

# **An Approach to Efficient Network Flow Algorithm for Solving Maximum Flow Problem**

**Thesis submitted in partial fulfillment of the requirements for the award  
of degree of**

**Master of Engineering  
in  
Computer Science & Engineering**

**By:  
Chintan Jain  
(800832020)**

**Under the supervision of:  
Dr. Deepak Garg  
Assistant Professor, CSED  
&  
Mrs. Shivani Goel  
Assistant Professor, CSED**



**COMPUTER SCIENCE AND ENGINEERING DEPARTMENT  
THAPAR UNIVERSITY  
PATIALA – 147004**

**JUNE 2010**

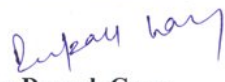
## Certificate

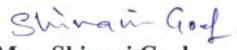
I hereby certify that the work which is being presented in the thesis entitled, “**An approach to efficient Network Flow Algorithm for solving Maximum Flow Problem**”, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Deepak Garg & Mrs. Shivani Goel** and refers other researcher’s works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

  
**Chintan Jain**


This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

  
**Dr. Deepak Garg**  
Assistant Professor  
Computer Science & Engineering Dept  
Thapar University  
Patiala

  
**Mrs. Shivani Goel**  
Assistant Professor  
Computer Science & Engineering Dept  
Thapar University  
Patiala

### Countersigned by

  
**Dr. RAJESH BHATIA**  
Head  
Computer Science & Engineering Department  
Thapar University  
Patiala

  
**Dr. R.K.SHARMA**  
Dean (Academic Affairs)  
Thapar University  
Patiala

## Acknowledgement

---

No volume of words is enough to express my gratitude towards my guide **Dr. Deepak Garg & Mrs. Shivani Goel**, Department of Computer Science & Engineering, Thapar University, Patiala, who have been very concerned and has aided for all the materials essential for the preparation of this thesis report. They have helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to **Dr. Rajesh Bhatia**, Head of Department, Computer Science & Engineering Department and **Mrs. Inderveer Channa**, P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there at the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis work.

Most importantly, I would like to thank my parents and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.



**Chintan Jain**

**(800832020)**

## Abstract

---

Network Flow Problems have always been among the best studied combinatorial optimization problems. These problems are central problems in operations research, computer science, and engineering and they arise in many real world applications. Flow networks are very useful to model real world problems like, current flowing through electrical networks, commodity flowing through pipes and so on. Maximum flow problem is the classical network flow problem. In this problem, the maximum flow which can be moved from the source to the sink is calculated without exceeding the maximum capacity. Once, the maximum flow problem is solved it can be used to solve other network flow problems also. Maximum flow problem is thoroughly studied in this thesis and the general algorithm is explained in detail to solve it. Then other network flow problems like, Minimum Cost Flow, Transshipment, Transportation, and Assignment problems are also briefly explained and shown that how they can be converted into maximum flow problem.

The Ford-Fulkerson algorithm is the general algorithm which can solve all the network flow problems. The improvement of the Ford Fulkerson algorithm is Edmonds-Karp algorithm which uses BFS procedure instead of DFS to find an augmenting path.

Next the modified Edmonds-Karp algorithm is designed to solve the maximum flow problem in efficient manner. One real world problem is taken, it is converted into network flow graph and the new algorithm is implemented to solve the problem. The same problem is solved using Edmonds-Karp algorithm also and both algorithms are compared in terms of different parameters. Finally, it is proved that the modified algorithm performs better in most cases and the new algorithm is implemented in C.

# Table of Contents

---

---

<b>Certificate</b> .....	i
<b>Acknowledgement</b> .....	ii
<b>Abstract</b> .....	iii
<b>Table of Contents</b> .....	iv
<b>List of Figures</b> .....	vi
<b>List of Tables</b> .....	vii
<b>Chapter 1 Introduction</b> .....	1
1.1 Introduction to Network Flow Problems .....	2
1.2 Relationship between Network Flow Problems.....	3
<b>Chapter 2 Literature Review</b> .....	5
2.1 Introduction to Network Flow Concepts.....	5
2.2 Networks with Multiple Sources and Sinks.....	7
2.3 Maximum Flow Problem .....	8
2.3.1 The Ford-Fulkerson Method .....	9
2.3.2 Residual Networks.....	10
2.3.3 Augmenting Paths.....	12
2.3.4 Cuts of Flow Networks .....	12
2.3.5 The Basic Ford-Fulkerson Algorithm.....	14
2.3.5.1 Analysis of Ford-Fulkerson .....	16
2.3.5.2 Limitation of Ford-Fulkerson Algorithm.....	17
2.3.6 The Edmonds-Karp Algorithm .....	17
2.4 Bipartite Matching .....	18
2.4.1 The Maximum Bipartite Matching Problem.....	18
2.4.2 Finding Maximum Bipartite Matching .....	19
2.5 Minimum Cost Flow .....	20

2.6 Transshipment Problem .....	23
2.7 Transportation Problem .....	25
2.8 The Assignment Problem .....	26
<b>Chapter 3 Problem Statement .....</b>	<b>29</b>
3.1 Methodology.....	29
<b>Chapter 4 Design and Implementation .....</b>	<b>30</b>
4.1 Converting the Problem into Graph.....	30
4.2 Solution using Modified Edmonds-Karp Algorithm .....	32
4.3 Solution using Edmonds-Karp Algorithm .....	37
4.4 Discussion and Results.....	38
4.5 Results .....	40
4.6 Implementation .....	41
<b>Chapter 5 Conclusion and Future Scope .....</b>	<b>42</b>
5.1 Future Scope .....	42
<b>References .....</b>	<b>43</b>
<b>List of Publications .....</b>	<b>45</b>
<b>Appendix A .....</b>	<b>46</b>
A.1 Implementation Code.....	46
A.2 Input File .....	49
A.3 Output File .....	50

## List of Figures

---

---

Figure 1.1 Relationship between network flow problems .....	4
Figure 2.1 Sample flow network graph .....	5
Figure 2.2 Maximum-flow problem with multiple-source, multiple-sink .....	8
Figure 2.3(a)-(d) The flow network showing augmenting path and the flow .....	11
Figure 2.4 A cut in the flow network.....	13
Figure 2.5(a)-(d) The execution of the basic Ford-Fulkerson algorithm.....	15
Figure 2.6 A flow network showing limitation of the Ford-Fulkerson algorithm.....	17
Figure 2.7 A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$ .....	19
Figure 2.8 The flow network corresponding to a bipartite graph.....	20
Figure 2.9 Side-by-side computation showing difference when considering the minimum cost flow.....	22
Figure 2.10 Sample Transshipment problem instance converted to Minimum Cost Flow Problem instance.....	24
Figure 2.11 A simple transportation model.....	26
Figure 2.12 The assignment problem cast as a transportation network .....	27
Figure 4.1 The initial flow network corresponding to the problem to be solved .....	31
Figure 4.2 Residual Graph after 0 augmentation & flow graph after 1 <sup>st</sup> augmentation ....	33
Figure 4.3 Residual Graph after 1 <sup>st</sup> augmentation & flow graph after 2 <sup>nd</sup> augmentation .	33
Figure 4.4 Residual Graph after 2 <sup>nd</sup> augmentation & flow graph after 3 <sup>rd</sup> augmentation .	34

Figure 4.5 Residual Graph after 3 <sup>rd</sup> augmentation & flow graph after 4 <sup>th</sup> augmentation ..	34
Figure 4.6 Residual Graph after 4 <sup>th</sup> augmentation & flow graph after 5 <sup>th</sup> augmentation .	35
Figure 4.7 Residual Graph after 5 <sup>th</sup> augmentation & flow graph after 6 <sup>th</sup> augmentation .	36
Figure 4.8 Residual Graph after 6 <sup>th</sup> augmentation .....	36



## List of Tables

---

---

Table 2.1 Person-task relationship for assignment problem .....	27
Table 4.1 Defined capacities of each pipeline between two areas .....	30
Table 4.2 Correspondence between areas and vertices .....	31
Table 4.3 Iteration – augmentation comparison of two algorithms for the problem .....	38
Table 4.4 Comparison of complexities of two algorithms .....	39

# Chapter 1

## Introduction

---

Network flow problems are central problems in operations research, computer science, and engineering and they arise in many real world applications [5]. Everywhere we look in our daily lives, networks are apparent. Electrical and power networks bring lighting and entertainment into our homes. Telephone networks permit us to communicate with each other almost effortlessly within our local communities and across regional and international borders. National highway systems, rail networks, and airline service networks provide us with the means to cross great geographical distances to accomplish our work, to see our loved ones, and to visit new places and enjoy new experiences. Manufacturing and distribution networks give us access to life's essential food stock and to consumer products. And computer networks, such as airline reservation systems, have changed the way we share information and conduct our business and personal lives.

In all of these problem domains, and in many more, we wish to move some entity (electricity, a consumer product, a person or a vehicle, a message) from one point to another in an underlying network, and to do so as efficiently as possible, both to provide good service to the users of the network and to use the underlying (and typically expensive) transmission facilities effectively. In other words, this aim is what this thesis is all about. This thesis is about to learn how to model application settings as mathematical objects known as network flow problems and to study various ways (algorithms) to solve the resulting models [3].

Starting with early work in linear programming and the study of such problems has led to continuing improvements in the efficiency of network flow algorithms. Here the algorithms whose running time is small as a function of the size of the network and the numbers involved (e.g. capacities, nodes, or flows) are taken into consideration. In spite of the long history of this study, many substantial results have been obtained within the last several years. In this thesis some of these recent developments and the ideas behind them are examined [5].

Network flows is a problem domain that lies at the cusp between several fields of inquiry, including applied mathematics, computer science, engineering, management, and operations research. The field has a rich and long tradition, tracing its roots back to the work of Gustav Kirchhoff and other early pioneers of electrical engineering and mechanics who first systematically analyzed electrical circuits. This early work set the foundations of many of the key ideas of network flow theory and established networks (graphs) as useful mathematical objects for representing many physical systems. Much of this early work was descriptive in nature, answering such questions as: If we apply a set of voltages to a given network, what will be the resulting current flow? If we have alternative ways to use a network (i.e., send flow), which alternative will be most cost-effective?

In this thesis, the main classical network flow problems are the *maximum flow problem* and the *minimum-cost flow problem* [3].

## **1.1 Introduction to Network Flow Problems [1]**

There are numerous problems that can be viewed as a network of vertices and edges, with a capacity associated with each edge over which commodities flow. The algorithms found in this thesis are, in many ways, the direct product of the need to solve these specific classes of problems.

### **Maximum Flow**

Given a network that shows the potential capacity over which goods can be shipped between two locations, compute the maximum flow supported by the network.

### **Bipartite Matching**

Given a set of applicants, who have been interviewed for a set of job openings, find a matching that maximizes the number of applicants selected for jobs for which they are qualified.

### **Minimum Cost Flow**

To solve a Minimum Cost Flow problem we need only construct a flow network graph and ensure that it satisfies three criteria —capacity constraint, flow conservation, and skew symmetry—as well as two additional criteria:

-> Supply satisfaction

-> Demand satisfaction

**Transportation**

Determine the most cost-effective way to ship goods from a set of supplying factories to a set of retail stores selling these goods.

**Transshipment**

Determine the most cost-effective way to ship goods from a set of supplying factories to a set of retail stores selling these goods, while potentially using a set of warehouses as intermediate stations.

**Assignment**

Given a set of tasks to be carried out by a set of employees, find an assignment that minimizes the overall expense when different employees may cost different amounts based upon the task to which they are assigned.

**1.2 Relationship Between Network Flow Problems**

One way to explain how these specialized problems are solved is to describe the relationship between network flow problems. Figure 1.1 shows the relationships between these problems in thin, labeled rectangles, with brief descriptions in the larger boxes. A more general instance of a problem is related to a more specific instance of the problem by a directed edge. For example, the Transportation problem is a specialized instance of the Transshipment problem because transportation graphs do not contain intermediate transshipment nodes. Thus a program that solves the Transshipment problem can be immediately applied to solve Transportation problems.

The Ford-Fulkerson algorithm, which solves the Maximum Flow problem, can be immediately applied to solve Bipartite Matching problems, as shown in Figure 1.1 [1]. Upon further reflection, the approach outlined in Ford-Fulkerson can be generalized to solve the more powerful Minimal Cost Flow problem, which enables us to immediately solve the Transshipment, Transportation, and Assignment problems [7].

