國 立 成 功 大 學

工 業 與 資 訊 管 理 研 究 所

碩 士 論 文

應用子網路擴張流量方法求解最大流量問題之研究

Solving The Maximum Flow Problem by Augmenting Flows on Subnetworks

指導教授：王逸琳 博士

研 究 生：李俊賢

中 華 民 國 九 十 八 年 七 月

國立成功大學

碩士論文

應用子網路擴張流量方法求解最大流量問題之研究

Solving the Maximum Flow Problem by Augmenting Flows on Subnetworks

研究生：李俊賢

本論文業經審查及口試合格特此證明

論文考試委員：

王逸琳

洪政煌

李宇欣 [illegible]

指導教授：王 逸 琳

系(所)主管：謝中奇

中 華 民 國 98 年 6 月 1 日

應用子網路擴張流量方法求解最大流量問題之研究

國立成功大學工業與資訊管理研究所碩士班

摘　　要

最大流量問題為一基本之網路最佳化問題，旨在於一具有流量上限的網路中尋找從起點到訖點間可流通之最大可能流量，此問題已有超過五十年以上的研究歷史。傳統的最大流量演算法大致可分為兩大類：其一為不斷自起點找出一條可擴張流量至訖點之擴張路徑來擴張流量；而另一則為先塞滿起點連結出去之弧，允許節點暫存流量，再將具有暫存流量的節點沿其鄰近之一條可行弧逐次擴張流量於其鄰近點，直至所有之暫存流量擴張至訖點或返回起點而止。這些解法在每次擴張流量時，皆僅能沿網路圖之單一路徑或單一弧進行，本論文以能一次即利用網路中所有的可行弧來擴張流量為目標，使用最短路徑所建構之子網路架構來擴張流量，提出三個新的具多項式時間複雜度之最大流量演算法。

第一個最大流量演算法根據可行子網路中各弧殘餘流量上限的比例來分配每次之擴張流量，稱為proportional arc augmenting (PAA) 演算法。PAA演算法依據可行子網路中各節點所連結出去之可行弧的殘餘流量上限比例，計算出該子網路所有弧之擴張流量向量δ，再根據此擴張流量向量計算出該次從起點到訖點之擴張流量，此種等比例擴張流量方式符合具較大流量上限之弧能擴張更多流量的直覺，並保證每次至少可塞滿一個節點。在一個具有n個節點及m條弧的網路上，PAA演算法可在$O(n^2m)$時間內算出其最大流量。此外，本論文亦提出一些加速PAA演算法實作效率之方法。

第二個最大流量演算法稱為flow splitting augmenting (FSA) 演算法，旨在以平分

流量的方式，快速地將子網路中各節點所接收之流量依其連結出去之弧個數來平均分配，如此可保證每次擴張流量時至少會塞滿一條可行弧。在理論複雜度上，FSA演算法可在$O(nm^2)$時間內算出其最大流量。

第三個最大流量演算法以求解線性規劃時可避免退化問題之最小平方對偶-主(least-squares dual-primal, LSDP)演算法為基礎，在最短路徑所建構之子網路架構上使用電路學的克西荷夫定律，計算子網路中每弧應該推送之單位流量比例以擴張流量，稱為modified least-squares dual-primal (MLSDP)演算法。此演算法每次亦可保證塞滿一條節線，並可在$O(nm^5)$時間內算出其最大流量，解決了先前文獻中無法推估如何運用LSDP法求解最大流量問題之理論複雜度問題。由於此法必須重複計算克西荷夫定律之稀疏矩陣線性聯立方程式，因此在實作上我們亦使用專業的聯立方程式求解軟體以加快求解效率。

本論文所提出之三種最大流量演算法皆可能會得到非整數之最佳流量，因此我們提出一個流量分解的方式，以在$O(m^2)$時間內將其轉換為整數最佳流量。在求解效率測試上，我們將所提出之演算法與現今最常見之數種最大流量演算法在不同的模擬測試網路上做求解時間及運算次數之比較。測試結果顯示，雖然此三種新的演算法在平均求解表現上並不十分突出，我們仍可發現其求解機制可能會對某些特殊網路架構有利。最後，我們提出數種演算法效率改善之可能作法，並建議一些具挑戰性的未來研究方向。

關鍵字：最大流量問題、網路最佳化、擴張流量、等比例擴張、最小平方對偶-主、克西荷夫定律

ABSTRACT

Solving The Maximum Flow Problem by Augmenting Flows on Subnetworks

Chun-Hsien Lee

The $s-t$ maximum flow problem is a fundamental network optimization problem which computes for the largest amount of flow sent through a network from a source node s to a sink node t. This problem often appears in many business applications as an optimization subproblem, and has been investigated extensively over the recent five decades. Conventional maximum flow algorithms either augment flow via a simple path to t, or via admissible arcs from those nodes of excess flow to their neighbors at each iteration. In this thesis, we propose three new maximum flow algorithms that ship flow via all the arcs in an admissible subnetwork composed by shortest augmenting paths at each iteration.

With the intuition to ship more flow along more spacious admissible arcs at each iteration, our proposed proportional arc augmenting algorithm (PAA) calculates an augmenting-flow vector δ along all arcs in an admissible subnetwork based on the proportion of the residual capacity for each arc emanating from a node to their sum. In particular, PAA ships as much flow as possible along the augmenting-flow vector until one admissible arc is saturated, then all of its neighbor admissible arcs are saturated as well. PAA iteratively saturates at least one node at each iteration, and calculates the maximum flow in polynomial time. Several speed-up techniques are also proposed to improve the practical efficiency of PAA.

In order to save more overhead than PAA in calculating a different δ while still shipping flow via all the admissible arcs at each iteration, we proposed flow splitting augmenting algorithm (FSA) which equally distributes all the flow entering a node to each of its outgoing arcs in an admissible subnetwork. FSA saturates at least one arc at each iteration, instead of one node as PAA. We also show FSA calculates the maximum flow in polynomial time.

Our third algorithm, called as modified least-squares dual primal algorithm (MLSDP), exploits the least-squares primal-dual algorithm that was designed for solving LPs without degenerate pivots. In particular, the augmenting-flow vector calculated by MLSDP guarantees nondegenerate pivots at each iteration, which may lead to fewer iterations of flow augmentation than other algorithms. Instead of solving a quadratic nonnegative least squares problem for calculating an augmenting-flow vector, MLSDP solves systems of linear equations composed by Kirchhoff's circuit laws. We show the polynomial-time complexity for MLSDP, and also suggest more efficient implementation that exploits sparse matrix operations.

All of our proposed maximum flow algorithms give fractional optimal flow, which can be further converted into integral optimal flow within polynomial-time by techniques based on flow decomposition. Comprehensive computational experiments have been conducted to analyze the practical efficiency of our proposed algorithms in comparison to several state-of-the-art maximum flow algorithms over several families of simulated networks. Although the results indicate our current implementations on proposed algorithms perform less efficient than the state-of-the-art preflow-push algorithms, we observe that our proposed algorithms tend to terminate in fewer iterations and make several speed-up suggestions for future research.

Keywords: maximum flow problem, network optimization, augmenting flow, proportional arc augmenting, least-squares dual-primal algorithm, Kirchhoff's circuit laws

ACKNOWLEDGEMENTS

本論文得以順利完成，最重要的必須感謝指導教授 王逸琳老師給予的細心指導，使我對最佳化領域有深入的瞭解，並能有所突破。老師所給予的諸多良善的建言，使學生的論文能更臻完備，更讓學生瞭解到做學問所需要的嚴謹態度。論文提案期間承蒙 蔡青志老師對本論文提供研究方向的建議。論文口試期間則承蒙 李宇欣老師、洪政煌老師、以及 黃耀廷老師對本論文提供寶貴的意見，使學生的論文更具內涵與價值。謹此表達我的敬意與謝忱。

在這兩年研究所求學生涯中，感謝 LAB61205 的好伙伴橙坤、姿君、正楠、建傑、姿儀、家宜、志偉、旻哲、立蓁、依巧、乃文、佳琦一起在研究室奮鬥，使得研究室充滿活力與歡笑，使我的研究所生活多采多姿。同時也感謝我協助指導的專題生家瑜，在討論的過程中引發我許多想法，使我的論文更加完善。

最後，感謝父母及家人長期以來的支持與包容，讓我求學能無後顧之憂，完成學業。謹將這份論文以及碩士學位的榮耀與喜悅獻給你們。

Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

In the competitive business environment nowadays, cooperative and competitive relationship among customers, retailers, distributors, manufacturers, and vendors in a supply chain becomes much more complicated to achieve a better supply chain management. In general, manufacturers either produce final products made of materials purchased from several vendors, or put part of their production out to contract with other manufacturers. Thus a final product may be made through several channels in different stages of a supply chain. Identifying the bottleneck for a supply chain becomes an important issue to improve the entire efficiency. In other words, calculating the maximum throughput (i.e. product flow) for either the entire or part of the chain helps us to evaluate the efficiency and capacity of the chain. This problem in fact is a maximum flow problem, which often appears in business applications as an optimization subproblem that seeks the critical or bottleneck processes for performance improvement. In other words, if we can calculate the maximum flow more efficiently, we can solve the original problem in shorter time.

The maximum flow problem has been studied for over five decades and finding efficient solution methods is the most main part in this field. In this chapter, we will first introduce the background and motivation of our study, and then we will propose our objective and assumption. Finally, we will describe the structure of the thesis in the end.

1.1. Background and Motivation

Network flow problem is an important class of optimization problem. There are three fundamental network optimization problems: (1) the shortest path problem, (2) the maximum flow problem, and (3) the minimum cost flow problem (MCF). This thesis focus on the $s-t$ maximum flow problem. In particular, the objective is to find the maximum value of flow in a capacitated network from the source node s and the sink node t. On the other hand, the MCF seeks an optimal arc flow assignments such that the total flow costs is minimized while the node flow balance and arc capacity constraints are maintained. The $s-t$ maximum flow problem can also be modeled as a specialized minimum cost flow problem by adding an uncapacitated artificial arc (t,s) with negative unit arc cost while all other original arcs have zero cost, as shown in Figure 1.1.

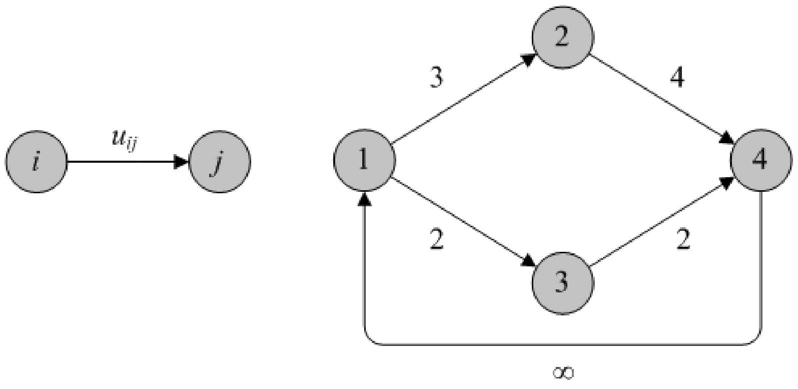


Figure 1.1. A maximum flow example

A *cut* in a network is a set of arcs whose removal separates the network into two parts, one containing the source node and the other containing the sink node. A *minimum cut* is a set of arcs whose sum of capacities is the smallest, and the sum of capacities of minimal cut must equal to the value of maximum flow, which is called the *max-flow min-cut theorem*, proposed by Ford and Fulkerson [**22**]. Indeed, by solving the maximum flow problem, one can also solve the minimum cut problem. The max-flow min-cut problem can be easily expressed as a pair of primal and dual linear programs. Given a capacitated network $G=(N,A,s,t,u)$ where N is the node set with $|N|=n$ and A is the arc set with $|A|=m$. The two special distinguished nodes of the network is the source node s and the sink node t. Let l and u be respectively a lower and an upper bound function that assign to each arc $(i, j) \in A$ a nonnegative integer value l_{ij} and u_{ij}. Let x_{ij} be feasible flow variables denoting flow on each arc $(i, j) \in A$. The primal linear programming (LP) model for the maximum flow problem can be formulated as follows:

$$\max\ x_{ts} \qquad \text{(MAX\_FLOW\_PRIMAL)}$$

$$s.t. \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} x_{ts} & \forall\ i = s \\ 0 & \forall\ i \in N\setminus\{s,t\} \\ -x_{ts} & \forall\ i = t \end{cases} \tag{1.1}$$

$$l_{ij} \le x_{ij} \le u_{ij}\ \forall\ (i,j) \in A \tag{1.2}$$

Let p_i denote the dual variable of equation 1.1 and q_{ij} and r_{ij} respectively denote the dual variables of the upper and lower bound constraint of equation 1.2. The dual formulation can be described as follows.

$$\min \sum_{(i,j)\in A} q_{ij}u_{ij} \qquad \text{(MAX\_FLOW\_DUAL)}$$

$$s.t.\ p_i - p_j + q_{ij} - r_{ij} = \begin{cases} 0 & \forall (i,j) \in A\setminus\{(t,s)\} \\ 1 & \text{for } (i,j) = (t,s) \end{cases} \tag{1.3}$$

$$p_i \ge 0\ \forall i \in N;\ q_{ij}, r_{ij} \ge 0\ \forall (i,j) \in A \tag{1.4}$$

These LP formulations can be solved by the simplex method. With the special structure for the network optimization problems, the maximum flow problem can be solved more efficiently by graphical methods than algebraic methods. Conventional graphical maximum flow algorithms either augment flows via simple paths or ship flows from nodes with excess flows via admissible arcs to their neighbors. In particular, Ford and Fulkerson [**22**] proposed the first maximum flow algorithm called augmenting path method, which iteratively seeks an augmenting path connecting s to t in residual network, ships as much flow as possible to saturate at least one arc, and then repeat these processes until no more augmenting path exists. The generic augmenting path method runs in pseudo-polynomial time (see [**1**] for details), which can be improved to polynomial-time, if the augmenting path is selected as the one with smallest number of arcs at each iteration. On the other hand, the most efficient maximum flow algorithm is the preflow-push method proposed by Goldberg and Tarjan [**35**], which first saturates all arcs emanating from s, then iteratively ships flows from a node with positive excess flow to its neighbor that is one arc closer to t, until no more flow can be shipped to t, and then ships the excess flow back to s. All of these maximum flow algorithms more or less ship flows via shortest paths composed by admissible arcs to achieve better theoretical running time. Such a technique will also be exploited in our proposed maximum flow algorithms.

Recently, Fujishige [**25**] proposed a new maximum flow algorithm using maximum adjacency (MA) orderings by Nagamochi and Ibaraki [**43, 44**]. Unlike conventional maximum flow algorithms, this new maximum flow algorithm augments flow by at least one (and possibly more) augmenting path at each iteration. Wang et al. [**51**] proposed another new maximum flow algorithm based on the least-squares primal-dual algorithm which guarantees nondegenerate pivots when solving LPs. Since both methods augment flow along a subnetwork that contains more augmenting paths at each iteration, they are expected to converge in fewer iterations than conventional methods. Based on similar idea to ship flow along a subnetwork, this thesis investigates different efficient techniques to augment flows via different subnetworks.

