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 $\boldsymbol{c}$.*
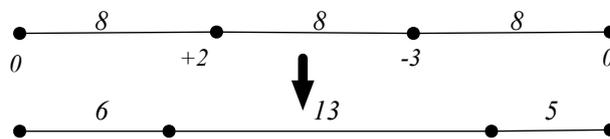

Figure 7.1: This figure shows how a price vector changes lengths of a path. Assume for simplicity that the path's start and end each have price zero. Then the length of the path does not change, despite the changes in the lengths of the darts.

Corollary 7.1.2 is very useful because it alllows us to transform a shortest-path instance with respect to one cost vector, c, into a shortest-path instance with respect to another, c_ρ. 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 c and ρ if

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

and *tense* otherwise. Note that the quantity on the left-hand side is the slack cost of d with respect to ρ. 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 c if the slack costs of all darts are nonnegative. That is, ρ is a consistent price vector if every dart is relaxed with respect to ρ.

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 ρ is the *from-r distance vector* with respect to c if, for every vertex v, $\rho[v]$ is the minimum cost with respect to c of a r-to-v path of darts.

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

Problem 7.1.4. *Prove Lemma 7.1.3.*

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 7.1.5. *Let ρ be a price vector that is consistent with respect to c. If T is a rooted spanning tree every dart of which is tight then T is a shortest-path tree with respect to 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 c_ρ and hence (by Corollary 7.1.2) with respect to c. $\qquad\square$

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 7.1.6. *If ρ is a consistent price vector for G with respect to 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 ρ is a consistent price vector then every dart of d has nonnegative slack cost, so C has nonnegative slack cost and hence nonnegative cost. $\qquad\square$

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 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_∞
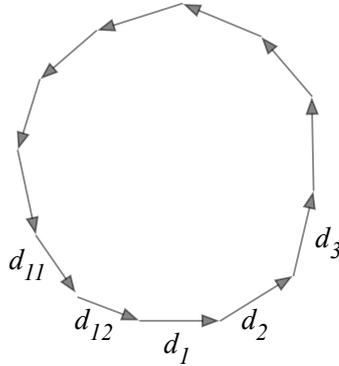
where $\text{tail}(d_1)$ is the root of the given shortest-path tree T_0. For $i = 1, \ldots, k$, let T_i be the shortest-path tree rooted at $\text{head}(d_i)$. We assume that each shortest-path tree T_i includes only finite-length darts. The algorithm will describe the changes required to transform 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 $\text{head}(d_i)$, and inserts the dart $\text{rev}(d_i)$ (now T is $\text{head}(d_i)$-rooted), and

- a sequence of *ordinary* pivots each of which ejects a dart d' and inserts a dart $\hat{d}$ with the same head.

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 amortized 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

In this section we prove a contiguity property for the trees T_i that is crucial in bounding the number of steps of the MSSP algorithm. To facilitate the

---

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

exposition we first consider the non-degenerate case, where shortest paths are unique. We analyze the degenerate case in section **??**. The same arguments are used in the analysis of the single-source max-flow algorithm presented in Chapter 10.

Let G be a graph, let c be a dart vector, and let d be a dart. For a vertex u, we say d is *u-tight with respect to c* if d is tight with respect to the from-u distance vector with respect to 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 7.1.5.*

Note that our unique shortest-paths assumption implies that a dart is u-tight if and only if is in the shortest path tree rooted at u.

Lemma 7.3.3 (Consecutive-Roots Lemma)**.** *Let G be a planar embedded graph with infinite face f_∞, and let 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

$$\{i\ :\ d\ is\ in\ T_i\} \tag{7.2}$$

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

Proof. Let v be a vertex. Let T' be the v-rooted shortest-path tree using darts in the reverse direction. (Formally, we use the costs c' defined by $c'[d] = c[rev(d)]$). For each vertex r on f_∞, if d is the last dart in the shortest r-to-v path, then the first dart in the v-to-r path in T' is $rev(d)$. That is, r is a descendent of $tail(d)$ in T'. Since the tree T' does not cross itself, the set $\{r_i \in f_\infty : r_i \in T'_{\text{tail}(d)}\}$ is a consecutive subsequence of the boundary of f_∞. The lemma follows. $\square$

Corollary 7.3.4. *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-length darts.*