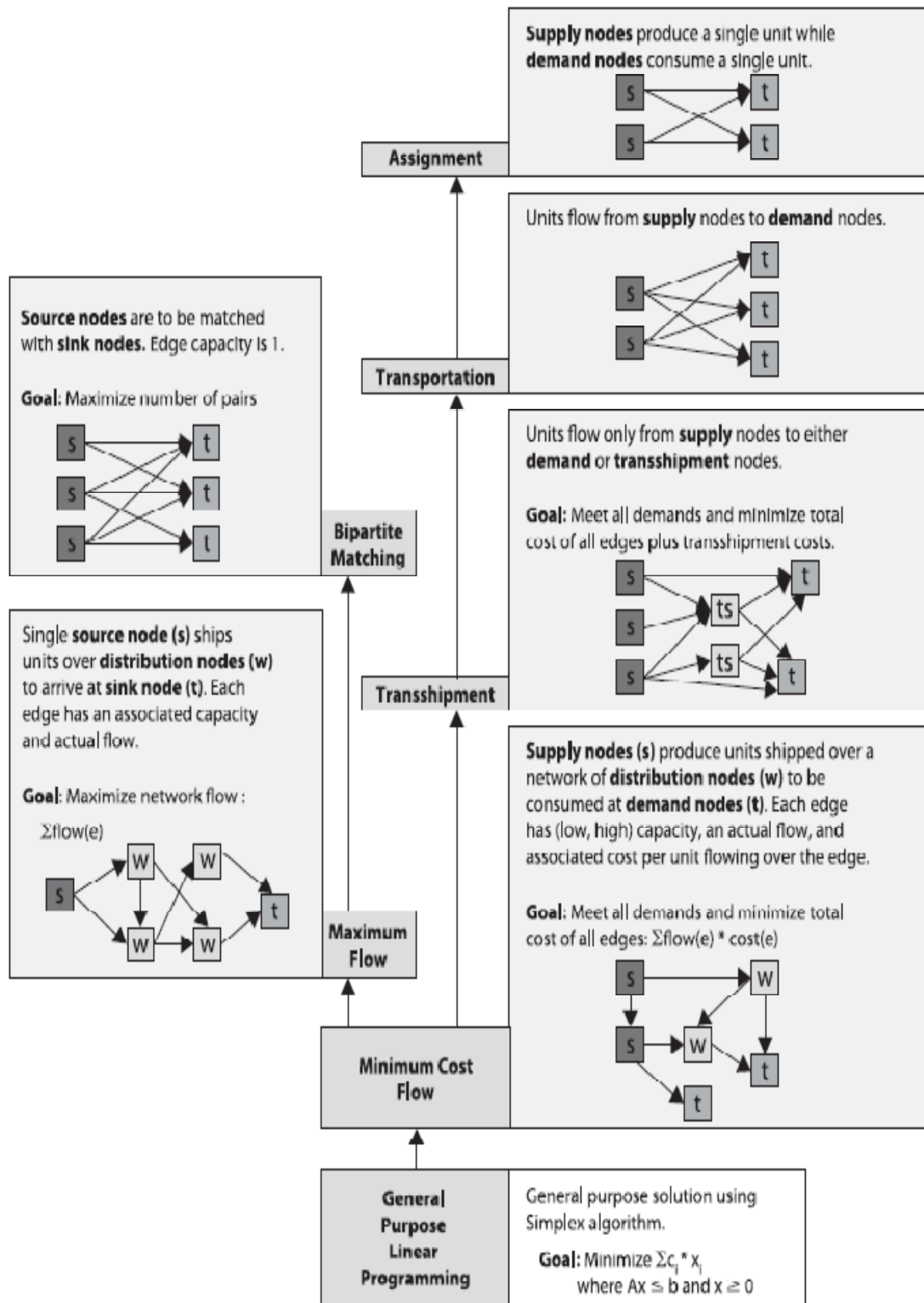


Figure 1.1 Relationship between network flow problems

## 2.1 Introduction to Network Flow Concepts

As depicted in Figure 2.1 [1], the common abstraction that models a flow network is a directed graph  $G = (V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges over these vertices. The graph itself is typically connected (though not every edge need be present). A special source vertex  $s$  belongs to  $V$  produces units of a commodity that flow through the edges of the graph to be consumed by a sink vertex  $t$  belongs to  $V$  (also known as the *target* or *terminus*). A flow network assumes that the supply of units produced is infinite and that the sink vertex can consume all units it receives

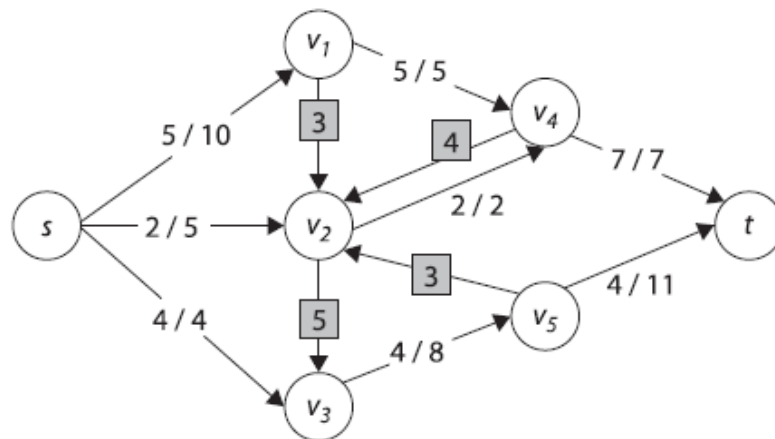


Figure 2.1 Sample flow network graph

Each edge  $(u, v)$  has a flow  $f(u, v)$  that defines the number of units of the commodity that flows from  $u$  to  $v$ . An edge also has a capacity  $c(u, v)$  that constrains the maximum number of units that can flow over that edge. In Figure 2.1, each vertex is numbered (with vertices  $s$  and  $t$  clearly marked) and each edge is labeled as  $f/c$ , showing the flow over that edge and the maximum possible flow. The edge between  $s$  and  $v_1$ , for example, is labeled  $5/10$ , meaning that 5 units flow over that edge, which can sustain a capacity of up to 10. When no units are flowing over an edge (as is the case with the edge between  $v_5$  and  $v_2$ ), only the capacity is shown, outlined in a gray box. The following criteria must be satisfied for any feasible flow  $f$  through a network:

### Capacity constraint

The flow  $f(u, v)$  through an edge cannot be negative and cannot exceed the capacity of the edge  $c(u, v)$ ,  $0 \leq f(u, v) \leq c(u, v)$ . If an edge  $(u, v)$  doesn't exist in the network, then  $c(u, v) = 0$ .

### Flow conservation

Aside from the source vertex  $s$  and sink vertex  $t$ , each vertex  $u$  belongs to  $V$  must satisfy the property that the sum of  $f(v, u)$  for all edges  $(v, u)$  in  $E$  (the flow into  $u$ ) must equal the sum of  $f(u, w)$  for all edges  $(u, w)$  belongs to  $E$  (the flow out of  $u$ ). This property ensures that flow is neither produced nor consumed in the network, except at  $s$  and  $t$ .

### Skew symmetry

For consistency, the quantity  $f(v, u)$  represents the net flow from vertex  $u$  to  $v$ . This means that it must be the case that  $f(u, v) = -f(v, u)$ ; this holds even if both edges  $(u, v)$  and  $(v, u)$  exist in a directed graph (see Figure 2.1).

In the ensuing algorithms a network path referred is a non-cyclic path of unique vertices  $\langle v_1, v_2, \dots, v_n \rangle$  involving  $n-1$  consecutive edges  $(v_i, v_j)$  in  $E$ . In the directed graph shown in Figure 2.1, one possible network path is  $\langle v_3, v_5, v_2, v_4 \rangle$ . In a network path, the direction of the edges can be ignored [1].

The **value** of a flow  $f$  is defined as

$$|f| = \sum_{v \in V} f(s, v),$$

That is, the total flow out of the source. Here, the  $|\cdot|$  notation denotes flow value, not absolute value or cardinality.

Now the three flow properties are briefly explored. The capacity constraint simply says that the flow from one vertex to another must not exceed the given capacity. Skew symmetry is a notational convenience that says that the flow from a vertex  $u$  to a vertex  $v$  is the negative of the flow in the reverse direction. The flow-conservation property says that the total flow out of a vertex other than the source or sink is 0. By skew symmetry, we can rewrite the flow-conservation property as

$\sum_{u \in V} f(u, v) = 0$ , for all  $v \in V - \{s, t\}$ .

That is, the total flow into a vertex is 0.

When neither  $(u, v)$  nor  $(v, u)$  is in  $E$ , there can be no flow between  $u$  and  $v$ , and  $f(u, v) = f(v, u) = 0$ . Now the flow properties deal with flows that are positive. The **total positive flow** entering a vertex  $v$  is defined by

$$\sum_{u \in V, f(u, v) > 0} f(u, v)$$

The total positive flow leaving a vertex is defined symmetrically. The **total net flow** at a vertex is defined to be the total positive flow leaving a vertex minus the total positive flow entering a vertex. One interpretation of the flow-conservation property is that the total positive flow entering a vertex other than the source or sink must equal the total positive flow leaving that vertex. This property, that the total net flow at a vertex must equal 0, is often informally referred to as “flow in equals flow out” [2].

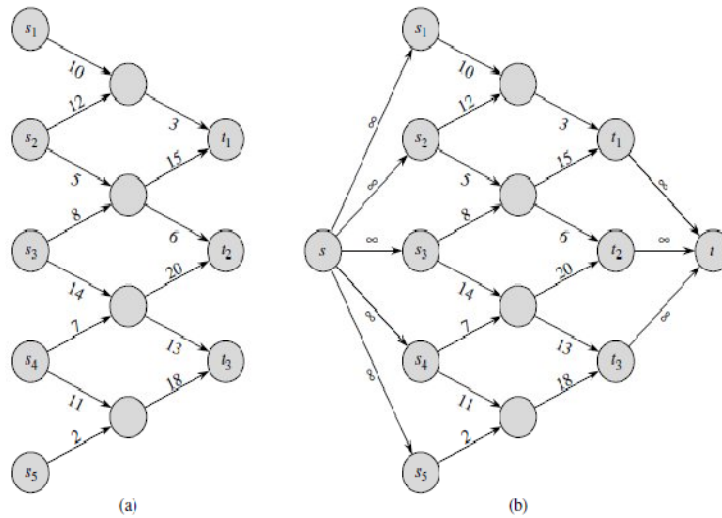
## 2.2 Networks with Multiple Sources and Sinks [2]

A maximum-flow problem may have several sources and sinks, rather than just one of each. The Lucky Puck Company, for example, might actually have a set of  $m$  factories  $\{s_1, s_2, \dots, s_m\}$  and a set of  $n$  warehouses  $\{t_1, t_2, \dots, t_n\}$ , as shown in Figure 2.2(a) [2]. Fortunately, this problem is no harder than ordinary maximum flow.

The problem of determining a maximum flow in a network with multiple sources and multiple sinks is reduced to an ordinary maximum-flow problem. Figure 2.2(b) [2] shows how the network from (a) can be converted to an ordinary flow network with only a single source and a single sink. A **supersource**  $s$  and a directed edge  $(s, s_i)$  is added with capacity  $c(s, s_i) = \infty$  for each  $i = 1, 2, \dots, m$ . A new **supersink**  $t$  and a directed edge  $(t_i, t)$  with capacity  $c(t_i, t) = \infty$  for each  $i = 1, 2, \dots, n$ . Intuitively, any flow in the network in (a) corresponds to a flow in the network in (b), and vice versa. The single source  $s$  simply provides as much flow as desired for the multiple sources



$s_i$ , and single sink  $t$  likewise consumes as much flow as desired for the multiple sinks  $t_i$ .



**Figure 2.2** Converting a multiple-source, multiple-sink maximum-flow problem into a Problem with a single source and a single sink. **(a)** A flow network with five sources  $S = \{s_1, s_2, s_3, s_4, s_5\}$  and three sinks  $T = \{t_1, t_2, t_3\}$ . **(b)** An equivalent single-source, single-sink flow network.

### 2.3 Maximum Flow Problem

The maximum flow problem is a classical optimization problem with many applications. 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 this problem have been studied for over four decades [4]. Recently, significant improvements have been made in theoretical performance of maximum flow algorithms.

A directed graph can be interpreted as a “flow network”. Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate. The “flow” of the material at any point in the system is intuitively the rate at which the material moves. Flow networks are useful to model situations like, liquids flowing through pipes, parts through assembly lines, current through electrical networks, information through communication networks, and so forth.

Each directed edge in a flow network can be thought of as a conduit for the material. Each conduit has a stated capacity, given as a maximum rate at which the material can flow through the conduit, such as 200 gallons of liquid per hour through a pipe or 20 amperes of electrical current through a wire. Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. We call this property “flow conservation,” and it is equivalent to Kirchoff’s Current Law when the material is electrical current.

In the maximum-flow problem, the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraints is calculated. It is one of the simplest problems concerning flow networks and this problem can be solved by efficient algorithms. Moreover, the basic techniques used in maximum-flow algorithms can be adapted to solve other network-flow problems [2].

Let the graph  $G = (V, E)$  be a flow network with source  $s$ , sink  $t$ , and an integer capacity  $c(u, v)$  on each edge  $(u, v) \in E$ . A *flow* in graph  $G$  is a real-valued function  $f: V \times V \rightarrow \mathbb{R}$  that satisfies the following three properties:

**Capacity constraint:** For all  $u, v \in V$ , it required that  $f(u, v) \leq c(u, v)$ .

**Skew symmetry:** For all  $u, v \in V$ , it required that  $f(u, v) = -f(v, u)$ .

**Flow conservation:** For all  $u \in V - \{s, t\}$ , it required that

$$\sum_{v \in V} f(u, v) = 0.$$

The quantity  $f(u, v)$ , which can be positive, zero, or negative, is called the **flow** from vertex  $u$  to vertex  $v$ . The **value** of a flow  $f$  is defined as

$$|f| = \sum_{v \in V} f(s, v)$$

that is, the total flow out of the source. (Here, the  $|\cdot|$  notation denotes flow value, not absolute value or cardinality.) The **maximum flow problem** is to maximize  $|f|$ , that is, to route as much flow as possible from  $s$  to  $t$  [12].

