4 Graphs

Graphs are a fundamental model for representing various relations between data. The aim of this chapter is to present some basic problems and techniques relating to graphs, mainly for finding particular paths in directed and undirected graphs. In the following chapters, we will deal with various problems in biological sequence analysis that can be reduced to one of these basic ones.

Unless stated otherwise, in this chapter we assume that the graphs have n vertices and m arcs.

4.1 Directed acyclic graphs (DAGs)

A directed graph is called *acyclic* if it does not contain a directed cycle; we use the shorthand DAG to denote a directed acyclic graph. DAGs are one of the most basic classes of graphs, being helpful also in many problems in bioinformatics. DAGs admit special properties that not all graphs enjoy, and some problems become simpler when restricted to DAGs.

4.1.1 Topological ordering

The acyclicity of the arc relation of a DAG allows us to build recursive definitions and algorithms on its vertices. In such a case, the value of a function in a vertex v of a DAG depends on the values of the function on the in-neighbors of v. In order to implement its computation, we need an ordering of the vertices of the DAG such that any vertex appears in this ordering after all its in-neighbors. Such an ordering is called a *topological ordering* of the DAG. See Figure 4.1 for an example.

Any DAG admits a topological ordering, but it need not be unique. Exercise 4.1 asks the reader to construct a DAG in which the number of topological orderings is exponential in the number of vertices. A consequence of the existence of a topological ordering is that every DAG has at least one source (the first node in a topological ordering) and at least one sink (the last node in a topological ordering).

A topological ordering can be found in time linear in the number of vertices and arcs of the DAG. This can be done by performing a depth-first search (DFS): when all the vertices reachable from a vertex v have been visited, v is added at the beginning of a queue. At the end, the queue traversed from the beginning to its last element gives a topological ordering.

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

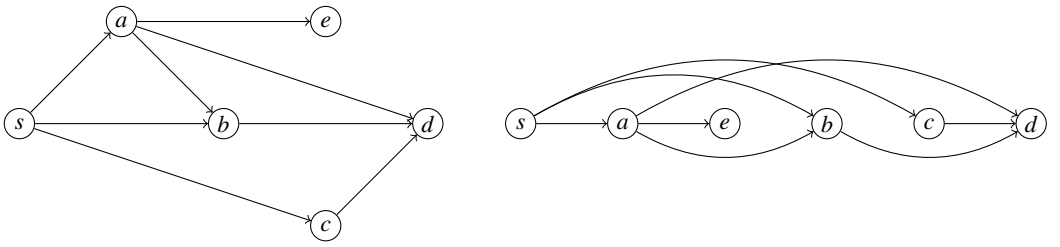

Figure 4.1 On the left, a DAG; on the right, a topological ordering of it.

Algorithm 4.1: Depth-first visit of a graph

Input: An undirected graph $G = (V, E)$.
Output: For every vertex $v \in V$, its value `finishTime`.

```
1  foreach v ∈ V do
2      visited(v) ← false;
3  time ← 0;
4  def DFS-from-vertex(G, v):
5      visited(v) ← true;
6      foreach u ∈ N+(v) do
7          if not visited(u) then
8              DFS-from-vertex(G, u);
9      finishTime(v) ← time;
10     time ← time + 1;
11 def DFS(G):
12     foreach v ∈ V do
13         if not visited(v) then
14             DFS-from-vertex(G, v);
15     return finishTime;
```

THEOREM 4.1 *A topological ordering of a DAG G can be computed in time $O(n+m)$.*

Proof We perform a DFS as in Algorithm 4.1. Let $v_1, \ldots, v_n$ be the order of the vertices in decreasing order of `finishTime`. Since we record the finish time of v_i only after all out-neighbors of v_i have been visited, v_i appears in the ordering $v_1, \ldots, v_n$ after all its out-neighbors. □

4.1.2 Shortest paths

One of the most basic problems on a graph is that of finding the shortest path between two vertices. Given a DAG G, we show that we can find in time $O(n + m)$ the shortest paths from a given vertex s to all other vertices of G. The acyclicity of G (in particular,

