Flatworlds: Optimization Algorithms for Planar Graphs

Philip N. Klein

October 21, 2011

Contents

Chapter 1

Rooted forests and trees

The notion of a rooted forest should be familiar to the reader. For completeness, we will give formal definitions.

Let N be a finite set. A *rooted tree on* N is defined by the pair (N, p) where p is a function $p : N \longrightarrow N \cup \{\perp\}$ such that

- there is no positive integer k such that $p^k(x) = x$ for some element $x \in N$, and

- there is exactly one element $x \in N$ such that $p(x) = \perp$.

This element is called the *root.* The elements of N are called the *nodes* of the tree. They are also called the *vertices* of the tree.

For each nonroot node x, $p(x)$ is called the *parent* of x in the tree, and the ordered pair $xp(x)$ is called the *parent arc* of x in the tree. An *arc* of the tree is the parent arc of some nonroot node. If the parent of x is y then x is a *child* of y and xy is the *child arc* of y.

A rooted tree has *arity* k if every node has at most k children. A *binary* tree is a tree that has arity 2.

We say two nodes are *adjacent* if one is the parent of the other. We say the arc $xp(x)$ is *incident* to the nodes x and $p(x)$.

The *ancestors* of x are defined inductively: x is its own ancestor, and (if x is not the root) the ancestor's of x's parent are also ancestors of x. If x is the ancestor of y then y is a *descendant* of x. We say y is a *proper* ancestor of x (and x is a *proper* descendant of y) if y is an ancestor of x and $y \neq x$. The *depth* of a node is the number of proper ancestors it has.

We say an arc $xp(x)$ is an *ancestor arc* of y if x is an ancestor of y. We say $xp(x)$ is a *descendant arc* of y if $p(x)$ is a descendant of y.

A *subtree* of (N, p) is a tree (N', p') such that N' is a subset of N. A *rooted forest* is a collection of disjoint rooted trees. That is, (N, p) is a forest if there are trees $(N_1, p_1), \ldots, (N_k, p_k)$ such that

$$N = N_1 \dot\cup N_2 \dot\cup \cdots \dot\cup N_k$$

and p_i is the restriction of p to N_i.

Deletion of an arc $xp(x)$ from a rooted forest (N, p) is an operation that yields the forest (N, p') where

$$p'(x) = \begin{cases} \perp & \text{if } x = \hat{x} \\ p(x) & \text{otherwise} \end{cases}$$

If T is a rooted forest and e is an arc of T then we use $T - \{e\}$ to denote the result of deleting e.

Deletion of a node $\hat{x}$ from a rooted forest (N, p) is an operation that yields the forest $(N - \{\hat{x}\}, p')$ where

$$p'(x) = \begin{cases} \perp & \text{if } p(x) = \hat{x} \\ p(x) & \text{otherwise} \end{cases}$$

If T is a rooted forest and $\hat{x}$ is a node of T then we use $T - \{x\}$ to denote the result of deleting x.

More generally, if S is a set of nodes or a set of arcs, $T - S$ denotes the forest obtained by deleting every element of S.

For a tree T and a node x of T, the *subtree rooted at x* is the tree obtained from T by deleting every node that is not a descendant of x.

For a forest T and a node x of T, the *root-to-x path* is the sequence $x_0 x_1 \ldots x_k$ where x_0 is the root of T, x_k is x, and x_i is the parent of x_{i+1} for $i = 0, \ldots, k-1$. We denote this path by $T[x]$.

Ancestorhood defines a partial order among nodes of a forest. Given a set S of nodes of a forest, a *rootmost* node of S in the forest is a node v such that no proper ancestor of v is in S. A *leafmost* node of S is a node v such that no proper descendant of v is in S.

Given two nodes u and v of a forest, we say u is *leafward* of v and v is *rootward* of u if u is a descendant of v. A sequence $v_1, \ldots, v_k$ of nodes of the forest is a *leafward* path if v_i's parent is v_{i+1} for $i = 1, \ldots, k-1$.

1.1 Rootward computations

Suppose T is a rooted tree and $w(\cdot)$ is an assignment of weights to the nodes. There is a simple, linear-time algorithm to compute, for each node u, the total weight of all descendants of u:

```
def totalWeight($u$):
    return $w(u) + \sum\{$totalWeight($v$) : $v$ a child of $u\}$
```

This algorithmic schema, though simple, comes up again and again: in finding separators for trees (in the next section), in algorithms that exploit interdigitating trees in planar graphs (Section 4.5, in processing a breadth-first-search tree (Section 5.4), in dynamic-programming algorithms on trees (Section 13.1) and on graphs of bounded carvingwidth (Section 13.3.1) and bounded branchwidth (Section 13.5.1).

1.2 Separators for rooted trees

Should have pictures of separators.

A *separator* for a tree is a vertex or edge whose deletion results in trees that are "small" in comparison to the original graph.

Lemma 1.2.1 (Leafmost Heavy Vertex)**.** *Let T be a rooted tree. Let $\hat{w}(\cdot)$ be an assignment of weights to vertices such that the weight of each vertex is at least the sum of the weights of its children. Let W be the weight of the root, and let α be a positive number less than 1. Then there is a linear-time algorithm to find a vertex v_0 such that $\hat{w}(v_0) > \alpha W$ and every child v of v_0 satisfies $\hat{w}(v) \leq \alpha W$.*

Proof. Call the procedure below on the root of T.

```
define $f(v)$:
1    if some child $u$ of $v$ has $\hat{w}(u) > \alpha W$,
2        return $f(u)$
3        else return $v$
```

By induction on the number of invocations, for every call $f(v)$, we have $\hat{w}(v) > \alpha W$. If v is a leaf then the condition in Line 1 is not satisfied, so the procedure terminates. Let v_0 be the vertex returned by the procedure. Since the condition in Line 1 did not hold for v_0, every child v of v_0 satisfies $\hat{w}(v) \leq \alpha W$. □

1.2.1 Vertex separator

Lemma 1.2.2 (Tree Vertex Separator)**.** *Let T be a rooted tree, and let $w(\cdot)$ be an assignment of weights to vertices. Let W be the sum of weights. There is a linear-time algorithm to find a vertex v_0 such that every component in $T - \{v_0\}$ has total weight at most $W/2$.*

Proof. For each vertex u, define $\hat{w}(u) = \sum\{w(v) \ : \ v \text{ a descendant of } u\}$. Then $\hat{w}(\text{root}) = W$. The values $\hat{w}(\cdot)$ can be computed using a rootward computation as in Section 1.1. Let v_0 be the vertex of the Leafmost-Heavy-Vertex Lemma with $\alpha = 1/2$. Let $v_1, \ldots, v_p$ be the children of v_0. For each child v_i, the subtree rooted at v_i has weight at most $W/2$. Each such subtree is a tree of $T - \{v_0\}$. The remaining tree is $T - \{v \ : \ v \text{ is a descendant of } v_0\}$. Since the sum $\sum_{v \text{ is a descendant of } v_0} w(v) = \hat{w}(v_0)$ exceeds $W/2$, the weight of the remaining tree is less than $W/2$. □

1.3 Edge separators

For some separators, we need to impose a condition on the weight assignment. We say a weight assignment is *α-proper* if no element is assigned more than an α fraction of the total weight.

Lemma 1.3.1 (Tree Edge Separator of Edge-Weight)**.** *Let T be a tree of degree at most three, and let $w(\cdot)$ be a $\frac{1}{3}$-proper assignment of weights to edges. There is a linear-time algorithm to find an edge $\hat{e}$ such that every component in $T - \{\hat{e}\}$ has at most two-thirds of the weight.*

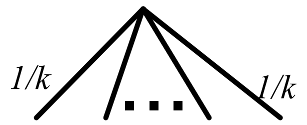
Proof. Assume for notational simplicity that the total weight is 1. Choose a vertex of degree one as root. For each nonroot vertex v, define

$$\hat{w}(v) = \sum\{w(e) \ : \ e \text{ a descendant edge of } v\} \cup \{\text{parent edge of } v\}$$

Define $\hat{w}(root) = 1$. Let v_0 be the vertex of the Leafmost-Heavy-Vertex Lemma with $\alpha = 1/3$. Let e_0 be the parent edge of v. Then $T - \{e_0\}$ consists of two trees. One tree consists of all descendants of v_0, and the other consists of all nondescendants.

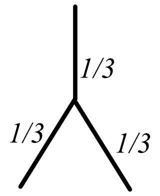
The weight of all edges among the nondescendants is $1 - \hat{w}(v_0)$, which is less than $1 - 1/3$ since $\hat{w}(v_0) > 1/3$. Let $v_1, \ldots, v_p$ be the children of v_0. (Note that $1 \leq p \leq 2$.) The weight of all edges among the descendants is $\sum_{i=1}^{p} \hat{w}(v_i)$. Since $w(v_i) \leq 1/3$ for $i = 1, \ldots, p$ and $p \leq 2$, we infer $\sum_i \hat{w}(v_i) \leq 2/3$. □

The following example shows that the restriction on the arity of the trees in Lemma 1.3.1 cannot be discarded:



If the number of children is k then removal of any edge leaves weight $(k-1)/k$.

The following example shows that, for trees of degree at most three, the factor two-thirds in Lemma 1.3.1 cannot be improved upon.



Lemma 1.3.2 (Tree Edge Separator of Vertex Weight)**.** *Let T be a tree of degree at most three, and let $w(\cdot)$ be a $\frac{3}{4}$-proper assignment of weights to vertices such that each nonleaf vertex is assigned at most one-fourth of the weight. There is a linear-time algorithm to find an edge $\hat{e}$ such that every component $T - \{\hat{e}\}$ has at most three-fourths of the weight.*

Proof. Assume the total weight is 1. Root T at a leaf. For each vertex v, define

$$\hat{w}(v) = \sum\{w(v') \ : \ v' \text{ a descendant of } v\}$$

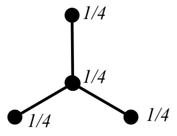
Let v be the vertex of the Leafmost-Heavy-Vertex Lemma with $\alpha = 3/4$. Let $v_1, \ldots, v_p$ be the children of v. Note that $0 \leq p \leq 2$. Since $w(v) \leq \frac{3}{4}$ but $\hat{w}(v) > \frac{3}{4}$, we must have $p > 0$, so $w(v) \leq \frac{1}{4}$.

For $1 \leq i \leq p$, let W_i be the weight of descendants of v_i. Let $\hat{i} = \text{maxarg}_{1 \leq i \leq p} W_i$. By choice of v, $W_{\hat{i}} \leq \frac{3}{4}$. By choice of $\hat{i}$,

$$W_{\hat{i}} \geq \frac{1}{2}\sum_{i=1}^{p} W_i > \frac{1}{2}(\frac{3}{4} - w(v_0)) \geq \frac{1}{2}(\frac{3}{4} - \frac{1}{4}) = W/4$$

This shows that choosing $\hat{e}$ to be the edge $v_{\hat{i}}v$ satisfies the balance condition. □

The following example shows that the factor three-fourths in Lemma 1.3.2 cannot be improved upon.



By changing our goal slightly, we can get a better-balanced separator.

Lemma 1.3.3 (Tree edge separator of Vertex/Edge Weight)**.** *Let T be a binary tree, and let $w(\cdot)$ be a $\frac{1}{3}$-proper assignment of weight to the vertices and edges such that degree-three vertices are assigned zero weight. There is a linear-time algorithm to find an edge e such that every component of $T - e$ has at most two-thirds of the weight.*

Problem 1.3.4. *Prove Lemma 1.3.3.*

1.4 Recursive tree decomposition

Problem 1.4.1. *A recursive edge-separator decomposition for an unrooted tree T is a rooted tree D such that*

- *the root r of D is labeled with an edge e of T;*
- *for each connected component K of $T - e$ (there are at most two), r has a child in D that is the root of a recursive edge-separator decomposition of K.*

Show that there is an $O(n \log n)$ algorithm that, given a tree T of maximum degree three and n nodes, returns a recursive edge-separator decomposition of depth $O(\log n)$

1.5 Data structure for sequences and rooted trees

In the Appendix, we describe data structures for representing sequences and rooted trees.

Problem 1.5.1. *Show that the data structure for representing trees can be used to quickly find edge-separators in binary trees. Use this idea to give a fast algorithm that, given a tree of maximum degree three, returns a recursive edge-separator decomposition of depth $O(\log n)$.* Note: *A running time of $O(n)$ can be achieved.*

Chapter 2

Basic graph definitions

To quote Berge,

> It would be convenient to say that there are two theories and two kinds of graphs: directed and undirected. This is not true. *All graphs are directed,* but sometimes the direction need not be specified.

That is, for specific graph problems it is convenient to ignore the distinction between endpoints.

We define one combinatorial structure, a *graph.*[1] There are three ways to interpret this combinatorial structure, as an *undirected* graph, as a *directed* graph, and as a *bidirected* graph. Each kind of graph has its uses, and it is convenient to be able to view the underlying graph from these different perspectives.

In the traditional definition of graphs, vertices are in a sense primary, and edges are defined in terms of the vertices. We used this approach in defining rooted trees in Chapter 1. In defined graphs, we choose to make edges primary, and we will define vertices in terms of edges.

There are three reasons for choosing the edge-centric view:

- Self-loops and multiple edges, which occur often, are more simply handled by an edge-centric view.
- Contraction, a graph operation we discuss later, transforms a graph in a way that changes the identity of vertices but not of edges. The edge-centric view is more natural in this context, and simplifies the tracking of an edge as the graph undergoes contractions.
- The dual of an embedded graph is usefully viewed as a graph with the same edges, but where those edges form a different topology.

There is one seeming disadvantage: our definition of graphs does not permit the existence of isolated vertices, vertices with no incident edges. This disadvantage

[1] Our definition allows for self-loops and multiple edges, a structure traditionally called a *multigraph.*

is mitigated by another odd aspect of our approach: a subgraph of a graph is not in itself an independent graph but depends parasitically on the original graph.

2.1 Edge-centric definition of graphs

For any finite set E, a *graph on* E is a pair $G = (V, E)$ where V is a partition of the set $E \times \{1, -1\}$, called the *dart set* of G. That is, V is a collection of disjoint, nonempty, mutually exhaustive subsets of $E \times \{1, -1\}$. Each subset is a *vertex* of G. (The word *node* is synonymous with *vertex*). For any $e \in E$, the *darts of* e are the pairs $(e, +1)$ and $(e, -1)$, of which the *primary dart of* e is $(e, +1)$. For brevity, we can write $(e, +1)$ as e^+ and $(e, -1)$ as e^-

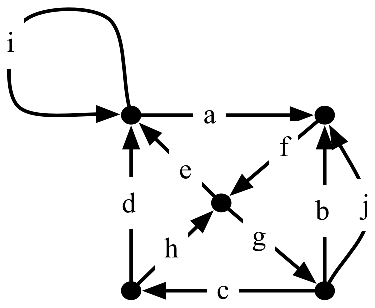


Figure 2.1: The vertex v is the subset of darts $\{(e, 1), (f, -1), (g, 1), (h, -1)\}$. An example of a walk is $(j, 1)$ $(a, -1)$ $(i, 1)$ $(i, -1)$ $(d, -1)$ $(d, 1)$.

rev Define the bijection rev on darts by $\text{rev}((e, \sigma)) = (e, -\sigma)$. For a dart d, $\text{rev}(d)$ is called the *reverse* of d, and is sometimes written as d^R.

endpoints, head and tail, self-loops, parallel edges The *tail* of a dart (e, σ) is the block $v \in V$ such that v contains (e, σ). The *head* of (e, σ) is the tail of $\text{rev}((e, -\sigma))$.

Each element $e \in E$ has two *endpoints*, namely the head and tail of $(e, 1)$. If the endpoints are the same vertex, we call e a *self-loop*. In Figure 2.1, i is a self-loop. If two elements have the same endpoints, we say they are *parallel*, for example, b and j are parallel in Figure 2.1.

Edges and arcs We can interpret an element $e \in E$ as a directed *arc*, in which case we distinguish between its head and tail, which are, respectively, the head and tail of the primary dart $(e, +1)$. If we interpret e as an undirected *edge*, we do not distinguish between its endpoints. Thus use of the word *edge* or *arc* indicates whether we intend to interpret the element as undirected or directed. The edge or arc of a dart (e, σ) is defined to be e.

Parallel arcs/edges and self-loops If two arcs have the same tail and the same head, we say they are *parallel arcs*. If two edges have the same pair of endpoints, we say they are *parallel edges*. If the endpoints of an edge/arc are the same, we say it is a *self-loop*. Our definition of graph permits parallel edges and self-loops.

Incidence, degree We say an edge/arc/dart is *incident* to a vertex v if v is one of the endpoints. The *degree* of a vertex v (written degree(v)) is the number of occurences of v as an endpoint of elements of E (counting multiplicity[2]). The outdegree of v (written outdegree(v)) is the number of arcs having v as a tail, and the indegree (written indegree(v)) is the number of arcs having v as a head.

Endpoint notation We sometimes write an arc as uv to indicate that its tail is u and its head is v, and we sometimes write an edge the same way to indicate that its endpoints are u and v. This notation has the potential to be ambiguous because of the possibility of parallel edges.

$V(G)$ **and** $E(G)$ For a graph $G = (V, E)$, we use $V(G)$ and $E(G)$ to denote V and E, respectively, and we use $n(G)$ and $m(G)$ to denote $|V(G)|$ and $|E(G)|$. We use $D(G)$ to denote the set of darts of G. We may leave the graph G unspecified if doing so introduces no ambiguity.

Is $n(G)$ or $m(G)$ ever used?

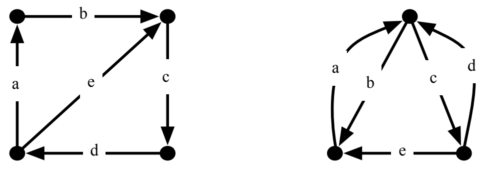


Figure 2.2: Two graphs corresponding to the edges $a, \ldots, e$.

2.2 Walks, paths, and cycles

Walks As illustrated in Figure 2.1, a non-empty sequence

$$d_1 \ldots d_k$$

of darts is a *walk* if the head of d_i is the tail of d_{i+1} for every $1 \leq i \leq k$. To be more specific, it is a *x-to-y walk* if x is d_1 or the tail of d_1 and y is d_k or the head of d_k. We define d_i to be the *successor* in W of d_i to be d_{i+1} and we define *predecessor* of d_{i+1} to be d_i. We may designate a walk to be a *closed walk* if the tail of d_1 is the head of d_k, in which case we define the successor of d_k to be d_1 and the predecesor of d_1 to be d_k.

[2]That is, a self-loop contributes two to the degree of a vertex.

Paths and cycles A walk is called a *path of darts* if the darts are distinct, a *cycle of darts* if in addition it is a closed walk. A path/cycle of darts is called a *path/cycle of arcs* if each dart is of the form $(e, +1)$. It is called a *path/cycle of edges* if no edge is represented twice.

Simple paths and cycles, internal vertices A cycle is *simple* if every vertex occurs at most once as the head of some d_i. A path is simple if it is not a cycle and every vertex occurs at most once as the head of some d_i. A vertex is said to belong to the path or cycle if the vertex is an endpoint of some d_i. The *internal vertices* of a path $d_1 \ldots d_k$ are the heads of $d_1, \ldots, d_{k-1}$. Two paths/cycles are dart-disjoint if they share no darts, and are vertex-disjoint if they share no vertices. Two paths are *internally vertex-disjoint* if they share no internal vertices.

Walks, paths, and cycles of arcs/edges A sequence $e_1, \ldots, e_k$ of elements of E is a *directed walk* (or *diwalk*) if the sequence of corresponding darts $(e_1, 1), \ldots, (e_k, 1)$ is a walk. It is a *directed path* (or *dipath*) if, in addition, $e_1, \ldots, e_k$ are distinct. It is an undirected walk if there exist $i_1, \ldots, i_k \in \{1, -1\}$ such that the sequence of darts $(e_1, i_1), \ldots, (e_k, i_k)$ is a walk. It is an undirected path if in addition $e_1, \ldots, e_k$ are distinct. The other definitions given for sequences of darts apply straightforwardly to paths consisting of elements of E.

Empty walks and paths In the above, we neglected to account for the possibility of an *empty* walk or path. Empty walks and paths are defined by a vertex in the graph; they contain no darts. We do not allow for the existence of empty cycles.

Lemma 2.2.1. *A u-to-v walk of darts contains a u-to-v path of darts as a subsequence.*

2.2.1 Connectedness

Given a graph $G = (V, E)$, for a vertex or dart x and a vertex or dart y, we say x and y are *connected* in G if there is a v_1-to-v_2 path of darts in G. Similarly, edges e_1 and e_2 are connected in G if there is a path of darts that starts with a dart of e_1 and ends with a dart of e_2.

More generally, given a subset E' of E, we say that v_1, v_2 are connected via E' in G if there is a v_1-to-v_2 path using only darts corresponding to edges of E'.

A subset of V is connected in a graph if every two vertices in the subset are connected. Connectedness is an equivalence relation on the vertex set. A *connected component* is an equivalence class of this equivalence relation. Equivalently, a connected component is a maximal connected vertex subset. Let $\kappa(G)$ denote the number of connected components of G.

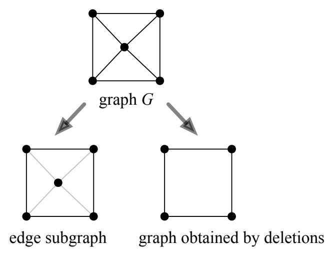


Figure 2.3: This figure illustrates the difference between an edge subgraph (shown on the bottom-left) and a traditional subgraph, a graph obtained by edge deletions (shown on the bottom-right). In the graph obtained by deletions, the center vertex does not exist since all its incident edges have been deleted. The edge subgraph does not formally include the grayed-out edges but still contains the center vertex. There are other advantages to the edge subgraph that we will discuss in the context of graph embeddings.

2.2.2 Subgraphs and edge subgraphs

We will use the term *subgraph* in two ways. According to the traditional definition, a *subgraph* of a graph $G = (V, E)$ is simply a graph $H = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Because we often want to relate features of a subgraph to the graph from which it came, we will define an *edge subgraph* of G as a pair (G, E') where $E' \subseteq E(G)$.

If it is clear which graph G is intended, we will sometimes use an edge-set E' to refer to the corresponding edge subgraph (G, E').

One significant distinction between a graph and an edge subgraph is this: according to our definition, a graph G cannot contain a vertex with no incident edges, whereas an edge subgraph (G, E') can contain a vertex v (a vertex of G) none of whose incident edges belong to E'.

The usual definitions (walk, path, cycle, connectedness) extend to an edge subgraph by restricting the darts comprising these structures to those darts corresponding to edges in E'. For example, two vertices x and y of G are connected in (G, E') if there is an x-to-y path of darts belonging to E'. As in graphs, a connected component of an edge subgraph of G is a maximal connected subset of $V(G)$. We define $\kappa((G, E'))$ to be the number of connected components in this sense. For example, the edge subgraph on the bottom-left in Figure 2.3 has two connected components. (The graph on the bottom-right has only one.)

2.2.3 Deletion of edges and vertices

Deleting a set S of *edges* from G is the operation on a graph that results in the subgraph or edge subgraph of G consisting of the edges of G not in S. We denote this subgraph or edge subgraph by $G - S$.

The result of *deleting* a set V' of *vertices* from G is the graph (not the edge subgraph) obtained by deleting all the edges incident to the vertices in V'. This subgraph is denoted $G - V'$. Since isolated vertices (vertices with no incident edges) cannot exist according to our definition of graphs, deleted vertices cease to exist when deleted.

Deletion of multiple edges and/or vertices results in a graph or edge-subgraph that is independent of the order in which the deletions occured.

2.2.4 Contraction of edges

For a graph $G = (V, E)$ and an edge $uv \in E$, the *contraction of* e *in* G is an operation that produces the graph $G' = (V', E')$, where

- $E' = E - \{uv\}$, and
- the part of V containing u and the part of V containing v are merged (and uv is removed) to form a part V'.

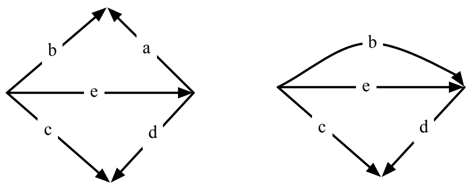


Figure 2.4: (a) A graph with an edges $a, \ldots, e$. (b) The graph after the contraction of edge a.

Like deletions, the order of contractions of edges does not affect the result. For a set S of edges, the graph obtained by contracting the edges of S is denoted G/S.

2.2.5 Minors

A graph H is said to be a *minor* of a graph G if H can be obtained from G by edge contractions and edge deletions. The relation "is a minor of" is clearly reflexive, transitive, and antisymmetric.

Note that each vertex v of H corresponds to a *set* of vertices in G (the set merged to form v).

Chapter 3

Elementary graph theory

3.1 Beginning of Graph Theory: Euler Tours

The story of graphs began with Leonhard Euler (1707-1783). Euler was a remarkable mathematician (both pure and applied) and managed to contribute to the foundation for so many subfields of mathematics (number theory and algebra, complex analysis, calculus, differential geometry, fluid mechanics, and even music theory and cartography).

One field that began with Euler was topology. He learned of the bridges-of-Königsberg problem and published his solution in an article called "The solution of a problem relating to the geometry of position.". The phrase "geometry of position" reflected his realization that the *geometry* of the problem, in the traditional sense of geometry, was irrelevant. According to Euler's geometry of *position*, structure and adjacency were important but geometric distance was not. Euler abstracted what we now call a *graph* from the map of Königsberg. He then gave necessary and sufficient conditions for a graph to admit a tour that visits every edge exactly once.

