865	4,630	61.61	3,214	12.18	12.18	11,033	2	285.10	11,003	12	284.05	10,977	283.14	283.14
msc10848	10,848	790	792	1,053	33.29	864	9.37	9.09	1,235	1	56.33	1,235	8	56.33	1,206	52.66	52.27
msc23052	23,052	606	606	1,067	76.07	854	40.92	40.92	1,316	4	117.16	1,219	11	101.16	1,210	99.67	99.67
nasa1824	1,824	118	129	164	38.98	150	27.12	16.28	263	1	122.88	232	1	96.61	220	86.44	70.54
nasa4704	4,704	171	183	234	36.84	209	22.22	14.21	345	1	101.75	336	1	96.49	336	96.49	83.61
pwt	36,519	137	137	312	127.74	258	88.32	88.32	341	1	148.91	340	2	148.18	338	146.72	146.72
pwtk	217,918	814	814	1,950	139.56	1,687	107.25	107.25	2,035	10	150.00	2,035	82	150.00	2,035	150.00	150.00
shuttle_eddy	10,429	79	104	128	62.03	119	50.63	14.42	176	1	122.78	173	1	118.99	173	118.99	66.35
skirt	12,598	112	112	186	66.07	155	38.39	38.39	314	1	180.36	314	2	180.36	314	180.36	180.36
sstmodel	3,345	53	53	79	49.06	69	30.19	30.19	87	1	64.15	85	1	60.38	84	58.49	58.49
twotone	120,750	7,727	7,727	15,045	94.71	13,091	69.42	69.42	21,847	49	182.74	21,847	342	182.74	21,719	181.08	181.08
vibrobox	12,328	1,360	1,476	2,377	74.78	1,823	34.04	23.51	4,053	1	198.01	3,712	1	172.94	3,599	164.63	143.83
Average					**54.26**		**32.57**	**27.80**		**4.31**	**100.63**		**53.41**	**90.13**		**80.36**	**74.14**

Table A.7: Solver performance evaluation: Numbers of subproblems.

