

Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πανεπιστήμιο Πατρών

ΣΗΜΕΙΩΣΕΙΣ

Αλγόριθμοι και Συνδυαστική Βελτιστοποίηση

Χρήστος Ζαρολιάγκης

Οκτώβριος 2004

Πρόλογος

Οι σημειώσεις καλύπτουν το πρώτο τμήμα του βασικού μαθήματος επιλογής 505 “Αλγόριθμοι και Συνδυαστική Βελτιστοποίηση”, που διδάσκεται στους φοιτητές του Τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Πατρών. Οι σημειώσεις αφορούν κλασσικές και προηγμένες αλγοριθμικές τεχνικές για την αποδοτική επίλυση τριών θεμελιωδών προβλημάτων βελτιστοποίησης δικτύων: εύρεση συντομότερων διαδρομών, εύρεση μέγιστης ροής, και εύρεση μέγιστης ροής ελάχιστου κόστους. Εκτός από την ερευνητική και διδακτική εμπειρία του συγγραφέα σε τέτοια ζητήματα, πολύτιμες πηγές για αυτές τις σημειώσεις αποτέλεσαν τα βιβλία [1, 2, 3, 4, 5], τα οποία ο αναγνώστης μπορεί να συμβουλευθεί για περαιτέρω εμβάθυνση.

Οκτώβριος 2004

Χρήστος Δ. Ζαρολιάγκης

Contents

Πρόλογος	3
1 Introduction	7
1.1 A Sample of Applications	8
2 Preliminaries and Fundamental Algorithms	15
2.1 Basic Graph-Theoretic Definitions	15
2.2 Graph and Network Representation	20
2.3 Complexity of Algorithms	22
2.4 Fundamental Graph Algorithms	25
3 Shortest Paths	29
3.1 Introduction	29
3.2 The SSSP Problem	32
3.3 Optimality Conditions	33
3.4 A Generic Label-Correcting Algorithm	35
3.5 An Improved Implementation of the Label-Correcting Algorithm	38
3.6 Detecting Negative Cycles	39
3.7 The SSSP Problem: Special Cases	40
3.7.1 Acyclic Networks	40
3.7.2 Non-Negative Edge Weights: Dijkstra's Algorithm	41
3.7.3 Heap Implementations of Dijkstra's Algorithm	43
3.7.4 Integer Edge Weights: Dial's Algorithm	45
3.7.5 Integer Edge Weights: Radix Heap Implementation	46
3.8 All-Pairs Shortest Paths	49
3.8.1 Solving APSP through SSSP	50
3.8.2 APSP through Label-Correcting Algorithms	52

4	Maximum Flows: Basic Algorithms	57
4.1	Introduction	57
4.2	Preliminaries and the Max-Flow Min-Cut Theorem	58
4.3	Four Fundamental Theorems	61
5	Maximum Flows: Improved Algorithms	67
5.1	Introduction	67
5.2	Maximum Capacity and Capacity Scaling Algorithms	68
5.3	Shortest Augmenting Path Algorithm	70
5.3.1	Distance Labels, Admissible Edges and Admissible Paths	71
5.3.2	The Algorithm	72
5.4	Preflow-Push Algorithms	77
6	Minimum Cost Flows	85
6.1	Introduction	85
6.2	Optimality conditions	86
6.3	Optimal Flows and Optimal Vertex Potentials	88
6.4	Algorithms for Finding a Min-Cost Flow	90
	Bibliography	95

Chapter 1

Introduction

A *graph* G is a pair of sets (V, E) , where V is the set of *vertices* (or nodes) and E is a set of pairs of distinct vertices called *edges* (or arcs). A *network* is a graph whose vertices and edges are associated with certain numerical values. Graphs and networks are two of the most useful mathematical objects for representing many physical systems. Except for computer science, they appear in numerous fields, including (among others) applied mathematics, electrical/mechanical/civil engineering, chemistry, management and operations research.

Networks (as an abstract mathematical term) take their name from the physical networks that appear in the real world: electric/power networks, computer networks, telephone networks, transportation networks, manufacturing/distribution networks, etc.

In all of the physical (real world) networks, we want to move some entity (e.g., electricity, message, vehicle, product) from one point to another in the network. And we want to do it as efficiently as possible satisfying a two-fold purpose: (a) to provide good service to the end-users; and (b) to use the underlying (and usually expensive) transmission facilities in an effective way.

This objective is the goal of this course. We will consider classical and advanced algorithmic techniques to tackle efficiently several fundamental problems on graphs and networks. The main part of the course will be network optimization problems, an area with rich and long tradition due to its fundamental importance. More precisely, we will address three problems:

- *shortest path problem*: which is the cheapest way to move from one point of a network to another, assuming that the network has costs on its edges.

- *maximum flow problem*: how much flow can we send from one point of a network to another, assuming certain capacity values on the network's edges.
- *minimum cost flow problem*: if the network imposes a cost per unit of flow, then which is the maximum flow that we can send from one point of the network to another at the minimum possible cost? The minimum cost flow problem constitutes a synthesis of the shortest path and maximum flow problems.

1.1 A Sample of Applications

Networks arise in numerous application settings. The following table shows the ingredients of some common physical networks and their correspondence to vertices, edges and flows of (abstract) networks.

Applications	Physical analog of nodes/vertices	Physical analog of arcs/edges	Flow
Communication Systems	Telephone exchanges, computers, transmission facilities, satellites	Cables, fiber optic links, microwave relay links	Voice messages, data, video transmissions
Hydraulic Systems	Pumping stations, Reservoir, lakes	Pipelines,	Water, gas, oil hydraulic fluids
Integrated Circuits	Gates, registers, processors	Wires	Electrical current
Mechanical Systems	Joints	Rods, beams, strings	Heat, Energy
Transportation Systems	Intersections, airports rail yards	Highways, railbeds airline routes	Passengers, freight, vehicles, operators

We shall discuss two applications.

(A1) DNA Sequence alignment problem

Molecular biologists model strands of DNA as a sequence of letters over a specific alphabet. Given any two such sequences $B = b_1, b_2, \dots, b_p$ and $D = d_1, d_2, \dots, d_q$, scientists are interested to determine how similar are these sequences to each other. A natural way to do this is to determine the minimum “cost” required to transform B into D . The transformation can be done by the following operations:

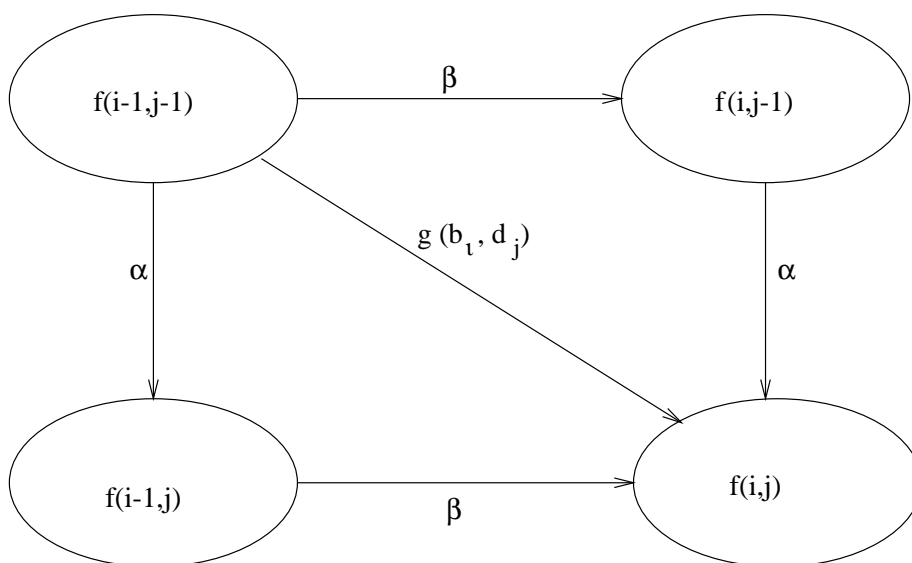


Figure 1.1: Basic network module for the DNA sequence alignment problem.

1. insert an element in B (at any place) at a cost of α units;
2. delete an element from B at a cost of β units;
3. mutate (i.e., change) an element b_i into an element d_j at a cost of $g(b_i, d_j)$ units.

Since it is possible to transform B into D in many ways, we want to find the transformation of total minimum cost. Let $f(i, j)$ denote the minimum cost of transforming the subsequence b_1, b_2, \dots, b_i into the subsequence d_1, d_2, \dots, d_j . The goal is to find $f(p, q)$. It suffices to find $f(i, j)$ for all $i = 0, 1, \dots, p$ and $j = 0, 1, \dots, q$. Clearly, $f(0, 0) = 0$. We also have:

- (a) $\forall i, f(i, 0) = \beta \cdot i$ (cost of transforming a sequence of i elements into a null sequence = cost of deleting i elements).
- (b) $\forall j, f(0, j) = \alpha \cdot j$ (cost of transforming the null sequence into one with j elements = cost of inserting j elements).

To determine $f(i, j)$, there are three cases to consider. Let B' be the optimal aligned sequence of B , before its transformation into D .

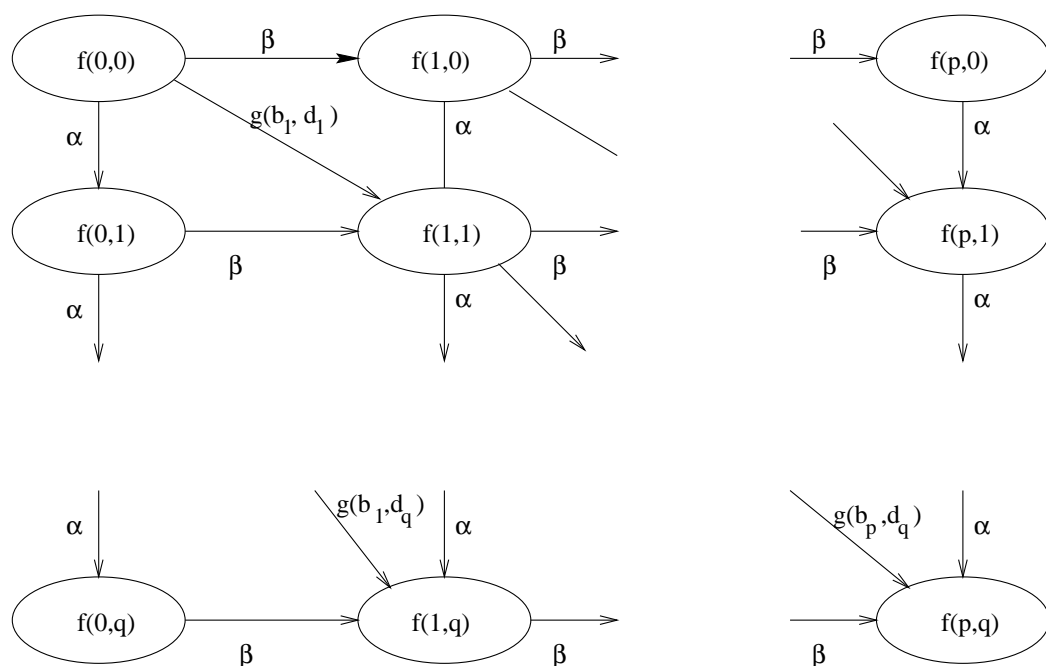


Figure 1.2: Complete network for the DNA sequence alignment problem.

1. B' contains b_i which is aligned to d_j :

$$\begin{aligned} B' &= b_1 \cdots b_{i-1} b_i \\ D &= d_1 \cdots d_{j-1} d_j \end{aligned}$$

Then, $f(i, j) = f(i - 1, j - 1) + g(b_i, d_j)$

2. B' contains b_i which is not aligned to d_j : then notice that b_i is on the left of d_j .

$$\begin{aligned} B' &= b_1 \cdots b_i \cdots \\ D &= d_1 \cdots d_j \cdots \end{aligned}$$

Hence, a newly added element must be aligned with d_j . Therefore, $f(i, j) = f(i, j - 1) + \alpha$.

3. B' does not contain b_i : this implies that b_i has been deleted from B and therefore the optimal cost of transformation equals the cost of deleting this element plus the cost of transforming the remaining sequence into D ; i.e., $f(i, j) = f(i - 1, j) + \beta$.

The above discussion implies that

$$(c) \quad f_{i,j} = \min\{f(i-1, j-1) + g(b_i, d_j), f(i, j-1) + \alpha, f(i-1, j) + \beta\}$$

We can represent (c) as a network with costs on its edges as shown in Figure 1.1. Based on this representation, we can construct the network shown in Figure 1.2 for the sequence alignment problem.

It is easy to verify that a shortest path in the above network, from vertex $f(0, 0)$ to $f(p, q)$, determines the minimum cost transformation of sequence B into D .

(A2) Scheduling on uniform parallel machines

Assume that we want to schedule J jobs on M uniform, parallel machines. Each job $j \in J$ has a:

- **processing requirement** p_j : machine time (hours) to complete the job.
- **release time** r_j : exact time (hour) that job j is available for processing.
- **due time** $d_j \geq r_i + p_j$: exact time (hour) by which the job j must be completed.

We will assume that a machine can work only one job at a time and each job can be processed by at most one machine. However, *preemptions* are allowed (i.e., interrupt a job and process it on different machines in different hours).

The *scheduling problem* is to determine a feasible schedule that completes all jobs before their due times, or to show that such a schedule does not exist. The scheduling problem reduces to a maximum flow problem. We will illustrate the reduction with an example.

Let $|M| = 3$ and $|J| = 4$. Consider the following data:

j	1	2	3	4
p_j	1.5	1.25	2.1	3.6
r_j	3	1	3	5
d_j	5	4	7	9

Rank (in ascending order) the release and due times. (In our example, 1,3,4,5,7,9). Determine $P \leq 2|J| - 1$ mutually disjoint intervals of time between consecutive milestones (i.e., release and due times). Let $T_{k,l}$ be the interval that starts at hour k and ends at hour $l + 1$. In our example, we

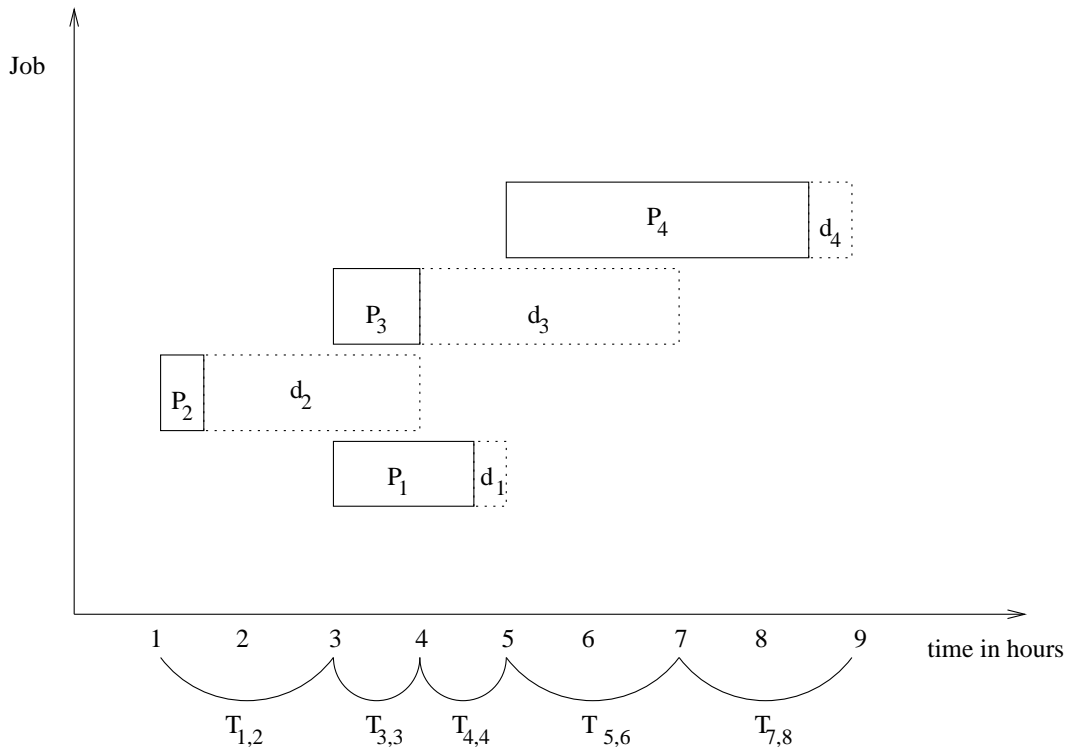


Figure 1.3: Time intervals.

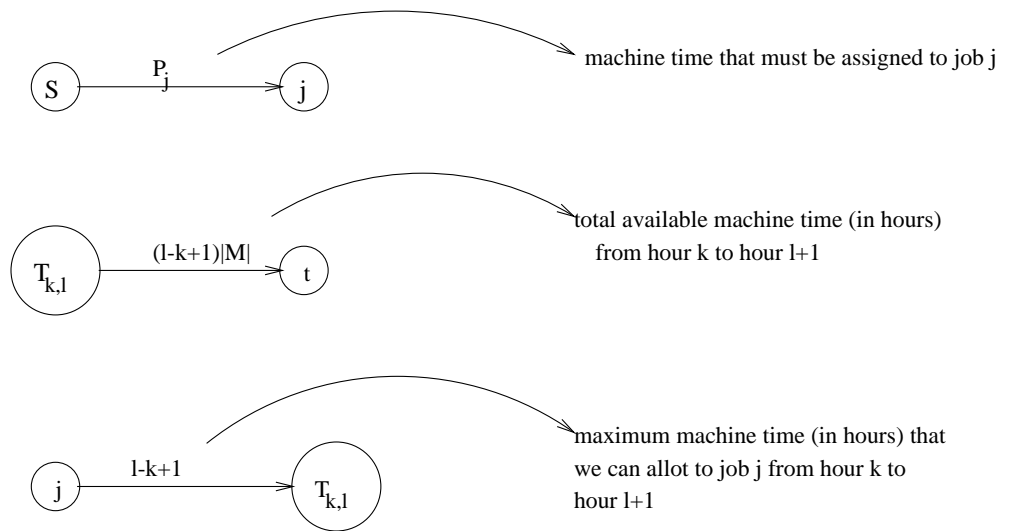


Figure 1.4: Capacities associated with the edges for the scheduling problem.

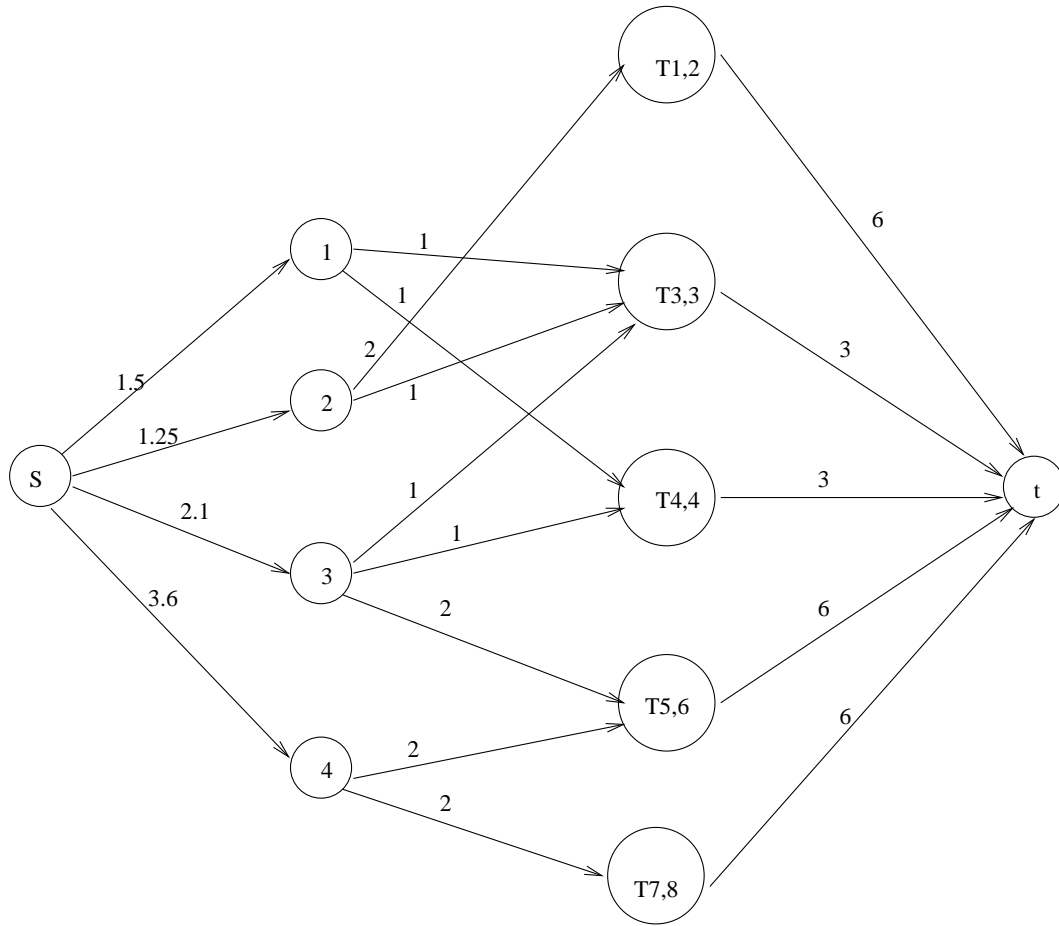


Figure 1.5: The complete network for the scheduling problem.

have 5 intervals as shown in Figure 1.3. We construct a network $G = (V, E)$ as follows.

$$V = J \cup \{T_{k,l} : T_{k,l} \text{ is a time interval}\} \cup \{s, t\}$$

where s and t are two special vertices, and

$$E = \{(s, j) : j \in J\} \cup \{(T_{k,l}, t) : T_{k,l} \text{ is a time interval}\} \\ \cup \{(j, T_{k,l}) : r_j \leq k \text{ and } d_j \geq l + 1\}$$

The capacities associated with the edges are shown in Figure 1.4. In our example, the resulting network G is illustrated in Figure 1.5. It can be easily

verified that a feasible schedule exists iff the value of the maximum flow in G equals $\sum_{j \in J} p_j$ ($= 8.45$, in our example).

Chapter 2

Preliminaries and Fundamental Algorithms

2.1 Basic Graph-Theoretic Definitions

A graph G is a pair of sets $G = (V, E)$, where V is the set of *vertices* (or nodes) and E is a set of pairs of distinct vertices called *edges* (or arcs). If the set E contains unordered (resp. ordered) pairs of vertices, the graph –as well as its edges– is/are called *undirected* (resp. *directed*); an example of graphs is presented in Figure 2.1. A directed (resp. undirected) *network* is a directed (resp. undirected) graph whose vertices and/or edges are associated with certain numerical values (e.g., costs, capacities, supplies/demands).

A directed edge (u, w) has two *endpoints* u and w , where u (resp. w) is called the *tail* (resp. *head*) of the edge. The edge (u, w) is incident to both u and w , and it is an *outgoing* or *outcoming* edge of vertex u and an *incoming* or *incoming* edge of vertex w . We also say that w is *adjacent* to u . An undirected edge (u, w) does not have tail and head and is treated as two directed edges (u, w) and (w, u) .

The *indegree* (resp. *outdegree*) of a vertex u is the number of incoming (resp. outgoing) edges of u . The *degree* of a vertex u in a directed graph is the sum of its indegree and outdegree. In an undirected graph, the degree of a vertex u is the total number of its adjacent vertices.

Two or more edges with the same endpoints are called *multiple edges*. An edge whose endpoints are the same vertex (e.g., (u, u)) is called a *loop*.

A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V', E')$ of G is called an *induced subgraph* on the set V' if $E' = \{(u, w) : \text{both } u \text{ and } w \in V'\}$. A graph $G' = (V', E')$

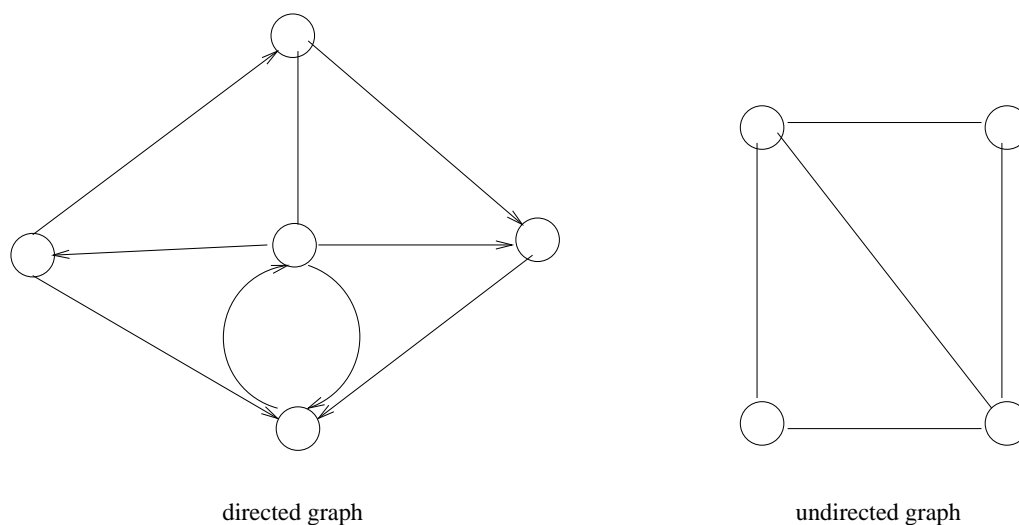


Figure 2.1: A directed and an undirected graph.

is called a *spanning subgraph* of a graph $G = (V, E)$, if $V' = V$ and $E' \subseteq E$. Figure 2.2 illustrates these definitions.