1.2. Objective and Proposed Approaches

In this thesis, we first study the Dinic's algorithm (DA) [**18**]. By using the concept of sending flow along paths in the shortest path network, called the *layered network*, we propose three new maximum flow algorithms that augment flow along layered networks.

With different intuitions, our algorithms calculate different augmenting-flow vector, denoted by δ, which is an improving direction to update arc flow from x to $x + \theta \times \delta$, where θ represents a step length and is calculated by minimum ratio test. The δ determined by conventional augmenting-path based algorithms has value 1 along arcs on an augmenting path and 0 for all other arcs. Such δ is straight forward to satisfy the flow balance constraints, so that would correspond to the bottleneck residual capacity along the augmenting path. On the other hand, if all the arcs in a subnetwork have to be used to ship flows, it is not trivial to calculate a δ that guarantees flow balance. Here we propose three different ways to generate different qualified δ. In particular, the proportional arc augmenting algorithm (PAA) intends to ship more (or less) flow along those arcs of larger (or smaller) residual capacities; the flow splitting augmenting algorithm (FSA) equally distributes all the flow entering a node to each of its outgoing arcs in a layered network; and the modified least-squares dual-primal algorithm (MLSDP) solves systems of equations composed by Kirchhoff's circuit laws for δ, so that nondegenerate pivots at each iteration is guaranteed. We will show these three new methods calculate the maximum flow within time strictly polynomial to the number of nodes and arcs (denoted by n and m, respectively) of the input network. We will also prove the complexity of these three algorithms and compare the efficiency with other maximum flow algorithms. Besides, we also try speed-up techniques on the implementations for these new maximum flow algorithms to gain more empirical efficiency.

1.3. Scope and Limitations

This research develops efficient methods for the maximum flow problem. Our methodology is based on the following assumptions and limitations:

(1) The network is directed (i.e. each arc has a direction from its tail node to head node).

(2) We assume no self-loop (i.e. an arc that starts and ends on the same node).

(3) Each arc has zero flow lower bound and a positive integer upper bound. If an arc has unlimited capacity, we will use a very large integer capacity to represent it so that the arc will never be saturated in any case.

(4) All the parameters associated with nodes and arcs are deterministic. That is, we only consider the case where the parameters are given in advance and will not be changed.

(5) Each arc admits a fractional feasible flow. In other words, arc flow is not necessarily integral.

(6) There is only one source node and one sink node. The objective is to solve for the maximum flow from the source node to the sink node.

(7) The amount of the maximum flow is a finite integral number.

1.4. Overview of Thesis

The rest of this thesis is organized as follow. Chapter 2 reviews maximum flow algorithms in literature; Chapter 3 introduces two new maximum flow algorithms called the proportional arc augmenting algorithm and flow splitting augmenting algorithm and proves the complexity can be bounded in polynomial time; Chapter 4 proposes a modified LSDP algorithm and proves its theoretical complexity to be polynomial time; Chapter 5 records and analyzes the results of computational experiments, in which we compare the efficiency of our proposed algorithms with several state-of-the-art maximum flow algorithms over randomly generated test networks of different sizes and topologies; Conclusions and contributions of our works are summarized in Chapter 6, where we also suggest a few topics for future research in related fields.

CHAPTER 2

PRELIMINARIES

We first introduce notations and definitions used in this paper in Section 2.1, and then review and summarize the existing maximum flow algorithms in Section 2.2-2.8.

2.1. Notations and Definitions

We consider the maximum flow problem over a capacitated network $G = (N, A, s, t, u)$. The flow x is a function $x : A \rightarrow R$ satisfying the following equation 2.1 and 2.2:

$$\sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} = \ 0 \quad \forall\, i \in N - \{s,t\} \tag{2.1}$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall \ (i,j) \in A \tag{2.2}$$

where 2.1 is the flow balance constraints of nodes and 2.2 is the capacity constraints of arcs. If we relax the equation 2.1 to be equation 2.3 while satisfying equation 2.2, the *preflow* x is a function $x : A \rightarrow R$ satisfying the equation 2.2 and 2.3.

$$\sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} \geq \ 0 \quad \forall\, i \in N - \{s,t\} \tag{2.3}$$

Given a feasible flow vector x, a residual network $G(x) = (N(x), A(x), s, t, r(x))$ can be constructed as follows: for each arc (i,j) with flow x_{ij}, we replace by two arcs, (i,j) and (j,i) with residual capacities $r_{ij}(x) = u_{ij} - x_{ij}$ and $r_{ji}(x) = x_{ij}$, respectively. An arc $(i,j) \in A(x)$ is saturated when $r_{ij}(x) = 0$. The residual network only contains arcs of positive residual capacities. Figure 2.1 shows an example to construct the residual network $G(x)$.

For each node $i \in N(x)$, we define its *distance label* $d(i)$ as the lower bound on the number of arcs connecting from the source node s to it in $G(x)$. Note that this

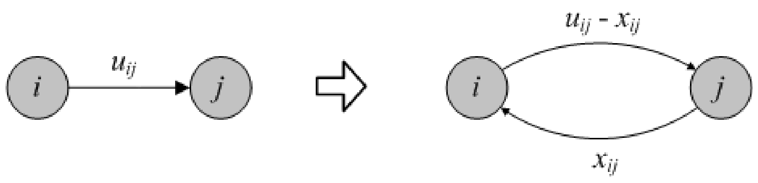


Figure 2.1. Constructing the residual network $G(x)$

definition is different from the conventional one used in literature (e.g. Ahuja and Orlin [**3**]) for measuring the minimum number of arcs connecting to t. We can conduct a BFS on $G(x)$ starting from s to calculate the distance labels, denoted by $d(i)$ for each node i in $N(x)$. Moreover, a distance label is *valid* if it satisfies the two conditions:

$$d(s) = 0 \tag{2.4}$$

$$d(j) \leq d(i) + 1 \text{ for every arc } (i, j) \in A \text{ with } r_{ij}(x) > 0 \tag{2.5}$$

The following two properties show that our modified distance labels can be use in designing maximum flow algorithms.

If a $d(i)$ equals the length of the shortest path from source to node i in the residual network, we say the distance label $d(i)$ is *exact*. For example, in Figure 2.2(a), if node 1 is the source node and node 4 is the sink node, the distance vector $d = (0, 1, 1, 2)$ is an exact distance label. Because the $d(t)$ is the lower bound distance from source to sink along a path in the residual network, $d(t) = \infty$ implies that there exists no path from s to t in the residual network.

Then, we can define the *admissible* arc set, denoted by $A'(x)$, as the set of arcs in $A(x)$ satisfying $d(j) = d(i) + 1$ and $r_{ij} > 0$. A *layered network*, denoted by $G'(x) = (N', A', s, t, r')$, is an admissible subnetwork of $G(x)$ that contains all the admissible arcs A' of $A(x)$ with residual capacity r', and their associated node set N'. For example, in Figure 2.2(a), the arc $(1, 2)$ is an admissible arc because $d(1) = d(2) - 1$ and $r_{12} > 0$. The arc $(2, 3)$ is inadmissible because $d(2) \neq d(3) - 1$ and Figure 2.2(b) shows the admissible subnetwork of Figure 2.2(a). Note that $G'(x)$ is acyclic and there are at

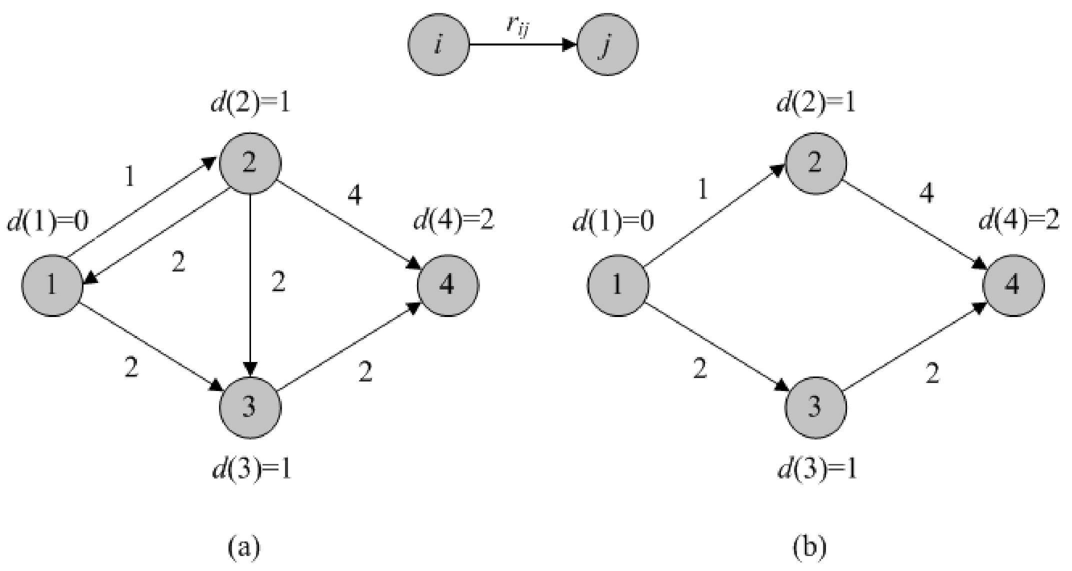


Figure 2.2. An example of residual network and its corresponding admissible subnetwork

most $O(n)$ different layered networks in our algorithms since $d(t)$ in each new layered network increases at least one and $d(i) < n$ for each node i in $N(x)$ at any time.

2.2. Network Simplex Methods

The network simplex method is an effective way to solve the MCF problems. Since the maximum flow problem can be formulated as the MCF problem, Fulkerson and Dantzig [**24**] proposed a specialized the network simplex algorithm for the maximum flow problem. However, this algorithm performs at most nU nondegenerate pivots and converge in $O(n^2U)$ pseudo-polynomial time, where $U = \max_{(i,j)\in A} \{u_{ij}\}$. Later, Goldfarb and Hao [**37**] proposed the polynomial time bound pivot rule, called the admissible pivot rule, which performs at most nm pivots and the total running time can be bounded in $O(n^2m)$. Goldberg et al. [**34**] used the dynamic tree data structure to speed up the Goldfarb-Hao version of the network simplex algorithm and run in at most $O(nm \log n)$ time.

The polynomial time dual network simplex algorithm was proposed by Armstrong et al. [**5**] which required at most $2nm$ pivots and $O(n^2m)$ times. Armstrong et al. also proposed the dual-like simplex method which behaves like a dual simplex method in that the objective function is to be maximized but the basis is not necessarily dual feasible by each pivot. The dual-like algorithm also needs $2nm$ pivots and the total running time can be bounded in $O(n^3)$.

2.3. Augmenting Path Methods

The augmenting path algorithm was first proposed by Ford and Fulkerson [**22**], called *labeling algorithm*. The idea behinds this algorithm is simple: If there exists a path from s to t in the residual network, we send flow along one of these paths and the flow is decided by the minimal residual capacity of arcs in the selected path. The selected path is called the *augmenting path* and the minimal residual capacity of arc will be the *saturated* arc. The labeling algorithms terminates when there exists no more $s-t$ augmenting path in the residual network, at which time the maximum flow has been achieved since saturated arcs form cuts that separates s and t in the residual network. However, the labeling algorithm has a pseudo-polynomial time complexity time bound $O(nmU)$.

To have strongly polynomial time complexity, an augmenting path should be selected in a shortest path fashion. Edmonds and Karp [**20**] suggested two polynomial time implementations of the labeling algorithm. The first implementation, called the maximum capacity augmenting path algorithm, sends flow along a path with maximum residual capacity and performs $O(m \log U)$ augmentations. The second implementation runs in $O(nm^2)$ time if a flow is augmented along the shortest path identified by a breadth-first-search (BFS) at each iteration. Independently, Dinic [**18**] proposed an $O(n^2m)$ time algorithm which constructs a layered network (i.e. an admissible subnetwork composed by shortest augmenting paths) and ships flow along each s-t path (which would be the shortest) in layered networks at each iteration. Later, Malhotra et al. [**42**] selected the *reference node* that has the minimum throughput in a layered network, and constructed augmenting paths through it to calculate the maximum flow in $O(n^3)$ time.

We briefly introduce the DA in Figure 2.3. First DA uses the BFS to construct a layered network and finds paths to augment flow until there exists no path in current layered network. Then DA reconstructs a new layered network in the current residual network and repeats to augment flow along paths. Each step to construct a layered network and augment flows along current layered network takes $O(nm)$ time and the number of constructing layered networks is at most n. Therefore, the total running time of DA is $O(n^2m)$.

Algorithm 1 *Dinic's algorithm*

```
$x := 0$
Do while
    using BFS starting from $s$ to construct a layered network $L$;
    If $t$ is not reached then
        Return $x$;
    Else then
        While $L$ contains a directed path from node $s$ to node $t$ do
            identify an augmenting path $P$ from $s$ to $t$ in $L$;
            $\delta := \min\{r_{ij} : (i,j) \in P\}$;
            augment $\delta$ units of flow along $P$ and update $L$;
        End while;
End while;
```

Figure 2.3. Dinic's algorithm

There are various approaches to speed up DA. Even and Itai [**21**] combined BFS and depth-first-search (DFS) which constructs a layered network by BFS and finds augmenting paths by DFS. Sleator and Tarjan [**48**] used the dynamic tree data structure to achieve a theoretical running time as $O(nm \log n)$. Without explicitly constructing layered networks, Ahuja and Orlin [**3**] used the distance label technique introduced by Goldberg and Tarjan [**35**] and proposed a distance-directed algorithm which updates distance labels whenever necessary to improve the practical efficiency of Dinic's algorithm.

2.4. Preflow Push Methods

The *preflow-push algorithms* are more efficient than augmenting path algorithms in both theory and practice. The preflow-push algorithms sends flow along arcs rather than paths. The first preflow concept was proposed by Karzanov [**40**] which obtains a blocking flow in a layered network in $O\left(n^2\right)$ time and total running time is $O\left(n^3\right)$. Galil [**28**] improved the time bound of Karzanov's algorithm in $O\left(n^{\frac{5}{3}}m^{\frac{2}{3}}\right)$.

Because flow conservation is not always satisfied in preflow-push algorithms, it may temporarily store some flow on the nodes and these nodes are called *excess* nodes. Given a flow x, we define the excess $e\left(i\right)$ of each node $i \in N - \{s,t\}$ is:

$$e\left(i\right) = \sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} \tag{2.6}$$

A node is *active* if it has a positive excess. The basic operation of preflow-push algorithms is to select an active node and try to send the excess flow toward adjacent nodes in the residual network. Because we want to send *excess* flow to t, we select the adjacent nodes that are closer to t. Goldberg and Tarjan [**35**] introduced distance labels and used the concept of preflow to establish the $O\left(n^2m\right)$ generic preflow-push algorithm. In particular, the preflow-push algorithm first saturates arcs emanating from s, allows active nodes to store temporary excess flow, iteratively pushes flow from an active node to its neighbor that is one arc closer to t according to distance labels, until no more flow shippable to t, and then pushes the excess flow back to s. They also used some data structure techniques to improve efficiency. The first-in, first-out (FIFO) preflow-push algorithm uses the concept of queue and can be implemented in $O\left(n^3\right)$ time. The dynamic tree data structure implementation improves the time bound to $O(nm\log(\frac{n^2}{m}))$. The highest label preflow-push algorithm proposed by Cheriyan and Maheshwari [**10**] pushes flows from an active node of the highest distance label and can be implemented in $O\left(n^2\sqrt{m}\right)$ time.

