New Algorithms for Three Combinatorial Optimization Problems on Graphs

by

Mark Velednitsky

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Industrial Engineering and Operations Research

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ilan Adler, Chair
Professor Alper Atamtürk
Professor Paul Grigas
Professor Satish Rao

Spring 2020

New Algorithms for Three Combinatorial Optimization Problems on Graphs

Abstract

New Algorithms for Three Combinatorial Optimization Problems on Graphs

by

Mark Velednitsky

Doctor of Philosophy in Industrial Engineering and Operations Research

University of California, Berkeley

Professor Ilan Adler, Chair

In this dissertation, we study three NP-hard combinatorial optimization problems phrased on graphs. For each problem, we introduce one or more new algorithms tailored for the problem.

The first problem is the minimum k-`terminal cut` problem. Given a graph and a distinguished subset of vertices ("terminals"), we would like to remove the minimum weight set of edges that disconnect the terminals. We present a fixed-parameter-tractable branch-and-bound algorithm for the problem. We also show that $(k-1)$-stable instances of k-`terminal cut` can be solved optimally by calculating $k-1$ minimum isolating cuts: minimum cuts which separate one terminal from the rest. This analysis is tight: there exist $(k-1-\epsilon)$-stable instances for which the isolating cuts do not return the optimal solution.

The second problem concerns valid distance drawings of signed graphs. A valid distance drawing of a signed graph in $\mathbb{R}^k$ is an embedding of the graph in $\mathbb{R}^k$ such that, for every vertex, all its positive neighbors are closer than its negative neighbors. We address the question of finding $L(n)$, the smallest dimension such that every signed graph with n nodes has a valid distance drawing in $\mathbb{R}^{L(n)}$. In general, we show that $\lfloor \log_5(n-3) \rfloor + 1 \leq L(n) \leq n-2$. We introduce a new algorithm for computing $L(n)$ and then compute $L(n)$ exactly up to $n = 7$. We offer a conjecture for the value of $L(n)$ for all n.

The third problem is the maximum online perfect bipartite matching problem with i.i.d. arrivals. We introduce seven algorithms in two classes: three "flow-guided" algorithms and four "evaluation-guided" algorithms. Two of the evaluation-guided algorithms can be interpreted as derandomizations of flow-guided algorithms. We prove that at least three of the algorithms are 0.5-competitive, which is the best possible competitive ratio. The seven algorithms can be partially ordered: for some pairs, one may be expected to perform at least as well as the other on all instances. Through theoretical and empirical results, we determine all but four of the pairwise relations in this partial ordering.

To my family.

For their unwavering support.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation would not have been possible without the support of many.

Thank you to the National Physical Science Consortium (NPSC) for awarding me your fellowship. Your generous financial support allowed me to focus on doing the best research possible.

Thank you to my colleagues for providing feedback on my presentations, for encouraging my best ideas, and for making the graduate student office a fun place to work. Thank you Yonatan, Erik, Salar, Han, Dean, Titouan, Rebecca, Alfonso, Matt, and Quico.

Thank you to my frequent collaborator Quico. I am proud of all the papers we worked on. Together, we were more than the sum of parts!

Thank you to the IEOR department staff for your incredible compassion and commitment to the well-being of students, which goes beyond the call of duty. Thank you Rebecca, Heather, and Keith.

Thank you to my friend outside of the IEOR department and outside of Berkeley. I am very fortunate to have spent the last five years surrounded by a community of smart and adventurous people. I have fond memories of late-night discussions at Hillel, board game nights, trips to national parks, and so much more.

Thank you to my committee members for your guidance, both in research and professionally. Thank you Satish, Paul, and Alper.

An enormous thank you to my advisor, Ilan. I am incredibly lucky to have had the opportunity to work with such a distinguished scholar. I learned more from you in a few short months than I ever thought possible. Thank you for engaging with my work, reminding me of the joys of discovery that make research so rewarding, and pushing me to be my best.

Finally, thank you to my family. To my parents, Robin and Boris, thank you for giving me every opportunity. To my sister, Leanne, thank you for always sharing your witty sense of humor. To my wife, Shanté, thank you for being someone that I can always count on. I love you! To my half-sisters Bailee and Veronica... I hope you enjoy math enough to read this some day!

Chapter 1

Introduction

1.1 Motivation

When discussing problem and solutions in operations research and computer science, it is important to distinguish between properties of *problems* and properties of *algorithms*.

For example, consider the problem of comparison-based sorting. There are a myriad of sorting algorithms. Bubble Sort has complexity $O(n^2)$ and Merge Sort has complexity $O(n \log n)$. These complexities are properties of the algorithms, not the problem. When we say the "complexity of the comparison-based sorting problem," we mean the best possible theoretical complexity of any algorithm. From the analysis of Merge Sort, we know that the complexity of comparison-based sorting is at most $O(n \log n)$. On the other hand, from a separate analysis, we know that the complexity is at least $\Omega(n \log n)$. Thus, we conclude that the complexity of comparison-based sorting is exactly $\theta(n \log n)$.

Similar properties appear throughout operations research and computer science, where the same property can be the property *of an algorithm* or a property *of a problem*. For example, approximation ratios and competitive ratios.

There are many techniques for proving inapproximability or other hardness results. These often include specifying reductions from other problems with known hardness results. On the other hand, showing a problem can be approximated or can be solved in a certain complexity is often constructive.

Sometimes, new algorithms are devised with wide applicability to many problems. Consider, for example, linear programming. The great advantage is that these new algorithms can be implemented once and used many times. However, they may not advance our theoretical understanding of any one problem, in particular.

In this dissertation, we focus on the latter case. Rather than developing a new general-purpose algorithm for several problems, we tailor one or more algorithms to the problem at hand. In doing so, we push the theoretical understanding of a problem's asymptotic behavior.

Over three chapters, we consider three combinatorial optimization problems on graphs.

The problems are unrelated. The unifying theme is our approach. For each problem, we propose one or more new algorithms. Our analysis of the algorithms improves the current state-of-the-art understanding of the limiting behavior of the problem.

1.2 Outline

In chapter 2, the problem we consider is the minimum k-`terminal cut` problem. Given a graph and a distinguished set of vertices, called "terminals," we would like to determine the minimum weight set of edges that must be removed to disconnect the terminals. The k-`terminal cut` problem is known to be APX-hard for $k \geq 3$. We present a new, fixed-parameter-tractable branch-and-bound algorithm for k-`terminal cut`. Our branch-and-bound algorithm uses a novel relaxation of the Călinescu-Karloff-Rabani [11] integer programming formulation of k-`terminal cut`. We call our relaxation the *isolating cut* relaxation because we solve it using *minimum isolating cuts*, minimum (s, t)-cuts which separate one terminal from the rest. We call our algorithm Isolating Cut Branch-and-Bound. In an empirical experiment, we compare Isolating Cut Branch-and-Bound to linear-programming-based branch-and-bound with Gurobi, a commercial mixed-integer programming solver. On the twenty-four real-world benchmark instances, Isolating Cut Branch-and-Bound outperforms linear-programming-based branch-and-bound with Gurobi.

We continue our treatment of the k-`terminal cut` problem by considering γ-stable instances. An instance of k-`terminal cut` is γ-stable ($\gamma > 1$) if the optimal cut is unique and the weight of edges in the optimal cut can be multiplied by up to γ without changing the unique optimal cut. We prove that, in any $(k-1)$-stable instance of k-`terminal cut`, the source sets of the minimum isolating cuts return the optimal solution to that k-`terminal cut` instance. Thus, an optimal solution to $(k-1)$-stable instances of k-`terminal cut` can be computed in the time it takes to compute $k-1$ minimum (s, t)-cuts. Our result is tight. We prove this by constructing $(k-1-\epsilon)$-stable instances of the k-`terminal cut` problem, for $0 < \epsilon < k-2$, in which the source set of the minimum isolating cuts for each terminal are just the terminals themselves and do not return the optimal solution.

In chapter 3, we consider the problem of finding valid distance drawings of signed graphs in $\mathbb{R}^k$. We call an embedding of a signed graph G into $\mathbb{R}^k$ a *valid distance drawing* if, for every vertex v, all the of positive neighbors of v are closer than all of its negative neighbors (in terms of euclidean distance). We address the question of finding $L(n)$, the smallest dimension such that every signed graph with n nodes has a valid embedding in $\mathbb{R}^{L(n)}$. In general, we show that $\lfloor \log_5(n-3) \rfloor + 1 \leq L(n) \leq n-2$. We also phrase the embedding problem as an optimization problem, introducing the algorithm Valid Drawing, which attempts to find a valid drawing for a given input graph. Using our algorithm, we compute $L(n)$ exactly for small values of n (up to $n = 7$). From the results of these experiments, we conjecture that the complete signed graph whose positive subgraph is the complete bipartite graph $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$ is the n-node signed graph requiring the most dimensions to embed. We calculate an embedding for this graph up to $n = 24$, which suggests a pattern for the

asymptotic behavior of $L(n)$. We conjecture that $L(n) \sim \frac{3}{4}n$.

In chapter 4, we consider a variant of the online bipartite matching problem. The variant we consider is the *maximum* online *perfect* bipartite matching problem *with i.i.d. arrivals*. We are given a set of n workers, a distribution over k job types, and non-negative utility weights for each pair of worker and job type. This set-up is naturally modeled as a bipartite graph on workers and job types. At each time step, a job is drawn i.i.d. from the distribution over job types. Upon arrival, the job must be irrevocably assigned to a worker and cannot be dropped. After n jobs have arrived, the assignment of workers to jobs is a perfect matching. The goal is to maximize the expected sum of utilities of this perfect matching.

In the chapter, we introduce two classes of algorithms for the problem. The first is the class of "flow-guided" algorithms, where the decision of which worker to assign at each step is guided by an offline transportation problem between the workers and job types. The second is the class of "evaluation-guided" algorithms. In evaluation-guided algorithms, the problem is modeled as a Markov Decision Process (MDP). Instead of computing the exact expected value of states in the MDP, which requires exponential time, an *evaluation function* is used as a proxy. When the evaluation function is derived from the expected performance of an algorithm, the resulting evaluation-guided algorithm can be thought of as a derandomization of the algorithm used for evaluation.

A total of seven specific algorithms are introduced, including three flow-guided algorithms and four evaluation-guided algorithms. We prove that one of the flow-guided algorithms, which we call $\mathcal{OPT}$-$\mathcal{FLOW}$, is optimal among all flow-guided algorithms. We also prove that at least three of the algorithms are 0.5-competitive, which is the best possible competitive ratio. On the other hand, at least two of the baseline algorithms, a greedy algorithm ($\mathcal{GREEDY}$) and a purely random algorithm ($\mathcal{RAND}$), have arbitrarily bad competitive ratios. The seven algorithms can be partially ordered: for some pairs of the algorithms, we can guarantee that one will always perform at least as well as the other, in expectation, on all instances of the problem. Some of these relations are surprising. For example, it turns out that a greedy algorithm always performs at least as well, in expectation, as the purely random algorithm on all instances. Through a combination of theoretical and empirical results, we determine all but four of the pairwise relations in this partial ordering.

We conclude the dissertation in chapter 5.

Chapter 2

The k-terminal cut Problem

2.1 Introduction

In the k-terminal cut problem, also known as the multiterminal cut problem, we are given an graph with edge weights and k distinct vertices called "terminals." The goal is to remove a minimum weight collection of edges from the graph such that there is no path between any pair of terminals. The k-terminal cut problem is APX-hard for $k \geq 3$ [19].

The k-terminal cut problem has a number of applications. Specific application areas include distributing computational jobs in a parallel computing system [26], partitioning elements of a circuit into sub-circuits that will be put on different chips [19], scheduling tasks [35], understanding transportation bottlenecks [35], planning the "divide" step in divide-and-conquer algorithms [29], and pre-processing an image for computer vision [7]. More generally, graph cut problems, including k-terminal cut, have applications to graph clustering [24]. Minimizing the weight of edges between clusters is equivalent to maximizing the weight within clusters. In a setting where the weights measure similarity between vertices, the result is a graph clustering algorithm. Thus, k-terminal cut gives an explicit combinatorial objective function for supervised graph clustering.

In an instance of k-terminal cut, minimum isolating cuts are minimum cuts which separate one terminal from the rest of the terminals. They can give useful information about the optimal solution: the source set of a terminal's minimum isolating cut is a subset of that terminal's source set in an optimal solution [19]. Furthermore, the union of all the edges of all the minimum isolating cuts, except for the cut with largest weight, is a $(2-2/k)$-approximation for the k-terminal cut problem. This approximation was first proven by Dahlhaus et al. [19].

Many approaches to the k-terminal cut problem have been studied. These approaches include devising approximation algorithms for quickly finding approximate solutions, devising fixed-parameter tractable algorithms for solving the problem to optimality in exponential time, or finding polynomial-time algorithms for special instances of the problem.

The $(2 - 2/k)$-approximation algorithm of Dahlhaus et al., using minimum isolating

cuts, is considerably different from most other currently-known approximation algorithms for k-terminal cut. Most of the other currently-known approximation algorithm for k-terminal cut work by probabilistically rounding the linear programming relaxation of the Călinescu-Karloff-Rabani (CKR) Integer Programming formulation, giving an approximation guarantee in expectation [11, 33, 10, 55]. The earliest such rounding scheme, by Călinescu et al. [11], delivered an approximation factor of 1.5. The best-to-date approximation factor, by Sharma et al. [55], is 1.2965.

There also exist fixed-parameter tractable algorithms for solving k-terminal cut optimally. It was first proven in 2004 that k-terminal cut is fixed-parameter tractable with respect to the value of the optimal solution [47]. That proof was not constructive. A constructive proof in the form of an algorithm was given by Chen, Liu, and Lu [14], with running time $O(w(E_{\text{OPT}})4^{w(E_{\text{OPT}})}n^3)$, where $w(E_{\text{OPT}})$ is the weight of the optimal cut and n is the number of vertices in the graph. The algorithm of Chen, Liu, and Lu is of theoretical value. To our knowledge, none of the aforementioned fixed-parameter tractable algorithms have ever been implemented.

Bilu and Linial [6] introduced the concept of stability for graph cut problems. An instance of k-terminal cut is said to be γ-stable ($\gamma > 1$) if the optimal cut remains uniquely optimal when every edge in the cut is multiplied by a factor up to γ. The concepts of sensitivity analysis and robustness in linear programming are closely related [53, 5]. Makarychev et al. [43] showed that, for 4-stable instances of k-terminal cut, the solution to the linear programming relaxation of the CKR formulation will necessarily be integer. The result was later improved to $(2 - 2/k)$-stable instances using the same linear programming technique [2].

Our contributions in this chapter are as follows:

1. We devise a fixed-parameter tractable branch-and-bound algorithm for the k-terminal cut problem which relies on a new relaxation of the CKR integer programming formulation. The new relaxation can be solved efficiently with minimum isolating cuts and, for this reason, we call it the "isolating cut" relaxation. We call our branch-and-bound algorithm Isolating Cut Branch-and-Bound.

2. We establish a connection between minimum isolating cuts and stability. We prove that, in $(k-1)$-stable instances of k-terminal cut, the minimum isolating cuts return the optimal solution. Our result is tight: we also exhibit $(k - 1 - \epsilon)$-stable instances $(0 < \epsilon < k-2)$ in which the minimum isolating cuts do not return the optimal solution. As a corollary, we prove that Isolating Cut Branch-and-Bound terminates on $(k - 1)$-stable instances after computing just k minimum isolating cuts.

3. We conduct an empirical study of optimization algorithms for k-terminal cut, in which the performance of Isolating Cut Branch-and-Bound is compared to branch-and-bound using the linear programming relaxation of the CKR Integer Programming formulation of k-terminal cut. The linear programming branch-and-bound algorithm is implemented with Gurobi, a commercial mixed-integer program-

ming solver. The performance is evaluated for real-world instances and simulated data. On twenty-four real-world benchmark data sets, with up to tens of thousands of vertices and hundreds of thousands of edges, Isolating Cut Branch-and-Bound runs more than 10× faster than Gurobi. On simulated data, the relative speedup of Isolating Cut Branch-and-Bound over Gurobi grows with the size of the instance.

2.2 Preliminaries

The notation $\{G = (V, E, w), T\}$ refers to an instance of the k-`terminal cut` problem, where $G = (V, E, w)$ is an undirected graph with vertices V and edges E. $T = \{t_1, \ldots, t_k\} \subseteq V$ is a set of k terminals ($k \geq 2$). The weight function, w, is a function from E to $\mathbb{R}^+$.

For an instance $\{G, T\}$ of the k-`terminal cut` problem, we can refer to feasible solutions in two equivalent ways. The first is in terms of the edges that are cut and the second is in terms of the partition of V.

A set of edges $E_{\text{FEAS}} \subseteq E$ is a feasible solution to k-`terminal cut` if removing all the edges in E_{FEAS} ensures that there is no path between any pair of terminals. The notation $w(E_{\text{FEAS}})$ denotes the total weight of edges in E_{FEAS}:

$$w(E_{\text{FEAS}}) = \sum_{e \in E_{\text{FEAS}}} w(e).$$

An optimal solution, E_{OPT} is a minimum-weight feasible solution.

An equivalent way to define a feasible solution is as partition of V into a collection of k subsets $(V_1, \ldots, V_k)$, where $t_i \in V_i \subset V$ for $i \in \{1, \ldots, k\}$. The implied feasible solution is the set of edges with endpoints in different sets. That is, $E_{\text{FEAS}} = \{\{v_i, v_j\} | v_i \in V_i, v_j \in V_j, i \neq j\}$. We notate the total weight of edges that go between distinct subsets:

$$w(V_1, \ldots, V_k) = \sum_i \sum_{j>i} w(V_i, V_j).$$

Referring to the optimal cut in terms of vertices, we use the notation $V_1^*, \ldots, V_k^*$.

Combining the notation introduced in this section,

$$w(E_{\text{OPT}}) = w(V_1^*, \ldots, V_k^*).$$

To introduce Isolating Cut Branch-and-Bound, we will sometimes want to work with a collection of k disjoint sets $(V_1, \ldots, V_k)$ where $t_i \in V_i$ but where some vertices in V are not assigned to any of the k sets. We will call such collections *sub-feasible* collections.

Definition 2.1 (Sub-Feasible Collection)**.** *A collection of k disjoint subsets of V, $(V_1, \ldots, V_k)$, is called* sub-feasible *if $t_i \in V_i \subset V$ for $i \in \{1, \ldots, k\}$.*

Equivalently, we could say that a sub-feasible collection is a partition of a *subset* of V such that $t_i \in V_i$. A sub-feasible collection is a feasible solution if and only if $\bigcup_{i=1}^k V_i = V$. Given a sub-feasible collection, we can generate a feasible solution to k-`terminal cut` by assigning the unassigned vertices to sets in the collection.

Isolating Cuts

To introduce minimum isolating cuts, it is helpful to recall the minimum (s,t)-cut problem. Given a graph $G = (V, E, w)$ and terminals s and t, a minimum (s,t)-cut is a partition of the set of vertices into two sets, a *source set* containing s and a *sink set* containing t, such that the total weight of edges between the source set and the sink set is minimized.

Definition 2.2 (Minimum V_i-Isolating Cut)**.** *Given a sub-feasible collection $(V_1, \ldots, V_k)$, a minimum isolating cut for V_i is a minimum cut set which separates V_i from all the vertices in $\cup_{j \neq i} V_j$. The notation $\mathcal{I}(V_i)$ denotes the source set of this minimum isolating cut.*

The problem of calculating a minimum isolating cut for V_i can be reduced to the problem of computing a minimum (s,t)-cut. This is accomplished by contracting all of the vertices in V_i into a single source vertex s, contracting all the vertices $\cup_{j \neq i} V_j$ into a single sink vertex t, then calculating a minimum (s,t)-cut. It may be the case that there are several minimum cuts. In such cases, it is always possible to efficiently compute the minimum cut with maximum source set [25].

The definition we have stated here for minimum isolating cuts is slightly more general than the definition that is typically used in the literature. We have defined isolating cuts for any sub-feasible collection. Often, the only sub-feasible collection that is considered is the collection of terminals: $(\{t_1\}, \ldots, \{t_k\})$. When dealing with this sub-feasible collection, we will refer to the minimum $\{t_i\}$-isolating cut as the t_i-isolating cut and notate it $\mathcal{I}(t_i)$ instead of $\mathcal{I}(\{t_i\})$, omitting the set notation. Dahlhaus et al. [19] prove the following lemma:

Lemma 2.1 (Isolation Lemma)**.** *Let $\{G = (V, E, w), T\}$ be an instance of k-terminal cut. For all i, there exists an optimal solution $(V_1^*, \ldots, V_k^*)$ in which $\mathcal{I}(t_i) \subseteq V_i^*$.*

As an example, consider figure 2.1. In the unique optimal k-terminal cut, $(V_1^*, V_2^*, V_3^*, V_4^*)$, $|V_i^*| = 3$ for all $i \in \{1, 2, 3, 4\}$. The weight of the cut is $w(V_1^*, V_2^*, V_3^*, V_4^*) = 8$ (cutting the four edges that form the central square). The t_1-isolating cut consists of two vertices:

$$\mathcal{I}(t_1) = \{t_1, u_1\}.$$

The corresponding minimum cut has weight 3:

$$w(\mathcal{I}(t_1), V \setminus \mathcal{I}(t_1)) = 3.$$

The t_4-isolating cut consists of three vertices: $|\mathcal{I}(t_4)| = \{t_4, u_4, v_4\}$. The corresponding minimum cut has weight 4. For all four terminals, $\mathcal{I}(t_i) \subseteq V_i^*$. The isolation lemma proves that this is always the case.

The CKR Formulation

In the literature, the following Integer Programming formulation of k-terminal cut is often referred to as the Călinescu-Karloff-Rabani (CKR) formulation [11]. Assuming the Unique

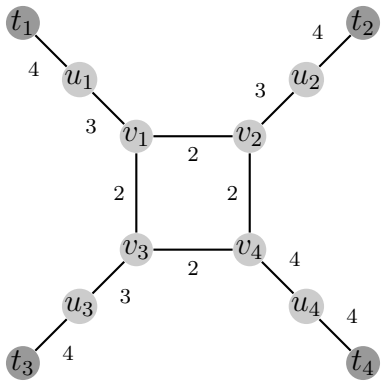


Figure 2.1: Example k-terminal cut instance, where $k = 4$

Games Conjecture, it has been proven that no other formulation can have a smaller integrality gap [45].

The variable x_i^t is a binary variable: it is 1 if vertex i is assigned to terminal t and 0 otherwise. If x_i^t and x_j^t differ, then z_{ij}^t is forced to be 1. In total, if there are n nodes and m edges in G, the CKR formulation has $k(n+m)$ variables and $n + 2km$ constraints.

$$\begin{aligned}
\min \quad & \sum_{\{i,j\}\in E, t\in T} \frac{1}{2} w(\{i,j\}) z_{ij}^t && && \text{(CKR)}\\
s.t. \quad & z_{ij}^t \geq x_i^t - x_j^t && \forall \{i,j\} \in E, t \in T\\
& z_{ij}^t \geq x_j^t - x_i^t && \forall \{i,j\} \in E, t \in T\\
& \sum_{t\in T} x_i^t = 1 && \forall i \in V\\
& x_i^t \in \{0,1\} && \forall i \in V, t \in T\\
& z_{ij}^t \in \{0,1\} && \forall \{i,j\} \in E, t \in T\\
& x_t^t = 1 && \forall t \in T
\end{aligned}$$

In this chapter, we consider two possible relaxations of the CKR formulation. The first is the linear programming relaxation, in which we relax the constraints that x_i^t and z_{ij}^t are integer:

$$\begin{aligned}
x_i^t \in \{0,1\} \rightarrow 0 \leq x_i^t \leq 1\\
z_{ij}^t \in \{0,1\} \rightarrow 0 \leq z_{ij}^t \leq 1
\end{aligned}$$

This relaxation can be solved efficiently with linear programming. The full relaxation is

$$\begin{aligned}
\min \quad & \sum_{\{i,j\}\in E, t\in T} \frac{1}{2} w(\{i,j\}) z_{ij}^t && && \text{(CKR-LP)}\\
s.t. \quad & z_{ij}^t \geq x_i^t - x_j^t && \forall \{i,j\} \in E, t \in T\\
& z_{ij}^t \geq x_j^t - x_i^t && \forall \{i,j\} \in E, t \in T\\
& \sum_{t\in T} x_i^t = 1 && \forall i \in V\\
& 0 \leq x_i^t \leq 1 && \forall i \in V, t \in T\\
& 0 \leq z_{ij}^t \leq 1 && \forall \{i,j\} \in E, t \in T\\
& x_t^t = 1 && \forall t \in T.
\end{aligned}$$

The second relaxation we can consider is one in which the variables remain integer, but we relax the requirement that each vertex be assigned to a terminal, allowing some vertices to be assigned to no terminals. Mathematically,

$$\sum_{t\in T} x_i^t = 1 \rightarrow \sum_{t\in T} x_i^t \leq 1$$

We call this relaxation the "isolating cut" relaxation because, as we will see, it can be solved efficiently using isolating cuts. The full relaxation is

$$\begin{aligned}
\min \quad & \sum_{\{i,j\}\in E, t\in T} \frac{1}{2} w(\{i,j\}) z_{ij}^t && && \text{(CKR-IC)}\\
s.t. \quad & z_{ij}^t \geq x_i^t - x_j^t && \forall \{i,j\} \in E, t \in T\\
& z_{ij}^t \geq x_j^t - x_i^t && \forall \{i,j\} \in E, t \in T\\
& \sum_{t\in T} x_i^t \leq 1 && \forall i \in V\\
& x_i^t \in \{0,1\} && \forall i \in V, t \in T\\
& z_{ij}^t \in \{0,1\} && \forall \{i,j\} \in E, t \in T\\
& x_t^t = 1 && \forall t \in T.
\end{aligned}$$

2.3 A Branch-and-Bound Algorithm

In this section, we introduce our branch-and-bound algorithm for k-terminal cut: Isolating Cut Branch-and-Bound. Our algorithm follows the same blueprint as linear

programming branch-and-bound, but uses the relaxation CKR-IC instead. At each node in the branch-and-bound tree, we solve the isolating cut relaxation CKR-IC. This solution can be calculated with minimum isolating cuts. The objective value of the solution provides a *bound*. When we *branch*, we take a vertex which is not assigned to any terminal and consider assigning it to each terminal in turn, creating k children nodes.

Before we get into the details, some general notation: let $\mathbb{T}$ denote the incumbent branch-and-bound tree and let $d \in \mathbb{T}$ be a node in the tree. We will use *nodes* when referring to the branch-and-bound tree $\mathbb{T}$ and *vertices* when referring to V in the original graph G.

Solving the Isolating Cut Relaxation

Consider the relaxation CKR-IC. We will show that this relaxation can be solved efficiently by computing k minimum isolating cuts. Consider a feasible setting of the x_i^t variables. Let V_t be the set of vertices which are assigned to terminal t in this solution:

$$V_t = \{i | x_i^t = 1\}.$$

The constraint $\sum_{t \in T} x_i^t \leq 1$ ensures that each vertex in the graph is in at most one of the V_t. Thus, $(V_1, \ldots, V_k)$ is a sub-feasible collection.

Because we are minimizing, in an optimal solution the variable z_{ij}^t takes on the value 1 if and only if $i \in V_t$, $j \notin V_t$ (or vice versa). The objective function can thus be rewritten

$$\sum_{t \in T} \sum_{\{i,j\} \in E} \frac{1}{2} w(\{i,j\}) z_{ij}^t = \sum_{t \in T} \frac{1}{2} w(V_t, V \setminus V_t).$$

The cut $(V_t, V \setminus V_t)$ isolates the terminal t, so this objective is the sum of k isolating cuts, one for each terminal. Thanks to lemma 2.1, we know that it is possible to compute k minimum isolating cuts, one for each terminal, which do not overlap. Thus, to find the optimal solution to relaxation CKR-IC, we take k minimum isolating cuts and set $x_i^t = 1$ if and only if i is in the minimum isolating cut for terminal t.