Instance	n	m	BW(φ)	**Hebas 0.9**	**Hebas 1.0**	**Hebas 1.1**	**Hebas 1.2**	**Hebas 1.3**	**Hebas 1.3.1**	**Hebas 1.4**	**Hebas 2.3**	**Hebas 2.4**
bcsstk22	110	254	9	_	7.88×10^{8}	1.17×10^{8}	9.34×10^{6}	9.34×10^{6}	5.36×10^{6}	7.91×10^{6}	5.17×10^{5}	4.97×10^{5}
fs_183_1	183	701	52	_	6.42×10^{8}	_	_	1.84×10^{2}	1.84×10^{2}	1.84×10^{2}	1.67×10^{4}	1.67×10^{4}
			53	_	_	_	_	2.94×10^{2}	2.94×10^{2}	2.94×10^{2}	1.67×10^{4}	1.67×10^{4}
			54	_	_	_	_	3.14×10^{2}	3.14×10^{2}	3.14×10^{2}	1.93×10^{4}	1.93×10^{4}
			55	_	_	_	_	1.21×10^{3}	1.21×10^{3}	1.21×10^{3}	1.94×10^{4}	1.94×10^{4}
			56	_	_	_	_	3.74×10^{5}	3.74×10^{5}	3.74×10^{5}	2.70×10^{6}	2.70×10^{6}
gent113	104	549	24	5.59×10^{4}	5.59×10^{4}	4.81×10^{4}	4.02×10^{4}	4.02×10^{4}	8.19×10^{3}	4.02×10^{4}	3.03×10^{4}	3.03×10^{4}
			25	4.66×10^{6}	4.66×10^{6}	4.36×10^{6}	3.75×10^{6}	3.75×10^{6}	3.70×10^{6}	3.75×10^{6}	3.80×10^{5}	3.80×10^{5}
gre___115	115	267	19	1.97×10^{6}	1.97×10^{6}	2.74×10^{5}	3.00×10^{4}	3.00×10^{4}	3.12×10^{4}	3.00×10^{4}	1.83×10^{4}	1.83×10^{4}
			20	_	_	_	_	_	_	_	1.17×10^{9}	_
gre___185	185	650	16	3.10×10^{3}	3.10×10^{3}	2.46×10^{3}	1.64×10^{3}	1.86×10^{2}	1.86×10^{2}	1.86×10^{2}	1.70×10^{4}	1.70×10^{4}
			17	_	_	_	_	1.86×10^{2}	1.86×10^{2}	1.86×10^{2}	1.70×10^{4}	1.70×10^{4}
impcol_b	59	281	19	4.82×10^{8}	4.82×10^{8}	1.03×10^{9}	6.76×10^{8}	6.76×10^{8}	_	6.76×10^{8}	1.29×10^{7}	1.29×10^{7}
impcol_c	137	352	26	_	_	_	_	8.64×10^{5}	8.64×10^{5}	8.64×10^{5}	6.08×10^{4}	6.07×10^{4}
lns___131	123	275	17	6.60×10^{5}	6.60×10^{5}	6.60×10^{5}	3.67×10^{4}	4.37×10^{4}	4.37×10^{4}	3.93×10^{4}	2.83×10^{4}	2.83×10^{4}
			18	2.26×10^{8}	2.26×10^{8}	2.25×10^{8}	2.25×10^{8}	2.25×10^{8}	_	2.06×10^{8}	3.44×10^{7}	3.37×10^{7}
lund_a	147	1151	18	1.36×10^{5}	1.36×10^{5}	1.10×10^{5}	8.97×10^{4}	1.48×10^{2}	1.48×10^{2}	1.48×10^{2}	1.07×10^{4}	1.07×10^{4}
lund_b	147	1147	18	1.36×10^{5}	1.36×10^{5}	1.10×10^{5}	8.97×10^{4}	1.48×10^{2}	1.48×10^{2}	1.48×10^{2}	1.07×10^{4}	1.07×10^{4}
west0132	132	404	24	2.93×10^{4}	2.93×10^{4}	1.50×10^{3}	6.32×10^{2}	1.33×10^{2}	1.33×10^{2}	1.33×10^{2}	8.65×10^{3}	8.65×10^{3}
			25	_	_	_	_	1.96×10^{2}	1.96×10^{2}	1.96×10^{2}	8.65×10^{3}	8.65×10^{3}
			26	_	_	_	_	7.96×10^{2}	7.96×10^{2}	7.96×10^{2}	1.05×10^{4}	1.05×10^{4}
			27	_	_	_	_	5.05×10^{3}	5.05×10^{3}	5.05×10^{3}	8.35×10^{4}	8.35×10^{4}
west0156	156	371	31	2.83×10^{6}	2.83×10^{6}	2.04×10^{6}	1.35×10^{6}	1.35×10^{6}	6.41×10^{5}	1.35×10^{6}	1.14×10^{6}	1.14×10^{6}
			32	3.12×10^{8}	3.12×10^{8}	1.26×10^{8}	4.63×10^{7}	4.63×10^{7}	1.93×10^{7}	4.63×10^{7}	1.60×10^{7}	1.60×10^{7}
			33	_	_	_	_	_	_	_	5.61×10^{8}	_
west0167	167	489	30	2.69×10^{6}	2.69×10^{6}	8.58×10^{2}	3.94×10^{2}	1.68×10^{2}	1.68×10^{2}	1.68×10^{2}	1.39×10^{4}	1.39×10^{4}
			31	_	_	_	_	_	_	_	3.19×10^{7}	3.19×10^{7}
will199	199	660	53	1.22×10^{8}	1.22×10^{8}	1.05×10^{8}	7.79×10^{7}	7.79×10^{7}	7.61×10^{7}	_	1.18×10^{8}	1.18×10^{8}
			54	_	_	_	2.36×10^{8}	2.36×10^{8}	_	_	2.61×10^{8}	2.61×10^{8}

Table A.8: Solver performance evaluation: Running times (in seconds).

