Chapter 8

Shortest paths with negative lengths

In this chapter we describe a linear-space, nearly linear-time algorithm that, given a directed planar graph G with real positive and negative lengths, but no negative-length cycles, and given a vertex $s \in V(G)$, computes single-source shortest paths from s to all vertices of G.

The algorithm triangulates G by adding edges with sufficiently large length so that shortest paths are not affected. It then uses the cycle separator algorithm (Theorem 5.8.1) to separate G into two parts; an interior subgraph G_1 and an exterior subgraph G_0. The edges and vertices of the cycle separator C are the only ones that belong to both parts. The vertices V_c of C are called boundary vertices. Since C is a simple cycle, it forms the boundary of a single face both in G_0 and in G_1. The *dense distance graph* $\mathrm{DDG}(G_i)$ of G_i is the complete graph on V_c, where for any $u, v, \in V_c$ the length of the arc uv is the length of the u-to-v path in G_i. The dense distance graph $\mathrm{DDG}(G)$ of G is the union of the dense distance graphs of its parts.

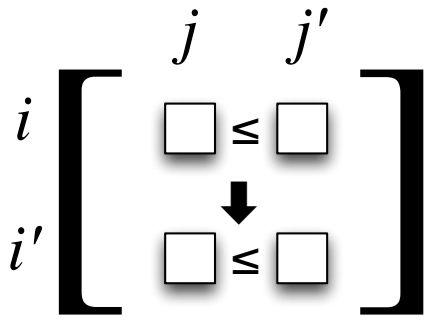
Lemma 8.0.1. *For any $u, v \in V_c$, the u-to-v distance in G equals the u-to-v distance in $DDG(G)$.*

Problem 8.1. *Prove Lemma 8.0.1*

Section 8.1 we discuss the *Monge* property, a special structural property of dense distance graphs, and some of its algorithmic consequences. In Section 8.3 we describe how to the Monge property is used to implement the Bellman-Ford algorithm on the dense distance graph in time $|V_c| \log |V_c|$. This is the main step in the overall recursive algorithm.

8.1 Total Monotonicity and the Monge Property

A matrix $M = (M_{ij})$ is *totally monotone* if for every i, i', j, j' such that $i < i'$, $j < j'$ and $M_{ij} \leq M_{ij'}$, we also hae $M_{i'j} \leq M_{i'j'}$.



A matrix $M = (M_{ij})$ is *convex Monge* (*concave Monge*) if for every i, i', j, j' such that $i < i'$, $j < j'$, we have $M_{ij} + M_{i'j'} \geq M_{ij'} + M_{i'j}$ ($M_{ij} + M_{i'j'} \leq M_{ij'} + M_{i'j}$). It is immediate that if M is convex Monge then it is totally monotone. It is also easy to see that the matrix obtained by transposing M is also totally monotone.

8.1.1 Boundary distances and the Monge Property

Consider one of the subgraphs G_i ($i \in \{1, 2\}$). Since all boundary vertices lie on the boundary of a single face of G_i, there is a cyclic clockwise order $v_1, v_2, \ldots, v_{|V_c|}$ on the vertices in V_c. Let A be the $|V_c| \times |V_c|$ weighted incidence matrix of $DDG(G_i)$. That is, $A_{k\ell}$ equals to the v_k-to-v_ℓ distance in G_i. We define the *upper triangle* of A to be the elements of A on or above the main diagonal. More precisely, the upper triangle of A is the portion $\{A_{k\ell} : k \leq \ell\}$ of A. Similarly, the lower triangle of A consists of all the elements on or below the main diagonal of A.

Lemma 8.1.1. *For any four indices k, k', ℓ, ℓ' such that either $A_{k\ell}, A_{k\ell'}, A_{k'\ell}$ and $A_{k'\ell'}$ are all in A's upper triangle, or are all in A's lower triangle (i.e., either $1 \leq k \leq k' \leq \ell \leq \ell' \leq |V_c|$ or $1 \leq \ell \leq \ell' \leq k \leq k' \leq |V_c|$), the convex Monge property holds:*

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

Proof. Consider the case $1 \leq k \leq k' \leq \ell \leq \ell' \leq |V_c|$, as in Fig. 8.1. Since G_i is planar, any pair of paths in G_i from k to ℓ and from k' to ℓ' must cross at some node w of G_i. Let $\delta_i[u, v]$ denote the u-to-v distance in G_i for any two vertices