Beyond the root node of the Isolating Cut Branch-and-Bound tree, we will stipulate that certain vertices must be assigned to certain terminals. This is equivalent to adding the constraint $x_i^t = 1$ for certain pairs $i \in V$ and $t \in T$. We say that $i \in V$ is *fixed* to $t \in T$ if we have added such a constraint. Let $F_{d,t}$ be the set of vertices fixed to terminal t in node d.

With these added constraints, the relaxation can still be solved by a series of minimum isolating cuts. The key realization is that the cuts $(V_t, V \setminus V_t)$ (as defined earlier) are $F_{d,t}$-isolating for the sub-feasible collection $(F_{d,1}, \ldots, F_{d,k})$. As before, an optimal solution can be found by taking k minimum $F_{d,t}$-isolating cuts, one for each $t \in \{1, \ldots, k\}$.

Algorithm 2.1 ISOLATING CUT BRANCH-AND-BOUND

Input: $G = (V, E)$: an instance of k-`terminal cut`.
Output: $(V_1, \ldots, V_k)$: an optimal solution to k-`terminal cut`.

```
1: set d to root node
2: while optimal solution not found do
3:     solve relaxation CKR-IC at d
4:     if d has unassigned vertices then
5:         variable selection: choose vertex ℓ unassigned in d        ▷ see section 2.6
6:         for t = 1 ... k do
7:             Create a new child of d with the constraint x_ℓ^t = 1
8:         end for
9:     end if
10:    node selection: choose node d unexplored in 𝕋                  ▷ see section 2.6
11: end while
12: return optimal the optimal solution.
```

Branching

At each node of the tree, $d \in \mathbb{T}$, we keep track of two sub-feasible collections. The first is the collection of fixed vertices $(F_{d,1}, \ldots, F_{d,k})$. The second is the feasible solution to relaxation CKR-IC: $V_{d,i} \subset V$, $i \in \{1, \ldots, k\}$.

We will say that a vertex $\ell \in V$ is *unassigned* in $d \in \mathbb{T}$ if it is not in any of the $V_{d,t}$. In our branching step, we choose an unassigned vertex ℓ in d and create k children of d in $\mathbb{T}$ by assigning ℓ to each of the $F_{d,t}$. Equivalently, we create k children where $x_\ell^t = 1$ for each $t = \{1, \ldots, k\}$.

Algorithm 2.1 is the pseudo-code of the ISOLATING CUT BRANCH-AND-BOUND algorithm. Figure 2.2 provides an illustration.

An important realization is that we do not need to recompute all k minimum isolating cuts at every node in $\mathbb{T}$. In fact, at every node except the root node we only need to recompute one minimum isolating cut: the minimum isolating cut corresponding to the set to which ℓ was added. To be precise, assume that from node d to its child we add the unassigned vertex to $F_{d,i}$:

$$(F_{d,1}, \ldots, F_{d,i}, \ldots, F_{d,k}) \to (F_{d,1}, \ldots, F_{d,i} \cup \ell, \ldots, F_{d,k}).$$

For $j \neq i$, the minimum $F_{d,j}$-isolating cut in the former, $(V_{d,j}, V \setminus V_{d,j})$, must still be a minimum $F_{d,j}$-isolating cut in the latter. This is because ℓ was already in $V \setminus V_{d,j}$, so $(V_{d,j}, V \setminus V_{d,j})$ continues to be a minimum isolating cut. Thus, we only need to recompute the minimum isolating cut for $F_{d,i} \cup \ell$. When we recompute this minimum cut, we will take the minimum isolating cut with maximum source set. For the reasoning behind this, see section 2.5.

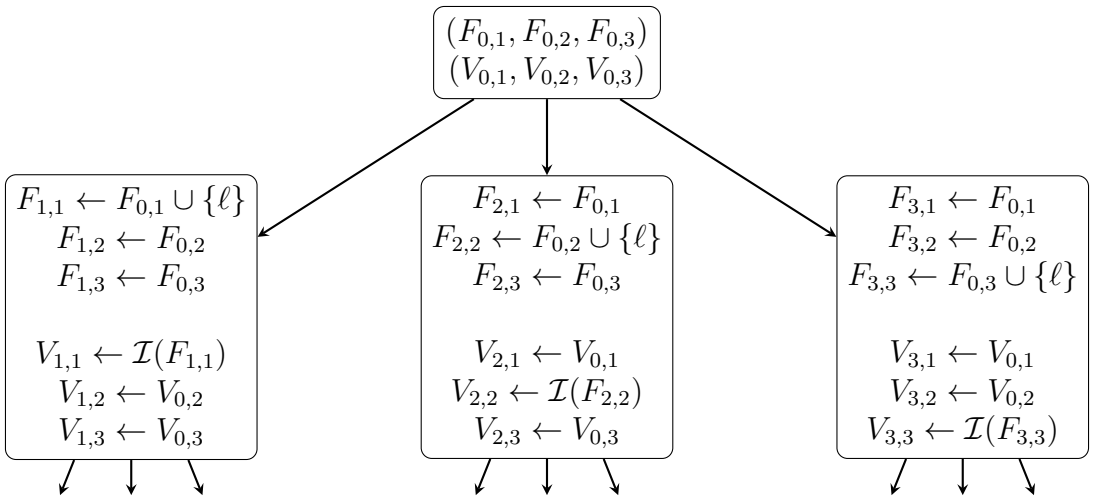


Figure 2.2: Example of the branch-and-bound tree when $k = 3$.

Bounding

As in most branch-and-bound algorithms, at node d in the branch-and-bound tree we derive a bound equal to the objective value of the relaxation at d. The bound at node d serves as a lower bound on any feasible solution that will be found at any descendent of d. Thus, we can prune nodes for which the bound is greater than or equal to the best feasible solution found so far.

At node d in the tree, we compute $L(d)$, the objective function of CKR-IC:

$$L(d) = \frac{1}{2} \sum_{i=1}^{k} w(V_{d,i}, V \setminus V_{d,i}).$$

If there are no unassigned vertices, then d is a leaf node. At a leaf node, the sets $(V_{d,1}, \ldots, V_{d,k})$ induce a feasible solution to the k-`terminal cut` instance.

Even if we are not at a leaf node, we can heuristically generate a good feasible solution at a node d by assigning all the unassigned vertices to the same terminal. To be precise, let

$$U_d = V \setminus \bigcup_{t=1}^{k} V_{d,t}$$

be the unassigned vertices at node d. Without loss of generality, assume that $w(V_k, V \setminus V_k)$ is the largest among the $w(V_i, V \setminus V_i)$. Our proposed feasible solution is

$$(V_{d,1}, V_{d,2}, \ldots, V_{d,k-1}, V_{d,k} \cup U_d).$$

2.4 Isolating Cuts in Stable Instances

Stable instances of k-terminal cut are special instances in which the optimal solution remains optimal even when the weights of the graph are perturbed. In order to formally define γ-stable instances, we first define the notion of a γ-perturbation.

Definition 2.3 (γ-Perturbation)**.** *Let $G = (V, E, w)$ be a weighted graph with edge weights w. Let $G' = (V, E, w')$ be a weighted graph with the same set of vertices V and edges E and a new set of edge weights w' such that, for every $e \in E$ and some $\gamma > 1$,*

$$w(e) \leq w'(e) \leq \gamma w(e).$$

Then G' is a γ-perturbation of G.

Stable instances are instances where the optimal solution remains uniquely optimal for any γ-perturbation of the weighted graph.

Definition 2.4 (γ-Stability)**.** *An instance $\{G = (V, E, w), T\}$ of k-terminal cut is γ-stable ($\gamma > 1$) if there is an optimal solution E_{OPT} which is uniquely optimal for k-terminal cut for every γ-perturbation of G.*

Note that the optimal solution need not be γ times as good as *any* other solution, since two solutions may share many edges. Given an alternative feasible solution, E_{FEAS}, to the optimal cut, E_{OPT}, in a γ-stable instance, we can make a statement about the relative weights of the edges where the cuts differ. The following equivalence was first noted in [43]:

Lemma 2.2 (γ-Stability)**.** *Let $\{G = (V, E, w), T\}$ be an instance of k-terminal cut with optimal cut E_{OPT}. G is γ-stable ($\gamma > 1$) if and only if, for every alternative feasible k-terminal cut $E_{FEAS} \neq E_{OPT}$, we have*

$$w(E_{FEAS} \setminus E_{OPT}) > \gamma w(E_{OPT} \setminus E_{FEAS}).$$

Now, we present a new characterization of γ-stable instances. A key realization is that we do not need to consider every γ-perturbation of G in order to check for γ-stability. In fact, there is one γ-perturbation which is, in a sense, the "worst" one: the γ-perturbation in which we multiply the edges in E_{OPT} by γ and leave the rest unchanged. If we find a set of edges with this property, then our instance is γ-stable. We summarize this idea in lemma 2.3.

Lemma 2.3 (Checking γ-Stability)**.** *Consider an instance of k-terminal cut: $\{G = (V, E, w), T\}$. Let $E^* \subseteq E$ be a subset of the edges. For some $\gamma > 1$, consider the instance $\{G' = (V, E, w'), T\}$, where*

$$w'(e) = \begin{cases} \gamma w(e) & e \in E^* \\ w(e) & e \notin E^*. \end{cases}$$

If E^* is the unique optimal solution to k-terminal cut in $\{G', T\}$, then the instance $\{G, T\}$ is γ-stable and E^* is the unique optimal solution to k-terminal cut in $\{G, T\}$.

Proof. First, we claim that E^* is the unique optimal solution in G. For any alternative feasible cut $\bar{E}$,

$$w'(E^*) < w'(\bar{E}).$$

By construction, $\gamma w(E^*) = w'(E^*)$ and $w'(\bar{E}) < \gamma w(\bar{E})$. Thus,

$$w(E^*) < w(\bar{E}).$$

Thus, E^* is optimal in G.

Now, consider an alternative feasible cut $\bar{E}$. We express its weight in G' in terms of its weight in G:

$$\begin{aligned} w'(\bar{E}) &= w'(\bar{E} \setminus E^*) + w'(\bar{E} \cap E^*) \\ &= w(\bar{E} \setminus E^*) + \gamma w(\bar{E} \cap E^*). \end{aligned}$$

We do the same for E^*:

$$\begin{aligned} w'(E^*) &= w'(E^* \setminus \bar{E}) + w'(E^* \cap \bar{E}) \\ &= \gamma w(E^* \setminus \bar{E}) + \gamma w(E^* \cap \bar{E}). \end{aligned}$$

Thus,

$$w'(E^*) < w'(\bar{E}) \iff \gamma w(E^* \setminus \bar{E}) < w(\bar{E} \setminus E^*).$$

From lemma 2.2, we conclude that $\{G, T\}$ is γ-stable. □

We make a few observations about γ-stability:

Fact 2.1. *Any k-terminal cut instance that is stable with $\gamma > 1$ must have a unique optimal solution.*

Proof. By Definition 2.3, any graph is a γ-perturbation of itself. Thus, by Definition 2.4, the optimal solution must be unique. □

Fact 2.2. *Any k-terminal cut instance that is γ_2-stable is also γ_1-stable for any $1 < \gamma_1 < \gamma_2$.*

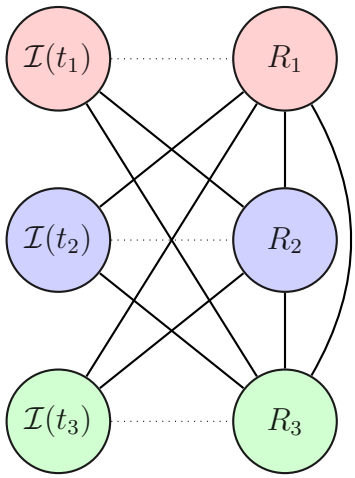
Proof. The set of γ_1-perturbations is a subset of the set of γ_2-perturbations, since

$$w(e) \leq w'(e) \leq \gamma_1 w(e) \implies w(e) \leq w'(e) \leq \gamma_2 w(e).$$

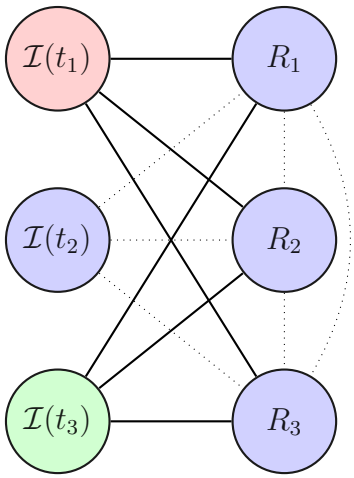
□

Thus, for example, every instance which is 4-stable is necessarily 2-stable. On the other hand, there exist instances which are 2-stable but not 4-stable.

Now we turn to studying the special behavior of isolating cuts in stable instances of the k-terminal cut problem.



(a) The optimal partition, in which each set of vertices, R_i, is in a source set with its respective $\mathcal{I}(t_i)$.



(b) The alternative partition used in theorem 2.1 when $i = 2$, where all of the R are in a source set with $\mathcal{I}(t_2)$.

Figure 2.3: The sets $\mathcal{I}(t_1), \mathcal{I}(t_2), \mathcal{I}(t_3)$ and R_1, R_2, R_3 defined in theorem 2.1 when $k = 3$. Solid lines represent edges which are in the cut. Dashed lines represent edges which are not in the cut.

In lemma 2.1, the qualification that *there exists* an optimal solution in which $\mathcal{I}(t_i) \subseteq V_i^*$ can create complications, since the equation $\mathcal{I}(t_i) \subseteq V_i^*$ need not be simultaneously true for all i. Conveniently, when an instance is γ-stable ($\gamma > 1$), it has a unique optimal solution (fact 2.1). Thus, in such instances, the condition $\mathcal{I}(t_i) \subseteq V_i^*$ will be simultaneously true for all i.

Theorem 2.1. *Let $\{G = (V, E, w), T\}$ be a $(k-1)$-stable instance of k-terminal cut. Then, for all i, $\mathcal{I}(t_i) = V_i^*$.*

Proof. We will primarily be working with the k vertex sets $\mathcal{I}(t_1), \ldots, \mathcal{I}(t_k)$ and the k vertex sets $V_1^* \setminus \mathcal{I}(t_1), \ldots V_k^* \setminus \mathcal{I}(t_k)$. For convenience, we will use the notation $R_i = V_i^* \setminus \mathcal{I}(t_i)$. As a consequence of lemma 2.1, $V_i^* = \mathcal{I}(t_i) \cup R_i$. We will assume, for the sake of contradiction, that at least one R_i is non-empty.

Since $\mathcal{I}(t_i)$ is the source set for the minimum isolating cut for terminal t_i:

$$\begin{aligned}
& w(\mathcal{I}(t_i), V \setminus \mathcal{I}(t_i)) \leq w(V_i^*, V \setminus V_i^*) \\
\iff \quad & w(\mathcal{I}(t_i), V \setminus \mathcal{I}(t_i)) \leq w(R_i, V \setminus V_i^*) + w(\mathcal{I}(t_i), V \setminus V_i^*) \\
\iff \quad & -w(\mathcal{I}(t_i), V \setminus V_i^*) + w(\mathcal{I}(t_i), V \setminus \mathcal{I}(t_i)) \leq w(R_i, V \setminus V_i^*) \\
\iff \quad & w(\mathcal{I}(t_i), R_i) \leq w(R_i, V \setminus V_i^*) \\
\iff \quad & w(\mathcal{I}(t_i), R_i) \leq \sum_{\{j|j\neq i\}} w(R_i, R_j) + \sum_{\{j|j\neq i\}} w(R_i, \mathcal{I}(t_j))
\end{aligned}$$

Summing these inequalities over all the i:

$$\begin{aligned}
& \sum_i w(\mathcal{I}(t_i), R_i) \leq \sum_i \sum_{\{j|j\neq i\}} w(R_i, R_j) + \sum_i \sum_{\{j|j\neq i\}} w(R_i, \mathcal{I}(t_j)) \\
\iff \quad & \sum_i w(\mathcal{I}(t_i), R_i) \leq 2w(R_1, \ldots, R_k) + \sum_i \sum_{\{j|j\neq i\}} w(R_i, \mathcal{I}(t_j)) \qquad (2.1)
\end{aligned}$$

Next, we will consider alternatives to the optimal cut $(V_1^*, \ldots, V_k^*)$ and apply lemma 2.2. The optimal cut can be written as

$$(V_1^*, \ldots, V_k^*) = (\mathcal{I}(t_1) \cup R_1, \ldots, \mathcal{I}(t_k) \cup R_k).$$

We will consider alternative cuts $E_{\text{FEAS}}^{(i)}$ where all the R_j are in the same set of the partition, associated with $\mathcal{I}(t_i)$. That is, we will consider

$$\Big(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_{i-1}), \mathcal{I}(t_i) \cup (R_1 \cup \ldots \cup R_k), \mathcal{I}(t_{i+1}), \ldots, \mathcal{I}(t_k)\Big).$$

See figure 2.3 for an illustration. We assumed that at least one of the R_i is non-empty, so at least $k-1$ of these alternative cuts are distinct from the optimal one[1]. In order to apply lemma 2.2, we need to calculate $w(E_{\text{OPT}} \setminus E_{\text{FEAS}}^{(i)})$ and $w(E_{\text{FEAS}}^{(i)} \setminus E_{\text{OPT}})$.

To calculate $w(E_{\text{FEAS}}^{(i)} \setminus E_{\text{OPT}})$, consider the edges in $E_{\text{FEAS}}^{(i)}$ with one endpoint in $\mathcal{I}(t_j)$ $(j \neq i)$. The only edges which are *not* counted in E_{OPT} are those which go to R_j. Thus,

$$w(E_{\text{FEAS}}^{(i)} \setminus E_{\text{OPT}}) = \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_j)).$$

To calculate $w(E_{\text{OPT}} \setminus E_{\text{FEAS}}^{(i)})$, we must consider the set of edges which are in E_{OPT} but not in $E_{\text{FEAS}}^{(i)}$. For an edge not to be in $E_{\text{FEAS}}^{(i)}$, it must be internal to one of the $\mathcal{I}(t_j)$ $(j \neq i)$

[1]If only one R_i is non-empty, then $E_{\text{OPT}} = E_{\text{FEAS}}^{(i)}$ for this i. The corresponding inequality in Equation 2.2 is not strict (both sides are 0), but the other $k-1$ inequalities are strict and so the average (Equation 2.3) is still a strict inequality.

or internal to $\mathcal{I}(t_i) \cup (R_1 \cup \ldots \cup R_k)$. None of the internal edges of the $\mathcal{I}(t_j)$ are in E_{OPT}, so we need only consider the internal edges of $\mathcal{I}(t_i) \cup (R_1 \cup \ldots \cup R_k)$:

$$w(E_{\text{OPT}} \setminus E^{(i)}_{\text{FEAS}}) = w(R_1, \ldots, R_k) + \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_i)).$$

We apply lemma 2.2, with $\gamma = k - 1$:

$$(k-1) \cdot w(E_{\text{OPT}} \setminus E^{(i)}_{\text{FEAS}}) < w(E^{(i)}_{\text{FEAS}} \setminus E_{\text{OPT}}) \tag{2.2}$$

Substituting in the formulas derived earlier for $w(E_{\text{OPT}} \setminus E^{(i)}_{\text{FEAS}})$ and $w(E^{(i)}_{\text{FEAS}} \setminus E_{\text{OPT}})$:

$$(k-1) \cdot w(R_1, \ldots, R_k) + (k-1) \cdot \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_i)) < \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_j)).$$

Averaging over the k inequalities (one for each i)[1]:

$$(k-1) \cdot w(R_1, \ldots, R_k) + \frac{k-1}{k} \sum_i \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_i)) < \frac{k-1}{k} \sum_i w(R_i, \mathcal{I}(t_i)). \tag{2.3}$$

We combine this with the inequality derived in Equation 2.1:

$$\begin{aligned} & (k-1) \cdot w(R_1, \ldots, R_k) + \frac{k-1}{k} \sum_i \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_i)) \\ < \quad & 2\frac{k-1}{k} w(R_1, \ldots, R_k) + \frac{k-1}{k} \sum_i \sum_{\{j|j\neq i\}} w(R_i, \mathcal{I}(t_j)). \end{aligned}$$

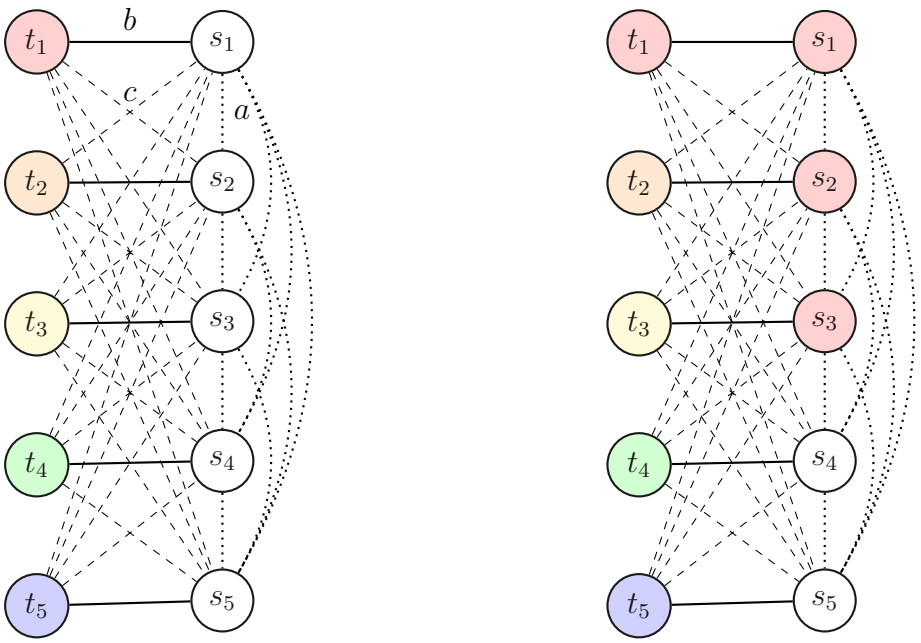
Notice that

$$\sum_i \sum_{\{j|j\neq i\}} w(R_j, \mathcal{I}(t_i)) = \sum_i \sum_{\{j|j\neq i\}} w(R_i, \mathcal{I}(t_j)).$$

Therefore,

$$(k-1) \cdot w(R_1, \ldots, R_k) < 2\frac{k-1}{k} w(R_1, \ldots, R_k).$$

This is a contradiction, so it must be the case that $R_i = \emptyset$ for all i. Thus, $\mathcal{I}(t_i) = V_i^*$ for all i. □



(a) Dotted lines have weight a (between s_i and s_j, $i \neq j$) and are in E^*. Solid lines have weight b (between t_i and s_i) and are not in E^*. Dashed lines have weight c (between t_i and s_j, $i \neq j$) and are in E^*.

(b) When $p = 3$, we assume $V_1^* = \{t_1, s_1, s_2, s_3\}$. It follows that $V_2^* = \{t_2\}$ and $V_3^* = \{t_3\}$. We eventually arrive at a contradiction, showing that this cannot be the optimal cut.

Figure 2.4: The construction used in theorem 2.2 when $k = 5$

To prove that that the factor of $(k-1)$ is tight, it would be sufficient to find an instance which is $(k-1-\epsilon)$-stable, for arbitrarily small $\epsilon > 0$, for which $\mathcal{I}(t_i) \neq V_i^*$ for some i. In fact, we will exhibit an instance with a stronger property: $\mathcal{I}(t_i) \neq V_i^*$ for *all* i.

Theorem 2.2. *For all $0 < \epsilon < k-2$, there exists a $(k-1-\epsilon)$-stable instance of k-terminal cut for which $\mathcal{I}(t_i) = \{t_i\} \neq V_i^*$ for all $i \in \{1, \ldots, k\}$.*

Proof. Consider a graph with $2k$ vertices. There are k terminals, $T = \{t_1, \ldots, t_k\}$, and k other vertices $S = \{s_1, \ldots, s_k\}$. The $\binom{k}{2}$ edges between s_i and s_j $(i \neq j)$ have weight $a \in \mathbb{R}^+$. The k edges from t_i to s_i have weight $b \in \mathbb{R}^+$. The $k(k-1)$ edges from t_i to s_j $(i \neq j)$ have weight $c \in \mathbb{R}^+$. No other edges exist. Call this graph G_k. See figure 2.4a for an illustration. We will show that this graph has the property $\mathcal{I}(t_i) = \{t_i\} \neq \{t_i, s_i\} = V_i^* \ \forall i$

for the following choices of $a, b, c \in \mathbb{R}^+$:

$$\begin{aligned} a &= 2\epsilon \\ b &= k(k-1)(k-1-\epsilon) \\ c &= k(k-1) - (k+1)\epsilon. \end{aligned}$$

For $0 < \epsilon < k-2$, these values of a, b, and c are all well-defined and G_k is $(k-1-\epsilon)$-stable.

We will be using the idea presented in lemma 2.3. Consider the collection of k sets $V_i = \{t_i, s_i\}$ for $i \in \{1, \ldots, k\}$. Let E^* be the set of edges which go between these sets. That is, E^* consists of all the edges of weight a and all the edges of weight c. Let G'_k be the graph in which the edges of weight b have their weight divided by $(k-1-\epsilon)$ (which is equivalent to multiplying the edges of E^* by $(k-1-\epsilon)$ and then re-scaling):

$$\begin{aligned} a' &= 2\epsilon \\ b' &= k(k-1) \\ c' &= k(k-1) - (k+1)\epsilon. \end{aligned}$$

We will show that $V_i = \{t_i, s_i\}$ is the unique optimal solution in G'_k. From lemma 2.3, it will follow that the instance $\{G_k, T\}$ is $(k-1-\epsilon)$-stable with optimal cut E^*.

Let $(V_1^*, \ldots, V_k^*)$ be the optimal solution to k-`terminal cut` in G'_k, where $t_i \in V_i^*$. Note that minimizing the cut between the V_i^* is equivalent to maximizing the edges internal to to the V_i^*. For this proof, it is easier to think in terms of this equivalent maximization problem.

First, we claim that

$$|V_i^*| > 1 \implies s_i \in V_i^*.$$

The condition $|V_i^*| > 1$ means that V_i^* consists of more than just the terminal t_i. Assume, for the sake of contradiction, that $|V_i^*| > 1$ but $s_i \notin V_i^*$. There must be some $s_j \neq s_i$ in V_i^* and there must be some V_ℓ^* such that $s_i \in V_\ell^*$. Consider a "swap" operation, where we assigned s_i to V_i^* and s_j to V_ℓ^*. The total number of edges of weight a' internal to V_i^* and V_ℓ^* remains unchanged. However, the edge $\{t_i, s_i\}$ of weight b' is now internal to V_i^*, replacing the edge $\{t_i, s_j\}$ of weight c'. This improves the cut. Thus, it must be the case that $s_i \in V_i^*$.

Without loss of generality, assume that V_1^* is the largest set among the V_i^* and that V_1^* contains $\{s_1, \ldots, s_p\}$. From our claim in the previous paragraph, it follows that $|V_2^*| = \ldots = |V_p^*| = 1$. See figure 2.4b for an illustration. Internal to V_1^* are $\binom{p}{2}$ edges of weight a', 1 edge of weight b', and $p-1$ edges of weight c'. What if, instead, we re-assigned the s_j to their respective V_j, for $j \in \{2, \ldots, p\}$? If we did this, consider the edges internal to $V_1^*, \ldots, V_p^*$: 0 edges of weight a', p edges of weight b', and 0 edges of weight c'. The improvement would be:

$$pb' - (\binom{p}{2}a' + b' + (p-1)c') = (p-1)(k+1)\epsilon - p(p-1)\epsilon > 0.$$

This proves that $V_1^* = \{t_1, s_1\}$ and, therefore, $V_i^* = \{t_i, s_i\}$, as desired.

