Chapter 9

Shortest paths in dense distance graphs

The crux of Chapter 8 was in showing (Section 8.3) how to emulate the Bellman-Ford algorithm on the dense distance graph in time roughly proportional to the square of the number of boundary vertices. In this chapter we will show[1] how to emulate Dijkstra's algorithm on the dense distance graph in time roughly proportional to the number of boundary vertices, even though the number of edges in those graphs is proportional to the square of the number of boundary vertices!

We recall Dijkstra's algorithm. The algorithm maintains a heap of vertices labeled by estimates $d(\cdot)$ of their distances from the root. At each iteration, the vertex v with minimum estimated distance is extracted from the heap (its estimate is in fact the correct distance), and all the arcs whose tail is v are relaxed. In the pseudocode, this relaxation is done by the procedure ACTIVATE, which also updates the heap whenever the distance label of a vertex changes.

Algorithm 9.1 Dijkstra(G, s)

```
1: for all v ∈ V(G): d(v) ← ∞
2: d(s) ← 0
3: initialize an empty heap Q
4: UpdateHeap(Q, s, d(s))
5: S ← ∅
6: while S ≠ V(G) do
7:     v ← ExtractMin(Q)
8:     Activate(Q, G, v, d)
9:     S ← S ∪ {v}
```

In Chapter 8, the speedup is obtained by decomposing the dense distance

[1] Parts of our description were exracted and adapted from [Kaplan et al., 2012]

graph into subgraphs that have the Monge property. The Monge property is exploited to relax the edges in each subgraph efficiently. We show that a similar idea works for Dijkstra's algorithm. The graph can be decomposed into subgraphs, which allow for a fast implementation of ExtractMin and Activate. In the case of Dijkstra's algorithm additional coordination is required between the subgraphs since the algorithm must identify, at each step, the vertex with minimum label in the whole graph, whereas the efficient implementation handles each subgraph separately.

Recall the definition of the dense distance graph from Chapter 8. A cycle separator C with $n = O(\sqrt{N})$ vertices separates a planar graph G with N vertices into an external and internal subgraphs, G_0 and G_1. Let K_i ($i \in \{0, 1\}$) be the complete graph on the boundary vertices of G_i (the vertices of C), where the length of arc uv of K_i is the u-to-v distance in G_i. The union $\cup_i K_i$ is called the dense distance graph of G w.r.t. C.

We note that in some cases the graph G is decomposed into more than two regions G_i. The definition of the dense distance graph trivially extends to such cases. The algorithm described in this chapter can be generalized to handle more than two regions as well, under the restriction that the boundary vertices within each region lie on a constant number of faces. For simplicity, we describe the two-region case.

9.1 Decomposing a DDG into bipartite graphs

To be able to implement ExtractMin and Activate efficiently, the algorithm decomposes the dense distance graph into bipartite graphs. Each complete graph K_i is decomposed into complete bipartite graphs as follows. Refer to Figure 9.1. Consider the sequence of vertices of K_i according to their clockwise order on C. The algorithm splits the vertices into two consecutive halves A and B, and adds the complete bipartite graph on A and B to the decomposition. Next, it recurses on A and on B. Since A and B are disjoint, and their size is reduced by a factor of two at each level of the recursion, each vertex of K_i appears in $O(\log |C|) = O(\log n)$ bipartite subgraphs. Let $\mathcal{H} = \{H_i\}$ denote the set of all bipartite graphs in the decompositions of both K_i's. The total number of bipartite graphs is $|\mathcal{H}| = O(|C|) = O(n) = O(\sqrt{N})$.

The decomposition procedure can also be illustrated by considering the weighted incidence matrix of K_i. While this matrix is not Monge, the submatrices that correspond to bipartite graphs are Monge. See Figure 9.2.

Lemma 9.1.1. *Let M be the (weighted) incidence matrix of K_i. Let H be a graph in the decomposition of K_i into bipartite graphs. The restriction of M to the vertices of H is Monge.*

The proof is nearly identical to that of Lemma 8.1.1.

