Abstract of “Efficient Algorithms for Shortest-Path and Maximum-Flow Problems in Planar Graphs” by Shay Mozes, Ph.D., Brown University, May 2013.

Large graphs are ubiquitous in many diverse fields ranging from sociology and economics to biology and engineering. Applications in numerous areas such as transportation, geographical routing, computer vision and VLSI design involve solving optimization problems on large *planar* graphs. The sheer growth in the number and size of available data sets drives a need to design optimization algorithms that are not merely efficient in the qualitative sense that their computational resource consumption is polynomial in the size of the input, but in the stronger quantitative sense that the amount of required resources (specifically, running time and space) is as close to linear as possible. In this thesis we study the combinatorial structural properties of directed planar graphs, and exploit these properties to devise efficient algorithms for shortest-path and maximum-flow problems in such graphs. Among the results we present are:

- A linear-space algorithm for the shortest-path problem in directed planar graphs with real (positive and negative) arc lengths that runs in near-linear time. Our algorithm is more efficient and conceptually simpler than previously known algorithms for this problem.
- Data structures for efficiently reporting exact distances in directed planar graphs (distance oracles) with significantly improved space-to-query-time ratio over previously known exact distance oracles.
- A near-linear time algorithm for computing a maximum flow in directed planar graphs with multiple sources and sinks. This problem has been open for more than 20 years. No planarity exploiting algorithm for the problem was previously known. Our algorithm is significantly faster than previous algorithms for this problem.

These problems, and the tools and techniques used in solving them, are interrelated.

Abstract of “Efficient Algorithms for Shortest-Path and Maximum-Flow Problems in Planar Graphs” by Shay Mozes, Ph.D., Brown University, May 2013.

Large graphs are ubiquitous in many diverse fields ranging from sociology and economics to biology and engineering. Applications in numerous areas such as transportation, geographical routing, computer vision and VLSI design involve solving optimization problems on large *planar* graphs. The sheer growth in the number and size of available data sets drives a need to design optimization algorithms that are not merely efficient in the qualitative sense that their computational resource consumption is polynomial in the size of the input, but in the stronger quantitative sense that the amount of required resources (specifically, running time and space) is as close to linear as possible. In this thesis we study the combinatorial structural properties of directed planar graphs, and exploit these properties to devise efficient algorithms for shortest-path and maximum-flow problems in such graphs. Among the results we present are:

- A linear-space algorithm for the shortest-path problem in directed planar graphs with real (positive and negative) arc lengths that runs in near-linear time. Our algorithm is more efficient and conceptually simpler than previously known algorithms for this problem.
- Data structures for efficiently reporting exact distances in directed planar graphs (distance oracles) with significantly improved space-to-query-time ratio over previously known exact distance oracles.
- A near-linear time algorithm for computing a maximum flow in directed planar graphs with multiple sources and sinks. This problem has been open for more than 20 years. No planarity exploiting algorithm for the problem was previously known. Our algorithm is significantly faster than previous algorithms for this problem.

These problems, and the tools and techniques used in solving them, are interrelated.

Efficient Algorithms for Shortest-Path and Maximum-Flow Problems in Planar Graphs

by
Shay Mozes
B. Sc. Summa Cum Laude Computer Science and Physics, The Hebrew University of Jerusalem, Jerusalem, Israel, 2002
M. Sc. Physics, Tel Aviv University, Tel Aviv-Yafo, Israel 2004
Sc. M. Computer Science, Brown University, Providence, RI, 2009

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2013

This dissertation by Shay Mozes is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____
Philip N. Klein, Director

Recommended to the Graduate Council

Date _____ _____
Jeff Erickson, Reader
(University of Illinois at Urbana-Champaign)

Date _____ _____
Jonathan A. Kelner, Reader
(Massachusetts Institute of Technology)

Date _____ _____
Claire Mathieu, Reader

Approved by the Graduate Council

Date _____ _____
Peter M. Weber
Dean of the Graduate School

Vita

Shay Mozes was born in Jerusalem, Israel on January eleventh, 1976. He is married to Edya Ladan Mozes, and father to Amit and Einat. He attended the Hebrew university in Jerusalem from 1999 to 2002, earning a Bachelor of Science degree in Computer Science and in Physics Summa cum Laude. He attended Tel-Aviv University between 2002 and 2004, earning a Masters degree in Physics (Quantum Information). Shay has earned an M.Sc. degree in Computer Science in 2009 while working towards a doctorate in Computer Science at Brown University. At the Hebrew University he was part of the *Amirim* Honors program for studies of the natural sciences. At Tel-Aviv University, he received the Ann and Maurice Cohen award. At Brown he received two Kanellakis graduate fellowship, and a Brown University dissertation fellowship. He received the best paper award at the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007), and the best student paper award at the 18th Annual European Symposium on Algorithms (ESA 2010).

Acknowledgements

First I would like to thank Philip Klein for bringing me into the beautiful world of planar graphs, for his enthusiasm, patience, flexibility and respect. I am grateful for the opportunity to work closely with Philip, learning about creativity, determination, and the process of doing research. I had enjoyed the process as much as the outcome, and look forward to collaborating with him in the future.

For me, one of the greatest pleasures in research is collaboration, and the ability to share the excitement of discovery, as well as the daily burden of feeling lost or stuck. I thank all of my collaborators - Cora Borradaile, Aparna Das, Erik Demaine, Eli Fox-Epstein, Crystal Kahn, Haim Kaplan, Claire Mathieu, Yahav Nussbaum, Krzysztof Onak, Ben Raphael, Ben Rossman, Micha Sharir, Christian Sommer, Dekel Tzur, Oren Weimann, Christian Wulff-Nilsen and Michal Ziv-Ukelson - for sharing this experience with me and for making this thesis possible.

I owe a special debt to Oren Weimann. It was with him that I discovered my pleasure and excitement with research in computer science during our first years in the Boston area. Our friendship and collaboration in the past years are priceless, and I look forward to continuing and expanding those in the years to come.

The past couple of years were especially enjoyable thanks to the close collaboration with Christian Sommer, whose friendship I cherish. I also thank Christian for commenting on parts of this thesis.

Teaching 6.889 last fall at MIT was a great experience that I have enjoyed and learned a lot from. I thank Erik Demaine for letting me have this opportunity, as well as for his advice on teaching. It was a pleasure to co-teach this class with him and with Christian Sommer and Siamak Tazari. I am grateful to Philip Klein for encouraging me to teach the class, for accommodating this unusual arrangement, and for allowing us to use the draft of his book.

I spent the two years prior to starting at Brown, as well as large portions of the time since then as a guest at MIT. I would like to thank the institution as well as the individuals for letting me use the facilities and for creating a welcoming environment for an outsider. I thank Eddie Farhi for welcoming me into his group when we just moved to Cambridge; Piotr Indyk, David Karger, Manolis Kellis, Jon Kelner, Peter Shor and Ike Chuang for allowing me to sit in their classes; and Erik Demaine, Charles Leiserson, Shafi Goldwasser and Nir Shavit for their help and advice.

Last, but not least, I would like dedicate this thesis to my family. To my wife Edya - for *everything*.

To Amit and Einat, for distracting me, and for accepting that sometimes I am distracted. To my parents, for their unconditional support and love. To our extended families in Israel - for being there for us.

Contents

⋆ Parts of this thesis have appeared in print before [KMW09, KMW10, MWN10, MS12, BKM+11, KM11]. I thank my coauthors for their permission in using portions of these papers in this thesis.

List of Figures

Part I

Introduction

This thesis is concerned with the study and use of combinatorial structural graph properties in obtaining efficient algorithms for fundamental optimization problems in planar graphs.

Graphs, or networks, are among the most basic and powerful objects to model relations and processes in sociology, economy, physics, chemistry, computer science, biology, engineering, and anthropology, as well as many other fields. While networks were around long before the modern computer entered the scene, the tremendous growth in the availability of data and in the size of networks is driving an increasing need to devise computational methods to represent, store, and process large graph data in a cost-effective and timely manner.

Computer scientists are using various approaches to meet those challenges. For example, on the systems side we have witnessed the emergence of large-scale data-centers and multi-core processors. On the theoretical algorithmic side, a lot of effort has been put into developing new techniques that give rise to various kinds of *approximation* algorithms. These are algorithms that provide weaker, relaxed output guarantees, but typically run faster and require less memory space than exact algorithms. In theoretical computer science the resources, such as time and space, that are required to solve a problem are measured according to their asymptotic dependency on the size of the input. This makes particular sense for large networks, where the input size is such that the asymptotic behavior is actually observed in practice. However, for such large networks, algorithms whose resource consumption is more than slightly linear in the size of the inputs are impractical. Resorting to sub-optimal solutions is partially based on the assumption that computing an exact optimal solution for big problem instances using traditional combinatorial methods requires too much resources to be considered practical. In this thesis we show that, contrary to this implicit assumption, combinatorial methods can be used to quickly compute optimal solutions. Specifically, we show the following:

> *There exist exact algorithms for fundamental important optimization problems in planar graphs that require nearly-linear time and space. The methods used in these algorithms are purely combinatorial.*

Informally, planar graphs are graphs that can be drawn on the plane such that edges only intersect at their endpoints. This is an important family of graphs that has been extensively studied and has many practical applications in areas such as transportation networks [Eul41, HR55], geographical routing for communication networks [BMSU99], computer vision [GPS89, Epp97], VLSI design [Lei80, RBK90], and finite-element problems [Zie77, LRT79]. Planar graphs exhibit a wealth of beautiful structural properties that can be exploited to achieve efficient algorithms. We find the study of these structural properties and the relations they reveal between different optimization problems interesting and rewarding in its own right.

Contributions and Organization

We consider two fundamental optimization problems on planar graphs. In Part II of this thesis we present our results on shortest-path problems. In Part III we present our results on maximum flow problems. As we shall see, the two problems are closely related in planar graphs. Indeed, techniques

and tools we devise for the shortest-path problems are used in our solutions for the maximum flow problem.

In Chapter 2 we give an algorithm for the single source shortest-path problem in directed planar graphs with real (positive and negative) arc lengths. Like previous algorithms for this problem [LT79, HKRS97, FR06], Our algorithm uses divide-and-conquer based on small planar separators. We achieve linear space and $O(n \log^2 n / \log\log n)$ time by exploiting a structural property of shortest paths known as the *Monge* property. To date, our algorithm is the fastest known for this problem, and is arguably simpler than the previous fastest algorithm [FR06], which required $O(n \log^3 n)$ time and $O(n \log n)$ space. Our algorithm has been generalized for graphs with bounded genus [CEN09]. The results in this chapter are based on work with Philip Klein and Oren Weimann [KMW10], and with Christian Wulff-Nilsen [MWN10].

In Chapter 3 we present an extension of a data structure of Fakcharoenphol and Rao [FR06] for efficiently implementing Dijkstra's algorithm in certain dense graphs. This extension will be later used in Chapter 5. Fakcharoenphol and Rao's data structure and its variants have emerged as a useful tool in quite a few works in recent years [BSWN10, INSWN11, MS12, BKM+11, KMNS12, ŁS11, Nus11, CR10]. To obtain this result we use a technique related to Monge matrices that was developed for the shortest-path problem in Chapter 2, and a data structure for range minimum query in Monge matrices. We are hopeful that this detailed treatment of Fakcharoenphol and Rao's data structure, as well as the clear statement of its known extension, will assists others in using this algorithmic tool. The results in this chapter are based on work with Cora Borradaile, Philip Klein, Yahav Nussbaum and Christian Wulff-Nilsen [BKM+11], and with Haim Kaplan, Yahav Nussbaum and Micha Sharir [KMNS12].

In Chapter 4 we develop data structures for efficiently reporting exact distances in a planar graph. The first contribution is a (strictly) linear size data structure that reports distances in sublinear time ($O(n^{1/2+\epsilon})$ time for any constant $\epsilon > 0$). No such data structure was known prior to this work.[1] The main contribution presented in this chapter is the following. Given a desired space allocation $S \in [n \lg\lg n, n^2]$, we show how to construct in $\tilde{O}(S)$ time[2] a data structure of size $O(S)$ that answers distance queries in $\tilde{O}(n/\sqrt{S})$ time per query. Such a tradeoff between space and query time was previously only known for $S \in [n^{4/3}, n^2]$ [Dji96, CX00, Cab12]. For $\sigma \in (1, 4/3)$ and space $S = n^\sigma$, we essentially improve the query time from $\tilde{O}(n^2/S)$ [Dji96, ACC+96] to $\tilde{O}(\sqrt{n^2/S})$. To obtain these results we employ non-uniform recursive r–divisions using cycle separators, and extensively use Fakcharoenphol and Rao's efficient implementation of Dijkstra's algorithm. The results in this chapter are based on work with Christian Sommer [MS12].

Part III, which discusses flow problems, includes two main results. In Chapter 5 we present a nearly-linear time algorithm for solving the maximum flow problem in a directed planar graph with multiple sources and sinks. This problem was studied by Miller and Naor [MN95], who gave efficient algorithms, that exploit planarity, for the special case where the sources and sinks all lie

[1] Yahav Nussbaum has independently obtained a similar result [Nus11]

[2] The $\tilde{O}(\cdot)$ notation hides polylogarithmic factors

on a small number of faces. However, the problem of maximum flow with multiple sources and sinks without any additional restrictions remained open for more than twenty years. The problem can be reduced to the single-source single-sink case by introducing an artificial source and sink and connecting them to all the sources and sinks, respectively—but this reduction does not preserve planarity. Indeed, no planarity-exploiting algorithm was known for the problem, and the fastest known algorithm for computing multiple-source multiple-sink max-flow in a planar graph has been to use this reduction in conjunction with a general maximum-flow algorithm such as that of Sleator and Tarjan [ST83], which leads to a running time of $O(n^2 \log n)$. For integer capacities less than U, one could instead use the algorithm of Goldberg and Rao [GR98], which leads to a running time of $O(n^{1.5} \log n \log U)$. Our algorithm, on the other hand, uses planar cycle separators, and an adaptation of a method to compute max st-flow when s and t are adjacent [Has81] to work with our extension of Fakcharoenphol and Rao's efficient implementation of Dijkstra's algorithm (Chapter 3). The algorithm runs in $O(n \log^3 n)$ time. The results in this chapter are based on work with Cora Borradaile, Philip Klein, Yahav Nussbaum and Christian Wulff-Nilsen [BKM+11].

Finally, in Chapter 6, we present an algorithm that solves the maximum flow problem with multiple sources and a single sink in a directed planar graph in $O(\text{diameter} \cdot n \log n)$, where diameter is the diameter of the face-vertex incidence graph of G. While the running time is greater than that of the multiple-source multiple-sink maximum flow algorithm of Chapter 5, the techniques used are interesting and different. The algorithm may be thought of as an extension of Klein's multiple-source shortest-path algorithm [Kle05], and we are hopeful that the technique will find applications in related problems.

Preliminary definitions as well as necessary concepts and tools that had appeared in the literature prior to our work are covered in Chapter 1. In addition, the technical contributions on each of the subsequent chapters are preceded by discussions of related work and context.

Chapter 1

Preliminaries

1.1 Graphs

This thesis is mostly concerned with directed graphs. We adopt the point of view that every graph is directed, but sometimes the direction is not specified or may be ignored. Throughout, we use the terms *arc* and *edge* for the directed and undirected case, respectively. In this text we shall use both the natural intuitive geometric view of embedded graphs and the formal algebraic formulation. We will comment about the relation between the two where appropriate.

1.1.1 Vertex- and Edge-Oriented Definitions of Graphs

There are two almost equivalent definitions of graphs. The common one is the vertex-oriented approach, in which a graph is a pair $G = \langle V, A \rangle$, where V is the vertex (or node[1]) set, and A, the arc set, is a subset of $V \times V$. For nodes $u, v \in V$ we use the notation uv to denote the element (u, v) of $V \times V$, and say that the arc uv is oriented from u to v. The node u is called the tail of uv, written $u = \text{tail}(uv)$. Similarly, $v = \text{head}(uv)$ is the head of uv. We use $\text{tail}_G(uv)$ whenever the graph G in question is not clear from the context. We use $V(G)$ to denote the set of nodes of a graph G, and use $|V(G)|$, or sometimes just $|G|$ to denote the number of nodes of G.

We associate with each arc $a \in A$ two darts a^{+1} and a^{-1} (we will also use just a^+ and a^-). One may identify the dart a^+ with the arc a and a^- with the arc a after reversing its orientation. We define $\text{rev}(\cdot)$ to be the bijection on darts $\text{rev}(a^\sigma) = a^{-\sigma}$. Darts are convenient since they allow us to argue about the reverses of arcs without requiring that these reverse-arcs be present in the graph. As we will see later, they are particularly convenient for describing planar graphs and their embeddings. We strongly believe that, where appropriate, darts should become a standard concept in graph theory, similar to edges and arcs.

In the edge-oriented approach, A is a set of elements (arcs) and V is a partition of the set of darts $A \times \{+1, -1\}$. A vertex v is a block of the partition V. The set of darts in that block are said

[1] We make no distinction between the terms vertex and node, and use them interchangeably.

to be oriented towards v, which we call the head of each of these darts. A vertex v is called the tail of a dart a^σ if v is the head of $a^{-\sigma}$. The edge-centric approach is very convenient for representing embedded graphs and their duals. We will use the two approaches interchangeably as convenient.

1.1.2 Paths, Cycles, Trees and Cuts

An x-to-y *walk* is a sequence of darts $d_1 \dots d_k$ such that tail$(d_1) = x$, head$(d_k) = y$ and, for $i = 2, \dots, k$, head$(d_{i-1}) =$ tail(d_i). A walk in which no dart appears more than once is called a *path*. An empty sequence represents the trivial path consisting of a single vertex. We use start(P) to denote the first vertex, x, of P and end(P) to denote the last vertex, y, of P. If additionally head$(d_k) =$ tail(d_1) then the walk is a *cycle*. A walk is *simple* if no vertex occurs twice as the head of a dart in the walk.

If $P = d_1 \dots d_k$ and $Q = e_1 \dots e_\ell$ are walks such that end$(P) =$ start(Q), we use $P \circ Q$ to denote the walk $d_1 \dots d_k e_1 \dots e_\ell$. If u and v are vertices in path P such that $u =$ tail(d_i), $v =$ head(d_j) and $i \leq j$, we use $P[u, v]$ to denote the subpath P' such that start$(P') = u$ and end$(P') = v$. Since walks may visit vertices or darts multiple times, when we use this notation, we intend u and v to refer to specific occurrences of vertices within the path. If P is a cycle, and u occurs before v in P then $P[u, v]$ denotes a subpath of the cycle. We use $P[u, v)$ to denote the walk obtained from the walk $P[u, v]$ by deleting the last dart; $P(u, v]$ and $P(u, v)$ are defined analogously. $P[\cdot, v]$ denotes the subpath P' with start$(P') =$ start(P) and end$(P') = v$. $P[u, \cdot]$ is similarly defined. The reverse of P, rev(P) is defined as the sequence rev(d_k), rev$(d_{k-1}), \dots,$ rev(d_1).

A graph G is *connected* if for every pair of nodes $u, v \in V$ there exists a u-to-v path of darts in G. G is said to be k-*connected* if there does not exist a set of $k-1$ nodes whose removal disconnects the graph.

A subset $A' \subseteq A$ induces a subgraph $\langle V', A' \rangle$ of G, where V' is the set of nodes that are endpoints of arcs in A' (in the edge-centric definition, V' is the restriction of the partition V to the dart set of A'). For a set A' of arcs, we denote by $G - A'$ the subgraph of G induced by $A - A'$.

A subset $V' \subseteq V$ induces a subgraph $\langle V', A' \rangle$ of G, where A' is the set of arcs that have both their endpoints in V'. For a set V' of nodes, we denote by $G - V'$ the subgraph of G induced by $V - V'$.

A *spanning tree* T of G is a connected subgraph of G that includes all nodes of G, and contains no cycles *of edges or arcs*. Note that T does contain cycles of darts, but we regard T primarily as a set of edges, disregarding orientation. We say that a dart $d = a^\sigma$ is a tree dart (also, a dart of T, in T, or belongs to T) if $a \in T$. Otherwise, d is called a *non-tree dart*. For vertices u and v, $T[u, v]$ denotes the unique simple u-to-v path through T. Given a vertex designated as the *root* of T, $T[u]$ denotes the u-to-root path in T.

For a spanning tree T of G and a non-tree dart d, the *fundamental cycle of T with respect to d* is the cycle composed of d and the unique head(d)-to-tail(d) path in T. For a rooted spanning tree T, a non-tree dart d is called a *back edge* of T if head(d) is and ancestor in T of tail(d).

A *cut* X is a partition of V into two disjoint subsets $(X, V - X)$. The sets X and $V - X$ are sometimes referred to as the sides of the cut. The darts crossing the cut X are those with one

endpoint in X and the other not in X. This set of darts is denoted $\Gamma(X)$. Similarly, the set of darts leaving the cut is $\Gamma^+(X) = \{\text{dart } d : \text{tail}(d) \in X \text{ and head}(d) \notin X\}$, and the set of darts entering the cut is $\Gamma^-(X) = \{\text{dart } d : \text{head}(d) \in X \text{ and tail}(d) \notin X\}$. A cut X is called simple if both $G - X$ and $G - (V - X)$ are connected.

1.2 Planar Graphs

1.2.1 Embeddings

There are several equivalent ways to define planar graphs. The most intuitive one is the geometric definition. A graph is planar if it can be drawn (embedded) on the plane (or, equivalently, on the surface of the sphere) such that each node v is mapped to a distinct point, each arc uv is mapped to a continuous curve segment whose endpoints are the points corresponding to u and to v, and such that curves may only intersect at their endpoints. A plane graph is a planar graph equipped with an embedding.

Kuratowski [Kur30] was the first to give a combinatorial characterization of planar graphs as the family of graphs that do not contain $K_{3,3}$ nor K_5 as a minor (see Section 1.2.5 for a formal definition of minors). Here, $K_{3,3}$ is the complete bipartite graph with three nodes on each side of the bipartition, and K_5 is the complete graph on 5 nodes. Whitney [Whi33] gave an alternative definition in terms of combinatorial duals. MacLane [Mac37] gave another formulation in terms of the existence of a particular basis of the cycle space.

Consider the set of points in the plane that are not assigned to any node or arc of G. The connected components of this set are called the set F of *faces* of G. The unbounded component is called the *infinite face* of G, often denoted f_∞. Each face is bounded by a cycle of edges (not necessarily simple). We associate a face with the clockwise cycle of darts that forms its boundary.

Planar graphs can also be defined in terms of a combinatorial embedding using a rotation system [Hef91, Edm60, You63]. A rotation system is a permutation π over the darts of G. The nodes of an embedded graph $G = \langle \pi, A \rangle$ are the orbits (cycles) of π. The correspondence to a geometric embedding is that the darts in an orbit v of π correspond to the darts whose head is the node v in the geometric embedding which are listed, by convention, in counterclockwise order, see Figure 1.1.

We can also define the faces in terms of the rotation system π. Consider the permutation $\pi^* = \text{rev} \circ \pi$. The faces of G are defined to be the orbits of π^*. The choice of the infinite face in a combinatorial embedding is arbitrary. Note that there is a slight difference between the geometric and combinatorial definitions of the infinite face in graphs with multiple connected components. In the combinatorial definition, there is an infinite face for each connected component, while from the geometric definition, there is only one unbounded component. We will not encounter this issue because we will mostly be concerned with connected graphs, but in general, considering just a single infinite face may lead to problems. Another difference between the geometric and combinatorial definition is that the geometric definition allows for isolated vertices, whereas the combinatorial

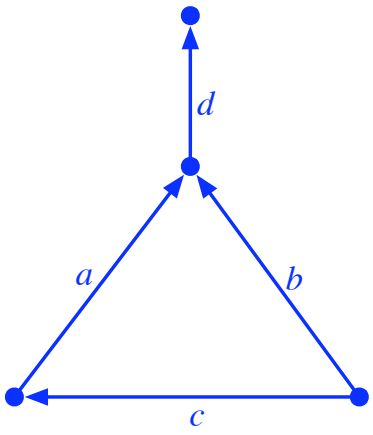

Figure 1.1: Illustration of the relation between the geometric and combinatorial embeddings of a graph. The combinatorial embedding is $\pi = (a^+b^+d^-)(a^-c^+)(c^-b^-)(d^+)$. Note the correspondence between the orbits of π and the darts oriented towards the nodes in the geometric embedding, listed in counterclockwise order. The permutation π^* is $\pi^* = \text{rev} \circ \pi = (a^+b^-c^+)(a^-c^-b^+d^+d^-)$. Note the correspondence of the orbits of π^* and the faces in the geometric embedding.
The darts of orbits that corresponds to bounded faces in the geometric embedding (in this example there is only one such orbit, $(a^+b^-c^+)$) form clockwise cycles in the geometric embedding.

definition does not.

Connected embedded graphs satisfy Euler's formula:

$$|V| - |A| + |F| = 2 - 2g,$$

where g is called the *genus* of the graph. Planar graphs are embedded graphs with genus zero.

A planar graph is *triangulated* if every face consists of exactly 3 darts. Any planar graph can be triangulated by introducing additional edges. This procedure can be easily implemented in linear time.

An easy consequence of Euler's formula is that planar graphs are sparse. More precisely, for a planar graph in which every face has size at least three (i.e., excluding parallel edges and self-loops), $|A| \leq 3|V| - 6$. Therefore, $|A| = O(|V|)$, so it suffices to specify the asymptotic dependency of, say, the running time of an algorithm on an input planar graph G is just in terms of $|V|$. For convenience we use $O(|G|)$ for this purpose.

The *face-vertex incidence graph* (also called the *radial* graph) of an embedded graph G with node set V and face set F is the graph $R(G)$ whose node set is $V \cup F$, such that there is an edge vf in $R(G)$ if and only if $v \in V$ and $f \in F$ and the node v is incident to the face f in G.

1.2.2 Vector Spaces and Circulations

Let $G = \langle V, A \rangle$ be a directed graph. The *dart space* of G is $\mathbb{R}^{A \times \{\pm 1\}}$, the set of vectors $\boldsymbol{\alpha}$ that assign a real number $\boldsymbol{\alpha}[d]$ to each dart d.

The *arc space* of G is the subspace of G's dart space satisfying antisymmetry with respect to rev. Namely, for every dart d, $\boldsymbol{\alpha}[d] = -\boldsymbol{\alpha}[\text{rev}(d)]$. We call the vectors of the dart and arc spaces *dart vectors* and *arc vectors*, respectively. For a dart d we define $\boldsymbol{\eta}(d)$ to be the arc vector such that

$$\boldsymbol{\eta}(d)[d'] = \begin{cases} 1 & \text{if } d' = d, \\ -1 & \text{if } d' = \text{rev}(d), \\ 0 & \text{otherwise.} \end{cases}$$

We extend this definition to any set S of darts by $\boldsymbol{\eta}(S) = \sum_{d \in S} \boldsymbol{\eta}(d)$. The vectors $\{\boldsymbol{\eta}(a) : a \in A\}$ form a basis of the arc space.

The *cycle space* of G is the subspace of the arc space spanned by

$$\mathcal{C} = \{\boldsymbol{\eta}(C) \ : \ C \text{ a cycle of darts in } G\}.$$

We refer to vectors of the cycle space as *circulations*. If G is planar then $\{\boldsymbol{\eta}(f) : f \text{ a face of } G, f \neq f_\infty\}$ is a basis of the cycle space. Therefore, any circulation $\boldsymbol{\rho}$ can be expressed as a linear combination

$$\boldsymbol{\rho} = \sum_{f \neq f_\infty} \boldsymbol{\phi}_{\boldsymbol{\rho}}[f]\boldsymbol{\eta}(f). \tag{1.1}$$

The coefficients $\boldsymbol{\phi}_{\boldsymbol{\rho}}$ are called the *face potentials associated with* $\boldsymbol{\rho}$. More generally, a *potential function* is a function assigning real numbers to the faces of G. We adopt the convention that $\boldsymbol{\phi}[f_\infty] = 0$.

1.2.3 Clockwise, Left and Right

A circulation $\boldsymbol{\rho}$ is *clockwise* if $\boldsymbol{\phi}_{\boldsymbol{\rho}}[f] \geq 0$ for all faces f. A circulation $\boldsymbol{\rho}$ is *counterclockwise* if $\boldsymbol{\phi}_{\boldsymbol{\rho}}[f] \leq 0$ for all faces f. Note that a circulation may be neither clockwise nor counterclockwise. Let C be a cycle of darts in G. We say that C is (counter) clockwise if the circulation $\boldsymbol{\eta}(C)$ is (counter) clockwise. See Figure 1.1 for an illustration of the geometric intuition for this definition.

Let P and Q be two u-to-v paths of darts in G. We say that Q *is left of* P if the cycle $Q \circ \text{rev}(P)$ is clockwise. We say that Q *is right of* P if the P is left of Q.

We say a face f is *external* to a circulation $\boldsymbol{\rho}$ if $\boldsymbol{\phi}_{\boldsymbol{\rho}}[f] = 0$, and is *internal* otherwise. We say that a dart d is external if the faces to d's left and right are external, and we say d is internal if the faces to d's left and right are internal. We say a dart is *belongs* to $\boldsymbol{\rho}$ if the dart is neither internal nor external to $\boldsymbol{\rho}$. A dart and its reverse have the same property (external, internal, or contained) with respect to a circulation. We say a vertex v is external if every dart incident to v is external, and is internal if every dart incident to v is internal.

For a cycle C, we say C *strictly encloses* a face (dart or vertex, resp.) if the face (dart or vertex, resp.) is internal to the circulation $\boldsymbol{\eta}(C)$. A dart or vertex is *enclosed* if it is strictly enclosed by $\boldsymbol{\eta}(C)$ or belongs to $\boldsymbol{\eta}(C)$. For a simple cycle C with a geometric embedding, this definition corresponds to the notion of a face, dart, or vertex being embedded inside C, with respect to f_∞. We define the *interior* of C to be the subgraph induced by the nodes enclosed by C. We define the *exterior* of C

to be the subgraph induced by the nodes not strictly enclosed by C. Note that, according to this definitions, C is considered to be both in its interior and in its exterior.

Let $G = \langle V, A \rangle$ be a planar embedded graph. Let d_1, d_2, d_3 be darts such that head$(d_1) =$ tail$(d_2) =$ tail(d_3). We say that d_2 is *right of* d_3 *w.r.t.* d_1 if d_3 occurs strictly between d_2 and rev(d_1) in counterclockwise order. For a simple path P containing d_3, we say dart d_2 *emanates right of* P if (a) the dart preceding d_3 in P is d_1, and d_2 is right of d_3 with respect to d_1, or (b) d_3 is the first dart of P, and tail(d_3) is on the boundary of f_∞, and d_2 is right of d_3 with respect to an imaginary edge that enters tail(d_3) from within the infinite face. We say that d *enters* P *from the right* if rev(d) emanates right of P. We extend these definitions to paths and say, e.g., that a path Q emanates right of path P at v if there is a dart d of Q that emanates right of P and tail$(d) = v$.

For two paths P and Q, we say that P *crosses* Q from right to left with entry node u and exit node v if P and Q have a common u-to-v subpath R (R may consist of a single vertex when u=v) such that P enters Q at u from the right and P emanates left of Q at v. This definition can be naturally extended to curves in the plane.

1.2.4 The Planar Dual

The *dual* of an embedded graph $G = \langle \pi, A \rangle$ is the embedded graph $G^* = \langle \pi^*, A \rangle$. G is called the primal graph. Since rev $\circ$ rev is the identity, $(\pi^*)^* =$ rev $\circ$ (rev $\circ\, \pi) = \pi$, which shows that the dual of the dual is the primal. If G is a planar connected graph, then so is G^*. By definition, the faces of G are the nodes of G^*, and vice-versa. Note that according to our definition the set of darts of G and of G^* is identical. Whenever it is not clear from the context we will explicitly specify whether we refer to a dart d in the primal graph G or in its dual G^*.

To define the geometric dual, embed a dual node f^* inside each primal face f. If f and g are the faces to the left and right of arc a in the primal, then the arc a^* is oriented from f^* to g^*. This roughly corresponds to rotating each arc by 90 degrees clockwise. See Figure 1.2. Note, however, that when using this geometric embedding of the dual on top of the primal embedding, the orientations in the dual are inverted. For example, in Figure 1.2, the dual node f corresponds to the orbit $(a^+ b^- c^+)$ of π^*. In the figure, this corresponds to listing the darts whose head is f in clockwise order, whereas the convention we use for the relation between the geometric and combinatorial embedding is listing the darts in counterclockwise order. Note that the combinatorial definitions of clockwise, left and right in the primal and in the dual are self consistent. Therefore, when overlaying the geometric embedding of the dual on top of that of the primal in this way, we will use the inverted directions and orientations; a dual cycle that *appears to be* clockwise in such a drawing will be said to be counterclockwise, which is consistent with our combinatorial definitions.

One of the basic properties we will use in our maximum-flow algorithms is the duality of cuts and cycles. See Figure 1.3.

Lemma 1.2.1 (Cycle-Cut duality [Whi33])**.** *A set of darts forms a simple cut in G iff it forms a simple cycle in the dual G^**

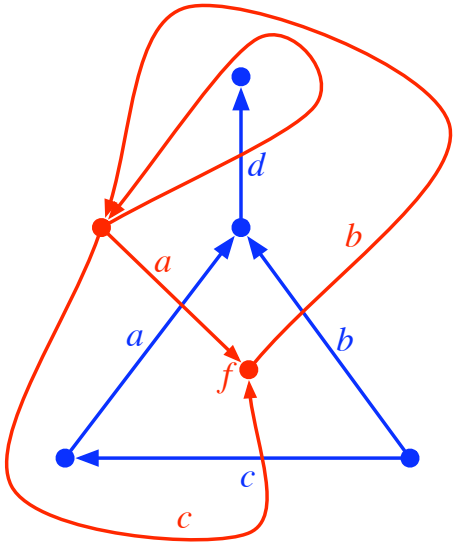

Figure 1.2: Illustration of the relation between the geometric and combinatorial embeddings of a primal planar graph (in blue) and its dual (in red).
The combinatorial embedding of the primal is $\pi = (a^+b^+d^-)(a^-c^+)(c^-b^-)(d^+)$.
The combinatorial embedding of the dual is $\pi^* = \text{rev} \circ \pi = (a^+b^-c^+)(a^-c^-b^+d^+d^-)$.
The orbit that corresponds to the dual node f is $(a^+b^-c^+)$. Note that the dual (red) darts in this orbit are listed in what appears to be clockwise order rather than in our counterclockwise convention. The geometric embedding used in this figure is convenient because it overlays the dual on top of the primal, but the orientations of the dual graph are inverted.

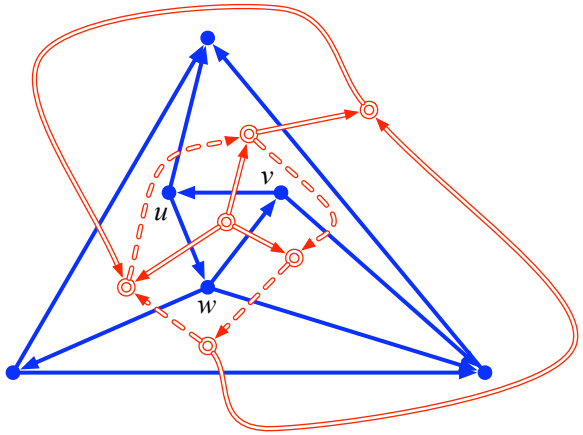

Figure 1.3: A primal graph (blue) and its dual (red). The cut $\Gamma^+(\{u, v, w\})$ in the primal corresponds to a counterclockwise dual cycle (dashed red arcs). Note that the dashed red cycle appears to be clockwise, but since the orientations of the dual are inverted, this is a counterclockwise dual cycle.

Another well-known relation between the primal and the dual is the duality of spanning trees. The idea can be traced back to a 1847 proof of Euler's theorem due to von Staudt [vS47, Som29]. See also [EIT+90].

Lemma 1.2.2. *Let T be a spanning tree of G. The set of edges not in T $\{a : a \notin T\}$ forms a spanning tree T^* of G^*, the dual of G.*

In some of our algorithms we will refer to T and T^* in Lemma 1.2.2 as the primal and dual trees.

1.2.5 Deletion and Contraction

In our algorithms we will use deletions of edges and contractions of (non-self-loop) edges. Intuitively, the result of contracting an edge uv in G is the graph G' in which the nodes u and v are identified with a single node. Formally, deleting an edge e is defined as deleting both darts of e. Deleting a dart $\hat{d}$ from a permutation π of a dart set D is obtaining the permutation π' of $D - \{\hat{d}\}$ defined as follows.

$$\pi'[d] = \begin{cases} \pi[\pi[d]] & \text{if } \pi[d] = \hat{d} \\ \pi(d) & \text{otherwise} \end{cases}$$

The graph G' obtained by contracting a non-self loop e of G is the dual of the graph obtained by deleting e from G^*.

We say that a graph G' is a *minor* of G if G' can be obtained from G by edge deletions and contractions.