Lemma 3.1.1 (Euler's Lemma). *Any nonempty graph with no vertices of degree one has a simple cycle.*

Proof. Let $G = (V, E)$ be a nonempty graph with all degrees ≥ 2, and let d be a dart of one of the edges. Execute the following algorithm, where *select* is a function that returns one element of its (set-valued) argument.

```
1    Let $v$ be the tail of $d$
2    While the head of $d$ has not been previously assigned to $v$,
3        Let $v$ be the head of $d$.
4        Let $d :=$ select$(v - \{\text{rev}(d)\})$
```

Let v_i and d_i be the values, respectively, of the variables v and d after i iterations. The condition in step 1 guarantees that the vertices $v_0, v_1, \ldots$ are distinct, which implies that the darts $d_0, d_1, \ldots$ are distinct. In step 4, therefore, since v has at least one dart (namely $\text{rev}(d)$) and does not have degree one, it has at least one dart in addition to $\text{rev}(d)$. Thus step 4 cannot fail. Since there are a finite number of vertices, the algorithm must terminate after some number k of iterations, when the vertex containing $\text{rev}(d)$ is v_i for some $i \leq k$. At this point, the sequence $d_i, d_{i+1}, \ldots, d_k$ of darts corresponds to a simple cycle. □

Corollary 3.1.2 (Euler's Corollary)**.** *The edges of a graph can be written as a disjoint union of simple cycles iff all degrees are even.*

Proof. (*only if*) Suppose the edges of G can be written as a disjoint union $C_1 \cup \cdots \cup C_k$ of simple cycles. For each vertex v and for $i = 1, \ldots, k$, the number of edges of C_i incident to v is two if v is a vertex of C_i and zero otherwise. The degree of v is $2|\{i : v \in C_i\}|$, which is even.
(*if*) The proof is by induction on the number of edges. The corollary holds trivially for an empty graph. Suppose G is a nonempty graph with all degrees even. By Euler's Lemma, G contains a simple cycle C. For each vertex v, the number of edges of C incident to v is zero or two, and therefore even. Deleting the edges of C from G therefore yields a graph G' that still has all degrees even. By the inductive hypothesis, the edges of G' can be decomposed into simple cycles. Adding C to this decomposition yields a decomposition for G. □

Theorem 3.1.3 (Euler's Theorem)**.** *An undirected graph has a path of edges containing every edge iff the graph is connected and every node has even degree.*

The directed version of Euler's Corollary states that if every vertex of a directed graph has outdegree equal to indegree, then its arcs can be decomposed into a disjoint union of simple directed cycles.

3.2 Spanning forests and trees

An edge subgraph of G that has no undirected cycles is called a *forest* of G, and is called a *tree* of G if it is connected. A forest is a disjoint union of trees.

A forest F of G is a *spanning forest* if every pair of vertices that are connected in G are also connected in F. A spanning forest that is a tree is called a *spanning tree.*

Let F be a spanning forest of G. An edge of G is a *tree edge* (or *tree arc*) with respect to F if e belongs to F, and otherwise is a *nontree edge* (or arc).

Lemma 3.2.1. *If F is a spanning forest, $|E(F)| = |V(F)| - \kappa(F)$.*

Lemma 3.2.2. *Suppose F is a forest of G, and uv is an edge in $E(G) - E(F)$ such that u and v are not connected in F. Then $F \cup \{uv\}$ is a forest.*

Proof. Let $F' = F \cup \{uv\}$, and suppose F' has a simple cycle C. Then C must include the edge uv, for otherwise C is a cycle in F. But $C - \{uv\}$ is a path in F connecting u and v, a contradiction. □

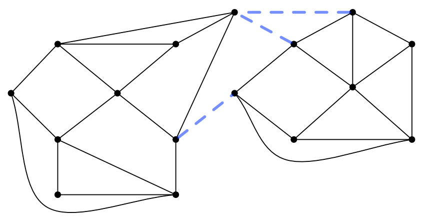


Figure 3.1: A graph is shown; the dashed edges form a cut.

We say an edge-subgraph F of G is a *spanning forest* if every pair u, v of vertices that are connected in G are also connected in F. Note that in this case $\kappa(F) = \kappa(G)$. If G is connected then a spanning forest is a tree, so we call it a *spanning tree* of G.

Corollary 3.2.3 (Matroid property of forests)**.** *For any forest F of G, there exists a set M of edges in $E(G) - F$ such that $F \cup M$ is a spanning forest of G.*

Corollary 3.2.4. *If F is a forest of G and $|E(F)| = |V(G)| - 1$ then F is a spanning tree of G.*

Proof. By Corollary 3.2.3, there exists a set M of edges in $E(G) - E(F)$ such that $F \cup M$ is a spanning forest of G. By Corollary 3.2.1,

$$\begin{aligned} |V(G)| - \kappa(G) &= |F| + |M| \\ &= |V(G)| - 1 + |M| \end{aligned}$$

so $1 - |M| = \kappa(G)$. Since $\kappa(G) \geq 1$, we can infer that $|M| = 0$ and $\kappa(G) = 1$. $\square$

3.2.1 Nontree edges and fundamental cycles

Let G be a graph, and let F be a spanning forest of G. For a dart d of an nontree edge, there is a simple head(d)-to-tail(d) path $d_1 \cdots d_k$ of darts in G whose edges belong to F. Write $d_0 = d$ so $d_0 \cdots d_k$ is a simple cycle C_e of darts, called the *fundamental cycle of d with respect to F*. For an arc e of $E(G) - F$, we define the fundamental cycle of e to be the fundamental cycle of the primary dart $(e, +1)$.

3.3 Cuts

3.3.1 (Undirected) cuts

For a graph G and a set S of vertices of G, we define $\delta_G(S)$ to be the set of edges having one endpoint in S and one endpoint not in S. We say a set of edges of G is a *cut* of G if it has the form $\delta_G(S)$.

3.3.2 (Directed) dicuts

We define $\delta^+_G(S)$ to be the set of arcs whose tails are in S and whose heads are not in S. Note that $\delta^+_G(S)$ is a subset of $\delta_G(S)$. A set of arcs of G is a *directed cut* (a.k.a. *dicut*) if it has the form $\delta^+_G(S)$.

3.3.3 Dart cuts

For a set S of vertices of G, we define $\vec{\delta}_G(S)$ to be the set of *darts* whose tails are in S and whose heads are not in S. Note that $\vec{\delta}_G(S)$ has one dart for each edge in $\delta_G(S)$. We refer to $\vec{\delta}_G(S)$ as the *dart boundary of S in G*.

3.3.4 Bonds/simple cuts

Let G be a graph, let K be a connected component of G, and let S be a subset of the vertices of K. We say a cut $\delta_G(S)$ or a dart cut $\vec{\delta}_G(S)$ is a *bond* or a *simple cut* if S is connected in G and $V(K) - S$ is connected in G.

Note that a cut in G is minimal among nonempty cuts iff it is a simple cut. Any cut can be written as a disjoint union of simple cuts.

Vertex cuts Now let S be a set of edges. We define $\partial_G(S)$ to be the set of vertices v such that at least one edge incident to v is in S and at least one edge incident to v is *not* in S. We refer to $\partial_G(S)$ as the *vertex boundary of S in G*. A vertex of $\partial_G(S)$ is a *boundary vertex* of S in G.

Notational conventions We may omit the subscript and write $\delta(S)$ or $\delta^+(S)$ when doing so introduces no ambiguity.

For a vertex v, we may write $\delta(v)$ or $\vec{\delta}(v)$ to mean $\delta(\{v\})$ or $\vec{\delta}(\{v\})$.

Question 3.3.1. *Give a graph G and a vertex v for which $\vec{\delta}_G(v)$ is not identical to the set v of darts.*

3.3.5 Tree edges and fundamental cuts

Let G be a graph, and let F be a spanning forest of G.

For a tree edge $e = u_1u_2$ (where $u_1 = \text{tail}(e)$ and $u_2 = \text{head}(e)$), let K be the connected component of F that contains e. For $i = 1, 2$, let

$$S_i = \{\text{vertices reachable from } u_i \text{ via edges of } F - e\}$$

Claim: S_1 and S_2 form a partition of the vertices of K.

Proof. Let T be the tree connecting the vertices of F. For any vertex v of K, for $i = 1, 2$, let P_i be the simple v-to-u_i path in T. If e were in neither P_1 nor P_2 then $P_1 \circ \text{rev}(P_2) \circ e$ would be a simple cycle in T, so e is in one of them, say P_1. Then the prefix of P_1 ending just before e is a simple v-to-u_2 path not using e, so v is in S_2. □

We call $\vec{\delta}_G(S_1)$ the *fundamental cut of e in G with respect to F*. Since S_1 and S_2 are connected in K, the claim implies the following.

Lemma 3.3.2 (Fundamental-Cut Lemma)**.** *For any tree edge e, the fundamental cut of e is a simple cut.*

Lemma 3.3.3. *For distinct tree edges e, e', e' is not in the fundamental cut of e.*

Problem 3.3.4. *Prove Lemma 3.3.3.*

3.3.6 Paths and Cuts

Lemma 3.3.5 (Path/Cut Lemma)**.** *Let G be a graph, and let u and v be vertices of G.*

- Dipath/Dicut *For a set A of arcs, every u-to-v dipath contains an arc of A iff there is a dicut $\delta^+(S) \subseteq A$ such that $u \in S, v \notin S$.*
- Path/Cut *For a set E of edges, every u-to-v path contains an edge of A iff there is a cut $\delta(S) \subseteq A$ such that $u \in S, v \notin S$.*
- Dart Path/Dart Cut *For a set D of darts, every u-to-v path of darts contains a dart of D iff there is a dart cut $\vec{\delta}(S) \subseteq D$ such that $u \in S, v \notin S$.*

Proof. We give the proof for the first statement; the proofs for the others are similar.

Let S be the set of vertices reachable from x via paths that avoid arcs in A.

(*only if*) Suppose every x-to-y dipath contains an arc of A. Then $x \in S, y \notin S$. Let uv be an arc in $\delta^+(S)$. then $u \in S$, so there is an x-to-u path P that avoids arcs in A. On the other hand, $v \notin S$, so every x-to-v path contains an arc in A. Consider the path $P\ uv$. It is an x-to-v path, so contains some arc in A, but P has no arcs of A, so $uv \in A$.

(*if*) Suppose there is a dicut $\delta^+(S) \subseteq A$ such that $x \in S, y \notin S$. Let P be any x-to-y path, and let v be the first vertex in P that does not belong to S (there is at least one such vertex, since $y \notin S$). Let u be the predecessor of v in P. By choice of v, we know $u \in S$. Hence $uv \in \delta^+(S)$, so $uv \in A$. □

3.3.7 Two-edge-connectivity and cut-edges

We define an equivalence relation, two-edge-connectivity, on edges. Edges e_1 and e_2 are *two-edge-connected* in G if G contains a cycle of edges containing both of them. The equivalence classes of this relation are called *two-edge-connected components.* Check definition.

An edge e of G is a *cut-edge* if the two-edge-connected component containing e contains no other edges.

Lemma 3.3.6 (Cut-Edge Lemma)**.** *An edge e of G is a cut-edge iff every path between its endpoints uses e.*

PICTURE

3.4 Vector Spaces

Dart space Let $G = (V, E)$ be a graph. The *dart space* of G is $\mathbb{R}^{E\times\{\pm 1\}}$, the set of vectors $\boldsymbol{\alpha}$ that assign a real number $\boldsymbol{\alpha}[d]$ to each dart d. For a vector $\boldsymbol{c}$ in dart space and given a set S of darts, $\boldsymbol{c}(S)$ denotes $\sum_{d\in S} \boldsymbol{c}[d]$.

Vertex space The *vertex space* of G is $\mathbb{R}^V$. A vector of vertex space is called a *vertex vector*.

Arc space and arc vectors The *arc space* of G is a vector subspace of the dart space, namely the set of vectors $\boldsymbol{\alpha}$ in the dart space that satisfy *antisymmetry*:

$$\text{for every dart } d, \boldsymbol{\alpha}[d] = -\boldsymbol{\alpha}[\text{rev}(d)] \tag{3.1}$$

A vector in arc space is called an *arc vector*. We will mostly be working with arc vectors.

$\boldsymbol{\eta}(d)$ For a dart d, define $\boldsymbol{\eta}(d)$ to be the arc vector such that $\boldsymbol{\eta}(d)[d] = 1$ and $\boldsymbol{\eta}(d)[d'] = 0$ for all darts d' such that $d' \neq d$ and $d' \neq \text{rev}(d)$.

Fact 3.4.1. *The vectors* $\{\boldsymbol{\eta}((a, +1)) \ : \ a \in E\}$ *form a basis for the arc space.*

We extend this notation to sets of darts: $\boldsymbol{\eta}(S) = \sum_{d\in S} \boldsymbol{\eta}(d)$. Formally, a vertex v is the set of darts having v as tail, so $\boldsymbol{\eta}(v) = \sum_d \boldsymbol{\eta}(d)$ where the sum is over those darts whose tails are v.

For a dart d,

$$\boldsymbol{\eta}(v)[d] = \begin{cases} 1 & \text{if only } a\text{'s tail is } v \\ -1 & \text{if only } a\text{'s head is } v \\ 0 & \text{otherwise} \end{cases}$$

A self-loop cancels itself out.

The dart-vertex incidence matrix A_G For a graph G, we denote by A_G the *dart-vertex incidence matrix*, the matrix whose columns are the vectors $\{\boldsymbol{\eta}(v) \ : \ v \in V(G)\}$. That is, A_G has a row for each dart d and a column for each vertex v, and the dv entry is 1 if v is the tail of d, -1 if v is the head of d, and zero otherwise.

3.4.1 The cut space

Maybe call it the vertex basis.

Let G be a graph. The vector space spanned by the set $\{\boldsymbol{\eta}(\vec{\delta}_G(S)) \ : \ S \subset V\}$ is called the *cut space* of G. To define a basis for this vector space, let $K_1, \ldots, K_{\kappa(G)}$ be the connected components of G, and let $v_1, \ldots, v_{\kappa(G)}$ be representative vertices chosen arbitrarily from the vertex sets of the components. Let $\text{CUT}_G = \{\boldsymbol{\eta}(\vec{\delta}_G(v)) \ : \ v \in V - \{\hat{v}_1, \ldots, \hat{v}_{\kappa(G)}\}\}$. Note that $|\text{CUT}_G| = |V| - \kappa(G)$. Clearly each vector in CUT_G belongs to the cut space. We will eventually prove that CUT_G is a basis for the cut space. (For brevity, we may omit the subscript when the choice of graph G is clear.)

Lemma 3.4.2. *The vectors in CUT are linearly independent (so span(CUT) has dimension $|CUT|$).*

Proof. Suppose $\boldsymbol{\psi} = \sum_v \psi_v \boldsymbol{\eta}(v)$ is a nonzero linear combination of vectors in CUT. We show that the sum is not the all-zeroes vector. Let H be the subgraph induced by the set of vertices v such that $\psi_v \neq 0$. Let K be a connected component of H. Since K is a proper subgraph of some connected component K' of G itself (K includes no representative vertex $\hat{v}_i$), there is some arc uv having exactly one endpoint in K. Assume without loss of generality that u belongs to K. Its other endpoint v cannot lie in another component of H, else u and v would be in the same component. Hence $\psi_v = 0$. This implies that the component of $\boldsymbol{\psi}$ corresponding to uv is nonzero. □

3.4.2 The cycle space

We turn to another vector space. We define the *cycle space* of G to be the orthogonal complement of the cut space in the arc space. That is, the cycle space is

$$\{\boldsymbol{\theta} \in \text{cut space} \;:\; \boldsymbol{\theta} \cdot \boldsymbol{\eta}(v) = 0 \text{ for all } v \in V\} \tag{3.2}$$

It follows via elementary linear algebra that the dimension of the cycle space plus the dimension of the cut space is $|E|$.

To define a basis for the cycle space, consider G as an undirected graph, and let F be a spanning forest. For each arc e in $E(G) - F$ (i.e., each nontree arc), we use C_e to denote the fundamental cycle of e with respect to F (defined in Section 3.2.1). We define $\boldsymbol{\beta}_F(e)$ to be $\boldsymbol{\eta}(C_e)$. We may omit the subscript F when it is clear which forest is intended.

We will show that the vectors in the set $\textsc{Cyc}_F = \{\boldsymbol{\beta}_F(e) \;:\; e \in E - F\}$ are independent and belong to the cycle space. Note that $|\textsc{Cyc}_F| = |E| - |F|$.

Lemma 3.4.3. *The vectors in $\textsc{Cyc}_F$ are independent, (so span($\textsc{Cyc}_F$) has dimension $|\textsc{Cyc}_F|$).*

Proof. For any $e \in E - F$, the only vector in $\textsc{Cyc}_F$ with a nonzero entry in the position corresponding to e is $\boldsymbol{\beta}_F(e)$, so this vector cannot be written as a linear combination of other vectors in $\textsc{Cyc}_F$. □

The following lemma partially explains the name of the cycle space.

Lemma 3.4.4. *If W is a closed walk then $\boldsymbol{\eta}(W)$ is in the cycle space.*

The proof is illustrated in Figure 3.2.

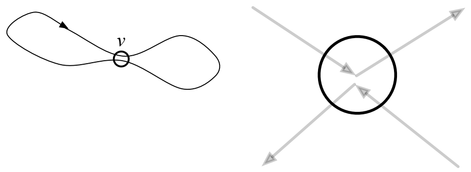


Figure 3.2: Consider a walk W and a vertex v. The darts of W that enter v are matched up with darts of W that leave v. In the vector $\boldsymbol{\eta}(v)$, darts leaving v are represented by!+1, and darts entering v are represented by -1, so $\boldsymbol{\eta}(v) \cdot \boldsymbol{\eta}(W) = 0$.

Proof. Let v be a vertex. For each dart d in W whose tail is v, v is the head of the predecessor of d in W, and for each dart d in W whose head is v, v is the tail of the successor of d. This shows that the number of darts of W whose tail is v equals the number of darts of W whose head is v, proving $\boldsymbol{\eta}(v) \cdot \boldsymbol{\eta}(W) = 0$. This shows that $\boldsymbol{\eta}(W)$ lies in the cycle space as defined in 3.2. □

Corollary 3.4.5. *The vectors in* $\textsc{Cyc}_F$ *belong to the cycle space (so span*$(\textsc{Cyc}_F)$ *has dimension at most that of the cycle space).*

Proof. Every vector $\boldsymbol{\beta}_F(e)$ in $\textsc{Cyc}_F$ is equal to $\boldsymbol{\eta}(C_e)$ where C_e is a cycle, so by Lemma 3.4.4 belongs to the cycle space. □

3.4.3 Bases for the cut space and the cycle space

Now we put the pieces together.

Corollary 3.4.6. *CUT is a basis for the cut space, and* $\textsc{Cyc}_F$ *is a basis for the cycle space.*

Proof. By Lemma 3.4.2, for some nonnegative integer j_1,

$$\begin{aligned} \dim(\text{cut space}) &= j_1 + |\text{CUT}| && (3.3)\\ &= j_1 + |V| - \kappa(G) && (3.4) \end{aligned}$$

By Corollary 3.4.5 and Lemma 3.4.3, for some nonnegative integer j_2,

$$\begin{aligned} \dim(\text{cycle space}) &= j_2 + |\textsc{Cyc}_F| && (3.5)\\ &= j_2 + |E| - |F| && (3.6) \end{aligned}$$

Since the cut space and the cut space are orthogonal,

$$\begin{aligned} |E| &= \dim(\text{arc space}) && (3.7)\\ &= \dim(\text{cut space}) + \dim(\text{cycle space}) && (3.8)\\ &= j_1 + |V| - \kappa(G) + j_2 + |E| - |F| && (3.9)\\ &= j_1 + |V| - \kappa(G) + j_2 + |E| - (|V| - \kappa(G)) && (3.10)\\ &= j_1 + j_2 + |E| && (3.11) \end{aligned}$$

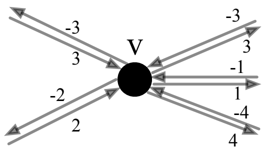


Figure 3.3: The two darts that enter v from its left carry positive amounts (3 and 2) of the commodity. The three darts that leave v to its right carry positive amounts (3, 1, and 4). The net outflow is $3 + 1 + 4 - (3 + 2) = 3$.Because of antisymmetry, a simpler way to calculate net outflow at v is to sum the values assigned to all darts leaving v: $(-3) + (-2) + 3 + 1 + 4 = 3$.

so $j_1 = 0$ and $j_2 = 0$, proving that CUT is a basis for the cut space and $\textsc{Cyc}_F$ is a basis for the cycle space. □

We call $\textsc{Cyc}_F$ the *fundamental-cycle basis with respect to F.*

3.4.4 Another basis for the cut space

Let F be a spanning forest of G. Consider the set of vectors

$$\{\boldsymbol{\eta}(\text{fundamental cut of } e \text{ with respect to } F) \ : e \in F\}$$

Clearly each vector is in the cut space. Since distinct tree edges are not in each other's fundamental cuts (Lemma 3.3.3), these vectors are linearly independent. The set consists of $|F|$ vectors. Since F is a spanning forest of G, $|F| = |V(G)| - \kappa(G)$. The set of vectors therefore has the same cardinality as $|\text{CUT}|$. It follows that this set of vectors is another basis for the cut space, and we call it the *fundamental-cut basis* with respect to F.

3.4.5 Conservation and circulations

Let $\boldsymbol{\gamma}$ be an arc vector. We can interpret $\boldsymbol{\gamma}$ as a plan for transporting amounts of some commodity (e.g. oil) along darts of the graph. If $\boldsymbol{f}[d] > 0$ for some dart d then we think of $\boldsymbol{\gamma}[d]$ units of the commodity being routed along dart d. Because $\boldsymbol{\gamma}$ satisfies antisymmetry (3.1), $\boldsymbol{\gamma}[\text{rev}(d)] < 0$ in this case.

The *(net) outflow* of $\boldsymbol{\gamma}$ at a vertex v is defined to be $\sum\{\boldsymbol{\gamma}[d] \ : \ d \in \vec{\delta}(v)\}$ This is the net amount of the commodity that leaves v (see Figure 3.3).

We say $\boldsymbol{\gamma}$ satisfies *conservation* at v if the net outflow at v is zero.

It follows from (3.2) that an arc vector that satisfies conservation at every vertex belongs to the cycle space. A vector $\boldsymbol{\theta}$ in the cycle space of G is called a *circulation* of G. We can interpret a circulation as a plan for transporting a commodity through the graph in such a way that no amount is created or consumed at any vertex. Circulations will play an essential role in our study of max-flow algorithms for planar graphs.

3.5 Embedded graphs

In solving the problem of the bridges of Königsberg, Euler discovered the power of abstracting a topological structure, a graph, from an apparently geometric structure. Perhaps it was this experience that enabled him to make another discovery. Polyhedra had been studied in ancient times but nobody seems to have noticed that every three-dimensional polyhedron without holes obeyed a simple relation:

$$n - m + \phi = 2$$

where n is the number of vertices, m is the number of edges, and ϕ is the number of faces. This equation is known as Euler's formula.[1]

This equation does not describe the geometry of a polyhedron; in fact, one can stretch and twist a polyhedron, and the formula will remain true (though the edges and faces will get distorted). We presume it was Euler's ability to think *beyond* the geometry that enabled him to realize the truth of this formula.

Planar embedded graphs are essentially the mathematical abstraction of our stretched and twisted polyhedra. Turning Euler's observation on its head, we will end up defining planar embedded graphs as those embedded graphs that satisfy Euler's formula. The traditional definition of planar embedded graphs is geometric. Our definition of embedded graphs will not involve geometry at all. Instead, we will use the notion of a *combinatorial embedding.* The advantage of this approach is that it's easier to prove things about combinatorial embeddings. For that, we need to review permutations.

Permutations For a finite set S, a *permutation* on S is a function $\pi : S \longrightarrow S$ that is one-to-one and onto. That is, the inverse of π is a function. A permutation π on S is a *cycle* (sometimes called a *cyclic permutation*) if S can be written as $\{a_0, a_1, \ldots, a_{k-1}\}$ such that $\pi(a_i) = a_{(i+1) \bmod k}$ for all $0 \leq i < k$. According to the traditional notation for a cyclic permutation, we would then write π as $(a_0\ a_1\ \ldots\ a_{k-1})$. This notation is not unique; for example, $(a_1\ a_2\ \ldots\ a_{k-1}\ a_0)$ represents the same permutation. The length of the cycle is defined to be k.

Orbits The *orbits* of a permutation π are the equivalence classes under the equivalence relation where $c \sim d$ if there exists k such that $\pi^k(c) = d$. Here the exponent indicates k-fold composition, so $c \sim d$ if one can get from c to d by some number of applications of π.

Decomposition of a permutation into cyclic permutations For any orbit of a permutation π, the restriction of π to that orbit is a cyclic permutation. It follows that any permutation can be decomposed uniquely into nonempty cyclic permutations.

[1]There is another "Euler's formula," $e^{ix} = \cos x + i \sin x$.

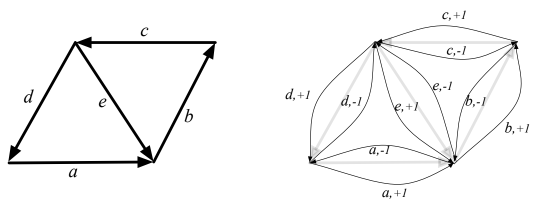


Figure 3.4: An embedded graph is illustrated. The cyclic permutation associated with the top-right vertex is $((d, +1)\ (e, +1)\ (c, -1))$, and the one associated with the bottom-left vertex is $((b, +1)\ (e, -1)\ (a, -1))$. This drawing reflects the convention that the cyclic associated with a vertex represents the *counterclockwise* arrangement of darts around the vertex. In the drawing on the right, the individual darts are shown. This drawing is in accordance with the US "rules of the road": if the two darts are interpreted as two lanes of traffic, one travels in the right lane.