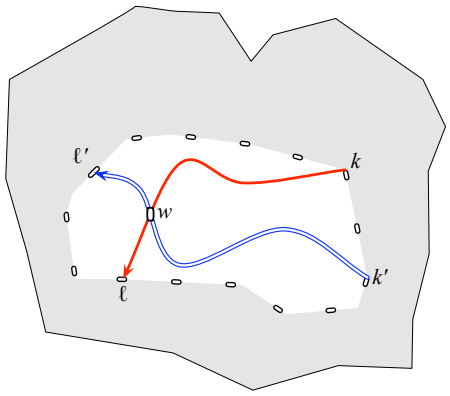


Figure 8.1: Vertices $k < k' < \ell < \ell'$ in clockwise order on the boundary vertices. Paths from k to ℓ and from k' to ℓ' must cross at some node w. This is true both in the internal and the external subgraphs of G

u, v of G_i. Note that $A_{k\ell} = \delta_i[v_k, v_\ell]$. We have

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

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

8.1.2 Finding all column minima of a Monge matrix

We will be interested in computing all column minima of a Monge matrix A. Let n denote the number of rows (or columns) of A, and assume A has the convex Monge property. For a subset R of the rows of A, the *lower envelope* of the rows R of A is the real-valued function $\mathcal{E}_R$ on the integers in $[1, n]$ defined by $\mathcal{E}_R(j) = \min_{i \in R} A_{ij}$.

Lemma 8.1.2. *Let A be a convex Monge matrix. For any subset R of the rows of A, the function $\text{argmin}_{i \in R} A_{ij}$ is monotonically non-increasing.*

Proof. Suppose $k = \text{argmin}_{i \in R} A_{i\ell}$. Then, for $k' > k$, $A_{k\ell} \leq A_{k'\ell}$. Since A is convex Monge, $A_{k\ell} + A_{k'\ell'} \geq A_{k\ell'} + A_{k'\ell}$. Therefore, for $\ell' > \ell$, $A_{k'\ell'} \geq A_{k\ell'}$, so $\text{argmin}_{i \in R} A_{i\ell'} \leq k$. □

A *breakpoint* of $\mathcal{E}_R$ is a pair of integers (k, ℓ) such that $k = \text{argmin}_{i \in R} A_{i\ell}$, and $k \neq \text{argmin}_{i \in R} A_{i(\ell+1)}$. For $k = \text{argmin}_{i \in R} A_{in}$, we always consider (k, n) to be a breakpoint of $\mathcal{E}_R$. For notational convenience we also consider the pair $(\perp, 0)$ as a trivial breakpoint. If $B = (k_1, \ell_1), (k_2, \ell_2), \ldots$ are the breakpoints of the lower envelope of A, ordered such that $k_1 < k_2 < \ldots$, then the minimum elements in each column in the range $(\ell_{j-1}, \ell_j]$ is at row k_j.

Problem 8.2. *Give a tight bound (i.e., matching upper and lower bounds) on the number of breakpoints of the lower envelope of a convex Monge matrix*

We focus our attention on finding the breakpoints of the lower envelope of A, since, given the breakpoints, all column minima can be recovered in $O(n)$ time. Lemma 8.1.2 suggests the following procedure for finding the breakpoints of the lower envelope of a convex Monge matrix A. The procedure maintains the lower envelope of an increasingly larger prefix R of the rows of A. Initially R consists of just the first row of A, and the breakpoints of $\mathcal{E}_R$ are just $(\perp, 0)$ and $(1, n)$. Let $B = (k_1, \ell_1), (k_2, \ell_2), \ldots$ be the non-trivial breakpoints of the lower envelope of the first i rows of A, ordered such that $k_1 > k_2 > \ldots$. Note that, by Lemma 8.1.2, this implies $\ell_1 < \ell_2 < \ldots$. To obtain the breakpoints of the lower envelope of the first $i + 1$ rows of A, the procedure compares $A_{(i+1)\ell_j}$ and $A_{k_j \ell_j}$ for increasing values of j, until it encounters an index j such that $A_{(i+1)\ell_j} \geq A_{k_j \ell_j}$. By Lemma 8.1.2, and by the definition of breakpoints, this implies that