1.2.6 Separators in Planar Graphs

Separators are the basis for most divide-and-conquer algorithms in planar graphs. In this method the graph is divided into two or more subgraphs. The problem is first solved recursively in each subgraph. Then the solutions are combined into a solution for the entire graph. The best known separator theorem for planar graphs was given by Lipton and Tarjan [LT79]. They proved that for any planar graph with n nodes, the vertex set can be partitioned into three sets A, B, C such that no edge connects A and B, neither A nor B contains more than $2n/3$ nodes, and C contains at most $2\sqrt{2n}$ nodes. The nodes of C are referred to in the literature as *separator, border* or *boundary* nodes. The result can be stated in terms of weights, rather than number of nodes.

Cycle Separators

In this thesis we will use a stronger result due to Miller [Mil86].

Theorem 1.2.3 ([Mil86])**.** *There exists a linear-time algorithm that, for any 2-connected triangulated planar graph with weights on the nodes, edges and faces that sum to 1, and no face has weight $> 2/3$, finds a simple cycle with at most $2\sqrt{2n}$ nodes such that the total weight strictly enclosed by C is at most $2/3$ and the total weight not enclosed by C is at most $2/3$.*

The cycle C is called a simple cycle separator.

For an n-node planar embedded graph G that is not necessarily triangulated or two-connected, we define a *Jordan separator* to be a non-self-intersecting curve C that intersects the embedding of the graph only at nodes such that at most $2/3$ of the weight is strictly enclosed by the curve and at most $2/3$ of the total weight is not enclosed. The nodes intersected by the curve are called *boundary nodes*. To find a Jordan separator with at most $2\sqrt{2}\sqrt{n}$ boundary nodes, add zero-weight artificial edges to triangulate the graph and make it two-connected. Next, apply Miller's cycle separator algorithm. The cycle separator produced visits at most $2\sqrt{2}\sqrt{n}$ nodes. Replace each edge of the cycle with a curve between its endpoint that is embedded in one of the faces adjacent to that edge. This can always be done in a way that preserves the balance, provided that no single edge weighs more than $1/3$ of the total weight.

Consider the subgraph G' of G induced by the interior of a Jordan or cycle separator C. The nodes of C all lie on a single face of G'. The same is true for the subgraph induced by the exterior of C.

r–divisions

We next consider a generalization of separators. Define a *region* R of G to be the subgraph of G induced by a subset V_R of V. We sometimes use the term *piece* instead of region. The two terms have the same meaning. In G, the vertices of V_R adjacent to vertices in $V - V_R$ are called *boundary vertices* (of R) and the set of boundary vertices of R is called the *boundary* of R. Vertices of V_R that are not boundary vertices of R are called *interior vertices* (of R).

An r–*division* [Fre87] of G is a partition of G into $O(n/r)$ edge-disjoint regions, each with $O(r)$ nodes and $O(\sqrt{r})$ boundary nodes. Fredrickson showed how to construct an r–division in $O(n \log r + nr^{-1/2} \log n)$ time by recursively applying the separator theorem of Lipton and Tarjan.

Define the holes of a region R to be the faces of R that are not faces of G. Note that every boundary node of R is incident to some hole of R. In our algorithms we require a variant of r–division with the additional property that every region has a constant number of holes. We call such an r–division an r–division with a constant number of holes. Such a division can be obtained within the same time bounds by modifying Fredrickson's construction in two ways. First use Miller's cycle separator instead of Lipton and Tarjan's. Second, when finding the a cycle separator, alternate between separating according to the number of nodes and separating according to the number of holes. This technique was first used by Klein and Subramanian [KS98, Sub95]. See [INSWN11, WN10b] for a more detailed treatment.

1.3 Lengths and Shortest Paths

Graphs can be augmented with different kinds of attributes. One of the most common attributes is lengths (sometimes called weights). A length assignment is a function length$(\cdot)$ from the set of darts to the real numbers. Very often we talk of arc lengths, in which case the length of the dart a^+ is length(a) and the length of a^- is infinite. In general, length assignments may be negative, although in various circumstances only non-negative lengths are considered. When no confusion arises we may refer to lengths of the corresponding arcs instead of the darts. We extend the notation to sets of darts (and to paths in particular). For a set D of darts, the length (or weight) of D is length$(D) = \sum_{d \in D}$ length(d). For two nodes u, v, the distance from u to v is the length of a shortest u-to-v path in the graph. The *single-source shortest-path problem* is to find, for a specified node s, the distances from s to all nodes in the graph as well as the actual paths realizing these distances. In directed graphs with no negative length cycles, these paths can be always be chosen so that they form a tree rooted at s which is called a *shortest-path tree*.

1.3.1 Dijkstra's Algorithm

Dijkstra's algorithm is the common name for an algorithm for computing single source shortest paths in a directed graph with non-negative arc lengths. It is named after E. J. Dijkstra [Dij59], although several very similar algorithms were independently discovered and published in the second half of the 1950's by Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, and Seitz [LGJ+57] and by Dantzig [Dan58]. See Schrijver [Sch05] for a historical survey.

The pseudocode we present is slightly different than the usual one, to make it suitable for the efficient implementation discussed in Section 1.3.7 and in Chapter 3.

The procedure UPDATEHEAP$(\mathcal{Q}, s, d)$ decreases the key of element s in the heap $\mathcal{Q}$ to be d (or inserts the element if it is not already in the heap). The procedure EXTRACTMIN$(\mathcal{Q})$ returns the

Algorithm 1.1 Dijkstra(G, s)

1: for all $v \in V(G)$: $d(v) \leftarrow \infty$
2: $d(s) \leftarrow 0$
3: initialize an empty heap $\mathcal{Q}$
4: UPDATEHEAP$(\mathcal{Q}, s, d(s))$
5: $S \leftarrow \emptyset$
6: **while** $S \neq V(G)$
7: $\quad v \leftarrow$ EXTRACTMIN$(\mathcal{Q})$
8: $\quad$ ACTIVATE$(\mathcal{Q}, G, v, d)$
9: $\quad S \leftarrow S \cup \{v\}$

element with minimum key in the heap $\mathcal{Q}$.

For an arc vu, *relaxing* vu is an attempt to decrease the distance label of u by considering a path that gets to u through v using vu [For56]. The procedure ACTIVATE relaxes all the arcs whose tail is v.

Algorithm 1.2 Activate$(\mathcal{Q}, G, v, d)$

relaxes all the arcs in G whose tail is v in the heap $\mathcal{Q}$ using the labels $d(\cdot)$

1: **for** each arc $vu \in E(G)$
2: $\quad d(u) \leftarrow \min\{d(u), d(v) + \ell(vu)\}$
3: $\quad$ UPDATEHEAP$(\mathcal{Q}, u, d(u))$

In Dijkstra's algorithm each node v is extracted from the heap exactly once. At that time the arcs emanating from v are relaxed. It follows that each arc is relaxed exactly once. Dijkstra's algorithm runs in $O((n + m) \log n)$ time if elementary data structures such as a binary heap are used [Joh77], and in $O(n \log n + m)$ time when implemented with Fibonacci heaps [FT87]. It is easy to augment the algorithm to produce a shortest-path tree within the same running time.

1.3.2 The Bellman-Ford Algorithm

The Bellman-Ford algorithm is the common name for an algorithm for computing single-source shortest paths in directed graphs in the presence of negative lengths, but no negative-length cycles. The algorithm can be used to detect negative-length cycles. It is named after R. Bellman, who published the algorithm in 1958 [Bel58], and after L. R. Ford Jr., who set forward the framework of edge relaxations [For56]. Very similar algorithms were independently obtained and published by Shimbel [Shi55] and Moore [Moo57]. Again, see Schrijver [Sch05] for a detailed historical review.

The algorithm consists of phases. On each phase all the arcs of the graph are relaxed. It follows that at the end of the phase k, the distance label $d(v)$ stores the length of a shortest s-to-v path that consists of at most k arcs. Therefore, after at most n iterations, the distance labels store the true distances from s in the entire graph, and the running time is $O(mn)$. As with Dijkstra's algorithm,

Algorithm 1.3 Bellman-Ford(G, s)

```
1: for all v ∈ V(G): d(v) ← ∞
2: d(s) ← 0
3: repeat |V(G)| times
4:    for each arc uv
5:       d(v) ← min{d(v), d(u) + ℓ(uv)}
```

it is easy to augment the algorithm to produce a shortest-path tree within the same running time. In Chapter 2 we will present efficient implementations of the Bellman-Ford algorithm in a specific scenario that comes up in planar graphs.

1.3.3 Price Functions and Reduced Lengths

Let G be a directed graph with arc lengths $\ell(\cdot)$. A *price function*[2] is a function ϕ from the nodes of G to the reals. For a dart (or arc) d, the *reduced length with respect to* ϕ is $\ell_\phi(d) = \ell(d) + \phi(\text{tail}(d)) - \phi(\text{head}(d))$. A price function is called *feasible*[3] if it induces nonnegative reduced lengths on *all* darts of G.

Feasible price functions are useful in transforming a shortest-path problem involving positive and negative lengths into one involving only nonnegative lengths, which can then be solved using Dijkstra's algorithm. The best known use of this technique is probably Johnson's algorithm for all-pairs shortest paths [Joh77]. The idea appeared much earlier, for example in Ford and Fulkerson's algorithm for minimum-cost flow [FF57].

For any nodes s and t, for any s-to-t path P, the reduced length $\ell_\phi(P)$ of P is

$$\begin{aligned} \ell_\phi(P) &= \sum_{d \in P} \ell_\phi(d) + \phi(\text{tail}(d)) - \phi(\text{head}(d)) \\ &= \phi(\text{start}(P)) - \phi(\text{end}(P)) + \sum_{d \in P} \ell(d) \\ &= \ell(P) + \phi(s) - \phi(t). \end{aligned} \tag{1.2}$$

This shows that an s-to-t path is shortest with respect to $\ell_\phi(\cdot)$ iff it is shortest with respect to $\ell(\cdot)$. Moreover, the s-to-t distance with respect to the original lengths $\ell(\cdot)$ can be recovered by adding $\phi(t) - \phi(s)$ to the s-to-t distance with respect to $\ell_\phi(\cdot)$. Finally, we note that the reduced length of a cycle is equal to its length.

Suppose ϕ is a feasible price function. Running Dijkstra's algorithm with the reduced lengths and modifying the distances thereby computed to obtain distances with respect to the original lengths will be called *running Dijkstra's algorithm with* ϕ.

An example of a feasible price function comes from single-source distances. Suppose that, for some node r of G, for every node v of G, $\phi(v)$ is the r-to-v distance in G with respect to $\ell(\cdot)$. Then

[2] different names have been used for this term in the literature. In particular, Miller and Naor [MN95] used the term *potential*.

[3] Miller and Naor [MN95] use the term *consistent*.

for every arc uv, $\phi(v) \leq \phi(u) + \ell(uv)$, so $\ell_\phi(uv) \geq 0$. Here we assume, without loss of generality, that all distances are finite (i.e., that all nodes are reachable from r) since we can always add arcs with sufficiently large lengths to make all nodes reachable without essentially affecting the shortest paths in the graph.

Let T be a rooted spanning tree of G. The *reduced length of d with respect to T* are reduced lengths with respect to distances in T. Namely,

$$\text{length}_T(d) = \text{length}(d) + \text{length}(T[\text{tail}(d)]) - \text{length}(T[\text{head}(d)]) \tag{1.3}$$

1.3.4 Multiple-Source Shortest Paths

Klein [Kle05] gave a multiple-source shortest-path algorithm with the following properties. The input consists of a directed planar embedded graph G with non-negative arc-lengths, and a face f. For each node u in turn on the boundary of f, the algorithm computes (an implicit representation of) the shortest-path tree rooted at u. The basic algorithm takes a total of $O(n \log n)$ time and $O(n)$ space on an n-node input graph. In addition, given a set of pairs (u, v) of nodes of G where u is on the boundary of f, the algorithm computes the u-to-v distances. The time per distance computed is $O(\log n)$. In particular, given a set S of $O(\sqrt{n})$ nodes on the boundary of a single face, the algorithm can compute all S-to-S distances in $O(n \log n)$ time. We refer to this algorithm as the MSSP algorithm.

Using persistence techniques [DSST89], the MSSP algorithm can be turned into a $O(n \log n)$-space data structure, whose construction takes $O(n \log n)$ time, that can report the distance in G between any node on the distinguished face f and any other node in G in $O(\log n)$ time.

In fact, the MSSP algorithm does not require that the arc-lengths be nonnegative if the input also includes a table of distances to all nodes from some node on the face f. This observation follows from a careful inspection of the algorithm [Kle05] itself. Alternatively, it also follows from the price-function technique of Section 1.3.3; the table of distances can be used to obtain nonnegative reduced lengths, and these lengths can be supplied as input to Klein's algorithm.

We use Klein's MSSP as a black box in our shortest-path algorithms in Chapter 2. We extend it to report distances from all nodes of a cycle in Chapter 4 and generalize it to solve the maximum-flow problem with multiple sources and a single sink in Chapter 6. We do not provide here a complete description of the algorithm, but only include definitions that are used in Chapter 6.

We note that, when using MSSP as black box, one may also use the algorithm of Cabello and Chambers [CC07], which is considered simpler, provides the same interface, and runs in the same time bounds.

Leafmost and Right-Short

Let T be a spanning tree of a planar graph G. A dart d is *unrelaxed* if $\text{length}_T(d) < 0$. Note that, by definition, only darts not in T can be unrelaxed. Let T^* be the dual spanning tree (as in

Lemma 1.2.2). A *leafmost unrelaxed* dart is an unrelaxed dart d such that no proper descendant of d in T^* is unrelaxed.

Right-first search [RLWW95] is depth-first search on a planar graph, with the restriction that, for each node v visited, the edges vw out of v are explored in right-to-left order with respect to the the edge uv by which v was first visited (or, if v is the root and is on the boundary of the infinite face, with respect to an imaginary edge that enters v from within the infinite face). Right-first search induces a *right-first search tree* consisting of the set of edges explored by the search.

A tree T is *right-short* if for all nodes $v \in T$ there is no *simple* root-to-v path P that is as short as $T[v]$ and strictly right of $T[v]$. Clearly, a right-first search tree is right-short.

1.3.5 Dense Distance Graphs

Recall that for a subgraph (piece) P of a planar graph G, the boundary nodes ∂P of P are the nodes of P that are adjacent to nodes in $G - P$. The *dense distance graph* of P, which we sometime denote by DDG_P, is the complete graph on ∂P, such that the length of an arc corresponds to the distance (in P) between its endpoints. The dense distance graph for all pieces P in an r–division with a constant number of holes can be computed in $O(|V(G)| \log |V(G)|)$ time and space using Klein's MSSP; for each piece P, run the MSSP algorithm in $O(|P| \log |P|)$ time and space a constant number of times, specifying a different hole of P as the distinguished face at each run. These invocations of MSSP compute the distances between the boundary nodes of P in $O(|P| \log |P| + |\partial P|^2 \log |P|)$ time. Since in an r–division $|\partial P| = \sqrt{|P|}$, this is $O(|P| \log |P|)$ time and space per piece, for a total of $O(|V(G)| \log |V(G)|)$ for all pieces.

1.3.6 The Monge Property

Monge [Mon81] observed in 1781 that if unit quantities have to be transported from locations X and Y in the plane to locations Z and W (not necessarily respectively) in such a way as to minimize the total distance travelled, then the paths followed in transporting these quantities must not intersect. In 1961 Hoffman [Hof61] elaborated on this idea and coined the term *Monge property*. A matrix has the Monge property (also called a Monge matrix) if for every pair of rows with indices $i < j$ and every pair of columns with indices $k < \ell$ we have

$$M_{ik} + M_{j\ell} \leq M_{i\ell} + M_{jk} \tag{1.4}$$

When the inequality hods in the reverse direction the property is called the inverse Monge property (also called contra-Monge, anti-Monge or convex Monge). See [BKR96] for a survey of Monge matrices and their applications.

Monge matrices and the efficient algorithms described in the remainder of this section are used in the algorithms in Chapter 2. In the context of a planar graph G, let $v_1, v_2, \ldots, v_r$ be the nodes incident to some face of G in, say, clockwise order. Consider a partition of these nodes into two consecutive sets $A = \{v_1, v_2, \ldots v_q\}$ and $B = \{v_{q+1}, v_{q+2}, \ldots v_r\}$. Monge's observation implies that

for any $1 \leq i < j \leq q$ and $q < k \leq \ell \leq r$, a shortest v_i-to-v_k path and a shortest v_j-to-v_ℓ path must cross. Let M be a matrix such that M_{ik} is the distance between v_i and v_k. It easily follows that M is an inverse Monge matrix (see Lemma 2.1.4 in Chapter 2).

An immediate consequence of the inverse Monge property is that for any $i < j$, $k < \ell$ such that $M_{ik} \leq M_{i\ell}$, we also have $M_{jk} \leq M_{j\ell}$. A Matrix that satisfies this weaker property is called *totally monotone*. Totally Monotone matrices were defined and studied in a seminal paper by Aggarwal, Klawe, Moran, Shor and Wilber [AKM+86], where an algorithm is given to find all row-maxima of an n-by-m totally monotone matrix in $O(m+n)$ time. This algorithm is referred to in the literature by the nickname SMAWK. It is easy to see that by negating each element of a totally monotone matrix and reversing the order of its columns, SMAWK can be used to find the row-minima within the same time bound.

Since for an inverse Monge matrix M, both M and its transpose are totally monotone, SMAWK can also be used to find the *column* minima and maxima of an inverse Monge matrix (the same is true for Monge matrices).

The SMAWK algorithm was generalized to totally monotone partial matrices. A *falling staircase matrix* is defined in [AK90] and [KK90] to be a lower triangular fragment of a totally monotone matrix. More precisely, $(M, \{f(i)\}_{0 \leq i \leq n+1})$ is an $n \times m$ falling staircase matrix if

1. for $i = 0, \ldots, n+1$, $f(i)$ is an integer with $0 = f(0) < f(1) \leq f(2) \leq \cdots \leq f(n) < f(n+1) = m+1$.
2. M_{ij}, is a real number if and only if $1 \leq i \leq n$ and $1 \leq j \leq f(i)$. Otherwise, M_{ij} is blank.
3. (total monotonicity) for $i < k$ and $j < l \leq f(i)$, and $M_{ij} \leq M_{il}$, we have $M_{kj} \leq M_{kl}$.

Finding the row-maxima in a falling staircase matrix can easily be done using SMAWK in $O(n+m)$ time after replacing the blanks with sufficiently small numbers so that the resulting matrix is totally monotone. However, this trick does not work for finding the row-minima. Aggarwal and Klawe [AK90] gave an $O(m \log\log n)$ time algorithm for finding row-minima in falling staircase matrices of size $n \times m$. Klawe and Kleitman gave in [KK90] a more complicated algorithm that computes the row-minima of an $n \times m$ falling staircase matrix in $O(m\alpha(n)+n)$ time, where $\alpha(n)$ is the inverse Ackerman function (cf. [Tar75]). If M satisfies the above conditions with total monotonicity replaced by the inverse Monge property then M and the matrix obtained by transposing M and reversing the order of rows and of columns are falling staircase. In this case both algorithms can be used to find the column-minima as well as the row-minima of M.

1.3.7 Fakcharoenphol and Rao's Efficient Dijkstra Implementation

In this section we describe a data structure due to Fakcharoenphol and Rao [FR06] that efficiently implements Dijkstra's algorithm in specific scenarios. Our description somewhat deviates from the original one in [FR06].

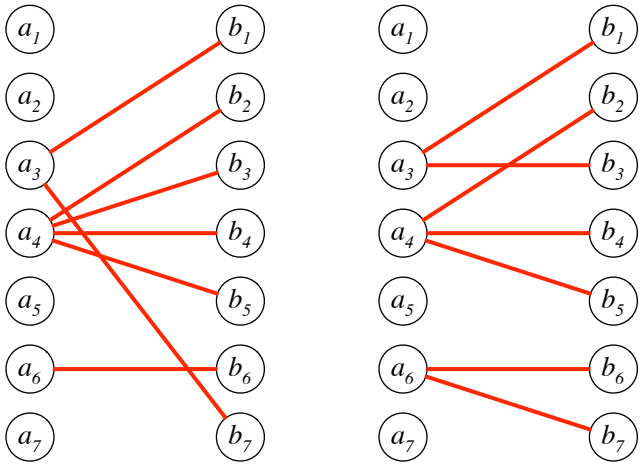

Figure 1.4: Schematic examples for sets with interleaving parent relationship (right) and non-interleaving parent relationship (left). The sets A and B are shown. An edge between a node a_i and a node b_j indicates that $p(b_j) = a_i$. Namely, that a_i is the parent of b_j. The parent relationship on the left is non-interleaving since no edges cross when the nodes of B are shifted cyclically one position down.

Non-interleaving parent relationship

Let $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_m$ be two sequences of nodes. Let $H = \langle A \cup B, A \times B \rangle$ be a complete directed bipartite graph with arc lengths M_{ab} (all arcs are directed from A to B). Given real labels $d(\cdot)$ for the nodes of A, the label of each node $b \in B$ is defined to be

$$d(b) = \min_{a \in A} d(a) + M_{ab}. \tag{1.5}$$

The *parent* $p(b)$ of a node $b \in B$ is the node $a \in A$ with minimum index that achieves the minimum in (1.5).

We consider arc lengths M that have the following property. For every choice of labels $d(\cdot)$ for the nodes of A, there exists a cyclic shift of the order of nodes of B such that for every $k < \ell$, $p(b_k) \leq p(b_\ell)$. We refer to this property as the *non-interleaving of the parent relationship*. See Figure 1.4 for an example.

Before we continue with the description of the data structure, we give two examples where the parent relationship is indeed non-interleaving. The first situation is when the length matrix M is Monge. Suppose $p(b_\ell) = a_i$ then, for $i < j$,

$$d(a_i) + M_{i\ell} \leq d(a_j) + M_{j\ell}.$$

By the Monge property, for $i < j, k < \ell$, $M_{ik} + M_{j\ell} \leq M_{i\ell} + M_{jk}$. Hence,

$$d(a_i) + M_{ik} + d(a_j) + M_{j\ell} \leq d(a_i) + M_{i\ell} + d(a_j) + M_{jk}.$$

Combining the two inequalities we get

$$d(a_i) + M_{ik} + d(a_j) + M_{j\ell} \leq d(a_j) + M_{j\ell} + d(a_j) + M_{jk},$$

or equivalently,

$$d(a_i) + M_{ik} \leq d(a_j) + M_{jk}.$$

This shows that $p(b_k) \leq i = p(b_\ell)$. As we had noted in section 1.3.6, Monge distance matrices arise when the sets A and B correspond to a partition of the nodes incident to a single face of a planar graph into two consecutive sets. Another situation where the parent relation is non-interleaving is when the sets A and B correspond to nodes around two distinct faces of an underlying planar graph G. In this case the distance matrix M is not Monge. However, it is not difficult to see that the parent relationship is non-interleaving.[4] Informally, the reason is that, if the parent relationship is interleaving, then there are corresponding shortest paths in the underlying planar graph G that must cross. This leads to a contradiction. Both situations are encountered when separating a graph into pieces using cycle separators (individually, or in an r–division).

Monge Heaps

We now proceed with describing the data structure. A range-minimum-query data structure [GBT84, BV93, BFC00] for a row i of M is a data structure that, given a range $[j_1, j_2]$ of columns, returns $\min_{j_1 \leq j \leq j_2} M_{ij}$.

Recall that in Dijkstra's algorithm, the estimates for the label of each node are initialized to infinity. The label of each node is fixed when the node is extracted as the minimum element in the heap. At that time, all incident arcs are relaxed, thus updating the estimates for the labels of other nodes. It is important to note that the labels set by Dijkstra's algorithm are monotonically increasing. After a node is extracted from the heap with label d, all subsequent labels that will be set by the algorithm will be at least d.

Fakcharoenphol and Rao [FR06, Section 4.1] described a data structure that, given a complete bipartite graph with edge-length matrix M with the non-interleaving parent relationship and a range-minimum data structure with running time $O(T_{RMQ})$ for each row of M, supports the following operations, which are useful in implementing Dijkstra's algorithm:

1. FR-ACTIVATE(a, d)[5] - Sets the label of node $a \in A$ to be d, and implicitly relaxes all arcs incident to a. This operation may be called at most once per node, and runs in $O(\log |A| + T_{RMQ})$ amortized time.

2. FR-FINDMIN - Returns the node v in B with minimum label. Labels of nodes in A that were not yet activated are assumed to be infinite. This operation also returns the label d of v and takes $O(1)$ time.

[4] In Lemma 2.2.1 we shall see a way to represent such a matrix M by two Monge matrices in a way that preserves distances.

[5] In [FR06] this procedure is called ACTIVATELEFT.

3. FR-EXTRACTMIN - Returns the node v in B with minimum label and removes it from the set B. This operation also returns the label d of v and takes $O(\log|B| + T_{RMQ})$ time. The integrity of the data structure is only guaranteed if all subsequent FR-ACTIVATE operations have labels at least d.

Fakcharoenphol and Rao call this data structure *an online bipartite Monge searching data structure*. We will refer to it as a *Monge heap*. It is noted in [FR06] that a range-minimum data structure that answers queries in $T_{RMQ} = O(\log|A|)$ can be constructed for each row of the input matrix in linear time (at the time the matrix itself is constructed). In fact, range-minimum data structures with constant-time query and linear-time construction are also known. See [BFC00] and the references therein.

Observe that Dijkstra's algorithm implements these procedures in a straightforward manner using a heap. The difference is that Dijkstra's algorithm assume neither bipartite graphs nor the non-interleaving property of the parent relationship. On the other hand, Dijkstra's ACTIVATE (Algorithm 1.2) explicitly relaxes all the arcs incident to a, which takes $O(|A|)$ time, not $O(\log|A|)$ time.

FR-Dijkstra

Monge heaps can be used to implement Dijkstra's algorithm in the following setting. Let $\{P_i\}_{i=1,2,\ldots}$ be a set of (not necessarily disjoint) subgraphs of a planar embedded graph. Let X_i be a set of nodes on a constant h number of faces of P_i. Let K_i be the complete graph on X_i such that the length of an arc uv in K_i corresponds to the u-to-v distance in P_i. In typical applications, the P_i's are pieces in an r–division with a constant number of holes, the nodes X_i are the boundary nodes of P_i, and the graphs K_i are the dense distance graphs of the pieces P_i.

Each complete graph K_i can be decomposed into complete bipartite subgraphs in which the parent relationship is non-interleaving, such that each node of X_i appears in $O(\log|X_i|)$ of these subgraphs. The decomposition is as follows. Refer to Figure 1.5. There are $O(h^2)$ pairs of faces to which nodes of X_i belong. For each such pair f_1, f_2, let A and B be the set of nodes of X_i on f_1 and f_2, respectively. If $f_1 \neq f_2$ then the complete bipartite graph on A and B is in the decomposition since it satisfies the non-interleaving property. Otherwise, $A = B$ and we further decompose it to satisfy the non-interleaving property. List the nodes of A according to their clockwise order on f_1. Split A into two consecutive halves A' and B' and add the complete bipartite graph on A' and B' to the decomposition. Next, recurse on A' and on B'. Since A' and B' are disjoint, and their size is reduced by a factor of two at each level of the recursion, each node of X_i appears in $O(h^2 \log|A|) = O(\log|X_i|)$ bipartite subgraphs.

Let $\mathcal{G} = \{G_j\}$ denote the set of all bipartite graphs in the decompositions of all the K_i's. Note that $\bigcup_i K_i = \bigcup_j G_j$, but that $|\mathcal{G}|$ is $O(\sum_i |X_i|)$ since each graph K_i is decomposed into $O(|X_i|)$ bipartite subgraphs. Fakcharoenphol and Rao's algorithm takes as input the set $\mathcal{G}$ of bipartite graphs as well as a range-minimum data structure for each row of the distance matrix for each G_j. Given a

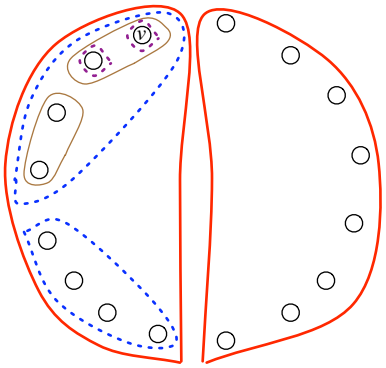

Figure 1.5: Decomposition of a complete graph over a set of nodes on a single face into bipartite complete graphs such that each node appears in a logarithmic number of bipartite graphs. The bipartitions in all the bipartite graphs in which the node v appears are indicated.

node s in $\bigcup K_i$, it computes a shortest-path tree in $\bigcup K_i$ rooted at s. We will refer to Fakcharoenphol and Rao's algorithm as FR-DIJKSTRA. The running time is $O\left(\sum_i |X_i| \log |X_i| \log \left(\sum |X_i|\right)\right)$, which is $O\left(\sum_i |X_i| \log^2 \left(\sum |X_i|\right)\right)$.

It may seem curious that, although we are interested in computing a shortest-path tree in $\bigcup K_i$, our description of FR-DIJKSTRA exposes the decomposition of the graphs $\{K_i\}$ into bipartite graphs $\{G_j\}$ that obey the non-interleaving property of the parent relationship. Had we not exposed this decomposition, the implementation of FR-DIJKSTRA would have had to compute the range-minimum data structures for each G_j internally. This task seems to require time that is linear in the number of entries in the distance matrix of each G_j, which may be $\Theta(|X_i|^2)$. This computation would then dominate the running time of FR-DIJKSTRA. Instead, we choose to expose the decomposition, and assume that the range-minimum data structures are computed at the time the graphs $\{K_i\}$ and $\{G_j\}$ are computed. In Section 3 we will show how to avoid this obstacle, so that the required range-minimum data structures are constructed within the $O\left(\sum_i |X_i| \log^2 \left(\sum |X_i|\right)\right)$ time bound. This will be useful for further generalizing FR-DIJKSTRA to work with dynamically changing reduced lengths, rather than with a fixed set of lengths.

Recall that Dijkstra's algorithm maintains a heap with distance labels for all nodes. At each step, the node u with minimum label is extracted from the heap since its distance label is guaranteed to be correct, and the arcs leaving u are relaxed. The key idea in FR-DIJKSTRA is that the edges of each bipartite subgraph can be implicitly relaxed efficiently using a dedicated Monge heap $\mathcal{M}_i$ for each bipartite subgraph G_i. The minimum elements from each Monge heap $\mathcal{M}_i$ are maintained in a regular global heap $\mathcal{Q}$ from which the node with global minimum label is extracted in each iteration of the main loop. Note that since a node v may appear in multiple G_i's, v may be extracted as the minimum element of the global heap $\mathcal{Q}$ multiple times, once for each Monge heap it appears in. However, the label $d(v)$ of v is finalized at the first time v is the minimum element of $\mathcal{Q}$. The algorithm activates v using FR-ACTIVATE only at that time. At all subsequent occurrences the algorithm

just extracts v from the corresponding Monge heap, and updates the representative of that Monge heap in $\mathcal{Q}$.

Algorithm 1.4 FR-DIJKSTRA$(\mathcal{G}, s)$

Input: a set $\mathcal{G} = \{G_j\}$, where each G_j is a complete bipartite graph on $V(G_j) = A_j \cup B_j$ with a non-interleaving parent relationship. G_j is represented by its arc length matrix M_j, accompanied by a range-minimum data structure for each row of M_j.
a node s in $\bigcup G_j$.
Output: the distances $d(\cdot)$ from s in $\bigcup G_j$.

```
1: initialize a Monge heap M_j for each G_j ∈ G
2: for all v ∈ ⋃V(G_j): d(v) ← ∞
3: d(s) ← 0
4: for all G_j ∈ G s.t. s ∈ A_j: FR-ACTIVATE(M_j, s, d(s))
5: initialize an empty regular heap Q
6: for all G_j ∈ G: UPDATEHEAP(Q, M_j, FR-FINDMIN(M_j))
7: S ← {s}
8: while S ≠ ⋃V(G_j)
9:    M_ĵ, v, d_v ← EXTRACTMIN(Q)
10:   FR-EXTRACTMIN(M_ĵ)
11:   UPDATEHEAP(Q, M_ĵ, FR-FINDMIN(M_ĵ))
12:   if v ∉ S then
13:      d(v) ← d_v
14:      S ← S ∪ {v}
15:      for each G_j s.t. v ∈ A_j
16:         FR-ACTIVATE(M_j, v, d(v))
17:         UPDATEHEAP(Q, M_ĵ, FR-FINDMIN(M_j))
18: return d
```

The stated running time follows from the fact that FR-EXTRACTMIN and FR-ACTIVATE are called at most once for each node v in each bipartite subgraph v belongs to. Since a node of X_i belongs to $O(\log |X_i|)$ bipartite subgraphs, the number of iterations is $O(\sum_i |X_i| \log |X_i|)$. The running time of each iteration is dominated by that of EXTRACTMIN, which is $O\left(\log\left(|\sum |X_i|\right)\right)$. (FR-EXTRACTMIN and FR-ACTIVATE take only $O(\log |X_i|)$ amortized time). Hence, the total running time is $O\left(\sum_i |X_i| \log |X_i| \log\left(|\sum |X_i|\right)\right)$.

A simple extension of FR-DIJKSTRA

We note that given a (not necessarily planar) graph $H = \langle V_H, E_H \rangle$, where V_H is possibly not disjoint from $\bigcup X_i$, FR-DIJKSTRA can be easily extended to compute a shortest-path tree in $H \cup \bigcup_j G_j$ in $O\left(\sum_i |X_i| \log |X_i| \log\left(|\sum |X_i|\right) + |E_H| \log\left(|V_H|\right)\right)$ time. This is done by relaxing the edges of H using a separate regular heap $\mathcal{Q}_H$ (rather than in a Monge heap). The edges of H are relaxed explicitly by calling ACTIVATE (rather than FR-ACTIVATE). The edges of each bipartite graph G_j are handled in a Monge heap $\mathcal{M}_j$, as before. The minimum element from each Monge heap $\mathcal{M}_j$ as well as the minimum element from the regular heap $\mathcal{Q}_H$ are maintained in a regular global heap $\mathcal{Q}$. This

use was first suggested in [BSWN10]. Since the details were not published, we provide the extended pseudocode for completeness (see Algorithm 1.5).

Algorithm 1.5 FR-DIJKSTRA$(\mathcal{G}, H, s)$

Input: a set $\mathcal{G} = \{G_j\}$, where each G_j is a complete bipartite graph on $V(G_j) = A_j \cup B_j$ with a non-interleaving parent relationship. G_j is represented by its arc length matrix M_j, accompanied by a range-minimum data-structure for each row of M_j.
a graph $H = \langle V_H, E_H \rangle$ represented by its arc lengths.
a node s in $\bigcup G_j$
Output: the distances $d(\cdot)$ from s in $H \cup \bigcup G_j$.

initialize a Monge heap $\mathcal{M}_j$ for each $G_j \in \mathcal{G}$
for all $v \in \bigcup V(G_j)$: $d(v) \leftarrow \infty$
$d(s) \leftarrow 0$
for all $G_j \in \mathcal{G}$ s.t. $s \in A_j$: FR-ACTIVATE$(\mathcal{M}_j, s, d(s))$
initialize an empty regular heap $\mathcal{Q}_H$
if $s \in V_H$ **then** ACTIVATE$(\mathcal{Q}_H, H, s, d)$
initialize an empty regular heap $\mathcal{Q}$
UPDATEHEAP$(\mathcal{Q}, \mathcal{Q}_H, \text{FINDMIN}(\mathcal{Q}_H))$
for all $G_j \in \mathcal{G}$: UPDATEHEAP$(\mathcal{Q}, \mathcal{M}_j, \text{FR-FINDMIN}(\mathcal{M}_j))$
$S \leftarrow \{s\}$
while $S \neq \bigcup V(G_j)$
  $\mathcal{M}_{\hat{j}}, v, d_v \leftarrow$ EXTRACTMIN$(\mathcal{Q})$
  if $\mathcal{M}_{\hat{j}} = \mathcal{Q}_H$ **then**
    EXTRACTMIN$(\mathcal{Q}_H)$
    UPDATEHEAP$(\mathcal{Q}, \mathcal{Q}_H, \text{FINDMIN}(\mathcal{Q}_H))$
  else
    FR-EXTRACTMIN$(\mathcal{M}_{\hat{j}})$
    UPDATEHEAP$(\mathcal{Q}, \mathcal{M}_{\hat{j}}, \text{FR-FINDMIN}(\mathcal{M}_{\hat{j}}))$
  if $v \notin S$ **then**
    $d(v) \leftarrow d_v$
    $S \leftarrow S \cup \{v\}$
    if $v \in V_H$ **then** ACTIVATE$(\mathcal{Q}_H, H, v, d)$
    UPDATEHEAP$(\mathcal{Q}, \mathcal{Q}_H, \text{FINDMIN}(\mathcal{Q}_H))$
    for each G_j s.t. $v \in A_j$
      FR-ACTIVATE$(\mathcal{M}_j, v, d(v))$
      UPDATEHEAP$(\mathcal{Q}, \mathcal{M}_{\hat{j}}, \text{FR-FINDMIN}(\mathcal{M}_j))$
return d

We already mentioned above that in the basic applications of FR-DIJKSTRA the subgraphs P_i are regions in an r–division with a constant number of holes. In Chapter 4 we use FR-DIJKSTRA with a more complicated set of subgraphs. In Chapter 3 we extend FR-DIJKSTRA to use reduced lengths, an extension that has uses in computing maximum flow (see Chapter 5) and in constructing dynamic distance oracles that can handle negative lengths [KMNS12].