A *walk* in a graph $G = (V, E)$ is a sequence of vertices u_1, u_2, \dots, u_k , $k \geq 1$, such that $(u_i, u_{i+1}) \in E$ or $(u_{i+1}, u_i) \in E$ for $1 \leq i \leq k-1$. A *directed walk* is a walk u_1, u_2, \dots, u_k for which $(u_i, u_{i+1}) \in E$, $\forall 1 \leq i \leq k-1$. A (directed) *path* is a (directed) walk without repeated vertices. A (directed) *cycle* is a path u_1, u_2, \dots, u_k with $k > 2$ and $u_k = u_1$.

A directed graph $G = (V, E)$ is called *acyclic* (or DAG as a shorthand), if G does not contain directed cycles. A directed acyclic network is a network whose underlying graph is a DAG.

Two vertices u and w in a graph G are *connected* if G contains at least one path from u to w or from w to v . A graph is *connected* if every pair of its vertices is connected. Otherwise, it is called *disconnected*. A directed graph is called *strongly connected* if for every two of its vertices u and w , there is a directed path from u to w . Figure 2.3 illustrates these definitions.

A *cut* in a graph $G = (V, E)$ is a partition of the vertex set V into two sets S and $\bar{S} = V - S$. A cut defines a set of edges having one endpoint in S and the other in \bar{S} . We will refer to this set of edges as $[S, \bar{S}]$. Note that the set S determines uniquely a cut. An *s-t cut* is defined with respect to two distinguished vertices $s, t \in V$, and is a cut $[S, \bar{S}]$ such that $s \in S$ and $t \in \bar{S}$. The cut shown in Figure 2.4 is an 1-6 cut.

A *tree* is a connected graph that contains no cycles. A *forest* is a collec-

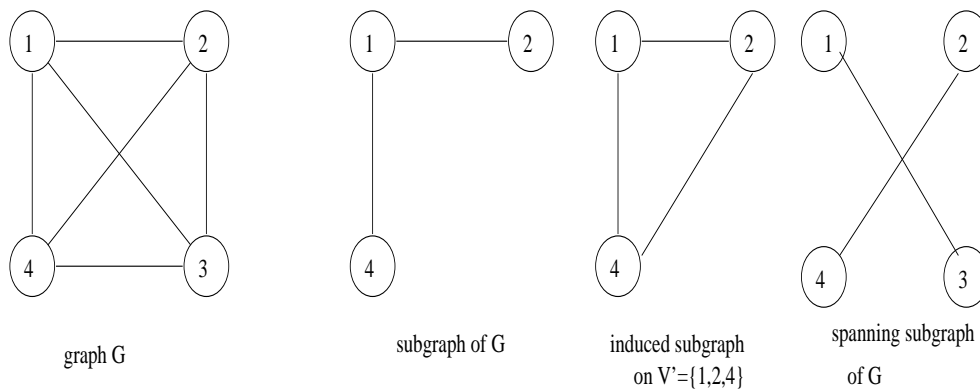


Figure 2.2: Various subgraphs of a graph.

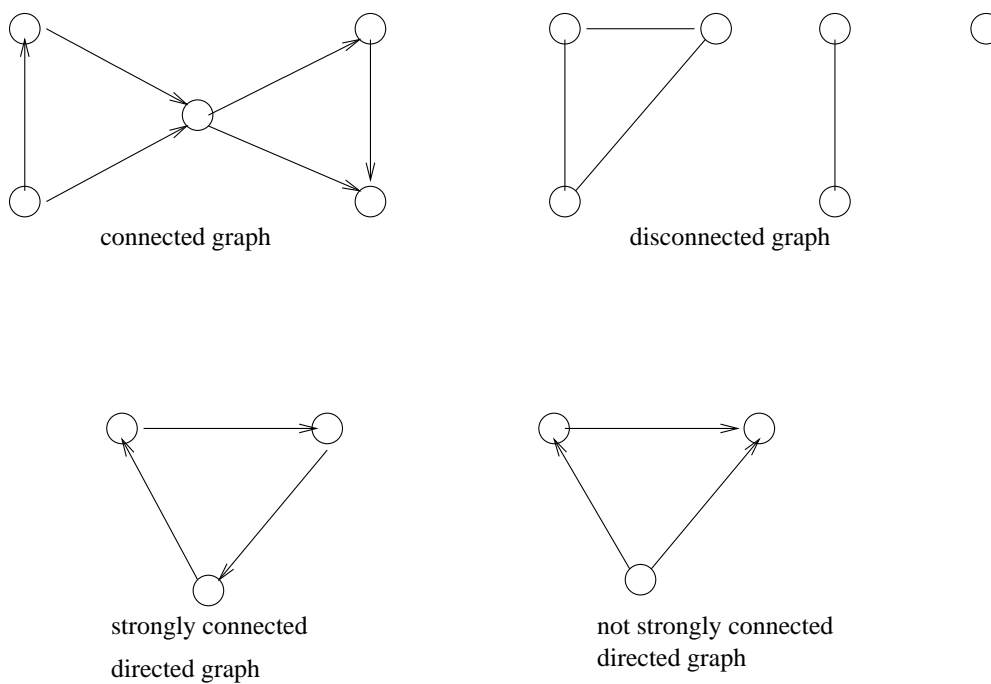


Figure 2.3: Graph connectivity.

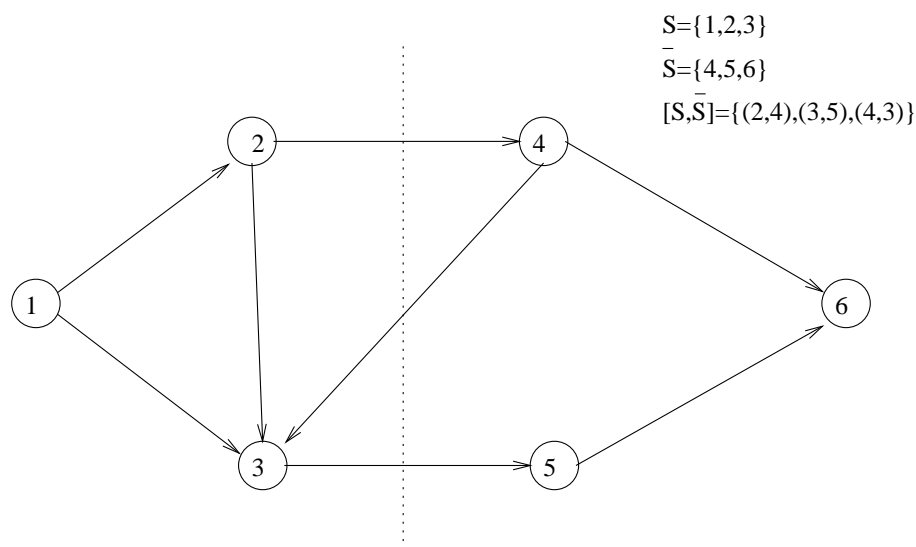


Figure 2.4: A cut in a graph.

tion of trees. A *subtree* is a connected subgraph of a tree. These definitions are illustrated in Figure 2.5.

EXERCISE 2.1. Prove that: (i) a tree with n vertices has exactly $n - 1$ edges; (ii) there are at least two leaves (i.e., vertices with degree 1) in a tree; (iii) every two vertices in a tree are connected by a unique path.

A *rooted tree* (see Figure 2.6) is a tree with a distinguished vertex called its *root*. In a rooted tree $T = (V, E)$ we view its edges as establishing parent-child relationships. In the unique path from the root r of T to any vertex u , every vertex (except for u) has a successor called its *child*, and every vertex (except for r) has a unique predecessor called its *parent*. The *level* of a vertex u in T (denoted as $\text{level}(u)$) is defined as the number of edges in the unique path from r to u . A vertex w is called *ancestor* (resp. *descendant*) of a vertex u , if $\text{level}(w) < \text{level}(u)$ (resp. $\text{level}(w) > \text{level}(u)$) in the unique path from r to u (resp. from r to w). A tree T is called a *spanning tree* of a graph G , if T is a spanning subgraph of G .

A graph $G = (V, E)$ is called *bipartite* if there is a partition of its vertex set into two sets V_1 and V_2 (i.e., $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$) such that $\forall (u, w) \in E$, either $u \in V_1$ and $w \in V_2$, or $u \in V_2$ and $w \in V_1$. Examples of bipartite graphs are shown in Figure 2.7.

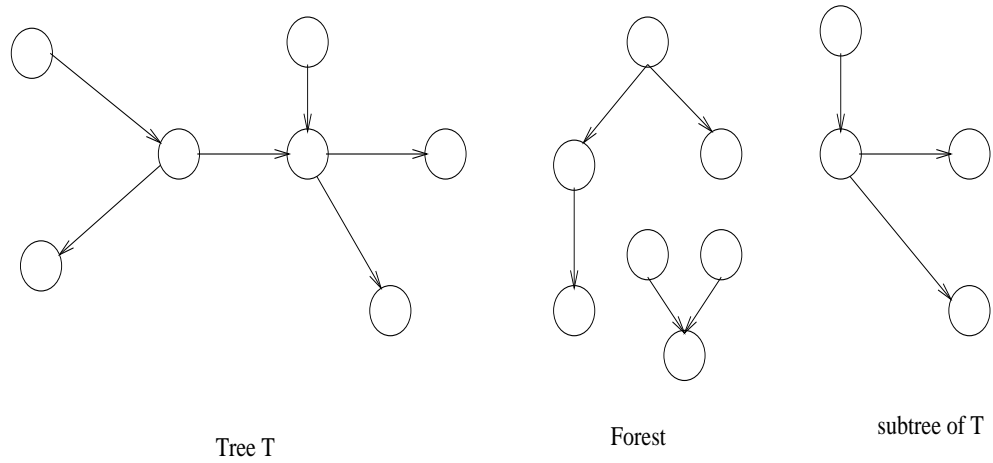


Figure 2.5: Tree, forest and subtree.

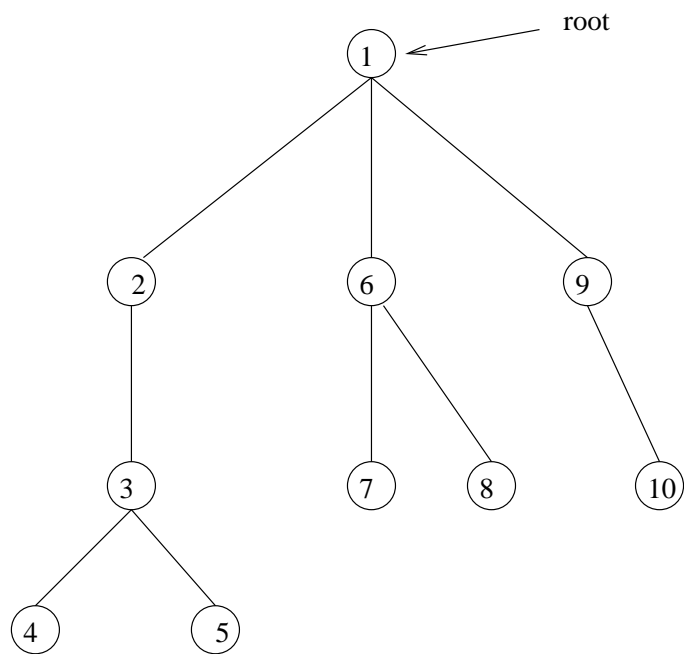


Figure 2.6: A rooted tree.

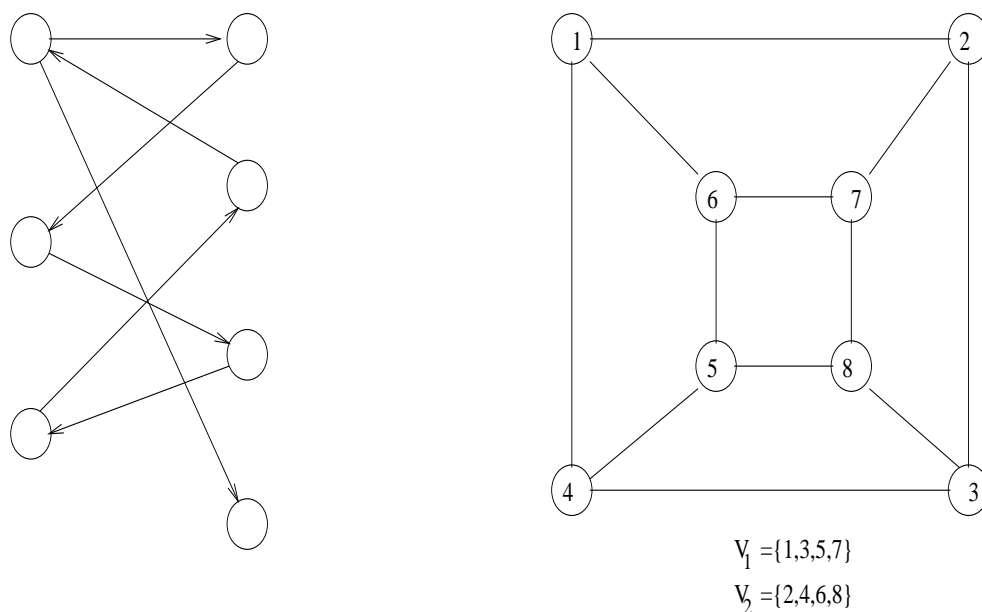


Figure 2.7: Bipartite graphs.

2.2 Graph and Network Representation

Let $G = (V, E)$ be a graph (or network), where $|V| = n$ and $|E| = m$. There are two very common and widely acceptable representations: adjacency matrix and adjacency lists.

Adjacency Matrix

In this representation, the graph (network) is stored in an $n \times n$ matrix M , where every row and column corresponds to a vertex in G . Let $M_{i,j}$ denote the (i, j) -th entry of M . Then, $M_{i,j} = 1$ if $(i, j) \in E$; otherwise, $M_{i,j} = 0$. In the case where G is a network, we construct additional $n \times n$ matrices to store the numerical values associated with its edges. An example is shown in Figure 2.8.

The adjacency matrix has n^2 elements, out of which only m are non-zero. Hence, this representation is space-efficient only if G is very dense. On the other hand, we can determine if two vertices are adjacent, or obtain any numerical value associated with an edge by simply looking at a particular entry of a matrix.

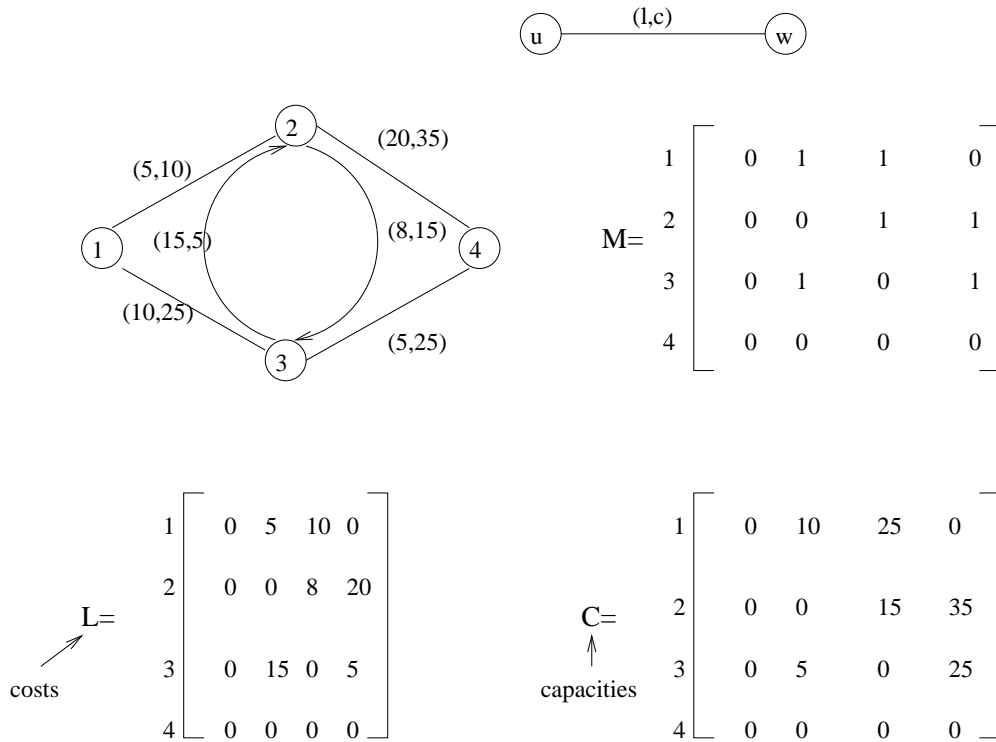


Figure 2.8: Adjacency matrix representation of a graph or network. Each edge (u, w) is associated with a cost $l = l(u, w) = L_{uw}$ and a capacity $c = c(u, w) = C_{uw}$.

Adjacency Lists

The adjacency list $A(u)$ of a vertex u in a graph (network) $G = (V, E)$ is defined as the set $A(u) = \{w : (u, w) \in E\}$. The adjacency list representation of a graph (network) G stores the adjacency list of every vertex in G as a linked list (i.e., a collection of cells each one containing one or more fields). There is an additional array of length n for storing the head of the lists. An example, for the graph of Figure 2.8, is shown in Figure 2.9.

The adjacency list representation contains

$$\sum_{u \in V} |A(u)| = \sum_{u \in V} \text{degree}(u) \leq 2m$$

elements, plus an $n \times 1$ array. Hence, it consists of $\leq n + 2m$ space locations and it is therefore space-efficient (especially when the graph or network is

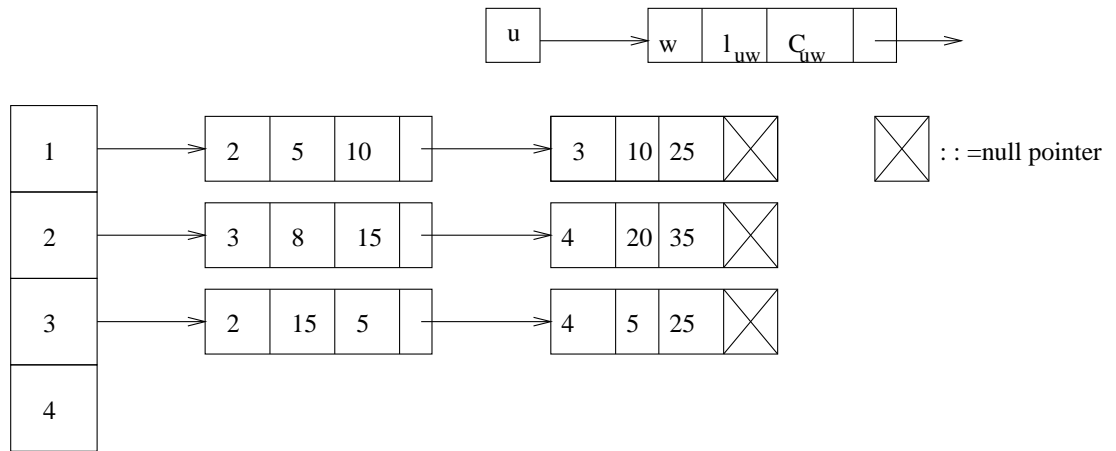


Figure 2.9: Adjacency list representation for the graph in Figure 2.8.

non-dense). On the other hand, to test if two vertices are adjacent, or to obtain a numerical value associated with an edge, one may need to search the whole adjacency list of a vertex.

Remark: with some preprocessing in the adjacency list representation, one can answer the above questions in a number of steps which is logarithmic to the number of elements in an adjacency list.

2.3 Complexity of Algorithms

An algorithm is a step-by-step procedure to solve a problem. We will say that an algorithm A *solves* a problem Π , if the application of A to any instance of Π always produces a solution. The goal is to find the most “efficient” algorithm to solve a problem. But how we measure the efficiency, or generally the performance of an algorithm?

As mentioned above, an algorithm performs a number of steps to solve a problem. The steps that an algorithm performs are generally of three types: (i) read or write (i.e., assignment) steps; (ii) arithmetic steps (additions, multiplications, etc.); and (iii) logical steps (e.g., comparison of two numbers). We will call the above *elementary* steps. The *complexity* (or the number of steps) of an algorithm is the total number of elementary steps that the algorithm performs. There are three approaches to measure the performance of an algorithm.

1. *Empirical analysis*: estimates how the algorithm behaves in practice.

In this analysis, a computer program is written that implements the algorithm and then the program's performance is tested for various problem instances.

2. *Average-case analysis*: estimates the expected number of steps performed by the algorithm. In this analysis, a probability distribution is chosen for the problem instances. Then, using probabilistic methods, the expected number of steps, performed by the algorithm, is determined.
3. *Worst-case analysis*: provides upper bounds on the number of steps performed by the algorithm on *any* problem instance. Here, the maximum possible number of steps is counted that the algorithm performs to solve any problem instance.

The major drawbacks of the empirical analysis are that it depends very much on the computing environment and the programmer's skills and hence cannot provide an objective measure of comparison among several algorithms.

The average-case analysis has some drawbacks also: (a) the analysis depends crucially on the chosen probability distribution; (b) it is often difficult to determine appropriate distributions for many problems met in practice; and (c) it is usually difficult to carry out.

The worst-case analysis avoids many of the above drawbacks (it is independent of the computing environment, it is usually easy to be determined and provides an objective measure of comparison between two algorithms). On the other hand, it permits certain "pathological" (and probably very rare) instances to determine the performance of an algorithm. This is exactly the advantage of the average-case analysis which gives an upper bound of the performance of the algorithm in the majority of problem instances.

However, the advantages of the worst-case analysis have traditionally established it as the most popular method for measuring algorithmic performance in the literature and we will also follow it throughout the course.

The complexity or *running time* of an algorithm is expressed as a function of the problem size. We will assume (as it is widely acceptable) that a data item of value x needs $\log x$ (instead of x) size (i.e., bits) to be stored. For example, a network with n vertices, m edges, largest capacity value C and largest cost value L , needs $n \log n + m \log m + m \log C + m \log L$ bits to be stored (in the adjacency list representation). Hence, the running time of an algorithm for a problem on such a network will be expressed as a function of $n, m, \log C$ and $\log L$.

If we want to be precise with the complexity of an algorithm, we should specify the values of one or more constants (which are independent of the problem size and generally do not affect the algorithm's performance). Since determining these constants is usually a non-trivial task and also depends very much on the computer and other factors as well, the following notation – called *asymptotic notation* – is used which permits one to ignore constants when s/he analyzes the complexity of an algorithm. Let $T(n)$ and $f(n)$ be functions of n .

O notation: $T(n) = O(f(n))$, if \exists a constant c and a number n_0 , such that $T(n) \leq c \cdot f(n)$, $\forall n \geq n_0$.

Ω notation: $T(n) = \Omega(f(n))$, if \exists a constant d and a number n_0 , such that $T(n) \geq d \cdot f(n)$, $\forall n \geq n_0$.

Θ notation: $T(n) = \Theta(f(n))$, if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Examples: $100n + 2n^2 + \frac{n^3}{1000} = O(n^3)$, $500n \log n + 40 \log^2 n = \Omega(n \log n)$, $\frac{\sqrt{n}}{2} = \Theta(\sqrt{n})$

We shall call an algorithm a *polynomial time algorithm* if its running time is bounded by a polynomial function on the problem's size (e.g., a polynomial in $n, m, \log C$ and $\log L$). Examples of polynomial time bounds are: $O(nm \log(\frac{n^2}{m}))$, $O(n^3)$, $O(m + n \log C)$, $O(nm + n^2 \log L)$.

If the running time of an algorithm grows as a function that cannot be polynomially bounded on the problem's size, then the algorithm is called an *exponential-time algorithm*. Examples of exponential time bounds are: $O(2^n)$, $O(nC)$, $O(n^{\log n})$, $O(n!)$.

There are several reasons to prefer polynomial time algorithms rather than exponential ones. The following table gives an indication.

n	10	100	1000	10000
$\log n$	3.32	6.64	9.97	13.29
\sqrt{n}	3.16	10	31.62	100
n^2	10^2	10^4	10^6	10^8
n^3	10^3	10^6	10^9	10^{12}
2^n	10^3	1.27×10^{30}	1.07×10^{304}	0.99×10^{3010}
$n!$	3.6×10^6	9.33×10^{157}	4.02×10^{2567}	2.85×10^{35659}

Even if computer technology improves, polynomial time algorithms are still preferable. To see this, consider a polynomial time algorithm whose

complexity is $O(n^2)$. Assume that the algorithm solves a problem of size n_1 in 1 hour with computing speed S_1 instructions per second. Suppose that we increase the computer speed to $S_2 = 100S_1$. Then, $(\frac{n_2}{n_1})^2 = \frac{S_2}{S_1}$ and consequently $n_2 = n_1 \sqrt{\frac{S_2}{S_1}}$. This implies that a 100-fold increase in computer speed allows us to solve problems which are 10 times larger ($n_2 = 10n_1$). Consider now the above situation, but with an exponential time algorithm of complexity $O(2^n)$. We have that $2^{n_2}/2^{n_1} = S_2/S_1$ and hence $n_2 = n_1 + \log(\frac{S_2}{S_1})$. In this case, a 100-fold increase in computer speed allows us to solve problems that are only 7 units (!) larger ($n_2 \leq n_1 + 7$).