Instance	n	m	BW(φ)	**Hebas 0.9**	**Hebas 1.0**	**Hebas 1.1**	**Hebas 1.2**	**Hebas 1.3**	**Hebas 1.3.1**	**Hebas 1.4**	**Hebas 2.3**	**Hebas 2.4**
bcsstk22	110	254	9	_	37,020.0	5,751.0	598.0	604.0	1,244.0	496.0	24.0	23.0
fs_183_1	183	701	52	_	32,779.0	_	_	0.1	0.1	0.1	0.6	0.6
			53	_	_	_	_	0.1	0.1	0.1	0.6	0.6
			54	_	_	_	_	0.1	0.1	0.1	2.3	2.3
			55	_	_	_	_	1.0	0.9	0.9	3.1	3.1
			56	_	_	_	_	451.0	441.0	451.0	2,836.0	3,054.0
gent113	104	549	24	1.6	1.8	1.6	1.4	1.5	1.1	1.4	0.8	0.8
			25	130.0	133.0	127.0	122.0	123.0	295.0	117.0	11.0	11.0
gre___115	115	267	19	55.0	55.7	8.0	1.4	1.5	4.2	1.4	0.6	0.6
			20	_	_	_	_	_	_	_	30,942.0	_
gre___185	185	650	16	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.6	0.6
			17	_	_	_	_	0.1	0.1	0.1	0.6	0.6
impcol_b	59	281	19	7,656.0	7,848.0	18,819.0	15,864.0	15,985.0	_	15,831.5	172.0	175.0
impcol_c	137	352	26	_	_	_	_	198.0	197.0	198.0	7.6	7.5
lns___131	123	275	17	23.8	23.5	23.5	2.0	6.8	6.7	5.9	1.6	1.6
			18	6,195.0	6,214.0	6,271.0	6,731.0	7,017.0	_	5,927.8	801.0	787.0
lund_a	147	1151	18	13.7	16.0	12.8	10.8	0.1	0.1	0.1	0.3	0.3
lund_b	147	1147	18	13.7	16.0	12.9	10.8	0.1	0.1	0.1	0.3	0.3
west0132	132	404	24	1.0	1.0	0.1	0.1	0.1	0.1	0.1	0.3	0.3
			25	_	_	_	_	0.1	0.1	0.1	0.3	0.3
			26	_	_	_	_	0.2	0.2	0.2	1.0	1.0
			27	_	_	_	_	0.6	0.6	0.6	3.6	3.6
west0156	156	371	31	104.0	105.0	77.0	55.0	55.0	67.0	92.0	38.7	40.0
			32	12,433.0	12,589.0	5,318.0	2,148.0	2,139.0	2,731.0	7,485.0	565.0	600.0
			33	_	_	_	_	_	_	_	22,199.0	_
west0167	167	489	30	115.0	115.0	0.1	0.1	0.1	0.1	0.1	0.4	0.7
			31	_	_	_	_	_	_	_	7,558.0	13,079.0
will199	199	660	53	5,383.0	5,442.0	4,621.0	3,733.0	3,808.0	3,984.0	_	4,790.0	31,436.0
			54	_	_	_	11,582.0	11,865.0	_	_	10,707.0	74,412.0

Symbols and Notation

G	An undirected graph $G = (V, E)$.
V	Set of vertices.
E	Set of edges.
n	Number of vertices in G.
m	Number of edges in G.
$d(u, v)$	Distance between u and v.
$N_1(v)$	Set of vertices adjacent to v (not including v).
$N_h(v)$	Set of vertices at distance at most h from v (not including v).
π	A permutation of the vertices.
π^L	A left partial permutation.
$\pi^{L,R}$	A both-sided partial permutation.
$\pi(v)$	Label of vertex v under π.
f_v	Min label of v in a partial permutation.
l_v	Max label of v in a partial permutation.
φ	Search parameter in a BANDWIDTH (BW) problem.
$\alpha(G)$	Lower bound Alpha. Refer to formula (3.1).
$\gamma(G)$	Lower bound Gamma. Refer to formula (3.2).
Φ_t	Initial lower bound. $\Phi_t = \max\{\alpha(G), \gamma(G)\}$.
Φ_l	Lower bound obtained by a solution method.
Φ_u	Upper bound obtained by a solution method.
imp	Lower bound improvement (in %). Refer to formula (7.1).
dev	Upper bound deviation (in %). Refer to formula (7.2).
gap	Gap (in %) between lower and upper bound. Refer to formula (7.3).
S_p	Speedup factor of a parallel system. Refer to formula (2.1).
E_p	Efficiency factor of a parallel system. Refer to formula (2.2).

References

[1] Blum, A., Konjevod, G., Ravi, R., and Vempala, S. (2000). Semidefinite relaxations for minimum bandwidth and other vertex-ordering problems. *Theoretical Computer Science*, *235*(1), 25–42.

[2] Boisvert, R. F., Pozo, R., Remington, K., Barrett, R. F., and Dongarra, J. J. (1997). Matrix market: A web resource for test matrix collections. In R. F. Boisvert (Ed.), *The Quality of Numerical Software: Assessment and Enhancement* (pp. 125–137). London: Chapman and Hall.

