COMPUTATIONAL APPROACHES FOR HARD DISCRETE OPTIMIZATION PROBLEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Daniel Fleischman

August 2016

COMPUTATIONAL APPROACHES FOR HARD DISCRETE OPTIMIZATION PROBLEMS

Daniel Fleischman, Ph.D.

Cornell University 2016

In order to create a simple algorithm that is easy to implement and to understand one must have a deeper understanding of the problem being solved. In spite of that, when studying algorithms, simplicity, implementability, and other "real world" considerations are aspects often shadowed by asymptotic runtime and complexity results.

We study three different scenarios where we match or beat the asymptotic runtime of the best known algorithms for several problems, and have solutions that run faster in practice than any previously known solution, sometimes by several orders of magnitude.

The first problem presented deals with extending an important subclass of Dynamic Programming solutions; when the underlying DAG is a *Grid DAG*. It is a general framework that allows us to solve windowed and cyclic versions of the original problem with little of no increase of runtime. The framework is particularly useful in computational biology where cyclic strings are common, but it can also be applied outside this realm.

The second problem is the *graph bisection* problem which can be loosely defined as finding a cut of minimum cost in a graph, such that both sides of the cut have roughly the same number of vertices, and it is strongly NP-Hard. This problem has several applications ranging from compiler optimization to VLSI circuit design. By understanding characteristics of instances that arise in

practice we created a branch-and-bound solution that solves incredibly large instances in a reasonable amount of time.

The last problem is the so called *a priori TSP* where we must decide on a master tour before knowing the actual subset of cities that should be visited. We focus on solving the problem when a subset of all cities is chosen according to a probability P given explicitly. We show how to extend any approximation algorithm or heuristics for the metric TSP to a solution to the a priori TSP, while giving approximation guarantees. The solution proposed is useful when the support of P is small.

BIOGRAPHICAL SKETCH

Daniel Fleischman received B.S. degrees in Mathematics and Computer Engineering, and a M.S. degree in Computer Science from the Pontifícia Universidade Católica, in Rio de Janeiro, RJ, where he graduated first-in-class. He continued his graduate work in Operations Research and Information Engineering at Cornell University, in Ithaca, NY, where he earned a Ph.D. degree in August 2016. He spent the summers of 2012 and 2014 at Microsoft Research in Mountain View, CA and the summer of 2015 at Amazon in Seattle, WA.

Daniel advanced to the world finals of the Association for Computing Machinery International Collegiate Programming Contest in 2006 and 2007 as a contestant, and to the 2010, 2015, and 2016 world finals as coach, the latter two instances leading the Cornell team.

To my grandmother Frida.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor David Shmoys for his guidance during my time at Cornell. His passion for the field is apparent to anyone, and the amount of effort he puts in everything he does is remarkable – quoting Bob Bland, "David sleeps negative four hours a day and works for the remaining twenty eight hours". Observing the way he approaches problems taught me a lot, and certainly will shape my future research career.

I extend my gratitude to all the professors in the ORIE department for their passion and their friendship. Thank you Krish Iyer for liking Mathematical puzzles as much as I do, and for providing me with numerous variations of the "prisoners and hats" puzzle. I was fortunate to have Mike Todd and David Williamson in my committee. The conversations I had with them throughout these years, and the opportunity to TA for them were invaluable experiences. I became a better teacher and researcher by observing them closely.

I am also grateful to my many friends at Cornell. This includes my good friends inside the ORIE department, like Chaoxu Tong, Kenneth Chong, and James Davis, and outside, like my cousin, roommate, cachaça-buddy and much more Ilan Shomorony, and the co-host of so many "Great Parties" Leo Caroli.

I would like to also thank all my students and mentees for their excitement about learning something new, for asking questions that went far beyond the scope of what being taught. In particular I would like to thank Saketh Are, Victor Reis, Eduardo Ferreira, Jake Silverman, and Rafael Marinheiro for the effort put in the training provided and for representing Cornell so well in the *ACM-ICPC* World Finals twice. The memories from these two trips will certainly last forever.

Last but definitely not least I would like to thank my family for all the

support. My parents and brother provided emotional support whenever I needed. Them together with my grandparents, aunts, uncles, cousins, sister-in-law, nieces and nephews form what is perhaps the best family ever to exist. On the other side of the Americas, closer to Ithaca, my girlfriend Melanie Sand and her family gave me an introduction to the American culture, receiving me at their home for Thanksgiving and Christmas. I am grateful even for the many times they made fun of me for mispronouncing a word.

This work was supported by the National Science Foundation, grants CMMI-1537394, CCF-1526067, CCF-1017688, and CCF-0832782.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Some of the most important algorithms were created by a combination of theoretical achievements and implementation considerations. Take Ford-Fulkerson's algorithm for maximum flow [44], for instance. It is simple to understand and to implement, but it is based on a solid scaffold of theorems culminating on the famous max flow-min cut theorem. Other algorithms that could be used as example are Dijkstra's [35], Floyd-Warshall [43, 104], Kruskal's [71], Knuth-Morris-Pratt [69], the Miller-Rabin primality test [87], etc.

Perhaps a more interesting example has to do with Highway Dimension [2] where the authors were solving the shortest path problem in continental sized graphs. Their algorithm was correct for all graphs [49] but worked particularly well on road networks. While trying to understand why road networks were particularly nice for their algorithm they formally defined what aspects of a graph make it look like a road network; they defined a metric called *highway dimension* that measures how similar to a road network a given graph is (smaller numbers mean more similar) [2]. They then proved that their algorithm indeed perform well in graphs with low highway dimension. Finally, by using the concept of highway dimension they came up with a better algorithm [1].

We are interested in studying this interchange between theory and practice, and how it leads to algorithms that are simple and implementable. Simplicity is usually overlooked, but we argue that it should be a central component of algorithm design. A simple algorithm requires more intuition about the problem being solved, and usually provides bigger insights on why the algorithm works.

We will discuss mainly three algorithms for three completely different problems to show several techniques for when designing algorithms. All these algorithms have a unifying characteristic to them; they were designed while taking implementability into account. By taking input from practical aspects we developed intuition on the problem, which in turn made it possible for us to create solutions that match or outperform the best known asymptotic runtime, but are faster and simpler in practice.

The first problem (in Chapter 2) is the problem of how to find several longest path trees (for different sources) in a DAG with special structure. This algorithm can then be used in a myriad of applications [14, 21, 28, 85]; it is particularly useful in computational biology where cyclic DNA strings are common (either because we are considering bacteria, whose DNA is circular [105] or because the sequencing method used cannot differentiate between cyclic permutations of the same DNA [101]).

We present a clean and easy algorithm to solve it that requires nothing but elementary understanding of Mathematics. By doing so we have solutions to several problems that at least match the asymptotic runtime of the best known ad-hoc algorithms. Our method achieves better results in practice, and is considerably simpler than previous approaches.

The main technique of Chapter 2 is the extensive use of data structures. We make the point that by storing data in different ways (usually implicitly) one can achieve better results.

The problem solved in Chapter 3 is finding the *minimum ϵ-balanced bisection* of an input graph G (or simply a minimum bisection); the problem is strongly

NP-Complete [46]. This problem has several applications including computer vision [72], load balancing [55], and Jostle [103]. The minimum bisection has been studied by several people and numerous heuristics were written to give solutions for some classes of instances; it was not expected that anyone would solve to optimality large instances in a reasonable amount of time.

Our algorithm is presented in details in Chapter 3. The solution is based on theoretical results, and also a lot of careful implementation. Most of the intuition necessary to develop the algorithm came from looking at graphical representation of instances and understanding what make them treatable or untreatable.

The lesson from Chapter 3 is that you can gain a lot of intuition by looking at instances, formulate hypotheses and test them. Sometimes we should put *science* back in *Computer Science*.

Finally, in Chapter 4 we approach an extension of the classic *Traveling Salesman Problem* (TSP) where we do not know a priori which subset of cities will have to be served; it is called *a priori TSP*. The problem asks for a tour over all cities that will be then shortcut to a random subset of vertices, and we are interested in minimizing the expected length of such shortcut tour. We present a simple algorithm to solve the a priori TSP when the probability distribution on which a subset of the nodes will be selected is given explicitly. Other models also exist, and solutions for them will be discussed in Section 4.3.

Our algorithm uses an approximation algorithm or a heuristic to TSP as a black box, and in the case the black box have approximation guarantees, our algorithm will also have.

CHAPTER 2

A SIMPLE FRAMEWORK FOR WINDOWED PROBLEMS

2.1 Introduction

Many dynamic programming (DP) solutions for optimization problems can be described as finding a longest path in a directed acyclic graph (DAG) [29]. An important collection of such DAGs can be laid in a 2-dimensional grid of horizontal and vertical arcs and additional diagonal arcs. Rows are indexed from 0 to N and columns from 0 to M, and we want the longest path from $[0, 0]$ to $[N, M]$.

Define a window as a set of indices $0 \leq a \leq b \leq N$ and set its value to be the length of the longest path from $[a, 0]$ to $[b, M]$. In this chapter we will show how to proceed to find the value of all windows with no increase on runtime when compared to finding a single longest path tree from $[0, 0]$.

This can be used to solve several problems from computational biology. Problems like Episode Matching [28], Longest Common Cyclic Subsequence [14, 21, 85], Episode Quasi-Matching, and others can be put into this framework. In most of these problems the runtime complexity of our algorithm at least matches the best available, and in a few cases we beat the state-of-the-art. Furthermore our method is a unifying framework that is faster in practice, and simpler to understand and to implement than most ad-hoc solutions to these problems.

Throughout this chapter we will use the *Longest Common Cyclic Subsequence* (LCCS) in most of the images and examples. We will use only unit diagonal arcs

in most examples not to clutter the figures.

The remainder of this chapter is organized as follows. In Section 2.2 we show how to view DP solutions as the longest path in a DAG. In Section 2.3 we give the formal definition of Grid DAGs and the problem we will solve. In Section 2.4 we show the outline of the method to solve it. In Section 2.5 we study the structure of the longest path trees. In Section 2.6 we present the main algorithm. In Section 2.7 we show how to modify the main algorithm for the case where all diagonal arcs have unit length. In Section 2.8 we present computational results. In Section 2.9 we make some final remarks.

2.2 DP Solutions as Longest Paths in DAGs

Dynamic Programming is a very broad technique that can be used to solve optimization problems, counting problems, decision problems, and others. In all its generality it can be characterized by a method to compute a function f over a finite set S of *states*. Every state $x \in S$ *depends on* a set of states D_x, i.e. $f(x)$ is defined recursively as a function of $f(y)$ for all $y \in D_x$, or $f(x) = R_x(f|_{D_x})$ where $f|_A$ is f restricted to A.

Define the directed graph $G = (S, A)$ where $(i, j) \in A$ if $i \in D_j$. To be a properly defined DP, the graph G must be a Directed Acyclic Graph (DAG), and we can compute $f(x)$ for all $x \in S$ using Algorithm 1. There is an implicit requirement that the functions R_x should be "computationally simple".

Algorithm 1 Dynamic Programming

```
1: function DP(S, D, R)
2:     G ← DAG(D)
3:     S' ← topological order of S in G
4:     for x ∈ S' do
5:         f(x) ← R_x(f|_{D_x})          ▷ f(y) already computed ∀y ∈ D_x
6:     return f
```

Example 1 (Maximum Contiguous Sum)**.** *Consider the following problem:*

Input *A sequence of integers* $A = [a_1, a_2, \ldots, a_N]$.

Output *Two indices* $i \leq j$ *such that* $a_i + a_{i+1} + \ldots + a_j$ *is maximized.*

We can solve this problem with Dynamic Programming by having a set $S = \{1, 2, \ldots, N\}$ *and defining* $f(x)$ *to be the maximum possible value with indices* $(i, j = x)$ *for some* i.

It is easy to check that $f(1) = a_1$ *and* $f(x) = \max(f(x-1) + a_x, a_x)$ *where the two arguments of the maximum are the maximum possible sum having* $i < j$ *and* $i = j$ *respectively.*

Once we have evaluated f *at all points of* S *using Algorithm 1 we can get the answer by setting* $j = \arg\max_x f(x)$ *and finding* i *naively for a* $O(n)$ *algorithm in total.*

This DP is formally defined by $S = \{1, 2, \ldots, N\}$, $D_1 = \emptyset$ *and* $D_x = \{x - 1\}$ *for* $x > 1$, *and* $R_1() = a_1$ *and* $R_x(f(x-1)) = \max(f(x-1) + a_x, a_x)$.

Several DPs have a special form where $D_b = \emptyset$ for exactly one b, $f(b) = 0$, and $R_x(f|_{D_x}) = \max_{y \in D_x} f(y) + d[y, x]$ for all $x \neq b$ where $d[y, x]$ is independent of f. In these cases $f(x)$ is the length of the longest path from b to x in the DAG defined by D with $d[a, b]$ as lengths of the corresponding arcs. See Example 2.

Example 2 (Maximum Contiguous Sum as a longest path)**.** *We will revisit the problem in Example 1, and give a slightly different DP formulation to it.*

Let $S = \{0, 1, 2, \ldots, N\}$ *and define* $f(0) = 0$ *and* $f(x)$ *the same way as before. Now it is easy to see that* $f(x) = \max(f(x-1)+a_x, f(0)+a_x)$ *for* $x \geq 1$*, therefore* $D_x = \{0, x-1\}$*. This second formulation has the form we want.*

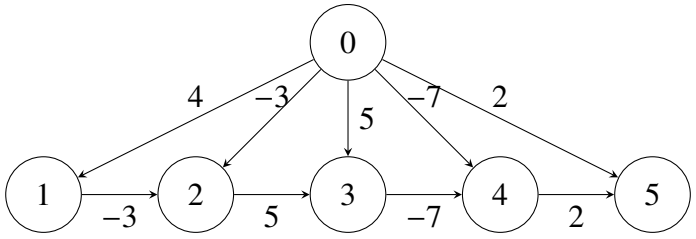


Figure 2.1: Example of the DAG formed from the sequence $[4, -3, 5, -7, 2]$

2.3 Preliminaries

We will consider DPs that has the form as in Example 2, where the problem can be reduced to finding the longest path from a unique vertex b to all other nodes. Furthermore, we will restrict ourselves to an important family of DAGs, namely *Grid DAGs*.

Definition 1. *A DAG* $G = (W, A)$ *is a* Grid DAG *if* $W = \{[i, j] : i \in [N], j \in [M]\}$ *(*$[N] = \{0, 1, \ldots, N\}$*) and* $A = V \cup H \cup D$ *where:*

- $V = \{[i, j] \to [i + 1, j] : \forall i, j\}$ *is the set of "vertical arcs".*
- $H = \{[i, j] \to [i, j + 1] : \forall i, j\}$ *is the set of "horizontal arcs".*
- $D \subseteq \{[i, j] \to [i + 1, j + 1] : \forall i, j\}$ *is the set of "diagonal arcs".*

Arcs in V and H have length 0 and arcs in D have integer length between 1 and L for a fixed $L > 0$.

Several dynamic programming solutions with a quadratic state space can be represented in this form. E.g. Longest Common Subsequence, Longest Increasing Subsequence [24, 68], and Episode Matching [28].

Example 3 (Longest Common Subsequence)**.** *Consider the Longest Common Subsequence (LCS) problem:*

Input *Sequences $A = [a_1, \ldots, a_N]$ and $B = [b_1, \ldots, b_M]$.*

Output *Sequence C which is a subsequence of both A and B with maximal length.*

This problem can be solved by finding a longest path tree in a Grid DAG by taking $D = \{[i-1, j-1] \to [i, j] : a_i = b_j\}$ and assigning length 1 to all diagonal arcs (See Figure B.1.) The application of Algorithm 1 to this DAG is taught as one of the classic examples of DP [24, 68].

Note also that C can be recovered by following the longest path from $[0, 0]$ to $[N, M]$ and inserting one character in C for each diagonal arc traversed.

Fix a Grid DAG $G = (W, A)$ and let $0 \leq a \leq b \leq N$. We are interested in finding the longest path from $[a, 0]$ and $[b, M]$ for some (or all) pairs (a, b). We call such a pair of indices a *window* of size $b - a$ of G whose value $l_G(a, b)$ is equal to the length of the corresponding longest path.

Example 4. *Consider the Longest Common Cyclic Subsequence, an important problem in Computational Biology [21, 59, 78, 80, 84].*

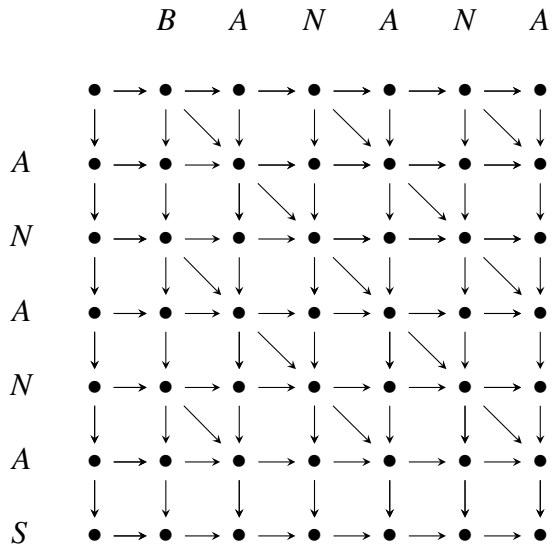


Figure 2.2: Example of the DAG formed by an LCS instance with A = ANANAS and B = BANANA

Input *Sequences* $A = [a_1, \ldots, a_N]$ *and* $B = [b_1, \ldots, b_M]$.

Output *A sequence C which is a subsequence of a cyclic permutation of A and a cyclic permutation of B with maximal length.*

A cyclic permutation A is defined as the concatenation A_1A_2 *where* A_1 *is a suffix of A,* A_2 *is a prefix of A and* $|A| = |A_1| + |A_2|$.

Let C be an optimal solution for a particular input. Then a cyclic permutation of C will be a subsequence of B and of a cyclic permutation of A. So without loss of generality we only consider the trivial cyclic permutation of B. Namely B itself.

We will show how to solve this problem by finding the value of N windows of size N in a Grid DAG G.

Let G be the DAG that would be used to solve the LCS problem with input $A' = [a_1, \ldots, a_N, a_1, a_2, \ldots, a_{N-1}]$ *and B (Note that* $|A'| = 2|A| - 1$*, and G has* $2N(M + 1)$ *nodes).*

The value of the windows $(i, i+N)$ of G for $0 \leq i \leq N-1$ consider all cyclic permutations of A. See Figure 2.3 for an example.

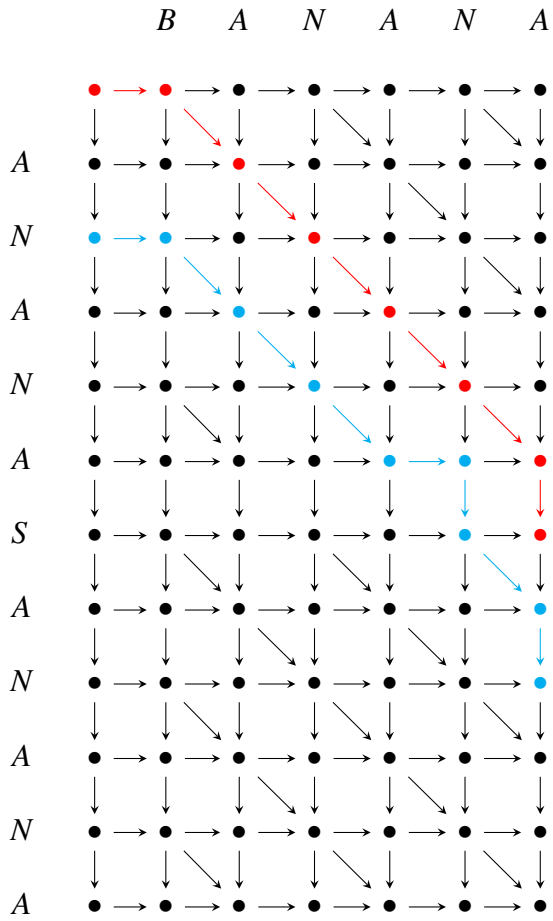


Figure 2.3: Solution for two windows of size 6; $(0, 6)$ and $(2, 8)$ corresponding to cyclic permutations "ANANAS" and "ANASAN" respectively.

We note that it is easy to compute the value of one particular window (a, b) in $O(NM)$ by following Algorithm 1. A topological order is readily given by $([0,0],[0,1],\ldots,[0,M],[1,0],\ldots,[1,M],\ldots,[N,0],\ldots,[N,M])$.

In fact if a is fixed we can compute the value of windows (a, b) for all b by following Algorithm 1 in $O(NM)$. This gives a trivial $O(N^2M)$ algorithm to compute the value of all windows of a Grid DAG.

In the following few sections we will show how to improve this runtime in a simple way.

2.4 Outline

Suppose you are given a Grid DAG and the longest path tree P from $[a, 0]$. With these in hand the values of all windows of the form (a, b) are readily available. We will show how to compute the longest path tree from $[a + 1, 0]$ in sublinear time (i.e. in $o(NM)$) if $N, M = \omega(1)$ and L is fixed.

Let $S_a(i, j)$ be the length of the longest path between $[a, 0]$ and $[i, j]$.

Lemma 1. *Let G be a Grid DAG and let $a \in [N]$. Then $S_a(i, j)$ is nondecreasing in both i and j.*

Proof. *Since there exists an arc $[i, j] \to [i + 1, j]$ of length 0 we have $S_a(i + 1, j) \geq S_a(i, j)$.*

A similar conclusion is reached by using vertical arcs instead of horizontal. □

For $i \geq a$ and $[i, j] \neq [a, 0]$ let $P_a(i, j) \in H \cup V \cup D$ be the arc that "explains" the value of $S_a(i, j)$, i.e., P_a describes the longest path tree starting from $[a, 0]$. See Figure 2.4.

We will abuse notation and say that $P_a(i, j) = H$ if $P_a(i, j)$ is the unique arc in H whose head is in $[i, j]$. In the same way we can say $P_a(i, j) = V$ or $P_a(i, j) = D$. Abusing notation even more we can say that $P_a(i, j) = [i', j']$ if $P_a(i, j) = [i', j'] \to [i, j]$.

If there are multiple longest path trees we choose one by preferring H over D over V as a tie breaking rule. The reason for this choice will become clear soon.

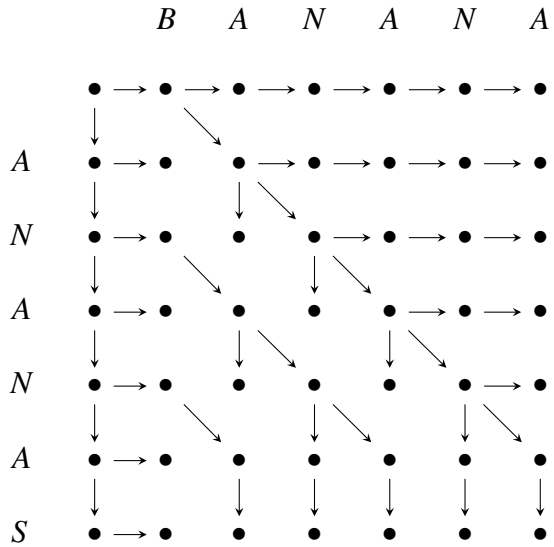


Figure 2.4: Example of P_0 for all nodes (i, j)

Note that P_0 and S_0 can be computed in $O(NM)$ by following Algorithm 1.

2.5 Properties of S_a and P_a

In this section we will study the relationship between the pair (S_a, P_a) and the pair (S_{a+1}, P_{a+1}). The key observations made here will make a fast update from a to $a + 1$ possible.

Note that we can compute $P_a(i, j)$ from S_a by trying the (at most) three alternatives and seeing which ones give the correct value for $S_a(i, j)$.

Let $\Delta_a(i, j) = S_a(i, j) - S_{a+1}(i, j)$. In other words, $\Delta_a(i, j)$ is by how much the value of $S_a(i, j)$ decreases. It is easy to see that $0 \leq \Delta_a(i, j) \leq L$ since we at most take one diagonal arc from the longest path.

Theorem 1. *$\Delta_a(i, j)$ is nondecreasing on j and nonincreasing on i. In other words $\Delta_a(i, j) \leq \Delta_a(i, j + 1)$ and $\Delta_a(i, j) \leq \Delta_a(i - 1, j)$.*

Theorem 1 is central to our results, but its proof is long and was therefore moved to Appendix A.

Define $B_a = \{[i, j] : \Delta_a(i, j) > \Delta_a(i, j - 1)\}$ to be the set of *border vertices*. The next theorem is the main theorem of this section.

Theorem 2. *Let $[i, j] \notin B_a$. Then $P_a(i, j) = P_{a+1}(i, j)$.*

Note that Theorem 2 does not imply that $P_{a+1}(i, j)$ changes for $[i, j] \in B_a$.

Proof. *Let $[i, j] \notin B_a$ and let $\Delta_a = \Delta_a(i, j) = \Delta_a(i, j - 1)$. Let also l be the length of the diagonal arc $[i - 1, j - 1] \to [i, j]$ if such an arc exists. Note that if $P_a(i, j) = P$ then $\Delta_a(i, j) \leq \Delta_a(P)$ (equality can be achieved by doing $P_{a+1}(i, j) = P$.) We will consider each possible value of $P_a(i, j)$.*

We have $P_a(i, j) = H$ if and only if $S_a(i, j) = S_a(i, j - 1)$ if and only if $S_{a+1}(i, j) = S_a(i, j) - \Delta_a = S_a(i, j - 1) - \Delta_a = S_{a+1}(i, j - 1)$ if and only if $P_{a+1}(i, j) = H$. We use our preference of H over V and D for the first and last steps.

Let now $P_a(i, j) = D$. If $\Delta_a(i - 1, j - 1) = \Delta_a$ we trivially have $P_{a+1}(i, j) = D$ by following an argument similar to the one above, so assume not. By Theorem 1 we have $\Delta_a < \Delta_a(i - 1, j - 1) \leq \Delta_a(i - 1, j)$, and

$$\begin{aligned} S_{a+1}(i,j) &= S_a(i,j) - \Delta_a \\ &\geq S_a(i-1,j) - \Delta_a \\ &> S_a(i-1,j) - \Delta_a(i-1,j) \\ &= S_{a+1}(i-1,j) \end{aligned}$$

, so $P_{a+1}(i,j) \neq V$.

Finally let $P_a(i,j) = V$. *Note that* $\Delta_a(i-1,j-1) \geq \Delta_a(i,j-1) = \Delta_a$ *by Theorem 1. Therefore:*

$$\begin{aligned} S_{a+1}(i,j) &= S_a(i,j) - \Delta_a \\ &> S_a(i-1,j-1) + l - \Delta_a && \textit{Since we prefer } D \textit{ over } V \\ &\geq S_a(i-1,j-1) + l - \Delta_a(i-1,j-1) \\ &= S_{a+1}(i-1,j-1) + l \end{aligned}$$

, so $P_{a+1}(i,j) \neq D$. □

Theorem 1 and the trivial fact that $0 \leq \Delta_a(i,j) \leq L$ imply that at most L elements of each row are in B_a . By Theorem 2 only those elements of P_a may change. Let $R_d = \{[i,j] : \Delta_a(i,j) = d\}$ be the region with $\Delta_a = d$.

Note that the regions R_d do not need to be connected, and do not have to start on the top row.

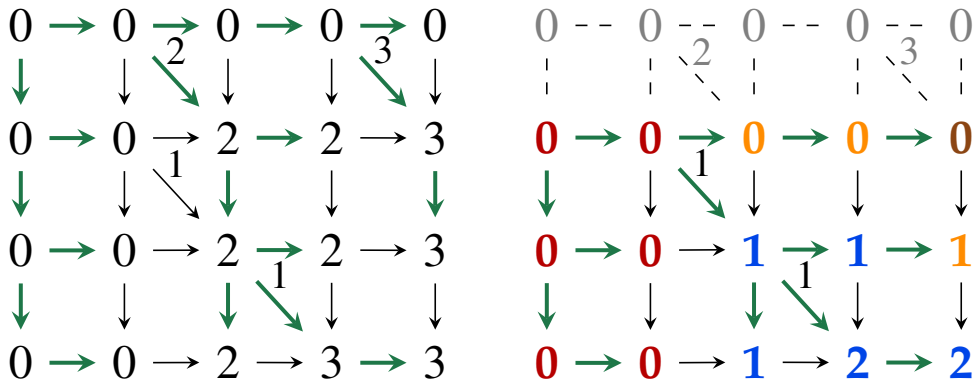


Figure 2.5: Longest path trees P_a and P_{a+1} and regions R_0, R_1, R_2, and R_3

2.6 The Algorithm

The algorithm proceeds row by row maintaining an array of integers x_d for $0 \leq d \leq L$ with the meaning that region R_d on the current row i goes from x_d to $x_{d+1} - 1$. We implicitly maintain $x_0 = 0$ and $x_{L+1} = M + 1$. Note that if R_d does not intersect the current row then $x_d = x_{d+1}$.