3.5.1 Embeddings

The idea of a combinatorial embedding was implicit in the work of Heffter (1891). Edmonds (1960) first made the idea explicit in his Master's thesis, and Youngs (1963) formalized the idea. A combinatorial embedding is sometimes called a *rotation system.*

Here's the idea. Suppose we start with an embedding of a graph in the plane. For each vertex, walk around the vertex counterclockwise and you will encounter edges in some order, ending up in the same place you started. The embedding thus defines a cyclic permutation (called a *rotation*) of the edges incident to the vertex. There is a sort of converse: given a rotation for each vertex, there is an embedding on *some* orientable surface that is consistent with those rotations.

Embedding For a graph $G = (V, E)$, an *embedding* of G is a permutation π of the set of darts $E \times \{\pm 1\}$ whose orbits are exactly the parts of V. Thus π assigns a cyclic permutation to each part of V, i.e. each vertex. For each vertex v, we define $\pi|v$ to be the cyclic permutation that is the restriction of π to v.

Our *interpretation* is that $\pi|v$ tells us the arrangement of darts counterclockwise around v in the embedding. Such an interpretation is useful in drawings of embedded graphs, e.g. the drawing on the left of Figure 3.4.

The drawing on the right of Figure 3.4 should not be considered a drawing of an embedded graph. This drawing shows the

We use $V(\pi)$ to denote the set of orbits of π.

Embedded graph We define the pair $G = (\pi, E)$ to be an *embedded graph.* To be consistent with the definitions of unembedded graphs, we define $E((\pi, E)) =$

E and $V((\pi, E)) = V(\pi)$. We also define $\pi((\pi, E)) = \pi$. The *underlying graph* of an embedded graph G is defined to be the (unembedded) graph $(V(G), E(G))$.

MORE HERE

$\vec{\delta}_G(S)$ For an embedded graph G and a set S of vertices, $\vec{\delta}_G(S)$ is a permutation on the set of darts whose tails are in S and whose heads are not in S....

Faces To define the faces of the embedded graph, we define another permutation dual(π) of the set of darts by composing π with rev:

$$\text{dual}(\pi) = \pi \circ \text{rev}$$

Then the *faces* of the embedded graph (π, E) are defined to be the orbits of dual(π). We typically denote dual(π) by π^*.

Consider, for example, the embedded graph of Figure 3.4.

$$\begin{aligned} \pi^*[(a,-1)] &= \pi[(a,+1)] \\ &= (d,-1) \\ \pi^*[(d,-1)] &= (e,+1) \\ \pi^* * [(e,+1)] &= (a,-1) \end{aligned}$$

so one of the faces is $\{(a,-1),(d,-1),(e,+1)\}$.

Note that, in the figure, the face's cyclic permutation of darts traces out a clockwise cycle of darts such that no edges are embedded within the cycle. Consider, though, the face consisting of $\{(a,+1),(b,+1),(c,+1),(d,+1)\}$. In the drawing, the darts of this face appear to form a counterclockwise cycle, and the rest of the graph is embedded within this cycle. According to traditional nomenclature for planar embedded graphs, this face is called the *infinite face* because the edge-free region it bounds is infinite. However, imagine the same embedding on the surface of a sphere; there is no infinite face. All faces have equal status.

Since the combinatorial definition of embedded graphs and faces does not distinguish any faces, it is often convenient to imagine that it describes an embedding on a sphere (or other closed, orientable surface). However, for some purposes, it is convenient to distinguish one face, and designate it as the infinite face. Any face of the embedded graph can be chosen to be the infinite face, and this freedom is exploited in the design of some algorithms.

]bf Remark: *In the traditional, geometric definition of embedded graphs, one considers the set of points that are* not *in the image of the embedding; a face is a connected component of this set of points. This definition works for connected graphs. However, for disconnected graphs, this definition leads to a face being in a sense shared by two components of the graph. Such a face has a disconnected boundary. This is a flawed definition; we later mention a couple of bad consequences of adopting this definition.*

3.5.2 Euler characteristic and genus

Let n, m, and ϕ be the number of vertices, edges, and faces of an embedded graph. The *Euler characteristic* of the embedding is $n - m + \phi$. The *genus* of the embedding is the integer g that satisfies the formula

$$n - m + \phi = 2 - 2g$$

As we discussion in Chapter 4, an embedding is planar if its genus is 0

3.5.3 Remark on the connection to geometric embeddings

From a combinatorial embedding, one can construct a surface and an embedding of the underlying graph in that surface. For each face $(d_1 \ \dots \ d_r)$, construct an r-sided polygon and label the sides $d_1, \dots, d_r$ in clockwise order. Now we have one polygon per face. For each edge e, glue together the two polygon sides labeled with the two darts of e. The result can be shown to be a closed, orientable surface. The graph is embedded in it.

Conversely, given any embedding of a graph G onto an closed, orientable surface, define π by the rotations at the vertices. Then the embedding defined by the gluing construction is homeomorphic to the given embedding. Thus, up to homeomorphism the rotations determine the embedding. The proof of this theorem is very involved, and won't be covered here. Since for the purpose for proofs we use combinatorial embeddings rather than geometric embeddings, the theorem will not be needed.

3.5.4 The dual graph

For an embedded graph $G = (\pi, E)$, the *dual graph* (or just *dual*) is defined to be the embedded graph $\text{dual}(G) = (\text{dual}(\pi), E)$. We typically denote the dual of G as G^*.

In relation to the dual, the original graph G is called the *primal graph* (or just the *primal*). Note that the vertices of G^* are the faces of G. It follows from the following lemma that the faces of G^* are the vertices of G.

Lemma 3.5.1 (The dual of the dual is the primal.)**.** *dual*(*dual*(G)) $= G$.

Problem 3.5.2. *Prove Lemma 3.5.1.*

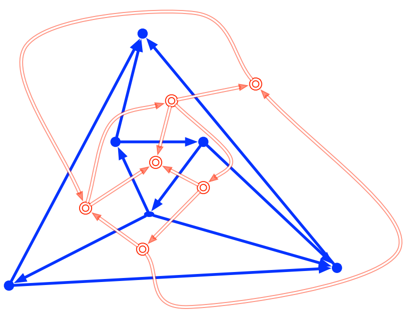


Figure 3.5: The the primal and the dual are shown together. The arcs and vertices of the primal are drawn thick and solid and blue; the arcs and vertices of the dual are drawn with thin double lines in red.

Formally, there is no need to say more about the dual. Each orbit of π^* is a subset of darts and so can be interpreted as a vertex, so (π^*, E) is an embedded graph. However, for the sake of intuition, it is often helpful to draw the dual of an embedded graph superimposed on a drawing of the primal, as shown in Figure 3.5. Each dual vertex is drawn in the middle of a face of the primal, and, for each arc a of the primal, the corresponding arc of the dual is drawn so that it crosses a at roughly a right angle.
We often adopt the convention of drawing $G = (\pi, E)$ in such a way that the counterclockwise order of darts about a vertex corresponds to their order in the corresponding permutation cycle of π. That is, for a dart d with tail v, the next counterclockwise dart with tail v is $\pi[d]$.
However, when we draw the dual superimposed on the primal as we have described, the ordering of darts in a permutation cycle corresponds in the drawing to a *clockwise* arrangement.

Is this definition necessary?

Face vectors Because the faces of G are the vertices of the dual G^*, a vertex vector of G^* is an assignment of numbers to the *faces* of G. We therefore refer to it as a *face vector* of G.

3.5.5 Connectedness properties of embedded graphs

Lemma 3.5.3. *For any face f of any embedded graph G, the darts comprising f are connected.*

Proof. Let d and d' be darts of f. $\pi^*|f = (d_0\ d_1\ \ldots\ d_{k-1})$ where $d_0 = d$. Suppose $d_i = d'$. We claim that $d_0\ d_1\ d_2\ \ldots\ d_i$ is a walk in G, which proves the lemma. For $j = 1, 2, \ldots, i$, we have $d_j = \pi^*(d_{j-1})$. By definition of π^*, $\pi(\text{rev}(d_{j-1})) = d_j$, so $\text{rev}(d_{j-1})$ and d_j have the same tail in G. Thus $\text{head}_G(d_{j-1}) = \text{tail}_G(d_j)$. $\square$

Corollary 3.5.4. *If d and d' are connected in G then they are connected in G^*.*

Proof. Let $d_0\ d_1\ d_2\ \ldots\ d_k$ be a walk in G. For $i = 1, 2, \ldots, k$, $\text{head}_G(d_{i-1}) = \text{tail}_G(d_i)$, so $\text{tail}_G(\text{rev}(d_{i-1}) = \text{tail}_G(d_i)$, so $\text{rev}(d_{i-1})$ and d_i are in the same orbit of π. Hence $\text{rev}(d_{i-1})$ and d_i belong to the same face of π^*. By Lemma 3.5.3, $\text{rev}(d_{i-1})$ and d_i are connected in G^*, so d_{i-1} and d_i are connected in G^*. □

Corollary 3.5.5 (Connectivity Corollary)**.** *For any embedded graph G, a set of darts forms a connected component of G iff the same set forms a connected component in G^*.*

3.5.6 Cut-edges and self-loops

Lemma 3.5.6. *If e is not a self-loop in an embedded graph G then e is not a cut-edge in G^*.*

Proof. Let f be a face of G^* containing one of the two darts of e. Since e is not a self-loop in G, f does not contain the other dart of e. Therefore e is not a cut-edge in G^*. □

We shall show later that the converse of Lemma 3.5.6 holds in *planar* embedded graphs. However, more generally....

Problem 3.5.7. *Show that the converse of Lemma 3.5.6 does not hold.*

3.5.7 Deletion

Deleting a dart $\hat{d}$ from a permutation π of E is obtaining the permutation π' of $E - \{\hat{d}\}$ defined as follows.

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

Deleting an edge $\hat{e}$ consists of deleting the two darts of $\hat{e}$ (in either order).

Let π' be the permutation obtained from π by deleting an edge $\hat{e}$, and let $G' = (\pi', E - \{\hat{e}\})$ be the corresponding embedded graph. It is easy to check that the orbits of π' are the same as the orbits of π except that darts of $\hat{e}$ have been removed (possibly some orbits go away). Hence the underlying graph of G' is the graph obtained by deleting $\hat{e}$ from the underlying graph of G. We write $G - \hat{e}$ for the embedded graph obtained by deleting $\hat{e}$.

3.5.8 Compression (deletion in the dual) and contraction

Lemma 3.5.8. *For an embedded graph G, if e is not a self-loop then the underlying graph of dual($G^* - e$) is the graph obtained from the underlying graph of G by contracting e.*

Proof. The proof is illustrated in Figure 3.6. Let u and v be the endpoints of e. Let $a_0, \ldots, a_k$ be the darts outgoing from u, and let $b_0, \ldots, b_\ell$ be the darts

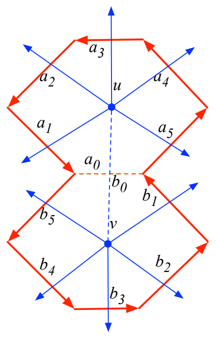


Figure 3.6: The primal graph G is shown in blue, and the dual G^* is shown in red. The edge e to be deleted from G^* has two darts, a_0 and b_0, which are in different faces of the dual G^*. When e is deleted, the two faces merge into one.

outgoing from v, where a_0 and b_0 are the darts of e. Since e is not a self-loop, a_0 does not occur among the b_i's and b_0 does not occur among the a_i's.

In G^*, u is a face with boundary $(a_0\ a_1\ \cdots\ a_k)$ and v is a face with boundary $(b_0\ b_1\ \cdots\ b_\ell)$. Let $G^{*\prime} = (\pi^{*\prime}, E - \{e\})$ be the graph obtained from G^* by deleting e.

$$\begin{aligned}\text{dual}(\pi^{*\prime})[d] &= \pi^{*\prime} \circ \text{rev}(d) \\ &= \begin{cases} \pi^*[\pi^*[\text{rev}(d)]] & \text{if } \pi^*[\text{rev}(d)] \text{ is deleted} \\ \pi[d] & \text{otherwise} \end{cases} \end{aligned} \tag{3.12}$$

For which two darts d is $\pi^*[\text{rev}(d)]$ deleted? Since $\pi^*[\text{rev}(d)] = \pi[d]$, the two darts are $\pi^{-1}[a_0]$, which is a_k, and $\pi^{-1}[b_0]$, which is b_ℓ.

Rewriting Equation 3.12, we obtain

$$\begin{aligned}\text{dual}(\pi^{*\prime})[d] &= \begin{cases} \pi^*[a_0] & \text{if } d = a_k \\ \pi^*[b_0] & \text{if } d = b_\ell \\ \pi[d] & \text{otherwise} \end{cases} \\ &= \begin{cases} b_1 & \text{if } d = a_k \\ a_1 & \text{if } d = b_\ell \\ \pi[d] & \text{otherwise} \end{cases}\end{aligned}$$

Thus $\text{dual}(\pi^{*\prime})$ has a permutation cycle $(a_1\ a_2\ \cdots\ a_k\ b_1\ b_2\ \cdots\ b_\ell)$. This permutation cycle defines the vertex obtained by merging u and v and removing the edge e. All other vertices are unchanged. This shows that the underlying graph is that obtained by contracting e. □

In view of Lemma 3.5.8, we define *compression* of an edge e in an embedded graph G to be deletion of e in the dual G^*. We denote this operation by G/e. That is, $G/e = \text{dual}(G^* - e)$. Compression of an edge of an embedded graph yields an embedded graph.

In the case when e is not a self-loop, we refer to the operation as *contraction* of e.

What about the case of compression when e is a self-loop? We discuss this later when we study planar graphs.

Chapter 4

Planar embedded graphs

4.1 Planar embeddings

We say that an embedding π of a graph $G = (V, E)$ is *planar* if it satisfies Euler's formula: $n - m + \phi = 2\kappa$, where n=number of nodes, m=number of arcs, ϕ=number of faces, and κ=number of connected components. In this case, we say (π, E) is a *planar embedded graph* or *plane graph.*

Problem 4.1.1. *Specify formally a smallest embedded graph that is not a planar embedded graph. (This is not the assume as giving the smallest graph that has no planar embedding.) You should give the embedding π and a drawing in which the darts are labeled. (You will have to find some way of drawing the embedding even though it is not planar.) Then give the dual in the same way, using a permutation and a drawing.*

The definition of planarity immediately implies the following lemma.

Lemma 4.1.2. *π is a planar embedding of G iff, for each connected component G' of G, the restriction π' of π to darts of G' is a planar embedding of G.*

Lemma 4.1.3. *The dual of a planar embedded graph is planar.*

Problem 4.1.4. *Prove Lemma 4.1.3.*

4.2 Contraction preserves planarity

Our goal for this section is to show that contracting an edge preserves planarity.

Lemma 4.2.1. *Let G be a planar embedded graph, and let e be an edge that is not a self-loop. Then G/e is planar.*

Proof. Let n, m, ϕ, κ be the number of vertices, edges, faces, and connected components of G. By planarity, $n - m + \phi = 2\kappa$. Let $G' = \text{dual}(G^* - e)$. Let

n', m', ϕ', κ' be the number of vertices, edges, faces, and connected components of G'. Clearly $m' = m - 1$. By Lemma 3.5.8, $n' = n - 1$. By Lemma 3.5.6, e is not a cut-edge in G^*. It follows from the Cut-Edge Lemma (Lemma 3.3.6) that $\kappa' = \kappa$. Therefore $n' - m' + \phi' = 2\kappa'$. □

4.3 Sparsity of planar embedded graphs

Lemma 4.3.1 (Sparsity Lemma). *For a planar embedded graph in which every face has size at least three, $m \leq 3n - 6$, where m is the number of edges and n is the number of vertices.*

Problem 4.3.2. *Prove the Sparsity Lemma, and show that the upper bound is tight by showing that, for every integer $n \geq 3$, there is an n-vertex planar embedded graph whose number of edges achieves the bound.*

Problem 4.3.3. *Prove a lemma analogous to the Sparsity Lemma in which faces of size one are permitted.*

4.3.1 Strict graphs and strict problems

A face of size two consists of two *parallel edges*, edges with the same endpoints. A face of size one consists of a self-loop. A graph with neither parallel edges nor self-loops is a *strict* graph.

Tutte, W. T. Graph Theory as I Have Known It. Oxford, England: Oxford University Press, 1998.

For many optimization problems, it is sufficient to consider strict graphs. Consider a optimization problem whose input includes a graph G and a dart vector $\boldsymbol{c}$. We say a graph optimization problem is *strict* if there is a constant-time procedure that, given an instance $\mathcal{I}$ and a pair of parallel edges or a self-loop, modifies the instance to eliminate one of the parallel edges or the self-loop, such that, given an optimal solution for the modified instance, an optimal solution for the original instance can be obtained in constant time.

Consider, for example, the problem of finding a *minimum-weight spanning tree* . A self-loop can simply be eliminated because it will never appear in any spanning tree. Given a pair of parallel edges, the one with greatest weight can be eliminated since it will not appear in the minimum-weight spanning tree. Therefore finding a minimum-weight spanning tree is a strict problem. Many other problems discussed in this book, such as shortest paths, maximum flow, the traveling salesman problem, and the Steiner tree problem, can similarly be shown to be strict. For a strict problem, we generally assume that the input graph is strict and therefore has at most three times as many edges as vertices.

We also discuss a problem, two-edge-connected spanning subgraph, that is not, strictly speaking, strict. However, by using a similar technique we can ensure that there are no *triples* of parallel edges. It follows that we can assume for this problem that there are at most six times as many edges as vertices.

4.3.2 Semi-strictness

Strictness is too strict. A weaker property, *semi-strictness*, can be more easily established and maintained. We say an embedded graph is *semi-strict* if every face has size at least three. The Sparsity Lemma applies to such graphs.

The strictness of a problem can be exploited more thoroughly to obtain an algorithm.

Theorem 4.3.4. *There is a linear-time algorithm to compute a minimum-weight spanning tree in a planar embedded graph.*

Here is a (not fully specified) algorithm for computing a minimum-weight spanning tree.

```
def MST($G$):
1 if $G$ has no edges, return $\emptyset$
2 let $\hat{e}$ be an edge of $G$ contained in some MST of $G$
3 contract $\hat{e}$
4 eliminate some parallel edges
  return $\{\hat{e}\} \cup$ MST($G$)
```

The choice of $\hat{e}$ in Line 2 is guided by the following observation.

Lemma 4.3.5. *Let G be a connected undirected graph with edge-weights, let v be a vertex of G, and let e be a minimum-weight edge incident to v. Then there is a minimum-weight spanning tree of G that contains e.*

Problem 4.3.6. *Prove Lemma 4.3.5, and then prove Theorem 4.3.4 by showing how to implement* MST *for semi-strict planar embedded graphs in such a way that each iteration takes constant time.*

4.3.3 Orientations with bounded outdegree

An *orientation* of a graph is a set $\mathcal{O}$ of darts consisting of exactly one dart of each edge. We say it is an α-orientation if each each vertex is the tail of at most α darts.

Corollary 4.3.7 (Orientation Corollary)**.** *Every semi-strict planar embedded graph has a 5-orientation.*

Problem 4.3.8. *Prove the Orientation Corollary.*

One simple application of the Orientation Corollary is maintaining a representation of a planar embedded graph to support queries of the form

"Is there an edge whose endpoints are u and v?"

Here is the representation. For each vertex u, maintain a list of u's outgoing darts. To check whether there is an edge with endpoints u and v, search in the list of u and the list of v. Since each list has at most five darts, answering the query takes constant time.

4.3.4 Maintaining a bounded-outdegree orientation for a dynamically changing graph

For an unchanging graph, the same bound can be obtained for all graphs simply by using a hash function. However, the orientation-based approach can be used even when the graph undergoes edge deletions and contractions, and we will see how this can be used in efficient implementations of other algorithms.

An orientation $\mathcal{O}$ is represented by an array adj$[\cdot]$ indexed by vertices. For vertex v, adj$[v]$ is a list consisting of the darts in $\mathcal{O}$ that are outgoing from v.

Let G be a semi-strict planar embedded graph, and let $\mathcal{O}$ be a 14-orientation of G. Suppose G' is obtained from G by deleting an edge e. Then $\mathcal{O} - \{\text{darts of } e\}$ is a 14-orientation for G', and the representation adj$[\cdot]$ can be updated as follows:

```
def DELETE($e$):
    let $v$ be the endpoint of $e$ such that adj[$v$] contains a dart $d$ of $e$
    remove $d$ from adj[$v$]
```

Suppose instead that G' is obtained from G by contracting e, and that G' remains semi-strict. The vertex resulting from coalescing the endpoints of e might have more than fourteen outgoing darts in $\mathcal{O} - \{\text{darts of } e\}$. However, a 14-orientation of G' can be found as follows:

```
def CONTRACT($e$):
    let $u$ and $v$ be the endpoints of $e$
    let $w$ be the vertex obtained by coalescing $u$ and $v$
    adj[$w$] := adj[$u$] $\cup$ adj[$v$] $-$ {darts of $e$}
    if $|$adj[$w$]$| > 14$,
        $S := \{w\}$
        while $S \neq \emptyset$,
            remove a vertex $x$ from $S$
            for each dart $xy \in$ adj[$x$],
                add $yx$ to adj[$y$]
                if $|$adj[$y$]$| > 14$, add $y$ to $S$
            adj[$x$] := $\emptyset$
```

4.3.5 Analysis of the algorithm for maintaining a bounded-outdegree orientation

The key to analyzing the algorithm is the following lemma.

Lemma 4.3.9. *For any semi-strict plane graph G, any orientation $\mathcal{O}$ of G, and any vertex v, there is a path of size at most $\lceil \log n(G) \rceil - 1$ from v to a vertex whose outdegree is at most 3.*

Proof. Let L_i denote the set of vertices reachable from v via paths of size at most i consisting of darts in $\mathcal{O}$. Assume for a contradiction that $L_{\lceil \log n(G) \rceil - 1}$ contains no vertex of outdegree at most 3. We prove that $|L_{i+1}| > 2|L_i|$ and hence that $|L_i| > 2^i$ for all $i \leq \lceil \log n(G) \rceil$. In particular, $|L_{\lceil \log n(G) \rceil}| > n(G)$, a contradiction.

Each vertex in L_i has outdegree at least 4, so the sum of outdegrees is at least $4|L_i|$. The plane graph induced by L_i has at most $3|L_i| - 6$ edges, so the heads of at least $|L_i| + 6$ of the outgoing darts are not in L_i. Therefore $|L_{i+1} - L_i| \geq |L_i| + 6$. □

We show a bound of $O((k+n)\log n)$ on the total time for maintaining a 14-orientation using Delete and Contract for k operations on an n-vertex graph.

Consider a sequence of semi-strict planar embedded graphs

$$G_0, \ldots, G_k$$

such that, for $i = 1, \ldots, k$, G_i is obtained from G_{i-1} by a deletion or a contraction. Let $n = \max_i n(G_i)$.

Lemma 4.3.10. *There exist 5-orientations $\mathcal{O}_0, \ldots, \mathcal{O}_k$ of $G_0, \ldots, G_k$ respectively, such that, for $i = 1, \ldots, k$, there are at most $\log n$ edges that whose orientations in O_i and $\mathcal{O}_{i-1}$ differ.*

Proof. We construct the sequence $\mathcal{O}_0, \ldots, \mathcal{O}_k$ backwards. Since G_k is semi-strict, it has a 5-orientation. Let $\mathcal{O}_k$ be this 5-orientation.

Suppose we have constructed a 5-orientation $\mathcal{O}_i$ of G_i for some $i \geq 1$. We show how to construct a 5-orientation $\mathcal{O}_{i-1}$ of G_{i-1}. If G_i was obtained from G_{i-1} by contraction of a non-leaf edge then $\mathcal{O}_{i-1} \cap D(G_{i-1})$ is a 5-orientation of G_{i-1} so we set $\mathcal{O}_i := \mathcal{O}_{i-1} \cap D(G_{i-1})$.

If G_i was obtained from G_{i-1} by deletion of an edge or contraction of a leaf edge then we proceed as follows. Let uv be one of the darts of the edge in G_{i-1} and not in G_i. Let $\mathcal{O}$ be the orientation $\mathcal{O}_i \cup \{uv\}$. Note that $\mathcal{O}$ might not be a 5-orientation because u might have outdegree 6. However, by Lemma 4.3.9, there is a path P of size at most $\lceil \log n \rceil - 1$ consisting of darts in $\mathcal{O}$ from u to a vertex of outdegree at most 3. Let $\mathcal{O}_{i-1}$ be the orientation obtained from $\mathcal{O}$ by replacing the darts of P with their reverses. This replacement reduces the outdegree of u by one and increases the outdegree of the end of P, so $\mathcal{O}_{i-1}$ is a 5-orientation. □

Now we can analyze the use of Delete and Contract in maintaining an 14-orientation of a changing semi-strict plane graph G. The time for a delete operation is $O(1)$. The time for a contract operation is $O(1)$ not including the time spent in the while-loop of Contract. The time spent in the while-loop is proportional to the number of changes to orientations of edges. The next theorem proves that the number of such changes is $O(m + k \log n)$, which shows that the total time is also $O(m + k \log n)$.

Theorem 4.3.11. *As G is transformed from G_0 to G_1 to $\cdots$ to G_k, the total number of changes to the orientation is $O(n + k \log n)$.*

Proof. Lemma 4.3.10 showed that there are 5-orientations $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_k$ of $G_0, G_1, \ldots, G_k$ such that each consecutive pair of orientations differ in at most $\lceil \log n \rceil$ edges. When G is one of $G_0, G_1, \ldots, G_k$, we denote by $\mathcal{O}[G]$ the corresponding 5-orientation.

We use $\tilde{\mathcal{O}}$ to denote the orientation maintained by the algorithm (and represented by adj$[\cdot]$).

For the purpose of amortized analysis, we define the potential function

$$\Phi(G, \widehat{\mathcal{O}}) = |\widehat{\mathcal{O}} - \mathcal{O}[G]|$$

We say an edge of G is *good* if $\widehat{\mathcal{O}}$ and $\mathcal{O}[G]$ agree on its orientation, and *bad* otherwise. Then Φ is the number of bad edges. The value of the potential is always nonnegative and is always at most m, the number of edges in G_0.

