**ANALYSIS AND IMPLEMENTATION OF NETWORK FLOW ALGORITHMS**

**Submitted By**

Vaibhav Devekar (081070015)

Aniket Alshi (081070103)

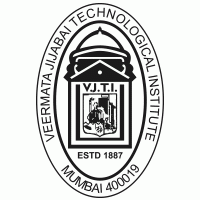
Aniket Divekar (081070117)

Bachelor of Technology, Computer Engineering

2011-2012

**Under The Guidance Of**

Mrs.S. C. Shrawne



**Department of Computer Technology**

**Veermata Jijabai Technological Institute**

(Autonomous Institute Affiliated to Mumbai University)

Mumbai-400019

**Statement by the Candidates**

We wish to state that work embodied in this report titled **“Analytical Study And Implementation Of Network Flow Algorithms”** forms our group’s contribution to the work carried out under the Guidance of Mrs. **S. C. Shrawne at** **Veermata Jijabai Technological Institute.** This work has not been submitted for any other Degree or Diploma of any University/Institute. Whenever references have been to previous works of others, it has been clearly indicated.

Aniket Alshi Vaibhav Devekar Aniket Divekar

# CERTIFICATE

This is to certify that **Vaibhav Devekar, Aniket Divekar and Aniket Alshi,** students of B.Tech(Computer Technology), as a team of students have completed the Project Report, **“Analytical Study and Implementation of Network Flow Algorithms”** to our satisfaction.

**Mrs. S. C. Shrawne Dr. B. B. Meshram**

Project Guide, Head of Department,

Computer Department, Computer Department,

VJTI VJTI

**Examiner**

ABSTRACT

In Network Flows, the maximum flow problem is the following optimization problem: in a capacitated network, we wish to send as much flow as possible between two special nodes, a source node *s* and a sink node *t,* without exceeding the capacity of any arc. In this thesis, we reviewed specific families of network flows algorithms – the augmenting path, the push-relabel and the scaling algorithms and also the applications with respect to maximum flow problem.

As part of the implementation, we have developed and integrated the code based on max-flow min-cut theorem in image segmentation. The code to separate the foreground and background in the image uses the heuristic based on Edmond-Karp method and push-relabel method using FIFO technique. Additionally we compared the performance of both algorithms on the graphs formed from image data.

# ACKNOWLEDGEMENTS

We would like to thank our project guide Mrs. S. C. Shrawne for contributing her time and effort to help us with our project. Her suggestions and guidelines have been of a great help during the course of our project. She has always been involved by discussing our project at each phase to make sure that the experiment is designed and carried out in an appropriate manner and that the our conclusions are appropriate, given their results. Her constant support and interaction have been a driving force which has constantly motivated us to explore the different aspects of our project.

Table of Contents

[Certificate 3](#_Toc323470274)

[Acknowledgements 5](#_Toc323470275)

[List of Tables 8](#_Toc323470276)

[List of Figures 9](#_Toc323470277)

1 [Introduction 10](#_Toc323470279)

2 [Preliminaries 11](#_Toc323470281)

[2.1 Flow networks and flows 11](#_Toc323470282)

[2.2 The Maximum Flow Problem 13](#_Toc323470283)

[2.3 Flows and Cuts 13](#_Toc323470284)

3 [Algorithms 15](#_Toc323470286)

[3.1 Augmenting Path algorithms 15](#_Toc323470287)

[3.1.1 Shortest Augmenting Path Algorithm 16](#_Toc323470288)

[3.1.2 Ford Fulkerson Method 22](#_Toc323470289)

[3.1.3 The Edmonds-Karp algorithm 30](#_Toc323470290)

[3.2 Preflow Push Algorithms 33](#_Toc323470291)

[3.2.1 Introduction 33](#_Toc323470292)

[3.2.2 The Generic Algorithm 36](#_Toc323470293)

[3.2.3 FIFO Preflow-Push Algorithm 38](#_Toc323470294)

[3.2.4 Highest-Label Preflow-Push Algorithm 40](#_Toc323470295)

[3.2.5 Dynamic Tree Algorithm 41](#_Toc323470296)

[3.3 Scaling Algorithms 48](#_Toc323470297)

[3.3.1 Capacity Scaling Algorithms 48](#_Toc323470298)

[3.3.2 Excess Scaling Algorithm 49](#_Toc323470299)

4 [Applications 54](#_Toc323470302)

[4.1 Feasible Flow Problem 54](#_Toc323470303)

[4.3 Matrix Rounding Problem 56](#_Toc323470304)

[4.4 Distributed Computing on a Two-Processor Computer 57](#_Toc323470305)

5 [Image Segmentation 60](#_Toc323470307)

[5.1 Reduction to Network Flow problem 60](#_Toc323470308)

[5.2 Implementation 63](#_Toc323470309)

[5.3 Conclusion 66](#_Toc323470310)

6 Conclusion………………………………………………………………………………….. 67

[Bibliography 69](#_Toc323470311)

# List of Tables

|  |  |  |
| --- | --- | --- |
| Table No. | Table Name | Pg. No. |
| 4.1 | Matrix rounding problem | 56 |
| 4.2 | Data for the distributed computing model | 58 |
| 5.1 | Experimental Results | 65 |
| 6.1 | Computational Complexity of algorithms | 68 |

# List of Figures

|  |  |  |
| --- | --- | --- |
| Figure No. | Figure Name | Pg. No. |
| 2.1 | Flow Network | 12 |
| 2.2 | Residual Network | 13 |
| 2.3 | Example of an s-t cut | 14 |
| 3.1.1 | The generic augmenting path algorithm | 16 |
| 3.1.2 | The shortest augmenting path algorithm | 18 |
| 3.1.3 | Ford-Fulkerson Method | 25 |
| 3.1.4 | The (A\*,B\*) cut | 28 |
| 3.2.1 | Generic preflow-push algorithm | 35 |
| 3.2.2 | FIFO preflow-push algorithm | 39 |
| 3.2.3 | Finding an augmenting path in the dynamic tree implementations | 43 |
| 3.3.1 | Illustrating residual network | 48 |
| 4.1 | System of direct representatives | 55 |
| 4.2 | Network for the matrix rounding problem | 56 |
| 4.3 | Network for the distributed computing model | 59 |
| 5.1 | Pixel graph and flow graph | 60 |
| 5.2 | S-T cut on graph of pixels | 62 |
| 5.3 | Image Segmentation: Original Image and results | 64 |
| 5.4 | Image Segmentation: Orignal Image | 65 |
| 5.5 | Ouput after applying edmond-karp and push relabel on fig 5.4 | 65 |

## Chapter 1

## Introduction

We encounter many different types of networks in our everyday lives, including electrical, telephone, cable, highway, rail, manufacturing, and, computer networks. Networks consists of special points called *nodes* and links connecting pairs of nodes called *edges*. In all of these networks, we wish to send some commodity, which we generically call *flow*, from one node to another, and do so as efficiently as possible, subject to certain constraints. Network flow theory is the study of designing computationally efficient algorithms to solve such problems .

The problem of finding a maximum flow in a directed graph with edge capacities arises in many settings in operations research and other fields, and efficient algorithms for the problem have received a great deal of attention. The first maximum-flow algorithm, due to Ford and Fulkerson , works by finding augmenting paths. Edmonds and Karp observed that augmenting along shortest paths leads to a polynomial-time algorithm (algorithm 1). To improve the efftciency further, Dinic proposed a method to find all shortest augmenting paths in one phase.

We are interested in algorithms whose running time is small as a function of size of the network and the numbers involved (e.g. capacities, costs, or gains). In this thesis we would like to highlight the main ideas involved in designing highly efficient algorithms for network flow problems. In chapter 2, we mention the preliminaries that are required for comprehending the topics that are mentioned in later chapters. Chapter also talks about the famous maximum flow problem and associated concepts. In Chapter 3, we have presented different algorithms to solve the maximum flow problem and also have given the detailed analysis of every algorithm. Initial section talks of augmenting path algorithm which proceeds by finding an augmenting path in graph and pushing flow across it. Then we discuss the Preflow-Push algorithm and its associated variants. Later section speaks of scaling algorithm for efficient handling the maxflow problem. Chapter 4, consist of the areas where network flow concepts can be readily applied and its algorithms can be used to simplify the solution. In chapter 5, we have given detailed mention of image segmentation carried out by computing minimum cut. We have also given detailed report of our implementation of Ford-Fulkerson and Push-Relabel algorithm for image segmentation and have also weighed their advantages and disadvantages for the same.

## Chapter 2

## Preliminaries

In this chapter, we give a graph-theoretic definition of flow networks, discuss their properties, and define the maximum-flow problem precisely. We also introduce some helpful notation.

## 2.1 Flow networks and flows

A ***flow network*** G = (V, E) is a directed graph in which each edge (u, v) ∈ E has a nonnegative ***capacity*** c(u, v) ≥ 0. If (u,v) E, then for convenience we define c(u, v) = 0, and we disallow self-loops. We distinguish two vertices in a flow network: a ***source*** s and a ***sink*** t. For convenience, we assume that each vertex lies on some path from the source to the sink. That is, for each vertex v ∈ V, the flow network contains a path s → v → t. The graph is therefore connected and, since each vertex other than s has at least one entering edge, |E| = |V | - 1. Figure 26.1 shows an example of a flow network.

We are now ready to define flows more formally. Let G = (V, E) be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function f: V × V → R that satisfies the following two properties:

**Capacity constraint:** For all u, v ∈ V, we require 0 ≤ f(u, v) ≤ c(u, v).

**Flow conservation:** For all u ∈ V - {s, t}, we require

When (u, v) ∉ E, there can be no flow from u to v, and f (u, v) = 0.

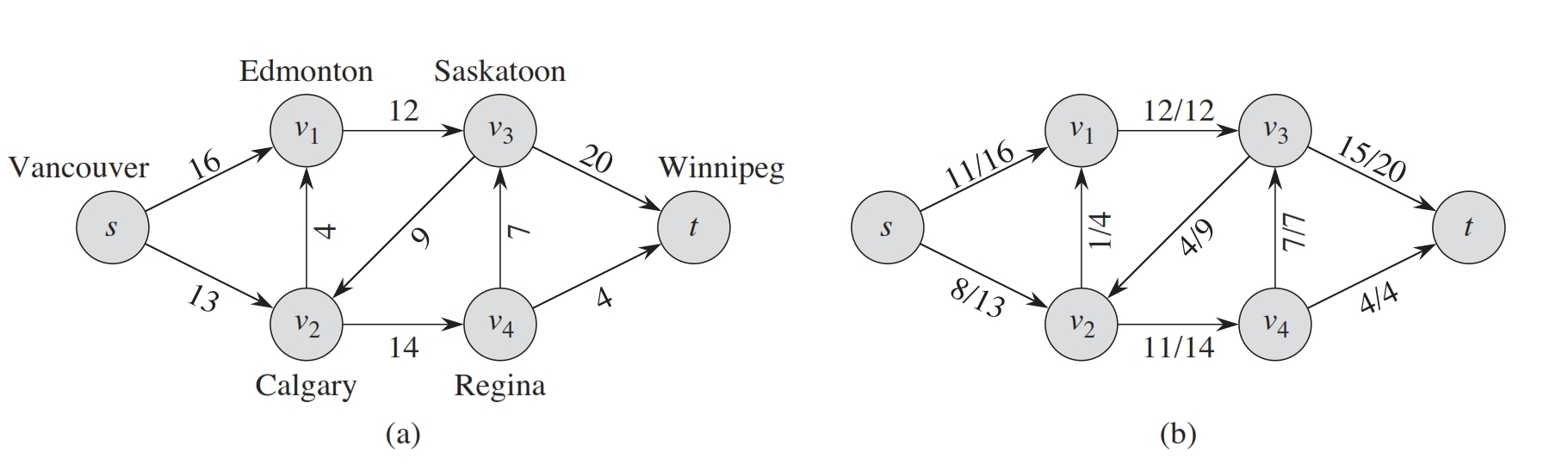


Figure 0.1 (a) A flow network G = (V, E) for the Lucky Puck Company’s trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships pucks through intermediate cities, but only c(u, v)crates per day can go from city u to city v. Each edge is labeled with its capacity. (b) A flow f in G with value |f | = 19. Each edge (u, v) is labeled by f(u, v) / c(u, v). The slash notation merely separates the flow and capacity; it does not indicate division.

We call the nonnegative quantity f(u,v) ≥ 0 the flow from vertex u to vertex v. The value|f | of a flow f is defined as

that is, the total flow out of the source minus the flow into the source. (Here, the |.| notation denotes flow value, not absolute value or cardinality.) Typically, a flow network will not have any edges into the source, and the flow into the source, given by the summation , will be 0. We include it, however, because when we introduce residual networks later in this chapter, the flow into the source will become significant. In the ***maximum-flow problem***, we are given a flow network G with source s and sink t , and we wish to find a flow of maximum value.

## 2.2 The Maximum Flow Problem

To introduce the maximum flow problem, we need the following definitions in addition to the definitions of the previous section. Consider a flow network *(G,u,s,t).* A *preflow* is a pseudo-flow such that the excess function is nonnegative for all vertices other than *s* and *t.* A *flow f* on G is a pseudo-flow satisfying the conservation constraints for all vertices except s and *t.* The *value* |f| of a flow f is the net flow into the sink t. A *maximum flow* is a flow of maximum value (also called *an optimal flow).* The maximum flow problem is that of sending a maximum flow in a given flow network.

Given a flow f, we define an *augmenting path* to be a source-to-sink path in the residual graph. The following theorem, due to Ford and Fulkerson, gives an optimality criterion for maximum flows.

**Theorem 2.1:**  *A flow is optimal if and only if its residual graph contains no augmenting path.*

## 2.3 Flows and Cuts

**Residual network:**

The concept of *residual network* plays a central role in the development of all the maximum flow algorithms we consider. Given a flow *x,* the residual capacity *rij* of any edge *(i,* j) ∈ Eis the maximum additional flow that can be sent from node *i* to node j using the edge *(i,* j*)* and (j, i). The residual capacity *rij* has two components: (1) *Uij* - *Xij,* the unused capacity of edge *(i, j),* and (2) the current flow *Xji* on edge (j, *i),* which we can cancel to increase the flow from node *i* to node *j.* Consequently, *rij* = *Uij* - *Xij +* *Xji.* We refer to the network *G(x)* consisting of the edges with positive residual capacities as the *residual network* (with respect to the flow *x).* Figure 2.2 gives an example of a residual network.

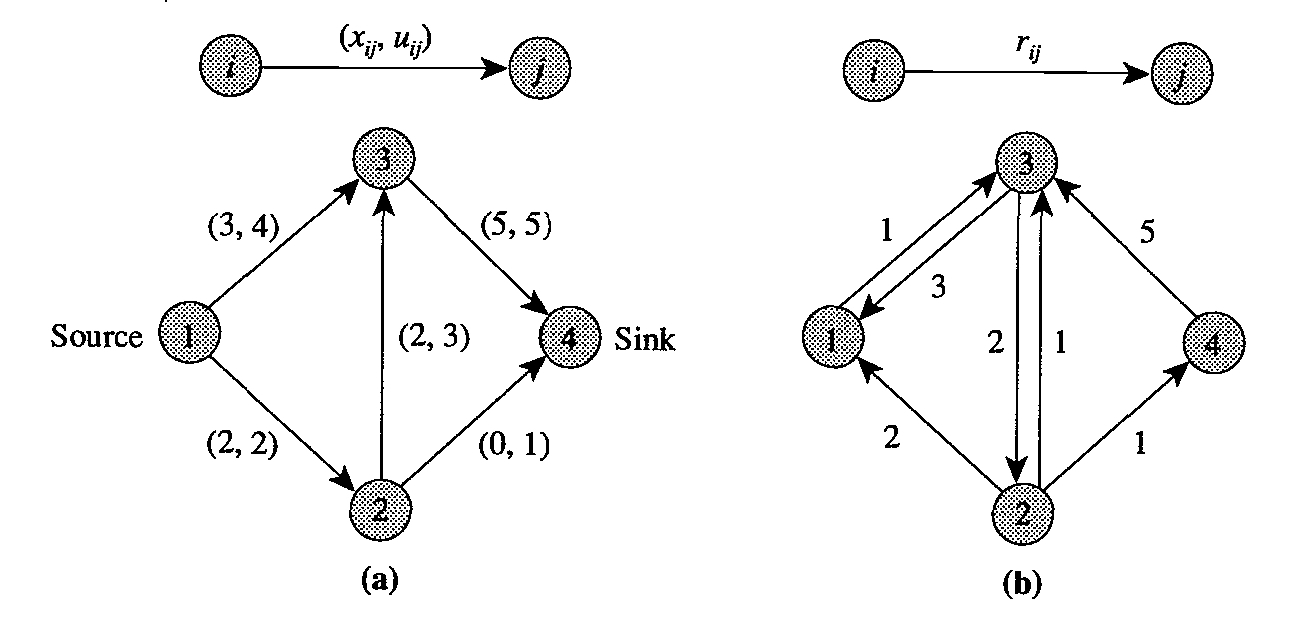


Figure 0.2 Illustrating a residual network: (a) original network G with a flow *x;* (b) residual network *G(x).*

***s-t* cut:**

We now review notation about cuts. A cut is a partition of the node set *N* into two subsets S and S’ = *N* - S; we represent this cut using the notation *[S, S’].* Alternatively, we can define a cut as the set of edges whose endpoints belong to the different subsets Sand S’. We refer to a cut as an *s-t cut* if s ∈ Sand *t* ∈ S’. We also refer to an arc *(i,j)* with i ∈ *S* and j ∈ S’ as a *forward arc* of the cut, and an arc *(i,j)* with i ∈ S’ and j ∈ *S* as a backward *arc* of the cut [S, S’]. Let (S, S’) denote the set of forward arcs in the cut, and let (S’, S) denote the set of backward arcs. For example, in Figure 2.3, the dashed arcs constitute an *s-t* cut. For this cut, (S, S’) = {(1, 2), (3, 4), (5, 6)}, and (S’, *S)* = {(2, 3), (4, 5)}.