Treat the function QUERY of Algorithm 2 as a black box that returns the most updated value of S_{a+1} and the function UPDATESTRUCTURE as a black box that updates all data structures to reflect the fact that we found a new value of x_d. We will show how to implement these functions later. For ease of notation we will let $l_{i,j}$ be the length of the diagonal arc with its head on $[i, j]$ or $-\infty$ if such arc does not exist. The value $-\infty$ can be thought as "impossible" since we are solving a maximization problem.

The correctness of Algorithm 2 is a trivial consequence of Theorem 1, that shows that x_d does not decrease as we advance from row i to row $i + 1$. So we will only analyze the runtime of the algorithm. Let the function QUERY run in time $O(Q)$ and the function UPDATESTRUCTURE runs in $O(U)$.

Algorithm 2 Update P_a to P_{a+1} with diagonal lengths bounded by L

```
1:  function computeDelta(i, j, d)
2:      S_a(i, j) ← query(i, j) + (d − 1)        ▷ already updated row i d − 1 times
3:      S_{a+1}(i − 1, j) ← query(i − 1, j)
4:      S_{a+1}(i, j − 1) ← query(i, j − 1)
5:      S_{a+1}(i − 1, j − 1) ← query(i − 1, j − 1)
6:      S_{a+1}(i, j) ← max(S_{a+1}(i − 1, j), S_{a+1}(i, j − 1), l_{i,j} + S_{a+1}(i − 1, j − 1))
7:      return S_a(i, j) − S_{a+1}(i, j)
8:  procedure Update                              ▷ updates the value of a to a + 1
9:      x_d ← 1 for 1 ≤ d ≤ L
10:     x_0 ← 0
11:     a ← a + 1
12:     for i = a + 1, ..., N do
13:         for d = 1, ..., L do
14:             x_d ← max(x_d, x_{d−1})                          ▷ By Theorem 1
15:             while x_d ≤ M and d > computeDelta(i, x_d, d) do
16:                 x_d ← x_d + 1
17:             if x_d ≤ M then
18:                 updateStructure(i, x_d)
```

Lemma 2. *Algorithm 2 runs in $O(L(NU + (M + N)Q))$.*

Proof. *The runtime of* computeDelta *is clearly $O(Q)$. So let us focus on function* Update.

Note that the total number of times the condition on line 15 evaluates to true is bounded by ML throughout the algorithm, since at every time it does one of the variables x_d is incremented. The number of times it evaluates to false is exactly $(N - a - 1)L = O(NL)$. Therefore the helper function updateStructure *is called $O(NL)$ times. The total runtime is, as wanted, $O(MLQ + NLQ + NLU) = O(L(NU + (M + N)Q))$.* □

Now we present a way to implement query and updateStructure. We will use a data structure that implicitly maintains an array V of integers initialized to 0 and implements the following operations.

- increment(j) - Performs $V[j] \leftarrow V[j] + 1$.

- QUERY(j) - Returns $V[1] + \ldots + V[j]$.

If such a structure is provided to us we can maintain one V_i for each row i. Whenever $[i, j] \in B_a$ and $q = \Delta_a(i, j) - \Delta_a(i, j-1)$ we increment $V_i[j]$ q times. Now V_i.QUERY(j) returns $\sum_a \Delta_a(i, j)$ over all calls to Algorithm 2 so far. Therefore $S_a(i, j) = S_0(i, j) - V_i$.QUERY($j$) where a is the number of times UPDATE was called.

See Algorithm 3 for an implementation of functions QUERY and UPDATESTRUCTURE used in Algorithm 2.

Algorithm 3 Auxiliary functions for Algorithm 2

```
1:  function QUERY(i, j)
2:      if i ≤ a then
3:          return 0
4:      return S_0(i, j) − V_i.QUERY(j)

5:  function UPDATESTRUCTURE(i, j)
6:      V_i.INCREMENT(j)
7:      S_{a+1}(i, j) ← QUERY(i, j)          ▷ By Theorem 2 only these entries of P can
    change
8:      S_{a+1}(i − 1, j) ← QUERY(i − 1, j)
9:      S_{a+1}(i − 1, j − 1) ← QUERY(i − 1, j − 1)
10:     S_{a+1}(i, j − 1) ← QUERY(i, j − 1)
11:     if S_{a+1}(i, j) = S_{a+1}(i, j − 1) then
12:         P(i, j) ← H
13:     else if S_{a+1}(i, j) = S_{a+1}(i − 1, j − 1) + l_{i,j} then
14:         P(i, j) ← D
15:     else
16:         P(i, j) ← V
```

While there are several data structures that can implement increment and query efficiently we advocate for Fenwick Trees [40]. These structures implement both operations in time $O(\log M)$ where M is the size of the implicit array. It is very efficient in practice and its implementation is particularly simple (see for instance [42]). This gives a runtime of $O(L(N + M) \log M)$ to Algorithm 2.

We can therefore (if wanted) compute the values of all windows in total time $O(NM + LN(N + M) \log M) = O(LN(N + M) \log M)$. When L is small this shows a substantial improvement over the trivial $O(N^2 M)$ algorithm.

2.7 Faster algorithm when $L = 1$

When $L = 1$ we can modify Algorithm 2 to update P in $O(N + M)$, for a total runtime of $O(N(N + M))$ to compute all windows.

Let us start with a few simple observations. In this section we will always assume that $L = 1$.

Lemma 3. *The following properties about S_a and P_a hold for all a.*

1. *$S_a(i, j)$ is equal to the sum of the lengths of diagonal arcs in the unique path from $[a, 0]$ to $[i, j]$ in P_a.*
2. *$S_a(a + i, j) \leq \min(i, j)$.*
3. *$S_a(i, j) \leq S_a(i, j + 1) \leq S_a(i, j) + 1$.*

Proof. *We now prove each of these properties.*

1. *This is a direct consequence of the length of the arcs in Grid DAGs.*
2. *The previous observation implies trivially that $S_a(a + i, j) \leq \min(i, j)$.*
3. *The arc $[i, j] \to [i, j + 1] \in H \subseteq A$ has length 0, therefore $S_a(i, j) \leq S_a(i, j + 1)$.*

   We will now prove that $S_a(i, j + 1) \leq S_a(i, j) + 1$ by induction on i. It is clearly true at $i = a$, since $S_a(a, j) = 0$ (there are no diagonal arcs to cross).

   Now assume it is true at $i - 1$ and consider the value of $P_a(i, j + 1)$.

$\mathbf{P_a(i, j+1) = H}$ $S_a(i, j+1) = S_a(i, j)$.

$\mathbf{P_a(i, j+1) = V}$ *Then* $S_a(i, j+1) = S_a(i-1, j+1) \leq S_a(i-1, j) + L \leq S_a(i, j) + L$, *where the first inequality comes from the inductive hypothesis, and the second is implied by* S_a *being nondecreasing in i.*

$\mathbf{P_a(i, j+1) = D}$ $S_a(i, j+1) \leq L + S_a(i-1, j) \leq L + S_a(i, j)$.

□

Note also that since $\Delta_a(i, j) \leq 1$ we have that if $[i, j] \in B_a$ then $\Delta_a(i, j) = 1$. We are ready to prove the main theorem of this section.

Theorem 3. *If* $[i, j] \in B_a$ *then* $P_{a+1}(i, j) = H$.

Proof. *Note that since* $[i, j] \in B_a$ *we have* $\Delta_a(i, j) = 1$ *and* $\Delta_a(i, j-1) = 0$. *Therefore:*

$$\begin{aligned} S_{a+1}(i, j) &= S_a(i, j) - 1 \\ &\leq S_a(i, j-1) && \text{By Lemma 3} \\ &= S_{a+1}(i, j-1) \end{aligned}$$

Lemma 1 and the fact that we prefer H over V or D complete the proof. □

Theorem 3 allows us to update P without needing to query the current value of S, therefore becoming a logarithmic factor faster than Algorithm 2.

Note also that for $i \geq a + 1$ we have $\Delta_a(i, j) = 0$ if and only if $\Delta_a(P_a(i, j)) = 0$, which is easy to prove by induction. When $i = a + 1$ we have $[i, j] \in B_a$ if and only if $P_a(i, j) = D$, by Lemma 3. We will now present the algorithm for $L = 1$.

Algorithm 4 is clearly correct and clearly runs in $O(N + M)$. This result matches the best available runtime but is a lot simpler.

Algorithm 4 Algorithm 2 specialized for $L = 1$

```
1: procedure UPDATE
2:     x ← 1                                  ▷ x is equivalent to x_1 from Algorithm 2
3:     xp ← 0                                 ▷ Marks the final value of x on the previous row
4:     a ← a + 1
5:     for i = a, ..., N do
6:         while x ≤ M do
7:             [i', j'] ← P(i, x)             ▷ Note that we use an abuse of notation here
8:             if i' = i − 1 and j' ≥ xp then                                  ▷ [i, x] ∈ B_a
9:                 P(i, x) ← H
10:                S(i, M) ← S(i, M) − 1 ▷ We only care about the value of the last column of S
11:                break
12:            x ← x + 1
13:        xp ← x
```

2.8 Computational Results

All codes in this section were written in C++, compiled with gcc 5.1.0, and ran on Amazon Web Services, on a single core of instances of type r3.8xlarge.

Table 2.1 shows the time dependency to compute all windows for a square Grid DAG with increasing dimension. We generated instances randomly and used $L = 3$.

Table 2.1: Dependency on the size of the graph with fixed L

Small Instances			Large Instances		
$N = M$	L	Time (s)	$N = M$	L	Time (s)
100	3	<0.001	5000	3	4.22
300	3	0.008	7000	3	9.0
500	3	0.024	9000	3	15.616
700	3	0.052	10000	3	18.856
900	3	0.088	30000	3	283.5
1000	3	0.116	50000	3	968.948
3000	3	1.356	100000	3	4087.123

The slightly super quadratic dependency on the size is clear in Table 2.1.

Next we analyze the effect of L in the runtime. The results appear on Table 2.2.

Table 2.2: Effect of L in the runtime.

$N = M$	L	Time (s)
20000	5	152.672
20000	25	170.644
20000	45	193.168
20000	65	206.816
20000	85	222.916
20000	100	240.004
20000	300	316.86
20000	500	438.152
20000	700	544.124
20000	900	704.444

As it can be clearly seen on Table 2.2 while the worst case analysis gives a linear dependency between runtime and L, in practice the dependency is way more mild. This comes from line 14 of Algorithm 2, when we can increase the value of a particular x_d significantly "for free".

Finally table 2.3 shows results of LCCS instances from several bacteria DNAs [102], with $L = 1$. Bacterial DNA is circular [105] so it is a perfect candidate for LCCS. We run both the general algorithm (Algorithm 2) and the specialized algorithm for $L = 1$ (Algorithm 4).

Table 2.3: Computational Results for the generic algorithm and the specialized algorithm for $L = 1$. Times marked with a single dash mean the machine did not have enough memory to finish.

Instance	N	M	Algorithm 2 (s)	Algorithm 4 (s)
007717x009926	3973	374162	15.76	8.14
007717x009929	3973	226681	11.56	5.6
007717x009930	3973	177163	8.78	4.4
007717x009937	3973	5369773	108.3	102.4
007717x013860	3973	261597	12.032	6.1
009929x009928	226681	273122	-	3523.7
013860x009928	261597	273122	-	2563.1
009926x009927	356088	374162	-	4170.6
009929x009927	226681	356088	-	2289.4
009937x013212	49962	5369773	-	3115.7
013093x013212	49962	8248145	-	6199.1

Note that on instance 007717x009937 the runtimes for Algo 1 and Algo 2 were very similar. This is due to the fact that the time to compute S_0 dominates the total time. In Grid DAGs where $N \ll M$ we usually observe a performance that is better than the worst case analysis predicts. This is due to the fact that the variables x_1 and x (for each algorithms) do not reach large values.

2.9 Final Remarks

We provide an algorithm to compute all windows of a Grid DAG in time $O(LN(N+M)\log M)$ or $O(N(N+M))$ when $L = 1$. This result improves on the state-of-the-art [21, 59, 78, 80, 84] in runtime and in simplicity.

This algorithm is still too expensive to be used on computational biology, where the use of heuristics is still, unfortunately, the only way to approach the

problems.

One might be able to improve on the results presented here by using known structure of typical Grid DAGs that arise in a particular application.

There are several other problems that fit this framework, and would be implemented almost without any change in the implementation. To name a few:

- Episode Matching - Given two sequences A and B we are interested in finding the smallest window of A that contains B as a subsequence.
- Episode Quasi-Matching - Given two sequences A and B, and an integer t, find the smallest window of A that contains B' as a subsequence where B' differs to B in at most t positions.
- Forbidden subsequence - Given two sequences A and B find the largest window of A that does not contain B as a subsequence.
- Longest Increasing Cyclic Subsequence - Given a sequence A of elements of a totally ordered set find the largest increasing sequence C that is a subsequence of some cyclic permutation of A. This is solved by realizing that LIS(A) = LCS(A, SORTED(A)) where SORTED(A) is formed by the unique elements of A sorted in increasing order.

CHAPTER 3

AN EXACT COMBINATORIAL ALGORITHM FOR MINIMUM GRAPH BISECTION

3.1 Introduction

We consider the *minimum graph bisection* problem. It takes as input an undirected graph $G = (V, E)$, and its goal is to partition V into two sets A and B of roughly equal weight so as to minimize the total cost of all edges between A and B. This fundamental combinatorial optimization problem is a special case of *graph partitioning* [45], which may ask for more than two cells. It has numerous applications, including image processing [97, 106], computer vision [72], divide-and-conquer algorithms [77], VLSI circuit layout [15], distributed computing [79], compiler optimization [62], load balancing [55], and route planning [13, 30, 32, 57, 58, 63, 75]. The minimum bisection problem is NP-hard [46] for general graphs, with a best known approximation ratio of $O(\log n)$ [88]. Only some restricted graph classes, such as grids without holes [38] and graphs with bounded treewidth [60], have known polynomial-time solutions.

In practice, there are numerous general-purpose heuristics for graph partitioning, including CHACO [56], METIS [67], SCOTCH [20, 86], Jostle [103], and KaHiP [92, 93], among others [11, 19, 53, 74, 99]. Successful heuristics tailored to particular graph classes, such as DibaP [82] (for meshes) and PUNCH [31] (for road networks), are also available. These algorithms can be quite fast (often running in near-linear time) and handle very large graphs, with tens of millions of vertices. They cannot, however, prove optimality or provide approximation guarantees. Moreover, with a few notable exceptions [93, 99], most of these al-

gorithms only perform well if a certain degree of imbalance is allowed.

There is also a vast literature on practical exact algorithms for graph bisection (and partitioning), mostly using the branch-and-bound framework [73]. Most of these algorithms use sophisticated machinery to obtain lower bounds, such as multicommodity flows [95,96] or linear [7,16,41], semidefinite [5,7,66], and quadratic programming [52]. Computing such bounds, however, can be quite expensive in terms of time and space. As a result, even though the branch-and-bound trees can be quite small for some graph classes, published algorithms can only solve instances of moderate size (with hundreds or a few thousand vertices) to optimality, even after a few hours of processing. (See Armbruster [5] for a survey.) Combinatorial algorithms can offer a different tradeoff: weaker lower bounds that are much faster to compute. An algorithm by Felner [39], for example, works reasonably well on random graphs with up to 100 vertices, but does not scale to larger instances.

This article introduces a new purely combinatorial exact algorithm for graph bisection that is practical on a wide range of graph classes. It is based on two theoretical insights: a *packing bound* and a *decomposition* strategy. The packing bound is a novel combinatorial lower bound in which we take a collection of edge-disjoint trees with certain properties and argue that any balanced bisection must cut a significant fraction of them. The decomposition strategy allows us to contract entire regions of the graph without losing optimality guarantees; we show that one can find the optimal solution to the input problem by independently solving a small number of (usually much easier) subproblems.

Translating our theoretical findings into a practical algorithm is not trivial. Both the packing bound and the decomposition technique make use of certain

nontrivial combinatorial objects, such as collections of disjoint trees or edges. Although building feasible versions of these objects is straightforward, the quality of the bounds depends strongly on the properties of these structures, as does the size of the resulting branch-and-bound tree. This motivates another important contribution of this article: efficient algorithms to generate structures that are good enough to make our bounds effective in practice. While these algorithms are heuristics (in the sense that they may not necessarily lead to the best possible lower bound), the lower bounds they provide are provably valid. This is all we need to ensure that the entire branch-and-bound routine is correct: it is guaranteed to find the optimal solution when it finishes.

Finally, we present experimental evidence that combining our theoretical contributions with the appropriate implementation does pay off. Our algorithm works particularly well on instances with relatively small minimum bisections, solving large real-world graphs (with tens of thousands to more than a million vertices) to optimality. See Figure 3.1 for a simple example and Figure C.1 for an example in a real instance. In fact, our algorithm outperforms previous techniques on a wide range of inputs, often by orders of magnitude. We can solve several benchmark instances that have been open for decades, sometimes in a few minutes or even seconds. That said, our experiments also show that there are classes of inputs (such as high-expansion graphs with large cuts) in which our algorithm is asymptotically slower than existing approaches.

The remainder of this chapter is organized as follows. After establishing in Section 3.2 the notation we use, we explain our new packing bound in detail in Section 3.3. Section 3.4 then proposes sophisticated algorithms to build the combinatorial objects required by the packing bound computation. We then show,

in Section 3.5, how we can fix some vertices to one of the cells without actually branching on them; this may significantly reduce the size of the branch-and-bound tree and is crucial in practice. Section 3.6 introduces our decomposition technique and proposes practical implementations of the theoretical concept. Section 3.7 explains how all ingredients are put together in our final branch-and-bound routine and discusses missing details, such as branching rules and

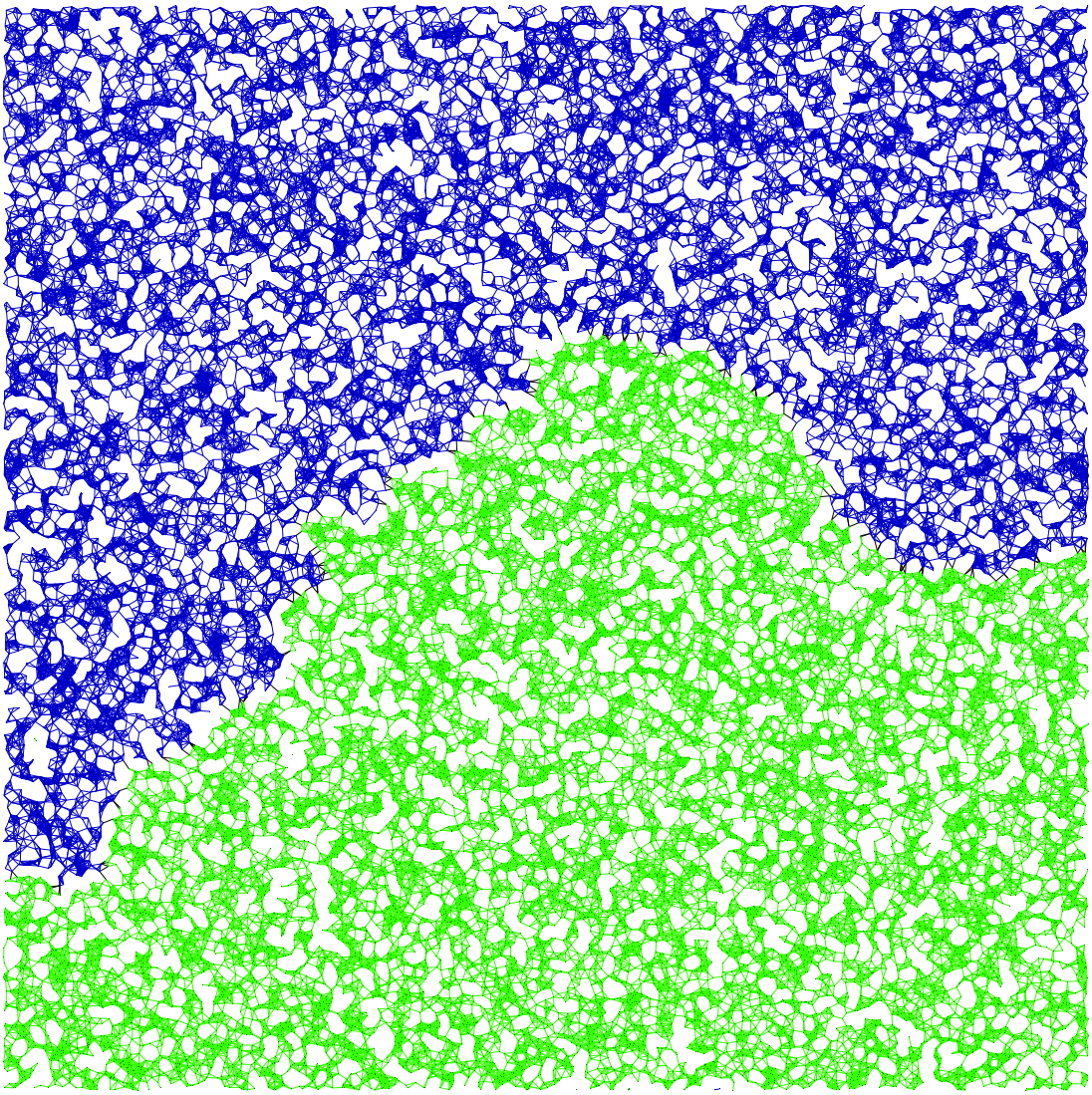


Figure 3.1: Minimum bisection of rgg15*, a random geometric graph with 32768 vertices and 160240 edges. Each cell (with a different color) has exactly 16384 vertices, and there are 181 cut edges.*

upper bound computation. Section 3.8 then shows how our techniques can be extended to handle large edge costs efficiently. Section 3.9 introduces techniques to handle very large graphs where the cost of computing maximum flows is prohibitive, and Section 3.10 shows how to handle graphs with small degree by simulating fractional flow. Finally, Section 3.11 presents extensive experiments showing that our approach outperforms previous algorithms on many (but by no means all) graph classes, including some corresponding to real-world applications.

Most of the figures of this chapter are present in Appendix C to make this chapter flow easier.

3.2 Preliminaries

Consider an undirected graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges. Each vertex $v \in V$ has an integral nonnegative *weight* $w(v)$, and each edge $e \in E$ has an associated positive integral *cost* $c(e)$. By extension, for any set $S \subseteq V$, let $w(S) = \sum_{v \in S} w(v)$ and let $W = w(V)$ denote the total weight of all vertices. A *partition* of G is a partition of V, i.e., a set of subsets of V which are disjoint and whose union is V. We say that each such subset is a *cell*, whose weight is defined as the sum of the weights of its vertices. The *cost* of a partition is the sum of the costs of all edges whose endpoints belong to different cells. A *bisection* is a partition into two cells. For a given $\epsilon \geq 0$, we are interested in computing a partition of V into exactly two sets (cells) such that (1) the weight of each cell is at most $W_+ = \lfloor (1 + \epsilon)\lceil W/2 \rceil \rfloor$ (we say that such a partition is ϵ-balanced) and (2) the total cost of all edges between cells (*cut size*) is minimized. Conversely,

$W_- = W - W_+$ is the *minimum allowed cell size*. If $\epsilon = 0$, we say the partition is *perfectly balanced* (or just *balanced*). Formally we have the following problem.

Input Graph $G = (V, E)$, weights $w(v)$, costs $c(e)$ and imbalance allowed $\epsilon \geq 0$

Output ϵ-balanced bisection (A, B) of V of minimal cost

To simplify exposition, we will describe our algorithms assuming that all edges have unit costs. Small integral edge costs can be dealt with by creating parallel unit edges; Section 3.8 shows how we can use a scaling technique to handle arbitrary edge costs.

A standard technique for finding exact solutions to NP-hard problems is *branch-and-bound* [47,73]. It performs an implicit enumeration of all possible solutions by dividing the solution space of the original problem into two or more "simpler" parts (subproblems,) solving them recursively, and picking the best solution found. Each node of the branch-and-bound tree corresponds to a distinct subproblem. In a minimization context, the algorithm keeps a global *upper bound* U on the solution of the original problem; this bound is updated whenever better solutions are found. To process a node in the tree, we first compute a *lower bound* L on any solution to the corresponding subproblem. If $L \geq U$, we *prune* the node: it cannot lead to a better solution. Otherwise, we *branch*, creating two or more simpler subproblems.

In the case of graph bisection, each node of the branch-and-bound tree corresponds to a *partial assignment* (A, B), where $A, B \subseteq V$ and $A \cap B = \emptyset$. We say the vertices in A or B are *assigned*, and all others are *free* (or *unassigned*). This node implicitly represents all valid bisections (A^+, B^+) that are *extensions* of (A, B), i.e., such that $A \subseteq A^+$ and $B \subseteq B^+$. In particular, the *root* node, which represents all

valid bisections, has the form $(A, B) = (\{v\}, \emptyset)$. Note that we can fix an arbitrary vertex v to one cell to break symmetry.

To process an arbitrary node (A, B), we must compute a lower bound $L(A, B)$ on the value of any extension (A^+, B^+) of (A, B). The fastest exact algorithms [5,7, 16,41,52,66] usually apply mathematical programming techniques to find lower bounds. In this article, we use only combinatorial bounds that can be computed efficiently. In particular, our basic algorithm starts from the well-known [18,31] *flow bound*: the minimum A–B cut, taking the edge costs as capacities. This is a valid lower bound because any extension (A^+, B^+) must separate A from B; moreover, it can be computed rather quickly in practice [48]. If the minimum cut happens to be balanced, we can prune (and update U, if applicable). Otherwise, we choose a free vertex v and branch on it, generating subproblems $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$.

Note that the flow lower bound can only work well when A and B are large enough. In particular, if either set is empty, the flow bound is zero. Even when A and B have approximately the same size, the corresponding minimum cut is often far from balanced, with one side containing many more vertices than the other. This makes the flow bound rather weak by itself. To overcome these issues, we introduce a new *packing lower bound*, which we describe next.

3.3 Packing Bound

The *packing bound* is a novel lower-bounding technique that takes into account the fact that the optimal solution must be balanced. Consider a partial assignment (A, B). Let f be the value of the maximum A–B flow, and G_f be the graph

obtained by removing all flow edges from G (recall that we assume all edges have unit costs/capacities). Without loss of generality, assume that A is the *main side*, i.e., that the set of vertices reachable from A in G_f has higher total weight than those reachable from B (these two sets are disjoint since we removed edges from a A–B cut). We will compute our new bound on G_f, since this allows us to simply add it to f to obtain a unified lower bound. (A detailed example illustrating the concepts introduced in this section can be found in Appendix B.1.) The following structure is central in our algorithm.

Definition 2. *A* tree packing $\mathcal{T}$ *of* G_f *where* A *is the main side is a collection of trees such that: (1) the trees are edge-disjoint; (2) each tree contains exactly one edge incident to* A*; and (3) no edge can be added to* $\mathcal{T}$ *without violating the previous properties. See Figure B.1b (in Appendix C) for an example. Given a set* $S \subseteq V$*, let* $\mathcal{T}(S)$ *be the subset of* $\mathcal{T}$ *consisting of all trees that contain a vertex in* S*. Let* $\mathcal{T}(v) = \mathcal{T}(\{v\})$*.*

For now, assume a tree packing $\mathcal{T}$ is given; Section 3.4 will show how one can be built.