1.3.8 External Dense Distance Graphs

For a graph G and a subgraph (piece) P of G, let $G-P$ be the graph obtained from G by deleting the nodes in $V_P - \partial P$. The *external* dense distance graph of P, denoted by DDG_{G-P}, is the complete graph on ∂P such that the length of an arc corresponds to the distance between its endpoints in $G - P$. External dense distance graphs were used recently in [BSWN10]. Computing the external dense distance graphs for all pieces in an r–division cannot be done efficiently using Klein's MSSP. The reason is that $|V(G-P)|$ can be $\Theta(|V(G)|)$ for all pieces. Instead, the computation can be done in a top-down approach as follows (cf. [BSWN10]). Recall that an r–division is obtained as the set of pieces of the lowest level in a recursive division of the graph using Miller's simple-cycle separators. Consider the set $\mathcal{Q}$ of all pieces at all levels of the recursion (rather than just the set of pieces at the lowest level). Note that there are $O(\log |V(G)|)$ recursive levels since the size of the pieces decreases by a constant factor for every constant number of applications of Miller's cycle separator theorem. First, we compute DDG_Q for every piece $Q \in \mathcal{Q}$. As explained in Section 1.3.5, this can be done in $O(|V(G)| \log |V(G)|)$ for all the pieces in a specific level of the recursion, for a total of $O(|V(G)| \log^2 |V(G)|)$ for all pieces in $\mathcal{Q}$. Next, we consider a piece Q and denote the two subpieces of Q by Q_1 and Q_2. DDG_{G-Q_2} is obtained by computing distances in $DDG_{Q_1} \cup DDG_{G-Q}$ (see Figure 1.6), using multiple applications of FR-DIJKSTRA, once for each node in ∂Q_2. This takes $O(|Q| \log^2 |\partial Q|)$ time per piece, for an overall $O(|V(G)| \log^2 |V(G)|)$ time for all pieces in a specific level. Since the number of levels is bounded by $O(\log |V(G)|)$, the entire computation takes $O(|V(G)| \log^3 |V(G)|)$ time.

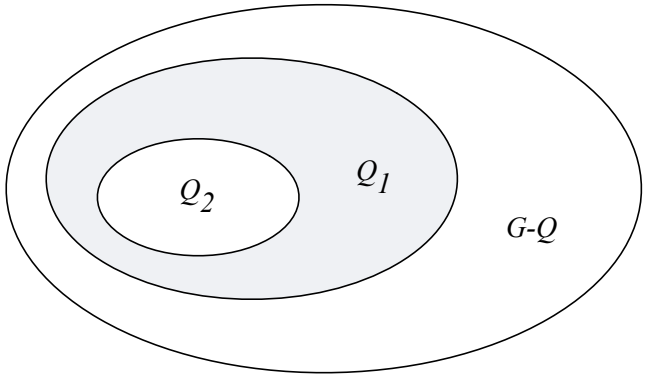

Figure 1.6: Pieces Q_1 and Q_2 in some level of the recursive application of Miller's cycle separator theorem. The piece Q is the union of Q_1 and Q_2. Distances in $G - Q_2$, the exterior of Q_2, are obtained by considering shortest paths in the interior of Q_1 and in $G - Q$, the exterior of Q.

1.4 Capacities and Maximum Flow

Another commonly used attribute is arc-capacity, which quantifies the amount of flow that can be pushed through an arc in a flow network.

Let $S \subset V$ be a set of nodes called sources, and let $T \subseteq V - S$ be a set of nodes called sinks.

A *flow assignment* $\boldsymbol{f}(\cdot)$ is an arc vector (a real-valued function on darts satisfying antisymmetry with respect to rev, see Section 1.2.2).

A *capacity assignment* $\boldsymbol{c}(\cdot)$ is a dart vector (a real-valued function on darts). In many cases only the capacities of arcs are specified. In this case, we extend the capacities to darts by $\boldsymbol{c}(a^+) = \boldsymbol{c}(a)$, and $\boldsymbol{c}(a^-) = 0$.

A flow assignment $\boldsymbol{f}(\cdot)$ *respects the capacity* of dart d if $\boldsymbol{f}(d) \leq \boldsymbol{c}(d)$. $\boldsymbol{f}(\cdot)$ is called a *pseudoflow* if it respects the capacities of all darts.

For a given flow assignment $\boldsymbol{f}(\cdot)$, the *net inflow* (or just *inflow*) of node v is:[6]

$$\text{inflow}_{\boldsymbol{f}}(v) = \sum_{\text{dart } d:\text{head}(d)=v} \boldsymbol{f}(d).$$

The *outflow* of v is $\text{outflow}_{\boldsymbol{f}}(v) = -\text{inflow}_{\boldsymbol{f}}(v)$.

A *preflow* is a pseudoflow $\boldsymbol{f}(\cdot)$ such that for each node $v \in V - S$, $\text{inflow}_{\boldsymbol{f}}(v) \geq 0$.

A flow assignment $\boldsymbol{f}(\cdot)$ is said to *obey conservation* at node v if $\text{inflow}_{\boldsymbol{f}}(v) = 0$. A *feasible flow* is a pseudoflow that obeys conservation at every node other than the sources and sinks. A *feasible circulation* is a pseudoflow that obeys conservation at all nodes. The *value* of a feasible flow $\boldsymbol{f}(\cdot)$ is the sum of inflow at the sinks, $\sum_{t \in T} \text{inflow}_{\boldsymbol{f}}(t)$ or, equivalently, the sum of outflow at the sources. The maximum-flow problem is that of finding a feasible flow with maximum value.

The famous max-flow min-cut theorem [FF56, EFS56] states that the value of a maximum flow from source s to sink t equals the capacity of the minimum cut separating s from t.

For two flow assignments $\boldsymbol{f}, \boldsymbol{f}'$, the *addition* $\boldsymbol{f} + \boldsymbol{f}'$ is the flow that assigns $\boldsymbol{f}(d) + \boldsymbol{f}'(d)$ to every dart d. A flow assignment $\boldsymbol{f}(\cdot)$ is a *quasi-feasible flow*[7] (respectively, *quasi-pseudoflow* or *quasi-preflow*) if there exists a circulation $\boldsymbol{\rho}$ such that $\boldsymbol{f} + \boldsymbol{\rho}$ is a feasible flow (respectively, pseudoflow, preflow).

A *residual path* in G is a path whose darts all have strictly positive capacities. For two sets of nodes A, B, $A \overset{G}{\rightarrow} B$ is used to denote the existence of some residual a-to-b path in G for some nodes $a \in A$ and $b \in B$. Conversely, $A \overset{G}{\not\rightarrow} B$ is used to denote that no such path exists. We will omit the graph G when it is clear from the context.

The *residual graph* of G with respect to a flow assignment $\boldsymbol{f}(\cdot)$ is the graph $G_{\boldsymbol{f}}$ with the same arc-set, node-set, sources and sinks, and with capacity assignment $\boldsymbol{c}_{\boldsymbol{f}}(\cdot)$ such that for every dart d, $\boldsymbol{c}_{\boldsymbol{f}}(d) = \boldsymbol{c}(d) - \boldsymbol{f}(d)$.

It follows from the max-flow min-cut theorem that a feasible flow $\boldsymbol{f}$ in G is maximum if and only if $S \overset{G_{\boldsymbol{f}}}{\not\rightarrow} T$. We use an analog of this condition to define maximum preflows and pseudoflows. Let V^+ denote the set of nodes $\{v \in V - (S \cup T) : \text{inflow}_{\boldsymbol{f}}(v) > 0\}$. Similarly, let V^- denote the set of nodes $\{v \in V - (S \cup T) : \text{inflow}_{\boldsymbol{f}}(v) < 0\}$. We say that a pseudoflow or preflow $\boldsymbol{f}$ is maximum if $S \cup V^+ \overset{G_{\boldsymbol{f}}}{\not\rightarrow} T \cup V^-$.

[6] An equivalent definition, in terms of arcs, is $\text{inflow}_{\boldsymbol{f}}(v) = \sum_{a \in E:\text{head}(a)=v} \boldsymbol{f}(a) - \sum_{a \in E:\text{tail}(a)=v} \boldsymbol{f}(a)$.

[7] the terms preflow and pseudoflow are well-known in the literature (e.g., [GT88, Hoc08, GT90]), the term quasi-feasible flow is new.

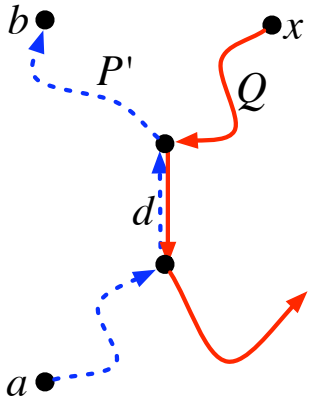

Figure 1.7: Proof of the Sources Lemma. The path P is dashed blue. Its suffix P' is indicated. The dart d is not residual before f is pushed. A flow carrying path of f that starts at x and goes through $\text{rev}(d)$ is shown in solid red.

We next give a few basic lemmas that characterize cases in which the nonexistence of residual paths is preserved when flow is pushed in a graph.

Lemma 1.4.1. *Let $\boldsymbol{\rho}$ be a circulation. Let u and v be nodes in a graph G. Then*

$$u \overset{G}{\not\to} v \Rightarrow u \overset{G_{\boldsymbol{\rho}}}{\not\to} v.$$

Proof. Pushing a circulation does not change the amount of flow crossing any cut. This implies that if there was no u-to-v residual path before $\boldsymbol{\rho}$ was pushed, then there is none after $\boldsymbol{\rho}$ is pushed. □

Lemma 1.4.2. *(Sources Lemma) Let f be a flow with source set X. Let A, B be two disjoint sets of nodes. Then*

$$A \cup X \overset{G}{\not\to} B \Rightarrow A \overset{G_f}{\not\to} B.$$

Proof. The flow f may be decomposed into a cyclic component (a circulation) and an acyclic component. By Lemma 1.4.1, it suffices to show the lemma for an acyclic flow f.

Suppose for the sake of contradiction that there exists a residual a-to-b simple path P after f is pushed for some $a \in A$ and $b \in B$. Let P' be the maximal suffix of P that was residual before the push. Maximality implies that the dart d of P whose head is $\text{start}(P')$ was non-residual before the push, and $f(\text{rev}(d)) > 0$. The fact that $f(\text{rev}(d)) > 0$ implies that before f was pushed there was a residual path Q from some node $x \in X$ to $\text{head}(d)$. Therefore, the concatenation of Q and P' was a residual x-to-b residual path before the push, a contradiction. See Figure 1.7. □

The proof of the following lemma is symmetric.

Lemma 1.4.3. *(Sinks Lemma) Let f be a flow with sink set X. Let A, B be two disjoint sets of nodes. Then*

$$A \overset{G}{\not\to} B \cup X \Rightarrow A \overset{G_f}{\not\to} B.$$

For node sets W, Y, Z, we will use the notation *sources lemma*(W, Y, Z) to indicate the use of the Sources Lemma to establish that there are no W-to-Y residual paths after a flow with source set Z is pushed. We will use *sinks lemma*(W, Y, Z) in a similar manner.

1.4.1 Circulations and Shortest Paths

In this section we discuss a relation between circulations in a directed planar graph G with arc capacities $\boldsymbol{c}$, and shortest paths in the dual graph G^*, where the length of a dart d in G^* is taken to be its capacity in G. Namely,

$$\text{length}_{G^*}(d) = \boldsymbol{c}[d]. \tag{1.6}$$

To the best of our knowledge, this connection was first used by Hassin [Has81], and was later made precise in [KNK93, MN95].

Recall that by our definition of circulations (Eq. (1.1)), every circulation $\boldsymbol{\rho}$ is associated with a face potential $\boldsymbol{\phi_\rho}$. We follow the convention that $\boldsymbol{\phi_\rho}[f_\infty] = 0$. Let d' be a dart of G. By our definition of the faces of G and of the dual graph G^*, d'^* is a dart of the face $\text{head}_{G^*}(d')$. Hence, by Eq. (1.1),

$$\begin{aligned} \boldsymbol{\rho}[d'] &= \sum_f \boldsymbol{\phi_\rho}[f]\boldsymbol{\eta}(f)[d'] \\ &= \boldsymbol{\phi_\rho}[\text{head}_{G^*}(d')] - \boldsymbol{\phi_\rho}[\text{tail}_{G^*}(d')]. \end{aligned} \tag{1.7}$$

Eq. (1.7) implies the following lemma:

Lemma 1.4.4. *If there exists a negative length cycle in G^* with respect to length_{G^*} then there is no feasible circulation in G.*

Proof. Let C be a negative length cycle in G^*. That is $\sum_{d\in C} \boldsymbol{c}[d] < 0$. Assume there exists a feasible circulation $\boldsymbol{\rho}$ in G. Then $\sum_{d\in C} \boldsymbol{\rho}[d] \leq \sum_{d\in C} \boldsymbol{c}[d] < 0$. On the other hand, observe that summing Eq. (1.7) over all darts of a cycle telescopes to zero, a contradiction. □

We will next see that the contrary is also true. Namely, if there is no negative length cycle in G^* then there exists a feasible circulation in G. Observe that $\boldsymbol{\rho}$ is a feasible circulation if and only if there exists a face potential $\boldsymbol{\phi}$ satisfying (1.7) such that for every dart d,

$$\boldsymbol{\phi}[\text{head}_{G^*}(d)] - \boldsymbol{\phi}[\text{tail}_{G^*}(d)] \leq \boldsymbol{c}[d],$$

or equivalently,

$$\boldsymbol{\phi}[\text{head}_{G^*}(d)] \leq \boldsymbol{\phi}[\text{tail}_{G^*}(d)] + \text{length}_{G^*}(d). \tag{1.8}$$

Equation (1.8) is the *shortest-path inequality* for G^* with respect to length_{G^*}.

Lemma 1.4.5. *There is a feasible circulation in G if and only if there exists no negative length cycle in G^* with respect to length_{G^*}. Furthermore, suppose there is no negative length cycle in G^* with respect to length_{G^*}, and let $\boldsymbol{\phi}[f]$ be the f'-to-f distances in G^* for some arbitrary face f'. The circulation $\boldsymbol{\rho}[d] = \boldsymbol{\phi}[\text{head}_{G^*}(d)] - \boldsymbol{\phi}[\text{tail}_{G^*}(d)]$ is a feasible circulation in G.*

Proof. Lemma 1.4.4 shows that if there is a negative length cycle in G^* then there is no feasible circulation in G. To show the other direction, observe that if there is no negative length cycle in G^*, then shortest paths in G^* are well defined. Hence, ϕ is well defined and satisfies the shortest-path inequality (Eq. (1.8)). Therefore $\boldsymbol{\rho}$ is a feasible circulation in G. □

The circulation $\boldsymbol{\rho}$ defined in Lemma 1.4.5 has a few additional properties:

Lemma 1.4.6. *Suppose distances in G^* with respect to length_{G^*} are well defined. Let $\boldsymbol{\phi}[f]$ be the f_∞-to-f distances in G^*. Among all face potential functions that satisfy the shortest-path inequality and assign zero to f_∞, $\boldsymbol{\phi}$ is one with maximum values.*

Proof. Let $\boldsymbol{\phi}'$ be a face potential function such that $\boldsymbol{\phi}'[f] > \boldsymbol{\phi}[f]$ for some face f. Let P be a shortest f_∞-to-f path in G^*. Since $\boldsymbol{\phi}'[f] > \boldsymbol{\phi}[f] = \text{length}_{G^*}(P)$, the reduced length of P with respect to $\boldsymbol{\phi}'$ is negative. Hence, there must exist some dart d of P whose reduced length with respect to $\boldsymbol{\phi}'$ is negative. Namely,

$$\text{length}_{G^*}(d) + \boldsymbol{\phi}'[\text{tail}_{G^*}(d)] - \boldsymbol{\phi}'[\text{head}_{G^*}(d)] < 0,$$

so $\boldsymbol{\phi}'$ violates the shortest-path inequality for the dart d. □

Lemma 1.4.6 has two consequences:

Corollary 1.4.7. *Suppose distances in G^* with respect to length_{G^*} are well defined. Let $\boldsymbol{\phi}[f]$ be the f_∞-to-f distances in G^*. Among all feasible circulations, the circulation $\boldsymbol{\rho}[d] = \boldsymbol{\phi}[\text{head}_{G^*}(d)] - \boldsymbol{\phi}[\text{tail}_{G^*}(d)]$ maximizes the flow on every dart whose tail in G^* is f_∞.*

Proof. Let d be a dart whose tail in G^* is f_∞. The flow assigned to d by $\boldsymbol{\rho}$ is $\boldsymbol{\rho}[d] = \boldsymbol{\phi}[\text{head}_{G^*}(d)] - \boldsymbol{\phi}[\text{tail}_{G^*}(d)] = \boldsymbol{\phi}[\text{head}_{G^*}(d)]$. By the preceding lemma, $\boldsymbol{\phi}$ is maximum among all face potential functions that satisfy the shortest-path inequality and assign zero to f_∞. The claim follows since a circulation is feasible only if it is induced by such a face potential function. □

Corollary 1.4.8. *Suppose distances in G^* with respect to length_{G^*} are well defined. Let $\boldsymbol{\phi}[f]$ be the f_∞-to-f distances in G^*, and let $\boldsymbol{\rho}[d] = \boldsymbol{\phi}[\text{head}_{G^*}(d)] - \boldsymbol{\phi}[\text{tail}_{G^*}(d)]$. There are no clockwise residual cycles in $G_{\boldsymbol{\rho}}$.*

Proof. Let $\boldsymbol{\rho}'$ be a clockwise feasible circulation in $G_{\boldsymbol{\rho}}$. That is, $\boldsymbol{\rho}' = \boldsymbol{\phi}'[\text{head}_{G^*}(d)] - \boldsymbol{\phi}'[\text{tail}_{G^*}(d)]$ for some non-negative face potential function $\boldsymbol{\phi}'$. Therefore, $\boldsymbol{\phi} + \boldsymbol{\phi}'$ is a feasible circulation in G. By Lemma 1.4.6, $\boldsymbol{\phi}$ is the maximum such circulation, so $\boldsymbol{\phi}'$ (and thus also $\boldsymbol{\rho}'$) must be a identically zero. □

1.4.2 Hassin's Algorithm for Maximum st-Planar Flow

An st-planar graph is a planar graph in which nodes s and t are incident to the same face. Hassin [Has81], relying on work of Itai and Shiloach [IS79], gave an algorithm for computing a maximum flow from s to t in an st-planar graph. We describe our interpretation of this algorithm, which is used by the algorithm in Chapter 5. The algorithm is a natural consequence of Corollary 1.4.7.

Hassin's algorithm starts by adding to G an artificial infinite capacity arc a from t to s. Let d be the dart that corresponds to a and whose head is s. Let $\phi[\cdot]$ denote the shortest-path distances from $\text{tail}_{G^*}(d)$ in G^*, where the length of a dual dart is defined as the capacity of the primal dart. Consider the flow

$$\boldsymbol{\rho}[d'] = \boldsymbol{\phi}[\text{head}_{G^*}(d')] - \boldsymbol{\phi}[\text{tail}_{G^*}(d')] \text{ for all darts } d' \tag{1.9}$$

Corollary 1.4.7 states that the flow assigned to d by $\boldsymbol{\rho}$ is maximum among all feasible circulations. This implies that if after removing the artificial arc a from G, $\boldsymbol{\rho}$ is a maximum feasible flow from s to t in G.

We will be interested in a *max flow with limit* x from s to t rather than a maximum flow, i.e., a flow whose value is at most a given number x but is otherwise maximal. It is not difficult to see that setting the capacity of the artificial arc a to x instead of infinity results in the desired limited max flow [KNK93].

Note that, instead of using an artificial arc from t to s, one can use an existing arc whose endpoints are s and t as the arc a above. In order for this to work, the capacity of the dart d that corresponds to a and whose head is t must be zero. The is achieved by first pushing flow on d to saturate it. Since a is now an edge of G, $\boldsymbol{\rho}$ is a feasible circulation, rather than a maximum flow (flow is being pushed back from t to s along a). To convert this circulation into a maximum flow, the algorithm must remove flow on a. If we define $\boldsymbol{f}$ by

$$\boldsymbol{f}[d'] = \begin{cases} -\boldsymbol{\rho}[d'] & \text{if } d' \text{ corresponds to } a \\ 0 & \text{otherwise} \end{cases}, \tag{1.10}$$

then $\boldsymbol{\rho} + \boldsymbol{f}$ is a maximum st-flow. We will use the fact that this flow can be represented jointly by the face potential vector $\boldsymbol{\phi}$ and the flow values $\boldsymbol{f}[d']$ for the two darts corresponding to a.

1.4.3 Converting a Maximum Pseudoflow into a Maximum Feasible Flow

Let $\boldsymbol{f}$ be a pseudoflow in a planar graph G with node set V, sources S and sinks T. Let V^+ denote the set of nodes $\{v \in V - (S \cup T) : \text{inflow}_{\boldsymbol{f}}(v) > 0\}$. Similarly, let V^- denote the set of nodes $\{v \in V - (S \cup T) : \text{inflow}_{\boldsymbol{f}}(v) < 0\}$. Suppose $\boldsymbol{f}$ is a maximum pseudoflow. That is, assume $S \cup V^+ \overset{G_{\boldsymbol{f}}}{\not\to} T \cup V^-$. In this section we show how to convert $\boldsymbol{f}$ into a maximum feasible flow $\boldsymbol{f}'$. This procedure was first described for planar graphs by Johnson and Venkatesan [JV82]. The original description of the procedure took $O(n \log n)$ time, but using the flow cycles cancellation technique of Kaplan and Nussbaum [KN09] the running time is $O(n)$ (the technique of Kaplan and Nussbaum applies Corollary 1.4.8 twice to eliminate both clockwise and counterclockwise residual cycles).

We begin by converting $\boldsymbol{f}$ to an acyclic pseudoflow in linear time [KN09]. That is, after the conversion there is no cycle C such that $\boldsymbol{f}[d] > 0$ for every dart d of C. Since $\boldsymbol{f}$ is acyclic, the darts with a positive flow induce a topological ordering on the nodes of the graph G. Let v be the last member of V^+ in the topological ordering, and let d be an arbitrary dart that carries flow into v. We set $\boldsymbol{f}[d] = \max\{\boldsymbol{f}[d] - \text{inflow}_{\boldsymbol{f}}(v), 0\}$, and set $\boldsymbol{f}[\text{rev}(d)]$ accordingly. The flow assignment $\boldsymbol{f}$ maintains the invariant $S \cup V^+ \overset{G_{\boldsymbol{f}}}{\not\to} T \cup V^-$ by *sinks lemma*($S \cup V^+$,$T \cup V^-$,$\{v\}$). As long as v is in

V^+, there must be a dart d which carries flow to v. By changing the flow on d we cannot add to V^+ a new node that appears later than v in the topological ordering. We repeat this process until V^+ is empty. Since we reduce the flow on each dart at most once, this takes linear time. Next we handle V^- while keeping the invariant $S \cup V^+ \overset{G_f}{\nrightarrow} T \cup V^-$ in a symmetric way, by repeatedly decreasing the flow on darts that carry flow from the first vertex of V^- in the topological ordering.

The total running time is $O(n)$, and since eventually both V^+ and V^- are empty sets, the invariant $S \cup V^+ \overset{G_f}{\nrightarrow} T \cup V^-$ implies that the resulting pseudoflow is a feasible flow. This is the required flow $\boldsymbol{f}'$.

Part II

Shortest Paths

Chapter 2

Shortest Paths in Directed Planar Graphs with Negative Lengths

In this chapter we describe algorithms for computing shortest paths in a directed planar graph with real (positive and negative) edge lengths and no negative length cycles. Our discussion will be focused on computing shortest-path distances, but the results apply to computing the shortest paths themselves as well as to identifying a negative length cycle if one exists. The shortest-path problem is a classical problem in combinatorial optimization, which was extensively studied. For general graphs, the Bellman-Ford algorithm (see Section 1.3.2) solves the problem in $O(mn)$ time, where m is the number of arcs and n is the number of nodes. For integer lengths whose absolute values are bounded by N, the algorithm of Gabow and Tarjan [GT89] takes $O(\sqrt{n}m\log(nN))$. For integer lengths exceeding $-N$, the algorithm of Goldberg [Gol95] takes $O(\sqrt{n}m\log N)$ time. For non-negative lengths, the problem is easier and can be solved using Dijkstra's algorithm (see Section 1.3.1) in $O((n+m)\log n)$ time if elementary data structures are used [Joh77], and in $O(n\log n+m)$ time when implemented with Fibonacci heaps [FT87].

For planar graphs, there has been a series of results yielding progressively better bounds. The first algorithm that exploited planarity was due to Lipton, Rose, and Tarjan [LRT79], who gave an $O(n^{3/2})$ algorithm. Henzinger et al. [HKRS97] gave an $O(n^{4/3}\log^{2/3} D)$ algorithm where lengths are assumed to be integers, and D is the sum of the absolute values of the lengths. Fakcharoenphol and Rao [FR06] gave an algorithm requiring $O(n\log^3 n)$ time and $O(n\log n)$ space. All of these algorithms use planar separators to divide and conquer the problem. Fakcharoenphol and Rao used cycle separators recursively, and observed that the dense distance graph for a set of k boundary nodes adjacent to a single face of a piece can be decomposed into Monge matrices (see Section 1.3.7 for a description of this decomposition). They then used the Monge property to implement the Bellman-Ford algorithm on a set of dense distance graphs with k boundary nodes in $O(k^2\log^2 k)$. This is done using a simple divide and conquer algorithm. They also implement Dijkstra's algorithm in $O(k\log^2 k)$ (see, again, Section 1.3.7). At each recursive level of their shortest-path algorithm, the

Bellman-Ford algorithm is executed once to obtain a feasible price function, and Dijkstra's algorithm is run $O(\sqrt{n})$ times to compute the distances between all pairs of boundary nodes.

The algorithm we describe in this chapter uses cycle separators as well, but in a simpler way than that of Fakcharoenphol and Rao's. A main technical contribution we introduce is the use of the more efficient SMAWK algorithm (see Section 1.3.6) to implement the Bellman-Ford step. This is a stronger use of the Monge property than the simple divide and conquer approach of Fakcharoenphol and Rao to the Bellman-Ford step. Then, instead of using FR-Dijkstra to compute the all pairs distances, we use the MSSP algorithm (Section 1.3.4). The resulting algorithm is more efficient and arguably simpler than Fakcharoenphol and Rao's algorithm.

In comparing our algorithms to that of Fakcharoenphol and Rao, it is worth lingering on the issue of holes. In our first algorithm (Section 2.1) holes are not a problem; because of the simple recursive way in which we apply the cycle separator, all boundary nodes at a given level of the recursion are adjacent to a single face of a piece. In our faster algorithm (Section 2.2), as well as in Fakcharoenphol and Rao's, boundary nodes are adjacent to a constant number of holes. One of the main technical contributions of Section 2.2 is in showing how to exploit the Monge property in the Bellman-Ford step in the case where the boundary nodes are adjacent to two distinct faces (Lemma 2.2.4). Boundary nodes on a constant number of holes are then handled by considering each pair of holes separately. Since the number of holes is constant, so is the number of pairs. Fakcharoenphol and Rao's paper, which is a little terse on details in several occasions, does not address this issue (they do address the issue of holes for the Dijkstra step, but not for the Bellman-Ford step).

Our algorithm was extended by Chambers, Erickson, and Nayyeri [CEN09] to handle graphs embedded on a genus g surface in $O(g^2 n \log^2 n)$ time.[1]

Applications

We had already discussed in Section 1.4 the relation between shortest paths with negative lengths, feasible circulations and maximum flow in planar graphs. See also [MN95]. In addition, several techniques for computer vision, including image segmentation algorithms by Cox, Rao, and Zhong [CRZ96] and by Jermyn and Ishikawa [JI01, IJ01], and a stereo matching technique due to Veksler [Vek02], involve finding negative-length cycles in graphs that are essentially planar grids. Our algorithms can be used to implement these techniques.

Organization

For ease of presentation we first describe a linear-space $O(n \log^2 n)$-time algorithm for the problem. This is a divide-and-conquer algorithm which decomposes the graph using recursive applications of Miller's cycle separator. This algorithm appeared in [KMW10]. Next, we describe a generalization

[1] important details are omitted in the conference version [CEN09]. A detailed description will appear in the journal version [Eri11].

of that algorithm that uses an r–division at each recursive level instead of a single cycle separator. This more complicated algorithm, which appeared in [MWN10], runs in $O(n \log^2 n / \log\log n)$ time and is the fastest algorithm for the problem to date.

2.1 A Linear Space $O(n \log^2 n)$-Time Algorithm

Algorithm 2.1 SHORTESTPATH(G, s)

Input: a directed embedded planar graph G with arc-lengths, and a node s of G
Output: a table d giving distances in G from s to all nodes of G

1: **if** G has at most 2 nodes, the problem is trivial; return the result
2: find a Jordan separator C of G with $O(\sqrt{n})$ boundary nodes
3: let R_0,R_1 be the external and internal regions of G with respect to C
4: let r be a boundary node

// recursive call
5: **for** $i = 0, 1$: $d_i \leftarrow$ SHORTESTPATH(R_i, r)

// intra-region boundary distances
6: **for** $i = 0, 1$: use d_i as input to the multiple-source shortest-path algorithm to compute a table δ_i such that $\delta_i[u, v]$ is the u-to-v distance in R_i for every pair u, v of boundary nodes

// single-source inter-region boundary distances
7: use δ_0 and δ_1 to compute a table B such that $B[v]$ is the r-to-v distance in G for every boundary node v

// single-source inter-region distances
8: **for** $i = 0, 1$: use tables d_i and B, and Dijkstra's algorithm to compute a table d'_i such that $d'_i[v]$ is the r-to-v distance in G for every node v of R_i

// rerooting single-source distances
9: define a price function ϕ for G such that $\phi[v]$ is the r-to-v distance in G:
$$\phi[v] = \begin{cases} d'_0[v] & \text{if } v \text{ belongs to } R_0 \\ d'_1[v] & \text{otherwise} \end{cases}$$
10: use Dijkstra's algorithm with price function ϕ to compute a table d such that $d[v]$ is the s-to-v distance in G for every node v of G
11: **return** d

The high-level description of the algorithm appears in Algorithm 2.1. After finding a Jordan separator and selecting a boundary node as a temporary source node, the algorithm consists of the following five steps, which are described in detail in the sequel:

Recursive call: Distances from r are computed recursively in each region.

Intra-region boundary distances: For each region R_i, distances between all pairs of boundary nodes in R_i are computed using Klein's multiple source shortest-path algorithm (Section 1.3.4).

Single-source inter-region boundary distances: A variant of Bellman-Ford is used to compute SSSP distances in G from r to all boundary nodes. See Section 2.1.1.

Single-source inter-region distances: Distances from the previous stages are used as a feasible price function and as an initialization for Dijkstra's algorithm to obtain distances from r in G. See Section 2.1.2.

Rerooting single-source distances The from-r distances in G are used as a feasible price function for Dijkstra's algorithm once more to finally compute distances from the given source.

2.1.1 Computing Single-Source Inter-Region Boundary Distances

In this section we describe how to efficiently compute the distances in G from r to all boundary nodes (i.e., the nodes of C). This is done using δ_0 and δ_1, the all-pairs distances in R_0 and in R_1 between nodes of C, which were computed in the previous step.

Theorem 2.1.1. *Let G be a directed plane graph with arbitrary arc-lengths. Let C be a Jordan separator in G and let R_0 and R_1 be the external and internal regions of G with respect to C. Let δ_0 and δ_1 be the all-pairs distances between nodes in C in R_0 and in R_1, respectively. Let $r \in C$ be an arbitrary node on the boundary. There exists an algorithm that, given δ_0 and δ_1, computes the from-r distances in G to all nodes in C in $O(|C|^2\alpha(|C|))$ time and $O(|C|)$ space.*

The rest of this section describes the algorithm, thus proving Theorem 2.1.1. The following structural lemma stands in the core of the computation. The same lemma has been implicitly used by previous planarity-exploiting algorithms.

Lemma 2.1.2. *Let P be a simple r-to-v shortest path in G, where $v \in C$. Then P can be decomposed into at most $|C|$ subpaths $P = P_1P_2P_3\ldots$, where the endpoints of each subpath P_j are boundary nodes, and P_j is a shortest path in some region R_i.*

Proof. Consider a decomposition of $P = P_1P_2P_3\ldots$ into maximal subpaths such that the subpath P_j consists of nodes of some region R_i. Since r and v are boundary nodes, and since the boundary nodes are the only nodes common to different regions, each subpath P_j starts and ends on a boundary node. If P_j were not a shortest path in R_i between its endpoints, replacing P_j in P with a shorter path would yield a shorter r-to-v path, a contradiction.

It remains to show that there are at most $|C|$ subpaths in the decomposition of P. Since P is simple, each node, and in particular each boundary node appears in P at most once. Hence P can be decomposed into no more than $|C| - 1$ subpaths. □

Lemma 2.1.2 gives rise to a dynamic-programming solution for calculating the from-r distances to nodes of C, which resembles the Bellman-Ford algorithm. The pseudocode is given in Algorithm 2.2. Note that, at this level of abstraction, there is nothing novel about this dynamic program. Our contribution is in an efficient implementation of Line 5.

Algorithm 2.2 Implementation of the algorithm in Theorem 2.1.1 for single-source inter-region boundary distances

1: $e_0[v] \leftarrow \infty$ for all $v \in C$
2: $e_0[r] \leftarrow 0$
3: **for** each $j = 1, 2, \ldots, |C|$
4: **for** each region R_i
5: $e_j[v] \leftarrow \min_{w \in C}\{e_{j-1}[w] + \delta_i[w, v]\}$ for all $v \in C$
6: $B[v] \leftarrow e_{|C|}[v]$ for all $v \in C$

Before we continue with the description of the algorithm, let us explicitly show the relation to the Bellman-Ford algorithm. Let H_{R_i} be the directed graph having the boundary vertices of R_i as vertices and having an arc (u, v) of length $d_{R_i}(u, v)$ between each pair of vertices u and v. We call H_{R_i} the dense distance graph for R_i. Note that H_{R_i} is not planar (it is a complete graph), but the distances among the nodes of H_{R_i} are induced by the planar graph R_i. In terms of the dense distance graphs, the algorithm computes a shortest-path tree rooted at r in the graph $\cup_{i \in \{1,2\}} H_{R_i}$ using an efficient implementation of the Bellman-Ford algorithm. It consists of $|C|$ iterations. On each iteration it considers all possible ways to extend the distances computed so far using one more edge in each region. This is called *relaxing* all the edges of each H_{R_i}. In terms of the original planar graph this corresponds to choosing the shortest way to extend the distances computed so far using one more shortest path between boundary nodes in each of the regions. This notion is made formal in the following lemma:

Lemma 2.1.3. *After the table e_j is updated by the algorithm, $e_j[v]$ is the length of a shortest path in G from r to v that can be decomposed into no more than j subpaths $P = P_1 P_2 P_3 \ldots P_j$, where each subpath P_i is a shortest path in some region whose endpoints are boundary nodes.*

Proof. By induction on j. For the base case, e_0 is initialized to be infinity for all nodes other than r, trivially satisfying the lemma. For $j > 0$, assume that the lemma holds for $j - 1$, and let P be a shortest path in G that can be decomposed into $P_1 P_2 \ldots P_j$ as above. Assume, without loss of generality, that P_j is a path in R_0 (the other case is symmetric). Let w and v be the start and end nodes of P_j, respectively. Consider the prefix P', $P' = P_1 P_2 \ldots P_{j-1}$. P' is a shortest r-to-w path in G that can be decomposed into no more than $j - 1$ subpaths as above. Hence, by the inductive hypothesis, when e_j is updated in Line 5, $e_{j-1}[w]$ already stores the length of P'. Thus $e_j[v]$ is updated in Line 5 to be at most $e_{j-1}[w] + \delta_0[w, v]$. Since, by definition, $\delta_0[w, v]$ is the length of the shortest w-to-v path in R_0, it follows that $e_j[v]$ is at most the length of P. For the opposite direction, since for any boundary node u, $e_{j-1}[u]$ is the length of some path that can be decomposed into at most $j - 1$ subpaths as above, $e_j[v]$ is updated in Line 5 to the length of some path that can be decomposed into at most j subpaths as above. Hence, since P is the shortest such path, $e_j[v]$ is at least the length of P. □

From Lemma 2.1.2 and Lemma 2.1.3, it immediately follows that the table $e_{|C|}$ stores the from-r shortest-path distances in G, so the assignment in Line 6 is justified, and the table B also stores

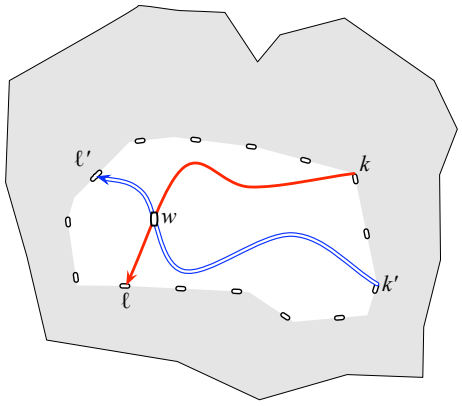

Figure 2.1: Nodes $k < k' < \ell < \ell'$ in clockwise order on the boundary nodes. Paths from k to ℓ and from k' to ℓ' must cross at some node w. This is true both in the internal and the external subgraphs of G

these distances.