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

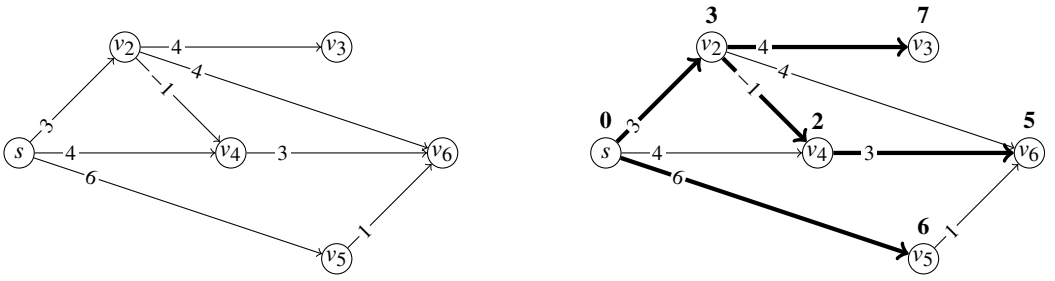

Figure 4.2 On the left, a DAG whose arcs are associated costs; on the right, each vertex v_i is labeled with $d(i)$; for every vertex we highlight an incoming arc minimizing (4.1).

the existence of a topological ordering) allows for such an algorithm even if the arcs have arbitrary positive or negative costs. In this case, the length (which we will also call the *weight*, or *cost*) of a path is the sum of the lengths (or weights, or costs) of its arcs.

Problem 4.1 Shortest path in a DAG

Given a DAG $G = (V, E)$, a vertex $s \in V$, and a cost function $c : E \to \mathbb{Q}$, find the paths of minimum cost from s to all other vertices $v \in V$.

THEOREM 4.2 *The shortest-path problem on a DAG can be solved in time $O(n + m)$.*

Proof A shortest path starting at s and ending at v must arrive at v from one of the in-neighbors, say u, of v. If the shortest path from s to u is already known, then a shortest path from s to v coincides with it, and then proceeds to v.

Since G is acyclic, the optimal solution for all the in-neighbors of v can be computed without considering v (and all other vertices further reachable from v). Thus, we can decompose the problem along *any* topological ordering $v_1, \ldots, v_n$ of the vertices of G.

For every $i \in \{1, \ldots, n\}$, we define

$$d(i) = \text{the length of the shortest path from } s \text{ to } v_i,$$

where $d(i) = \infty$ if no path from s to v_i exists. We compute $d(i)$ as

$$d(i) = \min\{c(v_j, v_i) + d(j) \mid v_j \in N^-(v_i)\}, \tag{4.1}$$

by initializing $d(i) = 0$, if $v_i = s$, and $d(i) = \infty$ otherwise. (Recall that $N^-(v_i)$ denotes the set of in-neighbors of v_i.) Since $v_1, \ldots, v_n$ is a topological ordering of G, this recursive definition makes sense. In order to compute also one shortest path, we can also store, for every $i \in \{1, \ldots, n\} \setminus \{s\}$, one in-neighbor v_j of v_i minimizing (4.1).

Computing a topological ordering of G can be done in time $O(n + m)$, by Theorem 4.1. Every arc (v_j, v_i) of G is inspected only once in the computation of d, namely when computing $d(i)$. Thus the complexity of this algorithm is $O(n + m)$. □

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

The solution of the shortest-path problem on a DAG is an elementary instance of *dynamic programming*, where one exploits the fact that the optimal solution to a larger instance (like the shortest path to the current vertex) can be constructed from the optimal solutions to smaller instances (the shortest paths to the in-neighbors of the current vertex). The *evaluation order* of a dynamic programming algorithm refers to the order in which the solutions to all required smaller instances become available, before computing the larger instance. In the case of DAGs, the topological ordering equals the evaluation order. We shall explore this connection in Chapter 6, but some of the exercises in this section already refer to dynamic programming.

4.2 Arbitrary directed graphs

4.2.1 Eulerian paths

A special kind of path (possibly with repeated vertices) in a connected directed graph is one using *every* arc *exactly* once, called an Eulerian path. An Eulerian path in which its first and last vertices coincide is called an *Eulerian cycle*. We say that a directed graph is *connected* if the undirected graph obtained from it by ignoring the orientation of its arcs is connected. A connected directed graph is said to be Eulerian if it has an Eulerian cycle.

Problem 4.2 Eulerian cycle

Given a connected directed graph G, check if G is Eulerian, and output an Eulerian cycle if this is the case.