The above discussion shows the importance of developing polynomial-time algorithms and also that improved hardware capabilities can only have a *marginal* impact on the problem-solving ability of exponential time algorithms.

A similar argument justifies why we have to look for more *efficient* algorithms within polynomial-time ones, i.e., a polynomial-time algorithm may not suffice in certain cases and we have to look for another one bounded by a polynomial of lower order. Consider two algorithms A_1 and A_2 for sorting n numbers, and let us assume that A_1 makes $n^2/2$ comparisons (e.g., insertion sort) and that A_2 makes $n \log n$ comparisons (e.g., mergesort) – a comparison is usually the dominating operation in sorting algorithms. Suppose now that we want to sort 10^6 numbers on a computer which performs 10^7 comparisons per second. Then, the time required by A_1 is $(10^6 \times 10^6)/(2 \times 10^7) = 10^5/2$ seconds which is approximately 13 hours. On the other hand, the time required by A_2 is at most $(10^6 \times 20)/10^7 = 2$ seconds. Even a 10-times faster computer would require 1.3 hours to sort the 10^6 numbers using A_1 , while a 10-times slower computer would sort the 10^6 numbers in just 20 seconds using A_2 . Hence, designing efficient algorithms, like designing fast hardware, is a *technology*. The total performance of a computer system depends on the choice of efficient algorithms at least as much as on the choice of fast hardware.

2.4 Fundamental Graph Algorithms

In this section we still study a family of algorithms called *search algorithms*. They are fundamental methods whose general use is to help us to explore a graph (e.g., find all vertices that satisfy a particular property).

The idea is the following. Let $G = (V, E)$ be a graph and let $s \in V$ be an arbitrary *start* vertex. Perform a search of G , starting from s , visiting s , and then perform the following step until there is no edge (u, w) such that

u has been visited and w has not (called *admissible* edge):

SEARCH STEP: select an admissible edge (u, w) (such that u has been visited and w has not) and examine it by visiting w .

Let a vertex called *marked* if it is visited and *unmarked* otherwise. The search algorithm designates all the vertices of G as being in two states: marked or unmarked. By examining admissible edges, the algorithm marks additional vertices. When a new vertex w is marked, by examining an admissible edge (u, w) , we will say that u is the predecessor of w in the search (i.e., $\text{pred}(w)=u$). The algorithm examines (or traverses) the marked vertices in a certain order. Let *order* be an array where its i -th entry $\text{order}(i)$ contains the i -th vertex in the search. A formal description of the search algorithm follows.

Algorithm SEARCH

Input: Graph $G = (V, E)$ and vertex $s \in V$.

Output: Arrays *pred* and *order*.

Method:

```

unmark all vertices of  $V$ ;
mark  $s$ ;  $\text{pred}(s)=0$ ;
 $\text{next}=1$ ;  $\text{order}(s)=\text{next}$ ;
LIST= $\{s\}$ ;
while LIST  $\neq \emptyset$  do
    select vertex  $u$  from LIST;
    if  $u$  is incident to admissible edge  $(u, w)$  then
        mark  $w$ ;  $\text{pred}(w)=u$ ;
         $\text{next}=\text{next}+1$ ;  $\text{order}(w)=\text{next}$ ;
        LIST=LIST  $\cup \{w\}$ ;
    else
        delete  $u$  from LIST;
od

```

In the above algorithm LIST represents a set of marked vertices which the algorithm has to examine because there may be some admissible edges incident on them. Note that the predecessor indices define a tree (of marked vertices) called *search tree*.

The search algorithm runs in $O(n + m) = O(m)$ time, where $|V| = n$ and $|E| = m$ (we assume, without loss of generality, that G is connected and

thus $m \geq n - 1$). To see this, consider an iteration of the while loop. Every such iteration either finds an admissible edge or not. In the former case, the algorithm marks a new vertex and adds it to LIST. In the latter case it deletes a marked vertex from LIST. Since the algorithm marks a vertex at most once, it executes the while loop at most $2n$ times (n steps for marking the vertices and n steps for deleting them from LIST). Now, how much does the algorithm spend to identify the admissible edges? For each vertex u , the algorithm scans every edge in the adjacency list $A(u)$ of u at most once *overall executions* of the while loop. Hence, the algorithm, in total, examines $\sum_{u \in V} |A(u)| \leq 2m$ edges. Therefore, the algorithm terminates in $O(m + n) = O(m)$ time.

The algorithm does not specify the manner of examining vertices or for adding the vertices to LIST. Different rules give rise to different search techniques. The two most popular (and most fundamental) search strategies are depth-first search and breadth-first search.

Depth-first search (DFS). Select the admissible edge (u, w) such that u was visited most recently. To achieve this search strategy it suffices to implement the set LIST as a *stack*: We always select the vertex which has been added last.

Breadth-first search (BFS). Select the admissible edge (u, w) such that u was visited least recently. To achieve this search strategy it suffices to implement the set LIST as a *queue*: we select vertices from the front (the least recent) while we add them to the rear.

We shall now give an application of the search algorithm. Let $G = (V, E)$ be a directed graph. Assume that we want to possess a labeling of the vertices of G in such a way that $\forall u, z \in V, \text{label}(u) \neq \text{label}(z)$ and $\forall (u, w) \in E, \text{label}(u) < \text{label}(w)$. Such a label is called a *topological ordering* (of the vertices) of G .

Note that not all directed graphs have a topological ordering. Figure 2.10 shows an example.

Actually only DAGs have topological ordering. It can be proved that a directed graph (or digraph) is acyclic iff it possesses a topological ordering of its vertices.

In the following, we shall give an algorithm which either detects the presence of a directed cycle in a digraph G , or produces a topological ordering of G . It is based on the following property.

Proposition 2.4.1 *If every vertex of a directed graph G has indegree at least 1, then G contains a directed cycle.*

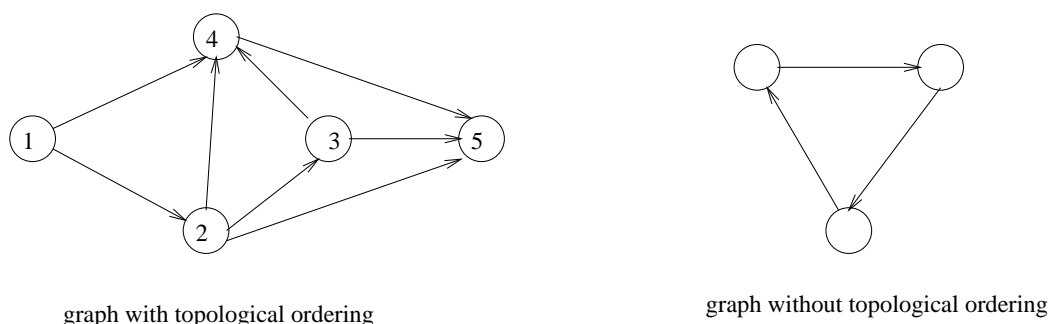


Figure 2.10: Topological ordering.

Proof. Reverse the direction of every edge in G yielding digraph G^* . Clearly, every vertex has outdegree at least 1 in G^* . Now, start from any vertex s and perform the search algorithm on G^* . Since every vertex has outdegree ≥ 1 and G^* has a finite number of vertices, we must at some point return to some vertex w that we have already visited. Let z be the vertex of G^* visited just before we return to w . Then, the path from w to z determined during the search algorithm plus the edge (z, w) define a cycle C^* in G^* . If we now reverse the direction of the edges in G^* , we have again a cycle C in G (defined on the same set of vertices as C^* but with reverse direction). ■

We are now ready for our algorithm: select any vertex s of the input digraph G of indegree 0. If such a vertex does not exist, then by Proposition 2.4.1 G has a directed cycle and stop. Otherwise, give s label 1. Delete s and all of its adjacent edges (s, u) in G . Take any vertex s' of indegree 0 in the remaining graph, give it label 2, delete it from the graph along with all of its adjacent edges. Repeat this process until there is no vertex of indegree 0 in the graph. If at this point, the remaining graph has vertices and edges, then by Proposition 2.4.1 it contains a directed cycle. Otherwise, G is acyclic and we have assigned labels to all of its vertices. Now notice that whenever we assign a label to some vertex during an iteration, the vertex has only outgoing edges whose heads are vertices that have not been labeled yet. These vertices will be labeled in subsequent iteration and therefore will be assigned a higher label. Consequently, the above algorithm produces a correct topological ordering of a DAG. The running time of the algorithm is clearly $O(m)$.

Chapter 3

Shortest Paths

3.1 Introduction

The shortest path problem is perhaps the simplest among the problems that we will study and appears to be fundamental in many combinatorial and other network optimization problems.

Let $G = (V, E)$ be a directed network whose edges have an associated real-valued weight function $wt : E \rightarrow \mathbb{R}$. We call $wt(u, v)$ the *weight* of edge $(u, v) \in E$. For $s, t \in V$, the weight of a directed path from s to t (henceforth, s - t path) is the sum of the weights of the edges in the path. A *shortest s - t path* is an s - t path whose weight is minimum among all s - t paths. The *distance* from s to t , denoted as $\delta(s, t)$, is the weight of a shortest s - t path in G . The *shortest path problem* is to find a shortest path between given pairs of vertices in G . There are four versions of the problem:

1. *Single-pair*: given two vertices s and t , find a shortest s - t path.
2. *Single-source*: given a vertex s , find a shortest s - v path for every vertex $v \in V$.
3. *Single-sink*: given a vertex t , find a shortest v - t path for every vertex $v \in V$.
4. *All-pairs*: for every pair of vertices $s, t \in V$, find a shortest s - t path.

Shortest paths do not always exist. The following lemma characterizes their existence and form.

Lemma 3.1.1 *Let $s, t \in V$ be two vertices of a directed network $G = (V, E)$ such that there is a directed walk from s to t in G . There is a shortest s - t walk in G iff no walk from s to t contains a directed cycle of total negative weight (called negative cycle). If there is any shortest s - t walk in G , then there is one which is an s - t path.*

Proof. A (shortest) s - t walk is defined similarly to a (shortest) s - t path. If there is an s - t walk in G that contains a negative cycle, then we can traverse this cycle an infinite number of times, each time reducing the weight of the s - t walk, and thus producing an s - t walk of arbitrarily small weight. Now, if no s - t walk contains a negative cycle, then we can convert any s - t walk to an s - t path by eliminating cycles. Elimination of cycles (with non-negative weight) does not increase the weight of the resulting s - t path. ■

The above lemma shows that the presence of a negative cycle makes the shortest path problem substantially harder to solve. (Actually, in this case the problem becomes NP-complete.) For this reason, we will assume henceforth that the input network G does not contain a negative cycle. Without loss of generality, we will also assume that there is a directed path from a vertex s to a vertex t in $G = (V, E)$ and also that G is connected. These assumptions can be easily enforced by adding additional edges in G : for any edge $(u, w) \in E$, if $(w, u) \notin E$ then add it with weight ∞ . If G is disconnected, with connected components (i.e., connected subgraphs of G) G_1, G_2, \dots, G_k , then $\forall 1 \leq i < k$ add edges (x, y) and (y, x) with weight ∞ where x is any vertex of G_i and y is any vertex of G_{i+1} .

We have defined the shortest path problem with respect to a directed network. In the case of an undirected network $G_u = (V_u, E_u)$ the problem can be similarly defined. Moreover, if all edge weights are non-negative, then the shortest path problem in G_u can be easily transformed into one on a directed network $G = (V, E)$: for every undirected edge $e = (v, w) \in E_u$, add two directed edges (v, w) and (w, v) in E with the same weight as e . Unfortunately, this transformation does not work if there are edges in G_u with negative weight. (Such an edge would immediately produce a negative cycle with the above transformation.) A much more complicated transformation is needed to handle this case. The above discussion justifies why we consider only directed networks.

Having characterized the existence and form of shortest paths in directed networks, we will see now two very simple but very powerful properties.

Proposition 3.1.1 *A subpath of a shortest path is a shortest path itself.*

Proof. Let $P = u_1, u_2, \dots, u_k$ be a shortest u_1 - u_k path in a directed network G . Let also S be a subpath of P from u_i to u_j , $1 \leq i < j \leq k$. Assume that S is not a shortest u_i - u_j path and let S^* be the shortest u_i - u_j path, i.e., with weight less than the weight of S . But in this case, the subpath u_1 - u_i of P , the path S^* , and the subpath u_j - u_k of P determine a u_1 - u_k path in G of total weight less than the weight of P , a contradiction. ■

Proposition 3.1.2 *Let $G = (V, E)$ be a directed network with weight function $wt : E \rightarrow \mathbb{R}$. A directed path $P = u_1, u_2, \dots, u_k$ is a shortest u_1 - u_k path iff $\delta(u_1, u_{i+1}) = \delta(u_1, u_i) + wt(u_i, u_{i+1})$, $\forall (u_i, u_{i+1}) \in P$, $1 \leq i < k$.*

Proof. (\Rightarrow) If P is a shortest u_1 - u_k path, then by Proposition 3.1.1 $\delta(u_1, u_{i+1}) = \delta(u_1, u_i) + wt(u_i, u_{i+1})$, $\forall (u_i, u_{i+1}) \in P$, $1 \leq i < k$, since u_1 - u_i and edge (u_i, u_{i+1}) are subpaths of a shortest u_1 - u_{i+1} path.

(\Leftarrow) To prove that P is indeed a shortest u_1 - u_k path, it suffices to prove that its weight is $\delta(u_1, u_k)$. We know that $\forall (u_i, u_{i+1}) \in P$, $1 \leq i < k$, $\delta(u_1, u_{i+1}) = \delta(u_1, u_i) + wt(u_i, u_{i+1})$ or $wt(u_i, u_{i+1}) = \delta(u_1, u_{i+1}) - \delta(u_1, u_i)$. The weight of P is

$$\begin{aligned} \sum_{i=1}^{k-1} wt(u_i, u_{i+1}) &= \sum_{i=1}^{k-1} (\delta(u_1, u_{i+1}) - \delta(u_1, u_i)) \\ &= (\delta(u_1, u_2) - \delta(u_1, u_1)) + (\delta(u_1, u_3) - \delta(u_1, u_2)) + \dots + \\ &\quad (\delta(u_1, u_{k-1}) - \delta(u_1, u_{k-2})) + (\delta(u_1, u_k) - \delta(u_1, u_{k-1})) \\ &= \delta(u_1, u_k) - \delta(u_1, u_1) \\ &= \delta(u_1, u_k) \end{aligned}$$

because $\delta(u_1, u_1) = 0$. ■

We are now ready to study the shortest path problem. Consider the four versions of the problem mentioned before. Version 1 is treated in the literature as a subcase of version 2 (or 4). Note also that version 3 is the directional dual of version 2: if we have an algorithm A that solves the single-source shortest path problem for a given vertex s of a directed network G , then by changing the direction of every edge in G (but keeping the same weights) we create a network G^* which is called the directional dual of G . Applying algorithm A with given vertex s on G^* , produces a solution to the single-sink shortest path problem on G with given (sink) vertex s . Therefore, we will focus only on the single-source shortest path (SSSP) and the all-pairs shortest path (APSP) problems.

3.2 The SSSP Problem

There are two main algorithmic approaches to solve the SSSP problem: *label-setting* and *label-correcting* algorithms. Both of them are iterative: they assign, at each step, a tentative distance label $d(u)$ to every vertex u that represents an upper bound on the real distance $\delta(s, u)$, where s is the given vertex for the SSSP problem. The approaches vary in many aspects, including:

- (a) The updating of distance labels: label setting algorithms designate one label as permanent (optimal) at each step. Label-correcting algorithms consider all distance labels as tentative until the final step, when they all become permanent.
- (b) The problems they solve: label-setting algorithms are applicable only to directed acyclic networks with arbitrary edge weights and to directed networks with non-negative edge weights. The label-correcting algorithms apply to every directed network including those with negative edge weights.
- (c) The complexity: in general, label-setting algorithms are more efficient (with respect to worst-case complexity) than the label-correcting ones and also their analysis is simpler.

Recall that in the SSSP problem we want to find all shortest s - u paths in a directed network $G = (V, E)$, where s is a specific vertex (called the *source*) and $u \in V$ is any vertex. Since we have to compute $n - 1$ shortest paths ($n = |V|$), one could argue that we need $O(n^2)$ space to store them (since in the worst-case a shortest path may contain $n - 1$ edges). Fortunately, this is not the case. There is a compact way to represent all $n - 1$ shortest s - u paths as a tree rooted at s . This tree is called *shortest path tree* and it is a spanning tree T of G where every s - u path in T (s is the root of T) is a shortest s - u path in G .

The following two lemmata characterize shortest path trees.

Lemma 3.2.1 *A shortest path tree rooted at a source vertex s of a directed network G always exist.*

Proof. Recall that without loss of generality we can assume that there is a directed path from s to every other vertex. Since there is a finite number of vertices in $G = (V, E)$, there is also a finite number of s - u paths, $\forall u \in V$. Consequently, there is a shortest s - u path in G , $\forall u \in V$.

Let T^* be the union (of vertices and edges) of all shortest s - u paths, $\forall u \in V$. (If for some u there is more than one shortest s - u path, then pick one arbitrarily.) By construction T^* is a spanning subgraph of G and is a DAG. Take any vertex u in T^* , $u \neq s$, and delete all but one of its incoming edges. It can be easily verified that the resulting graph T is a tree, from the fact that every vertex u ($u \neq s$) has indegree exactly one. Any s - u path in T is a shortest s - u path in G ($u \neq s$). To see this, consider the following two cases:

(a) u is a leaf of T . This means that either s - u was an s - u path in T^* (and hence a shortest s - u path in G), or u is a leaf because all edges (u, w) incident to u in T^* were deleted. In the latter case, the s - u path in T was a subpath of a path in T^* (i.e., a shortest path in G) and hence by Proposition 3.1.1 it is a shortest path in G .

(b) u is not a leaf in T . Then the s - u path in T is a subpath of an s - z path in T , where z is a leaf of T . By (a) above, any s - z path in T is a shortest path in G and hence, by Proposition 3.1.1, the s - u path in T is also a shortest path in G . ■

Lemma 3.2.2 T is a shortest path tree rooted at a given source vertex s of a directed network G iff \forall edge (u, w) of T , $\delta(s, w) = \delta(s, u) + wt(u, w)$.

Proof. (\Rightarrow) Since T is a shortest path tree rooted at s , every s - x path in T , $x \neq s$, is a shortest path. By Proposition 3.1.2, every edge (u, w) in this path satisfies $\delta(s, w) = \delta(s, u) + wt(u, w)$.

(\Leftarrow) Let T be a spanning tree of G rooted at s . By assumption, every edge (u, w) of T and hence of any s - x path in T , $\forall x$ in T ($x \neq s$), satisfies $\delta(s, w) = \delta(s, u) + wt(u, w)$. By Proposition 3.1.2, the s - x path in T is a shortest path in G . ■

EXERCISE 3.1. Let $G = (V, E)$ be a directed network and let s be a source vertex. Assume that all distances $\delta(s, u)$, $\forall u \in V$, are given but not the shortest paths. Prove that a BFS of G starting at s and selecting the edges $(u, w) \in E$ that satisfy $\delta(s, w) = \delta(s, u) + wt(u, w)$, produces a shortest path tree rooted at s .

3.3 Optimality Conditions

Optimality conditions allow us to certify when the distance labels (found by an SSSP algorithm) are optimal, i.e., equal to the actual distances of shortest

paths from the source vertex to all other vertices. They provide us with a certificate and hence with a termination condition for our algorithm. In some cases, optimality conditions help also in the development of algorithmic solutions.

Let $d(v)$ denote the distance label of vertex v in our input directed network $G = (V, E)$. As mentioned above, during the execution of a label-correcting algorithm $d(v)$ is an upper bound on $\delta(s, v)$, i.e., the weight of a shortest s - v path in G , where s is the source vertex. Upon termination of the algorithm, $d(v)$ is equal to $\delta(s, v)$.

Lemma 3.3.1 *Let $d(v)$ denote the weight of an s - v path in a directed network $G = (V, E)$. The numbers $d(v)$, $\forall v \in V$, represent shortest path distances iff $d(v) \leq d(u) + wt(u, v)$, $\forall (u, v) \in E$ (with equality holding for all edges belonging to the shortest path tree).*

Proof. (\Rightarrow) If for some edge $(u, v) \in E$, the optimality condition does not hold, we must have $d(v) > d(u) + wt(u, v)$. But in such a case we can improve the weight of a shortest s - v path by passing through vertex u , which contradicts the fact that $d(v)$ is the weight of a shortest s - v path. Obviously, for those edges (u, v) which belong to the shortest path tree, $d(v) = d(u) + wt(u, v)$.

(\Leftarrow) Let $s = v_1, v_2, \dots, v_k = v$ be a path P in G from the source vertex s to some vertex v . By assumption we have:

$$\begin{aligned} d(v_k) &\leq d(v_{k-1}) + wt(v_{k-1}, v_k) \\ d(v_{k-1}) &\leq d(v_{k-2}) + wt(v_{k-2}, v_{k-1}) \\ &\vdots \\ d(v_2) &\leq d(v_1) + wt(v_1, v_2) \end{aligned}$$

Adding the above inequalities (and taking into account that $d(v_1) = d(s) = 0$ and $d(v) = d(v_k)$), we get:

$$d(v) \leq wt(v_{k-1}, v_k) + wt(v_{k-2}, v_{k-1}) + \dots + wt(v_1, v_2) = wt(P)$$

Therefore, $d(v)$ is a lower bound on the weight of any s - v path in G , implying that $d(v) \leq \delta(s, v)$. By definition, $\delta(s, v)$ is also a lower bound on any s - v path in G , and consequently we must have $d(v) = \delta(s, v)$. ■

Lemma 3.3.2 *If G contains a negative cycle, then no set of distance labels satisfies the optimality conditions.*

Proof. Let $v_1, v_2, \dots, v_k, v_1$ be any negative cycle C in G . Assume that the optimality conditions are satisfied. This implies that:

$$\begin{aligned} d(v_2) &\leq d(v_1) + wt(v_1, v_2) \\ d(v_3) &\leq d(v_2) + wt(v_2, v_3) \\ &\vdots \\ d(v_k) &\leq d(v_{k-1}) + wt(v_{k-1}, v_k) \\ d(v_1) &\leq d(v_k) + wt(v_k, v_1) \end{aligned}$$

Summing up the above inequalities, we get $0 \leq \sum_{i=1}^{k-1} wt(v_i, v_{i+1}) + wt(v_k, v_1)$, implying that C is not a negative cycle, a contradiction. ■

3.4 A Generic Label-Correcting Algorithm

The generic label-correcting algorithm is a general procedure which successively updates distance labels (in a greedy manner) until they satisfy the optimality conditions. For every vertex $v \in V$ of the input network $G = (V, E)$, a distance label $d(v)$ is maintained. (Recall that we can assume w.l.o.g. that an s - v path always exist in G .) The label $d(v)$ either equals ∞ (denoting that the algorithm has not yet find an s - v path in G), or equals the weight of some s - v path. A predecessor index, $pred(v)$, is also maintained which stores the predecessor vertex of v in the current s - v path of weight $d(v)$. The algorithm is as follows:

Algorithm Generic label-correcting.

Input: Network $G = (V, E)$, $wt : E \rightarrow \mathbb{R}$, and source vertex $s \in V$.

Output: Shortest paths and distances from s to all other vertices in G .

Method:

```
(* Initialization step *)
for every  $v \in V$  do  $d(v) = \infty$ ;
 $d(s) = 0$ ;  $pred(s) = 0$ ;
(* Relaxation step *)
while  $\exists$  edge  $(u, v) : d(v) > d(u) + wt(u, v)$  do
     $d(v) = d(u) + wt(u, v)$ ;
     $pred(v) = u$ ;
od
```

We call the scanning of an edge (u, v) to check whether the while-loop condition is satisfied and if yes to update $d(v)$ and $pred(v)$, the *relaxation*

of (u, v) . The edges $(pred(v), v)$ define a graph G_π called *predecessor graph*. The generic label-correcting algorithm has a surprisingly long (though simple) proof of correctness and termination, and of establishing that upon termination G_π is a shortest path tree rooted at s . We shall need five properties before our main theorem.

Proposition 3.4.1 *After relaxing an edge (u, v) , $d(v) \leq d(u) + wt(u, v)$; that is, $d(v)$ does not increase.*

Proof. Obvious. ■

Proposition 3.4.2 *The generic label-correcting algorithm maintains the invariant that $d(v) \geq \delta(s, v)$, $\forall v \in V$. Moreover, once $d(v)$ achieves its lower bound $\delta(s, v)$ it never changes.*

Proof. If $v = s$ the claim is obviously true. Let us assume that the invariant is not maintained after a sequence of iterations. Let v be the first vertex for which $d(v) < \delta(s, v)$, after relaxing edge (u, v) . Then, we have $d(u) + wt(u, v) = d(v) < \delta(s, v) = \delta(s, u) + wt(u, v)$ which implies that $d(u) < \delta(s, u)$. But the relaxation of edge (u, v) does not change $d(u)$; therefore, we should have $d(u) < \delta(s, u)$ before relaxing (u, v) which contradicts the choice of v .