### 2.3.1 The Ford-Fulkerson Method [2]

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. It is called 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, which characterizes the value of a maximum flow in terms of cuts of the flow network. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method is iterative. We start with  $f(u, v) = 0$  for all  $u, v \in V$ , giving an initial flow of value 0. At each iteration, we increase the flow value by finding an “augmenting path,” which we can think of simply as a path from the source  $s$  to the sink  $t$  along which we can send more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

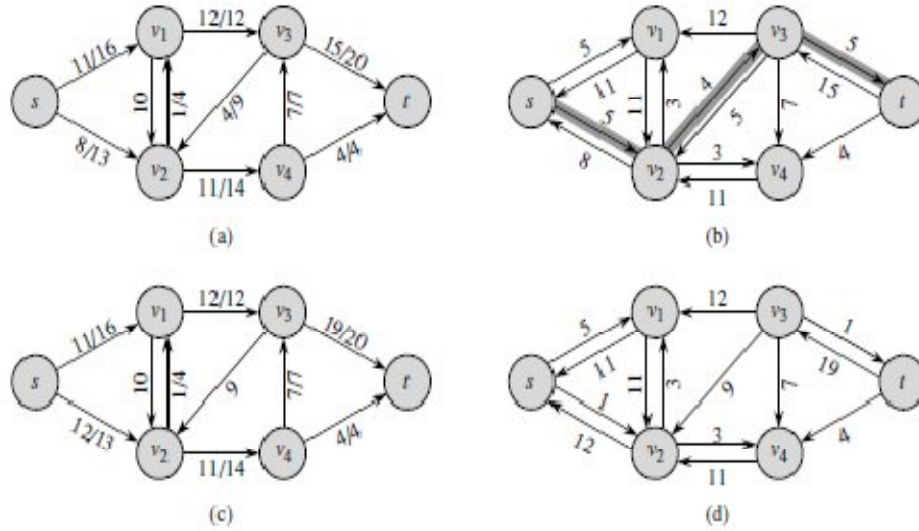
Ford-Fulkerson-Method( $G, s, t$ )

- [1] initialize flow  $f$  to 0
- [2] **while** there exists an augmenting path  $p$
- [3] **do** augment flow  $f$  along  $p$
- [4] **return**  $f$

### 2.3.2 Residual Networks

Intuitively, given a flow network and a flow, the residual network consists of edges that can admit more flow. More formally, Let  $G = (V, E)$  a flow network with source  $s$  and sink  $t$ . Let  $f$  be a flow in  $G$ , and consider a pair of vertices  $u, v \in V$ . The amount of *additional* flow which can be pushed from  $u$  to  $v$  before exceeding the capacity  $c(u, v)$  is the **residual capacity** of  $(u, v)$ , given by

$$c_f(u, v) = c(u, v) - f(u, v)$$



**Figure 2.3** (a) The flow network  $G$  and flow  $f$ . (b) The residual network  $G_f$  with augmenting path  $p$  shaded; its residual capacity is  $c_f(p) = c(v_2, v_3) = 4$ . (c) The flow in  $G$  that results from augmenting along path  $p$  by its residual capacity 4. (d) The residual network induced by the flow in (c).

For example, if  $c(u, v) = 16$  and  $f(u, v) = 11$ , then we can increase  $f(u, v)$  by  $c_f(u, v) = 5$  units before we exceed the capacity constraint on edge  $(u, v)$ . When the flow  $f(u, v)$  is negative, the residual capacity  $c_f(u, v)$  is greater than the capacity  $c(u, v)$ . For example, if  $c(u, v) = 16$  and  $f(u, v) = -4$ , then the residual capacity  $c_f(u, v)$  is 20. This situation can be interpreted as follows. There is a flow of 4 units from  $v$  to  $u$ , which we can cancel by pushing a flow of 4 units from  $u$  to  $v$ . We can then push another 16 units from  $u$  to  $v$  before violating the capacity constraint on edge  $(u, v)$ . We have thus pushed an additional 20 units of flow, starting with a flow  $f(u, v) = -4$ , before reaching the capacity constraint.

Given a flow network  $G = (V, E)$  and a flow  $f$ , the residual network of  $G$  induced by  $f$  is  $G_f = (V, E_f)$ , where

$$E_f = \{(u, v) \in V \times V: c_f(u, v) > 0\}.$$

That is, as promised above, each edge of the residual network, or residual edge, can admit a flow that is greater than 0. Figure 2.3(b) [2] shows the corresponding residual network  $G_f$ .

The edges in  $E_f$  are either edges in  $E$  or their reversals. If  $f(u, v) < c(u, v)$  for an edge  $(u, v) \in E$ , then  $c_f(u, v) = c(u, v) - f(u, v) > 0$  and  $(u, v) \in E_f$ . If  $f(u, v) > 0$  for an edge  $(u, v) \in E$ , then  $f(v, u) < 0$ . In this case,  $c_f(v, u) = c(v, u) - f(v, u) > 0$ , and so  $(v, u) \in E_f$ . If neither  $(u, v)$  nor  $(v, u)$  appears in the original network, then  $c(u, v) = c(v, u) = 0$ ,  $f(u, v) = f(v, u) = 0$ , and  $c_f(u, v) = c_f(v, u) = 0$ . It can be concluded that an edge  $(u, v)$  can appear in a residual network only if at least one of  $(u, v)$  and  $(v, u)$  appears in the original network, and thus  $|E_f| \leq 2|E|$ .

Observe that the residual network  $G_f$  is itself a flow network with capacities given by  $c_f$ . The following lemma shows how a flow in a residual network relates to a flow in the original flow network.

### 2.3.3 Augmenting Paths [2]

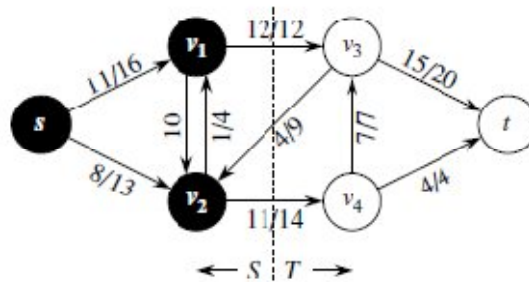
Given a flow network  $G = (V, E)$  and a flow  $f$ , an **augmenting path**  $p$  is a simple path from  $s$  to  $t$  in the residual network  $G_f$ . By the definition of the residual network, each edge  $(u, v)$  on an augmenting path admits some additional positive flow from  $u$  to  $v$  without violating the capacity constraint on the edge. The shaded path in Figure 2.3(b) is an augmenting path. Treating the residual network  $G_f$  in the figure as a flow network, we can increase the flow through each edge of this path by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is  $c_f(v_2, v_3) = 4$ . We call the maximum amount by which we can increase the flow on each edge in an augmenting path  $p$  the **residual capacity** of  $p$ , given by

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\}$$

Figure 2.3(c) shows the result of adding  $f_p$  in Figure 2.3(b) to  $f$  from Figure 2.3(a).

### 2.3.4 Cuts of Flow Networks [2]

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until a maximum flow has been found. The max-flow min-cut theorem, which we shall prove shortly, tells us that a flow is maximum if and only if its residual network contains no augmenting path. To prove this theorem, though, we must first explore the notion of a cut of a flow network.



**Figure 2.4** A cut  $(S, T)$  in the flow network of Figure 2.3 (a), where  $S = \{s, v_1, v_2\}$  and  $T = \{v_3, v_4, t\}$ . The vertices in  $S$  are black, and the vertices in  $T$  are white. The net flow across  $(S, T)$  is  $f(S, T) = 19$ , and the capacity is  $c(S, T) = 26$ .

A **cut**  $(S, T)$  of flow network  $G = (V, E)$  is a partition of  $V$  into  $S$  and  $T = V - S$  such that  $s \in S$  and  $t \in T$ . If  $f$  is a flow, then the **net flow** across the cut  $(S, T)$  is defined to be  $f(S, T)$ . The **capacity** of the cut  $(S, T)$  is  $c(S, T)$ . A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network. Figure 2.4 shows the cut  $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$  in the flow network of Figure 2.3(a).

The net flow across this cut is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19 \end{aligned}$$

and its capacity is

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 \end{aligned}$$

Observe that the net flow across a cut can include negative flows between vertices, but that the capacity of a cut is composed entirely of nonnegative values. In other words, the net flow across a cut  $(S, T)$  consists of positive flows in both directions; positive flow from  $S$  to  $T$  is added while positive flow from  $T$  to  $S$  is subtracted. On the other hand, the capacity of a cut  $(S, T)$  is computed only from edges going from  $S$  to  $T$ . Edges going from  $T$  to  $S$  are not included in the computation of  $c(S, T)$ .

The maximum flow in a network is bounded above by the capacity of a minimum cut of the network. The important max-flow min-cut theorem, which will be now stated and proved, says that the value of a maximum flow is in fact equal to the capacity of a minimum cut.

### Theorem 1 (Max-flow min-cut theorem)

If  $f$  is a flow in a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ , then the following conditions are equivalent:

1.  $f$  is a maximum flow in  $G$ .
2. The residual network  $G_f$  contains no augmenting paths.
3.  $|f| = c(S, T)$  for some cut  $(S, T)$  of  $G$  [10].

### 2.3.5 The Basic Ford-Fulkerson Algorithm

In each iteration of the Ford-Fulkerson method, we find *some* augmenting path  $p$  and increase the flow  $f$  on each edge of  $p$  by the residual capacity  $c_f(p)$ . The following implementation of the method computes the maximum flow in a graph  $G = (V, E)$  by updating the flow  $f[u, v]$  between each pair  $u, v$  of vertices that are connected by an edge. If  $u$  and  $v$  are not connected by an edge in either direction, we assume implicitly that  $f[u, v] = 0$ . The capacities  $c(u, v)$  are assumed to be given along with the graph, and  $c(u, v) = 0$  if  $(u, v) \notin E$ . The residual capacity  $c_f(u, v)$  is computed in accordance with the formula given in section 2.3.2. The expression  $c_f(p)$  in the code is actually just a temporary variable that stores the residual capacity of the path  $p$ .

Ford-Fulkerson( $G, s, t$ )

[1] **for** each edge  $(u, v) \in E[G]$

[2] **do**  $f[u, v] \leftarrow 0$

[3]  $f[v, u] \leftarrow 0$

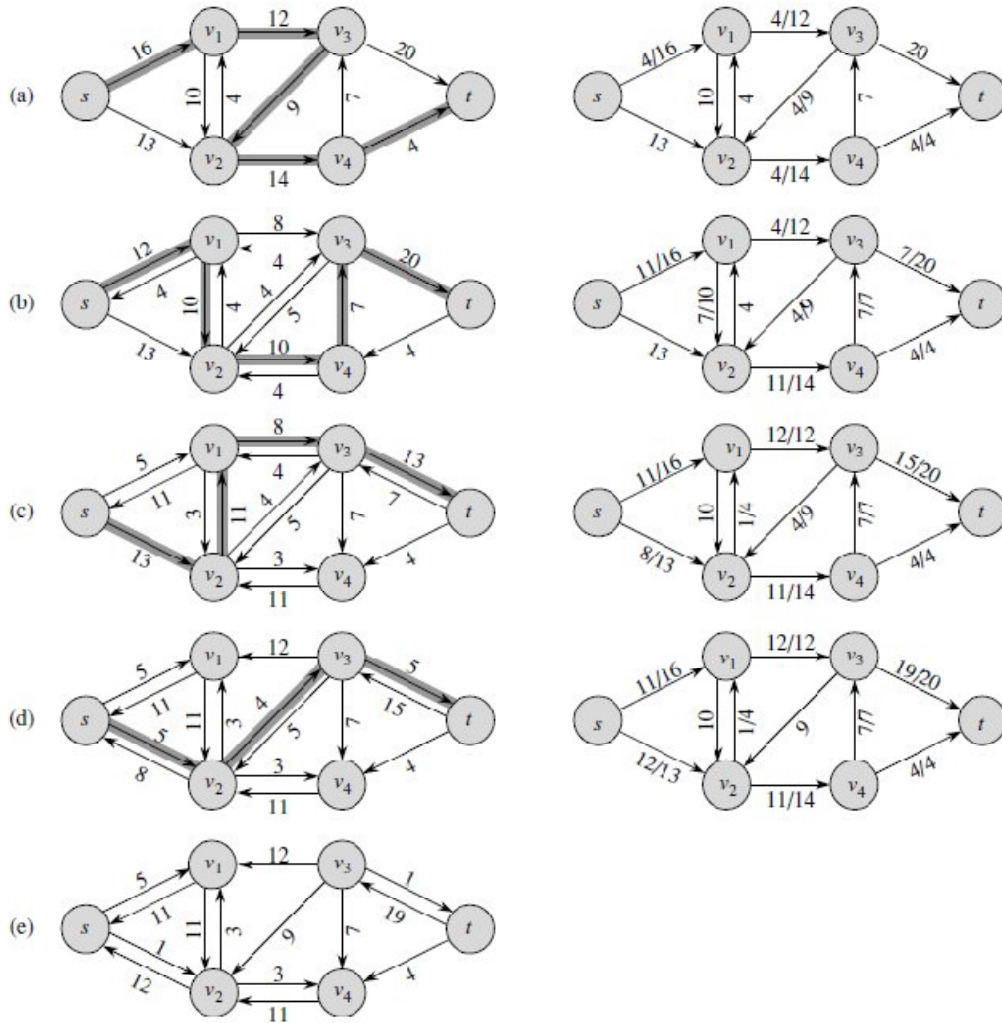
[4] **while** there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$

[5] **do**  $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$

[6] **for** each edge  $(u, v)$  in  $p$

[7] **do**  $f[u, v] \leftarrow f[u, v] + c_f(p)$

[8]  $f[v, u] \leftarrow -f[u, v]$



**Figure 2.5** The execution of the basic Ford-Fulkerson algorithm. (a)–(d) Successive iterations of the while loop. The left side of each part shows the residual network  $G_f$  from line 4 with a shaded augmenting path  $p$ . The right side of each part shows the new flow  $f$  that results from adding  $f_p$  to  $f$ . The residual network in (a) is the input network  $G$ . (e) The residual network at the last while loop test. It has no augmenting paths, and the flow  $f$  shown in (d) is therefore a maximum flow.

The Ford-Fulkerson algorithm simply expands on the Ford-Fulkerson-Method pseudo code given earlier. Figure 2.5 shows the result of each iteration in a sample run. Lines 1–3 initialize the flow  $f$  to 0. The **while** loop of lines 4–8 repeatedly finds an augmenting path  $p$  in  $G_f$  and augments flow  $f$  along  $p$  by the residual capacity  $c_f(p)$ . When no augmenting paths exist, the flow  $f$  is a maximum flow.