THEOREM 4.3 *A connected directed graph G is Eulerian if and only if $|N^-(v)| = |N^+(v)|$ for every vertex v. When this is the case, an Eulerian cycle can be computed in time $O(m)$.*

Proof If G is Eulerian, then $|N^-(v)| = |N^+(v)|$ must hold for every vertex v.

For the reverse implication, we proceed as follows. Start at an arbitrary vertex $v_0 \in V(G)$, and construct a cycle C (possibly with repeated vertices) as follows. Pick an arbitrary out-neighbor v_1 of v_0 and visit v_1 by removing the arc (v_0, v_1), and adding it to C. Since $|N^-(v_1)| = |N^+(v_1)|$ held before removing the arc (v_0, v_1), there exists an out-neighbor v_2 of v_1 in the graph. Remove again the arc (v_1, v_2), add it to C, and repeat this process from v_2.

Since the number of vertices of G is finite, this procedure stops on reaching v_0. If C already contains all the arcs of G, then we are done. Otherwise, thanks to the connectivity assumption (Exercise 4.7 asks the reader to detail how connectivity helps here), there must be a vertex w_0 on C for which $|N^-(w_0)| = |N^+(w_0)| \geq 1$. Repeat the above procedure starting at w_0 to obtain another cycle C'. Combine C and C' into one

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

Figure 4.3 Combining two cycles C and C' into C''.

cycle C'' formed by taking the path from v_0 to w_0 on C, then the cycle C', then the path from w_0 to v_0 on C (see Figure 4.3).

We can iterate this process by checking again whether there is a vertex with at least one remaining out- and in-neighbor. This procedure terminates since the number of vertices of G is finite. At the end, we have constructed an Eulerian cycle of G.

Observe that every arc of G is visited exactly once. Exercise 4.8 ask the reader to show that, using doubly-linked lists, the above procedure can be implemented in time $O(m)$. □

A consequence of the above theorem is the following corollary about an Eulerian path with different endpoints.

COROLLARY 4.4 *A connected directed graph G has an Eulerian path with different endpoints if and only if there exist $s, t \in V(G)$ such that*

- $|N^-(s)| + 1 = |N^+(s)|$,
- $|N^-(t)| = |N^+(t)| + 1$, *and*
- $|N^-(v)| = |N^-(v)|$, *for all* $v \in V(G) \setminus \{s, t\}$.

In this case, we can find an Eulerian path in time $O(m)$.

Proof If a connected directed graph G has an Eulerian path with different endpoints, then the above three conditions must hold.

For the reverse implication, add the arc (t, s) to G and find an Eulerian cycle in time $O(m)$ by Theorem 4.3. □

4.2.2 Shortest paths and the Bellman–Ford method

The algorithm given in Section 4.1.2 for the shortest-path problem does not work for arbitrary graphs, because one can no longer decompose the computation along a topological ordering. However, we can still decompose the problem by adding another parameter to a partial solution ending at a vertex v: the number of arcs of a path from s to v.

We restrict the problem to directed graphs without negative cost cycles.

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

Problem 4.3 Shortest path in a directed graph without negative cycles

Given a directed graph $G = (V, E)$, a vertex $s \in V$, and a cost function $c : E \to \mathbb{Q}$ such that G has no negative cost cycles, find the paths of minimum cost from s to every vertex $v \in V$.

THEOREM 4.5 (Bellman–Ford algorithm) *The shortest-path problem on a directed graph G without negative cycles can be solved in time $O(nm)$.*

Proof Let $v_1, \ldots, v_n$ be an arbitrary ordering of the vertices of G. To simplify notation in what follows we may assume $s = v_1$. For every $i \in \{1, \ldots, n\}$, we define

$$d(i,k) = \text{the length of a shortest path from } s \text{ to } v_i \text{ having at most } k \text{ arcs}, \tag{4.2}$$

and $d(i,k) = \infty$ if no path having at most k arcs exists from s to v_i. Since G is assumed not to have negative cost cycles, any shortest path from s to any other vertex has at most $n-1$ arcs. Thus, the solution is contained in $d(i, n-1)$, for every $i \in \{1, \ldots, n\}$. See Figure 4.4 for an example.