Lemma 4. *If* B^+ *is an extension of* B*, then* $f + |\mathcal{T}(B^+)|$ *is a lower bound on the cost of the corresponding bisection* $(V \setminus B^+, B^+)$*.*

Proof. *By definition, a tree* $T_i \in \mathcal{T}$ *contains a path from each of its vertices to* A*; if a vertex in* B^+ *is in* T_i*, at least one edge from* T_i *must be cut in* (A^+, B^+)*. Noting that each tree* $T_i \in \mathcal{T}(B^+)$ *contains an edge-disjoint path from* A *to* B^+ *in* G_f *completes the proof.* □

Lemma 4 applies to a fixed extension B^+ of B; we need a lower bound that applies to *all* (exponentially many) possible extensions. We must therefore reason about a *worst-case extension* B^*, i.e., an ϵ-balanced partition $(V \setminus B^*, B^*)$ with $B^* \supseteq B$ that minimizes the bound given by Lemma 4.

First, note that $w(B^*) \geq W_-$, since $(V \setminus B^*, B^*)$ must be a valid bisection.

Second, let $D_f \subseteq V$ be the set of all vertices that are *unreachable* from A in G_f (in particular, $B \subseteq D_f$). Without loss of generality, for the purposes of the lower bound we can assume that B^* contains D_f. We can do so because in Lemma 4 any vertex $v \in D_f$ is *deadweight*: it contributes to the weight of B^* (since v itself has nonnegative weight) but does not increase the lower bound (there is no path from v to A). Including D_f in B^* thus helps $w(B^*)$ reach W_- at no cost.

To reason about other vertices in B^*, we first establish a relationship between $\mathcal{T}$ and vertex weights by predefining a *vertex allocation*, a mapping from vertices to trees. We allocate each reachable free vertex v (i.e., $v \in V \setminus (D_f \cup A)$) to a tree $T(v) \in \mathcal{T}(v)$, as shown in Figure B.1c, in the appendix. (Section 3.4 will discuss how to compute such an allocation.) The *weight* $w(T_i)$ of a tree $T_i \in \mathcal{T}$ is the sum of the weights of all vertices allocated to T_i. Let $\mathcal{T}'(S) = \{T(v) : v \in S\}$ be the set of trees such that at least one vertex of S is allocated to it. Since $|\mathcal{T}'(S)| \leq |\mathcal{T}(S)|$ we have that $f + |\mathcal{T}'(B^+)|$ is a valid lower bound for the cost of an extension (A^+, B^+) of (A, B).

Note that if $v \in B^*$ we can add vertices w such that $T(w) = T(v)$ without improving the lower bound given by Lemma 4. Therefore we can assume without loss of generality (for the purposes of the lower bound) that if B^* contains a single vertex allocated to a tree T_i it will contain *all* vertices allocated to T_i (if all fit).

Moreover, to ensure that $w(B^*) \geq W_-$, B^* must contain a *feasible* set of trees $\mathcal{T}' \subseteq \mathcal{T}$, i.e., a set whose total weight $w(\mathcal{T}')$ (defined as $\sum_{T_i \in \mathcal{T}'} w(T_i)$) is at least as high as the *target weight* $W_f = W_- - w(D_f)$. Since B^* is the worst-case extension,

it must correspond to a feasible set $\mathcal{T}'$ of minimum cardinality.

Definition 3. *Given a partial assignment* (A, B)*, a flow* f*, a tree packing* $\mathcal{T}$*, and an associated vertex allocation, we define the* packing bound *as* $p(\mathcal{T}) = \min_{\mathcal{T}' \subseteq \mathcal{T}, w(\mathcal{T}') \geq W_f} |\mathcal{T}'|$.

Note that the exact value of this bound can be computed by a *greedy algorithm*: it suffices to pick trees in decreasing order of weight until their accumulated weight is at least W_f. The bound is the number of trees picked.

We can strengthen this bound further by allowing *fractional allocations*. Instead of allocating v's weight to a single tree, we can distribute $w(v)$ arbitrarily among all trees in $\mathcal{T}(v)$. For v's allocation to be *valid*, each tree must receive a nonnegative fraction of v's weight, and these fractions must add up to one. (Figure B.1d, in the appendix, has an example; Section 3.4 will discuss how such an allocation can be found.) The weight of a tree T is defined in the natural way, as the sum of all fractional weights allocated to T. By making trees more balanced, fractional allocations can improve the packing bound and are particularly useful when the average number of vertices per tree is small, or when some vertices have high degree. The fact that the packing bound is valid with fractional allocations is our first important theoretical result.

Theorem 4. *Consider a partial assignment* (A, B)*, a flow* f*, a tree packing* $\mathcal{T}$*, and a valid fractional allocation of weights. Then* $f + p(\mathcal{T})$ *is a lower bound on the cost of any valid extension of* (A, B)*.*

Proof. Let (A^*, B^*) be a minimum-cost extension of (A, B). Let $\mathcal{T}^* = \mathcal{T}(B^*)$ be the set of trees in $\mathcal{T}$ that contain vertices in B^*. The cut size of (A^*, B^*) must be at least $f + |\mathcal{T}^*|$ by Lemma 4. To prove our result we show that $p(\mathcal{T}) \leq |\mathcal{T}^*|$.

It suffices to show that $w(\mathcal{T}^*) \geq W_f$ as this ensures that $\mathcal{T}^*$ is one of the trees considered by the packing bound. Let $R^* = B^* \setminus D_f$ be the set of vertices in B^* that are reachable from A in G_f. Clearly, $w(R^*) = w(B^* \setminus D_f) \geq w(B^*) - w(D_f)$.

Moreover, $w(\mathcal{T}^*) \geq w(R^*)$ must hold because (1) every vertex $v \in R^*$ must hit some tree in $\mathcal{T}^*$ (the trees are maximal); (2) although $w(v)$ may be arbitrarily split among several trees in $\mathcal{T}(v)$, all these must be in $\mathcal{T}^*$ (by definition); and (3) vertices of $\mathcal{T}^*$ that are in A^* (and therefore not in R^*) can only contribute nonnegative weights to the trees. Finally, since B^* is a valid bisection, we must have $w(B^*) \geq W_-$. Putting everything together, we have $w(\mathcal{T}^*) \geq w(R^*) \geq w(B^*) - w(D_f) \geq W_- - w(D_f) = W_f$. This completes the proof. □

3.4 Bound Computation

Theorem 4 applies to any valid tree packing $\mathcal{T}$, but the quality of the bound it provides varies; intuitively, more balanced packings lead to better bounds. More precisely, we should pick $\mathcal{T}$ (and an associated weight allocation) so as to avoid heavy trees, which improves $p(\mathcal{T})$ by increasing the number of trees required to achieve the target weight W_f. For a fixed graph G_f, the number $|\mathcal{T}|$ of trees is the same for any tree packing $\mathcal{T}$, as is their total weight $w(\mathcal{T})$; therefore, in the ideal tree packing all trees would have the same weight.

Unfortunately, finding the best such tree packing is NP-hard. To prove this, we define a decision version of this problem as follows. Given a graph G with integral vertex weights and a partial assignment (A, B), decide whether there is a valid tree packing $\mathcal{T}$ rooted at A (with weight allocations) such that all trees have weight smaller than or equal an input parameter K. Tree Packing is NP-

Hard even when all nodes have weight 1.

Theorem 5. *Tree Packing is NP-Hard even when all nodes have weight* 1.

Proof. We prove it by reduction from 3-PARTITION, defined as follows: given $S = \{s_1, s_2, \ldots, s_p\}$ of p positive integers where $p = 3P$ decide whether it can be partitioned into P triplets, each summing to the same number. This problem is NP-Hard even if we bound s_i by a polynomial in P, in other words, 3-PARTITION is *strongly NP-Hard* [83].

Let $S = \{s_1, \ldots, s_p\}$ with $p = 3P$ and s_i bounded by a polynomial in P be an input to 3-PARTITION.

Let A be the maximum element in S and build $S' = \{s'_1, s'_2, \ldots, s'_p\}$ where $s'_i = (p+1)(s_i + pA)$. Note that since we only applied an affine transformation to each element of S, if a partition of S has the desired property (each part has 3 elements, and the sum is equal for all parts), the same partition of S' will also have this property. Note also that each element s'_i is still bounded by a polynomial on P. More precisely, $\sum_{i \in \mathcal{I}} s'_i = (p+1)\left[|\mathcal{I}|pA + \sum_{i \in \mathcal{I}} s_i\right]$.

Let $S_1, \ldots, S_P$ be a partition of S' with all parts summing to the same number. If S_i and S_j do not have the same number of elements their sums cannot be the same since pA is an upper bound on the sum of all numbers. So even without enforcing it directly we know that if a partition $S_1, \ldots, S_P$ of S' exists such that the sum of the elements in each part is the same, each part will have exactly three elements, and will therefore be a solution to 3-PARTITION.

Let $H = (V, E)$ where for each $s'_i \in S'$ there are vertices w_i and v_i^j for $j = 1, \ldots, s'_i$, and edges between v_i^j to v_i^{j+1} and between w_i and v_i^1. There are also P

parallel edges between w_i and w_{i+1}. Finally, create an extra node u and connect it with P parallel edges to w_1.

Create an instance of Tree Packing with the graph H, $A = \{u\}$, $B = \emptyset$, and $K = \frac{p+\sum_i s'_i}{P}$. ($K$ is the sum of all weights divided by P.) Note that any feasible solution to this instance will have P trees, and that the "P parallel paths" from u to w_k will be assigned to different trees. Also, if we assign a certain edge (w_i, v_i^1) to tree T we should assign all edges (v_i^j, v_i^{j+1}) to T, and it will be the only tree reaching nodes v_i^j.

If there is a solution with all trees with equal size, each tree should get exactly 3 edges of the form (w_i, v_i^1) as explained above. Each nodes of the form v_i^j will have exactly one tree containing it, so all its weight should go to this tree, adding weight s'_i to the tree that contains (w_i, v_i^1). Note that all s'_i are multiples of $p+1 \geq 4$, so the residue modulo 4 of the weight of a tree is exactly the amount of weight it receives from nodes w_i. Since all trees should have the same weight we have that each one should receive total weight exactly 3 from the nodes w_i. A solution to 3-PARTITION follows trivially.

It is easy to see, using the same arguments, that if a solution to this Tree Packing instance does not exist then the original 3-PARTITION instance is also infeasible. □

Given this result, in practice we resort to (fast) heuristics to find a valid tree packing. Ideally, the trees and weight allocations should be computed simultaneously, to account for the interplay between them. Since it is unclear how to do so efficiently, we use a two-stage approach instead: we first compute a valid tree packing, then allocate vertex weights to these trees appropriately. We

discuss each stage in turn.

3.4.1 Generating Trees

The goal of the first stage is to generate maximal edge-disjoint trees rooted at A that are as balanced and intertwined as possible, since this typically enables a more even distribution of vertex weights. We try to achieve this by growing these trees simultaneously, balancing their sizes (number of edges).

More precisely, each tree starts with a single edge (the one adjacent to A) and is marked *active*. In each step, we pick an active tree with minimum size (number of edges) and try to expand it by one edge in DFS fashion. A tree that cannot be expanded is marked as inactive. We stop when there are no active trees left. We call this algorithm *SDFS* (for *simultaneous depth-first search*).

An efficient implementation of SDFS requires a careful choice of data structures. In particular, a standard DFS implementation associates information (such as parent pointers and status within the search) with vertices, which are the entities added to and removed from the DFS stack. In our setting, however, the same vertex may be in several trees (and stacks) simultaneously, making an efficient implementation more challenging. We get around this by associating information with *edges* instead. Since each edge belongs to at most one tree, it has at most one parent and is contained in at most one stack. The combined size of all data structures we need is therefore $O(m)$.

Given this representation, we now describe our SDFS algorithm in more detail. We associate with each tree T_i a stack S_i, initially containing the root edge of

T_i. (Each non-flow edge with exactly one endpoint in A becomes the root edge of some T_i; edges with both endpoints in A belong to no tree.) The basic step of the SDFS algorithm is as follows. First, pick an active tree T_i of minimum size. (This can be done efficiently by maintaining the current active trees in *buckets* according to their current number of edges.) Let (u, v) be the edge on top of S_i (the stack associated with T_i), and assume v is farther from T_i's root than u is (i.e., u is v's parent). Scan vertex v, looking for an *expansion edge*. This is an edge (v, w) such that (1) (v, w) is free (not assigned to any tree yet) and (2) no edge incident to w belongs to T_i. The first condition ensures that the final trees are edge-disjoint, while the second makes sure they have no cycles. If no such expansion edge exists, we backtrack by popping (u, v) from S_i; if S_i becomes empty, T_i can no longer grow, so we mark it as inactive. If expansion edges do exist, we pick one such edge (v, w), push it onto S_i, and add it to T_i by setting $parent(v, w) \leftarrow (u, v)$. The algorithm repeats the basic step until there are no more active trees.

We must still define which expansion edge (v, w) to select when processing (u, v). We prefer an edge (v, w) such that w has several free incident edges (to help keep the tree growing) and is as far as possible from A (to minimize congestion around the roots, which is also why we do DFS). We use a preprocessing step to compute the distances from A to all vertices with a single breadth-first search (BFS).

To bound the running time of SDFS, note that a vertex v of degree $\deg(v)$ can be scanned $O(\deg(v))$ times, since each scan either eliminates a free edge or backtracks. When scanning v, we can process each outgoing edge (v, w) in $O(1)$ time using a hash table to determine whether w is already incident to v's tree.

The worst-case time is therefore $\sum_{v \in V}(\deg(v))^2 = O(m\Delta)$, where Δ is the maximum degree.

3.4.2 Weight Allocation

Once a tree packing $\mathcal{T}$ is built, we must allocate the weight of each vertex v to the trees $\mathcal{T}(v)$ it is incident to. Our final goal is to have the weights as evenly distributed among the trees as possible. We work in two stages. First, an *initial allocation* splits the weight of each vertex evenly among all trees it is incident to. We then run a *local search* to rebalance the weight allocation among the trees. We process one vertex at a time (in arbitrary order) by reallocating v's weight among the trees in $\mathcal{T}(v)$ in a locally optimal way. More precisely, v is processed in two steps. First, we reset v's existing allocation by removing v's share from all trees it is currently allocated to, thus reducing their weights. We then distribute v's weight among the trees in $\mathcal{T}(v)$ (from lightest to heaviest), evening out their weights as much as possible. In other words, we add weight to the lightest tree until it is as heavy as the second lightest, then add weight to the first two trees (at the same rate) until each is as heavy as the third, and so on. We stop as soon as v's weight is fully allocated. The entire local search runs in $O(m \log \Delta)$ time, since it must sort (by weight) the adjacency lists of each vertex in the graph once. In practice, we run the local search three times to further refine the weight distribution; additional runs typically have little effect.

3.5 Forced Assignments

In this section we will show how to extract information from a tree packing other than the tree bound to further reduce the size of the branch-and-bound tree. Consider a partial assignment (A, B). The quality of the bounds we use depends crucially on the sum of the degrees of all vertices already fixed to A or B, since they bound both the flow value and the number of trees created. If we could fix more vertices to A or B without explicitly branching on them, we would boost the effectiveness of both bounds, reducing the size of the branch-and-bound tree. This section explains how we can do this by exploiting some properties of the tree packing itself. These *forced assignments* work particularly well when the current lower bound for (A, B) is close enough to the upper bound U. Since most nodes of the branch-and-bound tree have this property, this technique can reduce the total running time by orders of magnitude.

Our goal is to infer, without branching, that a certain free vertex v must be assigned to A (or B). Intuitively, if we show that assigning v to one side would increase the lower bound to at least match the upper bound, we can safely assign v to the other side. Sections 3.5.1–3.5.3 propose specific versions of such forced assignments. They all require computing the packing bound for a slightly different set of trees; Section 3.5.4 shows how this can be done quickly, without a full recomputation. Without loss of generality, we once again assume that the weight of the vertices reachable from A in G_f is at least as high as the weight of those reachable from B (i.e., that A is the main side). In what follows, let $\mathcal{T}$ be a tree packing with weight allocations and $f + p(\mathcal{T})$ be the current lower bound. All techniques we present are illustrated in Appendix B.2.

3.5.1 Flow-based Assignments

We first consider *flow-based forced assignments*. Let v be a free vertex reachable from A in G_f, and consider what would happen if it were assigned to B. The flow bound would immediately increase by $|\mathcal{T}(v)|$ units, since each tree in $\mathcal{T}(v)$ contains a disjoin path from v to A. We cannot, however, simply increase the overall lower bound to $f + p(\mathcal{T}) + |\mathcal{T}(v)|$, since the packing bound may already be "using" some trees in $\mathcal{T}(v)$. Instead, we must compute a new packing bound $p(\mathcal{T}')$, where $\mathcal{T}' = \mathcal{T} \setminus \mathcal{T}(v)$ but the weights originally assigned to the trees $\mathcal{T}(v)$ are treated as deadweight (unreachable). If the updated bound $f + p(\mathcal{T}') + |\mathcal{T}(v)|$ is U or higher, we have proven that no solution that extends $(A, B \cup \{v\})$ can improve the best known solution. Therefore, we can safely assign v to A.

We can make a symmetric argument for vertices w that are reachable from B in G_f, as long as we also compute an edge packing $\mathcal{T}_B$ on B's side. If we assigned such a vertex w to A, the overall flow would increase by $|\mathcal{T}_B(w)|$. Since the extra flow is on B's side, the original packing bound $p(\mathcal{T})$ (which uses only edges reachable from A) is still valid. We can obtain a slightly better bound $p'(\mathcal{T})$, however, by using the fact that vertex w itself can no longer be considered deadweight (i.e., assumed to be on B's side), since we are explicitly assigning it (tentatively) to A. If the new bound $f + |\mathcal{T}_B(w)| + p'(\mathcal{T})$ is U or higher, we can safely assign w to B.

3.5.2 Extended Flow-based Assignments

As described, flow-based forced assignments are weaker than they could be. When we take a vertex reachable from A in G_f and tentatively assign it to B, we

argue that each tree containing v results in a new unit of flow, since following parent pointers within the tree leads to a vertex in A. This means that, even though a tree may have several edges incident to v, we only "send" new flow through the parent edge. This section shows how the bound can be strengthened by considering child edges as well. This is not trivial, however. Each child edge is the root of a different subtree, but no such subtree contains a vertex in A (by construction), so it cannot improve the flow bound by itself. To obtain new paths to A, we must use other trees as well.

For a precise description of the algorithm, we need some additional notation. For every edge e, let $tree(e)$ be the tree (from $\mathcal{T}$) to which e is assigned. Each edge $e = (v, u)$ is either a *parent* or a *child* edge of v, depending on whether u is on the path from v to the root of $tree(e)$ or not. Define $path(e)$ as the path (including e itself) from e to the root of $tree(e)$ following only parent edges. Moreover, let $sub(e)$ be the subtree of $tree(e)$ rooted at e. (Note that $tree(e)$, $path(e)$, and $sub(e)$ are undefined if e belongs to no tree in $\mathcal{T}$.) Let $\sigma(e)$ be the set of all trees $t \in \mathcal{T} \setminus \{tree(e)\}$ that intersect $sub(e)$, i.e., that have at least one vertex in common with $sub(e)$.

Given a vertex v reachable from A and an incident edge e, we can define an associated *expansion tree* $x_v(e)$, which is equal to $tree(e)$ if e is a parent edge, and equal to any tree in $\sigma(e)$ if e is a child edge (if e belongs to no tree or if $\sigma(e) = \emptyset$, $x_v(e)$ is undefined). The *expansion set* for v, denoted by $X(v)$, is the union of $x_v(e)$ over all edges e incident to v. Note that $X(v) \subseteq \mathcal{T}$ is a set of trees with the following useful property.

Lemma 5. *Assigning v to B would increase the flow bound by at least $|X(v)|$ units.*

Proof. We must show that, for each tree $T \in X(v)$, we can create an independent

path from v to A in G_f. If T is the expansion tree for a parent edge e of v, we simply take $path(e)$. Otherwise, if T is the expansion tree for a child edge e of v, we take the concatenation of two paths, one within $sub(e)$ and another within T itself. By construction (of $X(v)$), these paths are all disjoint. □

Note that we used a similar argument in Section 3.5.1 to justify the standard flow-based assignment; the difference here is that we (implicitly) send flow through potentially more trees. Accordingly, as long as we consider all affected trees as deadweight, a valid (updated) packing bound for $(A, B \cup \{v\})$ is given by $f + |X(v)| + p(\mathcal{T} \setminus X(v))$. If this is at least U, we can safely assign v to A.

A similar argument can be made if v is initially reachable from B in G_f. A valid lower bound for $(A \cup \{v\}, B)$ is $f + |X(v)| + p'(\mathcal{T})$, where $p'(\mathcal{T})$ is the standard packing bound, but with v no longer considered as deadweight.

Practical Issues

Although extended flow-based forced assignments are conceptually simple, they require access to several pieces of information associated with each edge. We now explain how they can be computed efficiently.

We can compute $\sigma(\cdot)$ (as defined above) for all edges in a tree T by traversing the tree in bottom-up fashion, as a "preprocessing" step before actually trying the extended forced assignments. Let $e = (v, w) \in T$ be the parent edge of some vertex v. We can compute $\sigma(e)$ as the union of the $\sigma(f)$ values for all child edges f of e in T, together with all other trees incident to v itself. More precisely, $\sigma(e) = (\cup_{f=(u,v)\in(T\setminus\{e\})}\sigma(f)) \cup (\mathcal{T}(v) \setminus \{T\})$. Note, however, that each set $\sigma(\cdot)$ can have $\Theta(|\mathcal{T}|) = O(m)$ trees, so maintaining all such sets in memory during the algo-

rithm would be impractical. In practice, for each edge e we keep only a subset $\tilde{\sigma}(e) \subseteq \sigma(e)$ (picked uniformly at random) of size at most κ, an input parameter; $\kappa = 3$ works well in practice. To process an edge e during the forced assignment routine, we pick a random unused element from $\tilde{\sigma}(e)$ as the expansion tree $x_v(e)$ of e. (We could maximize $|X(v)|$ by computing $x_v(e)$ for all edges e incident to v simultaneously using a maximum matching algorithm, but this is expensive.) Although using $\tilde{\sigma}(e)$ instead of $\sigma(e)$ may make the algorithm slightly less effective (we may run out of unused elements in $\tilde{\sigma}(e)$ even though some would be available in $\sigma(e)$), it does not affect correctness.

We must be careful to compute the $\tilde{\sigma}(e)$ sets in a space-efficient way: we still need $\sigma(e)$ in order to compute $\tilde{\sigma}(e)$, which is a random subset. But $\sigma(e)$ is the union of up to $O(m)$ distinct $\sigma(e')$ values, one for each child edge $e' = (v, u)$ of v. Instead of first building all these child subsets and then computing their union, we build $\sigma(e)$ incrementally; as soon as each $\sigma(e')$ is computed, we determine $\tilde{\sigma}(e')$, then merge $\sigma(e')$ into $\sigma(e)$ (initially empty) and discard $\sigma(e')$. Traversing the tree in DFS post-order ensures that, at any time, all edges with nonempty $\sigma(\cdot)$ sets form a contiguous path in T. Moreover, if we make the DFS visit the largest subtree first, this path will have length $O(\log n)$. To see why, note that $\sigma(e)$ is empty while we visit the first (largest) child subtree of e; after that, each subtree is at most half as large as the subtree rooted at e itself. Since the maximum subtree size is $O(n)$, this situation cannot happen more than $O(\log n)$ times, limiting the total space used by the temporary $\sigma(\cdot)$ sets to $O(|\mathcal{T}| \log n) = O(m \log n)$. The worse-case time required by this precomputation step is also reasonable.

Lemma 6. *All $\tilde{\sigma}(\cdot)$ sets can be computed in $O(m\Delta \log n)$ total time, where Δ is the maximum degree in the graph.*

Proof. First, note that each vertex v is scanned $O(\deg(v))$ times (once for each tree it belongs to), thus bounding the total scanning time to $O(m\Delta)$. Maintaining the $\sigma(\cdot)$ sets during the execution is slightly more expensive. To bound the total time to process a single tree T, we note that each edge $e \notin T$ that is adjacent to T will create an entry in some set $\sigma(e)$, then "bubble up" the tree as sets are merged with parent sets. Eventually, each such entry will either be discarded (if the parent already has an entry corresponding to $tree(e)$) or will end up at the root. The total time (over all trees) spent on such original insertions and on deletions is $O(m\Delta)$, since any edge may participate in at most $2(\Delta - 1)$ original insertions (once for each tree incident to its endpoints). We still have to bound the time spent copying the elements of some (final) set $\sigma(e)$ to another (temporary) set $\sigma(p)$, where p is the parent edge of e. First, note that we only need to insert elements from a smaller set into a bigger one; otherwise, we just swap the entire sets (in constant time by manipulating pointers). So consider the case in we transfer elements from a set J into another set K (with $|J| \leq |K|$) while processing a tree T. Only elements in $J' = J \setminus K$ are actually inserted into K; the remaining are deleted (and the corresponding cost has already been accounted for). If $|J'| \leq |J|/2$, we can charge each insertion to a corresponding deletion in J. If $|J'| > |J|/2$, we have a *heavy transfer*. Note that $|J \cup K| = |J'| + |K| > |J|/2 + |J| = 3|J|/2$. Since the target set ($J \cup K$) is bigger than the original set (J) by a constant factor, each entry (set element) can be involved in at most $O(\log |T|)$ heavy transfers. Considering that there are $O(m\Delta)$ entries overall (across all trees) and $|T| = O(n)$ for any tree T, the lemma follows. □

3.5.3 Subdivision-based Assignments

A third strategy we use is the *subdivision-based forced assignment*, which works by implicitly subdividing heavy trees in $\mathcal{T}$. Let v be a free vertex reachable from A in G_f. If v were assigned to A, we could obtain a new tree packing $\mathcal{T}'$ by splitting each tree $T_i \in \mathcal{T}(v)$ into multiple trees, one for each edge of T_i that is incident to v. If $f + p(\mathcal{T}') \geq U$, we can safely assign v to B.

Some care is required to implement this test efficiently. In particular, to recompute the packing bound we need to compute the total weight allocated to each newly-created tree. To do so efficiently, we must precompute some information about the original packing $\mathcal{T}$. (This precomputation happens once for each branch-and-bound node, after $\mathcal{T}$ is known.) We define $size(e)$ as the weight of the subtree of $tree(e)$ rooted at e: this is the sum, over all vertices descending from e in $tree(e)$, of the (fractional) weights allocated to $tree(e)$. (If e belongs to no tree, $size(e)$ is undefined.) The $size(e)$ values can be computed with bottom-up traversals of all trees, which takes $O(m)$ total time.

These precomputed values are useful when the forced assignment routine processes a vertex v. Consider an edge e incident to v. If e is a child edge for v, it will generate a tree of size $size(e)$. If e is a parent edge for v, the size of the new tree will be $size(r(e)) - size(e)$, where $r(e)$ is the root edge of $tree(e)$.

3.5.4 Recomputing Bounds

Note that all three forced-assignment techniques we consider need to compute a new packing bound $p(\mathcal{T}')$ for each vertex v they process. Although they need

only $O(\deg(v))$ time to (implicitly) transform $\mathcal{T}$ into $\mathcal{T}'$, actually computing the packing bound from scratch can be costly. Our implementation uses an incremental algorithm instead. When determining the original $p(\mathcal{T})$ bound (with the greedy algorithm described in Section 3.3), we remember the entire state of the computation, including the sorted list of all original tree weights as well as relevant partial sums. To compute $p(\mathcal{T}')$, we can start from this initial state, discarding original trees that are no longer valid and considering new ones appropriately.

3.6 Decomposition

Both lower bounds we consider depend crucially on the degrees of the vertices already assigned. More precisely, let D_A and D_B be the sum of the degrees of all vertices already assigned to A and B, respectively, with $D_A \geq D_B$ (without loss of generality). It is easy to see that the flow bound cannot be larger than D_B, and the packing bound is usually bounded by roughly $D_A/2$, half the number of trees (unless a significant fraction of the vertices is already assigned).[1] If the maximum degree in the graph is a small constant (which is often the case on meshes, VLSI instances, and road networks, for example), our branch-and-bound algorithm cannot prune anything until deep in the tree. Arguably, the dependency on degrees should not be so strong. The fact that increasing the degrees of only a few vertices could make a large instance substantially easier to solve is counter-intuitive.

[1] For an intuition on the $D_A/2$ bound, recall that the packing bound must accumulate enough trees to account for half the total weight reachable from A. Even if all D_A trees have exactly the same weight, roughly half of the trees will be enough for this. This is not a strict bound because it the number of trees needed also depends on how many vertices are already assigned to A and B.