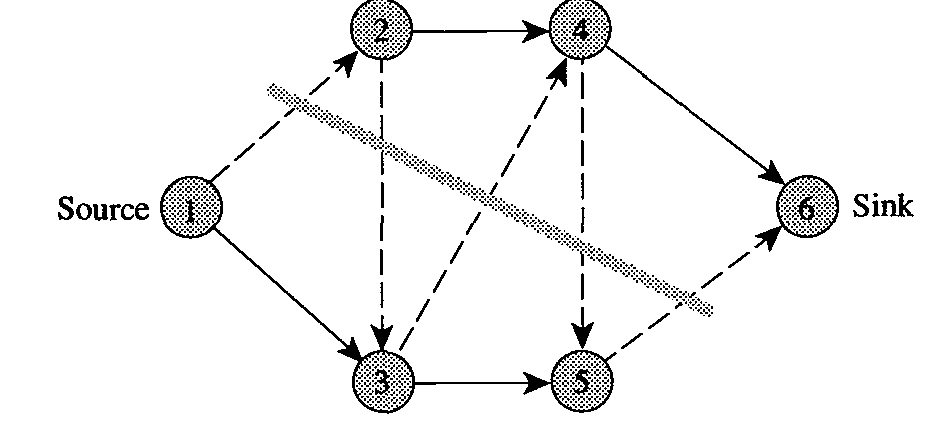


Figure 0.3 Example of an *s-t* cut.

**Capacity of an *s-t* cut:**

We define the capacity c [S, S’] of an *s-t* cut [S, S’]as the sum of the capacities of the forward arcs in the cut. That is,

Clearly, the capacity of a cut is an upper bound on the maximum amount of flow we can send from the nodes in *S* to the nodes in S’ while honoring arc flow bounds.

**Minimum cut:**

We refer to an *s-t* cut whose capacity is minimum among all *s-t* cuts as a *minimum cut.*

**Residual capacity of an *s-t* cut:**

We define the residual capacity *r[S, S’]* of an *s-t* cut *[S, S’]* as the sum of the residual capacities of forward arcs in the cut. That is,

## Chapter 3

## Algorithms

**3.1 Augmenting Path algorithms**

We refer to a directed path from the source to the sink in the residual network as an *augmenting path.* We define the residual capacity of an augmenting path as the minimum residual capacity of any arc in the path. The capacity of an augmenting path is always positive. Consequently, whenever the network contains an augmenting path, we can send additional flow from the source to the sink. The generic augmenting path algorithm is essentially based on this simple observation. The algorithm proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path.

**Algorithm** augmenting path;

**begin**

x: = 0;

**while** G(x) contains a directed path from node s to node t **do**

**begin**

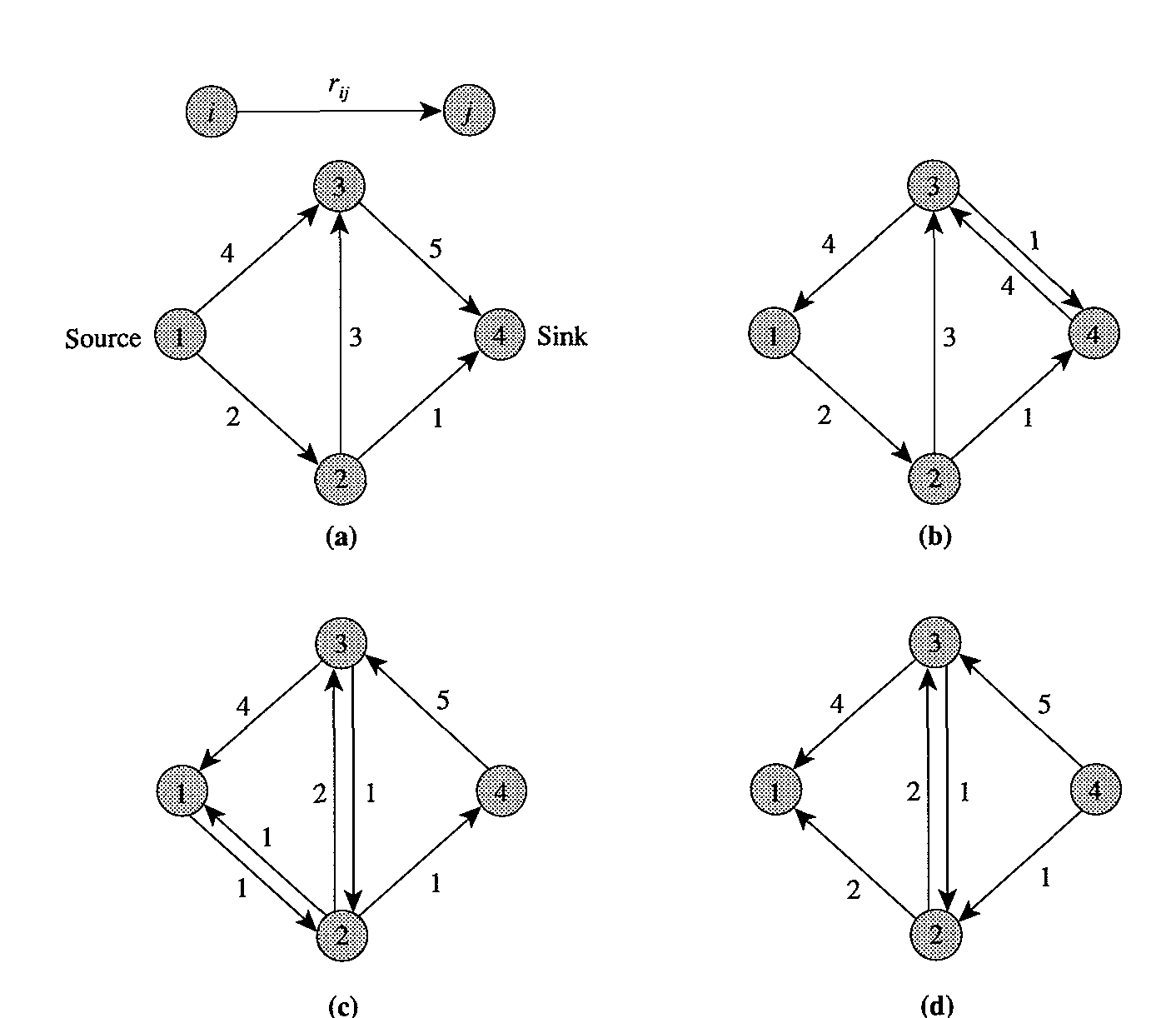
identify an augmenting path P from node s to node t;

: =

augment units of flow along P and update G(x);

**end;**

**end;**



**Figure 3.1.1 Illustrating the generic augmenting path algorithm: (a) residual network for the zero flow; (b) network after augmenting four units along the path 1-3-4; (c) network after augmenting one unit along the path 1-2-3-4; (d) network after augmenting one unit along the path 1-2-4.**

### 3.1.1 Shortest Augmenting Path Algorithm

The shortest augmenting path algorithm always augments flow along a shortest path from the source to the sink in the residual network. A natural approach for implementing this algorithm would be to look for shortest paths by performing a breadth first search in the residual network. If the labeling algorithm maintains the set *L* of labeled nodes as a queue, then by examining the labeled nodes in a first-in, first-out order, it would obtain a shortest path in the residual network. Each of these iterations would require steps in the worst case, and resulting computation time would be. Unfortunately, this computation time is excessive. We can improve it by exploiting the fact that the minimum distance from any node to the sink node is monotonically non-decreasing over all augmentations. By fully exploiting this property, we can reduce the average time per augmentation to*.*

The shortest augmenting path algorithm proceeds by augmenting flows along admissible paths. It constructs an admissible path incrementally by adding one arc at a time. The algorithm maintains a *partial admissible path* (i.e., a path from s to some node consisting solely of admissible arcs) and iteratively performs *advance* or *retreat* operations from the last node of the partial admissible path, which we refer to as the *current node.* If the current node has an admissible arc*,* we perform an advance operation and add arc to the partial admissible path; otherwise, we perform a retreat operation and backtrack one arc. We repeat these operations until the partial admissible path reaches the sink node at which time we perform an augmentation. We repeat this process until the flow is maximum.

**Algorithm** shortest augmenting path;

**begin**

;

*obtain the exact distance labels ;*

;

**while**  do

**begin**

**if** has an admissible arc then

**begin**

advance(i);

**if** then augment and set

**end**

**else** retreat(i)

**end;**

**end;**

**procedure** advance(i);

**begin**

let be an admissible arc in

and

**end;**

**procedure** retreat(i);

**begin**

;

**if** ≠ s **then**

**end;**

**procedure** *augment;*

**begin**

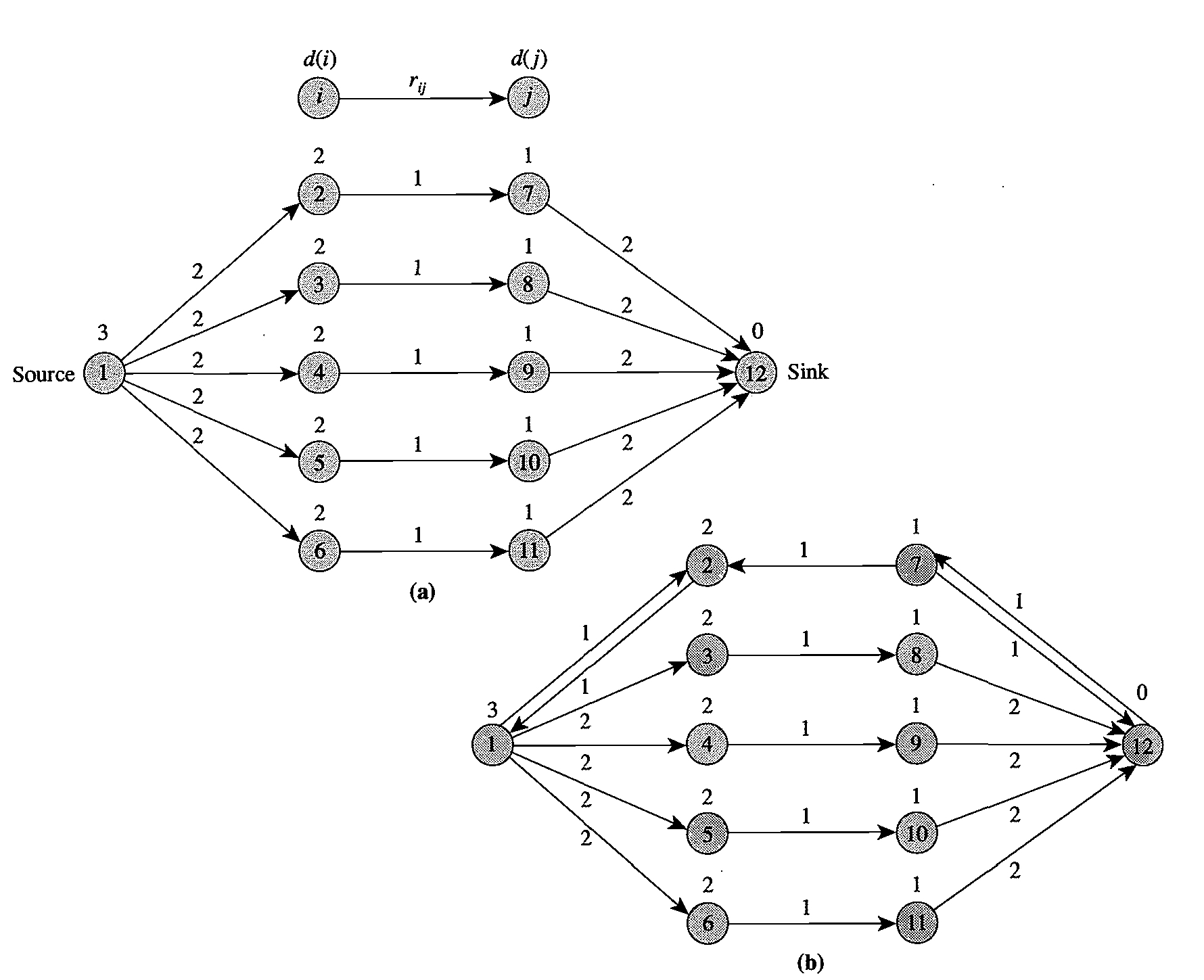
using the predecessor indices identify an augmenting

path P from the source to the sink;

*};*

augment units of flow along path P;

**end;**



**Figure 3.1.2 Illustrating the shortest augmenting path algorithm.**

**Correctness of the Algorithm**

In our analysis of the shortest augmenting path algorithm we first show that it correctly solves the maximum flow problem.

***Theorem 3.1.1:*** *The shortest augmenting path algorithm correctly computes a maximum flow.*

***Lemma 3.1.1:*** *The shortest augmenting path algorithm maintains valid distance labels at each step. Moreover, each re-label operation strictly increases the distance label of a node.*

***Proof:*** We show that the algorithm maintains valid distance labels at every step by performing induction on the number of augment and re-label operations. (The advance operation does not affect the admissibility of any arc because it does not change any residual capacity or distance label.) Initially, the algorithm constructs valid distance labels. Assume, inductively, that the distance labels are valid prior to an operation (i.e., they satisfy the validity conditions). We need to check whether these conditions remain valid (a) after an augment operation, and (b) after a re-label operation.

(a) Although a flow augmentation on arc might remove this arc from the residual network, this modification to the residual network does not affect the validity of the distance labels for this arc. An augmentation on arc might, however, create an additional arc with and therefore also create an additional inequality that the distance labels must satisfy. The distance labels satisfy this validity condition, though, since by the admissibility property of the augmenting path.

(b) The re-label operation modifies therefore, we must show that each incoming and outgoing arc at node satisfies the validity conditions with respect to the new distance labels, say The algorithm performs a re-label operation at node when it has no admissible arc; that is, no arc satisfies the conditions and . This observation, in light of the validity condition, implies that for all arcs with a positive residual capacity. Therefore,which is the new distance label after the re-label operation. We have thus shown that relabeling preserves the validity condition for all arcs emanating from node and that each re-label operation strictly increases the value of*.* Finally, note that every incoming arc satisfies the inequality (by the induction hypothesis).Since*,* the re-label operation again preserves validity condition for arc*.*

The shortest augmenting path algorithm terminates when*,* indicating that the network contains no augmenting path from the source to the sink. Consequently, the flow obtained at the end of the algorithm is a maximum flow. We have thus proved the following theorem.

**Complexity of the Algorithm**

We now show that the shortest augmenting path algorithm runs in time.

***Lemma 3.1.2***: *If the algorithm re-labels any node at most k times, the algorithm saturates arcs (i.e., reduces their residual capacity to zero) at most times.*

***Proof:*** We show that between two consecutive saturations of an arc*,* both and must increase by at least 2 units. Since, by our hypothesis, the algorithm increases each distance label at most times, this result would imply that the algorithm could saturate any arc at most times. Therefore, the total number of arc saturations would be*,* which the assertion of the lemma is.