Now, if $d(v) = \delta(s, v)$, then $d(v)$ cannot decrease because $d(v) \geq \delta(s, v)$ (as we just showed), and also $d(v)$, by Proposition 3.4.1, cannot increase. ■

Proposition 3.4.3 *The generic label-correcting algorithm maintains the invariant that for every edge (u, v) in G_π , $d(u) + wt(u, v) \leq d(v)$.*

Proof. We use induction on the number of iterations. Clearly, the claim is true after the initialization step of the algorithm. The algorithm adds (u, v) to G_π after relaxing it, which implies that $d(v) = d(u) + wt(u, v)$. If in subsequent iterations $d(u)$ decreases, then the invariant is still maintained. If $d(v)$ decreases, it may happen that $d(v) < d(u) + wt(u, v)$ for the edge (u, v) in G . But observe that in this case, (u, v) does not belong to G_π . The reason is that $d(v)$ decreases only after relaxing some edge (z, v) , which happens because the s - v path in G with last edge (z, v) has weight smaller than the s - v path passing through u . As a consequence, $pred(v) = z \neq u$, after the relaxation step, and hence (u, v) must have been deleted from G_π . ■

Proposition 3.4.4 *The generic label-correcting algorithm maintains the invariant that either G_π is a tree rooted at s spanning all vertices v with finite labels, or there is a (directed) cycle in G_π .*

Proof. A vertex $v \neq s$ has $\text{pred}(v) \neq 0$ iff $d(v) \neq \infty$ (i.e., $d(v)$ is finite). Moreover, if $d(v) \neq \infty$ and $\text{pred}(v) \neq 0$, $d(\text{pred}(v)) \neq \infty$. Therefore, if we start at v and follow predecessor indices, we either reach s and are unable to continue ($\text{pred}(s) = 0$), or we repeat a vertex. ■

Proposition 3.4.5 *If during the execution of the generic label-correcting algorithm there is a (directed) cycle in G_π , then G has a negative cycle.*

Proof. Assume that after relaxing edge (u, v) a cycle $v = v_1, v_2, \dots, v_{k-1}, v_k = u, v_1$ is created in G_π . Consider the situation just before the relaxation step. By Proposition 3.4.3, all edges (v_i, v_{i+1}) , $1 \leq i < k$, satisfy:

$$\begin{aligned} d(v_1) + wt(v_1, v_2) &\leq d(v_2) \\ d(v_2) + wt(v_2, v_3) &\leq d(v_3) \\ &\vdots \\ d(v_{k-1}) + wt(v_{k-1}, v_k) &\leq d(v_k) \end{aligned}$$

For (u, v) (i.e., (v_k, v_1)), we have that $d(v_k) + wt(v_k, v_1) < d(v_1)$. Adding this last inequality to the above ones, we get:

$$\sum_{i=1}^{k-1} wt(v_i, v_{i+1}) + wt(v_k, v_1) < 0$$

which implies that $v_1, v_2, \dots, v_{k-1}, v_k, v_1$ is a negative cycle in G . ■

We are now ready for our main theorem.

Theorem 3.4.1 *If G does not contain a negative cycle, then algorithm generic label-correcting terminates correctly. Moreover, upon termination the predecessor graph G_π is a shortest path tree rooted at the source vertex s .*

Proof. By Propositions 3.4.1 and 3.4.2, the iterations performed by the algorithm cause the distance labels to decrease monotonically towards the actual (shortest path) distances. By Lemma 3.3.2 such a set of distance labels exist, since G does not have a negative cycle. Moreover, as soon as distance labels reach their lower bounds they never change. If $d(v) = \delta(s, v)$, $\forall v \in V$, then there is no edge (u, v) in G which needs to be relaxed (i.e.,

satisfies $d(v) > d(u) + wt(u, v)$); otherwise, there is an s - v path in G with weight smaller than $\delta(s, v)$, contradicting the definition of $\delta(s, v)$. Therefore, by Lemma 3.3.1, the algorithm terminates correctly.

We now prove that upon termination, G_π is a shortest path tree rooted at s . By Propositions 3.4.4 and 3.4.5, G_π is a spanning tree of G rooted at s . To complete the proof it suffices to show that any s - v path in G_π is a shortest s - v path in G .

Let P be any $s = v_1, v_2, \dots, v_k = v$ path in G_π . Upon termination, we have that $d(v_i) = \delta(s, v_i)$, $1 \leq i < k$. Also by Proposition 3.4.3, $d(v_{i+1}) \geq d(v_i) + wt(v_i, v_{i+1})$, $1 \leq i < k$. Hence, $wt(v_i, v_{i+1}) \leq \delta(s, v_{i+1}) - \delta(s, v_i)$, $1 \leq i < k$. But then,

$$\begin{aligned} wt(P) &= \sum_{i=1}^{k-1} wt(v_i, v_{i+1}) \leq \sum_{i=1}^{k-1} (\delta(s, v_{i+1}) - \delta(s, v_i)) \\ &= \delta(s, v_k) - \delta(s, v_1) = \delta(s, v) \end{aligned}$$

Thus $wt(P) \leq \delta(s, v)$. Since $\delta(s, v)$ is a lower bound on the weight of any s - v path in G , we must have $wt(P) = \delta(s, v)$ and consequently G_π is a shortest path tree. ■

EXERCISE 3.2. Show that the generic label-correcting algorithm performs a total number of $O(2^n)$ iterations.

3.5 An Improved Implementation of the Label-Correcting Algorithm

Bellman (1958) proposed the following (quite efficient) implementation of the generic label-correcting algorithm.

Arrange the edges of the input network $G = (V, E)$ in some arbitrary order. Then, make passes through E relaxing the edges one by one (in the same order in every pass). That is, in each pass examine for each edge $(u, v) \in E$ whether the condition $d(v) > d(u) + wt(u, v)$ is satisfied and if yes, update $d(v) = d(u) + wt(u, v)$ and set $pred(v) = u$. The algorithm stops when no distance label changes during an entire pass.

Theorem 3.5.1 *The Bellman label-correcting algorithm solves the SSSP problem in $O(nm)$ time.*

Proof. It suffices to show that the algorithm performs at most $n - 1$ passes. Then, the running time follows immediately because each pass requires $O(m)$

time (since we examine every edge). To prove that the algorithm performs at most $n - 1$ passes, we show that at the end of the k -th pass, $1 \leq k \leq n - 1$, the algorithm sets $d(v) = \delta(s, v)$, for all shortest s - v paths with at most k edges. If we show this, we are done since no shortest path in G contains more than $n - 1$ edges.

We use induction on k . The basis ($k = 1$) holds trivially, since $d(s) = 0 = \delta(s, s)$. Assume that $d(u) = \delta(s, u)$ for all shortest s - u paths with at most $k - 1$ edges. Consider a vertex v , such that any s - v shortest path has exactly k edges. Let this path be $s = v_1, v_2, \dots, v_k = v$. By Proposition 3.1.1, the subpath $s = v_1, v_2, \dots, v_{k-1}$ is a shortest s - v_{k-1} path, and hence (by the induction hypothesis) $d(v_{k-1}) = \delta(s, v_{k-1})$. Since there is no shortest s - v path with less than k edges, edge (v_{k-1}, v_k) has to be relaxed during the k -th pass. Consequently, $d(v_k) = d(v_{k-1}) + wt(v_{k-1}, v_k) = \delta(s, v_{k-1}) + wt(v_{k-1}, v_k) = \delta(s, v_k)$. ■

3.6 Detecting Negative Cycles

All the algorithms described before can be easily modified to detect the presence of a negative cycle (and actually find it), if such a cycle exists in the input network $G = (V, E)$. We will give now three methods for the problem: the first two concern the generic label-correcting algorithm whereas the last one concerns Bellman's implementation.

The first method is as follows. Let M be the largest absolute value of an edge weight in G . Clearly, $-nM$ is a lower bound on any distance label, if there is no negative cycle. Therefore, if at some iteration of the generic label-correcting algorithm the distance label of some vertex is smaller than $-nM$, then G must have a negative cycle. The cycle can be obtained by following the predecessor indices.

The second method is based on Propositions 3.4.4 and 3.4.5. If at some iteration of the generic label-correcting algorithm a cycle in G_π is discovered, then G contains a negative cycle. It is easy to see that G_π can be tested for a cycle in $O(n)$ time.

The third method is based on the fact (see Theorem 3.5.1) that Bellman's algorithm performs at most $n - 1$ passes, because a shortest path in G contains at most $n - 1$ edges (i.e., at most n vertices). If the algorithm performs more than $n - 1$ passes, then some vertex in some shortest path has to be repeated. Consequently, there is a cycle in this shortest path which by Lemma 3.1.1 must be negative. The negative cycle can be found by tracing the predecessor indices. The above discussion and Theorem 3.5.1

lead to the following.

Theorem 3.6.1 *The Bellman label-correcting algorithm either solves the SSSP problem, or finds a negative cycle, in $O(nm)$ time.*

3.7 The SSSP Problem: Special Cases

In this section, we will study two important special cases of the single-source shortest path problem that frequently arise in practice. These are the case of directed acyclic networks and the case of non-negative edge weights. As we shall see, we can achieve from optimal to almost optimal results in these cases.

3.7.1 Acyclic Networks

Let $G = (V, E)$ be a directed acyclic network with real-valued edge weights. Recall from Chapter 2 that we can topologically order (or number) the vertices of G (i.e., assign them new names) such that $u < w$, $\forall (u, w) \in E$. Having this ordering, the solution of the SSSP problem is easy. The idea is as follows. Assume that we have determined the distances $\delta(s, u)$ from s to every vertex $u \in \{1, 2, \dots, k-1\}$. Consider vertex k . The topological ordering tells us that all incoming edges to vertex k have their tail in $\{1, 2, \dots, k-1\}$. By Proposition 3.1.1, a shortest s - k path is composed of a shortest s - u path, $u \in \{1, 2, \dots, k-1\}$, plus the edge (u, k) . Therefore, $\delta(s, k) = \min_{(u,k) \in E} \{\delta(s, u) + wt(u, k)\}$. The above discussion gives rise to the following algorithm.

Algorithm SSSP-DAG

Input: Directed acyclic network $G = (V, E)$, topologically ordered.

Output: Distances $d(\cdot)$, from vertex s to all other vertices in G .

Method:

```

for every  $u \in V$  do  $d(u) = \infty$ ;
 $d(s) = 0$ ;  $n = |V|$ ;
for  $u = s$  to  $n$  do
    for every  $(u, w) \in E$  do
        if  $d(w) > d(u) + wt(u, w)$  then
             $d(w) = d(u) + wt(u, w)$ 
    od
od

```

Theorem 3.7.1 *Algorithm SSSP-DAG solves the SSSP problem on a directed acyclic network $G = (V, E)$ in $O(m)$ time, where $m = |E|$.*

Proof. We first show the correctness. It suffices to show that whenever the algorithm examines a vertex, its distance label is optimal. We use induction. Assume that the algorithm has examined vertices $1, 2, \dots, k-1$, and examines vertex k . Let the shortest s - k path in G be s, \dots, z, k . By Proposition 3.1.1 the s - z subpath is also a shortest path. The fact that G is topologically ordered and $(z, k) \in E$ implies that $z \in \{1, 2, \dots, k-1\}$. By the induction hypothesis the distance label $d(z)$ is optimal, i.e., $d(z) = \delta(s, z)$. Consequently, when the algorithm examined vertex z it must have been scanned edge (z, k) and set the distance label $d(k)$ equal to the weight of the path s, \dots, z, k , i.e., $d(k) = d(z) + wt(z, k) = \delta(s, z) + wt(z, k) = \delta(s, k)$. Hence, when the algorithm examines vertex k its distance label $d(k)$ is optimal.

Concerning the running time of the algorithm, observe that the time for examining vertex u is proportional to the outdegree of u . Therefore, the running time of the algorithm is bounded by $O(\sum_{u \in V} \text{outdegree}(u)) = O(m)$. ■

3.7.2 Non-Negative Edge Weights: Dijkstra's Algorithm

Dijkstra's algorithm (1959) solves the SSSP problem on a directed network $G = (V, E)$ with non-negative edge weights. The algorithm is a label-setting one. It maintains a distance label $d(u)$, $\forall u \in V$, such that $\delta(s, u) \leq d(u)$. At any step, the algorithm divides V into two sets: those which are permanently labeled and those which are temporarily labeled. The algorithm selects a vertex u with the minimum temporary label, makes it permanent, and updates the distance labels of its adjacent vertices. The correctness of the algorithm lies on the fact that we can always make permanent the distance label of the minimum temporary-labeled vertex. A less informal description of the algorithm follows.

Algorithm DIJKSTRA

Input: Directed network $G = (V, E)$, $n = |V|$, $m = |E|$, with non-negative edge weights and source $s \in V$.

Output: Array $d(\cdot)$ of length n , where $d(u) = \delta(s, u)$, $\forall u \in V$, and array $pred(\cdot)$ of length n , where $pred(u)$ is the parent of vertex u in the shortest path tree rooted at s .

Method:

```

 $S = \emptyset; \bar{S} = V;$ 
for every  $u \in V$  do  $d(u) = \infty;$ 
 $d(s) = 0; pred(s) = 0;$ 
while  $|S| < n$  do
    (* vertex selection *)
    let  $u \in \bar{S}$  be the vertex for which  $d(u) = \min_{v \in \bar{S}} \{d(v)\};$ 
     $S = S \cup \{u\}; \bar{S} = \bar{S} - \{u\};$ 
    (* distance update *)
    for each  $(u, w) \in E$  do
        if  $d(w) > d(u) + wt(u, w)$  then
             $d(w) = d(u) + wt(u, w); pred(w) = u;$ 
        od
    od

```

Theorem 3.7.2 *Dijkstra's algorithm solves the SSSP problem in directed networks with non-negative edge weights in $O(n^2)$ time.*

Proof. We first show the correctness. It suffices to prove by induction on the size of S that (i) $\forall u \in S, d(u) = \delta(s, u)$ and (ii) $\forall u \in \bar{S}, d(u)$ is the weight of a shortest s - u path among those s - u paths whose internal vertices belong to S .

The basis of the induction ($|S| = 1$) is trivial. To prove (i), we show that the weight of any s - u path that contains some vertices in \bar{S} is at least $d(u)$. Let u be the selected vertex at some step $|S| > 1$. Note that by the induction hypothesis $d(u)$ is the weight of a shortest s - u path among all s - u paths that do not contain any vertex of \bar{S} as an internal vertex. Let P be any s - u path that contains some vertices in \bar{S} , and let w be the first such vertex of \bar{S} (see Figure 3.1). Denote by P_1 the s - w subpath of P and by P_2 its w - u subpath. By the induction hypothesis, $wt(P_1) \geq d(w)$. Since u has been selected by the algorithm, $d(u) \leq d(w)$. Hence, $wt(P) = wt(P_1) + wt(P_2) \geq d(w) + wt(P_2) \geq d(u)$, since $wt(P_2) \geq 0$ as all edge weights are non-negative.

To prove (ii), note that after some vertex z has been permanently labeled (i.e., $z \in S$), the algorithm examines every adjacent vertex u of z and

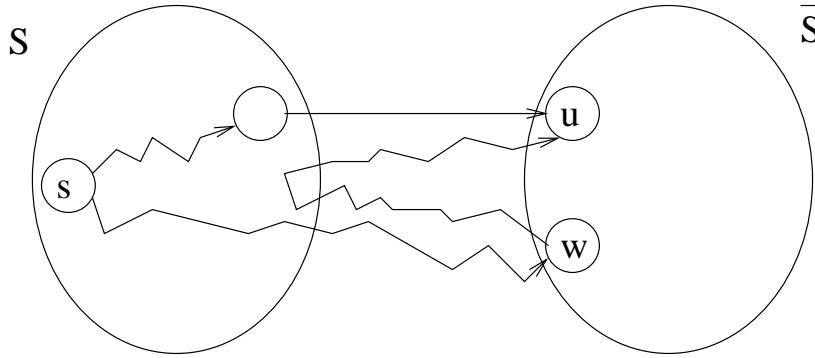


Figure 3.1: Proof of Dijkstra's algorithm; u is the vertex with smallest temporary distance label.

performs a distance update (if necessary) $d(u) = d(z) + wt(z, u)$. The s - u path is the concatenation of the s - z path and the edge (z, u) . Every vertex of the s - z path belongs to S and hence by the assumption hypothesis every distance label is optimal implying that $d(x) = d(y) + wt(y, x)$, for every edge (y, x) belonging to the s - z path. Since also the edge (z, u) satisfies the distance equality (i.e., $d(u) = d(z) + wt(z, u)$), it follows by Proposition 3.1.2 that the above s - u path is a shortest path in G among all shortest paths whose internal vertices belong to S .

Let us now analyze the running time of Dijkstra's algorithm. Clearly, it is equal to the total number of vertex selection steps plus the total number of distance update steps. Consider first the vertex selection steps. Once a vertex is selected it is removed from \bar{S} . Moreover, to find the smallest labeled vertex in \bar{S} requires $O(|\bar{S}|)$ time. Hence, the total number of vertex selection steps requires $O(n + (n - 1) + \dots + 1) = O(n^2)$ time. For the distance update steps, observe that for every selected vertex u the distance update is performed at most $outdegree(u)$ times. Therefore, the algorithm performs a total of $\sum_{u \in V} outdegree(u) = O(m)$ distance update steps. Since m is always less than or equal to n^2 , the time bound follows. ■

3.7.3 Heap Implementations of Dijkstra's Algorithm

The bottleneck in Dijkstra's algorithm is the vertex selection step. We shall see now how we can improve on the algorithm's performance using heap data structures to implement this step. A *heap* (or priority queue) is a data structure for efficiently storing and manipulating a collection H of objects where every object $i \in H$ has an associated real number called its *key* and

denoted by $key(i)$. The following operations are provided by a heap:

create-heap(H): creates an empty heap.

find-min(i, H): finds and returns an object i of minimum key.

insert(i, H): inserts a new object i with a predefined key.

decrease-key(val, i, H): sets $key(i) = val$, where $val < key(i)$.

delete-min(i, H): deletes an object i of minimum key.

In the implementation of Dijkstra's algorithm using a heap H , H stores the vertices with finite temporary distance labels and the key of such a vertex (object) is its distance label.

Algorithm HEAP-DIJKSTRA

Input: As in algorithm DIJKSTRA.

Output: As in algorithm DIJKSTRA.

Method: *create-heap*(H);
for every $u \in V$ **do** $d(u) = \infty$;
 $d(s) = 0$; $pred(s) = 0$; *insert*(s, H);
while $H \neq \emptyset$ **do**
 (* vertex selection *)
 find-min(u, H); *delete-min*(u, H);
 (* distance update *)
 for each $(u, w) \in E$ **do**
 $val = d(u) + wt(u, w)$;
 if $d(w) > val$ **then**
 if $d(w) = \infty$ **then**
 $d(w) = val$; $pred(w) = u$; *insert*(w, H);
 else
 $d(w) = val$; $pred(w) = u$; *decrease-key*(val, w, H);
 od
 od

We now analyze the running time of algorithm HEAP-DIJKSTRA using a d -heap and a Fibonacci heap.

d-heap implementation: for a given parameter $d \geq 2$, a d -heap requires $O(\log_d n)$ time for *insert* and *decrease key*, $O(d \log_d n)$ time for *delete-min*, and $O(1)$ time for the rest of the operations. Therefore, the running time

of the algorithm is $O(m \log_d n + nd \log_d n)$ using this data structure. The running time is optimized when $d = \max\{2, \lceil m/n \rceil\}$, and thus becomes $O(m \log_d n)$. Note that if $m = \Omega(n^{1+\varepsilon})$, for some constant $0 < \varepsilon < 1$, the running time of the algorithm is optimal, because $O(m \log_d n) = O(m \frac{\log n}{\log d}) = O(m \frac{\log n}{\log n^\varepsilon}) = O(\frac{m}{\varepsilon}) = O(m)$.

Fibonacci heap implementation: this data structure performs every heap operation in $O(1)$ amortized¹ time except for delete-min which requires $O(\log n)$ time. Using this data structure, it is easy to verify that algorithm HEAP-DIJKSTRA runs in $O(m + n \log n)$ time which is the fastest known strongly polynomial-time algorithm for solving the SSSP problem (with non-negative arbitrary weights).

3.7.4 Integer Edge Weights: Dial's Algorithm

Dial (1969) proposed a simple way to improve Dijkstra's algorithm in practice when the *edge weights are integers*. Similarly to the heap implementations, the idea is to maintain the temporary distance labels in a sorted fashion. The algorithm is based on the following property.

Proposition 3.7.1 *The distance labels which are permanently labeled in Dijkstra's algorithm are nondecreasing.*

Proof. The algorithm always labels permanently a vertex u with a smallest temporary label $d(u)$. Furthermore, when the algorithm scans a vertex w adjacent to u during a distance update step, it never decreases w 's temporary distance label $d(w)$ below $d(u)$, because all edge weights are non-negative. ■

Let Δ be the largest edge weight in the input network $G = (V, E)$. Clearly, $n\Delta$ is an upper bound on the distance label of any finitely labeled vertex.

Dial's algorithm stores vertices with finite temporary labels in a collection of $n\Delta + 1$ sets, called *buckets*, numbered $0, 1, 2, \dots, n\Delta$. Bucket k stores the vertices with temporary distance label equal to k . Let $B(k)$ denote the contents of bucket k . The two main steps (vertex selection and distance update) are implemented as follows.

Vertex selection: (i) Scan buckets $0, 1, 2, \dots$, until the first nonempty bucket, say k , is found. Each vertex $u \in B(k)$ has the minimum distance

¹The amortized complexity of an operation denotes the average worst-case complexity of performing this operation. When some operation takes $O(t)$ amortized time, this means that a sequence of k such operations requires $O(kt)$ time in the worst case.

label. (ii) Delete all $u \in B(k)$, one by one, designate them as permanently labeled, and proceed with the distance update step.

Distance update: (i) Let w , with distance label d_1 , be a vertex adjacent to vertex u which has been permanently labeled during the vertex selection step. Perform a distance update to w resulting into a new distance label d_2 for w . (ii) Move w from $B(d_1)$ to $B(d_2)$.

In the *next vertex selection step*, we scan buckets $k + 1, k + 2, \dots$, to find the next nonempty bucket. This is correct, because Proposition 3.7.1 implies that buckets $0, 1, 2, \dots, k$ will be empty in all subsequent iterations.

Let us analyze the running time of Dial's algorithm. Each $B(k)$, $0 \leq k \leq n\Delta$, is stored as a doubly linked list (which implies that checking whether $B(k)$ is empty, or adding/deleting an element to $B(k)$, requires $O(1)$ time). With this data structure, each distance update step takes $O(1)$ time and thus the algorithm requires $O(m)$ time for all distance update steps. The bottleneck operation is vertex selection which requires scanning of the $n\Delta + 1$ buckets. Proposition 3.7.1 ensures that each bucket has to be scanned only once. We have proved the following result.

Theorem 3.7.3 *Dial's algorithm runs in $O(m + n\Delta)$ time.*

Dial's algorithm is a pseudo-polynomial one (consider e.g., $\Delta = 2^n$). However, in many practical applications Δ is quite small (compared to n) and the algorithm performs very well.

3.7.5 Integer Edge Weights: Radix Heap Implementation

The radix heap implementation (proposed in 1990 by Ahuja, Mehlhorn, Orlin and Tarjan) is a hybrid of the original implementation of Dijkstra's algorithm and Dial's implementation. We assume that the edge weights of the input network are integers and Δ denotes the largest edge weight. The radix heap implementation gives one of the fastest polynomial-time algorithms for the SSSP problem on networks with non-negative integral weights.

Dijkstra's original implementation puts all the tentatively labeled vertices in one larger set (or bucket) and selects, among them, the vertex with minimum label. On the other hand, Dial's implementation puts vertices into a large collection of buckets, where different labeled vertices go to different buckets. The radix heap implementation improves upon these two extremes by following an intermediate approach: it stores many labels, but not all, in one bucket. This idea reduces the number of buckets used. However,

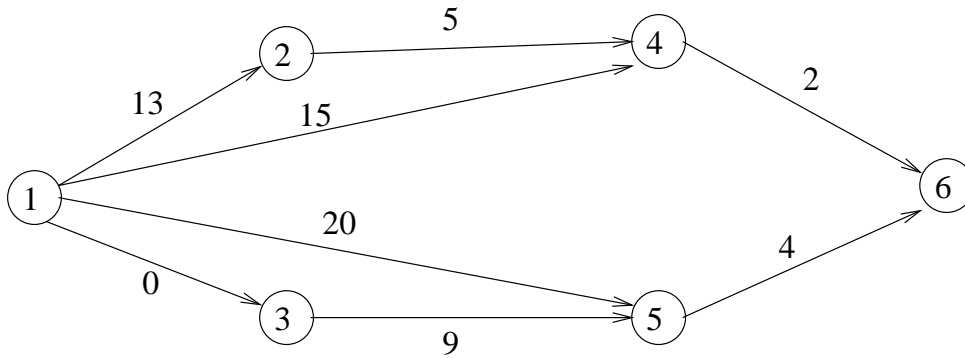


Figure 3.2: Network for radix heap implementation: $s = 1$, $\Delta = 20$, $K = \lceil \log(120) \rceil = 7$.

storing many tentative labels in one bucket, may require a lot of time to find, in the lowest-numbered bucket, the vertex with minimum distance label. Therefore, an implementation scheme should be employed such that the lowest-numbered bucket contains only a few (say 1) elements. Only in this manner, the advantage of reducing the number of buckets would be useful.