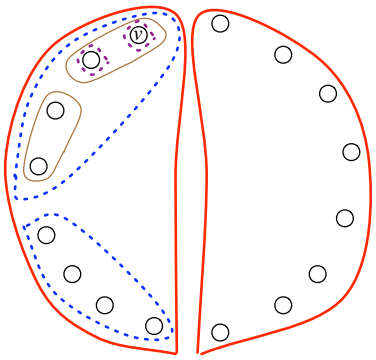

Figure 9.1: Decomposition of a complete graph over a set of vertices on a single face into bipartite complete graphs such that each vertex appears in a logarithmic number of bipartite graphs. The bipartitions in all the bipartite graphs in which the vertex v appears are indicated.

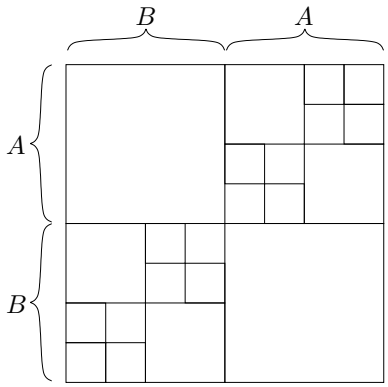

Figure 9.2: Illustration of the decomposition of M into Monge Submatrices.

9.2 The Monge heap

The algorithm maintains a data-structure, called a *Monge Heap*, for each bipartite graph $H \in \mathcal{H}$. Let A, B be the bipartition of the vertex set of H. The Monge heap maintains distance labels of the vertices in A, and implicitly maintains the distance estimates to the vertices of B. It supports the following operations to facilitate the implementation of Dijkstra's algorithm. The initials FR stand for Fakcharoenphol and Rao, who conceived this data structure.

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

2. FR-FINDMIN - Returns the vertex $v \in B$ with minimum label among those that have not yet been extracted. This operation also returns the label d of v and takes $O(1)$ time.

3. FR-EXTRACTMIN - Returns the vertex $v \in B$ with minimum label among those that have not yet been extracted, and removes v from the set B. This operation also returns the label d of v and takes $O(\log |B|)$ time. The integrity of the data structure is only guaranteed if all subsequent FR-ACTIVATE operations have labels at least d.

The next section describes how Monge heaps are used to implement Dijkstra's algorithm. The following section deals with the implementation of the Monge heap itself.

9.3 Implementing Dijkstra's algorithm using Monge heaps

The edges of each bipartite subgraph can be implicitly relaxed efficiently using a dedicated Monge heap $\mathcal{M}_j$ for each bipartite subgraph $H_j \in \mathcal{H}$. The algorithm initializes the Monge heaps and activates s in those Monge heaps $\mathcal{M}_j$ where $s \in A_j$.

The minimum elements from each Monge heap $\mathcal{M}_j$ are maintained in a regular global heap $\mathcal{Q}$.

In each iteration of the main loop, a vertex v with global minimum label is extracted from the global heap $\mathcal{Q}$. Let $\mathcal{M}_{\hat{j}}$ be the Monge heap that contributed v, and let d_v be its distance label. The algorithm extracts v from $\mathcal{M}_{\hat{j}}$, and updates the new minimum vertex in $\mathcal{M}_{\hat{j}}$ in the global heap $\mathcal{Q}$.

Note that since a vertex v appears in multiple H_j's, v may be extracted as the minimum element of the global heap $\mathcal{Q}$ multiple times, once for each Monge heap it appears in. However, the label $d(v)$ of v is finalized at the first time v is the minimum element of $\mathcal{Q}$. At that time, and only at that time, the algorithm adds v to the set of vertices whose distance label is finalized, activates v using FR-ACTIVATE in all Monge heaps $\mathcal{M}_j$ such that $v \in A_j$, and updates the representatives of those Monge heaps in $\mathcal{Q}$.