Suppose that an augmentation saturates an arc*.* Since the arc is admissible,

(i)

Before the algorithm saturates this arc again, it must send flow back from node to node*.* At this time, the distance labels and satisfy the equality

(ii)

In the next saturation of arc*,* we must have *(iii)*

Using (i) and (ii) in *(ii),* we see that

The inequalities in this expression follow from Lemma *2.1.* Similarly, it is possible to show that As a result, between two consecutive saturations of the arc*,* both and increase by at least 2 units, which is the conclusion of the lemma.

***Lemma 3.1.3:***

1. *In the shortest augmenting path algorithm each distance label increases at most n times. Consequently, the total number of re-label operations is at most.*
2. *The number of augment operations is at most nm/2.*

***Proof:*** Each re-label operation at node increases the value of by at least 1 unit. After the algorithm has relabeled nodeat most *n* times, *.* From this point on, the algorithm never again selects node during an advance operation since for every node in the partial admissible path, *.* Thus the algorithm re-labels a node at most times and the total number of re-label operations is bounded by*.* In view of Lemma 3.1.2, the preceding result implies that the algorithm saturates at most arcs. Since each augmentation saturates at least one arc, we immediately obtain a bound of on the number of augmentations.

***Property 3.1-1:*** *If the algorithm re-labels any node at most k times, the total time spent in finding admissible arcs and relabeling the nodes is*

***Theorem 3.1.2:*** *The shortest augmenting path algorithm runs in time.*

***Proof:*** Using Lemma 3.1.3 and 3.1-1 we find that the total effort spent in finding admissible arcs and in relabeling the nodes is*.* Lemma 3.1.3 implies that the total number of augmentations is*.* Since each augmentation requires time, the total effort for the augmentation operations is*.* Each retreat operation re-labels a node, so the total number of retreat operations is*.* Each advance operation adds one arc to the partial admissible path, and each retreat operation deletes one arc from it. Since each partial admissible path has length at most *n,* the algorithm requires at most advance operations. The first term comes from the number of retreat (re-label) operations, and the second term from the number of augmentations. The combination of these bounds establishes the theorem.

### 3.1.2 Ford Fulkerson Method

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. We call it a “method” rather than an “algorithm” because it encompasses several implementations with differing running times. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts. These ideas are essential to the important max-flow min-cut theorem. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method iteratively increases the value of the flow. We start with for all giving an initial flow of value 0. At each iteration, we increase the flow value in by finding an “augmenting path” in an associated “residual network” . Once we know the edges of an augmenting path in, we can easily identify specific edges in for which we can change the flow so that we increase the value of the flow. Although each iteration of the Ford-Fulkerson method increases the value of the flow, we shall see that the flow on any particular edge of may increase or decrease; decreasing the flow on some edges may be necessary in order to enable an algorithm to send more flow from the source to the sink. We repeatedly augment the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

In order to implement and analyze the Ford-Fulkerson method, let us first understand several additional concepts.

**Residual Network**

Given a flow network and a flow, the residual network consists of edges with capacities that represent how we can change the flow on edges of. An edge of the flow network can admit an amount of additional flow equal to the edge’s capacity minus the flow on that edge. If that value is positive, we place that edge into with a “residual capacity” of. The only edges of that are in are those that can admit more flow; those edge whose flow equals their capacity have, and they are not in .

The residual network may also contain edges that are not in, however. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge. In order to represent a possible decrease of a positive flow on an edge in , we place an edge into with residual capacity —that is, an edge that can admit flow in the opposite direction to ,at most canceling out the flow on These reverse edges in the residual network allow an algorithm to send back flow.

**Augmenting Paths in a Residual Graph**

Now we want to make precise the way in which we push flow from source *s* to sink *t* in*.* Let be a simple *s-t* path in that is does not visit any node more than once. We define to be the minimum residual capacity of any edge on, with respect to the flow*.* We now define the following operation which yields a new flow in.



Let

For each edge

If is a forward edge then

increase in by *;*

Else is a backward edge, and let)

decrease in by

Endif

Endfor

Return

The result of is a new flow in, obtained by increasing and decreasing the flow values on edges of. Let us first prove that obtained as a result of is indeed a flow.

***Lemma 3.1.4:***  *is a flow in*.

***Proof:*** We must verify the capacity and conservation conditions.

Since differs from only on edges of, we need to check the capacity conditions only on these edges. Thus, let be an edge of. Informally, the capacity condition continues to hold because if is a forward edge, we specifically avoided increasing the flow on e above *;* and if is a backward edge arising from edge, we specifically avoided decreasing the flow on e below 0. More concretely, note that is no larger than the residual capacity of If is a forward edge, then its residual capacity is*;* thus we have

So the capacity condition holds.

We need to check the conservation condition at each internal node that lies on the path. Let be such a node; we can verify that the change in the amount of flow entering is the same as the change in the amount of flow exiting *;* since satisfied the conservation condition at *,* so must .

**Ford Fulkerson Method:**

***Assumption: the capacity values are always integers.***

Max-Flow

Initially for all in

While there is an *s-t* path in the residual graph

Let be a simple *s-t* path in

*=*

Update to be

Update the residual graph to be

Endwhile

Return

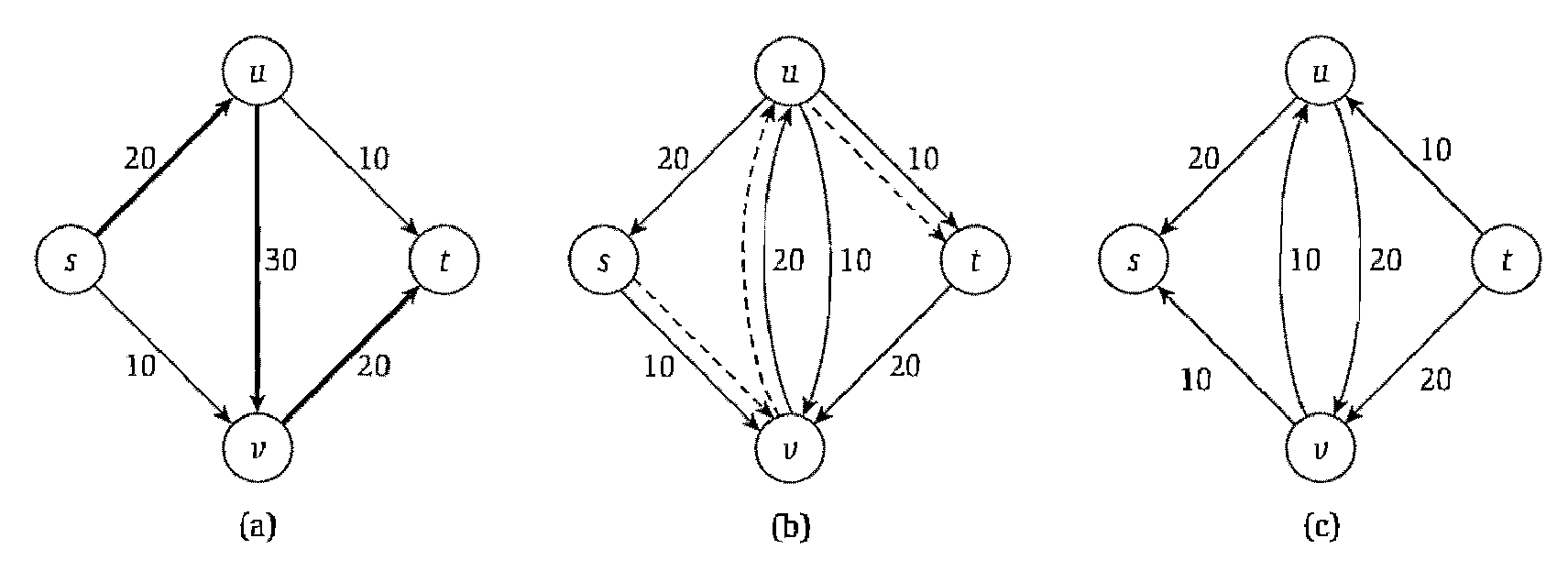


Figure 3.1.3 (a) the graph G with the path *s,* n, P, *t* used to push the first 20 units of flow. (b) The residual graph of the resulting flow *t,* with the residual capacity next to each edge. The dotted line is the new augmenting path. (c) The residual graph after pushing an additional 10 units of flow along the new augmenting path *s,* P, n, *t.*

**Analysis:**

We now analyze the Ford-Fulkerson algorithm based on the whether the algorithm terminates for all possible input values and the running time of the algorithm.

**Termination and Running time**

***Property 3.1-2:*** *At every intermediate stage of the Ford-Fulkerson Algorithm, the flow values {)} and the residual capacities in are integers.*

***Property 3.1-3:*** *Let f be a flow* in, *and let be a simple s-t path in Then*; *and since*, *we have.*

We need one more observation to prove termination: We need to be able to bound the maximum possible flow value. If all the edges out of s could be completely saturated with flow, the value of the flow would be .Let denote this sum. Thus we have for all *s-t* flows*.*

***Lemma 3.1.5:*** *If all capacities in the flow network*  *are integers, then the Ford-Fulkerson Algorithm terminates in at most*  *iterations of the While* *loop.*

***Proof:*** We noted above that no flow in G can have value than C, due to the capacity condition on the edges leaving S. Now, by (3.1-3), the value of the flow maintained by the Ford-Fulkerson Algorithm increases in each iteration; so by (3.1-2), it increases by at least 1 in each iteration. Since it starts with the value 0, and cannot go higher than C, the While loop in the Ford-Fulkerson Algorithm can run for at most C iterations.

***Lemma 3.1.6:*** *If all capacities in the flow network*  *are integer, then the Ford-Fulkerson Algorithm can be implemented to* run *in time.*

***Proof:*** We know from 3.1.5 that the algorithm terminates in at most iterations of the while loop. We, therefore, consider the amount of work involved in one iteration when the current flow is*.*

The residual graph has at most edges, since each edge of gives rise to at most two edges in the residual graph. We will maintain using an adjacency list representation; we will have two linked lists for each node*,* one containing the edges entering*,* and one containing the edges leaving*.* To find an *s-t* path in*,* we can use breadth-first search or depth-first search, which run in time; by our assumption that is the same as *.* The procedure takes time*,* as the path P has at most edges. Given the new flow, we can build the new residual graph in time: For each edge of, we construct the correct forward and backward edges in*.*

**Flows and Cuts**

Consider dividing the nodes of the graph into two sets, and, so that and*.* Any such division places an upper bound on the maximum possible flow value, since all the flow must cross from to somewhere. Formally, we say that an *s-t cut* is a partition of the vertex set, so that and. The *capacity* of a, denoted by, is simply the sum of the capacities of all edges out of**.**

***Property 3.1-4:*** *Let be any s-t flow, and any s-t cut. Then*

*OR*

*Let be any s-t flow, and any s-t cut. Then*

***Lemma 3.1.7:*** *Let be any s-t flow, and any s-t cut. Then*

***Proof:***

*.*

***It basically derives the fact that the value of every flow is upper-bounded by the capacity of every cut.*** In other words, if we exhibit any *s-t* cut in of some value *c\*,* we know immediately by 3.1.7 that there cannot be an *s-t* flow in of value greater than *c\*.* Conversely, if we exhibit any *s-t* flow in of some value *v\*,* we know immediately by 3.1.7 that there cannot be an *s-t* cut in of value less than *v\*.*

**Max-Flow Equals Min-Cut**

Let denote the flow that is returned by the Ford-Fulkerson Algorithm. We want to show that has the maximum possible value of any flow in, and we do this by the method discussed above: We exhibit an *s-t* cut *(A\*, B\*)* for which *\**, *\*.* This immediately establishes that has the maximum value of any flow, and that *\*\** has the minimum capacity of any *s-t* cut.



The Ford-Fulkerson Algorithm terminates when the flow has no *s-t* path in the residual graph  *.* This turns out to be the only property needed for proving its maximality.

***Lemma 3.1.8:*** *If f is an s-t-flow such that there is no s-t path in the residual graph, then there is an s-t cut (A\*, B\*) in for which v(f)* = *c(A\*, B\*). Consequently, f has the maximum value of any flow in*, *and (A\*, B\*) has the minimum capacity of any s-t* cut *in*.

***Proof:*** The statement claims the existence of a cut satisfying a certain desirable property; thus we must now identify such a cut. To this end, let *A*\* denote the set of all nodes *v* in for which there is an *s-t* path in  *.* Let B\* denote the set of all other nodes: *B\*= V* - *A\*.*

First we establish that *(A*\*, *B\*)* is indeed an *s-t* cut. It is clearly a partition of *V.* The source *s* belongs to *A\** since there is always a path from *s* to *s.* Moreover, by the assumption that there is no *s-t* path in residual graph,.

Next, suppose that is an edge in for which and*,* as shown in figure below. We claim that For if not, would be a forward edge in the residual graph *,* and since *,* there is an *s-u* path in *;* appending to this path, we would obtain an *s-v* path in *,* contradicting our assumption that *.*



Figure 3.1.4 The *(A\*,B\*)* cut

Now suppose that *)* is an edge in for which and*.* We claim that. For if not, would give rise to a backward edge in the residual graph*,* and since*,* there is an *s-v'* path in*;* appending to this path, we would obtain an *s-u'* path in*,* contradicting our assumption that*.*

So all edges out of are completely saturated with flow, while all edges into are completely unused. We can now use *3.1-4* to reach the desired conclusion:

### 3.1.3 The Edmonds-Karp algorithm

If we look at the bad examples for Ford/Fulkerson, we see that the bad behavior can be attributed to two causes:

• We use paths with little capacity.

• Our paths put flow on more edges than necessary.

The bound on Ford-Fulkerson method can be improved if we implement the computation of the augmenting path with a breadth-first search, that is, if the augmenting path is a *shortest* path from *s* to *t* in the residual network, where each edge has unit distance. We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***. We now prove that the Edmonds-Karp algorithm runs in time.

The analysis depends on the distances to vertices in the residual network. The following lemma uses the notation (*u*, *v*) for the shortest-path distance from *u* to *v* in, where each edge has unit distance.

***Lemma 3.1.3.1:*** *If the Edmonds-Karp algorithm is run on a flow network with source and sink, then for all vertices the shortest-path distance in the residual network increases monotonically with each flow augmentation.*

***Proof:*** We will suppose that for some vertex there is a flow augmentation that causes the shortest-path distance from to to decrease, and then we will derive a contradiction. Let  *be* the flow just before the first augmentation that decreases some shortest path distance, and let be the flow just afterward. Let be the vertex with the minimum whose distance was decreased by the augmentation, so that. Let be a shortest path from *s* to *v* in, so that and .

Because of how we chose, we know that the distance label of vertex did not decrease i.e. .

We claim that. If we had, then we would also have

.

.

.

Which contradicts our assumption that

The augmentation must have increased the flow from to The Edmonds-Karp algorithm always augments flow along shortest paths, and therefore the shortest path from to in have as its last edge. Therefore,

.

.

Which contradicts our assumption that we conclude that our assumption that such a vertex exists is incorrect.

***Theorem 3.1.3.1:*** *If the Edmonds-Karp algorithm is run on a flow network with source and sink, then the total number of flow augmentations performed by the algorithm is .*

***Proof:*** We say that an edge in a residual network is ***critical*** on an augmenting path if the residual capacity of is the residual capacity of, that is, if . After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We will show that each of the edges can become critical at most times.

Let and be vertices in that are connected by an edge in. Since augmenting paths are shortest paths, when is critical for the first time, we have

Once the flow is augmented, the edge disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from to is decreased, which occurs only if appears on an augmenting path. If ' is the flow in when this event occurs, then we have

Since, we have

.

.

Consequently, from the time becomes critical to the time when it next becomes critical, the distance of from the source increases by at least 2. The distance of from the source is initially at least 0. The intermediate vertices on a shortest path from to cannot contain or (since on the critical path implies that). Therefore, until becomes unreachable from the source, if ever, its distance is at most. Thus, can become critical at most times. Since there are pairs of vertices that can have an edge between them in a residual graph, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is. Each augmenting path has at least one critical edge, and hence the theorem follows.

## 3.2 Preflow Push Algorithms

In this section, we present the “push-relabel” approach to computing maximum flows. To date, many of the asymptotically fastest maximum-flow algorithms are push-relabel algorithms, and the fastest actual implementations of maximum-flow algorithms are based on the push-relabel method. Push-relabel methods also efficiently solve other flow problems, such as the minimum-cost flow problem.