Consider the effect on Φ of a delete or contract operation. Since the operation is accompanied by a change in $\mathcal{O}[G]$ in the orientations of at most $\log n$ edges, the potential Φ goes up by at most $\log n$. Since there are k operations, the total increase due to these changes is at most $k \log n$.

The loop in CONTRACT also has an effect on the value of the potential. In each iteration, a vertex x with outdegree greater than 14 is removed from S and the outgoing darts of x are replaced in $\widehat{\mathcal{O}}$ with their reverses. The replacement turns good edges into bad edges and bad edges into good edges. Before the replacement, at most five of x's outgoing darts were in $\mathcal{O}[G]$ so at most five edges were good. Thus at most five good edges turn to bad, and at least $15 - 5 = 10$ bad edges turn to good. The net reduction in Φ is therefore at least $10 - 5 = 5$.

Since the initial value of Φ is at most m and the increase due to operations (not counting the loop) is at most $k \log n$, the total reduction in Φ throughout is at most $m + k \log n$. Since each iteration of the loop reduces Φ by at least 5, the number of iterations is at most $\frac{m+k\log n}{5}$.

Each iteration changes the orientations of many edges; we next analyze the total number of orientation changes. Since each iteration changes at most five edges from good to bad, the total number of edges changed from good to bad is as at most $m + k \log n$. Initially the number of bad edges is at most m, so there are at most $2m + k \log n$ changes of edges from bad to good. Thus the total number of orientation changes is $3m + 2k \log n$. $\square$

1

4.4 Cycle-space basis for planar graphs

Lemma 4.4.1. *Let G be a planar embedded graph. For each vertex v of G and each vertex f of G^* $\boldsymbol{\eta}(v) \cdot \boldsymbol{\eta}(f) = 0$.*

Proof. Let D^- be the set of darts in f having v as head. Let D^+ be the set of darts in f having v as tail. Since $\boldsymbol{\eta}(v)$ assigns 1 to darts in D^+ and -1 to darts in D^-, the dot product is $|D^+| - |D^-|$. Note that, for each $d \in D^-$, $\pi^*(d)$ belongs to D^+, and, for each $d \in D^+$, $(\pi^*)^{-1}(d)$ belongs to D^-. Hence $|D^+| = |D^-|$. This shows the dot product is zero. □

Corollary 4.4.2 (Cut-Space/Cycle-Space Duality)**.** *The cut space of G^* is the cycle space of G.*

Proof. For simplicity, assume G is connected. Let v_∞ and f_∞ be vertices of G and G^*, respectively. A basis for the cut space of G is

$$\{\boldsymbol{\eta}(v) \ : v \in V(G) - \{v_\infty\}\}$$

and a basis for the cut space of G^* is

$$\{\boldsymbol{\eta}(f) \ : f \in V(G^*) - \{f_\infty\}\}$$

By Lemma 4.4.1, the vectors in the basis for the cut space of G^* are orthogonal to the vectors in the basis for the cut space of G, so the former belong to the orthogonal complement of the cut space of G, i.e. to the cycle space of G. Moreover, the former basis has cardinality exactly one less than the number of faces in G, which equals $|E(G)| - |V(G)| + 1$, which is the dimension of the cycle space of G. This proves the corollary. □

MacLane [S. MacLane, "A combinatorial condition for planar graphs," *Fund. Math.* 28 (1937), p.22–32.] in fact formulated a criterion for planarity based on cycle-cut duality.

4.4.1 Representing a circulation in terms of face potentials

Recall from Section 3.4.5 that a vector in the cycle space of G is called a *circulation* in G. It follows from Cut-Space/Cycle-Space duality (Corollary 4.4.2) that an arc vector $\boldsymbol{\theta}$ is a circulation iff it can be written as a linear combination of basis vectors

$$\boldsymbol{\theta} = \sum\{\rho_f \boldsymbol{\eta}(f) \ : \ f \in V(G^*) - \{f_\infty\}\}$$

The sum does not include a term corresponding to f_∞. It is convenient to adopt the convention that $\rho_{f_\infty} = 0$ and include the term $\rho_{f_\infty}\boldsymbol{\rho}(f_\infty)$ in the sum. We can then represent the coefficients by a face vector $\boldsymbol{\rho}$, and so write

$$\boldsymbol{\theta} = \sum\{\boldsymbol{\rho}[f]\boldsymbol{\eta}(f) \ : \ f \in V(G^*)\} \tag{4.1}$$

Even if $\boldsymbol{\rho}[f_\infty] \neq 0$, since $\boldsymbol{\eta}(f_\infty)$ is a circulation, the sum 4.1 is a circulation. In this context, we refer to the coefficients $\boldsymbol{\rho}[f]$ as *face potentials.*

We can write the relation between a circulation and face potentials more concisely using the dart-vertex incidence matrix A_{G^*} of the dual G^*. (We could call this the *dart-face incidence matrix* of G.)

$$\boldsymbol{\theta} = A_{G^*}\boldsymbol{\rho}$$

4.5 Interdigitating trees

Lemma 4.5.1. *Suppose G is a connected plane graph with a spanning tree T. Every cycle in G^* has an edge in T.*

Proof. Let C^* be a cycle in G^*. Since $\boldsymbol{\eta}(C^*)$ is in the cycle space of G^*, it is in the cut space of G by Cut-Space/Cycle-Space duality, so it can be written in terms of the fundamental cut basis of G with respect to T:

$$\boldsymbol{\eta}(C^*) = \sum_{e \in T} \alpha_e \boldsymbol{\eta}(\text{fundamental cut of } e)$$

Since the left-hand side is nonzero, there is at least one edge $\hat{e} \in T$ such that $\alpha_{\hat{e}} \neq 0$. Since different edges of T are not in each other's fundamental cuts, it follows that the sum assigns a nonzero value to a dart of $\hat{e}$. This proves the lemma. □

Corollary 4.5.2. *Let G be a plane graph. If T is a spanning tree of G then the edges $E(G) - E(T)$ form a spanning tree of G^*.*

Problem 4.5.3. *Prove Corollary 4.5.2.*

Maybe this goes in solution section at end of book!

Should have figure of two interdigitating hands.

If T is a spanning tree of a plane graph G_π, we use T^* to denote the spanning tree of G^* whose edges are $E(G) - E(T)$. We refer to T^* as the dual spanning tree with respect to T in G_π. The trees T and T^* are called *interdigitating trees.*

Interdigitating trees combined with rootward computations give rise to simple algorithms for some problems in planar graphs, as illustrated in the following problems. Beware, however, that the choice of the root of T^* might be significant.

Problem 4.5.4. *Using rootward computation (Section 1.1) on the dual tree, give a simple linear-time algorithm for the following problem.*

- input: *a planar embedded graph G, a spanning tree T, and a vertex r*
- output: *a table that, for each nontree edge uv of G, gives the least common ancestor of u and v in T rooted at r*

Problem 4.5.5. *Using the result of Problem 4.5.4, give a simple linear-time algorithm for the following problem.*

- input: *a planar embedded graph G with edge-weights and a spanning tree T*
- output: *a table that, for each nontree edge e of G, gives the total weight of the fundamental cycle of e with respect to T.*

Problem 4.5.6. *Show that a connected planar graph G with edge-weights can be represented so as to support the following operations in $O(\log n)$ amortized time:*

- *Given an edge e of G, determine whether e is in a minimum-weight spanning tree of G.*
- *Given an edge e of G and a number λ, set the weight of e to λ.*

4.6 Simple-cut/simple-cycle duality

Lemma 4.6.1 (Fundamental-Cut/Fundamental-Cycle Duality). *Let G be a connected planar embedded graph with spanning tree T and let $\hat{e}$ be an edge of T. Then*

$$\begin{aligned}&\{\textit{darts of the fundamental cut of } \hat{e} \textit{ in } G \textit{ with respect to } T\}\\ &\quad = \{\textit{darts of the fundamental cycle of } \hat{e} \textit{ in } G^* \textit{ with respect to } T^*\}\end{aligned}$$

Proof. Let C^* be the fundamental cycle of $\hat{e}$ in G^* with respect to T^*. As in the proof of Lemma 4.5.1, we write $\boldsymbol{\eta}(C^*)$ in terms of the fundamental cut basis of G with respect to T:

$$\boldsymbol{\eta}(C^*) = \sum_{e \in T} \alpha_e \boldsymbol{\eta}(\text{fundamental cut of } e)$$

Since different edges are not in each other's fundamental cuts, for each edge $e \in T$, if $\alpha_e \neq 0$ then $\boldsymbol{\eta}(C^*)$ assigns nonzero values to the darts of e. However, the only edge of T with darts in C^* is $\hat{e}$, so the sum in the right-hand side is just $\alpha_{\hat{e}}\boldsymbol{\eta}(\text{fundamental cut of } \hat{e})$. Furthermore, since the primary dart of $\hat{e}$ is assigned 1 by both $\boldsymbol{\eta}(C^*)$ and $\boldsymbol{\eta}(\text{fundamental cut of } \hat{e})$, we conclude that $\alpha_{\hat{e}} = 1$. Thus $\boldsymbol{\eta}(C^*) = \boldsymbol{\eta}(\text{fundamental cut of } \hat{e})$, which proves the lemma. □

Theorem 4.6.2 (Simple-Cycle/Simple-Cut Theorem). *Let G be a planar embedded graph. A nonempty set of darts forms a simple cycle in G^* iff the set forms a simple cut in G.*

Proof. We prove the theorem for the case in which G is connected. The result immediately follows for disconnected graphs as well.

(*only if*) Let C^* be a simple cycle in G^*. Let $\hat{e}$ be an edge of C^*, and let P^* be the simple path in G^* such that $C^* = P^* \circ \hat{e}$. By the matroid property of forests (Corollary 3.2.3), there exists a spanning tree T^* of G^* containing the edges of P^*. Note that C^* is the fundamental cycle of $\hat{e}$ with respect to T^*. Therefore, by Fundamental-Cut/Fundamental-Cycle Duality (Lemma 4.6.1), the darts forming C^* are the darts forming a fundamental cut in G, and such a cut is a simple cut by the Fundamental-Cut Lemma (Lemma 3.3.2).

(*if*) Let S_1 be a set of vertices of G such that $\vec{\delta}_G(S_1)$ is a simple cut. Let $S_2 = V(G) - S_1$. By definition of simple cut in a connected graph, for $i = 1, 2$, the vertices of S_i are connected; let T_i be a tree connecting exactly the vertices of S_i. Let d be a primary dart such that d is in $\vec{\delta}(S_1)$ or $\vec{\delta}(S_2)$, and let e be the edge of d. Let $T = T_1 \cup T_2 \cup \{e\}$. Then $\vec{\delta}(S_1)$ or $\vec{\delta}(S_2)$ is a fundamental cut with respect to T, and so by Fundamental-Cut/Fundamental-Cycle Duality (Lemma 4.6.1), its darts form a simple cycle in G^*. □

4.6.1 Compressing self-loops

The Simple-Cycle/Simple-Cut Theorem immediately yields the following.

Corollary 4.6.3. *If e is a self-loop in a planar embedded graph G then e is a cut-edge in G^*.*

We use the corollary to help analyze the effect of compressing a self-loop in a planar graph.

Lemma 4.6.4. *If G is a planar embedded graph and e is a self-loop then G/e is planar.*

Proof. Let n, m, ϕ, κ be the number of vertices, edges, faces, and connected components of G. By planarity, $n - m + \phi - 2\kappa = 0$. Let n', m', ϕ', κ' be the numbers for $G' = \text{dual}(G^* - e)$. In order to prove that G' is planar, it suffices to show that $n' - m' + \phi' - 2\kappa' = n - m + \phi - 2\kappa$.

Clearly $m' = m - 1$. By Corollary 4.6.3, e is a cut-edge in G^*.

First suppose each endpoint of e in G^* has degree greater than one. In this case, deletion of e does not cause the elimination of its endpoints in G^*. Therefore $\phi' = \phi$. Since e is a cut-edge, deleting it increases the number of connected components, so $\kappa' = \kappa + 1$ (using the Connectivity Corollary, which is Corollary 3.5.5). Let v be the common endpoint of the self-loop e in G, and let the corresponding permutation cycle be $(d_0\ d_1\ \cdots\ d_k\ \cdots\ d_\ell)$, where d_0 and d_k are the darts corresponding to e. In G^*, v is a face. Deletion of e in G^* breaks the face up into $(d_0\ d_1\ \cdots\ d_{k-1})$ and $(d_{k+1}\ \cdots\ d_\ell)$ and leaves all other faces alone. Since faces of G^* are vertices of G, we infer $n' = n + 1$. Thus

$$n' - m' + \phi' - 2\kappa' = (n+1) - (m-1) - \phi - 2(\kappa+1) = n - m - \phi - 2\kappa$$

If exactly one of the endpoints of e in G^* has degree one, that endpoint will disappear when e is deleted, so $\phi' = \phi - 1$, and there is no change to the number of connected components. In this case,

$$n' - m' + \phi' - 2\kappa' = n - (m-1) + (\phi - 1) - 2\kappa = n - m + \phi - 2\kappa$$

If both endpoints of e in G^* have degree one, deleting e eliminates both endpoints (vertices of G^*), a connected component, and a face of G^*, so $\phi' = \phi - 2$, $\kappa' = \kappa - 1$, and $n' = n - 1$.

$$n' - m' + \phi' - 2\kappa' = (n-1) - (m-1) + (\phi - 2) - 2(\kappa - 1) = n - m + \phi - 2\kappa$$

□

4.6.2 Compression and deletion preserve planarity

Combining Lemma 4.6.4 with Lemma 4.2.1 shows that compression preserves planarity. Since compression in the dual is deletion in the primal, it follows that deletion preserves planarity. We state these results as follows

Theorem 4.6.5. *For a planar embedded graph G and an edge e, $G-e$ and G/e are planar embedded graphs.*

Figure 4.1 shows some examples of compressing edges.

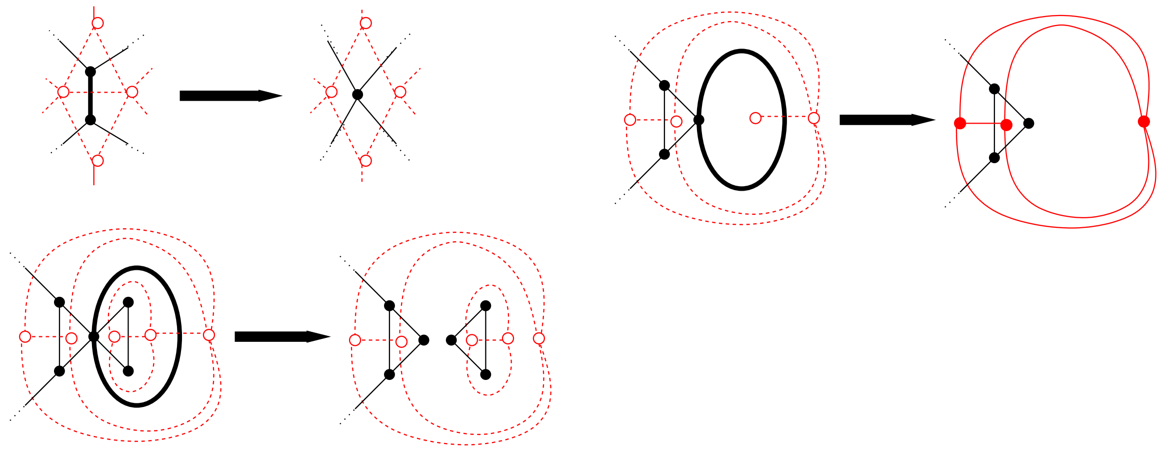


Figure 4.1: Examples of compressing an edge $\hat{e}$ in G (solid lines and filled vertices), i.e. deleting $\hat{e}$ from G^* (dashed lines and open vertices).

Compressing a self-loop in a planar embedded graph is an interesting operation. The graph can be divided into two parts, the part enclosed by the self-loop and the part not enclosed. These parts have only one vertex in common, namely the endpoint of the self-loop. Compression has the effect of duplicating the common vertex, and attaching each part to its own copy.

4.7 Faces, edges, and vertices enclosed by a simple cycle

Let C be a simple cycle of darts in a connected plane graph G_π. Let f_∞ be an arbitrary face, designated the *infinite face*. We say the cycle C *encloses* a face f with respect to f_∞ if $E(C) = \delta(S)$ where $f \in S, f_\infty \notin S$.

Using the Path/Cut Lemma, we immediately obtain

Proposition 4.7.1. *Let G_π be a connected plane graph, and let C be a cycle of G_π. Every path in G^* from a face enclosed by C to a face not enclosed by C goes through an edge of C.*

We say C encloses an edge or vertex if C encloses a face whose boundary contains the edge or vertex, and *strictly* encloses the edge or vertex if in addition the edge or vertex is not on C.

Using Proposition 4.7.1 and Corollary 62.of Lecture 2, we obtain

Proposition 4.7.2. *Let G_π be a connected plane graph, and let C be a cycle of G_π. Every path in G from a vertex enclosed by C to a face not enclosed by C goes through a vertex of C.*

The *interior* of a cycle C is the subgraph consisting of edges and vertices enclosed by C. The *exterior* is the subgraph consisting of edges and vertices not strictly enclosed by C. The *strict* interior and exterior are similarly defined.

4.8 Crossing

Two kinds of crossing: sets that cross (violate laminarity) and paths/cycles that cross in a graph sense.

4.8.1 Crossing paths

4.8.2 Non-self-crossing paths and cycles

4.9 Representing embedded graphs in implementations

It makes sense to base our computer representation of embedded graphs on the mathematical representation. We will even use this representation when we don't care about the embedding.

For the purpose of specifying algorithms, our finite set E will consist of positive integers. For example, if $|E| = m$ then we can use the integers $1 \dots m$. We also need a way to represent darts, remembering that each element of E corresponds to two darts. We use some convention to represent each dart as an integer. (Two ways: use +/- or use a low-order bit). A permutation π of darts is represented by a pair of arrays, one for the forward direction and one for the backward direction. That way, it takes $O(1)$ time to go from a dart d to the darts $\pi[d]$ and $\pi^{-1}[d]$.

Must first discuss deleting an arc from an embedded graph.

We also have to discuss the implementation of arc deletion. It will be necessary to delete arcs in constant time. The key is to allow some integers to become unused.

Deletion of an arc consists of deletion of its two darts from the representation of the permutation π.

Chapter 5

Separators in planar graphs

5.1 Triangulation

We say a planar embedded graph is *triangulated* if each face's boundary has at most three edges.

Problem 5.1.1. *Provide a linear-time algorithm that, given a planar embedded graph G, adds a set of artificial edges to obtain a triangulated planar embedded graph. Show that the number of artificial edges is at most twice the number of original edges.*

5.2 Weights and balance

Let G be a planar embedded graph, and let α be a number between 0 and 1. An assignment of nonnegative weights to the faces, vertices, and edges of G is an *α-proper* assignment if no element is assigned more than α times the total weight. A subpartition of these elements is *α-balanced* if, for each part, the sum of the weights of the elements of that part is at most α times the total weight.

5.3 Fundamental-cycle separators

Let G be a plane graph. A simple cycle C of G defines a subpartition consisting of two parts, the strict interior and the strict exterior of the cycle. If the subpartition is α-balanced, we say that C is an *α-balanced cycle separator.* The subgraph induced by the (nonstrict) interior, i.e. including C, is one *piece* with respect to C, and the subgraph induced by the (nonstrict) exterior is the other piece.

First we give a result on fundamental-cycle separators that are balanced with respect to an assignment of weights only to *faces*.

Lemma 5.3.1 (Fundamental-cycle separator of faces)**.** *For any plane graph G, $\frac{1}{4}$-proper assignment of weights to faces, and spanning tree T of G such that the* *boundary of each face of G has at most three nontree edges, there is a nontree edge $\hat{e}$ such that the fundamental cycle of $\hat{e}$ with respect to T is a $\frac{3}{4}$-balanced cycle separator for G.*

Figure

Proof. Let T^* be the interdigitating tree, the spanning tree of G^* consisting of edges not in T. The vertices of T^* are faces of G, and are therefore assigned weights. The property of T ensures that T^* has degree at most three. Using Lemma 1.3.1, let $\hat{e}$ be an edge separator in T^* of vertex weight. By the Fundamental-Cut Lemma (Lemma 3.3.2), the fundamental cut of $\hat{e}$ in G with respect to T^* is a simple cut in G^* (each side of which has weight at most three-fourths of the total) so is, by the Simple-Cycle/Simple-Cut Theorem (Theorem 4.6.2), a simple cycle in G that encloses between one-fourth and three-fourths of the weight. □

Assignments of weight to faces, vertices, and edges can also be handled:

Lemma 5.3.2. *There is a linear-time algorithm that, given a triangulated plane graph G, a spanning tree T of G, and a $\frac{1}{4}$-proper assignment of weights to faces, edges, and vertices, returns a nontree edge $\hat{e}$ such that the fundamental cycle of $\hat{e}$ with respect to T is a $\frac{3}{4}$-balanced cycle separator for G.*

Problem 5.3.3. *Prove Lemma 5.3.2 using Lemma 5.3.1*

We can give a better balance guarantee if we are separating just edge-weight.

Lemma 5.3.4 (Fundamental-cycle separator of edges)**.** *There is a linear-time algorithm that, given a triangulated plane graph G with a $\frac{1}{3}$-proper assignment of weights to edges, and a spanning tree T, returns a nontree edge $\hat{e}$ such that the fundamental cycle of $\hat{e}$ with respect to T is a $\frac{2}{3}$-balanced cycle separator for G.*

Problem 5.3.5. *Prove Lemma 5.3.4 by following the proof of Lemma 5.3.1 but using tree edge separators of vertex/edge weight (Lemma 1.3.3) instead of tree edge separators of vertex weight (Lemma 1.3.2).*

5.4 Breadth-first search

Let G be a connected, undirected graph, and let r be a vertex. For $i = 0, 1, 2, \ldots$, we say a vertex v of G has *level i with respect to r* and we define level$(v) = i$ if i is the minimum number of edges on an r-to-v path in G. (That is, the level of a vertex v is the distance of v from r where the edges are assigned unit length.) For $i = 0, 1, 2, \ldots$, let $L_i(G, r)$ denote the set of vertices having level i.

An edge uv is said to have level i (and we write level$(uv) = i$) if u has level i and v has level $i + 1$. (An edge whose endpoints have the same level is not assigned a level.)

Breadth-first search from r is a linear-time algorithm that finds

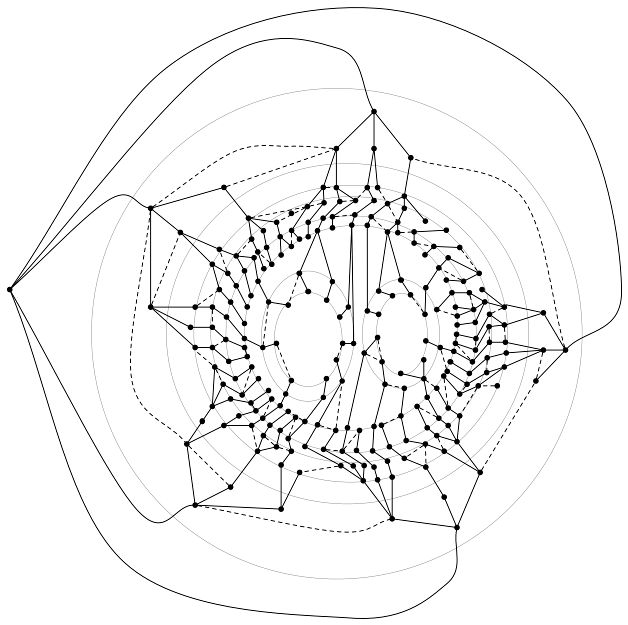


Figure 5.1: Shows the levels of breadth-first search.

- the levels of vertices and edges, and
- a spanning tree rooted at r such that, for each vertex v other than r, the parent of v has level one less than that of v (a *breadth-first-search tree*).

Need to elaborate on caption for Figure 5.1. Use copies of the figure to show the forest.

5.5 $O(\sqrt{n})$-vertex separator

We use fundamental-cycle separators to prove a fundamental separator result for planar graphs. [2]

Theorem 5.5.1 (Planar-Separator Theorem with Edge-Weights)**.** *There is a linear-time algorithm that, for a plane graph G and $\frac{1}{3}$-proper assignment of weights to edges, returns subgraphs G_1, G_2 such that*

- *$E(G_1), E(G_2)$ is a partition of $E(G)$,*
- *The partition $E(G_1), E(G_2)$ is $\frac{2}{3}$-balanced, and*
- *$|V(G_1) \cap V(G_2)| \leq 4\sqrt{|V(G)|}$*

The subgraphs G_1, G_2 are called the *pieces*. The set $V(G_1) \cap V(G_2)$ of vertices common to the two subgraphs is called the *vertex separator*.

The Planar-Separator Theorem can be used with an assignment of weight to vertices instead of edges. In this case, in evaluating the resulting balance, we do not count the weight of the vertices in the vertex separator.

Theorem 5.5.2 (Planar-Separator Theorem with Vertex-Weights)**.** *There is a linear-time algorithm that, for a plane graph G and $\frac{1}{3}$-proper assignment of weights to vertices, returns subgraphs G_1, G_2 such that*

- *$E(G_1), E(G_2)$ is a partition of $E(G)$,*
- *The subpartition $V(G_1) - V(G_2), V(G_2) - V(G_1)$ of $V(G)$ is $\frac{2}{3}$-balanced, and*
- *$|V(G_1) \cap V(G_2)| \leq 4\sqrt{|V(G)|}$*

Problem 5.5.3. *Show that the Vertex-Weights version (Theorem 5.5.2) follows easily from the Edge-Weights version (Theorem 5.5.1).*

3

Now we give the proof of the Edge-Weights version. Let $w(\cdot)$ denote the edge-weight assignment. Assume for notational simplicity that the sum of edge-weights is 1. The algorithm first performs a breadth-first search of G from r. Let T be the breadth-first-search tree. Let V_i denote the set of vertices at level i.