Algorithm 9.2 FR-DIJKSTRA($\mathcal{H}, s$)

Input: a set $\mathcal{H} = \{H_j\}$, where each H_j is a complete bipartite graph on $V(H_j) = A_j \cup B_j$.
a vertex s in $\bigcup H_j$.
Output: the distances $d(\cdot)$ from s in $\bigcup H_j$.

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

9.3.1 Analysis

Since each vertex appears in $O(\log n)$ bipartite graphs H_j in $\mathcal{H}$, the number of times each vertex is extracted from the global heap $\mathcal{Q}$ is $O(\log n)$. Since $\mathcal{Q}$ contains one representative element from each Monge heap $\mathcal{M}_j$, a single call to ExtractMin on $\mathcal{Q}$ takes $O(\log(\sqrt{n})) = O(\log n)$ time. Therefore, the total time spent on extracting vertices from $\mathcal{Q}$ is $O(n \log^2 n)$.

As for the cost of operations on the Monge heaps, FR-Activate and FR-ExtractMin are called at most once per vertex in each Monge heap, and the number of calls to FR-FindMin is bounded by the number of calls to FR-Activate and FR-ExtractMin. Since each vertex appears in $O(\log n)$ Monge heaps, the total number of these operations is $O(n \log n)$. Since each operation takes $O(\log n)$ amortized time, the total time spent on operations on the Monge heaps is $O(n \log^2 n)$ as well.

It follows that the FR-Dijkstra takes $O(n \log^2 n) = O(\sqrt{N} \log^2 N)$ time to complete.

9.4 Implementing Monge heaps

Let $\mathcal{M}$ be the Monge heap of a bipartite subgraph H with corresponding weighted incidence matrix M with a set of n rows A and a set of n columns B.

The Monge heap $\mathcal{M}$ stores a *range-minimum data structure* over the rows of M. Given a row $u \in A$ and a range of columns $[v_1, v_2]$ in B, the range-minimum data structure returns $\operatorname{argmin}_{v_1 \le v \le v_2} M_{uv}$ in $O(\log n)$ time.

$\mathcal{M}$ maintains a distance label $\delta(u)$ for every vertex $u \in A$. Initially $\delta(u) = \infty$ for every $u \in A$. The distance label of a vertex $v \in B$ is defined to be $\delta(v) = \min_{u \in A}\{\delta(u) + M_{uv}\}$ and is not stored explicitly. For every vertex $v \in B$ the algorithm maintains a bit indicating whether v has already been extracted from $\mathcal{M}$ or not. Initially all vertices of B are in $\mathcal{M}$. A vertex v is extracted from $\mathcal{M}$ when it has the smallest distance label among all vertices in the global heap $\mathcal{Q}$. The monotonicity of the distance labels of nodes extracted from the global heap in Dijkstra's algorithm implies that after v is extracted from $\mathcal{M}$ a subsequent decrease in $\delta(u)$ for a vertex $u \in A$ cannot cause $\delta(v)$ to decrease: the value of $\delta(u)$ cannot be smaller than the value of $\delta(v)$ when v was extracted from $\mathcal{M}$.

We say that $u \in A$ is the *parent* of $v \in B$ (and v is the *child* of u) if $\delta(u)$ is finite and $\delta(v) = \delta(u) + M_{uv}$; see below for handling ties. As the execution of Dijkstra's algorithm progresses, the distance labels $\delta(u)$ of vertices $u \in A$ may change and thereby the parents of vertices $v \in B$ may change. Assuming that the parent of every member of $v \in B$ is uniquely defined (at some fixed time), the Monge property of M implies the following two properties:

1. The set of vertices of B for which a specific vertex $u \in A$ is the parent are consecutive in B.

2. If vertex u' follows u in A, then the set of vertices of which u' is the parent follows in B the set of vertices of which u is the parent.