### 3.2.1 Introduction

Push-relabel algorithms work in a more localized manner than the Ford-Fulkerson method. Rather than examine the entire residual network to find an augmenting path, push-relabel algorithms work on one vertex at a time, looking only at the vertex’s neighbors in the residual network. Furthermore, unlike the Ford-Fulkerson method, push-relabel algorithms do not maintain the flow-conservation property throughout their execution. They do, however, maintain a *preflow*.

For a given preflow x, we define for each node *i ∈ N – {s, t}*, the *excess e(v)* of a vertex *v* is defined as , net flow into *v*. If vertex v has positive excess and edge (v, w) has positive residual capacity, then an amount of flow excess up can be moved from *v* to w by δ to *f(v, w).*

A push-relabel algorithm performs two basic operations: pushing flow excess from a vertex to one of its neighbors and relabeling a vertex. The situations in which these operations apply depend on the heights of vertices. A distance or height function *d : V → Z+* for a preflow *f* is a function from the set of nodes to the nonnegative integers. We say that a distance function *d* is valid if it also satisfies the following two conditions:

1) *d(v) = 0;*

2) *d(v) ≤ d(w) + 1;* for all *(v,w) ∈ E* where *(v,w)* is a residual edge.

We call a vertex *v* active if *v ∈ V –{s,t}, d(v) < ∞* and *e(v) > 0.*

The basic operations in any push-relabel algorithm are given below:

**PUSH (v,w)**

* 1. **// Applies when**: v is overflowing, rf(v,w)>0, and d(v) = d(w) + 1.
  2. **// Action:** Push δ=min(e(v), rf(v,w)) units of flow from v to w
  3. δ = min(d(v), rf(v,w))
  4. if (v,w) ∈ E
  5. f(v,w) = f(v,w) + δ
  6. else f(w,v) = f(w,v) - δ
  7. e(v) = e(v) − δ
  8. e(w) = e(w) + δ

The code for PUSH operates as follows. Because vertex *v* has a positive excess *e(v)* and the residual capacity of *(v,w)* is positive, we can increase the flow from *v* to *w* by *δ = min(d(v), rf(v,w))* without causing *e(v)* to become negative or the capacity *rf(v,w)* to be exceeded. Line 3 computes the value δ, and lines 4–6 update f. Line 5 increases the flow on edge *(v,w)* because we are pushing flow over a residual edge that is also an original edge. Line 6 decreases the flow on edge *(w,v)* because the residual edge is actually the reverse of an edge in the original network. Finally, lines 7–8 update the excess flows into vertices *v* and *w*. Thus, if *f* is a preflow before PUSH is called, it remains a preflow afterward.

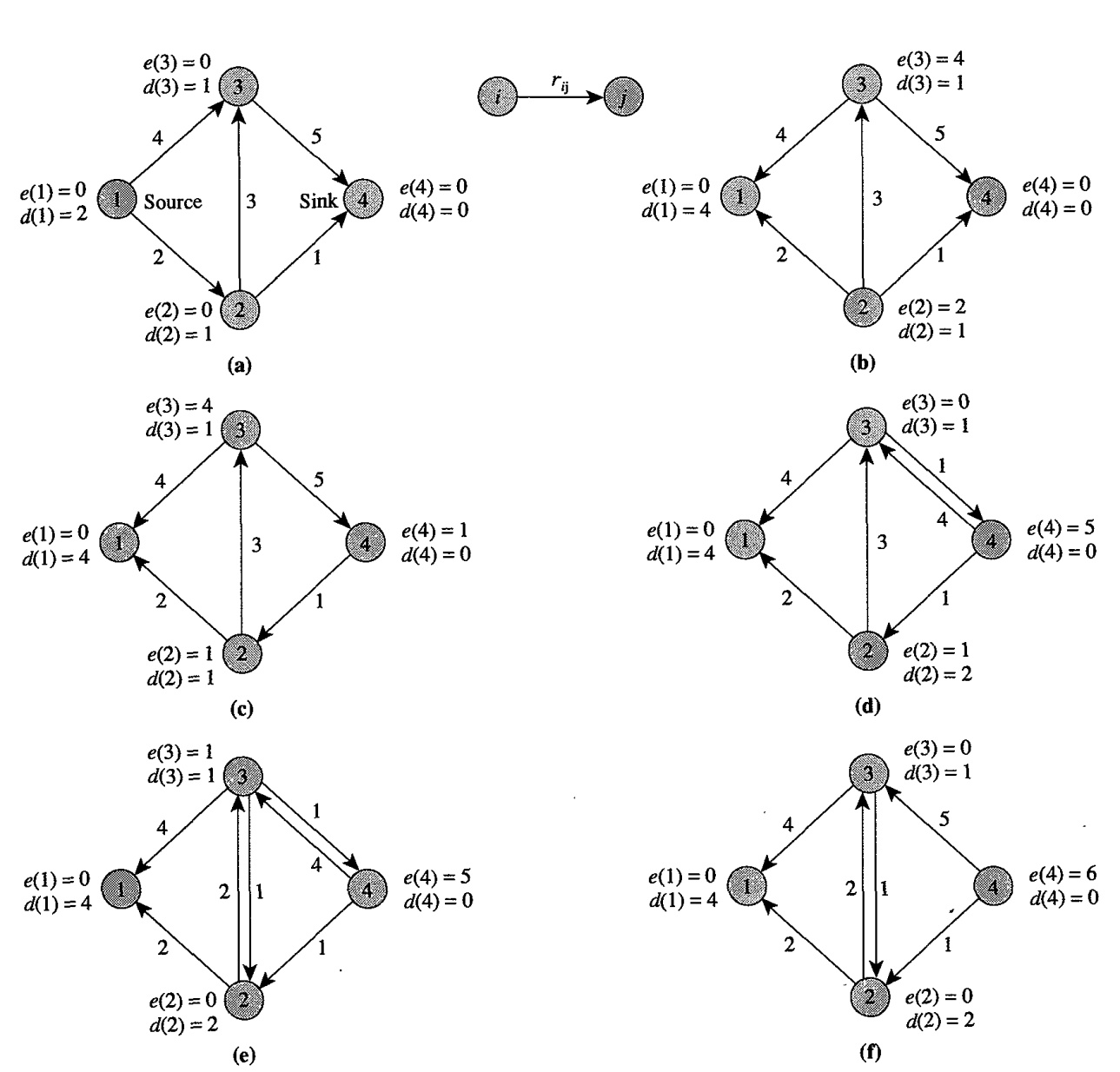
**RELABEL (v)**

1. **// Applies when:** v is overflowing and for all w ϵ V such that (v,w) ϵ Ef

we have d(v) ≤ d(w)

1. **// Action:** Increase the height of u.
2. d(v) = 1 + min(d(w) | (v,w) ∈ Ef)

The basic operation RELABEL (v) applies if *v* is overflowing and if *d(v) ≤ d(w)* for all edges *(v,w) ϵ Ef* . In other words, we can relabel an overflowing vertex *v* if for every vertex *w* for which there is residual capacity from *v* to *w*, flow cannot be pushed from *v* to *w* because *w* is not downhill from *v*.

****

**Figure 3.2.1: Illustration of generic preflow-push algorithm**

### 3.2.2 The Generic Algorithm

The generic push-relabel algorithm uses the following subroutine to create an initial preflow in the flow network. That is, we fill to capacity each edge leaving the source s, and all other edges carry no flow.

The generic algorithm may use initialize distance labels to zero or it may use reverse breadth-first search to initialize to appropriate values

**INITIALIZE-PREFLOW (G,s)**

1. x: = 0;
2. compute the exact distance labels d(i);
3. f(s,v): = c(s,v) for each arc (s, v) ∈ s.Adj;
4. d(s) : = n;

Initialization, followed by a sequence of push and relabel operations, executed in no particular order, yields the GENERIC-PUSH-RELABEL algorithm:

**GENERIC-PUSH-RELABEL (G)**

1. INITIALIZE-PREFLOW (G,s)
2. **while** there exists an applicable push or relabel operation
3. select an applicable push or relabel operation and perform it

**Analysis:**

***Lemma 3.2.1****: At any time during the execution of the algorithm and for any vertex v ∈ V,*

*d(v) ≤ 2n-1*

***Proof****:* The lemma is trivial for *v = s* and *v = t*. Suppose *v ∈ V – {s,t}.* Since the algorithm changes only labels of active vertices, it is enough to prove the lemma for an active vertex *v*. If *v* is active, then *e(v) > 0*, there is a simple path from *v* to *s* in Gf . Let v = v0, v1, ….. vi = s be such a path. The length 1 of the path is at most n - 1. Since d is a valid labeling and (vi,vi+1) ∈ Ef , we have *d(vi) ≤ d(vi+1) + 1*. Therefore, we have *d(v) = d(vo) ≤ d(vl) + 1 ≤ d(s) + (n - 1) = 2n - 1.*

***Lemma 3.2.2****: The number of relabeling operations is at most 2n - 1 per vertex and at most*

*(2n - 1)(n - 2) < 2n2 overall.*

***Proof:*** Relabeling operations apply only to vertices v ∈ V – {s, t}. A relabeling of v increases *d(v).* The label *d(v)* is zero initially, and the label can grow to at most *2n - 1*. Therefore, there are at most *(2n - 1)* relabelings of each vertex in v ∈ V – {s,t}, and the total number of relabelings is at most *(2n - l)(n - 2).*

***Lemma 3.2.3:*** *The number of saturating push operations is at most 2nm.*

***Proof****:* For any pair of vertices v and w, consider the saturating pushes from v to w and from w to v. If there are any such pushes, then either (v,w) ∈ E or (w,v) ∈ E. Consider a saturating push from v to w. In order to push flow from v to w again, the algorithm must first push flow from w to v, which cannot happen until d(w) increases by at least 2. Similarly, d(v) must increase by at least 2 between saturating pushes from w to v. Since d(v) + d(w) ≥ 1 when the first push between v and w occurs and d(v)+d(w) ≤ 4n-3 when the last such push occurs, the total number of saturating pushes between v and w is at most 2n-1. Thus the total number of saturating pushes is at most 2n - 1 per edge, for a total over all edges of at most (2n - 1)m < 2nm.

***Lemma 3.2.4****: The number of non-saturating pushing operations is at most 4n2m.*

***Proof****:* Let . Each non-saturating push from a vertex v to another vertex w causes ϕ to decrease by at least 1, since the push makes v inactive and d(w) = d(v) - 1. A saturating pushing operation causes ϕ to increase by at most 2n - 1. The total increase in ϕ due to saturating pushes is at most (2n - 1) × 2nm. The total increase in ϕ over the entire algorithm due to relabeling operations is at most (2n - l)(n - 2) by Lemma 2.3. Before the first push or relabel operation ϕ is zero, and at the end of the algorithm ϕ is also zero. Thus the total decrease in ϕ, and hence the total number of non-saturating pushing operations, is equal to the total increase in ϕ, which is at most (2n - 1)2nm + (2n - l)(n - 2) ≤ 4n2m (recall the assumption m ≥ n - 1).

***Theorem 3.2.5****: The generic algorithm terminates after O(n2m) basic operations.*

***Proof****:* Immediate from Lemmas 3.2.2,3.2.3 and 3.2.4.

### 3.2.3 FIFO Preflow-Push Algorithm

We need some data structures to represent the network and the preflow. We associate the three values *u(v,w), u(w,v),* and *f(v,w)= -f(w,v)* with each undirected edge *{v,w} of G.* Each vertex *v* has a list of the incident edges *{v,w},* in fixed but arbitrary order. Thus each edge *{v,w}* appears in exactly two lists, the one for *v* and the one for *w.* Each vertex *v* has a *current edge {v, w},* which is the current candidate for a pushing operation from *v.* Initially, the current edge of *v* is the first edge on the edge list of *v.* The main loop of the implementation consists of repeating the *discharge* operation described until there are no active vertices. The *discharge* operation is applicable to an active vertex *v.* This operation iteratively attempts to push the excess at *v* through the current edge *{v,w}* of *v* if a pushing operation is applicable to this edge. If not, the operation replaces *{v,w}* as the current edge of *v* by the next edge on the edge list of *v;* or, if *{v,w}* is the last edge on this list, it makes the first edge on the list the current one and relabels *v.* The operation stops when the excess at *v* is reduced to zero or *v* is relabeled.

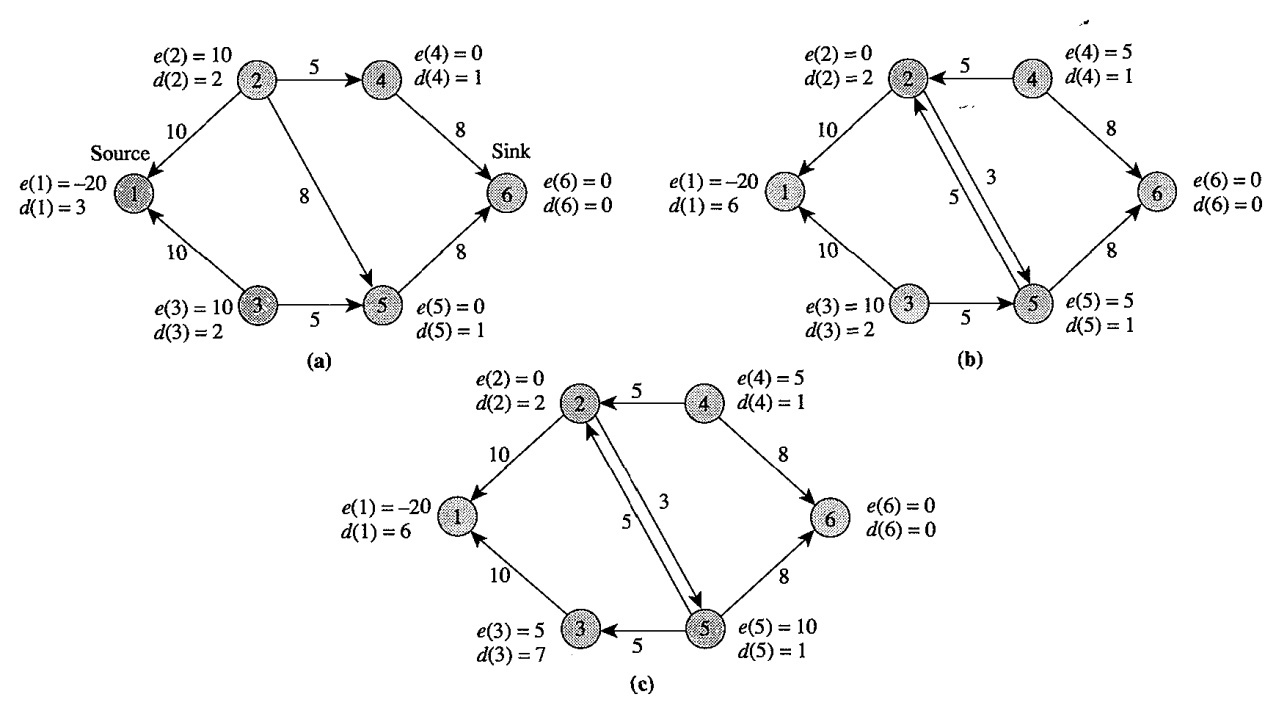
**discharge(v) :**

1. **Applicability**: *v* is active.
2. **Action**: Let *{v, w}* be the current edge of u;
3. **If** push(v, w) is applicable then push(v, w)
4. **else**
5. **if** {v, w) is not the last edge on the edge list of v then
6. replace {v, w) as the current edge of v
7. by the next edge on the edge list of v
8. **else begin**
9. make the first edge on the edge list of v the current edge;
10. relabel(v);
11. **end**.

**FIFO:**

1. **Applicability**: Q ≠ 0.
2. **Action**: Remove the vertex v on the front of Q.
3. (Vertex v must be active.)
4. **Repeat**
5. discharge(v);
6. **if** w becomes active during this discharge operation **then**
7. add w to the rear of Q;
8. **until** e(v) = 0 or d(v) increases.
9. **If** v is still active then add v to the rear of Q.

We maintain Q as a queue. The first-in, first-out algorithm consists of applying the FIFO operation until Q is empty. The FIFO operation consists of removing the vertex on the front of Q, applying discharge operations to this vertex at least until the excess becomes zero or the label of the vertex increases, and adding any newly active vertices to the rear of Q (including v if it is still active).