1. For all $\ell \geq \ell_j$, $\mathcal{E}_{[1\ldots i+1]}(\ell) = \mathcal{E}_{[1\ldots i]}(\ell) \neq i + 1$, and

2. For all $\ell \leq \ell_{j-1}$, $\mathcal{E}_{[1\ldots i+1]}(\ell) = i + 1$.

Hence, the lower envelope of the first $i + 1$ rows of A consists of a suffix of the breakpoints of B starting at (k_j, ℓ_j) plus, perhaps, a new breakpoint $(i+1, \ell)$ for some $\ell \in [\ell_{j-1}, \ell_j)$. The exact column ℓ where the new breakpoint occurs can be found in by binary search for the largest column ℓ such that $A_{(i+1)\ell} < A_{k_j, \ell}$.

We summarize the procedure in the following theorem

Theorem 8.1.3. *There exists a procedure that computes the breakpoints of the lower envelope of a n-by-n convex Monge matrix in $O(n \log n)$ time.*

Proof. We have already described the procedure. To analyze the running time, observe that a breakpoint at row i is created at most once, at the iteration that adds row i to the set R. Hence, the total number of breakpoints considered by the procedure is $O(n)$. At each iteration, each comparison takes constant time, and all but the last comparison of the iteration eliminate one existing breakpoint. Hence the time required for all comparisons at all iterations is $O(n)$. Finally, the binary search at each iteration takes $O(\log n)$ time. Hence the total running time of the procedure is $O(n \log n)$. □

8.1.3 Finding all the column minima of a triangular Monge matrix

The procedure of Theorem 8.1.3 can be easily adapted to find the breakpoints of the lower envelope of triangular convex Monge matrix. The procedure needs to be adapted since Lemma 8.1.2 does not apply to triangular matrices.

Problem 8.3. *Give a tight bound (i.e., matching upper and lower bounds) on the number of breakpoints of the lower envelope of a convex Monge triangular matrix*

Consider first an upper triangular convex Monge matrix. At the beginning of iteration i the breakpoints of the lower envelope of the submatrix that consists of the first $i-1$ rows and all columns are known. Note that, since the matrix is upper triangular, no row greater than $i-1$ contributes to the lower envelope of the first $i-1$ columns. It follows that the breakpoints in the first $i-1$ columns have already been computed before iteration i, and can be disregarded for the remainder of the procedure. On the other hand, the submatrix defined by the first i rows and columns with index at least i is rectangular, so Lemma 8.1.2 does apply. Therefore, the procedure correctly computes the breakpoints of the lower envelope of this rectangular matrix.

For lower triangular matrices, a symmetric procedure that handles the rows in reverse order should be used.

Problem 8.4. *Adjust the procedure of Theorem 8.1.3 to work with lower triangular convex Monge matrices.*

8.2 The Algorithm

We now turn back to the description of the algorithm for computing shortest paths in the presence of negative arc lengths and no negative length cycles. It consists of five stages. The first four stages alternate between working with negative lengths and working with only positive lengths. Let r be an arbitrary boundary vertex.

Recursive call: The first stage recursively computes the distances from r within G_i for $i = 0, 1$. The remaining stages use these distances in the computation of the distances in G.

Intra-part boundary-distances: For each graph G_i the algorithm uses the MSSP algorithm of Chapter 7 to compute all distances in G_i between boundary vertices. This takes $O(n \log n)$ time.

Single-source inter-part boundary distances: The algorithm uses a fast implementation of Bellman-Ford in the dense distance graph of G to compute the distances in G from r to all other boundary vertices. DDG(G) has $O(\sqrt{n})$ vertices, so the number of iterations of Bellman-Ford is $O(\log n)$. Each iteration consists of relaxing all the edges DDG(G). Because for each

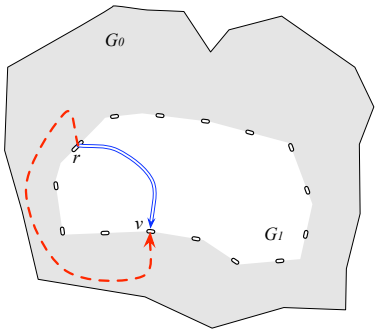


Figure 8.2: A graph G and a decomposition using a cycle separator into an external subgraph G_0 (in gray) and an internal subgraph G_1 (in white). Only boundary vertices are shown. r and v are boundary vertices. The double-lined blue path is an r-to-v shortest path in G_1. The dashed red path is an r-to-v shortest path in G_0.