A natural approach to deal with this weakness is branching on entire regions (connected subgraphs) at once. We would like to pick a region and add *all* of its vertices to A in one branch, and all to B in the other. Since the "degree" of the region (i.e., the number of outgoing edges) is substantially larger, lower bounds should increase much faster as we go down the branch-and-bound tree. The obvious problem with this approach is that the optimal bisection may actually split the region itself; assigning the entire region to A or to B does not exhaust all possibilities.

One could overcome this by making the algorithm probabilistic: if we contract a small number of random edges (merging both endpoints of each edge into a single, heavier vertex), with reasonable probability none of them will actually be cut in the minimum bisection. If this is the case, the optimum solution to the contracted problem is also the optimum solution to the original graph. We can boost the probability of success by repeating this procedure multiple times (with multiple randomly selected contracted sets) and picking the best result found. Such probabilistic contraction is a natural approach for cut problems, having been used in Karger and Stein's randomized global minimum-cut algorithm [65], as well as in a graph-bisection algorithm by Bui et al. [18] that runs in expected polynomial time on a certain class of d-regular graphs with small enough bisections.

Since our goal is to find provably optimum bisections, however, probabilistic solutions are inadequate. Instead, we propose a contraction-based *decomposition algorithm*, which is *guaranteed* to output the optimum solution for *any input*. It is (of course) still exponential in the worst case, but for many inputs it has much better performance than our standard branch-and-bound algorithm.

The algorithm works as follows. Let U be a known upper bound on the optimum bisection. First, we partition the set E of edges into $U + 1$ disjoint sets $(E_0, E_1, \ldots, E_U)$. For each subset E_i, we create a corresponding (weighted) graph G_i by first taking all the edges of the input graph G, then contracting those in E_i. (We contract an edge (u, v) by combining u and v into a single vertex with weight $w(u) + w(v)$, redirecting edges incident to u or v to the new vertex, and discarding self-loops.) Note that no edge of E_i actually belongs to G_i. Then, we use our standard algorithm to find the optimum bisection U_i of each graph G_i independently, and return the best (lowest-value) such solution.

Theorem 6. *The decomposition algorithm finds the minimum bisection of G.*

Proof. Let $U^* \leq U$ be the cost of the minimum bisection. We have to prove that $\min_{0 \leq i \leq U}(U_i) = U^*$. First, note that $U_i \geq U^*$ for every i, since any bisection of G_i can be trivially converted into a valid bisection of G with the same value. Moreover, we argue that the solution of at least one G_i must correspond to the optimum solution of G itself. Let E^* be the set of cut edges in an optimum bisection of G. (If there is more than one optimum bisection, take one arbitrarily.) Because $|E^*| = U^*$ and the E_i sets are disjoint, $E^* \cap E_i$ can only be nonempty for at most U^* sets E_i. Therefore, there is at least one j such that $E^* \cap E_j = \emptyset$. Contracting the edges in E_j preserves the optimum bisection, proving our claim. □

The decomposition algorithm solves $U + 1$ subproblems, but the high-degree vertices introduced by the contraction routine should make each subproblem much easier for our branch-and-bound routine. Besides, the subproblems are not completely independent: they can all share the same best upper bound. In fact, we can think of the algorithm as traversing a single branch-and-bound tree whose root node has $U + 1$ children, each responsible for a distinct contraction

pattern. The subproblems are not necessarily disjoint (the same partial assignment may be reached in different branches), but this does not affect correctness.

Finally, we note that solving $U + 1$ subproblems is necessary if we actually want to find a solution of value U if one exists. If we already know a solution with U edges and just want to prove it is optimal, it suffices to solve U subproblems: at least one of them will preserve a solution with fewer than U edges, if it exists. In particular, if we start running the algorithm with upper bound U and find an improving solution $U' < U$, we can stop after the first U' subproblems are solved. (To see why U' is the optimum bisection value in this case, apply Theorem 6 with U' sets of edges; even though these sets are not a partition of E, they are disjoint, which is all the theorem requires.) For the remainder of this section, we assume that only U initial subproblems are generated.

3.6.1 Finding a Decomposition

Our decomposition algorithm is correct regardless of how edges are partitioned among subproblems, but its performance may vary significantly. To make all subproblems have comparable degree distribution of difficulty, a natural approach is to allocate roughly the same number of edges to each subproblem. Moreover, the choice of *which* edges to allocate to each subproblem G_i also matters. The effect on the branch-and-bound algorithm is more pronounced if we can create vertices with much higher degree than in the original graph. We can achieve this by making sure the edges assigned to E_i induce relatively large connected components (or *clumps*) in G. (In contrast, if all edges in E_i are far apart, the degrees of the contracted vertices in G_i will not be much higher that those

of other vertices.) Finally, the shape of each clump matters: among clumps of the same size, we would like the expansion (number of incident edges) of each clump to be as large as possible.

To achieve these goals, we perform the decomposition in two stages: *clump generation* partitions all the edges in the graph into clumps, while *allocation* assigns the clumps to subproblems so as to balance the effort required to solve each subproblem. We discuss each stage separately.

See Figure C.2 for an example of decomposition and Figure C.3 for an example of its effect in bounds.

Clump Generation

Our clump generation routine must build a set F of clumps (initially empty) that partition all the edges in the graph. We combine two approaches to accomplish this. The *basic* clump generation routine is based on extracting paths from random BFS trees within the graph. This approach works reasonably well, and can be used to find a complete set of clumps, i.e., one that covers all edges in the graph. Our second approach, *flow-based* clump generation, is more focused: its aim is to find a small number (comparable to U) of good-quality clumps. We discuss each approach in turn, then explain how they can be combined into a robust clump-generation scheme.

Basic clump generation. Our basic clump generation routine maintains a set C of candidate clumps, consisting of high-expansion paths that are not necessarily disjoint and may not include all edges in the graph. The algorithm adds

new clumps to C until there are enough candidates, then transfers some of the clumps from C to F. This process is repeated until F is complete (its clumps contain all edges in the graph).

More precisely, our algorithm works in iterations, each consisting of two phases: generation and selection. Consider iteration i, with a certain *expansion threshold* τ_i (which decreases as the algorithm progresses).

The *generation phase* creates new clumps to be added to the candidate set C. Our clumps are paths obtained by following parent pointers in BFS trees, which tend to have relatively high expansion because the subgraph of G induced by such a path is itself a path. (Note that this is not true for arbitrary non-BFS paths.) More precisely, we pick 5 vertices at random and grow BFS trees from them, breaking ties in favor of parents that lead to paths with higher expansion. From each tree T, we greedily extract a set of edge-disjoint paths to add to C (in decreasing order of expansion). When doing so, we restrict ourselves to paths that (1) have at most $\lceil m/(4U) \rceil$ edges (ensuring each subproblem has at least four clumps), (2) contain no edge that is already in F, and (3) have expansion at least $\tau_i/4$ (avoiding very small clumps to save edges for future iterations).

The *selection phase* extracts from C all clumps with expansion at least τ_i in decreasing order of expansion. A clump c is added to F if no edge in c is already in F; otherwise, it is simply discarded. To save space, we also discard from C clumps that have at least one edge that either (1) belongs to F or (2) appears in at least five other clumps with higher expansion.

For the next iteration, we set $\tau_{i+1} = \lfloor 0.9\tau_i \rfloor$. We stop when the threshold reaches 0, at which point we simply add all remaining unused edges to F as

separate clumps. For speed, we grow smaller trees as the algorithm progresses, and only pick as roots vertices adjacent to at least one unused edge.

Flow-based clumps. We now consider an alternative clump generation scheme aimed at finding a small number of good-quality clumps. It is based on the observation that clumps containing a cut edge from the optimum bisection tend to lead to large packing bounds. To understand why, consider the alternative: if a clump is entirely contained within one side of the optimum bisection, the associated packing bound cannot be higher than *opt* (the optimum bisection value), since at most *opt* disjoint trees will reach the other side. This indicates that to obtain large packing bounds we should prefer clumps that actually cross the optimum bisection. Although the basic scheme described above can create such clumps, it does not actively look for them. We now describe an alternative that does; we call it the *flow-based* clump generation scheme.

It works in three stages. First, we quickly find an approximate minimum bisection, i.e., a small cut that roughly divides the graph in half. (The actual optimal solution would be ideal.) Second, we define two regions that are far from this cut, one on each side. Finally, we compute the maximum flow between these two regions, and use the paths in the corresponding flow decomposition as clumps. The remainder of this section describes how we implement each of these steps; other implementations are possible.

To find a quick approximate bisection, we first compute the *Voronoi diagram* with respect to two random vertices v_0 and v_1. In other words, we split V into two sets, R_0 and R_1, such that each element in R_i is closer to v_i than to v_{1-i}; this can be done with a single BFS [81]. Our tentative cut is given by the set S of

boundary vertices in the Voronoi diagram, i.e., those with a neighbor in another region. Although this approach is very fast, we may be unlucky in our choice of v_0 and v_1 and end up with a very unbalanced cut, with $w(R_0) \ll w(R_1)$. (Without loss of generality, assume that $w(R_0) \leq w(R_1)$.) To avoid this we try U random pairs (v_0, v_1), where U is the upper bound on the solution value. We then pick the cut with the highest *score* $s = r^2/b$, where $r = w(R_0)/w(R_1)$ and b is the number of boundary edges in the Voronoi diagram. Intuitively, the score measures how close we are to the optimal bisection: we want both regions to have roughly the same size and, among those, prefer smaller cuts.

After picking the highest-scored pair (R_0, R_1), we define subsets $Q_0 \subseteq R_0$ and $Q_1 \subseteq R_1$ to act as source and sink during our flow computation. We do so in a natural way. For a given parameter $0 < \alpha < 1$ and for each $i \in \{0, 1\}$, we run a BFS restricted to R_i starting from $S \cap R_i$ and stopping after the total weight of all scanned vertices is about to exceed $\alpha \cdot w(R_i)$; we take the vertices that remained in the BFS queue as Q_i.

Initially, we pick α such that the distance from Q_0 to Q_1 is closest to $\lceil m/(4U) \rceil$, the target clump size. (To avoid pathological cases, we also enforce that $\alpha \leq 0.8$.) Once the initial flow is computed, we remove the edges in the corresponding flow from the graph and repeat the computation using the same Q_0 and Q_1, but with $\alpha' \leftarrow 0.8\alpha$. We stop this process when α falls below 0.2. This choice of parameters is somewhat arbitrary; the algorithm is not very sensitive to them.

Final approach. As already observed, when the sets Q_0 and Q_1 happen to be in opposite sides of the minimum bisection, the flow-based approach cannot create more than U clumps. In practice, the actual number of clumps is indeed close

to U. We take these as the initial clumps in our set F, then apply the BFS-based clump generation scheme to obtain the remaining clumps. The initial threshold τ_0 for the basic clump generation scheme is set to the maximum expansion among all initial flow-based clumps.

Clump Allocation

Once clumps are generated, we run an *allocation phase* to distribute them among the U subproblems $(E_0, E_1, \ldots, E_{U-1})$, which are initially empty. We allocate clumps one at a time, in decreasing order of expansion (high-expansion clumps are allocated first). In each step, a clump c is assigned to the set E_i whose distance to c is maximum (with random perturbations to handle approximate ties). The distance from E_i to c is defined as the distance between their vertex sets, or infinity if E_i is empty. For efficiency, we keep the Voronoi diagram associated with each E_i explicitly and update it whenever a new clump is added.

This unbiased distribution tends to allocate comparable number of edges to each subproblem. In some cases, however, this is not desirable: if a subproblem E_i already has much better clumps than E_j, it pays to add more clumps to E_j. To accomplish this, we propose a *biased distribution*. It works as before, but we associate with each subproblem an extra *bias* parameter, to be multiplied by the standard distance parameter. The bias depends on the quality of the first clump added to each subproblem; we refer to it as the *anchor clump*. The bias makes subproblems with worse anchor clumps more likely to receive additional clumps.

To maximize the effectiveness of this approach, we take special care when

selecting the set of anchor clumps. We first compute the packing bound associated with the $\lceil 4U/3 \rceil$ clumps in F with highest expansion, then pick the U clumps with highest (packing) value as the anchors of the U subproblems. We define the *gap* associated with subproblem i as $g(i) = U - \phi(i)$, where $\phi(i)$ is the packing bound associated with i's anchor clump, computed by assigning the clump to A and making B empty in the partial assignment. Note that smaller gaps indicate better anchor clumps, and therefore should result in smaller biases. Simply making the bias proportional to the gap is too aggressive for some instances, resulting in very uneven distributions. Instead, we define $bias(i)$ as zero if $g(i) \leq 0$ (the anchor clump by itself is enough to prune the branch-and-bound tree at the root), and as $bias(i) = 1 + \log_2(U - \phi(i))$ otherwise. The log factor makes the distribution smoother, and the additive term ensures the expression is strictly positive.

3.7 The Algorithm in Full

Having described the main ingredients of our approach, we are now ready to explain how they fit together within our branch-and-bound routine. We first present an overview of the entire algorithm, then proceed to explain the missing details.

3.7.1 Overview

We take as input the graph G to be bisected, the maximum imbalance ϵ allowed, and an upper bound U on the solution value. The algorithm returns the optimal

solution if its value is less than U; otherwise, it proves that U is a valid lower bound. (Section 3.7.4 explains how one can pick U if no good upper bound is known.)

We start by running a simple heuristic (detailed in Section 3.7.2) to decide whether to use the decomposition technique or not. If not, we simply call our core branch-and-bound routine (detailed below). Otherwise, we create a series of U subproblems (as described in Section 3.6.1), call the core branch-and-bound routine on each subproblem separately, and return the best solution found. As an optimization, if we find an improving solution U' for a subproblem, we use it as an upper bound for subsequent subproblems.

Our core branch-and-bound routine starts with an assignment $(A, B) = (\{v\}, \emptyset)$ at the root (Section 3.7.5 explains how the initial assigned vertex v is picked). It then traverses the branch-and-bound tree in DFS order (to keep space usage low). Each node (A, B) of the tree is processed in four steps.

Step 1 computes the flow bound f; the exact algorithm is explained in Section 3.7.3. If $f \geq U$, we prune (discard the node) and backtrack. If $f < U$ and the corresponding minimum cut (A', B') is balanced enough (i.e., if $w(A') \geq W_-$ and $w(B') \geq W_-$), we have found a better feasible solution; we thus update the best known upper bound ($U \leftarrow f$), remember the corresponding bisection, and prune (since the flow bound f matches the new upper bound U). Otherwise (if $f < U$ but the cut is not balanced), we remove the flow edges from G (creating G_f), and proceed to the second step.

Step 2 computes a tree packing $\mathcal{T}$ in G_f and the corresponding packing bound $p(\mathcal{T})$, as described in Section 3.4. If $f + p(\mathcal{T}) \geq U$, we prune. Otherwise,

we proceed to the third step.

Step 3 applies the forced assignment rules introduced in Section 3.5, which may result in some vertices being added to A or B. If an inconsistency is detected (if assigning the same vertex to either A or B would push the lower bound above U, or if either side becomes too heavy), we prune. Otherwise, we proceed to the fourth step.

Step 4 picks a vertex $v \in V \setminus \{A, B\}$ to branch on (using rules explained in Section 3.7.5) and creates two subproblems, $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$, which are added to the branch-and-bound tree.

Steps 2 and 3 have already been discussed in detail. The remainder of this section focuses on the other aspects of the algorithm: criteria for using decomposition, flow computation, upper bound updates, and branching rules.

3.7.2 Decomposition

The first step of our algorithm is to decide whether to use decomposition or not. Intuitively, one should use decomposition when the upper bound U on the solution value is significantly higher than the degrees of the first few vertices we branch on, since they would lead to a very deep branch-and-bound tree.

We implement this idea by first computing a best-case estimate of the depth of the branch-and-bound tree without decomposition. We use the assumption that branching on a vertex of degree d increases the overall packing bound by $d/2$ (the intuition for this was explained in Section 3.6: in good cases, roughly half of the d new trees will be selected by the greedy algorithm). We sort all ver-

tices in the graph in decreasing order of degree and count how many vertices x would be required for the accumulated degree to reach $2U$. If $x \leq \max\{5, \log_2 U\}$, we do not use decomposition, since there are potentially enough high-degree vertices to ensure we have a small branch-and-bound tree. Intuitively, decomposition would require at least one node in each of the U branch-and-bound trees, which is roughly equivalent to a single tree of height $\log_2 U$. Otherwise (if $x > \max\{5, \log_2 U\}$), we compute a rough estimate *cdeg* of the degree of all clumps that would be assigned to each subproblem, given by the average number of edges per subproblem (m/U) multiplied by the average degree of each vertex in the graph ($2m/n$). We only use decomposition if $cdeg \geq 2U$.

We stress that this is (once again) just a heuristic, and it is not hard to come up with instances for which it would make the wrong decision. That said, this heuristic is easy to compute and it works well enough for the large (and diverse) set of instances considered in our experiments, allowing us to use the same settings for all inputs we test, with no manual tuning. In applications in which a more robust approach is needed, one could simply run both versions of the algorithm (with and without decomposition) in parallel and stop after one of them finishes. This method is guaranteed to be within a factor of two of the best choice in the worst case. Also note that, in general, using decomposition is a safer choice: although it can slow down the entire algorithm by a factor of at most U (even when it is not effective), it can lead to exponential speedups for some instances.

3.7.3 Flow Computation

Although the flow-based lower bound is valid regardless of the actual edges in the flow, the packing bound is better if it has more edges to work with, since fewer vertices tend to be unreachable and trees are more intertwined. We therefore prefer maximum flows that use relatively few edges: instead of using the standard push-relabel approach [48], we prefer an augmenting-path algorithm that greedily sends flows along shortest paths. Specifically, we use the IBFS (*incremental breadth first search*) algorithm [50], which is sometimes slower than push-relabel by a small constant factor on the instances we test, but still faster than computing the packing bound. Note that we could minimize the total number of edges using a minimum-cost maximum flow algorithm, but this would be considerably slower.

3.7.4 Upper Bound

As already mentioned, we only update the best upper bound U when the minimum A–B cut happens to be balanced; we use no additional heuristics to find improved valid solutions. This approach works well enough in practice, at least if the initial upper bound U is relatively close to the actual solution. If the initial value is significantly higher, our pruning techniques are ineffective, and the DFS traversal of the branch-and-bound tree can go extremely deep. For robustness, we thus avoid using the upper bound provided by a pure heuristic, such as CHACO [56], METIS [67], or SCOTCH [20, 86]. Although they work well in general, they can be off by a large margin for some instances, severely compromising the performance of our algorithm.

When no good initial upper bound is known, we therefore simply call the branch-and-bound algorithm repeatedly with increasing values of U, and stop when the bound we prove is actually lower than the input bound. We use $U_1 = 1$ for the first call and $U_i = \lceil 1.05\, U_{i-1} \rceil$ for call $i > 1$. Since the algorithm has exponential dependence on U, solving the last problem (the only one where U_i is greater than the optimum) takes a significant fraction of the total time. The overhead of previous calls is a relatively modest constant factor.

That said, we use a couple of simple strategies that sometimes allow us to increase U_i faster between iterations, thus saving us a small factor. Consider a run of our branch-and-bound routine with upper bound U_i. If we run it without decomposition and compute a lower bound $L_i > U_i$ on the root of the branch-and-bound tree, we set $U_{i+1} = \max\{\lceil 1.05\, U_i \rceil, L_i + 1\}$. If we run the algorithm with decomposition, recall from Section 3.6.1 that we calculate packing bounds for $4U_i/3$ subproblems to guide the distribution process. Let h be the largest integer such that at least h such subproblems have a lower bound of h or higher. From the proof of Theorem 4, it follows that h is a lower bound on the solution of the original problem. We can then set $U_{i+1} = \max\{\lceil 1.05\, U_i \rceil, h + 1\}$.

3.7.5 Branching

If the lower bound for a given subproblem (A, B) is not high enough to prune it, we must branch on an unassigned vertex v, creating subproblems $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$. Our experiments show that the choice of branching vertices has a significant impact on the size of the branch-and-bound tree and on the total running time. Intuitively, we should branch on vertices that lead to higher lower

bounds on the child subproblems. Given our lower-bounding algorithms, we can infer some properties the branching vertex v should have. Based on these properties, we compute a score for each vertex v, then branch on the vertex with the highest score. We discuss each property in turn (in roughly decreasing order of importance), then explain how they are combined into an overall score.

The first criterion we use is the degree $\deg(v)$ of each vertex v. Branching on high-degree vertices helps both the flow bound (by increasing the capacity out of the source or into the sink) and the packing bound (more trees can be created).

Second, we prefer to branch on vertices that are incident to heavier trees, since this would allow these trees to be split when a new packing bound is computed. To measure this, we define a branching parameter $\text{tweight}(v)$. If v is reachable from either A or B in G_f, we set $\text{tweight}(v)$ to the average weight of the trees adjacent to v (the average is weighted by v's own allocation). If v is unreachable from neither A nor B, $\text{tweight}(v)$ is set to the average weight of all trees in the packing (rooted at either A or B).

Third, we avoid branching on vertices reachable from B (the smaller side) in G_f, since splitting trees on this side will not (immediately) help improve the packing bound. We do so by associating a *side penalty* $\text{sp}(v)$ to each vertex v, set to 1 if v is reachable from B, and 10 otherwise.

Fourth, we branch on vertices that are far from both A and B, since having assigned vertices spread around the graph helps maintain the trees balanced in the packing bound. Accordingly, we define $\text{dist}(v)$ as the distance in G from v to the closest vertex in $A \cup B$. A single BFS is enough to find the distances between

$A \cup B$ and all vertices. If vertex v is not reachable from $A \cup B$ we set $\text{dist}(v) = M+1$, where M is the maximum finite distance from $A \cup B$.

Finally, it pays to treat disconnected graphs in a special way. Intuitively, it makes sense to branch on larger components first, since they tend to have more impact on the packing bound. We thus associate with each vertex v a value $\text{comp}(v)$, defined as the total vertex weight of the component that contains v.

Putting it all together. The relative importance of these five criteria varies widely across instances. For most instances, using just the degree is enough. The remaining four criteria are useful for robustness: they never hurt much, and can be very helpful for some specific instances. The $\text{comp}(v)$ parameter is crucial for disconnected instances, for example.

We combine all parameters in the most natural way, essentially by taking their product. More precisely, we branch on the vertex v that maximizes $q(v) = (\deg(v) + 1)^2 \cdot (\text{tweight}(v) + 1) \cdot (\text{dist}(v) + 1) \cdot \text{sp}(v) \cdot \text{comp}(v)$. The "+1" terms ensure all factors are strictly positive; a single zero in the expression would render all other factors irrelevant. We take the square of $\deg(v)$ term to reflect the fact that degrees are the most relevant criterion. We emphasize that there is nothing special about this particular expression; other approaches could be used as well, but this one showed good results in practice.

3.8 Edge Costs

Our description so far has assumed that all edges in the input graph have unit cost, but our actual implementation supports arbitrary integral costs. This section shows how we accomplish this.

First, we note that all algorithms we described can handle parallel edges. Therefore, if the original edges have small costs, we can simply replace them by parallel (unit) edges. In fact, this is how we deal with potential high-cost edges after contraction. To handle arbitrarily large integral costs efficiently, however, we need something more elaborate.

Extending the flow bound is straightforward: we still perform a standard maximum-flow computation between A and B, using edge costs as capacities. The only difference is when we "remove" the flow to create G_f and compute the packing bound. If the flow through an edge e of capacity $c(e)$ is $f(e)$, removing the flow still leaves an edge of (residual) capacity $r(e) = c(e) - f(e)$. If $c(e) = f(e)$, we just remove the edge, as before.

Tree packing. Extending the packing bound is less straightforward. We generalize tree packings as follows. First, we associate to each tree in the packing an integral *thickness*. Each edge e may belong to more than one tree, but the sum of the thicknesses of these trees must not exceed $r(e)$, the residual capacity of e. Intuitively, each edge allocates portions of its cost to different trees, with the portion allocated to tree t equal to t's thickness. In practice, we split each edge into multiple edges with smaller (potentially different) costs; we then build trees as before, but ensure that every edge within the same tree has exactly the same

cost (corresponding to the thickness of the tree).

More precisely, we handle large costs by splitting each edge into a collection of parallel subedges, each with a cost taken from a discrete set of values. The exact parameters of this split depend on the average edge cost $\gamma = [\sum_{e \in E} cost(e)]/|E|$. If $\gamma \leq 10$, we simply split each original edge into unit-cost edges. If $\gamma > 10$, we use a scaling approach to compute disjoint sets of trees in decreasing order of thickness. For each threshold θ_i (with $\theta_0 > \theta_1 > \ldots > \theta_k = 1$), we create a *partial graph* G_i in which every edge has cost θ_i. For each edge e in the original graph G, we create $\lfloor cost(e)/\theta_i \rfloor$ parallel edges in G_i. We then use our standard algorithm to compute a tree packing $\mathcal{T}_i$ in G_i. For each edge in $\mathcal{T}_i$, we subtract τ_e from the current cost of the corresponding edge e in G. Note that every tree in $\mathcal{T}_i$ has thickness exactly θ_i. To create our final tree packing $\mathcal{T}$, we simply take the union of the trees in all $\mathcal{T}_i$ packings, with their original thicknesses. We set the initial threshold to $\theta_0 = \lfloor \gamma / \sqrt{10} \rfloor$, then set $\theta_i = \eta \lfloor \theta_{i-1} \rfloor$ between iterations. The step $\eta < 1$ is chosen so as to make the total number of steps approximately equal to $\log_{10} \gamma$.

After the trees themselves are created, we use the algorithm described in Section 3.4.2 to allocate vertex weights to the trees, but taking thicknesses into account. Our goal is to have the ratio between the weight and the thickness to be roughly the same across all trees, the reason will become clear soon. Both phases described in Section 3.4.2 (initial allocation and local search) can be trivially generalized to accomplish this, with no penalty in asymptotic performance. Once again, it suffices to interpret a thick tree as a collection of identical trees with unit thickness.

Calculating the Packing Bound. We now consider how to compute the packing bound associated with a tree packing $\mathcal{T}$ that is *heterogeneous*, i.e., contains trees with different thicknesses. First, note that correctness is not an issue: we could simply create a homogeneous packing $\mathcal{T}'$ from $\mathcal{T}$ (by replacing each tree of thickness of k by a collection of k trees of unit thickness); the bound given by Theorem 4 ($f + p(\mathcal{T}')$) is still valid. Since $\mathcal{T}'$ may be quite large, however, computing such bound explicitly would be expensive.

We therefore generalize the greedy algorithm to deal with heterogeneous tree packings directly. Note that this is essentially the fractional knapsack problem, so instead of picking trees in decreasing order of weight, we process them in decreasing order of *profit*, defined as the ratio between weight and thickness. We take the trees in this order their cumulative weight is about to exceed the target weight W_f; the packing bound is the sum of their thicknesses, together with the fractional weight (rounded up) of the tree at the threshold.

Forced Assignments. The forced assignment routines described in Section 3.5 enable us, in some situations, to assign a vertex v to one side of the bisection without the need for an explicitly branch. These routines can be generalized to handle heterogeneous tree packings.

For the plain flow-based assignments, we must add up the thicknesses of the parent edges of v. In the extended flow-based assignment, when considering a path through a child edge e, the increase in flow is equal to minimum between the thicknesses of $tree(e)$ (the tree containing e) and $x_v(e)$ (the expansion tree). Finally, the subdivision-based assignment remains essentially unchanged, except for the fact that newly-created trees may have thickness greater than one.

Decomposition. Our decomposition technique can be generalized as in the packing bound. First one splits each high-cost original edge into a small number of cheaper edges. For each such cost, one partitions the corresponding edges into clumps, which are then distributed to different subproblems, as before. One then solves each such subproblem independently (contracting the clump edges as usual), and return the best solution found. We require all clumps assigned to a given subproblem to have the same thickness k, therefore solving this subproblem is equivalent to solving k independent subproblems with thickness one. For correctness, therefore, it suffices to have the sum of the thicknesses of all subproblems be at least U, the upper bound on the solution value.

Branching. When branching, we define the degree $\deg(v)$ of a vertex v as the sum of the costs of its incident edges.

3.9 Very Large Graphs

Our experiments showed that our algorithm produces a relatively small branch-and-bound tree when the size of the optimal bisection is small when compared to the number of vertices in the graph. We are still unable to solve to optimality instances in this case when the graph is very large, because of the time required to compute lower bounds and forced assignments in each node of the branch-and-bound tree.