**Figure 3.2.2 Illustration of FIFO preflow-push algorithm**

**Analysis:**

To analyze the first-in, first-out algorithm, we need to introduce the concept of passes over the queue. Pass one consists of the FIFO operations applied to the vertices added to the queue during the initialization. Given that pass i is defined, pass i + 1 consists of the FIFO operations applied to vertices on the queue that were added during pass i.

***Lemma 3.2.6****: The number of passes over the queue is at most 4n2.*

***Proof****:* Let ϕ = max{d(v) | v is active}. Consider the effect on ϕ of a single pass over the queue. If no distance label changes during the pass, each vertex has its excess moved to lower-labeled vertices, so ϕ decreases during the pass. If ϕ is not changed by the pass, some vertex label must increase by at least 1. If ϕ increases, some vertex label must increase by at least as much as ϕ increases. The total number of passes in which ϕ stays the same or increases is thus at most 2n2 by Lemma 3.2.1. Since ϕ = 0 initially and at the end of the algorithm, the total number of passes in which ϕ decreases is also at most 2n2. Hence the total number of passes is at most 4n2.

***Corollary 3.2.7****: The number of non-saturating pushes during the first-in, first out algorithm is at most 4n3.*

***Proof:*** There is at most one non-saturating push per vertex in V – {s, t } per pass.

**Theorem 3.2.8:** The first-in, first-out algorithm runs in O(n3) time.

**Proof:** Immediate from Theorem 3.2.5 and Corollary 3.2.7.

### 3.2.4 Highest-Label Preflow-Push Algorithm

The highest-label preflow-push algorithm always pushes flow from an active node with the highest distance label.The implementation maintains an array of sets Bi, 0 < i < *2n* - 1, and an index *b* into the array. Set B, consists of all active vertices with label i, represented as a doubly-linked list, so that insertion and deletion take O(l) time. The index *b* is the largest label of an active vertex. During the initialization, when the arcs going out of the source are saturated, the resulting active vertices are placed in *B0,* and *b* is set to 0. At each iteration, the algorithm removes a vertex from *Bi,* processes it using the *discharge* operation, and updates *b.* The algorithm terminates when *b* becomes negative, *i.e.* when there are no active vertices. This processing of vertices, which implements the *while* loop of the generic algorithm, is described.

**process-vertex:**

1. remove a vertex v from Bb;
2. old-label ← d(v);
3. discharge(v);
4. add each vertex w made active by the discharge to Bd(w)
5. **if** d(v) ≠ old-label **then** **begin**
6. b ← d(v)
7. add v to Bb
8. **end**
9. **else** **if** Bb = ∅ then b ← b – 1;

To understand why the *process-vertex* procedure correctly maintains *b,* note that *discharge(V)* either relabels *v* or gets rid of all excess at *v,* but not both. In the former case, *v* is the active vertex with the largest distance label, so *b* must be increased to *d(v).* In the latter case, the excess at *v* has been moved to vertices with distance labels of *b -* 1, so if *Bb* is empty, then *b* must be decreased by one. The total time spent updating *b* during the course of the algorithm is *O(n2).*

### 3.2.5 Dynamic Tree Algorithm

A dynamic tree is an important data structure that researchers have used extensively to improve the worst-case complexity of several network algorithms. In this section we describe the use of this data structure for the shortest augmenting path algorithm. We do not describe how to actually implement the dynamic tree data structure; rather, we show how to use this data structure as a "black box" to improve the computational complexity of certain algorithms.

The following observation serves as a motivation for the dynamic tree structure. The shortest augmenting path algorithm repeatedly identifies a path consisting solely of admissible arcs and augments flows on these paths. Each augmentation saturates some arcs on this path, and by deleting all the saturated arcs from this path we obtain a set of *path fragments:* sets of partial paths of admissible arcs. The path fragments contain valuable information. If we reach a node in any of these path fragments using any augmenting path, we know that we can immediately extend the augmenting path along the path fragment. The standard implementation of the shortest augmenting path algorithm discards this information and possibly regenerates it again at future steps. The dynamic tree data structure cleverly stores-these path fragments and uses them later to identify augmenting paths quickly.

The dynamic tree data structure maintains a collection of node-disjoint rooted trees, each arc with an associated value, called *val.* Each rooted tree is a directed in-tree with a unique root. Notice that, according to our definitions, each node is its own ancestor and descendant.

Dynamic Tree Operations are as followed:

*1)* ***find-root (i).***Find and return the root of the tree containing node *i.*

2) **find-value (i).** Find and return the value of the tree arc leaving node *i.* If *i* is a root node, return the value ∞.

*3)* ***find-min (i).***Find and return the ancestor *w* of *i* with the minimum value of find-value (w). In case of a tie, chose the node *w* closest to the tree root.

*4)* ***change-value (i,val).***Add a real number *val* to the value of every arc along the path from node *i to* find-root(i).

5)***link (i,*j,*val).***This operation assumes that *i* is a tree root and that *i* and j belong to different trees. The operation combines the trees containing nodes *i* and j by making node j the parent of node *i* and giving arc *(i,* j) the value *val.*

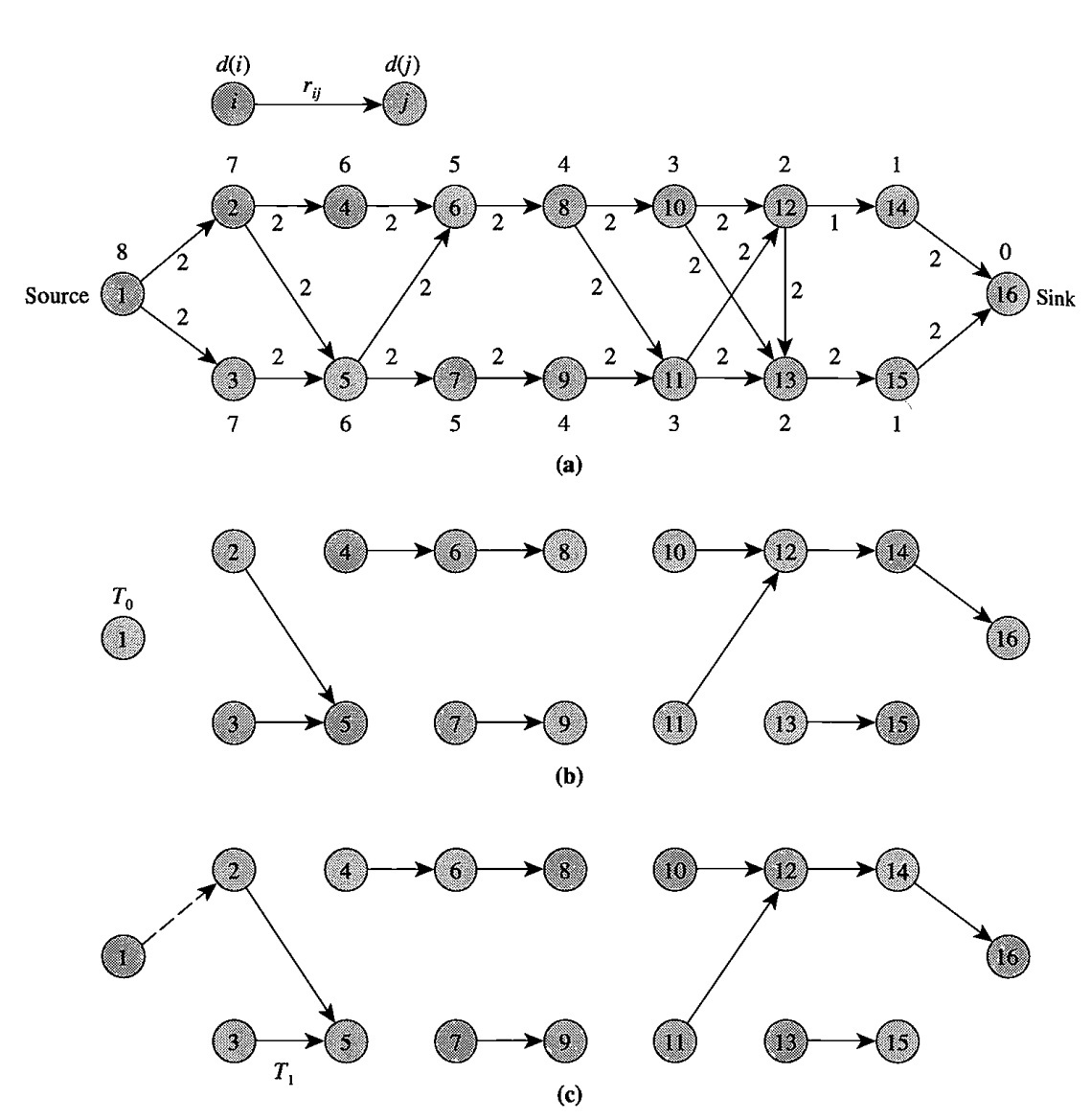
6)***cut (i).***Break the tree containing node *i* into two trees by deleting the arc joining node *i* to its parent and returning the value of the deleted arc. We perform this operation when *i* is not a tree root.

The following important result, which we state without proof, lies at the heart of the efficiency of the dynamic tree data structure.

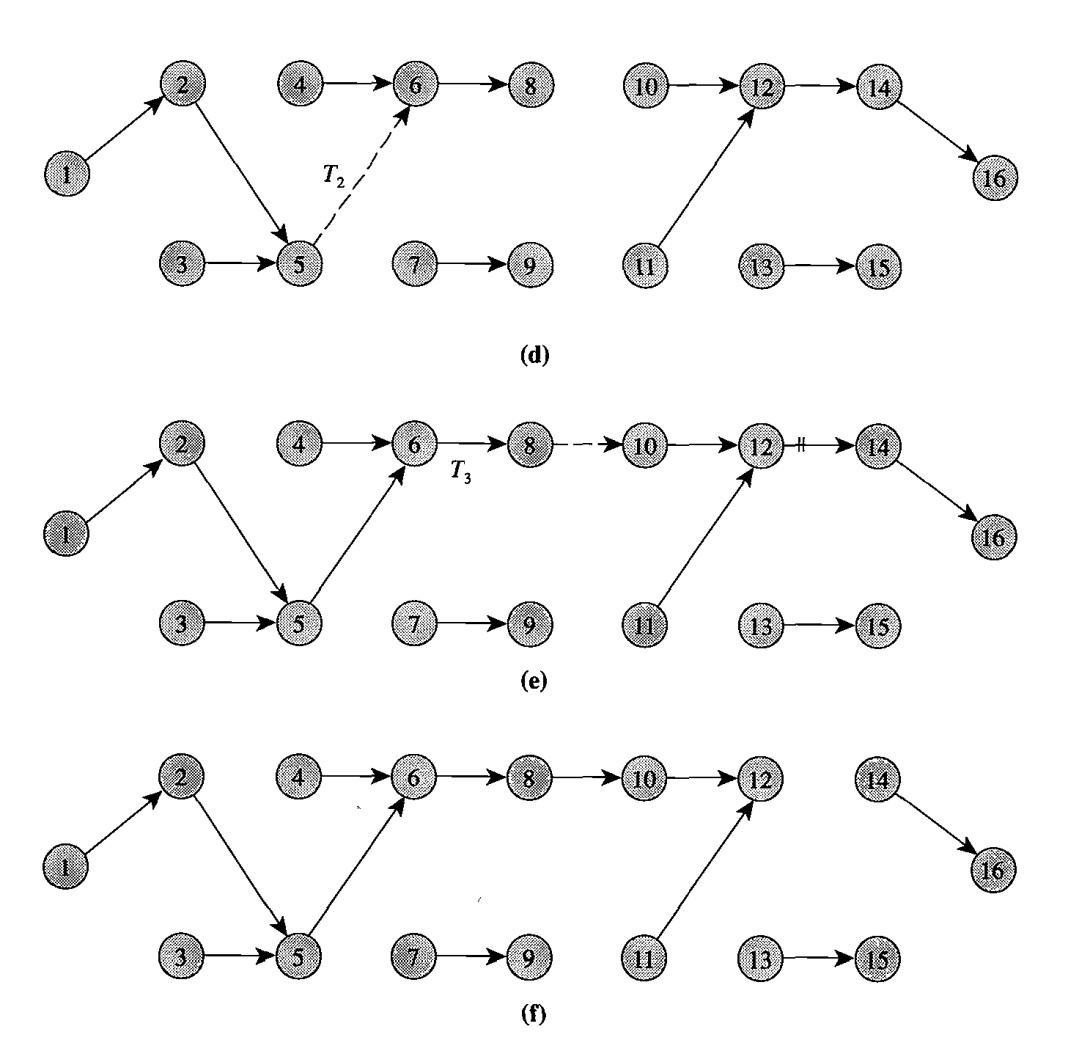
***Lemma 3.2.9****: If z is the maximum tree size (i.e., maximum number of nodes in any tree), a sequence of l tree operations, starting with an initial collection of singleton trees, requires a total of O(l log(z + l)) time.*

The following basic idea underlies the algorithmic speed-up. In the dynamic tree implementation, each arc in the rooted tree is an admissible arc (recall that an arc *(i,* j) is admissible if *rij* > 0 and *d(i)* = *d(j)* + 1). The value of an arc is its residual capacity. For example, consider the residual network given in Figure 3.2.3(a), which shows the distance labels next to the nodes and residual capacities next to the arcs. Observe that in this network, every arc, except the arc (12, 13), is admissible; moreover, the residual capacity of every arc is 2, except for the arc (12, 14) whose residual capacity is 1. Figure 3.2.3(b) shows one collection of rooted trees for this example. Notice that although every tree arc is admissible, every admissible arc need not be in some tree.

We first show how the algorithm works on our example. It maintains a rooted tree containing the source node and progressively expands this tree until it contains the sink node, at which point the algorithm performs an augmentation. To grow the tree containing the source, the algorithm repeatedly performs link operations. In our example, the algorithm starts with the singleton tree *To* containing only the source node 1 [see Figure 3.2.3(b)]. It identifies an admissible arc emanating from node 1. Suppose that we select arc(1, 2). The algorithm performs the operation link (1, 2, 2), which joins two rooted trees, giving us a larger tree *T1* containing node 1 [see Figure 3.2.3(c)]. The algorithm then identifies the root of *T1 ,* by performing the operation find-root(1), which identifies node 5. The algorithm tries to find an admissible arc emanating from node 5. Suppose that the algorithm selects the arc (5, 6). The algorithm performs the operation link(5, 6, 2) and obtains a larger tree *T2* containing node 1 [see Figure 3.2.3(d)]. In the next iteration, the algorithm identifies node 8 as the root of *T2 .* Suppose that the algorithm selects arc (8, 10) as the admissible arc emanating from node 8.



**Figure 3.2.3: Finding an augmenting path in the dynamic tree implementations.**



**Figure 3.2.3 :( Continued)**

The algorithm performs the operation link(8, 10, 2) and obtains a rooted tree *T3* that contains both the source and sink nodes [see Figure 3.2.3(e)].

Observe that the unique path from the source to the sink in *T3* is an admissible path since by construction every arc in a rooted tree is admissible. The residual capacity of this path is the minimum value of the arcs in this path. The operation find-min(1) would determine an ancestor of node 1 with the minimum value of find-value, which is node 12 in our example. Performing find-value(12) will give us the residual capacity of this path, which is 1 in this case. We have thus discovered the possibility of augmenting 1 unit of flow along the admissible path and that arc (12, 14) is the blocking arc. We perform the augmentation by executing change-value(1, -1). This augmentation reduces the residual capacity of arc (12, 14) to zero. The arc (12, 14) now becomes inadmissible and we must drop it from the collection of rooted trees. We do so by performing cut(12). This operation gives us the collection of rooted trees shown in Figure 3.2.3(f).

The tree-augment procedure can be explained as follows. If node *p* is an ancestor of node s with the minimum value of find-value(p) and, among such nodes in the path, it is Closest to the sink, then find-value(p) gives the residual capacity of the augmenting path. The operation change(s, -δ) implicitly updates the residual capacities of all the arcs in the augmenting path. This augmentation might cause the capacity of more than one arc in the path to become zero. The whileloop identifies all such arcs, one by one, and deletes them from the collection of rooted trees.