[3] Computational Infrastructure for Operations Research (COIN-OR). Project Home Page: http://www.coin-or.org.

[4] Campos, V., Piñana, E., and Martí, R. (2011). Adaptive memory programming for matrix bandwidth minimization. *Annals of Operations Research*, *183*(1), 7–23.

[5] Campos, V., Piñana, E., and Martí, R. (no date). Bandwidth minimization problem. Computational Results. Retrieved December 7, 2010 from http://www.uv.es/rmarti/paper/results/Best BRP Results.pdf.

[6] Caprara, A. and Salazar-González, J.-J. (2005). Laying out sparse graphs with provably minimum bandwidth. *INFORMS Journal on Computing*, *17*(3), 356–373.

[7] Chinn, P. Z., Chvátalová, J., Dewdney, A., and Gibbs, N. (1982). The bandwidth problem for graphs and matrices - a survey. *Journal of Graph Theory*, *6*(3), 223–254.

[8] Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 151–158. New York: ACM.

[9] Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the 1969 24th National Conference*, 157–172. New York: ACM.

[10] Dantzig, G. B. (1951). Maximization of a linear function of variables subject to linear inequalities. In T. C. Koopmans (Ed.), *Activity Analysis of Production and Allocation* (pp. 339–347). New York: John Wiley & Sons.

[11] Davis, T. A. and Hu, Y. (in press). The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*. Collection Home Page: http://www.cise.ufl.edu/research/sparse/matrices.

[12] Del Corso, G. M. and Manzini, G. (1999). Finding exact solutions to the bandwidth minimization problem. *Computing*, *62*(3), 189–203.

[13] Díaz, J., Petit, J., and Serna, M. (2002). A survey of graph layout problems. *ACM Computing Surveys*, *34*(3), 313–356.

[14] Diestel, R. (2005). *Graph Theory* (3rd ed.). Springer-Verlag Berlin Heidelberg.

[15] Eckstein, J., Phillips, C. A., and Hart, W. E. (2000, August). *Pico: An object-oriented framework for parallel branch and bound*. RUTCOR Research Report RRR 40-2000, Rutgers University. http://rutcor.rutgers.edu/pub/rrr/reports2000/40.ps.

[16] Eckstein, J., Phillips, C. A., and Hart, W. E. (2006, August). *PEBBL 1.0 User's Guide*. RUTCOR Research Report RRR 19-2006, Rutgers University. http://rutcor.rutgers.edu/pub/rrr/reports2006/19_2006.ps.

[17] Esposito, A., Malucelli, F., and Tarricone, L. (1998). Bandwidth and profile reduction of sparse matrices: an experimental comparison of new heuristics. In R. Battiti and A. A. Bertossi (Eds.), *Proceedings of "Algorithms and Experiments" (ALEX98)*, Trento, Italy, 19–26.

[18] Feige, U. (2000). Approximating the bandwidth via volume respecting embeddings. *Journal of Computer and System Sciences*, *60*(3), 510–539.

[19] Foster, I. (1995). *Designing and Building Parallel Programs*. Reading, MA: Addison-Wesley.

[20] Garey, M. R., Graham, R. L., Johnson, D. S., and Knuth, D. E. (1978). Complexity results for bandwidth minimization. *SIAM J. on Applied Mathematics*, *34*(3), 477–495.

[21] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman.

[22] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press.

[23] Gibbs, N. E., Poole, W. G., and Stockmeyer, P. K. (1976). An algorithm for reducing the bandwidth and profile of sparse matrix. *SIAM Journal on Numerical Analysis*, *13*(2), 236–250.

[24] Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message Passing Interface* (2nd ed.). Cambridge, MA: MIT Press.

[25] Gurari, E. M. and Sudborough, I. H. (1984). Improved dynamic programming algorithms for bandwidth minimization and the MinCut Linear Arrangement problem. *Journal of Algorithms*, *5*(4), 531–546.

[26] Heidelberg Linux Cluster System (HELICS). Home page: http://helics.uni-hd.de.

[27] Hwang, K. and Xu, Z. (1998). *Scalable Parallel Computing: Technology, Architecture, Programming*. Boston, MS: WCB/McGraw-Hill.

[28] Jünger, M., Reinelt, G., and Thienel, S. (1995). Practical problem solving with cutting plane algorithms in combinatorial optimization. In W. Cook, L. Lovász, and P. Seymour (Eds.), *Combinatorial Optimization: Papers from the DIMACS Special Year* (Volume 20 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science) (pp. 111–152). Providence, RI: American Mathematical Society.