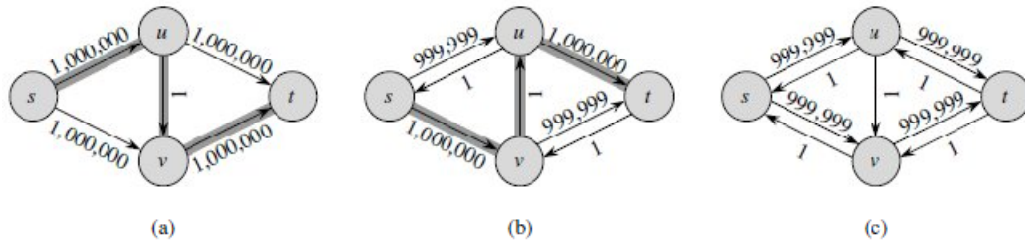
### 2.3.5.1 Analysis of Ford-Fulkerson

The running time of Ford-Fulkerson depends on how the augmenting path  $p$  in line 4 is determined. If it is chosen poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value. If the augmenting path is chosen by using a breadth-first search, however, the algorithm runs in polynomial time. Before proving this result, however, a simple bound is obtained for the case in which the augmenting path is chosen arbitrarily and all capacities are integers [14].

Most often in practice, the maximum-flow problem arises with integral capacities. If the capacities are rational numbers, an appropriate scaling transformation can be used to make them all integral. Under this assumption, a straightforward implementation of Ford-Fulkerson runs in time  $O(E |f^*|)$ , where  $f^*$  is the maximum flow found by the algorithm. The analysis is as follows. Lines 1–3 take time  $\Theta(E)$ . The **while** loop of lines 4–8 is executed at most  $|f^*|$  times, since the flow value increases by at least one unit in each iteration.

The work done within the **while** loop can be made efficient if the data structure used to implement the network  $G = (V, E)$  is efficiently managed. Let us assume that we keep a data structure corresponding to a directed graph  $G' = (V, E')$ , where  $E' = \{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$ . Edges in the network  $G$  are also edges in  $G'$ , and it is therefore a simple matter to maintain capacities and flows in this data structure. Given a flow  $f$  on  $G$ , the edges in the residual network  $G_f$  consist of all edges  $(u, v)$  of  $G'$  such that  $c(u, v) - f[u, v] \neq 0$ . The time to find a path in a residual network is therefore  $O(V + E') = O(E)$  if we use either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes  $O(E)$  time, making the total running time of Ford-Fulkerson  $O(E |f^*|)$ .

### 2.3.5.2 Limitation of Ford-Fulkerson Algorithm



**Figure 2.6** (a) A flow network for which Ford-Fulkerson can take  $\Theta(E|f^*|)$  time, where  $f^*$  is a maximum flow, shown here with  $|f^*| = 2,000,000$ . An augmenting path with residual capacity 1 is shown. (b) The resulting residual network. Another augmenting path with residual capacity 1 is shown. (c) The resulting residual network.

When the capacities are integral and the optimal flow value  $|f^*|$  is small, the running time of the Ford-Fulkerson algorithm is good. Figure 2.6(a) shows an example of what can happen on a simple flow network for which  $|f^*|$  is large. A maximum flow in this network has value 2,000,000: 1,000,000 units of flow traverse the path  $s \rightarrow u \rightarrow t$ , and another 1,000,000 units traverse the path  $s \rightarrow v \rightarrow t$ . If the first augmenting path found by Ford-Fulkerson is  $s \rightarrow u \rightarrow v \rightarrow t$ , shown in Figure 2.6(a), the flow has value 1 after the first iteration. The resulting residual network is shown in Figure 2.6(b). If the second iteration finds the augmenting path  $s \rightarrow v \rightarrow u \rightarrow t$ , as shown in Figure 2.6(b), the flow then has value 2. Figure 2.6(c) shows the resulting residual network. It can be continued, choosing the augmenting path  $s \rightarrow u \rightarrow v \rightarrow t$  in the odd-numbered iterations and the augmenting path  $s \rightarrow v \rightarrow u \rightarrow t$  in the even-numbered iterations. So, it would perform a total of 2,000,000 augmentations, increasing the flow value by only 1 unit in each [8].

### 2.3.6 The Edmonds-Karp Algorithm

The bound on Ford-Fulkerson can be improved if we implement the computation of the augmenting path  $p$  in line 4 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 (weight). The Ford-Fulkerson method so implemented is called the **Edmonds-Karp algorithm**. Now it is proved that the Edmonds-Karp algorithm runs in  $O(VE^2)$  time. The analysis depends on the distances to vertices in the residual network  $G_f$  [15].

### **Theorem 2**

If the Edmonds-Karp algorithm is run on a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ , then the total number of flow augmentations performed by the algorithm is  $O(VE)$ .

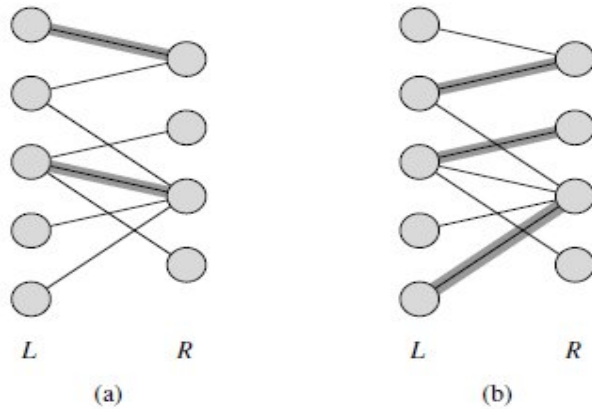
Since each iteration of Ford-Fulkerson can be implemented in  $O(E)$  time when the augmenting path is found by breadth-first search, the total running time of the Edmonds-Karp algorithm is  $O(VE^2)$  [6].

## **2.4 Bipartite Matching [2]**

Some combinatorial problems can easily be cast as maximum-flow problems. The multiple-source, multiple-sink maximum-flow problem gave us one example. There are other combinatorial problems that seem on the surface to have little to do with flow networks, but can in fact be reduced to maximum-flow problems. This section presents one such problem: finding a maximum matching in a bipartite graph. In order to solve the problem, we shall take advantage of an integrality property provided by the Ford-Fulkerson method. We shall see that the Ford-Fulkerson method can be made to solve the maximum bipartite-matching problem on a graph  $G = (V, E)$  in  $O(VE)$  time.

### **2.4.1 The Maximum Bipartite Matching Problem**

Given an undirected graph  $G = (V, E)$ , a **matching** is a subset of edges  $M \subseteq E$  such that for all vertices  $v \in V$ , at most one edge of  $M$  is incident on  $v$ . A vertex  $v \in V$  is **matched** by matching  $M$  if some edge in  $M$  is incident on  $v$ ; otherwise,  $v$  is **unmatched**. A **maximum matching** is a matching of maximum cardinality, that is, a matching  $M$  such that for any matching  $M'$ ,  $|M| \geq |M'|$ .



**Figure 2.7** A bipartite graph  $G = (V, E)$  with vertex partition  $V = L \cup R$ . A matching with cardinality 2. (b) A maximum matching with cardinality 3.

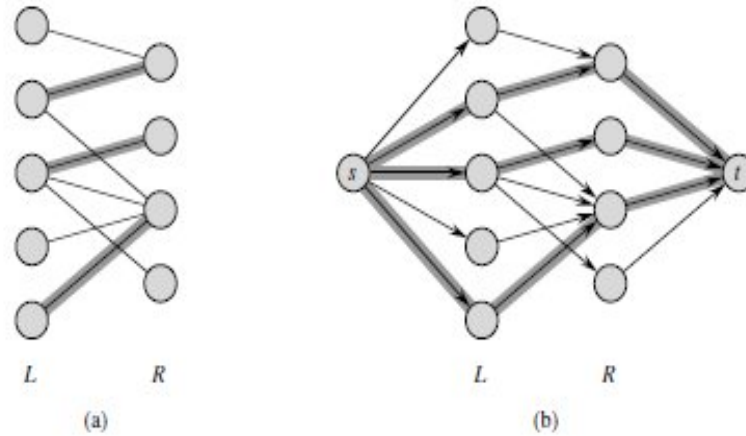
In this section, attention is restricted to finding max. matchings in bipartite graphs. It is assumed that vertex set can be partitioned into  $V = L \cup R$ , where  $L$  and  $R$  are disjoint and all edges in  $E$  go between  $L$  and  $R$ . It is further assume that every vertex in  $V$  has at least one incident edge. Figure 2.7 illustrates the notion of a matching.

The problem of finding a maximum matching in a bipartite graph has many practical applications. As an example, consider matching a set  $L$  of machines with a set  $R$  of tasks to be performed simultaneously. The presence of edge  $(u, v)$  in  $E$  is taken to mean that a particular machine  $u \in L$  is capable of performing a particular task  $v \in R$ . A maximum matching provides work for as many machines as possible.

### 2.4.2 Finding Maximum Bipartite Matching

The Ford-Fulkerson method can be used to find a maximum matching in an undirected bipartite graph  $G = (V, E)$  in time polynomial in  $|V|$  and  $|E|$ . The trick is to construct a flow network in which flows correspond to matchings, as shown in Figure 2.8. The **corresponding flow network**  $G' = (V', E')$  for the bipartite graph  $G$  is defined as follows. Let the source  $s$  and sink  $t$  be new vertices not in  $V$ , and let  $V' = V \cup \{s, t\}$ . If the vertex partition of  $G$  is  $V = L \cup R$ , the directed edges of  $G'$  are the edges of  $E$ , directed from  $L$  to  $R$ , along with  $V$  new edges:

$$\begin{aligned}
 E' = & \{(s, u): u \in L\} \\
 & \cup \{(u, v): u \in L, v \in R, \text{ and } (u, v) \in E\} \\
 & \cup \{(v, t): v \in R\}.
 \end{aligned}$$



**Figure 2.8** The flow network corresponding to a bipartite graph. (a) The bipartite graph  $G = (V, E)$  with vertex partition  $V = L \cup R$  from Figure 26.7. A maximum matching is shown by shaded edges. (b) The corresponding flow network  $G'$  with a maximum flow shown. Each edge has unit capacity. Shaded edges have a flow of 1, and all other edges carry no flow. The shaded edges from  $L$  to  $R$  correspond to those in a maximum matching of the bipartite graph.

To complete the construction, unit capacity is assigned to each edge in  $E'$ . Since each vertex in  $V$  has at least one incident edge,  $|E| \geq |V|/2$ . Thus,  $|E| \leq |E'| = |E| + |V| \leq 3|E|$ , and so  $|E'| = \Theta(|E|)$  [9].

## 2.5 Minimum Cost Flow [1]

To solve a Minimum Cost Flow problem need only construct a flow network graph and ensure that it satisfies the criteria discussed earlier—capacity constraint, flow conservation, and skew symmetry—as well as two additional criteria:

### Supply satisfaction

For each source vertex  $s_i \in S$ , the sum of  $f(s_i, v)$  for all edges  $(s_i, v) \in E$  (the flow out of  $s_i$ ) minus the sum of  $f(u, s_i)$  for all edges  $(u, s_i) \in E$  (the flow into  $s_i$ ) must be less than or equal to  $\text{sup}(s_i)$ . That is, the supply  $\text{sup}(s_i)$  at each source vertex is a firm upper bound on the net flow from that vertex.

### Demand satisfaction

For each sink vertex  $t_j \in T$ , the sum of  $f(u, t_j)$  for all edges  $(u, t_j) \in E$  (the flow into  $t_j$ ) minus the sum of  $f(t_j, v)$  for all edges  $(t_j, v) \in E$  (the flow out of  $t_j$ ) must be less than or equal to  $\text{dem}(t_j)$ . That is, the  $\text{dem}(t_j)$  at each target vertex is a firm upper bound on the net flow into that vertex.

To simplify the algorithmic solution, the flow network graph is further constrained to have a single source vertex and sink vertex. This can be easily accomplished by taking an existing flow network graph with any number of source and sink vertices and adding two new vertices. First, add a new vertex (which we refer to as  $s_0$ ) to be the source vertex for the flow network graph, and add edges  $(s_0, s_i)$  for all  $s_i \in S$  whose capacity  $c(s_0, s_i) = \text{sup}(s_i)$  and whose cost  $d(s_0, s_i) = 0$ . Second, add a new vertex (which is often referred to as  $t_{gt}$ , for target) to be the sink vertex for the flow network graph, and add edges  $(t_j, t_{gt})$  for all  $t_j \in T$  whose capacity  $c(t_j, t_{gt}) = \text{dem}(t_j)$  and whose cost  $d(t_0, t_j) = 0$ . Adding these vertices and edges does not increase the cost of the network flow, nor do they reduce or increase the final computed flow over the network.

The supplies  $\text{sup}(s_i)$ , demands  $\text{dem}(t_j)$ , and capacities  $c(u, v)$  are all greater than 0. The shipping cost  $d(u, v)$  associated with each edge may be greater than or equal to zero. When resulting flow is computed, all  $f(u, v)$  values will be greater than or equal to 0.

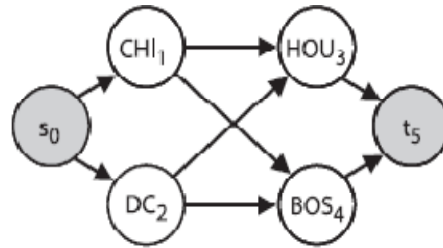
Figure 2.9 [1] shows the side-by-side computation on a small example comparing a straightforward Maximum Flow computation with a Minimum Cost Flow computation. The result, at the bottom of the figure, is the maximum flow found by each approach.