We now show how to perform all the minimizations in the j^{th} iteration of Line 5 in $O(|C|\alpha(|C|))$ time. Consider a region R_i. Since all boundary nodes lie on the boundary of a single face of R_i, there is a natural cyclic clockwise order $v_1, v_2, \ldots, v_{|C|}$ on the nodes of C. Define a $|C| \times |C|$ matrix A with elements $A_{k\ell} = e_{j-1}(v_k) + \delta_i(v_k, v_\ell)$. Note that computing all minima in Line 5 is equivalent to finding the column-minima of A. We define the *upper triangle* of A to be the elements of A on or above the main diagonal. More precisely, the upper triangle of A is the portion $\{A_{k\ell} : k \leq \ell\}$ of A. Similarly, the lower triangle of A consists of all the elements on or below the main diagonal of A.

Lemma 2.1.4. *For any four indices k, k', ℓ, ℓ' such that either $A_{k\ell}, A_{k\ell'}, A_{k'\ell}$ and $A_{k'\ell'}$ are all in A's upper triangle, or are all in A's lower triangle (i.e., either $1 \leq k \leq k' \leq \ell \leq \ell' \leq |C|$ or $1 \leq \ell \leq \ell' \leq k \leq k' \leq |C|$), the convex Monge property holds:*

$$A_{k\ell} + A_{k'\ell'} \geq A_{k\ell'} + A_{k'\ell}.$$

Proof. Consider the case $1 \leq k \leq k' \leq \ell \leq \ell' \leq |C|$, as in Fig. 2.1. Since R_i is planar, any pair of paths in R_i from k to ℓ and from k' to ℓ' must cross at some node w of R_i. Let $b_k = e_{j-1}(v_k)$ and let $b_{k'} = e_{j-1}(v_{k'})$. Let $\Delta(u, v)$ denote the u-to-v distance in R_i for any nodes u, v of R_i. Note that

$\Delta(u,v) = \delta_i(u,v)$ for $u, v \in C$. We have

$$\begin{aligned} A_{k,\ell} + A_{k',\ell'} &= (b_k + \Delta(v_k, w) + \Delta(w, v_\ell)) \\ &\quad + (b_{k'} + \Delta(v_{k'}, w) + \Delta(w, v_{\ell'})) \\ &= (b_k + \Delta(v_k, w) + \Delta(w, v_{\ell'})) \\ &\quad + (b_{k'} + \Delta(v_{k'}, w) + \Delta(w, v_\ell)) \\ &\geq (b_k + \Delta(v_k, v_{\ell'})) + (b_{k'} + \Delta(v_{k'}, v_\ell)) \\ &= (b_k + \delta_i(v_k, v_{\ell'})) + (b_{k'} + \delta_i(v_{k'}, v_\ell)) \\ &= A_{k,\ell'} + A_{k',\ell}. \end{aligned}$$

The case ($1 \leq \ell \leq \ell' \leq k \leq k' \leq |C|$) is similar.

□

Lemma 2.1.5. *A single iteration of Line 5 of the algorithm in Algorithm 2.2 can be computed in $O(|C|\alpha(|C|))$ time.*

Proof. We need to show how to find the column-minima of the matrix A. We compute the column-minima of A's lower and upper triangles separately, and obtain A's column-minima by comparing the two values obtained for each column.

It follows directly from Lemma 2.1.4 that replacing the upper triangle of A with blanks yields a falling staircase matrix. By [KK90], the column-minima of this falling staircase matrix can be computed in $O(|C|\alpha(|C|))$ time. Another consequence of Lemma 2.1.4 is that the column-minima of the upper triangle of A may also be computed using the algorithm in [KK90]. To see this consider a counterclockwise ordering of the nodes of C, $v'_1, v'_2, \ldots, v'_{|C|}$, such that $v'_k = v_{|C|+1-k}$. This reverses the order of both the rows and the columns of A, thus turning its upper triangle into a lower triangle. Again, replacing the upper triangle of this matrix with blanks yields a falling staircase matrix.

We thus conclude that A's column-minima can be computed in $O(2|C| \cdot \alpha(|C|) + |C|) = O(|C| \cdot \alpha(|C|))$ time. Note that we never actually compute and store the entire matrix A as this would take $O(|C|^2)$ time. We compute the entries necessary for the computation on the fly in $O(1)$ time per element.

□

Lemma 2.1.5 shows that the time it takes the algorithm described in Algorithm 2.2 to compute the distances between r and all nodes of C is $O(|C|^2 \cdot \alpha(|C|))$. We have thus proved Theorem 2.1.1. The choice of separator ensures $|C| = O(\sqrt{n})$, so this computation is performed in $O(n\alpha(n))$ time.

2.1.2 Computing Single-Source Inter-Region Distances

In the previous section we showed how to compute a table B that stores the distances from r to all the boundary nodes in G. We now describe how to compute the distances from r to all other nodes of G. This is in each region R_i separately. We use $d_i[\cdot]$, the from-r distances in R_i, as a feasible

price function in a Dijkstra's computation in which we initialize the label of every boundary node u according to $B[u]$, its distance from R in G. More precisely, since Dijkstra's algorithm is invoked with the reduced lengths with respect to d_i, the label of u is initialized to the reduced r-to-u distance in G, namely to $B[u] + d_i[r] - d_i[u]$. Since $d_i[r] = 0$, the initialization for boundary node u is just $B[u] - d_i[u]$.

The correctness of this procedure follows from the following lemma:

Lemma 2.1.6. *Let P be an r-to-v shortest path in G, where $v \in R_i$. Then P can be expressed as $P = P_1P_2$, where P_1 is a (possibly empty) shortest path from r to a boundary node $u \in C$, and P_2 is a (possibly empty) shortest path from u to v that only visits nodes of R_i.*

Proof. Let u be the last boundary node visited by P. Let P_1 be the r-to-u prefix of P, and let P_2 be the u-to-v suffix of P. Since P_1 and P_2 are subpaths of a shortest path in G, they are each shortest as well. By choice of u, P_2 has no internal boundary nodes, so it is a path in R_i. □

The time required for this step is dominated by the $O(n \log n)$ running time of Dijkstra's algorithm. We note that one may use the algorithm of Henzinger et al. [HKRS97] instead of Dijkstra to obtain a linear running time for this stage. This however does not change the overall running time of our algorithm.

2.1.3 Correctness and Analysis

We will show that at each step of the algorithm, the necessary information has been correctly computed and stored. The recursive call in Step 4 computes and stores the from-r distances in R_i. The conditions for applying Klein's algorithm in Step 5 hold since all boundary nodes lie on the boundary of a single face of R_i and since the from-r distances in R_i constitute a feasible price function for R_i (see also the discussion at the end of Section 1.3.4). The correctness of the single-source inter-region boundary distances stage in Step 6 and of the single-source inter-region distances stage in Step 7 was proved in Sections 2.1.1 and 2.1.2. Thus, the r-to-v distances in G for all nodes v of G are stored in d'_0 for $v \in R_0$ and in d'_1 for $v \in R_1$. Note that d'_0 and d'_1 agree on distances from r to boundary nodes. Therefore, the price function ϕ defined in Step 8 is feasible for G, so the conditions to run Dijkstra's algorithm in Step 9 hold, and the from-s distances in G are correctly computed. We have thus established the correctness of our algorithm.

To bound the running time of the algorithm we bound the time it takes to complete one recursive call to SHORTESTPATH. Let $|G|$ denote the number of nodes in the input graph G, and let $|R_i|$ denote the number of nodes in each of its subgraphs. Computing the intra-subgraph boundary-to-boundary distances using Klein's algorithm takes $O(|R_i| \log |R_i|)$ for each of the two subgraphs, which is in $O(|G| \log |G|)$. Computing the single-source distances in G to the boundary nodes is done in $O(|G|\alpha(|G|))$, as we explain in Section 2.1.1. The extension to all nodes of G is again done in $O(|R_i| \log |R_i|)$ for each subgraph. Distances from the given source are computed in an additional $O(|G| \log |G|)$ time. Thus the total running time of one invocation is $O(|G| \log |G|)$. Therefore the

running time of the entire algorithm is given by

$$\begin{aligned} T(|G|) &= T(|R_0|) + T(|R_1|) + O(|G| \log |G|) \\ &= O(|G| \log^2 |G|). \end{aligned}$$

Here we used the properties of the separator, namely that $|R_i| \leq 2|G|/3$ for $i = 0, 1$, and that $|R_0| + |R_1| = |G| + O(\sqrt{|G|})$. The formal proof of this recurrence is given in the following lemma.

Lemma 2.1.7. *Let $T(n)$ satisfy the recurrence $T(n) = T(n_1)+T(n_2)+O(n \log n)$, where $n \leq n_1+n_2 \leq n + 4\sqrt{n}$ and $n_i \leq \frac{2n}{3}$. Then $T(n) = O(n \log^2 n)$.*

Proof. We show by induction that for any $n \geq N_0$, $T(n) \leq Cn \log^2 n$ for some constants N_0, C. For a choice of N_0 to be specified below, let C_0 be such that for any $N_0 \leq n \leq 3N_0$, $T(n) \leq C_0 n \log^2(n)$. Let C_1 be a constant such that $T(n) \leq T(n_1)+T(n_2)+C_1(n \log n)$. Let $C = \max\left\{C_0, \frac{C_1}{\log(3/2)}\right\}$. Note that this choice serves as the base of the induction since it satisfies the claim for any $N_0 \leq n \leq 3N_0$. We assume that the claim holds for all n' such that $3N_0 \leq n' < n$ and show it holds for n. Since for $i = 1, 2$, $n_i \leq 2n/3$ and $n_1 + n_2 \geq n$, it follows that $n_i \geq n/3 > N_0$. Therefore, we may apply the inductive hypothesis to obtain:

$$\begin{aligned} T(n) &\leq C(n_1 \log^2 n_1 + n_2 \log^2 n_2) + C_1 n \log n \\ &\leq C(n_1 + n_2) \log^2(2n/3) + C_1 n \log n \\ &\leq C(n + 4\sqrt{n}) \left(\log(2/3) + \log n\right)^2 + C_1 n \log n \\ &\leq Cn \log^2 n + 4C\sqrt{n} \log^2 n - 2C \log(3/2) n \log n \\ &\quad + C(n + 4\sqrt{n}) \log^2(2/3) + C_1 n \log n. \end{aligned}$$

It therefore suffices to show that

$$4C\sqrt{n} \log^2 n - 2C \log(3/2) n \log n + C(n + 4\sqrt{n}) \log^2(2/3) + C_1 n \log n \leq 0,$$

or equivalently that

$$\left(1 - \frac{C_1}{2C \log(3/2)}\right) n \log n \geq \frac{2}{\log(3/2)} \sqrt{n} \log^2 n + \frac{\log(3/2)}{2}(n + 4\sqrt{n}).$$

Let N_0 be such that the right hand side is at most $\frac{1}{2} n \log n$ for all $n > N_0$. The above inequality holds for every $n > N_0$ since we chose $C > \frac{C_1}{\log(3/2)}$ so the coefficient in the left hand side is at least $\frac{1}{2}$.

□

We have thus proved that the total running time of our algorithm is $O(n \log^2 n)$. We turn to the space bound. The space required for one invocation is $O(|G|)$. Each of the two recursive calls can use the same memory locations one after the other, so the space is given by

$$\begin{aligned} S(|G|) &= \max\{S(|R_0|), S(|R_1|\} + O(|G|) \\ &= O(|G|) \quad \text{because} \max\{|R_0|, |R_1|\} \leq 2|G|/3. \end{aligned}$$

We have proved Theorem 2.1.1.

2.2 Improving the running time to $O(n \log^2 n/ \log\log n)$

In this section we show how to extend the algorithm in Section 2.1 to decompose the graph, at each level of the recursion, using an r–division rather than using a single cycle separator. The rationale is we may be able to achieve smaller recursion depth when decomposing the graph into multiple regions, which will result in reduced running time.

Note that the bottleneck in the running time analysis of the non-recursive part of the algorithm in Section 2.1 is the intra-region boundary distances step, which uses Klein's MSSP algorithm to compute in $O(|R_i| \log |R_i|)$ time the all-pair distances among the boundary nodes in each of the subgraphs R_i. This sums up to $O(|G| \log |G|)$ whether G is decomposed into just two regions, as in Section 2.1, or into n/r regions if we were using an r–division. On the other hand, the single-source inter-region boundary distances step takes time proportional (up to an $\alpha(n)$ term) to the square of the number of boundary nodes. When using a single cycle separator, the number of boundary nodes is $O(\sqrt{|G|})$. In an r–division, the graph is decomposed into $p = O(|G|/r)$ regions, and the total number of boundary nodes is $O(p\sqrt{r}) = O(\sqrt{|G|p})$. This suggests that we can afford using a division into roughly $\log |G|$ regions without increasing the overall asymptotic running time. Using such a division the depth of the recursion reduces to $O(\log n \log\log n)$. Since each recursive level takes $O(n \log n)$ time, the total running time is reduced to $O(n \log^2 n/ \log\log n)$.

The high-level description of the algorithm stays the same. We next describe the changes required in each of the five steps. In most steps dealing with multiple regions instead of just two is straight forward. The main difficulty is in dealing with the fact that in an r–division the boundary nodes of a region lie on more than one face of that region. The Monge property that we used in the inter-region boundary distance step in Section 2.1.1 does not hold when the boundary nodes are not incident to the same face. We present a technique to deal with this difficulty.

Let $p = \log n/\alpha(n)$. We compute a $\frac{n\alpha(n)}{\log n}$–division of G. For each region R_i in this division, we pick an arbitrary boundary vertex r_i.

Recursive call: Distances from r_i are computed recursively in each region R_i.

Intra-region boundary distances: For each region R_i, distances between all pair of boundary nodes in R_i are computed using a constant number of applications of Klein's MSSP algorithm; for each hole h of R_i, we run the MSSP algorithm with h designated as the infinite face of R_i. This yields the distances from the boundary nodes on h to all other boundary nodes of R_i. Since the number of holes is bounded by a constant, the total number of invocations of Klein's MSSP algorithm on R_i is constant, so the running time is $O(|R_i| \log |R_i|)$.

Single-source inter-region boundary distances: A variant of Bellman-Ford is used to compute SSSP distances in G from r to all boundary nodes. This is where the algorithm differs most significantly from the one in Section 2.1. See Section 2.2.1

Single-source inter-region distances: Distances from the previous stages are used to as a feasible price function and as an initialization for Dijkstra's algorithm to obtain distances from r in G. This step is identical to the one in Section 2.1.2. The only difference is that here there are $O(\log n/\alpha(n)$ regions instead of just two. The time required for this step is $\sum O(|R_i| \log |R_i|) = O(n \log n)$.

Rerooting single-source distances The from-r distances in G are used as a feasible price function for Dijkstra's algorithm once more to finally compute distances from the given source. This step is identical to the one in Section 2.1.

2.2.1 Computing Single-source Inter-region Boundary Distances

Recall that this step implements the Bellman-Ford algorithm. In Section 2.1.1, we explained that on each iteration it considers all possible ways to extend the distances computed so far using one more shortest path between boundary nodes in each of the regions. There, this was achieved by considering, for each region, the paths between all pairs of nodes on the separator C, which is shared by both regions. Here, different regions have different bounding cycles. In fact, each region has a constant number of bounding cycles instead of just one. The revised algorithm therefore considers all possible shortest paths between boundary nodes of each region by considering all possible pairs of bounding cycles for each region.

For a region R, let ∂R denote the boundary of R which consists of a constant number of cycles. Let χ denote the set of all boundary nodes of all regions $\chi = \bigcup_i \partial R_i$.

Algorithm 2.3 Extension of Algorithm 2.2 for finding single-source inter-region boundary distances using an r–division with a constant number of holes

1: $e_0[v] \leftarrow \infty$ for all $v \in \chi$
2: $e_0[r] \leftarrow 0$
3: **for** $j = 1, 2, \ldots, |\chi|$
4: **for** each region R_i
5: **for** each pair of cycles C_1, C_2 in ∂R_i
6: $e_j[v] \leftarrow \min_{w \in C_1}\{e_{j-1}[w] + \delta_i[w, v]\}$ for all $v \in C_2$
7: $B[v] \leftarrow e_{|\chi|}[v]$ for all $v \in \chi$

The correctness of the algorithm follows from a trivial generalization of Lemma 2.1.2, where the cycle separator C should be replaced by the set of separator nodes χ, and from Lemma 2.1.3 which applies both to cycle separators and to r–divisions. All that remains is to show how to efficiently implement Line 6 which, for a given pair of bounding cycles C_1 and C_2 of a region R, considers augmenting the current distances with shortest paths in R starting at nodes of C_1 and ending at nodes of C_2. We will refer to this operation as relaxing the edges of H_R starting at C_1 and ending at C_2, and will show how to implement it in $O((|C_1| + |C_2|)\alpha(|C_1| + |C_2|))$ time, with $O(|V_R| \log |V_R|)$ preprocessing time.

If $C_1 = C_2$ we can relax all the edges as described in Section 2.1.1. We therefore assume that $C_1 \neq C_2$.

Our solution is based on the following lemma, whose correctness is proved in the remainder of this section.

Lemma 2.2.1. *Let $H = \{H_i\}$ be set of directed graphs with real arc lengths. Let R be a planar graph, and let C_1, C_2 be two distinct faces of R. Assume some graph $H' \in H$ is a bipartite graph $H' = \langle C_1 \cup C_2, C_1 \times C_2 \rangle$ equipped with arc lengths M, such that for $a \in C_1, b \in C_2$, the length M_{ab} of the arc ab is the length of a shortest $a - to - b$ path in R. There exist two Monge matrices M_1, M_2 such that all shortest paths in $\bigcup_{H_i \in H} H_i$ remain the same when H' is replaced with two copies of H', one equipped with arc lengths M_1 and the other with M_2. Furthermore, M_1 and M_2 can be computed in the time required for a single shortest-path computation in R plus additional $O(|V_R| \log |V_R|)$ time.*

Before going into the details, let us give an intuitive and informal overview of our approach. Consider C_1 as the external face of R. Let P be a simple path from some vertex $r_1 \in C_1$ to some vertex $r_2 \in C_2$. Let R_P be the graph obtained by "cutting along P" (see Figure 2.2). Note that every shortest path in R_P corresponds to a shortest path in R that does not cross P. We will show that relaxing all edges in H_R from C_1 to C_2 with respect to distances in R_P can be done efficiently. Unfortunately, relaxing edges w.r.t. R_P does not suffice since shortest paths in R that do cross P are not represented in R_P. To overcome this obstacle we will identify two particular paths P_r and P_ℓ such that for any $u \in C_1, v \in C_2$ there exists a shortest path in R that does not cross both P_r and P_ℓ. Then, relaxing all edges between boundary vertices once in R_{P_r} and once in R_{P_ℓ} suffices to compute shortest-path distances in R. More specifically, let T be a shortest-path tree in R from r_1 to all vertices of C_2. The rightmost and leftmost paths in T satisfy the above property (see Figure 2.3).

We proceed with the formal description. In the following, we define graphs, obtained from R, required in our algorithm. It is assumed that these graphs are constructed in a preprocessing step. Later, we bound construction time.

We transform R in such a way that C_1 is the external face of R and C_2 is a hole of R. We may assume that there is a shortest path in R between every ordered pair of vertices, say, by adding a pair of oppositely directed edges between each consecutive pair of vertices of C_i in some simple walk of C_i, $i = 1, 2$ (if an edge already exists, a new edge is not added). The lengths of the new edges are chosen sufficiently large so that shortest paths in R and their lengths do not change. Where appropriate, we regard R as some fixed planar embedding of that region.

For a simple path P from a vertex $r_1 \in C_1$ to a vertex $r_2 \in C_2$, take a copy R_P of R and remove P and all edges incident to P in R_P. let $\overleftarrow{E}$ resp. $\overrightarrow{E}$ be the set of edges that either emanate left resp. right of P or enter P from the left resp. right. Refer to Section 1.2.3 for the definitions of emanates and enters. Add two copies, $\overleftarrow{P}$ and $\overrightarrow{P}$, of P to R_P. Connect path $\overleftarrow{P}$ resp. $\overrightarrow{P}$ to the rest of R_P by attaching the edges of $\overleftarrow{E}$ resp. $\overrightarrow{E}$ to the path, see Figure 2.2. If $(u, v) \in E_R$, where $(v, u) \in E_P$, we add (u, v) to both $\overleftarrow{P}$ and $\overrightarrow{P}$ in R_P.

A simple, say counter-clockwise, walk $u_1, u_2, \ldots, u_{|C_1|}, u_{|C_1|+1}$ of C_1 in R where $u_1 = u_{|C_1|+1} =$

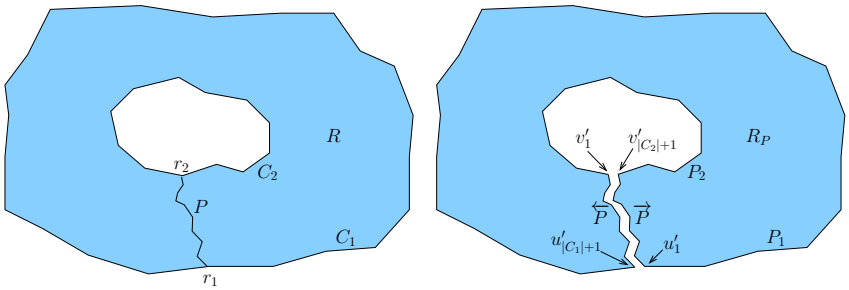

Figure 2.2: Region R_P is obtained from R essentially by cutting open at P the "ring" bounded by C_1 and C_2.

r_1 corresponds to a simple path $P_1 = u'_1, \ldots, u'_{|C_1|+1}$ in R_P. In the following, we identify u_i with u'_i for $i = 2, \ldots, |C_1|$. The vertex r_1 in R corresponds to two vertices in R_P, namely u'_1 and $u'_{|C_1|+1}$. We will identify both of these vertices with r_1. Similarly, a simple, say clockwise, walk of C_2 in R from r_2 to r_2 corresponds to a simple path $P_2 = v'_1, \ldots, v'_{|C_2|+1}$ in R_P. We make a similar identification between vertices of C_2 and P_2.

In the following, when we say that we relax all edges in R_P starting in vertices of C_1 and ending in vertices of C_2, we really refer to relaxing edges in H_R with respect to the distances between the corresponding vertices of P_1 and P_2 in R_P. More precisely, suppose we are in iteration j. Then relaxing all edges entering a vertex $v \in C_2$ in R_P means updating

$$e_j[v] \leftarrow \min_{u \in C_1} \{e_{j-1}[v], e_{j-1}[u] + d_{R_P}(u', v')\}.$$

It is implicit in this notation that if $u = r_1$, we relax w.r.t. both u'_1 and $u'_{|C_1|+1}$ and if $v = r_2$, we relax w.r.t. both v'_1 and $v'_{|C_2|+1}$. The fact that in R_P P_1 and P_2 both belong to the external face implies:

Lemma 2.2.2. *Let paths P_1 and P_2 in R_P be defined as above. Consider iteration j. Define a $|P_1| \times |P_2|$ matrix A with elements $A_{k\ell} = e_{j-1}[u_k] + d_{R_P}(u'_k, v'_\ell)$. The Matrix A is Monge.*

Proof. Since $P_1 \overleftarrow{P} P_2 \overrightarrow{P}$ is the external face of R_P, it follows by the argument in Lemma 2.1.4 that for $1 \leq k \leq k' \leq |P_1|$ and $1 \leq l \leq l' \leq |P_2|$, $A_{kl} + A_{k'l'} \geq A_{kl'} + A_{k'l}$. Hence A is Monge. □

An immediate consequence of Lemma 2.2.2 is:

Lemma 2.2.3. *Relaxing all edges from V_{C_1} to V_{C_2} in R_P can be done in $O(|C_1| + |C_2|)$ time in any iteration of Algorithm 2.3.*

Proof. Observe that relaxing all edges from V_{C_1} to V_{C_2} in R_P is equivalent to finding all column-minima of the matrix A in Lemma 2.2.2. Since A is Monge, its column-minima can be found in $O(|C_1| + |C_2|)$ time using SMAWK [AKM+87]. □

As we have mentioned, relaxing edges between boundary vertices in R_P does not suffice since shortest paths in R that cross P are not represented in R_P. Let T be a shortest-path tree in R from r_1 to all vertices of C_2. A *rightmost (leftmost)* path P in T is a path such that no other path Q in

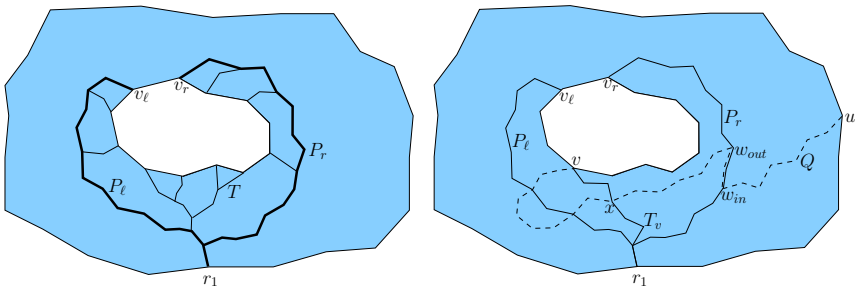

Figure 2.3: (a): The rightmost root-to-leaf simple path P_r and the leftmost root-to-leaf simple path P_ℓ in T. (b): In the proof of Lemma 2.2.4, if Q first crosses P_r from right to left and then crosses P_ℓ from right to left then there is a u-to-v shortest path in R that does not cross P_ℓ.

T emanates right (left) of P. Let P_r and P_ℓ be the rightmost and leftmost root-to-leaf simple paths in T, respectively; see Figure 2.3(a). Let $v_r \in C_2$ and $v_\ell \in C_2$ denote the leaves of P_r and P_ℓ, respectively.

Lemma 2.2.4. *For any $u \in C_1$ and any $v \in C_2$, there is a simple u-to-v shortest path in R that does not cross both P_r and P_ℓ.*

Proof. Let Q be a simple u-to-v shortest path in R which is minimal with respect to the total number of times it crosses P_r and P_ℓ. If Q does not cross P_r or P_ℓ, we are done, so assume it crosses both. Also assume that Q crosses P_r first. The case where Q crosses P_ℓ first is symmetric. Let w_{in} and w_{out} be the entry and exit vertices of the first crossing, see Figure 2.3(b). There are two cases:

- Q first crosses P_r from left to right. In this case Q must cross P_ℓ at the same vertices. In fact, it must be that all root-to-leaf paths in T coincide until w_{out} and that Q crosses them all. In particular, Q crosses the root-to-v path in T, which we denote by T_v. Since T_v does not cross P_r, the path $Q[u, w_{out}]T_v[w_{out}, v]$ is a shortest u-to-v path in R that does not cross P_r.

- Q first crosses P_r from right to left. Consider the path $S = Q[u, w_{out}]P_r[w_{out}, v_r]$. We claim that Q does not cross S. To see this, assume the contrary and let w' denote the exit point corresponding to the crossing. Since Q is simple, $w' \notin Q[u, w_{out}]$. So $w' \in P_r[w_{out}, v_r]$, but then $Q[u, w_{out}]P_r[w_{out}, w']Q[w', v]$ is a shortest path from u to v in R that crosses P_r and P_ℓ fewer times than Q. But this contradicts the minimality of Q.

  Since Q first crosses P_r from right to left and never crosses S, its first crossing with P_ℓ must be right-to-left as well, see Figure 2.3(b). This implies that Q enters all root-to-leaf paths in T before (not strictly before) it enters P_ℓ. In particular, Q enters T_v. Let x be the entry vertex. Then $Q[u, x]T_v[x, v]$ is a u-to-v shortest path in R that does not cross P_ℓ.

□

Algorithm 2.4 Initialization required for Bellman-Ford step (Algorithm 2.3)

1: **for** each region R_i
2: **for** each ordered pair of cycles C_1, C_2 in ∂R_i
3: let r_1 be an arbitrary node of C_1
4: compute a shortest-path tree in R_i rooted at r_1
5: let P_r and P_ℓ be the rightmost and leftmost paths in T
6: compute table $d^r_{R_i,C_1,C_2}$ of distances between nodes of C_1 and C_2 in $R_{i\,P_r}$ using MSSP
7: compute table $d^\ell_{R_i,C_1,C_2}$ of distances between nodes of C_1 and C_2 in $R_{i\,P_\ell}$ using MSSP

To complete the proof of Lemma 2.2.1 it only remains to show that R_{P_r} and R_{P_ℓ} and distances between boundary vertices in these graphs can be precomputed in the time required for a single shortest-path computation in R_i plus $O(|V_R| \log |V_R|)$. The procedure for computing this information, for all regions R_i is given in Algorithm 2.4. It is described below for the region R.

A shortest-path tree T in R with source r_1 can be found in $O(|V_R| \log |V_R|)$ time with Dijkstra using the recursively computed distances in R as a feasible price function ϕ. Given T, we can find its rightmost path in $O(|V_R|)$ time by starting at the root r_1. When entering a vertex v using the edge uv, leave that vertex on the edge that comes after vu in counterclockwise order. Computing R_{P_r} given P_r also takes $O(|V_R|)$ time. We can next apply Klein's algorithm [Kle05] to compute distances between all pairs of boundary vertices in R_{P_r} in $O(|V_R| \log |V_R|)$ time (here, we use the non-negative edge lengths in R defined by the reduced cost function induced by ϕ). We similarly compute P_ℓ and pairwise distances between boundary vertices in R_{P_ℓ}.

This concludes the proof of Lemma 2.2.1.

The algorithm: We can now describe our efficient implementation of the Bellman-Ford step (Algorithm 2.3). The pseudocode is given in Algorithm 2.5.

Assume that R_{P_l} and R_{P_r} and distances between pairs of boundary vertices in these graphs have been precomputed by Algorithm 2.4. In each iteration j, we relax edges from vertices of C_1 to all $v \in C_2$ in R_{P_ℓ} and in R_{P_r} (lines 9 and 10). Lemma 2.2.4 implies that this corresponds to relaxing all edges in R from vertices of C_1 to vertices of C_2. As we have discussed in the beginning of this section, this suffices to show the correctness of the algorithm.

Lemma 2.2.3 shows that lines lines 9 and 10 can each be implemented to run in $O(|C_1| + |C_2|)$ time. Thus, each iteration of lines 6–10 takes $O((|C_1| + |C_2|)\alpha(|C_1| + |C_2|))$ time, as claimed.

Algorithm 2.5 Efficient implementation of Bellman-Ford step (Algorithm 2.3)

$e_0[v] \leftarrow \infty$ for all $v \in \chi$
$e_0[r] \leftarrow 0$
for $j = 1, 2, \ldots, |\chi|$
  for each region R_i
    for each pair of cycles C_1, C_2 in ∂R_i
      if $C_1 = C_2$ **then**
        $e_j[v] \leftarrow \min_{w \in C_1}\{e_{j-1}[w] + \delta_i[w, v]\}$ for all $v \in C_2$ by Lemma 2.1.5
      else
        $e_j[v] \leftarrow \min\{e_j[v], \min_{w \in V_{C_1}}\{e_{j-1}[w] + d^r_{R_i,C_1,C_2}(w', v')\}\}$ $\forall v \in C_2$ by Lemma 2.2.3
        $e_j[v] \leftarrow \min\{e_j[v], \min_{w \in V_{C_1}}\{e_{j-1}[w] + d^\ell_{R_i,C_1,C_2}(w', v')\}\}$ $\forall v \in C_2$ by Lemma 2.2.3
$B[v] \leftarrow e_{|\chi|}[v]$ for all $v \in \chi$

Chapter 3

An Extension of FR-Dijkstra

In Section 1.3.7 we have introduced Fakcharoenphol and Rao's Monge heaps and how they are used to efficiently implement Dijkstra's algorithm (a procedure we called FR-DIJKSTRA). In this chapter we develop an extension of FR-DIJKSTRA that works with respect to reduced lengths induced by a feasible price function. This variant was first announced in [BKM+11] in the context of computing a maximum flow (see Chapter 5). The extension is achieved by using a data structure from [KMNS12].

Recall the efficient procedure to implement Dijkstra's algorithm from Section 1.3.7. Let $\{P_i\}_{i=1,2,\ldots}$ be a set of (not necessarily disjoint) subgraphs of a planar graph G. Let X_i be a set of nodes on a constant h number of faces of P_i. Let K_i be the complete graph on X_i such that the length of an arc uv in K_i corresponds to the u-to-v distance in P_i. Let $\{G_j\}$ denote the set of all bipartite graphs in the decompositions of all the $\{K_i\}$'s. Recall the discussion in Section 1.3.7, where we described the decomposition of the K_i's into bipartite graphs $\{G_j\}$ whose parent relationship is non-interleaving. Each of the bipartite graphs G_j consists of nodes on the boundary of either a single face or a pair of faces of the corresponding P_i. In the former case, the distance matrix that corresponds to G_j is Monge. In the latter case, the distance matrix M_j that corresponds to G_j is not Monge. However, using Lemma 2.2.1, we can replace the graph G_j and its distance matrix M_j with two copies of G_j, denoted G_j^1 and G_j^2, each with its own distance matrix M_j^1, M_j^2, such that both M_1 and M_2 are Monge, and such that distances computed using M are identical to those computed using both M_1 and M_2. By doing so we guarantee that the distance matrices that correspond to all of the bipartite graphs $\{G_j\}$ provided as input to FR-DIJKSTRA are Monge matrices (rather than just respecting the non-interleaving property).

In Section 1.3.7 we elaborated on our choice to expose this decomposition into bipartite subgraphs in the interface of FR-DIJKSTRA. The reason was that a range-minimum data structure needs to be computed for each row of each distance matrix of each graph in $\{G_j\}$. The time required for construct these range-minimum data structures is of the same order of the construction time of the graphs $\{G_j\}$, which is asymptotically larger than the running time of FR-DIJKSTRA.

Kaplan and Sharir [KMNS12] developed a data structure that, given a m-by-n Monge matrix M, reports the minimum in any contiguous range of any row of M in $O(\log n)$ time. The construction

time is $O(n(\log m + \log n)$ and the required space is $O(n \log n)$. In the context of FR-DIJKSTRA, each graph K_i is a complete graph on a node-set X_i. For every $0 \leq k \leq \log |X_i|$, K_i is decomposed into $O(2^k)$ bipartite graphs on $|X_i|/2^i$ nodes. Hence, constructing the Monge row-range-minimum data structure of [KMNS12] for all of the bipartite subgraphs $\{G_j\}$ takes $O(\sum |X_i| \log^2 |X_i|)$, which is of the same order of the running time of FR-DIJKSTRA. This enables us to include the construction of the range-minimum data structure in the specification of FR-DIJKSTRA. We replace the individual row range-minimum data structures used by Fakcharoenphol and Rao with one Monge row-range-minimum data structure of [KMNS12] per bipartite graph G_j. By doing so we can provide a simpler and more natural interface for FR-DIJKSTRA, where the input consists of the complete graphs $\{K_i\}$ and their corresponding distance matrices rather than their decomposition into bipartite matrices.

This change has benefits beyond just simplifying the interface. Suppose we are given the graphs K_i with their distance matrices M_i, and a feasible price function ϕ for the nodes $\bigcup X_i$ of $\bigcup K_i$. We wish to perform a Dijkstra computation in $\bigcup K_i$ with respect to the reduced lengths induced by ϕ. We would like the computation to take $O\left(\sum |X_i| \log^2(\sum |X_i|)\right)$ time.

The reduced length of an arc ab is $M_{ab} + \phi(a) - \phi(b)$. Hence, transforming a matrix M to store reduced lengths corresponds to adding $\phi(a)$ to each row a of M, and subtracting $\phi(b)$ from each column b of M. Furthermore, it follows from the definition of the Monge property (Eq. (1.4)) that if M is Monge, then it remains Monge after this change. However, actually performing the change would take $\Theta(\sum |X_i|^2)$, which is more than our desired $O\left(\sum |X_i| \log^2(\sum |X_i|)\right)$ time bound. Instead, whenever the reduced length of some element of M is required, it is computed in constant time from M and ϕ. The use of reduced lengths has another consequence. Since we add a different constant to each column of M, the results of range-minimum queries on the rows of a matrix M are not preserved, so we need to recompute the range-minimum data structures. This is only possible since the construction time of the Monge row range-minimum data structure of [KMNS12] is $O(\sum |X_i| \log^2 |X_i|)$.