For each $k \in \{1, \ldots, n-1\}$ in increasing order, we compute $d(i,k)$ as

$$d(i,k) = \min\left(\min\{d(j,k-1) + c(v_j, v_i) \mid v_j \in N^-(v_i)\}, \{d(i,k-1)\}\right), \tag{4.3}$$

where $d(1,0) = 0$, and $d(i,0) = \infty$ for every $i \in \{2, \ldots, n\}$. In other words, a shortest path from s to v_i having at most k arcs is either

- a shortest path from s to an in-neighbor v_j of v_i having at most $k-1$ arcs, plus the arc (v_j, v_i), or
- a shortest path from s to v_i having at most $k-1$ arcs.

Observe that this recursive definition makes sense because a solution for a subproblem parametrized by k depends only on the solutions to subproblems parametrized by $k-1$, which have already been computed.

In order to compute also one shortest path, for every (i,k), $k > 1$, we store not only $d(i,k)$, but also the parent pair $p(i,k) = (j,k')$ minimizing (4.3).

For every $k \in \{1, \ldots, n-1\}$, every arc (v_j, v_i) of G is inspected only once, when computing $d(i,k)$. Therefore, the complexity of the entire algorithm is $O(nm)$. □

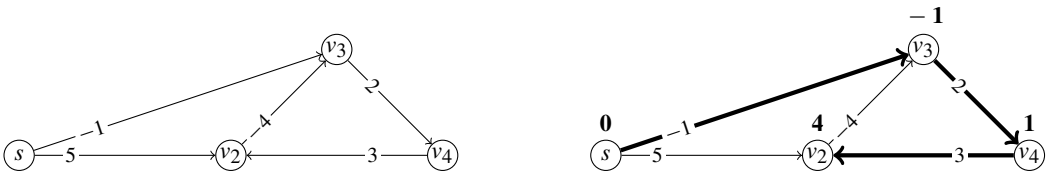

Figure 4.4 On the left, a directed graph whose arcs are associated costs; on the right, each vertex v_i is labeled with $d(i, n-1)$; for every vertex we highlight an incoming arc minimizing (4.3) when $k = n-1$.

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

The proof of Theorem 4.5 can also be interpreted as the operation of creating a DAG consisting of $n-1$ copies of the vertices of G, one for each value of k, and making each arc (v_j, v_i) go from each v_j in one copy to the v_i in the next copy. More precisely, we can construct the graph G^* (see Figure 4.5):