Last but not least, we need to check that, as claimed, the minimum isolating cuts in G_k have trivial source sets. Knowing that $V_i^* = \{t_i, s_i\}$ and, thanks to lemma 2.1, that

$\mathcal{I}(t_i) \subseteq V_i^*$, there are only two possibilities for $\mathcal{I}(t_i)$. Either $\mathcal{I}(t_i) = \{t_i\}$ or $\mathcal{I}(t_i) = \{t_i, s_i\}$. $\mathcal{I}(t_i) = \{t_i\}$ if and only if

$$\begin{aligned} & b + (k-1)c < (k-1)a + 2(k-1)c \\ \iff \quad & b < (k-1)(a+c) \\ \iff \quad & k(k-1-\epsilon) < k(k-1) - (k+1)\epsilon + 2\epsilon \\ \iff \quad & 0 < \epsilon \end{aligned}$$

Thus, our construction has the property $\mathcal{I}(t_i) = \{t_i\} \neq \{t_i, s_i\} = V_i^* \ \forall i$ for any ϵ in $(0, k-2)$. □

Approximation Revisited

From theorem 2.2, it follows that the $(2-2/k)$-approximation algorithm of Dahlhaus et al. does *not* deliver an optimal solution on the constructed $(k-1-\epsilon)$-stable instance. It is interesting to consider whether the approximation ratio is better than $(2-2/k)$. Let E_{ISO} be the union of the edges in all the isolating cuts except the one with largest weight. We calculate, for $\epsilon \in (0, k-2)$:

$$\begin{aligned} \frac{w(E_{\text{ISO}})}{w(E_{\text{OPT}})} &= \frac{(k-1)(b+(k-1)c)}{\binom{k}{2}a + k(k-1)c} \\ &= \frac{(k-1)(2k(k-1)(k-1-\epsilon) - (k-1)\epsilon)}{k(k-1)\epsilon + k(k-1)k(k-1-\epsilon) - k(k-1)\epsilon} \\ &= \frac{2k(k-1)(k-1-\epsilon) - (k-1)\epsilon}{k^2(k-1-\epsilon)} \\ &= (2-2/k) - \frac{(k-1)\epsilon}{k^2(k-1-\epsilon)}. \end{aligned}$$

When $\epsilon = 0$, $w(E_{\text{ISO}}) = w(E_{\text{OPT}})$. Thus,

$$\frac{w(E_{\text{ISO}})}{w(E_{\text{OPT}})} = \begin{cases} (2-2/k) - \frac{(k-1)\epsilon}{k^2(k-1-\epsilon)} & \epsilon \in (0, k-2) \\ 1 & \epsilon = 0. \end{cases}$$

This result is somewhat surprising. While one might expect the approximation ratio to improve on more stable instances, we find the opposite: as $\epsilon \to 0$, the $(k-1-\epsilon)$-stable instance becomes more stable, but the approximation ratio gets worse (larger).

2.5 Complexity Analysis

In Isolating Cut Branch-and-Bound, the number of children of each tree node is k, because we consider adding the selected unassigned vertex to each of the k possible terminals.

Recall that

$$L(d) = \frac{1}{2}\sum_{i=1}^{k} w(V_{d,i}, V \setminus V_{d,i})$$

and that $V_{d,t} = \mathcal{I}(F_{d,t})$. As we go from a parent to its child, we fix an unassigned vertex ℓ to a terminal t and take a new isolating cut: $\mathcal{I}(F_{d,t} \cup \ell)$. This isolating cut also isolates $F_{d,t}$. If we assume that $\mathcal{I}(F_{d,t})$ had maximum source set, then $\mathcal{I}(F_{d,t} \cup \ell)$ cannot be a minimum isolating cut for $F_{d,t}$, so it must have strictly larger weight. When the edge weights are integer, the increase must be at least $\frac{1}{2}$. $L(d)$ is at least $\frac{w(E_{\text{OPT}})}{2}$ at the root node ($d = 0$) and exactly $w(E_{\text{OPT}})$ at a node with an optimal solution. Thus, $w(E_{\text{OPT}})$ is a bound on the depth of the tree. If we sum over the number of possible nodes at depths $1, 2, \ldots, w(E_{\text{OPT}})$, we see that the number of nodes considered is at most

$$1 + k + k^2 + \ldots + k^{w(E_{\text{OPT}})} < 2k^{w(E_{\text{OPT}})}.$$

Let $C(n, m)$ be the complexity of evaluating a minimum (s, t)-cut on a graph with n vertices and m edges. The complexity of Isolating Cut Branch-and-Bound is thus $O(2k^{w(E_{\text{OPT}})}C(n, m))$ on general instances. From this, we have fixed-parameter tractability. In $(k-1)$-stable instances, Isolating Cut Branch-and-Bound terminates after computing the initial k isolating cuts. The complexity of computing k minimum isolating cuts is $O(kC(n, m))$.

Using the push-relabel maximum flow algorithm, we have $C(n, m) = O(mn \log \frac{n^2}{m})$. If we ignore logarithmic factors and assume $m > n$, then the complexity of calculating k minimum isolating cuts is $\tilde{O}(km^2)$. Earlier, Angelidakis et al. proved that $(2 - 2/k)$-stable instances of `k-terminal cut` can be solved to optimality by solving the linear programming relaxation of the CKR formulation (formulation CKR-LP). In these stable instances, the solution to the relaxation will necessarily be integer [2]. The number of variables in the CKR relaxation is $k(n + m)$ and the number of constraints is $n + 2km$. Because the CKR linear program has a binary constraint matrix, it is known that it can be solved in strongly polynomial time [54]. That said, even if we knew exactly which variables were in the basis in the optimal solution, determining their values would require solving a system of linear equations with $\min\{k(n + m), n + 2km\}$ variables and equations. Such an operation would be at least quadratic in the number of variables. This suggests that solving the CKR relaxation has complexity $\Omega(k^2m^2)$. In fact, using the current best-known algorithms for matrix inversion, the complexity of solving the relaxation would be $\Omega(k^{2.37}m^{2.37})$ [15, 41].

2.6 Empirical Study

Isolating Cut Branch-and-Bound Implementation

Our implementation is available online at

`https://github.com/marvel2010/k-terminal-cut`

and works as a Python package (ktcut). It represents graphs using NetworkX [28]. We chose Python for ease of implementation and portability, even though it is not the fastest language in terms of its practical running time [58].

As in linear programming branch-and-bound, in Isolating Cut Branch-and-Bound we must specify two strategies: the variable-selection strategy (line 5, algorithm 2.1) and the node-selection strategy (line 10, algorithm 2.1).

Branching Variable At each tree node, we must select an unassigned graph vertex, ℓ, to branch on. Our branching strategy affects the k variables associated with vertex ℓ, one for each terminal. For this reason, we use "branching variable" and "branching vertex" interchangeably.

Branching Node After exploring a node in the tree, we must decide which node to explore next.

Branching Variable Strategy: For choosing the branching vertex, we considered a few options. The options included choosing a vertex randomly, choosing the vertex farthest from an existing source set, or choosing the vertex of largest degree. Initial experiments suggested that the last strategy was best (largest degree), so our results use that strategy. In our implementation, we contract source sets into a single terminal vertex at each node in the branch-and-bound tree. This allows subsequent minimum cuts to be evaluated on smaller graphs.

Branching Node Strategy: For the branching node, we chose the "least bound" heuristic: that is, we always chose the unexplored node with the smallest lower bound, $L(d)$, with ties broken by fewer unassigned vertices. We considered two other heuristics: choosing the node with the fewest unassigned vertices and choosing the node with smallest sum ($L(d)$ plus unassigned vertices). We found that the "least bound" heuristic performed best, in practice.

Comparison to Linear Programming Branch-and-Bound

To compare our algorithm, Isolating Cut Branch-and-Bound, to branch-and-bound using the linear programming relaxation (relaxation CKR-LP), we used Gurobi, a popular commercial software package for mixed-integer programming. Strictly speaking, the constraint $z^t_{ij} \in \{0, 1\}$ did not need to be specified, since it is implied. However, we found that specifying it explicitly helped Gurobi solve instances faster. We suspect this is because the z^t_{ij} variables are good variables to branch on. All of Gurobi's hyper-parameters were set to their default values. No additional performance tuning was undertaken.

Data Sets

Real Data Sets: The twenty-four real-world data sets we used for experimentation were gathered from two sources. The first source was the DIMACS Implementation Challenge.

According to the website, "These real-world graphs are often used as benchmarks in the graph clustering and community detection communities." These data sets are available online at

`https://www.cc.gatech.edu/dimacs10/`.

The second source was KONECT, an online repository of popular graph datasets. These data sets are available online at

`http://konect.uni-koblenz.de/`.

In most of the data sets, the graphs are already connected. In the rest, we only considered the largest connected component, otherwise the k-`terminal cut` problem decomposes into smaller problems on each component.

Simulated Data Sets: To systematically study the running time scaling of Isolating Cut Branch-and-Bound, we used simulated graphs. It has been observed that many real-world graphs, from social networks to computer networks to metabolic networks, exhibit both a power-law degree distribution and high clustering [30]. The Power Cluster model, introduced by Holme and Kim [30], generates random graphs which exhibit both of these properties. NetworkX includes a tool for randomly generating graphs according to the Powerlaw Cluster model with three parameters: the number of vertices, the number of random edges to add for each vertex, and the probability of creating a triangle. In our scaling experiment, we vary the first parameter (the number of vertices) while leaving the latter two fixed at 10 and 0.1, respectively.

Terminals: In the data sets, terminals are not specified. In order to find suggested terminals, we do the following: first, we perform spectral clustering on the graph to get an approximate clustering of the graph into k clusters (by performing k-means clustering on the spectral embedding of the graph). Next, we choose the largest-degree vertex in each cluster and set those vertices to be our k terminals. By selecting one terminal in each cluster, we hope to increase our odds of creating k-`terminal cut` instances with non-trivial solutions.

Results

Properties of Isolating Cut Branch-and-Bound

Before presenting the main comparison, we break down the performance of Isolating Cut Branch-and-Bound. On the twenty-four real data sets, we report the number of minimum isolating cuts calculated, as well as the value of the optimal objective. The analysis is repeated in real graphs with five terminals (table A.1) and with ten terminals (table A.3). The graphs and method for choosing the terminals were described in section 2.6.

One property of the objective function of k-`terminal cut` is that it can occasionally lead to solutions where most of the graph is assigned to one component of the partition.

Since this is of interest, in addition to reporting the value of the optimal solution we also report the fraction of the graph which is in the largest partition of the optimal solution.

From this data, it appears that the running time of Isolating Cut Branch-and-Bound is correlated with the size of the graph and the number of minimum isolating cuts performed. It does not appear to be correlated with the properties of the optimal solution we considered.

Comparison to Linear Programming Branch-and-Bound

Next, we compare Isolating Cut Branch-and-Bound to linear programming (LP) branch-and-bound with Gurobi. We compare on the twenty-four real-world test graphs, first with five terminals (table A.2) and then with ten terminals (table A.4). Isolating Cut Branch-and-Bound is faster than LP branch-and-bound on twenty-three of the twenty-four instances with five terminals and nineteen of the twenty-four instances with ten terminals. Isolating Cut Branch-and-Bound provides a median speedup of 18× in instances with 5 terminals and a median speedup of 26× in instances with 10 terminals.

To systematically investigate the scaling of Isolating Cut Branch-and-Bound, we simulated random instances (as described in section 2.6). Properties of the optimal solution can be found in table A.5. Running time comparisons can be found in table A.6 and figure A.1. The running time is the average running time of each algorithm across ten randomly generated k-`terminal cut` instances. The error bars in the figure reflect the standard deviation of running time across those instances. In these simulated data sets, Isolating Cut Branch-and-Bound scales better than LP branch-and-bound to large instances. In simulated instances with $30,000$ edges, Isolating Cut Branch-and-Bound provides a factor of 3× speedup. In simulated instances with $90,000$ edges, the speedup is a factor of 14×. We believe that this behavior can largely be attributed to the slow growth in the number of minimum isolating cuts performed by Isolating Cut Branch-and-Bound. In simulated instances with $\sim 30,000$ edges, Isolating Cut Branch-and-Bound performs 70 minimum isolating cuts on average. In simulated instances with $\sim 90,000$ edges, Isolating Cut Branch-and-Bound performs only 130 minimum isolating cuts on average.

2.7 Conclusions

In this chapter, we developed two new ways to use isolating cuts to solve the k-`terminal cut` problem. First, we introduced Isolating Cut Branch-and-Bound, a new fixed-parameter tractable branch-and-bound algorithm devised for solving the k-`terminal cut` problem. In our empirical study, we found that Isolating Cut Branch-and-Bound is an order of magnitude faster than linear programming branch-and-bound on twenty-four real-world benchmark instances. On simulated data, the Isolating Cut Branch-and-Bound algorithm scales better from small to large instances. Second, we proved that, in $(k-1)$-stable instances of k-`terminal cut`, the source sets of the minimum isolating cuts

recover the unique optimal solution to that k-`terminal cut` instance. As an immediate corollary, we concluded that, on $(k-1)$-stable instances, Isolating Cut Branch-and-Bound returns the optimal solution after calculating just k minimum cuts. On the other hand, we constructed $(k-1-\epsilon)$-stable instances of k-`terminal cut` $(0 < \epsilon < k-2)$ where the isolating cuts in the root node do not return the optimal solution.

Chapter 3

Valid Distance Drawings of Signed Graphs

3.1 Introduction

A signed graph is an undirected graph where each edge has an associated sign, positive or negative. Kermarrec and Thraves [36] introduced the definition of a *valid distance drawing* for signed graphs. A drawing of a signed graph in $\mathbb{R}^k$ is an injection of the set of vertices into $\mathbb{R}^k$. A drawing is said to be *valid distance*, or simply *valid*, if, for every vertex, its positive neighbors are closer than its negative neighbors with respect to the Euclidean distance. The problem of drawing signed graphs in $\mathbb{R}^k$ has received increasing attention in recent years due to its applications in social networks, such as in opinion formation [50], consensus decision-making [1], the evolution of beliefs [56], and community detection [12].

Cygan et al. [18] gave a characterization of the set of complete signed graphs with a valid distance drawing in $\mathbb{R}^1$: a complete signed graph, G, has a valid distance drawing in $\mathbb{R}^1$ if and only if its positive subgraph is a proper interval graph. A *proper* interval graph, also called a *unit* interval graph, is an interval graph in which no interval is contained inside another or, equivalently, in which every interval has unit length. Determining if a graph is a proper interval graph can be done in linear time [16]. However, for general signed graphs (not necessarily complete), it was shown that deciding whether or not a graph has a valid distance drawing in $\mathbb{R}^1$ is an NP-complete problem [18]. This result implies that finding the smallest k such that a *given* signed graph has a valid distance drawing in $\mathbb{R}^k$ is an NP-hard problem.

Several researchers have designed algorithms that provide meaningful representations of signed graphs in $\mathbb{R}^2$ [59, 60, 61]. These representations do not necessarily satisfy the condition required for a drawing to be a valid distance drawing, since there are signed graphs without a valid distance drawing in $\mathbb{R}^2$ [36].

The following question remains open: What is the smallest dimension, $L(n)$, such that *any* signed graph with n vertices has a valid distance drawing in $\mathbb{R}^{L(n)}$? In this work we provide

upper and lower bounds on $L(n)$: $\lfloor \log_5 (n-3) \rfloor + 1 \leq L(n) \leq n-2$. We also determine exact values for $L(n)$ up to $n = 7$. In order to prove the upper bound $L(n) \leq n-2$, we apply a result from distance geometry. For the lower bound $L(n) \geq \lfloor \log_5 (n-3) \rfloor + 1$, we bound the size of a graph that has a valid distance drawing with a hypersphere packing problem.

This chapter is organized as follows. In section 3.2, we discuss preliminaries. In section 3.3, we prove that $L(n) \leq n-2$. In section 3.4, we prove that $L(n) \geq \lfloor \log_5 (n-3) \rfloor + 1$. In section 3.5, we compute exact values for $L(n)$ up to $n = 6$. In section 3.6, we determine the exact value for $L(7)$ and an upper bound for $L(8)$ via computational experiments. We offer some final remarks in section 3.7.

3.2 Preliminaries

We formally define signed graphs and valid distance drawings. We consider only finite, undirected graphs with no parallel edges and no self-loops. A *signed graph* is defined as follows:

Definition 3.1. *A* signed graph $G = (V, (E^+, E^-))$ *consists of a set of vertices, V, as well as two disjoint sets of edges: the* positive *edges E^+ and the* negative *edges E^-.*

Given a signed graph $G = (V, (E^+, E^-))$, we define *positive neighbors* and *negative neighbors* for each vertex in G. Let us define the set of *positive neighbors* of a vertex v as the set

$$N_v^+ = \{u \in V : \{v, u\} \in E^+\}.$$

Similarly, let us define the set of *negative neighbors* of vertex v as the set

$$N_v^- = \{u \in V : \{v, u\} \in E^-\}.$$

The *positive subgraph* of a signed graph $G = (V, (E^+, E^-))$ is the signed graph $G^+ = (V, (E^+, \emptyset))$ on the same set of vertices as G but with only the positive edges of G. If a signed graph is complete, it is sufficient to specify only its positive subgraph. The remaining edges are negative.

One type of signed graph we will use frequently is the complete signed graph on $a + b$ nodes where the positive subgraph equals the complete bipartite graph with a nodes on one side and b nodes on the other side. We define special notation for this graph:

$$Q_{a,b} = (V, (K_{a,b}, K_{a+b} \setminus K_{a,b})),$$

where $|V| = a + b$, K_{a+b} is the complete graph on $a + b$ vertices and $K_{a,b}$ is the complete bipartite graph between a and b vertices. See figure 3.1 for an illustration of $Q_{2,3}$.

For a signed graph $G = (V, (E^+, E^-))$, let $D : V \to \mathbb{R}^k$ be an injection of the set of vertices of G into $\mathbb{R}^k$. We call D a *drawing* of G in $\mathbb{R}^k$. Moreover, we define the validity of a drawing as follows:

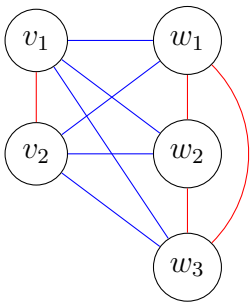


Figure 3.1: An example of $Q_{a,b}$ when $\{a, b\} = \{2, 3\}$. The blue edges have positive sign and the red edges have negative sign.

Definition 3.2 (Valid Distance Drawing)**.** *Let $G = (V, (E^+, E^-))$ be a signed graph, and let D be a drawing of G in $\mathbb{R}^k$. We say that D is a* valid distance drawing *if, $\forall v \in V$, $\forall u^+ \in N_v^+$, and $\forall u^- \in N_v^-$:*

$$d(D(v), D(u^+)) < d(D(v), D(u^-)), \tag{3.1}$$

where $d(x, y)$ denotes the Euclidean distance between $x, y \in \mathbb{R}^k$.

Definition 3.2 captures the requirement that every vertex is closer to its positive neighbors than to its negative neighbors. In the case that there exists a valid distance drawing of a given signed graph G in $\mathbb{R}^k$, we say that G *has a valid distance drawing* in $\mathbb{R}^k$. Now we formally define the function L.

Definition 3.3. *$L(n)$ is the smallest dimension such that every signed graph on n vertices has a valid distance drawing in $\mathbb{R}^{L(n)}$.*

To determine $L(n)$, it is sufficient to consider only complete signed graphs, since a valid distance drawing of a complete signed graph remains valid when some of its edges are removed.

3.3 Upper Bound on L(n)

In this section, we show that any signed graph on n vertices has a valid distance drawing in a Euclidean space of dimension $n - 2$. We rely on a previous result from distance geometry about drawing polytopes that are close to a simplex [21].

Consider the set of $\binom{n+1}{2}$ positive real numbers $\{\ell_{ij} | 1 \leq i < j \leq n + 1\}$. Dekster and Wilker [21] give conditions such that there exists $p_1, \ldots, p_{n+1} \in \mathbb{R}^n$ such that $d(p_i, p_j) = \ell_{ij}$. Their result can be thought of as a generalization of the triangle inequality to higher dimensions. They prove that, for some function $\lambda(n)$, if $\lambda(n) \leq \ell_{ij} \leq 1$ for all i, j, then the set of points $p_1, \ldots, p_{n+1}$ necessarily exists. For example, $\lambda(2) = \frac{1}{2}$.

Given a signed graph G on $n+1$ vertices and a sufficiently small ϵ, we can require that every vertex is *exactly* at a distance of $1-\epsilon$ from its positive neighbors and 1 from its negative neighbors. As long as $\lambda(n) \leq 1-\epsilon$, this set of edge lengths forms an approximate simplex which can be injected in $\mathbb{R}^n$ with no violation of restriction (3.1). This shows that any signed graph on $n+1$ vertices has a valid distance drawing in $\mathbb{R}^n$, so $L(n) \leq n-1$.

Using ideas presented in [21], we go a step further and show that any graph on $n+2$ vertices has a valid distance drawing in $\mathbb{R}^n$. Thus, $L(n) \leq n-2$.

Theorem 3.1. *Let $G = (V, (E^+, E^-))$ be a signed graph such that $|V| = n+2$. Then, G has a valid distance drawing in $\mathbb{R}^n$.*

Proof. Assume that at least one edge, e, is negative. Otherwise, the existence of a valid distance drawing is trivial since any drawing will satisfy condition (3.1). Label the vertices incident to e as v_{n+1} and v_{n+2}. The rest of the vertices are vertices v_1 through v_n.

We will draw G such that every positive edge has distance $1-\epsilon$ and every negative edge except e has distance 1. To accomplish this, let

$$D_1 : V \setminus \{v_{n+1}\} \to \mathbb{R}^n$$

be a valid injection of the vertices except v_{n+1} in which every positive edge has length exactly $1-\epsilon$ and every negative edge has length exactly 1. Such a drawing necessarily exists for sufficiently small ϵ, by the theorem of [21]. Now, let

$$D_2 : V \setminus \{v_{n+2}\} \to \mathbb{R}^n$$

by a valid injection of the vertices except v_{n+2} in which every positive edge has length exactly $1-\epsilon$ and every negative edge has length exactly 1. The drawings D_1 and D_2 are congruent on the vertices $v_1, \ldots, v_n$. We can thus require that the drawing D_2 agrees with D_1 on the vertices $v_1, \ldots, v_n$. Let H be a hyperplane of dimension $n-1$ which passes through vertices $v_1, \ldots, v_n$[1]. We can further require that $D_1(v_{n+2})$ and $D_2(v_{n+1})$ are on opposite sides of H.

Let D be the drawing which equals D_1 and D_2 on vertices v_1 through v_n and $D(v_{n+2}) = D_1(v_{n+2})$ and $D(v_{n+1}) = D_2(v_{n+1})$. We claim that D is a valid distance drawing. In the drawing D, every edge except for e had length $1-\epsilon$ if it is positive and 1 if it is negative. By construction, the negative edge e has length at least $\sqrt{2} - O(\epsilon)$, which is strictly larger than 1 for sufficiently small ϵ. Thus, the properties of a valid distance drawing are satisfied. □

3.4 Lower Bound on $L(n)$

In this section, we lower bound $L(n)$ by establishing a correspondence between finding valid drawings and hypersphere packing. In particular, we consider the graph $Q_{2,p(k)}$. We show that this graph has no valid distance drawing in $\mathbb{R}^k$ when $p(k) > 5^k$.

[1] If $\epsilon < 1 - \lambda(n)$ (strictly), then the drawings D_1 and D_2 are "non-degenerate" (to use the terminology of [21]) and the hyperplane H is unique. This uniqueness of H is not required for our proof.

Lemma 3.1. *The signed graph $Q_{2,p(k)}$ has no valid distance drawing in $\mathbb{R}^k$ for $p(k) > 5^k$.*

Proof. Assume, for the sake of contradiction, there exists a valid distance drawing of $Q_{2,p(k)}$ in $\mathbb{R}^k$. Let v_1 and v_2 denote the two vertices on one side of $Q_{2,p(k)}$ and w_1 through $w_{p(k)}$ denote the vertices on the other side. See figure 3.2 for a helpful illustration.

Without loss of generality, we scale the drawing such that the distance between v_1 and v_2 is exactly 1. For every vertex w_i, the distance to its farthest positive neighbor is at least $\frac{1}{2}$, since it is connected to both v_1 and v_2 via a positive edge: $\max\{d(w_i, v_1), d(w_i, v_2)\} \geq \frac{1}{2}$.

This implies that the distance between w_i and w_j for any $j \neq i$ is strictly larger than $\frac{1}{2}$, since w_i and w_j are connected via a negative edge: $d(w_i, w_j) > \frac{1}{2}$.

As a result, we can draw $p(k)$ balls of radius $\frac{1}{4}$, one around each of the w_i, that do not intersect. The centers of these $p(k)$ balls are inside $B(v_1, 1)$, since $d(v_1, w_i)$ must be less than $d(v_1, v_2)$. The balls lie entirely inside $B(v_1, 1 + \frac{1}{4})$ because the radius of each ball is $\frac{1}{4}$.

We bound $p(k)$ with a packing problem. When there exists a valid distance drawing, $p(k)$ is at most the number of balls of radius $\frac{1}{4}$ that can be packed into a ball of radius $1 + \frac{1}{4}$. The ratio between the volumes of the balls implies that:

$$p(k) \leq \frac{(1 + 1/4)^k}{(1/4)^k} = 5^k.$$

Thus, when $p(k) > 5^k$, there is no valid distance drawing of $Q_{2,p(k)}$ in $\mathbb{R}^k$. □

From lemma 3.1, we see that we need at least $k + 1$ dimensions to embed every graph on $5^k + 3$ vertices. In other words, we need at least $\lfloor \log_5 (n - 3) \rfloor + 1$ dimensions for every graph on n vertices to have a valid distance drawing. Thus, we have

Theorem 3.2. $L(n) \geq \lfloor \log_5 (n - 3) \rfloor + 1$.

It is tempting to try and improve this lower bound using a tighter analysis of the same construction. While improvement is possible, asymptotic improvement with this construction is *not* possible. That is, the lower bound derived for $L(n)$ by tightening this analysis will still be $\Theta(\log n)$. To prove this, in lemma 3.2 we will construct a valid distance drawing of $Q_{2,p(k)}$ exists where $p(k)$ is an exponential function of k. Thus, obtaining a stronger asymptotic bound on $L(n)$ will require a different construction.