In this example, there are two factories in Chicago ( $v_1$ ) and Washington, D.C. ( $v_2$ ) that can each produce 300 widgets daily. The shipping manager in charge of two factories must ensure that two customers in Houston ( $v_3$ ) and Boston ( $v_4$ ) each receive 300 widgets a day. Manager has several options for shipping, as shown in the figure. For example, between Washington, D.C. and Houston, he may ship up to 280 widgets daily at \$4 per widget, but the cost increases to \$6 per widget if he ship from Washington, D.C. to Boston (although he can then send up to 350 widgets per day along that route).

### Shipping Info

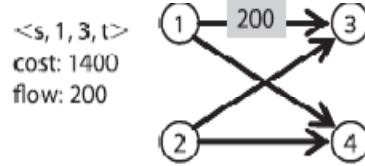
	HOU <sub>3</sub>	BOS <sub>4</sub>
CHI <sub>1</sub>	200 @ 7	200 @ 6
DC <sub>2</sub>	280 @ 4	350 @ 6

### Flow Network

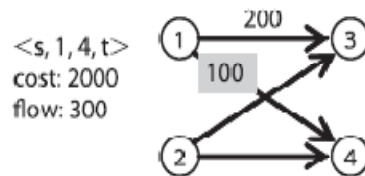


### Maximum Flow

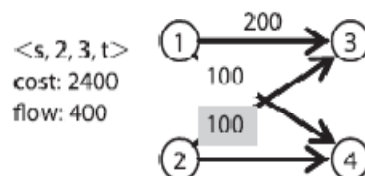
Iteration 1



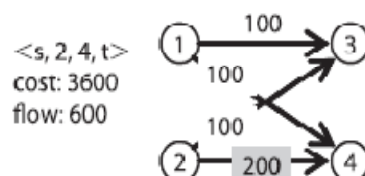
Iteration 2



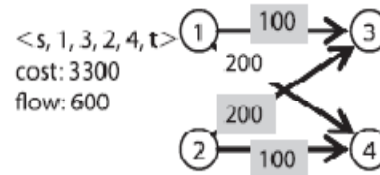
Iteration 3



Iteration 4



Iteration 5



### Minimum Cost Flow

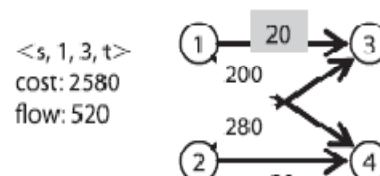
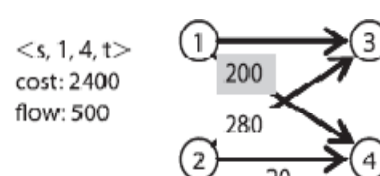
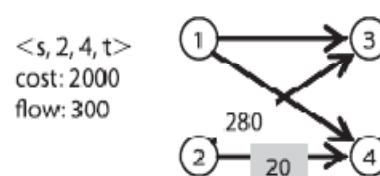
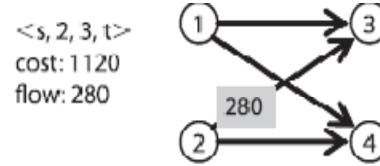


Figure 2.9 Side-by-side computation showing difference when considering the minimum cost flow

It may not even be clear that Ford-Fulkerson can be used to solve this problem, but note that we can create a graph  $G$  with a new source vertex  $s_0$  that connects to the two factory nodes ( $v_1$  and  $v_2$ ) and the two customers ( $v_3$  and  $v_4$ ) connect to a new sink vertex  $t_5$ . On the left hand side of Figure 2.9, the Edmonds-Karp variation is executed to demonstrate that we can meet all of our customer needs as requested, at the total daily shipping cost of \$3,600. To save space, the source and sink vertices  $s_0$  and  $t_5$  are omitted. During each of the four iterations by Ford-Fulkerson, the impact of the augmented path is shown (when an iteration updates the flow for an edge, the flow value is shaded gray).

Is this the lowest cost we can achieve? The right-hand side of Figure 2.9 shows the execution of Ford Fulkerson using short path as the search strategy. Note how the first augmented path found takes advantage of the lowest-cost shipping rate. Also shortest path only uses the costliest shipping route from Chicago ( $v_1$ ) to Houston ( $v_3$ ) when there is no other way to meet the customer needs; indeed, when this happens, note how the augmented path reduces the existing flows between Washington, D.C. ( $v_2$ ) and Houston ( $v_3$ ), as well as between Washington, D.C. ( $v_2$ ) and Boston ( $v_4$ ).

Now the constructions can be presented that allow solving each of the remaining flow network problems listed in Figure 1.1. For each problem it is described that how to reduce the problem to Minimum Cost Flow.

## 2.6 Transshipment Problem [1]

There exists  $m$  supply stations  $s_i$ , each capable of producing  $sup(s_i)$  units of a commodity. There are  $n$  demand stations  $t_j$ , each demanding  $dem(t_j)$  units of the commodity. There are  $w$  warehouse stations  $w_k$ , each capable of receiving and reshipping (known as “transshipping”) a maximum  $max_k$  units of the commodity at the fixed warehouse processing cost of  $wp_k$  per unit. There is a fixed shipping cost of  $d(i, j)$  for each unit shipping from supply station  $s_i$  to demand stations  $t_j$ , a fixed transshipping cost of  $ts(i, k)$  for each unit shipped from supply station  $s_i$  to warehouse station  $w_k$ , and a fixed transshipping cost of  $ts(k, j)$  for each unit shipped from warehouse station  $w_k$  to demand station  $t_j$ . The goal is to determine the flow  $f(i, j)$  of



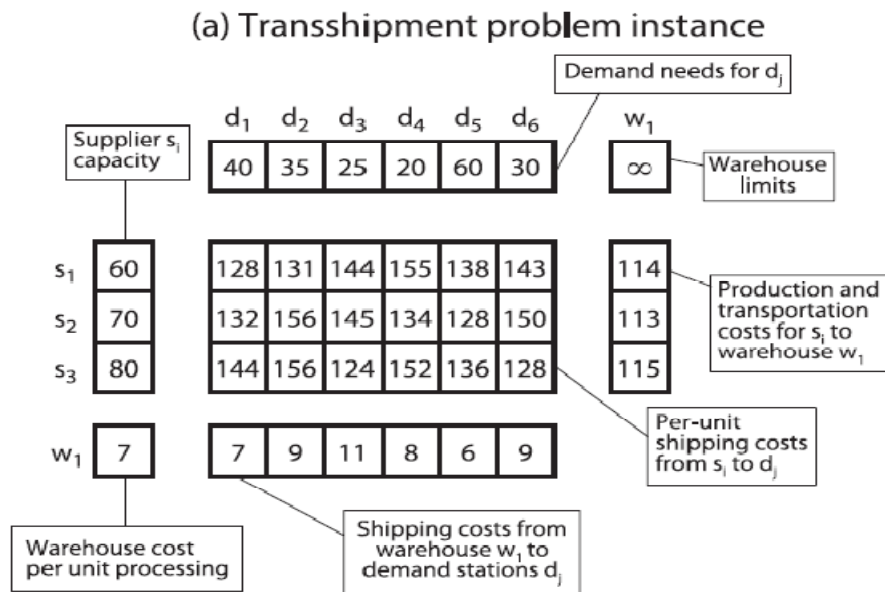
units from supply station  $s_i$  to demand station  $t_j$  that minimizes the overall total cost, which can be concisely defined as:

$$\text{Total Cost (TC)} = \text{Total Shipping Cost (TSC)} + \text{Total Transshipping Cost (TTC)}$$

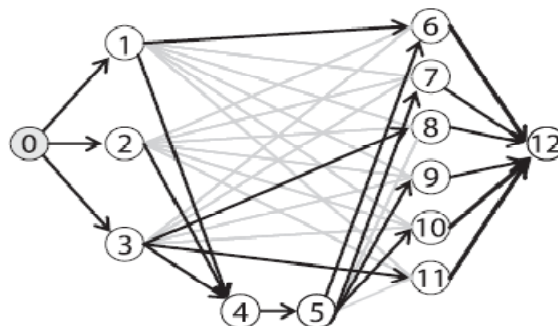
$$TSC = \sum_i \sum_j d(i, j) * f(i, j)$$

$$TTC = \sum_i \sum_k ts(i, k) * f(i, k) + \sum_j \sum_k ts(j, k) * f(j, k)$$

The goal is to find integer values for  $f(i, j) \geq 0$  that ensure that  $TC$  is a minimum while meeting all of the supply and demand constraints. Finally, the net flow of units through a warehouse must be zero, to ensure that no units are lost (or added!). The supplies  $sup(s_i)$  and demands  $dem(t_i)$  are all greater than 0. The shipping costs  $d(i, j)$ ,  $ts(i, k)$ , and  $ts(k, j)$  may be greater than or equal to zero.



(b) Minimum Cost Flow problem instance



**Figure 2.10** Sample Transshipment problem instance converted to Minimum Cost Flow problem instance

## Solution

The Transshipment problem instance can be converted into Minimum Cost Flow problem instance as shown in Figure 2.10 [1] by constructing a graph  $G = (V, E)$  such that:

### **V contains $n + m + 2 * w + 2$ vertices**

Each supply station  $s_i$  maps to a vertex numbered  $i$ . Each warehouse  $w_k$  maps to two different vertices, one numbered  $m + 2 * k - 1$  and one numbered  $m + 2 * k$ . Each demand station  $t_j$  maps to  $1 + m + 2 * w + j$ . Create a new source vertex  $src$  (labeled 0) and a new target vertex  $tgt$  (labeled  $n + m + 2 * w + 1$ ).

### **E contains $(w + 1) * (m + n) + m * n + w$ edges**

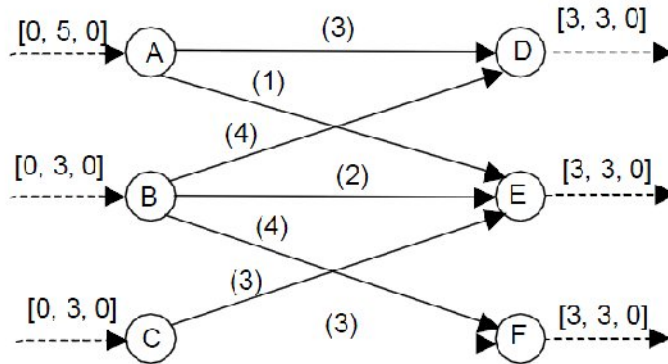
The process for constructing edges from the Transshipment problem instance can be found in the Transshipment class in the code repository.

Once the Minimum Cost Flow solution is available, the transshipment schedule can be constructed by locating those edges  $(u, v) \in E$  whose  $f(u, v) > 0$ . The cost of the schedule is the sum total of  $f(u, v) * d(u, v)$  for these edges.

## 2.7 Transportation Problem [20]

The Transportation problem instance can be converted into a Transshipment problem instance with no intermediate warehouse nodes [1]. The transportation problem is simple in form, but surprisingly useful in practice. It consists of a set of sources of some product (e.g. factories producing canned vegetables), which are directly linked to sinks of the product (e.g. markets in various cities which buy the canned vegetables). Each link has an associated cost per unit of flow (e.g. cost per delivered truckload in this case).

Consider the example in Figure 2.11 [20], which has three factories (A, B, and C) shipping to three markets (D, E, and F). The “transportation arcs” are the arcs which directly connect the sources (factories) to the sinks (markets); these are labeled only with the cost per unit of flow because the lower flow bounds are all at the default of zero, and the upper flow bounds are all at the default of infinity.



**Figure 2.11** A simple transportation model. The transportation arcs are labeled with the cost per unit of flow

The question in this case is: how many truckloads per day should be produced at each factory and shipped to each sink to meet the market demands at minimum total cost? After solution, the flows in the transportation arcs will be known; hence the number of truckloads to ship from each factory to each market will be known.

It's always good to give a network model of this type a simple "idiot test" at first glance. Figure 2.11 shows that the factories can produce up to a total of 11 units of flow while the markets demand exactly nine units of flow. This model passes the idiot test: there is sufficient supply to meet the demand. Of course, the model may fail for other reasons, e.g. the demand at a particular market cannot be met from the supply available to it. The linear programming solver will detect any of these problems, and appropriate infeasibility analysis routines can be brought into play.

## 2.8 The Assignment Problem [20]

Assignment problem is simply a more restricted version of the Transportation problem: each supply node must supply only a single unit, and the demand for each demand node is also one.

The assignment problem is a classic that also appears in the integer programming literature. In the usual form of the problem, there is a need to assign a set of people to a set of tasks. Each person takes a certain number of minutes to do a certain task, or cannot do a particular task at all, and each person can be assigned to exactly one task.

How should the people be assigned to the tasks to minimize the total time taken to do all of the tasks?

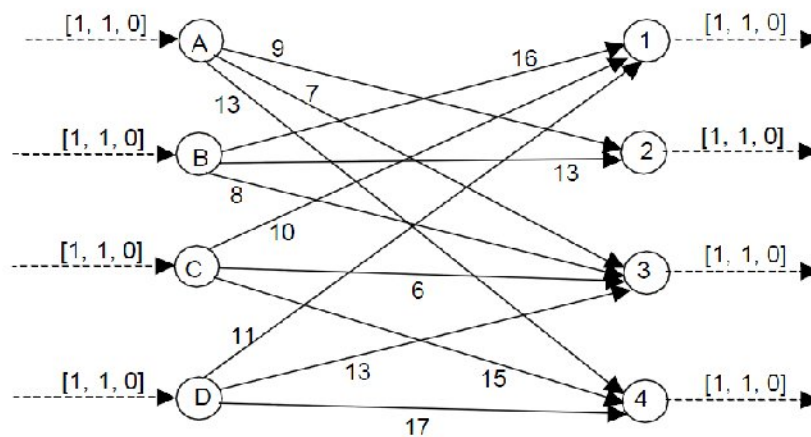
The data for an assignment problem is often collected in a table, as shown in table 2.1 for example. The number in each cell indicates the number of minutes required for a particular person to do a particular task. The notation “NA” in a cell indicates that the associated person cannot do the task associated with the cell. It’s not obvious how to assign the people to the tasks by simple inspection of the table. For example, it may be tried looking at each task and simply choosing the best person for that task. But as it can be seen, person A is the best for tasks 2, 3, and 4. How should the tie be broken? Other ad hoc procedures also soon run into trouble. A more organized approach is needed.