As an example, consider the instance USA, which consists of a complete road map of the USA, with almost 24 million vertices and 29 million edges, whose optimal solution is only 87 where each node takes roughly 20 minutes to be

solved (initialization, maximum flow, packing bound, forced assignment, and branch node selection). This can also be observed in other similar graphs.

One observation to be made is that usually the branch-and-bound tree is very unbalanced, with most branches being pruned in a very low height, and the branch in the neighborhood of the optimal solution going deeper (see Figure 3.2). This suggests an approach that makes all shallow nodes cheaper with possibly weaker bounds, and concentrate effort on the (few) deep nodes.

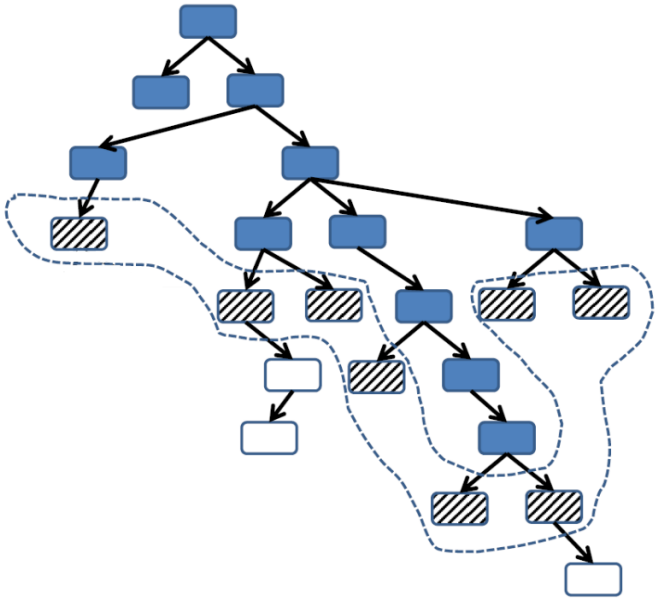


Figure 3.2: Example of a branch-and-bound tree showing frontier. Note that only the branches close to where the solution is (bottom right) are deep.

Another observation is that a relatively small number of edges is used to compute the flow bound in graphs of this nature, and that several small subgraphs will always be given to the same tree if none of its nodes are fixed (for instance, a cul-de-sac in a road network).

To exploit those observations we will create a graph G^{flow} with *pre-assembled flow paths* and a graph G^{tree} with *pre-assembled trees* that will be small compared

to G. Let $V' \subseteq V$ be a set of *important vertices* of G (we will show how to compute them in Subsection 3.9.1.) In this section we restrict ourselves again to unit edge costs/capacities for ease of exposition.

Graph $G^{\text{flow}} = (V', E_f)$ has edges corresponding to edge-disjoint paths in G. If $(u, v) \in E_f$ then there is a path $P(u, v)$ in G from u to v, where $P(x, y)$ for all $(x, y) \in E_f$ are disjoint.

Graph $G^{\text{tree}} = (V_t, E_t)$ is a little more complex. Vertices $t \in \mathbb{T}$ correspond to edge-disjoint trees of G each touching exactly one node of $t(V') \in V'$. There is an edge (t_1, t_2) if the edge-disjoint trees t_1 and t_2 share a vertex different from $t_1(V')$.

Suppose we restrict ourselves to only branch on nodes in V'. A max A–B-flow in G^{flow} with value f' is clearly a lower bound to a max A–B-flow in G. Let $\bar{E}$ be the set of edges of G used by such a maximum flow F in G^{flow}. In other words, $\bar{E} = \bigcup_{(u,v):F(u,v)=1} P(u, v)$.

Definition 4. *An* edge-disjoint connected subgraph packing *of a graph $G = (V, E)$ and subset $A \subseteq V$ is a collection of connected subgraphs of G such that (1) each edge belongs to at most one subgraph, (2) each subgraph contains exactly one edge connecting a vertex in A to a vertex not in A, and (3) we cannot add an edge to any subgraph while maintaining the previous two conditions.*

Note that an edge-disjoint connected subgraph packing where all subsets are acyclic is an edge-disjoint tree packing. Note also that the proof of Theorem 4 does not use the fact that the subsets are acyclic, only that they are connected. Therefore the lower bound is valid for edge-disjoint connected subgraph packings also. We cannot perform subdivision and extended flow assignments in an efficient way anymore, since there is no notion of "weight of a subtree" in

connected graphs with cycles.

Let $\bar{V}_t$ be formed by pre-assembled trees parts t that do not contain any edge of $\bar{E}$. As before assume that A is the main side, and let $\mathbb{T}' \subseteq \mathbb{T}$ be the set of pre-assembled trees that touch A, in other words $\mathbb{T}' = \{t : t(V') \in A\}$. Then a *vertex-disjoint* tree packing with roots on $\mathbb{T}'$ on the subgraph induced by $\bar{V}_t$ corresponds to an edge-disjoint connected subgraph packing on G, say, with value p. Therefore $f' + p$ is a valid lower bound for the current branch-and-bound node by Theorem 4. We will show how to compute such vertex-disjoint tree packing in Subsection 3.9.2. Another way of thinking of the vertex-disjoint tree packing on G^{tree} is by considering assembling subgraphs by using several pre-assembled trees.

Note that while the lower bound might be smaller we deal with each branch-and-bound node considerably faster since we run our algorithms in graphs smaller by orders of magnitude.

We proceed by using G^{flow} and G^{tree} if the depth of the current node in the branch-and-bound tree is not greater than a threshold γ. When the depth is larger than γ we use the algorithm described in Section 3.7. This clearly leads to a correct algorithm, regardless of V', G^{flow} and G^{tree}. Experiments showed that $\gamma = 10$ gives good results for graphs with a few tens of millions of vertices.

3.9.1 Computing G^{flow} and G^{tree}

We need to select the subset $V' \subseteq V$ of special nodes. We want V' to be small, but to represent V well. Intuitively we want to have at least one vertex $v \in V'$ in

each *highly connected component* of V.

Selecting V' We do it by selecting random pairs of vertices with probability proportional to the square of their degrees and computing the maximum flow between them. If it is smaller than the known upper bound U we select such a node pair as a candidate pair, and record the maximum flow between them. They are not highly connected.

After such a selection of candidates is done we sort the pairs in increasing order of flow between them, and consider including both vertices in V'. We include both if (1) it is the first candidate pair considered, (2) one of the vertices is already in V', or (3) the average maximum flow from the pair to vertices already in V' is smaller than a threshold. We test condition 3 by selecting a random subset of V'.

We aim for a V' whose size is close to 0.5% if the size of V. To give concrete numbers, in the road map of Italy, for instance (a graph with 6.6M vertices and 7M edges) we create V' with 41K vertices, 66K flow paths and 160K trees. G^{flow} is roughly 0.1%, and G^{tree} is roughly 2.5% the size of the original graph.

After computing G^{flow} and G^{tree} we store the following data structures. For each vertex $v \in V'$ we store which vertex $v(V) \in V$ it corresponds to. For each edge $(u, v) \in E_f$ we store the list of edges of $P(u, v)$ in an array. For each edge of G we store in what pre-assembled tree t_i that edge belongs to (if any).

Computing G^{flow} After we have V' we compute G^{flow} by finding maximum flows between random pairs of vertices $u, v \in V'$ with probability of choosing u proportional to the maximum flow associated with the candidate pair u came

from. We then decompose the flow in paths, and create an edge of G^{flow} that represents such path (recording the edges of G that are being represented by the new edge). We remove the edges with flow, and repeat for a fixed number of iterations ($10 \log_2 N$ works well in practice).

After all iterations we should add a few more edges to G^{flow} as to guarantee that the selected paths correctly cuts all pairs of nodes $u, v \in V'$. Let $V' = \{v_0, v_1, \ldots, v_{|V'|-1}\}$, and think of the index i of v_i written in binary. Let L be the number of bits necessary to write $|V'| - 1$. We compute maximum flows between V'^0_j and V'^1_j for $j = 0, \ldots, L-1$ where V'^k_j is the set of nodes of V' where the j-th bit is k, removing the flow edges after each maximum flow computation. Since each pair of vertices of V' will be in different sides of the partition at least once, the selected paths will correctly cut all pairs of vertices.

Computing G^{tree} We compute the vertices of G^{tree} (pre-assembled trees of G) simply by calling our tree packing procedure on V'. The weight of a vertex of G^{tree} is the total weight allocated to it by its vertices in the algorithm. To compute the edges of G^{tree} we have to find, for each tree, the set of subtrees it touches. Suppose the vertices of G^{tree} (trees of G) are ordered as $t_1, \ldots, t_r$. We process them in order from t_1 to t_r. We also maintain a *timestamp* for each tree t_j that tells to which trees list tree j was last added.

While processing tree t_i we maintain a go to each of its vertices in BFS order (the order is not important for correctness but BFS is the most efficient tree traversal method) and scan all its edges for trees. If an edge belong to tree t_j with $i \neq j$ and the timestamp of t_j is strictly smaller than i we update the timestamp of t_j to i and add edge (t_i, t_j) to G^{tree} if $i > j$ (to avoid two edges to be

added). The whole traversal runs in $O(\sum_{v \in V} deg(v)) = O(m\Delta)$, since each node v is scanned $O(deg(v))$ times.

3.9.2 Computing Bounds Based on G^{flow} and G^{tree}

As before, suppose we have a partial assignment (A, B) of V, and suppose furthermore that $A, B \subseteq V'$, the set of important vertices, and let A be the main side.

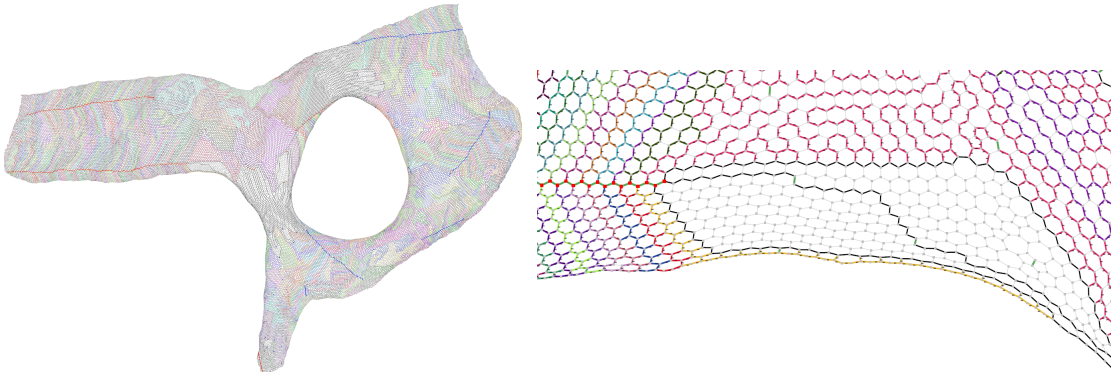
Flow Bound Start by finding a maximum A–B-flow on G^{flow} with value f using IBFS [50]. Then, for each edge (u, v) of G^{flow} with positive flow, go through the list of edges of $P(u, v)$ in G, and for each edge, if it is present in a pre-assembled tree (vertex of G^{tree}) t_i, mark t_i as "invalid".

Packing Bound We then compute a vertex-disjoint tree packing of G^{tree} without using any invalid vertices t_i. Note that since the pre-assembled trees are edge-disjoint (in G), we have that a vertex-disjoint tree packing will correspond to a packing of G that is clearly (1) edge-disjoint and (2) connected (by the way we define edges of G^{tree}). In other words, it will be an edge-disjoint connected subgraph packing of G. To compute it we follow an adaptation of the simultaneous depth-first search from Subsection 3.4.1, where when looking for an expanding edge (u, w) all we require is that w is free, i.e. it does not belong to any tree already. Since the trees are vertex-disjoint the vertex allocation is trivial, our only option is give the entire weight of a vertex to the unique tree containing it, if such tree exists.

After that a similar greedy algorithm gives the packing bound.

3.10 Graphs With Big Unreachable Areas After Removing Flow Edges

When the graph has several vertices of small degree the flow edges can create a "barrier" for the tree packing. In an extreme example, suppose we are dealing with a 3-regular graph. Every free node touched by flow edges will have two of its edges not present in G_f. This essentially disconnects it to "one side of the graph", and creates a big amount of deadweight, which renders the packing bound useless. See Figure 3.3.



(a) Full Instance (b) Details of one of the "bad" areas

Figure 3.3: Tree Packing on instance t60k with big amount of deadweight

In Figure 3.3 bold edges have flow in them, while colors different from gray represent trees of the tree packing. In Figure 3.3a we can see some big regions of uncolored vertices (deadweight) on the left of "central hole". In Figure 3.3b we can see details of one such big region. As seen trees cannot break through the barrier created by the flow edges.

Note that this is an issue present in several important classes of graphs, like road networks and meshes.

Our solution to this problem is essentially to allow "fractional flows" where we only use "part of an edge". Instead of modifying IBFS to allow fractional flows we expanded the graph by replacing each edge (u, v) with k parallel paths between u and v, and give cost/capacity of $1/k$ to each one of them. We also introduce new artificial vertices so that the k parallel paths between u and v have length $1, 2, \ldots, k$ (see Figure 3.4).

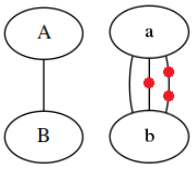


Figure 3.4: Fractional flow transformation with $k = 3$

3.11 Experiments

We now evaluate the performance of our branch-and-bound algorithm on several benchmark instances from the literature. Subsection 3.11.1 starts our experimental analysis by comparing our method with state-of-the-art exact approaches. We show that, although our algorithm is outperformed on some graphs (notably those with high expansion), it is much faster than existing approaches on a wide variety of realistic graph classes, such as sparse graphs with relatively small bisections. This motivates our second set of experiments, reported in Subsection 3.11.2, which shows that our approach can solve, for the first time, large instances (with tens of thousands of vertices) to optimality. In Subsection 3.11.3 we present results from very large graphs by using the algo-

rithm of Section 3.9. In Subsection 3.11.4 we present the improvements achieved by the fractional flow technique developed on Section 3.10. Subsection 3.11.5 studies the relative importance of each of the many elements of our algorithm, such as forced assignments and decomposition strategies.

We implemented our algorithm in C++ and compiled it with full optimization on Visual Studio 2010. All experiments were run on a single core of an Intel Core 2 Duo E8500 with 4 GB of RAM running Windows 7 Enterprise at 3.16 GHz. All instances tested, as well as the solutions found by our algorithm, are available at `http://www.cs.princeton.edu/~rwerneck/bisection/` or upon request.

3.11.1 Exact Benchmark Instances

Our first experiment compares our algorithm with the results reported by Hager et al. [52]. They use standard benchmark instances (originally considered by Brunetta et al. [16]) to compare their own quadratic-programming-based algorithm (CQB) with other state-of-the-art methods from the literature: BiqMac [90] (semidefinite programming), KRC [66] (also semidefinite programming), and SEN [96] (multicommodity flows).

Table 3.1 and Table 3.2 compare the performance of our algorithm against the results reported by Hager et al. [52]. For each instance, we show the number of vertices (n) and edges (m), the value of the optimum bisection (using $\epsilon = 0$), followed by the number of branch-and-bound nodes (BB) and the running time of our algorithm (TIME). In this and other experiments we run our algorithm a single time. While the algorithm is random preliminary experiments indicated

Table 3.1: Performance on standard benchmark instances. Columns indicate number of vertices (n) and edges (m), optimum bisection value (opt), number of branch-and-bound nodes (BB) for our algorithm, and running times in seconds for our method (TIME) and others (CQB, BiqMac, KRC, SEN); "—" means "not tested" and DNF means "not finished in 2.5 hours".

CLASS	NAME	n	m	*opt*	BB	TIME	CQB	BiqMac	KRC	SEN
grid	2x16	32	46	8	29	**0.01**	0.05	0.06	0.10	—
	18x2	36	52	6	32	**0.01**	0.06	0.04	0.05	—
	2x19	38	55	6	60	**0.02**	0.09	2.91	1.44	—
	5x8	40	67	18	36	**0.02**	0.08	0.06	0.05	—
	3x14	42	67	10	47	**0.02**	0.10	0.56	0.47	—
	5x10	50	85	22	47	**0.03**	0.22	0.25	0.24	—
	6x10	60	104	28	63	**0.03**	0.26	3.59	8.38	—
	7x10	70	123	23	83	**0.05**	0.41	17.05	14.63	—
fem	m4.i	32	50	6	31	**0.01**	0.03	0.03	0.02	0.11
	ma.i	54	72	2	17	**0.01**	0.13	0.04	0.08	0.22
	me.i	60	96	3	12	**0.01**	0.16	0.11	0.10	0.22
	m6.i	70	120	7	30	**0.01**	0.35	0.54	0.97	1.13
	mb.i	74	120	4	39	**0.01**	0.29	0.49	0.77	0.90
	mc.i	74	125	6	41	**0.01**	0.32	0.39	1.21	1.13
	md.i	80	129	4	38	**0.01**	0.42	0.69	0.76	1.01
	mf.i	90	146	4	36	**0.01**	0.53	1.12	0.63	1.46
	m1.i	100	155	4	57	**0.02**	0.78	14.77	28.76	2.36
	m8.i	148	265	7	46	**0.02**	3.43	2.41	8.43	3.26
mixed	2x17m	34	561	316	32971	22.07	1.21	0.91	**0.76**	—
	10x4m	40	780	436	180068	158.64	2.99	**0.04**	0.05	—
	5x10m	50	1225	670	6792445	8006.36	223.14	0.11	**0.05**	—
	13x4m	52	1326	721	DNF	DNF	853.07	**0.50**	0.89	—
	4x13m	52	1326	721	DNF	DNF	571.58	**0.52**	0.89	—
	9x6m	54	1431	792	DNF	DNF	2338.03	0.42	**0.32**	—
	10x6m	60	1770	954	DNF	DNF	2051.07	0.24	**0.20**	—
	10x7m	70	2415	1288	DNF	DNF	3975.26	**0.31**	0.36	—

that changing the seed of the random generator does not affect the results too much. The final four columns show the running times reported by Hager et al. [52] for all competing algorithms (when available). The times reported by Hager et al. [52] are already scaled to reflect the approximate execution time on their machine, a 2.66 GHz Xeon X5355. To make the resulting times consistent with our (slightly faster) machine, we further multiply those results by 0.788.[2]

[2]The scaling factor was obtained from `http://www.cpubenchmark.net/`

Table 3.2: Performance on standard benchmark instances. Columns indicate number of vertices (n) and edges (m), optimum bisection value (opt), number of branch-and-bound nodes (BB) for our algorithm, and running times in seconds for our method (TIME) and others (CQB, BiqMac, KRC, SEN); "—" means "not tested" and DNF means "not finished in 2.5 hours".

CLASS	NAME	*n*	*m*	*opt*	BB	TIME	CQB	BiqMac	KRC	SEN
random	q0.00	40	704	1606	DNF	DNF	294.11	0.19	**0.10**	—
	q0.10	40	647	1425	DNF	DNF	159.23	2.83	**1.07**	—
	q0.20	40	566	1238	4122814	5541.47	52.02	3.03	**0.65**	—
	q0.30	40	506	1056	949311	1261.00	29.48	4.42	**1.62**	—
	q0.80	40	145	199	1004	0.52	**0.16**	5.65	1.81	—
	q0.90	40	78	63	175	**0.06**	0.09	0.23	0.10	—
	c0.00	50	1108	2520	DNF	DNF	1892.41	10.98	**3.68**	—
	c0.10	50	1003	2226	DNF	DNF	1406.71	12.21	**4.18**	—
	c0.30	50	802	1658	DNF	DNF	1743.65	18.45	**4.30**	—
	c0.70	50	350	603	105712	107.94	12.49	13.31	**6.35**	—
	c0.80	50	235	368	11056	8.61	**1.46**	9.91	4.83	—
	c0.90	50	130	122	388	0.22	**0.18**	0.28	0.26	—
	c2.90	52	137	123	354	0.21	**0.20**	0.36	0.32	—
	c4.90	54	149	160	993	0.53	**0.32**	0.52	2.60	—
	c6.90	56	166	177	1340	0.79	0.52	**0.30**	0.79	—
	c8.90	58	179	226	4993	2.94	**1.56**	22.42	13.81	—
	s0.90	60	195	238	2630	1.76	**1.16**	4.38	7.80	—
tori	8x5	40	80	33	108	**0.03**	0.09	0.13	0.16	—
	21x2	42	63	9	24	**0.01**	0.02	0.17	0.13	—
	23x2	46	69	9	78	**0.02**	0.13	8.63	3.28	—
	4x12	48	96	24	39	**0.02**	0.16	0.23	0.44	—
	5x10	50	100	33	80	**0.03**	0.17	0.30	0.16	—
	6x10	60	120	35	77	**0.05**	0.34	4.75	9.19	—
	7x10	70	140	45	140	**0.08**	0.57	0.64	15.02	—
	10x8t	80	160	43	93	**0.05**	0.64	3.75	24.79	—
debruijn	debr5	32	61	10	145	0.02	0.06	0.06	0.16	**0.00**
	debr6	64	125	18	2583	0.60	**0.38**	0.66	12.32	0.79
	debr7	128	253	30	109039	19.67	2436.76	371.89	2204.00	**8.10**

Note that we do not give any upper bound U to our algorithm; it simply tries increasing values of U as needed, as described in Section 3.7.4. The statistics we report aggregate over all such runs. For each value of U, we use the automated approach described in Section 3.7.2 to decide whether to use decomposition or not.

`singleThread.html`.

The instances in this experiment are divided in classes according to their properties. Class grid corresponds to $h \times k$ rectangular grids with integral edge costs picked uniformly at random from the range $[1, 10]$. Class tori is similar, but with extra edges to make the grids toroidal. Each instance in class mixed consists of an $h \times k$ grid (with random edge costs in the range $[1, 100]$) to which extra edges (with random costs in the range $[1, 10]$) are added in order to create a complete graph. Class random contains random graphs of various densities and edge costs in the range $[1, 10]$. Class fem contains finite element meshes. Finally, debruijn contains de Bruijn graphs, which are useful in parallel computer architecture applications.

Table 3.1 shows that our method can solve all grid, fem, and tori instance in a few hundredths of a second; the difference to other approaches increases significantly with the size of the instances, indicating that our algorithm is asymptotically faster. Our algorithm also does well on debruijn instances, on which it is never far from the best algorithm.

For mixed and denser random instances, however, our method is clearly outperformed, notably by the methods based on semidefinite programming (BiqMac and KRC). For these instances, the ratio between the solution value and the average degree is quite large, so we can only start pruning very deep in the tree. Decomposition would not help, since it would contract very few edges per subproblem.

Intuitively, our method does well when the number of edges in the bisection is a small fraction of the total edge cost, which is not the case for mixed and random, but is definitely the case for grid-like graphs.

Our second experiment considers benchmark problems compiled by Armbruster [6] from previous studies. They include instances used in VLSI design (alue, alut, diw, dmxa, gap, taq) [41, 64], finite element meshes (mesh) [41], random graphs (G) [61], random geometric graphs (U) [61], as well as graphs derived from sparse symmetric linear systems (KKT) [54] and compiler design (cb) [62]. In each case, we use the same value of ϵ tested by Armbruster, which is either 0 or 0.05.

Table 3.3 reports the results obtained by our algorithm. For comparison, we also show the best running times obtained by Armbruster et al. [5, 7, 8] (using linear or semidefinite programming, depending on the instance) and by Hager et al. [52] (using CQB). As before, we multiply the times reported by Hager et al. by 0.788; similarly, we multiply the times reported by Armbruster [5] (on a 3.2 GHz Pentium 4 540) by 0.532 for consistency with our machine. Results for other algorithms (such as KRC and BiqMac) are only available for a tiny fraction of the instances in this table, and are thus discussed in the text only whenever appropriate.

The table includes all instances that can be solved by at least one method in less than 150 minutes in our machine (roughly corresponding to the 5-hour time limit set by Armbruster [5]), except those that can be solved in less than 5 seconds by both our method and Armbruster's. Note that we can solve every instance in the table to optimality. Although our method can be slightly slower than Armbruster's (notably on alue6112.16896), it is usually much faster, often by orders of magnitude. We can solve in minutes (or even seconds) several instances no other method can handle in hours.

Our approach is significantly faster than Hager et al.'s for mesh, KKT, and

Table 3.3: Performance of our algorithm compared with the best times obtained by Armbruster [5] and Hager et al. [52]. Columns indicate number of nodes (n), number of edges (m), allowed imbalance (ϵ), optimum bisection value (opt), number of branch-and-bound nodes (BB), and running times in seconds for our method (TIME) and others ([Arm07], CQB); "—" means "not tested" and DNF means "not finished in 2.5 hours".

NAME	n	m	ϵ	*opt*	BB	TIME	[Arm07]	CQB
G124.02	124	149	0.00	13	376	**0.06**	7.40	3.32
G124.04	124	318	0.00	63	166799	**35.11**	2334.24	751.46
G250.01	250	331	0.00	29	42421	**10.86**	974.76	7963.64
KKT_capt09	2063	10936	0.05	6	30	**0.15**	619.72	3670.97
KKT_skwz02	2117	14001	0.05	567	902	**10.97**	DNF	—
KKT_plnt01	2817	24999	0.05	46	1564	**22.32**	DNF	—
KKT_heat02	5150	19906	0.05	150	4346	**83.63**	DNF	—
U1000.05	1000	2394	0.00	1	45	**0.03**	28.53	—
U1000.10	1000	4696	0.00	39	315	**1.09**	883.46	—
U1000.20	1000	9339	0.00	222	16502	**149.40**	DNF	—
U500.05	500	1282	0.00	2	34	**0.02**	10.54	—
U500.10	500	2355	0.00	26	95	**0.16**	263.82	—
U500.20	500	4549	0.00	178	27826	**104.34**	DNF	—
U500.40	500	8793	0.00	412	232593	**2159.34**	DNF	—
alut2292.6329	2292	6329	0.05	154	17918	**201.44**	208.42	—
alue6112.16896	6112	16896	0.05	272	239815	7778.05	**2539.85**	—
cb.47.99	47	99	0.00	765	141	**0.08**	2.81	0.23
cb.61.187	61	186	0.00	2826	193	**0.63**	43.28	**0.63**
diw681.1494	681	1494	0.05	142	3975	**8.25**	DNF	—
diw681.3103	681	3103	0.05	1011	5659	**129.82**	DNF	—
diw681.6402	681	6402	0.05	330	1234	**8.26**	2436.09	—
dmxa1755.10867	1755	10867	0.05	150	2417	**31.98**	DNF	—
dmxa1755.3686	1755	3686	0.05	94	8233	**45.93**	1049.22	—
gap2669.24859	2669	24859	0.05	55	11	**0.24**	185.64	—
gap2669.6182	2669	6182	0.05	74	2424	**24.93**	346.35	—
mesh.138.232	138	232	0.00	8	87	**0.02**	5.44	5.45
mesh.274.469	274	469	0.00	7	57	**0.03**	4.53	19.40
taq170.424	170	424	0.05	55	246	**0.51**	15.26	—
taq334.3763	334	3763	0.05	341	2816	**5.07**	DNF	—
taq1021.2253	1021	2253	0.05	118	3009	**9.88**	90.25	—

sufficiently sparse random graphs (G). For the cb class, the algorithms have similar performance (KRC—not shown in the table—has comparable running times as well). These instances are small but its edges are costly, which means our algorithm is only effective because of the scaling strategy described in Section 3.8.

Without scaling, it would be two orders of magnitude slower.

With longer runs, Armbruster [5], Hager et al. [52], and BiqMac [90] can solve denser random graphs G124.08 and G124.16 in a day or less (not shown in the table). We would take about 3 days on G124.08, and a month or more for G124.16. Once again, this shows that there are classes of instances in which our method is clearly outperformed.