Lemma 3.2. *The signed graph $Q_{2,p(k)}$ has a valid distance drawing in $\mathbb{R}^k$ for some value $p(k) \geq \frac{1}{4}(\frac{4}{\sqrt{15}})^{k-2}$.*

Proof. We continue using the notation from lemma 3.1. Let v_1 and v_2 denote the two vertices in on one side of $Q_{2,p(k)}$ and let w_1 through $w_{p(k)}$ denote the vertices on the other side. We begin by drawing v_1 and v_2 in $\mathbb{R}^k$:

$$D(v_1) = (-\frac{1}{2}, 0, 0, \ldots, 0).$$

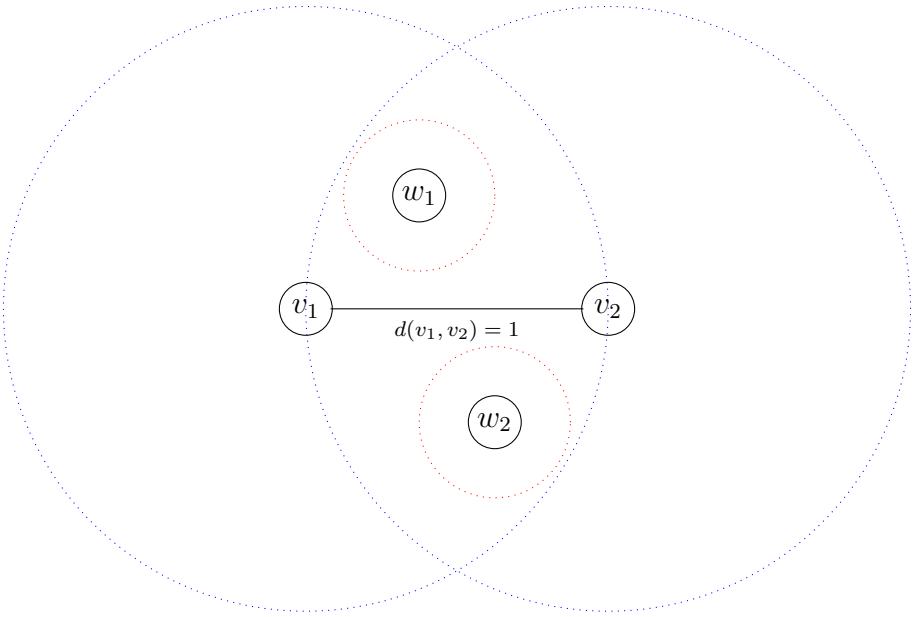


Figure 3.2: The construction used in lemma 3.1 when $p(k) = 2$. The w_i must lie inside the blue circles, but outside of each other's red circles.

$$D(v_2) = (\frac{1}{2}, 0, 0, \ldots, 0).$$

Thus, $d(D(v_1), D(v_2)) = 1$. To ensure that $d(D(v_i), D(w_j)) = \sqrt{1-\epsilon}$, we will draw each $w_1, \ldots, w_{p(k)}$ at a point $D(w_j) = (x_{j1}, x_{j2}, \ldots, x_{jk})$ such that

$$x_{j1} = 0 \;\; \& \;\; x_{j2}^2 + x_{j3}^2 + \ldots + x_{jk}^2 = \frac{3}{4} - \epsilon.$$

Notice that the $D(w_j)$ lie on a hypersphere of dimension $k-1$ with radius $\sqrt{\frac{3}{4} - \epsilon}$. A *spherical code* is defined as a set of points on a hypersphere such that the angle between every pair of points is at least θ [31]. Consider what happens when we stipulate that the w_j must have angle $\theta = \cos^{-1} \frac{1}{4}$ between them:

$$d(D(w_i), D(w_j)) = \sqrt{(\frac{3}{4} - \epsilon) + (\frac{3}{4} - \epsilon) - 2(\frac{3}{4} - \epsilon)\frac{1}{4}} = \sqrt{\frac{9}{8} - \frac{3}{2}\epsilon}.$$

A drawing constructed this way is valid when ϵ is sufficiently small:

$$\begin{aligned} d(D(w_j), D(v_1)) = d(D(w_j), D(v_2)) < d(D(w_j), D(w_i)) &\iff \sqrt{1-\epsilon} < \sqrt{\frac{9}{8} - \frac{3}{2}\epsilon} \\ &\iff \epsilon < \frac{1}{4}. \end{aligned}$$

Jenssen et al. [31] give a lower bound on the size of the largest spherical code in the for any dimension and angle. In dimension $k-1$ with angle θ, it is:

$$p(k) \geq (1+o(1))\sqrt{2\pi(k-1)}\frac{\cos\theta}{\sin^{(k-2)}\theta}$$

When $\theta = cos^{-1}\frac{1}{4}$, as it is in our construction,

$$p(k) \geq \frac{1/4}{(\sqrt{15}/4)^{k-2}} = \frac{1}{4}(\frac{4}{\sqrt{15}})^{k-2}.$$

□

3.5 Exact Computation for $n \leq 6$

In this section, we calculate the exact values of $L(n)$ for $n \leq 6$.

$L(1) = 0$**.** A signed graph with only one vertex has a valid distance drawing in $\mathbb{R}^0$ because there is no constraint (3.1) to be satisfied.

$L(2) = 1$**.** Again, there is no constraint (3.1) to be satisfied. However, at least two distinct points are needed for the valid distance drawing to be an injection. Hence, at least dimension 1 is required.

$L(3) = 1$**.** Any signed graph on 3 vertices has a proper interval graph as its positive subgraph. Hence, every signed graph on 3 vertices has a valid distance drawing in $\mathbb{R}^1$.

$L(4) = 2$**.** $K_{2,2}$ is not a proper interval graph. Therefore, $Q_{2,2}$ does not have a valid distance drawing in $\mathbb{R}^1$. From theorem 3.1, $L(4) \leq 2$. Thus, we conclude that $L(4) = 2$.

$L(5) = 3$**.** We again apply theorem 3.1 and the fact that the signed $Q_{2,3}$ does not have a valid distance drawing in $\mathbb{R}^2$, as was shown in [36].

The remainder of this section is devoted to showing that $Q_{3,3}$ does not have a valid distance drawing in $\mathbb{R}^3$. Therefore $L(6) = 4$. First, we present a key lemma. See figure 3.3 for a helpful illustration.

Lemma 3.3. *Let C be a disk in $\mathbb{R}^2$ defined by a center c and a point p on its border. Let C_1, C_2, and C_3 be disks in $\mathbb{R}^2$ with centers c_1, c_2, and c_3, respectively, such that the following properties hold:*

- *$c_i \in C$ for $i = 1, 2, 3$.*
- *$c \in C_i$ for $i = 1, 2, 3$.*
- *$p \in C_i$ for $i = 1, 2, 3$.*

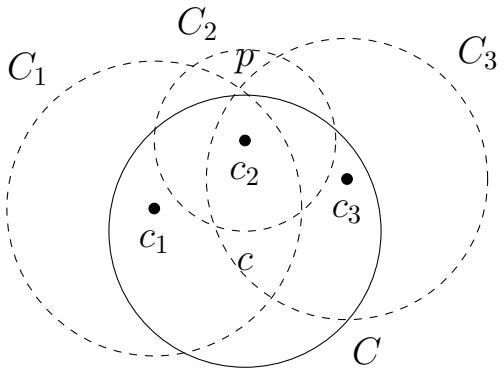


Figure 3.3: An example of lemma 3.3. The points c_1, c_2, and c_3 are the centers of the dashed disks C_1, C_2, and C_3. Each of these disks contains points c and p, and their centers lie inside the disk C. The point c is the center of disk C, and p lies on the boundary of C. The statement of lemma 3.3 is satisfied since c_2 is strictly contained in C_1. For this instance, c_2 is also contained in C_3.

Then, for some i and j in $\{1, 2, 3\}$ such that $i \neq j$, c_i is strictly inside C_j.

Proof. The disk C is defined as $C = \{q : d(c, q) \leq r\} \in \mathbb{R}^2$, where $r = d(c, p)$. Without loss of generality, assume C is rotated such that p is its north pole. The line that passes through c and p divides C in two halves, say the left and right halves.

Let c_1, c_2, and c_3 be the centers of three disks that contain c and p such that $c_i \in C$. By the pigeonhole principle, one half of C contains at least two of them. Without loss of generality, assume that c_1 and c_2 are in the left half of C. Let c_1' be the point on the border of C where C and the ray from c through c_1 intersect. Define c_2' analogously. Without loss of generality, assume that the angle $\angle(pcc_1)$ is at least as large as the angle $\angle(pcc_2)$. We will argue that c_2 is contained in C_1.

Consider the function $d(c_1, x)$, where x is a point on the border of C. This function has a unique minimum value at c_1' and a unique maximum value at the antipodal point (with respect to C). Between the minimum and maximum values, it is monotonically increasing. Thus, because p is contained in C_1, it follows that all points on the arc between c_1' and p are strictly contained in C_1, including c_2'. Finally, since c_2 can be expressed as a convex combination of c and c_2', both of which are inside C_1, it follows that c_2 is strictly inside C_1. □

Theorem 3.3. *The signed graph $Q_{3,3}$ does not have a valid distance drawing in $\mathbb{R}^3$.*

Proof. Assume, for the sake of contradiction that $Q_{3,3}$ has a valid distance drawing, D, in $\mathbb{R}^3$. Let $v_1, v_2, v_3 \in V$ be the vertices on one side and $w_1, w_2, w_3 \in V$ be the vertices on the other side. Edges of the form $\{v_i, w_j\}$ are positive, while edges of the form $\{v_i, v_j\}$ and $\{w_i, w_j\}$ are negative.

Since D is valid, $D(w_1), D(w_2)$ and $D(w_3)$ lie strictly inside all of the three spheres S_i, one for each $i \in \{1, 2, 3\}$, centered at $D(v_i)$ and with radius equal to $r_i^- = \min_{j \neq i} d(D(v_i), D(v_j))$. By construction, none of the S_i strictly contain $D(v_j)$ for $j \neq i$. Furthermore, $D(v_1), D(v_2)$ and $D(v_3)$ cannot be collinear, otherwise the interior of the intersection of the S_i is empty. Let H_v be the plane defined by $D(v_1), D(v_2)$ and $D(v_3)$.

By the pigeonhole principle, at least two $D(w_i)$ lie in the same side of H_v. Without loss of generality, assume that $D(w_1)$ and $D(w_2)$ are in the same closed half-space defined by H_v and that $D(w_1)$ is at least as far from H_v as $D(w_2)$. Let c and p, respectively, be the projections of $D(w_1)$ and $D(w_2)$ onto H_v. Let $C_i = S_i \cap H_v$. We will show that our construction of c, p, C_1, C_2, C_3 satisfies the conditions of lemma 3.3.

First, we will show that c and p are inside C_i for $i = 1, 2, 3$. The following two inequalities are from the properties of projections and from the property of valid distance drawings:

$$d(c, D(v_i)) \leq d(D(w_j), D(v_i)) < r_i^-.$$

The same argument applies for p in place of c.

Now, let S be the sphere with radius $r^+ = \max_i d(D(v_i), D(w_1))$ centered at $D(w_1)$. Because w_2 and w_1 are negative neighbors, $D(w_2)$ must lie outside S.

Consider the intersection $S \cap H_v$. By construction, $S \cap H_v$ contains all the $D(v_i)$ for $i = 1, 2, 3$. However, since $D(w_2)$ is at least as close to H_v as $D(w_1)$, p is not contained in $S \cap H_v$.

Let C be the disk centered at c which exactly passes through p. By the argument in the previous paragraph, C contains $S \cap H_v$ and thus contains $D(v_i)$ for $i = 1, 2, 3$.

Notice that we have satisfied all the conditions of lemma 3.3. Thus, one of the C_i must strictly contain the center of another C_j. This contradicts the validity of D, completing the proof. □

3.6 Computational Experiments

In this section, we will devise an algorithm, Valid Drawing, for finding valid distance drawings of a given signed graph. The inputs to Valid Drawing are a signed graph G and a dimension k. The output is either a valid distance drawing of G or $\emptyset$ if no valid distance drawing was found.

For small n, we can then upper bound $L(n)$ with a brute-force search over all signed graphs on n vertices. We only need to check the complete signed graphs on n vertices. The number of such graphs equals the number of distinct graphs on n vertices, up to isomorphism (Online Encyclopedia of Integer Sequences, sequence A000088 [57]).

For a given signed graph $G = (V, (E^+, E^-))$, consider the problem of finding a valid distance drawing $D : V \to \mathbb{R}^k$. Due to the non-convex nature of the constraint (3.1), it is difficult to formulate this problem as an optimization problem with convex constraints. Instead, we formulate the problem as an unconstrained optimization problem with a non-convex objective function.

Consider the following optimization problem:

$$\min_{\boldsymbol{x}_{v_i}} \sum_{v_i \in V} \max\left(\max_{v_j \in N^+(v_i)} d(\boldsymbol{x}_{v_i}, \boldsymbol{x}_{v_j}) - \min_{v_j \in N^-(v_i)} d(\boldsymbol{x}_{v_i}, \boldsymbol{x}_{v_j}) + 1, 0 \right), \tag{3.2}$$

where $\boldsymbol{x}_{v_i} = D(v_i)$ for $v_i \in V$. The term $\max_{v_j \in N^+(v_i)} d(\boldsymbol{x}_{v_i}, \boldsymbol{x}_{v_j})$ is the distance to the farthest positive neighbor of vertex v_j. The term $\min_{v_j \in N^-(v_i)} d(\boldsymbol{x}_{v_i}, \boldsymbol{x}_{v_j})$ is the distance to the closest negative neighbor of vertex v_i. The 1 inside the outer maximization may be confusing at first glance. Recall that constraint (3.1) is a strict inequality. The 1 exists to ensure that there is a slack of at least 1 in this constraint.

Lemma 3.4. *Given a signed graph G, the optimal objective value for optimization problem (3.2) is 0 if and only if G has a valid distance drawing in $\mathbb{R}^k$.*

Proof. If there exists a valid distance drawing of G, then there exists a scaling of the valid distance drawing with slack at least 1 on each constraint (3.1). This scaled drawing is a feasible solution to (3.2) where each term in the summation over V has a value of 0. On the other hand, if there is a feasible solution with value 0 then every constraint (3.1) has slack of at least 1. Ergo, the drawing is a valid distance drawing. □

For a given signed graph G, we attempt to find a valid distance drawing into $\mathbb{R}^k$ with the Valid Drawing algorithm (algorithm 3.1). The algorithm works by performing gradient descent on the objective function (3.2). First, the injection x_{v_i} of each vertex $i \in V$ is initialized as a k-dimension random vector, drawn from a uniform distribution on $[-\sqrt{3/(n \times k)}, \sqrt{3/(n \times k)}]^k$. At each iteration, we evaluate the objective function (3.2) to check if it is 0. If it is, then we have identified a valid distance drawing. Otherwise, the injection is updated by taking a step of gradient descent. This process is repeated until a valid distance drawing is identified or the maximum number of iterations, n_{iter}, is reached.

The specific variant of gradient descent that we used was Adam [39]. Adam is a popular gradient descent algorithm. Rather than maintaining a single learning rate, Adam calculates a separate learning rate for each parameter. Adam also combines the gradient with a "momentum" term, which is a weighted average of the gradients calculated in previous steps. We set the parameters for Adam to their default values. In our experiments, we set $n_{\text{iter}} = 1000$. Our code is available at

`https://github.com/marvel2010/signed-graph-valid-drawing`.

Unfortunately, our approach has limitations. Our objective function (3.2) is not convex, so Adam is not guaranteed to converge to an optimal solution. Thus, if we do not find a feasible solution with objective value 0 within the allotted iterations, we cannot know if no such feasible solution exists or if we were unable to find it. In some applications of gradient descent, when the improvement in the objective function is sufficiently small it is assumed a local minimum has been found and, to save time, the gradient descent is terminated. In our

Algorithm 3.1 VALID DRAWING

Input: $G = (V, (E^+, E^-))$: the signed graph we wish to inject.
Input: k: the dimension of Euclidean space to inject the signed graph.
Input: n_{iter}: the maximum number of iterations.
Output: $(x_{v_1}, x_{v_2}, \ldots, x_{v_n})$: a valid distance drawing or $\emptyset$ if none found.

$n \leftarrow |V|$
$x_{v_i} \leftarrow$ Random vector selected uniformly in $\left[-\sqrt{\frac{3}{n \times k}}, \sqrt{\frac{3}{n \times k}}\right]^k \quad \forall v_i \in V$
$x \leftarrow (x_{v_1}, x_{v_2}, \ldots, x_{v_n})$
$f(x) \leftarrow \sum_{v_i \in V} \max\left(\max_{v_j \in N^+(v_i)} d(x_{v_i}, x_{v_j}) - \min_{v_j \in N^-(v_i)} d(x_{v_i}, x_{v_j}) + 1, 0\right)$
for $t = 1, \ldots, n_{\text{iter}}$ **do**
    if $f(x) = 0$ **then**
        return x ▷ Valid distance drawing found.
    else
        $x \leftarrow$ ADAM(x) ▷ Update the drawing with a step of gradient descent.
    end if
end for
return $\emptyset$ ▷ No valid distance drawing found.

Table 3.1: Known bounds on $L(n)$

Number of vertices	1	2	3	4	5	6	7	8
Upper bound	0	1	1	2	3	4	4	5
Lower bound	0	1	1	2	3	4	4	4

case, we only care whether the optimal objective value is 0, so there is no benefit to stopping early when the objective value is greater than 0.

Using VALID DRAWING, we were able to find a valid distance drawing in $\mathbb{R}^4$ for every signed graph with 7 vertices. Combining this with the result that $L(6) = 4$ from the previous section, we conclude that $L(7) = 4$.

We also identified a valid distance drawing for almost all complete signed graphs with 8 vertices in $\mathbb{R}^4$. The only three signed graphs for which we did not find a valid distance drawing in $\mathbb{R}^4$ were the signed K_8 with positive subgraphs $K_{3,5}$, $K_{4,4}$, or $K_{4,4}$ with one edge removed. We were unable to draw these three graphs in $\mathbb{R}^4$ even after running VALID DRAWING hundreds of times and setting $n_{\text{iter}} = 100000$. All signed graphs on 8 vertices have a valid distance drawing in $\mathbb{R}^5$, showing that $L(8) \leq 5$.

Table 3.2: Upper bounds on the smallest dimension such that $Q_{\frac{n}{2},\frac{n}{2}}$ has a valid distance drawing.

Number of vertices n	2	4	6	8	10	12	14	16	18	20	22	24
Upper bound for $L(K_{\frac{n}{2},\frac{n}{2}})$	1	2	4	5	7	8	10	11	13	14	16	17

3.7 Final Remarks

Our results are summarized in table 3.1. Determining an exact formula for $L(n)$ in general remains an open problem. Based on our computational experiments, and the bounds on $L(n)$ for $n \leq 7$, we state the following conjecture:

Conjecture 3.1. *$L(n)$ is the smallest dimension such that $Q_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ (the signed K_n with positive subgraph $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$) has a valid distance drawing.*

In other words, we conjecture that $Q_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ is the most difficult graph to inject.

Motivated by our own conjecture, we used Valid Drawing to draw $Q_{\frac{n}{2},\frac{n}{2}}$ for even values of n from 8 to 24. Our results are presented in table 3.2. All the values we obtained for the upper bounds fit the formula $\lfloor \frac{(3n-1)}{4} \rfloor$, leading to our final conjecture about the asymptotic behavior of $L(n)$.

Conjecture 3.2. $L(n) \sim \frac{3}{4}n$.

Chapter 4

Maximum Online Perfect Bipartite Matching with i.i.d. Arrivals

4.1 Introduction

In this chapter, we consider the problem of *maximum online perfect bipartite matching*. Suppose that we have a set of jobs and a set of workers. At every time step, a single job arrives to be served by one of the workers. Upon a job's arrival, we observe the utility of assigning that job to each of the workers. We must immediately decide which worker will serve the job. Once a worker is assigned a job, the worker is unavailable and cannot be assigned to another job. Jobs continue to arrive until no workers are available.

It is natural to model this problem setup as a bipartite graph, where there is an edge between each worker and job. The weight of the edge between a job and a worker equals the non-negative utility of assigning that worker to that job. Once a number of jobs arrives equal to the number of workers, the assignment of workers to jobs will form a perfect matching in this bipartite graph. Our goal is to design a dispatching algorithm that maximizes the expected sum of utilities of the perfect matching.

In this work, we consider the maximum online perfect bipartite matching problem with *independent and identically distributed (i.i.d.) arrivals*. This means that, at each time step, a job is drawn i.i.d. from a known distribution over job types.

We introduce two classes of algorithms for the problem of online perfect bipartite matching with i.i.d. arrivals, which we call flow-guided algorithms and evaluation-guided algorithms. Several of the algorithms are $\frac{1}{2}$-competitive: the total expected utility of the perfect matching produced by the algorithm is at least half of the total expected utility of an optimal offline algorithm that knows the job arrival sequence in advance. We also describe a family of problem instances for which $\frac{1}{2}$ is the best-possible competitive ratio. In contrast, the same problem with adversarial job arrivals cannot be bounded, as observed by Feldman et al. [22].

Related Work

Our work resides in the space of online matching problems. We review several variants of online matching, including the maximum online bipartite matching problem (not necessarily perfect) and the minimum online perfect bipartite matching problem. We also review the closely-related k-server problem. For each of these problems, several arrival models are considered. Arrival models including adversarial, where the adversary chooses jobs and their arrival order; random order, where the adversary chooses jobs but not their arrival order; and i.i.d., where the adversary specifies a probability distribution over job types and each arrival is sampled independently from the distribution. We briefly describe each of these problems and present best-known results, contrasting it to the setting considered here. A summary is provided in table 4.1.

Maximum Online Bipartite Matching

The maximum online bipartite matching problem (not necessarily perfect) is defined on a bipartite graph with n known workers and n jobs that arrive one at a time. Jobs either get assigned to a worker or are discarded. The goal is to maximize the cardinality (or sum of weights) of the resulting matching. In contrast to our problem, jobs may be the discarded and the resulting matching may be *imperfect*.

For the unweighted problem with adversarial arrivals, Karp, Vazirani, and Vazirani [34] showed a best-possible algorithm that achieves a competitive ratio of $1 - \frac{1}{e} \approx 0.632$. Variations of the problem have been proposed: addition of edge or vertex weights, the use of budgets, different arrival models, etc. Mehta et al. [48] provide an excellent overview of this literature. When the arrivals are in a random order, it is possible to do better than $1 - \frac{1}{e}$. Mahdian and Yan [42], in 2011, achieved a competitive ratio of 0.696. Manshadi, Gharan, and Saberi [46] showed that you cannot do better than 0.823. If the problem also has weights, then the best-possible competitive ratio is 0.368 by a reduction from the secretary problem as shown by Kesselheim et al. [37]. They also give an algorithm that attains this competitive ratio.

The problem has also been studied when the jobs are drawn i.i.d. from a known distribution. This problem is also referred to as *online stochastic matching*. The first result to break the $1 - \frac{1}{e}$ barrier for the unweighted case was the 0.67-competitive algorithm of Feldman et al. [23] in 2009. To date, the best-known competitive ratio of 0.730 is due to Brubach et al. [8]. This is close the best-known bound of 0.745 by Correa et al. [17].

Minimum Online Perfect Bipartite Matching

The minimum online perfect bipartite matching addresses the question of finding a minimum cost perfect matching on a bipartite graph with n workers and n jobs. Given any arbitrary sequence of jobs arriving one by one, each job needs to be irrevocably assigned to worker on arrival. This problem is the minimization version of the problem considered in this work. However, the obtained competitive ratios do not transfer.

The problem was first considered by Khuller, Mitchell, and Vazirani [38] and independently by Kalyanasundaram and Pruhs [32]. If the weights are arbitrary, then the competitive ratio cannot be bounded. To address this, both papers considered the restriction where the edge weights are distances in some metric on the set of vertices. They give a $2n-1$ competitive algorithm, which is the best-possible for deterministic algorithms. When randomized algorithms are allowed, the best-known competitive ratio is $O(\log^2(n))$ by Bansal et al. [4].

It is possible to attain better competitive ratios when the arrival model is restrict. If the arrival order is also randomized, then Raghvendra [52] shows that a competitive ratio of $2\log(n)$ is attainable. He also shows that this is the best possible. If the arrivals are drawn i.i.d. from a known distribution, then Gupta et al. [27] present an algorithm with competitive ratio $O((\log\log\log n)^2)$. In fact, the setting studied by Gupta et al. is exactly the minimization version of the setting we study in this chapter.

k-Server Problem

In the k-server problem, k workers are distributed at initial positions in a metric space. Jobs are elements of the same metric space and arrive one at a time. When a job arrives, it must be assigned to a worker which moves to the job's location. The goal in the k-server problem is to minimize the total distance traveled by all workers to serve the sequence of jobs. After an assignment, the worker remains available for assignment to new jobs. This *reassignment* distinguishes the k-server problem from ours, where workers are fixed to a job once assigned.

The k-server problem was introduced by Manasse, McGeoch, and Sleator [44]. A review of the k-server problem literature was written by Koutsoupias [40]. For randomized algorithms in discrete metrics, the competitive ratio $O(\log^2(k)\log(n))$ was attained by Bubeck et al. [9], where n is the number of points in the discrete metric space. On the other hand, $\Omega(\log(k))$ is a known lower bound. In the i.i.d. setting, Dehghani et al. [20] consider a different kind of competitive ratio: they give an online algorithm with a cost no worse than $O(\log(n))$ times the cost of the optimal *online* algorithm.

Structure of this Chapter

This chapter is organized as follows. Section 4.2 formally introduces the problem of maximum online perfect bipartite matching with i.i.d. arrivals and defines the concept of competitive ratio. Section 4.3 introduces a family of instances for which no online algorithm performs better than $\frac{1}{2}$ in terms of competitive ratio. In section 4.4 we introduce the class of flow-guided algorithms and in section 4.5 we introduce the class of evaluation-guided algorithms. In section 4.6, we present a comparison of all the algorithms introduced in this chapter, combining theoretical and empirical results. We conclude in section 4.7.

Table 4.1: Best-known competitive ratios and impossibility bounds for various online bipartite matching problems. ★: Results presented in this chapter.

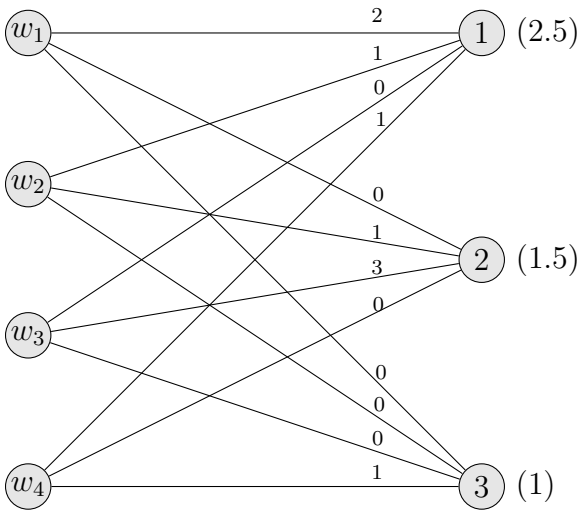
Sense	Discarding	Arrivals	Restrictions	Best Known	Best Possible
Max	Allowed	Advers.	0/1	0.632 [34]	0.632 [34]
Max	Allowed	Rand. Ord.	0/1	0.696 [42]	0.823 [46]
Max	Allowed	Rand. Ord.	None	0.368 [37]	0.368 [37]
Max	Allowed	i.i.d.	None	0.730 [8]	0.745 [17]
Min	No	Advers.	Metric	$O(\log^2(n))$ [4]	$\Omega(\log(n))$ [49]
Min	No	Rand. Ord.	Metric	$2\log(n)$ [52]	$2\log(n)$ [52]
Min	No	i.i.d.	Metric	$O((\log\log\log n)^2)$ [27]	$\Omega(1)$
Max	No	Adversarial	None	0	0 [22]
Max	**No**	**i.i.d.**	**None**	$\frac{1}{2}^{\bigstar}$	$\frac{1}{2}^{\bigstar}$

4.2 Preliminaries

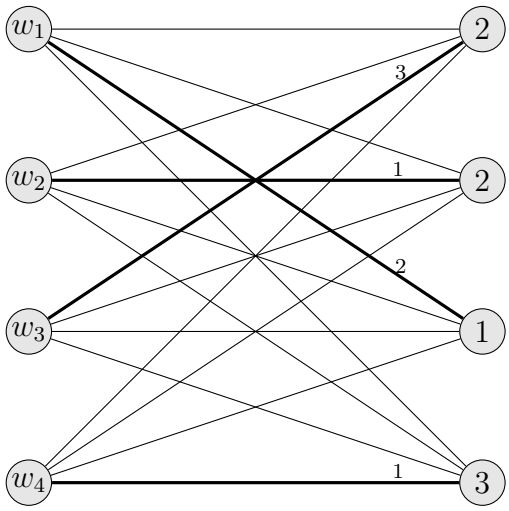
The set of workers is denoted by W, with size $|W| = n$. The set J denotes the set of job types, with size $|J| = k$. For every worker $w \in W$ and job type $j \in J$ there is a utility of $u_{wj} \geq 0$ for assigning a job of type j to worker w. Let $\mathcal{D}(J)$ be a known probability distribution over the job types.

At every time step, $t = 1, \ldots, n$, a single job is drawn i.i.d. from J according to $\mathcal{D}$. Let p_j denote the probability that job type j arrives, so np_j is the expected number of jobs of type j that arrive. The job must be irrevocably assigned to a worker before the next job arrives. Workers are no longer available after they have been assigned a job. After n steps, each worker is assigned to exactly one job and the resulting assignment forms a perfect matching. Our goal is to design a algorithm such that the expected sum of the utilities of the resulting perfect matching is as high as possible.