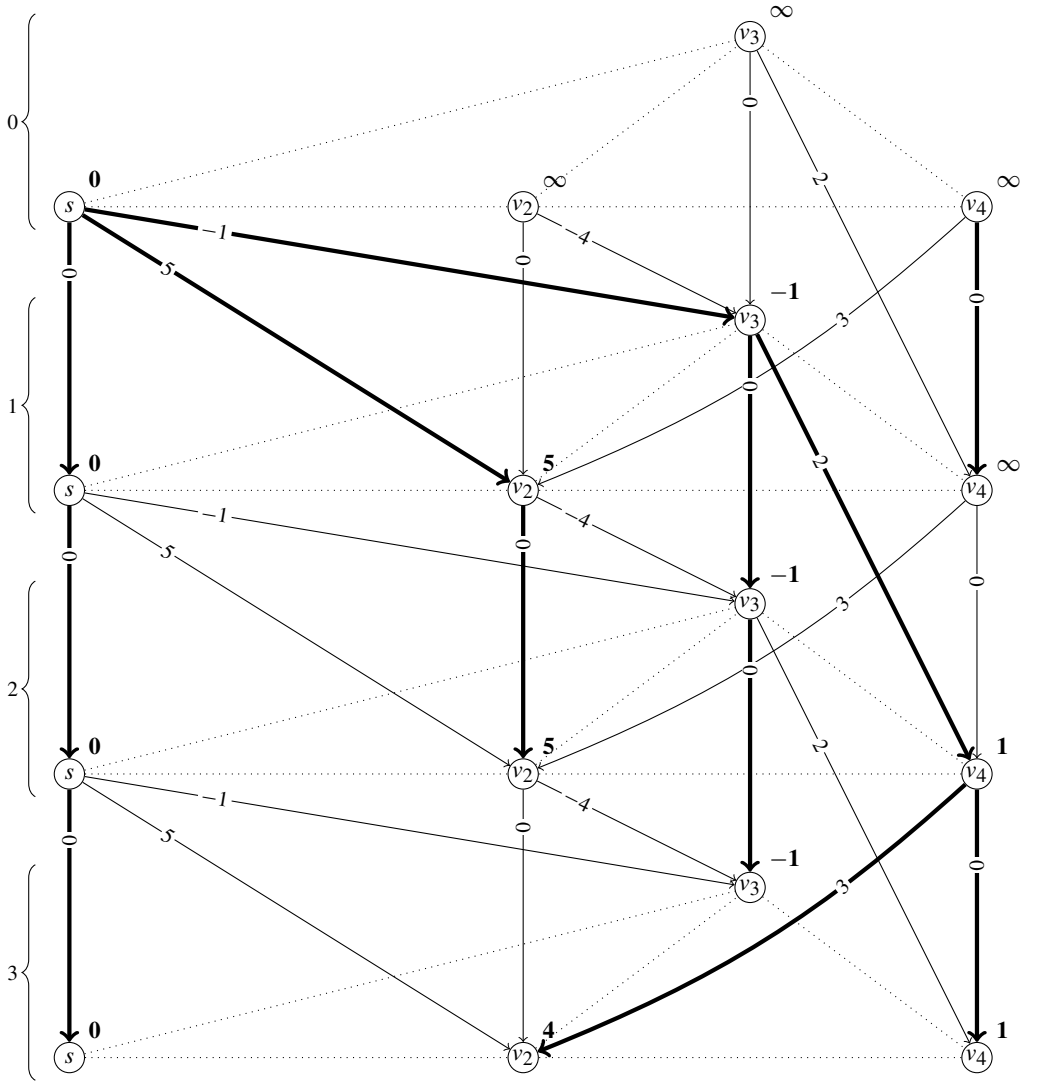

Figure 4.5 A reduction of the shortest-path problem in an arbitrary graph without negative cycles to the shortest-path problem in a DAG; each vertex (v_i, k) is labeled with $d(i, k)$; for every vertex we highlight an incoming arc minimizing (4.1).

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

- for every vertex $v_i \in V$, we add n vertices named $(v_i, 0), \ldots, (v_i, n-1)$;
- for every arc (v_j, v_i), we add the $n-1$ arcs $\big((v_j, 0), (v_i, 1)\big), \ldots, \big((v_j, n-2), (v_i, n-1)\big)$, each of cost $c(v_j, v_i)$; and
- for every vertex $v_i \in V$, we add $n-1$ arcs $\big((v_i, 0), (v_i, 1)\big), \ldots, \big((v_i, n-2), (v_i, n-1)\big)$, each of cost 0.

The shortest path from s to each of the vertices $(v_1, n-1), \ldots, (v_n, n-1)$ gives the shortest paths from s to each of $v_1, \ldots, v_n$, respectively. The DAG G^* has $n(n-1)$ vertices and $m(n-1)$ arcs. Assuming that G is connected (otherwise we can apply this reduction to each connected component of G), we have $m \geq n-1$. Thus, Theorem 4.5 also follows directly from Theorem 4.2.

The Bellman–Ford method also allows one to check whether a directed graph $G = (V, E)$, with a cost function $c : E \to \mathbb{Q}$, has a negative cost cycle.

Problem 4.4 Detection of a negative cycle

Given a directed graph G, and a cost function $c : E(G) \to \mathbb{Q}$, check whether G has a negative cost cycle, and output one if this is the case.

THEOREM 4.6 *Given a directed graph G, and a cost function $c : E(G) \to \mathbb{Q}$ the problem of detection of negative cycles on G can be solved in time $O(nm)$.*

Proof First, assume that the underlying graph of G is connected, otherwise we can work on each connected component of it. This assumption implies that $m \geq n-1$. We also assume that there is a vertex s from which all other vertices are reachable, otherwise we can add a dummy $(n+1)$th vertex s with arcs of cost 0 to all other vertices of G.

Let $v_1, \ldots, v_n$ be an arbitrary ordering of the vertices of G. For every $i \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, n\}$ we compute $d(i, k)$ (lengths of shortest paths starting from s) as in the proof of Theorem 4.5, in time $O(nm)$.

If $d(i, n) < d(i, n-1)$ for some $i \in \{1, \ldots, n\}$, then, following the parent links from v_i, we can find a path P consisting of exactly n arcs, starting at s and ending at v_i. Since P has $n+1$ vertices, at least one vertex must appear twice on P. Thus, P contains a cycle C made up of the arcs of P between two consecutive occurrences of a vertex v such that no other vertex between these two occurrences is repeated. On removing C from P we obtain a path P' with at most n arcs.