Recently, Cerulli et al. [**7**] merged the preflow-push and augmenting path algorithms and proposed the *budget algorithm* to avoid the *ping-pong* effect, which is the main drawback of the preflow-push algorithms. The ping-pong effect appears in the relabel process that attempts to let the active nodes be nonactive but without increasing any preflow value between these nodes. When sending flow from an active node i with distance label $d(i)$, the budget algorithm first checks this flow can reach a node l such that $d(i) - d(l) \geq k$, for a given constant k. Then a path is constructed by admissible arcs from node i to a node l and sending flow along this path. If $k = 0$, the budget algorithm becomes the highest label preflow-push and the running time is $O(n^2\sqrt{m})$.

2.5. Scaling Methods

The *scaling method* is a simple and efficient technique that can be applied in conjunction to other maximum flow algorithms. The concept of the scaling algorithm is similar to the maximum capacity augmenting path algorithm that augments flow along a path with a sufficiently large residual capacity. A threshold Δ in either distance label or residual capacity defines a scaling phase, in which only qualified arcs of larger residual capacity or nodes of larger distance label are kept to construct a smaller subnetwork for shipping flows. When no more flow can be shipped during a scaling phase (which means Δ-optimal), the threshold is decreased to enter the next scaling phase. The procedures repeat until Δ is sufficiently small.

Let $\Delta = 2^{\lfloor \log_2 U \rfloor}$, Gabow [**27**] introduced the bit-scaling algorithm using DA can run in $O(nm \log U)$. Similarly, Ahuja and Orlin [**3**] proposed a capacity scaling algorithm with the same worst case complexity based on a distance-directed method.

Ahuja and Orlin [**2**] incorporated the scaling phase with the generic preflow-push called the *excess scaling algorithm* with $O(nm + n^2 \log U)$ complexity. Then, Ahuja et al. [**4**] proposed two modifications of the excessing scaling algorithm. The first is the *stack scaling algorithm* which pushes flow from a node of large excess with the highest distance label and obtains a small improvement in running time to $O\left(nm + n^2\frac{n\log U}{\log\log U}\right)$. The second is using the dynamic tree data structure and gets the time bound in $O\left(nm\log\left(\frac{n\log U}{m\log\log U}+2\right)\right)$. Later, based on the excess scaling algorithm, Cheriyan et al. [**9**] introduced an incremental scaling algorithm which can be run in $O\left(\frac{n^3}{\log n}\right)$. Sedeño-Noda and González-Martín [**45**] proposed the *two-phase capacity scaling algorithm* which uses the capacity scaling algorithm in the first phase and labeling algorithm in the second phase and runs in $O\left(nm\log\left(\frac{U}{n}\right)\right)$ time.

2.6. Maximum Adjacency Ordering Methods

The *MA ordering method* by Nagamochi and Ibaraki [**43, 44**] is an efficient way to solve the graph connectivity problem by identifying the minimum cut between some node pair. Because the MA ordering is similar to the Dijkstra's shortest path algorithm, it can be adapted Dijkstra's shortest path algorithm with the Fibonacci heap and run in $O\left(m + n\log n\right)$ time. Fujishige [**25**] showed an application of MA ordering to the maximum flow problem to get a polynomial time algorithm. It can be regarded as the acceleration of Edmonds and Karp [**20**] maximum capacity augmenting path algorithm and reduces the total number of augmentation to $O\left(n\log nU\right)$. So the MA ordering maximum flow algorithm runs in $O\left(n\left(m+n\log n\right)\log nU\right)$ time. Fujishige and Isotani [**26**] also proposed a scaling version in [**25**] and showed its complexity to be $O(nm\log U)$.

Later, Matsuoka and Fujishige [**41**] improved the original version of the MA ordering algorithm by using preflows with the same time bound. This new algorithm contains two steps. The first step seeks a preflow of the maximum value in the residual network, which corresponds to an infeasible optimal solution. The second step converts the preflow to a feasible flow of the maximum value. Although their improved algorithm has the same theoretical complexity, in practice it performs better than the highest label preflow-push algorithm on some special networks.

2.7. Least-square Dual-primal Method

The least-squares dual-primal (LSDP) algorithm is modified from the least-squares primal-dual (LSPD) algorithm by exchanging the roles of the primal and dual formulations. LSPD method is a strictly improving method that maintains primal feasible until the complementary dual solution is feasibility for solving LPs. Because of the special constraint structure of the network problem, the LSPD method can solve the network problem efficiently. Gopalakrishnan et al. [**30**] proposed the least-squares network flow (LSNF) algorithm for solving the MCF problem. Wang [**50**] applied the LSPD method for solving one-to-one and one-to all shortest path problems and showed that LSPD gives exactly the same steps with the Dijkstra's algorithm. Wang et al. [**51**] modified the LSPD algorithm which exchanges the role of primal and dual and proposed the LSDP algorithm for solving the maximum flow problem without any degenerate pivots. In LSDP, many phase-I nonnegative least-squares (NNLS) subproblems are iteratively solved for improving the flows nondegenerately. Instead of directly solving the quadratic programming problem (NNLS), the NNLS subproblem can also be solved by the Kirchhoff's laws for current on some electrical network with diodes. We briefly introduce the LSDP algorithm for solving the maximum flow problem and how the NNLS subproblem can transform to the electrical network problem and use the Kirchhoff's law to solve it.

The LSDP algorithm for solving the maximum flow problem can be divided into two steps. The first step solves NNLS subproblem and to find a feasible improving direction δ for each arc. The second step finds the maximum step length θ by calculating the augmenting flow $\Delta x_{ij} = \theta \times \delta_{ij}$ for each arc (i, j). The NNLS subproblem is based on the dual formulation of the maximum flow problem. Let $L(x) = \{(i, j) : x_{ij} = l_{ij}, \forall (i, j) \in A\}$ and $U(x) = \{(i, j) : x_{ij} = u_{ij}, \forall (i, j) \in A\}$ be the set of arcs containing flows of the lower and the upper bounds, respectively. Let $F(x) = \{(i, j) : l_{ij} < x_{ij} < u_{ij}, \forall (i, j) \in A\}$ be the set of arcs containing flow between lower and upper bounds. For each arc (i,j), let δ_{ij} be the artificial variable associated with equation 1.3 and define $\alpha_{ij} = 1$ if $x_{ij} = u_{ij}$, and $\alpha_{ij} = 0$, otherwise; $\beta_{ij} = 1$ if $x_{ij} = l_{ij}$ and $\beta_{ij} = 0$, otherwise. The NNLS subproblem can be formulated as follows:

$$\min \sum_{(i,j)\in A} \delta_{ij}^2 \tag{NNLS}$$

$$s.t.\ p_i - p_j + \alpha_{ij} q_{ij} - \beta_{ij} r_{ij} + \delta_{ij} = \begin{cases} 0 & \forall (i,j) \in A \setminus \{(t,s)\} \\ 1 & \text{for } (i,j) = (t,s) \end{cases} \tag{2.7}$$

$$p_i \geq 0\ \forall i \in N;\ q_{ij} \geq 0\ \forall (i,j) \in U(x);\ r_{ij} \geq 0\ \forall (i,j) \in L(x) \tag{2.8}$$

Summing up equation 2.7 along arcs in a cycle C will cancel out p. Wang et al. [**51**] showed that variables q_{ij} and r_{ij} will not affect the optimal solution of NNLS, and thus can be removed. Let Q be the set of all cycles in $G(x)$ and we can rewrite the NNLS subproblem as follows:

$$\min \sum_{(i,j)\in A} \delta_{ij}^2 \tag{NNLS\_CYCLE}$$

$$s.t. \sum_{(i,j)\in C\cap F(x)} \delta_{ij} + \sum_{(i,j)\in C\cap U(x)} \delta_{ij} + \sum_{(i,j)\in C\cap L(x)} \delta_{ij} = \begin{cases} 0 & \text{if } (t,s) \notin C \\ 1 & \text{if } (t,s) \in C \end{cases},\ \forall C \in Q \tag{2.9}$$

$$\delta_{ij} : \text{free } \forall (i,j) \in F(x);\ \delta_{ij} \leq 0\ \forall (i,j) \in U(x),\ \delta_{ij} \geq 0\ \forall (i,j) \in L(x) \tag{2.10}$$

The subproblem NNLS_CYCLE can be viewed as an optimization problem over a physical electrical network. Let $G''(x) = (N, A''(x))$ where $A''(x) = F(x) \cup U(x) \cup L(x)$ and δ_{ij} be the electron flow along arc (i,j). We can image that there is a one-unit resistance on each arc $(i,j) \in A''(x)$; a diode on each arc $(i,j) \in L(x)$ and $(j,i) \in U(x)$ to restrict the orientation of δ_{ij} from node i to node j; a unit-voltage electric power source on arc (t,s). The NNLS_CYCLE subproblem is equivalent to seeking the equilibrium electron flow in the electrical network and we can use the Kirchhoff's current law (KCL) and Kirchhoff's voltage law (KVL) to solve the NNLS_CYCLE subproblem. The KCL contains $n-1$ equations where each one associated with a node $i \in N$ has a form as

$$\sum_{(i,j)\in A'} \delta_{ij} - \sum_{(j,i)\in A'} \delta_{ji} = 0 \tag{2.11}$$

and the KVL contains $m-(n-1)$ equations where each one associated with a cycle $C \in Q$ has a form as

$$\begin{aligned} \sum_{(i,j)\in C \text{ \& same orientation}} \delta_{ij} \quad - \sum_{(j,i)\in C \text{ \& opposite orientation}} \delta_{ij} = 0, \text{ if } (t,s) \notin C \\ \sum_{(i,j)\in C \text{ \& same orientation}} \delta_{ij} \quad - \sum_{(j,i)\in C \text{ \& opposite orientation}} \delta_{ij} = 1, \text{ if } (t,s) \in C \end{aligned} \tag{2.12}$$

We give a small example to illustrate how the Kirchhoff's law can be used to solve electron flow in a given network. Figure 2.4(a) is the original network which can be converted to an electrical network in 2.4(b) by setting a unit resister to each arc and a unit voltage battery to arc (t,s). Then we formulate the KCL and KVL linear system as 2.13 to solve for electron flow $\delta = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{2}{4}\right)$.

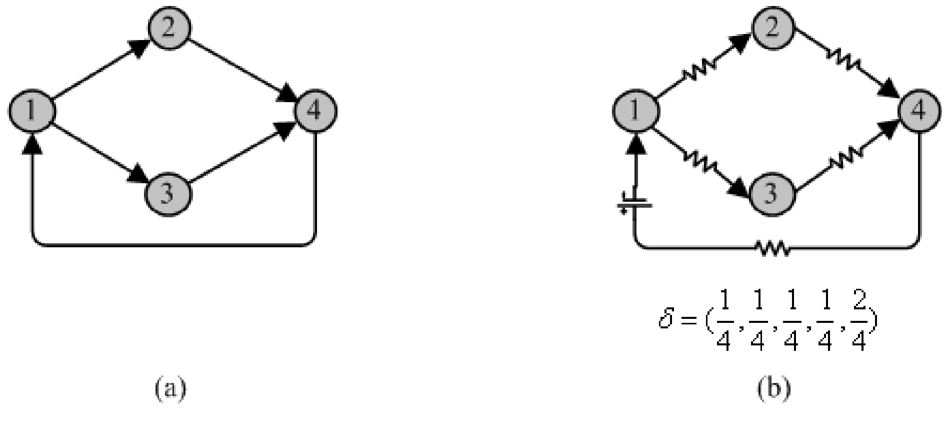

1
2
3
4
(a)
1
2
3
4
$\delta = (\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{2}{4})$
(b)

Figure 2.4. An example of solving electron flow by using the Kirchhoff's laws

$$\begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ -1 & 1 & -1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \delta_{12} \\ \delta_{13} \\ \delta_{24} \\ \delta_{34} \\ \delta_{41} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.13}$$

Wang et al. [**51**] proposed a LSDP maximum flow algorithm called DR, as illustrated in Figure 2.5. First, DR will calculate the electron flow δ by using Kirchhoff's laws. If all electron flows are nonnegative, then δ is a feasible improving direction to augment flow x. Otherwise, when there exists some electron flows such that $\delta_{ij} < 0$ and $(i, j) \in L(x)$ or $\delta_{ij} > 0$ and $(i, j) \in U(x)$, we have to adjust the sets of arcs $A''(x)$.

```
Procedure 2 $DR(G'(x), \delta)$
$A''(x) = A'(x)$;
$\delta := 0$;
While there exists some paths from $s$ to $t$ in $G''(x) = (N, A''(x))$ do
    solve $\delta$ by KCL and KVL
    If $\delta_{ij} > 0 \; \forall (i,j) \in L(x)$ and $\delta_{ij} < 0 \; \forall (i,j) \in U(x)$ then Break;
    Else then
        $p_s := 0$ and $p_j := p_i + \delta_{ij} \; \forall i \in N, \forall (i,j) \in A'(x)$;
        If arc $(i,j) \in L(x)$, $(i,j) \notin A''(x)$ and $p_j > p_i$ then
            $A''(x) := A''(x) \cup \{(i,j)\}$;
        Else if arc $(i,j) \in U(x)$, $(i,j) \notin A''(x)$ and $p_j < p_i$ then
            $A''(x) := A''(x) \cup \{(i,j)\}$;
        Else then
            select arc $(k,l) = \max\{|\delta_{ij}|\}$;
            $A''(x) := A''(x) \setminus \{(k,l)\}$;
End while;
```

Figure 2.5. DR algorithm

First we calculate the electrical potential p_i for each node $i \in N$. Then we check the arcs was disconnected during some inner steps. If $(i, j) \in L(x)$ and $p_j > p_i$ or $(i, j) \in U(x)$ and $p_i > p_j$, we know that the electrical potential forces the electron flow to pass through these arcs. As a result, those arcs should be reconnected to $A''(x)$. On the other hand, those arcs such that $\delta_{ij} < 0$ and $(i, j) \in L(x)$ or $\delta_{ij} > 0$ and

$(i, j) \in U(x)$ should have electron flows move along the opposite way of their original orientation, which is impossible, and thus such arcs should be disconnected. Among those arcs to be disconnected, DR will select the one with the most electron flow to disconnect. Then DR repeats the steps to obtain a nonnegative electron flow and augment flow.

We briefly illustrate how algorithm DR works by solving a small maximum flow example. In Figure 2.6(a), we would like to send as much flow as possible from 1 to 4. Thus $N = \{1, 2, 3, 4\}$ and $A''(x) = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (4, 1)\}$. Starting with $x = (0, 0, 0, 0, 0, 0)$, we construct an electrical network as illustrated in Figure 2.6(b). The electron flow on $A''(x)$ is $\delta = (\frac{1}{4}, \frac{1}{4}, 0, \frac{1}{4}, \frac{1}{4}, \frac{2}{4})$ and the maximum step length is $\theta = \min\{\frac{2}{\frac{1}{4}}, \frac{3}{\frac{1}{4}}, \frac{4}{\frac{1}{4}}, \frac{6}{\frac{1}{4}}\} = 8$ which can be used to improve flow to be $x = (2, 2, 0, 2, 2, 4)$.