Throughout this work, we will repeatedly refer two bipartite graphs; the *expectation graph* G and the *realization graph* $\widehat{G}$. See figure 4.1 for an example. The expectation graph G is a complete bipartite graph defined over the set of workers W and the set of job types J. Every pair $\{w, j\}$ has associated utility $u_{wj} \geq 0$. $U = \{u_{wj} \forall w \in W, j \in J\}$ is the set of edges of the expectation graph. Each job type is assigned a weight equal to $p_j n$, the expected number of arrivals of that job type over n time steps. The realization graph $\widehat{G}$ is the random bipartite graph obtained after all n jobs have arrived. On one side of $\widehat{G}$ is W. On the other side is $\widehat{J}$, the set of n jobs that arrived. We use $\hat{j}_t \in \widehat{J}$ to denote the job that arrives at time t and $j_t \in J$ to denote the job type of job $\hat{j}_t$. $\widehat{G}$ is a complete bipartite graph defined over W and $\widehat{J}$. Every edge $\{w, \hat{j}\}$ has utility u_{wj}, where j is the job type of job $\hat{j}$. It is important to remember that the expectation graph G is deterministic and known in advance, whereas the realization graph $\widehat{G}$ is a random graph representing a realization of the job arrival process



(a) An expectation graph with 4 workers and 3 job types. On one side, the workers. On the other side, the job types. In this example, $p_1 = \frac{1}{2}$ and $p_3 = \frac{1}{5}$.



(b) A realization graph in which 2 workers of type 2 where sampled, then 1 worker of type 1, then 1 worker of type 3. The bold edges represent a greedy assignment of jobs to workers.

Figure 4.1: An expectation graph and realization graph on an instance where $|W| = 4$ and $|J| = 3$.

and is revealed over time.

An instance of the maximum online perfect bipartite matching problem with i.i.d. arrivals is defined by the set of workers W, the job types J, a set of utilities U, and a distribution over the job types $\mathcal{D}(J)$. Equivalently, the expectation graph G and the distribution $\mathcal{D}(J)$ defines an instance of this problem.

Given an algorithm $\mathcal{ALG}$ (potentially randomized) that returns a perfect matching $\hat{M}$ on $\widehat{G}$, we would like to assess the expected performance of the algorithm. We do this in two steps: first we take the expectation over the internal randomness of the algorithm and second we take the expectation over samples of the realization graph $\widehat{G}$ from the expectation graph G. Throughout this chapter, we will always use curly letters to refer to algorithms (for example, $\mathcal{ALG}$) and block letters to refer to their expected performance (for example, ALG).

The performance of an algorithm $\mathcal{ALG}$ for a single realization $\widehat{G}$, the bipartite graph over W and $\widehat{J}$, is given by:

$$\text{ALG}(W, \widehat{J}) = \mathbb{E}_{\mathcal{ALG}}\left[\sum_{w \in W, \hat{j} \in \widehat{J}} u_{w\hat{j}} I_{w\hat{j}}\right],$$

where $I_{w\hat{j}}$ is a random indicator variable that equals 1 if $\mathcal{ALG}$ assigned the job $\hat{j}$ to worker w (that is, $\{w, \hat{j}\} \in \widehat{M}$) and equals 0 otherwise. The expectation is taken over the random choices that the algorithm makes, if any.

For a given problem instance defined by expectation graph G and distribution $\mathcal{D}(J)$,

$$\mathrm{ALG}(W, J) = \mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\mathrm{ALG}(W, \widehat{J})\right]$$

measures the algorithm's expected performance over samples of $\widehat{J}$ from $\mathcal{D}(J)$.

Remark 4.1. *Technically, the performance of an algorithm $\mathcal{ALG}$ should always be written in terms of four parameters: W, the set of workers; J, the set of job types; U the set of utilities between workers and job types (equivalently, edges in the bipartite expectation graph); and $\mathcal{D}(J)$, the distribution over job types. However, writing $\mathcal{ALG}(W, J, U, \mathcal{D}(J))$ and $\mathrm{ALG}(W, J, U, \mathcal{D}(J))$ is cumbersome. For that reason, throughout this chapter we write $\mathcal{ALG}(W, J)$ when referring to an algorithm and $\mathrm{ALG}(W, J)$ when referring to its expected performance, assuming that we are also given U and $\mathcal{D}(J)$.*

As an example, consider the algorithm $\mathcal{RAND}(W, J)$. When a job arrives, $\mathcal{RAND}(W, J)$ selects a worker uniformly at random and assigns the job that arrived to that worker. We calculate its expected performance:

$$\begin{aligned}
\mathrm{RAND}(W, J) &= \mathbb{E}_{\mathcal{RAND}}\left[\mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\sum_{w\in W, \hat{j}\in\widehat{J}} u_{w\hat{j}} I_{w\hat{j}}\right]\right] \\
&= \mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\sum_{w\in W, \hat{j}\in\widehat{J}} u_{w\hat{j}} \mathbb{E}_{\mathcal{RAND}}\left[I_{w\hat{j}}\right]\right] \\
&= \mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\sum_{w\in W, \hat{j}\in\widehat{J}} u_{w\hat{j}} \frac{1}{n}\right] \\
&= \sum_{w\in W} \mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\sum_{\hat{j}\in\widehat{J}} u_{w\hat{j}} \frac{1}{n}\right] \\
&= \sum_{w\in W, j\in J} p_j n u_{wj} \frac{1}{n} \\
&= \sum_{w\in W, j\in J} u_{wj} p_j.
\end{aligned}$$

The worst-case performance across instances is measured by the *competitive ratio*. Let $\mathcal{OPT}\text{-}\mathcal{OFF}(W, \widehat{J})$ be the maximum weight perfect matching in the realization graph $\widehat{G}$ and

let OPT-OFF$(W, \widehat{J})$ be its weight. Consider

$$\text{OPT-OFF}(W, J) = \mathbb{E}_{\widehat{J} \sim \mathcal{D}(J)} \left[\text{OPT-OFF}(W, \widehat{J}) \right].$$

OPT-OFF(W, J) measures the expected performance of an optimal offline algorithm, which is given full information about the arrival sequence in advance. The ratio

$$\frac{\text{ALG}(W, J)}{\text{OPT-OFF}(W, J)}$$

measures the excepted performance of $\mathcal{ALG}$ relative to the optimal offline algorithm for a given instance of the problem. The competitive ratio is the worst-case, i.e. lowest, ratio among all possible instances of the expectation graph $G = (W \cup J, U)$ and distributions $\mathcal{D}(J)$:

Definition 4.1 (Competitive Ratio)**.** *An algorithm, $\mathcal{ALG}$, is said to have a competitive ratio of α when α is the infimum, over all instances $(W, J, U, \mathcal{D}(J))$, of $\frac{\text{ALG}(W,J,U,\mathcal{D}(J))}{\text{OPT-OFF}(W,J,U,\mathcal{D}(J))}$. Using our shorthand for referring to instances (see remark 4.1),*

$$\alpha = \inf_{W,J,U,\mathcal{D}(J)} \frac{\text{ALG}(W, J)}{\text{OPT-OFF}(W, J)}.$$

4.3 Best-Possible Competitive Ratio

We present here a family of instances for which no online algorithm attains a competitive ratio more than $\frac{1}{2}$.

Theorem 4.1. *For the online perfect bipartite matching problem with i.i.d. arrivals, no online algorithm can achieve a competitive ratio better than $\frac{1}{2}$.*

Proof. Consider an expectation graph G with $|J| = k = n + 1$. Let the job types be indexed from 0 to n and the workers from 1 to n. Job types 1 to n each arrive with probability $\frac{1}{n^2}$ and job type 0 arrives with probability $\frac{n-1}{n}$. For this graph, we set $u_{wj} = 1$ if $w = j$ and to 0 otherwise. See figure 4.2 for an illustration.

The value of OPT-OFF$(W, \widehat{J})$ equals the number of unique job type in $\{1, \ldots, n\}$ that arrives. The probability that a given job type *never* arrives is

$$(1 - \frac{1}{n^2})^n.$$

Thus, the probability that at least one job of that job type arrives is

$$1 - (1 - \frac{1}{n^2})^n.$$

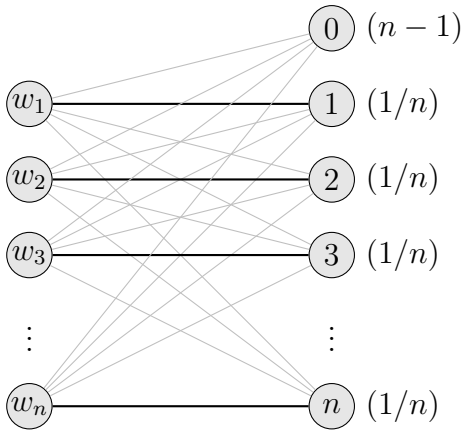


Figure 4.2: Expectation graph G used in the proof of theorem 4.1. Edges in black have a utility of 1 and edges in gray have a utility of 0.

By linearity of expectation,

$$\text{OPT-OFF}(W, J) = \mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\text{OPT-OFF}(W, \widehat{J})\right] = n\left(1 - \left(1 - \frac{1}{n^2}\right)^n\right).$$

For any online algorithm $\mathcal{ALG}$, $t-1$ workers are no longer available at time step t regardless of the strategy. Thus, with probability $\frac{n-1}{n} + \frac{1}{n}\frac{t-1}{n}$ the utility attained at time step t is 0. With probability $\frac{1}{n}\frac{n-(t-1)}{n}$, it is 1. The total expected utility obtained by $\mathcal{ALG}$ is:

$$\mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\text{ALG}(W, \widehat{J})\right] = \frac{1}{n}\left(\frac{n}{n} + \frac{n-1}{n} + \frac{n-2}{n} + \cdots + \frac{1}{n}\right) = \frac{1}{2}\frac{1}{n}(n+1)$$

We compute the relevant ratio

$$\frac{\text{ALG}(W, J)}{\text{OPT-OFF}(W, J)} = \frac{\mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\text{ALG}(W, \widehat{J})\right]}{\mathbb{E}_{\widehat{J}\sim\mathcal{D}(J)}\left[\text{OPT-OFF}(W, \widehat{J})\right]} = \frac{\frac{1}{2}\frac{1}{n}(n+1)}{n\left(1 - \left(1 - \frac{1}{n^2}\right)^n\right)}$$

We take the limit as n goes to infinity:

$$\lim_{n\to\infty} \frac{\frac{1}{2}(n+1)}{n^2\left(1 - \left(1 - \frac{1}{n^2}\right)^n\right)} = \frac{1}{2}.$$

□

4.4 Flow-Guided Algorithms

In this section, we will introduce the general class of *flow-guided* algorithms. The idea is that, at each time step, the assignment of worker is guided by a feasible solution to an offline

transportation problem between the remaining available workers and J. When a job of type j arrives, the probability of selecting worker w is proportional to the flow from w to j.

First, some additional notation. Let the random variable W_t^A represent the worker assigned for the job arriving at time t. Furthermore, let the random set AW_t consist of the available workers when the job at time t arrives. Note that $|AW_t| = n - (t-1)$.

We will show two important results. First, we will define an update rule which we call *uniform redistribution*. If we use the uniform redistribution update rule, then the expected performance of a flow-guided algorithm can be written in a succinct closed form and computed efficiently. We show that an algorithm previously proposed by Chang et al. [13], $\mathcal{DISPATCH}$, is optimal among all flow-guided algorithms that use the uniform redistribution update rule. Second, we argue that there is a singular algorithm, $\mathcal{OPT}$-$\mathcal{FLOW}$, which is the optimal flow-guided algorithm. In $\mathcal{OPT}$-$\mathcal{FLOW}$, we update the flow at each step to be the optimal solution to the offline transportation problem between AW_t and J.

Offline Transportation Problem

In expectation, $p_j n$ jobs of job type $j \in J$ will arrive in the realization graph $\hat{G}$. We will consider a transportation problem on the expectation graph G, where each job type has a demand of $p_j n$ and each worker has a supply of 1.

Formally, let $f_{wj} \geq 0$ be the flow from worker $w \in W$ to job type $j \in J$. We define the transportation problem $\mathcal{TPP}(W, J)$ and we let $\text{TPP}(W, J)$ denote the value of the optimal solution. See figure 4.3 for an example. As before, we use W and J as parameters, assuming that U and $\mathcal{D}(J)$ are implied (see remark 4.1):

$$\text{TPP}(W, J) = \max \sum_{w \in W} \sum_{j \in J} u_{wj} f_{wj}, \qquad (\mathcal{TPP}(W, J))$$

$$\sum_{w \in W} f_{wj} = p_j n \qquad \forall j \in J. \qquad (4.1)$$

$$\sum_{j \in J} f_{wj} = 1 \qquad \forall w \in W. \qquad (4.2)$$

Let $f^*_{wj}(W, J)$ be the value of the optimal flow from w to j in $\mathcal{TPP}(W, J)$. When the instance is clear from context, we omit the W and J and write f^*_{wj}.

We claim that $\text{OPT-OFF}(W, J) \leq \text{TPP}(W, J)$. The reason is that the weighted average of the perfect matchings $\mathcal{OPT}$-$\mathcal{OFF}(W, \widehat{J})$ forms a feasible solution to the transportation problem $\mathcal{TPP}(W, J)$.

Lemma 4.1. *For any instance* $(W, J, U, \mathcal{D}(J))$*,*

$$\text{OPT-OFF}(W, J) \leq \text{TPP}(W, J).$$

Proof. Assign each edge in $G = (W \cup J, U)$ an indicator variable I_{wj}, which takes on the value 1 if $\mathcal{OPT}$-$\mathcal{OFF}(W, J)$ assigns worker w to a job of type j in $\widehat{G}$ and 0 otherwise. We claim

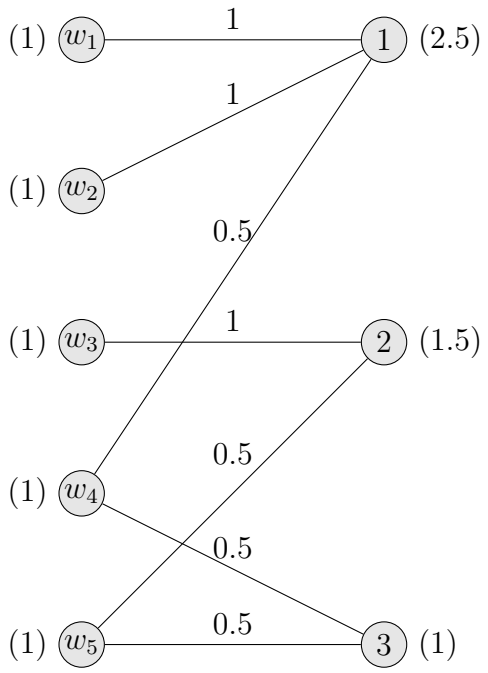


Figure 4.3: Example of a feasible solution to $\mathcal{TPP}(W, J)$ on an instance where $|W| = 5$, $|J| = 3$, and $(p_1, p_2, p_3) = (\frac{5}{10}, \frac{3}{10}, \frac{2}{10})$. The value of this feasible solution is $\text{TPP}(W, J) = u_{11} + u_{21} + 0.5u_{41} + u_{32} + 0.5u_{52} + 0.5u_{43} + 0.5u_{53}$.

that $\mathbb{E}[I_{wj}]$ forms a feasible solution to the transportation problem $\mathcal{TPP}(W, J)$. Indeed,

$$\sum_{w \in W} \mathbb{E}[I_{wj}] = \mathbb{E}\left[\sum_{w \in W} I_{wj}\right] = p_j n, \qquad \sum_{j \in J} \mathbb{E}[I_{wj}] = \mathbb{E}\left[\sum_{j \in J} I_{wj}\right] = 1.$$

Since $\mathbb{E}[I_{wj}]$ is feasible for $\mathcal{TPP}(W, J)$, it has objective at most $\text{TPP}(W, J)$:

$$\text{OPT-OFF}(W, J) = \mathbb{E}\left[\sum_{w \in W, j \in J} u_{wj} I_{wj}\right] = \sum_{w \in W, j \in J} u_{wj} \mathbb{E}[I_{wj}] \leq \text{TPP}(W, J).$$

□

General Template for Flow-Guided Algorithms

Algorithm 4.1 gives the generic outline for all flow-guided algorithms. The flow at step t, $f^t_{wj} \geq 0$, is a feasible solution to the offline transportation problem $\mathcal{TPP}(AW_t, J)$. Recall that $|AW_t| = n - (t-1)$, so constraint 4.1 becomes

$$\sum_{w \in AW_t} f^t_{wj} = p_j(n - (t-1)) \forall j \in J. \tag{4.3}$$

In algorithm 4.1, we see that in order to fully specify a flow-guided algorithm, we need to specify how it initializes the offline solution (line 3) and how it updates the offline solution (line 9).

Algorithm 4.1 Template for Flow-Guided Algorithms

Input: W, J, U, $\mathcal{D}(J)$: the set of workers, set of job types, set of utilities, and distribution over job types.
Output: $\hat{M}$: a perfect matching between W and $\widehat{J}$.
1: $\hat{M} \leftarrow \emptyset$.
2: $AW_1 \leftarrow W$.
3: **Initialize Offline Solution**: f^1_{wj} feasible for $\mathcal{TPP}(AW_1, J)$
4: **for** $t = 1, \ldots, n$ **do**
5: $\quad \hat{j}_t$ arrives (type: $j_t \in J$).
6: $\quad$ select assigned worker w^A_t proportional to $\frac{f^t_{wj_t}}{\sum_{w' \in AW_t} f^t_{w'j_t}}$
7: $\quad \hat{M} \leftarrow \hat{M} \cup [w^A_t, \hat{j}_t]$
8: $\quad AW_{t+1} \leftarrow AW_t \setminus \{w^A_t\}$
9: $\quad$ **Update Offline Solution**: f^{t+1}_{wj} feasible for $\mathcal{TPP}(AW_{t+1}, J)$
10: **end for**
11: **return** $\hat{M}$

All flow-guided algorithms have an important property in common. At each time step, the assigned worker, W^A_t, is selected uniformly at random from the set of available workers, AW_t. Thus, the order in which workers are assigned can be viewed as a random permutation of the workers, selected uniformly at random from all $n!$ possible permutations. We prove this in the following lemma.

Lemma 4.2. *In any flow-guided algorithm, at each time step t the assigned worker W^A_t is drawn uniformly from the available workers:*

$$\mathbb{P}\left(W^A_t = w | w \in AW_t\right) = \frac{1}{|AW_t|}.$$

Proof. The probability that worker w is the assigned worker at time t can be calculated by summing over all the job types and calculating the probability that job type is selected times the conditional probability that worker w is selected. See figure 4.4 for a helpful example.

$$\mathbb{P}\left(W^A_t = w | w \in AW_t\right) = \sum_{j \in J} \mathbb{P}\left(W^A_t = w | w \in AW_t, j_t = j\right) \mathbb{P}\left(j_t = j\right)$$

The probability that a job of type j is selected at step t is p_j. The conditional probability of selecting worker w is given by the flow (algorithm 4.1, line 6).

$$= \sum_{j \in J} \frac{f^t_{wj}}{\sum_{w' \in AW_t} f^t_{w'j}} p_j$$

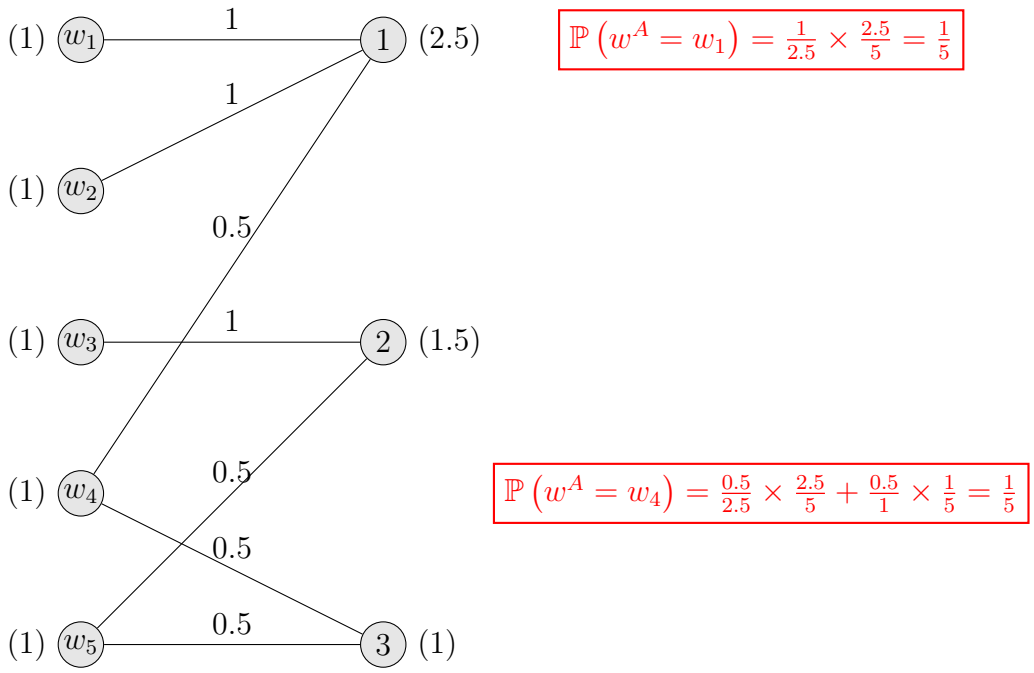


Figure 4.4: An illustration of lemma 4.2 when there are 5 workers available. There are two ways in which worker 4 can be selected. Either a job of type 1 arrives and then worker 4 is selected or a job of type 3 arrives and then worker 4 is selected.

By the stipulation that the flows must sum to $|AW_t|p_j$ at each job type (constraint 4.1),

$$= \sum_{j \in J} \frac{f^t_{wj}}{|AW_t|p_j} p_j$$
$$= \frac{1}{|AW_t|} \sum_{j \in J} f^t_{wj}$$

By the stipulation that the flows must sum to 1 at each worker (constraint 4.2),

$$= \frac{1}{|AW_t|}.$$

□

Updating by Uniform Redistribution

Up to this point, we have proved a property that all flow-guided algorithms have in common (lemma 4.2), regardless of their initialization rule (algorithm 4.1, line 3) and their update

rule (algorithm 4.1, line 9). In this subsection, we will consider only flow-guided algorithms that use a specific update rule, which we call the "uniform redistribution" update rule. We will show that flow-guided algorithms that use the uniform redistribution update rule have the extraordinary property that their expected performance can be easily calculated from the initial flow.

Let w_t^A be the worker assigned at time step t. We call the following rule for updating the feasible guiding flow *uniform redistribution*:

$$f_{wj}^{t+1} = \begin{cases} \frac{|AW_{t+1}|}{|AW_t|}(f_{wj}^t + \frac{1}{|AW_{t+1}|} f_{w_t^A j}^t) & w \in AW_{t+1} \\ 0 & w \notin AW_{t+1}. \end{cases} \tag{4.4}$$

In words, we take the flow which was assigned to the worker w_t^A and reassign it uniformly to the remaining $|AW_{t+1}|$ available workers. We then re-scale such that the flow satisfies the feasibility conditions. For an example, see figure 4.5.

We claim that the uniform redistribution updating rules is a valid updating rule, in the sense that, at each step t, the flow is a valid feasible solution to the offline transportation problem $\mathcal{TPP}(AW_t, J)$.

Lemma 4.3. *For the uniform redistribution updating rule, specified in equation 4.4, if the initial flow f_{wj}^1 is feasible for the transportation problem $\mathcal{TPP}(W, J)$, then the flow at time t, f_{wj}^t, is feasible for the transportation problem $\mathcal{TPP}(AW_t, J)$.*

Proof. We proceed by induction on t. The base case, $t = 1$, is assumed. Now, assume that the feasibility conditions are satisfied up to t. Then, for $t + 1$,

$$\begin{aligned} \sum_{j \in J} f_{wj}^{t+1} &= \frac{n-t}{n-(t-1)}(\sum_{j \in J} f_{wj}^t + \frac{1}{n-t} \sum_{j \in J} f_{w_t^A j}^t) \\ &= \frac{n-t}{n-(t-1)}(1 + \frac{1}{n-t}) \\ &= 1. \end{aligned}$$

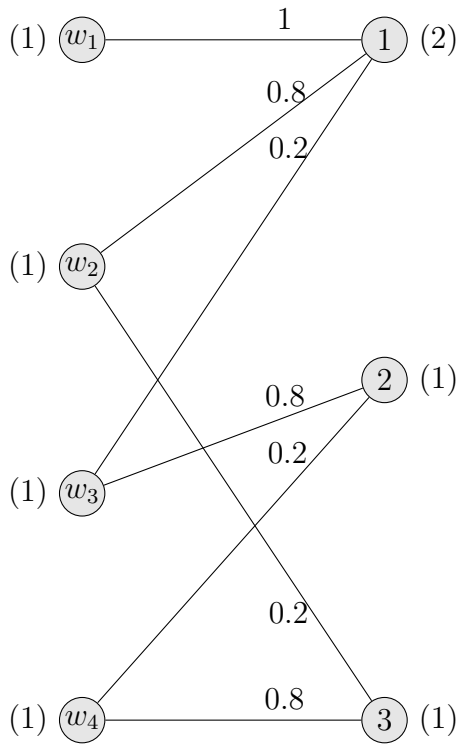
Thus, constraint 4.2 is satisfied. Likewise,

$$\sum_{w \in AW_{t+1}} f_{wj}^{t+1} = \frac{n-t}{n-(t-1)} \left(\sum_{w \in AW_{t+1}} f_{wj}^t + \frac{1}{n-t} \sum_{w \in AW_{t+1}} f_{w_t^A j}^t \right)$$

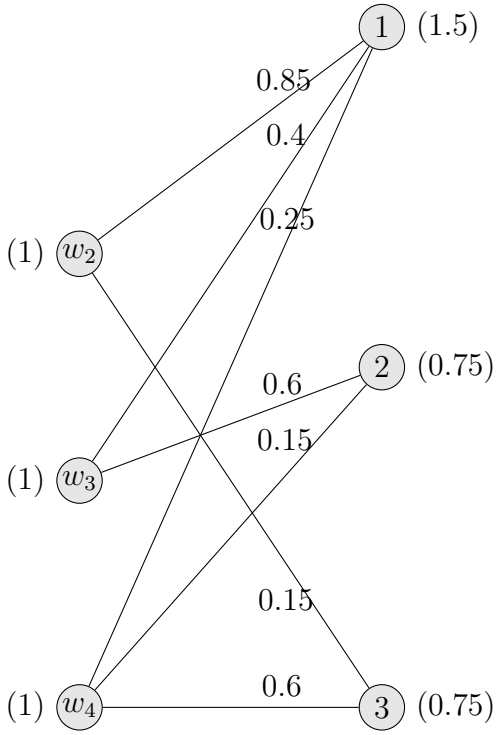
Since $AW_{t+1} = AW_t \setminus \{w_t^A\}$,

$$\begin{aligned} &= \frac{n-t}{n-(t-1)} \left(((n-(t-1))p_j - f_{w_t^A j}^t) + \frac{1}{n-t}(n-t) f_{w_t^A j}^t \right) \\ &= (n-t)p_j. \end{aligned}$$

Thus, constraint 4.3 is satisfied. This completes the induction. □



(a) The expectation graph for an instance with 4 workers and 3 job types. The arrival probabilities are $(p_1, p_2, p_3) = (\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$. The flows are an example of a feasible solution to the transportation problem $\mathcal{TPP}(W, J)$. In this example, we will assume that a job of type 1 arrives and worker 1 is assigned.



(b) After the assignment, we re-scale the flow by $\frac{3}{4}$ (since there were 4 workers and now there are 3) and then re-assign the flow that went from worker 1 to job type 1. The flow is reassigned to workers 2, 3, and 4. Notice that the node weights on the right have decreased, since there are fewer workers.

Figure 4.5: An example of the uniform redistribution update rule, before and after.