Let us call the *range* of bucket i , and denote it as $R(i)$, the different tentative labels that can be stored in that bucket. $R(i)$ is a (possibly empty) closed interval of integers. The quantity $|R(i)|$ is called the *width* of bucket i . The radix heap implementation uses variable widths and changes the ranges dynamically. Vertices with tentative distance labels are reallocated in such a way that the bucket containing the minimum distance label has width 1.

The radix heap consists of $1 + \lceil \log(n\Delta) \rceil$ buckets, numbered $0, 1, 2, \dots, \lceil \log(n\Delta) \rceil = K$. The ranges of them are initially defined as follows: $R(0) = [0]$, $R(i) = [2^{i-1}, 2^i - 1]$, $1 \leq i \leq K$. Note that the widths of the buckets are $1, 1, 2, 4, 8, 16, \dots, 2^{K-1}$. A vertex u with tentative label $d(u)$ is stored in bucket i , if $d(u) \in R(i)$. (Permanently labeled vertices are not stored.)

The algorithm works as follows: it changes the ranges of the buckets dynamically, and each time it changes the ranges, it redistributes the vertices in the buckets. Note that the width of the buckets never increase beyond their initial width.

Let us now illustrate the algorithm with an example. In the following, let $B(i)$ denote the contents of bucket i , $0 \leq i \leq k$. Consider the network of Figure 3.2. The radix heap after the examination of vertex 1 looks as follows:

u	1	2	3	4	5	6
$d(u)$	0	13	0	15	20	∞

<i>Bucket i</i>	0	1	2	3	4	5	6	7
$R(i)$	[0]	[1]	[2,3]	[4,7]	[8,15]	[16,31]	[32,63]	[64,127]
$B(i)$	{3}	\emptyset	\emptyset	\emptyset	{2, 4}	{5}	\emptyset	\emptyset

The algorithm designates vertex 3 as permanently labeled, it deletes it from the heap, and examines edge (3,5). Then it changes the distance label $d(5)$ from 20 to 9. Since the new $d(5)$ is not contained in $R(5)$, vertex 5 has to move to a lower-numbered bucket. This is done, by sequentially scanning the buckets from right to left (starting at bucket 5) to locate the first bucket j , such that $d(5) \in R(j)$. In our example, $j = 4$ and vertex 5 moves to this bucket now.

In the next iteration, we want to find the vertex with minimum label. Scanning the buckets from left to right (starting with bucket 0), bucket $l = 4$ is identified as the first nonempty bucket. Since $R(l)$ contains more than one integer, the first vertex in $B(l)$ is not necessarily the one with minimum label. Proposition 3.7.1 implies that no tentative distance label will be less than the minimum distance label in $B(l)$ and consequently the algorithm will never use ranges $R(0), R(1), \dots, R(l-1)$ to store tentative distance labels. Hence, we can redistribute $R(l)$ into the buckets $0, 1, 2, \dots, l-1$, and reinsert the vertices in $B(l)$ into these lowered-numbered buckets. In our example, the minimum distance label in $B(4)$ is $9(= d(5))$ and the range $[9,15]$ is redistributed as follows: $R(0) = [9]$, $R(1) = [10]$, $R(2) = [11, 12]$, $R(3) = [13, 15]$, and $R(4) = \emptyset$. Since now $R(4) = \emptyset$, the vertices in $B(4)$ have to be reassigned in buckets 0,1,2,3. This is done by successively selecting a vertex $w \in B(4)$, scanning sequentially buckets 3,2,1,0, and insert w into the appropriate bucket. The radix heap looks now as follows:

u	1	2	3	4	5	6
$d(u)$	0	13	0	15	9	∞

<i>Bucket i</i>	0	1	2	3	4	5	6	7
$R(i)$	[9]	[10]	[11,12]	[13,15]	[0]	[16,31]	[32,63]	[64,127]
$B(i)$	{5}	\emptyset	\emptyset	{2, 4}	\emptyset	\emptyset	\emptyset	\emptyset

We are now ready to see the general description of the algorithm and its complexity.

Vertex selection: Scan buckets $0, 1, 2, \dots$, (from left to right) to find the first nonempty bucket, say l . This requires $O(K)$ time per iteration and

hence $O(nK)$ time overall. If $l = 0$, or $l = 1$, there is only one vertex in $B(l)$ and this vertex has the minimum distance label. If $l \geq 2$, redistribute the “useful” range of bucket l into the buckets $0, 1, \dots, l - 1$, and reinsert the elements of $B(l)$ into these buckets. Let $R(l) = [p, r]$ and let d^* be the smallest distance label of a vertex in $B(l)$. Then, the useful range of bucket l is $[d^*, r]$. The useful range is redistributed as follows: the first integer of $[d^*, r]$ is assigned to $R(0)$, the second integer to $R(1)$, the next two integers to $R(2)$, the next four integers to $R(3)$, etc. This can be done because bucket l has width $\leq 2^l - 2^{l-1} = 2^{l-1}$, and buckets $0, 1, \dots, l - 1$ have a total width of $1 + 1 + 2 + 4 + \dots + 2^{l-2} = 1 + (2^{l-1} - 1) = 2^{l-1}$. The redistribution of ranges requires $O(K)$ time per iteration and thus $O(nK)$ time in total. The reinsertion of vertices to buckets $0, 1, \dots, l - 1$ is done as follows. Let $u \in B(l)$. If $d(u) \notin R(l)$, then scan buckets $l - 1, l - 2, \dots, 0$ sequentially until an appropriate bucket j is found such that $d(u) \in R(j)$. Every time a vertex u moves, it moves to a lower-indexed bucket. Since we have $K + 1$ buckets, the total number of vertex movements is $O(nK)$. Hence, the vertex selection step requires $O(nK)$ time in total.

Distance update: Let w be a vertex adjacent to vertex u which has been permanently labeled during the vertex selection step. Let also $w \in B(l)$. Perform a distance update to w . If $d(w) \notin R(l)$, then scan buckets $l - 1, l - 2, \dots, 0$ sequentially until an appropriate bucket j is found such that $d(w) \in R(j)$. Insert w to $B(j)$. As before, the total number of vertex movements (because of a distance update) is $O(nK)$. There are a total of m distance updates and therefore the distance update step requires $O(m + nK)$ time in total.

The above discussion is summarized in the following.

Theorem 3.7.4 *The radix heap implementation of Dijkstra’s algorithm runs in $O(m + n \log(n\Delta))$ time.*

3.8 All-Pairs Shortest Paths

In this section we will give algorithms for solving the all-pairs shortest path (APSP) problem. We will study two approaches: the first one treats the problem as the solution of n SSSP problems, while the second approach solves the APSP problem directly by using a label-correcting method. Recall that the solution of an SSSP problem requires $\Omega(n)$ space, since we want to find shortest paths and distances from a source vertex to all other vertices of the network. Analogously, a solution to the APSP problem requires $\Omega(n^2)$

space, because we have to report shortest paths and distances between $\Theta(n^2)$ pair of vertices.

3.8.1 Solving APSP through SSSP

It is clear that the APSP problem, for a directed network $G = (V, E)$, can be solved by applying n times an SSSP algorithm (where each time a different vertex from the set V is considered as the source vertex). This approach is space-optimal because each SSSP solution (i.e., a shortest path tree) takes $O(n)$ space.

If G has non-negative weights, then the solution of the APSP problem takes $O(nT_S(n, m))$ time, where $T_S(n, m)$ is the time required to solve an SSSP problem on G . In most of the cases, this approach solves the APSP problem quite efficiently. For example, if we use the Fibonacci heap implementation of Dijkstra's algorithm, then $T_S(n, m) = O(m + n \log n)$ and consequently the solution to the APSP problem takes $O(nm + n^2 \log n)$ time, which performs very well in the case of sparse networks (i.e., networks with less than $O(n \log n)$ edges). In general, this approach never takes more than $O(n^3)$ time.

If G has negative weights, then we have to use Bellman's algorithm. We select a vertex s (as a source) and run the algorithm. The algorithm either identifies a negative cycle in G , or solves the SSSP problem. In the former case, the APSP problem does not have a solution and we stop. In the latter case, we have a solution to the APSP problem (by repeating Bellman's algorithm from every vertex in G) which takes $O(n^2m)$ time. Note that this solution performs very poor: in the best-case it requires $O(n^3)$ time; however, if G is dense (i.e., it contains $\Theta(n^2)$ edges), then it requires $O(n^4)$ time.

We will now see how we can solve the APSP problem in networks with negative edge weights (but no negative cycles) in only $O(nT_S(n, m))$ time (thus matching the bound of the case with non-negative weights). To achieve this, we have to transform our initial network G into another one G' whose edge weights are non-negative. To do the transformation, we need the concept of reduced edge weights.

Let $d(v)$, $\forall v \in V$, be the distance labels returned by the execution of an algorithm for solving the SSSP problem (assuming w.l.o.g. that G does not have a negative cycle; otherwise, shortest paths are not defined.) For each edge $(u, v) \in E$, define its *reduced edge weight* $\rho(u, v)$ with respect to these distance labels as $\rho(u, v) = wt(u, v) + d(u) - d(v)$. Let P be a path v_1, v_2, \dots, v_k in G . The reduced weight $\rho(P)$ of P is defined as $\rho(P) =$

$\sum_{i=1}^{k-1} \rho(v_i, v_{i+1})$. (The reduced weight of a cycle C is defined similarly.) We will need the next lemma about reduced edge weights.

Lemma 3.8.1 *Reduced edge weights satisfy the following properties:*

- (i) *If $d(v) = \delta(s, v)$, $\forall v \in V$, then $\rho(u, v) \geq 0$ for every edge $(u, v) \in E$.*
- (ii) *For any x - y path P in G , $\rho(P) = wt(P) + d(x) - d(y)$.*
- (iii) *For any cycle C in G , $\rho(C) = wt(C)$.*

Proof. Part (i) is immediate from Lemma 3.3.1. For part (ii), let P be the path $x = v_1, v_2, \dots, v_k = y$. We have that:

$$\begin{aligned} \rho(P) &= \sum_{i=1}^{k-1} \rho(v_i, v_{i+1}) \\ &= \sum_{i=1}^{k-1} (wt(v_i, v_{i+1}) + d(v_i) - d(v_{i+1})) \\ &= \sum_{i=1}^{k-1} wt(v_i, v_{i+1}) + \sum_{i=1}^{k-1} (d(v_i) - d(v_{i+1})) \\ &= wt(P) + d(v_1) - d(v_k) = wt(P) + d(x) - d(y). \end{aligned}$$

For part (iii), let $v_1, v_2, \dots, v_k, v_1$ be the cycle C . Then,

$$\begin{aligned} \rho(C) &= \rho(v_1, v_2, \dots, v_k) + \rho(v_k, v_1) \\ &= wt(v_1, v_2, \dots, v_k) + d(v_1) - d(v_k) \text{ (by (ii))} \\ &\quad + wt(v_k, v_1) + d(v_k) - d(v_1) \\ &= wt(C) \end{aligned}$$

■

We are now ready to describe the promised method. It consists of the following steps: (1) Select (arbitrarily) a source vertex s and run Bellman's algorithm. If the algorithm finds a negative cycle, then stop. Otherwise, let $d(v)$, $\forall v \in V$, be the distance labels returned by the algorithm. (2) Replace the weight of every edge $(u, v) \in E$ by its reduced weight $\rho(u, v)$ and call the new (transformed) network G' . By Lemma 3.8.1(i), all edge weights in G' are non-negative. (3) Run n times in G' an algorithm for solving the SSSP problem in networks with non-negative edge weights, resulting in a solution to the APSP problem in G' . Let $\delta'(x, y)$ be the distance from x to y in G' (i.e., the weight of a shortest x - y path in G'). (4) Obtain the actual distance $\delta(x, y)$ (i.e., the weight of a shortest x - y path in G), by adding $d(y) - d(x)$ to $\delta'(x, y)$ (according to Lemma 3.8.1(ii)).

The above method requires $O(nm)$ time to solve the first SSSP problem (or to identify a negative cycle), and if there is no negative cycle in G , an additional time of $O(nT_S(n, m))$ to compute the required shortest paths and distances. Hence, the total time taken by this method is $O(nm + nT_S(n, m)) = O(nT_S(n, m))$ (because $T_S(n, m) \geq m$). We summarize the above discussion as follows.

Theorem 3.8.1 *The APSP problem in a directed network with arbitrary edge weights can be solved in time $O(nT_S(n, m))$, where $T_S(n, m)$ is the time to solve one SSSP problem on a directed network with non-negative edge weights.*

3.8.2 APSP through Label-Correcting Algorithms

We start with the optimality conditions for the APSP problem which, as in the case of the SSSP problem, will immediately suggest a generic label-correcting algorithm.

Lemma 3.8.2 *Let $d(u, v)$ represent the weight of a (directed) u - v walk in a directed network $G = (V, E)$. Then, distances $d(u, v)$, $\forall u, v \in V$, represent shortest path distances iff $d(u, v) \leq d(u, w) + d(w, v)$, for all $u, v, w \in V$.*

Proof. (\Rightarrow) Assume that there is a triple of vertices u, v and w for which $d(u, v) > d(u, w) + d(w, v)$. This implies that there is a u - v walk (or path) in G (passing through w) with weight smaller than $d(u, v)$. However, by assumption, $d(u, v) = \delta(u, v)$. Consequently, there is a u - v walk (or path) in G with weight smaller than $\delta(u, v)$ contradicting the definition of $\delta(u, v)$.

(\Leftarrow) Let P be a path v_1, v_2, \dots, v_k of weight $d(v_1, v_k)$. By assumption, we have:

$$\begin{aligned} d(v_1, v_k) &\leq d(v_1, v_2) + d(v_2, v_k) \leq wt(v_1, v_2) + d(v_2, v_k) \\ d(v_2, v_k) &\leq d(v_2, v_3) + d(v_3, v_k) \leq wt(v_2, v_3) + d(v_3, v_k) \\ &\vdots \\ d(v_{k-1}, v_k) &\leq wt(v_{k-1}, v_k) \end{aligned}$$

The above inequalities imply that $d(v_1, v_k) \leq wt(v_1, v_2) + wt(v_2, v_3) + \dots + wt(v_{k-1}, v_k) = wt(P)$. Hence, $d(v_1, v_k)$ is a lower bound on any v_1 - v_k walk (or path) in G and therefore $d(v_1, v_k) \leq \delta(v_1, v_k)$. But from the definition of $d(v_1, v_k)$ we also have $d(v_1, v_k) \geq \delta(v_1, v_k)$. Consequently, $d(v_1, v_k) = \delta(v_1, v_k)$. \blacksquare

The generic APSP label-correcting algorithm is based on the same principle as the generic label-correcting algorithm for the SSSP problem: it starts with some initial (tentative) distance labels $d(u, v)$. At each iteration, a tentative distance label is decreased. The algorithm proceeds until all distance labels converge to their corresponding shortest path distances (if the network does not have a negative cycle), i.e., until they satisfy the optimality conditions. The algorithm is as follows.

Algorithm Generic APSP label-correcting.

Input: Network $G = (V, E)$, with $wt : E \rightarrow \mathbb{R}$.

Output: Shortest paths and distances between every pair of vertices in G .

Method:

```
(* Initialization step *)
for each pair  $u, v \in V$  do
     $d(u, v) = \infty$ ;
     $pred(u, v) = 0$ ;
od
for every  $v \in V$  do  $d(v, v) = 0$ ;
for each edge  $(u, v) \in E$  do
     $d(u, v) = wt(u, v)$ ;
     $pred(u, v) = u$ ;
od
(* Relaxation step *)
while  $\exists u, v, w \in V : d(u, v) > d(u, w) + d(w, v)$  do
     $d(u, v) = d(u, w) + d(w, v)$ ;
     $pred(u, v) = pred(w, v)$ ;
od
```

The algorithm maintains a predecessor index, $pred(u, v)$, for each pair of vertices $u, v \in V$. These indices allow us to discover the actual shortest u - v path in G . The path is discovered in a reversed fashion: $pred(u, v)$ determines the last (internal) vertex x in the shortest u - v path. Then, $pred(u, x)$ determines the vertex z prior to x in the shortest u - v path. The process is repeated until u is reached.

EXERCISE 3.3. Show that if G does not contain a negative cycle, then the generic APSP label-correcting algorithm terminates correctly.

Similarly to the SSSP case, the generic APSP label-correcting algorithm may perform an exponential number of iterations. (To see this, consider for example the case where all edge weights are integers and the largest absolute

edge weight Δ has a value of 2^n .) We will now see an efficient implementation of the above APSP algorithm, known as Floyd-Warshall algorithm, which solves the APSP problem in $O(n^3)$ time. It is worth noting that the time bound is quite satisfactory in the sense that given an $n \times n$ matrix of distances $d(u, v)$, we need $\Omega(n^3)$ time to test the optimality conditions.

The Floyd-Warshall algorithm (1962) constructs such a distance matrix in $O(n^3)$ time by applying the relaxation steps in a clever way. Assume that $V = \{1, 2, \dots, n\}$. Let $d^{k+1}(u, v)$ represent the weight of a shortest u - v path such that all its internal vertices belong to $\{1, 2, \dots, k\}$. Clearly, $d^{n+1}(u, v)$ represents $\delta(u, v)$. The algorithm starts by computing $d^1(u, v)$, $\forall u, v \in V$, which is actually the initialization step (see below). Then, it computes $d^2(u, v)$ using $d^1(u, v)$. In the $(k+1)$ -th step, $1 \leq k \leq n$, the algorithm computes $d^{k+1}(u, v)$ using the distance labels $d^k(u, v)$. The computation (and the correctness of the algorithm) is based on the following property.

Proposition 3.8.1 $d^{k+1}(u, v) = \min\{d^k(u, v), d^k(u, k) + d^k(k, v)\}$.

Proof. A shortest u - v path that uses, as its internal vertices, only vertices in $\{1, 2, \dots, k\}$ either does not pass through vertex k implying $d^{k+1}(u, v) = d^k(u, v)$, or passes through k , in which case $d^{k+1}(u, v) = d^k(u, k) + d^k(k, v)$. ■

A less informal description of the algorithm follows.

Algorithm Floyd-Warshall APSP.

Input: Network $G = (V, E)$, with $wt : E \rightarrow \mathbb{R}$.

Output: Shortest paths and distances between every pair of vertices in G .

Method:

```
(* Initialization step *)
for each pair  $u, v \in V$  do
     $d(u, v) = \infty$ ;
     $pred(u, v) = 0$ ;
od
for every  $v \in V$  do  $d(v, v) = 0$ ;
for each edge  $(u, v) \in E$  do
     $d(u, v) = wt(u, v)$ ;
     $pred(u, v) = u$ ;
od
(* Relaxation step *)
for  $k = 1$  to  $n$  do
    for each pair  $u, v \in V$  do
        if  $d(u, v) > d(u, k) + d(k, v)$  then
             $d(u, v) = d(u, k) + d(k, v)$ ;
             $pred(u, v) = pred(k, v)$ ;
        od
    od
od
```

From the description of the algorithm, Proposition 3.8.1, and the above discussion, the following result is straightforward.

Theorem 3.8.2 *The Floyd-Warshall algorithm computes APSP in a directed network, with arbitrary edge weights but no negative cycles, in $O(n^3)$ time.*

We will now discuss how the above algorithms can be modified to detect a negative cycle (if such a cycle exists) in the input network G . Let M be the largest absolute value of an edge weight in G . Then, incorporate the following tests in the generic APSP label-correcting algorithm whenever a distance label is updated during a relaxation step.

- (I) If $u = v$, check whether $d(u, u) < 0$.
- (II) If $u \neq v$, check whether $d(u, v) < -nM$.

Proposition 3.8.2 *If either of the above tests is true during the execution of the generic APSP label-correcting algorithm, then G contains a negative cycle.*

Proof. (I) Consider the first time that $d(u, u) < 0$, due to a distance update in a relaxation step. At this point, $d(u, u) = d(u, w) + d(w, u)$ for some $w \neq u$. This implies that there is a (directed) u - w walk and also a (directed) w - u walk such that the sum of their weights is negative. The union of these two walks is a closed (u - u) walk consisting of a set of (directed) cycles. Since $d(u, u) < 0$, at least one of these cycles must be negative.

(II) Consider the first time that $d(u, v) < -nM$. At this point G contains a (directed) u - v walk of weight $-nM$. This walk consists of a u - v path and a collection of cycles. Since no path in G has weight less than $-(n-1)M$, at least one of these cycles must be negative. ■

EXERCISE 3.4. In the Floyd-Warshall algorithm: (a) Show that test (I) suffices for the detection of a negative cycle. (b) How a negative cycle can be found, if such a cycle exists?

It is clear that checking whether these tests are satisfied does not add to the worst-case complexity of the algorithms. The above discussion and Theorem 3.8.2 imply:

Theorem 3.8.3 *The Floyd-Warshall algorithm computes APSP in a directed network with arbitrary edge weights, or finds a negative cycle (if it exists), in $O(n^3)$ time.*

Chapter 4

Maximum Flows: Basic Algorithms

4.1 Introduction

The maximum flow problem is the second fundamental problem in network optimization and is complementary to the shortest paths problem.

Let $G = (V, E)$ be a directed network with a nonnegative real capacity $c(v, w)$ for every edge $(v, w) \in E$. Let s and t be two distinguished vertices, called the *source* and the *sink* respectively. An s - t *flow* (or simply a *flow*) on G is a function $f : E \rightarrow \mathbb{R}^+ \cup \{0\}$ satisfying the following properties:

1. Capacity constraint: $\forall (v, w) \in E, 0 \leq f(v, w) \leq c(v, w)$.
2. Flow conservation: $\forall v \in V - \{s, t\}$,

$$\sum_{(v,w) \in E} f(v, w) - \sum_{(w,v) \in E} f(w, v) = 0.$$

The *value* $|f|$ of a flow f is the net flow out of the source, that is:

$$|f| = \sum_{(s,v) \in E} f(s, v) - \sum_{(v,s) \in E} f(v, s) = \sum_{(v,t) \in E} f(v, t) - \sum_{(t,v) \in E} f(t, v)$$

The *maximum flow problem* is the problem of finding a flow of maximum value from s to t in G , called *max-flow*.

We will make the following three assumptions concerning our input network G :

A1: G does not contain a directed s - t path where every edge of this path has an infinite capacity. If this assumption does not hold, then we can always

send an infinite amount of flow from s to t and consequently the value of a max-flow is unbounded. Note that such a path can be detected in time linear to the size of G , by the search algorithm given in Chapter 2.

A2: If $(v, w) \in E$, then $(w, v) \in E$ as well. This is not a restriction, since we allow edges with capacity zero. In other words, if an edge is not in the network, we can assume its existence with zero capacity.

A3: G contains no parallel edges. This assumption also possesses no restriction. Assume that there are two parallel edges e_1 and e_2 with tail v and head w . Let also f_i and c_i be the flow and the capacity resp. of edge e_i , $i = 1, 2$. Then, we can remove e_1 and e_2 and substitute them with a single edge (v, w) having flow $f_1 + f_2$ and capacity $c_1 + c_2$. It can be easily verified that this transformation does not change the value of a max-flow.

4.2 Preliminaries and the Max-Flow Min-Cut Theorem

Given a flow f on G , the *residual capacity* $r(v, w)$ of an edge $(v, w) \in E$ is defined as $r(v, w) = c(v, w) - f(v, w) + f(w, v)$. The residual capacity of (v, w) represents the additional amount of flow that can be sent from v to w using the edges (v, w) and (w, v) . The term $c(v, w) - f(v, w)$ represents the unused capacity of edge (v, w) , while the term $f(w, v)$ represents the current flow on edge (w, v) (which we try to cancel in order to increase the flow from v to w).

The network $G(f) = (V, E(f))$, where $E(f) = \{e \in V \times V : r(e) > 0\}$, is called the *residual network* of G with respect to the flow f . Figure 4.1 illustrates the definition of a residual network.

A directed s - t path in $G(f)$ is called an *augmenting path* for f . (Observe that an augmenting path is an undirected s - t path in G whose edges have positive capacity.)

Recall from Chapter 2 that a *cut* is a partition of V into two subsets S and $\bar{S} = V - S$ denoted as $[S, \bar{S}]$. A cut defines (uniquely) a set of edges with one endpoint in S and the other in \bar{S} . Let $(S, \bar{S}) = \{(v, w) \in E : v \in S \wedge w \in \bar{S}\}$ and $(\bar{S}, S) = \{(v, w) \in E : v \in \bar{S} \wedge w \in S\}$. Then, $[S, \bar{S}] = (S, \bar{S}) \cup (\bar{S}, S)$ and (S, \bar{S}) (resp. (\bar{S}, S)) is called the *forward* (resp. *backward*) edges of the cut. If $s \in S$ and $t \in \bar{S}$, then $[S, \bar{S}]$ is called an *s - t cut*. In the following, with the term cut we shall always refer to an s - t cut, unless it is stated otherwise.