At the second iteration, because arc $(1, 2)$ is saturated, we reverse its orientation and add it to $U(x)$. A new $A''(x) = \{(2, 1), (1, 3), (2, 3), (2, 4), (3, 4), (4, 1)\}$ is constructed and the electron flow on $A''(x)$ is $\delta = (\frac{1}{4}, \frac{1}{4}, 0, \frac{1}{4}, \frac{1}{4}, \frac{2}{4})$. Since the electron flow on arc $(2, 1)$ moves in its opposite orientation, the arc $(2, 1)$ has to be disconnected. We then solve the current $A''(x)$ to get $\delta = (0, \frac{3}{8}, \frac{-1}{8}, \frac{1}{8}, \frac{2}{8}, \frac{3}{8})$. The electron flow on arc $(2, 3)$ also has to be disconnected. Algorithm DR continues to solve for $A''(x) = \{(1, 3), (2, 4), (3, 4), (4, 1)\}$ and obtains $\delta = (0, \frac{1}{3}, 0, 0, \frac{1}{3}, \frac{1}{3})$ in Figure 2.6(d). This electron flow is feasible and the maximum step length $\theta = 3$ is calculated to achieve a better feasible flow $x = (2, 3, 0, 2, 3, 5)$.

In Figure 2.6(e), arc $(1, 3)$ is also saturated, so we reverse its orientation and add it to $U(x)$. Thus $A''(x) = \{(2, 1), (3, 1), (2, 3), (2, 4), (3, 4), (4, 1)\}$. When using the KCL and KVL to solve for electron flow on $A''(x)$, algorithm DR disconnects arc $(2, 1)$, $(2, 3)$, $(3, 1)$. The remaining electrical network has no more path which connects from 1 to 4 as illustrated in Figure 2.6(f), thus $\delta = 0$. Algorithm DR then terminates and the optimality is attained.

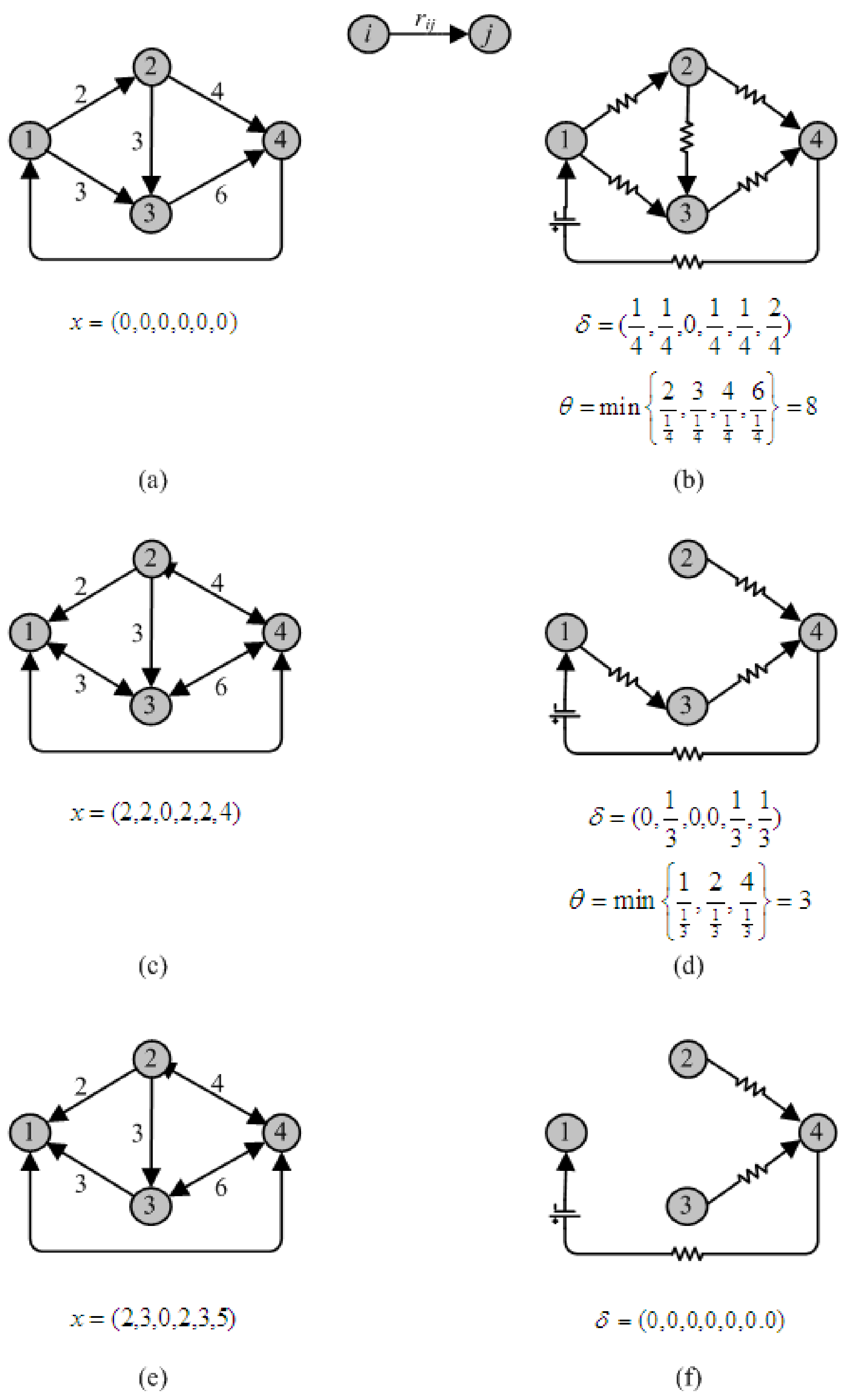

r_{ij}
$x = (0,0,0,0,0,0)$
$\delta = (\frac{1}{4},\frac{1}{4},0,\frac{1}{4},\frac{1}{4},\frac{2}{4})$
$\theta = \min\left\{\frac{2}{\frac{1}{4}},\frac{3}{\frac{1}{4}},\frac{4}{\frac{1}{4}},\frac{6}{\frac{1}{4}}\right\} = 8$
(a)
(b)
$x = (2,2,0,2,2,4)$
$\delta = (0,\frac{1}{3},0,0,\frac{1}{3},\frac{1}{3})$
$\theta = \min\left\{\frac{1}{\frac{1}{3}},\frac{2}{\frac{1}{3}},\frac{4}{\frac{1}{3}}\right\} = 3$
(c)
(d)
$x = (2,3,0,2,3,5)$
$\delta = (0,0,0,0,0,0.0)$
(e)
(f)

Figure 2.6. An example of LSDP for max-flow problem

2.8. Summary

Basic network notations about the maximum flow problem and literature survey on various important maximum flow algorithms are introduced and summarized in this chapter. Table 2.1 summarizes the complexities for different maximum flow algorithms introduced in this chapter.

Table 2.1. Summary of maximum flow algorithms

Algorithm	**Complexity**	**Reference**
Fulkerson and Dantzig (1955)	$O\left(n^2U\right)$	[**24**]
Ford and Fulkerson (1956)	$O\left(nmU\right)$	[**22**]
Dinic (1970)	$O\left(n^2m\right)$	[**18**]
Edmonds and Karp(1972)	$O\left(nm^2\right)$	[**20**]
Karzanov (1974)	$O\left(n^3\right)$	[**40**]
Malhotra et al. (1978)	$O\left(n^3\right)$	[**42**]
Galil (1980)	$O\left(n^{\frac{5}{3}}m^{\frac{2}{3}}\right)$	[**28**]
Sleator and Tarjan (1983)	$O\left(nm\log n\right)$	[**48**]
Tarjan (1984)	$O(n^3)$	[**49**]
Gabow (1985)	$O\left(nm\log U\right)$	[**27**]
Goldberg (1985)	$O(n^3)$	[**31**]
Goldberg and Tarjan (1986)	$O(nm\log(\frac{n^2}{m}))$	[**35**]
Ahuja and Orlin (1989)	$O\left(nm + n^2\log U\right)$	[**2**]
Ahuja et al. (1989)	$O\left(nm\log\left(\frac{n\log U}{m\log\log U}+2\right)\right)$	[**4**]
Cheriyan and Maheshwari (1989)	$O\left(n^2\sqrt{m}\right)$	[**10**]
Goldfarb and Hao (1990)	$O\left(n^2m\right)$	[**37**]
Ahuja and Orlin (1991)	$O\left(nm\log U\right), O(n^2m)$	[**3**]
Goldberg et al. (1991)	$O(nm\log n)$	[**34**]
Cheriyan et al. (1996)	$O\left(\frac{n^3}{\log n}\right)$	[**9**]
Armstrong et al. (1998)	$O\left(n^3\right)$	[**5**]
Sedeño-Noda and González-Martín (2000)	$O\left(nm\log\left(\frac{U}{n}\right)\right)$	[**45**]
Fujishige (2003)	$O\left(n\left(m + n\log n\right)\log nU\right)$	[**25**]
Fujishige and Isotani (2003)	$O\left(nm\log U\right)$	[**26**]
Matsuoka and Fujishige (2005)	$O\left(n\left(m + n\log n\right)\log nU\right)$	[**41**]
Wang et al. (2006)	unknown	[**51**]
Cerulli et al. (2008)	$O\left(n^2\sqrt{m}\right)$	[**7**]

n : the number of nodes, m : the number of arcs, $U = \max\limits_{(i,j)\in A} |u_{ij}|$

CHAPTER 3

ALGORITHMS TO FAIRLY AUGMENT FLOWS

In this chapter, we will introduce two new maximum flow algorithms called the proportional arc augmenting (PAA) algorithm and flow splitting augmenting (FSA) algorithm. Unlike previous algorithms that only ship flow along part of the admissible arcs, PAA and FSA augment flow via all arcs in a layered network (thus these arcs are implicitly admissible) at each iteration. To do so, the calculated δ has to satisfy the flow conservation constraint. To this end, for each node PAA assigns the flow it receives to each of its outgoing arcs according to the relative proportion of its residual capacity, so that arcs of larger (or smaller) residual capacities will receive more (or less) flow. On the other hand, for each node FSA equally distributes the flow it receives to each of its outgoing arcs. Therefore, we say both PAA and FSA augment flows in different "fair" senses. We first give an introduction of PAA algorithm for the maximum flow problem in section 3.1. We examine the complexity of PAA algorithm in section 3.2. Section 3.3 gives a small example to show how PAA solve the maximum flow problem. Since PAA will obtain the fractional flow by solving the maximum flow problem, we also propose the flow decomposition method to transform the fractional flow to integral flow in section 3.4. In section 3.5, we propose three methods trying to improve the efficiency of PAA. Section 3.6 we propose another algorithm called flow splitting augmenting (FSA) algorithm which is similar to PAA. Section 3.7 gives a summary of PAA algorithm.

3.1. Proportional Arc Augmenting Algorithm (PAA)

Our proposed PAA algorithm exploits an algorithmic framework similar to DA [**18**]. In particular, in each outer iteration, a layered network $G'(x)$ is constructed and solved for its maximum flow, until no more layered network can be constructed. To solve the maximum flow for each layered network, DA seeks an augmenting-flow vector, denoted by $\delta = [\delta_{ij}]$ for each admissible arc $(i, j) \in A'$, to augment flows along single augmenting path P at each inner iteration. In other words, $\delta_{ij} = 1$ for each $(i, j) \in P$ and $\delta_{ij} = 0$ for each $(i, j) \in A' - P$. Then DA finds the maximum step length $\theta = \min \{r_{ij}\ \forall (i, j) \in P\}$ to improve flow $\Delta x = \theta \times \delta$. On the other hand, the augmenting-flow vector by PAA guarantees $\delta_{ij} > 0$ for each $(i, j) \in A'$ based on the relative proportions of the residual capacities between each arcs emanating from the same node in layered networks. If arc (i, j) has a large residual capacity, it will be distributed a large δ_{ij}. For example, if the residual capacity of arc (i, j) is twice as large as the capacity of arc (i, k), we will distribute unit augmenting-flow on arc (i, j) and (i, k) based on the ratio of $2 : 1$. In a sense, PAA may tend to ship more flows than conventional augmenting-path based algorithms at each inner iteration, since all the admissible arcs are used to ship flows rather than via only those admissible arcs on a single path. Detailed pseudo-code of PAA is illustrated in Figure 3.5, which contains four major procedures that will be described in the following paragraphs and Figure 3.1-3.4.

3.1.1. Steps and Properties of PAA

PAA requires constructing or updating a layered network in each iteration, which can be done by procedure *construct-layered-network* in Figure 3.1 and procedure *delete-node* in Figure 3.2.