Now we state and prove an exact formula for the expected performance of any flow-guided algorithm with the uniform redistribution updating rule.

Theorem 4.2. *For the maximum online perfect bipartite matching problem with i.i.d. arrivals, the expected performance of any flow-guided algorithm $\mathcal{ALG}(W, J)$ that uses the uniform redistribution update rule (equation 4.4) is*

$$\mathrm{ALG}(W, J) = \frac{1}{2} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{1}{2} \sum_{w \in W} \sum_{j \in J} u_{wj} p_j. \tag{4.5}$$

Proof. We proceed by induction on the number of workers.

First, the base case. When $|W| = 1$, $f^1_{wj} = p_j$ is the only feasible solution to the transportation problem $\mathcal{TPP}(W, J)$. Let w be the lone worker. The expected performance of the flow-guided algorithm is $\sum_{j \in J} u_{wj} p_j$. This concludes the base case.

Assume the claim holds for $|W| = n-1$. We will show it for $|W| = n$. The expected utility in just the first time step equals exactly $\frac{1}{n} \sum_{w \in W, j \in J} u_{wj} f^1_{wj}$. In addition, each available worker has a $\frac{1}{n}$ chance of being selected as the assigned worker (lemma 4.2). Let $\overline{\mathrm{ALG}}(W, w, J)$ be the expected performance of $\mathcal{ALG}(W, J)$ from the point where the set of available workers is $W \setminus \{w\}$ onward. Thanks to the linearity of expectation, we can write the expected performance of $\mathcal{ALG}(W, J)$ as

$$\mathrm{ALG}(W, J) = \frac{1}{n} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{1}{n} \sum_{w \in W} \overline{\mathrm{ALG}}(W, w, J).$$

$\overline{\mathcal{ALG}}(W, w, J)$ is also a flow-guided algorithm that uses the uniform redistribution update rule and starts with one fewer worker than $\mathcal{ALG}(W, J)$. The initial flow in $\overline{\mathcal{ALG}}(W, w, J)$ can be expressed in terms of the initial flow in $\mathcal{ALG}(W, J)$ after one step of uniform redistribution. Assuming the inductive hypothesis, we can expand $\overline{\mathrm{ALG}}(W, w, J)$, substituting the formula for the uniformly redistributed flow:

$$\begin{aligned} &= \frac{1}{n} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} \\ &\quad + \frac{1}{n} \sum_{w \in W} \left[\frac{1}{2} \sum_{w' \in W \setminus \{w\}} \sum_{j \in J} \left(\frac{n-1}{n} f^1_{w'j} + \frac{1}{n} f^1_{wj} \right) u_{w'j} + \frac{1}{2} \sum_{w' \in W \setminus \{w\}} \sum_{j \in J} u_{w'j} p_j \right] \end{aligned}$$

Distributing the sum over W and reordering such that the sum over J is on the outside:

$$\begin{aligned} &= \frac{1}{n} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{1}{2n} \sum_{j \in J} \sum_{w \in W} \sum_{w' \in W \setminus \{w\}} \frac{n-1}{n} f^1_{w'j} u_{w'j} \\ &\quad + \frac{1}{2n} \sum_{j \in J} \sum_{w \in W} \sum_{w' \in W \setminus \{w\}} \frac{1}{n} f^1_{wj} u_{w'j} + \frac{1}{2n} \sum_{j \in J} \sum_{w \in W} \sum_{w' \in W \setminus \{w\}} u_{w'j} p_j \end{aligned}$$

When the term being summed does not depend on w, the sum over $w \in W$ then $w' \in W \setminus \{w\}$ is equivalent to $n-1$ times the sum over $w \in W$. When the term being summed does depend on w, we can swap the order of summation to sum over w first:

$$\begin{aligned} &= \frac{1}{n} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{(n-1)^2}{2n^2} \sum_{w \in W} \sum_{j \in J} f^1_{wj} u_{wj} \\ &\quad + \frac{1}{2n^2} \sum_{j \in J} \sum_{w \in W'} u_{w'j} \sum_{w \in W \setminus \{w'\}} f^1_{wj} + \frac{n-1}{2n} \sum_{j \in J} \sum_{w \in W} u_{wj} p_j \end{aligned}$$

Resolving the sum over $w \in W \setminus \{w'\}$:

$$= \frac{n^2+1}{2n^2} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{1}{2n^2} \sum_{j \in J} \sum_{w' \in W} u_{w'j}(p_j n - f^1_{w'j}) + \frac{n-1}{2n} \sum_{w \in W} \sum_{j \in J} u_{wj} p_j$$

Simplifying the expression,

$$= \frac{1}{2} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{1}{2} \sum_{w \in W} \sum_{j \in J} u_{wj} p_j.$$

This completes the inductive step. □

Earlier, we introduced the algorithm $\mathcal{RAND}$. When a job arrives, $\mathcal{RAND}$ assigns the job to a worker uniformly at random. $\mathcal{RAND}$ can be viewed at a flow-guided algorithm, where the flow from worker w to job type j is always p_j if worker w is available and 0 otherwise. To be precise,

$$f^t_{wj} = \begin{cases} p_j & w \in AW_t \\ 0 & w \notin AW_t. \end{cases}$$

This flow satisfies uniform redistribution (equation 4.4), so $\mathcal{RAND}$ fits the condition of theorem 4.2. This gives us an alternative means to calculate the expected performance of $\mathcal{RAND}(W, J)$, using equation 4.5:

$$\text{RAND}(W, J) = \frac{1}{2} \sum_{w \in W} \sum_{j \in J} u_{wj} p_j + \frac{1}{2} \sum_{w \in W} \sum_{j \in J} u_{wj} p_j = \sum_{w \in W} \sum_{j \in J} u_{wj} p_j.$$

Notice that the second term in equation 4.5 is exactly $\frac{1}{2}\,\text{RAND}(W, J)$, regardless of which initialization strategy is being used.

The $\mathcal{DISPATCH}$ Algorithm

The algorithm $\mathcal{DISPATCH}$, introduced by Chang et al. [13], is a special flow-guided algorithm in which the initial offline flow is $f^*_{wj}(W, J)$ and the uniform redistribution update rule is used. Using the closed form from theorem 4.2, we will show that $\mathcal{DISPATCH}$ is $\frac{1}{2}$-competitive and that it is the best algorithm among all flow-guided algorithms that use the uniform redistribution update rule.

From theorem 4.2 and lemma 4.1, the $\frac{1}{2}$-competitiveness of $\mathcal{DISPATCH}$ follows. For any instance $(W, J, U, \mathcal{D}(J))$,

$$\begin{aligned} \text{DISPATCH}(W, J) &= \frac{1}{2}\,\text{TPP}(W, J) + \frac{1}{2}\,\text{RAND}(W, J) \\ &\geq \frac{1}{2}\,\text{TPP}(W, J) \\ &\geq \frac{1}{2}\,\text{OPT-OFF}(W, J). \end{aligned}$$

Furthermore, since $\text{TPP}(W, J)$ is the largest possible value of $\sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj}$, it is clear that $\mathcal{DISPATCH}$ is the optimal algorithm among all flow-guided algorithms that use the uniform redistribution update rule.

From the closed form in theorem 4.2, we get a sense for which instances $\mathcal{DISPATCH}$ is expected to perform well on and which instances it is expected to perform poorly on. Rather than just calculating the competitive ratio of $\mathcal{DISPATCH}$ in a worst-case instance, we can compute its competitive ratio for *any* instance. In particular,

$$\frac{\text{DISPATCH}(W, J)}{\text{OPT-OFF}(W, J)} \geq \frac{\frac{1}{2}\,\text{TPP}(W, J) + \frac{1}{2}\,\text{RAND}(W, J)}{\text{TPP}(W, J)} = \frac{1}{2} + \frac{1}{2}\frac{\text{RAND}(W, J)}{\text{TPP}(W, J)} \tag{4.6}$$

In addition, thanks to our formula for $\text{DISPATCH}(W, J)$, we can express an interesting relationship between $\text{TPP}(W, J)$ and $\text{OPT-OFF}(W, J)$, in terms of $\text{RAND}(W, J)$. We know that the ratio $\frac{\text{OPT-OFF}(W,J)}{\text{TPP}(W,J)}$ is bounded above by 1. Now, we can lower bound it as well:

$$\begin{aligned} &\frac{1}{2}\,\text{TPP}(W, J) + \frac{1}{2}\,\text{RAND}(W, J) \leq \text{OPT-OFF}(W, J) \leq 1 \\ \implies \qquad &\frac{1}{2} + \frac{1}{2}\frac{\text{RAND}(W, J)}{\text{TPP}(W, J)} \leq \frac{\text{OPT-OFF}(W, J)}{\text{TPP}(W, J)} \leq 1. \end{aligned} \tag{4.7}$$

Both lower bounds (4.6 and 4.7) depend on the ratio $\frac{\text{RAND}(W,J)}{\text{TPP}(W,J)}$. In general, $\frac{\text{RAND}(W,J)}{\text{TPP}(W,J)} \in [0, 1]$. In the worst case instance we presented earlier (section 4.3), $\frac{\text{RAND}(W,J)}{\text{TPP}(W,J)} \to 0$ as $n \to \infty$. On the other hand, consider an instance where all the $u_{wj} = 1$. On such an instance, $\text{RAND}(W, J) = \text{TPP}(W, J)$. From equation 4.6, we see that, as $\frac{\text{RAND}(W,J)}{\text{TPP}(W,J)} \to 1$, the bound on the competitive ratio of $\mathcal{DISPATCH}(W, J)$ grows stronger.

An Optimal Flow-Guided Algorithm

In this section, we will introduce an algorithm and prove that it has the best expected value among all flow-guided algorithms. For this reason, we will call the algorithm $\mathcal{OPT\text{-}FLOW}$.

In $\mathcal{OPT\text{-}FLOW}(W, J)$, we start by initializing the offline solution to be the optimal solution to the transportation problem $\mathcal{TPP}(W, J)$. We update the offline solution at every time step to match the optimal solution to the transportation problem on the remaining workers:

$$f^t_{wj} = f^*_{wj}(AW_t, J).$$

Theorem 4.3. *Let $\mathcal{ALG}$ be any flow-guided algorithm. For any instance $(W, J, U, \mathcal{D}(J))$,*

$$\text{ALG}(W, J) \leq \text{OPT-FLOW}(W, J).$$

Proof. We proceed by induction on the size of W.

If $|W| = 1$, all flow algorithms are the same as $\mathcal{OPT\text{-}FLOW}$, so the claim holds at equality.

Assume the claim holds for $|W| = n-1$. Let $\overline{\text{ALG}}(W, w, J)$ be the expected performance of $\mathcal{ALG}(W, J)$ from the point where the set of available workers is $W \setminus \{w\}$ onward. Recall (from the same construction in the proof of theorem 4.2) that we can write the expected performance of $\mathcal{ALG}(W, J)$ as

$$\text{ALG}(W, J) = \frac{1}{n} \sum_{w \in W} \sum_{j \in J} u_{wj} f^1_{wj} + \frac{1}{n} \sum_{w \in W} \overline{\text{ALG}}(W, w, J).$$

Since f^1 is feasible,

$$\leq \frac{1}{n} \text{TPP}(W, J) + \frac{1}{n} \sum_{w \in W} \overline{\text{ALG}}(W, w, J).$$

$\overline{\mathcal{ALG}}(W, w, J)$ is a flow-guided algorithm. Thus, by the inductive hypothesis,

$$\leq \frac{1}{n} \text{TPP}(W, J) + \frac{1}{n} \sum_{w \in W} \text{OPT-FLOW}(W \setminus \{w\}, J).$$

This is exactly the formula for the expected performance of $\mathcal{OPT}\text{-}\mathcal{FLOW}(W, J)$:

$$= \text{OPT-FLOW}(W, J).$$

This completes the inductive step. □

In fact, we can take this analysis a step further and write out an explicit formula for OPT-FLOW(W, J). We break down the calculation of the expected performance of $\mathcal{OPT}\text{-}\mathcal{FLOW}(W, J)$ as follows. For each time step, we calculate the probability that the set of available workers at time t is AW. Second, we calculate the expected utility gained at time step t conditional on the set of available workers being AW. Finally, we sum over all time steps t.

$$\text{OPT-FLOW}(W, J) = \sum_{t=1}^{n} \sum_{\substack{AW \subseteq W \\ |AW| = n-(t-1)}} \mathbb{E}\left[u_{W_t^A j_t} | AW_t = AW\right] \mathbb{P}\left(AW_t = AW\right)$$

Substituting the expected utility at time t, given that AW is the set of available workers,

$$= \sum_{t=1}^{n} \sum_{\substack{AW \subseteq W \\ |AW| = n-(t-1)}} \frac{\text{TPP}(AW, J)}{|AW|} \mathbb{P}\left(AW_t = AW\right)$$

Recall that the order in which workers are assigned is a random permutation of the set of workers (lemma 4.2), so each of the $\binom{|W|}{|AW|}$ set of size $|AW|$ are equally likely for AW_t.

$$= \sum_{t=1}^{n} \sum_{\substack{AW \subseteq W \\ |AW| = n-(t-1)}} \frac{\text{TPP}(AW, J)}{|AW|} \frac{1}{\binom{|W|}{|AW|}}$$

Rewriting the sum,

$$= \sum_{AW \subseteq W} \frac{1}{\binom{|W|}{|AW|}} \frac{\text{TPP}(AW, J)}{|AW|}.$$

This formula requires summing over all subsets of W, so it takes exponential time to compute.

4.5 Evaluation-Guided Algorithms

In this section, we introduce a class of algorithms where, when a job j arrives, the worker that it is assigned to is selected deterministically (as a function of AW_t, J, and j). To motivate this class of algorithms, we first describe how our problem can be modeled as a Markov Decision Process and an optimal online algorithm computed in exponential time.

Best-Possible Online Algorithm

The problem of maximum online bipartite matching with i.i.d. arrivals can be modeled as a Markov Decision Process (MDP) with an exponential number of states, which can be described as a linear program of exponential size [51]. We will detail the MDP solution and its dynamic programming solution, here.

In the MDP, there are $2^n - 1$ states, each one corresponding to a different non-empty subset of W. The state, $AW \subseteq W$, tells us the subset of workers that remain available. See figure 4.6 for an illustration. Let OPT-ON(AW, J) denote the value of the optimal MDP policy from this state.

The value of the optimal MDP policy can be computed with dynamic programming. Assume we are in the state where the set of remaining workers is AW. If a job of type j arrives, then we would like to assign it to a worker $w \in AW$ so that we maximize

$$u_{wj} + \text{OPT-ON}(AW \setminus \{w\}, J).$$

Combining this with the probability that a job of type j arrives, we have the recursive formula

$$\text{OPT-ON}(AW, J) = \sum_{j \in J} p_j \max_{w \in AW} \{u_{wj} + \text{OPT-ON}(AW \setminus \{w\}, J)\}.$$

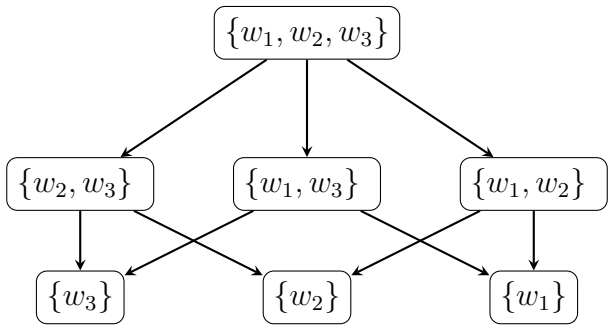


Figure 4.6: Illustration of the states in the Markov Decision Process.

The dynamic programming solution first considers all subsets AW where $|AW| = 1$, then where $|AW| = 2$, etc. At each state, it takes the sum over $|J| = k$ terms, so its total running time is at least $\Omega(k2^n)$. This running time is prohibitively large for anything but very small instances.

General Template for Evaluation-Guided Algorithms

Algorithm 4.2 gives a template for a generic *evaluation-guided* algorithm. The crux is that, at each step, after a job type j arrives, a worker w is selected so as to maximize the current utility, u_{wj}, plus the expected utility from the state $AW \setminus \{w\}$. Since the exact value of the state $AW \setminus \{w\}$ cannot be calculated efficiently, a proxy is used. This comes in the form of EFUNC, an evaluation function. For a given job type arriving, the choice of which worker to assign is deterministic.

Algorithm 4.2 Template for Evaluation-Guided Algorithms

Input: W, J, U, $\mathcal{D}(J)$: the set of workers, set of job types, set of utilities, and distribution over job types.
Input: An evaluation function $\text{EFUNC} : \mathcal{P}(W) \times \{J\} \to \mathbb{R}$.
Output: $\hat{M}$: a perfect matching between W and $\hat{J}$.

$\hat{M} \leftarrow \emptyset$.
$AW_1 \leftarrow W$.
for $t = 1, \ldots, n$ **do**
    $\hat{j}_t$ arrives (type: $j_t \in J$).
    $w_t^A = \text{argmax}_{w \in AW_t} \{u_{wj_t} + \text{EFUNC}(AW_t \setminus \{w\}, J)\}$.
    $\hat{M} \leftarrow \hat{M} \cup [w_t^A, \hat{j}_t]$.
    $AW_{t+1} \leftarrow AW_t \setminus \{w^A\}$.
end for
return $\hat{M}$

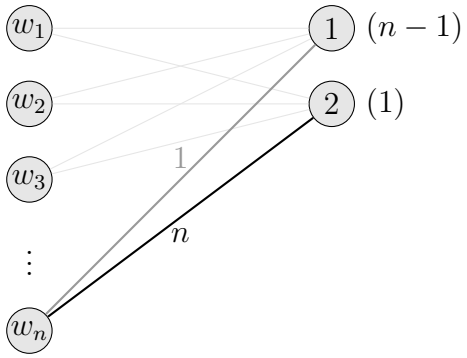


Figure 4.7: Instances in which the competitive ratio of a greedy algorithm tends to zero as the number of workers tends to infinity.

Note that the class of evaluation-guided algorithms is *not* a subset of the class of flow-guided algorithms. In flow-guided algorithms, there is a uniform probability of choosing each worker at each time step. In evaluation-guided algorithms, this is not necessarily the case.

When we set the evaluation function, EFUNC(AW, J), to be OPT-ON(AW, J), then the resulting evaluation-guided algorithm is the optimal online algorithm. However, we are more interested in considering cases where the evaluation function can be computed efficiently.

Greedy

The purely greedy algorithm, $\mathcal{GREEDY}$, is an evaluation-guided algorithm in which the evaluation function is uniformly zero. That is, EFUNC$(AW, J) = 0$ for all $AW \subseteq W$.

$\mathcal{GREEDY}$ has arbitrarily bad competitive ratio. Consider the instance illustrated in figure 4.7. There are n workers and 2 job types. Job type 1 arrives with probability $\frac{n-1}{n}$. Job type 2 arrives with probability $\frac{1}{n}$. The utility of assigning either job to worker 1 through $n-1$ is 0. The utility of assigning job 1 to worker n is 1. The utility of assigning job 2 to worker n is n. Thus, $\mathcal{GREEDY}(W, J)$ always assigns the first job to arrive to worker n. On the other hand $\mathcal{OPT}\text{-}\mathcal{ON}(W, J)$ attains utility 1 if only jobs of type 1 arrive and utility n if any jobs of type 2 arrive. We can calculate

$$\begin{aligned}\frac{\text{GREEDY}(W, J)}{\text{OPT-OFF}(W, J)} &= \frac{p(j_1 = 1) \times 1 + p(j_1 = 2) \times n}{p(j_1 = \ldots = j_n = 1) \times 1 + (1 - p(j_1 = \ldots = j_n = 1)) \times n} \\ &= \frac{\frac{n-1}{n} \times 1 + \frac{1}{n} \times n}{(\frac{n-1}{n})^n \times 1 + (1 - (\frac{n-1}{n})^n) \times n}.\end{aligned}$$

As $n \to \infty$, this ratio goes to 0.

Even though it has arbitrarily bad competitive ratio, the greedy algorithm will still perform better than a purely random algorithm, in expectation, on all instances. The intuition

is as follows: even though the greedy algorithm is deterministic *given the arrivals*, the fact that the arrival order is inherently random introduces some randomness into the behavior of the greedy algorithm.

Theorem 4.4. *For any instance* $(W, J, U, \mathcal{D}(J))$*,*

$$\text{GREEDY}(W, J) \geq \text{RAND}(W, J)$$

Proof. For the sake of this proof, it is helpful to re-frame the random arrival process. Instead of sampling a random arrival $\hat{j}_t$ from $\mathcal{D}(J)$ at each time step, imagine that we first sample a set of n jobs from $\mathcal{D}(J)$ and *then* decide their arrival order. Because of the i.i.d. nature of the problem, this two-step random processes is equivalent to our arrival model. Let $\{\hat{j}_{a_1}, \hat{j}_{a_2}, \ldots, \hat{j}_{a_n}\}$ be the set of n jobs selected in the first step. The second step determines the values of $(a_1, \ldots, a_n)$, which is some permutation of $(1, \ldots, n)$.

Consider any some fixed $\hat{j}_\alpha$, where its type, j_α, has been decided but its arrival position, α, has not. Without loss of generality, assume that we order the workers by their utility for being matched to $\hat{j}_\alpha$. That is,

$$u_{1\hat{j}_\alpha} \geq u_{2\hat{j}_\alpha} \geq \ldots \geq u_{n\hat{j}_\alpha}.$$

If $\hat{j}_\alpha$ is the first to arrive ($\alpha = 1$), then it is necessarily assigned to worker 1. This occurs with probability $\frac{1}{n}$. If $\hat{j}_\alpha$ is the second to arrive ($\alpha = 2$), then it is necessarily assigned to either worker 1 or worker 2. This also occurs with probability $\frac{1}{n}$. Combining these first two events, we see that $\hat{j}_\alpha$ is assigned to worker 1 or 2 with probability at least $\frac{2}{n}$. This logic continues. The probability that $\hat{j}_\alpha$ is assigned to worker z or better (meaning worker i, where $i \leq z$) can be bounded below by the probability that $\alpha \leq z$, which is exactly $\frac{z}{n}$. Let $I^G_{i\hat{j}_\alpha}$ be the indicator random variable for the greedy algorithm assigning worker i to job $\hat{j}_\alpha$. Then,

$$\sum_{i=1}^{z} \mathbb{P}\left(I^G_{i\hat{j}_\alpha} = 1\right) \geq \mathbb{P}\left(\alpha \leq z\right) = \frac{z}{n}.$$

Let $I^R_{i\hat{j}_\alpha}$ be a similar indicator random variable for the random algorithm. For the random algorithm, the arrival order is irrelevant. For all i,

$$\mathbb{P}\left(I^R_{i\hat{j}_\alpha} = 1\right) = \frac{1}{n}.$$

We will consider two random variables. The first is

$$U^G_\alpha = \sum_{i=1}^{n} u_{i\hat{j}_\alpha} I^G_{i\hat{j}_\alpha},$$

which is the expected value of the assignment $\hat{j}_\alpha$ in the greedy algorithm. The second is

$$U^R_\alpha = \sum_{i=1}^{n} u_{i\hat{j}_\alpha} I^R_{i\hat{j}_\alpha},$$

which is the expected value of the assignment of $\hat{j}_\alpha$ in the random algorithm. Both U_α^G and U_α^R are random variables which depend on α.

Recall that the $u_{i\hat{j}_\alpha}$ were assumed to be ordered (without loss of generality). The random variable U_α^G has first-order stochastic dominance over the random variable U_α^R, because

$$\begin{aligned}\mathbb{P}\left(U_\alpha^G \geq u_{z\hat{j}_\alpha}\right) &= \sum_{i=1}^{z} \mathbb{P}\left(I_{i\hat{j}_\alpha}^G = 1\right) \\ &\geq \frac{z}{n} \\ &= \sum_{i=1}^{z} \mathbb{P}\left(I_{i\hat{j}_\alpha}^R = 1\right) \\ &= \mathbb{P}\left(U_\alpha^R \geq u_{z\hat{j}_\alpha}\right).\end{aligned}$$

From this stochastic dominance, we can conclude that the expected value of U_α^G is necessarily greater than or equal to the expected value of U_α^R:

$$\mathbb{E}\left[U_\alpha^G\right] \geq \mathbb{E}\left[U_\alpha^R\right].$$

There was nothing special about how we chose $\hat{j}_\alpha$. So, by the linearity of expectation, we can write:

$$\text{GREEDY}(W, J) = \sum_{\alpha \in \{a_1, \ldots, a_n\}} \mathbb{E}\left[U_\alpha^G\right] \geq \sum_{\alpha \in \{a_1, \ldots, a_n\}} \mathbb{E}\left[U_\alpha^R\right] = \text{RAND}(W, J).$$

□

Derandomizing $\mathcal{DISPATCH}$

Recall from theorem 4.2 that DISPATCH(AW, J) can be computed efficiently for any set of available workers AW. This gives rise to an interesting prospect. What if, instead of running $\mathcal{DISPATCH}$, we used DISPATCH as an evaluation function? The resulting algorithm, $\mathcal{EVAL}$-$\mathcal{DISPATCH}(W, J)$, is an evaluation-guided algorithm. Like any evaluation-guided algorithm, when a job arrives, a worker is assigned deterministically. Thus, $\mathcal{EVAL}$-$\mathcal{DISPATCH}$ can be viewed as a derandomized version of $\mathcal{DISPATCH}$. In $\mathcal{EVAL}$-$\mathcal{DISPATCH}(W, J)$, when a job of type j arrives, the worker assigned, w, maximizes

$$u_{wj} + \text{DISPATCH}(AW \setminus \{w\}, J).$$

In the next theorem, we prove that $\mathcal{EVAL}$-$\mathcal{DISPATCH}(W, J)$ will perform at least as well as $\mathcal{DISPATCH}(W, J)$, in expectation, on every instance.

Theorem 4.5. *For any instance* $(W, J, U, \mathcal{D}(J))$*,*

$$\text{DISPATCH}(W, J) \leq \text{EVAL-DISPATCH}(W, J).$$

Proof. We proceed by induction on the size of the set W. If $|W| = 1$, then both algorithms have only one choice of which worker to assign, so the claim holds with equality. This concludes the base case.