[29] Kliewer, G. and Tschöke, S. (2000). A general parallel simulated annealing library and its application in airline industry. *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, 55–61.

[30] Kliewer, G. and Klohs, K. (no date). *Parallel Simulated Annealing Library (parSA): User Manual Version 2.2*. Library Home Page: http://www2.cs.uni-paderborn.de/fachbereich/AG/monien/SOFTWARE/PARSA.

[31] Korte, B. and Vygen, J. (2008). *Combinatorial Optimization: Theory and Algorithms* (4th ed.). Springer-Verlag Berlin Heidelberg.

[32] Lai, Y.-L. and Williams, K. (1999). A survey of solved problems and applications on bandwidth, edgesum, and profile of graphs. *Journal of Graph Theory*, *31*(2), 75–94.

[33] Land, A. H. and Doig, A. G. (1960). An automatic method for solving discrete programming problems. *Econometrica*, *28*(3), 497–520.

[34] Lim, A., Rodrigues, B., and Xiao, F. (2006). Heuristics for matrix bandwidth reduction. *European Journal of Operational Research*, *174*(1), 69–91.

[35] Linderoth, J. T. and Savelsbergh, M. W. P. (1999). Computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, *11*(2), 173–187.

[36] Martí, R., Campos, V., and Piñana, E. (2008). A branch and bound for the matrix bandwidth minimization. *European Journal of Operational Research*, *186*(2), 513–528.

[37] Martí, R., Laguna, M., Glover, F., and Campos, V. (2001). Reducing the bandwidth of a sparse matrix with tabu search. *European Journal of Operational Research*, *135*(2), 450–459.

[38] MPI. Home Page: http://www.mcs.anl.gov/research/projects/mpi.

[39] MPICH2. Home Page: http://www-unix.mcs.anl.gov/mpi/mpich.

[40] Mueller, C. (2004). Sparse matrix reordering algorithms for cluster identification. Research Report, Indiana University. http://osl.iu.edu/~chemuell/projects/bioinf/sparse-matrix-clustering-chris-mueller.pdf.

[41] Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, *27*(8), 983 – 993.

[42] Papadimitriou, C. H. (1976). The NP-completeness of the bandwidth minimization problem. *Computing*, *16*(3), 263–270.

[43] Piñana, E., Plana, I., Campos, V., and Martí, R. (2004). GRASP and Path relinking for the matrix bandwidth minimization. *European Journal of Operational Research*, *153*(1), 200–210.

[44] Ralphs, T. K. (2003). Parallel branch and cut for capacitated vehicle routing. *Parallel Computing*, *29*(5), 607–629.

[45] Ralphs, T. K., Güzelsoy, M., and Mahajan, A. (2010). *SYMPHONY 5.2.3 User's Manual*. Project Home Page: https://projects.coin-or.org/SYMPHONY.

[46] Rockafellar, R. T. and Wets, R. J.-B. (2004). *Variational Analysis* (2nd ed.). Springer-Verlag Berlin Heidelberg.

[47] Rodriguez-Tello, E., Hao, J. K., and Torres-Jimenez, J. (2004). An improved evaluation function for the bandwidth minimization problem. *Lecture Notes in Computer Science*, *3242/2004*, 652–661.

[48] Rodriguez-Tello, E., Hao, J. K., and Torres-Jimenez, J. (2008). An improved simulated annealing algorithm for bandwidth minimization. *European Journal of Operational Research*, *185*(3), 1319–1335.

[49] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

[50] Safro, I., Ron, D., and Brandt, A. (2009). Multilevel algorithms for linear ordering problems. *ACM Journal of Experimental Algorithmics*, *13*, 1.4:1–1.4:20.

[51] Saxe, J. B. (1980). Dynamic-programming algorithms for recognizing small bandwidth graphs in polynomial time. *SIAM. J. on Algebraic and Discrete Methods*, *1*(4), 363–369.

[52] Schnörr, C. (2010, winter semester 2009-2010). *Digital Image Processing*. Class Lecture. University of Heidelberg.

[53] SGI. *Standard Template Library*. Home Page: http://www.sgi.com/tech/stl.