The solutions we report were in most cases known; we just proved their optimality. We did find better solutions than those reported by Armbruster [5] in three cases: KKT_plnt01, diw681.6402, and taq334.3763. For the last two, however, Armbruster claims matching lower bounds for his solutions. This discrepancy could in principle be due to slightly different definitions of W_+, the maximum allowed cell size. We define it as in the 10th DIMACS Implementation challenge [10] ($W_+ \leq \lfloor(1+\epsilon)\lceil W/2\rceil\rfloor$) whereas Armbruster [5] uses $W_+ \leq \lceil(1+\epsilon)W/2\rceil$; these values can differ very slightly for some combinations of W and ϵ. The solutions we found, however, obey both definitions. For KKT_plnt01, we found a solution with cut size 46, $w(A) = 1479$, $w(B) = 1338$, and an imbalance (defined as $\max\{w(A), w(B)\}/\lceil W/2\rceil$) of 4.968%. For diw681.6402, we computed a solution of 330 with $w(A) = 1350$ and $w(B) = 1221$ (imbalance 4.977%), while for taq334.3763 our solution of 341 has $w(A) = 556$ and $w(B) = 503$ (imbalance 4.906%). We conjecture that Armbruster's code actually uses $W_+ \leq (1+\epsilon)W/2$ (without the ceiling). Indeed, with $\epsilon = 0.049$ our code finds the same solutions as Armbruster does (49, 331, and 342, respectively). Our running times are essentially the same with either value of ϵ.

3.11.2 Larger Benchmark Instances

We now show that we can actually solve real-world instances that are much larger than those shown in Table 3.3. In particular, we consider instances from the 10th DIMACS Implementation Challenge [10], on Graph Partitioning and Graph Clustering. They consist of social and communication networks (class clustering), road networks (streets), Delaunay triangulations (delaunay), random geometric graphs (rgg), planar maps representing adjacencies among census blocks in US states (redistrict), and assorted graphs (walshaw) from the Walshaw benchmark [100] (mostly finite-element meshes). We stress that these testbeds were created to evaluate heuristics and were believed to be beyond the reach of exact algorithms. To the best of our knowledge, no exact algorithm has been successfully applied to these instances.

Table 3.4 and Table 3.5 report the detailed performance of our algorithm. For each case, we show the number of vertices (n) and edges (m), the optimum bisection value (opt), the total number of nodes in the branch-and-bound tree (BB), and the total running time of our algorithm in seconds. We use $\epsilon = 0$ for all classes but one: since vertices in redistrict instances are weighted (by population), we allow a small amount of imbalance ($\epsilon = 0.03$). We only report instances that our algorithm can solve within a few hours. Since we deal with very large instances in this experiment, we save time by running our algorithm with $U = opt + 1$. (The opt values were gathered over time from preliminary runs of our own algorithm.) Running times would increase by a small constant factor if we ran the algorithm repeatedly with increasing values of U. As usual, we use the heuristic described in Section 3.7.2 to decide whether to use decomposition in each case.

Table 3.4: Performance of our algorithm on DIMACS Challenge instances starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds. We use $\epsilon = 0$ for all classes but redistrict, which uses $\epsilon = 0.03$.

CLASS	NAME	n	m	opt	BB	TIME
clustering	karate	34	78	10	4	0.02
	chesapeake	39	170	46	26	0.03
	dolphins	62	159	15	32	0.02
	lesmis	77	254	61	17	0.03
	polbooks	105	441	19	7	0.02
	adjnoun	112	425	110	12 488	4.13
	football	115	613	61	2 046	1.17
	jazz	198	2 742	434	70 787	160.37
	celegansneural	297	2 148	982	83	0.78
	celegans_metabolic	453	2 025	365	359	0.66
	polblogs	1 490	16 715	1 213	327 740	7 748.64
	netscience	1 589	2 742	0	363	0.20
	power	4 941	6 594	12	71	0.32
	PGPgiantcompo	10 680	24 316	344	49 381	1 100.62
	as-22july06	22 963	48 436	3 515	4 442	279.31
delaunay	delaunay_n10	1 024	3 056	63	1 422	3.76
	delaunay_n11	2 048	6 127	86	3 698	17.46
	delaunay_n12	4 096	12 264	118	9 322	100.49
	delaunay_n13	8 192	24 547	156	18 245	442.18
	delaunay_n14	16 384	49 122	225	599 012	31 281.53
redistrict	de2010	24 115	58 028	36	43	3.08
	hi2010	25 016	62 063	44	887	42.10
	ri2010	25 181	62 875	107	921	73.33
	vt2010	32 580	77 799	112	1 398	137.09
	ak2010	45 292	108 549	48	61	6.62
	nh2010	48 837	117 275	146	12 997	2 125.96
	ct2010	67 578	168 176	150	11 081	2 988.57
	me2010	69 518	167 738	140	12 855	3 215.64
	nv2010	84 538	208 499	126	883	259.03
	wy2010	86 204	213 793	190	6 113	2 098.37
	ut2010	115 406	286 033	198	12 112	5 639.60
	mt2010	132 288	319 334	209	3 728	2 129.32
	wv2010	135 218	331 461	222	76 644	48 675.42
	id2010	149 842	364 132	181	3 822	2 493.53
	nj2010	169 588	414 956	150	26 454	21 455.21
	la2010	204 447	490 317	125	5 116	4 299.09

Table 3.5: Performance of our algorithm on DIMACS Challenge instances starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds. We use $\epsilon = 0$ for all classes but redistrict, which uses $\epsilon = 0.03$.

CLASS	NAME	n	m	opt	BB	TIME
rgg	rgg15	32 768	160 240	181	3 072	615.04
	rgg16	65 536	342 127	314	28 301	14 064.83
streets	luxembourg	114 599	119 666	17	318	38.22
walshaw	data	2 851	15 093	189	26 098	304.59
	3elt	4 720	13 722	90	860	10.45
	uk	4 824	6 837	19	1 429	6.25
	add32	4 960	9 462	11	22	0.15
	whitaker3	9 800	28 989	127	992	25.66
	crack	10 240	30 380	184	15 271	481.37
	fe_4elt2	11 143	32 818	130	1 189	37.40
	4elt	15 606	45 878	139	1 903	89.83
	fe_pwt	36 519	144 794	340	2 569	458.34
	fe_body	45 087	163 734	262	42 373	6 176.41
	brack2	62 631	366 559	731	34 308	20 327.60
	finan512	74 752	261 120	162	219	37.85

The table shows that our method can solve surprisingly large instances, with up to hundreds of thousands of vertices and edges. In particular, we can easily handle luxembourg, a road network with more than 100 thousand vertices; the fact that the bisection value is relatively small certainly helps in this case. Instances of comparable size from the redistrict class are harder, but can still be solved in a few minutes. We can also find the minimum bisections of rather large walshaw instances [100], which are mostly finite element meshes. For every such instance reported in the table, we show (for the first time, to the best of our knowledge) that the best previously known bisections, which were found by heuristics [11,19,53,56,74,99] with no time limit, are indeed optimal.

Our algorithm can also deal with fairly large Delaunay triangulations (delaunay) and random geometric graphs (rgg). The sets of vertices in both classes correspond to points picked uniformly at random within a square, and

differ only in how edges are added. Although random geometric graphs are somewhat denser than the triangulations, the actual cut sizes (bisections) are not much bigger. Our algorithm can then allocate more edges to each subproblem (during decomposition), which explains why it works better on rgg graphs. Our algorithm can also find exact solutions for some clustering graphs, even though the minimum bisection value is much larger relative to the graph size.

The rule described in Section 3.7.2 causes our algorithm to use decomposition for all instances in Table 3.4, except those from classes clustering (which can have very high-degree vertices) and redistrict (which are sparse but have a few high-degree vertices). For a small fraction of the redistrict instances, however, using decomposition would be slightly faster: solving nj2010, for example, would be four times quicker with decomposition. As anticipated, the rule is not perfect, but works well enough for most instances.

To further illustrate the usefulness of our approach, Table 3.6 considers additional natural classes of large instances with relatively small (but nontrivial) bisections. We test three classes of inputs: cgmesh (meshes representing various objects [91], commonly used in computer graphics applications), road (road networks from the 9th DIMACS Implementation Challenge [34]), and steinlib (sparse benchmark instances for the Steiner problem in graphs [70], mostly consisting of grid graphs with holes representing large-scale circuits). Once again, we use $\epsilon = 0$ and $U = opt + 1$. Our algorithm uses decomposition for all instances tested.

As the table shows, we can find the optimum bisection of some road networks with more than a million vertices in less than an hour while traversing very few branch-and-bound nodes. Decomposition is particularly effective on

these instances, since a very small fraction of the edges belong to the minimum bisection. Since the graphs themselves are large, however, processing each node of the branch-and-bound tree can take a few seconds even with our almost-linear-time combinatorial algorithms. The main bottlenecks are finding the tree packing and the flow, which take comparable time. All other elements are relatively cheap: decomposition, allocating vertex weights to trees, the greedy computation of the packing bound (given the trees), and forced assignments. This relative breakdown holds for all instances tested, not just road networks.

The other two classes considered in Table 3.6 (steinlib and mesh) have larger bisections and need substantially more branch-and-bound nodes. Even so, we can handle instances with tens of thousands of vertices in a few minutes.

Finally, Table 3.7 shows the results of longer runs of our algorithm on some large benchmark instances. For most instances we use $\epsilon = 0$. The only exception is taq1021.5480, which is usually tested with $\epsilon = 0.05$ in the literature [5]; it is also the only one in the table that does not use decomposition. All runs use $U = opt + 1$.

We employ the standard parameter settings (used for all experiments so far) for all instances but t60k, for which we set the target number of edges per subproblem to $\lceil m/(150U) \rceil$ instead of the usual $\lceil m/(4U) \rceil$. This is a planar mesh in which most vertices have degree three. This is challenging for our algorithm because trees from the tree packing cannot "cross" the flow, which often causes large fractions of the vertices to be unreachable from either A and B (the assigned vertices). With so much deadweight, the packing bound becomes much less effective.

Table 3.6: Performance on additional large instances with $\epsilon = 0$, starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds.

CLASS	NAME	n	m	opt	BB	TIME
cgmesh	dolphin	284	846	26	101	0.16
	mannequin	689	2 043	61	3 217	5.26
	venus-711	711	2 127	43	182	0.43
	beethoven	2 521	7 545	72	388	2.36
	venus	2 838	8 508	83	443	3.35
	cow	2 903	8 706	79	1 305	7.85
	fandisk	5 051	14 976	137	5 254	68.10
	blob	8 036	24 102	205	756 280	17 127.04
	gargoyle	10 002	30 000	175	18 638	572.57
	face	12 530	36 647	174	48 036	1 704.89
	feline	20 629	61 893	148	13 758	443.24
	dragon-043571	21 890	65 658	148	64 678	4 111.25
	horse	48 485	145 449	355	71 616	12 552.44
road	ny	264 346	365 050	18	976	380.98
	bay	321 270	397 415	18	537	248.13
	col	435 666	521 200	29	3 672	2 164.13
	fla	1 070 376	1 343 951	25	901	1 640.38
	nw	1 207 945	1 410 387	18	166	463.35
	ne	1 524 453	1 934 010	24	206	751.48
	cal	1 890 815	2 315 222	32	733	2 658.27
steinlib	alue5067	3 524	5 560	30	574	2.33
	gap3128	10 393	18 043	52	942	10.89
	diw0779	11 821	22 516	49	150	3.14
	fnl4461fst	17 127	27 352	24	402	8.08
	es10000fst01	27 019	39 407	22	452	14.84
	alut2610	33 901	62 816	93	802	43.74
	alue7065	34 046	54 841	80	4 080	194.52
	alue7080	34 479	55 494	80	4 065	200.88
	alut2625	36 711	68 117	99	791	50.36
	lin37	38 418	71 657	131	14 989	1 184.92

Note that two of the instances (delaunay_n15 and t60k) took months of CPU time. For them, we ran a distributed version of the code using the DryadOpt framework [17]. DraydOpt is written in C# and calls our native C++ code to solve individual nodes of the branch-and-bound tree. The distributed version was run on a cluster where each machine has two 2.6 GHz dual-core AMD

Table 3.7: Performance on harder instances with $\epsilon = 0$ (except for taq1021.5480), starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds. The setup varies depending on the instance; see text for details.

CLASS	NAME	n	m	opt	BB	TIME
delaunay	delaunay_n15	32 768	98 274	320	126 053 466	45 358 801
exact	taq1021.5480	1 021	5 480	1 650	3 102 902	105 973
ptv	bel	463 514	591 882	80	34 280	27 049
	nld	893 041	1 139 540	54	17 966	26 466
rgg	rgg17	131 072	728 753	517	34 800	64 072
	rgg18	262 144	1 547 283	823	192 966	671 626
streets	belgium	1 441 295	1 549 970	72	59 562	192 064
	netherlands	2 216 688	2 441 238	45	8 330	56 052
walshaw	t60k	60 005	89 440	79	56 681 055	14 378 268

Opteron processors, 16 GB of RAM, and runs Windows Server 2003. We used 100 machines only, and report the total CPU time (the sum of the times spent by our C++ code on all cores). Note that this excludes the communication overhead, which is negligible. The remaining instances in the table were solved sequentially.

To the best of our knowledge, none of these instances has been provably solved to optimality before. Solutions matching the optimum were known for t60k [103], taq1021.5480 [5], and rgg17 [93]. We are not aware of any published solutions for belgium and netherlands. For the remaining three instances, we improve the best previously known solutions, all found by the state-of-the-art heuristic of Sanders and Schulz [93]: 867 for rgg18, 81 for bel, and 64 for nld. (These are the best results for multiple executions of their algorithm; the average results are 1151, 104, and 120, respectively.) This shows that, in some cases, our exact algorithm can be a viable alternative to heuristics. For nld, we improved the best known result by more than 18%, probably because the heuristics may have missed the fact that one of the cells in the optimum solution is disconnected. As we have seen, for smaller or more regular instances, state-of-

the-art heuristics can find solutions that are much closer to the optimum. Even in such cases, our algorithm can still be useful to calibrate the precise quality of the solutions provided.

3.11.3 Very Big Graphs

For a few graphs the reduction in processing time was great. We solved for the first time gigantic road networks like Italy and USA, and found the best known solution for Europe. Table 3.8 give a summary of the results.

Table 3.8: Performance on large road network instances with $\epsilon = 0$, starting from U = best previously known bound + 1; TIME *is the total CPU time, and* PREPROCESS *is the time to build G^{flow} and G^{tree}. Europe was not proven optimal, but we have the best known solution.*

NAME	n	m	*opt*	TIME	PREPROCESS
Italy	6.6M	7M	38	8 hours	2 hours
USA	24M	29M	87	12 days	9 hours
East	3.6M	4.3M	65	13 hours	1 hour
Europe	18M	22M	150*	5 days	7 hours

The instances from Table 3.8 are very large and were not expected to be solved by an exact method. Note that the value of the solution is highly correlated with how hard a problem is.

For Europe we found a decomposition with 170 subproblems (that was the best known solution), and solved the first few branch-and-bound nodes of each of them to find the most promising to have the optimal solution. We then proceeded to solve that subproblem to optimality. It is possible that a better solution exists.

3.11.4 Fractional Flows

By using fractional flows as in Section 3.10 we solved some low degree instances for the first time, and greatly reduced the time to solve other instances. See Table 3.9.

Table 3.9: Performance on instances with small average degree, $\epsilon = 0$, starting from U = best previously known bound + 1, and using $k = 2$ parallel edges for each original edge; BB *is the number of branch-and-bound nodes, and* TIME *is the total CPU time.*

				fractional flow		integral flow	
NAME	n	m	*opt*	BB	TIME	BB	TIME
t60k	60K	90K	79	71K	12 hours	56M	5.5 months
lks	2.8M	3.4M	77	81K	14 days	-	DNF

A few things to note about Table 3.9. First is the huge improvement seen in instance t60k when compared to our previous solution. Second, the best known (heuristic) solution for lks was 188. Our solution has a value of about 40% of the previously best known solution.

3.11.5 Parameter Evaluation

We now discuss the relative importance of some of the techniques introduced in this article. One of our main contributions is the packing lower bound. Although we also use the (known) flow bound, it is extremely weak by itself, particularly when most the nodes are assigned to the same side. As a result, almost all instances we tested would not finish without the packing bound, even if we allowed a few hours of computation. The only exceptions are tiny instances, which can be solved but are orders of magnitude slower.

In the remainder of this section, we evaluate other important aspects of the

algorithm: the decomposition technique, forced assignments, and the branching criterion. In every case, we pick a small set of instances for illustration (chosen to highlight the differences between the strategies), and always run the full algorithm with $U = opt + 1$ and the same value of ϵ as in our main experiments (usually 0). We set a time limit of 12 hours for the experiments in this section.

Table 3.10: Total running times (in seconds) of our algorithm on assorted instances with different decomposition strategies: no decomposition, random partition of the edges, using BFS-based clumps, and using both flow-based and BFS-based clumps.

CLASS	NONE	RANDOM	BFS	FLOW
G124.04	14.52	499.99	315.36	366.25
alue5067	441.22	225.46	3.12	2.61
cow	572.23	5537.97	12.41	8.53
delaunay_n10	25.94	191.46	4.22	4.02
delaunay_n11	658.88	4885.87	19.86	18.35
dragon-043571	DNF	DNF	21744.69	4128.44
gargoyle	DNF	DNF	1248.64	578.62
mannequin	57.17	198.18	5.40	6.18

We first consider the decomposition technique. Table 3.10 compares the total running time of our algorithm when different decomposition techniques (introduced in Section 3.6) are used. We consider four different approaches: (1) no decomposition; (2) random decomposition (each edge is independently assigned to each of the U subproblems at random); (3) decomposition with BFS-based clumps; and (4) decomposition with flow-based and BFS clumps. All remaining aspects of the algorithm remain unchanged. Note that the standard version of our algorithm (used in previous experiments) automatically picks either strategy (1) or strategy (4). Strategies (2) and (3) are shown here for comparison only.

As anticipated, decomposition is useful, but only for a subset of the instances. Instances with relatively large bisections, such as G124.04 (a random graph) become significantly slower if decomposition is used: it has to solve

many subproblems, each about as hard as the original ones. In contrast, decomposition is helpful for instances with smaller bisections, such as meshes (cow, mannequin, dragon-043571, and gargoyle), grid graphs with holes (alue5067), and Delaunay triangulations. The way we distribute edges to subproblems is important, however. Random distribution is not enough to make the subproblems much easier; creating clumps is essential. Flow-based clumps have only minor effect in most cases, but never hurt much and are occasionally quite helpful (as in dragon-043571, which is a long and narrow mesh).

Table 3.11: Total running times (in seconds) on assorted instances using different combinations of forced-assignment techniques (based on flows, extended flows, and subdivisions).

CLASS	NONE	FLOW	EXTENDED	SUBDIV	FLOW+SUBDIV	FULL
G124.04	14.91	14.48	15.30	14.82	14.64	14.65
alue5067	10.63	3.10	2.65	17.44	3.09	2.62
cow	24.43	9.23	8.61	23.72	9.29	8.51
delaunay_n10	4.82	3.62	3.93	4.91	3.48	3.53
luxembourg	DNF	42.72	38.76	DNF	39.33	39.44

Table 3.12: Total running times (in seconds) on assorted instances using different branching techniques. All columns use degree as a branching criterion, by itself (column DEGREE*) or in combination with one additional criterion (columns* TREE, SIDE, DISTANCE, CONNECTED*). The last column refers to our default branching criterion, which combines all five methods.*

CLASS	DEGREE	TREE	SIDE	DISTANCE	CONNECTED	ALL
G124.04	24.69	27.14	24.59	15.85	24.44	15.12
ak2010	560.65	24.04	458.03	2088.31	78.46	6.76
alue5067	5.34	2.71	4.89	2.93	5.37	2.51
cow	8.99	9.32	8.74	8.85	8.91	8.62
delaunay_n10	4.10	4.05	4.06	4.01	4.13	4.07
hi2010	895.37	DNF	574.30	351.95	24.87	42.48

Another important contribution is the notion of *forced assignments*. We proposed three strategies: flow-based (Section 3.5.1), extended flow-based (Section 3.5.2), and subdivision-based (Section 3.5.3). Table 3.11 compares total running times on some instances using all possible combinations of these methods: (1) no forced assignments; (2) flow-based only; (3) extended flow-based only; (4)

subdivision-based only; (5) flow-based + subdivision-based; (6) extended flow-based + subdivision-based (the default version used in our experiments). Note that other combinations are redundant, since extended flow-based assignments are strictly stronger than flow-based assignments.

The table shows that forced assignments are crucial to make our method robust. While it does not help much in some cases (such as random graphs), it introduces very little overhead. For other instances, forgoing forced assignments makes the overall algorithm orders of magnitude slower (this is the case for luxembourg, a road network). Forced assignments are especially helpful in guiding the algorithm towards the optimum bisection (the actual balanced cut). Note that most gains come from the simpler flow-based strategy; extended flows and subdivisions are generally helpful, but not crucial.

Finally, we evaluate our branching criteria. Recall, from Section 3.7.5, that we evaluate each potential branching vertex using a combination of five criteria: the *degree* of the vertex, *weight* of the tree it belongs to, which *side* it is reachable from in G_f, its *distance* to the closest assigned vertex, and the total weight of its *connected* component. Among those, the degree is absolutely crucial: our algorithm becomes orders of magnitude slower without this criterion. In Table 3.12 we consider the effect of each of the remaining methods (when applied in conjunction with degree) on the total running time of our algorithm.

Although branching based only on degrees is often good enough (as in cow and delaunay_n10), additional criteria usually have a significant positive effect. In particular, taking components into account is crucial for disconnected instances (such as ak2010 and hi2010, redistricting instances representing Alaska and Hawaii), and distance information can be helpful in instances with rela-

tively flat degree distributions, such as G124.04 (a random graph). We stress, however, that these are heuristics, and sometimes they actually hurt, as when one uses only tree weights (in addition to degrees) on hi2010 or distances for ak2010. On balance, however, the combination of all five criteria leads to a fairly robust algorithm.

3.12 Conclusion

We have introduced new lower bounds for graph bisection that provide excellent results in practice. They outperform previous methods on a wide variety of instances, and find provably optimum bisections for several long-standing open instances (such as U500.20 [61]). While most previous approaches keep the branch-and-bound tree small by computing very good (but costly) bounds at the root, our bounds are only useful if some vertices have already been assigned. This sometimes causes us to branch more, but we usually make up for it with a faster lower bound computation. A large number of nodes is also ideal for distributed computing since the branch-and-bound nodes are almost independent (with the exception of the global upper bound).

A notable characteristic of our approach is that it relies strongly on heuristics at various steps, such as generating clumps for decomposition, creating valid tree packings, and picking branching vertices. A natural direction for future research is to improve these heuristics so as to realize the full potential of the main theoretical techniques we introduce (packing bound and decomposition). For several classes of instances (such as Delaunay triangulations), it is likely that better methods for generating trees and clumps will lead to much tighter

bounds (and smaller branch-and-bound trees) in practice.

For some other graph classes (such as high-expansion graphs), however, the tree packings generated by our heuristics are already perfectly balanced. For those, improvements would have to come from additional theoretical insights. Of course, it would be straightforward to create a hybrid algorithm that simply applies another technique (based on semidefinite programming or multicommodity flows, for example) when confronted with such instances. A more interesting question is whether bounds based on such techniques can be combined in a nontrivial way with the ones we propose here. Our decomposition technique, in particular, could be used in conjunction with any exact algorithm for graph bisection, though it is not very effective for such instances.

Another avenue for future research is proving nontrivial bounds for the running time of our algorithm (or a variant) for some graph classes. In particular, Demaine et al. [33] show that one can partition the edges of a minor-free graph into k classes (for any integer k) so that contracting all edges of any class leads to a graph with treewidth $O(k)$. This implies an exact algorithm for the minimum bisection problem on minor-free graphs with running time $O(2^{O(opt)}n^{O(1)})$ [60]. (More recently, Cygan et al. [26] have shown that the problem can be solved in $O(2^{O(opt^3)}n^3 \log^3 n)$ for general graphs.) Although far from practical, this theoretical algorithm has some parallels with our approach, and may potentially shed some light into the good empirical performance we observe.

CHAPTER 4

A METHOD FOR EXTENDING TSP APPROXIMATION ALGORITHMS TO SOLVE A PRIORI INSTANCES

4.1 Introduction

It is hard to overestimate the scientific community interest in the traveling salesman problem (TSP). For decades journals have been seeing a myriad of works presenting exact solutions [3, 27], heuristics [25, 76], and approximation algorithms [9, 22] for the TSP or for extensions or restrictions of it. The TSP is one of the quintessential NP-Complete problems with many books devoted exclusively to studying its structure and possible solutions [4, 23, 36, 89].

In this chapter we will study the *a priori TSP*, which is a two-stage stochastic extension of the TSP. The first stage of the a priori TSP consists on computing a tour π over all cities. In the second stage a particular subset A of the cities is sampled according to a known distribution P and π is shortcut to cover only the cities in A. The goal of the problem is to determine a tour π that minimizes the expected length of the shortcut tour. The a priori TSP will be defined formally in Section 4.2.

The a priori TSP was motivated by the "Meals On Wheels" program of Senior Citizen Services Inc., which prepares lunches for elderly or ill people who are unable to cook for themselves. The set of clients who should actually be served at a particular day is very volatile as "[They] may die, or recover from illness, or receive care elsewhere" [12].

This chapter presents a method that can use an λ-approximative algorithm

$\mathcal{A}$ for the TSP and using it as a black box, provide a $K\lambda$ approximation to an a priori TSP instance, where K is the size of the support of the distribution P, i.e. the number of subsets A such that $P(A) > 0$.

The rest of this chapter proceeds as follows. In Section 4.2 we will formally define the a priori metric TSP and will introduce some notation. In Section 4.3 we will discuss the state-of-the-art on the a priori TSP. In Section 4.4 we will present our method and prove some of its properties.

4.2 Preliminaries

The metric Traveling Salesman Problem (TSP) is defined as follows:

Input A metric space (S, d) with size $N = |S|$.

Output A list $\pi = [\pi_0, \ldots, \pi_{N-1}]$ containing each element of S once that minimizes $d(\pi) = \sum_{i=0\ldots N-1} d(\pi_i, \pi_{i+1})$ where the indices of π are taken modulo N.

Each element of S is usually called a "city", this nomenclature coming from the original motivation to the problem, where a traveling salesman has to visit a list of cities and return to his home town in the shortest possible time [27].

A central operation in several approximation algorithms and heuristics for the TSP is the notion of a shortcut, which allows us to reduce a list π of elements in S, possibly with repetitions to a valid solution of the TSP with input (A, d) where $A \subseteq S$ by "ignoring useless elements of π".

Definition 5. *Fix a set $A \subseteq S$ and let $\pi = [\pi_0, \ldots, \pi_{|\pi|-1}]$ be a list of elements of S*

that covers A, i.e. each element of A appears in π at least once. Note that π can contain repetitions and that it may not cover S.

Let $\pi' = [\pi'_0, \ldots, \pi'_{|A|-1}]$ be the elements of A in the order they first appear in π. We call π′ the "restriction of π to A", or "π shortcut to A" and denote it by $\pi|_A$.

A list π that contains each element of S exactly once will be called a *permutation* of S. Note that $\pi|_A$ is a permutation of A. The following lemma shows why $\pi|_A$ is useful in the design of algorithms for the TSP.

Lemma 7. *Let π be a list of elements of S covering $A \subseteq S$. Then $d(\pi|_A) \leq d(\pi)$.*

Proof. *Let $\pi = [\pi_0, \ldots, \pi_{|\pi|-1}]$, and let $i_0 \leq \ldots \leq i_{|A|-1}$ be a set of indices such that $(\pi|_A)_k = \pi_{i_k}$, i.e. i_k gives the indices of the first appearances of each element of A in π. Then:*

$$
\begin{aligned}
d(\pi_A) &= \sum_{k=0}^{|A|-1} d((\pi_A)_k, (\pi_A)_{k+1}) \\
&= \sum_{k=0}^{|A|-1} d(\pi_{i_k}, \pi_{i_{k+1}}) \\
&\leq \sum_{k=0}^{|A|-1} \sum_{j=i_k}^{i_{k+1}-1} d(\pi_j, \pi_{j+1}) && \textit{By the triangle inequality} \\
&= d(\pi)
\end{aligned}
$$

Where $k+1$ is taken modulo A and $j+1$ is taken modulo $|\pi|$. The inner sum of the last line "wraps around" $|\pi|$; If b ¡ a $\sum_{j=a}^{b} f(j)$ is defined as $f(a) + \ldots + f(|\pi|-1) + f(0) + \ldots + f(b)$. □

We now define formally the a priori TSP problem.

Input A metric space (S, d) and a probability distribution P over subsets of S.

Output A permutation π of S that minimizes $E[d(\pi|_A)]$ where $A \sim P$.