G_i, the upper and lower triangles of DDG(G_i) each has the Monge property, all of the edges can be relaxed in $O(\sqrt{n}\log n)$ time. Thus this step is implemented in $O(n \log n)$ time. This step is described in Section 8.3.

Single-source inter-part distances: For each G_i, the distances obtained in the previous stages are used, in a Dijkstra computation, to compute the distances in G from r to all the vertices of G_i. Dijkstra's algorithm requires the lengths in G_i to be non-negative, so the recursively computed distances are used as a consistent price vector. This stage takes $O(n \log n)$ time. (This stage can actually be implemented in $O(n)$ using the algorithm of Chapter 6. This however does not change the overall running time of the algorithm.) This stage is described in Section 8.4

Rerooting single-source distances: The algorithm has obtained distances in G from r. In the last stage these distances are used as a consistent price vector to compute, again using Dijkstra's algorithm, distances from s in G. This stage also requires $O(n \log n)$ time.

```
def SSSP(G, s):
    pre: G is a directed embedded graph with arc-lengths.
         s is a vertex of G.
    post: returns a table d giving distances in G from s to all vertices of G
1   if G has ≤ 2 vertices, the problem is trivial; return the result
2   find a cycle separator C of G with O(√n) boundary vertices
3   let G0,G1 be the external and internal parts of G with respect to C
4   for i = 0, 1: let d_i = SSSP(G_i, r)
```

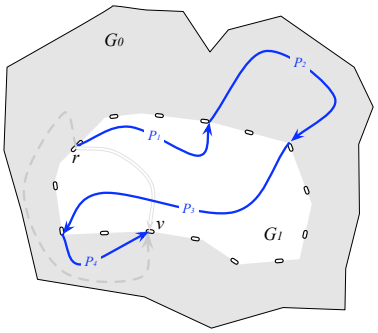


Figure 8.3: The solid blue path is an r-to-v shortest path in G. It can be decomposed into four subpaths. The subpaths P_1 and P_3 (P_2 and P_4) are shortest paths in G_1 (G_0) between boundary vertices. The r-to-v shortest paths in G_0 and G_1 are shown in gray in the background.

5 for $i = 0, 1$:
  use d_i as input to the multiple-source shortest-path algorithm to compute a table δ_i such that $\delta_i[u, v]$ is the u-to-v distance in G_i for every pair u, v of boundary vertices
6 use δ_0 and δ_1 to compute a table B such that $B[v]$ is the r-to-v distance in G for every boundary vertex v
7 for $i = 0, 1$:
  use tables d_i and B, and Dijkstra's algorithm to compute a table d'_i such that $d'_i[v]$ is the r-to-v distance in G for every vertex v of G_i
8 define a price vector $\boldsymbol{\phi}$ for G such that $\boldsymbol{\phi}[v]$ is the r-to-v distance in G:
$$\boldsymbol{\phi}[v] = \begin{cases} d'_0[v] & \text{if } v \text{ belongs to } G_0 \\ d'_1[v] & \text{otherwise} \end{cases}$$
9 use Dijkstra's algorithm with price vector $\boldsymbol{\phi}$ to compute a table d such that $d[v]$ is the s-to-v distance in G for every vertex v of G

8.3 Computing Single-Source Inter-Part Boundary Distances

We now describe how to efficiently compute the distances in G from r to all boundary vertices (Line 6). The algorithm computes the from-r distances in G by computing the from-r distances in the dense distance graph DDG(G). For $i = 1, 2$, the edge lengths of the dense distance graph of G_i have already been computed and stored in δ_i in the previous step (Line 5).

Theorem 8.3.1. *Let G be a directed graph with arbitrary arc-lengths. Let C be a cycle separator in G and let G_0 and G_1 be the external and internal parts*

of G with respect to C. Let δ_0 and δ_1 be the all-pairs distances between vertices in V_c in G_0 and in G_1 respectively. Let $r \in V_c$ be an arbitrary boundary vertex. There exists an algorithm that, given δ_0 and δ_1, computes the from-r distances in G to all vertices of C in $O(|V_c|^2 \log(|V_c|))$ time and $O(|V_c|)$ space.