Furthermore, in case of ties, i.e. when there are two different vertices $u_1, u_2 \in A$ such that $\delta(v) = \delta(u_1)+M_{u_1v} = \delta(u_2)+M_{u_2v}$, the Monge property of M implies that ties can always be broken by picking a parent for v such that Properties (1) and (2) above continue to be satisfied. The algorithm indeed (implicitly) maintains parents in this manner such that Properties (1) and (2) are satisfied. Note that vertices in B with infinite distance labels also have parents in A. For such a vertex v any vertex $u \in A$ satisfies that $\delta(v) = \delta(u) + M_{uv}$ and the algorithm can choose any vertex to be a parent of v as long as it maintains Properties (1) and (2).

The algorithm maintains a binary search tree T of triplets of the form (a, b_1, b_2) where $a \in A$, $b_1, b_2 \in B$, and the set of vertices of B between b_1 and b_2 (inclusive) is a maximal interval of vertices of B that are currently in $\mathcal{M}$ and whose parent is a. See Figure 9.3. Note that the same vertex $a \in A$ may appear in more than one triplet of T because, after extracting vertices of B from the Monge heap, the set of non-extracted vertices of which a is a parent might not be a contiguous subsequence of the vertices of B.[2] That is, the extracted elements form gaps in the sequence, so it is required to maintain the remaining elements as the union of smaller contiguous subsequences. The order of the triplets in T is lexicographic. Namely: (1) If vertex a' follows a in A then all the triplets (a', b'_1, b'_2) follow all the triplets (a, b_1, b_2). (2) The set of triplets (a, b_1, b_2) of the same vertex a are ordered in T by the order of their (pairwise disjoint) intervals in B.

The Monge heap structure also consists of a standard heap Q_B containing, for every triplet $(a, b_1, b_2) \in T$, a vertex b between b_1 and b_2 (inclusive) that minimizes $\delta(b) = \delta(a) + M_{ab}$. The key of b in Q_B is $\delta(b)$.[3]

The operations on a Monge heap are implemented as follows:

- FR-FindMin: Return a vertex b with minimum distance label in Q_B.
- FR-ExtractMin: Extract the vertex b with minimum distance label from Q_B. The algorithm marks b as removed from $\mathcal{M}$. It finds the (unique) triplet (a, b_1, b_2) containing b in T. Let b' and b'' be the members of B that precede and follow b, respectively, within this triplet, if they exist. The algorithm splits the triplet (a, b_1, b_2) and replaces it with two triplets (a, b_1, b') and (a, b'', b_2) (if these intervals are defined). In each of these new triplets it finds the vertex b^* that minimizes $\delta(b^*) = \delta(a)+M_{ab^*}$, using the range minimum data structure of M, and then inserts b^* into Q_B.
- FR-Activate(u, d): First the algorithm sets $\delta(u) = d$ and then finds the children of u in B.

[2]For example, as depicted in Figure 9.3, z may be the parent of vertices 9–16 of B just before we extract vertex 13 of B from the subheap. After this extraction z is the parent of vertices 9–12 of B which form one triplet $(z, 9, 12)$, and of vertices 14–16 which form another triplet $(z, 14, 16)$.

[3]This heap can be implemented within the tree T by maintaining subtree minima at the nodes of T.