Let $out(i)$ and $in(i)$ denote the number of outgoing and incoming admissible arcs for each admissible node. Perform a BFS on $G'(x)$ starting from s to calculate distance labels and record the traversal order for each node in N'. Based on the distance label of each node, we can construct the admissible arcs set A' and record $out(i)$ and $in(i)$ for each node $i \in N'$. If $out(i) = 0$ and $i \in N'$, which means node i can not reach

t along any path, thus we should remove node i by adding it to DEL_LIST. After conducting BFS, we use procedure *delete-node* to remove those nodes in DEL_LIST and their associated arcs from N' and A', respectively. The algorithm thus constructs a layered network and store N' and A' by the BFS traversal order (which corresponds to a topological ordering as well).

```
Procedure 3 Construct-layered-network($G(x), out, in, d, S, \Delta$)
$LIST := \{s\}$; $N' := \{s\}$; $DEL\_LIST := \varnothing$;
$d(s) := 0; A' := \varnothing; \Delta = 0; reach(s) := 1$;
While $LIST \neq \varnothing$ do
    select the first node $i$ in $LIST$;
    If $reach(j) = 0$ and $r_{ij} > \Delta$ then
        mark node $j$;$LIST := LIST \cup j$;
        $d(j) := d(i) + 1; A' := A' \cup \{(i,j)\}$;
        $out(i) := out(i) + 1; in(j) := in(j) + 1; S(j) := S(j) + r_{ij}$;
    Else if $reach(j) = 1$, $d(j) = d(i) + 1$, and $r_{ij} > \Delta$ then
        $A' := A' \cup \{(i,j)\}; out(i) := out(i) + 1; in(j) := in(j) + 1; S(j) := S(j) + r_{ij}$;
    If $out(i) = 0$ then $DEL\_LIST := DEL\_LIST \cup i$;
    Else then $N' := N' \cup i$;
    $LIST := LIST \setminus i$;
End while;
If $DEL\_LIST \neq \varnothing$ then Delete-node($DEL\_LIST, G'(x), out, in$);
Return $G' = (N', A')$;
```

Figure 3.1. Pseudo-code for constructing a layered network

```
Procedure 4 Delete-node($DEL\_LIST, G'(x), out, in, S$)
While $DEL\_LIST \neq \varnothing$ do
    select the first node $k$ in $DEL\_LIST$;
    For each arc $(i,k) \in A'$ do
        If arc $(i,k) \in A$ then $out(i) := out(i) - 1$;
        Else then $in(i) := in(i) - 1; S(i) := S(i) - r_{ik}$;
        $A' := A' \setminus \{(i,k)\}$;
        If $out(i) = 0$ or $in(i) = 0$ then
            $DEL\_LIST := DEL\_LIST \cup i; N' := N' \setminus i$;
    For each arc $(k,j) \in A'$ do
        If arc $(k,j) \in A$ then $in(j) := in(j) - 1; S(j) := S(j) - r_{kj}$;
        Else then $out(j) := out(j) - 1$;
        $A' := A' \setminus \{(k,j)\}$;
        If $out(j) = 0$ or $in(j) = 0$ then
            $DEL\_LIST := DEL\_LIST \cup j; N' := N' \setminus j$;
    $DEL\_LIST := DEL\_LIST \setminus k$;
End while;
```

Figure 3.2. Pseudo-code to delete nodes not accessible from s or to t

Let $b(s) := 1$ and $b(i) := \sum_{(i,j)\in A'} \delta_{ij}$ represent the sum of unit augmenting-flow for each admissible node i other than s, respectively. To satisfy the flow balance constraint, $b(i)$ has to be distributed to each admissible arc (i, j) emanating from i. Based on the intuition to distribute more (or less) flow to those arcs of larger (or smaller) residual capacities, for each $(i, j) \in A'$, procedure *calculate-proportional-flow* in Figure 3.3 calculates δ_{ij} by $\frac{r_{ij}}{S(i)} \times b(i)$, where $S(i) := \sum_{(i,j)\in A'} r_{ij}$ is the sum of residual capacities for all admissible arcs outgoing from i. Note that these operations can be conducted for each admissible node i, starting from s, according to the BFS traversal order obtained in procedure *construct-layered-network*. Since the layer network is acyclic and the BFS traversal order selects a node in topological ordering, it suffices to calculate $b(\cdot)$ and $S(\cdot)$ by scanning nodes in BFS traversal order for only one time. In particular, one can directly add δ_{ij} to $b(j)$ whenever δ_{ij} is calculated, since all incoming arcs to j will be scanned before we select j, by the BFS traversal order.

```
Procedure 5 Calculate-augmenting-flow($G'(x), b, \delta$)
$b(s) := 1; order := 1; i := 0; \theta := \infty$;
While $i \neq t$ and $i \in N'$ do
    $i := N'(order)$;
    For each arc $(i, j) \in A'$ do
        $\delta_{ij} := \frac{r_{ij}}{S(i)} \times b(i); b(j) := b(j) + \delta_{ij}$;
    If $\frac{S(i)}{b(i)} < \theta$ then $\theta := \frac{S(i)}{b(i)}$;
    $order := order + 1$;
End while;
Return $\theta$;
```

Figure 3.3. Pseudo-code for calculating the augmenting-flow vector δ

Note that δ is calculated by distributing the unit flow into s to each admissible arc. Thus, if we change the flow into s to be θ units, each admissible arc (i, j) will augment $\theta \times \delta_{ij}$ units of flow. Therefore the maximum step length θ can be calculated by a minimum ratio test $\theta := \min_{(i,j)\in A'} \left\{\frac{r_{ij}}{\delta_{ij}}\right\}$. Such a procedure takes $O(m)$ time, which can be further improved to $O(n)$ time. The following 1 shows that the step length for each node $i \in N'$ can be calculated by $\frac{S(i)}{b(i)}$.

Lemma 1. *Whenever an arc (i,j) is saturated by a step length θ_{ij}, all other neighboring arcs emanating from the same tail node i will also be saturated. In other words, all the arcs emanating from the same node i will be saturated by the same step length $\theta(i) = \frac{r_{ij}}{\delta_{ij}} = \frac{S(i)}{b(i)}$.*

Proof. To saturate arc (i,j), the step length has to be $\theta_{ij} = \frac{r_{ij}}{\delta_{ij}} = \frac{r_{ij}}{\frac{r_{ij}}{S(i)} \times b(i)} = \frac{S(i)}{b(i)}$. Note that this value only depends on $S(i)$ and $b(i)$. In other words, if we set the step length to $\frac{S(i)}{b(i)} = \theta(i)$, other arcs (i,k) also will also be saturated. □

Details of procedure *calculate-augmenting-flow* are illustrated in Figure 3.3. In fact, this nice property is a by-product of the intuition to ship flow based on the relative proportions of the residual capacities between each arc emanating from the same node. As a result, an admissible arc becomes bottleneck (i.e. saturated) if and only if each arc emanating from the same tail node becomes bottleneck. That is, whenever arc (i,j) is saturated, we can directly remove node i and its associated arcs in the layered network. Note that once an arc is removed from a layered network, its orientation will be reversed, which makes any path containing that arc has length longer than the shortest path and thus such an arc will never be considered in current layered network.

```
**Procedure 6** *Augment-flow*$(G'(x), \delta, b, S, \theta)$
$DEL\_LIST := \varnothing$;
**For** each arc $(i,j) \in A'$ **do**
    **If** arc $(i,j) \in A$ **then** $x_{ij} := x_{ij} + \delta_{ij} \times \theta$;
    **Else then** $x_{ij} := x_{ij} - \delta_{ij} \times \theta$;
**For** each node $i \in N'$ **do**
    $S(i) := S(i) - \theta \times b(i)$;
    **If** $S(i) = 0$ **then** $DEL\_LIST := DEL\_LIST \cup i$;
**Return** $DEL\_LIST$;
```

Figure 3.4. Pseudo-code for augmenting flow

After calculating θ, procedure *augment-flow* updates arc flow from x to $x + \theta \times \delta$, as illustrated in Figure 3.4. Note that we can directly update $S(i)$ by $S(i) - \theta \times b(i)$, instead of calculating them from the scratch by definition. If $S(i)$ decreases to zero, all the admissible arcs emanating from node i are saturated and we add node i to DEL_LIST. Then we update by procedure *delete-node*, as illustrated in Figure 3.2.

We conclude our PAA algorithm in Figure 3.5:

```
Algorithm 7 $Proportional\_arc\_augmenting(G(x))$
Initialization:
    For each node $i \in N$ do
        $out(i) := 0; in(i) := 0; d(i) := \infty;$
$G'(x) :=$ Construct-layered-network$(G(x), out, in, d, S, \Delta)$;
While $d(t) \neq \infty$ do
    While $s \in N'$ and $t \in N'$ do
        For each arc $(i,j) \in A'$ do $\delta_{ij} := 0$;
        $\theta :=$ Calculate-proportional-flow$(G'(x), b, \delta)$;
        $DEL\_LIST :=$ Augment-flow$(G'(x), \delta, b, S, \theta)$;
        Delete-node$(DEL\_LIST, G'(x), out, in, S)$;
    End while;
    $G'(x) :=$ Construct-layered-network$(G(x), out, in, d, S, \Delta)$;
End while;
```

Figure 3.5. Proportional arc augmenting algorithm

3.1.2. Comparison of the Augmenting Flow Step between PAA and DA

Here we show PAA may not necessarily augment more flow than DA over the same layered network. Let θ_{DA} and θ_{PAA} respectively be the total flow to be augment through a layered network in one iteration by DA and PAA. DA finds a path P and augments flow $\theta_{DA} = \min\{r_{ij}, \forall(i.j) \in P\}$ along P. PAA will calculate unit augmenting-flow for each arc in the layered network and calculate the step length $\theta_{PAA} = \min\left\{\frac{S(i)}{b(i)}, \forall i \in N'\right\} = \min\left\{\frac{r_{ij}}{\delta_{ij}}, \forall(i,j) \in A'\right\}$ to augment flow. Note that although $\delta_{ij} \leq 1$ and $\frac{r_{ij}}{\delta_{ij}} \geq r_{ij}$ by PAA, we can not guarantee that $\theta_{PAA} \geq \theta_{DA}$ since PAA may not necessarily saturate the same arc as DA does. On the other hand, if each arc has the same residual capacity, then $\theta_{DA} \leq \theta_{PAA}$ since $\frac{\theta_{DA}}{\theta_{PAA}} = \frac{\min\{r_{ij}, \forall(i.j) \in P\}}{\min\left\{\frac{r_{ij}}{\delta_{ij}}, \forall(i,j) \in A'\right\}} = \min\{\delta_{ij}, \forall(i,j) \in A'\} \leq 1$. Moreover, If all the arcs in the augmenting path identified by DA are not bridges, then PAA guarantees to augment more flows than DA.

Next we will compare the theoretical running time of PAA with which of DA for one iteration of augmentation. In a given layered network, PAA will calculate δ and θ by the *calculate-augmenting-flow* procedure in $O(m+n)$ time. When performing *augment-flow* step, flows are augmented to admissible arcs by $\delta_{ij} \times \theta$ in $O(m)$ time. Furthermore, updating $S(i)$ for each admissible node i takes $O(n)$ time. Finally, we remove the saturated nodes, their associated arcs, and other nodes no longer connecting to t via nonsaturated admissible arcs by *delete-node* procedure in $O(m)$ time. Overall, PAA will use $O(2n+3m)$ time to augment flow in one iteration. On the other hand, DA only spends $O(2n)$ time to find a path and augment flow along this path, which takes less time than PAA to augment flow in each iteration.

3.2. Complexity Analysis for PAA

We show that the running time of PAA algorithm is $O\left(n^2m\right)$ which is the same as DA [**18**] and distance-directed algorithm [**3**]. We prove that the *construct-layered-network* procedure takes $O(m)$ time and calculating the maximum flow over a given layered network takes $O(nm)$ time.

Lemma 2. *The construct-layered-network procedure runs in $O(m)$ time.*

Proof. Recall that we first perform a BFS to obtain the distance labels of nodes and record the order the labeled nodes as a queue. This step would require $O\left(m+n\right) = O\left(m\right)$ time because every node in N and arc in A will be explored at most once. Then we check each node j in a queue whether it can reach t or not. This step we can record $out(i)$ and $in(i)$ for each node $i \in N$. If $out(i)$ or $in(i)$ becomes zero, we know node i can not reach t along any path and then we trace the incoming admissible arcs (j,i) to node i to decrease $out(j)$ when $out(i)=0$, or trace the outgoing admissible arcs (i,j) from node i to decrease $in(j)$ when $in(i)=0$. The worst case of this step also takes $O\left(m+n\right) = O\left(m\right)$ time. Therefore the *construct-layered-network* procedure runs in $O\left(m\right)$ time. $\square$

Lemma 3. *Calculating the maximum flow for each layered network by PAA takes $O(nm)$ time*

Proof. The initial sum of the residual capacity $S(i)$ of admissible arcs from node $i \in N'$ can be calculated by *construct-layered-network* step. The $b(i)$ has been initially defined if node i is the source node or calculated by the former nodes of node i in N'. After calculating $b(i)$ for each node $i \in N'$, we can easily obtain unit augmenting-flow of each admissible arc from node i by scanning these admissible arcs and the step length of node i by using $\frac{S(i)}{b(i)}$. Note that we record the maximum step length until the other step length is smaller than this one. After exploring all nodes in N', we can determine the maximum step length θ and augmenting-flow vector δ of admissible arcs. If we record δ by an array, we can obtain improving flow x in $O(m)$ time. Then we update $S(i)$ for each node $i \in N'$ by decreasing $\theta \times b(i)$ and record the saturated nodes. This step takes $O(n)$ time.

Concluding the above steps, we will search at most m times on admissible arcs to obtain δ, improve flow x in $O(m)$ time, and update S in $O(n)$ time by one *calculate-augmenting-flow* and *augment-flow* inner-iteration. If node i is added to DEL_LIST, all of the admissible arcs from node i will be removed. Then this node will be removed from current layered network. We also scan the incoming admissible arcs from node i to node j to decrease $in(j)$ or outgoing admissible arcs from node j to i to decrease $out(j)$. If there exist some nodes associated with zero outgoing or incoming admissible arcs, we also remove these nodes at the same time. This step takes $O(m)$ time and we will remove at most n nodes in a given layered network. Therefore we perform $O(n(2n + 3m)) = O(nm)$ time in doing *calculate-proportional-flow* and *augment-flow* until the flows passing through current layered network is optimal. □

Finally we prove the convergence of PAA algorithm

Lemma 4. *Until flow x is a max-flow, we will perform the construct-layered-network at most n times*

Proof. Assume the current distance from s to t is d and there exists no admissible path from s to t with current distance labels. That means the algorithm has already ships max-flow along paths with lengths less than or equal to d, and any further flow, if there is any, has to be shipped via paths with length larger than d. Since the distance from s to t is at most $n-1$, we conclude that the *construct-layered-network* procedure can be performed at most n times. $\square$

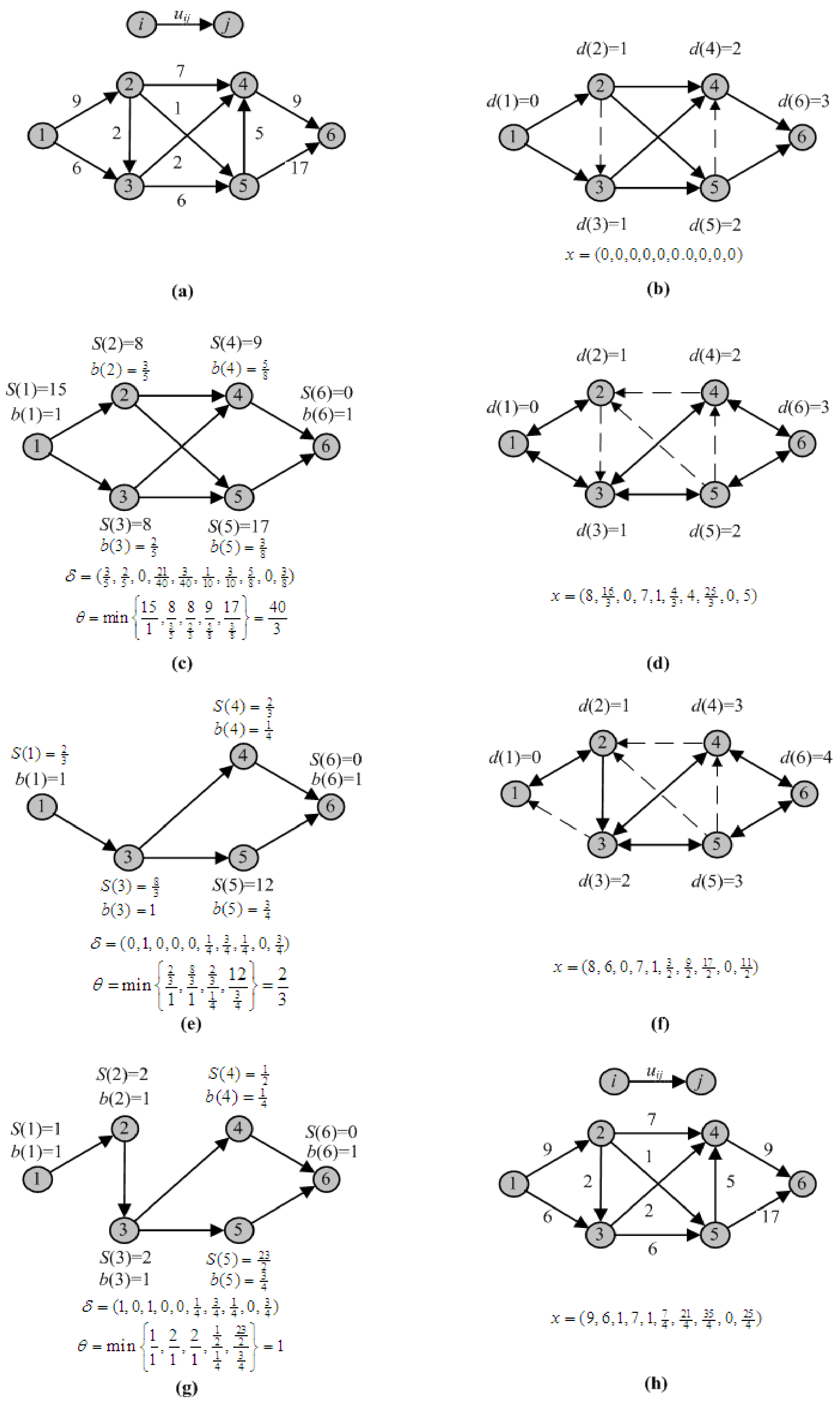
Based on the above Lemma 2, 3, and 4, we can conclude that the PAA algorithm runs in $O(n(m+nm)) = O(n^2m)$. We have thus established the following theorem

Theorem 5. *The proportional arc augmenting algorithm runs in $O(n^2m)$ time*

3.3. An Illustrative Example for PAA

Here we give a small maximum flow problem to illustrate PAA algorithm. In Figure 3.6(a), we would like to ship as much flow as possible from node 1 to 6. $N = \{1,2,3,4,5,6\}$, $A = \{(1,2),(1,3),(2,3),(2,4),(2,5),(3,4),(3,5),(4,6),(5,4),(5,6)\}$, $s=1$, $t=6$ and $u = (9,6,2,7,1,2,6,9,5,17)$.