5.5.1 Finding the middle level

Next, the algorithm finds the integer i_0 such that

- $w(\{uv : u \text{ and } v \text{ both have level } < i_0\}) \leq 1/2$, and
- $w(\{uv : u \text{ and } v \text{ both have level } > i_0\}) \leq 1/2$.

5.5.2 Finding small levels

Next, the algorithm finds the greatest level $i_- \leq i_0$ such that $|V_{i_-}| \leq \sqrt{n}$, and the least level $i_+ \geq i_0$ such that $|V_{i_+}| \leq \sqrt{n}$. Since $|V_0| = 1$ and $|V_n| = 0$, the levels i_- and i_+ exist.

Since each of levels $i_- + 1, i_- + 2, \ldots, i_- + (i_0 - i_- - 1)$ has greater than $\sqrt{n}$ vertices and each of levels $i_0, i_0 + 1, \ldots, i_0 + (i_+ - i_0 - 1)$ has greater than $\sqrt{n}$ vertices, the total number of vertices among these levels is greater than $(i_+ - i_- - 1)\sqrt{n}$, so $i_+ - i_- - 1 < n/\sqrt{n} = \sqrt{n}$.

5.5.3 Extracting a middle graph and low-depth spanning tree

Next, the algorithm obtains a graph G' from G as follows:

1. Delete every vertex whose level is greater than i_+.
2. For each edge uv of T such that the levels of u and v are at most i_-, contract uv.

Note that G' is a planar embedded graph.

Let T' be the set of edges of T that remain in G'. The result of Line 2 is that all vertices having level at most i_- are coalesced into a single vertex r'. For any vertex v in G', therefore, there is a v-to-r' path through T' that consists of at most $i_+ - i_-$ edges.

5.5.4 Separating the middle graph

The algorithm adds artificial zero-weight edges to G' to triangulate it, and, following Lemma 5.3.4, finds a $\frac{2}{3}$-balanced fundamental-cycle separator C with respect to T'.

Let E_1 be the set of edges whose endpoints have level at most i_-, let E_2 be the set of edges having at least one endpoint at level greater than i_+, let E_3 be the set of edges of G' assigned to the interior of C, and let E_4 be the set of edges of G' assigned to the exterior.

By choice of i_0, for $j = 1, 2$ we have $w(E_j) \leq 1/2$. By the properties of the separator C, for $j = 3, 4$ we have $|E_j| \leq \frac{2}{3} w(E(G')) \leq \frac{2}{3}$.

For $j = 1, 2, 3$, or 4, if $w(E_j) \geq \frac{1}{3}$ then the algorithm sets $G_1 := E_j$ and $G_2 :=$ all other edges of G. Since $\frac{1}{3} \leq w(E_j) \leq \frac{2}{3}$, it follows that $w(E(G_1)) \leq \frac{2}{3}$ and $w(E(G_2)) \leq \frac{2}{3}$.

Assume therefore that $w(E_j) < \frac{1}{3}$ for $j = 1, 2, 3, 4$. Let j be the minimum integer such that $w(E_1) + \cdots + w(E_j) \geq \frac{1}{3}$. Then $w(E_1) + \cdots + w(E_{j-1}) < \frac{1}{3}$ and $w(E_j) < \frac{1}{3}$ so $w(E_1) + \cdots + w(E_j) < \frac{2}{3}$. The algorithm sets $G_1 := E_1 \cup \cdots \cup E_j$ and $G_2 :=$ all other edges of G. Then $w(E(G_1)) \leq \frac{2}{3}$ and $w(G_2) \leq \frac{2}{3}$.

5.5.5 Analysis

Show that the separator really separates.

We have completed the description of the algorithm, and have ensured the balance property of the separator. It is easy to verify that E_1, E_2, E_3, and E_4 are disjoint, which implies that G_1 and G_2 are edge-disjoint.

In the uncontracted graph G, the vertices of the separator C comprise two leafward tree paths P_1, P_2 originating at vertices at level i_- and terminating at two vertices whose levels are in the range $[i_-, i_+]$, possibly joined by an edge of G.

The only vertices that are endpoints of edges in distinct sets E_i and E_j are the vertices at level i_-, the vertices at level i_+, and the vertices comprising P_1 and P_2. There are at most $\sqrt{n}$ in each of the first two categories. The number in the third category that are not in the other two is at most $2(i_+ - i_- - 1)$, which in turn is at most $2\sqrt{n}$. This completes the proof of the theorem.

Problem 5.5.4. *Show how to modify the algorithm so that the size of the separator is at most $c\sqrt{n}$ for some constant $c < 4$.* Hint: *The size criterion used to decide whether a level i qualifies to be designated level i_- can depend on $|i_0 - i_-|$.*

Give problem on fast algorithm for finding separators.

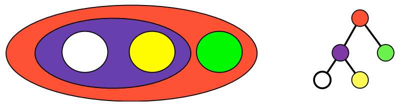


Figure 5.2: The diagram on the left shows a Venn diagram of some noncrossing sets, and the diagram on the right shows the corresponding rooted forest (a tree in this case).

5.6 Biconnectivity

Define biconnecti prove this result. haps describe the tree.

Lemma 5.6.1. *If a planar embedded graph G is biconnected then so is the planar dual G^*.*

Lemma 5.6.2. *If G is a planar embedded biconnected graph then every face is a simple cycle.*

5.7 Noncrossing families of subsets

Two nonempty sets A and B *cross* if they are neither disjoint nor nested, i.e. if $A \cap B \neq \emptyset$ and $A \not\subseteq B$ and $B \not\subseteq A$.

A family $\mathcal{C}$ of nonempty sets is *noncrossing* (also called *laminar*) if no two sets in $\mathcal{C}$ cross.

As illustarted in Figure 5.2, under the subset relation, a noncrossing family $\mathcal{C}$ of sets forms a rooted forest $F_{\mathcal{C}}$. That is, each set $X \in \mathcal{C}$ is a node, and its ancestors are the sets in $\mathcal{C}$ that include X as a subset. To see that this is a forest, let X_1 and X_2 be two supersets of X in $\mathcal{C}$. Since X is nonempty, X_1 and X_2 intersect, so one must include the other. This shows that the supersets of X are totally ordered by inclusion. Hence if X has any proper supersets, it has a unique minimal proper superset, which we take to be the parent of X.

5.8 The connected-components tree with respect to breadth-first search

For a connected graph G, a vertex r, and an integer $i \geq 0$, let $L_i^+(G, r)$ denote the set of vertices v having level at least i, and let $\mathcal{K}_i^+(G, r)$ denote the set of connected components of the subgraph of G induced by $L_i^+(G, r)$. We refer to a connected component $K \in \mathcal{K}_i^+(G, r)$ as a *level-i BFS component*, and we define $\text{level}(K) = i$.

Lemma 5.8.1. *For any connected graph G and vertex r, the BFS components $\bigcup_i \mathcal{K}_i^+(G, r)$ form a noncrossing family of subsets of $V(G)$.*

Proof. Let K_1 and K_2 be two BFS components. Assume without loss of generality that $\text{level}(K_1) \leq \text{level}(K_2)$. If any vertex of K_2 belongs to K_1 then *every* vertex of K_2 belongs to K_1. □

It follows from Lemma 5.8.1 that the vertex sets of BFS components of G form a rooted forest with respect to the subset relation.

Lemma 5.8.2. *The parent of a level-i BFS component is a level-$i-1$ BFS component if $i > 0$.*

Proof. Let K be a level-i BFS component where $i > 0$. Since the level-i BFS components are disjoint, K is not contained in any other level-i BFS component. Since K must contain some level-i vertex v of G, K is not contained in any level-j BFS component where $j > i$. Let u be the parent of v in the BFS tree. Then the level-$i-1$ BFS component K' containing u must contain K. Furthermore, for $j \leq i-1$, any level-j BFS component that contains K must also contain K'. This shows that K' is the uniqe minimal proper superset of K among the BFS components. $\square$

Lemma 5.8.2 shows that a root of the forest of BFS components must have level 0. Since we assume G is connected, there is only one level-0 BFS component, namely the whole graph G, so it is the only root, and the forest is in fact a tree. We call it the *BFS component tree*, and we denote it by $\mathcal{T}(G, r)$

Lemma 5.8.3. *The cuts $\{\delta_G(K) \ : \ K \in \mathcal{K}_i^+(G, r), i \geq 0\}$ are edge-disjoint.*

Proof. Suppose K is a level-i BFS component. Let uv be an edge of $\delta_G(K)$ where $u \in V(K)$. Then the level of v is less than i (else v would belong to K). Breadth-first search ensures that the levels of the endpoints of an edge differ by at most one, so level$(v) = i - 1$. Therefore, for any BFS component K' such that $uv \in \delta_G(K')$, we must have level$(K') = i$ and K' must contain u, hence $K' = K$. $\square$

Lemma 5.8.4. *For any BFS component $K \in \mathcal{K}_i^+(G, r)$, $\delta_G(K)$ is a simple cut.*

Proof. Clearly K is connected in G. We need to show that $V(G) - K$ is also connected in G. We show that, for every vertex $v \in V(G) - K$ other than r, there is an adjacent vertex u whose breadth-first-search level is less than that of v. An induction by breadth-first-search level then shows that every vertex in $V(G) - K$ is connected to r, and hence that $V(G) - K$ is connected.

Let T be a breadth-first-search tree of G rooted at r. For any vertex $v \in V(G) - K$ other than r, let u be the parent of v in T. By definition of the breadth-first-search tree, the level of u is less than that of v.

If u belongs to K then the level of u is at least i, so the level of v would also be at least i. Since v and u are adjacent, v would belong to K as well, a contradiction. Therefore u does not belong to K. $\square$

Lemma 5.8.5 (BFS-Component-Tree Construction)**.** *There is a linear-time algorithm to construct the BFS component tree $\mathcal{T}(G, r)$.*

Problem 5.8.6. *Prove the BFS-Component-Tree Construction Lemma.*

5.9 Cycle separators

Theorem 5.9.1 (Planar-Cycle-Separator Theorem)**.** *There is a linear-time algorithm that, for any simple undirected biconnected triangulated plane graph and any $\frac{3}{4}$-proper assignment of weights to faces, edges, and vertices, returns a $\frac{3}{4}$-balanced cycle separator C of size at most $\sqrt{8m}$.*

4

Assume for notational simplicity that the total weight is 1 and that weight is assigned only to faces. The algorithm first designates some face as f_∞ and performs breadth-first search from f_∞ in the planar dual G^*.

5.9.1 Finding the middle node

The algorithm constructs the BFS Component tree $\mathcal{T}(G^* f_\infty)$. Using a rootward computation, the algorithm computes

$$\hat{w}(K) := \sum\{w(f) \, : \, f \in V(K)\}$$

for each BFS component. The algorithm then applies the Leafmost-Heavy-Vertex Lemma with $\alpha = 1/2$ to select a BFS component K_0. Let $i_0 = \text{level}(K_0)$.

5.9.2 Finding small levels

The next step is similar to a step in the algorithm of the Planar-Separator Theorem. The algorithm will select two levels that are small. However, for simplicity we count edges instead of vertices.

Let E_i denote the set of edges at level i. Next the algorithm finds the greatest level $i_- \leq i_0$ such that $|E_{i_-}| \leq \sqrt{m/2}$, and the least level $i_+ \geq i_0$ such that $|E_{i+}| \leq \sqrt{m/2}$. Since $|E_0| = 0$ and $|E_n| = 0$, the levels i_- and i_+ exist.

Lemma 5.9.2. $i_+ - i_- - 1 \leq \sqrt{2m}$

Proof. Since each of levels $i_-+1, i_-+2, \ldots, i_+-1$ has greater than $\sqrt{m/2}$ edges, the total number of edges among these levels is greater than $(i_+ - i_- - 1)\sqrt{m/2}$, so $i_+ - i_- - 1 < m/\sqrt{m/2}$. □

5.9.3 Extracting a middle graph and a low-depth spanning tree

We assume in the following that $i_- > 0$. (The case $i_- = 0$ is similar but simpler.) Let $\hat{K}$ be the level-i_- BFS component that includes K_0. Lemma 5.8.4 shows that the set of vertices not in $\hat{K}$ is connected. The algorithm performs contractions in G^* to merge these vertices into a single vertex x_0 and assigns it a weight equal to the sum of the weights of the vertices it replaced.

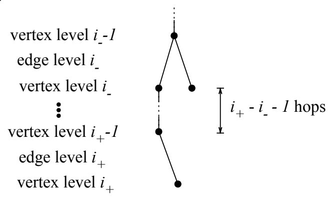
Let $K_1, \ldots, K_p$ be the level-i_+ BFS components contained in $\hat{K}$. The algorithm similarly merges the vertices of each K_j into a single vertex x_j and assigns it a weight equal to the sum of the weights of the vertices it replaced. Let H^* be the resulting graph.

By choice of ..., $\sqrt{m/2}$ is an upper bound on both the boundary size of x_0 and the sum of boundary sizes of $x_1, \ldots, x_p$.

Lemma 5.9.3. *For $j = 0, 1, \ldots, p$, the weight of x_j is at most $1/2$.*

Proof. Since K_0 has weight greater than $1/2$, the weight not in K_0 is less than $1/2$. The weight of x_0 corresponds to vertices not in K_0, so is less than $1/2$. Similarly, for any level-i_- BFS component K_j that is not a descendant of K_0, K_j is disjoint from K_0 so its weight is less than $1/2$. On the other hand, if K_j is a proper descendant of K_0 then its weight is at most $1/2$. □

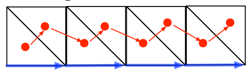
Every vertex except $x_1, \ldots, x_p$ is within $i_+ - 1 - i_-$ hops of a neighbor of x_0, as shown by this diagram:



Let $H = \text{dual}(H^*)$. Note that H can be obtained from G by edge deletions. Note also that $x_0, \ldots, x_p$ are faces in H. The boundaries of these faces are simple cycles.

Lemma 5.9.4. *Any vertex v in H is within at most $\sqrt{2m}/2$ hops of a vertex on the boundary of x_0.*

Proof. Suppose vertices $f_1, f_2, \ldots, f_q$ of H^* form a path, and each f_i has boundary size at most three. Let e_1 be an edge of f_1 and let e_q be an edge of f_q. A simple induction shows that H contains a path of size at most $\lfloor q/2 \rfloor$ from an endpoint of e_1 to an endpoint of e_q.



Now let v be any vertex in H, and let e be any incident edge. Since the sets $\{\delta_*(x_i) : i = 1, \ldots, p\}$ are disjoint by Lemma 5.8.3, H has some face f other than $x_1, \ldots, x_p$ that has v on its boundary. By Lemma 5.9.2, H^* has a path of size at most $\sqrt{2m}$ from f to a neighbor of x_0, so H has a path of size at most $\sqrt{2m}/2$ from v to a vertex on the boundary of x_0. □

The algorithm will next construct a spanning tree T of H. As in the proof of the Planar Separator Theorem, we want to ensure that every path in T is small so that every fundamental cycle is small. We would choose to make it a breadth-first-search tree except that we want it to have a special property: each of the cycles of H that form the boundaries of $x_0, \ldots, x_p$ should consist almost entirely of tree edges. More here.

Let H' be the graph obtained from H by contracting all but one edge in the boundary of x_i, for $i = 0, 1, \ldots, p$. Let T' be a breadth-first-search tree of H' rooted at the vertex resulting from contracting edges in the boundary of x_0. By Lemma 5.9.4, the depth of T' is at most $\sqrt{2m}/2$. Therefore every path in T' has size at most $\sqrt{2m}$.

Next the algorithm assigns

$$T := T' \cup \{\text{edges contracted to form } H \text{ from } H'\}$$

so T is a spanning tree of G. Since at most $2\sqrt{m/2} - 2$ edges were contracted, every path in T has size at most $\sqrt{2m} + 2\sqrt{m/2} - 2 = \sqrt{8m} - 2$, so every fundamental cycle has size at most $\sqrt{8m}$.

5.9.4 The cycle separator

The algorithm has ensured that every face has weight at most 3/4. If any face of G has weight greater than 1/4 then the boundary of such a face is a balanced cycle separator. Assume therefore that every face has weight at most 1/4. The algorithm applies Lemma 5.3.1, which states the existence of a fundamental-cycle separator. The lemma requires that the boundary of every face have at most three nontree edges. Every face of H that is also a face of G has at most three boundary edges; and the boundary of each of the new faces $(x_0, \ldots, x_p)$ has at most one nontree edge. Therefore H has a fundamental cycle with respect to T that is a $\frac{3}{4}$-balanced separator of face-weight. This cycle in H is also a cycle in G, and the way face weights were assigned in H is also $\frac{3}{4}$-balanced. This completes the proof of the Planar-Cycle-Separator Theorem.

5.10 Division into regions

5

For constants c_1, c_2, an *r-division* of an m-edge graph G is a decomposition of G into at most $c_1 m/r$ edge-disjoint subgraphs $G_1, \ldots, G_k$, each consisting of at most r edges, such that $|\partial_G(G_i)| \leq c_2\sqrt{r}$. The subgraphs are called *regions.* For each region G_i, the vertices of $\partial_G(G_i)$ are called the *boundary vertices of region G_i*, and $\partial_G(G_i)$ is called the *boundary.*

We show in this section that an r-division can be found by using calls to a procedure for finding an $O(\sqrt{n})$-vertex separator.

Theorem 5.10.1. *There is an $O(n \log n)$ algorithm that, given a planar embedded graph G and a positive integer r, computes an r-division.*

5.10.1 Concave function

In our analysis, we use *concave functions*, which we define in terms of weighted averages. Any number $0 \leq \mu \leq 1$ defines a weighted average of a pair x, y of numbers: $\mu x + (1-\mu)y$. A continuous function $f(\cdot)$ defined on an interval is *concave* if, for every number $0 \le \mu \le 1$,

$$f(\mu x + (1-\mu)y) \ge \mu f(x) + (1-\mu) f(y) \tag{5.1}$$

That is, the weighted average of the images of x and y under f is at most the image under f of the weighted average of x and y.

Fact 5.10.2. *If the second derivative of a function is nonpositive over some interval then the function is concave over that interval.*

For example, a little calculus shows that $\log x$ and $\sqrt{x}$ are monotone nondecreasing and concave.

5.10.2 Phase One: Finding regions with on-average small boundaries

The r-division algorithm consists of two phases. In Phase One, the algorithm finds a decomposition of G into $O(|E(G)|/r)$ regions such that the average number of boundary vertices per region is $O(\sqrt{r})$. Phase One consists of processing the input graph using the procedure RecursiveDivide:

```
def RecursiveDivide($G, r$):
  if $|E(G)| \le r$ then return $\{G\}$
  else
    assign equal weight to all edges of $G$
    find a vertex separator and let $G_1, G_2$ be the pieces
    return RecursiveDivide($G_1, r$) $\cup$ RecursiveDivide($G_2, r$)
```

Lemma 5.10.3. *After Phase One, the number of regions is at most* $2|E(G)|/r$.

Proof. Each final region is one of the two pieces obtained by applying a planar separator to a region with more than r edges. □

For each vertex v, let $b(v)$ be one less than the number of regions containing v.

Lemma 5.10.4. *After* RecursiveDivide *is applied to an m-edge graph G,* $\sum_{v \in V(G)} b(v) \le \frac{4m}{\sqrt{r}}$.

Proof. Let $B(m)$ be the maximum of $\sum_{v \in V(G)} b(v)$ over all m-edge graphs. The Planar-Separator Theorem applied to an m-edge graph G guarantees that the fraction of edges in each of piece lies in the interval $[\frac{1}{3}, \frac{2}{3}]$, and that the separator has size at most $4\sqrt{m}$. Therefore $B(\cdot)$ satisfies the recurrence

$$\begin{aligned} B(m) &= 0 \text{ for } m \le r \\ B(m) &\le 4\sqrt{m} + \max_{1/3 \le \alpha \le 2/3} B(\alpha m) + B((1-\alpha)m) \text{ for } m > r \end{aligned}$$

For a constant c to be determined, we show by induction that $B(m) \leq \frac{4m}{\sqrt{r}} - c\sqrt{m}$. To do this, we show that, for any α in the range $[1/3, 2/3]$,

$$4\sqrt{m} + B(\alpha m) + B((1-\alpha)m) \leq \frac{4m}{\sqrt{r}} - c\sqrt{m}.$$

By the inductive hypothesis,

$$\begin{aligned} B(\alpha m) &\leq \frac{4\alpha m}{\sqrt{r}} - c\sqrt{\alpha m} \\ B((1-\alpha)m) &\leq \frac{4(1-\alpha)m}{\sqrt{r}} - c\sqrt{(1-\alpha)m} \end{aligned}$$

so

$$\begin{aligned} & 4\sqrt{m} + B(\alpha m) + B((1-\alpha)m) && (5.2) \\ \leq \; & 4\sqrt{m} + \frac{4\alpha m}{\sqrt{r}} - c\sqrt{\alpha m} + \frac{4(1-\alpha)m}{\sqrt{r}} - c\sqrt{(1-\alpha)m} \\ = \; & 4\sqrt{m} + \frac{4m}{\sqrt{r}} - c\sqrt{m}\left(\sqrt{\alpha} + \sqrt{1-\alpha}\right) && (5.3) \end{aligned}$$

In order to select a value of c for which the induction step can be completed, we must show that $\sqrt{\alpha} + \sqrt{1-\alpha} > 1$. Let $f(x) = \sqrt{x} + \sqrt{1-x}$. Taking the second derivative shows that $f(x)$ is a concave function for $0 < x < 1$. For any $\frac{1}{3} \leq \alpha \leq \frac{2}{3}$, we can write

$$\alpha = \mu\frac{1}{3} + (1-\mu)\frac{2}{3}$$

and solve for μ. Since the resulting value of μ is between 0 and 1, we can apply Inequality 5.1 to obtain

$$f(\alpha) \geq \mu f(1/3) + (1-\mu) f(2/3)$$

Since a weighted average of two numbers is at least the minimum,

$$\mu f(1/3) + (1-\mu) f(2/3) \geq \min\{f(1/3), f(2/3)\}$$

Since $f(1/3) = f(2/3) = 1.39...$, we have shown $\sqrt{\alpha} + \sqrt{1-\alpha} \geq 1.39....$ We set $c = 4/0.39$, for then $c\sqrt{m}(\sqrt{\alpha} + \sqrt{1-\alpha}) - 4\sqrt{m} \geq c\sqrt{m}$, which shows that the right-hand side of Inequality 5.3 is bounded by $\frac{4m}{\sqrt{r}} - c\sqrt{m}$, completing the induction step. □

5.10.3 Phase Two: Splitting small regions into small regions with small boundaries

The result of Phase One is a decomposition of the input graph G into at most $2|E(G)/r$ regions of size at most r. Lemma 5.10.4 shows that the regions have small boundaries *on average* but Phase Two is responsible for ensuring that each region has a small boundary. Phase Two resembles Phase One; the difference is that weight is assigned to boundary vertices instead of to all edges. Phase Two is as follows: for each region G resulting from Phase One, call SPLIT on G and B where B is the set of vertices of G that also appear in other regions.

```
def $\text{SPLIT}_r(G, B)$:
  if $|B| \leq 16\sqrt{r}$ then return $\{G\}$
  else
    assign equal weight to all vertices in $B$ (others are assigned zero weight)
    find a vertex separator $S$ and let $G_1, G_2$ be the pieces
    return $\text{SPLIT}_r(G_1, (B \cup S) \cap V(G_1)) \cup \text{SPLIT}_r(G_2, (B \cup S) \cap V(G_2))$
```

Lemma 5.10.5. *A call* $\text{SPLIT}_r(G, B)$ *creates at most* $\max\{0, \frac{|B|}{8\sqrt{r}} - 1\}$ *new regions.*

Proof. Let $R(k)$ be the maximum number of new regions created upon invoking $\text{SPLIT}_r(G, B)$ where $|B| = k$. We show by induction that $R(k) \leq \max\{0, \frac{k}{8\sqrt{r}} - 1\}$.

For the base case, if $|B| \leq 16\sqrt{r}$ then no additional regions are created. Assume therefore that $|B| > 16\sqrt{r}$. In this case there are two recursive calls, $\text{SPLIT}_r(G_1, B_1)$ and $\text{SPLIT}_r(G_2, B_2)$ where $B_i = (B \cup S) \cap V(G_i)$.

We write $|B \cap V(G_1)| = \alpha|B|$ and $|B \cap V(G_2)| = (1-\alpha)|B|$. The separator's size guarantee ensures that $|S| \leq 4\sqrt{r}$. Hence $|B_1| \leq \alpha|B| + 4\sqrt{r}$ and $|B_2| \leq (1-\alpha)|B| + 4\sqrt{r}$. The separator's balance condition ensures that $\frac{1}{3} \leq \alpha \leq \frac{2}{3}$. This combined with the fact that $|B| > 16\sqrt{r}$ implies that $|B_i| < |B|$ for $i = 1, 2$.

Hence, by the inductive hypothesis, the number of new regions created by the call $\text{SPLIT}_r(G_1, B_1)$ is at most $\max\{0, \frac{|B_1|}{8\sqrt{r}} - 1\}$, which is in turn at most $\frac{\alpha|B|+4\sqrt{r}}{8\sqrt{r}} - 1$. Similarly, the number created by the call $\text{SPLIT}_r(G_2, B_2)$ is at most $\frac{(1-\alpha)|B|+4\sqrt{r}}{8\sqrt{r}} - 1$. Since the call $\text{SPLIT}_r(G, B)$ directly created one new region, the total number of new regions created is at most

$$1 + \frac{\alpha|B| + 4\sqrt{r}}{8\sqrt{r}} - 1 + \frac{(1-\alpha)|B| + 4\sqrt{r}}{8\sqrt{r}} - 1$$

which is at most $\frac{|B|}{8\sqrt{r}} - 1$, proving the induction step. □

Suppose Phase One and Phase Two are executed on a graph with m edges. After Phase One, the sum over all regions R of the number of vertices in R that also appear in other regions is at most $\sum_v 2b(v)$. By Lemma 5.10.4, this sum is at most $\frac{8m}{\sqrt{r}}$. By Lemma 5.10.4, the number of new regions introduced by Phase Two is therefore at most $\frac{m}{r}$. By Lemma 5.10.3, the number of regions resulting from Phase One is at most $2m/r$. Hence the total number of regions after Phase One and Phase Two is at most $3m/r$. Since Phase Two ensures that each region has at most $16\sqrt{r}$ boundary vertices, the resulting set of regions form an r-division.