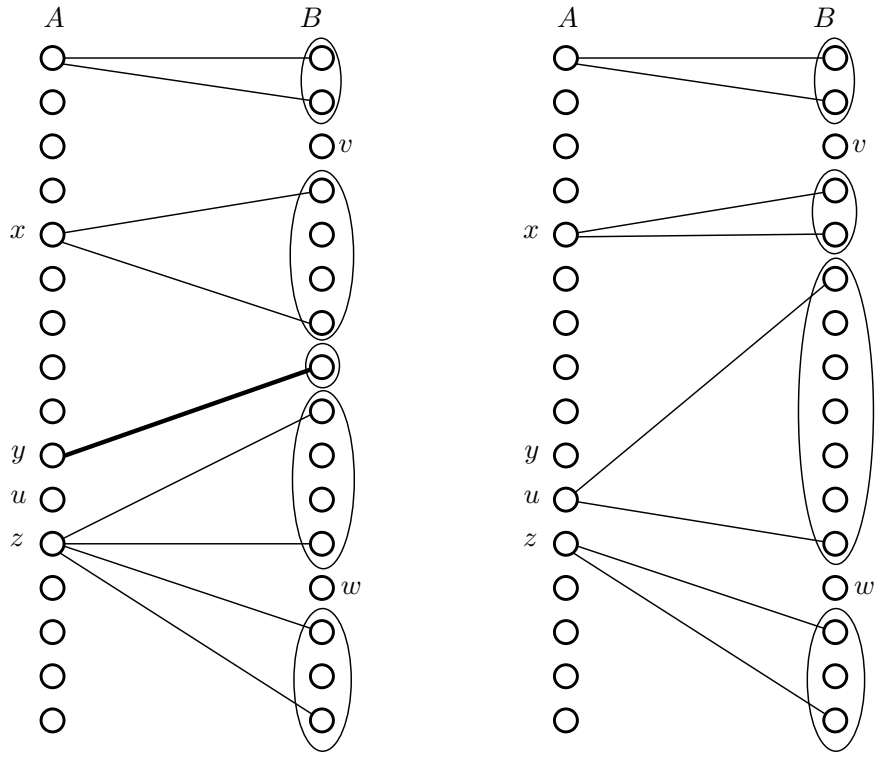

Figure 9.3: The triplets in the tree T of a Monge heap. Each triplet (a, b_1, b_2) is depicted by a line from a to b_1, a line from a to b_2 and an ellipse around the vertices of B between b_1 and b_2. On the left we see the triplets before the scan of u and on the right we see the triplets after that scan. Vertices v and w have already been deleted from this Monge heap and therefore do not belong to any triplet.

If u is the first vertex in the Monge heap for which this operation is applied then all the vertices of B are children of u (none of them could have already been removed, and u is the only vertex in A with a finite distance label). Otherwise, we next describe how to the algorithm finds the children of u which are in triplets associated with vertices preceding u in A. The algorithm finds children of u which are in triplets associated with vertices following u in A in a symmetric manner.

If there is no triplet $t = (w, f_1, f_2)$ such that w precedes u in A then u does not have children in triplets associated with vertices preceding it in A. Otherwise, the algorithm searches T and finds the last triplet

$t = (w, f_1, f_2)$ such that w precedes u in A. It then traverses the triplets in T one by one backwards, starting from t, until it reaches a triplet (a, b_1, b_2) such that $\delta(a) + M_{ab_1} < \delta(u) + M_{ub_1}$ or until it scanned all triplets preceding t without finding such a triplet (a, b_1, b_2). Then the algorithm performs one of the following steps.
(1) If no triplet (a, b_1, b_2) as above was found then all vertices in triplets preceding t become children of u.
(2) If $\delta(a) + M_{ab_2} \geq \delta(u) + M_{ub_2}$ then among vertices in triplets preceding t, the first vertex in B whose parent changes to u belongs to the triplet (a, b_1, b_2). It is found by a binary search on the subsequence of B between b_1 and b_2.
(3) If $\delta(a) + M_{ab_2} < \delta(u) + M_{ub_2}$ and (a, b_1, b_2) is not t then among vertices in triplets preceding t, the first vertex in B whose parent changes to u is the first vertex in the triplet following (a, b_1, b_2).
(4) If $\delta(a) + M_{ab_2} < \delta(u) + M_{ub_2}$ and (a, b_1, b_2) is t then none of the vertices in triplets associated with vertices preceding u in A acquires u as its parent.