7.4 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∞, T):
    pre: T is a shortest-path tree rooted at a vertex of f∞
    let (d₁ d₂ ··· dₛ) be the darts of f∞, where tail_G(d₁) is the root of T
    for i := 1, 2, . . . , s,
        T is a tail(dᵢ)-rooted shortest-path tree
        (Aᵢ, Bᵢ) :=CHANGEROOT(T, dᵢ)    transform T to a head(dᵢ)-rooted
                shortest-path tree by removing dart-set Aᵢ and adding Bᵢ
```

> *the darts of A_i are* not *head(d_i)-tight*
> return $(A_1, \ldots, A_s)$ and $(B_1, \ldots, B_s)$

The algorithm calls the subroutine CHANGEROOT, for each dart d_i of f_∞ in order. CHANGEROOT transforms T_{i-1} into T_i. It returns two sets of darts; the set A_i of darts in T_{i-1} but not in T_i, and the set B_i of darts in T_i but not in T_{i-1}.

7.4.1   Analysis of the abstract algorithm

Lemma 7.4.1. *For each dart d, there is at most one iteration i such that $d \in A_i$.*

Proof. By the Consecutive-Roots Lemma (Corollary 7.3.3), there is at most one integer i such that d is in T_i but is not in T_{i+1}.   □

Corollary 7.4.2. $\sum_i |A_i|$ *is at most the number of darts.*

7.5   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 7.4.2, 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 $\hat{d}$ into T*, which means

- removing from T the dart whose head is head($\hat{d}$) and

- adding $\hat{d}$ 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 head(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 maintain 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.
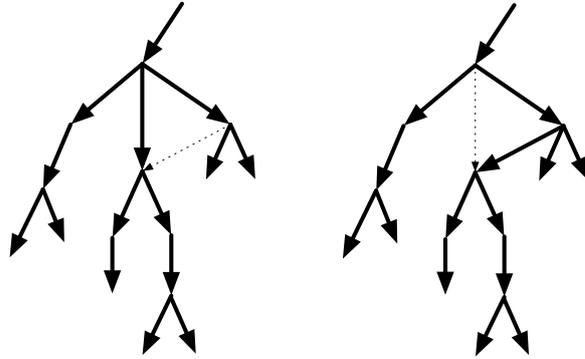
Figure 7.2: The two trees differ by a single pivot.

Let $\boldsymbol{c}_t$ denote the vector of costs. That is,

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

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.

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 7.5.1. *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 $\boldsymbol{c}_t$.

By Lemma 7.1.5, it remains only to show that $\text{rev}(d_i)$ itself is tight when it is first pivoted in. At this time, its cost $\boldsymbol{c}_t[\text{rev}(d_i)]$ 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 c_t and $\rho_{T,t}$. At this point, d is tight. Therefore, by Lemma 7.1.5, pivoting d into T yields a shortest-path tree. $\qquad\square$

7.6 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 c and $\rho_{T,t}$. A dart d is in danger of becoming tense only if its slack cost with respect to $\rho_{T,t}$ is decreasing.

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

Lemma 7.6.1. *Suppose that t increases by Δ. For a dart d not in T, the slack cost of d*