Proof. Consider the Bellman-Ford algorithm on DDG(G). There are $|V_c|$ vertices, so the number of iterations is $|V_c|$. In iteration j Bellman-Ford computes for every $v \in V_c$, the quantity $e_j[v]$, which is the length of a shortest r-to-v path consisting of at most j edges of DDG(G). The following procedure, BoundaryDistances describes this process in detail.

```
def BoundaryDistances(δ_0, δ_1, r)
      pre: δ_i is a table of distances between boundary nodes in G_i
           r is a boundary vertex
      post: returns a table B of from-r distances in G
1     e_0[v] := ∞ for all v ∈ V_c
2     e_0[r] := 0
3     for j = 1, 2, 3, ..., |V_c|
4           e_j[v] := min { min_{w∈V_c}{e_{j-1}[w] + δ_0[w, v]},  } , ∀v ∈ V_c
                          { min_{w∈V_c}{e_{j-1}[w] + δ_1[w, v]}  }
5     B[v] := e_{|V_c|}[v] for all v ∈ V_c
```

The correctness of BoundaryDistances follows from the fact that it is an implementation of the Bellman-Ford algorithm.

To complete the proof of Theorem 8.3.1 it remains to show that BoundaryDistances can be implemented in $O(|V_c|^2 \cdot \log(|V_c|))$ time. The number of iterations in BoundaryDistances is $|V_c|$. It therefore suffices to show how to implement Line 4 in $O(|V_c| \log(|V_c|))$ time. This is the crux of the algorithm. Consider the natural clockwise order on the boundary vertices, and regard the tables δ_i as square $|V_c|$-by-$|V_c|$ matrices. The key observation is that Line 4 can be implemented by computing all column minima of two matrices A^0, A^1, such that the (k, ℓ) element of A^i is $A^i_{k\ell} = e_{j-1}[v_k] + \delta_i[v_k, v_\ell]$, and then taking the minimum of the two values obtained for each column. Since δ_i stores the pairwise distances between vertices on a single face of G_i, it can be decomposed, by Lemma 8.1.1, into an upper triangular matrix and a lower triangular matrix, both of whom are convex Monge. Next note that adding a constant to each row of a Monge matrix preserves the Monge property. For $1 \le k \le k' \le \ell \le \ell' \le |V_c|$ or $1 \le \ell \le \ell' \le k \le k' \le |V_c|$,

$$\delta_i[v_k, v_\ell] + \delta_i[v_{k'}, v_{\ell'}] \ge \delta_i[v_k, v_{\ell'}] + \delta_i[v_{k'}, v_\ell]$$

implies

$$\begin{aligned}(e_{j-1}[v_k] + \delta_i[v_k, v_\ell]) + (e_{j-1}[v_{k'}] + \delta_i[v_{k'}, v_{\ell'}]) \quad \ge \\ (e_{j-1}[v_k] + \delta_i[v_k, v_{\ell'}]) + (e_{j-1}[v_{k'}] + \delta_i[v_{k'}, v_\ell])\end{aligned}$$

Hence each matrix A^i decomposes into a lower and an upper triangular convex Monge matrices. The algorithm can therefore find all column minima of each A^i by finding all column minima of each of the two triangular matrices comprising A^i. Using the extension of Theorem 8.1.3 to triangular matrices, this can be done in $O(|V_c| \log(|V_c|))$ time, as desired. □

8.4 Computing Single-Source Inter-Part Distances

We now describe how to implement Line 7 of SSSP which computes the distances from r to all other vertices of G, rather than just to the boundary vertices. The algorithm does so by computing tables d'_0 and d'_1 where $d'_i[v]$ is the r-to-v distance in G for every vertex v of G_i. Recall that in Line 4 of SSSP the algorithm had already computed the table d_i that stores the r-to-v distance in G_i for every vertex v of G_i.