**algorithm** *tree-augmenting-path;*

**begin**

x: = 0;

perform a reverse breadth-first search of the residual network

from node *t* to obtain the distance labels *d(i);*

let *T* be the collection of all singleton nodes;

*i:* = s;

**while** *d(s)* < *n* **do**

**begin**

**if** *i* has an admissible arc **then** *tree-advance(i)*

**else** *tree-retreat(t);*

**if** *i* = *t* **then** *tree-augment;*

**end;**

**end;**

**procedure** *tree-advance(t);*

**begin**

let *(i, j)* be an admissible arc in *A(i);*

*Iink(i, j, rij);*

*i:* = *find-root(});*

**end;**

**procedure** *tree-retreat(t);*

**begin**

*d(i)* : = *min{d(j)* + 1 : *(i, j)* ∈ *A(i)* and *rij>* O};

**for** each tree arc *(k, i)* **do** *cut(k);*

*i:* = *find-root(s);*

**end;**

**procedure** *tree-augment;*

**begin**

*p:* = *find-min(s);*

δ : = *find-value(p);*

*change-value(s, -𝛿);*

**while** *find-value(p)* = 0 **do** *cut(p)* and set *p* : = *find-min(s);*

*i:* = *find-root(s);*

**end;**

We now consider the worst-case complexity of the algorithm. The bottleneck operations in the original shortest augmenting path algorithm are the advance and augment operations, which require *O(n2m)* time. Each advance operation in the original algorithm adds one arc; in contrast, the tree implementation adds a collection of arcs using the link operation. Thus the dynamic tree implementation substantially reduces the number of executions of the link operation. Similarly, while augmenting flow, the tree implementation augments flow over a collection of arcs by performing the operation change-value, thus again substantially reducing the number of required updates. We now obtain a bound on the number of times the algorithm performs various tree operations. We will show that the algorithm performs each of the tree operations *O(nm)* times.

**cut(j).** The algorithm performs this operation during the tree-retreat and tree-augment operations. During the tree-retreat(i) operation, the algorithm might perform this operation as many times as the number of incoming arcs at node i. Since this operation relabels node i, and we can relabel a node at most *n* times, these operations sum to *O(n2)* over all nodes. Furthermore, during the tree-augment operation, we perform the cut operation for each arc saturated during an augmentation. Since the total number of arc saturations is *O(nm),* the number of these operations sums to *O(nm).*

**link(i, *j,* val).** Each link operation adds an arc to the collection of rooted trees. Observe that if an arc enters a rooted tree, it remains there until a cut operation deletes it from the tree. Therefore, the number of link operations is at most *(n -1)* plus the number of cut operations. The term *(n* - 1) arises because initially the collection might contain no arc, and finally, it might contain as many as *(n* - 1) arcs. Consequently, the total number of link operations is also *O(nm).* Since each tree-advance operation performs a link operation, the previous result also implies an *O(nm)* bound on the total number of tree-advance operations.

**change-value(i, val).** The algorithm performs this operation once per augmentation. Since the number of augmentations is at most *nml2,* we immediately obtain a bound of *O(nm)* on the number of change-value operations.

**find-value(i)** and **find-min(i).** The algorithm performs each of these two operations once per augmentation and once for each arc saturated during the augmentation. These observations imply a bound of *O(nm)* on the number of executions of these two operations.

**find-root(i).** The algorithm performs this operation once during each execution of the tree-advance, tree-augment, and tree-retreat operations. Since the algorithm executes the first two operations *O(nm)* times and the third operation *O(n2)* times, it executes the find-root operation *O(nm)* times.

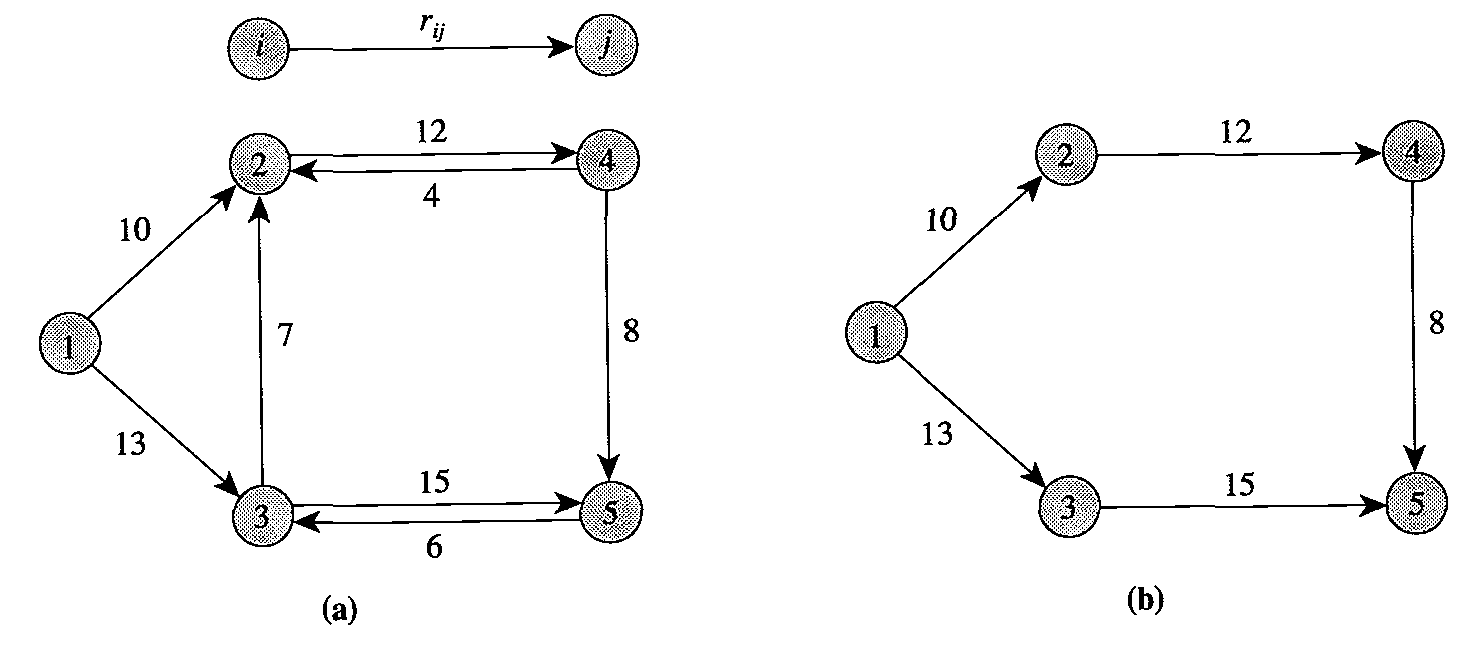
Using simple arguments, we have now shown that the algorithm performs each of the six tree operations *O(nm)* times. It performs each tree operation on a tree of maximum size *n.* The use of Lemma 3.2.9 establishes the following important result.

***Theorem 3.2.10:*** *The dynamic tree implementation of the shortest augmenting path algorithm solves the maximum flow problem in time.*

## 3.3 Scaling Algorithms

### 3.3.1 Capacity Scaling Algorithms

This algorithm always augments flow along a path with the maximum residual capacity. Let x be any flow and let v be its flow value. As we have seen, the maximum capacity augmentation algorithm reduces the number of augmentations in the generic labeling algorithm from O(nU) to O(mlogU). However, the algorithm performs more computations per iteration, since it needs to identify an augmenting path with the maximum residual capacity, not just any augmenting path. We now suggest a variation of the maximum capacity augmentation algorithm that does not perform more computations per iteration and yet establishes a maximum flow within O(mlogU). Since this algorithm scales the arc capacities implicitly, we refer to it as the capacity scaling algorithm.



**Figure 3.3.1: illustrating ∆ residual network (a) residual network G(x); (b) ∆-residual network G(x, ∆) for ∆ = 8**

The essential idea underlying the capacity scaling algorithm is conceptually quite simple. We augment flow along a path with a sufficiently large residual capacity, instead of a path with the maximum augmenting capacity because we can obtain a path with a sufficiently large residual capacity fairly easily-in O(m) time. To define the capacity scaling algorithm, let us introduce a parameter ∆ and, with respect to a given flow x, define the ∆-residual network as a network containing arcs whose residual capacity is at least ∆.et G(x,∆) denote the ∆-residual network. Note that G(x, 1) = G(x) and G(x, ∆) is a sub-graph of G(x). Figure 3.3.1(a) gives the residual network G(x) and Figure 3.3.2(b) gives the ∆-residual network G(x,∆) for ∆ = 8.

Let us refer to a phase of the algorithm during which ∆ remains constant as a scaling phase and a scaling phase with a specific value of ∆ as a ∆-scaling phase. Observe that in a ∆-scaling phase, each augmentation carries at least ∆ units of flow. The algorithm starts with and halves its value in every scaling phase until ∆ = 1. Consequently, the algorithm performs scaling phases. In the last scaling phase ∆ = 1, so G(x, ∆) = G(x). This result shows that the algorithm terminates with a maximum flow. The efficiency of the algorithm depends on the fact that it performs at most 2m augmentations per scaling phase.

**Capacity Scaling:**

x = 0

∆ =

**while** ∆ ≥ 1 **do**

**while** G(x, ∆) contains a path from node s to node t **do**

identify a path P in G(x, ∆)

δ = min(rij(i, j) ∈ P);

augment δ units of flow along P and update G(x, ∆) ;

**end**

∆ = ∆ = 2

**end**

***Theorem 3.3.1****: The capacity scaling algorithm solves the maximum flow problem within O(mlogU) augmentations and runs in O(m21ogU) time.*

### 3.3.2 Excess Scaling Algorithm

Our maximum flow algorithm improves the generic preflow-push algorithm of Section 3.2 by using excess scaling to reduce the number of non-saturating pushes from O(nm) to O(nlogU). The basic idea is to push flow from active nodes with sufficiently large excesses to nodes with sufficiently small excesses while never letting the excesses become too large. We refer to our algorithm as the excess scaling algorithm. The algorithm performs scaling iterations. For a scaling iteration, the excess-dominator is defined to be the least integer ∆ that is a power of 2 and satisfies e ≤ ∆ for all i ∈ N. In a scaling iteration, we guarantee that each non-saturating push sends at least ∆/2 units of flow and that the excess-dominator does not increase.

To ensure that each non-saturating push has a value of at least ∆/2, we consider only nodes with an excess more than ∆/2; and among these nodes with large excess, we select a node with a minimum distance label. This choice ensures that the flow will be sent to a node with a small excess. We show that after at most 8n2 non-saturating pushes, the excess-dominator decreases by a factor of at least 2, and a new scaling iteration begins. After at most K scaling iterations, all node excesses drop to zero and we obtain a maximum flow.

In order to select an active node with excess more than A/2 and with a minimum distance label among such nodes, we maintain the lists LIST(r) = iE ∈ N : e­i > ∆/2 and d(i) = r for each r = 1 … 2n - 1. These lists can be maintained in the form of either linked stacks or queues which enables insertion and deletion of elements in O(1) time. As per Goldberg and Tarjan, we use the following data structure to efficiently select the admissible arc for pushing flow out of a node. We maintain with each node i the list, A(i), of arcs directed out of it. Arcs in each list can be arranged arbitrarily, but once the order is decided, it remains unchanged throughout the algorithm. Each node i has a current arc (i; j), which is the current candidate.

**Complexity of Algorithm**

In this section, we show that the distance directed preflow-push algorithm with excess scaling correctly computes a maximum flow in O(nm + n2logU) time.

***Lemma 3.3.2****: The excess scaling algorithm satisfies the following two conditions:*

*1. Each non-saturating push from a node i to a node j sends at least ∆/2 units of flow.*

*2. No excess increases above ∆ (i.e., the excess-dominator does not increase subsequent to a push).*

***Proof****:* For every push on arc (i, j) we have ei > ∆/2 and ej ≤ ∆/2 because node i is a node with the smallest distance label among nodes whose excess is more than ∆/2, and d(j) = d(i) - 1 < d(i) by the property of the push operation. Hence, by sending units of flow, we ensure that in a non-saturating push the algorithm sends at least ∆/2 units of flow. Furthermore, the push operation increases the excess at node j only. Let e be the excess at node j after the push.

Then . All node excesses thus remain less than or equal to ∆.

***Lemma 3.3.3****: If each push satisfies conditions C3 and C4, then the number of non-saturating pushes per scaling iteration is at most 8n2.*

***Proof:*** Consider the potential function . The initial value of F at the beginning of the ∆-scaling iteration is bounded by 2n2 because ei is bounded by ∆ and d(i) is bounded by 2n. When the algorithm examines node i, one of the following two cases must apply.

Case 1: The algorithm is unable to find an arc along which flow can be pushed. This case occurs when the current arc of node i reaches the end of A(i). Observe that if an arc (i; j) is found to be inadmissible earlier, then it remains inadmissible until d(i) increases because d(j) is nondecreasing. Hence, there exists no admissible arc emanating from node i and the relabel operation increases d(i) by ϵ > 1 units. This increases F by at most ϵ units. Since the total increase in d(i) throughout the running of the algorithm for each i is bounded by 2n, the total increase in F due to relabelings of nodes is bounded by 2n2 in the scaling iteration. (Actually, the increase in F due to node relabelings is at most 2n2 over all scaling iterations.)

Case 2.The algorithm is able to identify an arc on which flow can be pushed and so it performs either a saturating or a nonsaturating push. In either case, F decreases. A nonsaturating push on arc (i, j) sends at least ∆/2 units of flow from node i to node j and since d(j) = d(i) - 1, this decreases F by at least 1/2 units. As the initial value of F for a scaling iteration plus the increases in F sum to at most 4n2, this case cannot occur more than 8n2 times.

***Theorem 3.3.4:*** *The scaling algorithm performs non-saturating pushes.*

***Proof****:* The initial value of the excess-dominator ∆ is. The value of the

Excess-dominator decreases by a factor of 2 within 8n2 nonsaturating pushes and a new scaling iteration begins. After such scaling iterations, ∆ < 1; and by the integrality of the flow ei = 0 for all I ∈ N – {s, t}. The algorithm thus obtains a feasible flow, which is a maximum flow.

***Theorem 3.3.5:*** *The complexity of the excess scaling algorithm is O(nm + n2logU).*

***Proof:*** The complexity of the algorithm depends upon the number of executions of the while loop in the main program. In each such execution, either a PUSH/RELABEL(i) step is performed or the value of the variable level increases.

Each execution of the procedure PUSH/RELABEL(i) results in one of the following outcomes.

Case 1: A push is performed. Since the number of saturating pushes is O(nm) and the number of non-saturating pushes is O(n2logU), this case occurs O(nm + n2logU) times.

Case 2: The distance label of node i goes up. By Corollary, this case can occur O(n) times for each node i and O(n2) in total.

Thus the algorithm calls the procedure PUSH/RELABEL O(nm + n21ogU) times. The effort needed to find an arc to perform the push operation is 0(1) plus the number of times the current arc of node i is replaced by the next arc in A(i). After |A(i)| such replacements for node i, Case 2 occurs and the distance label of node i goes up. Thus, the total effort needed is plus the number of PUSH/RELABEL operations. This is clearly O(nm + n2logU). Next consider the time needed for relabeling operations. Computing the new distance label of node I requires examining arcs in A(i). This yields a total of time for all relabeling operations. The lists LIST(r) are stored as linked stacks and queues, hence, addition and deletion of any element takes O(1) time. Consequently, updating these lists is not a bottleneck operation.

Finally, we need to bound the number of increases of the variable level. In each scaling iteration, level is bounded above by 2n and bounded below by 1. Hence, its number of increases per scaling iteration is bounded by the number of decreases plus 2n. Furthermore, level can decrease only when a push is performed and, in such a case, it decreases by . Hence, its increases over all scaling iterations are bounded by the number of pushes plus , which is again *O(nm + n2log U).*

|  |  |
| --- | --- |
| **Algorithm** | **Complexity** |
| Ford-Fulkerson | O(E |*f*|) |
| Edmond-Karp | O(VE2) |
| Generic Push-Relabel | O(V2E) |
| FIFO Push-Relabel | O(V3) |
| Highest Label Push-Relabel | O(V3) |
| Dynamic Tree Algorithm | O(VE log V) |
| Capacity Scaling | O(E2log U) |
| Excess Scaling | O(VE + V2log U) |

Table 3.1 Running time of max-flow algorithms

## 

## Chapter 4

## Applications

The maximum flow problem and the minimum cut problem, arise in a wide variety of situations and in several forms. The problem also arises directly in problems as far reaching as machine scheduling, the assignment of computer modules to computer processors, the rounding of census data to retain the confidentiality of individual households, and tanker scheduling.