Note that while shortcutting is only important in the design of algorithms for the TSP, it is part of the definition of the a priori TSP, where it plays a central role. Lemma 7 shows that $E[d(\pi|_A)] \leq d(\pi)$ for any metric space (S, d), distribution P, and π that covers S.

4.3 Literature Review

The a priori TSP is a family of problems as defined above, as it does not specify how P is given in the input. In this section we present several possibilities.

Suppose first the distribution P is given as a black box polynomial algorithm that returns samples from P. This is the variant with the least amount of information.

In this case there exists a randomized $O(\log n)$-approximative algorithm [94]. There is also no deterministic λ-approximative algorithm with $\lambda = o(\log n)$ [51]. So up to a constant factor and randomization this problem is solved with respect to approximation algorithms.

It is interesting to mention that the algorithm presented in [94] does not sample from P. Instead, it computes π that is $O(\log n)$-approximative in expectation for any distribution P, by approximating the input metric (S, d) with a tree metric [37].

It is also worth noting that construction that proves that the a priori TSP with

black box distribution model cannot be approximated better than $O(\log n)$ has a support of 2, i.e. $P(A) \neq 0$ only for 2 different $A \subseteq S$.

Now let P be such that if $A \sim P$ then events $A_x = \{x \in A\}$ are pairwise independent, and let $Pr(A_x) = p_x$. We call this the *independent activation* model. In this model there is a randomized 4-approximative algorithm, even if the distribution P is given as a black box [98]. Furthermore if p_x are known explicitly then a deterministic 8-approximative algorithm is known [98].

In the following sections we will discuss a method to find approximative solutions to the a priori TSP when P is given explicitly, that is you are given $(A, P(A))$ for all subsets $A \subseteq S$ with $P(A) \neq 0$.

4.4 Solving the A Priori TSP With Explicitly Known P

In this section we will show a method to solve the a priori TSP when the distribution P is given explicitly. Formally, we want to find the following problem.

Input A metric space (S, d) and tuples (A_i, p_i) for $i = 1 \dots K$ where $A_i \subseteq S$, $p_i > 0$ and $\sum_i p_i = 1$.

Output A permutation π of S that minimizes $\sum_i d(\pi|_{A_i})p_i = E[d(\pi|_A)]$.

Note that if $Q = S \setminus \bigcup_i A_i \neq \emptyset$ then the position of the cities of Q in a permutation π of S will not alter $E[d(\pi|_A)]$. Therefore we assume without loss of generality that $Q = \emptyset$.

We will use extensively the following operation, that combines permutations π_1 and π_2 of almost disjoint sets A_1 and A_2 into one permutation π of $A_1 \cup A_2$.

Definition 6. *Let π^i be a permutation of A_i for $i = 1, 2$ and let $A_1 \cap A_2 = \{a\}$ be a singleton. Let k_i be indices such that $\pi^i_{k_i} = a$.*

The permutation $\pi = [a, \pi^1_{k_1+1}, \pi^1_{k_1+2}, \ldots, \pi^1_{k_1+|A_1|-1}, \pi^2_{k_2+1}, \ldots, \pi^2_{k_2+|A_2|-1}]$ over $A_1 \cup A_2$ is called the concatenation *of π^1 and π^2 and is denoted $\pi = \pi^1 \oplus \pi^2$. (Indices of π^i are taken modulo $|A_i|$.) Note that $|\pi| = |\pi^1| + |\pi^2| - 1$.*

We will now prove a key property of concatenations. Let $\pi^1 \odot \pi^2$ as the list $[\pi^1_0, \ldots, \pi^1_{|\pi^1|-1}, \pi^2_0, \ldots, \pi^2_{|\pi^2|-1}]$. Note that $|\pi^1 \odot \pi^2| = |\pi^1| + |\pi^2|$.

Lemma 8. *Let $\pi = \pi^1 \oplus \pi^2$. Then $d(\pi) \leq d(\pi^1) + d(\pi^2)$.*

Proof. *Let $a = \pi_0$ be the unique element covered both by π^1 and π^2. Assume without loss of generality that $\pi^i_0 = a$, since $d(\pi) = d(\pi')$ for any cyclic permutation π' of π.*

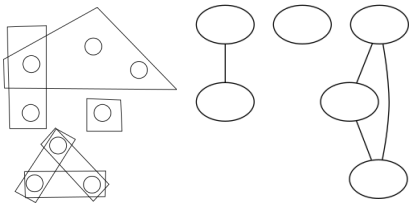
Let $\tilde{\pi} = \pi^1 \odot \pi^2$. Trivially $d(\tilde{\pi}) = d(\pi^1) + d(\pi^2)$ since $\pi^1_0 = \pi^2_0 = a$. Note that π is the restriction of $\tilde{\pi}$ to $A = A_1 \cup A_2$. Lemma 7 gives $d(\pi) \leq d(\tilde{\pi})$. □

Intuitively the algorithm will find a set of permutations π^i that are approximations to the optimal solution of instances of the TSP. The concatenation of all π^i will provide the required approximation guarantee to the a priori TSP. Before we can present the algorithm we need some more notation.

Let a metric space (S, d) and tuples (A_i, p_i) for $i = 1 \ldots K$ be an input tot he a priori TSP. We will assume without loss of generality that $p_1 \geq p_2 \geq \ldots \geq p_K$. Build a graph $H = (V, E)$ where $V = \{1, \ldots, K\}$ and $E = \{(i, j) : A_i \cap A_j \neq \emptyset\}$. For $B \subseteq V$ define $A(B) = \cup_{i \in B} A_i$ as the cities covered by B. See Figure 4.1 for an example.

Let H_k be the subgraph of H generated by nodes $1, \ldots, k$, and let H_0 be the empty graph. Let also C_k be the set of connected components of H_k, in other

words, $C_k = \{C_k^1, \ldots, C_k^l\}$ where $C_k^j \subseteq \{1, \ldots, k\}$ mark the nodes of a connected component of H_k. Note that $C_0 = \emptyset$.



(a) Subsets $A_1, \ldots, A_6$ (b) Corresponding graph H

Figure 4.1: Example of graph H

Finally, let $D_k = \{C \in C_{k-1} : A_k \cap A(C) \neq \emptyset\}$ be the set of connected components of H_{k-1} that are connected to k in H_k, and let $L_k = \{k\} \cup \left(\bigcup_{C \in D_k} C\right)$. Note that $C_k = (C_{k-1} \setminus D_k) \cup \{L_k\}$. For each $C \in D_k$ let $C^r = \max_{i \in C} i$ be the representative of C, and let $v_k(C) \in A_k$ be any element such that $v_k(C) \in \cup_{i \in C} A_i$. Note that $v_k(C)$ exists by the definition of D_k. We will abuse notation and say that $A(D_k) = \cup_{C \in D_k} A(C)$.

We are ready to present the algorithm.

Algorithm 5 Solving the A Priori TSP

```
1:  function COMBINE(x, D)
2:      π ← x
3:      for C ∈ D do
4:          π ← π ⊕ π_{C^r}
5:      return π

6:  function SOLVE((S, d), (A_i, p_i), 𝒜)      ▷ Where 𝒜 is an algorithm for the TSP
7:      for k = 1 ... K do
8:          B̄_k ← (A_k \ (⋃_{l<k} A_l))
9:          B_k ← B̄_k ∪ (⋃_{C∈D_k} v_k(C))
10:         π_k ← COMBINE(𝒜(B_k), D_k)
11:     return ⊙_{C∈C_K} π_{C^r}
```

The joining on line 11 is done on any order. Note that $\bar{B}_k$ is the set of cities

that appear for the first time in A_k.

Lemma 9. *Let $\bar{B}_k$ and B_k be as computed in Algorithm 5, and $\bar{D} \subseteq D_k$. Then $A(\bar{D})$, $\bar{B}_k$, and $A(D_k \setminus \bar{D})$ are pairwise disjoint. Moreover $B_k \cup A(\bar{D})$ is the disjoint union of $\bar{B}_k$, $A(\bar{D})$, and $\{v_k(C) : C \notin \bar{D}\}$.*

Proof. *Suppose $x \in A(\bar{D}) \cap A(D_k \setminus \bar{D})$. Then $x \in A_i$ for some $i \in \bigcup_{C \in \bar{D}} C$ and $x \in A_j$ for some $j \in \bigcup_{C \in D_k \setminus \bar{D}} C$ with $i, j < k$. Then by definition of H we would have $(i, j) \in E$ which contradicts the fact that $D_k \subseteq C_{k-1}$. Therefore $A(\bar{D}) \cap A(D_k \setminus \bar{D}) = \emptyset$. Note also that $\bar{B}_k$ is disjoint to $A(D_k)$ by definition.*

For the second part of the lemma, note that $\bar{B}_k$, $A(\bar{D})$ and $\{v_k(C) : C \notin \bar{D}\}$ are clearly pairwise disjoint. Let $X = B_k \cup A(\bar{D})$ and $Y = \bar{B}_k \cup A(\bar{D}) \cup \{v_k(C) : C \notin \bar{D}\}$.

Let $x \in B_k$. Then either $x \in \bar{B}_k$ or $x = v_k(C)$. In the latter case $C \in \bar{D}$ then $x \in A(\bar{D})$, and if not $x \in \{v_k(C) : C \notin \bar{D}\}$. In any case $x \in Y$, hence $X \subseteq Y$. The converse is trivially true by the definition of B_k. □

Corollary. *The concatenation on line 4 in Algorithm 5 is always well defined, and that* COMBINE *returns a permutation over $x \cup A(D)$.*

Lemma 10. *π_k as computed in Algorithm 5 is a permutation of $A(L_k)$.*

Proof. *We will prove by induction. For $k = 1$ note that $D_k = D_1 \subseteq C_0 = \emptyset$, and $B_1 = A_1 = A(L_1)$. Therefore $\pi_1 = \mathcal{A}(A(B_1))$ is a permutation of $A(L_1)$.*

Now let $k > 1$, and let $C \in D_k$. By induction we know that π_{C^r} is a permutation of $A(C)$. The rest of the proof is a simple consequence of the corollary of Lemma 9. □

Finally we are ready to prove the correctness of Algorithm 5, i.e. that it returns a permutation over S.

Theorem 7. *Function* SOLVE *of Algorithm 5 correctly returns a permutation of S.*

Proof. *The result is trivial if the graph H is connected. In this case $L_K = \{1, \ldots, K\}$ and Lemma 10 shows that the return is a permutation of $A(L_K) = S$.*

Suppose now that H is not connected. For each connected component C of H we know, again by Lemma 10 that π_{C^r} is a permutation of $A(C)$. Noting that $S = \bigcup_{C \in C_K} A(C)$ finishes the proof of the theorem. □

We have established the correctness of Algorithm 5 in Theorem 7. Note also that Algorithm 5 clearly runs in polynomial time assuming that $\mathcal{A}$ also does. We will now study the approximation guarantees of Algorithm 5, but first we will need some more notation.

Let $\mathcal{A}$ be a λ-approximative algorithm for the TSP, i.e. $d(\mathcal{A}(X)) \leq \lambda d(\pi^*(X))$ where $\pi^*(X)$ is an optimal solution for the instance (X, d). Let also π^* be an optimal solution for the a priori TSP. Note in particular that $d(\pi^*(X)) \leq d(\pi^*|_X)$ for all $X \subseteq S$.

Lemma 11. *Let π_k be as computed in Algorithm 5. Then $d(\pi_k) \leq \lambda \sum_{i \in L_k} d(\pi^*|_{A_i})$, for all k.*

Proof. *We will prove by induction. When $k = 1$ we have $L_1 = \{1\}$ and $\pi_1 = \mathcal{A}(A_1)$. So clearly $d(\pi_1) \leq \lambda d(\pi^*(A_1)) \leq \lambda d(\pi^*|_{A_1})$.*

Now let $k > 1$. Since $\bar{B}_k \subseteq A_k$ we have $d(\mathcal{A}(\bar{B}_k)) \leq \lambda d(\pi^*|_{A_k})$. By following COMBINE *in Algorithm 5 we have:*

$$
\begin{aligned}
d(\pi_k) &\leq d(\mathcal{A}(\bar{B}_k)) + \sum_{C \in D_k} d(\pi_{C^r}) && \text{By Lemma 8} \\
&\leq \lambda d(\pi^*|_{A_k}) + \sum_{C \in D_k} \lambda \sum_{i \in L_{C^r}} d(\pi^*|_{A_i}) && \text{By the induction hypothesis} \\
&= \lambda \sum_{i \in L_k} d(\pi^*|_{A_i})
\end{aligned}
$$

□

We are finally ready to prove the approximation guarantee.

Theorem 8. *Algorithm 5 is a $K\lambda$-approximation algorithm.*

Proof. *Assume without loss of generality that H is connected. Otherwise we are essentially running the algorithm in each component of H, and the result clearly follows. Let $\pi = \pi_K$ be the permutation returned by the algorithm, and note that $\pi|_{A_j} = \pi_j|_{A_j}$.*

$$
\begin{aligned}
E[d(\pi|_A)] &= \sum_{j=1}^{K} p_j d(\pi|_{A_j}) \\
&= \sum_{j=1}^{K} p_j d(\pi_j|_{A_j}) \\
&\leq \sum_{j=1}^{K} p_j d(\pi_j) && \text{By Lemma 8} \\
&\leq \lambda \sum_{j=1}^{K} p_j \sum_{i \in L_j} d(\pi^*|_{A_i}) && \text{By Lemma 11} \\
&= \lambda \sum_{i=1}^{K} d(\pi^*|_{A_i}) \sum_{j : i \in L_j} p_j \\
&\leq \lambda \sum_{i=1}^{K} d(\pi^*|_{A_i}) \sum_{j \geq i} p_j \\
&\leq \lambda \sum_{i=1}^{K} d(\pi^*|_{A_i}) \sum_{j \geq i} p_i && \text{Since } p_1 \geq p_2 \geq \ldots \geq p_K \\
&\leq \lambda \sum_{i=1^K} (K - i + 1) p_i d(\pi^*|_{A_i}) \\
&\leq K\lambda E[d(\pi^*|_A)]
\end{aligned}
$$

□

In the Example 5 we show that the guarantee of $K\lambda$ is tight at most up to a factor of 2.

Example 5. *Let (S, d) be the "worst case" of $\mathcal{A}$ with $N = |S|$. In other words, let $d(\pi) = \lambda d(\pi^*(S))$ where $\pi = \mathcal{A}(S)$. Extend the metric space (S, d) to (S', d') with $S' = S_1 \cup S_2 \cup \ldots \cup S_k$ where these unions are disjoint and for each $x \in S$ we have $x_i \in S_i$. Let $d'(x_i, y_j) = d(x, y)$ if $x \neq y$, and $d'(x_i, x_j) = |j - i|\epsilon$ where $\epsilon > 0$ is small enough. In other words, (S', d') is constructed by considering K copies of (S, d). Build an instance to the a priori TSP with support K where $A_i = S_1 \cup \ldots \cup S_i$ and $p_i = \frac{1}{K}$.*

Let $\pi^*(S) = [\alpha^0, \alpha^1, \ldots, \alpha^{N-1}]$. Note that $\rho^* = [\alpha_1^0, \alpha_2^0, \ldots, \alpha_K^0, \alpha_1^1, \ldots, \alpha_K^1, \ldots, \alpha_K^{N-1}]$ is a permutation of S'.

$$\sum_{i=1}^{K} p_i d(\rho^*|_{A_i}) = N(K-1)\epsilon + \sum_{i=1}^{K} \frac{d(\pi^*(S))}{K} = d(\pi^*(S)) + N(K-1)\epsilon$$

Let ρ be the return of Algorithm 5 given the input above. It is easy to see that one possible output is $\rho = [\beta_1^0, \beta_1^1, \ldots, \beta_1^{N-1}, \beta_1^0, \ldots, \beta_1^{N-1}, \ldots, \beta_K^{N-1}]$ where $\pi = [\beta^0, \beta^1, \ldots, \beta^{N-1}]$ is the return of $\mathcal{A}(S)$.

$$\sum_{i=1}^{K} p_i d(\rho|_{A_i}) = \sum_{i=1}^{K} \frac{id(\pi) + 2(i-1)\epsilon}{K} = \frac{(K+1)d(\pi)}{2} + (K-1)\epsilon$$

By $d(\pi) \leq \lambda d(\pi^*(S))$ we get $\frac{E[\rho|_A]}{E[\rho^*|_A]} \to \frac{(K+1)\lambda}{2}$ when $\epsilon \to 0$. This proves that the analysis in Theorem 8 is tight up to a constant.

If we are not provided with any structure of the metric d, the best known result is due to Christofides in 1976 [22], which gives $\lambda = \frac{3}{2}$. By using Christofides' algorithm as $\mathcal{A}$ we get a $\frac{3K}{2}$ for the a priori TSP with explicitly given distribution.

The $O(K)$ approximation seems like a weak result, but we remind the reader that the construction that proves it is impossible to get a $o(\log n)$ approximation with the black box model has $K = 2$. The strength of this result comes from the lack of dependency on n in the approximation guarantee.

APPENDIX A

PROOF OF THEOREM 1

Throughout this proof we will let l_a be the length of arc a. Note that $l_a = 0$ if $a \in H \cup V$.

We will start by proving a helping lemma.

Lemma 12. *If* $P_a(i, j) = [i', j']$ *then* $\Delta_a(i, j) \leq \Delta_a(i', j')$

Proof. *Let* $a = [i', j'] \rightarrow [i, j]$. *Then we know that* $S_a(i, j) = S_a(i', j') + l_a$ *and* $S_{a+1}(i, j) \geq S_{a+1}(i', j') + l_a$. *Therefore* $\Delta_a(i, j) \leq \Delta_a(i', j')$. □

Note that $S_{a+1}(a + 1, \cdot) = 0$ and by Lemma 1 $S_a(a + 1, j)$ is a nondecreasing function of j. This proves that the theorem is true for the row $a + 1$.

Fix $[i, j]$. We will prove that $\Delta_a(i, j) \leq \Delta_a(i, j + 1)$ and $\Delta_a(i, j) \leq \Delta_a(i - 1, j)$ by induction on i then on j using $i = a + 1$ as our base case.

First note that by induction hypothesis we have $\Delta_a(i, j - 1) \leq \Delta_a(i - 1, j - 1) \leq \Delta_a(i - 1, j)$ which are the three options for $[i', j'] = P_a(i, j)$. So $\Delta_a(i, j) \leq \Delta_a(i', j') \leq \Delta_a(i - 1, j)$ where the first inequality comes from Lemma 12. So we have established half the claim, that $\Delta_a(i, j) \leq \Delta_a(i - 1, j)$.

Now let $[i', j'] = P_a(i, j+1)$, and let $h = [i, j] \rightarrow [i, j+1]$, $d = [i-1, j] \rightarrow [i, j+1]$, and $v = [i-1, j+1] \rightarrow [i, j+1]$ be the possible values for P_{a+1}. If d does not exist let $l_d = -\infty$. We will abuse notation and identify the arcs with their first endpoint.

We say that $h < d < v$ meaning that h is preferred over d which is preferred over v. Let $x, y \in \{h, d, v\}$. Note that by induction hypothesis and the half of the

claim already established we have that $x < y$ implies $\Delta_a(x) \leq \Delta_a(y)$. In particular, if $x \in \{h, d, v\}$ we have $\Delta_a(x) \geq \Delta_a(h) = \Delta_a(i, j)$.

Let $x = P_a(i, j+1)$ and $y = P_{a+1}(i, j+1)$. Note that if $x = y$ we trivially have $\Delta_a(i, j+1) = \Delta_a(i', j') \geq \Delta_a(i, j)$. So let us assume $x \neq y$. If $x < y$ then:

$$S_a(x) + l_x \geq S_a(y) + l_y \qquad \text{From } x = P_a(i, j+1)$$

$$S_{a+1}(x) + l_x < S_{a+1}(y) + l_y \qquad \text{From } x > y = P_{a+1}(i, j+1)$$

$$\Delta_a(x) > \Delta_a(y)$$

Which is a contradiction, therefore x cannot be preferred over y. Now assume $x > y$. Then:

$$\begin{aligned} S_a(i, j+1) &= S_a(x) + l_x \\ &> S_a(y) + l_y \\ S_{a+1}(i, j+1) &= S_{a+1}(y) + l_y \\ \Delta_a(i, j+1) &> S_a(y) + l_y - S_{a+1}(y) - l_y \\ &= \Delta_a(y) \end{aligned}$$

, Therefore $\Delta_a(y) \geq \Delta_a(h) = \Delta_a(i, j)$ which concludes the proof of the theorem.

□

APPENDIX B

ALGORITHM EXAMPLES FOR BISECTION

B.1 Packing Bound

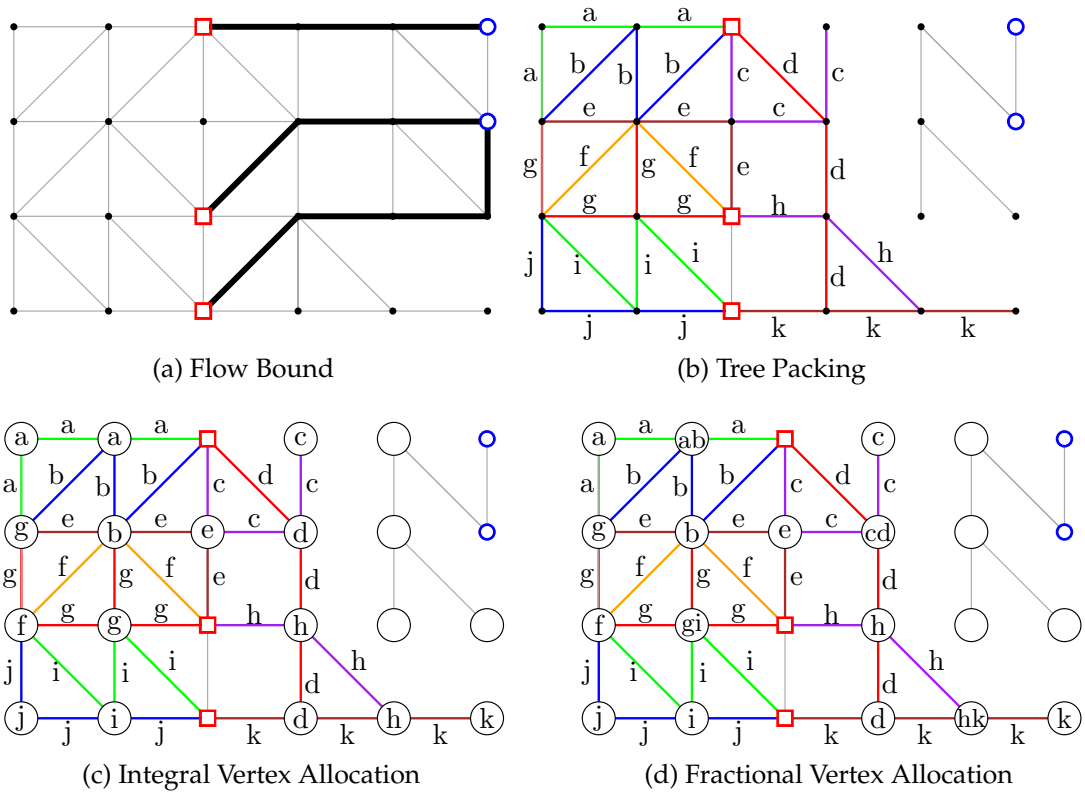
Figure B.1 illustrates our lower bounds. Assume that $\epsilon = 0$ and that all vertices have unit weight. Note that $W = 24$ and $W_- = W_+ = 12$, i.e., we must find a solution with 12 vertices on each side. For simplicity, we refer to vertices by their rows and columns in the picture: vertex (1,1) is at the top left, and vertex (4,6) at the bottom right.

Assume some vertices have already been assigned to A (red boxes) and B (blue disks). First, we compute a flow f (indicated in bold) from A to B (Figure B.1a). By removing these edges, we obtain the graph G_f. We then compute a tree packing on G_f, as shown in Figure B.1b. For each of the 11 edges (u, v) with $u \in A$ and $v \notin A$ we grow a tree (indicated by different colors and labeled $a, \ldots, k$) in G_f. Note that the deadweight (number of vertices unreachable from A) is 6, and that 15 free vertices are reachable from A. Finally, we allocate the weights of these 15 vertices to the trees.

Figure B.1c shows an allocation where each vertex is assigned in full to a single tree. This results in 7 trees of weight 1 (b, c, e, f, i, j, k), and 4 of weight 2 (a, d, g, h). Together, three of the heaviest trees have weight 6; with 6 units of deadweight, these trees are enough to reach the target weight of 12. Therefore, the packing bound is 3. Together with the flow bound, this gives a total lower bound of 6.

Figure B.1d shows an alternative allocation in which some vertices are split

equally among their incident trees. This results in 3 trees of weight 1 (e, f, j), and 8 trees of weight 1.5 (a, b, c, d, g, h, i, k). Now, we must add at least 4 trees to B to ensure its weight is at least 12. The packing bound is thus 4 and the total lower bound is 7, matching the optimum solution.



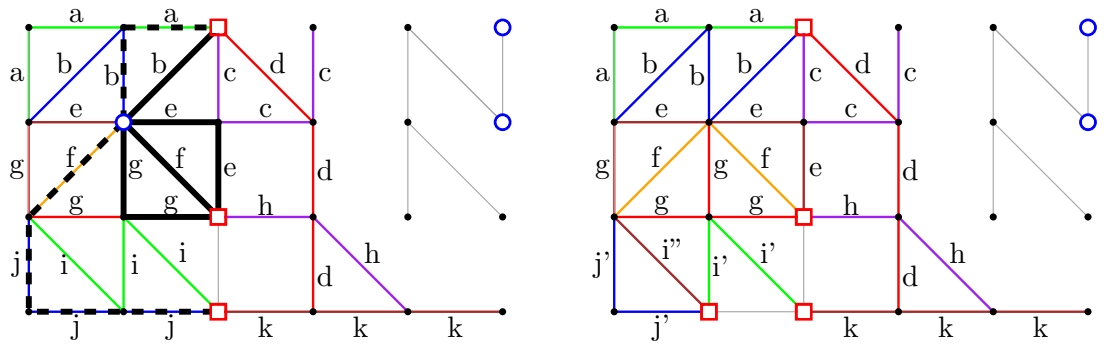
(a) Flow Bound (b) Tree Packing

(c) Integral Vertex Allocation (d) Fractional Vertex Allocation

Figure B.1: Example for lower bounds. Red boxes and blue circles are already assigned to A and B, respectively. The figures show (a) the maximum A-B flow; (b) a set of maximal edge-disjoint trees rooted at A; (c) an integral vertex allocation; and (d) a fractional allocation where vertices with two labels have their weights equally split among the corresponding trees.

B.2 Forced Assignment

Figure B.2 gives an example for our forced assignment techniques. We start from the tree packing in Figure B.1b. Figure B.2a shows how the flow-based forced assignment applies to vertex $(2, 2)$. It is incident to four trees (b, e, f, g). If



(a) Flow-based (b) Subdivision-based

Figure B.2: Examples of forced assignments. Figure (a) shows the additional flow that would be created if vertex $(2, 2)$ *were assigned to B (blue circles); solid edges correspond to the standard flow-based forced assignment and dashed edges to the extended version. Figure (b) shows (with primed labels) new trees that would be created if vertex* $(4, 2)$ *were assigned to A (red squares).*

it were assigned to B (blue circles), the flow bound would increase by 4 units, to 7. Using the extended flow-based forced assignment, we can increase the flow bound by another 2 units, sending flow along $f + j$ and $b + a$ to A (red squares). If the total lower bound, including the recomputed packing bound, is at least as high as the best solution seen so far, we can safely assign vertex $(2, 2)$ to A.

Figure B.2b illustrates our subdivision-based forced assignment. Consider what would happen if we were to assign vertex $(4, 2)$ to A. We implicitly split all trees incident to this vertex (i and j) into new trees i', i'', and j'. Tree i' is rooted at vertex $(4, 3)$, and the others at vertex $(4, 2)$. The vertex assignment remains consistent with the original one (as in Figure B.1c or B.1d, for example). We then recompute the packing bound for this set of trees. If the new lower bound is at least as high as the best solution seen so far, we can safely assign vertex $(4, 2)$ to B.

APPENDIX C

FIGURES FOR CHAPTER 3

We now present a few figures that demonstrate some of the concepts treated before, and then present a graphical representation for the optimal solution of several instances.