For any path Q in G, let $c(Q)$ denote the sum of the costs of its arcs. Since $c(P') \geq d(i, n-1) > d(i, n) = c(P)$ and $c(P) = c(P') + c(C)$, we get $c(C) < 0$. Therefore, we can output the cycle C of negative cost.

Otherwise, $d(i, n) = d(i, n-1)$ holds for all $i \in \{1, \ldots, n\}$, and we must prove that G cannot have a negative cost cycle. Thanks to the Bellman–Ford recurrence (4.3), this implies that

$$d(j, n) + c(v_j, v_i) = d(j, n-1) + c(v_j, v_i) \geq d(i, n), \text{for every arc } (v_j, v_i) \text{ of } G.$$

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

We argue by contradiction, and suppose that $v_{i_0}, v_{i_1}, \ldots, v_{i_\ell} = v_{i_0}$ is a cycle of negative cost $\sum_{t=1}^{\ell} c(v_{i_{t-1}}, v_{i_t}) < 0$. We sum up the above inequality for all vertices in the cycle and obtain

$$\sum_{t=1}^{\ell} d(i_t, n) \leq \sum_{t=1}^{\ell} \left(d(i_{t-1}, n) + c(v_{i_{t-1}}, v_{i_t}) \right) = \sum_{t=1}^{\ell} d(i_{t-1}, n) + \sum_{t=1}^{\ell} c(v_{i_{t-1}}, v_{i_t}).$$

Observe that $\sum_{t=1}^{\ell} d(i_t, n) = \sum_{t=1}^{\ell} d(i_{t-1}, n)$, which implies that $\sum_{t=1}^{\ell} c(v_{i_{t-1}}, v_{i_t}) \geq 0$, contradicting the initial assumption that the cycle $v_{i_0}, v_{i_1}, \ldots, v_{i_\ell} = v_{i_0}$ has negative cost. □

4.3 Literature

The reader can refer to the textbook *Introduction to Algorithms* (Cormen *et al*. 2009) for a gentle introduction to the algorithms presented here, or to the textbook *Introduction to Graph Theory* (West 2000) and the monograph *Combinatorial Optimization* (Schrijver 2003) for a more in-depth discussion. The Bellman–Ford method appeared in Shimbel (1955), Bellman (1958), Ford (1956), and Moore (1959).

Exercises

4.1 Find a family of DAGs in which the number of topological orderings is exponential in the number of vertices.

4.2 Find a family of DAGs with a number of distinct paths exponential in the number of vertices.

4.3 Show that a directed graph G is acyclic (does not contain a directed cycle) if and only if, for every subset of vertices $S \subseteq V(G)$, there exists a vertex $v \in S$ such that no out-neighbor of v belongs to S. Conclude that a DAG must have at least one sink.

4.4 Show that a directed graph G is acyclic if and only if it admits a topological ordering (the forward implication is Theorem 4.1). Conclude that we can check in $O(m)$ time whether a directed graph is acyclic.

4.5 Let G be a DAG with precisely one source s and one sink t. Show that for any $v \in V(G) \setminus \{s, t\}$ there exists a path from s to v and a path from v to t.

4.6 Let G be a DAG with a unique source. Prove that G is connected.

4.7 Explain in detail how the connectivity assumption is exploited in the proof of Theorem 4.3.

4.8 Show how the procedure in Theorem 4.3 can be implemented in time $O(m)$.

4.9 Consider a directed graph G in which for every pair of vertices $u, v \in V(G)$ exactly one of $(u, v) \in E(G)$ or $(v, u) \in E(G)$ holds (such a graph is called a *tournament*). Show that there exists a path in G passing through all vertices exactly once (such a path is called *Hamiltonian*).

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

4.10 Let G be an undirected graph. Show that either G or its complement $\overline{G}$ is connected.

4.11 Give an algorithm running in time $O(n + m)$ that checks whether an undirected graph G is connected, and if not, lists all of its connected components.

4.12 A connected undirected graph G is called *2-connected* (or *bi-connected*) if removing any vertex from G (along with its incident edges) results in a connected graph. Give an algorithm running in time $O(m)$ for checking whether a graph is 2-connected.

4.13 Refine the algorithm from Exercise 4.12 so that it also outputs the 2-connected components of G (that is, the maximal 2-connected subgraphs of G).