Now, assume that the claim holds for $|W| = n - 1$. For our inductive step, we will prove the claim for $|W| = n$. As in previous proofs, we let $\overline{\text{DISPATCH}}(W, w, J)$ denote the expected performance of $\mathcal{DISPATCH}(W, J)$ from the point where the set of available workers is $W \setminus \{w\}$ onward. Starting from W, assume that a job of type j arrives. Let π_w be the probability that $\mathcal{DISPATCH}(W, J)$ would assign worker w.

$$\text{DISPATCH}(W, J) = \sum_{w \in W} \pi_w (u_{wj} + \overline{\text{DISPATCH}}(W, w, J)) \tag{4.8}$$

Continuing to run $\mathcal{DISPATCH}(W, J)$ from the state $W \setminus \{w\}$ is no better than running $\mathcal{DISPATCH}(W \setminus \{w\}, J)$ (restarting from the state $W \setminus \{w\}$). Thus,

$$\leq \sum_{w \in W} \pi_w (u_{wj} + \text{DISPATCH}(W \setminus \{w\}, J)). \tag{4.9}$$

Because the π_w sum to 1,

$$\leq \max_{w \in W} (u_{wj} + \text{DISPATCH}(W \setminus \{w\}, J)).$$

Let w^* be the maximizer:

$$= u_{w^* j} + \text{DISPATCH}(W \setminus \{w^*\}, J).$$

From our inductive hypothesis,

$$\leq u_{w^* j} + \text{EVAL-DISPATCH}(W \setminus \{w^*\}, J).$$

This is exactly the formula for the expected performance of $\mathcal{EVAL}\text{-}\mathcal{DISPATCH}(W, J)$

$$= \text{EVAL-DISPATCH}(W, J).$$

Thus, induction is complete. □

Derandomizing Other Algorithms

In our proof of theorem 4.5, the only place where we made use of a property of $\mathcal{DISPATCH}$ was in going from equation 4.8 to equation 4.9. In particular, $\mathcal{DISPATCH}(W, J)$ has the property that continuing to run $\mathcal{DISPATCH}(W, J)$ from the state $W \setminus \{w\}$ onward is no

better than restarting $\mathcal{DISPATCH}$ from the state $W \setminus \{w\}$. To be precise, for any instance $(W, J, U, \mathcal{D}(J)$ and any $w \in W$,

$$\overline{\text{DISPATCH}}(W, w, J) \leq \text{DISPATCH}(W \setminus \{w\}, J).$$

This property is not unique to $\mathcal{DISPATCH}$. For example, it holds for $\mathcal{RAND}$ and $\mathcal{GREEDY}$. For any instance $(W, J, U, \mathcal{D}(J))$ and any $w \in W$,

$$\overline{\text{RAND}}(W, w, J) = \text{RAND}(W \setminus \{w\}, J)$$

and

$$\overline{\text{GREEDY}}(W, w, J) = \text{GREEDY}(W \setminus \{w\}, J).$$

Let $\mathcal{ALG}$ be an arbitrary algorithm for the online perfect bipartite matching problem and let $\mathcal{EVAL}$-$\mathcal{ALG}$ be the evaluation-guided algorithm which uses ALG as the evaluation function. That is, when a job j arrives, $\mathcal{EVAL}$-$\mathcal{ALG}$ always takes the worker w maximizing

$$u_{wj} + \text{ALG}(AW \setminus \{w\}, J).$$

We say that $\mathcal{EVAL}$-$\mathcal{ALG}$ is the *derandomization* of $\mathcal{ALG}$. We offer the following generalization of theorem 4.5, the proof of which is analogous:

Theorem 4.6. *Let $\mathcal{ALG}$ be an algorithm such that, for any instance $(W, J, U, \mathcal{D}(J))$ and any $w \in W$,*

$$\overline{\text{ALG}}(W, w, J) \leq \text{ALG}(W \setminus \{w\}, J). \tag{4.10}$$

Then, for any instance $(W, J, U, \mathcal{D}(J))$,

$$\text{ALG}(W, J) \leq \text{EVAL-ALG}(W, J).$$

Proof. Consider the proof of theorem 4.5, but replace every occurrence of $\mathcal{DISPATCH}$ with $\mathcal{ALG}$ and every occurrence of DISPATCH with ALG. The proof almost works as written, except when going from equation 4.8 to equation 4.9. In this theorem, we assume that $\overline{\text{ALG}}(W, w, J) \leq \text{ALG}(W \setminus \{w\}, J)$ as a condition. Thus,

$$\sum_{w \in W} \pi_w(u_{wj} + \overline{\text{ALG}}(W, w, J)) \leq \sum_{w \in W} \pi_w(u_{wj} + \text{ALG}(W \setminus \{w\}, J)).$$

The rest of the proof works as written (with the appropriate substitutions). □

The condition of theorem 4.6 holds for *any* memoryless algorithm. An algorithm is *memoryless* if its action from any state does not depend on the previous states it was in or the previous actions that it took. If an algorithm is memoryless, then equation 4.10 holds at equality. That said, even if an algorithm satisfies equation 4.10, it may not be useful to derandomize it, because it may not be possible to compute $\text{ALG}(AW \setminus \{w\}, J)$ efficiently. For example, while one could technically derandomize $\mathcal{OPT}$-$\mathcal{FLOW}$, it would require computing OPT-FLOW$(AW \setminus \{w\}, J)$ at each step, which takes an exponential amount of time and is therefore impractical.

In the following table, we consider three properties of the four algorithms we have presented so far:

	Memoryless	Satisfies Equation 4.10	ALG(W, J) Computable Efficiently
$\mathcal{RAND}$	✓	✓	✓
$\mathcal{DISPATCH}$		✓	✓
$\mathcal{OPT\text{-}FLOW}$	✓	✓	
$\mathcal{GREEDY}$	✓	✓	

For us to consider derandomizing $\mathcal{ALG}$, the latter two criteria must hold: equation 4.10 is satisfied (so that theorem 4.6 applies) and ALG(W, J) can be easily computed (so that running $\mathcal{EVAL}$-$\mathcal{ALG}$ is practical). From the table, we see that both $\mathcal{RAND}$ and $\mathcal{DISPATCH}$ satisfy these criteria, so we consider both $\mathcal{EVAL}$-$\mathcal{RAND}$ and $\mathcal{EVAL}$-$\mathcal{DISPATCH}$ to be practical algorithms.

Finally, recall that DISPATCH$(W, J) = \frac{1}{2}\,\text{RAND}(W, J) + \frac{1}{2}\,\text{TPP}(W, J)$ (theorem 4.2). Thus far, we have established that the evaluation-guided algorithms $\mathcal{EVAL}$-$\mathcal{RAND}$ and $\mathcal{EVAL}$-$\mathcal{DISPATCH}$ can be run efficiently and perform at least as well as $\mathcal{RAND}$ and $\mathcal{DISPATCH}$, in expectation, respectively. For the sake of completeness, we will also consider the evaluation-guided algorithm $\mathcal{EVAL}$-$\mathcal{TPP}$, in which TPP is used as the evaluation function. Unlike the evaluation functions RAND and DISPATCH, TPP is not derived from the expected performance of an algorithm, so theorem 4.6 does not apply. In the next section, we will see that the relative expected performance of $\mathcal{EVAL}$-$\mathcal{RAND}$, $\mathcal{EVAL}$-$\mathcal{DISPATCH}$, and $\mathcal{EVAL}$-$\mathcal{TPP}$ is not well-behaved. For each of the three proposed algorithms, there exist instances where it outperforms the other two, in expectation.

4.6 Empirical Study

In this chapter, we introduced seven algorithms that can be run efficiently (in the sense that they run in polynomial time for any expectation graph G and realization graph $\widehat{G}$), including three flow-guided algorithms and four evaluation-guided algorithms. The three flow-guided algorithms were $\mathcal{RAND}$, $\mathcal{DISPATCH}$, and $\mathcal{OPT\text{-}FLOW}$. The four evaluation-guided algorithms were $\mathcal{GREEDY}$, $\mathcal{EVAL}$-$\mathcal{RAND}$, $\mathcal{EVAL}$-$\mathcal{DISPATCH}$, and $\mathcal{EVAL}$-$\mathcal{TPP}$.

We have establish that certain algorithms perform at least as well as others, in expectation, on all instances. For example, for any instance, OPT-FLOW$(W, J) \geq$ DISPATCH(W, J). In this section, our goal is to experimentally discover which pairs of algorithms can not be simply compared. That is, for two algorithms, $\mathcal{ALG}_1$ and $\mathcal{ALG}_2$, we wish to find instances W, J and W', J', where $\text{ALG}_1(W, J) > \text{ALG}_2(W, J)$, but $\text{ALG}_2(W', J') > \text{ALG}_1(W', J')$. If we find such instances, then we know that we cannot make a simple comparison between $\mathcal{ALG}_1$ and $\mathcal{ALG}_2$ that holds on all instances.

All of the aforementioned algorithm, except $\mathcal{DISPATCH}$, can be viewed as memoryless policies for the Markov Decision Process. Given an expectation graph, we can compute the exact expected performance of each algorithm using dynamic programming. This computation takes an exponential amount of time, so it is practical only for a small number

	RAND	DISPATCH	OPTFLOW	GREEDY	EVALRAND	EVALDISPATCH	EVALTPP
RAND	=	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\{\leq_{\text{empir}}\}$
DISPATCH		=	$\leq$	<>	$\{\leq_{\text{empir}}\}$	$\leq$	$\{\leq_{\text{empir}}\}$
OPTFLOW			=	<>	<>	$\{\leq_{\text{empir}}\}$	<>
GREEDY				=	<>	<>	<>
EVALRAND					=	<>	<>
EVALDISPATCH						=	<>
EVALTPP							=

Table 4.2: Comparison of all the algorithms presented for the maximum online perfect bipartite matching problem with i.i.d. arrivals.

of workers. For $\mathcal{DISPATCH}$, we have an explicit formula (theorem 4.2). We tested 1000 randomly-generated instances with at most 10 workers and 10 job types and utilities selected uniformly at random in $[0, 1]$.

Table 4.2 presents all our results in one place, including theoretical results and practical ones. For the comparison between two algorithms, $\mathcal{ALG}_1$ and $\mathcal{ALG}_2$. There are three possible results:

- The notation $\mathcal{ALG}_1 \leq \mathcal{ALG}_2$ means that $\mathcal{ALG}_2$ provably performs at least as well as $\mathcal{ALG}_1$, in expectation, on all instances. These results follow from the preceding theoretical work in this chapter.
- The notation $\mathcal{ALG}_1 <> \mathcal{ALG}_2$ means that there exist instances where $\mathcal{ALG}_1$ outperforms $\mathcal{ALG}_2$ in expectation and there exist instances where $\mathcal{ALG}_2$ outperforms $\mathcal{ALG}_1$ in expectation. In appendix B, we present a set of four "representative" instances out of the 1000 which are sufficient to prove all the "<>" relations.
- The notation $\mathcal{ALG}_1\{\leq_{\text{empir}}\}\,\mathcal{ALG}_2$ means that that $\mathcal{ALG}_2$ performed at least as well as $\mathcal{ALG}_1$ on all 1000 instances in our empirical study, but we have not presented a rigorous proof that $\mathcal{ALG}_2$ will always outperform $\mathcal{ALG}_1$ in expectation on all instances. Thus, we cannot distinguish between $\mathcal{ALG}_1 \leq \mathcal{ALG}_2$ and $\mathcal{ALG}_1 <> \mathcal{ALG}_2$. However, the empirical evidence points to $\mathcal{ALG}_1 \leq \mathcal{ALG}_2$.

One major conclusion from our study was that, among the four evaluation-guided algorithms, every pair had the "<>" relation. This is particularly notable because we proved that $\text{RAND}(W, J) \leq \text{GREEDY}(W, J)$ for all instances (theorem 4.4). However,

$$\mathcal{GREEDY} <> \mathcal{EVAL\text{-}RAND}\,.$$

Another major conclusion was that, among $\mathcal{OPT\text{-}FLOW}$ and three of the four evaluation-guided algorithms ($\mathcal{GREEDY}$, $\mathcal{EVAL\text{-}RAND}$, and $\mathcal{EVAL\text{-}TPP}$), every pair had the "<>"

relation. Although we proved that DISPATCH$(W, J) \leq$ OPT-FLOW(W, J) for all instances (theorem 4.3), in our empirical study we found that

$$\mathcal{OPT\text{-}FLOW}\{\leq_{\text{empir}}\}\,\mathcal{EVAL\text{-}DISPATCH}\,.$$

This demonstrates the effectiveness of our derandomization technique.

4.7 Conclusion

In this chapter, we studied the maximum online perfect bipartite matching problem with i.i.d. arrivals. We introduced two classes of algorithms for the problems: the class of flow-guided algorithms and the class of evaluation-guided algorithms. A total of seven specific algorithms were studied, including the three flow-guided algorithms $\mathcal{RAND}$, $\mathcal{DISPATCH}$, and $\mathcal{OPT\text{-}FLOW}$, as well as the four evaluation-guided algorithms $\mathcal{GREEDY}$, $\mathcal{EVAL\text{-}RAND}$, $\mathcal{EVAL\text{-}DISPATCH}$, and $\mathcal{EVAL\text{-}TPP}$.

We showed that $\mathcal{DISPATCH}$, $\mathcal{EVAL\text{-}DISPATCH}$, and $\mathcal{OPT\text{-}FLOW}$ are 0.5-competitive and that 0.5 is the best possible competitive ratio. We also gave an instance-specific lower bound on the competitive ratio, which can provide a better-than-0.5 lower bound in non-worst-case instances. On the other hand, $\mathcal{RAND}$ and $\mathcal{GREEDY}$ have arbitrarily bad competitive ratios.

The seven algorithms can be partially ordered. In our theoretical work, we showed that, for some pairs of the algorithms, we can guarantee that one will always perform at least as well as the other, in expectation, on all instances of the problem: $\mathcal{GREEDY}$ will always perform at least as well as $\mathcal{RAND}$ in expectation, $\mathcal{OPT\text{-}FLOW}$ will always perform at least as well as any other flow-guided algorithm in expectation, and $\mathcal{EVAL\text{-}DISPATCH}$ and $\mathcal{EVAL\text{-}RAND}$ will always perform at least as well as $\mathcal{DISPATCH}$ and $\mathcal{RAND}$, in expectation, respectively.

Finally, we showed that several of the algorithms could not be ordered. In particular, for each pair of the four evaluation-guided algorithms, $\mathcal{ALG}_1$ and $\mathcal{ALG}_2$, there exists instances where $\mathcal{ALG}_1$ outperforms $\mathcal{ALG}_2$ in expectation and instances where $\mathcal{ALG}_2$ outperforms $\mathcal{ALG}_1$ in expectation.

Chapter 5

Conclusion

In this dissertation, we surveyed three NP-hard combinatorial optimization problems phrased on graphs. The problems were unrelated. The unifying theme was our approach. For each problem, we proposed a new, tailored algorithm for it. Our analysis of each algorithm furthered our understanding of a theoretical property of the corresponding problem, such as asymptotic complexity or competitive ratio.

In chapter 2, we considered the minimum k-`terminal cut` problem. We presented a practical branch-and-bound algorithm for this NP-hard problem. Our branch-and-bound algorithm used on a novel integer relaxation of the integer programming formulation known as the CKR formulation. Our relaxation could be solved with minimum isolating cuts. Thus, we called our algorithm Isolating Cut Branch-and-Bound. In an empirical experiment, we saw that Isolating Cut Branch-and-Bound was an order of magnitude faster than linear-programming-based branch-and-bound with Gurobi on twenty-four real-world instances. We proved that the complexity of Isolating Cut Branch-and-Bound is fixed-parameter tractable with respect to the size of the optimal solution. We also showed that, in any $(k-1)$-stable instance of k-`terminal cut`, the source sets of the minimum isolating cuts are the source sets of the unique optimal solution of that k-`terminal cut` instance. Thus, $(k-1)$-stable instances of k-`terminal cut` can be solved in the time it takes to compute $k-1$ minimum (s,t)-cuts.

In chapter 3, we considered the problem of finding valid distance drawings of signed graphs in $\mathbb{R}^k$. We addressed the question of finding $L(n)$, the smallest dimension such that every signed graph with n nodes has a valid embedding in $\mathbb{R}^{L(n)}$. In general, we showed that $\lfloor \log_5(n-3) \rfloor + 1 \leq L(n) \leq n-2$. We phrased the embedding problem as an optimization problem, introducing the algorithm Valid Drawing. Using our algorithm, we computed $L(n)$ for n up to 7. We also calculated the embedding of $K_{\frac{n}{2},\frac{n}{2}}$ for even n up to $n = 24$. This led us to conjecture that $L(n) \sim \frac{3}{4}n$.

In chapter 4, we considered the maximum online perfect bipartite matching problem with i.i.d. arrivals. We introduced two classes of algorithms for the problems: the class of flow-guided algorithms and the class of evaluation-guided algorithms. A total of seven specific algorithms were introduced, including the three flow-guided algorithms ($\mathcal{RANDOM}$,

$\mathcal{DISPATCH}$, and $\mathcal{OPT\text{-}FLOW}$), as well as four evaluation-guided algorithms ($\mathcal{GREEDY}$, $\mathcal{EVAL\text{-}RAND}$, $\mathcal{EVAL\text{-}DISPATCH}$, and $\mathcal{EVAL\text{-}RAND}$). We give an exact formula for the performance of $\mathcal{DISPATCH}$ on all instances and proved that it is the optimal algorithm among all flow-guided algorithms which use the uniform redistribution update rule. We also proved that $\mathcal{OPT\text{-}FLOW}$ is the optimal algorithm among all flow-guided algorithms, regardless of which update rule they use. In terms of competitive ratio, we proved that $\mathcal{DISPATCH}$, $\mathcal{EVAL\text{-}DISPATCH}$, and $\mathcal{OPT\text{-}FLOW}$ are 0.5-competitive and that 0.5 is the best possible competitive ratio. On the other hand, $\mathcal{RAND}$ and $\mathcal{GREEDY}$ have arbitrarily bad competitive ratios. The seven algorithms can be partially ordered: for some pairs of the algorithms, one is guaranteed to always perform at least as well as the other, in expectation, on all instances. For example, we showed that $\mathcal{GREEDY}$ will always perform at least as well as $\mathcal{RAND}$, in expectation, on all instances and $\mathcal{EVAL\text{-}DISPATCH}$ will always perform at least as well as $\mathcal{DISPATCH}$, in expectation, on all instances. On the other hand, in practical experiments, we showed that the evaluation-guided algorithms could not be ordered. For any two of the four evaluation-guided algorithms, we found at least one instance where the first was dominant and at least one instance where the second was dominant.

Bibliography

[1] Claudio Altafini. "Consensus problems on networks with antagonistic interactions". In: *IEEE Transactions on Automatic Control* 58.4 (2012), pp. 935–946.

[2] Haris Angelidakis, Konstantin Makarychev, and Yury Makarychev. "Algorithms for stable and perturbation-resilient problems". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. ACM. 2017, pp. 438–451.

[3] Haris Angelidakis, Yury Makarychev, and Pasin Manurangsi. "An Improved Integrality Gap for the Călinescu-Karloff-Rabani Relaxation for Multiway Cut". In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2017, pp. 39–50.

[4] Nikhil Bansal et al. "An $O(\log^2 k)$-competitive algorithm for metric bipartite matching". In: *European Symposium on Algorithms*. Springer. 2007, pp. 522–533. DOI: 10.1007/978-3-540-75520-3_47.

[5] Aharon Ben-Tal and Arkadi Nemirovski. "Robust solutions of linear programming problems contaminated with uncertain data". In: *Mathematical programming* 88.3 (2000), pp. 411–424.

[6] Yonatan Bilu and Nathan Linial. "Are stable instances easy?" In: *Combinatorics, Probability and Computing* 21.5 (2012), pp. 643–660.

[7] Yuri Boykov, Olga Veksler, and Ramin Zabih. "Markov random fields with efficient approximations". In: *Computer vision and pattern recognition, 1998. Proceedings. 1998 IEEE computer society conference on*. IEEE. 1998, pp. 648–655.

[8] Brian Brubach et al. "New Algorithms, Better Bounds, and a Novel Model for Online Stochastic Matching". In: *24th Annual European Symposium on Algorithms*. Vol. 57. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 24:1–24:16. DOI: 10.4230/LIPIcs.ESA.2016.24.

[9] Sebastien Bubeck et al. "K-Server via Multiscale Entropic Regularization". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. ACM. 2018, pp. 3–16. DOI: 10.1145/3188745.3188798.

[10] Niv Buchbinder, Joseph Seffi Naor, and Roy Schwartz. "Simplex partitioning via exponential clocks and the multiway cut problem". In: *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. ACM. 2013, pp. 535–544.

[11] Gruia Călinescu, Howard Karloff, and Yuval Rabani. "An improved approximation algorithm for multiway cut". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM. 1998, pp. 48–52.

[12] Cheng-Shang Chang et al. "Community detection in signed networks: An error-correcting code approach". In: *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*. IEEE. 2017, pp. 1–8.

[13] Minjun Chang et al. "An Optimally-Competitive Algorithm for Maximum Online Perfect Bipartite Matching with iid Arrivals". In: *Theory of Computing Systems* (2019), pp. 1–17.

[14] Jianer Chen, Yang Liu, and Songjian Lu. "An improved parameterized algorithm for the minimum node multiway cut problem". In: *Algorithmica* 55.1 (2009), pp. 1–13.

[15] Michael B Cohen, Yin Tat Lee, and Zhao Song. "Solving linear programs in the current matrix multiplication time". In: *Proceedings of the 51st annual ACM SIGACT symposium on theory of computing*. 2019, pp. 938–942.

[16] Derek G Corneil et al. "Simple linear time recognition of unit interval graphs". In: *Information processing letters* 55.2 (1995), pp. 99–104.

[17] José Correa et al. "Posted Price Mechanisms for a Random Stream of Customers". In: *Proceedings of the 2017 ACM Conference on Economics and Computation*. ACM. 2017, pp. 169–186. DOI: 10.1145/3033274.3085137.

[18] Marek Cygan et al. "Sitting closer to friends than enemies, revisited". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2012, pp. 296–307.

[19] Elias Dahlhaus et al. "The complexity of multiterminal cuts". In: *SIAM Journal on Computing* 23.4 (1994), pp. 864–894.

[20] Sina Dehghani et al. "Stochastic k-Server: How Should Uber Work?" In: *44th International Colloquium on Automata, Languages, and Programming*. Vol. 80. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 126:1–126:14.

[21] BV Dekster and JB Wilker. "Edge lengths guaranteed to form a simplex". In: *Archiv der Mathematik* 49.4 (1987), pp. 351–366.

[22] Jon Feldman et al. "Online ad assignment with free disposal". In: *International Workshop on Internet and Network Economics*. Springer. 2009, pp. 374–385. DOI: 10.1007/978-3-642-10841-9_34.

[23] Jon Feldman et al. "Online stochastic matching: Beating 1-1/e". In: *50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE. 2009, pp. 117–126. DOI: 10.1109/FOCS.2009.72.

[24] Xiaoli Zhang Fern and Carla E Brodley. "Solving cluster ensemble problems by bipartite graph partitioning". In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 36.

[25] Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in networks*. Princeton University Press, 1962.

[26] Andrew V Goldberg, Éva Tardos, and Robert E Tarjan. *Network flow algorithms*. Tech. rep. Cornell University Operations Research and Industrial Engineering, 1989.

[27] Anupam Gupta et al. "Stochastic Online Metric Matching". In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Ed. by Christel Baier et al. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 67:1–67:14. ISBN: 978-3-95977-109-2. DOI: 10.4230/LIPIcs.ICALP.2019.67.

[28] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[29] Dorit S Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.

[30] Petter Holme and Beom Jun Kim. "Growing scale-free networks with tunable clustering". In: *Physical review E* 65.2 (2002), p. 026107.

[31] Matthew Jenssen, Felix Joos, and Will Perkins. "On kissing numbers and spherical codes in high dimensions". In: *Advances in Mathematics* 335 (2018), pp. 307–321.

[32] Bala Kalyanasundaram and Kirk Pruhs. "Online weighted matching". In: *Journal of Algorithms* 14.3 (1993), pp. 478–488. DOI: 10.1006/jagm.1993.1026.

[33] David R Karger et al. "Rounding algorithms for a geometric embedding of minimum multiway cut". In: *Mathematics of Operations Research* 29.3 (2004), pp. 436–461.

[34] Richard M Karp, Umesh V Vazirani, and Vijay V Vazirani. "An optimal algorithm for on-line bipartite matching". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*. ACM. 1990, pp. 352–358. DOI: 10.1145/100216.100262.

[35] George Karypis and Vipin Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[36] Anne-Marie Kermarrec and Christopher Thraves. "Can everybody sit closer to their friends than their enemies?" In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2011, pp. 388–399.

[37] Thomas Kesselheim et al. "An Optimal Online Algorithm for Weighted Bipartite Matching and Extensions to Combinatorial Auctions". In: *European Symposium on Algorithms*. Springer. 2013, pp. 589–600. DOI: 10.1007/978-3-642-40450-4_50.

[38] Samir Khuller, Stephen G Mitchell, and Vijay V Vazirani. "On-line algorithms for weighted bipartite matching and stable marriages". In: *Theoretical Computer Science* 127.2 (1994), pp. 255–267. DOI: 10.1016/0304-3975(94)90042-6.

[39] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[40] Elias Koutsoupias. "The k-server problem". In: *Computer Science Review* 3.2 (2009), pp. 105–118. DOI: 10.1016/j.cosrev.2009.04.002.

[41] François Le Gall. "Powers of tensors and fast matrix multiplication". In: *Proceedings of the 39th international symposium on symbolic and algebraic computation*. 2014, pp. 296–303.

[42] Mohammad Mahdian and Qiqi Yan. "Online bipartite matching with random arrivals: an approach based on strongly factor-revealing lps". In: *Proceedings of the 43rd annual ACM symposium on Theory of computing*. ACM. 2011, pp. 597–606. DOI: 10.1145/1993636.1993716.

[43] Konstantin Makarychev, Yury Makarychev, and Aravindan Vijayaraghavan. "Bilulinial stable instances of max cut and minimum multiway cut". In: *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2014, pp. 890–906.

[44] Mark S Manasse, Lyle A McGeoch, and Daniel D Sleator. "Competitive algorithms for server problems". In: *Journal of Algorithms* 11.2 (1990), pp. 208–230. DOI: 10.1016/0196-6774(90)90003-W.

[45] Rajsekar Manokaran et al. "SDP gaps and UGC hardness for multiway cut, 0-extension, and metric labeling". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. ACM. 2008, pp. 11–20.

[46] Vahideh H Manshadi, Shayan Oveis Gharan, and Amin Saberi. "Online stochastic matching: Online actions based on offline statistics". In: *Mathematics of Operations Research* 37.4 (2012), pp. 559–573. DOI: 10.1287/moor.1120.0551.

[47] Dániel Marx. "Parameterized graph separation problems". In: *International Workshop on Parameterized and Exact Computation*. Springer. 2004, pp. 71–82.

[48] Aranyak Mehta et al. "Online matching and ad allocation". In: *Foundations and Trends in Theoretical Computer Science* 8.4 (2013), pp. 265–368. DOI: 10.1561/0400000057.

[49] Adam Meyerson, Akash Nanavati, and Laura Poplawski. "Randomized online algorithms for minimum metric bipartite matching". In: *Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2006, pp. 954–959.

[50] Linh Thi Hoai Nguyen et al. "Opinion formation over signed gossip networks". In: *SICE Journal of Control, Measurement, and System Integration* 10.3 (2017), pp. 266–273.

[51] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[52] Sharath Raghvendra. "A Robust and Optimal Online Algorithm for Minimum Metric Bipartite Matching". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Vol. 60. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 18:1–18:16. DOI: 10.4230/LIPIcs.APPROX-RANDOM.2016.18.

[53] Stephen M Robinson. "A characterization of stability in linear programming". In: *Operations Research* 25.3 (1977), pp. 435–447.

[54] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[55] Ankit Sharma and Jan Vondrák. "Multiway cut, pairwise realizable distributions, and descending thresholds". In: *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM. 2014, pp. 724–733.

[56] Guodong Shi et al. "The evolution of beliefs over signed social networks". In: *Operations Research* 64.3 (2016), pp. 585–604.

[57] N. J. A. Sloane. *The On-Line Encyclopedia of Integer Sequences*. https://oeis.org/A000088. Number of graphs on n unlabeled nodes.

[58] Martin Stein and Andreas Geyer-Schulz. "A Comparison of Five Programming Languages in a Graph Clustering Scenario." In: *Journal of Universal Computer Science* (2013).

[59] Suhang Wang et al. "Signed network embedding in social media". In: *Proceedings of the 2017 SIAM international conference on data mining*. SIAM. 2017, pp. 327–335.

[60] Shuhan Yuan, Xintao Wu, and Yang Xiang. "SNE: signed network embedding". In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2017, pp. 183–195.

[61] Quan Zheng and David B Skillicorn. "Spectral embedding of signed networks". In: *Proceedings of the 2015 SIAM international conference on data mining*. SIAM. 2015, pp. 55–63.