## 4.1 Feasible Flow Problem

The feasible flow problem requires that we identify a flow *x* in a network G = *(V, E)* satisfying the following constraints:

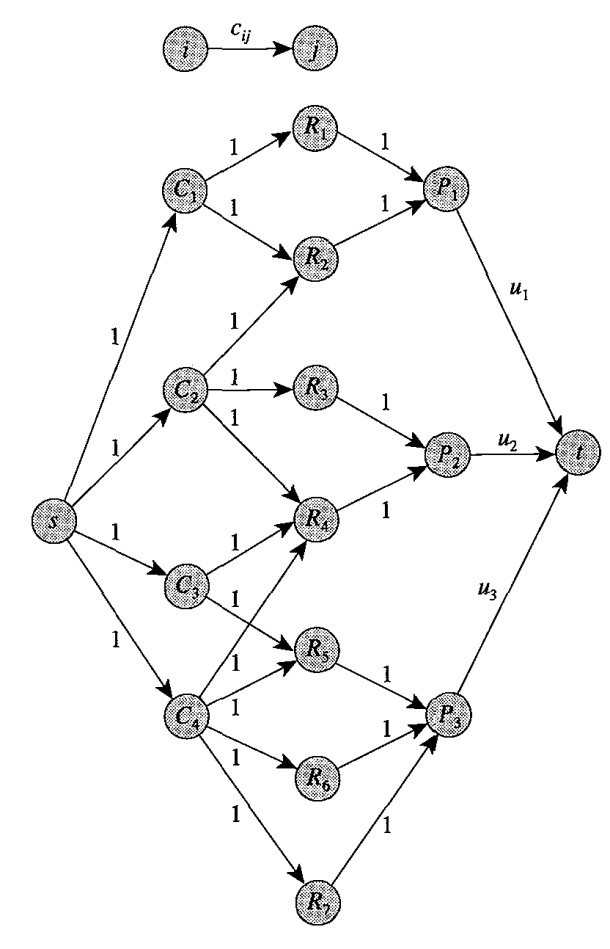
Here d(i) is the demand associated with vertex i. For feasible flow, we want that . The following distribution scenario illustrates how the feasible flow problem arises in practice. Suppose that merchandise is available at some seaports and is desired by other ports. We know the stock of merchandise available at the ports, the amount required at the other ports, and the maximum quantity of merchandise that can be shipped on a particular sea route. We wish to know whether we can satisfy all of the demands by using the available supplies.

We can solve the feasible flow problem by solving a maximum flow problem defined on an augmented network as follows. We introduce two new nodes, a source node *s* and a sink node *t.* For each node *i* with *d(i)* > 0, we add an arc *(s, i)* with capacity *d(i),* and for each node *i* with *d(i)* < 0, we add an arc *(i, t)* with capacity *-d(i).*

We refer to the new network as the *transformed network.* Then we solve a maximum flow problem from node *s* to node *t* in the transformed network. If the maximum flow saturates all the source and sink arcs, problem has a feasible solution; otherwise, it is infeasible.

4.2 Problem of Representatives

A town has *r* residents *RI , R2 ,* … *Rr; q* clubs CI , *C2 ,* … *Cq ;* and *p* political parties *P1, P2 ,* … *Pp .* Each resident is a member of at least one club and can belong to exactly one political party. Each club must nominate one of its members to represent it on the town's governing council so that the number of council members belonging to the political party *Pk* is at most u*k.* We want to find a council that satisfies above constraints. We illustrate this formulation with an example. We consider a problem with *r* = 7, *q* = 4, *p* = 3, and formulate it as a maximum flow problem in Figure 4.1.



**Figure 4.1: System of direct representatives**

The network also contains a source node *s* and a sink node t*.* It contains an arc *(s,*Ci) for each node *Ci* denoting a club, an arc *(Ci,,Rj)* whenever the resident *Rj* is amember of the club *Ci ,* and an arc *(Rj , Pk )* if the resident *Rj* belongs to the politicalparty *Pk.* Finally, we add an arc *(Pk,* t*)* for each *k* = 1... 3 of capacity u*k;* allother arcs have unit capacity.

We next find a maximum flow in this network. If the maximum flow value equals *q,* the town has a balanced council; otherwise, it does not. The proof of this assertion is easy to establish by showing that (1) any flow of value *q* in the network corresponds to a balanced council, and that (2) any balanced council implies a flow of value *q* in the network.

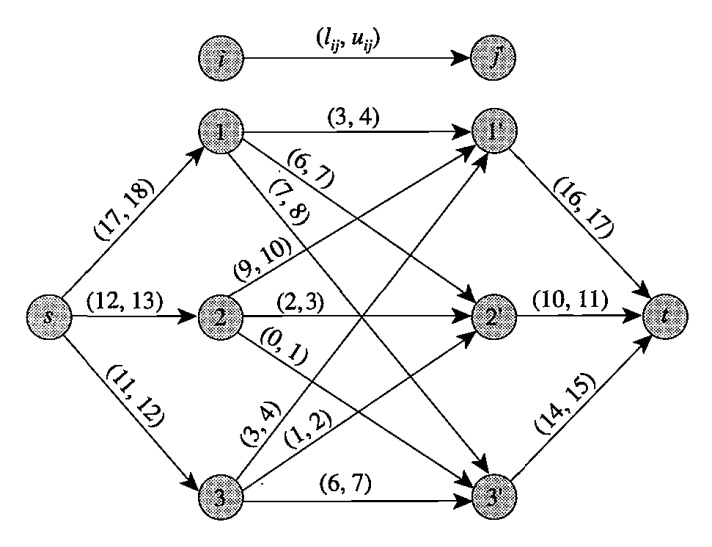
This type of model has applications in several resource assignment settings. For example, suppose that the residents are skilled craftsmen, the club *Ci* is the set of craftsmen with a particular skill, and the political party *Pk* corresponds to a particular seniority class. In this instance, a balanced town council corresponds to an assignment of craftsmen to a union governing board so that every skill class has representation on the board and no seniority class has a dominant representation.

## 4.3 Matrix Rounding Problem

This application is concerned with consistent rounding of the elements, row sums, and column sums of a matrix. We are given a *p* x *q* matrix of *real* numbers *D ={dij},* with row sums *𝛼i* and column sums *𝛽j.* We can round any real number *a* to the next smaller integer or to the next larger integer , and the decision to round up or down is entirely up to us. The matrix rounding problem requires that we round the matrix elements, and the row and column sums of the matrix so that the sum of the rounded elements in each row equals the rounded row sum and the sum of the rounded elements in each column equals the rounded column sum. We refer to such a rounding as a *consistent rounding.*

**

**Table 4.1 Matrix rounding problem**

**

**Figure 4.2 Network for the matrix rounding problem.**

We shall show how we can discover such a rounding scheme, if it exists, by solving a feasible flow problem for a network with nonnegative lower bounds on arc flows. We illustrate our method using the matrix rounding problem shown in Figure 4.2. The figure shows the maximum flow network for this problem. This network contains a node i corresponding to each row *i* and a node *j'* corresponding to each column *j.* Observe that this network contains an arc *(i, j')* for each matrix element *dij,* an arc *(s,* i) for each row sum, and an arc *(j', t)* for each column sum. The lower and the upper bounds of each arc *(i, j')* are and *,* respectively. It is easy to establish a one-to-one correspondence between the consistent roundings of the matrix and feasible flows in the corresponding network. Consequently, we can find a consistent rounding by solving a maximum flow problem on the corresponding network.

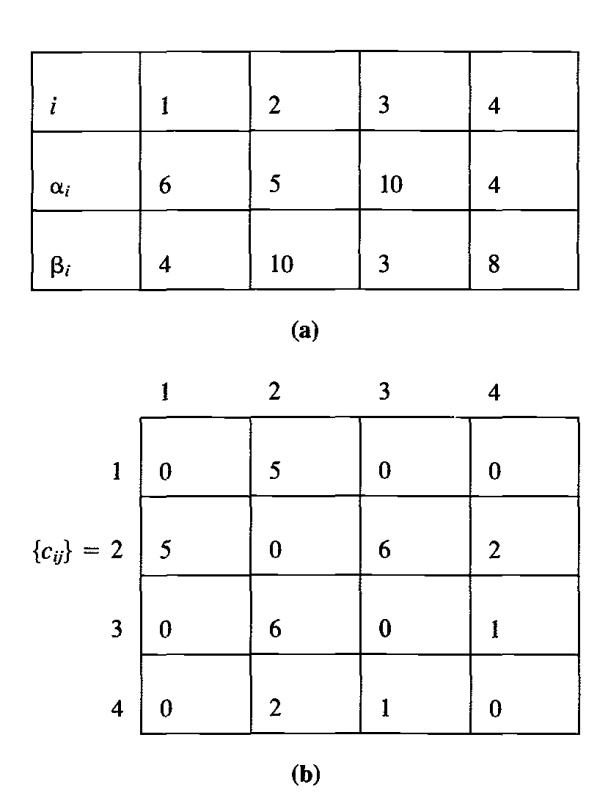
This matrix rounding problem arises in several application contexts. For example, the U.S. Census Bureau uses census information to construct millions of tables for a wide variety of purposes. By law, the bureau has an obligation to protect the source of its information and not disclose statistics that could be attributed to any particular person. We might disguise the information in a table as follows. We round off each entry in the table, including the row and column sums, either up or down to a multiple of a constant *k* (for some suitable value of *k),* so that the entries in the table continue to add to the (rounded) row and column sums, and the overall sum of the entries in the new table adds to a rounded version of the overall sums in the original table. This Census Bureau problem is the same as the matrix rounding problem discussed earlier except that we need to round each element to a multiple of *k* ≥ 1 instead of rounding it to a multiple of 1. We solve this problem by defining the associated network as before, but now defining the lower and upper bounds for any arc with an associated real number a as the greatest multiple of *k* less than or equal to a and the smallest multiple of *k* greater than or equal to a.

## 4.4 Distributed Computing on a Two-Processor Computer

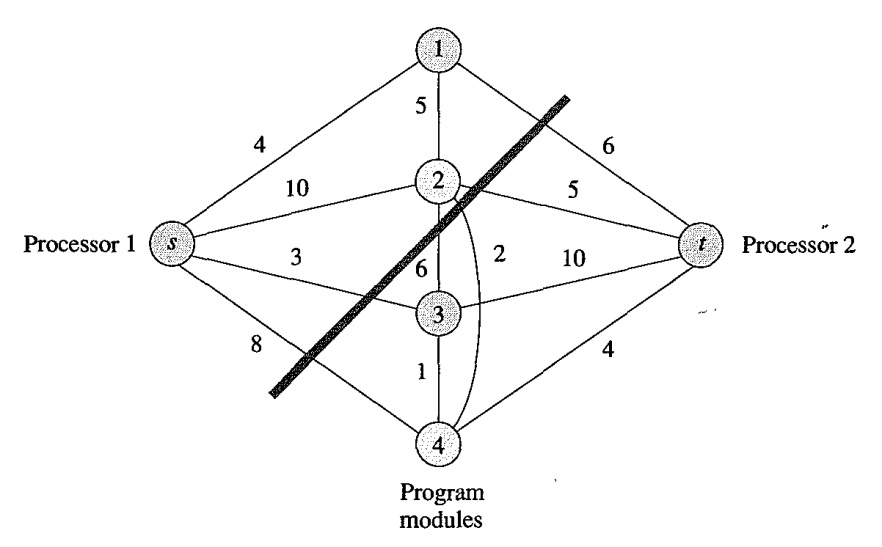
This application concerns assigning different modules (subroutines) of a program to two processors in a way that minimizes the collective costs of inter-processor communication and computation. We consider a computer system with two processors; they need not be identical. We wish to execute a large program on this computer system. Each program contains several modules that interact with each other during the program's execution. The cost of executing each module on the two processes is known in advance and might vary from one processor to the other because of differences in the processors' memory, control, speed, and arithmetic capabilities.

Let αi and βi denote the cost of computation of module *i* on processors 1 and 2, respectively. Assigning different modules to different processors incurs relatively high overhead costs due to interprocessor communication. Let cij denote the inter-processor communication cost if modules *i* and j are assigned to different processors; we do not incur this cost if we assign modules *i* and *j* to the same processor. The cost structure might suggest that we allocate two jobs to different processors-we need to balance this cost against the communication costs that we incur by allocating the jobs to different processors. Therefore, we wish to allocate modules of the program on the two processors so that we minimize the total cost of processing and inter-processor communication.

We formulate this problem as a minimum cut problem on an undirected network as follows. We define a source node s representing processor 1, a sink node *t* representing processor 2, and a node for every module of the program. For every node *i,* other than the source and sink nodes, we include an arc *(s, i)* of capacity βi and an arc *(i, t)* of capacity αi*.* Finally, if module *i* interacts with module *j* during program execution, we include the arc *(i,* j) with a capacity equal to *Cij'* Figures 4.4 and 4.5 give an example of this construction. Figure 4.4 gives the data for this problem, and Figure 4.5 gives the corresponding network.



**Table 4.2 Data for the distributed computing model.**



**Figure 4.3 Network for the distributed computing model.**

We now observe a one-to-one correspondence between *s-t* cuts in the network and assignments of modules to the two processors; moreover, the capacity of a cut equals the cost of the corresponding assignment. To establish this result, let *A1* and *A2* be an assignment of modules to processors 1 and 2, respectively. The cost of this assignment is .The *s-t* cut corresponding to this assignment is *({s}* U A1,*{t}* U *A2).* The approach we used to construct the network implies that this cut contains an arc *(i, t)* for every *i* ∈ *A1* of capacity *𝛼i,* an arc (s, i) for every *i* ∈ *A2* of capacity βi and all arcs (i, j) with *i* ∈ *A1* and j ∈ *A2* with capacity *Cij.* The cost of the assignment *A1* and *A2* equals the capacity of the cut *({s}* U A1,*{t}* U *A2).* Consequently, the minimum *s-t* cut in the network gives the minimum cost assignment of the modules to the two processors.

## Chapter 5

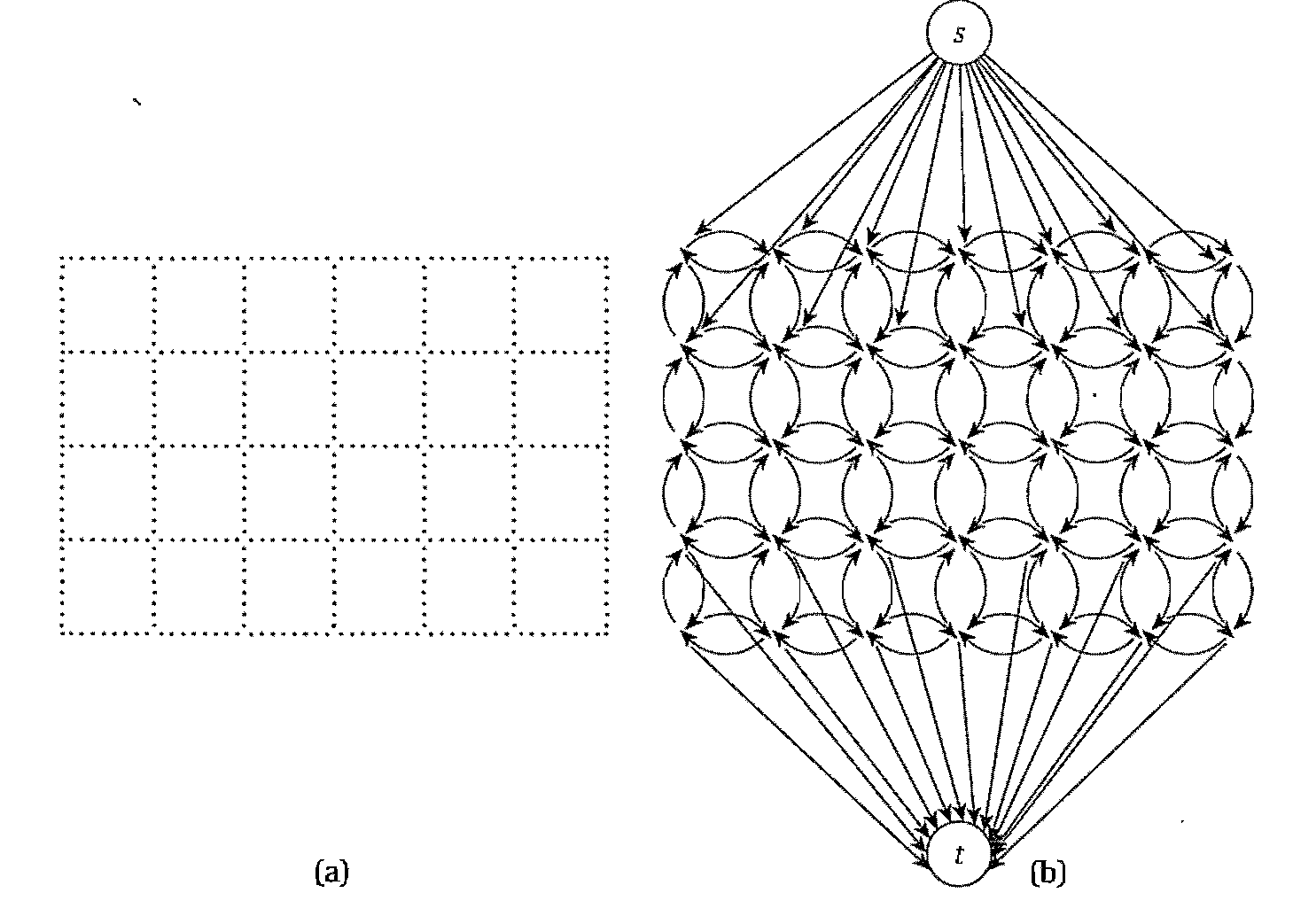
## Image Segmentation

A central problem in image processing is the segmentation of an image into various coherent regions. For example, you may have an image representing a picture of three people standing in front of a complex background scene. Natural but difficult goal is to identify each of the three people as coherent objects in the scene.