	Task 1	Task 2	Task 3	Task 4
Person A	NA	9	7	13
Person B	16	13	8	NA
Person C	10	NA	6	15
Person D	11	NA	13	17

**Table 2.1** Person-task relationship for assignment problem

**Solution**

The Assignment problem instance can be converted into a transportation problem instance, with the restriction that the supply nodes provide a single unit and the demand nodes require a single unit [1].



**Figure 2.12** The assignment problem cast as a transportation network

Each person is modeled as a source node which introduces exactly one unit of flow into the network, and each task is modeled as a sink node which removes exactly one unit of flow from the network, as shown in Figure 2.12 [20]. Each arc has the default upper and lower flow bounds, but the cost per unit of flow is set equal to the number of minutes for the person to do the job. To avoid diagram clutter, each arc is labelled only with the cost per unit of flow.

After the solution of the resulting network linear program, the flows in the arcs (i.e. the values of the variables in the linear program) will be known. The flow in any arc will be exactly zero or exactly one. It's because (i) the unimodularity property restricts the arc flows to integer values because all of the node equations have integer constants, and (ii) the sources and sink nodes in the model all have inflows or outflows of exactly one unit of flow. Given this, the optimal set of assignments is shown by the arcs that have a positive flow. Each positive-flow arc indicates a person-to-task assignment that should be made. The objective function value gives the minimum total time associated with this assignment.

A straightforward variation of the assignment problem is the case in which there are more people than jobs. This is easy to handle simply by making each person the source of *up to* one unit of flow, by labelling the phantom arc associated with each person as  $[0,1,0]$ .

## Chapter 3

### Problem Statement

---

Maximum flow problem is the classical network flow problem and once, it is solved other network flow problems can be converted into maximum flow problem. So among all the network flow problems, maximum flow problem is selected to be solved. Now the problem is to calculate the maximum flow value in the flow network graph which can be moved from source to sink without violating any capacity constraint. Any real world problem can be taken into as instance.

INSTANCE : Suppose one pipeline system is there in mumbai to supply water in different areas of mumbai. The pipeline between any two areas has a stated capacity in gallons per hour, given as a maximum flow at which water can flow through the pipe between those two areas. Now, suppose we want to supply water from the source area, say A to the sink area, say I and water passes through 7 other areas before reaching from source to sink. Suppose these 7 areas are B, C, D, E, F, G, H and pipeline between any two areas has defined capacity. So, the problem is to calculate the maximum amount of water which can flow from A to I.

OUTPUT: Output is the maximum amount of water in gallons per hour which can flow from A to I.

### 3.1 Methodology

The following steps are carried out to solve the problem.

- First of all, the problem discussed in the INSTANCE is converted into directed graph by representing area as a vertex of the graph and pipeline between any two areas as an edge of the graph.
- The capacity of the pipeline in gallons per hour is represented as capacity of an edge in units.
- Then new modified algorithm of the existing algorithm is designed.
- After this the problem is solved using both algorithms and comparison is done between them.
- Finally, the problem is implemented using implementation code of the new algorithm to achieve the desired output.

### 4.1 Converting the Problem into Graph

- Now the problem discussed in the INSTANCE is converted into directed graph by representing areas as vertices of the graph and pipelines between any two areas as edges of the graph.
- The capacity of the pipeline in gallons per hour is represented as capacity of an edge in units between vertices.

Now, the following table shows the defined capacities of each pipeline between any two areas which can flow between corresponding two areas. That is the following table is the input file to the problem discussed in INSTANCE.

Source Area	Destination Area	Capacity (gallons/hour)
A	B	34
A	B	20
A	E	35
B	D	20
B	E	8
C	E	8
C	F	15
D	F	10
D	G	22
D	H	18
E	D	35
E	F	15
F	H	35
G	I	48
H	G	20
H	I	30

**Table 4.1** Defined capacities of each pipeline between two areas.

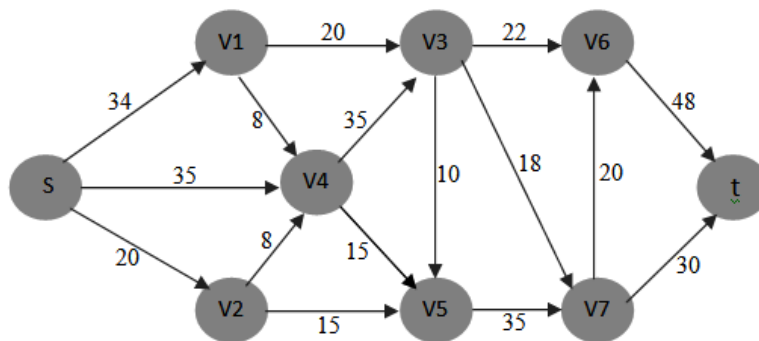
Let the graph  $G = (V, E)$  be a flow network with source  $s$ , sink  $t$ , and an integer capacity  $c(u, v)$  on each edge  $(u, v) \in E$ . Let  $C = \max_{(u, v) \in E} c(u, v)$ .

Now, following correspondence between areas and vertices is used to create a graph.

Area	Vertex
A	S
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	T

**Table 4.2** Correspondence between areas and vertices

So, the initial graph corresponding to the tables 4.1 and 4.2 is as follows.



**Figure 4.1** The initial flow network corresponding to the problem to be solved

Now the modified Edmonds-Karp Method is given below which can be used to compute a maximum flow in  $G$ .

**Algorithm 4.1 MOD\_EDMONDS-KARP ( $G, s, t$ )**

- [1] **for** each edge  $(u, v) \in E[G]$  [First initialize the flow  $f$  to 0]
- [2]     **do**  $f[u, v] \leftarrow 0$
- [3]      $f[v, u] \leftarrow 0$
- [4]  $C \leftarrow \max_{(u, v) \in E} c(u, v)$
- [5]  $I \leftarrow 2^{\text{floor}(\log_2 C)}$
- [6] **while**  $I \geq 1$
- [7]     **do while** there exists an augmenting path  $p$  from  $s$  to  $t$  in the residual



```

network  $G_f$  of capacity atleast I
[8]   do  $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
[9]   for each edge  $(u, v)$  in  $p$ 
[10]  do  $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
[11]   $f[v, u] \leftarrow -f[u, v]$ 
[12]   $I = I/2$ 
[13] return  $f$ 

```

## 4.2 Solution using Modified Edmonds-Karp Algorithm

Now, the maximum capacity of the graph is 48. So, the value of the variable  $C$  in the above algorithm will be 48.

So,  $2^{\text{floor}(\log_2 C)} = 32$ ,

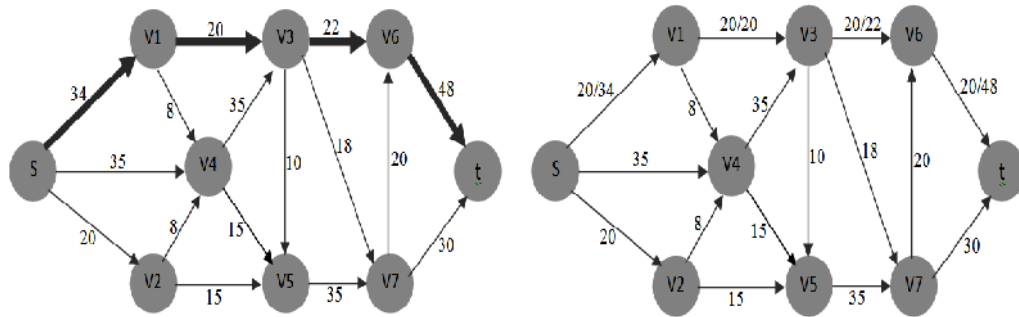
which will be value of the variable  $I$  in the 1<sup>st</sup> iteration of the algorithm.

**1<sup>st</sup> Iteration:**  $I = 32$ .

- So, the augmenting path with capacity at least 32 will be searched by the Breadth First Search procedure. But, there is no augmenting path with capacity at least 32.
- So, no flow will be added to the initial flow of the graph which is 0.

**2<sup>nd</sup> Iteration:**  $I = I/2 = 32/2 = 16$ .

- So, now the augmenting path with capacity at least 16 will be searched by the same BFS procedure in the residual graph which is given below in Figure 4.2 corresponding to the initial graph.
- The augmenting path will be searched till path with capacity at least 16 is found in the graph.
- Now, in the consecutive figures, the left side figure shows the residual graph and the right side figure shows the corresponding flow in the graph.

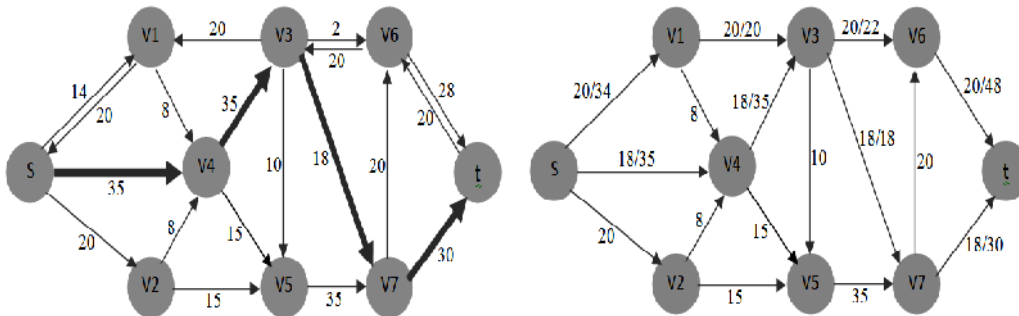


**Figure 4.2** Residual Graph before any augmentation and flow graph after 1<sup>st</sup> augmentation

Whenever the augmenting path is to be found in the graph, if there are more than 1 path satisfying the capacity criteria, then the path is determined by BFS procedure on the basis of sequential ordering of the vertices and corresponding edges which has been given as input.

**1<sup>st</sup> augmentation:**

- So, the augmenting path found in 2<sup>nd</sup> iteration is **s-v1-v3-v6-t** with capacity 20.
- So, the initial flow is augmented by 20 units and the flow in the graph is shown in the above right side figure 4.2 giving maximum flow value **f=20**.
- The residual graph after 1<sup>st</sup> augmentation is shown below in Figure 4.3.



**Figure 4.3** Residual Graph after 1<sup>st</sup> augmentation and flow graph after 2<sup>nd</sup> augmentation

**2<sup>nd</sup> augmentation:**

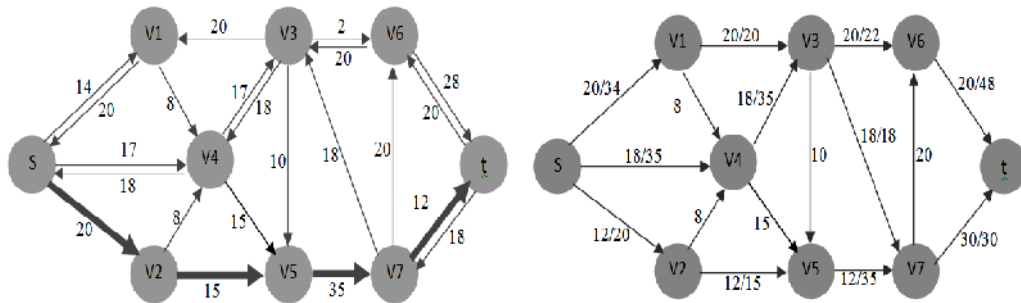
- Now, again there is a path with capacity at least 16 and the path found in the same 2<sup>nd</sup> iteration is **s-v4-v3-v7-t** with capacity 18.

- So, the maximum flow is augmented by 18 units and the flow in the graph is shown in the above figure 4.3 giving maximum flow value  $f = 20 + 18 = 38$ .
- Now, there is no path with capacity at least 16.

Same procedure will be followed until variable  $I$  becomes  $< 1$ .

**3<sup>rd</sup> Iteration:**  $I = I/2 = 16/2 = 8$ .

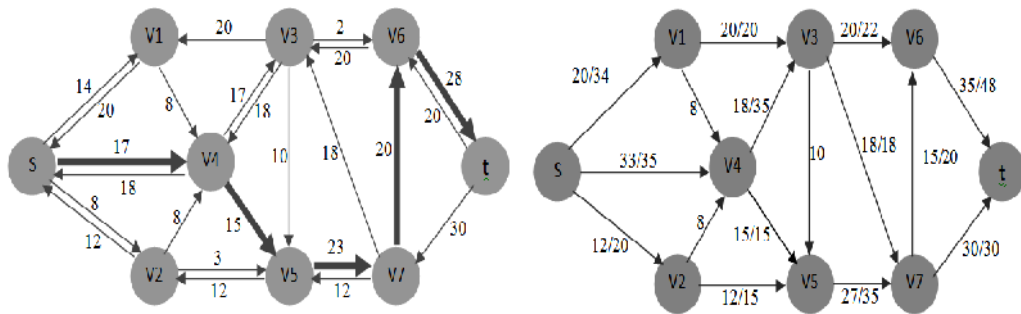
- So, now the augmenting path with capacity at least 8 will be searched.
- The residual graph after 2<sup>nd</sup> augmentation is shown below in figure 4.4.