- *decreases if $\text{tail}_G(d)$ is red and $\text{head}_G(d)$ is blue,*

- *increases if $\text{tail}_G(d)$ is blue and $\text{head}_G(d)$ 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 7.6.1, 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 7.6.2. *Suppose $\text{rev}(d_i)$ 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 $\text{head}_{G^*}(\text{rev}(d_i))$-to-$f_\infty$ path in T^*, and a decrease by Δ in the slack costs of the reverse darts.*

7.7 Efficient implementation

The concrete version of MSSP does not explicitly represent the distances $\rho_{T,t}[\cdot]$. Instead, it represents the slack costs of the darts whose edges are not in T. This permits an efficient implementation of CHANGEROOT.
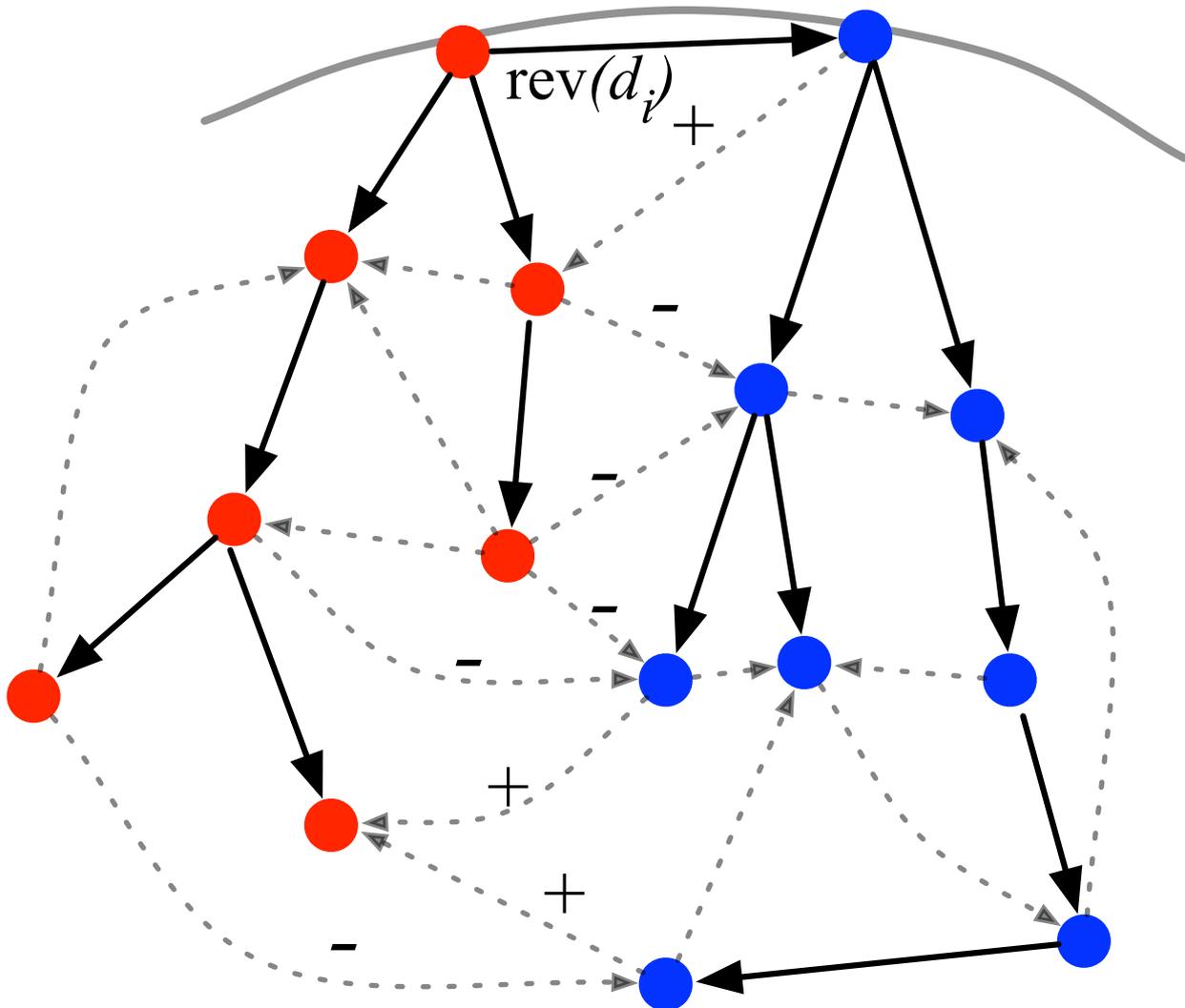
Figure 7.3: This figure shows the coloring of vertices. Those vertices reached through $rev(d_i)$ are blue and the others are red. The red-to-blue nontree arcs are labeled with **-** to indicate that their slack costs decrease as the algorithm progresses. The blue-to-red nontree arcs are labeled with **+** to indicate that their slack costs increase. The red-to-red and blue-to-blue arcs remain unchanged.
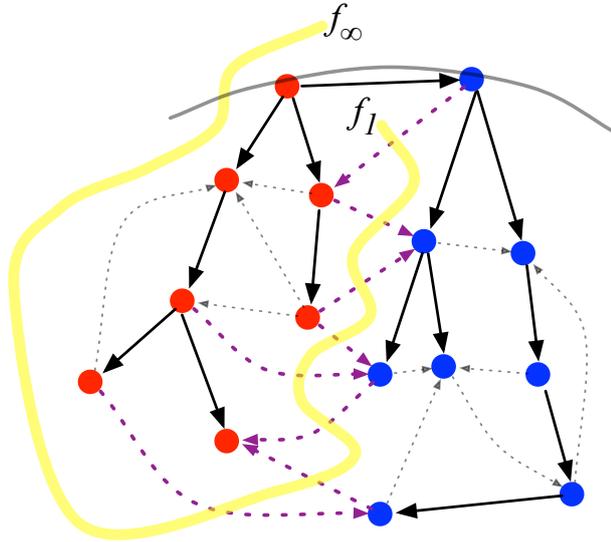
Figure 7.4: This figure illustrates that the edges whose slack costs are changing form a cycle, the fundamental cycle of $\text{rev}(d_i)$. The duals of darts on the f_1-to-f_∞ path decrease in slack cost, and their reverses increase in slack cost.

7.7.1   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 tail_G(d_i)
1    initialize A_i, B_i := ∅.
2    t := −c[d_i] + slack cost of d_i
3    remove from T the dart whose head is head_G(d_i) and add rev(d_i)
4    comment: now the root of T is head(d_i)
5    repeat
6       let P be the head_{G*}(rev(d_i))-to-f_∞ path in T*
7       find a dart d̂ in P whose slack cost s is minimum
8       Δ = min{s, c[rev(d_i)] − t}
9       subtract Δ from the slack costs of darts in P
10      add Δ to t and to the slack costs of reverses of darts in P
12      if Δ < c[rev(d_i)] − t
11         remove from T the dart whose head is head_G(d̂), and add it to A_i
12         add d̂ to T and to B_i
13   until t = c[rev(d_i)]
14   return (A_i, B_i)
```
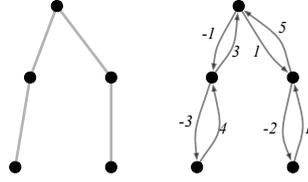
Figure 7.5: The left figure shows a tree. The right figure indicates that the tree must be represented by a data structure in such a way that each edge has two weights, the slack cost of the rootward dart and the slack cost of the leafward dart. The operation $\textsc{AddToAncestor}(v, \Delta)$ operates 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.