The *capacity* $c[S, \bar{S}]$ of a cut $[S, \bar{S}]$ is defined as the sum of the capacities

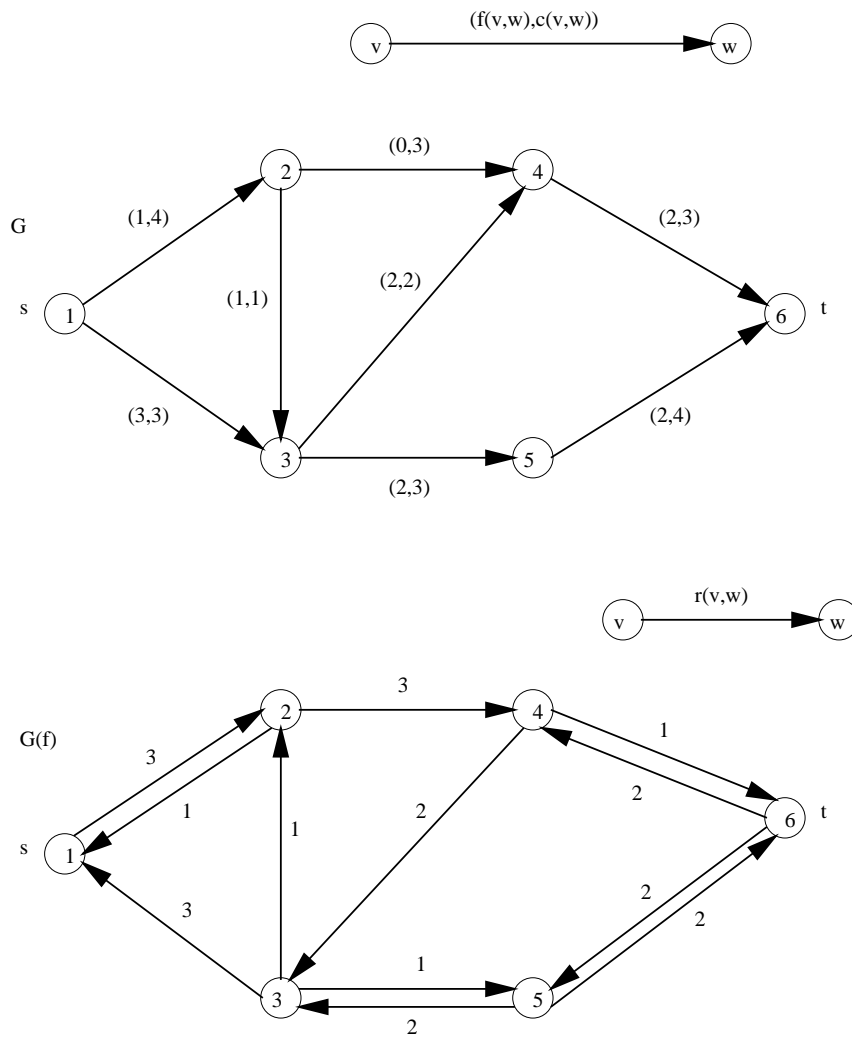


Figure 4.1: Network G with flow f and residual network $G(f)$.

of its forward edges. That is,

$$c[S, \bar{S}] = \sum_{(v,w) \in (S, \bar{S})} c(v, w)$$

Clearly, $c[S, \bar{S}]$ is an upper bound on the amount of flow that can be sent from S to \bar{S} . Similarly, the *residual capacity* of a cut $[S, \bar{S}]$ is defined as the sum of the residual capacities of its forward edges.

An *s-t minimum cut* (or simply *min-cut*) is an *s-t* cut whose capacity is minimum among all *s-t* cuts. The *flow across* an *s-t* cut $[S, \bar{S}]$ (or simply the total net flow) $f(S, \bar{S})$ is defined as the total flow sent from S to \bar{S} minus the total flow returned from \bar{S} to S . That is,

$$f(S, \bar{S}) = \sum_{(v,w) \in (S, \bar{S})} f(v, w) - \sum_{(w,v) \in (\bar{S}, S)} f(w, v)$$

The following lemma tells us that the net flow across any *s-t* cut is equal to the flow value $|f|$.

Lemma 4.2.1 *For any flow f and any s-t cut $[S, \bar{S}]$, $|f| = f(S, \bar{S})$.*

Proof.

$$\begin{aligned} |f| &= \sum_{(s,v) \in E} f(s, v) - \sum_{(v,s) \in E} f(v, s) \\ &= \sum_{(s,v) \in E} f(s, v) - \sum_{(v,s) \in E} f(v, s) + 0 \\ &= \sum_{(s,v) \in E} f(s, v) - \sum_{(v,s) \in E} f(v, s) + \sum_{\substack{v \in S - \{s\} \\ (v,w) \in E}} f(v, w) - \sum_{\substack{v \in S - \{s\} \\ (w,v) \in E}} f(w, v) \\ &= \left[\sum_{(s,v) \in E} f(s, v) + \sum_{\substack{v \in S - \{s\} \\ (v,w) \in E}} f(v, w) \right] - \left[\sum_{(v,s) \in E} f(v, s) + \sum_{\substack{v \in S - \{s\} \\ (w,v) \in E}} f(w, v) \right] \\ &= \sum_{\substack{v,w \in S \\ (v,w) \in E}} f(v, w) + \sum_{\substack{v \in S, w \in \bar{S} \\ (v,w) \in E}} f(v, w) - \sum_{\substack{v,w \in S \\ (w,v) \in E}} f(w, v) - \sum_{\substack{v \in S, w \in \bar{S} \\ (w,v) \in E}} f(w, v) \\ &= 0 + \sum_{\substack{v \in S, w \in \bar{S} \\ (v,w) \in E}} f(v, w) - \sum_{\substack{v \in S, w \in \bar{S} \\ (w,v) \in E}} f(w, v) \\ &= \sum_{(v,w) \in (S, \bar{S})} f(v, w) - \sum_{(w,v) \in (\bar{S}, S)} f(w, v) \\ &= f(S, \bar{S}). \end{aligned}$$

■

Observe that $|f|$ is also less than or equal to the capacity of any s - t cut. Intuitively, any s - t flow has to pass through any s - t cut in G , since such a cut divides G into two disjoint components and consequently $|f|$ cannot exceed the capacity of any s - t cut. The next lemma gives a formal proof of this claim.

Lemma 4.2.2 *For any flow f and any s - t cut $[S, \bar{S}]$, $|f| \leq c[S, \bar{S}]$.*

Proof. By Lemma 4.2.1,

$$\begin{aligned} |f| &= \sum_{(v,w) \in (S, \bar{S})} f(v,w) - \sum_{(w,v) \in (\bar{S}, S)} f(w,v) \\ &\leq \sum_{(v,w) \in (S, \bar{S})} c(v,w) \quad (\text{because } 0 \leq f(v,w) \leq c(v,w), \forall (v,w) \in E) \\ &= c[S, \bar{S}]. \end{aligned}$$

■

The above lemma implies the Max-Flow Min-Cut theorem.

Theorem 4.2.1 (Max-flow Min-cut Theorem – I) *If for some flow f and some s - t cut $[S, \bar{S}]$ we have that $|f| = c[S, \bar{S}]$, then f is a max-flow and $[S, \bar{S}]$ is a min-cut.*

Proof. Let f^* be a maximum flow and $[S^*, \bar{S}^*]$ be a minimum cut. By Lemma 4.2.2, $c[S^*, \bar{S}^*] \geq |f^*|$. By assumption we also have, $|f^*| \geq |f| = c[S, \bar{S}] \geq c[S^*, \bar{S}^*]$. Therefore, $|f^*| = |f| = c[S, \bar{S}] = c[S^*, \bar{S}^*]$. ■

4.3 Four Fundamental Theorems

Theorem 4.3.1 (Augmenting Path Theorem) *A flow f is maximum iff there is no augmenting path for f .*

Proof. (\Rightarrow) Assume on the contrary that there is an augmenting path for f , i.e., a directed s - t path in $G(f)$. By definition, every edge in P has a positive residual capacity. Let $r(P)$ be the minimum residual capacity of all edges in P . Since $r(P) > 0$, we can send additional flow of value $|f| + r(P)$ from s to t along P . Consequently, f is not a maximum flow, a contradiction.

(\Leftarrow) Since there is no augmenting path for f , the vertex set of $G(f)$ is partitioned into two subsets A and $\bar{A} = V - A$, such that $s \in A$ and $t \in \bar{A}$.

Note that A is the set of vertices z for which there exists a directed s - z path in $G(f)$. Then, $[A, \bar{A}]$ is an s - t cut for which $r(v, w) = 0$, $\forall (v, w) \in (A, \bar{A})$. Since $r(v, w) = c(v, w) - f(v, w) + f(w, v)$, $c(v, w) - f(v, w) \geq 0$ and $f(w, v) \geq 0$, the fact that $r(v, w) = 0$ implies that $c(v, w) - f(v, w) = 0$ and $f(w, v) = 0$, i.e., $f(v, w) = c(v, w)$, $\forall (v, w) \in (A, \bar{A})$, and $f(w, v) = 0$, $\forall (w, v) \in (\bar{A}, A)$. By Lemmas 4.2.1 and 4.2.2, $|f| = c[A, \bar{A}]$, and by Theorem 4.2.1 f is a maximum flow. ■

The augmenting path theorem provides an alternative proof to the max-flow min-cut theorem.

Theorem 4.3.2 (Max-flow Min-cut Theorem – II) *The maximum value of an s - t flow equals the minimum capacity of any s - t cut.*

Proof. Let f^* be a maximum s - t flow and let c^* be the capacity of a minimum s - t cut. Since f^* is maximum, by Theorem 4.3.1, there is no augmenting path for f^* and consequently (by the proof of Theorem 4.3.1) there is an s - t cut $[A, \bar{A}]$ for which $|f^*| = c[A, \bar{A}] \geq c^*$. However, $|f^*| \leq c^*$ by Lemma 4.2.2. Hence, $|f^*| = c^*$. ■

The augmenting path theorem suggests the following algorithm, called *augmenting path algorithm*, to find a maximum flow:

- (1) Start with the zero flow.
- (2) Repeat the following, until there is no directed s - t path in $G(f)$:
 - (2.a) Find a directed s - t path P in $G(f)$.
 - (2.b) Let $\Delta f = \min\{r(v, w) : (v, w) \in P\}$.
 - (2.c) Augment Δf units of flow along P , i.e., $|f| = |f| + \Delta f$.
 - (2.d) Update $G(f)$.

Let us now discuss a more detailed implementation of the augmenting path algorithm (i.e., how an augmenting path is identified) and whether the algorithm terminates. The detailed implementation is called the *labeling algorithm* (Ford and Fulkerson, 1958).

The labeling algorithm uses a search method (see Chapter 2) to find a directed s - t path in $G(f)$. At any step, the algorithm partitions V into two sets: *labeled* vertices (those $v \in V$ for which there is a directed s - v path in $G(f)$), and *unlabeled* vertices (those $v \in V$ for which there is not yet a directed s - v path in $G(f)$). The algorithm picks any labeled vertex and scans its adjacency list in $G(f)$ to label additional vertices. When t is labeled, a directed s - t path has been discovered and consequently the maximum possible flow is sent along this path. Then, the algorithm erases all labels and repeats the process. The labeling algorithm terminates when all

labeled vertices have been scanned but vertex t remains unlabeled (implying that there is no directed s - t path in $G(f)$). A less informal description of the algorithm follows.

Algorithm Labeling.

Input: Network $G = (V, E)$ with nonnegative edge capacities and two distinguished vertices $s, t \in V$.

Output: A maximum s - t flow f (when the algorithm terminates).

Method:

```

label  $t$ ;  $|f| = 0$ ;
while  $t$  is labeled do
    unlabel all vertices;
    for every  $v \in V$  do  $pred(v) = 0$ ;
    label  $s$ ;  $L = \{s\}$ ;
    while  $L \neq \emptyset$  or  $t$  is unlabeled do
        (* labeling step *)
        remove a vertex  $v$  from  $L$ ;
        for each  $(v, w) \in E(f)$  do
            if  $r(v, w) > 0$  and  $w$  is unlabeled then
                label  $w$ ;  $pred(w) = v$ ;
                 $L = L \cup \{w\}$ ;
            od
        (*augmentation step*)
    if  $t$  is labeled then
        use  $pred(\cdot)$  to find an augmenting path  $P$ ;
         $\Delta f = \min\{r(v, w) : (v, w) \in P\}$ ;
        augment  $\Delta f$  units of flow along  $P$ ;
         $|f| = |f| + \Delta f$ ;
        update  $G(f)$ ;
    od
od

```

The algorithm either performs an augmentation step, or terminates because it cannot label t . Hence, the correctness of the algorithm comes immediately by Theorem 4.3.1. Before discussing the complexity of the labeling algorithm, we give one more result.

Theorem 4.3.3 (Integrality Theorem) *If all edge capacities are integers, the max-flow is integral.*

Proof. Induction on the number of augmentations. We start with $f_1 = 0$. If there is no directed s - t path in $G(f_1)$, then f_1 is a maximum flow by Theorem 4.3.1 and we are done. Otherwise, we augment the flow along an s - t path in $G(f_1)$ by an amount $\Delta f_1 > 0$ which is integral. Consequently, the new flow value $|f_2| = |f_1| + \Delta f_1$ is integral and $|f_2| > |f_1|$. Continuing in this way, it is easy to see (by a simple induction argument) that we get a sequence $|f_1| < |f_2| < |f_3| < \dots$ of integers which by Lemma 4.2.2 is bounded by some number, say $|f_k|$. Hence, the labeling algorithm must terminate with an integral flow f_k for which there is no directed s - t path in $G(f_k)$ and consequently (by Theorem 4.3.1) f_k is a maximum flow. ■

Let us now determine the complexity of the labeling algorithm. In every iteration (except for the last one where t is not labeled), the algorithm performs a labeling step (which is actually a search algorithm) and an augmentation step. Clearly both steps take $O(m)$ time. Observe that the number of augmentation steps determine the number of labeling steps. Hence, the running time of the algorithm is $O(m) \times$ (number of augmentation steps). If all edge capacities are integers and C is the largest edge capacity, then the capacity of an s - t min-cut is bounded by nC . (To see this, consider the cut $[\{s\}, V - \{s\}]$.) Since the flow value is increased by at least one unit in each augmentation, the maximum number of augmentation steps is nC . Therefore:

Theorem 4.3.4 *The labeling algorithm solves the max-flow problem, in a network with integer edge capacities whose largest value is C , in $O(nmC)$ time.*

If edge capacities are rational, the labeling algorithm terminates but it may require an exponential number of steps. The same happens if the integer capacities are very large, e.g., $C = 2^n$, and the algorithm chooses a “bad” augmenting path. Consider the example shown in Figure 4.2. If the labeling algorithm alternates between these two augmenting paths, it will need $M = 2^{30}$ augmentation steps to terminate.

If edge capacities are irrational, the algorithm may not even terminate, or it terminates converging to a flow value strictly smaller than the max-flow value.

We will now present the last fundamental theorem on network flows. Let the *path flow* $f(P)$ of a path P be the flow passing through P . The *cycle flow* $f(W)$ of a cycle W is defined similarly.

Theorem 4.3.5 (Flow Decomposition Theorem) *Every path and cycle flow has a unique representation as nonnegative edge flows. Conversely,*

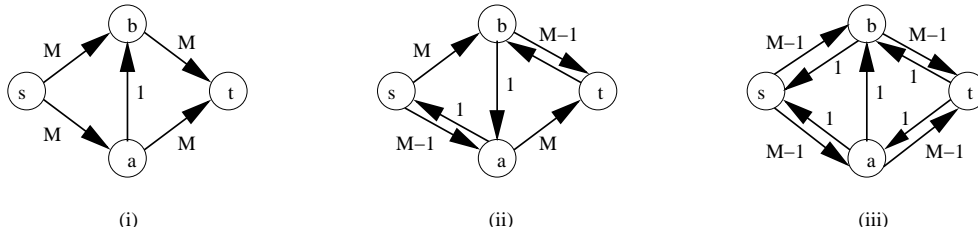


Figure 4.2: (i) Initial network; $M = 2^{30}$. (ii) Residual network after augmenting 1 unit of flow along path s, a, b, t . (iii) Residual network after augmenting 1 unit of flow along path s, b, a, t .

every nonnegative edge flow f can be represented as a collection of path and cycle flows such that at most m paths and cycles have nonzero flow.

Proof. The flow $f(v, w)$ on every edge $(v, w) \in E$ equals the sum of the flows $f(P)$ and $f(W)$ for all paths P and W having (v, w) as their edge. Hence, each path and cycle flow determines uniquely an edge flow.

To prove the converse, assume that there is a flow f on every edge in G with $|f| \neq 0$. Then, do the following. Start with any vertex v that has at least one outgoing edge (v, w) with positive flow. Traverse (v, w) and repeat this step from w . At some future step, either a vertex is repeated, or t is reached. In the former case, a cycle W has been found and let $f(W) = \min\{f(u, v) : (u, v) \in W\}$. Set $f(u, v) = f(u, v) - f(W)$, $\forall (u, v) \in W$, and repeat the process. In the latter case, a path P has been found and let $f(P) = \min\{f(u, v) : (u, v) \in P\}$. Set $f(u, v) = f(u, v) - f(P)$, $\forall (u, v) \in P$, and repeat the process. Terminate when the flow value on every edge is zero. Clearly, the original flow is the sum of the flows in the paths and cycles determined by the above method. Every time, during the above process, we find a path or a cycle, we set the flow value of at least one edge to zero. Consequently, the path and cycle representation of the given flow f contains at most m paths and cycles with nonzero flow. ■

Chapter 5

Maximum Flows: Improved Algorithms

5.1 Introduction

The augmenting path algorithm, presented in Chapter 4, has two significant drawbacks: (i) its worst-case complexity is bad when edge capacities (i.e., C) are large, and (ii) in networks with irrational capacities the algorithm converges to a non-optimal solution. Recall that the bottleneck is the total number of augmentation steps (or simply augmentations) which is $O(nC)$.

In this chapter, we discuss how we can improve on the $O(nC)$ bound for the total number of augmentations required by the augmenting path algorithm. We shall consider three approaches:

- (1) Augment flow along a path with *large* residual capacity. For example, along an augmenting path with maximum residual capacity. This idea results in a total of $O(m \log C)$ augmentations. As we will see, it also suffices to augment flow along an augmenting path with “sufficiently” large residual capacity (i.e., not necessarily maximum) and still being able to perform the same number of augmentations.
- (2) Restrict the choice of augmenting paths using a combinatorial strategy which is independent of the edge capacities. One such strategy is to augment flow along a “shortest path” from the source s to the sink t , where a shortest path is defined as a directed s - t path with the minimum number of edges in the residual network.
- (3) Relax the flow conservation constraints at intermediate steps of the algorithm. As a consequence, each flow change is not required to

be an augmentation starting at the source s and terminating at the sink t . This idea introduces the so-called “preflow-push” algorithms. These algorithms look for shortest augmenting paths, but they do not send flows along s - t augmenting paths; instead, the flows are sent on individual edges.

5.2 Maximum Capacity and Capacity Scaling Algorithms

In this section, we give algorithms that implement the first approach. We assume that all edge capacities of our input network G are nonnegative integers and that the largest such capacity has value C .

We start with the *maximum capacity augmenting path algorithm*. The structure of the algorithm is the same as those of Chapter 4; only the choice of an augmenting path changes. The algorithm always chooses as augmenting path a directed s - t path in the residual network with maximum residual capacity.

Let us discuss the complexity of this algorithm. Let f be any flow in G and let f^* be a max-flow. The flow decomposition theorem tells us that we can find at most m directed s - t paths whose residual capacities sum to $|f^*| - |f|$. Hence, the maximum capacity augmenting path P_{max} has residual capacity $r(P_{max}) \geq (|f^*| - |f|)/m$.

Consider a sequence of $2m$ augmentations. If each one of these augmentations augments $\geq (|f^*| - |f|)/2m$ units of flow, then at the end of this sequence of augmentations we will have a maximum flow. On the other hand, if any of this augmentation augments $< (|f^*| - |f|)/2m$ units of flow, then the new residual capacity of P_{max} is (by definition) at least $(|f^*| - |f|)/m - (|f^*| - |f|)/2m = (|f^*| - |f|)/2m$. This implies that the residual capacity of P_{max} has been reduced by a factor of (at least) 2. Consequently, after $2m$ augmentations either a max-flow has been found, or the residual capacity of P_{max} has been reduced by at least 2. Since $1 \leq r(P_{max}) \leq 2C$, after $O(m \log C)$ augmentations, a max-flow should be obtained.

Note, however, that although we reduced the total number of augmentations, the algorithm has to identify the maximum capacity augmenting path in every iteration. Such a path can be found using e.g., Dijkstra’s algorithm suitably modified, which implies that we may need more than $O(m)$ time per iteration.

We shall now describe a variation of the above algorithm which performs the same number of augmentations, but takes only $O(m)$ time per iteration.

This variation is called the *capacity scaling algorithm* (since the algorithm performs a kind of scaling to the edge capacities).

The idea (as mentioned before) is to augment along an augmenting path of *sufficiently large* residual capacity (instead of one with maximum residual capacity). The advantage is that such a path can be easily found in $O(m)$ time. Before giving the algorithm, we need some notation. Define the δ -*residual network* $G(f, \delta)$, with respect to a given flow f , to be the network containing edges with residual capacity at least δ . Note that $G(f, 1) = G(f)$ and $G(f, \delta)$ is a subgraph of $G(f)$. The capacity scaling algorithm is as follows.

Algorithm Capacity scaling.

Input: Network $G = (V, E)$ with nonnegative integral edge capacities, source s and sink t .

Output: An s - t max-flow f in G .

Method:

```

 $|f| = 0$ ;  $\delta = 2^{\lceil \log C \rceil}$ ;
while  $\delta \geq 1$  do
    while  $\exists$  a directed  $s$ - $t$  path in  $G(f, \delta)$  do
        find a directed  $s$ - $t$  path  $P$  in  $G(f, \delta)$ ;
         $\Delta f = \min\{r(v, w) : (v, w) \in P\}$ ;
        augment  $\Delta f$  units of flow along  $P$ ;
         $|f| = |f| + \Delta f$ ;
        update  $G(f, \delta)$ ;
    od
     $\delta = \delta/2$ ;
od

```

Let us refer to an iteration of the outer while-loop as a *scaling phase*, and to a total execution of the inner while-loop with a specific value of δ as a δ -scaling phase. Observe that a δ -scaling phase corresponds to an augmentation of at least δ units of flow. Since the algorithm starts with $\delta = 2^{\lceil \log C \rceil}$ and in every scaling phase reduces the value of δ by 2, it will perform a total of $1 + \lceil \log C \rceil = O(\log C)$ scaling phases. In the last scaling phase, $\delta = 1$ and thus $G(f, 1) = G(f)$. From the augmenting path theorem, it follows that the capacity scaling algorithm terminates with a maximum flow.

Let us now discuss the complexity of the algorithm. We claim that the algorithm performs at most $2m$ augmentations per scaling phase. To see this, consider a flow f' at the end of a δ -scaling phase. Let S be the subset

of vertices in $G(f', \delta)$ such that $\forall v \in S$ there exists a directed s - v path in $G(f', \delta)$. Since there is no directed s - t path in $G(f', \delta)$, the set S defines an s - t cut $[S, \bar{S}]$, where $\bar{S} = V - S$. Moreover, every edge in $[S, \bar{S}]$ has residual capacity less than δ . Hence, the residual capacity $r(S, \bar{S})$ of $[S, \bar{S}]$ is at most $m\delta$. From the definition of the residual capacity, we get:

$$\begin{aligned}
r(S, \bar{S}) &= \sum_{(v,w) \in (S, \bar{S})} r(v, w) \\
&= \sum_{\substack{(v,w) \in (S, \bar{S}) \\ (w,v) \in (\bar{S}, S)}} (c(v, w) - f'(v, w) + f'(w, v)) \\
&= \sum_{(v,w) \in (S, \bar{S})} c(v, w) - \left[\sum_{(v,w) \in (S, \bar{S})} f'(v, w) - \sum_{(w,v) \in (\bar{S}, S)} f'(w, v) \right] \\
&\geq |f^*| - f'(S, \bar{S}) \\
&= |f^*| - |f'|
\end{aligned}$$

Consequently, $|f^*| - |f'| \leq m\delta$. In the next scaling phase, each augmentation augments the flow by at least $\delta/2$ units. Hence, this scaling phase performs at most $2m$ augmentations and thus our claim is proved. It follows that the total number of augmentations is $O(m \log C)$. As discussed in the labeling algorithm, the identification of an augmenting path and the update of $G(f, \delta)$ take $O(m)$ time. The preceding discussion is summarized as follows.