Table d'_i is computed using a Dijkstra computation, where the distances of boundary nodes are initialized to their known distances in G (computed in Line 6 and stored in table B), and using the distances d_i as a price vector. By Lemma 7.1.3, d_i is a consistent price vector for G_i. The following lemma shows that this correctly computes the distances in G from r to all vertices of G_i.

Lemma 8.4.1. *Let P be an r-to-v shortest path in G, where $v \in G_i$. Then P can be expressed as $P = P_1P_2$, where P_1 is a (possibly empty) shortest path from r to a node $u \in V_c$, and P_2 is a (possibly empty) shortest path from u to v that only visits nodes of G_i.*

Problem 8.5. *Prove Lemma 8.4.1.*

8.5 Correctness and Analysis

We now show that at each stage of SSSP, the necessary information has been correctly computed and stored. The recursive call in Line 4 computes and stores the from-r distances in G_i. The conditions for applying the MSSP algorithm in Line 5 hold since all boundary vertices lie on the boundary of a single face of G_i and since, by Lemma 7.1.3, the from-r distances in G_i constitute a consistent price vector for G_i. The correctness of the single-source inter-part boundary distances stage in Line 6 and of the single-source inter-part distances stage in Line 7 was proved in Sections 8.3 and 8.4. Thus, the r-to-v distances in G for all vertices v of G are stored in d'_0 for $v \in G_0$ and in d'_1 for $v \in G_1$. Note that d'_0 and d'_1 agree on distances from r to boundary vertices. Therefore, the price vector ϕ defined in Line 8 is consistent for G, so the conditions to run Dijkstra's algorithm in Step 9 hold, and the from-s distances in G are correctly computed. This establishes the correctness of SSSP.

To bound the running time of the algorithm we bound the time it takes to complete one recursive call to SSSP. Let $|G|$ denote the number of nodes in

the input graph G, and let $|G_i|$ denote the number of nodes in each of its subgraphs. Computing the intra-subgraph boundary-to-boundary distances using the MSSP algorithm takes $O(|G_i| \log |G_i|)$ for each of the two subgraphs, which is in $O(|G| \log |G|)$. Computing the single-source distances in G to the boundary vertices is done in $O(|G| \log(|G|))$, as shown in Section 8.3. The extension to all vertices of G is again done in $O(|G_i| \log |G_i|)$ for each subgraph. Distances from the given source are computed in an additional $O(|G| \log |G|)$ time. Thus the total running time of one invocation is $O(|G| \log |G|)$. Therefore the running time of the entire algorithm is given by

$$\begin{aligned} T(|G|) &= T(|G_0|) + T(|G_1|) + O(|G| \log |G|) \\ &= O(|G| \log^2 |G|). \end{aligned}$$

Here we used the properties of the separator, namely that $|G_i| \leq 2|G|/3$ for $i = 0, 1$, and that $|G_0| + |G_1| = |G| + O(\sqrt{|G|})$. The formal proof of this recurrence is given in the following lemma.

Lemma 8.5.1. *Let $T(n)$ satisfy the recurrence $T(n) = T(n_1)+T(n_2)+O(n \log n)$, where $n \leq n_1 + n_2 \leq n + 4\sqrt{n}$ and $n_i \leq \frac{2n}{3}$. Then $T(n) = O(n \log^2 n)$.*

Problem 8.6. *Prove Lemma 8.5.1.*

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

$$\begin{aligned} S(|G|) &= \max\{S(|G_0|), S(|G_1|\} + O(|G|) \\ &= O(|G|) \quad \text{because} \max\{|G_0|, |G_1|\} \leq 2|G|/3. \end{aligned}$$

8.6 Chapter Notes

[Lipton et al., 1979] described an $O(n^{3/2}$-time algorithm for the problem. This algorithm uses small separators, but not the Monge property. [Henzinger et al., 1997] improved the running time to $O(n^{4/3} \log^{2/3} D$, where D is the sum of absolute values of the lengths. In the early 2000's, [Fakcharoenphol and Rao, 2006] gave the first nearly linear $O(n \log^3 n)$-time algoirthm for the problem, using the Monge property. The $O(n \log^2 n)$-time algorithm described in this chapter is the one by Klein, Mozes and Weimann [Klein et al., 2010]. The fastest algorithm currently known [Mozes and Wulff-Nilsen, 2010] runs in $O(n \log^2 n / \log \log n)$ time.