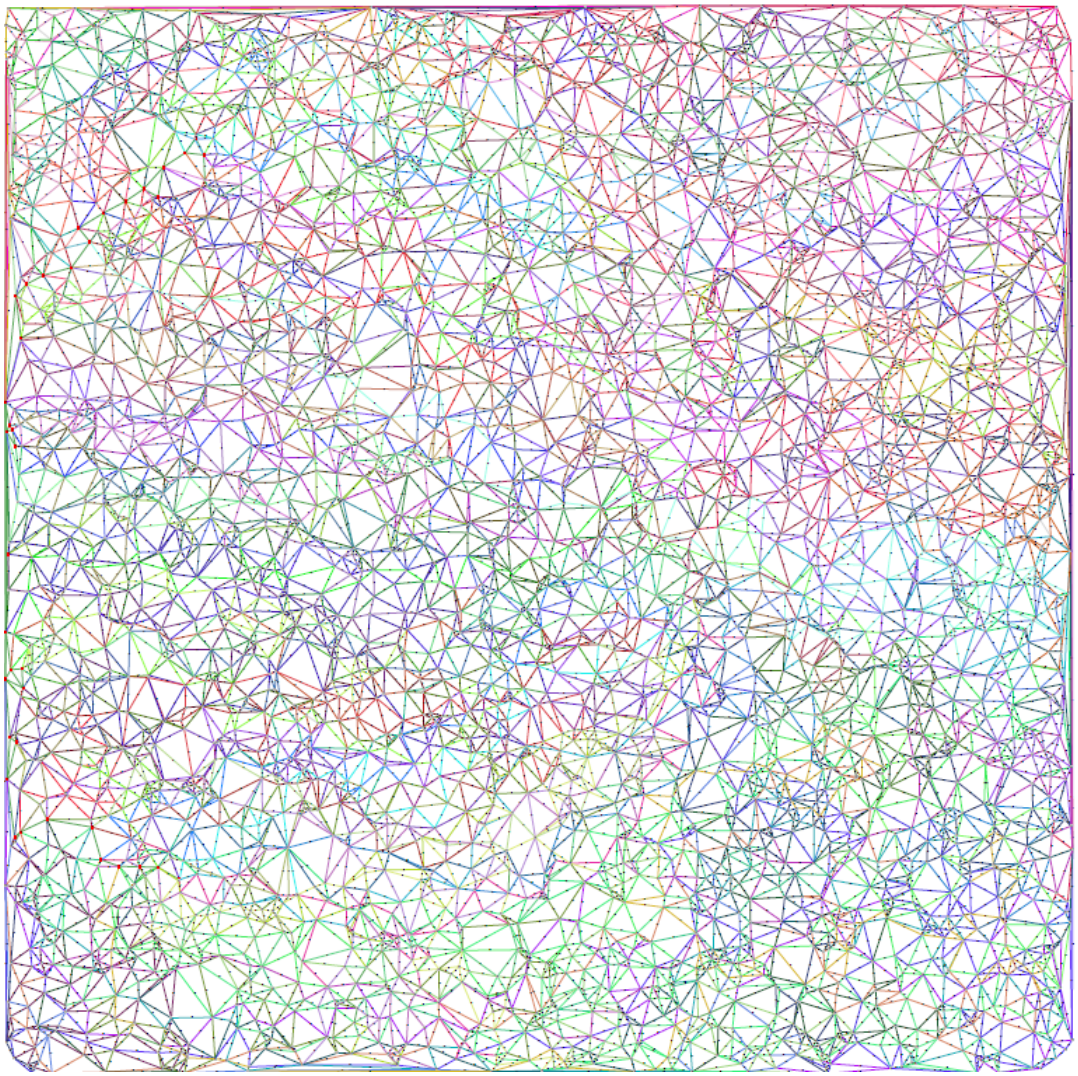
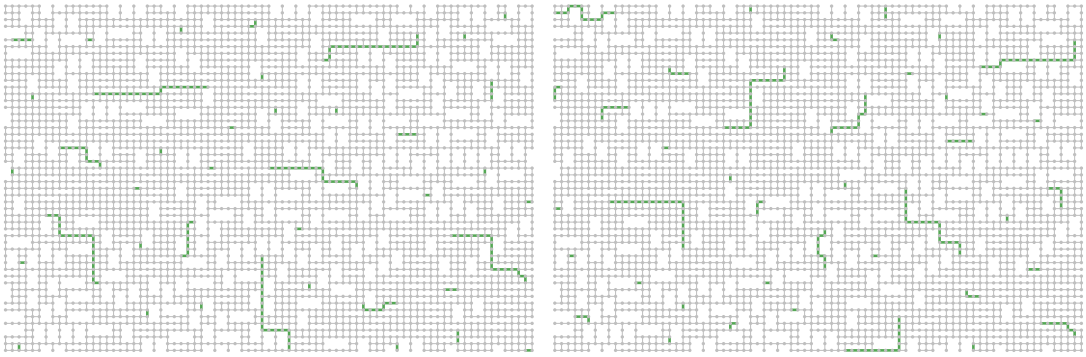


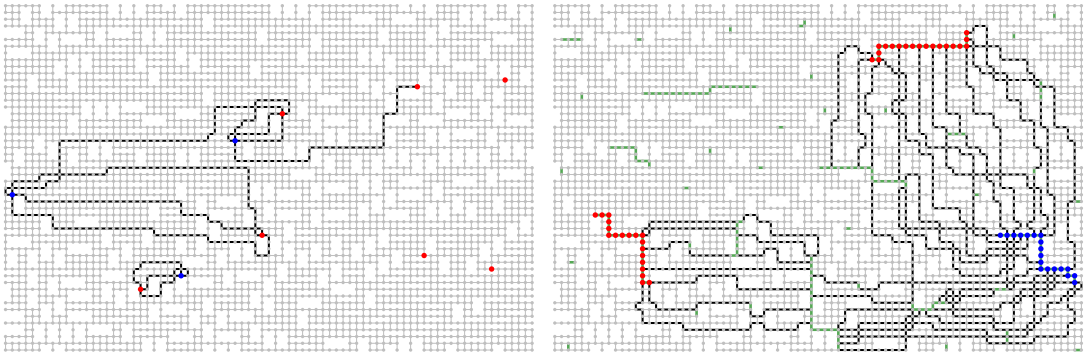
Figure C.1: Trees are represented in different colors. Note that trees intersect other trees in several places, making it easier to balance their weights with vertex fractional allocation, as well as doing forced assignments.



(a) Subproblem 1 of 30 (b) Subproblem 2 of 30

Figure C.2: Example of decomposition

As seen in Figure C.2 we generate nodes with very high degrees after edge contraction.



(a) Without Decomposition (b) With Decomposition

Figure C.3: Example of flow bound without and with decomposition.

Figure C.3 shows how useful decomposition can be. The total bound (flow + packing) of Figure C.3a is $17 = 11 + 6$ after fixing 10 vertices (i.e. at depth 10 of the branch-and-bound tree). The bound of Figure C.3b is $23 = 19 + 4$ after fixing only 3 vertices.

C.1 Solutions

Figure C.4 gives some intuition on why lks is such a hard instance. The problem is the existence of an almost balanced very small cut. Heuristic methods try to tweak the cut to make it balanced, while our approach without fractional flows give unsatisfactory packing bounds, since the region around the small cut will be full of flow edges, which will render them unreachable by our tree packing (i.e. they will be deadweight.) Instead, the optimal solution involves a not connected cell.

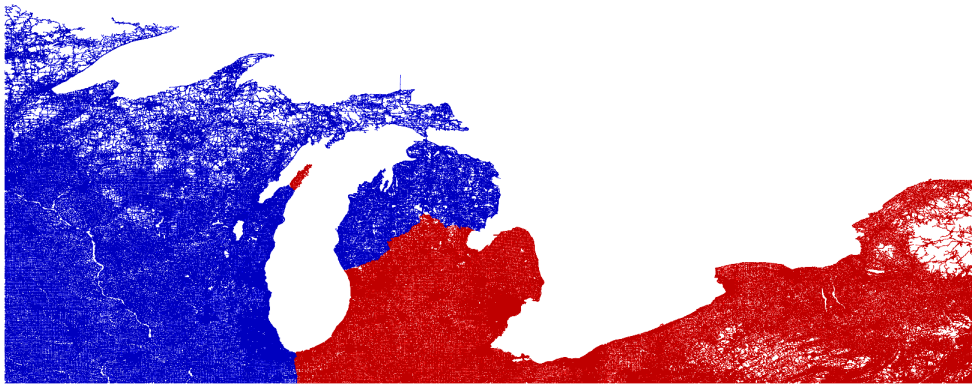
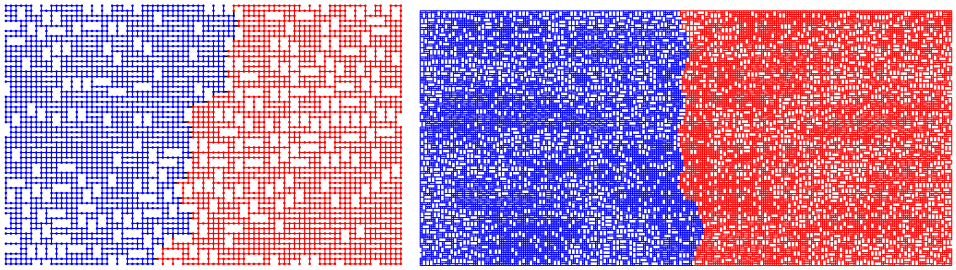


Figure C.4: Optimal bisection of lks which is a road map of the Great Lakes region of the USA.

Figure C.5 shows solutions arising from VLSI. These instances are generally solved well by our algorithm.

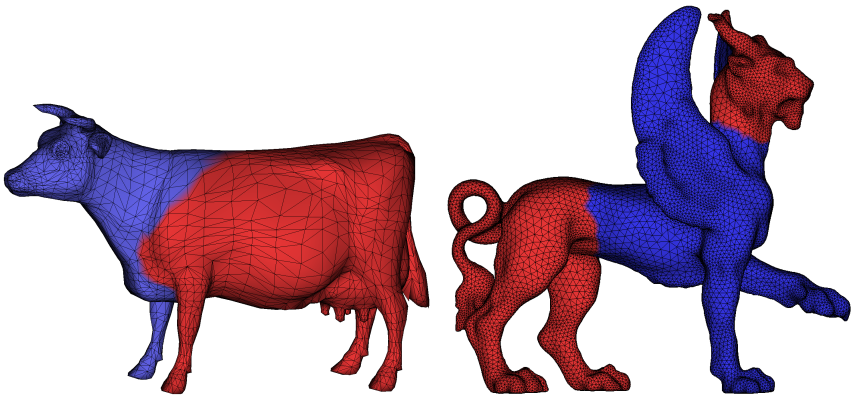


(a) alue5067 (b) alue7065

Figure C.5: VLSI instances

Figure C.6 shows optimal solutions for several 3d meshes. Note that the

optimal solution in Figure C.6b has a cell that is not connected. The reason for it is the big wings that are very heavy and highly connected.



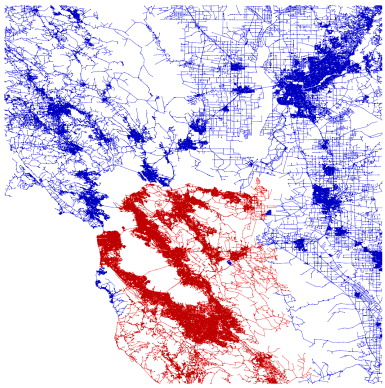
(a) cow (b) feline

(c) horse (d) dragon

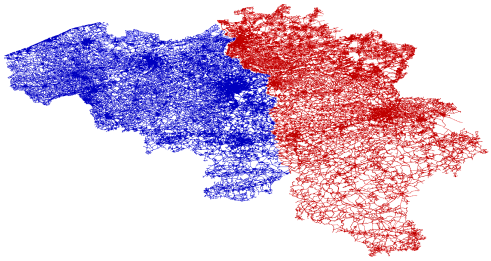
Figure C.6: Mesh Instances

Figure C.7 provides solutions for instances corresponding to road networks. These instances represent the shape of a highway by creating several vertices, which leads to a very sparse network (as several vertices have degree 2). Note that our algorithm finds geographical features like rivers in Figure C.7d. We reinforce that the only input to our algorithm is the graph, we do not take geographical information.

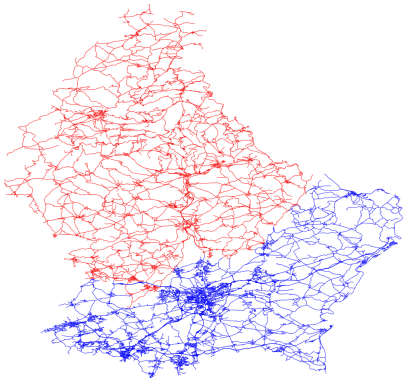
The last three figures show the largest instances we were able to solve. In Figure C.10 we can see how the Mississippi river is almost a perfect bisection. To gain the correct balance the optimal solution deviates slightly from it. In Figure C.9 note that we find the Alps, the Pyrenees, and a fjord that separates



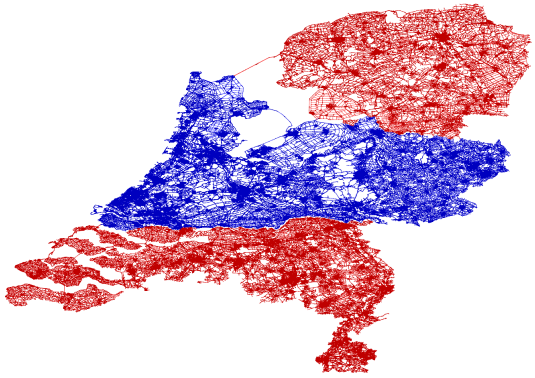
(a) bay



(b) belgium



(c) luxembourg



(d) netherlands

Figure C.7: Road Networks

Germany and Denmark.

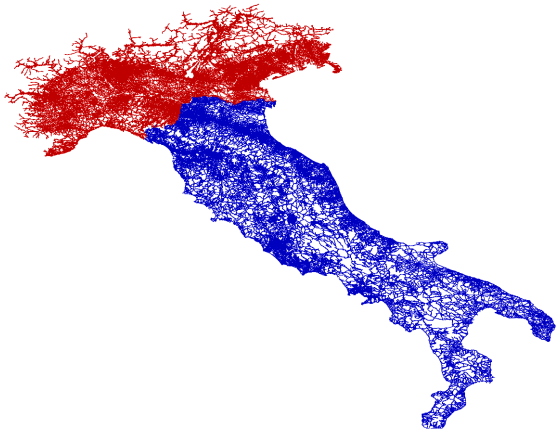


Figure C.8: Italy

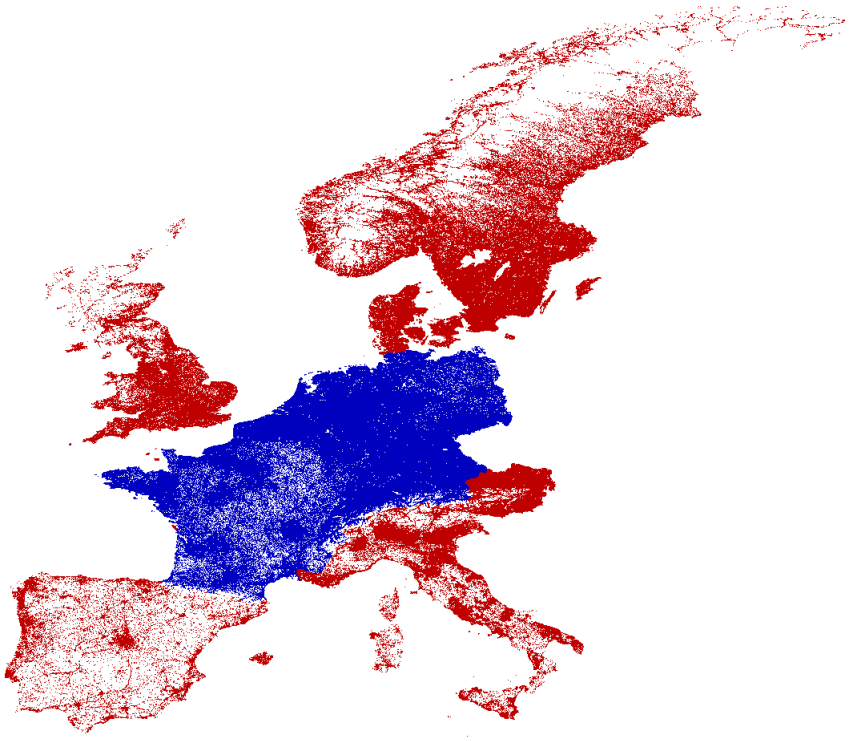


Figure C.9: Europe (not proven optimal)

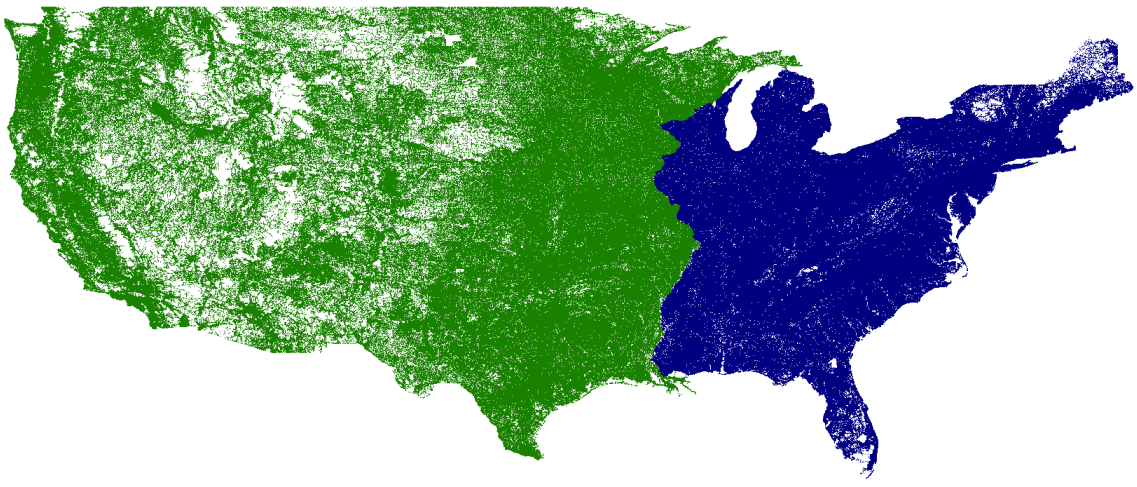


Figure C.10: USA

BIBLIOGRAPHY

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, pages 230–241, 2011.

[2] Ittai Abraham, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 782–793. Society for Industrial and Applied Mathematics, 2010.

[3] David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. Concorde tsp solver, 2006.

[4] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[5] Michael Armbruster. *Branch-and-Cut for a Semidefinite Relaxation of Large-Scale Minimum Bisection Problems*. PhD thesis, Technische Universität Chemnitz, 2007.

[6] Michael Armbruster. Graph bisection and equipartition, 2007. `http://www.tu-chemnitz.de/mathematik/discrete/armbruster/diss/`.

[7] Michael Armbruster, Marzena Fügenschuh, Christoph Helmberg, and Alexander Martin. A comparative study of linear and semidefinite branch-and-cut methods for solving the minimum graph bisection problem. In *Proc. Conf. Integer Programming and Combinatorial Optimization (IPCO)*, volume 5035 of *LNCS*, pages 112–124, 2008.

[8] Michael Armbruster, Marzena Fügenschuh, Christoph Helmberg, and Alexander Martin. LP and SDP branch-and-cut algorithms for the minimum graph bisection problem: A computational comparison. *Mathematical Programming Computation*, 4(3):275–306, 2012.

[9] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

[10] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering—10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013. `http://www.cc.gatech.edu/dimacs10/`.

[11] Stephen T. Barnard and Hort Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency and Computation: Practice and Experience*, 6(2):101–117, 1994.

[12] John J Bartholdi III, Loren K Platzman, R Lee Collins, and William H Warden III. A minimal technology routing system for meals on wheels. *Interfaces*, 13(3):1–8, 1983.

[13] Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009.

[14] Gary Benson. Tandem cyclic alignment. In Amihood Amir, editor, *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 118–130. Springer Berlin Heidelberg, 2001.

[15] Sandeep N. Bhatt and Frank Thomson Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984.

[16] L. Brunetta, M. Conforti, and G. Rinaldi. A branch-and-cut algorithm for the equicut problem. *Mathematical Programming*, 78:243–263, 1997.

[17] Mihai Budiu, Daniel Delling, and Renato F. Werneck. DryadOpt: Branch-and-bound on distributed data-parallel execution engines. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1278–1289, 2011.

[18] T. N. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[19] Pierre Chardaire, Musbah Barake, and Geoff P. McKeown. A PROBE-based heuristic for graph partitioning. *IEEE Transactions on Computers*, 56(12):1707–1720, 2007.

[20] Cédric Chevalier and Francois Pellegrini. PT-SCOTCH: A tool for efficient parallel graph ordering. *Parallel Computing*, 34:318–331, 2008.

[21] Shihabur Rahman Chowdhury, Masud Hasan, Sumaiya Iqbal, and M. Sohel Rahman. An $o(n^2)$ algorithm for computing longest common cyclic subsequence. *CoRR*, abs/0911.5031, 2009.

[22] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.

[23] William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.

[24] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[25] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[26] M. Cygan, D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. Minimum bisection is fixed parameter tractable. In *Proc. ACM Symposium on Theory of Computing (STOC)*, 2014. To appear.

[27] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[28] Gautam Das, Rudolf Fleischer, Leszek Gasieniec, Dimitris Gunopulos, and Juha Karkkainen. Episode matching, 1997.

[29] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[30] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proc. International Symposium on Experimental Algorithms (SEA)*, volume 6630 of *LNCS*, pages 376–387. Springer, 2011.

[31] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph partitioning with natural cuts. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1135–1146. IEEE, 2011.

[32] Daniel Delling and Renato F. Werneck. Faster customization of road networks. In *Proc. International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *LNCS*, pages 30–42. Springer, 2013.

[33] Erik D. Demaine, MohammadTaghi Hajiaghayi, and Ken-ichi Kawarabayashi. Contraction decomposition in h-minor-free graphs and algorithmic applications. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 441–450, 2011.

[34] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[35] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[36] A. H. G. R. Kan E. L. Lawler, J. K. Lenstra and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, 2015.

[37] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 448–455. ACM, 2003.

[38] Andreas Emil Feldmann and Peter Widmayer. An $O(n^4)$ time algorithm to compute the bisection width of solid grid graphs. In *Proc. European Symposium on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 143–154, 2011.

[39] Ariel Felner. Finding optimal solutions to the graph partitioning problem with heuristic search. *Annals of Math. and Artificial Intelligence*, 45(3–4):293–322, 2005.

[40] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24:327–336, 1994.

[41] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A.

Wolsey. The node capacitated graph partitioning problem: A computational study. *Mathematical Programming*, 81:229–256, 1998.

[42] Daniel Fleischman. Github repository with code snippets for programming competitions. `https://github.com/danielf/lib-puc-icpc/blob/master/BIT/bit.cpp`, 2016.

[43] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[44] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, 1979.

[46] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified $\mathcal{NP}$-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[47] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[48] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[49] Andrew V Goldberg. Point-to-point shortest path algorithms with preprocessing. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 88–102. Springer, 2007.

[50] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. Maximum flows by incremental breadth-first search. In *Proc. European Symposium on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 457–468, 2011.

[51] Igor Gorodezky, Robert D Kleinberg, David B Shmoys, and Gwen Spencer. Improved lower bounds for the universal and a priori tsp. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 178–191. Springer, 2010.

[52] William W. Hager, Dzung T. Phan, and Hongchao Zhang. An exact algorithm for graph partitioning. *Mathematical Programming*, 137:531–556, 2013.

[53] Matthias Hein and Thomas Bühler. An inverse power method for nonlinear eigenproblems with applications in 1-spectral clustering and sparse PCA. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 847–855, 2010.

[54] Christoph Helmberg. A cutting plane algorithm for large scale semidefinite relaxations. In Martin Grötschel, editor, *The Sharpest Cut*. SIAM, Philadephia, PA, 2004.

[55] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[56] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing*, page 28. ACM Press, 1995.

[57] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Demetrescu et al. [34], pages 41–72.

[58] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[59] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.

[60] Klaus Jansen, Marek Karpinski, Andrzej Lingas, and Eike Seidel. Polynomial time approximation schemes for MAX-BISECTION on planar and geometric graphs. *SIAM Journal on Computing*, 35:110–119, 2005.

[61] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; Part I, Graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[62] E. Johnson, A. Mehrotra, and G. Nemhauser. Min-cut clustering. *Mathematical Programming*, 62:133–152, 1993.

[63] Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.

[64] M. Jünger, A. Martin, G. Reinelt, and R. Weismantel. Quadratic 0/1 optimization and a decomposition approach for the placement of electronic circuits. *Mathematical Programming*, 63:257–279, 1994.

[65] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal the ACM*, 43(4):601–640, 1996.

[66] S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *INFORMS Journal on Computing*, 12:177–191, 2000.

[67] George Karypis and Gautam Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1999.

[68] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[69] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[70] T. Koch, A. Martin, and S. Voß. SteinLib: An updated library on Steiner tree problems in graphs. Technical Report 00-37, Konrad-Zuse-Zentrum Berlin, 2000.

[71] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[72] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Tr. on Graphics*, 22:277–286, 2003.

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

[74] Kevin J. Lang and Satish Rao. A flow-based method for improving the

expansion or conductance of graph cuts. In *Proc. Conf. Integer Programming and Combinatorial Optimization (IPCO)*, pages 325–337, 2004.

[75] Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In Demetrescu et al. [34], pages 19–40.

[76] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[77] Richard J. Lipton and Robert Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.

[78] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978.

[79] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *PODC*, page 6. ACM, 2009.

[80] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980.

[81] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[82] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009.

[83] R Garey Michael and S Johnson David. Computers and intractability: a guide to the theory of np-completeness. *WH Free. Co., San Fr*, 1979.

[84] A. Nguyen. Solving cyclic longest common subsequence in quadratic time. *ArXiv e-prints*, August 2012.

[85] William R. Pearson.

[86] Francois Pellegrini and Jean Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architec-

ture graphs. In *High-Performance Computing and Networking*, volume 1067 of *LNCS*, pages 493–498. Springer, 1996.

[87] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[88] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 255–263. ACM Press, 2008.

[89] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[90] Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121:307–335, 2010.

[91] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. on Graphics*, 27:144:1–144:9, 2008.

[92] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *Proc. Algorithm Engineering and Experiments (ALENEX)*, pages 16–29. SIAM, 2012.

[93] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proc. International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[94] Frans Schalekamp and David B Shmoys. Algorithms for the universal and a priori tsp. *Operations Research Letters*, 36(1):1–3, 2008.

[95] Meinolf Sellmann, Norbert Sensen, and Larissa Timajev. Multicommodity flow approximation used for exact graph partitioning. In *Proc. European Symposium on Algorithms (ESA)*, volume 2832 of *LNCS*, pages 752–764, 2003.

[96] Norbert Sensen. Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows. In *Proc. European Symposium on Algorithms (ESA)*, volume 2161 of *LNCS*, pages 391–403, 2001.

[97] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[98] David Shmoys and Kunal Talwar. A constant approximation algorithm for the a priori traveling salesman problem. In *Integer Programming and Combinatorial Optimization*, pages 331–343. Springer, 2008.

[99] A. J. Soper, Chris Walshaw, and Mark Cross. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[100] A. J. Soper, Chris Walshaw, and Mark Cross. The graph partitioning archive, 2004.

[101] R Staden. A strategy of dna sequencing employing computer programs. *Nucleic acids research*, 6(7):2601–2610, 1979.

[102] Paul Stothard, Gary H. Van Domselaar, Savita Shrivastava, Anchi Guo, Brian O'Neill, Joseph A. Cruz, Michael Ellison, and David S. Wishart. Bacmap: an interactive picture atlas of annotated bacterial genomes. *Nucleic Acids Research*, 33(Database-Issue):317–320, 2005.

[103] Chris Walshaw and Mark Cross. JOSTLE: Parallel multilevel graph-partitioning software – an overview. In Frédéric Magoulès, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.

[104] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[105] Wikipedia. Circular bacterial chromosome — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Circular_bacterial_chromosome`, 2016. [Online; accessed 30-April-2016].

[106] Zhenyu Wu and Richard Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993.