## 5.1 Reduction to Network Flow problem

One of the most basic problems to be considered along these lines is that of fore ground/ background segmentation. We wish to label each pixel in an image as belonging to either the foreground of the scene or the background. It turns out that a very natural model here leads to a problem that can be solved efficiently by a minimum cut computation.

Let V be the set of pixels in the underlying image that were analyzing. We will declare certain pairs of pixels to be neighbors, and use E to denote the set of all pairs of neighboring pixels. In this way, we obtain an undirected graph G = (V,E). We will be deliberately vague on what exactly we mean by a "pixel"; or what we mean by the "neighbor" relation. In fact, any graph G will yield an efficiently solvable problem, so we are free to define these notions in any way that we want.



**Figure 5.1: Figure (a) A pixel graph. (b) A sketch of the corresponding flow graph. Not all edges from the source or to the sink are drawn.**

Of course, it is natural to picture the pixels as constituting a grid of dots, and the neighbors of a pixel to be those that are directly adjacent to it in this grid, as shown in Figure.

For each pixel i, we have a likelihood *ai* that it belongs to the foreground, and a likelihood *bi* that it belongs to the background. For our purposes, we will assume that these likelihood values are arbitrary nonnegative numbers provided as part of the problem, and that they specify how desirable it is to have pixel i in the background or foreground. Beyond this, it is not crucial precisely what physical properties of the image they are measuring, or how they were determined. In isolation, we would want to label pixel i as belonging to the foreground if ai > bi , and to the background otherwise. However, decisions that we make about the neighbors of i should affect our decision about i. If many of its neighbors are labeled "background" for example, we should be more inclined to label i as "background" too; this makes the labeling "smoother" by minimizing the amount of foreground/background boundary. Thus, for each pair (i, j) of neighboring pixels, there is a separation penalty pij ≥0 for placing one of i or j in the foreground and the other in the background.

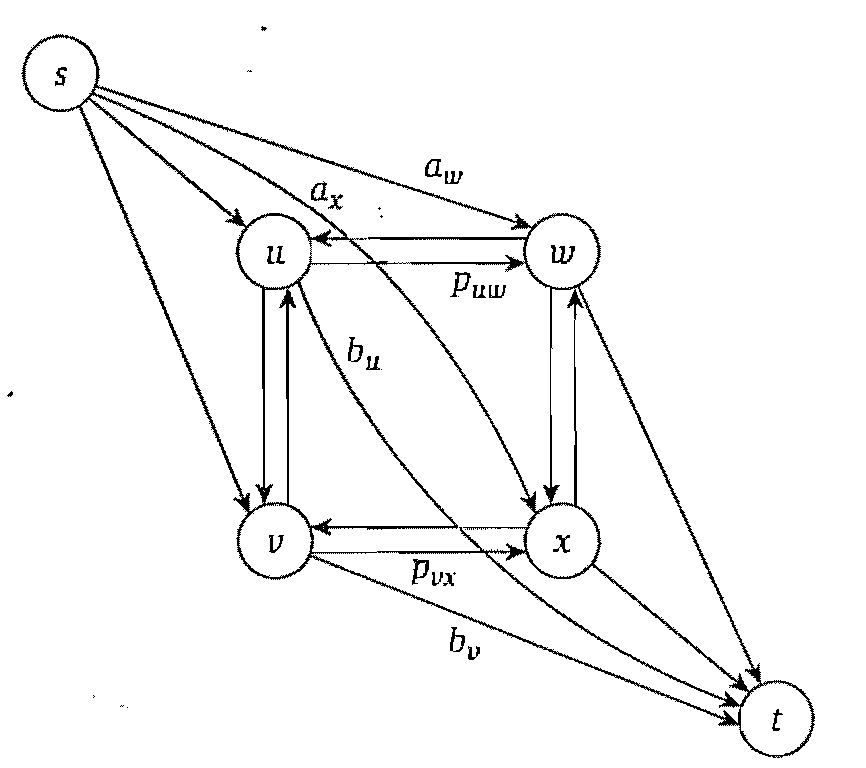
We can now specify our Segmentation Problem precisely, in terms of the likelihood and separation parameters: It is to find a partition of the set of pixels into sets A and B (foreground and background, respectively) so as to maximize

Thus we are rewarded for having high likelihood values and penalized for having neighboring pairs (i, j) with one pixel in A and the other in B. The problem, then, is to compute an optimal labeling - a partition (A,B) that maximizes q(A,B).

**Analyzing and Designing algorithm**

We deal with the fact that our Segmentation Problem is a maximization problem through the following observation. Let . The sum is same as the sum so we can write

Thus we see that the maximization of q(A;B) is the same problem as the minimization of the quantity

As for the missing source and the sink, we work by analogy with our constructions in previous sections: We create a new "super-source" s to represent the foreground, and a new "super-sink" t to represent the background. This also gives us a way to deal with the values ai and bi that reside at the nodes (whereas minimum cuts can only handle numbers associated with edges). Specifically, we will attach each of s and t to every pixel, and use ai and bi to define appropriate capacities on the edges between pixel i and the source and sink respectively.

**Figure 5.2: n s-t cut on a graph constructed from four pixels. Note how the three types of terms in the expression for q (A, B) are captured by the cut.**

Specifically, we define the following flow network G’ = (V’,E’) shown in Figure. The node set V 0 consists of the set V of pixels, together with two additional nodes s and t. For each neighboring pair of pixels i and j, we add directed edges (i,j) and (j, i), each with capacity pij For each pixel i, we add an edge (s, i) with capacity ai and an edge (i, t) with capacity bi. Now, an s-t cut (A,B) corresponds to a partition of the pixels into sets A and B. Let’s consider how the capacity of the cut c(A,B) relates to the quantity q’(A,B) that we are trying to minimize. We can group the edges that cross the cut (A,B) into three natural categories.

1. Edges (s, j), where j ∈ B; this edge contributes aj to the capacity of the cut.

2. Edges (i, t), where i ∈ A; this edge contributes to the capacity of the cut.

3. Edges (i, j) where i ∈ A and j ∈ B; this edge contributes pij to the capacity of the cut.

If we add up the contributions of these three kinds of edges, we get

So everything fits together perfectly. The flow network is set up so that the capacity of the cut (A,B) exactly measures the quantity q’(A,B). The three kinds of edges crossing the cut (A,B), as we have just defined them (edges from the source, edges to the sink, and edges involving neither the source nor the sink), correspond to the three kinds of terms in the expression for q’(A,B). Thus, if we want to minimize q’(A, B) (since we have argued earlier that this is equivalent to maximizing q(A, B)), we just have to find a cut of minimum capacity.

## 5.2 Implementation

**5.2.1 Implementation Details**

In this section we have illustrated our implementation of network flow algorithms to segment image into foreground and background sections. We have applied two different algorithms, Ford-Fulkerson algorithm and Push-Relabel algorithm, to the problem of image segmentation and have compared their space and time complexity.

We use the *imresize()* function to control the size of the image and hence the size of the graph, for instance, scaling to 5% of the original size. Initially we have created a python program graphcut.py which creates graph from the image. It contains the function *build\_bayes\_graph (im; labels; sigma =1e2; kappa = 2)* which is used to label image with values 1 for foreground training data and -1 for background training data. Based on this labeling, a Bayes classifier is trained on the RGB values. Then classification probabilities are computed for each pixel. These are then used as edge weights for the edges going from the source and to the sink. A graph with *nm+2* nodes is created. Note the index of the source and sink, we choose them as the last two to simplify the indexing of the pixel.

Function *cut\_graph (gr, imsize)* solves maximum flow of graph *gr* and return binary labels of the resulting segmentation. Once the graph is built it needs to be cut at the optimal location. The functions *edmond\_karp(s, t)* for Edmond-Karp method from Augmenting path algorithms and *push\_relabel(s, t)* for push-relabel method based on FIFO selection computes the min cut and reformats the output to a binary image of pixel labels. The cut is returned as a dictionary which needs to be copied to an image of segmentation labels. This is done using the .items() method that returns a list of (key, value) pairs. The graph is cut and the result plotted together with an image showing the segmented regions.

To visualize the labeling overlaid on the image we can use the function *contourf()* which fills the regions between contour levels of an image (in this case the label image). The *alpha* variable sets the transparency.

**5.2.2 Results**

We ran the algorithm against 569 x 800 which is down sampled to size 40 x 28. The following figures show the experimental results:

Example 1:



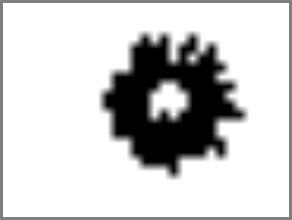
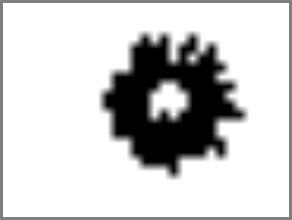
1. **(b) (c)**

**Figure 5.3: (a) Original image (b) Output after applying Edmond-Karp algorithm (b) Output after applying Push-Relabel algorithm**

Example 2:



**Figure 5.4: Original Image**

1. **(b)**

**Figure 5.5 (a) Output after applying Edmond-Karp algorithm to fig 5.4 (b) Output after applying Push-Relabel algorithm to fig 5.4**

In our implementation the running times of the algorithms for image segmentation are as follows:

|  |  |  |
| --- | --- | --- |
| Image Size | Edmond-Karp Algorithm | Push-Relabel Algorithm |
| 24x17 | 2.05288973127 | 1.56860416556 |
| 32x22 | 3.52507610797 | 4.87591196568 |
| 40x28 | 9.32956603157 | 17.740349804 |
| 48x34 | 27.5743096601 | 47.2430401756 |
| 56x39 | 68.5874806655 | 125.171154038 |

**Table 5.1: Experimental Results (running time in seconds)**

## 5.3 Conclusion

In our trials, Ford-Fulkerson algorithm was faster than Push-relabel algorithm. This can be attributed to sparseness of graph. Since E(G) would be of the order of V (G). So their run time is similar in terms of complexity but overhead involved in Push-relabel slows it down comparatively. We conclude that on large sparse graphs, Ford-Fulkerson is likely to be best option, however if graph is dense algorithmic bounds dictate that push-relabel will likely be faster, since its runtime is asymptotically independent of number of edges, and only depends on number of nodes.

**Chapter 6**

**Conclusion**

The worst-case bound for the augmentation time (which is O(n2 m)) grows much faster than the worst-case bound for the relabel time (which is O(nm)). We also observe from these figures that as the network density increases for layered networks, the share of the relabel time in the overall time slightly decreases.This behavior is exhibited by the shortest augmenting path algorithm because increasing the network density causes the number of augmentations to increase at a pace faster than the number of relabels, and thus the augmentations will constitute a larger proportion of the representative operation counts.

The capacity scaling algorithm for the maximum flow problem improves the worst-case complexity of the shortest augmenting path algorithm by incorporating capacity scaling. In the capacity scaling algorithm, we used a scale factor of 2. capacity scaling algorithm reduces the augmentation time but increases the relabel time; but overall the time decrease scaling algorithm. In contrast to the worst-case results, the capacity scaling algorithm is slower empirically than the shortest augmenting path algorithm. The capacity scaling algorithm saves on the augmentation time, but increases the relabel time; overall, more time is spent.

In several previous studies, researchers have investigated dynamic tree implementations of Dinic’s and preflow-push algorithms. The concensus conclusion of these studies is that dynamic tree implementations are slower than the original implementations. In general, dynamic trees decrease the time for nonsaturating pushes and augmentations at the expense of increasing the time for relabels. Moreover, the dynamic tree data structure is quite intricate, and involves a large constant overhead.

As was pointed out before, it is intuitively clear that the usage of current edges decreases the number of nonsaturating pushes. However, in terms of the worst-case performance, we proved that no matter how you choose the edge to push you will not get a worse bound than on the number of nonsaturating pushes.

We believe that this result encourages the attempts to get a better running time by using preflow-push algorithms, because this result provides flexibility in choosing which edge to push along.

Consider for instance the results of Cheriyan et al. , they start the execution of their algorithm with edges that have relatively large capacities and then add the other edges in the order of decreasing capacity as the execution progresses. Using their algorithm on dense graphs, they were able to get an *o(mn)-*time bound on running time. We believe that using more sophisticated rules for choosing the next edge, it might be possible to improve the bound on the number of pushes for preflow-push algorithms. It is evident that the highest-label preflow-push algorithm is fastest among the algorithms tested by us. The FIFO preflow-push algorithm is the second fastest algorithm for our tests. It is 1.5 to 2 times slower than the highest-label preflow-push algorithm. The performance of the stack-scaling algorithm is also attractive; it is about twice slower than the highest-label preflow-push algorithm. We find that the previous two best algorithms, namely Dinic’s and Karzanov’s algorithms, are substantially slower than the recently developed preflow-push algorithms. For large size grid networks, these two algorithms are about 7 to 10 times slower than the highest-label preflow-push algorithms is even more attractive.

|  |  |
| --- | --- |
| **Algorithm** | **Complexity** |
| Ford-Fulkerson | O(E |*f*|) |
| Edmond-Karp | O(VE2) |
| Generic Push-Relabel | O(V2E) |
| FIFO Push-Relabel | O(V3) |
| Highest Label Push-Relabel | O(V3) |
| Dynamic Tree Algorithm | O(VE log V) |
| Capacity Scaling | O(E2log U) |
| Excess Scaling | O(VE + V2log U) |

**Table 6.1 Computational complexity of algorithms**

# Bibliography

[1] Andrew V. Goldberg and Robert E. Tarjan, *A new approach to the maximum ﬂow problem*, Journal of the ACM, 1988, VOL 35 921-940

[2] Ahuja, R. K. AND Orlin, J. B. *A* *fast and simple algorithm for the maximum ﬂow problem*. Tech. Rep. 1905-87, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Mass., 1987.

[3] L. Tuncel. *On the complexity of preﬂow-push algorithms for maximum ﬂow problems.* Algorithmica, 11:353359, 1994.

[4] Cheriyan and Maheshwari. *Analysis of preﬂow push algorithms for maximum network ﬂow*. SIAM Journal of Computing, 18:10571086, 1989.

[5] SLEATOR, D. D., and R. E. TARJAN. 1983. *A data-structre for dynamic trees*, Journal of Computer and Systems Sciences 24, 362-391

[6] D. SLEATOR AND R. TARJAN*, Self-adjusting binary search trees*, J. ACM, 32 (1985), pp. 652-686.

[7] FORD, L. R. Jr., and D. R. FULKERSON. 1956. *Maximum ﬂow through a network.* Canadian Journal of Mathematics 8, 399-404.

[8] EDMONDS, J., and R. M. KARP. 1972. *Theoretical improvements in algorithmic efﬁciency for network ﬂow problems*. Journal of ACM 19, 248-264

[9] A. V. Goldberg, E. Tardos, and R. E. Tarjan, *Network Flow Algorithms, Technical Report* No. 860, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, 198

[10] R. K. AHUJA. T. L. MAGNANTI. AND J. B.ORLIN. *Network Flows, Theory. Algorithms. andApplications*, PrenticeHall, Englewood Cliffs, NJ, 1993