First performing *construct-layered-network* step to calculate the distance label of each node and we obtain admissible nodes $N' = \{1,2,3,4,5,6\}$ stored by BFS order. The admissible arcs $A' = \{(1,2),(1,3),(2,4),(2,5),(3,4),(3,5),(4,6),(5,6)\}$ and inadmissible arcs are $\{(2,3),(5,4)\}$, based on the current distance labels. A layered network can be constructed as Figure 3.6(b). Starting with the first node in N' and letting $b(1) = 1$, we calculate the unit augmenting-flow $\delta_{12} = \frac{3}{5}$ and $\delta_{13} = \frac{2}{5}$ and add these two values to $b(2)$ and $b(3)$. Repeat the above steps, we obtain $b = \left(1, \frac{3}{5}, \frac{2}{5}, \frac{5}{8}, \frac{3}{8}, 1\right)$ and $\delta = \left(\frac{3}{5}, \frac{2}{5}, 0, \frac{21}{40}, \frac{3}{40}, \frac{1}{10}, \frac{3}{10}, \frac{5}{8}, 0, \frac{3}{8}\right)$ as illustrate in Figure 3.6(c). Finally we calculate maximum step length $\theta = \min\left\{\frac{15}{1}, \frac{8}{\frac{3}{5}}, \frac{8}{\frac{2}{5}}, \frac{9}{\frac{5}{8}}, \frac{17}{\frac{3}{8}}\right\} = \frac{40}{3}$ and improve $x = \delta \times \theta =$



u_{ij}
9, 7, 1, 2, 5, 9, 6, 2, 17, 6
(a)
$d(1)=0$, $d(2)=1$, $d(4)=2$, $d(6)=3$, $d(3)=1$, $d(5)=2$
$x=(0,0,0,0,0,0,0,0,0,0)$
(b)
$S(1)=15$, $b(1)=1$; $S(2)=8$, $b(2)=\frac{3}{5}$; $S(4)=9$, $b(4)=\frac{5}{8}$; $S(6)=0$, $b(6)=1$; $S(3)=8$, $b(3)=\frac{2}{5}$; $S(5)=17$, $b(5)=\frac{3}{8}$
$\delta=(\frac{3}{5},\frac{2}{5},0,\frac{21}{40},\frac{3}{40},\frac{1}{10},\frac{3}{10},\frac{5}{8},0,\frac{3}{8})$
$\theta=\min\left\{\frac{15}{1},\frac{8}{\frac{3}{5}},\frac{8}{\frac{2}{5}},\frac{9}{\frac{5}{8}},\frac{17}{\frac{3}{8}}\right\}=\frac{40}{3}$
(c)
$d(1)=0$, $d(2)=1$, $d(4)=2$, $d(6)=3$, $d(3)=1$, $d(5)=2$
$x=(8,\frac{16}{3},0,7,1,\frac{4}{3},4,\frac{25}{3},0,5)$
(d)
$S(1)=\frac{2}{3}$, $b(1)=1$; $S(4)=\frac{2}{3}$, $b(4)=\frac{1}{4}$; $S(6)=0$, $b(6)=1$; $S(3)=\frac{8}{3}$, $b(3)=1$; $S(5)=12$, $b(5)=\frac{3}{4}$
$\delta=(0,1,0,0,0,\frac{1}{4},\frac{3}{4},\frac{1}{4},0,\frac{3}{4})$
$\theta=\min\left\{\frac{\frac{2}{3}}{1},\frac{\frac{8}{3}}{1},\frac{\frac{2}{3}}{\frac{1}{4}},\frac{12}{\frac{3}{4}}\right\}=\frac{2}{3}$
(e)
$d(1)=0$, $d(2)=1$, $d(4)=3$, $d(6)=4$, $d(3)=2$, $d(5)=3$
$x=(8,6,0,7,1,\frac{3}{2},\frac{9}{2},\frac{17}{2},0,\frac{11}{2})$
(f)
$S(1)=1$, $b(1)=1$; $S(2)=2$, $b(2)=1$; $S(4)=\frac{1}{2}$, $b(4)=\frac{1}{4}$; $S(6)=0$, $b(6)=1$; $S(3)=2$, $b(3)=1$; $S(5)=\frac{23}{2}$, $b(5)=\frac{3}{4}$
$\delta=(1,0,1,0,0,\frac{1}{4},\frac{3}{4},\frac{1}{4},0,\frac{3}{4})$
$\theta=\min\left\{\frac{1}{1},\frac{2}{1},\frac{2}{1},\frac{\frac{1}{2}}{\frac{1}{4}},\frac{\frac{23}{2}}{\frac{3}{4}}\right\}=1$
(g)
u_{ij}
9, 7, 1, 2, 5, 9, 6, 2, 17, 6
$x=(9,6,1,7,1,\frac{7}{4},\frac{21}{4},\frac{35}{4},0,\frac{25}{4})$
(h)

Figure 3.6. An example of PAA for max-flow problem

$\left(8, \frac{16}{3}, 0, 7, 1, \frac{4}{3}, 4, \frac{25}{3}, 0, 5\right)$ in Figure 3.6(d). Because all of the outgoing admissible arcs of node 2 are saturated, we remove node 2 from N' in the beginning of the second inner-iteration in current layered network.

Continuing to solve for $b = \left(1, 0, 1, \frac{1}{4}, \frac{3}{4}, 1\right)$ and $\delta = \left(0, 1, 0, 0, 0, \frac{1}{4}, \frac{3}{4}, \frac{1}{4}, 0, \frac{3}{4}\right)$ and $\theta = \frac{2}{3}$, then we calculate a better feasible flow $x = \left(8, 6, 0, 7, 1, \frac{3}{2}, \frac{9}{2}, \frac{17}{2}, 0, \frac{11}{2}\right)$ as illustrate in Figure 3.6(e) and (f), and then node 1 is removed which disconnected the current layered network.

Next, we recalculate the distance labels and construct a new layered network with $N' = \{1, 2, 3, 4, 5, 6\}$ and admissible arcs $A' = \{(1, 2), (2, 3), (3, 4), (3, 5), (4, 6), (5, 6)\}$ as illustrate in Figure 3.6(g). We then calculate $\delta = \left(1, 0, 1, 0, 0, \frac{1}{4}, \frac{3}{4}, \frac{1}{4}, 0, \frac{3}{4}\right)$ and a better feasible flow $x = \left(9, 6, 1, 7, 1, \frac{7}{4}, \frac{21}{4}, \frac{35}{4}, 0, \frac{25}{4}\right)$ by the maximum step length $\theta = 1$. After removing node 1, the layered network becomes disconnected from s to t. We then recalculate the distance labels and observe that $d(t) = \infty$. Thus we terminate PAA and the optimality is attained.

Note that PAA may augment fractional flows, rather than the integral flows as conventional maximum flow algorithms. If maximum integral flows are required, we can conduct another procedure similar to flow decomposition algorithm to convert all the fractional arc flows into integral arc flows within $O(m^2)$ time.

3.4. Converting Fractional Optimal Flow to Integral Optimal Flow

Here we illustrate how to convert the fractional flow to integral flow. Let Z^* be the calculated maximum flow of $G(x^*)$ by PAA algorithm. Note that Z^* has to be integral, since we assume all arc capacities are positive integers. Here we give a method based on the flow decomposition theorem (see Chapter 3 in Ahuja et al. [**1**]), which says any feasible arc flow x can be decomposed to path flows and cycle flows, to identify integral s-t path flows step by step. First we let $Z_r = Z^*$, conduct a modified DFS algorithm that only traverses along arcs of positive flows to identify an s-t path P that contains positive flows, and then ship an integral path flow $f(P) = \min\{\min\{\lceil x_{ij} \rceil, (i,j) \in P\}, Z_r\}$ so that at least one arc whose flow becomes negative or equal to zero, where x represents the updated arc flow over arc (i,j). Then, we update Z_r by $Z_r - f(P)$, update x_{ij} by $x_{ij} - f(P)$ for each (i,j) lying on path P, find another s-t path P that contains positive flows, and repeat these procedures until Z_r becomes zero. Then all the path flows we identify by these procedures will form an integral optimal flow.

Lemma 6. *The proposed flow decomposition techniques guarantees to convert a fractional flow x^* to be an integral flow y^* within $O(m^2)$ time, without affecting flow shipped from s to t.*

Proof. Since Z^* is a positive integral flow from s to t, which will not be affected by cycle flows, there must exist Z^* units of integral s-t path flows, by flow decomposition theorem. In other words, we can always identify an s-t path that ships positive flow, which can be identified by traversing along some arcs of positive arcs. Furthermore, since we only traverse along those arcs of positive flows, we know $\lceil x_{ij} \rceil \geq 1$, which means every iteration we will always find an s-t path that ships at least one unit of flow and remove at least one arc that contains positive flow (i.e. the arc defining $f(P)$). Therefore, Z_r will decrease at least one at each iteration and becomes zero within $O(m)$ iterations, since each iteration this method removes at least one arc and there are at most m arcs. The entire procedure takes $O(m^2)$ time since the modified DFS takes $O(m)$ time at each iteration and there are at most iterations. Note that when $Z_r = 0$, there may still exist some arcs with nonzero flows, which must be cycle flows and can be ignored or reset to be zero, without affecting the integral optimal path flow y^*. □

By a modified DFS that only traverses along arcs of positive flows, we can obtain directed paths.

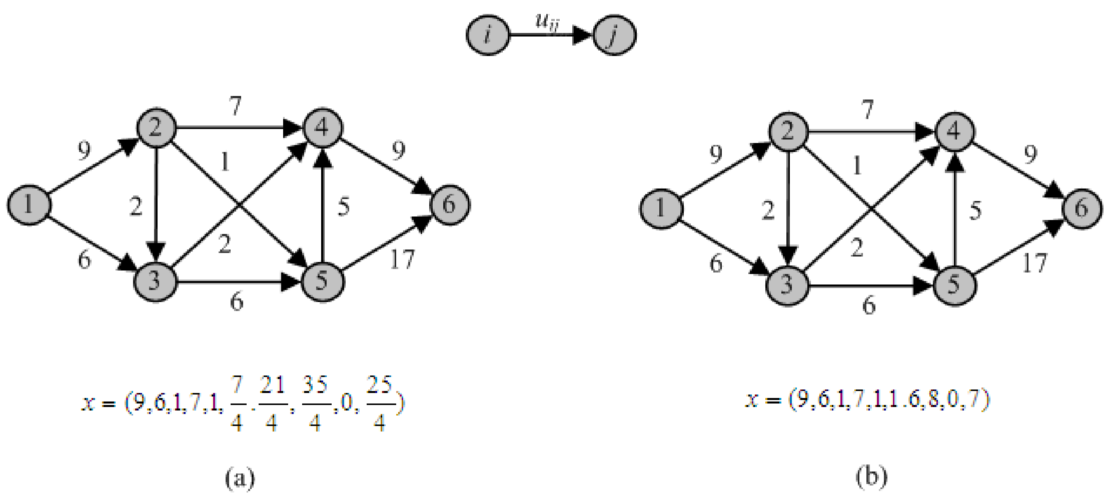

u_{ij}
$x = (9, 6, 1, 7, 1, \frac{7}{4}, \frac{21}{4}, \frac{35}{4}, 0, \frac{25}{4})$
(a)
$x = (9, 6, 1, 7, 1, 1, 6, 8, 0, 7)$
(b)

Figure 3.7. An example of converting fractional flow to integral flow

We use a small example to illustrate how our method works. In Figure 3.7(a), the maximum flow of this network is $Z = Z_r = 15$ and we observe that there exists some fractional flow x. By a modified DFS that only traverses along arcs of positive flows, we can obtain directed paths $P_1 = 1-2-4-6$ with $f(P_1) = 7$, $P2 = 1-2-5-6$ with $f(P_2) = 1$, $P_3 = 1-2-3-4-6$ with $f(P_3) = 1$, and $P_4 = 1-3-4-6$ with $f(P_4) = 6$ and decrease $Z_r = 0$. Then, we obtain the integral optimal flow as illustrated in Figure 3.7(b).

3.5. Speed-up Techniques for PAA

Here we propose some speed-up techniques to improve the practical efficiency of the PAA implementation. First, in order to avoid redundant operations such as scanning those exhausted nodes and their associated arcs, we use a simple data structure (i.e. doubly linked list) to record the previous and next nodes for each nonexhausted admissible node. Second, we point out different augmenting mechanisms, depending on whether pushing or pulling oriented, result in different improving effects which may help to prevent the algorithm to stall (i.e. consecutive small augmentations). Finally, we propose a scaling phase of PAA algorithm to prevent small augmentations.

3.5.1. Skipping Exhausted Nodes

Note that most operations of PAA are conducted over layered networks, whose size are usually smaller than the original one and will decrease after removing some nodes and arcs. Thus, one may store the BFS traversal order as a doubly linked list. In particular, for each admissible node, we store its index, previous node and next node in BFS traversal order. Therefore the operations to calculate δ, θ, and x can be conducted by scanning node from one to another by this traversal order very efficiently. Similar technique can also be applied for storing admissible arcs, and are especially beneficial since the removal of nodes or arcs can also be efficiently conducted by doubly linked list.