We will therefore use the interface FR-DIJKSTRA$(\mathcal{K}, H, \phi, s)$ for the extended variant of FR-DIJKSTRA. Here, $\mathcal{K} = \{K_i\}$ is a set of complete graphs on the nodes X_i (as defined in Section 1.3.7). Each K_i is specified by its arc-length matrix. The graph $H = \langle V_H, E_H \rangle$ is a (not necessarily planar) graph, whose vertex set is not disjoint from the vertex set of the graphs in $\mathcal{K}$. The graph H is specified by an array associating each arc of H to its length. The function ϕ is a feasible price function, and s is the source node at which the shortest-path computation is rooted. We note that lengths of arcs in $\mathcal{K}$ and H may be negative, but the reduced lengths with respect to ϕ are all non-negative. This modified procedure internally computes, for every complete graphs $K_i \in \mathcal{K}$, the decomposition of K_i into complete bipartite graphs, and constructs the Monge row-minimum-query data structures for every such bipartite graph. It then calls the extended variant of FR-DIJKSTRA (Algoritm 1.5). The computation takes $O\left(\sum |X_i| \log^2(\sum |X_i|) + |E_H| \log(|V_H|)\right)$ time.

Chapter 4

Exact Distance Oracles for Planar Graphs

Fast shortest-path query data structures, also known as distance oracles, may be of use whenever an application needs to compute shortest-path distances between some, but not necessarily all, pairs of nodes. This is an integral part of many applications [Som10], in particular in Geographic Information Systems (GIS) and intelligent transportation systems [JHR96]. These systems may help individuals in finding fast routes or they may also assist companies in improving fleet management, plant and facility layout, and supply chain management. A particular challenge for traffic information systems or public transportation systems is to process a vast number of queries on-line while keeping the space requirements as small as possible [Zar08]. Low space consumption is obviously very important when a query algorithm is run on a system with heavily restricted memory such as a hand-held device [GW05] but it is also important for systems with memory hierarchies [HMZ03, AT05], where caching effects can have a significant impact on the query time.

While many road networks are actually not exactly planar [EG08, AFGW10], they still share many properties with planar graphs; in particular, many road networks appear to have small separators as well. For this reason, planar graphs are often used to model various transportation networks.

In Section 4.1, we describe a (strictly) linear size data structure that reports distances in sublinear time ($O(n^{1/2+\epsilon})$ time for any constant $\epsilon > 0$). The main contribution presented in this chapter is the following. Given a desired space allocation $S \in [n \log\log n, n^2]$, we show how to construct in $\tilde{O}(S)$ time a data structure of size $O(S)$ that answers distance queries in $\tilde{O}(n/\sqrt{S})$ time per query. This data structure is described in Section 4.3. This description is preceded by Section 4.2 in which we develop a technical too, which we call a cycle-MSSP (multiple-source shortest paths) data structure. Cycle-MSSP is a space-efficient extension of the MSSP data structure (see Section 1.3.4). Given a directed planar graph G on n nodes and a simple cycle C, the cycle-MSSP data structure can answer the following queries in $O(c \log^2 c \log\log c)$ time: for a query node u, output the distance in G from u to all the nodes of C. The construction time is $O(n \log^3 n)$, and the size of the data structure is

$O(n \log\log c)$. we are hopeful that cycle-MSSP would be a useful tool in other algorithms as well.[1] The results in this chapter are based on joint work with Christian Sommer [MS12].

Related Work

Distance oracles for planar graphs have been studied extensively. In the following review of previous results we focus on the space–query time tradeoff. See Figure 4.1 for a summary of known results in comparison with ours. The tradeoffs previously had not been illustrated by a space vs. query time plot as in Figure 4.1;

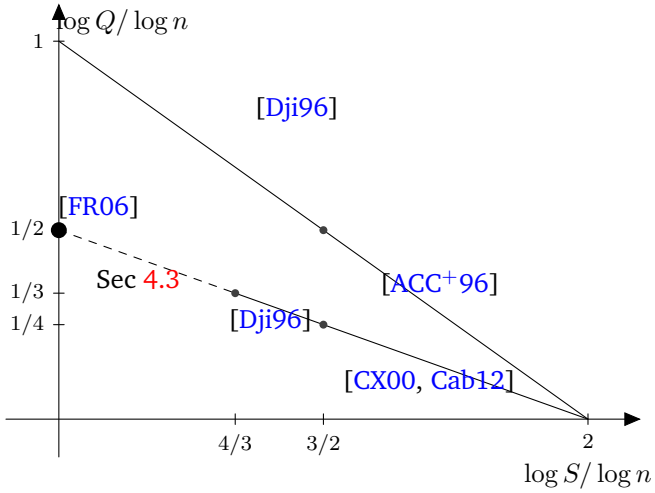

Figure 4.1: Tradeoff of the Space [S] vs. the Query time [Q] for different shortest-path query data structures on a doubly logarithmic scale, ignoring constant and logarithmic factors. The upper line represents the $Q = n^2/S$ tradeoff (completely covered by oracles of Djidjev [Dji96]; the oracles of Arikati, Chen, Chew, Das, Smid, and Zaroliagis [ACC+96] cover the range $S \in [n^{3/2}, n^2]$; SSSP ($S = n$) and APSP ($S = n^2$) also lie on this line). The lower line represents the $Q = n/\sqrt{S}$ tradeoff; the result of Djidjev [Dji96] covers the range $S \in [n^{4/3}, n^{3/2}]$; Chen and Xu [CX00] and Cabello [Cab12] extend this to $S \in [n^{4/3}, n^2]$. The data structure of Fakcharoenphol and Rao [FR06] covers the point $S = n$ and $Q = \sqrt{n}$. We extend their results to the full range.

For exact shortest-path queries, the currently best result in terms of the tradeoff between space and query time is by Fakcharoenphol and Rao [FR06]. Their data structure of space $\tilde{O}(n)$ can be constructed in time $\tilde{O}(n)$ and processes queries in time $\tilde{O}(\sqrt{n})$. The preprocessing time and space can be improved by logarithmic factors [Kle05, KMW10] ([KMW10] is concerned with single-source shortest path, and does not mention distance oracles explicitly. Its results, however, imply improvements to the distance oracle in [FR06]).

Some distance oracles have better query times. Djidjev [Dji96], improving upon earlier work of Feuerstein and Marchetti-Spaccamela [FMS91], proves that, for any $S \in [n, n^2]$, there is an exact

[1] There is at least one application for approximate distance oracles [Som11].

distance oracle with preprocessing time $O(S)$ (which increases to $O(n\sqrt{S})$ for $S \in [n, n^{3/2}]$), space $O(S)$, and query time $O(n^2/S)$. For a smaller range, he also proves that, for any $S \in [n^{4/3}, n^{3/2}]$, there is an exact distance oracle with preprocessing time $O(n\sqrt{S})$, space $O(S)$, and query time $\tilde{O}(n/\sqrt{S})$. Chen and Xu [CX00], extending the range, prove that, for any $S \in [n^{4/3}, n^2]$, there is an exact distance oracle using space $O(S)$ with preprocessing time $O(n\sqrt{S})$ and query time $\tilde{O}(n/\sqrt{S})$. Cabello [Cab12], mainly improving the preprocessing time, proves that, for any $S \in [n^{4/3}, n^2]$, there is an exact distance oracle with preprocessing time and space $O(S)$ and query time $\tilde{O}(n/\sqrt{S})$. Compared to Djidjev's construction, the query time is slower by a logarithmic factor but the range for S is larger and the preprocessing time is faster. Nussbaum [Nus11] provides a data structure that maintains both the tradeoff from [Dji96, CX00] and the fast preprocessing time from [Cab12]. Nussbaum also provides a different data structure with a "clean" tradeoff of space $O(S)$ and query time $O(n/\sqrt{S})$, however spending time $\tilde{O}(S^{3/2}/\sqrt{n})$ in the preprocessing phase. In our construction, we sacrifice another root-logarithmic factor in the query time (compared to [Cab12]) but we prove the bounds for essentially the whole range of S; our preprocessing time remains $\tilde{O}(S)$.

If constant query time is desired, storing a complete distance matrix is essentially the best solution we have. Wulff-Nilsen [WN10a] improved the space requirements to $o(n^2)$. If the space is restricted to linear, using the linear-time single-source shortest-path algorithm of Henzinger, Klein, Rao, and Subramanian [HKRS97] is the fastest known for exact shortest-path queries until the current work (Theorem 4.1.1). Nussbaum [Nus11] has simultaneously obtained a result similar to Theorem 4.1.1.

Efficient data structures for shortest-path queries have also been devised for restricted classes of planar graphs [DPZ00, CX00] and for restricted types of queries [Epp99, KK06, Sch98, Kle05]. If approximate distances and shortest paths are sufficient, a better tradeoff with $\tilde{O}(n)$ space and $\tilde{O}(1)$ query time has been achieved [Tho04, Kle02, Kle05, KKS11].

4.1 A Linear-Space Distance Oracle

In this section we describe a linear-space data structure. The techniques employed here are reused in the subsequent sections, particularly in the cycle MSSP data structure (Section 4.2).

Theorem 4.1.1. *For any directed planar graph G with non-negative arc lengths and for any constant $\epsilon > 0$, there is a data structure that supports exact distance queries in G with the following properties: the data structure can be created in time $O(n \log n)$, the space required is $O(n)$, and the query time is $O(n^{1/2+\epsilon})$.*

For non-constant $\epsilon > 0$, the preprocessing time is $O(n \log(n) \log(1/\epsilon))$, the space required is $O(n \log(1/\epsilon))$, and the query time is $O(n^{1/2+\epsilon} + n^{1/2} \log^2(n) \log(1/\epsilon))$.

Our distance oracle is an extension of the oracle in Fakcharoenphol and Rao [FR06]. The main ingredients of our improved space vs. query time tradeoff are *(i)* using recursive r–divisions instead

of cycle separators, and *(ii)* using an *adaptive* recursion,[2] where the ratio between the boundary sizes of piece at consecutive levels is $\sqrt{n}$.

We split the proof into descriptions and analysis of the preprocessing and query algorithms. Let $k = \Theta(\log(1/\epsilon))$.

Preprocessing. We compute the recursive $\boldsymbol{r}$–division of the graph with k recursive levels and values of $\boldsymbol{r} = r_0, r_1, \ldots, r_k$ to be specified below. This takes $O(kn \log n)$ time. We then compute the dense distance graph for each piece. This is done for a piece P, with r nodes and $O(\sqrt{r})$ boundary nodes on a constant number of holes, by applying the MSSP algorithm [Kle05] as described in Section 1.3.5. Thus, all of the boundary-to-boundary distances in P are computed in $O(r \log r)$ time. Summing over all $O(n/r_i)$ pieces at level i, the preprocessing time per level is $O(n \log r_i)$. The overall time to compute the dense distance graphs for all pieces over all recursive levels is therefore $O(kn \log n)$.

The space required to store DDG_P is $O((\sqrt{r})^2) = O(r)$; summing over all pieces at level i we obtain space $O(\frac{n}{r_i} r_i) = O(n)$ per level; the total space requirement is $O(kn)$.

Query. Given a query for the distance between nodes u and v, we proceed as follows. For simplicity of the presentation, we initially assume that neither u nor v are boundary nodes.

Let P_0 be the level–0 piece that contains u. We compute distances from u in P_0. This is done in $O(r_0)$ time using the algorithm of Henzinger et al. [HKRS97]. Denote these distances by $dist_{P_0}(u, w)$ for $w \in P_0$. Let H_0 denote the star graph with center u and leaves $w \in \partial P_0$. The arcs of H_0 are directed from u to the leaves, and their lengths are the corresponding distances in P_0.

Let S_u be the set of pieces that contain u. Note that S_u contains exactly one piece of each level. Let R_u be the union of subpieces of every piece in S_u. That is, $R_u = \bigcup_{P \in S_u} \{P' : P' \text{ is a subpiece of } P\}$. Let H_u be the union of the dense distance graphs of the pieces in R_u. We use FR-Dijkstra (see Section 1.3.7) to compute distances from u in $H_u \cup H_0$. Observe that any shortest path from u to a node of H_u can be decomposed into a shortest path in P_0 from u to ∂P_0 and shortest paths each of which is between boundary nodes of some piece in R_u (this is similar to Lemma 2.1.2).

Since all u-to-∂P_0 shortest paths in P_0 are represented in H_0, and since all shortest paths between boundary nodes of pieces in R_u are represented in H_u, this observation implies that distances from u to nodes of H_u in $H_u \cup H_0$ are equal to distances from u to nodes of H_u in G. We denote these distances by $dist_G(u, w)$ for nodes $w \in H_u$.

We repeat a similar procedure for v (reversing the direction of arcs) to compute $dist_G(w, v)$, the distances in G from every node $w \in H_v$ to v.

Let P_{uv} be the lowest-level piece that contains both u and v. Assume first that P_{uv} is not a level–0 piece. Let P_u (P_v) be the subpiece of P_{uv} that contains u (v). Since P_{uv} is both in S_u and in S_v, both P_u and P_v are in R_u as well as in R_v. This implies that we have already computed

[2] This appears to be the main difference to the distance oracle of Nussbaum [Nus11], which uses a non-adaptive recursion. Using our adaptive recursion is crucial whenever $S \in [n \log \log n, n \log n)$.

$dist_G(u,w)$ and $dist_G(w,v)$ for all $w \in \partial P_u$. Since we have assumed that P_{uv} is not a level–0 piece, the shortest u-to-v path must contain some node of ∂P_u. Therefore, the u-to-v distance can be found by computing

$$\min_{w \in \partial P_u} dist_G(u,w) + dist_G(w,v).$$

If P_{uv} is a level–0 piece, then $P_{uv} = P_0$, and the u-to-v distance can be found by computing

$$\min \left\{ dist_{P_0}(u,v), \min_{w \in \partial P_{uv}} \{dist_G(u,w) + dist_G(w,v)\} \right\}.$$

The case when u or v are boundary nodes is a degenerate case that can be solved by the above algorithm. Let Q_u be the highest-level piece of which u is a boundary node. We have the preprocessed distances in Q_u from u to all other nodes of ∂Q_u. Therefore, it suffices to replace S_u above with the set of pieces that contain Q_u as a subgraph in order to assure that H_u is small enough and that the distances computed by the fast implementation of Dijkstra's algorithm are the distances from u to nodes of H_u in G.

Query Time. Computing the distances $dist_{P_0}(\cdot,\cdot)$ takes $O(r_0)$ time. Let $|V(H_u)|$ denote the number of nodes of H_u. The FR-Dijkstra implementation runs in $O(|V(H_u)| \log^2 n)$ time. It therefore remains to bound $|V(H_u)|$. Let P_i be the level–i piece in S_u. P_i has $O(\frac{r_i}{r_{i-1}})$ subpieces, each with $O(\sqrt{r_{i-1}})$ boundary nodes. Therefore, the contribution of P_i to $|V(H_u)|$ is $O(\frac{r_i}{\sqrt{r_{i-1}}})$. The total running time is therefore

$$O\left(r_0 + \log^2 n \sum_{i=1}^{k} \frac{r_i}{\sqrt{r_{i-1}}} \right).$$

Recall that $r_k = |V(G)| = n$, and set $r_0 = \sqrt{n}$. For $i = 1 \ldots k-1$ we recursively define r_i so as to satisfy

$$\frac{r_i}{\sqrt{r_{i-1}}} = \sqrt{n}.$$

This implies

$$\begin{aligned} r_1 &= n^{\frac{1}{2}+\frac{1}{4}} = n^{1-\frac{1}{4}} \\ r_2 &= n^{\frac{1}{2}+\frac{3}{8}} = n^{1-\frac{1}{8}} \\ r_3 &= n^{\frac{1}{2}+\frac{7}{16}} = n^{1-\frac{1}{16}} \\ &\cdots \\ r_{k-1} &= n^{1-\frac{1}{2^k}}. \end{aligned}$$

The total running time is thus bounded by

$$\begin{aligned} O\left(\sqrt{n} + \log^2 n \left((k-1)\sqrt{n} + \frac{n}{\sqrt{n^{1-\frac{1}{2^k}}}} \right) \right) &\leq \\ O\left(\left(k\sqrt{n} + n^{\frac{1}{2}+\frac{1}{2^{k+1}}} \right) \log^2 n \right). \end{aligned} \tag{4.1}$$

By setting $k = \Theta(\log(1/\epsilon))$ we obtain the claimed running times.

4.2 A Cycle MSSP Data Structure for Planar Graphs

In this section we provide a technical tool, which is summarized in the following theorem.

Theorem 4.2.1. *Given a directed planar graph G on n nodes and a simple cycle C with $c = O(\sqrt{n})$ nodes, there is an algorithm that preprocesses G in $O(n \log^3 n)$ time to produce a data structure of size $O(n \log \log c)$ that can answer the following queries in $O(c \log^2 c \log \log c)$ time: for a query node u, output the distance from u to all the nodes of C.*

Comparison with the MSSP data structure Our data structure can be seen as an alternative to the MSSP data structure (see Section 1.3.4) with two main advantages (which we exploit in Section 4.3):

- our data structure can handle queries to any not-too-long cycle as opposed to a single face (the crucial difference and difficulty is that shortest paths may cross a cycle but not a face),
- the space requirements are only $O(n \log \log c)$ (we internally rely on the data structure in Theorem 4.1.1, so even $O(n)$ is possible at the cost of increasing the query time) as opposed to $O(n \log n)$,

and three main disadvantages:

- our data structure cannot efficiently answer queries from u to a *single* node on the cycle C; such a query requires the same time as computing the distances from u to all the nodes on C (in many existing applications, this disadvantage is not really problematic, since MSSP is used for all nodes on the face anyway),
- our data structure requires amortized time $O(\log^2 c \log \log c)$ per node on the cycle (as opposed to $O(\log n)$), which is slower for long cycles, and
- the preprocessing time of our data structure is $O(n \log^3 n)$ as opposed to $O(n \log n)$.

Let G be an embedded planar graph.

Preprocessing. Let G_0 be the exterior of C. That is, the graph obtained from G by deleting all nodes strictly enclosed by C. Consider C as the infinite face of G_0. Similarly, let G_1 be the interior of G. Namely, the graph obtained from G by deleting all nodes not enclosed by C. Consider C as the infinite face of G_1. Note that we have reduced the problem from query nodes on a cycle C to query nodes on a face. The query algorithm is supposed to handle paths that cross C. The preprocessing step consists of the following:

1. Computing DDG_C and DDG_{G-C}. This can be done in $O((n + c^2) \log n)$ time using the MSSP algorithm [Kle05]. Storing DDG_C and DDG_{G-C} requires $O(c^2) = O(n)$ space.

2. Computing an r–division of G_i ($i \in \{0,1\}$) with $r = c^2$. Each piece has $O(c^2)$ nodes and $O(c)$ boundary nodes incident to a constant number of holes. For every piece P in the r–division, consider the nodes of $C \cap P$ as boundary nodes of P. Note that each piece still has $O(c)$ boundary nodes. This step takes $O(n \log n)$ time.

3. Computing, for each piece P, a recursive $\boldsymbol{r}$–division of P as the one in the preprocessing step of the oracle in Section 4.1 (Theorem 4.1.1) with $\epsilon = 1/\log c$. That is, the number of levels in this recursive $\boldsymbol{r}$–division is $k = \Theta(\log \log c)$. The top-level (level–k) piece in this recursive division is the entire piece P. In the description in Section 4.1, the top-level piece is the entire graph and therefore it has no boundary nodes. Here, in contrast, we consider the boundary nodes of P as boundary nodes of the top-level piece in the decomposition (and thus, as boundary nodes of any lower-lever piece in which they appear). This does not asymptotically change the total number of boundary nodes at any level of the recursive decomposition since P has $O(c)$ boundary nodes, and every level of the recursive decomposition consists of a total of $\Omega(c)$ boundary nodes. The time to compute the recursive $\boldsymbol{r}$–division for all pieces is bounded by $O(n \log^2 n)$.

4. Computing, for each piece P, the dense distance graph for each of the pieces in the recursive decomposition of P. Let H_P denote the union of the dense distance graphs for all the pieces in the recursive decomposition of P. As discussed in Section 4.1, the space required to store H_P is $O(|P| \log \epsilon) = O(|P| \log \log c)$. Using the methods presented in Section 4.1, computing H_P takes $O(|P| \log |P| \log \log c) = O(c^2 \log c \log \log c)$. Thus, the total space and time required over all pieces P is $O(n \log \log c)$ and $O(n \log c \log \log c)$, respectively.

5. Computing, for each piece P, the dense distance graph comprised of all pair distances among $\partial P \cup C$ in $G_i - P$. Note that this can be thought of as $DDG_{G_i - P}$, the external dense distance graph of P, augmented with distances to the nodes of C (in $G_i - P$) as well. We will denote this dense distance graph by $DDG_{G_i - P \cup C}$. Since $|\partial P| = O(c)$, and $|C| = c$, the total number of nodes in each of these graphs is $O(c)$. These dense distance graphs can be computed top-down, as described in Section 1.3.8. The only difference is that the nodes of C are added to all external dense distance graphs for all intermediate cycle separators used to obtain the r–division. Since the bound on the number of nodes for any intermediate cycle separator is $\Omega(c)$, adding the nodes of C to all intermediate external dense distance graphs does not change the time required for the computation. As shown in Section 1.3.8, the entire computation (for all pieces combined) takes $O(n \log^3 n)$ time.

The time required for the preprocessing step is therefore $O(n \log^3 n)$ and the space required is $O(n \log \log c)$.

Query. When queried with a node u, the data structure outputs the distances from u to all the nodes of C. We describe the case where u is not enclosed by C. In this case, we use the dense

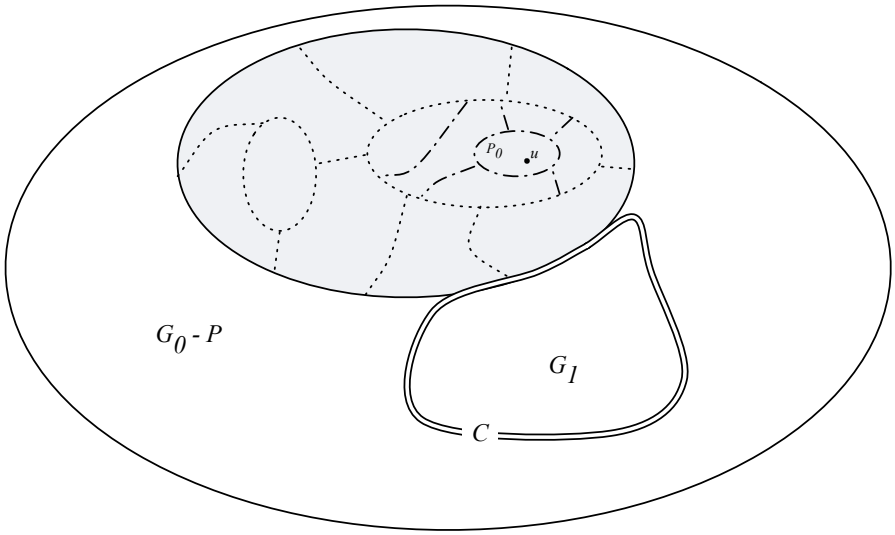

Figure 4.2: A schematic diagram showing the various subgraphs whose dense distance graphs are used in a query to the cycle MSSP data structure. The cycle C is double-lined. The interior of C is the subgraph G_1. The query node u is indicated by a small solid circle. The piece P in the r–division of the exterior of C (G_0) is shown as a grey region with solid boundary. The boundaries of the pieces whose dense distance graphs are in H_u are shown as dotted lines (one level) and dashed-dotted lines (another level). P_0 is the smallest piece of P that contains u. Any shortest path from u to C can be decomposed into a shortest path from u to ∂P_0 followed by shortest paths between nodes on the boundaries shown in the figure.

distance graphs computed in the preprocessing step for G_0. The symmetric case is handled similarly, by using the dense distance graphs computed for G_1.

Let P be the piece in the r–division of G_0 to which u belongs. Recall that P consists of $O(c^2)$ nodes. Consider the recursive $\boldsymbol{r}$–division of P computed in item 3 of the preprocessing stage. Let P_0 be the level–0 piece of P that contains u. P_0 consists of $O(\sqrt{c^2}) = O(c)$ nodes.[3]

We first compute [HKRS97], in $O(c)$ time, the distances from u to all nodes of P_0, and store them in a table $dist_{P_0}$. We then compute, using FR-Dijkstra, the distances from u in the union of the following dense distance graphs (see Figure 4.2):

1. H_0, the star graph with center u and leaves ∂P_0. The arcs of H_0 are directed from u to the leaves and their lengths are the corresponding lengths in $dist_{P_0}$.

2. H_u, the subset of dense distance graphs in H_P that correspond to pieces in the recursive decomposition of P that contain u and their subpieces. These dense distance graphs were computed in item 4 of the preprocessing step.

3. $DDG_{G_0 - P \cup C}$

4. DDG_C

[3]Here, as in Section 4.1, we assume that u is not a boundary node in the recursive $\boldsymbol{r}$–division of P. The case where u is such a boundary node is degenerate, see Section 4.1.

Note that the first two graphs are the analogs of H_0 and H_u from Section 4.1.

Distances from u in the union of the above graphs are equal to the distances from u in G. This is true since any u-to-C shortest path can be decomposed into

1. a shortest path in P_0 from u to ∂P_0.
2. shortest paths each of which is a shortest path in Q between boundary nodes of Q for some piece Q in the recursive r–division of P that is represented in H_u.
3. shortest paths in $G_0 - P$ between nodes of $\partial P \cup C$.
4. shortest paths in the interior of C between nodes of C.

To bound the running time of FR-Dijkstra we need to bound the number of nodes in all dense distance graphs used in the FR-Dijkstra computation. H_0 has $O(\sqrt{c})$ nodes. The analysis in Section 4.1 shows that the graphs in the set H_u consist of $O(\sqrt{|P|}\log\log|P|)$ nodes (substitute $k = 1/\log|P|$ in eq. (4.1)). $DDG_{G_0-P\cup C}$ has $O(c+\sqrt{|P|}) = O(c)$ nodes, and DDG_C has c nodes. Combined, the running time of the invocation of FR-Dijkstra is bounded by $O(c\log^2 c\log\log c)$. This dominates the $O(c)$ time required for the computation of $dist_{P_0}$, so the overall query time is $O(c\log^2 c\log\log c)$, as claimed.

4.3 Distance Oracles with Space $S \in [n\log\log n, n^2]$

In this section we prove the following Theorem. Using our new cycle MSSP data structure, the proof is rather straightforward.

Theorem 4.3.1. *Let G be a directed planar graph on n vertices. For any value S in the range $S \in [n\log\log n, n^2]$, there is a data structure with preprocessing time $O(S\log^3 n/\log\log n)$ and space $O(S)$ that answers distance queries in $O(nS^{-1/2}\log^2 n\log^{3/2}\log n)$ time per query.*

Proof. Let $r := (n^2\log\log n)/S$. Note that $r \in [\log\log n, n]$ for any $S \in [n\log\log n, n^2]$.

Preprocessing. We start by computing an r–division. Each piece has $O(r)$ nodes and $O(\sqrt{r})$ boundary nodes incident to a constant number of holes. For each piece P we compute the following:

1. A distance oracle as in Theorem 4.1.1 with $\epsilon = 1/\log r$. This takes $O(r\log r\log\log r)$ time and $O(r\log\log r)$ space.
2. For each hole of P (bounded by a cycle in G) we compute our new cycle MSSP data structure.[4] Since the number of holes per piece is constant, this requires requires $O(n\log^3 n)$ time and $O(n\log\log n)$ space per piece.

Summing over all pieces, the preprocessing time is $O(S\log^3 n/\log\log n)$ and the space needed is $O(S)$.

[4]Note that for $S = o(n\log n)$ we cannot even afford to store the MSSP data structure [Kle05].

Query. Given a pair of nodes (s,t), we compute a shortest s-to-t path as follows. Assume first that s and t are in different pieces. Let P denote the piece that contains s and let ∂P denote its boundary. We compute the distances in G from ∂P to t using the cycle MSSP data structures. These distances can be obtained in time $O(|\partial P| \log^2 n \log\log n) = O(\sqrt{r} \log^2 n \log\log n)$. Analogously, we compute the distances in G from s to ∂P. It remains to find the node $p \in \partial P$ that minimizes $d_G(s,p)+d_G(p,t)$, which can be done in $O(|\partial P|) = O(\sqrt{r})$ time using a simple sequential search.

If s and t lie in the same piece, the shortest s-to-t path might not visit ∂P. To account for this, we query the distance oracle for P, which takes $O(\sqrt{r} \log^2 r \log\log r)$ time. We return the minimum distance found. □

k–many distances As a consequence, we also obtain an improved algorithm for k–many distances, whenever $k = \Omega(\sqrt{n}/\log\log n)$. For some value of r to be specified below, we preprocess G in time $O((n^2/r)\log^3 n)$, and then we answer each of the k queries in time $O(\sqrt{r}\log^2 r \log\log r)$. The total time is $O((n^2/r)\log^3 n + k\sqrt{r}\log^2 r \log\log r)$. This is minimized by setting $r = n^{4/3}k^{-2/3}(\log n/\log\log n)^{2/3}$. Note that $r = O(n)$ since $k = \Omega(\sqrt{n}\log n/\log\log n)$. The total running time is thus $O((kn)^{2/3}(\log n)^{7/3}(\log\log n)^{2/3})$. This is faster than the running time in [Cab12], which is $\tilde{O}((kn)^{2/3} + n^{4/3})$, whenever $k = \Omega(\sqrt{n}/\log\log n)$.

Comparison and Discussion. The query time of our data structure is at most $O(\sqrt{r}\log^2 r \log\log r)$, which, in terms of S, is $O(nS^{-1/2}\log^2 n \log^{3/2} n \log n)$. Let us contrast this with Cabello's data structure [Cab12] that, for any $S \in [n^{4/3}\log^{1/3} n, n^2]$ has preprocessing time and space $O(S)$ and query time $O(nS^{-1/2}\log^{3/2} n)$. In our construction, we sacrifice a factor of $O(\sqrt{\log n(\log\log n)^3})$ in the query time but we gain a much larger regime for S.
For the range $S \in [\omega(n\log n/\log\log n), o(n^{4/3}\log^{1/3} n)]$, only data structures of size $O(S)$ with query time $O(n^2/S)$ had been known [Dji96] (see also Figure 4.1).

To conclude this section, let us observe what happens when we gradually decrease the space requirements S from n^2 down to n, and, *en passant*, let us pose some open questions. For quadratic space (or even slightly below [WN10a]), we can obtain constant query time. As soon as we require the space to be $O(n^{2-\epsilon})$ for some $\epsilon > 0$, the query time increases from constant or poly-logarithmic to a polynomial. It is currently not known whether $\tilde{O}(1)$ is possible or not — the known lower bounds [SVY09, PR10] on the space of distance oracles work for non-planar graphs only. Further restricting the space, as long as the space available is at least $\Omega(n\log n)$, the data structure can internally *store* MSSP data structures. The query time for this regime of S can actually be made slightly faster than what we claim in Theorem 4.3.1 (by avoiding the $O(\log\log n)$–factor due to the recursion needed in Theorem 4.1.1 and its manifestation in the cycle MSSP data structure). The data structure of Nussbaum [Nus11] obtains a "clean" tradeoff with query time proportional to $O(n/\sqrt{S})$ for $S \geq n^{4/3}$ (*without* logarithmic factors in the query time, currently at the cost of a slower preprocessing algorithm). The obvious open question is whether it is possible to obtain a data structure with space $O(S)$ and query time $O(n/\sqrt{S})$ for the whole range of $S \in [n, n^2]$ and without

substantial sacrifices with respect to the preprocessing time. Another open question is whether it is possible to improve upon this tradeoff. Note that, for quadratic space, an improvement of an (almost) logarithmic factor is possible [WN10a].

As soon as we require the space to be $o(n \log n)$, we cannot afford to store the MSSP data structure anymore and we are currently forced to rely on the cycle MSSP data structure (Theorem 4.2.1). The query time increases to the time bound claimed in Theorem 4.3.1. When we further restrict the space to $o(n \log \log n)$, say $S = \Theta(n \log(1/\epsilon))$ for some $\epsilon > 0$, the query time increases to $O(n^{1/2+\epsilon})$. A different recursion (maybe à la [HKRS97]) could potentially reduce this to $\tilde{O}(\sqrt{n})$.

Let us briefly consider the *additional space* of the data structure, assuming that storing the graph is free. It is known that, for *approximate* distances, a query algorithm can run efficiently using a data structure that occupies sublinear additional space [KKS11]. For exact distances and sublinear space, nothing better than the linear-time SSSP algorithm [HKRS97] is known.

Part III

Maximum Flow

Introduction

The two algorithms presented in Chapters 5 and 6 solve the maximum-flow problem in directed planar graphs with multiple sources or sinks (we shall refer to instances with multiple sources and sinks by the initials MSMS). In general (i.e., non-planar) graphs, the multiple sources and sinks case can be reduced to the single-source single-sink case by introducing an artificial source and sink and connecting them to all the sources and sinks, respectively—but this reduction does not preserve planarity. Therefore, algorithms for computing maximum st-flow (i.e., when there is a single source s and a single sink t) that work in general graphs can solve MSMS max-flow in planar graphs using this reduction. Indeed, prior to our work, this was the fastest known algorithm for planar MSMS max-flow.

For a survey of the early history of the maximum flow problem in general graphs, see Schrijver [Sch05]. For more recent history, see [GTT90, Gol98]. Using the max-flow algorithm of Sleator and Tarjan [ST83] leads to a running time of $O(n^2 \log n)$ for MSMS max-flow in planar graphs. For integer capacities less than U, one could instead use the algorithm of Goldberg and Rao [GR98], which leads to a running time of $O(n^{1.5} \log n \log U)$. Using the recent algorithm of Christiano, Kelner, Mądry, Spielman and Teng [CKM+11] leads to a running time of $O(n^{4/3}\epsilon^{-11/3})$ for obtaining a $(1-\epsilon)$ approximate solution.

In planar graphs, however, algorithms for maximum st-flow do not immediately imply algorithms for MSMS max-flow. It is to be expected, though, that the techniques and algorithms developed for maximum st-flow and min st-cuts in planar graphs are very relevant in solving planar MSMS max-flow. We have already discussed some of these techniques in Section 1.4. See Borradaile's Ph.D. thesis [Bor08] for the history of planar maximum st-flow. Here we only mention the currently best upper bounds. In undirected planar graphs, maximum st-flow and minimum st-cut can both be solved in $O(n \log \log n)$ [INSWN11]. In directed st-planar graphs, Hassin's algorithm [Has81] (see Section 1.4.2) can be implemented in linear time using the linear-time shortest-path algorithm [HKRS97]. In directed planar graph, maximum st-flow (and thus also minimum st-cut) can be solved in $O(n \log n)$ time [BK09, Eri10].

MSMS max-flow in planar graphs was studied by Miller and Naor [MN95]. When it is known how much of the flow is produced/consumed at each source and each sink, finding a consistent routing of flow that respects arc capacities can be reduced to *negative-length shortest paths*, which, as we saw in Chapter 2, can be solved in planar graphs in $O(n \log^2 n / \log \log n)$ time. Otherwise, Miller and Naor gave an $O(n \log^{3/2} n)$ algorithm for the case where all the sinks and the sources are on the boundary of a single face, and generalized it to an $O(k^2 n^{3/2} \log^2 n)$-time algorithm for the case where the sources and the sinks reside on the boundaries of k different faces.[5]

However, the problem of maximum flow with multiple sources and sinks in planar graphs without any additional restrictions remained open. For more than twenty years no planarity exploiting

[5] The time bound of the first algorithm can be improved to $O(n \log n)$ using the linear-time shortest-path algorithm of Henzinger et al. [HKRS97], and the time bound of the second algorithm can be improved to $O(k^2 n \log^2 n)$ using the $O(n \log n)$-time maximum st-flow algorithm of Borradaile and Klein [BK09].

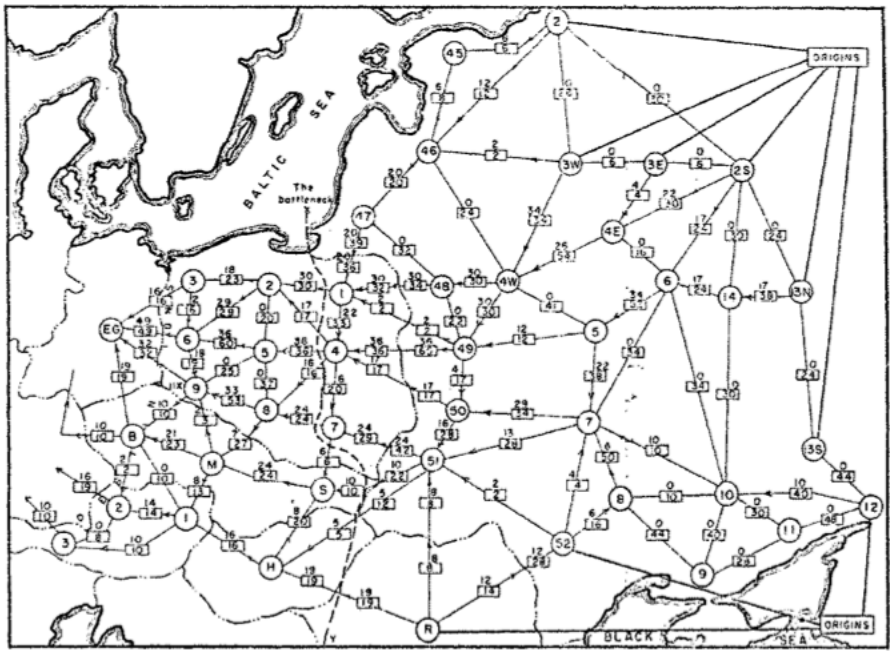

Figure 4.3: The soviet rail network from [HR55]

algorithms were known. In Chapter 5 we describe an $O(n \log^3 n)$-time algorithm for planar MSMS max-flow algorithm [BKM+11]. In Chapter 6 we describe a less efficient algorithm for the special case of multiple sources and a single sink [KM11]. While the latter algorithm is dominated by the former, it uses a novel interesting technique that may be useful in related problems.