[54] Suh, J., Jung, S., Pfeifle, M., Vo, K. T., Oswald, M., and Reinelt, G. (2007). Compression of digital road networks. *Lecture Notes in Computer Science*, *4605/2007*, 423–440.

[55] Vo, K. T. and Reinelt, G. (2009). Parallel computation for the bandwidth minimization problem. In B. Fleischmann, K.-H. Borgwardt, R. Klein, and A. Tuma (Eds.), *Operations Research Proceedings 2008*, 481–486. Springer-Verlag Berlin Heidelberg.

[56] Xu, Y., Ralphs, T. K., Ladányi, L., and Saltzman, M. (2005). ALPS: A framework for implementing parallel search algorithms. *The Proceedings of the Ninth INFORMS Computing Society Conference*, 319–334. Project Home Page: https://projects.coin-or.org/CHiPPS.

[57] Xu, Y. (2007). *Scalable Algorithms for Parallel Tree Search*. Doctoral dissertation, Lehigh University. http://coral.ie.lehigh.edu/~ted/files/papers/YanXuDissertation07.pdf.

Index

$N_1(F)$ 5
$N_1(v)$ 5
$N_k(v)$ 5
$O(f(n))$ 6
Φ_l 79
Φ_t 79
Φ_u 79
$\alpha(G)$ 14, 42, 71, 79
$\gamma(G)$ 14, 42, 71, 79
π^L 19
$\pi^{L,R}$ 19
dev 79
gap 79, 103
imp 79
$d(v, w)$ 5
f_v 22, 29, 46–48, 53
l_v 21, 29, 45–48, 53
2-labeling 63, 80, 86, 108, 109
  algorithm 64
  drawback 63

adjacency matrix 5
adjacent vertices 5
applications 1, 2, 40, 108, 110
approximate objective function 35, 37, 108
assigned set 19

BANDWIDTH (BW) 13, 18
bandwidth constraint 14, 18, 19, 29, 55, 57, 58, 93, 109
bandwidth minimization problem 1, 13, 108
  approximate objective function . . . 35
  optimal solution 18, 71, 72
  original objective function . . *see* IP formulation, of bandwidth problem
bandwidth under permutation π 13, 18, 32
benchmark instances
  large 77
  small 77, 87
  very large 78
benchmark suite
  the first . . . . . *see* the popular
  the popular 15, 39, 77, 110
  the second 15, 40, 78, 110
best known results
  for some hard instances 15
  lower bounds 15, 77
  LRX 77, 103–105
  MCP 77, 87–91, 100–105
  SRB 78, 103, 106
  upper bounds 15, 77, 103
binary search 71
branch-and-bound 8, 18, 64, 108
  bounding 8, 76
  branching 8, 76
  general algorithm 9
branch-and-bound for BW
  algorithm with 2-labeling 66, 67
  algorithm with constraints . . . 53, 54
  algorithm with dominance rules 61, 62
  basic algorithm 27
branch-and-bound tree 8, 26, 68, 86
branch-and-cut 7, 17, 73
BW 13, 18, 19, 25, 26, 46, 47, 53, 55, 60, 63, 68, 70, 71, 81, 108
  search space 71, 82, 109

child nodes 69
cluster 9
COIN-OR 72
complexity
  of bandwidth problem 2

theory 6
complexity class
$\mathcal{NP}$ 6
$\mathcal{NP}$-complete 7
$\mathcal{NP}$-hard 7
$\mathcal{P}$ 6
compression of topological information 42, 108
traditional approaches 41
computational results
benchmark suites *see* benchmark suite
hardware 78
lower bounds
large instances 88, 90, 91
parallel computation 95, 99, 101–103
small instances 87–89
very large instances . . 92, 94, 96, 97
optimal solutions 88, 91, 92
upper bounds 103
large instances 104, 105
small instances 103, 104
very large instances 106
computer node 9
constraint 19, 25, 44, 75, 108, 110
h_{heu} 46–48, 81
constraint 80
cut 49, 51
density cut constraint . 49–53, 80, 109
density near-cut constraint 52, 53, 80, 109
effect on the solver 84
excess-range 45–47, 80–82
fitting constraint 44–47, 80, 81, 84, 93, 109
generalized 45, 46
illustration convention 28
near-cut 51
pulling constraint 47, 48, 80, 81, 84, 93, 109
generalized 48
the first constraint 29, 80
the second constraint 30, 48–50
cycle 5
Hamiltonian 5
degree 5
delta information 42
dense graph 20
diameter 5
distance 5, 93
dominance relation 55–58, 60, 75, 80, 108, 109
effect on the solver 85
illustration convention 55
processing 60
processing algorithm 62
results 60
dominance rule 55–57, 60, 63, 64, 85, 108–110
dominate 55–58, 63
dominating 58