Theorem 5.2.1 *The capacity scaling algorithm solves the max-flow problem in time $O(m^2 \log C)$, in a network with integral edge capacities whose largest value is C .*

5.3 Shortest Augmenting Path Algorithm

The shortest augmenting path algorithm chooses an s - t path with the minimum number of edges (called shortest path), to augment the flow from s to t . Since “shortest” here means “shortest with respect to (w.r.t.) the number of edges”, a natural approach to implement this method is by employing a breadth-first search (BFS) in the residual network (i.e., implementing the set L in the labeling algorithm as a queue; see Chapter 2). Consequently, each iteration would take $O(m)$ time. We will see how this time can be reduced to $O(n)$ using the concept of *distance labels*. This, in combination

with the fact that the total number of augmentations is $O(nm)$, will result in an $O(n^2m)$ time algorithm for the max-flow problem (i.e., a strongly polynomial algorithm not depending on edge capacities).

5.3.1 Distance Labels, Admissible Edges and Admissible Paths

For a network $G = (V, E)$, let d be a *distance function* from V to the set of nonnegative integers. A distance function is called *valid* w.r.t. flow f if the following two *validity conditions* are satisfied:

- (1) $d(t) = 0$.
- (2) $d(v) \leq d(w) + 1, \forall (v, w) \text{ in } G(f)$.

The quantity $d(v)$ is called the *distance label* of vertex v . The next two properties establish useful results that will be required later.

Proposition 5.3.1 *If the distance labels are valid, then $d(v)$ is a lower bound on the length (i.e., the number of edges) of a shortest v - t path in the residual network.*

Proof. Let $v = v_1, v_2, \dots, v_{k+1} = t$ be any v - t path of length k . From the validity conditions, we have:

$$\begin{aligned} d(v_k) &\leq d(v_{k+1}) + 1 = d(t) + 1 = 1 \\ d(v_{k-1}) &\leq d(v_k) + 1 \\ &\vdots \\ d(v) = d(v_1) &\leq d(v_2) + 1 \end{aligned}$$

Summing up the above inequalities, we get $d(v) \leq k$. Hence, $d(v)$ is also a lower bound on the length of a shortest v - t path. ■

Proposition 5.3.2 *If $d(s) \geq n$, then the residual network has no directed s - t path.*

Proof. By Proposition 5.3.1, $d(s)$ is a lower bound on the length of a shortest s - t path. Thus, $d(s) \leq n - 1$, since no path in the residual network can be of length greater than $n - 1$. Consequently, $d(s) \geq n$ implies that there is no s - t path in the residual network. ■

For a vertex v , if $d(v)$ equals the length of a shortest v - t path in the residual network, then $d(v)$ is called the *exact* distance label of v . Exact

distance labels can be computed in $O(m)$ time by performing a backward BFS starting at t (i.e., by reversing temporarily the direction of every edge and then perform the BFS).

An edge (v, w) is called *admissible* if $d(v) = d(w) + 1$; otherwise, it is called *inadmissible*. An s - t path in the residual network is called admissible if all of its edges are admissible.

Proposition 5.3.3 *An admissible path is a shortest s - t augmenting path.*

Proof. Let P be an admissible path. Since every edge $(v, w) \in P$ is admissible, we have: (i) $r(v, w) > 0$; and (ii) $d(v) = d(w) + 1$. From (i) we have that P is an s - t augmenting path, and from (ii) that if the length of P is k , then $d(s) = k$. Since $d(s)$ is a lower bound on an s - t shortest path (by Proposition 5.3.1), it follows that P is a shortest s - t augmenting path. ■

5.3.2 The Algorithm

The shortest augmenting path algorithm augments flow along admissible paths. An admissible path is constructed incrementally by adding one edge at a time. The algorithm works as follows. During its execution, the algorithm maintains a *partial admissible path* P (i.e., an s - v path, for some $v \in V$, consisting only of admissible edges). At every iteration, the algorithm tries to add more admissible edges to P . In doing so, it performs an *advance* or a *retreat* operation from the last vertex v (called the *current vertex*) of P . If v is incident to an admissible edge (v, w) , the algorithm performs an advance operation and adds (v, w) to P . Otherwise, a retreat operation is performed: $d(v)$ is *relabelled* by increasing its value to $\min\{d(w) + 1 : (v, w) \in E \text{ and } r(v, w) > 0\}$ (this increment is necessary in order to create new admissible edges), and the last edge (u, v) of P is deleted. These operations are repeated until P reaches t . The above process is repeated until the flow in G is maximum. A less informal description of the algorithm follows.

Algorithm Shortest augmenting path.

Input: Network $G = (V, E)$ with nonnegative real edge capacities, source s and sink t .

Output: A maximum s - t flow f in G .

Method:

```

 $|f| = 0; v = s;$ 
perform a backward BFS starting at  $t$  to compute exact distance
labels  $d(v), \forall v \in V;$ 
while  $d(s) < n$  do
    if  $\exists$  admissible edge  $(v, w)$  then
        advance( $v$ );
        if  $v = t$  then
            augment;  $v = s;$ 
        else
            retreat( $v$ );
    od

```

Procedure advance(v);

(* let (v, w) be the identified admissible edge incident on v *)

```

 $pred(w) = v;$ 
 $v = w;$ 

```

end-of-procedure

Procedure retreat(v);

```

 $d(v) = \min\{d(w) + 1 : (v, w) \in E \text{ and } r(v, w) > 0\};$ 
if  $v \neq s$  then  $v = pred(v);$ 

```

end-of-procedure

Procedure augment;

```

use  $pred(\cdot)$  to find a shortest  $s$ - $t$  augmenting path  $P;$ 
 $\Delta f = \min\{r(v, w) : (v, w) \in P\};$ 
augment  $\Delta f$  units of flow along  $P;$ 
 $|f| = |f| + \Delta f;$ 
update  $G(f);$ 

```

end-of-procedure

Let us now discuss the correctness of the algorithm. To show that the shortest augmenting path algorithm computes a max-flow, it suffices to show that when the algorithm terminates (i.e., when $d(s) \geq n$) the residual network does not contain an s - t augmenting path (i.e., an admissible path, by Proposition 5.3.3). In view of Propositions 5.3.1 and 5.3.2, the proof boils

down to showing that the algorithm maintains valid distance labels during its execution. The next lemma provides a proof of this fact.

Lemma 5.3.1 *The shortest augmenting path algorithm maintains valid distance labels at each step. Further, each retreat (i.e., relabel) operation strictly increases the distance label of a vertex.*

Proof. We prove the lemma by induction on the number of operations. Since the advance operation does not affect distance labels, the induction is performed on the number of augment and retreat operations.

Initially, all distance labels are valid. For the induction hypothesis, assume that distance labels are valid (i.e., satisfy the validity conditions) prior to some augment or retreat operation. We have to show that the distance labels satisfy the validity conditions in two cases: (1) after an augment operation, and (2) after a retreat operation.

Consider case (1). Note that an augmentation of flow along an edge (v, w) may delete this edge from the residual network, but does not change either $d(v)$ or $d(w)$. On the other hand, an augmentation on (v, w) may create an additional edge (w, v) with $r(w, v) > 0$ in the residual network. Consequently, the distance labels must satisfy an additional inequality for that edge, namely $d(w) \leq d(v) + 1$. But, by Proposition 5.3.3, an augmenting path is an admissible path and therefore $d(v) = d(w) + 1$. Hence, $d(v)$ and $d(w)$ satisfy the above inequality. Consequently, after an augment operation distance labels remain valid.

Consider now case (2). Note that each retreat operation changes $d(v)$. Thus, we have to show that every incoming and outgoing edge incident on v satisfies the validity conditions w.r.t. the new distance labels, say $d'(v)$. A retreat operation is performed when there is no admissible edge incident on v ; i.e., $\nexists (v, w) \in E$ such that $r(v, w) > 0$ and $d(v) = d(w) + 1$. Since (v, w) belongs to the residual network, we must also have $d(v) \leq d(w) + 1$. Consequently, $d(v) < d(w) + 1$ for all $(v, w) \in E$ with $r(v, w) > 0$. Hence, $d(v) < \min\{d(w) + 1 : (v, w) \in E \text{ and } r(v, w) > 0\} = d'(v)$, which is the new distance label after the retreat operation. Thus, relabeling (performed during a retreat) preserves the validity of distance labels for all outgoing edges incident on v , and also strictly increases the distance label of v . Consider now an incoming edge (u, v) incident on v . By the induction hypothesis, $d(u) \leq d(v) + 1$. Since $d(v) < d'(v)$, this inequality is satisfied after the execution of a retreat operation. Hence, after a retreat operation, distance labels remain valid for all incoming edges incident on v . ■

Lemma 5.3.1 and the discussion preceding it, imply the following.

Theorem 5.3.1 *The shortest augmenting path algorithm computes correctly a max-flow.*

Before proceeding to the complexity, we describe how the algorithm searches for admissible edges incident on a vertex v and how the algorithm relabels the distance label of a vertex. We also establish two useful results.

Let $A(v)$ be the adjacency list of v . Recall that $A(v)$ determines all the edges incident on v . Hence, for convenience we shall refer to $A(v)$ as the list of edges incident on v . The order of edges in $A(v)$ is arbitrary (but fixed). During the execution of the shortest augmenting path algorithm, every vertex v has an associated *current edge* representing the edge of $A(v)$ which is the next candidate for admissibility testing. Initially, the current edge is the first edge in $A(v)$. Whenever during its execution, the algorithm has to search for an admissible edge incident on v , checks first if the current edge of v is admissible. If not, the next edge in $A(v)$ becomes the current edge. The algorithm repeats this process until either an admissible edge is found, or the end of $A(v)$ is reached. In the latter case, it performs a relabeling at v and sets the current edge of v equal to the first edge in $A(v)$.

The above implementation establishes the following (important) result.

Lemma 5.3.2 *If the shortest augmenting path algorithm relabels any vertex at most k times, the total time taken for finding admissible edges and relabeling the vertices is $O(km)$.*

Proof. It suffices to prove that the algorithm performs only one scan of $A(v)$ to identify the admissible edges incident on v , before a relabel (i.e., a retreat) operation is invoked. This is clearly true if the current edge is the first one in $A(v)$ and it is the only admissible edge incident on v . For the general case, assume that the current edge is not the first one in $A(v)$. To show that the algorithm performs only one scan, we have to show that when it reaches the end of $A(v)$ there is indeed no admissible edge in $A(v)$. This reduces in showing that if an edge (v, w) is inadmissible in previous iterations, it remains so until $d(v)$ increases (i.e., until a relabel operation is performed). To see this, observe that $d(v) \leq d(w) + 1$ during the execution of the algorithm (by Lemma 5.3.1). Since (v, w) is inadmissible we must have that either $d(v) < d(w) + 1$, or $d(v) = d(w) + 1$ and $r(v, w) = 0$. Since during the search for admissible edges residual capacities do not change, we have that if $r(v, w) = 0$ in previous iterations, it remains so during the search. Consequently, for (v, w) to become admissible $d(v)$ has to be increased which happens only after a relabel operation, i.e., when the algorithm reaches the end of $A(v)$.

The time spent to relabel a vertex is proportional to $|A(v)|$, since the retreat operation scans each edge in $A(v)$ once in order to compute the new distance label. Therefore, for every v the total time spent for finding admissible edges and relabeling it is $O(|A(v)|)$. Consequently, after at most k relabelings the total time taken for finding admissible edges and doing the relabelings is $O(k \sum_{v \in V} |A(v)|) = O(km)$. ■

The following two lemmas are useful in establishing the running time of the shortest augmenting path algorithm.

Lemma 5.3.3 *If the shortest augmenting path algorithm relabels every vertex at most k times, then it saturates edges (i.e., reduces their residual capacity to zero) at most $km/2$ times.*

Proof. Since the original network has m edges, it suffices to show that the algorithm saturates every edge at most $k/2$ times after the k relabelings. Since k relabelings increase the distance labels at most k times, we have to show that between two consecutive saturations of an edge (v, w) both $d(v)$ and $d(w)$ must increase by at least 2 units.

Assume that an augmentation saturates edge (v, w) . Since (v, w) is admissible, we have $d(v) = d(w) + 1$. Before the next saturation of (v, w) , the algorithm must send flow along edge (w, v) which implies that (w, v) must be admissible at that time, i.e., $d'(w) = d'(v) + 1$. In the next saturation, again (v, w) must be admissible implying $d''(v) = d''(w) + 1$. By Lemma 5.3.1, $d'(v) \geq d(v)$ and $d''(w) \geq d'(w)$. Consequently, $d''(v) = d''(w) + 1 \geq d'(w) + 1 = (d'(v) + 1) + 1 \geq d(v) + 2$. Similarly, we have $d(w) = d(v) - 1 \leq d'(v) - 1 = d'(w) - 2 \leq d''(w) - 2$, i.e., $d''(w) \geq d(w) + 2$.

Therefore, between every two consecutive saturations of an edge (v, w) both $d(v)$ and $d(w)$ increase by at least 2 units. ■

Lemma 5.3.4 *In the shortest augmenting path algorithm: (i) Each distance label increases at most n times and consequently the number of relabel operations is at most n^2 . (ii) The number of augment operations is at most $nm/2$.*

Proof. (i) Each relabel operation at vertex v increases $d(v)$ by at least 1 unit. After the algorithm relabels v (at most) n times, $d(v) \geq n$. Consequently, v will never be chosen by the algorithm to participate in an admissible path, because every vertex u in such a path satisfies $d(u) < d(s) < n$. Hence, the algorithm relabels v at most n times and the total number of relabel operations is at most n^2 .

(ii) Lemma 5.3.3 in combination with (i) above, implies that the algorithm saturates edges at most $nm/2$ times. Since each augment operation saturates at least one edge, it follows that the total number of augment operations is at most $nm/2$. ■

We are now ready to establish the running time of the algorithm.

Theorem 5.3.2 *The shortest augmenting path algorithm runs in $O(n^2m)$ time.*

Proof. Lemma 5.3.2 and Lemma 5.3.4(ii) imply that the total time for finding admissible edges and relabeling vertices is $O(nm)$.

The total number of augmentations is (by Lemma 5.3.4(ii)) $O(nm)$. Since each augment operation takes $O(n)$ time (because every admissible path can be now found in $O(n)$ time), it follows that the total time taken overall augmentations is $O(n^2m)$.

By Lemma 5.3.4(i), the total time overall retreat operations is $O(n^2)$, because every retreat relabels a vertex.

To bound the number of advance operations, note that a partial admissible path has at most n vertices. To find such a path the algorithm must perform at most n advance operations after each augmentation and one advance after each retreat operation (which deletes an edge from the partial admissible path). Consequently, the total number of advance operations is bounded by $O(n^2m + n^2) = O(n^2m)$.

The claimed running time follows now by the above bounds. ■

5.4 Preflow-Push Algorithms

In the previous sections, we studied various algorithms for solving the max-flow problem, all of which were based on the augmenting path method. In this section we study a different method for solving the max-flow problem. This method determines a new family of algorithms, called *preflow-push* algorithms. These algorithms are better than the augmenting path algorithms both in theory (i.e., they achieve better running times) and in practice (their implementations achieve faster execution times).

A major drawback in the augmenting path algorithms is the time to send additional flow along the augmenting path which can be $O(n)$ in the worst-case. The preflow-push algorithms overcome this drawback and improve substantially the running time.

As we already saw, the basic idea in the augmenting path algorithms is to send flow along a path. Sending flow along a path decomposes into the

elementary operation of sending flow along edges. Hence, sending Δf units of flow along a path with ℓ edges, decomposes into ℓ elementary operations of sending Δf units of flow along each edge of the path. Such an elementary operation is called a *push*. The preflow-push algorithms push flow on individual edges instead of augmenting paths. Because of this fact, these algorithms at intermediate steps do not satisfy the flow conservation constraint: they allow the flow entering a vertex to exceed the flow leaving the vertex. This situation is known as a *preflow*. Formally, a *preflow* is a nonnegative real-valued function f on the edges of a network $G = (V, E)$ such that f satisfies the capacity constraint (i.e., $0 \leq f(v, w) \leq c(v, w)$, $\forall (v, w) \in E$) and the following relaxation of the flow conservation constraint:

$$\sum_{(w,v) \in E} f(w, v) - \sum_{(v,w) \in E} f(v, w) = e(v) \geq 0, \quad \forall v \in V - \{s\}.$$

The term $e(v)$ is called the *excess* of vertex v . The preflow-push algorithms maintain a preflow at each intermediate step (which at the end converges to a maximum flow). In a preflow, $e(v) \geq 0$ for all $v \in V - \{s\}$ ($e(t) \geq 0$, because there are no outgoing edges incident on t) and s is the only vertex with negative excess.

A vertex v for which $e(v) > 0$ is called an *active* vertex. We also make the convention that s and t are never active. Active vertices play an important role in the preflow-push algorithms. Preflow-push algorithms try at each intermediate step to achieve feasibility of the solution (i.e., reduce the excess of every vertex in $V - \{s, t\}$ to zero), in contrast with the augmenting path algorithms which guarantee a feasible solution at every step and try to achieve optimality. Therefore, the presence of active vertices imply that the preflow-push algorithm has not found yet a feasible solution.

The preflow-push algorithm mimics (in some sense) the shortest augmenting path algorithm. However, it does not wait to secure first an admissible path and then send the flow, but at every step it sends flow through admissible edges. Informally, the preflow-push algorithm works as follows. It selects an active vertex and tries to remove its excess by pushing flow to its neighbors. The flow is sent to those vertices that are “closer” to t w.r.t. distance labels (as it happens with the shortest augmenting path algorithm). However, observe that sending flow to t through vertices that are “close” to it is equivalent to pushing flow through admissible edges. Therefore, as in the shortest augmenting path algorithm, the flow is sent only through admissible edges. If the selected active vertex has no admissible edges incident on it, then its distance label is increased in order to create admissible edges.

The algorithm terminates when there are no active vertices in the network. A less informal description of the algorithm follows.

Algorithm Generic preflow-push.

Input: Network $G = (V, E)$ with nonnegative real edge capacities, source s and sink t .

Output: A maximum s - t flow f in G .

Method:

```
(* Preprocessing phase *)
|f| = 0;
perform a backward BFS starting at  $t$  to compute exact distance
labels  $d(v)$ ,  $\forall v \in V$ ;
for each  $v : (s, v) \in E$  do  $f(s, v) = c(s, v)$ ;
 $d(s) = n$ ;
(* End of preprocessing phase *)
while  $\exists$  an active vertex in  $G$  do
    select an active vertex  $v$ ;
    push-relabel( $v$ );
od
```

Procedure push-relabel(v);

```
if  $\exists$  admissible edge  $(v, w)$  then
    push  $\Delta f = \min\{e(v), r(v, w)\}$  units of flow through  $(v, w)$ 
else
     $d(v) = \min\{d(w) + 1 : (v, w) \in E \text{ and } r(v, w) > 0\}$ ;
```

end-of-procedure

A push of Δf units of flow from v to w (along (v, w)) decreases both $e(v)$ and $r(v, w)$ and increases both $e(w)$ and $r(w, v)$. Such a push is called *saturating* if $\Delta f = r(v, w)$; otherwise, it is called *non-saturating*. Observe that a non-saturating push at vertex v reduces $e(v)$ to zero, i.e., v becomes inactive.

To get an intuitive description on how the algorithm works, consider the following visualization in terms of a water network, in which we want to send water from the source to the sink. Edges represent flexible water pipes, vertices represent pipes' joints (with a large reservoir), and $d(v)$ measures the distance of vertex v from the ground (i.e., how far is v above the ground). Flow in admissible edges is visualized as water flowing downhill. Initially, the source is moved upwards, so water flows to its neighbors. In general,

water flows downhill towards the sink. If at some point, water flow becomes trapped at some vertex v which has no downhill neighbors, then v is moved upwards (and higher than its neighbors) so that the water can again flow from v to its neighbors. After repeating this process a number of times, there will be a point after which no more water can flow into the sink. Also, as vertices are moved upwards, the remaining excess flow eventually flows back towards the source. The algorithm terminates when all the water flows into the sink or flows back to the source.

Let us concentrate for a moment on the preprocessing phase of the generic preflow-push algorithm. This phase plays an important role, for the following reasons: (a) It gives to every vertex adjacent to s a positive excess and consequently the algorithm has some active vertices to start with. (b) It saturates all the edges incident on s . As a consequence, none of these edges is admissible. (c) Setting $d(s) = n$ satisfies the validity conditions of distance labels and by Proposition 5.3.2 there is no s - t augmenting path in the residual network. Further, since distance labels are nondecreasing, there will be no s - t directed path in the residual network during subsequent iterations. Consequently, there will be no need to push flow from s again.

In view of the above, it is easy to establish the correctness of the algorithm, i.e., that the generic preflow-push algorithm finds a max-flow upon termination. Note that the algorithm terminates when $e(v) = 0$, for every $v \in V - \{s, t\}$ (i.e., when the excess of the flow resides on s or t). Upon termination, the residual network does not have an s - t directed path (because $d(s) = n$), and by the augmenting path theorem the flow must be maximum. The max-flow value equals $e(t)$.

To establish the complexity of the algorithm we need some intermediate results. The following lemma is crucial in the analysis.

Lemma 5.4.1 *At any step during the execution of the generic preflow-push algorithm, there is a directed v - s path in the residual network for every vertex v with $e(v) > 0$.*

Proof. Assume, for the purpose of contradiction, that there is no directed v - s path in the residual network $G(f)$ w.r.t. the preflow f , for some $v \in V$ with $e(v) > 0$. Let $X = \{w : \exists \text{ a directed } v\text{-}w \text{ path in } G(f)\}$ and let $\bar{X} = V - X$. Clearly, $s \in \bar{X}$. Let (w, u) be an edge such that $w \in X$ and $u \in \bar{X}$. Then, $r(w, u) = 0$. Consequently, $f(u, w) = 0$ (otherwise, $r(w, u) > 0$). Then,

$$\sum_{w \in X} e(w) = \sum_{\substack{u \in V \\ w \in X}} f(u, w) - \sum_{\substack{u \in V \\ w \in X}} f(w, u)$$

$$\begin{aligned}
&= \sum_{\substack{u \in \bar{X} \\ w \in X}} f(u, w) + \sum_{\substack{u \in X \\ w \in X}} f(u, w) - \sum_{\substack{u \in X \\ w \in X}} f(w, u) - \sum_{\substack{u \in \bar{X} \\ w \in X}} f(w, u) \\
&= 0 + \left[\sum_{\substack{u \in X \\ w \in X}} f(u, w) - \sum_{\substack{u \in X \\ w \in X}} f(w, u) \right] - \sum_{\substack{u \in \bar{X} \\ w \in X}} f(w, u) \\
&= 0 - \sum_{\substack{u \in \bar{X} \\ w \in X}} f(w, u) \\
&\leq 0
\end{aligned}$$

The term in brackets (in the third line) equals zero, because it is the value of the total preflow in X and every vertex in X has a nonnegative excess while the only vertex with negative excess (i.e., s) that could absorb the excess in the flow does not belong to X .

Hence, $\sum_{w \in X} e(w) \leq 0$, and since f is a preflow, $e(w) = 0, \forall w \in X$ implying that $e(v) = 0$, a contradiction. ■

Note that distance labels remain valid. This follows by Lemma 5.3.1, because the preflow-push algorithm pushes flow along admissible edges and relabels a vertex when there is no admissible edge incident on it. Further, distance labels do not increase much as the next lemma shows.

Lemma 5.4.2 (i) For every $v \in V$, $d(v) < 2n$. (ii) Each distance label increases at most $2n$ times, and consequently the total number of relabel operations is at most $2n^2$.

Proof. (i) The claim is obviously true for s and t . Consider the last time that the algorithm relabeled v . At that time $e(v) > 0$ and there was a v - s path P in the residual network (by Lemma 5.4.1). Note that P has at most $n - 2$ edges. Since $d(s) = n$ and $d(u) \leq d(w) + 1$, for every edge $(u, w) \in P$, we have that $d(v) \leq d(s) + (n - 2) < 2n$.

(ii) Since each time v is relabeled, $d(v)$ increases by at least 1, (i) above implies that each $d(v)$ increases at most $2n$ times. Since there are n vertices, the total number of relabel operations is at most $2n^2$. ■

Lemma 5.4.3 The generic preflow-push algorithm: (a) takes $O(nm)$ time to identify admissible edges and perform relabel operations; (b) performs at most nm saturating pushes.

Proof. (a) Immediate from Lemmas 5.3.2 and 5.4.2(ii). (b) Immediate from Lemmas 5.3.3 and 5.4.2(i). ■

Let us now count the number of non-saturating pushes.

Lemma 5.4.4 *The generic preflow-push algorithm performs $O(n^2m)$ non-saturating pushes.*

Proof. The proof is based on a potential function argument. Let A be the set of active vertices. Consider a potential function Φ defined as $\Phi = \sum_{v \in A} d(v)$. Initially, $\Phi = 0$ and at the end of the algorithm Φ is also zero.

Since $|A| < n$ and $d(v) < 2n$, $\forall v \in A$ (by Lemma 5.4.2(i)), we have that $\Phi < 2n^2$ after the preprocessing phase.

Note that a non-saturating push on edge (v, w) causes Φ to decrease by at least 1. This is because Φ decreases by $d(v)$ (since v becomes inactive), but it may simultaneously increase by $d(w) = d(v) - 1$ (since (v, w) is admissible and $w \notin A$). Hence, to bound the number of non-saturating pushes it suffices to count the total decrease in Φ . Since initially and after the termination of the algorithm $\Phi = 0$, the total decrease in Φ equals the total increase in Φ .