For example, Figure 3.8 shows a layered network with $N' = \{1, 2, 3, 4, 5, 6\}$ and $A' = \{(1,2), (1,3), (2,4), (2,5), (3,4), (3,5), (4,6), (5,6)\}$.

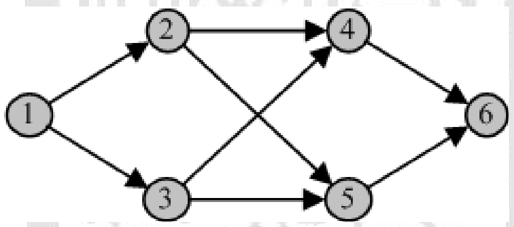


Figure 3.8. An example of storing additional informations to skip exhausted nodes

The three arrays are:

Node	**1**	**2**	**3**	**4**	**5**	**6**
pervious node array	0	1	2	3	4	5
next node array	2	3	4	5	6	0
start arc index array	0	2	4	5	7	8

Based on these three arrays, we can easily skip the saturated nodes and related arcs by using the order of the next array. For example, if node 2 and 5 are saturated, the previous and next array are changed into:

Node	**1**	**2**	**3**	**4**	**5**	**6**
pervious node array	0	0	1	3	0	4
next node array	3	0	4	6	0	0

The scanning order of the admissible nodes are following the next array, that is 1, 3, 4, and 6. Besides, we will only scan the admissible arcs $(1,2),(1,3),(3,4),(3,5),(4,6)$ based on the start arc index array. Though this may take more memory to store additional informations, the practical efficiency is improved about twice as fast as the original PAA. The detail comparison is described in subsection 5.3.1.

3.5.2. Pushing and Pulling Flow Augmentations

Since PAA calculates δ by pushing one unit of flow from s and distributing the unit flow along all outgoing arcs from each admissible node in a given layered network, it may augment small flow in some iteration because $b(i) := \sum\limits_{(i,j)\in A'} \delta_{ij}$ may be large and $S(i) := \sum\limits_{(i,j)\in A'} \delta_{ij}$ may be small for a bottleneck node i so that $\theta = \frac{S(i)}{b(i)}$ is small. If consecutive small flow augmentations take place, we say the algorithm is *stalling.*

On the other hand, instead of pushing one unit of flow to s, we can also calculate another δ by pulling one unit of flow from t and then collecting partial flow from its incoming arcs. In particular, the δ^f calculated by pushing augmentation considers the residual capacity for arcs emanating from a node, while the δ^b calculated by pulling augmentation considers the residual capacity for arcs entering a node. Thus δ^f may differ from δ^b.

Both δ^f and δ^b can satisfy the flow balance constraints and obey the intuition to augment flow proportionally to the residual arc capacity, except they improve arc flow in different directions. To have more understanding on the stalling affect, we record the history of flow augmentation θ for some network example, as shown in Figure 3.9. It appears that θ may oscillate without any pattern. However, it is expected, although without rigorous proof, that periodically altering the augmentation by δ^f

and δ^b (i.e. one iteration by pushing and one iteration by pulling) may help calculating consecutive large θ and avoid the stalling affect. To efficiently alter the pushing and pulling augmentations, one should have efficient way to record/update the sum of δ_{ij} and r_{ij}, with respected to either node i or node j. Whether it pays to store those information by some extra storage to speed the operation of

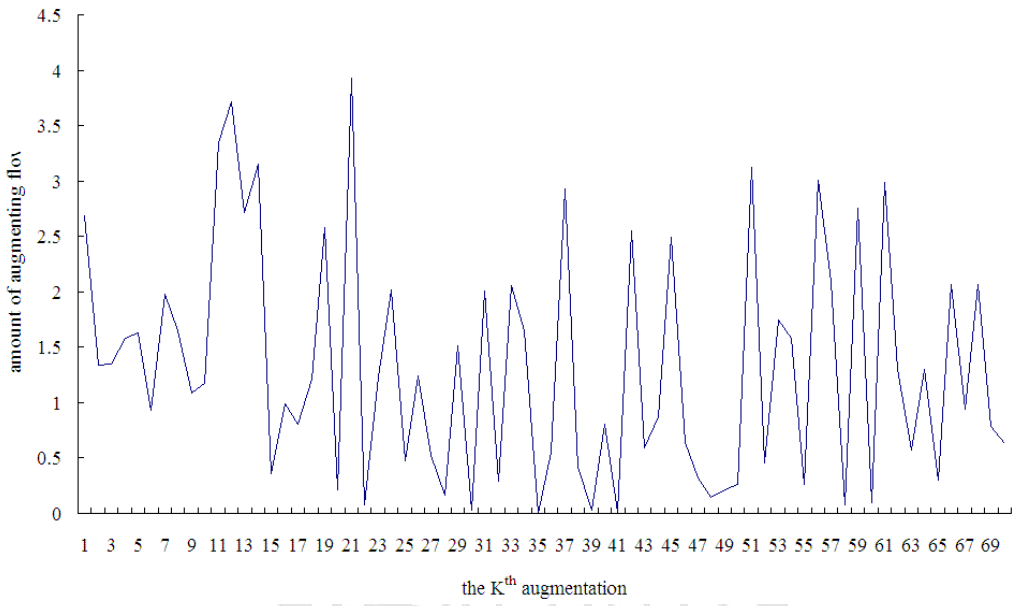


Figure 3.9. The amount of flow augmented in each iteration (i.e. θ) for a given layered network

Let $b^f(i) := \sum_{(i,j)\in A'} \delta_{ij}$ be the sum of incoming unit augmenting-flow for each admissible node i that can be distributed to the outgoing admissible arcs from each node and $b^b(i) := \sum_{(j,i)\in A'} \delta_{ji}$ be the sum of outgoing unit augmenting-flow for each admissible node that can be distributed to the incoming admissible arcs from each node, $S^f(i) := \sum_{(i,j)\in A'} \delta_{ij}$ be the sum of residual capacity for outgoing admissible arcs from each admissible node and $S^b(i) := \sum_{(j,i)\in A'} \delta_{ji}$ be the sum of residual capacity for incoming admissible arcs from each admissible node, $\delta^f = \left[\delta^f_{ij}\right]$ be the augmenting-flow vector calculating from source node and $\delta^b = \left[\delta^b_{ij}\right]$ be the augmenting-flow vector calculating from sink node for each admissible arc (i,j), and $\theta^f = \min\left\{\frac{S^f(i)}{b^f(i)}, i \in N'\right\}$ and $\theta^b = \min\left\{\frac{S^b(i)}{b^b(i)}, i \in N'\right\}$. In one iteration, we select the larger maximum step length between θ^f and θ^b. If θ^f is selected, the improving flow of admissible arcs are improved by $\delta^f \times \theta^f$. Otherwise, the improving flow of admissible arcs are improved by $\delta^b \times \theta^b$. Then we recalculate $b^f, b^b, S^f, S^b, \delta^f$, and δ^b to find the larger maximum step length to augment flow until the optimal of current layered network is attained.

Here we give a small example to show that θ^b may not necessarily be smaller than θ^f. In Figure 3.10(a), we would like to ship as much flow as possible from node 1 to 8. First we perform a pushing flow augmentation by PAA. The augmenting flow $\delta^f = \left(\frac{3}{8}, \frac{1}{8}, \frac{4}{8}, \frac{2}{8}, \frac{1}{8}, \frac{1}{8}, \frac{4}{8}, \frac{2}{8}, \frac{2}{8}, \frac{4}{8}\right)$ and a better feasible flow $x = (3, 1, 4, 2, 1, 1, 4, 2, 2, 4)$ by the maximum step length $\theta^f = 8$ as illustrated in Figure 3.10(b). Since arc $(4, 7)$ is saturated, we remove it from current network and then node 4 and 7 and associated arcs are also removed. In the next iteration, if we still perform a pushing flow augmentation, the augmenting flow $\delta^f = \left(\frac{3}{4}, \frac{1}{4}, 0, \frac{2}{4}, \frac{1}{4}, \frac{1}{4}, 0, \frac{2}{4}, \frac{2}{4}, 0\right)$ and $\theta^f = 4$ are calculated to improve flow $x = (6, 2, 4, 4, 2, 2, 4, 4, 4, 4)$ in Figure 3.10(c). However, if we perform a pulling flow augmentation, $\delta^b = \left(\frac{47}{438}, \frac{391}{438}, 0, \frac{4}{73}, \frac{23}{438}, \frac{391}{438}, 0, \frac{4}{73}, \frac{69}{73}, 0\right)$ and $\theta^b = \frac{438}{23} > \theta^f = 4$ are calculated to improve flow $x = \left(\frac{116}{23}, 18, 4, \frac{70}{23}, 2, 18, 4, \frac{70}{23}, 20, 4\right)$ in Figure 3.10(d). In this case, the amount of augmented flow passing through 1 to 8 by pulling flow augmentation is larger than the one by pushing flow augmentation.

Although considering pushing and pulling augmentation at the same time in each iteration may reduce the number of flow augmentation in a given layered network, calculating δ^f and δ^b and recalculate S^f and S^b need to take $O(4m)$ time by scanning all admissible arcs. To save more time, we only consider pushing augmentations in odd numbered layered networks and pulling augmentations in even numbered layered networks. Then we can update S^f or S^b in $O(n)$ time which is the same as PAA that only use pushing augmentation. We denote this modified algorithm as PAAFB, where F and B respectively stand for pushing (forward) and pulling (backward) augmentation. We also compare the practical efficiency of PAAFB with PAA. The computational result is showed in subsection 5.3.1.

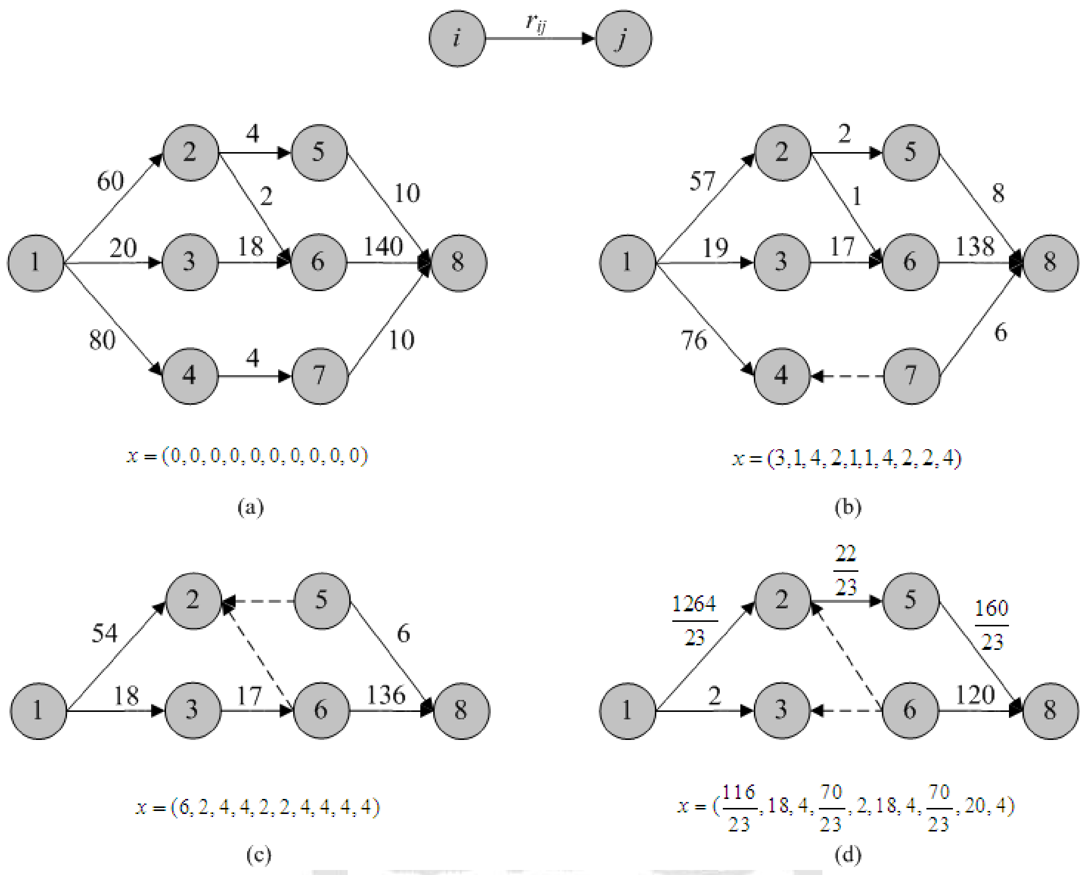

r_{ij}
$x = (0,0,0,0,0,0,0,0,0,0)$
(a)
$x = (3,1,4,2,1,1,4,2,2,4)$
(b)
$x = (6,2,4,4,2,2,4,4,4,4)$
(c)
$x = (\frac{116}{23},18,4,\frac{70}{23},2,18,4,\frac{70}{23},20,4)$
(d)

Figure 3.10. An example shows that pulling flow augmentation may augment more flow than pushing flow augmentation.

3.5.3. A Scaling Phase Technique

In this section we integrate the scaling phase technique with PAA to have an efficient implementation called PSA (i.e. PAA with scaling phase) to guarantee effective augmentation. Starting from a given threshold $\Delta = 2^{\lceil \log U \rceil}$, we perform *construct-layered-network* to construct a layered network that only contains those admissible arcs with residual capacity larger than or equal to Δ. If no such a layered network that connects s to t can be constructed, PSA replaces current Δ by $\Delta/2$ and continues to perform *construct-layered-network* procedure. After constructing a layered network, PSA augments flow by operations of PAA until the maximum flow is shipped in current layered network, and then PSA continues to decrease current Δ to be $\Delta/2$ and return to *construct-layered-network* step. These processes are repeated until $\Delta = 1$, and then setting $\Delta = 0$ to ship the remaining flow by PAA. The PSA algorithm is described as Figure 3.11.

```
**Algorithm 8** $Proportional\_arc\_augmenting\_scaling(G(x))$
**Initialization:**
    **For** each node $i \in N$ **do**
        $out(i) := 0; in(i) := 0; d(i) := \infty;$
    $\Delta := 2^{\lceil \log U \rceil}$ ;
**While** $\Delta \geq 1$ **do**
    **While** $d(t) = \infty$ **do**
        $G'(x) :=$ Construct-layered-network$(G(x), out, in, d, S, \Delta)$;
        **If** $d(t) = \infty$ **then** $\Delta := \Delta/2$;
    **End while**;
    **While** $s \in N'$ and $t \in N'$ **do**
        $\delta :=$ Calculate-proportional-flow$(G'(x), b)$;
        $DEL\_LIST :=$ Augment-flow$(G'(x), \delta, b, S)$;
        Delete-node$(DEL\_LIST, G'(x), out, in, S)$;
    **End while**;
    $\Delta := \Delta/2$;
**End while;**
$\Delta := 0$;
$proportional\_arc\_augmenting(G(x))$;
```

Figure 3.11. Proportional arc augmenting scaling algorithm

We examine the complexity of the PSA algorithm. Since we will construct at most $\log U$ layered networks before $\Delta = 1$ and at most n layered networks when $\Delta = 0$. Each layered network needs $O(nm)$ time to augment flow. Therefore the total running time of the scaling version of PAA algorithm is $O(nm \log U + n^2 m)$.