Appendix A

Isolating Cut Branch-and-Bound versus Linear Programming Branch-and-Bound: Tables and Figures

Table A.1: Real Data, 5 Terminals, Properties of Isolation Branching

Instance Properties				Iso Branching		Opt Solution	
Alias	**Category**	**#Vertices**	**#Edges**	**Seconds**	**#IsoCuts**	**Objective**	**MaxComponent**
pdzbase	Metabolic	161	209	0.034	5	8	0.652
adjnoun	Lexical	112	425	0.09	10	53	0.884
polbooks	Misc	105	441	0.14	75	50	0.514
ChicagoRegional	Infrastructure	823	822	0.185	10	4	0.417
netscience	Coauthorship	379	914	0.192	10	13	0.335
euroroad	Infrastructure	1039	1305	0.942	50	14	0.739
propro	Metabolic	1458	1948	0.175	5	6	0.974
celegans_metabolic	Metabolic	453	2025	0.089	5	84	0.956
jazz	HumanSocial	198	2742	0.104	5	67	0.960
ego-facebook	Social	2888	2981	0.344	5	7	0.631
email	Communication	1133	5451	4.76	35	136	0.994
vidal	Metabolic	2883	6007	0.585	5	6	0.957
powergrid	Infrastructure	4941	6594	25.5	75	19	0.789
petster-friendships	Social	1788	12476	0.711	5	4	0.987
as20000102	Computer	6474	12572	1.87	5	82	0.961
hep-th	Coauthorship	5835	13815	0.686	5	7	0.988
petster-hamster	Social	2000	16098	1.08	5	17	0.973
polblogs	Hyperlink	1222	16714	0.611	5	309	0.992
arenas-pgp	OnlineContact	10680	24316	2.24	5	17	0.965
as-22july06	Computer	22963	48436	5.78	5	187	0.923
as-caida20071105	Computer	26475	53381	5.45	5	138	0.938
astro-ph	Coauthorship	14845	119652	5.22	5	38	0.988
reactome	Metabolic	5973	145778	8.1	5	63	0.992
ca-AstroPh	Coauthorship	17903	196972	6.81	5	82	0.996

Properties of Isolation Branching in instances with 5 terminals: the number of minimum isolating cuts performed, the value of the optimal solution, and the fraction of the graph in the largest component of the optimal partition. The instances are sorted by the number of edges.

Table A.2: Real Data, 5 Terminals, Running Times

Instance Properties				Int Programming	Iso Branching	
Alias	**Category**	**#Vertices**	**#Edges**	**Seconds**	**Seconds**	**Ratio**
pdzbase	Metabolic	161	209	0.33	0.034	9.71
adjnoun	Lexical	112	425	0.52	0.09	5.78
polbooks	Misc	105	441	0.5	0.14	3.57
ChicagoRegional	Infrastructure	823	822	1.38	0.185	7.46
netscience	Coauthorship	379	914	1.14	0.192	5.94
euroroad	Infrastructure	1039	1305	2.35	0.942	2.49
propro	Metabolic	1458	1948	3.78	0.175	21.60
celegans_metabolic	Metabolic	453	2025	2.54	0.089	28.54
jazz	HumanSocial	198	2742	3.46	0.104	33.27
ego-facebook	Social	2888	2981	8.98	0.344	26.10
email	Communication	1133	5451	9.23	4.76	1.94
vidal	Metabolic	2883	6007	9.88	0.585	16.89
powergrid	Infrastructure	4941	6594	24.2	25.5	0.95
petster-friendships	Social	1788	12476	19.7	0.711	27.71
as20000102	Computer	6474	12572	21.4	1.87	11.44
hep-th	Coauthorship	5835	13815	17.7	0.686	25.80
petster-hamster	Social	2000	16098	29	1.08	26.85
polblogs	Hyperlink	1222	16714	20.5	0.611	33.55
arenas-pgp	OnlineContact	10680	24316	45	2.24	20.09
as-22july06	Computer	22963	48436	74.3	5.78	12.85
as-caida20071105	Computer	26475	53381	76.2	5.45	13.98
astro-ph	Coauthorship	14845	119652	160	5.22	30.65
reactome	Metabolic	5973	145778	179	8.1	22.10
ca-AstroPh	Coauthorship	17903	196972	240	6.81	35.24

The running time, in CPU seconds, of Isolation Branching versus Integer Programming with Gurobi in instances with five terminals. Both algorithms were run to optimality on all instances. The instances are sorted by the number of edges. The "improvement ratio" is the running time of Integer Programming divided by the running time of Isolation Branching.

Table A.3: Real Data, 10 Terminals, Properties of Isolation Branching

Instance Properties				Iso Branching		Opt Solution	
Alias	**Category**	**#Vertices**	**#Edges**	**Seconds**	**#IsoCuts**	**Objective**	**MaxComponent**
pdzbase	Metabolic	161	209	0.07	30	19	0.261
adjnoun	Lexical	112	425	1.32	120	84	0.804
polbooks	Misc	105	441	15	1530	93	0.438
ChicagoRegional	Infrastructure	823	822	0.188	20	9	0.142
netscience	Coauthorship	379	914	0.251	10	27	0.235
euroroad	Infrastructure	1039	1305	7.59	300	25	0.612
propro	Metabolic	1458	1948	0.182	10	18	0.892
celegans_metabolic	Metabolic	453	2025	0.165	10	125	0.934
jazz	HumanSocial	198	2742	1.05	20	182	0.934
ego-facebook	Social	2888	2981	61.5	12320	103	0.262
email	Communication	1133	5451	208	1140	248	0.976
vidal	Metabolic	2883	6007	0.385	10	12	0.948
powergrid	Infrastructure	4941	6594	392	1810	35	0.293
petster-friendships	Social	1788	12476	0.479	10	153	0.978
as20000102	Computer	6474	12572	1.98	10	296	0.942
hep-th	Coauthorship	5835	13815	0.87	10	21	0.968
petster-hamster	Social	2000	16098	0.522	10	28	0.960
polblogs	Hyperlink	1222	16714	0.618	10	27	0.981
arenas-pgp	OnlineContact	10680	24316	1.71	10	26	0.952
as-22july06	Computer	22963	48436	8.55	10	508	0.911
as-caida20071105	Computer	26475	53381	9.75	10	537	0.887
astro-ph	Coauthorship	14845	119652	7.41	10	54	0.982
reactome	Metabolic	5973	145778	6.98	20	75	0.986
ca-AstroPh	Coauthorship	17903	196972	7.2	10	96	0.993

Properties of Isolation Branching in instances with 10 terminals: the number of minimum isolating cuts performed, the value of the optimal solution, and the fraction of the graph in the largest component of the optimal partition. The instances are sorted by the number of edges.

Table A.4: Real Data, 10 Terminals, Running Times

Instance Properties				Int Programming	Iso Branching	
Alias	**Category**	**#Vertices**	**#Edges**	**Seconds**	**Seconds**	**Ratio**
pdzbase	Metabolic	161	209	0.66	0.07	9.43
adjnoun	Lexical	112	425	0.99	1.32	0.75
polbooks	Misc	105	441	1.01	15	0.07
ChicagoRegional	Infrastructure	823	822	2.28	0.188	12.13
netscience	Coauthorship	379	914	3.27	0.251	13.03
euroroad	Infrastructure	1039	1305	4.43	7.59	0.58
propro	Metabolic	1458	1948	5.64	0.182	30.99
celegans_metabolic	Metabolic	453	2025	5.73	0.165	34.73
jazz	HumanSocial	198	2742	6.61	1.05	6.30
ego-facebook	Social	2888	2981	7.76	61.5	0.13
email	Communication	1133	5451	15	208	0.07
vidal	Metabolic	2883	6007	16.9	0.385	43.90
powergrid	Infrastructure	4941	6594	440	392	1.12
petster-friendships	Social	1788	12476	34.6	0.479	72.23
as20000102	Computer	6474	12572	41.3	1.98	20.86
hep-th	Coauthorship	5835	13815	40.9	0.87	47.01
petster-hamster	Social	2000	16098	38.4	0.522	73.56
polblogs	Hyperlink	1222	16714	47	0.618	76.05
arenas-pgp	OnlineContact	10680	24316	64.3	1.71	37.60
as-22july06	Computer	22963	48436	253	8.55	29.59
as-caida20071105	Computer	26475	53381	209	9.75	21.44
astro-ph	Coauthorship	14845	119652	338	7.41	45.61
reactome	Metabolic	5973	145778	552	6.98	79.08
ca-AstroPh	Coauthorship	17903	196972	1120	7.2	155.56

The running time, in CPU seconds, of Isolation Branching versus Integer Programming with Gurobi in instances with ten terminals. Both algorithms were run to optimality on all instances. The instances are sorted by the number of edges. The "improvement ratio" is the running time of Integer Programming divided by the running time of Isolation Branching.

Table A.5: Simulated Data, 5 Terminals, Properties of Isolation Branching

Instance Properties	**Iso Branching**		
#Vertices	**Avg Time**	**Deviation**	**Avg #IsoCuts**
1000	17	4.3	67
2000	56	12.3	99
3000	65	17.6	69
4000	102	24.7	81
5000	234	49.6	138
6000	171	41.7	81
7000	345	64.5	135
8000	479	82.0	126
9000	466	80.3	131

Properties of Isolation Branching on simulated data sets with five terminals, generated according to the Powerlaw Cluster model, with 10 new edges per vertex and probability 0.1 of creating a triangle.

Table A.6: Simulated Data, 5 Terminals, Running Times

Instance Properties	**Integer Programming**		**Isolation Branching**		
#Vertices	**Avg Time**	**Deviation**	**Avg Time**	**Deviation**	**Ratio**
1000	13	0.3	17	4.3	0.74
2000	50	15.0	56	12.3	0.90
3000	192	86.6	65	17.6	2.94
4000	198	75.4	102	24.7	1.94
5000	585	161.0	234	49.6	2.50
6000	756	168.9	171	41.7	4.42
7000	1791	396.2	345	64.5	5.19
8000	2999	1322.6	479	82.0	6.26
9000	6573	1625.3	466	80.3	14.11

The average and standard deviation running time of Isolation Branching versus Integer Programming, measured in CPU seconds, on simulated data sets with five terminals, generated according to the Powerlaw Cluster model with 10 new edges per vertex and probability 0.1 of creating a triangle. For each size graph, 10 random instances were generated. Both algorithms were run to optimality on all instances. The "improvement ratio" is the running time of Integer Programming divided by the running time of Isolation Branching.

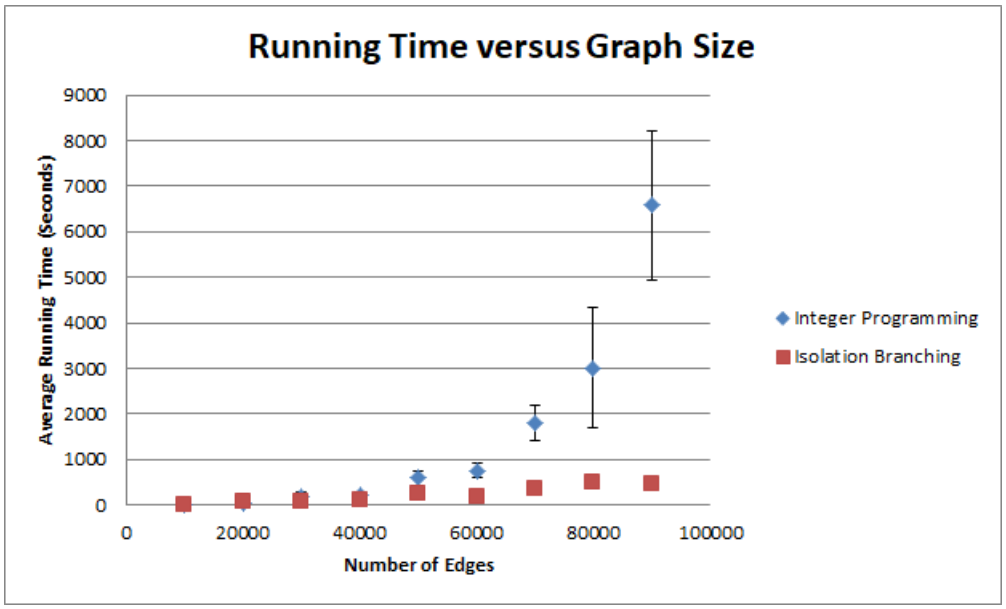


Figure A.1: The average running time of ISOLATING CUT BRANCH-AND-BOUND versus Gurobi Integer Programming on ten random instances of k-`terminal cut` generated using the PowerlawCluster model.

Appendix B

Notable Instances of Online Perfect Matching with i.i.d. Arrivals

In our empirical study, we generated 1000 random instances and computed the exact expected performance of all seven algorithms on each instance. The running time of computing the expected performance of each algorithm was exponential in the number of workers, so we limited ourselves to random instances with at most 10 workers and 10 job types. The utilities in each expectation graph were selected uniformly at random in $[0, 1]$. The relative expected performance of the algorithms is scale-invariant, so any instance on at most 10 workers and 10 job types can be represented this way.

From all 1000 instances, we generated table 4.2, comparing the relative expected performance of the seven algorithms. In this appendix, we present a set of four "representative" instances out of the 1000. If we wrote $\mathcal{ALG}_1 <> \mathcal{ALG}_2$ in table 4.2, then there exist two instances among these four, (W, J) and (W', J'), where $\text{ALG}_1(W, J) > \text{ALG}_2(W, J)$, but $\text{ALG}_1(W', J') < \text{ALG}_2(W', J')$. For the sake of presentation, we scaled and rounded the utilities of the four instances, checking to ensure that the relative expected performance of the algorithms was preserved.

The relative expected performance of the algorithms on these four instances is summarized in the following table:

Instance	Algorithms
B.1	GREEDY(W, J) > EVAL-TPP(W, J) > EVAL-DISPATCH(W, J) > EVAL-RAND(W, J) > OPT-FLOW(W, J)
B.2	OPT-FLOW(W, J) > EVAL-TPP(W, J) > DISPATCH(W, J) > GREEDY(W, J)
B.3	EVAL-RAND(W, J) > EVAL-DISPATCH(W, J) > GREEDY(W, J) > OPT-FLOW(W, J) > EVAL-TPP(W, J)
B.4	EVAL-TPP(W, J) > EVAL-DISPATCH(W, J) > OPT-FLOW(W, J) > EVAL-RAND(W, J) > GREEDY(W, J)

The code for our empirical study can be found here:

`https://github.com/marvel2010/dispatch-variants`.

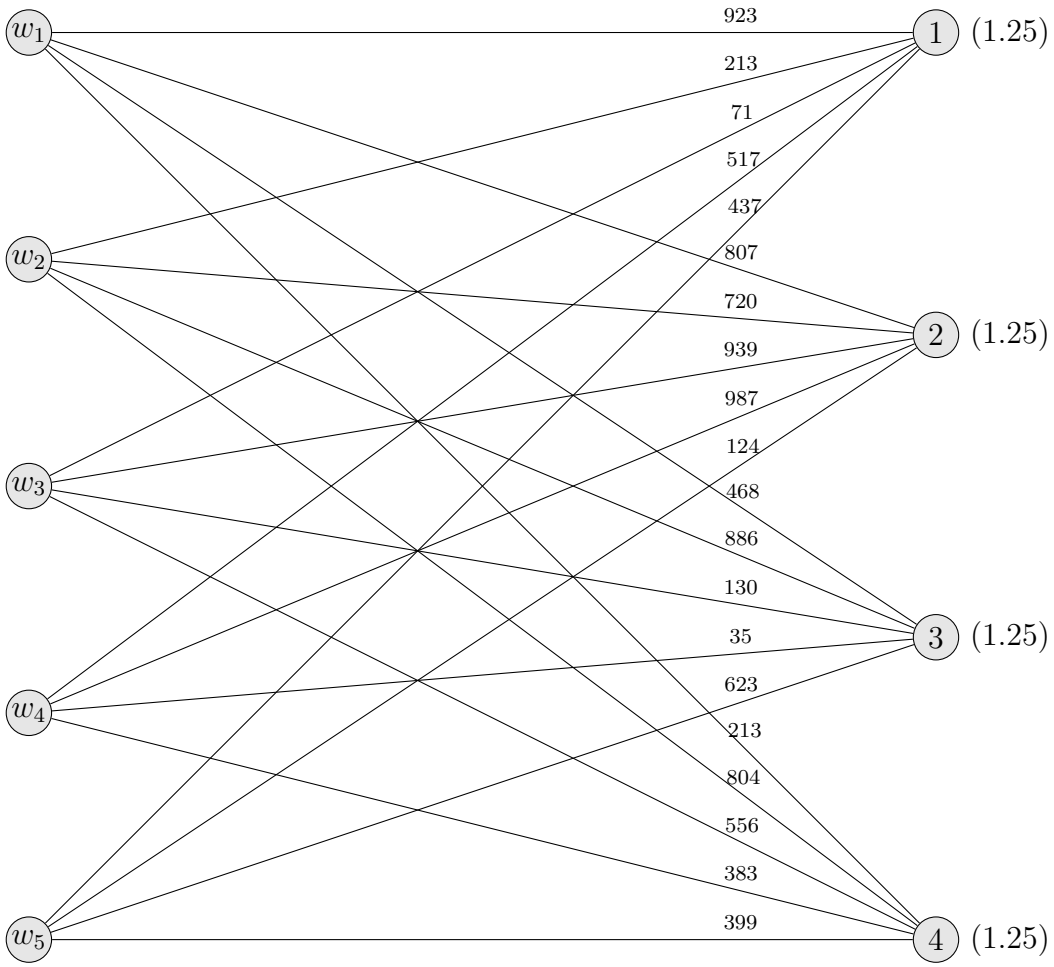


Figure B.1: An instance, discovered through random experiments, on which the greedy algorithm outperforms all the other evaluation-guided algorithms in expectation. Plus, all the evaluation-guided algorithms outperform the optimal flow-guided algorithm in expectation. To be precise, GREEDY(W, J) > EVAL-TPP(W, J) > EVAL-DISPATCH(W, J) > EVAL-RAND(W, J) > OPT-FLOW(W, J).

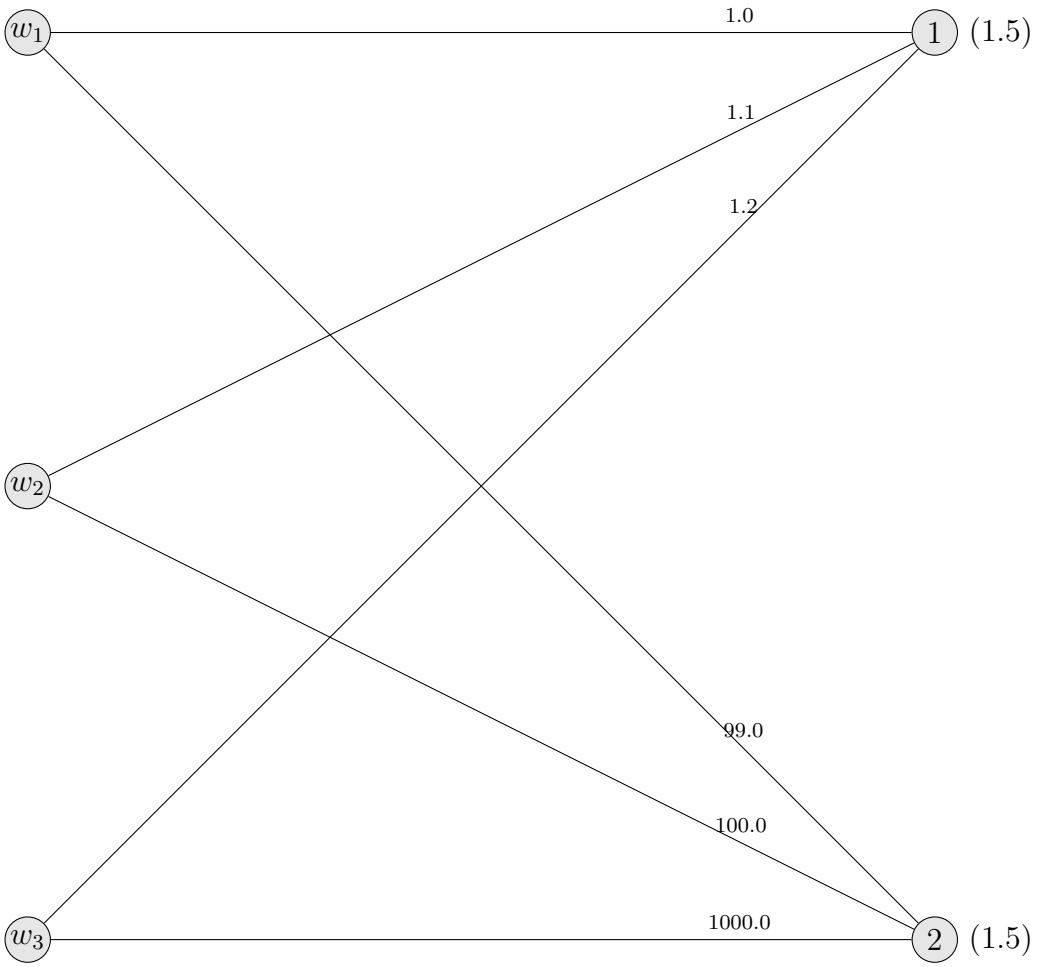


Figure B.2: An instance on which the optimal flow-guided algorithm outperforms the evaluation-guided algorithms EvalTPP and Greedy in expectation. In addition, the flow-guided algorithm Dispatch also outperforms the evaluation-guided algorithm Greedy in expectation. To be precise, OPT-FLOW$(W, J) >$ EVAL-TPP$(W, J) >$ DISPATCH$(W, J) >$ GREEDY(W, J).

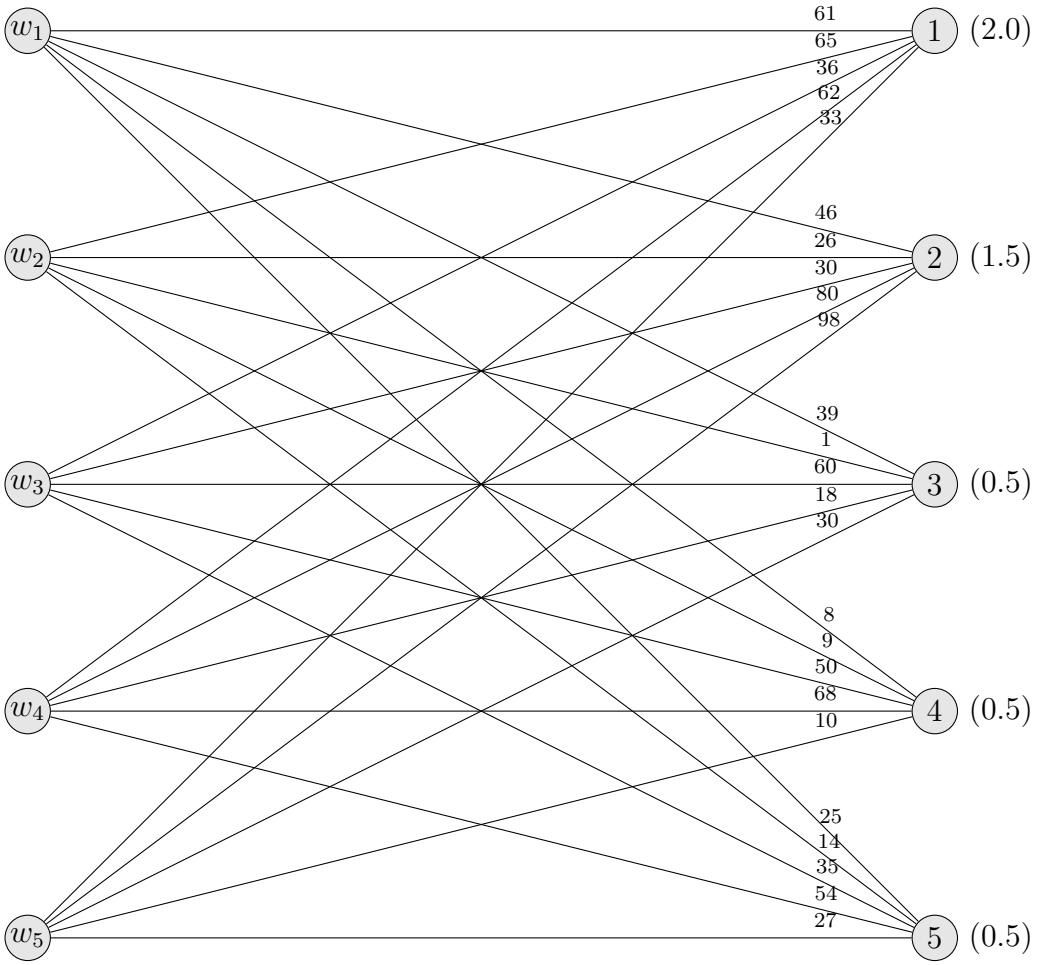


Figure B.3: An instance, discovered through random experiments, on which the evaluation-guided algorithm EvalRand outperforms all the other evaluation-guided algorithms in expectation. To be precise, EVAL-RAND(W, J) > EVAL-DISPATCH(W, J) > GREEDY(W, J) > OPT-FLOW(W, J) > EVAL-TPP(W, J).

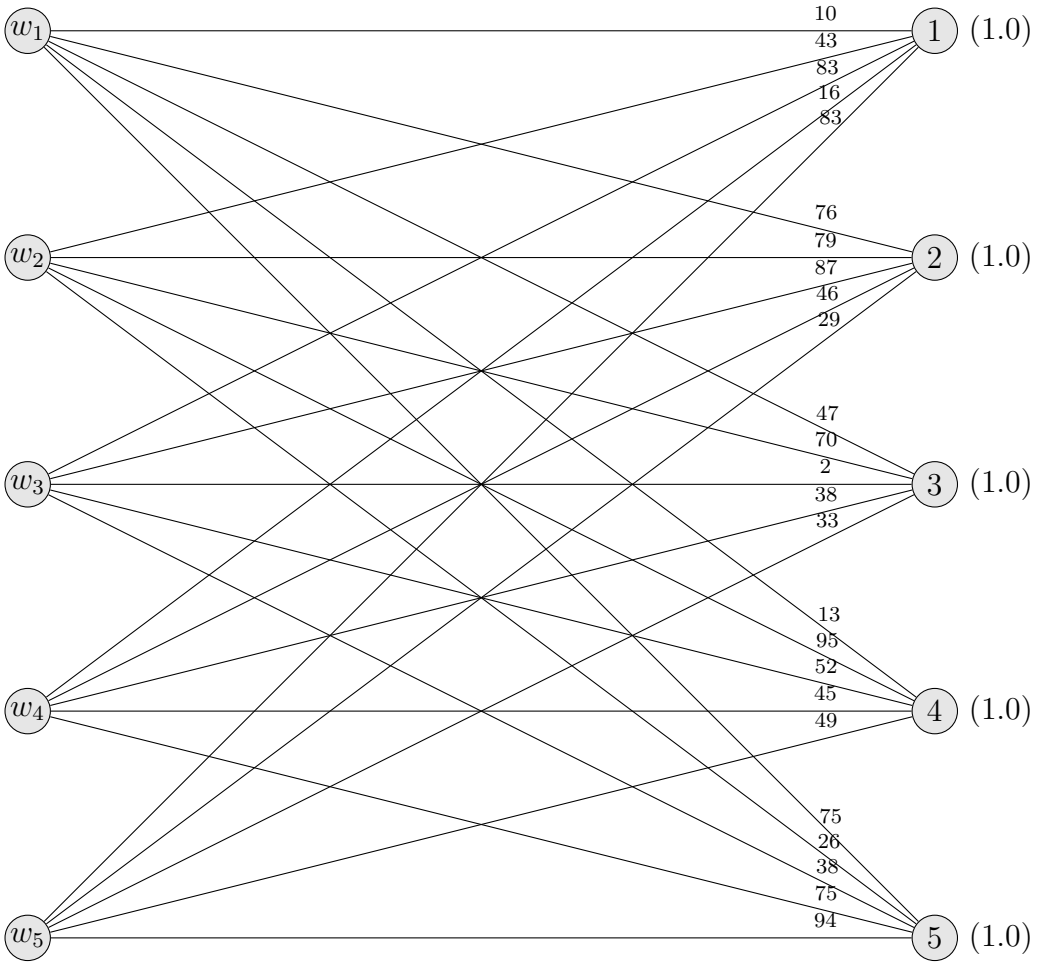


Figure B.4: An instance, discovered through random experiments, on which the optimal flow-guided algorithm outperforms the evaluation-guided algorithm EvalRand in expectation. To be precise, EVAL-TPP(W, J) > EVAL-DISPATCH(W, J) > OPT-FLOW(W, J) > EVAL-RAND(W, J) > GREEDY(W, J).