Notice that, as mentioned, the Monge property implies that the sequence of children that u acquires in B is contiguous. Moreover, since Dijkstra's algorithm finds the distances in monotonically increasing order, and since we extract a vertex b from $\mathcal{M}$ only when b is the minimum in the global heap $\mathcal{Q}$ of Dijkstra's algorithm, then once b is removed from $\mathcal{M}$ it cannot acquire a new parent (as that would mean that a shorter path to b was discovered). These two observations imply that if there are two consecutive triplets (w', x', y') and (w, x, y) such that w' precedes w in A and if there is a vertex z between y' and x in B that was already extracted from $\mathcal{M}$, then the search for children of u, while scanning u as described above, never proceeds beyond (w', x', y'). For an example consider the scan of u in Figure 9.3. Following this scan u acquires the children of z from one of the two triplets of z, the child of y, and two out of the four children of x. Since v and w have already been deleted, then u cannot acquire children preceding v or following w.

Let x (resp., y) be the first (resp., last) child of u in B, as obtained in the preceding step. The algorithm removes from T all triplets containing vertices between x and y, and removes from Q_B the elements contributed by these triplets. Let (a, b_1, b_2) be the removed triplet that contains x. If $x \neq b_1$ then the algorithm creates a new triplet to (a, b_1, z_1) where z_1 is the vertex preceding x in B. Similarly, let (a', b'_1, b'_2) be the removed triplet that contains y. If $y \neq b'_2$ the algorithm creates a new triplet (a', z_2, b'_2) where z_2 is the vertex following y in B. The algorithm creates a new triplet (u, x, y) and inserts into T the triplets (a, b_1, z_1), (u, x, y), and (a', z_2, b'_2) in this order. Finally, it updates the vertices of B that these new triplets contribute to Q_B. It finds these vertices by a range minimum query in the range minima data structure of the appropriate row of M.

9.4.1 Analysis

An elementary implementation of a range-minimum data structure using balanced binary trees requires space and construction time that are linear in the number of entries of M, namely $O(n^2)$, and answers range-minimum queries in $O(\log n)$ time. Such a data structure can be constructed at the time the matrix M is constructed (i.e., when computing the dense distance graph), and be given as input to the Monge heap. Alternatively, one may use the Monge submatrix range-minimum data structure of Kaplan et al. [Kaplan et al., 2012], which can be requires $O(n \log n)$ construction time and space, and answers range-minimum queries in $O(\log n)$ time.

Clearly, FR-FindMin() takes $O(1)$ time. We next argue that FR-ExtractMin() takes $O(\log n)$ (worst-case) time and FR-Activate() takes $O(\log n)$ amortized time: Both FR-ExtractMin() and FR-Activate() insert a constant number of new triplets to T in $O(\log n)$ time, make a constant number of range-minimum queries in $O(\log n)$ time, and update the representatives of the new triplets in the heap Q_B in $O(\log n)$ time. FR-Activate(u, d), however, may traverse many triplets to identify the children of u, remove these triplets from T and remove their representatives from Q_B. Since all except at most two of the triplets that it traverses are removed, we can charge their traversal and removal to their insertion in a previous FR-ExtractMin or FR-Activate.

9.5 Chapter Notes

The algorithm in this chapter is due to Fakcharoenphol and Rao [Fakcharoenphol and Rao, 2006]. It has been used creatively and slightly extended in many subsequent works, e.g., the minimum-cut oracle of Borradaile et al [Borradaile et al., 2010], the fast minimum-cut of Italiano et al [Italiano et al., 2011], maximum flow with multiple sources and sinks [Borradaile et al., 2011], and distance oracles [Fakcharoenphol and Rao, 2006, Cabello, 2012, Mozes and Sommer, 2012, Kaplan et al., 2012]. For a detailed treatment of the generalization of the algorithm to work on r–divisions with a constant number of holes, as well as for the Monge range-minimum query used by the algorithm, see the journal version of [Kaplan et al., 2012].