Now we consider the running time. Traditional analysis of Phase One, using the fact that finding a separator in an m-edge graph, shows that it runs in $O(m \log m)$ time. The analysis of Lemma 5.10.5 shows that calling $\text{Split}_r(G, B)$ results in at most $\max\{0, \frac{|B|}{8\sqrt{r}} - 1\}$ calls to the vertex-separator algorithm; as in the analysis of new regions, by Lemma 5.10.4 the total number of such calls for all of Phase Two is at most $\frac{m}{r}$. Each call is on a graph consisting of at most r edges, so the total time for Phase Two is $O(m)$. We have proved Theorem 5.10.1.

5.11 Recursive divisions

Let $\bar{r} = (r_0, r_1, \ldots)$ be an increasing sequence of positive integers. We say the *height* of a graph G with respect to $\bar{r}$ is the smallest integer i such that the graph has at most r_i arcs. For a fixed sequence $\bar{r}$, we denote the height of G by $\text{height}(G)$.

A *recursive $\bar{r}$-division* of a nonempty graph G is a rooted tree whose vertices are subgraphs of G, defined inductively as follows. The root of the tree is the graph G. If G has one arc, the root has no children. Otherwise, the children of the root are the regions $G_1, \ldots, G_k$ forming an $r_{\text{height}(G)-1}$-division of G. Moreover, each child is the root of a recursive $\bar{r}$-division. The parent of a region R is denoted by $\text{parent}(R)$. The region consisting of a single arc uv is denoted $R(uv)$. Such a region is said to be *atomic*.

5.12 History

Lipton and Tarjan [Lipton and Tarjan, 1979] proved the first separator theorem for planar graphs. Constant-factor Improvements were found by Djidjev [Djidjev, 1981] and Gazit [Gazit, 1986]. Goodrich [Goodrich, 1995] gave a linear-time algorithm to find a recursive decomposition using planar separators. Miller [Miller, 1986] proved the first cycle-separator theorem for planar graphs. A constant-factor improvement was found by Djidjev and Venkatesan [Djidjev and Venkatesan, 1997]. The notion of an r-division is due to Frederickson [Frederickson, 1987], who gave the algorithm for finding one.

Chapter 6

Shortest paths with nonnegative lengths

In this chapter, we give a linear-time algorithm for computing single-source shortest paths in a planar graph with nonnegative lengths. The algorithm uses recursive divisions (discussed in Section 5.10). We start with some basic concepts about shortest paths in arbitrary graphs.

6.1 Shortest-path basics: path-length property and relaxed and tense darts

Let G be a directed graph with dart-lengths given by a dart-vector $\boldsymbol{c}$. We say a vertex vector $\boldsymbol{d}$ has the *path-length property* with respect to $\boldsymbol{c}$ if, for each vertex v, $\boldsymbol{d}[v]$ is the length (with respect to $\boldsymbol{c}$ of some s-to-v path (not necessarily the shortest).

We say a dart vw is *relaxed* with respect $\boldsymbol{c}$ and the vertex vector $\boldsymbol{d}$ if $\boldsymbol{d}[w] \leq \boldsymbol{d}[v] + \text{length}(vw)$. An unrelaxed dart is said to be *tense*.

Let s be a vertex. If $\boldsymbol{d}[s] = 0$ and $\boldsymbol{d}$ satisfies the path-length property and every arc is relaxed then, for each vertex v, $\boldsymbol{d}[v]$ is the length of the *shortest* s-to-v path.

A basic step in several shortest-path algorithms is called *relaxing* a dart. Suppose $\boldsymbol{d}$ is a vertex vector with the path-length property and dart vw is tense. Relaxing vw consists of executing

$$\boldsymbol{d}[w] := \text{length}(vw) + \boldsymbol{d}[v]$$

after which vw is relaxed and $\boldsymbol{d}$ still has the path-length property.

For the case of nonnegative lengths, Dijkstra's algorithm [**?**] generates an ordering of darts to relax so that each dart is relaxed at most once, after which all dart are relaxed. If the algorithm uses a priority queue as suggested by

Johnson [**?**], the running time is $O(m \log n)$ where m is the number of arcs (we assume $m \geq n - 1$ since otherwise the graph is disconnected).

6.2 Using a division in computing shortest-path distances

The shortest-path algorithm operates on a graph equipped with a recursive division. The algorithm runs quickly because most queue operations are performed on smaller queues.

The algorithm has a limited "attention span." It chooses a region, then performs a number of steps of Dijkstra's algorithm (the number depends on the height of the region), then abandons that region until later. Thus it skips around between the regions.

To provide intuition, we briefly describe a simplified version of the algorithm. The simplified version runs in $O(n \log \log n)$ time. Divide the graph into $O(n/\log^4 n)$ regions of size $O(\log^4 n)$ with boundaries of size $O(\log^2 n)$. We associate a status, *active* or *inactive*, with each edge. Initialize by deactivating all edges and setting all node labels $\boldsymbol{d}[v]$ to infinity. Then set the label of the source to 0, and activate its outgoing edges. Now repeat the following three steps:

Step 1: Select the region containing the lowest-labeled node that has active outgoing edges in the region.

Step 2: Repeat $\log n$ times:

Step 2A: Select the lowest-labeled node v in the current region that has active outgoing edges in the region. Relax and deactivate all its outgoing edges vw in that region. For each of the other endpoints w of these edges, if relaxing the edge vw resulted in decreasing the label of w, then activate the outgoing edges of w.

Note that applying Dijkstra's algorithm to the region would entail repeating Step 2A as many times as there are nodes in the region. Every node would be selected exactly once. We cannot afford that many executions of Step 2A, since a single region is likely to be selected more than once in Step 1. In contrast to Dijkstra's algorithm, when a node is selected in Step 2A, its current label may not be the correct shortest-path distance to that node; its label may later be decreased, and it may be selected again. Since the work done within a region during a single execution of Step 2 is speculative, we don't want to do too much work. On the other hand, we don't want to execute Step 1 too many times. In the analysis of this algorithm, we show how to "charge" an execution of Step 1 to the $\log n$ iterations of Step 2A.

There is an additional detail. It may be that Step 2A cannot be repeated $\log n$ iterations because after fewer than $\log n$ times there are no active outgoing edges left in the region. In this case, we say the execution of Step 2 is *truncated.*

Since we cannot charge a truncated execution of Step 2 to $\log n$ iterations of Step 2A, we need another way to bound the number of such executions. It turns out that handling this "detail" is in fact the crux of the analysis. One might think that after a region R underwent one such truncated execution, since all its edges were inactive, the same region would never again be selected in Step 1. However, relax-steps on edges in another region R' might decrease the label on a node w on the boundary between R and R', which would result in w's outgoing edges being activated. If w happens to have outgoing edges within R, since these edges become active, R will at some later point be selected once again in Step 1.

This problem points the way to its solution. If R "wakes up" again in this way, we can charge the subsequent truncated execution involving R to the operation of updating the label on the boundary node w. The analysis makes use of the fact that there are relatively few boundary nodes to bound the truncated executions. Indeed, this is where we use the fact that the regions have small boundaries.

6.2.1 The algorithm

We assume without loss of generality that the input graph G is directed, that each node has at most two incoming and two outgoing edges, and that there is a finite-length path from s to each node. We assume that the graph is equipped with a recursive division.

The algorithm maintains a vertex vector $\boldsymbol{d}$ with the path-length property.

For each region R of the recursive division of G, the algorithm maintains a priority queue $Q(R)$. If R is nonatomic, the items stored in $Q(R)$ are the immediate subregions of R. For an atomic region $R(uv)$, $Q(R(uv))$ consists of only one item, the single arc uv contained in $R(uv)$; in this case, the key associated with the arc is either infinity or the label $\boldsymbol{d}[u]$ of the tail of the arc.

The algorithm is intended to ensure that for any region R, the minimum key in the queue $Q(R)$ is the minimum distance label $\boldsymbol{d}[v]$ over all arcs vw in R that need to be processed. We make this precise in Lemma 6.3.3. This idea of maintaining priority queues for nested sets is not new, and has been used, e.g. in finding the k^{th} smallest element in a heap [. Frederickson min-heap .]. Fix reference

We assume the priority queue data structure supports the operations

- *minItem*(Q), which returns the item in Q with the minimum key,
- *minKey*(Q), which returns the key associated with *minItem*(Q)
- *updateKey*(Q, x, k), which updates the key of x to k (x must be an item of Q) and returns a boolean indicating whether the update caused *minKey*(Q) to decrease.

We indicate an item is inactive by setting its key to infinity. Items go from inactive to active and back many times during the algorithm. We never delete items from the queue. This convention is conceptually convenient because it avoids the issue of having to decide, for a given newly active item, into which of the many priority queues it should be re-inserted.

SUMMARY, DEFINITION OF TRUNCATED—SEE BEGINNING OF SECTION 6.4 (ANALYSIS OF PROCESS, NOT INCLUDING UPDATE) FOR SOME INTUITION THAT CAN BE BORROWED FOR THIS PART.

The algorithm uses parameters α_i to specify an "attention span" for each level of the recursive division. We will specify their values in Section 6.5. The algorithm consists of the following two procedures.

```
def Process($R$)
        pre: $R$ is a region.
1       if $R$ contains a single edge $uv$,
2         if $\boldsymbol{d}[v] > \boldsymbol{d}[u] + \text{length}(uv)$,
3           $\boldsymbol{d}[v] := \boldsymbol{d}[u] + \text{length}(uv)$
4           for each outgoing edge $vw$ of $v$, call Update($R(vw), vw, \boldsymbol{d}[v]$).
5         updateKey($Q(R), uv, \infty$).
6       else ($R$ is nonatomic)
7         repeat $\alpha_{height(R)}$ times or until minKey($Q(R)$) is infinity:
8             $R'$ := minItem($Q(R)$)
9             Process($R'$)
10            updateKey($Q(R), R'$, minKey($Q(R')$)).
```

```
def Update($R, x, k$)
        : pre: $R$ is a region, $x$ is an item of $Q(R)$, and $k$ is a key value.
1         updateKey($Q(R), x, k$)
2         if the updateKey operation reduced the value of minKey($Q(R)$) then
3           Update(parent($R$), $R$, $k$).
```

To compute shortest paths from a source s, proceed as follows. Initialize all vertex-labels and keys to infinity. Then assign the label $\boldsymbol{d}[s] := 0$, and for each outgoing arc sv, call Update($R(sv), sv, 0$). Then repeatedly call Process(R_G), where R_G is the region consisting of all of G, until the call results in *minKey*($Q(R_G)$) being infinity. Since the vertex-labels $\boldsymbol{d}$ are only updated by steps in which arcs are relaxed, the vertex-labels satisfy the path-length property throughout the algorithm's execution. In the next section, we show that, when the algorithm terminates, all arcs are relaxed. It follows that the vertex-labels give distances from s.

We define an *entry vertex* of a region R as follows. The only entry vertex of the region R_G is s itself. For any other region R, v is an entry vertex if v is a boundary vertex of R such that some arc vw belongs to R.

When the algorithm processes a region R, it finds shorter paths to some of the vertices of R and so reduces their labels. Suppose one such vertex v is a boundary vertex of R. The result is that the shorter path to v can lead to shorter paths to arcs in a neighboring region R' for which v is an entry vertex. In order to preserve the property that the minimum key of $Q(R')$ reflects the labels of vertices of R', therefore, the algorithm might need to update $Q(R')$. Updating priority queues of neighboring regions is handled by the Update proedure. The reduction of *minKey*$(Q(R'))$ (which can *only* occur as a result of the reduction of the label of an entry vertex v) is a highly significant event for the analysis. We refer to such an event as a *foreign intrusion of region* R' *via entry vertex* v.

Lemma 6.2.1. *Let* R *be a region. Suppose there are two foreign intrusions of* R *via* v*, one at time* t_1 *and one at time* t_2*, where* $t_1 < t_2$*. Then minKey*$(Q(R))$ *is greater at time* t_1 *than at time* t_2*.*

Proof. $\boldsymbol{d}[v]$ must get smaller for the second intrusion to count. □

6.3 Correctness

We say that an arc uv is *active* if the key of uv in $Q(R(uv))$ is finite. To prove that every arc is relaxed at termination, we show that (a) if an arc is inactive, then it is relaxed, and (b) at termination all arcs are inactive.

Lemma 6.3.1. *If an arc* uv *is inactive then it is relaxed (except during Line 4).*

Proof. The lemma holds just before the first call to Process since at that point every node but s has label infinity, and every outgoing arc of s is active. The algorithm only deactivates an arc uv, i.e., uv is assigned a key of ∞ in Line 5, just after the arc is relaxed.

An arc vw could become tense when the labels of its endpoints change. Note that labels never go up. The label of v might go down in Line 4, but in the same step the algorithm calls Update$(R(vw), vw, \boldsymbol{d}[v])$ for each outgoing arc vw of v. In Line 1 of Update, the key of vw is updated to a finite value, so vw is again active. □

Lemma 6.3.2. *The key of an active arc* vw *is* $\boldsymbol{d}[v]$ *(except during Line 4).*

Proof. Initially all labels and keys are ∞. Whenever a label $\boldsymbol{d}[v]$ is assigned a value k (either in the initialization, where $v = s$, or in Line 4), Update$(R(vw), vw, k)$ is called for each outgoing arc vw. The first step of Update(R, v, k) is to update the key of vw to k. □

Next we show that the queues are, in a sense, consistent. The region of an invocation A of Process is simply the region that was the argument to that invocation of Process. The most recent invocation of Process that has not yet returned is called the *current invocation*, and that invocation's region is called the *current region*.

Lemma 6.3.3. *For any region* R *that is not an ancestor of the current region, the key associated with* R *in* $Q(parent(R))$ *is the min key of* $Q(R)$*.*

Proof. At the very beginning of the algorithm, all keys are infinity. Thus in this case the lemma holds trivially. Every time the minimum key of some queue $Q(R)$ is changed in Line 1 of Update, a recursive call to Update in Line 2 ensures that the key associated with R in $Q(\text{parent}(R))$ is updated.

We must also consider the moment when a new region becomes the current region. This happens upon invocation of the procedure Process, and upon return from Process.

- When Process(R) is newly invoked, the new current region R is a child of the old current region, so Lemma 6.3.3 applies to even fewer regions than before; hence we know it continues to hold.
- When Process(R') returns in step 9, the parent R of R' becomes current. Hence at that point Lemma 6.3.3 applies to R'. Note, however, that immediately after the call to Process(R'), the calling invocation updates the key of R' in $Q(R)$ to the value $minKey(Q(R'))$.

□

Corollary 6.3.4. *For any region R that is not an ancestor of the current region,*

$$minKey(Q(R)) = \min\{\boldsymbol{d}[v] : vw \text{ is an active arc contained in } R\} \tag{6.1}$$

Proof. By induction on the height of R, using Lemma 6.3.3. □

The algorithm terminates when $Q(R_G)$ becomes infinite. At this point, according to Corollary 6.3.4, G contains no active arc, so, by Lemma 6.3.1, all arcs are relaxed. Since the vertex-labels satisfy the path-length property, it follows that they are shortest-path distances. We have proved the algorithm is correct.

6.4 The Dijkstra-like property of the algorithm

Because of the recursive structure of Process, each initial invocation Process(R_G) is the root of a tree of invocations of Process and Update, each with an associated region R. Parents, children, ancestors, and descendants of invocations are defined in the usual way.

In this section, we give a lemma that is useful in the running-time analysis. This lemma is a consequence of the nonnegativity of the lengths; it is analogous to the fact that in Dijkstra's algorithm the labels are assigned in nondecreasing order.

For an invocation A of Process on region R, we define start(A) and end(A) to be the values of $minKey(Q(R))$ just before the invocation starts and just after the invocation ends, respectively.

Lemma 6.4.1. *For any invocation A, and for the children $A_1, \ldots, A_p$ of A,*

$$start(A) \leq start(A_1) \leq start(A_2) \leq \cdots \leq start(A_p) \leq end(A). \tag{6.2}$$

Moreover, every key assigned during A is at least $start(A)$.

Proof. The proof is by induction on the height of A. If A's height is 0 then it has an atomic region $R(uv)$. In this case the start key is the value of $\boldsymbol{d}[u]$ at the beginning of the invocation. The end key is infinity. The only finite key assigned (in UPDATE) is $\boldsymbol{d}[u]+\text{length}(uv)$, which is at least $\boldsymbol{d}[u]$ by nonnegativity of edge-lengths. There are no children.

Suppose A's height is more than 0. In this case it invokes a series $A_1, \ldots, A_p$ of children. Let R be the region of A. For $i = 1, \ldots, p$, let k_i be the value of $minKey(Q(R))$ at the time A_i is invoked, and let k_{p+1} be its value at the end of A_p. Line 8 of the algorithm ensures $k_i = \text{start}(A_i)$ for $i \leq p$. By the inductive hypothesis, every key assigned by A_i is at least $\text{start}(A_i)$, so $k_{i+1} \geq k_i$. Putting these inequalities together, we obtain

$$k_1 \leq k_2 \leq \cdots \leq k_{p+1}$$

Note that $k_1 = \text{start}(A)$ and $k_{p+1} = \text{end}(A)$. Thus (6.2) holds and every key assigned during A is at least $\text{start}(A)$. □

6.5 Accounting for costs

Now we begin the running-time analysis. The time required is dominated by the time for priority-queue operations. Lines 8 and 10 of PROCESS involve operations on the priority queue $Q(R)$. We charge a total cost of $\log |Q(R)|$ for these two steps. Similarly, we charge a cost of $\log |Q(R)|$ for Line 3 of UPDATE. Line 5 performs an operation on a priority queue of size one, so we only charge one for that operation. Our goal is to show that the total cost is linear.

To help in the analysis, we give versions of the procedures UPDATE and PROCESS that have been modified to keep track of the costs and to keep track of foreign intrusions. Note that the modifications are purely an expository device for the purpose of analysis; the modified procedures are not intended to be actually executed, and in fact one step of the modified version of PROCESS cannot be executed since it requires knowledge of the future!

Amounts of cost are passed around by UPDATE and PROCESS via return values and arguments. We think of these amounts as *debt obligations*. These debt obligations travel up and down the forest of invocations, and are eventually charged by invocations of PROCESS to pairs (R, v) where R is a region and v is an entry vertex of R.

The modified version of UPDATE, given below, handles cost in a simple way: each invocation returns the total cost incurred by it and its proper descendants (in Line 3 if there are proper descendants and in Line 3a if not).

The modified UPDATE has another job to do as well: it must keep track of foreign intrusions. There is a table entry$[\cdot]$ indexed by regions R. Whenever the reduction of the vertex-label $\boldsymbol{d}[v]$ causes $minKey(Q(R))$ to decrease, entry$[R]$ is set to v in step 2a. Initially the entries in the table are undefined. However, for any region R, the only way that $minKey(Q(R))$ can become finite is by an intrusion. We are therefore guaranteed that, at any time at which $minKey(Q(R))$ is finite, entry$[R]$ is an entry vertex of R.

```
def UPDATE($R, x, k, v$)
      pre: $R$ is a region, $x$ is an item of $Q(R)$, $k$ is a key value,
            and $v$ is a boundary vertex of $R$.
1     updateKey($Q(R), x, k, v$)
2     if the updateKey operation reduced the value of $minKey(Q(R))$ then
2a       entry[$R$] := $v$
3        return $\log|Q(R)|$ + UPDATE(parent($R$), $R, k, v$).
3a    else return $\log|Q(R)|$
```

The cost of the invocation UPDATE(R, x, k, v) is $\log|Q(R)|$ because of the *updateKey* operation in step 1.

We now turn to the modified PROCESS procedure. An invocation takes an additional argument, debt, which is a portion of the cost incurred by ancestor invocations. We refer to this amount as the debt *inherited* by the invocation.

Let R be the invocation's region. If R is atomic, the invocation's debt is increased by the cost incurred by calls to UPDATE, and then by 1 to account for the *updateKey* operation on the single-element priority queue.

If R is not atomic, the invocation will have some children, so can pass some of its debt down to its children. Since the parent invocation expects to have $\alpha_{\text{height}(R)}$ children, it passes on to each child

- a $1/\alpha_{\text{height}(R)}$ fraction of the parent's inherited debt, plus
- the $\log|Q(R)|$ cost the parent incurred in selecting the child's region from the priority queue.

The parent uses a variable *credit* to keep track of the amount of its inherited debt that it has successfully passed down to its children. If the parent has $\alpha_{\text{height}(R)}$ children, the parent's total credit equals its inherited debt. If not, we say the parent invocation is *truncated.*

Lemma 6.5.1. *A nontruncated invocation sends to its children all the debt it inherits or incurs.*

Debts also move *up the tree.* The parent invocation receives some debt from each child. The parent adds together

- the debt received from its children (upDebt) and
- the amount of inherited debt for which it has not received a credit (zero unless the parent is truncated)

to get the total amount the parent owes. The parent then either passes that aggregate debt to *its* parent or pays off the debt itself by withdrawing from an account, the account associated with the pair (R, v) where v is the value of entry[R] at the time of the invocation (step 10d).

Because of Lemma 6.5.1, as debt moves up the tree, no new debt is added by nontruncated invocations of PROCESS.

Clarify what "new" debt means?

```
def $\textsc{Process}(R, \text{debt})$
        pre: $R$ is a region.
1       if $R$ contains a single edge $uv$,
2         if $\boldsymbol{d}[v] > \boldsymbol{d}[u] + \text{length}(uv)$,
3           $\boldsymbol{d}[v] := \boldsymbol{d}[u] + \text{length}(uv)$
4           for each outgoing edge $vw$ of $v$, debt$+ = \textsc{Update}(R(vw), vw, \boldsymbol{d}[v], v)$.
5           $\textit{updateKey}(Q(R), uv, \infty)$.
5a          debt$+ = 1$
6       else ($R$ is nonatomic)
6a        upDebt:= 0, credit:=0
7         repeat $\alpha_{\text{height}(R)}$ times or until $\textit{minKey}(Q(R))$ is infinity:
8             $R' := \textit{minItem}(Q(R))$
8a            credit$+ = \text{debt}/\alpha_{\text{height}(R)}$
9             upDebt$+ = \textsc{Process}(R', \text{debt}/\alpha_{\text{height}(R)} + \log|Q(R)|)$
10            $\textit{updateKey}(Q(R), R', \textit{minKey}(Q(R')))$.
10a       debt$+ = \text{upDebt} - \textit{credit}$
10b     if $\textit{minKey}(Q(R))$ will decrease in the future,
10b       return debt
10c     else (this invocation is stable)
10d       pay off debt by withdrawing from account of $(R, \text{entry}[R])$
10e       return 0
```

We say an invocation A of Process is *stable* if, for every invocation $B > A$, the start key of B is at least the start key of A. If an invocation is stable, it pays off the debt by withdrawing the necessary amount from the account associated with $(R, \text{entry}[R])$. If not, it passes the debt up to its parent. The following theorem is proved in Section 6.6

Theorem 6.5.2 (Payoff Theorem). *For each region R and entry vertex v of R, the account (R, v) is used to pay off a positive amount at most once.*

Any invocation whose region is the whole graph is stable because there are no foreign intrusions of that region. Therefore such an invocation never tries to pass any debt to its nonexistent parent. We are therefore guaranteed that all costs incurred by the algorithm are eventually charged to accounts.

The total computational cost depends on the parameters $\bar{r} = (r_0, r_1, r_2, \ldots)$ of the recursive $\bar{r}$-division and on the parameters $\alpha_0, \alpha_1, \ldots$ that govern the number of iterations per invocation of Process. For now, we define the latter parameters in terms of the former parameters; later we define the former. Define $\alpha_i = \frac{4 \log r_{i+1}}{3 \log r_i}$.

Lemma 6.5.3. *Each invocation at height i inherits at most $4 \log r_{i+1}$ debt.*

Proof. By reverse induction on i. Each top-level invocation inherits no debt. Suppose the lemma holds for i, and consider a height-i invocation. By the inductive hypothesis, it inherits at most $4 \log r_{i+1}$ debt, so it passes down to each child a debt of at most $\frac{4\log r_{i+1}}{\alpha_i} + \log r_i$. The choice of α_i ensures that this is $4\log r_i$. □

Define $\beta_{ij} = \alpha_i \alpha_{i-1} \dots \alpha_{j+1}$ for $i > j$. β_{ii} is defined to be 1, and for $i < j$, we define β_{ij} to be zero.)

Lemma 6.5.4. *A* Process *invocation of height i has at most β_{ij} descendant* Process *invocations of height j.*

Debt incurred by the algorithm by a step of Process is called *process debt.*

Lemma 6.5.5. *For each height-i region R and boundary vertex x of R, the amount of process debt payed off by the account (R, x) is at most $\sum_{j \le i} \beta_{ij} 4 \log r_{j+1}$.*

Proof. By the Payoff Theorem, this account is used at most once. Let A be the invocation of Process that withdraws the payoff from that account. Each dollar of process debt paid off by A was sent back to A from some descendant invocation who inherited or incurred that dollar of debt. Thus, to account for the total amount of process debt paid off by A, we consider each of its descendant invocations. By Lemma 6.5.4, A has β_{ij} descendants of height j, and each such descendant inherited or incurred a debt of at most $4\log r_{j+1}$ dollars, by Lemma 6.5.3. □

Cost incurred by the algorithm by a step of Update is called *update debt.* The event (in Line 3 of Process) of reducing a vertex v's label $\boldsymbol{d}[v]$ initiates a chain of calls to Update in Line 4 for each outgoing arc vw. We say the debt incurred is *on behalf of v.*

Lemma 6.5.6. *If* Update *is called on the parent of region R during an invocation A of* Process *then R is not the region of A.*

Proof. Let R_A be the region of A. Recall that start(A) is the value of *minKey*$(Q(R_A))$ just before A begins. By Lemma 6.4.1, every key assigned during A is at least start(A).