edge 5
parallel edge 5
efficiency 10
endvertex 5
exact methods 2, 108
BB 15
BothWay 14
dynamic programming 13
Hebas 73
LeftToRight 14
MB_ID 14
MB_PS 14
Parallel BothWay 15
exponential time 6
extendability problem 20, 108
BPPE 21
LPPE 20
extendability test . . 14, 22, 23, 61, 63, 75
algorithm 22–24
violation 19, 26, 75

fathomed 8, 69
feasible permutation 19, 25, 26, 56, 57, 69, 75
forest 5
free set 19
free vertex 18

global upper bound 8

GPS 11, 32, 42, 103, 108
pseudo diameter 33
running time 34
vertex numbering procedure 33, 107, 108
graph
connected 5
simple 5
undirected 5

hash-table 58, 59, 80, 86, 109
Hebas 73, 79
changes for very large instances . . . 92
class diagram 74
configurations 79, 81
implementation 73
parallel version 80
performance evaluation . 81, 82, 84, 85
parallel version 97, 99, 100
heuristics 2, 108
CM 11, 32
GA 12
GPS 11, 32
GRASP 12
iGPS 32, 35
multilevel 12
NS 12
SA-σ 12, 38
SS_TS 12
TS 12
WBRA 11

iGPS 42, 71, 72, 103
running time 35
solution quality 35
incident 5
Integer Programming (IP) 7
IP formulation
of bandwidth problem 17

label 13, 18, 26
max . . . . . *see* max label
min . . . . . *see* min label
label domain 13, 19, 21, 71, 75, 109
tightened 19
LB Improvement 79
left set 19, 26
level structure 32
depth 33
rooted at v 32
width 33
linear layout 19, 49, 55, 109
Linear Programming (LP) 7
load balancing 73, 76
dynamic 76
static 76
lower bound
initial 71
lower bounds 14
$\alpha(G)$ 14, 16
$\gamma(G)$ 14, 16
LP relaxation 7, 17, 69

Matrix Market 15, 77
max label 26, 29, 44, 50
min label 26, 29, 30, 46, 48
MPI 9
MPICH 9, 73
multi-labeling 63

neighbor 5
node . . . . . *see* subproblem

objective function 7, 17

parallel computer 8
message-passing model 9
shared-memory model 9
parallel framework 72
ALPS 73
CHiPPS 73
PEBBL 73
SYMPHONY 72
parallel system 9
hub 73, 76
master 73, 76
master-hub-worker scheme . . . 73, 76
master-worker scheme 73
worker 73, 76
parSA 40
partial permutation 13, 18, 75, 108
assigned set 19
both-sided partial permutation . . . 19

extendability 13, 26
extended 19–22, 25, 26
free set 19
free vertices 13
illustration convention 28
label domain 19
left partial permutation 19
left set 19
right set 19
path 5
length of path 5
permutation 13, 18, 32, 33
polynomial time 6
problem
decision 6
optimization 7
search 7, 13, 18, 70
PVM 9, 73

relaxation-based 69, 73
right set 19, 26
root vertex 8, 26
rotation 39, 40
rounding heuristic 25, 72
running time 6

SA 38
SA-σ 38, 40, 71, 103, 108
algorithm 38
neighbor functions 39
parameters 40
scalability 10, 73
scalable 10
search parameter φ 18, 28, 45, 46, 71, 81, 97, 102, 105, 109
search strategy 8, 70
best-first 70
breath-first 70
depth-first 26, 70
hybrid 70, 71
Sigma 36–38, 71, 103, 108
computing 37
single-labeling 63, 64, 68, 86, 109
smoothing function 36
sorting
bucket sort 20–23, 93
introsort 93
sparse graph 14, 21, 41, 93
speedup 9
Standard Template Library (STL) . . . 93
subproblem 8
fathomed 8
queue 8, 70, 75
split 8, 69
swap 39, 40

The University of Florida Sparse Matrix Collection 15, 78
time complexity function 6
topological information 41, 43
tree 5

UB Deviation 79

vertex 5
vertex number 28, 41, 42, 55

zero-edge 77, 93, 96, 97