The total increase in Φ due to relabel operations is (by Lemma 5.4.2) at most $2n^2$.

A saturating push on edge (v, w) may create a new excess at w , and consequently increases Φ by $d(w)$. Since $d(w)$ increases at most $2n$ times (Lemma 5.4.2(ii)), and the number of saturating pushes is at most nm , the total increase in Φ due to saturating pushes is $2n^2m$.

Hence, the total increase in Φ is $2n^2 + 2n^2 + 2n^2m = O(n^2m)$, which ends the proof of the lemma. ■

To implement the algorithm, it is sufficient to maintain a set L of active vertices. The algorithm: adds to L those vertices that become active after a push and are not already in L ; selects an active vertex from L to perform a push-relabel operation; and deletes from L those vertices that become inactive after a non-saturating push. The set L can be easily implemented as a doubly linked list, so add, select and delete operations can be done in $O(1)$ time.

The above discussion and Lemmas 5.4.2, 5.4.3 and 5.4.4, establish the following.

Theorem 5.4.1 *The generic preflow-push algorithm finds a maximum flow in $O(n^2m)$ time.*

Although the running time of the generic preflow-push algorithm is comparable with that of the shortest augmenting path algorithm, it is possible to improve this bound by applying specific rules for selecting/examining active vertices. These rules improve substantially the number of non-saturating pushes which appears to be the bottleneck in the running time. Such improved implementations are:

- (1) *FIFO preflow-push algorithm*: the active vertices are examined in FIFO order. This is done by implementing L as a queue. When an active vertex is selected, the algorithm keeps pushing flow out of it until either $e(v) = 0$ or v is relabeled. With this implementation, the running time is improved to $O(n^3)$.
- (2) *Highest-label preflow-push algorithm*: this algorithm selects the active vertex v with the highest $d(v)$ to push flow. The set L is now maintained as a set of $2n-1$ lists $L(k)$, $1 \leq k \leq 2n-1$, where $L(k)$ contains those active vertices v with $d(v) = k$. Let ℓ be an upper bound on the highest value of k for which $L(k)$ is nonempty. To determine the vertex with the highest distance label, the lists $L(\ell), L(\ell-1), \dots$, are examined in order until a nonempty list $L(r)$ is found. Then, the algorithm sets $\ell = r$ and selects any vertex from $L(r)$ to push flow. If the distance level of a vertex is increased while the algorithm is examining it, then ℓ is set to the new distance label. It is not hard to see that the total time of selecting vertices from all lists is $O(n^2)$. This implementation (along with a careful analysis on the number of non-saturating pushes) results in a running time of $O(n^2\sqrt{m})$.
- (3) *Excess scaling preflow-push algorithm*: this algorithm pushes flow from a vertex with “sufficiently large” excess to a vertex with “sufficiently small” excess. This algorithm works for networks with integral capacities and runs in $O(nm + n^2 \log C)$ time, where C is the largest edge capacity.

Chapter 6

Minimum Cost Flows

6.1 Introduction

The minimum cost flow problem is a natural generalization of the max-flow problem; moreover, the problem is very useful in practice. We are given a network $G = (V, E)$, where every edge $(u, v) \in E$ has an associated nonnegative capacity $c(u, v)$ and an associated cost $wt(u, v)$. Further, every vertex $v \in V$ is associated with a number $b(v)$ indicating its *supply* (if $b(v) > 0$) or its *demand* (if $b(v) < 0$). The *minimum cost flow problem* (min-cost flow for short) is to find a nonnegative real-valued function f on the edges of G such that:

$$z(f) = \sum_{(u,v) \in E} wt(u,v)f(u,v) \quad \text{is minimized}$$

subject to

$$\sum_{(u,v) \in E} f(u,v) - \sum_{(v,u) \in E} f(v,u) + b(v) = 0 \quad \forall v \in V$$

$$0 \leq f(u,v) \leq c(u,v) \quad \forall (u,v) \in E.$$

The function $z(f)$ is called the objective function. We shall denote by C the largest value of any supply/demand or finite edge capacity, and by W the largest cost value. For convenience, we also make the following (non restrictive) assumptions:

A1: All costs, supplies/demands and capacities are integral. Further, all edge costs are nonnegative.

A2: The supplies/demands at the vertices satisfy the condition $\sum_{v \in V} b(v) = 0$ and the min-cost flow problem has a feasible solution.

The feasibility of the min-cost flow problem in a network G can be determined by solving a max-flow problem: add two new vertices s' and t' in G (that will play the roles of source and sink, respectively). Add new edges as follows: for each $v \in V$ with $b(v) > 0$, add an edge (s', v) with capacity $b(v)$; and for each $v \in V$ with $b(v) < 0$, add an edge (v, t') with capacity $-b(v)$. Now, solve an s' - t' max-flow problem in this transformed network. If the max-flow saturates all edges (s', v) and (v, t') , then the min-cost flow is feasible in G ; otherwise, it is infeasible.

EXERCISE 6.1. Prove the correctness of the above method.

The residual network $G(f)$ of G w.r.t. the flow f is defined in the same way as in Chapter 4 with the addition that if for an edge (u, v) , with cost $wt(u, v)$, its reversal (v, u) belongs to $G(f)$ (i.e. has positive residual capacity), then $wt(v, u) = -wt(u, v)$.

6.2 Optimality conditions

In Chapter 3, we discussed which particular properties the distance labels must satisfy in order to determine correct shortest path distances (Lemma 3.3.1), i.e., to be optimal. These so-called optimality conditions are very important in the design of algorithms as we saw in Chapter 3. Before discussing algorithms for solving the min-cost flow problem, we shall first give three different (however equivalent) optimality conditions for this problem.

Theorem 6.2.1 (Negative Cycle Optimality Conditions) *A feasible solution f^* is an optimal solution of the min-cost flow problem iff the residual network $G(f^*)$ does not contain a (directed) cycle of negative cost.*

Proof. (\Rightarrow) Assume that f is a feasible flow and $G(f)$ has a negative cycle. Then f is not optimal, because we can improve (i.e. reduce) the value of the objective function $z(f)$ by pushing positive flow along the cycle. Hence, if f^* is an optimal flow, then $G(f^*)$ cannot contain a negative cycle.

(\Leftarrow) Assume that f^* is a feasible flow and that $G(f^*)$ does not contain a negative cycle. Let f_o be an optimal flow, $f_o \neq f^*$. Consider the flow $f_o - f^*$. This flow can be decomposed into at most m cycle flows (in a way analogous to that described in the proof of Theorem 4.3.5; only now we start from a demand vertex). Since the costs of all cycles in $G(f^*)$ are nonnegative, $z(f_o) - z(f^*) \geq 0$, or $z(f_o) \geq z(f^*)$. Further, since f_o is an optimal flow, $z(f_o) \leq z(f^*)$. Hence, $z(f_o) = z(f^*)$ and since the cost function $wt(\cdot)$ is

the same, we must have $f_o = f^*$. Consequently, if $G(f^*)$ does not contain a negative cycle, then f^* is optimal. ■

Before proceeding to the other optimality conditions, we need to recall the concept of reduced weights (or costs) introduced in Chapter 3 for the solution of the all pairs shortest paths problem. Recall that the reduced cost $\rho(u, v)$ of an edge (u, v) , w.r.t. distance labels $d(\cdot)$ from a source vertex s , is defined as $\rho(u, v) = wt(u, v) + d(u) - d(v)$. Also recall the very useful properties of Lemma 3.8.1.

For the min-cost flow problem, we associate a real number $\pi(v)$, called the *potential* of vertex v , with each vertex $v \in V$. (The vertex potentials will play the role of distance labels.) For a given set of vertex potentials $\pi(\cdot)$, we define the *reduced cost* of an edge (u, v) as $\rho_\pi(u, v) = wt(u, v) - \pi(u) + \pi(v)$. The reduced costs apply to both the original and the residual network. The following property is a restatement of Lemma 3.8.1 in terms of $\rho_\pi(\cdot)$, and therefore can be easily verified.

Proposition 6.2.1

- (i) For any x - y path P , $\sum_{(u,v) \in P} \rho_\pi(u, v) = \sum_{(u,v) \in P} wt(u, v) - \pi(x) + \pi(y)$.
- (ii) For any directed cycle Q , $\sum_{(u,v) \in Q} \rho_\pi(u, v) = \sum_{(u,v) \in P} wt(u, v)$.

We are now ready for the second optimality condition.

Theorem 6.2.2 (Reduced Cost Optimality Conditions) *A feasible flow f^* is an optimal solution of the min-cost flow problem iff a set of vertex potentials $\pi(\cdot)$ satisfy $\rho_\pi(u, v) \geq 0$, for every edge (u, v) in $G(f^*)$.*

Proof. It suffices to prove that the negative cycle optimality conditions are equivalent to the reduced cost optimality conditions.

Assume that f^* satisfies the reduced cost optimality conditions. Then, for every directed cycle Q in $G(f^*)$, we have that $\sum_{(u,v) \in Q} \rho_\pi(u, v) \geq 0$. By Proposition 6.2.1(ii), $\sum_{(u,v) \in Q} \rho_\pi(u, v) = \sum_{(u,v) \in Q} wt(u, v) \geq 0$, and therefore $G(f^*)$ does not contain a negative cycle.

Conversely, assume that f^* satisfies the negative cycle optimality conditions, i.e., $G(f^*)$ does not contain a negative cycle. Let $d(\cdot)$ be the shortest path distances from a vertex s to all other vertices in $G(f^*)$. Then, from Lemmas 3.3.1 and 3.3.2, $d(\cdot)$ satisfy the conditions $d(v) \leq d(u) + wt(u, v)$ for every (u, v) in $G(f^*)$. These inequalities can be rewritten as $wt(u, v) - (-d(u)) + (-d(v)) \geq 0$, or $\rho_\pi(u, v) \geq 0$ by setting $\pi(\cdot) = -d(\cdot)$. Hence, f^* satisfies the reduced cost optimality conditions. ■

We will refer to a set of vertex potentials $\pi(\cdot)$ that satisfy the conditions $\rho_\pi(u, v) \geq 0$, for every edge (u, v) in $G(f)$ where f is a feasible flow, as the *optimal vertex potentials*.

Both previous optimality conditions refer to the residual network. We will now restate these conditions in terms of the original network.

Theorem 6.2.3 (Complementary Slackness Optimality Conditions)

A feasible flow f^ is an optimal solution of the min-cost flow problem iff for a set of vertex potentials $\pi(\cdot)$, the reduced costs and flow values satisfy the following conditions for every edge $(u, v) \in E$:*

- (a) *If $\rho_\pi(u, v) > 0$, then $f^*(u, v) = 0$.*
- (b) *If $0 < f^*(u, v) < c(u, v)$, then $\rho_\pi(u, v) = 0$.*
- (c) *If $\rho_\pi(u, v) < 0$, then $f^*(u, v) = c(u, v)$.*

Proof. As before, it suffices to prove that reduced cost optimality conditions are equivalent to the complementary slackness conditions.

Assume that the vertex potentials $\pi(\cdot)$ and the flow f^* satisfy the reduced cost optimality conditions. Consider the following cases for any edge $(u, v) \in E$:

(a) If $\rho_\pi(u, v) > 0$, then $G(f^*)$ cannot contain the edge (v, u) because $\rho_\pi(v, u) = -\rho_\pi(u, v) < 0$, contradicting the reduced cost optimality condition for that edge. Hence, $f^*(u, v) = 0$.

(b) If $0 < f^*(u, v) < c(u, v)$, then $G(f^*)$ contains both edges (u, v) and (v, u) . From the reduced cost optimality conditions we have that $\rho_\pi(u, v) \geq 0$ and $\rho_\pi(v, u) \geq 0$. Since $\rho_\pi(v, u) = -\rho_\pi(u, v)$, we must have that $\rho_\pi(u, v) = \rho_\pi(v, u) = 0$.

(c) If $\rho_\pi(u, v) < 0$, then $G(f^*)$ cannot contain the edge (u, v) because such an edge would contradict the reduced cost optimality condition. Hence, $f^*(u, v) = c(u, v)$.

For the converse, see Exercise 6.2. ■

EXERCISE 6.2. Show that if a flow f^* and a set of vertex potentials $\pi(\cdot)$ satisfy the complementary slackness conditions, then they also satisfy the reduced cost optimality conditions.

6.3 Optimal Flows and Optimal Vertex Potentials

The above discussion raises two questions: given an optimal flow, can we determine the optimal vertex potentials? And conversely, given optimal vertex potentials, can we obtain an optimal flow? We see now how the

former question is answered by solving a shortest path problem and the latter question by solving a max-flow problem. These results establish one more interesting connection between the min-cost flow problem and the shortest paths and max-flow problems.

Determining optimal vertex potentials

Let f^* be an optimal flow and let $G(f^*)$ be the residual network w.r.t. this flow. By Theorem 6.2.1, $G(f^*)$ does not contain a negative cycle. Let $d(\cdot)$ denote the shortest path distances from a vertex s to all other vertices in $G(f^*)$ with edge costs (weights) $wt(u, v)$. Since there is no negative cycle in $G(f^*)$, Lemmas 3.3.1 and 3.3.2 imply that $d(v) \leq d(u) + wt(u, v)$, for all (u, v) in $G(f^*)$. Compute these distances (by solving a shortest path tree problem) and set $\pi(\cdot) = -d(\cdot)$. Then, the previous inequality can be rewritten as $\rho_\pi(u, v) = wt(u, v) - \pi(u) + \pi(v) \geq 0$, for all (u, v) in $G(f^*)$. By Theorem 6.2.2, $\pi(\cdot)$ is a set of optimal vertex potentials.

Computing an optimal flow

Let $\pi(\cdot)$ be a given set of optimal vertex potentials. First, compute the reduced costs $\rho_\pi(u, v)$ for every edge $(u, v) \in E$. Then, examine all edges (u, v) one by one and make appropriate transformations on the original network G as they are determined by the following cases:

Case 1: $\rho_\pi(u, v) > 0$. Condition (a) of Theorem 6.2.3 implies that $f^*(u, v) = 0$. Enforce this constraint by setting $f^*(u, v) = 0$ and delete (u, v) from G .

Case 2: $\rho_\pi(u, v) < 0$. Condition (c) of Theorem 6.2.3 implies that $f^*(u, v) = c(u, v)$. Enforce this constraint by setting $f^*(u, v) = c(u, v)$ and delete (u, v) from G . Further, decrease $b(v)$ by $c(u, v)$ and increase $b(u)$ by $c(u, v)$, since $c(u, v)$ units of flow was sent on edge (u, v) .

Case 3: $\rho_\pi(u, v) = 0$. Then, allow the flow on edge (u, v) to take any value between 0 and $c(u, v)$.

Let $G' = (V, E')$ be the resulting network and b' be the new supplies/demands after the above transformations. Now, the problem reduces to finding a feasible flow in G' that meets the new supplies/demands. As we discussed in the Introduction (see also Exercise 6.1), finding a feasible min-cost flow in G' is equivalent to solving a max-flow problem (recall the necessary transformations that should be done to G').

6.4 Algorithms for Finding a Min-Cost Flow

The negative cycle optimality conditions suggest a simple algorithm for solving the min-cost flow problem, called the *cycle-canceling algorithm*. The algorithm maintains a feasible flow f and at every iteration tries to improve the value of its objective function $z(f)$ by sending flow along a negative cycle. The algorithm terminates when the residual network does not contain a negative cycle. In order to establish an initial feasible flow, the algorithm solves a max-flow problem as described before. A less informal description of the algorithm follows.

Algorithm Cycle-canceling.

Input: Network $G = (V, E)$ with nonnegative integral edge capacities and costs, and integral supplies/demands for each vertex.

Output: An min-cost flow f in G .

Method:

```

    find a feasible flow  $f$  in  $G$ ;
    while  $G(f)$  has a negative cycle do
        identify a negative cycle  $Q$ ;
         $\Delta f = \min\{r(v, w) : (v, w) \in Q\}$ ;
        augment  $\Delta f$  units of flow along  $Q$ ;
         $|f| = |f| + \Delta f$ ;
        update  $G(f)$ ;
    od
  
```

The correctness of the above algorithm follows immediately by Theorem 6.2.1. Before establishing its running time, we give an important (by-product) result.

Lemma 6.4.1 (Integrality Property) *If all edge capacities and the supplies/demands of vertices are integral, then the min-cost flow problem always has an integer flow of minimum cost.*

Proof. Induction on the number of iterations. The algorithm first finds a feasible flow by solving a max-flow problem in a network with integral edge capacities. By Theorem 4.3.3, this max-flow is integral and consequently the initial feasible flow is also integral. In every iteration the flow is augmented by an amount equal to the minimum residual capacity of the negative cycle canceled, which by the inductive hypothesis is integer. Thus, the modified residual capacities in the next iteration will again be integer. Consequently,

when the algorithm terminates, the flow is integral and by Theorem 6.2.1 is a flow of minimum cost. ■

Let us now determine the complexity of the cycle-canceling algorithm. Observe that for the min-cost flow problem, mCW is an upper bound on the initial flow cost and $-mCW$ is a lower bound on the optimal flow cost. In each iteration of the algorithm, the objective function is improved by $\Delta f(\sum_{(u,v) \in Q} wt(u,v))$, which is strictly negative. Since all data of the problem are integral, the algorithm terminates within $O(mCW)$ iterations. Each iteration is dominated by the time for finding a negative cycle, which by Theorem 3.6.1 takes $O(nm)$ time. Thus, we have established the following.

Theorem 6.4.1 *The cycle-canceling algorithm solves the min-cost flow problem in $O(nm^2CW)$ time.*

We now describe a different algorithm called the *successive shortest path* algorithm. This algorithm works in a manner opposite to that of cycle-canceling algorithm. The latter algorithm maintains at each step feasibility of the solution and attempts to achieve optimality. The successive shortest path algorithm maintains at each step optimality of the solution (as determined by Theorem 6.2.2) and attempts to achieve feasibility. It maintains a solution f which satisfies the nonnegativity and capacity constraints (i.e. it is nonnegative and its value on an edge does not exceed the capacity of the edge), but violates the balance constraints on the vertices. At every step, a vertex s is selected with excess supply (not yet sent to some demand vertex) and a vertex t with unsatisfied demand; the algorithm sends then flow from s to t along a shortest s - t path in the residual network. The algorithm terminates when the balance constraints on the vertices are satisfied.

Before giving the algorithm, we need some preliminaries. A *pseudoflow* is a function $f : E \rightarrow \mathbb{R}^+ \cup \{0\}$ satisfying only the capacity and nonnegativity constraints, but not necessarily the balance constraints on the vertices. For the pseudoflow f , the *imbalance* at vertex v is defined as

$$e(v) = b(v) + \sum_{(v,u) \in E} f(v,u) - \sum_{(u,v) \in E} f(u,v) \quad \forall v \in V.$$

If for some vertex $v \in V$ $e(v) > 0$ (resp. $e(v) < 0$), then $e(v)$ (resp. $-e(v)$) is called the *excess* (resp. *deficit*) of vertex v . If $e(v) = 0$, then v is called *balanced*. Let X and D be the sets of excess and deficit vertices in the network. Note that $\sum_{v \in V} e(v) = \sum_{v \in V} b(v)$, and therefore $\sum_{v \in X} e(v) = \sum_{v \in D} e(v)$. Consequently, if the network contains an excess vertex, it must also contain a deficit one. The residual network w.r.t. a pseudoflow is defined in the same way as that for a flow.

Lemma 6.4.2 *Suppose that a pseudoflow or flow f satisfies the reduced cost optimality conditions w.r.t. some vertex potentials $\pi(\cdot)$. Let $d(\cdot)$ denote the shortest path distances from a vertex s to all other vertices in $G(f)$ with $\rho_\pi(u, v)$ as the cost of edge (u, v) . Then, the following hold:*

(i) *f also satisfies the reduced cost optimality conditions w.r.t. the vertex potentials $\pi'(\cdot) = \pi(\cdot) - d(\cdot)$.*

(ii) *$\rho_{\pi'}(u, v) = 0$ for all edges (u, v) in a shortest path from s to every other vertex.*

Proof. (i) Since f satisfies the reduced cost optimality conditions w.r.t. $\pi(\cdot)$, $\rho_\pi(u, v) \geq 0$ for every (u, v) in $G(f)$. Further, since $d(\cdot)$ represent shortest path distances, with $\rho_\pi(u, v)$ as edge costs, they satisfy the shortest path optimality conditions (Lemma 3.3.1): $d(v) \leq d(u) + \rho_\pi(u, v)$, for every (u, v) in $G(f)$.

Substituting $\rho_\pi(u, v) = wt(u, v) - \pi(u) + \pi(v)$ in the above inequality, we get $d(v) \leq d(u) + wt(u, v) - \pi(u) + \pi(v)$. Rearranging the terms we get, $wt(u, v) - (\pi(u) - d(u)) + (\pi(v) - d(v)) \geq 0$, or $\rho_{\pi'}(u, v) \geq 0$. Now, the rest of the proof follows by Theorem 6.2.2.

(ii) Consider a shortest path from s to any other vertex. All edges (u, v) in this path satisfy $d(v) = d(u) + \rho_\pi(u, v)$. As before, substituting $\rho_\pi(u, v) = wt(u, v) - \pi(u) + \pi(v)$ in this equation, we get $\rho_{\pi'}(u, v) = 0$. ■

Lemma 6.4.3 *Suppose that a pseudoflow or flow f satisfies the reduced cost optimality conditions and f' is obtained from f by sending flow along a shortest path from s to another vertex y . Then, f' also satisfies the reduced cost optimality conditions.*

Proof. Define $\pi(\cdot)$ and $\pi'(\cdot)$ as in Lemma 6.4.2. It suffices to examine only the edges in the shortest s - y path. By the second property of that lemma, $\rho_{\pi'}(u, v) = 0$ for every edge (u, v) in a shortest s - y path, and hence the reduced cost optimality conditions are satisfied for these edges. Augmenting flow along such an edge (u, v) may add its reversal (v, u) to the residual network. But since $\rho_{\pi'}(u, v) = 0$ for each edge (u, v) in the shortest s - y path, $\rho_{\pi'}(v, u) = 0$ and consequently (v, u) also satisfies the reduced cost optimality conditions. ■

We are now ready to present the algorithm.

Algorithm Successive shortest path.

Input: Network $G = (V, E)$ with nonnegative integral edge capacities and costs, and integral supplies/demands for each vertex.

Output: A min-cost flow f in G .

Method:

```

 $f = 0$ ;
for all  $v \in V$  do
     $\pi(v) = 0$ ;  $e(v) = b(v)$ ;
od
 $X = \{v : e(v) > 0\}$ ;  $D = \{v : e(v) < 0\}$ ;
while  $X \neq \emptyset$  do
    select a vertex  $s \in X$  and a vertex  $t \in D$ ;
    find shortest path distances  $d(\cdot)$  to all other vertices
        in  $G(f)$  w.r.t.  $\rho_\pi(u, v)$ ;
    let  $P$  be the shortest  $s$ - $t$  path;
    for all  $v \in V$  do  $\pi(v) = \pi(v) - d(v)$ ;
     $\Delta f = \min\{e(s), -e(t), \min\{r(v, w) : (v, w) \in P\}\}$ ;
    augment  $\Delta f$  units of flow along  $P$ ;
     $|f| = |f| + \Delta f$ ;
    update  $G(f)$ ,  $X$ ,  $D$  and the reduced costs;
od

```

Theorem 6.4.2 *The successive shortest path algorithm solves the min-cost flow problem in $O(nC \cdot T_S(n, m, nW))$ time, where $T_S(n, m, A)$ is the time to compute a shortest path tree in an n -vertex, m -edge network with non-negative integral costs whose largest value is A .*

Proof. Let us first discuss the correctness. Initially $f = 0$ and this is a feasible pseudoflow. Also note that in this case $G(f) = G$ and f along with $\pi(\cdot) = 0$ satisfies the reduced cost optimality conditions because $\rho_\pi(u, v) = wt(u, v) \geq 0$ (recall assumption A1). As long as any vertex has a nonzero imbalance, both sets X and D must be nonempty since the total sum of excesses equals the total sum of deficits. Consequently, until all vertices become balanced the algorithm always finds an excess vertex s and a deficit vertex t . We can assume w.l.o.g. that an s - t path always exist in the residual network by adding appropriate edges of very large cost and very large capacity. Therefore, the distances $d(\cdot)$ are well defined. Hence, in view of Lemmas 6.4.2 and 6.4.3, the algorithm correctly maintains a pseudoflow which when all vertices become balanced is indeed the required flow of minimum cost.

Concerning the complexity of the algorithm, observe that every iteration strictly decreases the excess of some vertex (and also the deficit of some other vertex). Thus, if C is the largest value of all supplies/demands, then the

algorithm terminates in $O(nC)$ iterations. In every iteration, the dominating step is that of finding a shortest s - t path in a network with nonnegative integer costs. Since the costs in the residual network are bounded by nW , the claimed time bound follows. ■

Bibliography

- [1] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows* Prentice-Hall, 1993.
- [2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2nd Edition, 2001.
- [3] K. Mehlhorn, *Data Structures and Algorithms Vol 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, 1984.
- [4] K. Mehlhorn and S. Naeher, *LEDA: A platform for combinatorial and geometric computing*, Cambridge University Press, 1999.
- [5] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.