Consider a chain of Update calls initiated by the reduction of the label of vertex v. By the condition in Line 2 of Process and the condition in Line 2 of Update, in order for Update to be called on the parent of R, that label must have been less than the value of *minKey*$(Q(R))$. This shows $R \neq R_A$. □

For a vertex v, define

$$\text{height}(v) = \max\{j : v \text{ is a boundary vertex of a height-}j\text{ region}\}$$

Corollary 6.5.7. *A chain of calls to* Update *initiated by the reduction of the label of v has total cost at most $\sum_{k \le height(v)+1} \log r_k$.*

Proof. Let A_0 be the invocation of Process during which the initial call to Update was made, and let $R(uv)$ be the (atomic) region of A_0. Consider the chain of calls to Update, and let

$$R(vw) = R_0, R_1, \dots, R_p$$

be the corresponding regions. Note that $\text{height}(R_j) = j$. The cost of call j is $\log|Q(R_j)|$, which is at most $\log r_j$. Since R_{p-1} contains vw but (by Lemma 6.5.6) does not contain uv, v is a boundary vertex of R_{p-1}, so $p \leq \text{height}(v) + 1$. □

6.6 The Payoff Theorem

In this section, we prove the Payoff Theorem, repeated here for convenience:

> **Payoff Theorem** (Theorem 6.5.2): For each region R and entry vertex v of R, the account (R, v) is used to pay off a positive amount at most once.

In this section, when we speak of an *invocation*, we mean an invocation of Process. We define the partial order $\leq$ on the set of invocations of Process as follows: $A \leq B$ if A and B have the same region, and A occurs no later than B. We say in this case that A is a *predecessor* of B. We write $A < B$ if $A \leq B$ and $A \neq B$.

As we remarked earlier, the only way *minKey*$(Q(R))$ can decrease is if there is a foreign intrusion into R. We restate this as follows.

Lemma 6.6.1. *Let B be an invocation with region R. Suppose that between time t and the time B starts, there are no foreign intrusions of R. Then start(B) is at least the value of minKey$(Q(R))$ at time t.*

Lemma 6.6.2. *Suppose $A < B$ are two invocations such that no foreign intrusion occurs between A and B and such that B is stable. Then every child of A is stable.*

Proof. Let A' be a child of A, and let C' be any invocation such that $A' < C'$. If we can prove

$$\text{start}(A') \leq \text{start}(C') \tag{6.3}$$

then it will follow that A' is stable. Let C be the parent invocation of C' (so the region of C' is R). If $C = A$, then (6.3) follows from Lemma 6.4.1.

Assume therefore that $C > A$. It follows from Lemma 6.4.1 that $\text{start}(A') \leq \text{end}(A)$ and that $\text{start}(C) \leq \text{start}(C')$, so it suffices to show $\text{end}(A) \leq \text{start}(C)$. There are two cases.

- *Case 1: $B \leq C$* . In this case, $\text{end}(A) \leq \text{start}(C)$ by Lemma 6.6.1.
- *Case 2: $C > B$*. In this case, $\text{end}(A) \leq \text{start}(B)$ follows by Lemma 6.6.1, and $\text{start}(B) \leq \text{start}(C)$ follows by the stability of B, so $\text{end}(A) \leq \text{start}(C)$.

□

Now we can prove the Payoff Theorem, which states that each pair (R, v) is charged a positive amount at most once.

If (R, v) is never charged to, we are done. Otherwise, let A be the earliest invocation that pays off a positive amount from the account (R, v) in step 10d. Then R is the region of A, v is the value of entry$[R]$ at the time of A, and A is stable. Let t_1 be the time when entry$[R]$ was last set before A.

Assume for a contradiction that there is an invocation B such that $A < B$ and such that B also charges to (R, v). Then v is the value of entry$[R]$ at the time of B, and B is stable. Let t_2 be the time when entry$[R]$ was last set before B. If $t_2 > t_1$ then, by Lemma 6.2.1, *minKey*$(Q(R))$ at time t_2 was less than that at time t_1, which in turn was no more than start(A) by Lemma 6.2.1, contradicting the stability of A.

We conclude that no foreign intrusion of R occurs between A and B. By Lemma 6.6.2, therefore, every child of A is stable. It follows from step 10e that every child of A returns a zero debt, so in invocation A the value of upDebt is zero. Assume for a contradiction that A's credit does not cover its own inherited debt. Then A must be a truncated invocation, so end$(A) = \infty$. By Lemma 6.6.1, start$(B) = \infty$, a contradiction. Therefore, A pays off zero, a contradiction. This completes the proof of the Payoff Theorem.

6.7 Analysis

Let c_1, c_2 be the constants such that an r-division of an m-arc graph has at most $c_1 m/r$ regions, each having at most $c_2\sqrt{r}$ boundary vertices.

Lemma 6.7.1. *Let m be the number of edges of the input graph. For any nonnegative integer i, there are at most $c_1 c_2 m/\sqrt{m}$ pairs (R, x) where R is a height-i region and x is an entry vertex of R.*

Combining this lemma with Lemma 6.5.5, we obtain

Corollary 6.7.2. *The total process debt is at most*

$$\sum_i \frac{m}{\sqrt{r_i}} \sum_{j \leq i} \beta_{ij} 4 \log r_{j+1} \tag{6.4}$$

Lemma 6.7.3. *Let i, j be nonnegative integers, and let R be a region of height i. The total amount of update debt incurred on behalf of vertices of height at most j and paid off from accounts $\{(R, x) : x \text{ an entry vertex of } R\}$ is at most*

$$c_2 \sqrt{r_i} \beta_{i0} \sum_{k=0}^{j+1} \log r_k$$

Proof. The number of entry vertices x of R is at most $c_2\sqrt{r_i}$. For each, the Payoff Theorem ensures that all the debt paid off from account (R, x) comes from descendants of a single invocation A of Process. The number of height-0 descendants of A is β_{i0}. For each such level-0 descendant A_0, if the corresponding update debt is on behalf of a vertex of height at most j then by Corollary 6.5.7 the cost is at most $\sum_{k=0}^{j+1} \log r_k$. □

Lemma 6.7.4. *The total update debt is at most*

$$\sum_i c_1 c_2 \frac{m}{r_i} \sqrt{r_i} \beta_{i0} \sum_{k=0}^{i+1} \log r_k \tag{6.5}$$

$$+ \quad \sum_j c_1 \frac{m}{\sqrt{r_j}} 2 \sum_{i<j} c_2 \sqrt{r_i} \beta_{i0} \sum_{k=0}^{j+1} \log r_k \tag{6.6}$$

Proof. To each unit of update debt, we associate two integers: i is the height of the region R such that the debt is paid off from an account (R, x), and j is the height of the vertex v on whose behalf the debt was incurred. If $i \geq j$, we refer to the debt as *type 1* debt, and if $i < j$, we refer to it as *type 2* debt.

First we bound the type-1 debt. For each integer i, there are at most $c_1 \frac{m}{r_i}$ regions R of height i. By Lemma 6.7.3, the total type-1 debt is therefore

$$\sum_i c_1 c_2 \frac{m}{r_i} \sqrt{r_i} \beta_{i0} \sum_{k=0}^{i+1} r_k$$

Now we bound the type-2 debt. For each integer j, the number of regions of height j is at most $c_1 \frac{m}{r_j}$ and each has at most $c_2 \sqrt{r_j}$ entry vertices, so the total number of vertices of height j is at most $c_1 c_2 \frac{m}{\sqrt{r_j}}$. For each such vertex v, there are at most 2 incoming darts uv. Any height-0 invocation of PROCESS that reduced v's label involved one of these two darts. For each such dart uv, for each integer $i < j$, there is exactly one height-i region R that includes uv. By Lemma 6.7.3, the total type-2 debt is therefore

$$\sum_j c_1 \frac{m}{\sqrt{r_j}} 2 \sum_{i<j} c_2 \sqrt{r_i} \beta_{i0} \sum_{k=0}^{j+1} r_k$$

□

6.8 Parameters

In this section, we show that there is a way to choose the parameters $r_0, r_1, r_2, \ldots$ so that the total process cost and the total update cost are $O(m)$.

Problem 6.8.1. *For $r_0 = 1, r_1 = \log^4 m$, and $r_2 = m$, show that the total cost is $O(m \log \log m)$.*

We define $r_0, r_1, \ldots$ inductively by $r_0 = 1$ and $r_{j+1} = 16^{r_j^{1/6}}$. This defines an increasing sequence such that

$$\log r_{j+1} = 4 r_j^{1/6} \tag{6.7}$$

so

$$\log^2 r_{j+1} = 16 r_j^{1/3} \tag{6.8}$$

Lemma 6.8.2. *$r_j^{1/6} \geq 1.78^j$ for $j \geq 7$.*

Proof. Define $u_j = \log r_j^{1/6}$. Then $\log r_{j+1} = 4 \cdot 2^{u_j}$ and

$$u_{j+1} = \log r_{j+1}^{1/6} = \frac{1}{6} \log r_{j+1} = \frac{1}{6} 4 \cdot 2^{u_j} = \frac{2}{3} 2^{u_j}$$

A simple induction shows that $u_j \geq .838j$ for $j \geq 7$. Since $r_j^{1/6} = 2^{u_j}$, this implies the lemma.. □

Lemma 6.8.3. *The process-debt (6.4) is $O(m)$.*

Proof.

$$\begin{aligned}
&\sum_i \frac{m}{\sqrt{r_i}} \sum_{j \leq i} \beta_{ij} 4 \log r_{j+1} \\
&= \quad \sum_i \frac{m}{\sqrt{r_i}} \sum_{j \leq i} (4/3)^{i-j} \frac{\log r_{i+1}}{\log r_{j+1}} 4 \log r_{j+1} \text{ by definition of } \beta_{ij} \\
&= \quad 4m \sum_i r_i^{-1/2} \sum_{j \leq i} (4/3)^{i-j} \log r_{i+1} \\
&\leq \quad 4m \sum_i r_i^{-1/2} c(4/3)^i \log r_{i+1} \text{ for a constant } c \\
&\leq \quad 4cm \sum_i r_i^{-1/2} (4/3)^i 4 r_i^{1/6} \\
&= \quad 16cm \sum_i r_i^{-1/3} (4/3)^i
\end{aligned}$$

which is $O(m)$ by Lemma 6.8.2. □

Lemma 6.8.4. *The update-debt is $O(m)$.*

Proof. First we show that $\sum_j r_j^{-1/2} (4/3)^j \log^2 r_{j+1}$ is bounded by a constant c. (The recurrence relation for r_j was chosen to make this true.)

$$\begin{aligned}
\sum_j r_j^{-1/2} (4/3)^j \log^2 r_{j+1} \quad &\leq \quad \sum_i r_j^{-1/2} (4/3)^j 16 r_j^{1/3} \text{ by (6.8)} \\
&= \quad \sum_i r_j^{-1/6} (4/3)^j \\
&= \quad \sum_i 1.78^{-j} (4/3)^j \text{ by Lemma 6.8.2}
\end{aligned}$$

which is bounded by a constant c.

We also use the fact that $\sum_{k=0}^{i+1} \log r_k \leq c' \log r_{i+1}$ for a constant c'.

Next we bound the type-1 debt (6.5).

$$
\begin{aligned}
&\sum_i c_1 c_2 \frac{m}{r_i}\sqrt{r_i}\beta_{i0}\sum_{k=0}^{i+1}\log r_k \\
&\quad\le\ c_1 c_2 m \sum_i r_i^{-1/2}\sqrt{r_i}\beta_{i0} c' \log r_{i+1} \\
&\quad=\ \sum_i c_1 c_2 c' m r_i^{-1/2}\sqrt{r_i}(4/3)^i \frac{\log r_{i+1}}{\log r_1}\log r_{i+1} \text{ by definition of } \beta_{i0} \\
&\quad=\ \frac{c_1 c_2 c' m}{\log r_1}\sum_i \frac{1}{\sqrt{r_i}}(4/3)^i \log^2 r_{i+1} \\
&\quad=\ \frac{c_1 c_2 c' m}{\log r_1} c
\end{aligned}
$$

Now we bound the type-2 debt (6.6).

$$
\begin{aligned}
&\sum_j c_1 \frac{m}{\sqrt{r_j}} 2 \sum_{i<j} c_2 \sqrt{r_i}\beta_{i0}\sum_{k=0}^{j+1}\log r_k \\
&\quad\le\ \sum_j c_1 \frac{m}{\sqrt{r_j}} 2 \sum_{i<j} c_2 \sqrt{r_i}\beta_{i0}\ c' \log r_{j+1} \\
&\quad=\ 2c_1 c_2 c' m \sum_j r_j^{-1/2}\log r_{j+1}\sum_{i<j}\sqrt{r_i}\ (4/3)^i \frac{\log r_{i+1}}{\log r_1} \text{ by definition of } \beta_{i0} \\
&\quad\le\ \frac{2c_1 c_2 c' m}{\log r_1}\sum_j r_j^{-1/2}\log r_{j+1}\ c''\ r_{j-1}^{1/2}(4/3)^{j-1}\log r_j \text{ for a constant } c'' \\
&\quad\le\ \frac{2c_1 c_2 c' c'' m}{\log r_1}\sum_j r_j^{-1/2}\log r_{j+1}\ r_{j-1}^{1/2} r_{j-1}^{1/6}(4 r_{j-1}^{1/6}) \text{ by (6.7) and Lemma 6.8.2} \\
&\quad\le\ \frac{8c_1 c_2 c' c'' m}{\log r_1}\sum_j r_j^{-1/2}\log r_{j+1}\ r_{j-1} \\
&\quad\le\ \frac{8c_1 c_2 c' c'' m}{\log r_1}\sum_j r_j^{-1/2}\log^2 r_{j+1} \\
&\quad\le\ \frac{8c_1 c_2 c' c'' m}{\log r_1} c
\end{aligned}
$$

□

Theorem 6.8.5. *The shortest-path algorithm runs in $O(m)$ time.*

6.9 History

Frederickson [Frederickson, 1987] gave the first shortest-path algorithm for planar graphs that is faster than the one for general graphs. His algorithm runs in

$O(n\sqrt{\log n})$ time. His algorithm used an r-division (a concept he pioneered) to ensure that most priority-queue operations involved small queues.

Building on these ideas, Henzinger, Klein, Rao, and Subramanian [Henzinger et al., 1997] gave the linear-time algorithm presented here.

Chapter 7

Multiple-source shortest paths

7.1 Slack costs, relaxed and tense darts, and consistent price vectors

7.1.1 Slack costs

Recall that, for a graph G, we use A_G to denote the dart-vertex incidence matrix.

Let $\boldsymbol{\rho}$ be a vertex vector. The *slack cost vector* with respect to $\boldsymbol{\rho}$ is the vector $\boldsymbol{c_\rho} = \boldsymbol{c} + A_G\boldsymbol{\rho}$. That is, the *slack cost of dart d* with respect to $\boldsymbol{\rho}$ is

$$\boldsymbol{c_\rho}[d] = \boldsymbol{c}[d] + \boldsymbol{\rho}[\text{tail}(d)] - \boldsymbol{\rho}[\text{head}(d)]$$

In this context, we call $\boldsymbol{\rho}$ a *price vector*.

By a telescoping sum, we obtain

Lemma 7.1.1. *For any path P, $\boldsymbol{c_\rho}(P) = \boldsymbol{c}(P) + \boldsymbol{\rho}[start(P)] - \boldsymbol{\rho}[end(P)]$.*

Corollary 7.1.2 (Slack costs preserve optimality)**.** *For fixed vertices s and t, an s-to-t path is shortest with respect to $\boldsymbol{c_\rho}$ iff it is shortest with respect to $\boldsymbol{c}$.*

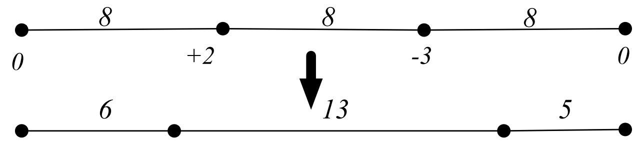


Figure 7.1: This figure shows how a price vector changes lengths of a path. Assume for simplicity that the path's start and end each have price zero. Then the length of the path does not change, despite the changes in the lengths of the darts.

Corollary 7.1.2 is very useful because it alllows us to transform a shortest-path instance with respect to one cost vector, $\boldsymbol{c}$, into a shortest-path instance with respect to another, $\boldsymbol{c}_{\boldsymbol{\rho}}$. This transformation is especially useful if the new costs are nonnegative.

7.1.2 Relaxed and tense darts

Recall from Section 6.1 that a dart d is *relaxed* with respect to $\boldsymbol{c}$ and $\boldsymbol{\rho}$ if

$$\boldsymbol{\rho}[\text{tail}(d)] + \boldsymbol{c}[d] - \boldsymbol{\rho}[\text{head}(d)] \geq 0 \tag{7.1}$$

and *tense* otherwise. Note that the quantity on the left-hand side is the slack cost of d with respect to $\boldsymbol{\rho}$. Thus a dart is relaxed if its slack cost is nonnegative and tense if its slack cost is negative.

A dart d is *tight* if Inequality 7.1 holds with equality, i.e. if its slack cost is zero.

7.1.3 *Consistent* price vectors

A price vector is *consistent* with respect to $\boldsymbol{c}$ if the slack costs of all darts are nonnegative. That is, $\boldsymbol{\rho}$ is a consistent price vector if every dart is relaxed with respect to $\boldsymbol{\rho}$.

Following up on the discussion in Section 7.1.1, a consistent price vector allows us to transform a shortest-path instance with respect to costs some of which are negative into a shortest-path instance in which all costs are nonnegative.

Now we discuss a way to obtain a consistent price vector. For a vertex r, we say that a price vector $\boldsymbol{\rho}$ is the *from-r distance vector* with respect to $\boldsymbol{c}$ if, for every vertex v, $\boldsymbol{\rho}[v]$ is the minimum cost with respect to $\boldsymbol{c}$ of a r-to-v path of darts.

Lemma 7.1.3. *Suppose $\boldsymbol{\rho}$ is the from-r distance vector with respect to $\boldsymbol{c}$ for some vertex r. Then $\boldsymbol{\rho}$ is a consistent price function, and every minimum-cost path starting at r consists of darts that are tight with respect to $\boldsymbol{\rho}$.*

Problem 7.1.4. *Prove Lemma 7.1.3.*

A from-r distance vector is not just any consistent price function; it is maximum in a sense.

Lemma 7.1.5. *Suppose $\boldsymbol{\rho}$ is the from-r distance vector with respect to $\boldsymbol{c}$ for some vertex r. For each vertex v,*

$$\boldsymbol{\rho}[v] = \max\{\boldsymbol{\gamma}[v] : \boldsymbol{\gamma} \text{ is a consistent price function such that } \boldsymbol{\gamma}[r] = 0\} \tag{7.2}$$

Proof. Let T be an r-rooted shortest-path tree. The proof is by induction on the number of darts in $T[v]$. The case $v = r$ is trivial. For the induction step, let v be a vertex other than r, and let d be the parent dart of v in T, i.e. head$(d) = v$. Let $u = \text{tail}(d)$. Let $\boldsymbol{\gamma}$ be the vertex vector attaining the maximum in 7.2.

By the inductive hypothesis,

$$\boldsymbol{\rho}[u] \geq \boldsymbol{\gamma}[u] \tag{7.3}$$

Since $\boldsymbol{\gamma}$ is a consistent price function,

$$\boldsymbol{\gamma}[v] \leq \boldsymbol{\gamma}[u] + \boldsymbol{c}[d] \tag{7.4}$$

By Lemma 7.1.3, $\boldsymbol{\rho}$ is itself a consistent price function, so $\boldsymbol{\gamma}[v] \geq \boldsymbol{\rho}[v]$, and d is tight with respect to $\boldsymbol{\rho}$, so

$$\boldsymbol{\rho}[v] = \boldsymbol{\rho}[u] + \boldsymbol{c}[d] \tag{7.5}$$

By Equations 7.3, 7.4, and 7.5, we infer $\boldsymbol{\rho}[v] \geq \boldsymbol{\gamma}[u]$. We have proved $\boldsymbol{\rho}[v] = \boldsymbol{\gamma}[v]$. □

One motivation for finding a consistent price vector is to transform a shortest-path instance into a simpler shortest-path instance; however, such a transformation seems useless if, as suggested by Lemma 7.1.3, carrying it out requires that we first solve the original shortest-path instance! Nevertheless, the transformation can be quite useful:

- Having distances from one vertex r simplify the computation of distances from other vertices (e.g. the all-pairs-distances algorithm of [**?**]).
- An algorithm can maintain a price function and preserve its consistency over many iterations (e.g. the min-cost flow algorithm of [**?**]).

This lemma suggests a way of using a price vector to certify that a tree is a shortest-path tree.

Lemma 7.1.6. *Let $\boldsymbol{\rho}$ be a price vector that is consistent with respect to $\mathbf{c}$. If T is a rooted spanning tree every dart of which is tight then T is a shortest-path tree with respect to $\mathbf{c}$.*

Proof. With respect to the slack costs, every dart has nonnegative cost, and every path in T has zero cost, so every path in T is a shortest path with respect to $\boldsymbol{c}_{\boldsymbol{\rho}}$ and hence (by Corollary 7.1.2) with respect to $\boldsymbol{c}$. □

Lemma 7.1.3 shows that distances form a consistent price function. However, distances do not exist if there are negative-cost cycles. The following lemma states that, in this case, consistent price functions also do not exist.

Lemma 7.1.7. *If $\boldsymbol{\rho}$ is a consistent price vector for G with respect to $\mathbf{c}$ then G contains no negative-cost cycles.*

Proof. By a telescoping sum, the slack cost of any cycle C equals its original cost. If $\boldsymbol{\rho}$ is a consistent price vector then every dart of d has nonnegative slack cost, so C has nonnegative slack cost and hence nonnegative cost. □

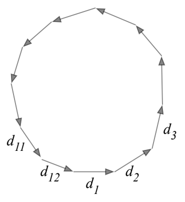
7.2 Specification of multiple-source shortest paths

In this chapter, we study a problem called the *multiple-source shortest paths* (MSSP).

- *input:* a directed planar embedded graph G with a designated infinite face f_∞, a vector $\boldsymbol{c}$ assigning lengths to darts, and a shortest-path tree T_0 rooted at one of the vertices of f_∞.
- *output:* a representation of the shortest-path trees rooted at the vertices on the boundary of f_∞.

Our goal is to give an algorithm that solves this problem in $O(n \log n)$ time where n is the number of vertices (assuming no parallel edges). There is an obvious obstacle: each shortest-path tree consists of $n-1$ edges, so explicitly outputting one for each of the possibly many vertices on the boundary of f_∞ could take far more than $\Omega(n \log n)$ time.

To resolve this difficulty, we use a simple implicit representation of the shortest-path trees. Let $d_1 \cdots d_k$ be the cycle of darts forming the boundary of f_∞



where tail(d_1) is the root of the given shortest-path tree T_0. For $i = 1, \ldots, k$, let T_i be the shortest-path tree rooted at head(d_i). The algorithm will describe the changes required to transform T_0 into T_1, the changes needed to transform T_1 into T_2, …, and the changes needed to transform T_{k-1} into T_k. We will show that the total number of changes is at most the number of finite-length darts of G.

7.2.1 Pivots

The basic unit of change in a rooted tree T, called a *pivot*, consists of ejecting one dart d_- and inserting another dart d_+ so that the result is again a rooted tree. A pivot is specified by the pair (d_-, d_+) of darts.[1]

Transforming T from T_{i-1} to T_i consists of

- a *special* pivot that ejects the dart whose head is head(d_i), and inserts the dart rev(d_i) (now T is head(d_i)-rooted), and
- a sequence of *ordinary* pivots each of which ejects a dart d' and inserts a dart $\hat{d}$ with the same head.

[1]The term *pivot* comes from an analogy to the network-simplex algorithm.

For $i = 1, \ldots, k$, the MSSP algorithm outputs the sequence of pivots that transform T_{i-1} into T_i. We shall show that the time per pivot is $O(\log n)$. The number of special pivots is k. In the next section, we show that the number of ordinary pivots is at most the number of finite-length darts. It follows that the running time is $O(n \log n)$.

7.3 Contiguity property of shortest-path trees in planar graphs

Let G be a graph, let $\boldsymbol{c}$ be a dart vector, and let d be a dart. For a vertex u, we say d is *u-tight with respect to $\boldsymbol{c}$* if d is tight with respect to the from-u distance vector with respect to $\boldsymbol{c}$.

Lemma 7.3.1. *There is a u-rooted shortest-path tree containing d iff d is u-tight.*

Problem 7.3.2. *Prove Lemma 7.3.1 using Lemma 7.1.6.*

The following lemma is the key to the analysis of the MSSP algorithm and the single-source max-flow algorithm presented in Chapter 9. It is illustrated in Figures 7.2 and 7.3,

Lemma 7.3.3. *Let G be a planar embedded graph with infinite face f_∞, and let $\boldsymbol{c}$ be a dart vector. Let r_1, r_2, r_3, r_4 be vertices on f_∞ in order. For any dart d, if d is r_1-tight and r_3-tight then it is r_2-tight or r_4-tight.*

Corollary 7.3.4 (Consecutive-Roots Corollary)**.** *Let G be a planar embedded graph with infinite face f_∞, and let $\boldsymbol{c}$ be a dart vector. Let $(d_1\ d_2\ \cdots)$ be the cycle of darts forming the boundary of f_∞.*

For each dart d, the set

$$\{i : d \text{ is } head(d_i)\text{-tight}\} \tag{7.6}$$

forms a consecutive subsequence of the cycle $(1\ 2\ \cdots)$.