It was pointed out to us by Yahav Nussbaum that a planar MSMS max-flow algorithm can be used to improve the running time of solving the planar MSMS max-flow problem with vertex capacities, where the number of vertices with finite capacity is moderate ($o(n^{1/3})$). This is done by plugging our MSMS max-flow algorithm into the algorithm of Hochstein and Weihe [HW07] for maximum st-flow in graphs with a bounded number of crossings.

Applications

Below, we discuss a few applications of MSMS max-flow in planar graphs. Ford and Fulkerson, who worked at RAND, were motivated by a classified memo of Harris and Ross [HR55] on interdiction of the Soviet railroad system. This report was downgraded to unclassified in 1999. It contains a diagram of a network that models the Soviet railroad system indicating sources ("ORIGINS") in the Soviet Union and sinks (destinations) in Eastern European satellite countries (Poland, Czechoslovakia, Austria, Eastern Germany).

Note that, Harris and Ross were interested in the minimum cut, not the maximum flow, as is the

case with most other applications. However, by the duality between maximum- flow and minimum-cut [FF56], given a maximum flow in G, one can find the minimum cut in linear time. Simply identify the connected components of the sources in the residual network.

Application to computer vision problems Multiple-source multiple-sink min-cut arises in addressing a family of problems associated with the terms *metric labeling* [KT02], *Markov Random Fields* [GG84], and *Potts Model* (see also [BVZ01, Hoc01]). In low-level vision problems such as *image restoration*, *segmentation*, *stereo*, and *motion*, the goal is to assign labels from a set to pixels so as to minimize a penalty function. The penalty function is a sum of two parts. One part, the *data component*, has a term for each pixel; the cost depends on the discrepancy between the observed data for the pixel and the label chosen for it. The other part, the *smoothing component*, penalizes neighboring pixels that are assigned different labels.

For the *binary* case (when the set of available labels has size two), finding the optimal solution is reducible to multiple-source multiple-sink min-cut. [GPS89].

For the case of more than two labels, there is a powerful and effective heuristic [BVZ01] using very-large-neighborhood [AEOP02] local search; the inner loop consists of solving the two-label case. The running time for solving the two-label case is therefore quite important. For this reason, researchers in computer vision have proposed new algorithms for max flow and done experimental studies comparing the run-times of different max-flow algorithms on the instances arising in this context (e.g. [BK04]). For the (common) case where the underlying graph of pixels is the two-dimensional grid graph, our result in Chapter 5 yields a theoretical speed-up for this important computer-vision subroutine.[6]

Hochbaum [Hoc01] describes a special case of the penalty function in which the data component is convex and the smoothing component is linear; in this case, she shows that an optimal solution can be found in time $O(T(m,n) + n \log U)$ where U is the maximum label, and $T(m,n)$ is the time for finding a minimum cut. She mentions specifically image segmentation, for which the graph is planar. For this case, by using our algorithm (Chapter 5), the optimal solution can be found in nearly linear time.

Application to maximum bipartite matching It is well known that finding a maximum matching in a bipartite planar graph can be reduced to computing multiple-source, multiple-sink maximum flow. Our result is the first planarity-exploiting algorithm for this problem (and the first near-linear one).

Application to finding edge disjoint paths The problem of finding edge disjoint paths may be thought of as a generalization of bipartite matching. Given two sets S, T, the goal is to find a maximum cardinality set of edge disjoint paths between nodes in S and nodes in T, such that each

[6] Note that the single-source, single-sink max-flow algorithm of [BK09] was implemented by computer-vision researchers [STC09] and found to be useful in computer vision and to be faster than the competitors.

node in S is paired with a distinct node in T. This problem can be reduced to MSMS max flow by attaching a source to each node of S and a sink to each node of T, and setting the capacities of all edges to one.

The problem arises in routing printed circuit boards and multiple chip modules [HS97], where it is known as the *breakout routing problem*, in ball grid array packages [YD95], where it is called the *fanout routing problem*, and in VLSI/WSI processor arrays in the presence of faulty processors [RBK90], where it is known as the *reconfiguration problem*. See also [CC00]. In a typical application the underlying graph is a rectangular n-by-m grid, and the number of sources is N. The sinks are usually located at the boundary of the grid. It can be shown [CC00] that without loss of generality $n, m \leq 2N$. The fastest algorithm that uses MSMS max-flow to solve this problem [CCT99] runs in $O(N^{9/4})$, whereas our algorithm would solve the problem in $O(N^2 \log^3 N)$ time. However, for grid graphs, a $O(N^2)$-time algorithm that does not use MSMS max-flow is known [CCT03]. Note that the situation where the terminals are all on the boundary of the grid is essentially the multiple sources single sink case (Chapter 6). It is also worth mentioning that our algorithms cannot handle the node disjoint paths problem since the resulting flow problem requires handling node capacities.

Application to flow in polyhedral domains Mitchell [Mit90] has shown how to reduce the maximum flow problem in a uniformly capacitated polyhedral region in two dimensions, with arbitrary sources and sinks to a maximum flow problem in a planar graph with multiple sources and sinks. Let n denote the number of vertices of the polyhedral domain. The corresponding planar graph has $O(n^4)$ edges. Using our result implies a running time of $O(n^4 \log^3 n)$ compared with the $O(n^8 \log n)$ time in [Mit90].

Chapter 5

Maximum Flow in Directed Planar Graphs with Multiple Sources and Sinks

Consider the following recursive approach for finding a max flow: split the input graph G in two using an $O(\sqrt{n})$ simple cycle separator C ([Mil86], see Section 1.2.6) and recursively solve the max flow problem in each of the two subgraphs. After the recursive calls, each subgraph contains no residual source-to-sink path. We would like to additionally ensure that there is no residual source-to-C/C-to-sink paths in order to ensure that G contains no source-to-sink residual path.

In order to achieve this, we sacrifice conservation; we allow nonzero inflow at nodes of the separator C. To address this, we introduce a new subproblem: pushing flow between the nodes of C so as to ensure that there is no residual path from a positive-inflow node of C to a negative-inflow node of C. Our algorithm for solving this problem (Section 5.3) deals with one node of C at a time. For each node, the algorithm moves flow between that node of C and the others to achieve the following: (i) the node has zero inflow, or (ii) the node has positive inflow but there is no residual path from that node to nodes of C with negative inflow, or (iii) the node has negative inflow but there is no residual path to that node from nodes of C with positive inflow.

Our algorithm for achieving this uses an implicit representation of the flow. This enables it to carry out each iteration in $O(\sqrt{n}\log^2 n)$ time by computing shortest paths in the planar dual using the extension of Fakcharoenphol and Rao's implementation of Dijkstra's algorithm described in Chapter 3. Thus the total time for solving this subproblem is $O(n\log^2 n)$, leading to an $O(n\log^3 n)$ running time for the main algorithm.

5.1 The Algorithm

Now we describe the algorithm, MSMSMaxFlow, in more detail. In order to treat nodes of the cycle separator both as sources and as sinks in recursive calls, we introduce a new node set A of size at most 6. To solve the original problem, one uses $A := \emptyset$.

MSMSMaxFlow($G, \boldsymbol{c}, S, T, A$)
Input: a directed planar graph G with non-negative capacities $\boldsymbol{c}$, a set S of source nodes, a set T of sink nodes, a set A of at most six nodes
Output: a pseudoflow $\boldsymbol{f}$ obeying conservation everywhere except S, T, A and such that $S \overset{G_{\boldsymbol{f}}}{\not\to} T$, $S \overset{G_{\boldsymbol{f}}}{\not\to} A$, $A \overset{G_{\boldsymbol{f}}}{\not\to} T$.

The algorithm finds a size-$O(\sqrt{n})$ simple cycle separator C. Following, e.g., [KS98, FR06], The separator is chosen to ensure each subgraph has at most two-thirds the nodes of G (if $|A| < 6$) or at most two-thirds the nodes of A (if $|A| = 6$). If C contains a source u, introduce an artificial node u' and artificial arc $u'u$ with high capacity, and designate u' a source instead of u. Sinks on C are handled similarly. From now on, we assume for simplicity of presentation that no nodes of C are sources or sinks.

The algorithm contracts all edges of C except one. This merges all the nodes of C into a single supernode v, and turns C into a self-loop. The algorithm then recursively solves the problem on the subgraph embedded on each side of the self-loop, adding v to the set A.

After the recursive calls, the algorithm uncontracts the edges of C. At this stage there are no residual paths between sources and sinks in the entire graph, but there might be positive or negative inflow at nodes of C. The algorithm then calls the procedure FixConservationOnPath, which pushes flow between the nodes of C so that there are no residual paths between nodes of C with positive inflow and nodes of C with negative inflow (the path in the name of the procedure is the cycle C minus one edge). This procedure is discussed in Section 5.3; the interface is:

FixConservationOnPath($G, P, \boldsymbol{c}, \boldsymbol{f}_0$)
Input: a directed planar graph G, a simple path P, a capacity function $\boldsymbol{c}$, and a pseudoflow $\boldsymbol{f}_0$
Output: a pseudoflow $\boldsymbol{f}$ such that $\boldsymbol{f} - \boldsymbol{f}_0$ satisfies conservation everywhere but P, and $\{v \in P : \text{inflow}(v) > 0\} \overset{G_{\boldsymbol{f}}}{\not\to} \{v \in P : \text{inflow}(v) < 0\}$.
Running Time: $O(n \log n + |P|^2 \log^2 n)$

Next, the algorithm iterates over the nodes a_i of A. The algorithm calls a procedure CycleToSinkMaxFlow that pushes as much excess flow as possible from C to a_i. Let C_i^+ be the set of nodes of C that are reachable via residual paths from some node of C with positive inflow at the beginning of iteration i. CycleToSinkMaxFlow pushes flow among the nodes of C_i^+ and from the nodes of C_i^+ to a_i so that the inflow at every node of C_i^+ remains non-negative and there are no residual paths to a_i from positive-inflow nodes of C. The interface is:

CYCLETOSINKMAXFLOW($G, \boldsymbol{c}, \boldsymbol{f}_0, C, t$)
Input: a directed planar graph G with capacities $\boldsymbol{c}$, a pseudoflow $\boldsymbol{f}_0$, a simple cycle C, a sink t.
Assumes: $\forall v \in C^+$, $\text{inflow}_{\boldsymbol{f}_0}(v) \geq 0$, where $C^+ = \{v \in C : \{x \in C : \text{inflow}_{\boldsymbol{f}_0}(x) > 0\} \overset{G_{\boldsymbol{f}_0}}{\to} v\}$.
Output: a pseudoflow $\boldsymbol{f}$ s.t. *(i)* $\boldsymbol{f} - \boldsymbol{f}_0$ obeys conservation everywhere but $C^+ \cup \{t\}$, *(ii)* $\forall v \in C^+$, $\text{inflow}_{\boldsymbol{f}}(v) \geq 0$, *(iii)* $\{v \in C : \text{inflow}_{\boldsymbol{f}}(v) > 0\} \overset{G_{\boldsymbol{f}}}{\not\to} t$.
Running Time: $O(n \log n + |C|^2 \log^2 n)$.

A symmetric procedure, SOURCETOCYCLEMAXFLOW, is called to push flow from a_i to C to reduce the amount of negative inflow. Finally, the algorithm pushes flow back from any nodes of C with positive inflow to S and pushes flow back from T into any nodes of C with negative inflow. This is done using the procedure described in Section 1.4.3.

Algorithm 5.1 MSMSMAXFLOW($G, \boldsymbol{c}$,S,T,A)
Input: a directed planar graph G with non-negative capacities $\boldsymbol{c}$, a set S of source nodes, a set T of sink nodes, a set $A = \{a_1, \ldots, a_k\}$ of size at most six.
Output: a pseudoflow $\boldsymbol{f}$ obeying conservation everywhere except $S \cup T \cup A$, such that $S \overset{G_{\boldsymbol{f}}}{\not\to} T$, $S \overset{G_{\boldsymbol{f}}}{\not\to} A$, $A \overset{G_{\boldsymbol{f}}}{\not\to} T$.

add zero-capacity arcs to triangulate and two-connect G (required for simple cycle separators)
find a cycle separator C in G
ensure that C contains no sources or sinks
$P \leftarrow C - \{e\}$, where e is one edge of C
contract all the edges of P, turning e into a self loop incident to the only remaining node v of C
let G_1 and G_2 be the subgraphs of G enclosed and not enclosed by e, respectively
$\boldsymbol{f} \leftarrow$ MSMSMAXFLOW($G_1, \boldsymbol{c}_{|G_1}, S \cap G_1, T \cap G_1, (A \cap G_1) \cup \{v\}$)
$\boldsymbol{f} \leftarrow \boldsymbol{f} +$ MSMSMAXFLOW($G_2, \boldsymbol{c}_{|G_2}, S \cap G_2, T \cap G_2, (A \cap G_2) \cup \{v\}$)
uncontract the edges of P
$\boldsymbol{f} \leftarrow$ FIXCONSERVATIONONPATH($G, P, \boldsymbol{c}, \boldsymbol{f}$)
for $i = 1, 2, \ldots, k$
    $\boldsymbol{f} \leftarrow$ CYCLETOSINKMAXFLOW($G, \boldsymbol{c}, \boldsymbol{f}, C, a_i$)
    $\boldsymbol{f} \leftarrow$ SOURCETOCYCLEMAXFLOW($G, \boldsymbol{c}, \boldsymbol{f}, a_i, C$)
push positive excess from C to S and negative excess to C from T
return $\boldsymbol{f}$

5.1.1 Running Time Analysis

The number of nodes of the separator cycle C used to partition G into G_1 and G_2 is $O(\sqrt{|G|})$. Therefore, each invocation of FIXCONSERVATIONONPATH, CYCLETOSINKMAXFLOW and SOURCETOCYCLEMAXFLOW in G takes $O(|G| \log |G| + |C|^2 \log^2 |C|) = O(|G| \log^2 |G|)$ time. A standard analysis of the recurrence (see, e.g., [FR06]) shows that the algorithm runs in $O(n \log^3 n)$ time.

5.2 Correctness of MSMSMaxFlow

To prove the correctness of MSMSMaxFlow, we prove that each step of the algorithm eliminates some residual paths without reintroducing residual paths that were already eliminated. The proofs rely on the sources and sinks lemmas from Section 1.4.

Lemma 5.2.1. *At any time in the running of the algorithm after the last execution of line 8 and before the execution of line 14,* $S \nrightarrow T$, $S \nrightarrow A$, $S \nrightarrow C$, $A \nrightarrow T$, $C \nrightarrow T$.

Proof. The properties hold just after line 8 by the properties of the recursive calls and by the fact that any residual path between G_1 and G_2 consists of a residual path to C and a residual path from C. Note that, because of the contractions in line 5, at this time the cycle C consists of just the node v. The nonexistence of residual paths w.r.t. v in the recursive calls implies the nonexistence of residual paths w.r.t. any node of C after the contractions are undone in line 9.

We next prove that the properties are maintained until just before the execution of line 14. By the above argument, there is a cut separating S from $T \cup A \cup C$ that is saturated just after line 9. The procedure in line 10 only pushes flow between vertices of C, and in lines 11– 13, flow is only pushed between the nodes of A and C. Since these sets are all on the same side of the cut, the cut stays saturated. It follows that $S \nrightarrow \{T, A, C\}$ at any point after line 8 and before line 14. A similar argument applied to a saturated cut separating $A \cup C$ from T shows $A \nrightarrow T$ and $C \nrightarrow T$. □

Recall that C_i^+ is the set of nodes of C that are reachable via residual paths from some node of C with positive excess at the beginning of iteration i of the loop in line 11, and that C_i^- is the set of nodes of C that have residual paths to some node of C with negative excess at that time.The next lemma follows from the definition of FixConservationOnPath.

Lemma 5.2.2. *Just after line 10 is executed,* $C_1^+ \nrightarrow C_1^-$.

Lemma 5.2.3. *For all* $i < j$, $C_j^+ \subseteq C_i^+$ *and* $C_j^- \subseteq C_i^-$

Proof. It suffices to show that, for all i, $C_{i+1}^+ \subseteq C_i^+$ and $C_{i+1}^- \subseteq C_i^-$. The flow pushed in iteration i of line 12 can be decomposed into a flow whose sources and sinks are all in C_i^+ and a flow whose sources are in C_i^+ and whose sink is a_i. Therefore, the set X of nodes of C with positive inflow immediately after iteration i of line 12 is a subset of C_i^+. By definition of SourceToCycleMaxFlow, the set of nodes of C with positive inflow does not change after line 13 is executed. Therefore, C_{i+1}^+ is the set of nodes reachable from X by a residual path after iteration i of line 13.

By definition of C_i^+, immediately before iteration i of line 12 there are no C_i^+-to-$(C - C_i^+)$ residual paths. By *sources lemma*$(C_i^+, C - C_i^+, C_i^+)$, there are no C_i^+-to-$(C - C_i^+$ residual paths immediately after iteration i of line 12 as well. This shows that there are no X-to-$(C - C_i^+)$ residual paths at that time.

The flow pushed in line 13 can be decomposed into a flow whose sources and sinks are all in C_i^- and a flow whose source is a_i and whose sinks are all in C_i^-. By *sinks lemma*$(C_i^+, C - C_i^+, C_i^-)$,

there are no C_i^+-to-$(C - C_i^+)$ residual paths immediately after iteration i of line 13. This shows that there are no X-to-$(C - C_i^+)$ residual paths at that time. Hence $C_{i+1}^+ \subseteq C_i^+$, as desired.

The proof of the analogous claim for C_{i+1}^- is similar. □

Lemma 5.2.4. *Just after line 12 is executed in iteration i, $C_i^+ \nrightarrow C_i^-$, $C_{i+1}^+ \nrightarrow \{a_j\}_{j \le i}$, $\{a_j\}_{j<i} \nrightarrow C_i^-$.*

Proof. The flow pushed in line 12 can be decomposed into a flow whose sources and sinks are all in C_i^+ and a flow whose sources are in C_i^+ and whose sink is a_i.

- $C_i^+ \nrightarrow C_i^-$ by *sources lemma*(C_i^+, C_i^-, C_i^+)
- $C_{i+1}^+ \nrightarrow \{a_j\}_{j<i}$ by *sources lemma*(C_{i+1}^+, a_j, C_i^+)
- $C_{i+1}^+ \nrightarrow \{a_i\}$ by definition of CycleToSinkMaxFlow
- $\{a_j\}_{j<i} \nrightarrow C_i^-$ by *sources lemma*(a_j, C_i^-, C_i^+)

□

The following lemma is proved similarly.

Lemma 5.2.5. *Just after line 13 is executed in iteration i, $C_i^+ \nrightarrow C_i^-$, $C_{i+1}^+ \nrightarrow \{a_j\}_{j \le i}$, $\{a_j\}_{j \le i} \nrightarrow C_{i+1}^-$.*

Let C^+ (C^-) denote the set of nodes in C with positive (negative) inflow just before line 14 is executed.

Corollary 5.2.6. *Just before line 14 is executed, $C^+ \nrightarrow C^-, C^+ \nrightarrow A, A \nrightarrow C^-$.*

Finally, the next lemma proves the algorithm is correct.

Lemma 5.2.7. *The following are true upon termination:*

1. *$\boldsymbol{f}$ is a pseudoflow*
2. *$\boldsymbol{f}$ obeys conservation everywhere except at S, T, A*
3. *$S \overset{G_{\boldsymbol{f}}}{\nrightarrow} T, S \overset{G_{\boldsymbol{f}}}{\nrightarrow} A, A \overset{G_{\boldsymbol{f}}}{\nrightarrow} T$.*

Proof. Since every addition to $\boldsymbol{f}$ along the algorithm respects capacities of all darts, $\boldsymbol{f}$ is a pseudoflow at all times. By induction, the only nodes that do not obey conservations after the recursive calls are those of S, T and A. Subsequent changes to $\boldsymbol{f}$ only violate conservation on the nodes of C, but any such violation is eliminated in line 14. Therefore upon termination $\boldsymbol{f}$ obeys conservation everywhere except S, T and A.

By Lemma 5.2.1 and Corollary 5.2.6, before line 14 is executed, $C^+ \nrightarrow A$ and $C \nrightarrow T$. Therefore the flow pushed back from C^+ in line 14 is pushed back to S. Similarly, the flow pushed back to C^- is pushed back from T. Let f_+ (f_-) be the flow pushed back from C^+ to S (from T to C^-) in line 14. Consider first pushing back f_+.

- $S \nrightarrow T$ by *sources lemma*(S, T, C^+)
- $S \nrightarrow A$ by *sources lemma*(S, A, C^+)
- $A \nrightarrow T$ by *sources lemma*(A, T, C^+)
- $S \nrightarrow C^-$ by *sources lemma*(S, C^-, C^+)
- $A \nrightarrow C^-$ by *sources lemma*(A, C^-, C^+)

Next consider pushing f_-

- $S \nrightarrow T$ by *sinks lemma*(S, T, C^-)
- $S \nrightarrow A$ by *sinks lemma*(S, A, C^-)
- $A \nrightarrow T$ by *sinks lemma*(A, T, C^-)

□

5.3 Eliminating residual paths between nodes on a path

In this section we present an efficient implementation of the fixing procedure which, roughly speaking, given a path with nodes having positive, negative, or zero inflow, pushes flow between the nodes of the path so that eventually there are no residual paths from nodes with positive inflow to nodes with negative inflow.

We begin by describing an abstract algorithm for the fixing procedure (Algorithm 5.2). It resembles a technique used by Venkatesan and Johnson [JV82]. Let M be the sum of capacities of all of the darts of G. The algorithm first increases the capacities of darts of the path P and their reverses by M. Let $p_1, p_2, \ldots, p_{k+1}$ be the nodes of P. The algorithm processes the nodes of P one after the other. Processing p_i consists of decreasing the capacities of $d_i = p_i p_{i+1}$ and $rev(d_i)$ by M (i.e., back to their original capacities), and trying to eliminate positive inflow w at p_i by pushing at most w units of flow from p_i to p_{i+1}. After the push, either the flow obeys conservation at p_i or there are no residual paths from p_i to any of the other nodes of P (this is where we use the large capacities on the darts between unprocessed nodes). Negative inflow at p_i is similarly handled by pushing flow from p_{i+1} to p_i.

5.3.1 Correctness of Algorithm 5.2

Lemma 5.3.1. *The following holds immediately after iteration i of the main loop (line 3).*

1. *For $j \leq i, j' > j$, if p_j has positive inflow, there is no residual path from p_j to $p_{j'}$. If p_j has negative inflow, there is no residual path from $p_{j'}$ to p_j.*

Algorithm 5.2 ABSTRACTFIXCONSERVATIONONPATH$(G, P, \boldsymbol{c}, \boldsymbol{f}_0)$

Input: directed planar graph G, simple path $P = d_1 d_2 \dots d_k$, capacity function $\boldsymbol{c}$, pseudoflow $\boldsymbol{f}_0$
Output: a pseudoflow $\boldsymbol{f}'$ s.t. *(i)* $\boldsymbol{f}' - \boldsymbol{f}_0$ satisfies conservation at nodes not on P, and *(ii)* with respect to $\boldsymbol{f}'$, there are no residual paths from nodes of P with positive inflow to nodes of P with negative inflow.

1: $\boldsymbol{f}' \leftarrow \boldsymbol{f}_0$
2: $\boldsymbol{c}[d] \leftarrow \boldsymbol{c}[d] + M$ for all darts d of $P \cup \text{rev}(P)$
3: **for** $i = 1, 2, \dots, k$
4: let p_i and p_{i+1} be the tail and head of d_i, respectively
// reduce the capacities of d and rev(d) by M and adjust the flow appropriately
5: **for** $d \in \{d_i, \text{rev}(d_i)\}$
6: $\boldsymbol{c}[d] \leftarrow \boldsymbol{c}[d] - M$
7: $\boldsymbol{f}'[d] \leftarrow \min\{\boldsymbol{f}'[d], \boldsymbol{c}[d]\}$
8: $\boldsymbol{f}'[\text{rev}(d)] \leftarrow -\boldsymbol{f}'[d]$
9: *excess* $\leftarrow$ inflow at p_i
10: **if** $excess > 0$ **then** $d \leftarrow d_i$ **else** $d \leftarrow \text{rev}(d_i)$ find in which direction flow should be pushed
11: add to $\boldsymbol{f}'$ a maximum feasible flow from tail(d) to head(d) with limit *excess*
12: **return** $\boldsymbol{f}'$

2. *For $j, j' \le i$, if p_j has positive inflow and $p_{j'}$ has negative inflow then there is no p_j-to-$p_{j'}$ residual path.*

Proof. By induction on the number of iterations i of the loop. For $i = 0$ the invariants are trivially satisfied.

First note that the adjustments to capacities and flow in lines 6–8 do not create any new residual paths since capacities are only reduced, and no residual capacity increases. Therefore, it suffices to argue just about the flow pushed in line 11.

Assume the invariants hold up until the beginning of the i^{th} iteration. We show that the invariants hold at the end of the iteration. Suppose that p_i has positive inflow at the end of the i^{th} iteration (the case of negative inflow is similar).

1. Since the flow pushed in line 11 is limited by $|\text{inflow}(p_i)|$, the fact that p_i has positive inflow at the end implies that the flow pushed was a maximum flow from p_i to p_{i+1}. Since the capacities of darts between p_{k+1} and p_k for $k > i$ are sufficiently large, the maximality of the flow implies that there are no p_i-to-p_k residual paths for any $k > i$.

   The invariant holds for nodes p_j with positive inflow and $j < i$ by *sinks lemma*$(\{p_j\}, \{p_{j'} : j' > j\}, \{p_{i+1}\})$.

2. Invariant 2 holds for $j, j' < i$ by
   sinks lemma$(\{p_j : j < i, \text{inflow}(p_j) > 0\}, \{p_j : j < i, \text{inflow}(p_j) < 0\}, \{p_{i+1}\})$.

   The invariant holds for p_i by invoking the *sources lemma*$(\{p_i\}, \{p_j : j < i, \text{inflow}(p_j) < 0\}, \{p_i\})$.

□

Corollary 5.3.2. *Algorithm 5.2 is correct*

Proof. The flow $\boldsymbol{f}' - \boldsymbol{f}_0$ satisfies conservation everywhere except at nodes of P since the algorithm only pushes flow between nodes of P. By Lemma 5.3.1, with respect to $\boldsymbol{f}'$, there are no residual paths from nodes of P with positive inflow to nodes of P with negative inflow. □

5.3.2 An Inefficient Implementation

Algorithm 5.3 Inefficient Implementation of line 11 of ABSTRACTFIXCONSERVATIONONPATH (Algorithm 5.2)

// push flow on d to make its residual capacity zero as required for Hassin's algorithm
1: *residual capacity* $\leftarrow \boldsymbol{c}[d] - \boldsymbol{f}[d] - \boldsymbol{\phi}[\text{head}_{G^*}(d)] + \boldsymbol{\phi}[\text{tail}_{G^*}(d)]$
2: *val* $\leftarrow \min\{$*residual capacity*$, |\text{inflow}(p_i)|\}$ — amount of flow to push on d
3: $\boldsymbol{f}[d] \leftarrow \boldsymbol{f}[d] + $ *val* ; $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[d]$

// push excess inflow from tail(d) to head(d) using Hassin's algorithm
4: $\ell[d'] \leftarrow \boldsymbol{c}[d'] - \boldsymbol{f}[d'] - \boldsymbol{\phi}[\text{head}_{G^*}(d')] + \boldsymbol{\phi}[\text{tail}_{G^*}(d')]$ for all darts $d' \in G$ — lengths are residual capacities
5: $\ell[\text{rev}(d)] \leftarrow |\text{inflow}(p_i)|$ — set the limit on the residual capacity of rev(d)
6: $\boldsymbol{\phi}_i(\cdot) \leftarrow$ DIJKSTRA$(G^*, \ell, \text{head}_{G^*}(d))$ — face potential are distances in G^* from head$_{G^*}(d)$ w.r.t. residual capacities
7: *val* $\leftarrow \boldsymbol{\phi}_i[\text{head}_{G^*}(d)] - \boldsymbol{\phi}_i[\text{tail}_{G^*}(d)]$ — the amount of flow assigned to d by the circulation corresponding to $\boldsymbol{\phi}_i$
8: $\boldsymbol{f}[d] \leftarrow \boldsymbol{f}[d] - $ *val* ; $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[(d)]$ — do not push the circulation on d and rev(d)
9: $\boldsymbol{\phi} = \boldsymbol{\phi} + \boldsymbol{\phi}_i$ — accumulate the current circulation

In this section, we give an *inefficient* implementation of line 11 of the abstract algorithm. This will facilitate the explanation of the efficient procedure in the next section.

The Inefficient Implementation

Recall that $\boldsymbol{f}_0$ is the flow at the beginning of the procedure. Observe that the change to the flow in iteration i of the abstract algorithm (line 11) is a flow between the endpoints of the dart d_i. As discussed in Section 1.4.2, this flow can be represented as the sum of (i) a circulation $\boldsymbol{\rho}_i$ and (ii) a flow on d_i and rev(d_i). Summing over the first i iterations, the flow $\boldsymbol{f}'$ at that time can be represented as the sum

$$\boldsymbol{f}' = \boldsymbol{\rho} + \boldsymbol{f} \tag{5.1}$$

where $\boldsymbol{\rho} = \sum_{j=1}^{i} \boldsymbol{\rho}_j$ is a circulation and $\boldsymbol{f}$ is a flow assignment that differs from $\boldsymbol{f}_0$ only on the darts of $\{d_j\}_{j=1}^{i}$ and their reverses.

Now we describe the inefficient implementation of line 11 of ABSTRACTFIXCONSERVATIONONPATH. The total flow $\boldsymbol{f}'$ is maintained by representing $\boldsymbol{f}$ and the circulation $\boldsymbol{\rho}$ as in Eq. (5.1). $\boldsymbol{f}$ is represented explicitly, but, in preparation for the efficient implementation, the circulations $\boldsymbol{\rho}_j$ are represented implicitly by the face-potentials $\boldsymbol{\phi}_j$. By linearity of Eq. (1.9), the sum $\boldsymbol{\phi} = \sum_{j=1}^{i} \boldsymbol{\phi}_j$ is the face potential vector that induces the circulation $\boldsymbol{\rho}$.

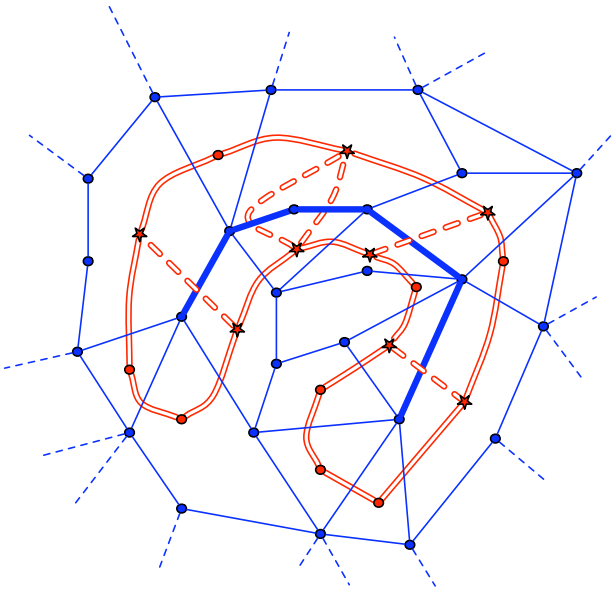

Figure 5.1: An example illustrating that the nodes of X^* are on the boundary of a single face of H^*. The diagram shows part of the graph G and some edges of its dual G^*. Edges of G are solid blue. Edges of P are bold. Dual edges are double lined red. The dual edges of P are in double lined dashed red. The nodes of X^* are represented by stars.

Recall that d is the dart of C such that flow needs to be sent from tail(d) to head(d) (line 10 of AbstractFixConservationOnPath). In lines 1 – 3, the procedure pushes as much as possible on d itself. Consequently, either d is saturated or conservation at p_i is achieved.

Next, an implementation of Hassin's algorithm pushes a maximum flow with limit $|\text{inflow}(p_i)|$ from tail(d) to head(d). The procedure first sets the length of darts to their residual capacities (line 4) and sets the length of rev(d) to be the flow limit $|\text{inflow}(p_i)|$ (line 5). Since the flow maintained is feasible, all residual capacities are non-negative. The procedure then computes all the head$_{G^*}(d)$-to-f distances $\boldsymbol{\phi}_i[f]$ in G^* using Dijkstra's algorithm (line 6). Let $\boldsymbol{\rho}_i$ be the circulation corresponding to the face-potential vector $\boldsymbol{\phi}_i$. The procedure sets *val* equal to $\boldsymbol{\rho}_i[d]$ in line 7, then subtracts *val* from $\boldsymbol{f}[d]$ and adds it to $\boldsymbol{f}[\text{rev}(d)]$. Finally, in the last line, the current circulation is added to the accumulated circulation by adding the potential $\boldsymbol{\phi}_i$ to $\boldsymbol{\phi}$.

5.3.3 An Efficient Implementation

In this section we give an *efficient* implementation of AbstractFixConservationOnPath. A tool that we use is the extension of FR-Dijkstra (see Chapter 3) that works with reduced lengths.

The efficient implementation keeps track of the inflow at each node of P in an array $\boldsymbol{v}[\cdot]$. As in the inefficient implementation, the procedure will maintain the total flow as the sum of a circulation $\boldsymbol{\rho}$ and a flow assignment $\boldsymbol{f}$ that differs from $\boldsymbol{f}_0$ only on the darts of $P \cup \text{rev}(P)$. Initially $\boldsymbol{f}$ is set

Algorithm 5.4 Efficient Implementation of line 11 of ABSTRACTFIXCONSERVATIONONPATH

// push flow on d to make its residual capacity zero as required for Hassin's algorithm
1: *residual capacity* $\leftarrow \boldsymbol{c}[d] - \boldsymbol{f}[d] - \boldsymbol{\phi}^X[\text{head}_{G^*}(d)] + \boldsymbol{\phi}^X[\text{tail}_{G^*}(d)]$
2: *val* $\leftarrow \min\{$*residual capacity*$, |\boldsymbol{v}[p_i]|\}$ amount of flow pushed on d
3: $\boldsymbol{f}[d] \leftarrow \boldsymbol{f}[d] +$ *val* ; $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[d]$
4: $\boldsymbol{v}[\text{tail}(d)] \leftarrow \boldsymbol{v}[\text{tail}(d)] -$ *val* ; $\boldsymbol{v}[\text{head}(d)] \leftarrow \boldsymbol{v}[\text{head}(d)] +$ *val* update the inflow at p_i and p_{i+1}

// push excess inflow from tail(d) to head(d) using Hassin's algorithm
5: $\ell[d'] \leftarrow \boldsymbol{c}[d'] - \boldsymbol{f}[d']$ for all darts $d' \in P \cup \text{rev}(P)$ lengths of explicit darts are residual capacities (excluding circulation component)
6: $\ell[\text{rev}(d)] \leftarrow |\boldsymbol{v}[p_i]| - \boldsymbol{\phi}^X[\text{tail}_{G^*}(\text{rev}(d))] + \boldsymbol{\phi}^X[\text{head}_{G^*}(\text{rev}(d))]$ limit residual capacity of rev(d) (adjusted by circulation component)
7: $\boldsymbol{\phi}_i^X(\cdot) \leftarrow$ FR-DIJKSTRA$(\text{D}^*, \ell, \boldsymbol{\phi}^X, \text{head}_{G^*}(d))$ distances in G^* from $\text{head}_{G^*}(d)$ w.r.t. the reduced lengths induced by $\boldsymbol{\phi}^X$
8: *val* $\leftarrow \boldsymbol{\phi}_i^X[\text{head}_{G^*}(d)] - \boldsymbol{\phi}_i^X[\text{tail}_{G^*}(d)]$ the amount of flow assigned to d by the circulation corresponding to ϕ_i^X
9: $\boldsymbol{f}[d] \leftarrow \boldsymbol{f}[d] -$ *val* ; $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[(d)]$ remove the flow assigned by the circulation to d and rev(d)
10: $\boldsymbol{\phi}^X = \boldsymbol{\phi}^X + \boldsymbol{\phi}_i^X$ accumulate the current circulation
11: $\boldsymbol{v}[\text{tail}_G(d)] = \boldsymbol{v}[\text{tail}_G(d)] -$ *val* ; $\boldsymbol{v}[\text{head}_G(d)] = \boldsymbol{v}[\text{head}_G(d)] +$ *val* update the inflow at p_i and p_{i+1}

equal to $\boldsymbol{f}_0$. The circulation $\boldsymbol{\rho}$ will be represented by a face-potential vector $\boldsymbol{\phi}$. However, we will show that it suffices to maintain just those entries of $\boldsymbol{\phi}$ that correspond to faces incident to P.