**Figure 4.4** Residual Graph after 2<sup>nd</sup> augmentation and flow graph after 3<sup>rd</sup> augmentation

**3<sup>rd</sup> augmentation:**

- The augmenting path found in 3<sup>rd</sup> iteration is  $s-v2-v5-v7-t$  with capacity 12.
- So, the maximum flow is augmented by 12 units and the flow in the graph is shown in the above figure 4.4 giving maximum flow value  $f = 38 + 12 = 50$ .
- The residual graph after 3<sup>rd</sup> augmentation is shown below in figure 4.5.



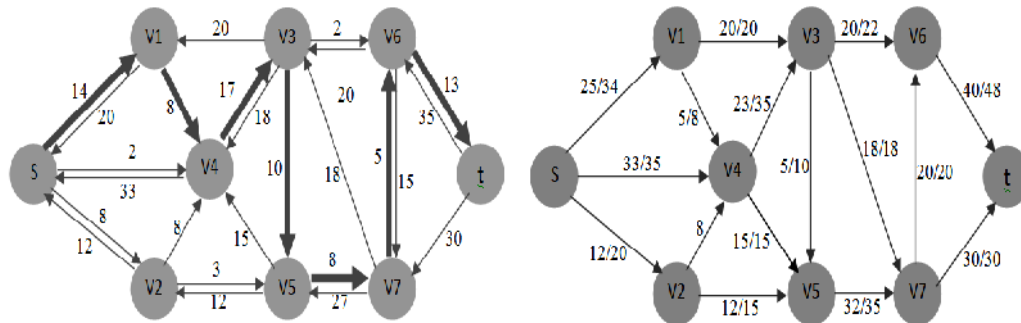
**Figure 4.5** Residual Graph after 3<sup>rd</sup> augmentation and flow graph after 4<sup>th</sup> augmentation

**4<sup>th</sup> augmentation:**

- Now, again there is a path with capacity at least 8 and the path found in the same 3<sup>rd</sup> iteration is **s-v4-v5-v7-v6-t** with capacity 15.
- Maximum flow value  $f = 50 + 15 = 65$ .
- Now, there is no path with capacity at least 8.

**4<sup>th</sup> Iteration:**  $I = I/2 = 8/2 = 4$ .

- Now the augmenting path with capacity at least 4 will be searched.
- The residual graph after 4<sup>th</sup> augmentation is shown below in figure 4.6.



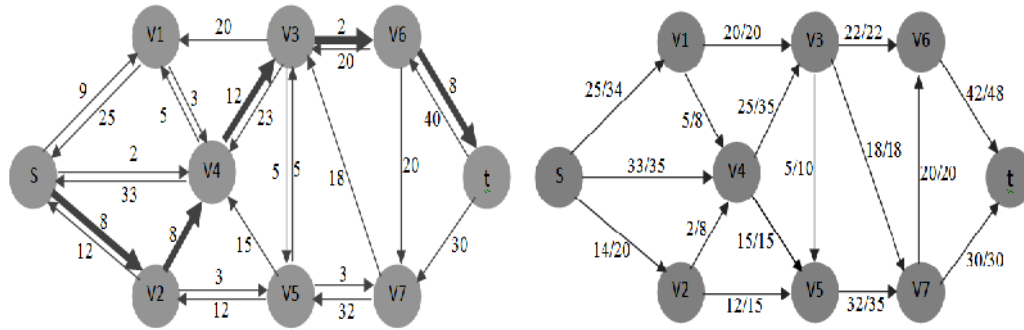
**Figure 4.6** Residual Graph after 4<sup>th</sup> augmentation and flow graph after 5<sup>th</sup> augmentation

**5<sup>th</sup> augmentation:**

- The augmenting path found in 4<sup>th</sup> iteration is **s-v1-v4-v3-v5-v7-v6-t** with capacity 4.
- Maximum flow value  $f = 65 + 5 = 70$ .
- Now, there is no path with capacity at least 4.

**5<sup>th</sup> Iteration:**  $I = I/2 = 4/2 = 2$ .

- Now the augmenting path with capacity at least 2 will be searched.
- The residual graph after 5<sup>th</sup> augmentation is shown below 4.7.



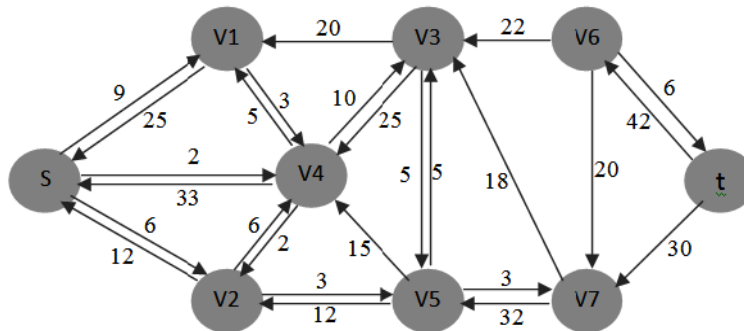
**Figure 4.7** Residual Graph after 5<sup>th</sup> augmentation and flow graph after 6<sup>th</sup> augmentation

**6<sup>th</sup> augmentation:**

- The augmenting path found in 5<sup>th</sup> iteration is **s-v2-v4-v3-v6-t** with capacity 2.
- Maximum flow value  $f = 70 + 2 = 72$ .
- Now, there is no path with capacity at least 2.

**6<sup>th</sup> Iteration:**  $I = I/2 = 2/2 = 1$ .

- Now the augmenting path with capacity at least 1 will be searched.
- The residual graph after 6<sup>th</sup> augmentation is shown below in figure 4.8.



**Figure 4.8** Residual Graph after 6<sup>th</sup> augmentation

Now, there is no path with capacity at least 1. So, no augmentation is possible in 6<sup>th</sup> iteration.

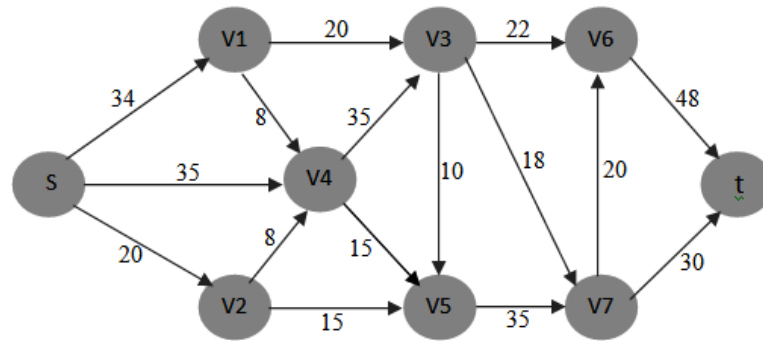
Now,  $I = I/2 = 1/2 = 0$ . So, here the algorithm halts and the resulting flow in graph returns the maximum flow.

So, Maximum flow value  $f = 72$ .

### 4.3 Solution using Edmonds-Karp Algorithm

Now, if the same problem is solved using **Edmonds-Karp** algorithm, then the maximum flow is calculated as shown below.

Consider the same graph as in figure 4.1.



**Figure 4.1** The initial flow network corresponding to the problem to be solved

Whenever there are more than 1 path satisfying the BFS search criteria, the path is determined by BFS procedure on the basis of sequential ordering of the vertices and corresponding edges which has been given as input.

#### 1<sup>st</sup> Iteration

- The augmenting path found is **s-v1-v3-v6-t** with capacity 20.
- Maximum flow value  $f = 20$

#### 2<sup>nd</sup> Iteration

- The augmenting path found is **s-v2-v5-v7-t** with capacity 15.
- Maximum flow value  $f = 20 + 15 = 35$

#### 3<sup>rd</sup> Iteration

- The augmenting path found is **s-v4-v3-v6-t** with capacity 2.
- Maximum flow value  $f = 35 + 2 = 37$

#### 4<sup>th</sup> Iteration

- The augmenting path found is **s-v4-v3-v7-t** with capacity 15.
- Maximum flow value  $f = 37 + 15 = 52$

#### 5<sup>th</sup> Iteration

- The augmenting path found is **s-v4-v3-v7-v6-t** with capacity 3.
- Maximum flow value  $f = 52 + 3 = 55$

### 6<sup>th</sup> Iteration

- The augmenting path found is **s-v4-v5-v7-v6-t** with capacity 15.
- Maximum flow value  $f = 55 + 15 = 70$

### 7<sup>th</sup> Iteration

- The augmenting path found is **s-v1-v4-v3-v5-v7-v6-t** with capacity 2.
- Maximum flow value  $f = 70 + 2 = 72$

Now, there is no augmenting path with capacity at least 1. So, here the algorithm halts and the resulting flow in graph returns the maximum flow.

Thus, Maximum flow value  $f = 72$ .

## 4.4 Discussion and Results

In the modified algorithm, **0 or more** augmentation (increment in the current flow) is possible in the same iteration while in the Edmonds-Karp algorithm, **only 1** augmentation is possible in each iteration.

The following table shows the comparison between two algorithms and also shows the above fact.

Iteration No.	Modified Edmonds-Karp (No. of augmentation)	Edmonds-Karp (No. of augmentation)
1 <sup>st</sup>	0	1
2 <sup>nd</sup>	2	1
3 <sup>rd</sup>	2	1
4 <sup>th</sup>	1	1
5 <sup>th</sup>	1	1
6 <sup>th</sup>	0	1
7 <sup>th</sup>	Halts in 6 <sup>th</sup> Iteration	1

**Table 4.3** Iteration – augmentation comparison of two algorithms for the problem

- As the table shows, the modified algorithm calculates the **maximum flow** after **6 augmentations** with **6 iterations** while the Edmonds-Karp algorithm calculates the **maximum flow** after **7 augmentations** with **7 iterations**.

- Thus, it can be seen that for such a graph with less vertices also, the modified algorithm takes 1 less iteration than the Edmonds-Karp algorithm.
- And if the no. of vertices in the graph is much larger, then the modified algorithm will perform extremely better than the Edmonds-Karp algorithm.
- The Edmonds-Karp algorithm runs in  $O(E^2V)$  while the modified algorithm runs in  $O(E^2\log_2C)$  where  $C$  is maximum edge capacity in the graph.
- Hence it can be said that the modified algorithm will perform better than Edmonds-Karp algorithm for any graph where  $V > \log_2C$  and this condition generally holds for most graphs and if no. of vertex ( $V$ ) is more, then obviously no. of edges ( $E$ ) will be more.

The following table shows some cases of graph where it can be seen that even though the difference between  $V$  and  $\log_2C$  is less, the difference between  $E^2\log_2C$  and  $E^2V$  considerably larger because of  $E^2$ .

No. of vertices $V$	No. of edges $E$	Maximum capacity $C$	$\log_2C$	$E^2\log_2C$	$E^2V$
15	25	128	7	4375	9375
10	18	75	6.229	2018	3240
12	20	30	4.907	1963	4800
15	23	32	5	2645	7935

**Table 4.4** Comparison of complexities of two algorithms

Though the modified algorithm performs better in most cases, in some cases where the maximum capacity of the graph is larger compared to no. of vertices in the graph then this algorithm is less suitable. In those cases, Edmonds-Karp and Ford-Fulkerson algorithms can perform better.



## 4.5 Some Other Results

### The modified Edmonds-Karp algorithm returns a maximum flow

- This algorithm uses the Ford-Fulkerson method. It repeatedly augments the flow along an augmenting path until there are no augmenting paths of capacity greater  $\geq 1$ .
- Since all the capacities are integers, and the capacity of an augmenting path is positive, this means that there are no augmenting paths whatsoever in the residual graph.
- Thus, by the max-flow min-cut theorem, the modified Edmonds-Karp algorithm returns a maximum flow.

### For a given number $I$ , an augmenting path of capacity at least $I$ can be found in $O(E)$ time, if such a path exists.

- The capacity of an augmenting path is the minimum capacity of any edge on the path, so we are looking for an augmenting path whose edges *all* have capacity at least  $I$ .
- Do a breadth-first search or depth-first-search as usual to find the path, considering only edges with residual capacity at least  $I$ . (Treat lower-capacity edges as though they don't exist.)
- This search takes  $O(V + E) = O(E)$  time. (Note that  $|V| = O(E)$  in a flow network.)

### The new modified Edmonds-Karp algorithm runs in $O(E^2 \log_2 C)$

- Now, consider the algorithm 4.1 given at the starting of this chapter.
- The time complexity is dependent on the loop of steps 6-12 because the steps 1-5 take  $O(E)$  time.
- The outer **while** loop executes  $O(\log C)$  times, since  $I$  is initially  $O(C)$  and is halved on each iteration, until  $K < 1$ . The inner **while** loop executes  $O(E)$  times for each value of  $I$  and as shown in above result each iteration takes  $O(E)$  time.
- Thus, the total time is  $O(E^2 \log_2 C)$ .

## 4.6 Implementation

The modified Edmonds-Karp algorithm can be applied to any graph to calculate the maximum flow value which can be moved from source to sink in the graph. The implementation code in C language is given in Appendix A. Implementation code reads the data from input file **maxflow.in**. Finally, the output file shows the maximum flow value and execution of the implementation code. Input file and output file are also shown in Appendix A.

## Chapter 5

### Conclusion and Future Scope

---

The basic classical techniques used in maximum-flow algorithms can be adapted to solve other network-flow problems also. By solving maximum flow problem we can consider a class of similar problems that seek to maximize the flow through a flow network while at the same time minimizing the cost of that flow. So it can be concluded that computing the Maximum Flow in the flow network graph produces a maximal matching set for the original Bipartite Matching problem. And on further reflections, it can be used to solve the more powerful Minimal Cost Flow problem, which enables us to immediately solve the Transshipment, Transportation, and Assignment problems.

The modified Edmonds-Karp algorithm returns a maximum flow and as discussed in the previous chapter, we can say that this algorithm takes less no. of iterations and less augmentation to calculate the maximum flow. An augmenting path of desired capacity in each iteration can be found in  $O(E)$  time, if such a path exists. The new modified algorithm runs in  $O(E^2 \log_2 C)$  while Edmonds-Karp algorithm runs in  $O(E^2 V)$ . So, finally it can be concluded that the modified Edmonds-Karp algorithm performs better in most cases compared to Edmonds-Karp algorithm.