7.7.2 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.

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 $\textsc{AncestorFindMin}$ 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 $\textsc{AddToAncestors}(\text{head}_{G^*}(\text{rev}(d_i)), \Delta)$.

The topological changes to T in Lines 11 and 12 are carried out by making topological changes to T^* using operations $\textsc{Cut}$, $\textsc{Evert}$, and $\textsc{Link}$. First, the dart $\hat{d}$ being pivoted into T must be removed from T^* by a $\textsc{Cut}$ 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 7.6, an $\textsc{Evert}$ 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 $\textsc{Link}(\text{head}_{G^*}(d'), \text{tail}_{G^*}(d'))$.

The link-cut tree must support $\textsc{GetWeight}$, $\textsc{Evert}$, $\textsc{AddToAncestor}$, and $\textsc{AncestorFindMin}$. 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
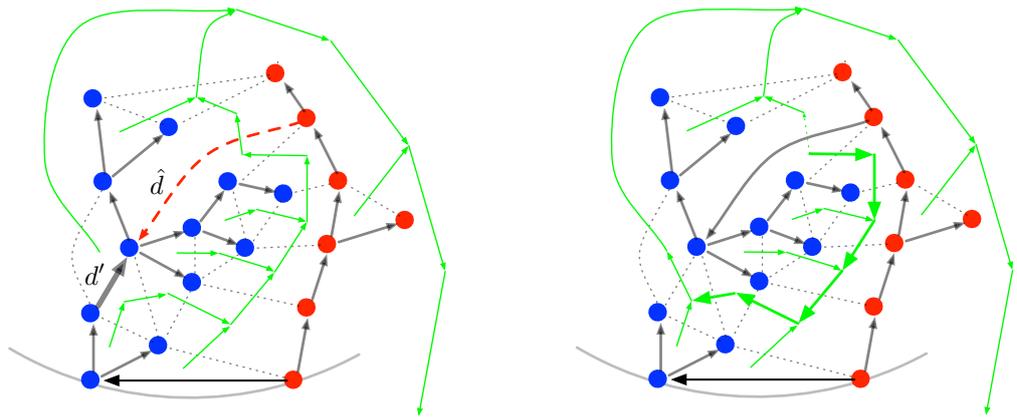
Figure 7.6: This figure shows the need for reversing the direction of a path in the dual tree T^*. The dual tree is shown using rootward darts. The diagram on the left shows the situation just before a pivot. The red dashed edge $\hat{d}$ is about to be pivoted into T, which causes the corresponding 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.