Define each dart d's length by $\ell(d) = \boldsymbol{c}[d] - \boldsymbol{f}[d]$. Let X^* be the set of endpoints in the planar dual G^* of the darts of P (i.e. the primal faces incident to P). Let H^* be the graph obtained from G^* by removing the darts of P. Note that in H^*, all the nodes of X^* that did not disappear (i.e., that have degree greater than zero) are on the boundary of a single face, as illustrated in Figure 5.1.

The procedure uses a multiple-source shortest paths (MSSP) algorithm [Kle05, CC07] to compute a table $\text{D}^*[\cdot,\cdot]$ of X^*-to-X^* distances in H^* with respect to the lengths $\ell(\cdot)$.

The implementation of lines 1– 10 of ABSTRACTFIXCONSERVATIONONPATH using the chosen representation of $\boldsymbol{f}'$ is straightforward. We therefore focus on the implementation of line 11.

Consider iteration i of the algorithm. The main difference between the inefficient implementation and the efficient one is in implementing the Dijkstra step for computing shortest paths in the dual. Instead of computing the entire shortest-path tree, the procedure computes just the distance labels of nodes in X^*. This is done using the FR-DIJKSTRA procedure (see Chapter 3), whose initialization requires the X^*-to-X^* distances in H^* with respect to the residual capacities. We now explain how these distances can be obtained.

The flow on a dart d in the primal is

$$\boldsymbol{f}[d] + \boldsymbol{\phi}[\text{head}_{G^*}(d)] - \boldsymbol{\phi}[\text{tail}_{G^*}(d)]. \tag{5.2}$$

Therefore the residual capacity of d is

$$(\boldsymbol{c}[d] - \boldsymbol{f}[d]) - \boldsymbol{\phi}[\text{head}_{G^*}(d)] + \boldsymbol{\phi}[\text{tail}_{G^*}(d)] \tag{5.3}$$

which is its *reduced* length $\ell_{\boldsymbol{\phi}}[d]$ with respect to the length $\ell(\cdot)$ and price function $\boldsymbol{\phi}$.

Suppose that d belongs to H^*, i.e. d is not one of the darts of $P \cup \text{rev}(P)$. The procedure never changes $\boldsymbol{f}[d]$, so $\boldsymbol{f}[d] = \boldsymbol{f}_0[d]$. Therefore $\ell(d) = \boldsymbol{c}[d] - \boldsymbol{f}_0[d]$. These lengths are known at the beginning of the procedure's execution. The reduced length of an X^*-to-X^* path $Q = d'_1, d'_2, \cdots, d'_j$ in H^* is

$$\sum_{i=1}^{j} (\ell(d'_i) - \boldsymbol{\phi}[\text{head}_{G^*}(d'_i)] + \boldsymbol{\phi}[\text{tail}_{G^*}(d'_i)]) = \left(\sum_{i=1}^{j} \ell(d'_i)\right) - \boldsymbol{\phi}[\text{end}(Q)] + \boldsymbol{\phi}[\text{start}(Q)]. \tag{5.4}$$

Therefore, for any nodes $x, y \in X^*$, the x-to-y distance in H^* w.r.t. the residual capacity is given by $\text{D}^*[x, y] - \boldsymbol{\phi}[y] + \boldsymbol{\phi}[x]$. Since the procedure maintains the restriction of $\boldsymbol{\phi}$ to nodes of X^*, this distance can be obtained in constant time.

It follows from the above discussion that, after executing line 11 of AbstractFixConservationOnPath using the efficient implementation for the last time, the flow assignment $\boldsymbol{f}$ maintained by the efficient implementation is the same as the one that would have been computed if the inefficient implementation were used. Moreover, the potential function $\boldsymbol{\phi}^X$ computed by the efficient implementation is the restriction to X^* of the potential function $\boldsymbol{\phi}$ that would have been computed by the inefficient implementation.

As discussed in Section 5.3.2, the flow $\boldsymbol{f}'$ to be returned is $\boldsymbol{\rho} + \boldsymbol{f}$ where $\boldsymbol{\rho}$ is the circulation induced by $\boldsymbol{\phi}$. It suffices, however, to return $\boldsymbol{\chi} + \boldsymbol{f}$ where $\boldsymbol{\chi}$ is any feasible circulation in $G_{\boldsymbol{f}}$: this change does not alter the inflow at any node, and, since it does not change the residual capacity of any cut, does not introduce new residual paths.

To compute $\boldsymbol{\chi}$, the algorithm computes shortest-path distances in the dual of $G_{\boldsymbol{f}}$ from a node $x \in X^*$. This computation consists of two steps. In the first step, the algorithm again uses FR-Dijkstra with price function $\boldsymbol{\phi}$ to compute the distances just to nodes of X^*. In the second step, the algorithm extends this distance labeling to include all nodes of G^*. Any shortest x-to-v path consists of an x-to-x' prefix (where $x' \in X^*$) and a x'-to-v suffix that passes through no nodes of X^*. The distance to v, therefore, is

$$\min_{x' \in X^*} \min\{d(x') + \ell(Q) \ : \ Q \text{ an } x'\text{-to-}v \text{ path not passing through } X^*\} \tag{5.5}$$

where $d(x')$ is the x-to-x' distance. Note that every edge of G^* whose endpoints are not both in X^* has nonnegative length. Therefore the distances (5.5) can be computed by a variant of Dijkstra's algorithm in which each node $x' \in X^*$ is initially inserted into the priority queue with label $d(x')$.

For completeness we include in Algorithm 5.5 the full pseudocode for the efficient implementation of FixConservationOnPath.

Algorithm 5.5 FIXCONSERVATIONONPATH$(G, P, \boldsymbol{c}, \boldsymbol{f}_0)$

Input: a directed planar graph G, simple path $P = d_1 d_2 \dots d_k$, capacity function $\boldsymbol{c}$, and a pseudoflow $\boldsymbol{f}_0$

Output: a pseudoflow $\boldsymbol{f}$ s.t. (1) $\boldsymbol{f} - \boldsymbol{f}_0$ satisfies conservation at nodes not on P and (2) w.r.t $\boldsymbol{f}$, no residual path exists from $\{v \in P : \text{inflow}(v) > 0\}$ to $\{v \in P : \text{inflow}(v) < 0\}$.

let X^* be the set of endpoints in the planar dual G^* of the darts of P
initialize to zero an array $\boldsymbol{\phi}^X[\cdot]$ indexed by the nodes of X^*
let H^* be the graph obtained from G^* by removing the darts of P
compute, using the MSSP algorithm [Kle05, CC07], a table $\text{D}^*[\cdot,\cdot]$ of distances in H^* between nodes of X^* w.r.t. the lengths $\boldsymbol{c} - \boldsymbol{f}_0$
$\boldsymbol{f} \leftarrow \boldsymbol{f}_0$
$\boldsymbol{c}[d] \leftarrow \boldsymbol{c}[d] + M$ for all darts of $P \cup \text{rev}(P)$
initialize an array of length k by $\boldsymbol{v}[i] \leftarrow$ inflow at p_i with respect to $\boldsymbol{f}$
for $i = 1, 2, \dots, k$
  let p_i and p_{i+1} be the tail and head of d_i, respectively
  // reduce the capacities of d and $\text{rev}(d)$ by M and adjust the flow appropriately
  for $d \in \{d_i, \text{rev}(d_i)\}$
    $\boldsymbol{c}[d] \leftarrow \boldsymbol{c}[d] - M$
    old flow $\leftarrow \boldsymbol{f}[d] + \boldsymbol{\phi}^X[\text{head}_{G^*}(d)] - \boldsymbol{\phi}^X[\text{tail}_{G^*}(d)]$
    new flow $\leftarrow \min\{$*old flow*$, \boldsymbol{c}[d]\}$ — adjusted flow must not exceed adjusted capacity
    $\boldsymbol{f}[d] \leftarrow$ *new flow* $- \boldsymbol{\phi}^X[\text{head}_{G^*}(d)] + \boldsymbol{\phi}^X[\text{tail}_{G^*}(d)]$ — the explicit flow does not include the circulation component
    $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[d]$
    $\boldsymbol{v}[\text{head}(d)] \leftarrow \boldsymbol{v}[\text{head}(d)] +$ *new flow* $-$ *old flow* — update the inflow at p_i and p_{i+1}
    $\boldsymbol{v}[\text{tail}(d)] \leftarrow \boldsymbol{v}[\text{tail}(d)] -$ *new flow* $+$ *old flow*
  if $\boldsymbol{v}[p_i] > 0$ **then** $d \leftarrow d_i$ **else** $d \leftarrow \text{rev}(d_i)$ — find in which direction flow should be pushed
  // push flow on d to make its residual capacity zero as required for Hassin's algorithm
  residual capacity $\leftarrow \boldsymbol{c}[d] - \boldsymbol{f}[d] - \boldsymbol{\phi}^X[\text{head}_{G^*}(d)] + \boldsymbol{\phi}^X[\text{tail}_{G^*}(d)]$
  val $\leftarrow \min\{$*residual capacity*$, |\boldsymbol{v}[p_i]|\}$ — amount of flow pushed on d
  $\boldsymbol{f}[d] = \boldsymbol{f}[d] +$ *val* ; $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[d]$
  $\boldsymbol{v}[\text{tail}(d)] = \boldsymbol{v}[\text{tail}(d)] -$ *val* ; $\boldsymbol{v}[\text{head}(d)] = \boldsymbol{v}[\text{head}(d)] +$ *val* — update the inflow at p_i and p_{i+1}
  // push excess inflow from $\text{tail}(d)$ to $\text{head}(d)$ using Hassin's algorithm
  let $\ell[d'] \leftarrow \boldsymbol{c}[d'] - \boldsymbol{f}[d']$ for all darts $d' \in P \cup \text{rev}(P)$ — lengths of explicit darts are residual capacities (not including circulation component)
  $\ell[\text{rev}(d)] \leftarrow |\boldsymbol{v}[p_i]| - \boldsymbol{\phi}^X[\text{tail}_{G^*}(\text{rev}(d))] + \boldsymbol{\phi}^X[\text{head}_{G^*}(\text{rev}(d))]$ — limit on the residual capacity of $\text{rev}(d)$ (adjusted by circulation component)
  $\boldsymbol{\phi}_i^X(\cdot) \leftarrow$ FR-DIJKSTRA$(\text{D}^*, \ell, \boldsymbol{\phi}^X, \text{head}_{G^*}(d))$ — distances in G^* from $\text{head}_{G^*}(d)$ w.r.t. the reduced lengths induced by $\boldsymbol{\phi}^X$
  val $\leftarrow \boldsymbol{\phi}_i^X[\text{head}(\text{rev}(d))] - \boldsymbol{\phi}_i^X[\text{tail}(\text{rev}(d))]$ — the amount of flow assigned to d by the circulation corresponding to $\boldsymbol{\phi}_i^X$
  $\boldsymbol{f}[d] \leftarrow \boldsymbol{f}[d] +$ *val* ; $\boldsymbol{f}[\text{rev}(d)] \leftarrow -\boldsymbol{f}[(d)]$ — do not push the circulation on d and $\text{rev}(d)$
  $\boldsymbol{\phi}^X = \boldsymbol{\phi}^X + \boldsymbol{\phi}_i^X$ — accumulate the current circulation
  $\boldsymbol{v}[\text{tail}(d)] = \boldsymbol{v}[\text{tail}(d)] -$ *val* ; $\boldsymbol{v}[\text{head}(d)] = \boldsymbol{v}[\text{head}(d)] +$ *val* — update the inflow at p_i and p_{i+1}
// find a feasible circulation
let x be an arbitrary node in G^*
let $\ell[d] \leftarrow \boldsymbol{c}[d] - \boldsymbol{f}[d]$ for all darts $d \in G$
$\boldsymbol{\chi} \leftarrow$ SINGLESOURCESHORTESTPATHS(G^*, ℓ, x) — distances of the nodes of G^* from x
$\boldsymbol{f}'[d] \leftarrow \boldsymbol{f}[d] + \boldsymbol{\chi}[\text{head}_{G^*}(d)] - \boldsymbol{\chi}[\text{tail}_{G^*}(d)]$ for every dart $d \in G$ — add explicit flow and implicit circulation
return $\boldsymbol{f}'$

5.3.4 Alternative to Line 32 of Algorithm 5.5

In this section we show that it is not necessary to use a generic shortest path algorithm that works with negative lengths to compute a feasible circulation $\boldsymbol{\chi}$ in line 32 of Algorithm 5.5. Instead of choosing x to be an arbitrary node in G^*, let x be an arbitrary node of X^*. Let $\boldsymbol{\chi}(y)$ denote the x-to-y distance in G^* w.r.t. the lengths ℓ. Instead of computing $\boldsymbol{\chi}$ using a shortest-path algorithm that accepts negative lengths, we will compute it more efficiently in two steps. Pseudocode is given below as Algorithm 5.6. In the first step we use FR-DIJKSTRA to compute the distances to just the nodes of X^*. In the second step we extend these distances to all other nodes using Dijkstra's algorithm.

In the first step the algorithm computes distances in G^* from x to all nodes of X^* w.r.t. $\ell_{\boldsymbol{\phi}^X}$, the reduced lengths of ℓ induced by the feasible price function $\boldsymbol{\phi}^X$. This is done by an additional invocation of FR-DIJKSTRA (line 2). Let $\boldsymbol{\psi}^X$ denote these distance labels. By definition of reduced lengths and a telescoping sum similar to the one in the derivation of Eq. (5.4),

$$\boldsymbol{\psi}^X[y] = \boldsymbol{\chi}(y) + \boldsymbol{\phi}^X[x] - \boldsymbol{\phi}^X[y]. \tag{5.6}$$

Since both $\boldsymbol{\phi}^X$ and $\boldsymbol{\psi}^X$ are known, the algorithm can compute the unreduced distances $\boldsymbol{\chi}[y]$ for all $y \in X^*$ (line 4). Next, the algorithm runs Dijkstra's algorithm on G^*, initializing the label of each node y of X^* to its correct value $\boldsymbol{\chi}[y]$. Since in the dual the darts of $P \cup \text{rev}(P)$ are only incident to nodes of X^*, Dijkstra's algorithm initialized in this manner correctly outputs the distance labels for all nodes of G^* even if some of the darts of $P \cup \text{rev}(P)$ may have negative lengths.

Algorithm 5.6 Alternative to line 32 of Algorithm 5.5

1: let x be an arbitrary node in X^*
2: $\boldsymbol{\psi}^X = \text{FR-DIJKSTRA}(\text{D}^*, \ell, \boldsymbol{\phi}^X, x)$ find distances of the nodes of X^* from x w.r.t. price function $\boldsymbol{\phi}^X$
3: initialize to ∞ an array $\boldsymbol{\chi}$ indexed by the nodes of G^*
4: $\boldsymbol{\chi}[y] = \boldsymbol{\psi}^X[y] - \boldsymbol{\phi}^X[x] + \boldsymbol{\phi}^X[y]$ for every $y \in X^*$ distances of the nodes of X^* from x
5: extend ℓ to all darts of G^* by $\ell[d] \leftarrow \boldsymbol{c}[d] - \boldsymbol{f}_0[d]$ for every dart $d \in G$
6: $\boldsymbol{\chi} \leftarrow \text{DIJKSTRA}(G^*, \ell, x, \boldsymbol{\chi})$ distances of the nodes of G^* from x

5.3.5 Running Time Analysis

Let n and k be the number of nodes in G and in P, respectively. The initialization time is dominated by the $O(n \log n + k^2 \log n)$ time for MSSP. The execution of each iteration of the main loop is dominated by the call to FR-DIJKSTRA, which takes $O(k \log^2 k)$ time. The number of iterations is $k - 1$, leading to a total of $O(k^2 \log^2 k)$ time for execution of the main loop. Computing the circulation $\boldsymbol{\chi}$ requires one more call to FR-DIJKSTRA and one Dijkstra computation in the entire graph, which takes $O(n \log n)$ time. Thus the total running time of the efficient implementation of FIXCONSERVATIONONPATH is $O(n \log n + k^2 \log^2 k)$.

5.4 Pushing Excess Inflow from a Cycle

In this section we describe the procedure CYCLETOSINKMAXFLOW that pushes excess inflow from a cycle to a node not on the cycle.

Algorithm 5.7 CYCLETOSINKMAXFLOW$(G, \boldsymbol{c}, \boldsymbol{f}_0, C, t)$

Input: a directed planar graph G with capacities $\boldsymbol{c}$, a pseudoflow $\boldsymbol{f}_0$, a simple cycle C, a sink t.
Assumes: $\forall v \in C^+$, $\text{inflow}_{\boldsymbol{f}_0}(v) \geq 0$, where $C^+ = \{v \in C : \{x \in C : \text{inflow}_{\boldsymbol{f}_0}(x) > 0\} \overset{G_{\boldsymbol{f}_0}}{\to} v\}$.
Output: a pseudoflow $\boldsymbol{f}$ s.t. *(i)* $\boldsymbol{f} - \boldsymbol{f}_0$ obeys conservation everywhere but $C^+ \cup \{t\}$, *(ii)* $\forall v \in C^+$, $\text{inflow}_{\boldsymbol{f}}(v) \geq 0$, *(iii)* $\{v \in C : \text{inflow}_{\boldsymbol{f}}(v) > 0\} \overset{G_{\boldsymbol{f}}}{\nrightarrow} t$.

1: let $C^+ \leftarrow \{v \in C$: there exists a residual path to v from a node $x \in C$ with $\text{inflow}_{\boldsymbol{f}_0}(x) > 0\}$
2: delete the nodes of $C - C^+$
3: let $v_1, v_2, \ldots, v_\ell$ be the nodes of C^+, labeled according to their cyclic order on C
4: add artificial arcs $v_i v_{i+1}$ for $1 \leq i < \ell$
5: let P be the v_1-to-v_ℓ path of artificial arcs
6: contract all the edges of P, collapsing C^+ into the single node v_1
7: $\boldsymbol{f} \leftarrow$ ST-MAXFLOW$(G, \boldsymbol{c} - \boldsymbol{f}_0, v_1, t)$
8: undo the contraction of the edges of P
9: $\boldsymbol{f} \leftarrow$ FIXCONSERVATIONONPATH$(G, P, \boldsymbol{c}, \boldsymbol{f}_0 + \boldsymbol{f}) - \boldsymbol{f}_0$
10: modify $\boldsymbol{f}$ to push back flow to nodes of C^+ whose inflow w.r.t $\boldsymbol{f}_0 + \boldsymbol{f}$ is negative
11: $\boldsymbol{f} \leftarrow \boldsymbol{f}_0 + \boldsymbol{f}$
12: **return** $\boldsymbol{f}$

To compute C^+ in Line 1, consider the residual graph of G w.r.t. $\boldsymbol{f}_0$. Add a node v and non-zero capacity arcs vw for every node w whose inflow w.r.t. $\boldsymbol{f}_0$ is positive (these arcs may not preserve planarity). In $O(|G|)$ time, find the set X of nodes that are reachable from v via darts with non-zero capacity. Then $C^+ = C \cap X$.

Since C^+ consists of all nodes of C reachable via residual paths from the nodes of C with positive inflow, the flow computed by the procedure involves no darts incident to nodes in $C - C^+$. Thus, restricting the computation to the graph obtained from G by deleting the nodes in $C - C^+$ (line 2) does not restrict the computed flow. After deletion, adding artificial arcs between consecutive nodes of C^+ (line 4) will not violate planarity. Contracting the artificial arcs effectively turns C^+ into a single node v_1. Next, the procedure computes a maximum flow $\boldsymbol{f}$ from C^+ to t w.r.t. residual capacities $\boldsymbol{c} - \boldsymbol{f}_0$. This is done by invoking a single-source single-sink maximum flow algorithm [BK09] with source v_1 and sink t (line 7). Uncontracting the artificial arcs turns $\boldsymbol{f}$ into a maximum C^+-to-t flow in G w.r.t. the residual capacities $\boldsymbol{c} - \boldsymbol{f}_0$. However, some of the nodes of C^+ may have negative inflow w.r.t. $\boldsymbol{f}_0 + \boldsymbol{f}$. In line 9, the procedure calls FIXCONSERVATIONONPATH to reroute the flow $\boldsymbol{f}$ among the nodes of C^+ so that, w.r.t. $\boldsymbol{f}_0 + \boldsymbol{f}$, there are no residual paths from nodes of C^+ with positive inflow to nodes of C^+ with negative inflow. This implies that any node of C^+ that still has negative inflow has pushed too much flow. Line 10 modifies $\boldsymbol{f}$ to push back such excess flow so that no node of C^+ has negative inflow w.r.t. $\boldsymbol{f}_0 + \boldsymbol{f}$. This is done using the technique discussed in Section 1.4.3. Finally, the procedure returns $\boldsymbol{f}_0 + \boldsymbol{f}$.

5.4.1 Correctness of CycleToSinkMaxFlow

Any flow that is pushed by the procedure originates at C^+ and terminates at $C^+ \cup \{t\}$. Therefore, $\boldsymbol{f} - \boldsymbol{f}_0$ violates conservation only at $C^+ \cup \{t\}$. In line 10, flow of $\boldsymbol{f}$ is pushed back so that no node of C^+ has negative inflow w.r.t. $\boldsymbol{f}_0 + \boldsymbol{f}$. It is possible to do so by only pushing back flow of $\boldsymbol{f}$ (rather than flow of $\boldsymbol{f}_0 + \boldsymbol{f}$) since by assumption no node of C^+ has negative inflow w.r.t. $\boldsymbol{f}_0$.

By maximality of the flow pushed in line 7, just after line 7 is executed there are no C^+-to-t residual paths. This remains true when the edges of P are uncontracted in Line 8. In line 9 flow is pushed among the nodes of C^+, so by *sources lemma*(C^+, t, C^+), there are no C^+-to-t residual paths after line 9 either. Moreover, by definition of FixConservationOnPath, there are no $C_{>0}$-to-$C_{<0}$ residual paths immediately after line 9 is executed, where $C_{>0}$ ($C_{<0}$) is the set of nodes of C^+ with positive (negative) inflow at that time. Line 10 pushes flow into $C_{<0}$, making all the nodes of $C_{<0}$ obey conservation. By *sinks lemma*$(C_{>0}, t, C_{<0})$ there are no $C_{>0}$-to-t residual paths upon termination. This completes the proof since $C_{>0}$ is the set of nodes of C with positive inflow upon termination.

5.4.2 Running Time Analysis

We next analyze the running time of this procedure on an n-node graph G and a k-node cycle C. The st-maximum flow computation in line 7 takes $O(n \log n)$ time using the algorithm of Borradaile and Klein [BK09]. The running time of the procedure is therefore dominated by the call to FixConservationOnPath in line 9 which takes $O(n \log n + k^2 \log^2 k)$ time.

Chapter 6

Maximum Flow in Direceted Planar Graphs with Multiple Sources and a Single Sink

In this chapter we present an alternative algorithm for the maximum flow problem with multiple sources and a single sink in planar graphs that runs in $O(\text{diameter} \cdot n \log n)$ time. The diameter of a graph is defined as the maximum over all pairs of nodes of the minimum number of edges in a path connecting the pair. Here, we refer to the diameter of the face-vertex incidence graph. Essentially, we start with a non-feasible flow that dominates a maximum flow, and convert it into a feasible maximum preflow by eliminating negative-length cycles in the dual graph. The main algorithmic tool is a modification of an algorithm of Klein [Kle05] for finding multiple-source shortest paths in planar graphs with nonnegative lengths; our modification identifies and eliminates negative-length cycles. This approach is significantly different than all previously known maximum-flow algorithms. While the relation between flow in the primal graph and shortest paths in the dual graph has been used in many algorithms for planar flow, considering fundamentally non-feasible flows and handling negative cycles is novel. We believe that this is an interesting algorithmic technique and are hopeful it will be useful beyond the current context.

6.1 Relation to Prior Work

Most of the algorithms for computing maximum flow in general (i.e., non-planar) graphs build a maximum flow by starting from the zero flow and iteratively pushing flow without violating arc capacities. Traditional augmenting path algorithms, as well as more modern blocking flow algorithms, push flow from the source to the sink at each iteration, thus maintaining a feasible flow (i.e., a flow

assignment that respects capacities and obeys conservation at non-terminals) at all times. Push-relabel algorithms relax the conservation requirement and maintain a feasible preflow rather than a feasible flow. However, none of these algorithms maintains a flow assignment that violates arc capacities.

There are algorithms for maximum flow in planar graphs that do use flow assignments that violate capacities [Rei83, MN95]. However, these violations are not fundamental in the sense that the flow does respect capacities up to a circulation (a flow with no sources or sinks). In other words, the flow may over-saturate some arcs, but no cut is over-saturated. We call such flows *quasi-feasible* flows.

The value of a flow that does over-saturate some cuts is higher than that of a maximum flow. This situation can be identified by detecting a negative-length cycle in the dual of the residual graph. One of the algorithms in [MN95] uses this property in a parametric search for the value of the maximum flow. When a quasi-feasible flow with maximum value is found, it is converted into a feasible one. This approach is not suitable for dealing with multiple sources because the size of the search space grows exponentially with the number of sources.

Our approach is the first to use and handle fundamentally infeasible flows. Instead of interpreting the existence of negative cycles as a witness that a given flow should not be used to obtain a maximum feasible flow, we use the negative cycles to direct us in transforming a fundamentally non-feasible flow into a maximum feasible flow. A negative-length cycle whose length is $-c$ corresponds to a cut that is over saturated by c units of flow. This implies that the flow should be decreased by pushing c units of flow back from the sink across that cut.

6.2 The Algorithm

We describe an algorithm that, given a graph G with n nodes, a sink t incident to the infinite face f_∞, and a set S of source nodes, computes a maximum flow from the sources to t in time $O(\text{diameter} \cdot n \log n)$, where diameter is the diameter of the face-vertex incidence graph of G. Initially, each dart d has a non-negative capacity, which we denote by $\text{length}(d)$ since we will interpret it as a length in the dual. During the execution of the algorithm, the length assignment $\text{length}(\cdot)$ is modified. Even though the algorithm does not explicitly maintain a flow at all times, we will refer throughout the description to *the flow pushed by the algorithm*. At any given point in the execution of the algorithm we can interpret the lengths of darts in the dual as their residual capacities in the primal. By *the flow pushed by the algorithm* we mean the flow that would induce these residual capacities.

The algorithm starts by pushing flow from every source s to t such that, for each source s, the edges $\Gamma^+(\{s\})$ are saturated (Line 4). In general, this flow is not feasible since it might violate the capacities of some darts. It then starts to reduce that flow in order to make it feasible. This is done by using a spanning tree T of G and a spanning tree T^* of G^* such that each edge is in exactly one of these trees. The algorithm repeatedly identifies a negative cycle C in G^*, which corresponds in G to

Algorithm 6.1 Multiple-source single-sink maximum flow (G, S, t, c_0)

Input: planar directed graph G with capacities c_0, source set S, sink t incident to f_∞
Output: a maximum feasible flow f

1: length$(d) := c_0(d)$ for every dart d
2: initialize spanning tree T^* of G^* rooted at f_∞ using right-first-search
3: let T be the spanning tree of G consisting of edges not in T^*, and root T at t
4: **for** each source $s \in S$
5: **for** each dart d on the s-to-t path in T
6: length$(d) :=$ length$(d) -$ length$(\Gamma^+(\{s\}))$
7: length(rev$(d)) :=$ length(rev$(d)) +$ length$(\Gamma^+(\{s\}))$
8: **while** there exist unrelaxed darts in G^*
9: let $\hat{d}$ be an unrelaxed dart that is leafmost in T
10: **if** $\hat{d}$ is not a back-edge in T^* **then** //perform a pivot
11: remove from T^* the parent edge of head$_{G^*}(\hat{d})$ and insert $\hat{d}$ into T^*
12: **else** // fix a negative cycle by pushing back flow
13: let C denote the fundamental cycle of $\hat{d}$ with respect to T^* in G^*
14: **for** each dart d of the $\hat{d}$-to-t path of darts in the primal spanning tree T

$$\text{length}(d) := \text{length}(d) + |\text{length}(C)|$$
$$\text{length}(\text{rev}(d)) := \text{length}(\text{rev}(d)) - |\text{length}(C)|$$

15: **for** every dart d strictly enclosed by C
16: $f(d) := c_0(d) -$ length$_{T^*}(d)$
17: in G, contract d //in G^*, delete d
18: $f(d) := c_0(d) -$ length$_{T^*}(d)$ for every dart d
19: convert the preflow f into a flow

an over-saturated cut. Line 14 decreases the lengths of darts on a primal path in T that starts at the sink t and ends at some node v (a face in G^*) that is enclosed by C. We call such a path a *pushback path*. This change corresponds to pushing flow back from the sink to v along the pushback path, making the over-saturated cut exactly saturated. We call the negative-turned-zero-length cycle C a *processed cycle*. Processed cycles enclose no negative cycles. This implies that there exists a preflow that saturates the cut corresponding to a processed cycle (see Lemma 6.3.5). The algorithm records that preflow (Line 16) and contracts the source-side of the cut into a single node referred to as a *super-source*.

When no negative length cycles are left in the contracted graph, the flow pushed by the algorithm is quasi-feasible. That is, the flow pushed by the algorithm is equivalent, up to a circulation, to a feasible flow in the contracted graph. Combining this feasible flow in the contracted graph and the preflows recorded at the times cycles were processed yields a maximum preflow for the original uncontracted graph. In a final step, this maximum preflow is converted into a maximum feasible flow.

We now describe in more detail how negative cycles are identified and processed by the algorithm. The algorithm maintains a spanning tree T^* of G^* rooted at the infinite face f_∞ of G, and a spanning tree T of G, rooted at the sink t. The tree T consists of the edges not in T^*. The algorithm tries to transform T^* into a shortest-path tree by pivoting into T^* unrelaxed darts according to some

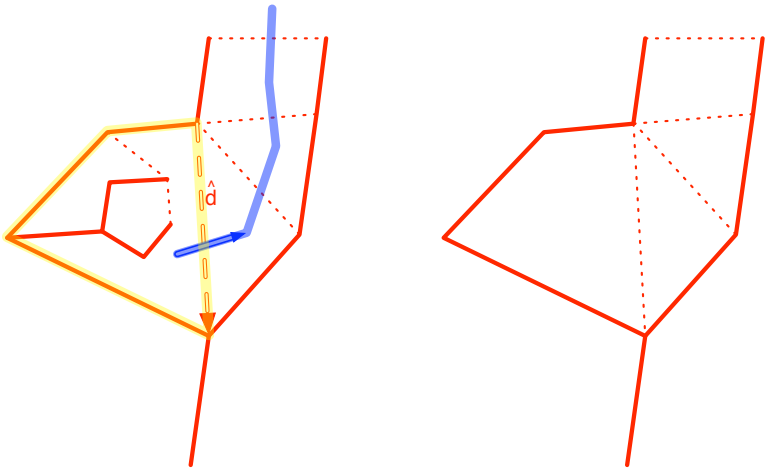

Figure 6.1: Identifying and processing a negative cycle. On the left, the tree T^* is shown in solid red. Non-tree edges are dashed. The unrelaxed dart $\hat{d}$ is double lined dashed red. In the primal $\hat{d}$ is shown as a double lined blue dart. The fundamental cycle of $\hat{d}$ w.r.t. T^* (marked yellow) is a counterclockwise dual cycle whose length is negative. The primal pushback path is indicated in blue. It originates from the sink t and ends at the tail of the primal dart $\hat{d}$. After the negative cycle is processed, its interior is deleted (on the right).

particular order (line 9). However, if an unrelaxed dart $\hat{d}$ happens to be a back-edge in T^*, the corresponding fundamental cycle C is a negative-length cycle. To process C, flow is pushed from t to tail$(\hat{d})$ along the t-to-$\hat{d}$ path in T (line 14). The amount of flow the algorithm pushes is $|\text{length}(C)|$, so after C is processed its length is zero, and $\hat{d}$ is no longer unrelaxed. The algorithm then records a preflow for C, deletes the interior of C, and proceeds to find the next unrelaxed dart. See Fig. 6.1 for an illustration[1]. When all darts are relaxed, T^* is a shortest-path tree, which implies no more negative-length cycles exist.

To control the number of pivots we initialize T^* to be *right-short* (see Section 1.3.4), and show that right-shortness is preserved by the algorithm and that it implies that the number of pivots of any dart is bounded by the diameter of the face-vertex incidence graph. We later discuss how data structures enable the iterations to be performed efficiently.

6.3 Correctness and Analysis

We begin with a couple of basic lemmas. Consider the sink t as the infinite face of G^*. By our conventions, any clockwise (counterclockwise) cycle C in G^* corresponds to a primal cut $(X, V - X)$ such that $t \in X$ ($t \notin X$); see Fig. 1.3.

[1]Recall the discussion in Section 1.2.4 about inverted orientations of the dual in figures like Fig. 6.1. In the figure the cycle appears to be clockwise, but this is a counterclockwise dual cycle.

Lemma 6.3.1. *Consider the sink t as the infinite face of G^*. The length of any clockwise (counterclockwise) dual cycle does not decrease (increase) when flow is pushed to t. The length of any clockwise (counterclockwise) dual cycle does not increase (decrease) when flow is pushed from t.*

Proof. The change in the length of a dual cycle is equal to the change in the residual capacity of the corresponding primal cut. Since clockwise cycles correspond to cuts $(X, V - X)$ s.t. $t \in X$, it follows that the residual capacity of a cut corresponding to a clockwise dual cycle can only increase when flow is pushed to t. The proofs of the other claims are similar. □

The following is a restatement of a theorem of Miller and Naor [MN95].

Lemma 6.3.2. *Let G be a planar graph. Let c_0 be a capacity function on the darts of G. Let f be a flow assignment. Define the length of a dart d to be its residual capacity $c_0(d) - f(d)$. If f is quasi-pseudoflow then $f'(d) = c_0(d) - length_{T^*}(d)$ is a pseudoflow, where T^* is a shortest-path tree in G^*. Furthermore, $f' = f + \rho$ for some circulation ρ.*

Proof. If f is a quasi-pseudoflow then there exists a feasible circulation ρ in G_f. In Section 1.4.1 we proved that $\rho(d) = \text{length}(T^*[\text{head}_{G^*}(d)]) - \text{length}(T^*[\text{tail}_{G^*}(d)])$ is such a feasible circulation, where the length of darts are defined to be their capacity in G_f, namely $c(d) = c_0(d) - f(d)$. Thus, $c(d) - \rho(d) \geq 0$. Interpret $c(d) - \rho(d)$ as residual capacity with respect to some flow f' in G. Namely, $c(d) - \rho(d) = c_0(d) - f'(d)$. Therefore, $f'(d) = c_0(d) - c(d) + \rho(d) = f(d) + \rho(d)$. Furthermore,

$$\begin{aligned} f'(d) &= c_0(d) - c(d) + \rho(d) \\ &= c_0(d) - c(d) + \text{length}(T^*[\text{head}_{G^*}(d)]) - \text{length}(T^*[\text{tail}_{G^*}(d)]) \\ &= c_0(d) - (c(d) + \text{length}(T^*[\text{tail}_{G^*}(d)]) - \text{length}(T^*[\text{head}_{G^*}(d)])) \\ &= c_0(d) - \text{length}_{T^*}(d) \end{aligned}$$

where the last equality follows from the definition of the reduced lengths with respect to T^*. □

Correctness

To prove the correctness of the algorithm, we first show that a fundamental cycle w.r.t. a back-edge is indeed a negative-length cycle.

Lemma 6.3.3. *Let $\hat{d}$ be an unrelaxed back-edge w.r.t. T^*. The length of the fundamental cycle of $\hat{d}$ w.r.t. T^* is negative.*

Proof. Consider the price function induced by from-root distances in T^*. Every tree dart whose tail is closer to the root than its head has zero reduced length. The reduced length of an unrelaxed dart is negative. Since $\hat{d}$ is a back edge w.r.t. T^*, its fundamental cycle C uses darts of T^* whose lengths are zero. Therefore the reduced length of C equals the reduced length of just $\hat{d}$, which is negative. The lemma follows since the length and reduced length of any cycle are the same. □

Lemma 6.3.4. *Let C be the negative-length cycle defined in line 13. After C is processed in line 14, length$(C) = 0$ and $\hat{d}$ is relaxed. Furthermore, the following invariants hold just before line 14 is executed:*

1. *the flow pushed by the algorithm satisfies flow conservation at every node other than the sources (including super-sources) and the sink.*
2. *the outflow at the sources is non-negative.*
3. *there are no clockwise negative-length cycles.*

Proof. The proof is by induction on the number of iterations of the main while loop of the algorithm. The initial flow pushed by the algorithm is from the sources to the sink. Therefore, conservation is satisfied at every non-terminal, the outflow at the sources is non-negative, and there is a one-to-one correspondence between over-saturated cuts in the primal and counterclockwise negative-length cycles in the dual.