#### 5.1 Future Scope

- The Ford Fulkerson algorithm is the general algorithm to solve the network flow problems and its improvement is Edmonds-Karp algorithm which performs better than it.
- In this thesis, the new modified algorithm is designed and implemented and proved that in most cases it performs better than Edmonds-Karp. Some other algorithms with approximately equivalent complexities are also available.
- So, In future more optimized algorithms can be developed to solve the network flow problems in more efficient manner.

## References

---

- [1] George T. Heinemen, Garry Pollice and Stanley Selkow. Algorithms in a Nutshell, A Desktop Quick Reference. O’rielly.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001). Introduction to Algorithms. McGraw-Hill, New York, 2nd edition, 2001.
- [3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin(1993). Network Flows: Theory, Algorithms, and Applications. Prentice Hall.
- [4] Andrew V. Goldberg, Eva Tardos and Robert E. Tarjan (1988). A new approach to the maximum-flow problem. Journal of the ACM. 35:921–940,
- [5] Andrew V. Goldberg, Eva Tardos, and Robert E. Tarjan. “Network flow algorithms. Paths, Flows, and VLSI-Layout”. Springer-Verlag, 1990, pages 101–164..
- [6] Jack Edmonds and Richard M. Karp (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". Journal of the ACM 19 (2): 248–264. doi:10.1145/321694.321699
- [7] Bazarra, M. and J. Jarvis. Linear Programming and Network Flows. John Wiley & Sons, 1977.
- [8] Ford, L. R. Jr. and D. R. Fulkerson. Flows in Networks. Princeton University Press, 1962.
- [9] John E. Hopcroft and Richard M. Karp. “An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs”. SIAM Journal on Computing, 2(4):225–231, 1973.

- [10] Christos H. Papadimitriou, Kenneth Steiglitz (1998). The Max-Flow, Min-Cut Theorem. Combinatorial Optimization: Algorithms and Complexity. Dover.
- [11] Eugene Lawler (2001). "4. Network Flows". Combinatorial Optimization: Networks and Matroids. Dover.
- [12] "Maximum Flow Problem"  
[http://en.wikipedia.org/wiki/Maximum\\_flow\\_problem](http://en.wikipedia.org/wiki/Maximum_flow_problem)
- [13] Ahuja, R.K., Orlin, J.B. (1987). "A fast and simple algorithm for the maximum flow problem". Sloan School of Management, M.I.T. Sloan Working Paper 1905-87.
- [14] "Recent Developments in Maximum Flow Algorithms"  
[www.springerlink.com/index/tfdyr7n9xyer80p7.pdf](http://www.springerlink.com/index/tfdyr7n9xyer80p7.pdf)
- [15] "Theoretical improvements in algorithmic efficiency for network flow problems." <http://portal.acm.org/citation.cfm?id=321699>
- [16] "A Capacity Scaling Algorithm for the Constrained Maximum Flow Problem"  
<http://dspace.mit.edu/bitstream/handle/1721.1/2480/SWP-3587-28521208.pdf>.
- [17] Andrew V. Goldberg and Satish Rao. "Beyond the flow decomposition" barrier. *Journal of the ACM*, 45:783–797, 1998.
- [18] Ravindra K. Ahuja and James B. Orlin. "A fast and simple algorithm for the maximum flow problem". *Operations Research*, 37(5):748–759, 1989.
- [19] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. "Improved time bounds for the maximum flow problem". *SIAM Journal on Computing*, 18(5):939–954, 1989.
- [20] "Network Flow Programming"  
[www.sce.carleton.ca/faculty/chinneck/po/Chapter10.pdf](http://www.sce.carleton.ca/faculty/chinneck/po/Chapter10.pdf)

## List of Publications

---

- [1] Chintan Jain “Network Flow Problems and the Generalized Ford-Fulkerson Algorithm”, National Conference on Emerging Trends in Information Technology & Computing (ETIC-2010). Gurgaon Institute of Technology and Management, Gurgaon, Haryana, March 29, 2010.

### A.1 Implementation Code

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

#define UNVISITED 0
#define VISITED 1
#define PROCESSED 2
#define MAX_NODES 100
#define temp 500

int n; // number of nodes
int e; // number of edges
int MaxCap[MAX_NODES][MAX_NODES]; // MaxCap matrix
int CurrentFlow[MAX_NODES][MAX_NODES]; // CurrentFlow matrix
int Status[MAX_NODES]; // needed for breadth-first search
int Preceding[MAX_NODES]; // array to store augmenting path

int head,tail,I;
int q[MAX_NODES];

void ReadInputFile()
{
    int a,b,c,i,j;
    FILE* input = fopen("C:\\Users\\Tiya\\Desktop\\maxflow.in","r");
    // read number of nodes and edges
    fscanf(input,"%d %d",&n,&e);
    // initialize empty MaxCap matrix
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
            MaxCap[i][j] = 0;
    }
    // read edge capacities
    for (i=0; i<e; i++)
    {
        fscanf(input,"%d %d %d",&a,&b,&c);
        MaxCap[a][b] = c;
    }
    fclose(input);
}

void Enqueue (int x)
{
    q[tail] = x;
    tail++;
    Status[x] = VISITED;
}
```

```

int Dequeue ()
{
    int x = q[head];
    head++;
    Status[x] = PROCESSED;
    return x;
}

int BreadthFirstSearch (int start, int target)
{
    int u,v;
    for (u=0; u<n; u++)
    {
        Status[u] = UNVISITED;
    }
    head = tail = 0;
    Enqueue(start);
    Preceding[start] = -1;
    while (head!=tail)
    {
        u = Dequeue();
        // Search all adjacent white nodes v. If the MaxCap
        // from u to v in the residual network is positive,
        // enqueue v.
        for (v=0; v<n; v++)
        {
            if(Status[v]==UNVISITED && MaxCap[u][v]-CurrentFlow[u][v]>=1)
            {
                Enqueue(v);
                Preceding[v] = u;
            }
        }
    }
    // If the status of the target node is processed now,
    // it means that we reached it.
    return Status[target]==PROCESSED;
}

int min (int x, int y)
{
    return x<y ? x : y; // returns minimum of x and y
}

int MaximumFlow (int source, int sink)
{
    int i,j,u,max,p=1,q=1;
    // Initialize empty currentflow.
    int MaxFlow = 0;
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
            CurrentFlow[i][j] = 0;
    }
}

```



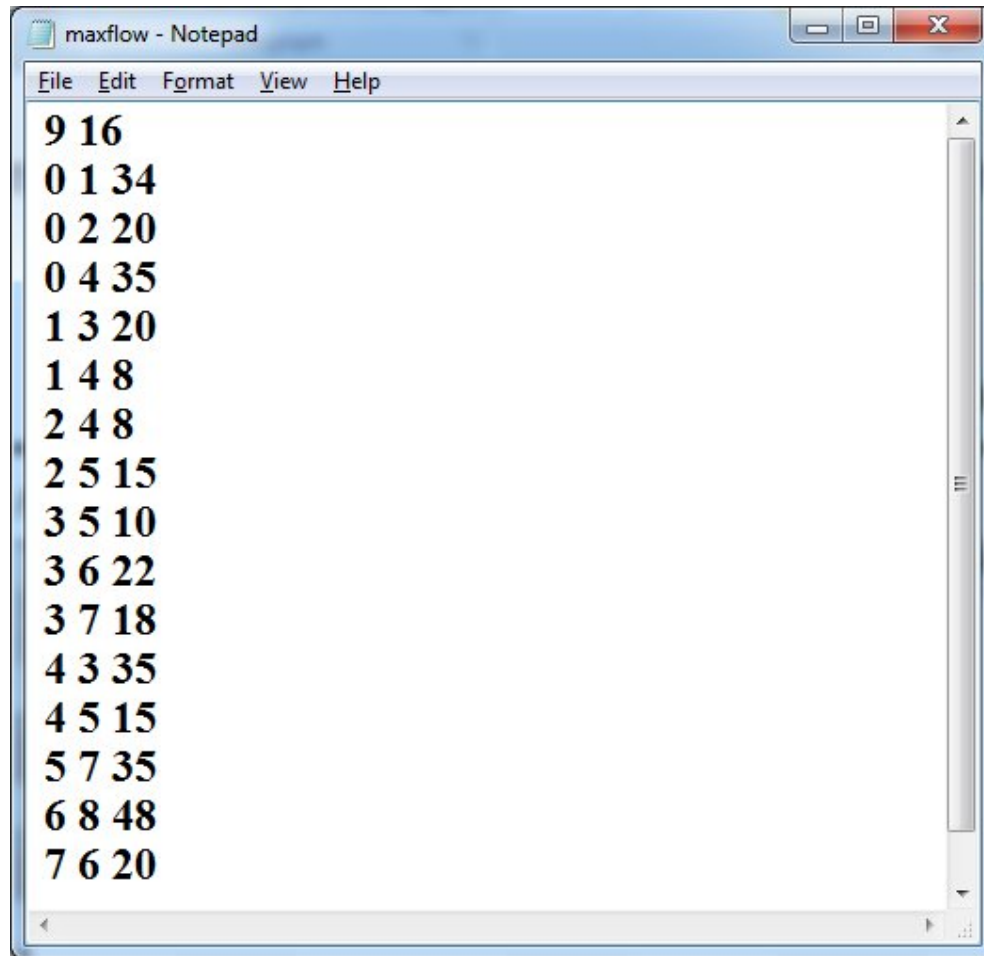
```

// While there exists an augmenting path,
// increment the currentflow along this path.
max=-1;
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        if(max<MaxCap[i][j])
            max=MaxCap[i][j];
    }
}
I=pow(2,floor((log(max)/log(2)))));
while(I>=1)
{
    printf("Iteration %d : I is %d\n",q++,I);
    while (BreadthFirstSearch(source,sink))
    {
        // Determine the amount by which we can increment the currentflow.
        int increment = temp;
        for (u=n-1; Preceding[u]>=0; u=Preceding[u])
        {
            increment = min(increment,MaxCap[Preceding[u]][u]-
                CurrentFlow[Preceding[u]][u]);
        }
        // Now increment the currentflow.
        printf("Augmentation %d is : %d\n",p++,increment);
        for (u=n-1; Preceding[u]>=0; u=Preceding[u])
        {
            CurrentFlow[Preceding[u]][u] += increment;
            CurrentFlow[u][Preceding[u]] -= increment;
        }
        MaxFlow += increment;
    }
    printf("\n\n");
    I=I/2;
}
// No augmenting path anymore. We are done.
return MaxFlow;
}

int main ()
{
    ReadInputFile();
    printf("The Maximum Flow value in the Flow Network is : %d
        units\n",MaximumFlow(0,n-1));
    getch();
    return 0;
}

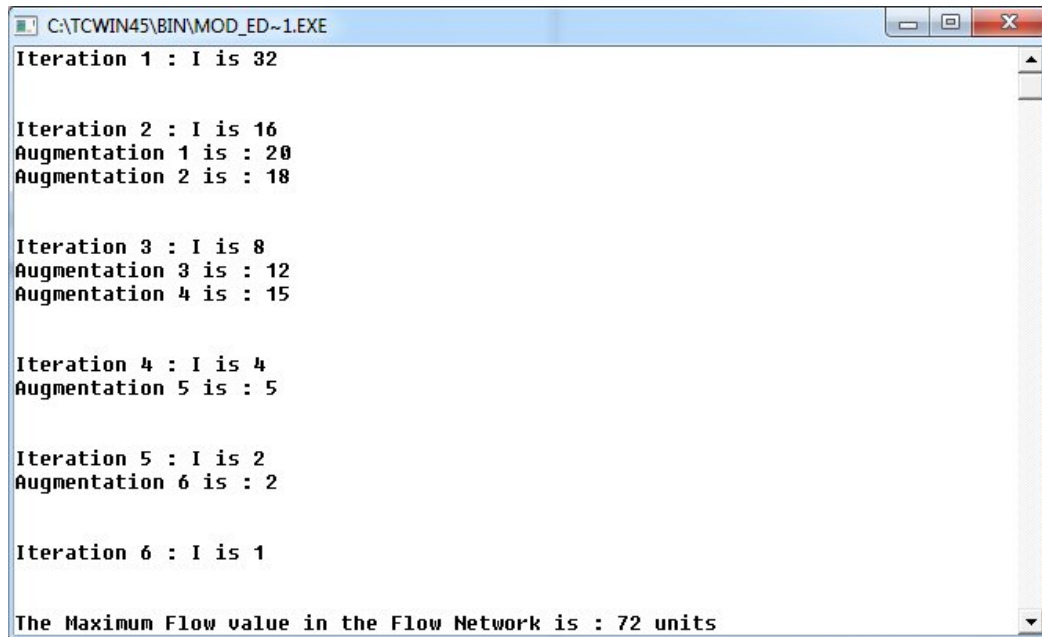
```

## A.2 Input File



```
maxflow - Notepad
File Edit Format View Help
9 16
0 1 34
0 2 20
0 4 35
1 3 20
1 4 8
2 4 8
2 5 15
3 5 10
3 6 22
3 7 18
4 3 35
4 5 15
5 7 35
6 8 48
7 6 20
```

### A.3 Output File



```
C:\TCWIN45\BIN\MOD_ED~1.EXE
Iteration 1 : I is 32

Iteration 2 : I is 16
Augmentation 1 is : 20
Augmentation 2 is : 18

Iteration 3 : I is 8
Augmentation 3 is : 12
Augmentation 4 is : 15

Iteration 4 : I is 4
Augmentation 5 is : 5

Iteration 5 : I is 2
Augmentation 6 is : 2

Iteration 6 : I is 1

The Maximum Flow value in the Flow Network is : 72 units
```