Corollary 7.3.5. *The number of pivots required to transform shortest-path tree T_{i-1} into T_i, summed over all i, is at most the number of finite-cost darts.*

7.4 Using the output of the MSSP algorithm

There are several ways to use the output of the MSSP algorithm.

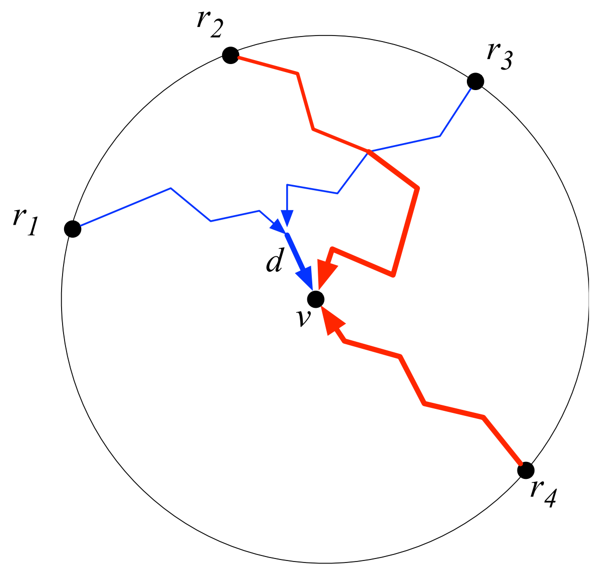


Figure 7.2: If the shortest r_1-to-v and r_3-to-v paths use d but the shortest r_2-to-v and r_4-to-v paths do not, one of the latter must cross one of the former.

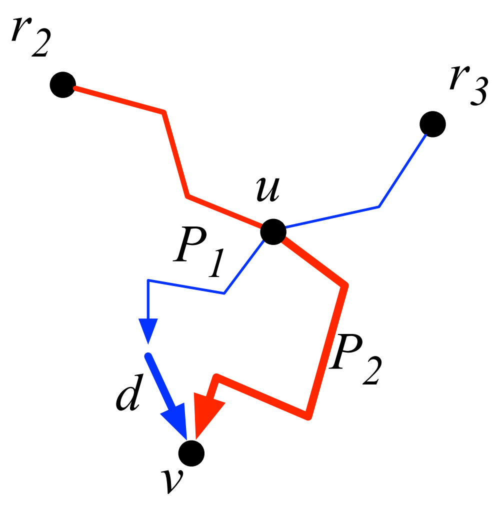


Figure 7.3: Let u denote a vertex at which they cross. Since $P_1 \circ d$ is a shortest u-to-v path, it is no longer than P_2. Replacing P_2 with $P_1 \circ d$ in the shortest r_2-to-v path shows that d is r_2v-tight, a contradiction.

7.4.1 Paths

We can use the output to build a data structure that supports queries of the form ShortestPath(head(d_i), v) where d_i is a dart of the boundary of f_∞ and v is an arbitrary vertex.

Fix a vertex v, and let $b_1, \ldots, b_{\text{degree}(v)}$ be the darts entering v. For each dart d_i on the boundary of f_∞, define

$$g(i) = \min\{j : b_j \text{ is head}(d_i)\text{-tight}\}$$

Note that $b_{g(i)}$ is the last dart in a shortest head(d_i)-to-v path.

Corollary 7.3.4 implies that, for each entering dart b_j, the set $\{i : g(v, i) = j\}$ forms a consecutive subsequence of $(1\ 2\ \cdots)$. For different entering darts, the subsequences are disjoint. The data structure of [**?**] enables one to Using, e.g., the data structure of [**?**], one can represent $g(\cdot)$ in $O(\text{degree}(v))$ space so that computing $g(i)$ takes $O(\log \log k)$ time where k is the number of darts on the boundary of f_∞. Overall all vertices v, the space required is linear in the size of the graph. Given these data structures, a query ShortestPath(head(d_i), v) can be answered iteratively by constructing the path backwards from v to head(d_i), one dart per iteration. The time for each iteration is $O(\log \log k)$, so constructing a shortest-path consisting of ℓ edges takes time $O(\ell \log \log n)$.

7.4.2 Distances

We describe a Distances algorithm that processes the output of the MSSP algorithm to answer a given set of q queries of the form distance(tail(d_i), v) in time $O((n + q) \log n)$. The Distances algorithm maintains a link-cut tree representation of T as T goes from T_0 to T_1 to ... to T_k. The link-cut tree assigns weights to the vertices. The Distances algorithm ensures that the weight of v is the length of the root-to-v path in T. For $i = 0, 1, \ldots, k - 1$, when T is the tail(d_i)-rooted shortest-path tree, the Distances algorithm queries the link-cut tree to find the weights of those vertices v for which the tail(d_i)-to-v distance is desired. The time per pivot and per query is $O(\log n)$.

Now we give the algorithm more formally. We represent the evolving shortest-path tree T using a link-cut-tree data structure that supports AddToDescendants and GetWeight. We maintain the invariant that the weight assigned in T to each vertex v is the length of the root-to-v path in T.

```
def DISTANCES($d_1 \cdots d_k, T_0, \mathcal{P}, \mathcal{Q}$):
  pre: $d_1 \cdots d_k$ are the darts of $f_\infty$,
       $T_0$ is the tail($d_1$)-rooted shortest-path tree,
       $\mathcal{P}$ is the sequence of pivots $(d', \hat{d})$ produced by the MSSP algorithm
       $\mathcal{Q}$ is an array such that $\mathcal{Q}[i]$ is a set of vertices
  post: returns the set $\mathcal{D} = \{(i, v, dist(\text{tail}(d_i), v)) : v \in Q[i]\}$ of distances
1 initialize $T$ :=link-cut tree representing $T_0$ with weight($v$) := $\boldsymbol{c}(T_0[v])$)
2 initialize $\mathcal{D} = \emptyset$, $i := 0$
3 for each pivot $(d', \hat{d})$ in $\mathcal{P}$,
4   if $(d', \hat{d})$ is a special pivot,// about to transform to next tree...
5     $i := i + 1$
      // ... but first find distances in current tree
6     for each $v \in Q[i]$,
7       append $(i, v, T.\text{GETWEIGHT}(v))$ to $\mathcal{D}$
    // perform pivot
8   $T$.CUT(tail($d'$)) // remove $d'$ from $T$
9   $T$.ADDTODESCENDANTS(head($d'$), $-T$.GETWEIGHT(head($d$))
10  $T$.LINK(tail($\hat{d}$), head($\hat{d}$)) // add $\hat{d}$ to $T$
11  $T$.ADDTODESCENDANTS(head($d'$), $\boldsymbol{c}[\hat{d}] + T$.GETWEIGHT(tail($d$))
12return $\mathcal{Q}$
```

In Line 4, if the next pivot in the sequence is a special pivot, it means that the current tree is a shortest-path tree, so now is the time to find distances from the current root. The algorithm maintains the invariant that the weight of each vertex v in T is the length of the root-to-v path in T, so in Line 7 the distance to v is the weight of v.

Lines 8-11 carry out a pivot. Line 8 ejects d', breaking the tree into two trees, one with the same root as before and one rooted at head(d'). Line 9 updates the weights of vertices in the latter tree to preserve the invariant. Line 10 inserts $\hat{d}$, forming a single tree once again. Line 11 again updates the weights of the vertices that were in the latter tree to preserve the invariant.

7.5 The abstract MSSP algorithm

We present an abstract description of an algorithm for MSSP. Later we will present a more detailed description.

```
def MSSP($G, f_\infty, T$):
  pre: $T$ is a shortest-path tree rooted at a vertex of $f_\infty$
  let $(d_1\ d_2\ \cdots\ d_s)$ be the darts of $f_\infty$, where tail$_G(d_1)$ is the root of $T$
  for $i := 1, 2, \ldots, s$,
    // $T$ is a tail($d_i$)-rooted shortest-path tree
    $(A_i, B_i)$ :=CHANGEROOT($T, d_i$)  // transform $T$ to a head($d_i$)-rooted
```

```
            // shortest-path tree by removing dart-set $A_i$ and adding $B_i$
    // the darts of $A_i$ are not head($d_i$)-tight
return $(A_1, \ldots, A_s)$ and $(B_1, \ldots, B_s)$
```

7.5.1 Analysis of the abstract algorithm

Lemma 7.5.1. *For each dart d, there is at most one iteration i such that $d \in A_i$.*

Proof. By the Consecutive-Roots Corollary (Corollary 7.3.4), there is at most one occurence of a vertex r on f_∞ such that d is r-tight but is not r'-tight where r' is the next vertex in consecutive order on f_∞. □

Corollary 7.5.2. $\sum_i |A_i|$ *is at most the number of darts.*

7.6 ChangeRoot: the inner loop of the MSSP algorithm

The procedure ChangeRoot(T, d_i) called in MSSP consists of a loop each iteration of which selects a dart to add to T and a dart to remove from T. We show later that setting up the loop takes $O(\log n)$ time, the number of iterations is $|A_i|$, and each iteration takes amortized $O(\log n)$ time. The time for ChangeRoot is therefore $O((|A_i|+1)\log n)$. Summing over all i and using Corollary 7.5.2, we infer that the total time for ChangeRoot and therefore for MSSP is $O(n \log n)$.

In each iteration, the algorithm determines that a dart $\hat{d}$ not in T must be added to T. It then *pivots d into T*, which means

- removing from T the dart whose head is head(d) and
- adding d to T.

This operation is called a *pivot* because it resembles a step of the network-simplex algorithm.

Conceptually, ChangeRoot is as follows. Let c_0 denote the tail(d_i)-to-head(d_i) distance in G. To initialize, ChangeRoot temporarily sets the cost of the dart rev(d_i) to $-c_0$, removes from T the dart whose head is tail(rev(d_i)), and inserts rev(d_i) into T. It then gradually increases the cost of rev(d_i) to its original cost while performing pivots as necessary to maintaing that T is a shortest-path tree.

After the initialization, the procedure uses a variable t to represent the current modified cost of rev(d_i). The costs of other darts remain unmodified. Let $\boldsymbol{c}_t$ denote the vector of costs. That is,

$$\boldsymbol{c}_t[d] = \begin{cases} t & \text{if } d = \text{rev}(d_i) \\ \boldsymbol{c}[d] & \text{otherwise} \end{cases}$$

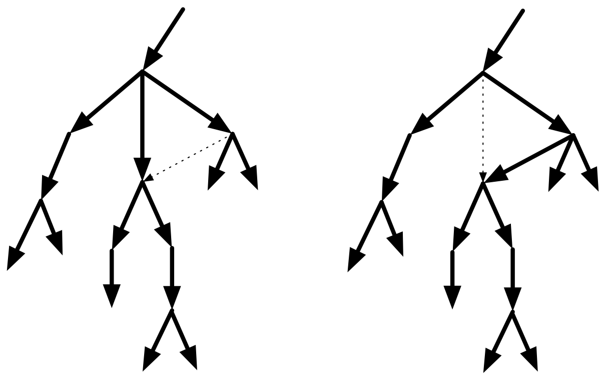


Figure 7.4: The two trees differ by a single pivot.

Under these costs, the cost of the path in T to a vertex v is denoted $\boldsymbol{\rho}_{T,t}[v]$. The vector $\boldsymbol{\rho}_{T,t}$ is not represented explictly by the algorithm; we use it in the analysis and proof of correctness.

Lemma 7.6.1. *There is no negative-cost cycle in G with respect to the costs $\boldsymbol{c}_t$.*

Proof. Let C be any simple cycle of darts in G. If C does not contain $\text{rev}(d_i)$ then $\boldsymbol{c}_t(C) = \boldsymbol{c}(C) \geq 0$. Otherwise, write $C = \text{rev}(d_i) \circ P$. Then

$$\begin{aligned} \boldsymbol{c}_t(C) &= \boldsymbol{c}_t(\text{rev}(d_i)) + \boldsymbol{c}_t(P) \\ &\geq -c_0 + c_0 \end{aligned}$$

□

After initialization, the procedure ChangeRoot repeats the following two steps until t equals the original cost of $\text{rev}(d_i)$:

- Increase t until any further increase would result in some dart d becoming tense with respect to $\boldsymbol{c}_t$ and $\boldsymbol{\rho}_{T,t}$.
- Pivot d into T.

Lemma 7.6.2. *Throughout the execution, T is a shortest-path tree with respect to the costs $\boldsymbol{c}_t$.*

Proof. We prove the lemma by induction. For the basis of the induction, we must prove that T is a shortest-path tree immediately after $\text{rev}(d_i)$ is pivoted in.

Define $\boldsymbol{\rho}$ to be the from-tail(d_i) distance vector with respect to $\boldsymbol{c}$. At the very beginning of ChangeRoot, before $\text{rev}(d_i)$ is pivoted in, T is a shortest-path tree, so by Lemma 7.1.3 its darts are tight with respect to $\boldsymbol{\rho}$ and $\boldsymbol{c}$. Since $\boldsymbol{c}_t$ is identical to $\boldsymbol{c}$ on these darts, they are also tight with respect to $\boldsymbol{c}_t$.

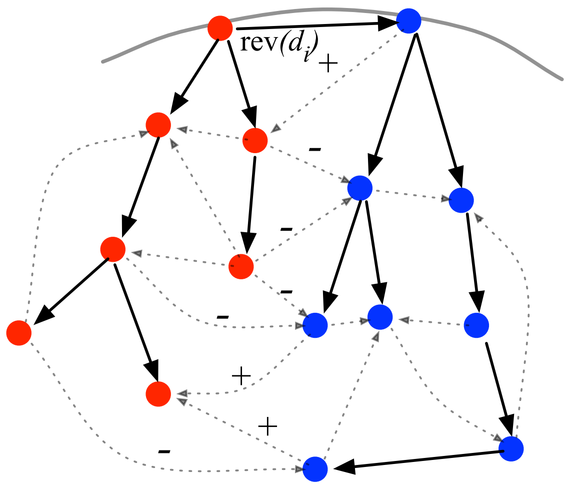


Figure 7.5: This figure shows the coloring of vertices. Those vertices reached through rev(d_i) are blue and the others are red. The red-to-blue nontree arcs are labeled with - to indicate that their slack costs decrease as the algorithm progresses. The blue-to-red nontree arcs are labled with + to indicate that their slack costs increase. The red-to-red and blue-to-blue arcs remain unchanged.

By Lemma 7.1.6, it remains only to show that rev(d_i) itself is tight when it is first pivoted in. At this time, its cost $\boldsymbol{c}_t$[rev(d_i)] is

$$-c_0 = -(\boldsymbol{\rho}[\text{tail}(\text{rev}(d_i))] - \boldsymbol{\rho}[\text{head}(\text{rev}(d_i))])$$

so its slack cost is zero. This completes the basis of the induction.

Now for the induction step. We assume T is a shortest-path tree. The variable t is increased until further increase would result in some dart d becoming tense with respect to $\boldsymbol{c}_t$ and $\boldsymbol{\rho}_{T,t}$. At this point, d is tight. Therefore, by Lemma 7.1.6, pivoting d into T yields a shortest-path tree. $\square$

7.7 Which darts are candidates for pivoting in

In this section, we consider the loop of ChangeRoot in which t is increased.

The algorithm pivots in a dart d only if necessary, i.e. if continuing the shrinking or growing phase without pivoting in d would result in T not being a shortest-path tree, in particular if d would become tense with respect to $\boldsymbol{c}$ and $\boldsymbol{\rho}_{T,t}$. A dart d is in danger of becoming tense only if its slack cost with respect to $\boldsymbol{\rho}_{T,t}$ is decreasing.

We define a labeling of the vertices with colors. For each vertex v, if the root-to-v path contains rev(d_i) then we say v is *blue*, and otherwise we say v is *red*.

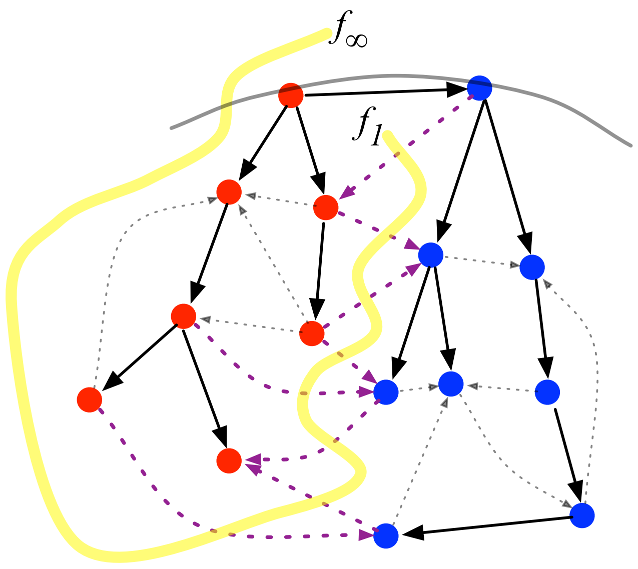


Figure 7.6: This figure illustrates that the edges whose slack costs are changing form a cycle, the fundamental cycle of $\text{rev}(d_i)$. The duals of darts on the f_1-to-f_∞ path decrease in slack cost, and their reverses increase in slack cost.

Lemma 7.7.1. *Suppose that t increases by Δ. For a dart dart d not in T, the slack cost of d*

- *decreases if $tail_G(d)$ is red and $head_G(d)$ is blue,*
- *increases if $tail_G(d)$ is blue and $head_G(d)$ is red, and*
- *otherwise does not change.*

Suppose $\text{rev}(d_i)$ is in T. Then the fundamental cut of $\text{rev}(d_i)$ with respect to T has the form $\vec{\delta}_G(S)$ where S is the set of red vertices. When the cost of $\text{rev}(d_i)$ increases by Δ, by Lemma 7.7.1, the darts of this cut (not including $\text{rev}(d_i)$) undergo a decrease of Δ in their slack costs. By Fundamental-Cut/Fundamental-Cycle Duality (4.6.1), these are the darts belonging to the the fundamental cycle of $\text{rev}(d_i)$ in the dual G^* with respect to T^*.

Since $\text{tail}_{G^*}(\text{rev}(d_i))$ is f_∞, this cycle, minus the dart $\text{rev}(d_i)$ itself, is the path in T^* from $\text{head}_{G^*}(\text{rev}(d_i))$ to f_∞. We summarize this result as follows:

Lemma 7.7.2. *Suppose $rev(d_i)$ is in T and its cost increases by Δ. This results in a decrease by Δ in the slack costs of the darts of the $head_{G^*}(rev(d_i))$-to-f_∞ path in T^*, and a decrease by Δ in the slack costs of the reverse darts.*

7.8 Efficient implementation

The concrete version of MSSP does not explicitly represent the distances $\boldsymbol{\rho}_{T,t}[\cdot]$. Instead, it represents the slack costs of the darts whose edges are not in T. This permits an efficient implementation of ChangeRoot.

7.8.1 ChangeRoot

Now we give the procedure ChangeRoot. (We later describe how it is implemented using data structures.)

```
def ChangeRoot($T, d_i$):
  pre: The root of $T$ is $\text{tail}_G(d_i)$
1 initialize $A_i, B_i := \emptyset$.
2 $t := -\boldsymbol{c}[d_i] +$ slack cost of $d_i$
3 remove from $T$ the dart whose head is $\text{head}_G(d_i)$ and add $\text{rev}(d_i)$
4 comment: now the root of $T$ is $\text{head}(d_i)$
5 repeat
6    let $P$ be the $\text{head}_{G^*}(\text{rev}(d_i))$-to-$f_\infty$ path in $T^*$
7    find a dart $\hat{d}$ in $P$ whose slack cost $\Delta$ is minimum
8    if $t + \Delta > \boldsymbol{c}[\text{rev}(d_i)]$ then return $(A_i, B_i)$
9    subtract $\Delta$ from from the slack costs of darts in $P$
10   add $\Delta$ to $t$ and to the slack costs of reverses of darts $P$
11   remove from $T$ the dart whose head is $\text{head}_G(\hat{d})$, and add it to $A_i$
12   add $\hat{d}$ to $T$ and to $B_i$
13   if $T$ no longer contains $\text{rev}(d_i)$, return $(A_i, B_i)$
```

The correctness of the analysis in Section 7.5.1 depends on the following lemma.

Lemma 7.8.1. *Each dart added to A_i is not $\text{head}_G(d_i)$-tight.*

Proof. Conside an iteration of the repeat-loop. Since the dart $\hat{d}$ selected in Line 7 is in P, its head is blue and its tail is red. Let $\tilde{d}$ be the dart that in Line 11 is removed from T and added to A_i. Consider the vertex-coloring before Line 11 is executed. Since $\hat{d}$ and $\tilde{d}$ share heads, the head of $\tilde{d}$ is blue. Since $\tilde{d}$ is in T, and no dart other than $\text{rev}(d_i)$ is in both T and the fundamental cycle of $\text{rev}(d_i)$ with respect to T^*, it follows that the the tail of $\tilde{d}$ is also blue.

Consider the moment just after Lines 11 and 12 are executed. Since $\hat{d}$ has been added to T, its head has become red. □

7.8.2 Data structure

To support efficient implementation, MSSP represents T in two ways. It represents T directly via a table, and represents T indirectly by representing the interdigitating tree T^* (rooted at f_∞) by a link-cut tree.

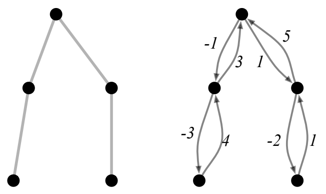


Figure 7.7: The left figure shows a tree. The right figure indicates that the tree must be represented by a data structure in such a way that each edge has two weights, the slack cost of the rootward dart and the slack cost of the leafward dart. The operation AddToAncestor(v, Δ) operates on the edges of the v-to-root path. For each edge, the operation subtracts Δ from the rootward dart and adds Δ to the leafward dart.

The table parentD[$\cdot$] stores, for each vertex v that is not the root of T, the dart of T whose head is v (the parent dart of v).

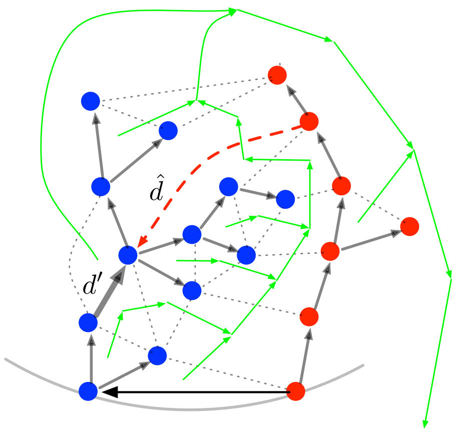
The link-cut tree representing T^* has a node for each vertex of G^* and a node for each edge of T^*. The edge-nodes are assigned pairs of weights; the weights associated with edge e are $(w_R(e), w_L(e))$, where w_R is the slack cost of the dart of e oriented towards the root f_∞ and w_L is the slack cost of the dart of e oriented away from the root.

In each iteration, in Line 7 the algorithm uses a AncestorFindMin operation on the link-cut tree representing T^* to select the dart $\hat{d}$ to insert into T and obtain the dart's slack cost Δ. In Line 11 the algorithm uses the table parentD[$\cdot$] to find which dart must therefore be removed from T. In Lines 9 and 10, the algorithm updates the slack costs along P using the operation AddToAncestors$(\text{head}_{G^*}(\text{rev}(d_i)), \Delta)$.

The topological changes to T in Lines 11 and 12 are carried out by making topological changes to T^* using operations Cut, Evert, and Link. First, the dart $\hat{d}$ being pivoted into T must be removed from T^* by a Cut operation. This breaks T^* into two trees, one rooted at f_∞ and one rooted at $\text{tail}_{G^*}(\hat{d})$. Next, as shown in Figure 7.8, an Evert operation must be performed on the latter tree to reroot it at $\text{head}_{G^*}(d')$ where $d' = \text{parentD}[\text{head}_G(\hat{d})]$ is the dart to be removed from T in Line 11. This reorients the path from $\text{head}_{G^*}(d')$ to $\text{head}_{G^*}(\hat{d})$. Finally, now that $\text{head}_{G^*}(d')$ is the root, this tree is linked to the one rooted at f_∞ by performing Link$(\text{head}_{G^*}(d'), \text{tail}_{G^*}(d'))$.

The link-cut tree must support GetWeight, Evert, AddToAncestor, and AncestorFindMin. Moreover, since eversion changes which dart of an edge e is oriented towards the root, the weights must be handled carefully.

Each iteration of the repeat-loop of Line 5 thus requires a constant number of link-cut-tree operations It follows that the time for iteration i of MSSP requires amortized time $O((1 + |A_i|) \log n)$. By Corollary 7.5.2, $\sum_i (1 + |A_i|)$ is at most the number of darts plus the size of the boundary of f_∞. Since the initialization of the data structures takes linear time, it follows that MSSP requires time $O(n \log n)$.



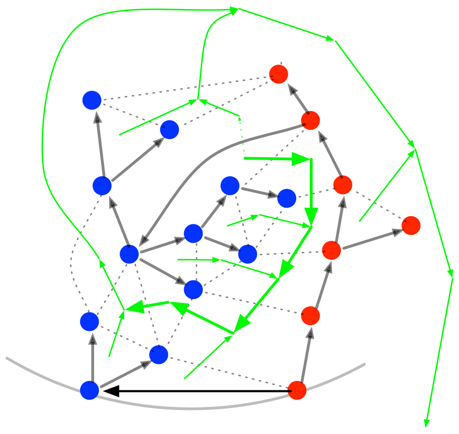


Figure 7.8: This figure shows the need for reversing the direction of a path in the dual tree T^*. The dual tree is shown using rootward darts. The diagram on the left shows the situation just before a pivot. The red dashed edge $\hat{d}$ is about to be pivoted into T, which causes the corresponding dual edge to be removed from T^*. The thick black edge must be removed from T, which causes the corresponding dual edge to be added to T^*. The figure on the right shows the resulting situation. Note that the thick edges in the dual tree have reversed direction.