amortized time $O((1 + |A_i|) \log n)$. By Corollary 7.4.2, $\sum_i (1 + |A_i|)$ 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 $O(n \log n)$.

7.8 Number of pivots - the degenerate case

In this section we show that it is not necessary to assume unique shortest paths. Note that we had used this assumption only to bound the number of pivots. For the correctness we only used the fact that the dart $\hat{d}$ chosen in line 7 of CHANGEROOT is a dart with minimum slack cost on the path P, so that T remains a shortest path tree at all times. When shortest paths are not unique, there may be more than one dart with minimum slack cost in P. We show that if $\hat{d}$ is chosen to be the leafmost such dart (i.e., the one farthest from f_∞ on P), then each dart is ejected from T at most once during the entire execution of the algorithm, and hence the number of pivots is bounded by the number of darts in the graph.

Lemma 7.8.1. *Let v be any vertex. For $j = 0, 1, 2, \ldots \ldots$, let P_j^v be the root-to-v path in the shortest-path tree T after j pivots. The paths $P_0^v, P_1^v, \ldots$ are mutually noncrossing.*

Proof. □

Lemma 7.8.2. *Once a dart pivots out of T it does not pivot back in within the same call to* CHANGEROOT.

Proof. Consider an iteration of the repeat-loop. Let $\hat{d}$ be the dart selected in Line 7 to be inserted into T, and let $\tilde{d}$ be the dart that in Line 11 is removed from T. $\hat{d}$ and $\tilde{d}$ have the same head v. Since the dart $\hat{d}$ selected in Line 7 is in P, its head v is blue and its tail is red. Consider the moment just after Lines 11 and 12 are executed. Since $\hat{d}$ has been added to T, v becomes red. Since along the execution of CHANGEROOT vertices only change color from blue to red, no dart whose head is v will be inserted into T in the remainder of the execution of CHANGEROOT. □

Lemma 7.8.3. *Each dart is ejected from T at most once during the entire execution of the algorithm*

Proof. Let d be a dart. Assume d is ejected from T twice during the execution of the algorithm. By Lemma 7.8.2, it must do so in two distinct calls to CHANGEROOT. Let $i < j$ be such that d is ejected during the calls to CHANGEROOT on T_i and T_j. Then $d \in T_i$, $d \notin T_{i+1}$, $d \in T_j$, $d \notin T_{j+1}$. Therefore, two of the root-to-head(d) paths in the trees $T_i, T_{i+1}, T_j, T_{j+1}$ must cross, contradicting Lemma 7.8.1. □

7.9   Using the output of the MSSP algorithm

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

7.9.1   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 7.3.3 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. Using, e.g., the data structure of [Mehlhorn and Naher, 1990], 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_∞. Over 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 $O(\ell \log \log n)$.

7.9.2   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 ADDTODESCEN-DANTS 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 ··· d_k, T_0, P, Q):
    pre: d_1 ··· d_k are the darts of f_∞,
         T_0 is the tail(d_1)-rooted shortest-path tree,
         P is the sequence of pivots (d', d̂) produced by the MSSP algorithm
         Q is an array such that Q[i] is a set of vertices
    post: returns the set D = {(i, v, dist(tail(d_i), v)) : v ∈ Q[i]} of distances
1   initialize T :=link-cut tree representing T_0 with weight(v) := c(T_0[v]))
2   initialize D = ∅, i := 0
3   for each pivot (d', d̂) in P,
4     if (d', d̂) 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 ∈ Q[i],
7         append (i, v, T.GETWEIGHT(v)) to 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(d̂), head(d̂)) // add d̂ to T
11    T.ADDTODESCENDANTS(head(d'), c[d̂] + T.GETWEIGHT(tail(d)))
12  return 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.

7.9.3   Distance data structure