4.14 Given an undirected graph G, consider the graph $B(G)$ of 2-connected components of G: the vertices of $B(G)$ are the 2-connected components of G, and we add an edge between two vertices of $B(G)$ if the corresponding 2-connected components of G have a vertex in common. Show that $B(G)$ is a tree.

4.15 Consider a directed graph G, a vertex $s \in V(G)$, and $c : E(G) \to \mathbb{Q}$ so that no cycle of G has negative cost. Show that if G has only two 2-connected components C_1 and C_2, sharing a vertex v, such that $s \in V(C_1)$, then we can solve the shortest-path problem on G by solving it independently on C_1 and C_2, by appropriately initializing the dynamic programming algorithm on C_2 at vertex v.

4.16 Consider a DAG G, a vertex $s \in V(G)$, a cost function $c : V(G) \to \mathbb{Q}$, a partition $\mathcal{S} = \{S_1, \ldots, S_k\}$ of $V(G)$, and an integer $t \leq k$. Give a dynamic programming algorithm for computing a shortest path from s to any other vertex of G that passes through at most t sets of the partition $\mathcal{S}$. What is the complexity of your algorithm? What if G is not acyclic?

4.17 Consider a DAG G, a partition $\mathcal{S} = \{S_1, \ldots, S_k\}$ of $V(G)$, and an integer $t \leq k$. We say that a path $P = u_1, u_2, \ldots, u_\ell$ in G is *t-restricted* if every maximal subpath of P of vertices from the same set of $\mathcal{S}$ has at least t vertices. (In other words, if P starts using vertices from a set $S_i \in \mathcal{S}$, then it must do so for at least t vertices.) Give a dynamic programming algorithm that is additionally given a vertex $s \in V(G)$ and a cost function $c : V(G) \to \mathbb{Q}$, and finds a t-restricted shortest path from s to any other vertex of G. What is the complexity of your algorithm? What if G is not acyclic?

4.18 Given a directed graph $G = (V, E)$, $n = |V|$, $m = |E|$, and a cost function $c : E \to \mathbb{Q}$, we say that the *length* of a cycle $C = v_1, v_2, \ldots, v_t, v_{t+1} = v_1$ in G, denoted $l(C)$ is t, its *cost*, denoted $c(C)$, is $\sum_{i=1}^{t} c(v_i, v_{i+1})$, and its *mean cost* is $\mu(C) = c(C)/l(C)$. Denote by $\mu(G)$ the minimum mean cost of a cycle of G, namely

$$\mu(G) = \min_{C \text{ cycle of } G} \mu(C).$$

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press

(*a*) For each $v \in V(G)$, and each $k \in \{0, \ldots, n\}$, let $d(v, k)$ be the minimum cost of a path in G with exactly k edges, ending at v (where $d(v, 0) = 0$ by convention). Show that the bi-dimensional array d can be computed in time $O(nm)$.

(*b*) Show that

$$\mu(G) = \min_{v \in V} \max_{0 \le k \le n-1} \frac{d(v, n) - d(v, k)}{n - k}, \tag{4.4}$$

by showing the following facts. Consider first the case $\mu(G) = 0$, and show that

- for any $v \in V$, there exists a $k \in \{0, \ldots, n-1\}$ such that $d(v, n) - d(v, k) \geq 0$, thus, the right-hand side of (4.4) is greater than or equal to 0;
- each cycle of G has non-negative cost: if C is a cycle of G, then there exists a vertex v on C such that, for every $k \in \{0, \ldots, n-1\}$, it holds that $d(v, n) - d(v, k) \leq c(C)$; and
- a cycle C of minimum mean cost $\mu(C) = 0$ also has $c(C) = 0$; use the above bullet to show that the right-hand side of (4.4) is equal to 0.

Conclude the proof of (4.4) by considering the case $\mu(G) \neq 0$. Show that

- if we transform the input (G, c) into (G, c') by subtracting $\mu(G)$ from the cost of every edge, then the minimum mean cost of a path of (G, c') is 0, and the paths of minimum mean cost of (G, c) are the same as those of (G, c'); and
- since relation (4.4) holds for (G, c'), it holds also for the original input (G, c).

(*c*) Use (*a*) and the proof of (*b*) to conclude that a minimum mean cost cycle C in G can be found in time $O(nm)$.

https://doi.org/10.1017/CBO9781139940023.006 Published online by Cambridge University Press