For the inductive step, since by the inductive assumption there are no clockwise cycles, Lemma 6.3.3 implies that the cycle C in line 13 must be counterclockwise. Therefore, the unrelaxed dart $\hat{d}$ in line 9 points in T towards the root t. Thus, the length of $\hat{d}$ is increased in line 14 by $|\text{length}(C)|$, and the length of C becomes zero. This shows the main claim of the lemma. To complete the proof of the invariants, note that the interior of C is deleted in G^*, so the flow pushed in line 14 is now a flow from the sink to the newly created super-source. This shows that invariant (1) is preserved. Having length$(C) = 0$ implies that the total flow pushed so far by the algorithm from the newly created super-source is non-negative. This shows (2).

Since the outflow at all sources is non-negative after the pushback, the inflow at the sink must still be non-negative and equals the sum of outflows at the sources. Therefore, for any negative-length dual cycle, the corresponding primal over-saturated cut $(X, V - X)$ must be such that $t \notin X$. Hence any negative-length cycle is counterclockwise. This shows (3). □

We now prove properties of the flow computed by the algorithm. The following lemma characterizes the flow recorded in line 16. Intuitively, this shows that it is a saturating feasible flow for the cut corresponding to the processed cycle.

Lemma 6.3.5. *Let C be a cycle currently being processed. Let $(X, V - X)$ be the corresponding cut, where $t \notin X$. The flow assignment f computed in the loop in Line 16 satisfies:*

1. *$f(d) \leq c_0(d)$ for all darts whose endpoints are both in X.*
2. *every node in X except sources and tails of darts of $\Gamma^+(X)$ satisfies conservation.*
3. *for every $d' \in \Gamma^+(X)$ such that tail(d') is not a source,*

$$\textit{inflow}_f(\textit{tail}(d')) \geq \sum_{d \in \Gamma^+(X) : tail(d) = tail(d')} c_0(d)$$

Proof. let ${G^*}_c$ denote the region of G^* enclosed by C. Let G_c denote the graph obtained from G by contracting $V - X$ into a single node. Note that ${G^*}_c$ is the dual of G_c. Let $\hat{d}$ be the non-tree dart of C. Since $\hat{d}$ is leafmost unrelaxed, there are no unrelaxed darts in ${G^*}_c$ other than $\hat{d}$. By Lemma 6.3.4, after the loop in line 14 is executed $\hat{d}$ is no longer unrelaxed as well. Therefore, the restriction of T^* to ${G^*}_c$ is a shortest-path tree for ${G^*}_c$, and ${G^*}_c$ contains no negative cycles. The conditions of Lemma 6.3.2 are satisfied, so when $f(d)$ is set to $c_0(d) - \text{length}_{T^*}(d)$ in Line 16, it respects the capacities $c_0(d)$ for all $d \in G_c$. This shows 1.

By Lemma 6.3.4, just before C is processed, the restriction of the flow pushed by the algorithm to G_c satisfies conservation everywhere except at the sources and at the sink. In Line 14 flow is pushed back to tail$(\hat{d})$, so there is more flow entering tail$(\hat{d})$ than leaving it (unless tail$(\hat{d})$ happens to be a source). Therefore, the restriction of the flow pushed by the algorithm to G_c satisfies conservation everywhere except at the sources, the sink (which in G_c is the only node not in X) and tail$(\hat{d})$. By Lemma 6.3.2, the flow $f(d) = c_0(d) - \text{length}_{T^*}(d)$ differs from the flow pushed by the algorithm by a circulation, so $f(\cdot)$ satisfies conservation at all those nodes as well.

In particular, for every dart $d' \in \Gamma^+(X)$ such that tail(d') is not a source, $0 \leq \sum_{d:\text{head}(d)=\text{tail}(d')} c_0(d) - \text{length}_{T^*}(d)$ (this is an inequality only for $d' = \hat{d}$). However, note that the darts of $\Gamma^+(X)$ are not strictly enclosed by C, and that $\text{length}_{T^*}(d) = 0$ for every $d \in C$. Since Line 16 assigns $f(d) = c_0(d) - \text{length}_{T^*}(d)$ only to darts strictly enclosed by C (and implicitly assigns zero to all other darts), we get that, for the darts of $\Gamma^+(X)$, $0 \leq \sum_{d:\text{head}(d)=\text{tail}(d')} f(d) + \sum_{d\in\Gamma^+(X):\text{head}(d)=\text{tail}(d')} c_0(d)$. Or alternatively, for every $d' \in \Gamma^+(X)$, $\sum_{d:\text{head}(d)=\text{tail}(d')} f(d) \geq \sum_{d\in\Gamma^+(X):tail(d)=tail(d')} c_0(d)$. This shows 2 and 3. □

Lemma 6.3.6. *The following invariant holds. In G^* every source is enclosed by some zero-length cycle that encloses no negative-length cycles.*

Proof. Initially, the face of G^* corresponding to each source is such a cycle. At the time a cycle C is processed and a super-source s is created, C is a zero-length cycle enclosing s that encloses no negative-length cycles. The length of C only changes if the pushback path ends at a dart of C in the execution of Line 14 when processing a cycle C' at some later time. Since the interior of C is deleted when C is processed, C' encloses C. At this time, the interior of C', is contracted into a single new super-source which is enclosed by the zero-length cycle C' that encloses no negative cycles. □

The following lemma is an easy consequence of lemma 6.3.6.

Lemma 6.3.7. *The following invariant holds. There exists no feasible flow f' s.t. inflow$_{f'}(t)$ is greater than inflow$_f(t)$, where f is the flow pushed by the algorithm.*

Proof. Let f_s be the amount of flow source s delivers to the sink in the flow pushed by the algorithm. Consider a flow f' and let f'_s be the amount of flow source s delivers to the sink in f'. By Lemma 6.3.6, at any time along the execution of the algorithm, every source is enclosed in a zero length cycle. Consider such a zero-length cycle C. If $\sum_{s \text{ enclosed by } C} f'(s) > \sum_{s \text{ enclosed by } C} f(s)$ then the dual of the residual graph w.r.t. f' contains a negative cycle, so f' is not feasible. □

Lemma 6.3.8. *The flow f computed in Line 18 is a maximum feasible flow in the contracted graph G w.r.t. the capacities c_0.*

Proof. The flow pushed by the algorithm satisfies conservation by Lemma 6.3.4. By Lemma 6.3.7, there is no feasible flow of greater value. Since there are no unrelaxed darts, T^* is a shortest-path tree in G^* and G^* contains no negative cycles. Therefore, the flow pushed by the algorithm is quasi-feasible. This shows that the conditions of Lemma 6.3.2 are satisfied. It follows that f computed in Line 18 satisfies conservation, respects the capacities c_0 and has maximum value. □

Lemma 6.3.9. *The flow assignment $f(d)$ is a maximum preflow in the (uncontracted) graph G w.r.t. the capacities c_0.*

Proof. f is well defined since each dart is assigned a value exactly once; in Line 16 at the time it is contracted, or in Line 18 if it was never contracted. By Lemma 6.3.8 and by part 1 of Lemma 6.3.5, $f(d) \leq c_0(d)$ for all darts d, which shows feasibility. By Lemma 6.3.8 and by part 2 of Lemma 6.3.5, flow is conserved everywhere except at the sources, the sink, and nodes that are tails of darts of processed cycles. However, for a node v that is the tail of some dart of a processed cycle, $\sum_{d:head(d)=v} f(d) \geq 0$. This is true by part 3 of Lemma 6.3.5, and since $f(d) \leq c_0(d)$ for any dart. $f(\cdot)$ is therefore a preflow. Finally, the value of $f(\cdot)$ is maximum by Lemma 6.3.8. □

Lemma 6.3.9 completes the proof of correctness since line 19 converts the maximum feasible preflow into a feasible flow of the same value.

Efficient implementation

The dual tree T^* is represented by a table parentD$[\cdot]$ that, for each nonroot node v, stores the dart parentD$[v]$ of T^* whose head in G^* is v. The primal tree T is represented using a dynamic-tree data structure such as self-adjusting top-trees [AHdT05]. Each node and each edge of T is represented by a node in the top-tree. Each node of the top-tree that represents an edge e of T has two weights, $w_L(e)$ and $w_R(e)$. The values of these weights are the reduced lengths of the two darts of e, the one oriented towards leaves and the one oriented towards the root.[2] The weights are represented so as to support an operation that, given a node x of the top-tree and an amount Δ, adds Δ to $w_R(e)$ and subtracts Δ from $w_L(e)$ for all edges e in the x-to-root path in T.

This representation allows each of the following operations be implemented in $O(\log n)$ time: lines 9, 11, 13, 16, and 17, the loop of line 5, and the loop of line 14.[3]

Running-Time Analysis

Lines 16 and 17 are executed at most once per edge. To analyze the running time it therefore suffices to bound the number of pivots in line 11 and the number of negative cycles encountered

[2] Note that the length of the the cycle C in line 13 is exactly the reduced length of $\hat{d}$.

[3] The whole initialization in the loop of line 4 can instead be carried out in linear time by working up from the leaves towards the root of T.

by the algorithm. Note that every negative length cycle strictly encloses at least one edge. This is because the length of any cycle that encloses just a single source is initially set to zero, and since the length of the cycle that encloses just a single super-source is set to zero when the corresponding negative cycle is processed and contracted.

Since the edges strictly enclosed by a processed cycle are deleted, the number of processed cycles is bounded by the number of edges, which is $O(n)$.

It remains to bound the number of pivots. We will prove that the tree T^* satisfies the *right-shortness* property (see Section 1.3.4). This property implies that the number of times a given dart pivots into T^* is bounded by the diameter of the graph (Lemma 6.3.15). Thus, the total running time of the algorithm is $O(\text{diameter} \cdot n \log n)$.

Since T^* is initialized using right first search, initially, for every node v there is no simple path in G^* that is strictly right of $T^*[v]$. Therefore, initially, T^* is right-short. The algorithm changes T^* in two ways; either by making a pivot (line 11) or by processing a negative-length cycle (line 14). The following lemma was proved in [Kle05]:

Lemma 6.3.10. *[Kle05] leafmost dart relaxation (line 11) preserves right-shortness.*

Lemma 6.3.11. *Right-shortness is preserved when processing the counterclockwise negative-length cycle C in line 14.*

Proof. For a node v, let P denote the tree path $T^*[v]$. Let Q be a simple path that is strictly right of P. By right-shortness of T^*, before line 14 is executed $\text{length}(Q) > \text{length}(P)$. We show that this is also the case after line 14 is executed. By definition of right-of, $P \circ \text{rev}(Q)$ is a clockwise cycle. Note that since C is counterclockwise, the changes in lengths of darts in line 14 correspond to pushing back flow from the sink t. Therefore, by Lemma 6.3.1, the length of $P \circ \text{rev}(Q)$ can only decrease. Since no dart of P changes its length (only lengths of darts of T are changed in line 14), this implies that $rev(Q)$ can only decreases in length. By antisymmetry, the length of Q can only increase, so $\text{length}(Q) > \text{length}(P)$ after the execution as well. □

We have thus established that

Corollary 6.3.12. *T^* is right-short at all times.*

In the remainder of this section we bound the number of times a dart d is ejected from T^*.

Lemma 6.3.13. *Let G be a planar graph. Let P, Q be two v-to-u simple edge disjoint paths of darts for some nodes u, v. Let m be a bound on the number of darts in either P or Q. Let ϕ be the potential function associated with $P \circ rev(Q)$, normalized to zero on the infinite face of G. For any face f, $|\phi[f]| \leq 4m$.*

Proof. Consider the faces of the subgraph G' of G induced by $P \circ \text{rev}(Q)$. Since P and Q are simple, each such face consists of at least one edge of P and one of Q. Each edge of P or Q is in at most two faces, so the number of faces of G' is bounded by $2m$. Assume $|\phi[f]| > 4m$ for some face f of G'. Then there must exist a dart d such that $|\phi[\text{head}_{G'^*}(d)] - \phi[\text{tail}_{G'^*}(d)]| > 2$. However, since P and

Q are simple, each dart appears in $P \circ \text{rev}(Q)$ at most twice, so $|\phi[\text{head}_{G'^*}(d)] - \phi[\text{tail}_{G'^*}(d)]| \leq 2$, a contradiction. □

For an iteration i of the algorithm, let T_i^* be the tree T^* at the end of that iteration. Consider a dart d of G^* with head f. Let P_i denote $T_i^*[f]$. Let C_i be the dual cycle $P_i \circ \text{rev}(P_{i-1})$. Let ϕ_i be the potential function associated with C_i (see Eq. (1.1)).

Lemma 6.3.14. *For any i, ϕ_i is non-negative. Furthermore, if d is ejected from T^* in iteration i, then ϕ_i is strictly positive for at least one of the two faces (primal nodes) to which d is adjacent.*

Proof. Since P_i is at least as short as P_{i-1} and by right-shortness of T^*, P_i is left of P_{i-1}. Hence, C_i is a clockwise dual cycle, so ϕ_i is non-negative.

Consider the iterations of the algorithm when d is ejected from T^*. Let i be such an iteration. That is, $d \in T_{i-1}^*$, but $d \notin T_i^*$. Since $d \in P_{i-1}$, $d \notin P_i$, one of the two faces (primal nodes) to which d is adjacent is assigned a strictly positive potential by ϕ_i □

Lemma 6.3.15. *A dart d is ejected from T^* at most $16 \cdot$ diameter times throughout the execution of the algorithm, where diameter is the diameter of the face-vertex incidence graph $R(G^*)$ of G^*.*

Proof. Let γ be a f_∞-to-f path in the face-vertex incidence graph $R(G^*)$ of G^* with at most diameter edges. For the sake of the proof, we may consider adding edges to G^* without violating planarity so that γ is a path in the augmented G^*.

For any k, let $\phi_{\gamma,k}$ be the potential function that corresponds to $P_k \circ \text{rev}(\gamma)$. Observe that $P_k \circ \text{rev}(\gamma)$ can be written as a telescopic sum:

$$P_k \circ \text{rev}(\gamma) = P_k \circ \text{rev}(P_{k-1}) \circ P_{k-1} \circ \text{rev}(P_{k-2}) \circ \cdots \circ P_1 \circ \text{rev}(P_0) \circ P_0 \circ \text{rev}(\gamma). \tag{6.1}$$

By linearity of the relation of circulations and face potentials (Eq. (1.7)), $\phi_{\gamma,k}$ can be written as:

$$\phi_{\gamma,k} = \phi_{\gamma,0} + \sum_{i=1}^{k} \phi_i \tag{6.2}$$

Let j be the number of times d is ejected from T^*. By Lemma 6.3.14 and Eq. (6.2), for at least one face (primal node) v to which d is adjacent $\phi_{\gamma,k}[v] - \phi\gamma, 0[v] > \lfloor \frac{j}{2} \rfloor$. By Lemma 6.3.13, $|\phi_{\gamma,k}[v] - \phi\gamma_0[v]| < 8 \cdot$ diameter. Hence j is at most 16 times the diameter of the face vertex incidence graph of G^*. □

Part IV

Conclusions

In this thesis we have shown how to exploit the combinatorial structure of planar graphs to obtain nearly linear-time algorithms for shortest paths and flow problems. In Chapter 2, we saw how to efficiently exploit the Monge property to solve the shortest-path problems in planar graphs with negative lengths. The main contributions there were in showing decompositions into Monge matrices, avoiding the $\Theta(\log n)$ overhead incurred by earlier such decompositions, and in showing how that the SMAWK algorithm and its generalizations to partial matrices can be used in this context.

We used the Monge property once more in Chapter 3, where we observed that a range-minimum query data structure for Monge matrices can be used to extend Fakcharoenphol and Rao's efficient implementation of Dijkstra's algorithm to work with reduced lengths. Such an extension was necessary in our approach for solving the maximum flow problem with multiple sources and sinks in planar graphs in chapter 5, which uses the connection between shortest paths in the dual graph and circulations in the primal.

We saw an additional use of the relation between primal circulations and dual shortest paths in Chapter 6, where we developed a new technique for computing maximum feasible flow. An interesting feature of this technique is that throughout the computation, the flow maintained by the algorithm is neither capacity-respecting, nor does it obey conservation. Instead, the algorithm seeks and eliminates cuts whose overall capacity is violated. When no such cuts exist, it is easy to convert the flow into a feasible one. We find this technique appealing and hope it will find uses in the future.

We conclude with open problems and future research directions that are directly related to the problems studied here.

Open Questions

The current fastest algorithm for shortest paths with negative arc lengths in planar graphs is the $O(n \log^2 n / \log\log n)$-time algorithm described in Chapter 2. For non-negative lengths there exists a linear-time algorithm [HKRS97]. Closing this gap is an important open problem. While achieving a linear-time algorithm may be unlikely (compare to the gap between negative and non-negative lengths in the general case), it is reasonable to believe that $O(n \log n)$ is achievable. The running-time bottleneck currently stems from the use of the MSSP algorithm ([Kle05, CC07], see Section 1.3.4) for computing the all-pair distances between $O(\sqrt{n})$ boundary nodes in a piece with n nodes, which takes $O(n \log n)$ time. Many of the algorithms that use MSSP use it for this exact purpose (e.g., the distance oracles in Chapter 4). In a sense, the MSSP data structure provides more than we actually need. We can query it for the distance between any boundary node and any node in the graph in $O(\log n)$ time. For our purposes we only require distances between pairs of boundary nodes, and we know in advance we require all such pairs. Is it possible to design a linear time algorithm for this special case? Such an algorithm would imply an $O(n \log n)$ running time for the algorithm in Section 2.1.

Another possible direction is to abandon the divide-and-conquer recursive approach and focus on more direct approaches. One such direction is extending the *sweep* technique, which was successfully

used in Klein's MSSP, and for maximum s-t flow in planar graphs [BK09]. We note that a slight variant on the MSSP algorithm can used to compute shortest paths with negative lengths in planar graphs. Instead of initializing the algorithm with a shortest-path tree, initialize it with a right-first-search tree. An analysis similar to that of the multiple-source single-sink maximum flow algorithm in Chapter 6 shows that the running time is $O(\text{diameter} \cdot n \log n)$. Can a similar approach lead to a more efficient algorithm?

Fakcharoenphol and Rao's implementation of Dijkstra's algorithm for a set of dense distance graphs on n nodes runs in time $O(n \log^2 n)$. Improving this running time would imply better running time for quite a few algorithms that use FR-Dijkstra, such as the multiple-source multiple-sinks maximum flow algorithm of Chapter 5, the exact distance oracles of Chapter 4, and the minimum st-cut oracle in [BSWN10]. One of the logarithmic factors in the running time of FR-Dijkstra arises from the decomposition of the distance matrix for a dense distance graph on n nodes into Monge matrices, such that each node is represented in $O(\log n)$ Monge matrices. As we have shown in Section 2.1, this decomposition can be avoided in the context of the Bellman-Ford algorithm. Can one avoid the logarithmic blowup in the case of FR-Dijkstra as well?

Currently, the best space-to-query-time tradeoff known for exact distance oracles is $\tilde{O}(n/\sqrt{s})$ query-time for an oracle requiring $O(S)$ space. Our work on exact distance oracles (Chapter 4) filled in the gap, so that such oracles are now known for the entire range $S \in [n \log \log n, n^2]$. A modest goal in this context would be to get rid of the logarithmic factors in the space-to-query-time tradeoff. The natural important open problem is whether exact distance oracles with better space-to-query-time tradeoff exist. Note that a much better tradeoff is known for approximate distance oracles [KKS11]. Another related problem is the design exact dynamic distance oracle - data structures that are capable of handling both distance queries and updates to the graph such as length updates or insertions and deletions of edges. The best such data structures require linear space support each distance query or update operation in $\tilde{O}(n^{2/3})$ amortized time [Kle05, KMNS12].

As we have seen in Chapter 5, maximum flow with multiple sinks and sources in planar graphs can now be solved in $O(n \log^3 n)$ time. It would be interesting to get rid of at least some of the logarithmic factors. As we mentioned above, one way to achieve that, would be to improve the running time of FR-Dijkstra. Another interesting question is whether the same approach can be used to solve the problem quickly in more general classes of graphs such as bounded-genus graphs. As we have mentioned in the introduction to Part III, our MSMS max-flow algorithm can be used to improve the running time required for solving the maximum flow problem with multiple sources and sinks and vertex capacities, in the regime of a moderate number of nodes with finite capacity. Can the running time be improved without any assumptions on the number of nodes with finite capacity? Is near-linear time possible?

Finally, the technique used in the algorithm for maximum flow with multiple sources and a single sink in Chapter 6 is currently a solution looking for a problem. Perhaps this technique can be used in solving related problems such as minimum cost circulation, or other variants of maximum flow.

Bibliography

[ACC+96] S. R. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. H. M. Smid, and C. D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *Proceedings of the 4th European Symposium on Algorithms (ESA)*, pages 514–528, 1996. 3, 53

[AEOP02] R. Ahuja, Ö. Ergun, J. Orlin, and A. Punnen. A survey of very large scale neighborhood search techniques. *Discrete Applied Mathematics*, 23:75–102, 2002. 66

[AFGW10] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010. 52

[AHdT05] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005. 90

[AK90] A. Aggarwal and M. M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2):3–23, 1990. 19

[AKM+86] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix searching algorithm. In *Proceedings of the 2nd Annual Symposium on Computational Geometry (SOCG)*, pages 285–292, New York, NY, USA, 1986. ACM. 19

[AKM+87] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987. 46

[AT05] Lars Arge and Laura Toma. External data structures for shortest path queries on planar digraphs. In *Proceedings of the 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 328–338, 2005. 52

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. 15

[BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94, 2000. 21, 22

[BK04] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26:1124–1137, September 2004. 66

[BK09] G. Borradaile and P. N. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *J. ACM*, 56(2), 2009. Preliminary version in SODA 2006. 64, 66, 81, 82, 95

[BKM+11] G. Borradaile, P. N. Klein, S. Mozes, Y. Nussbaum, and C. Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 170–179, 2011. ix, 3, 4, 50, 65

[BKR96] R. E. Burkard, B. Klinz, and R. Rudolf. Perspectives of monge properties in optimization. *Discrete Applied Mathematics*, 70:95–161, 1996. 18

[BMSU99] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M)*, pages 48–55, 1999. 2

[Bor08] G. Borradaile. *Exploiting Planarity for Network Flow and Connectivity Problems*. PhD thesis, Brown University, 2008. 64

[BSWN10] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen. Min st-cut oracle for planar graphs with near-linear preprocessing time. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 601–610, 2010. 3, 25, 26, 95

[BV93] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993. 21

[BVZ01] Y. Boykov, O. Veksler, and R. Zabih. Efficient approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1222–1239, 2001. 66

[Cab12] S. Cabello. Many distances in planar graphs. *Algorithmica*, 62(1-2):361–381, February 2012. Preliminary version in SODA 2006. 3, 53, 54, 61

[CC00] W.-T. Chan and F. Y. L. Chin. Efficient algorithms for finding the maximum number of disjoint paths in grids. *Journal of Algorithms*, 34(2):337–369, 2000. Preliminary version in SODA 1997. 67

[CC07] S. Cabello and E.W. Chambers. Multiple source shortest paths in a genus g graph. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 89–97, 2007. 17, 77, 79, 94

[CCT99] W.-T. Chan, F. Y. L. Chin, and H.-F. Ting. A faster algorithm for finding disjoint paths in grids. In *Proceedings of the 10th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 393–402, 1999. 67

[CCT03] W.-T. Chan, F. Y. L. Chin, and H.-F. Ting. Escaping a grid by edge-disjoint paths. *Algorithmica*, 36(4):343–359, 2003. Preliminary version in SODA 2000. 67

[CEN09] E. W. Chambers, , J. Erickson, and A. Nayyeri. Homology flows, cohomology cuts. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 273–282, 2009. 3, 35

[CKM+11] P. Christiano, J. A. Kelner, A. Mądry, D. A. Spielman, and S. Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 273–282, 2011. 64

[CR10] S. Cabello and G. Rote. Obnoxious centers in graphs. *SIAM Journal on Discrete Mathematics*, 24(4):1713–1730, 2010. 3

[CRZ96] I. Cox, S. Rao, and Y. Zhong. Ratio regions: A technique for image segmentation. *International Conference on Pattern Recognition*, 02:557, 1996. 35

[CX00] D.Z. Chen and J. Xu. Shortest path queries in planar graphs. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 469–478, 2000. 3, 53, 54

[Dan58] G. B. Dantzig. On the shortest route through a network. RAND Report P-1345, The Rand Corporation, Santa Monica, CA, April 1958. published in Management Science 6 (1960) 18–190. 14

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390. 14

[Dji96] Hristo Djidjev. Efficient algorithms for shortest path problems on planar digraphs. In *22nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 151–165, 1996. 3, 53, 54, 61

[DPZ00] H. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. Improved algorithms for dynamic shortest paths. *Algorithmica*, 28(4):367–389, 2000. 54

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86– 124, 1989. 17

[Edm60] J. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices of the American Mathematical Society*, 7:646, 1960. 7

[EFS56] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *IEEE Transactions on Information Theory*, 2(4):117–119, 1956. 27

[EG08] David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *16th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (ACM-GIS)*, page 16, 2008. 52

[EIT+90] D. Eppstein, G. Italiano, R. Tamassia, R. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1 – 11, 1990. 12

[Epp97] D. Eppstein. Dynamic connectivity in digital images. *Information Processing Letters*, 62(3):121–126, May 1997. 2

[Epp99] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999. 54

[Eri10] J. Erickson. Maximum flows and parametric shortest paths in planar graphs. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 794–804, 2010. 64

[Eri11] J. Erickson. personal communication, 2011. 35

[Eul41] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741. 2

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. 27, 66

[FF57] L. R. Ford and D. R. Fulkerson. Construction of maximal dynamic flows in networks. RAND Report P-1079 (RM-1981), The Rand Corporation, Santa Monica, CA, May 1957. published in Operations Research 6 (1958) 419–433. 16

[FMS91] E. Feuerstein and A. Marchetti-Spaccamela. Dynamic algorithms for shortest paths in planar graphs. In *17th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 187–197, 1991. 53

[For56] L. R. Ford. Network flow theory. RAND Report P-923, The Rand Corporation, Santa Monica, CA, August 1956. 15

[FR06] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006. Preliminary version in FOCS 2001. 3, 19, 21, 22, 34, 53, 54, 69, 70

[Fre87] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987. Preliminary version in FOCS 1983. 14

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. 15, 34

[GBT84] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984. 21

[GG84] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian relation of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–742., 1984. 66

[Gol95] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995. 34

[Gol98] A. V. Goldberg. Recent developments in maximum flow algorithms. In Stefan Arnborg and Lars Ivansson, editors, *Algorithm Theory SWAT'98*, volume 1432 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 1998. 64

[GPS89] D. Greig, B. Porteous, and A. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B*, 51(2):271–279, 1989. 2, 66

[GR98] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998. Preliminary version in FOCS 1997. 4, 64

[GT88] A. V. Goldberg and R. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988. 27

[GT89] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989. 34

[GT90] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990. 27

[GTT90] A. V. Goldberg, Éva Tardos, and R. E. Tarjan. Network flow algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Algorithms and Combinatorics Volume 9: Flows, Paths, and VLSI-Layout*, pages 101–164. Springer-Verlag, 1990. 64

[GW05] A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In *7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 26–40, 2005. 52

[Has81] R. Hassin. Maximum flow in (s, t) planar networks. *Information Processing Letters*, 13:107, 1981. 4, 29, 30, 64

[Hef91] L. Heffter. Über das problem der nachbargebiete. *Mathematische Annalen*, 38:477–508, 1891. 7

[HKRS97] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. 3, 34, 41, 54, 55, 59, 62, 64, 94

[HMZ03] D. A. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126(1):55–82, 2003. Preliminary version in COCOON 1999. 52

[Hoc01] D. S. Hochbaum. An efficient algorithm for image segmentation, Markov random fields and related problems. *Journal of the ACM*, 48(4):686–701, 2001. 66

[Hoc08] D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008. 27

[Hof61] A. J. Hoffman. On simple linear programming problems. In *Convexity. Proceedings of the 7th Symposium in Pure Mathematics of the American Mathematical Society*, pages 317–327, 1961. 18

[HR55] T. E. Harris and F. S. Ross. Fundamentals of a method for evaluating rail net capacities. Research Memorandum RM-1573, The RAND Corporation, Santa Monica, CA, October 1955. declassified 1999. 2, 65

[HS97] J. Hershberger and S. Suri. Efficient breakout routing in printed circuit boards. In *Proceedings of the 13th Annual Symposium on Computational Geometry (SOCG)*, pages 460–462, 1997. 67

[HW07] J. M. Hochstein and K. Weihe. Maximum $s - t$-flow with k crossings in $O(k^3 n \log n)$ time. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 843–847, 2007. 65

[IJ01] H. Ishikawa and I. Jermyn. Region extraction from multiple images. In *Proceedings of the 8th International Conference on Computer Vision (ICCV)*, pages 509–516, 2001. 35

[INSWN11] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 313–322, New York, NY, USA, 2011. ACM. 3, 14, 64

[IS79] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8:135–150, 1979. 30

[JHR96] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportation applications. In *Proceedings of the 5th International Conference on Information and Knowledge Management (CIKM)*, pages 261–268, 1996. 52

[JI01] I. H. Jermyn and H. Ishikawa. Globally optimal regions and boundaries as minimum ratio weight cycles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1075–1088, 2001. 35

[Joh77] D. B. Johnson. Efficient algorithms for shortest paths in sparse graphs. *Journal of the ACM*, 24:1–13, 1977. 15, 16, 34

[JV82] D. B. Johnson and S. Venkatesan. Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time. In *Proceedings of the 20th Annual Allerton Conference on Communication, Control, and Computing*, pages 898–905, 1982. 31, 73

[KK90] M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal On Discrete Math*, 3(1):81–97, 1990. 19, 40

[KK06] L. Kowalik and M. Kurowski. Oracles for bounded-length shortest paths in planar graphs. *ACM Transactions on Algorithms*, 2(3):335–363, 2006. Preliminary version in STOC 2003. 54

[KKS11] K. Kawarabayashi, P. N. Klein, and C. Sommer. Linear-space approximate distance oracles for planar, bounded-genus, and minor-free graphs. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 135–146, 2011. 54, 62, 95

[Kle02] P. N. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 820–827, New York, January 6–8 2002. ACM Press. 54

[Kle05] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 146–155, 2005. 4, 17, 48, 53, 54, 55, 57, 60, 77, 79, 83, 91, 94, 95

[KM11] P. N. Klein and S. Mozes. Multiple-source single-sink maximum flow in directed planar graphs in $o(\text{diameter} \cdot n \log n)$ time. In *Proceedings of the 14th International Workshop on Algorithms and Data Structures (WADS)*, pages 571–582, 2011. ix, 65

[KMNS12] H. Kaplan, S. Mozes, Y. Nussbaum, and M. Sharir. Submatrix maximum queries in monge matrices and monge partial matrices, and their applications. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 338–355, 2012. 3, 25, 50, 51, 95

[KMW09] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA '09, pages 236–245, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. ix

[KMW10] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. *ACM Transactions on Algorithms*, 6(2):1–18, 2010. Preliminary version in SODA 2009. ix, 3, 35, 53

[KN09] H. Kaplan and Y. Nussbaum. Maximum flow in directed planar graphs with vertex capacities. In *Proceedings of the 17th European Symposium on Algorithms (ESA)*, pages 397–407, 2009. 31

[KNK93] S. Khuller, J. Naor, and P. Klein. The lattice structure of flow in planar graphs. *SIAM Journal on Discrete Mathematics*, 6(3):477–490, 1993. 29, 31

[KS98] P. N. Klein and S. Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22(3):235–249, 1998. 14, 69

[KT02] J. M. Kleinberg and Éva Tardos. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and Markov random fields. *Journal of the ACM*, 49(5):616–639, 2002. Preliminary version in FOCS 1999. 66

[Kur30] K. Kuratowski. Sur le probléme des courbes gauches en topologie. *Fundamenta Mathematicae*, 15:271–283, 1930. 7

[Lei80] C. E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 270–281, 1980. 2

[LGJ+57] M. Leyzorek, R. S. Gray, A.A. Johnson, W. C. Ladew, S. R. Meaker, R. M. Petry, and R. N. Seitz. A study of model techniques for communication systems. First annual report, Case Institute of Technology, Cleveland, Ohio, 1957. 14

[LRT79] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979. 2, 34

[ŁS11] J. Łkacki and P. Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Proceedings of the 19th European Symposium on Algorithms (ESA)*, pages 155–166, 2011. 3

[LT79] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. 3, 13

[Mac37] S. MacLane. A combinatorial condition for planar graphs. *Fundamenta Mathematicae*, pages 22–32, 1937. 7

[Mil86] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986. Preliminary version in STOC 1984. 13, 68

[Mit90] J. S. B. Mitchell. On maximum flows in polyhedral domains. *Journal of Computer and System Sciences*, 40(1):88–123, 1990. 67

[MN95] G. L. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24(5):1002–1017, 1995. Preliminary version in FOCS 1989. 3, 16, 29, 35, 64, 84, 87

[Mon81] G. Monge. Mémoire sur la théorie des déblais et ramblais. *Mém. Math. Phys. Acad. Roy. Sci. Paris*, pages 666–704, 1781. 18

[Moo57] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, pages 285–292, 1957. 15

[MS12] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 209–222, 2012. ix, 3, 53

[MWN10] S. Mozes and C. Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log\log n)$ time. In *Proceedings of the 18th European Symposium on Algorithms (ESA)*, pages 206–217, 2010. ix, 3, 36

[Nus11] Y. Nussbaum. Improved distance queries in planar graphs. In *Proceedings of the 14th International Workshop on Algorithms and Data Structures (WADS)*, pages 642–653, 2011. 3, 54, 55, 61

[PR10] M. Patrascu and L. Roditty. Distance oracles beyond the Thorup–Zwick bound. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 815–823, 2010. 61

[RBK90] V. P. Roychowdhury, J. Bruck, and T. Kailath. Efficient algorithms for reconfiguration in VLSI/WSI arrays. *IEEE Transactions on Computers*, 39(4):480–489, 1990. 2, 67

[Rei83] J. Reif. Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM Journal on Computing*, 12:71–81, 1983. 84

[RLWW95] H. Ripphausen-Lipa, D. Wagner, and K. Weihe. The vertex-disjoint Menger problem in planar graphs. *SIAM Journal on Computing*, 24(5):1002–1017, 1995. 18

[Sch98] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998. Preliminary version in ISTCS 1995. 54

[Sch05] A. Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Handbook of Dicrete Optimization*, pages 1–68. Elsevier, 2005. 14, 15, 64

[Shi55] A. Shimbel. Structure in communication nets. In *Proceedings of the Symposium on Information Networks, New York (1954)*, pages 199–203, Brooklyn, NY, 1955. Polytechnic Press of the Polytechnic Institute of Brooklyn. 15

[Som29] D. Sommerville. *An introduction to the geometry of n dimensions*. London, 1929. 12

[Som10] Christian Sommer. *Approximate Shortest Path and Distance Queries in Networks*. PhD thesis, The University of Tokyo, 2010. 52

[Som11] C. Sommer. More compact oracles for approximate distances in planar graphs. *CoRR*, abs/1109.2641, 2011. 53

[ST83] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. 4, 64

[STC09] F. R. Schmidt, E. Toeppe, and D. Cremers. Efficient planar graph cuts with applications in computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Miami, Florida, June 2009. 66

[Sub95] S. Subramanian. *Parallel and Dynamic Shortest-Path Algorithms for Sparse Graphs*. PhD thesis, Brown University, 1995. Available as Brown University Computer Science Technical Report CS-95-04. 14

[SVY09] C. Sommer, E. Verbin, and W. Yu. Distance oracles for sparse graphs. In *Proceedings of the 50th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 703–712, 2009. 61

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975. 19

[Tho04] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004. 54

[Vek02] O. Veksler. Stereo correspondence with compact windows via minimum ratio cycle. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(12):1654–1660, 2002. 35

[vS47] G. K. C. von Staudt. *Geometrie der Lage*. Verlag Bauer und Raspe, Nürnberg, 1847. Par. 49, pages 20–21. 12

[Whi33] H. Whitney. Planar graphs. *Fundamenta mathematicae*, 21:73–84, 1933. 7, 10

[WN10a] C. Wulff-Nilsen. *Algorithms for Planar Graphs and Graphs in Metric Spaces*. PhD thesis, University of Copenhagen, 2010. 54, 61, 62

[WN10b] C. Wulff-Nilsen. Min st-cut of a planar graph in $O(n \log \log n)$ time. *CoRR*, abs/1007.3609, 2010. 14

[YD95] M.-F. Yu and W. W.-M. Dai. Single-layer fanout routing and routability analysis for ball grid arrays. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design (ICCAD)*, pages 581–586, 1995. 67

[You63] J. Youngs. Minimal imbeddings and the genus of a graph. *Journal of Mathematical Mechanic*, 12:303–315, 1963. 7

[Zar08] C. Zaroliagis. Engineering algorithms for large network applications. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer US, 2008. 52

[Zie77] O. C. Zienkiewicz. *The Finite Element Method in Engineering Science*. McGraw-Hill, New York, NY, USA, 1977. 2