Theorem 7. *The PSA algorithm solves the maximum flow problem within $O(nm \log U)$ augmentations when $\Delta \geq 1$ and $O(n^2 m)$ when $\Delta = 0$. The total running time for PSA is $O(nm \log U + n^2 m)$.*

3.6. Ideas of Flow Splitting

This section we propose another idea to calculate different augmenting-flow vector δ. Although PAA can efficiently calculate δ, it is not clear whether it pays off to calculate the overhead $S(i) := \sum_{(i,j) \in A'} r_{ij}$. To further save such overhead while satisfying the flow balance constraint, we propose another idea for calculating a qualified δ more efficiently. In particular, for each node i that receives a total amount of augmenting-flow vector $b(i) := \sum_{(i,j)\in A'} \delta_{ij}$, our second maximum flow algorithm equally distributes $b(i)$ to each of its outgoing admissible arcs. Calculating δ not only takes less time, but also guarantees the flow balance constraints are satisfied. However, such a δ will only saturates at least one admissible arc, instead of one node at each iteration as does by PAA.

3.6.1. Flow Splitting Augmenting Algorithm (FSA)

FSA can exploit most procedures of PAA, except calculating different δ_{ij} and θ. When performing the *construct-layered-network* procedure, we obtain a layered network to improve flow and the number of outgoing and incoming arcs of each admissible node. Same as PAA, first we set $b(s) := 1$ and distribute $\delta_{ij} = \frac{b(i)}{out(i)}$ for each arc $(i,j) \in A'$ from each node $i \in N'$ based on the BFS traversal order. The maximum step length θ is determined by a bottleneck arc $(i^*, j^*) = \arg \min_{(i,j)\in A'} \left\{ \frac{r_{ij}}{\delta_{ij}} \right\}$, and the improving flow of each admissible arc can be obtained by $\delta_{ij} \times \theta$. The *split-flow* and *augment-split-flow* procedure are illustrated in Figure 3.12 and 3.13. We conclude the FSA algorithm in Figure 3.14.

```
**Procedure 9** *Split-flow*$(G'(x), b, out, \delta)$
$b(s) := 1; order := 1; i := 0; \theta := \infty$
**While** $i \neq t$ and $i \in N'$**do**
    $i := N'(order)$;
    **For** each arc $(i,j) \in A'$ **do**
        $\delta_{ij} := \frac{b(i)}{out(i)}; b(j) := b(j) + \delta_{ij}$;
        **If** $\frac{r_{ij}}{\delta_{ij}} < \theta$ **then** $\theta := \frac{r_{ij}}{\delta_{ij}}$;
    $order := order + 1$;
**End while**;
**Return** $\theta$;
```

Figure 3.12. Pseudo-code for splitting proportional flow

```
**Procedure 10** *Augment-split-flow*$(G'(x), \delta, out, in, \theta)$
$DEL\_LIST := \varnothing$;
**For** each arc $(i,j) \in A'$ **do**
    **If** arc $(i,j) \in A$ **then** $x_{ij} := x_{ij} + \delta_{ij} \times \theta$;
        **If** $x_{ij} = u_{ij}$ **then** $out(i) := out(i) - 1; in(j) := in(j) - 1$;
            **If** $out(i) = 0$ **then** $DEL\_LIST := DEL\_LIST \cup i$;
            **Else if** $in(j) = 0$ **then** $DEL\_LIST := DEL\_LIST \cup j$;
    **Else then** $x_{ij} := x_{ij} - \delta_{ij} \times \theta$;
        **If** $x_{ij} = 0$ **then** $out(j) := out(j) - 1; in(i) := in(i) - 1$;
            **If** $out(j) = 0$ **then** $DEL\_LIST := DEL\_LIST \cup j$;
            **Else if** $in(i) = 0$ **then** $DEL\_LIST := DEL\_LIST \cup i$;
**Return** $DEL\_LIST$;
```

Figure 3.13. Pseudo-code for augmenting split flow

```
**Algorithm 11** *Flow_ splitting_ augmenting*$(G(x))$
**Initialization:**
    **For** each node $i \in N$ **do**
        $out(i) := 0; in(i) := 0; d(i) := \infty$;
$G'(x) :=$ *Construct-layered-network*$(G(x), out, in, d, \varnothing, \Delta)$;
**While** $d(t) \neq \infty$ **do**
    **While** $s \in N'$ and $t \in N'$ **do**
        **For** each arc $(i,j) \in A'$ **do** $\delta_{ij} := 0$;
        $\theta :=$ *Split-flow*$(G'(x), b, out, \delta)$;
        $DEL\_LIST :=$ *Augment-splitted-flow*$(G'(x), \delta, out, in, \theta)$;
        *Delete-node*$(DEL\_LIST, G'(x), out, in, \varnothing)$;
    **End while**;
    $G'(x) :=$ *Construct-layered-network*$(G(x), out, in, d, \varnothing, \Delta)$;
**End while;**
```

Figure 3.14. Flow splitting augmenting algorithm

3.6.2. Complexity Analysis for FSA

We show the running time of FSA algorithm is $O(nm^2)$. Same as PAA, FSA will construct at most n layered networks by using *construct-layered-network* procedure. The total iterations of executing *split-flow*, *augment-splitted-flow*, and *delete-node* in a given layered network are $O(m^2)$.

Lemma 8. *The total iterations of executing split-flow, augment-splitted-flow and delete-node steps in a given layered network are* $O(m^2)$.

Proof. Note that $b(i)$ has been initially defined if node i is the source node, or has been calculated by those nodes traversed before node i in N'. After calculating $b(i)$ for each node $i \in N'$, we can easily obtain the unit augmenting-flow for each admissible arc emanating from node i by fairly dividing $b(i)$ based on the number of its outgoing admissible arcs. We also record the maximum step length until the other step length is smaller than this one. After exploring all nodes in N', we can determine the maximum step length θ and augmenting-flow vector δ. If we record δ by an array, we can improve flow x and recalculate the number of outgoing and incoming admissible arcs for each node in $O(m)$ time. Then we can remove at least one saturated arc in A'.

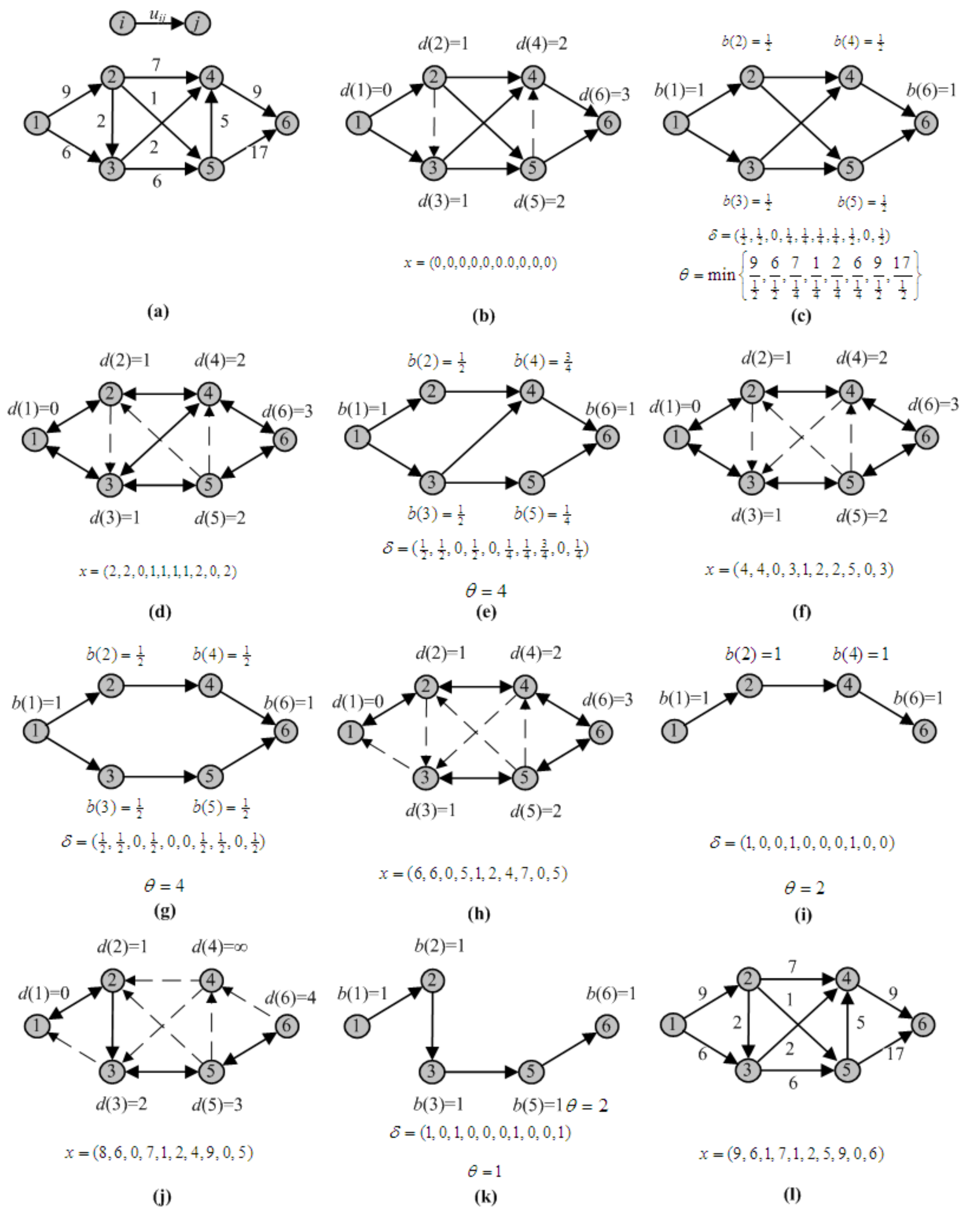
Concluding the above steps, we will search at most m times on admissible arcs to calculate δ and improve flow x in $O(m)$ time by one *split-flow* and *augment-splitted-flow* inner-iteration. If node i is added to DEL_LIST, all of the admissible arcs from node i will be removed. Then this node will be removed from current layered network. We also scan the incoming admissible arcs from node i to node j to decrease $in(j)$ or outgoing admissible arcs from node j to i to decrease $out(j)$. If there exist some nodes which has no outgoing or incoming admissible arcs, we also remove these nodes at the same time. This step needs $O(m)$ time and we will remove at most m arcs in a given layered network. Therefore we spend $O(m(3m)) = O(m^2)$ time in doing *split-flow* and *augment-splitted-flow* until the flows passing through current layered network is optimal. □

Based on 8, we can conclude that the FSA algorithm runs in $O(nm^2)$. We have thus established the following theorem

Theorem 9. *The flow splitting augmenting algorithm runs in $O(nm^2)$ time*

3.6.3. An Illustrative Example for FSA

Here we give a small maximum flow problem to illustrate PAA algorithm. In Figure 3.15(a), we would like to ship as much flow as possible from node 1 to 6. $N = \{1, 2, 3, 4, 5, 6\}$, $A = \{(1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (3,5), (4,6), (5,4), (5,6)\}$, $s = 1$, $t = 6$ and $u = (9, 6, 2, 7, 1, 2, 6, 9, 5, 17)$.



u_{ij}
(a)
d(1)=0
d(2)=1
d(3)=1
d(4)=2
d(5)=2
d(6)=3
x = (0,0,0,0,0,0,0,0,0,0)
(b)
b(1)=1
b(2) = 1/2
b(3) = 1/2
b(4) = 1/2
b(5) = 1/2
b(6)=1
δ = (1/2, 1/2, 0, 1/4, 1/4, 1/4, 1/4, 1/2, 0, 1/2)
θ = min{9/(1/2), 6/(1/2), 7/(1/4), 1/(1/4), 2/(1/4), 6/(1/4), 9/(1/2), 17/(1/2)}
(c)
x = (2, 2, 0, 1, 1, 1, 1, 2, 0, 2)
(d)
b(4) = 3/4
b(5) = 1/4
δ = (1/2, 1/2, 0, 1/2, 0, 1/4, 1/4, 3/4, 0, 1/4)
θ = 4
(e)
x = (4, 4, 0, 3, 1, 2, 2, 5, 0, 3)
(f)
δ = (1/2, 1/2, 0, 1/2, 0, 0, 1/2, 1/2, 0, 1/2)
θ = 4
(g)
x = (6, 6, 0, 5, 1, 2, 4, 7, 0, 5)
(h)
b(2) =1
b(4) =1
δ = (1, 0, 0, 1, 0, 0, 0, 1, 0, 0)
θ = 2
(i)
d(4)=∞
d(3)=2
d(5)=3
d(6)=4
x = (8, 6, 0, 7, 1, 2, 4, 9, 0, 5)
(j)
b(2)=1
b(3)=1
b(5)=1 θ = 2
δ = (1, 0, 1, 0, 0, 0, 1, 0, 0, 1)
θ = 1
(k)
x = (9, 6, 1, 7, 1, 2, 5, 9, 0, 6)
(l)

Figure 3.15. An example of FSA for max-flow problem

First we perform *construct-layered-network* step to obtain admissible nodes $N' = \{1, 2, 3, 4, 5, 6\}$ and $A' = \{(1,2), (1,3), (2,4), (2,5), (3,4), (3,5), (4,6), (5,6)\}$. A layered network can be constructed as Figure 3.15(b). Starting with the first node in N' and letting $b(1) = 1$, since there are two admissible arcs outgoing from node 1 is two, we fairly distribute unit augmenting-flow to $\delta_{12} = \frac{1}{2}$ and $\delta_{13} = \frac{1}{2}$ and add these two values to $b(2)$ and $b(3)$. Repeat these steps, we get $b = \left(1, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1\right)$ and $\delta = \left(\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, 0, \frac{1}{2}\right)$ as illustrate in Figure 3.15(c). Then we calculate the maximum step length $\theta = \min\left\{\frac{9}{\frac{1}{2}}, \frac{6}{\frac{1}{2}}, \frac{7}{\frac{1}{4}}, \frac{1}{\frac{1}{4}}, \frac{2}{\frac{1}{4}}, \frac{6}{\frac{1}{4}}, \frac{9}{\frac{1}{2}}, \frac{17}{\frac{1}{2}}\right\} = 4$ and improve $x = \delta \times \theta = (2, 2, 0, 1, 1, 1, 1, 2, 0, 2)$ in Figure 3.15(d). Since arc $(2, 5)$ is saturated, we remove this arc from A' and repeat these procedures to calculate δ and θ to improve flow. New $\delta = \left(\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, 0, \frac{1}{4}, \frac{1}{4}, \frac{3}{4}, 0, \frac{1}{4}\right)$ is calculated in Figure 3.15(e). A better feasible flow $x = (4, 4, 0, 3, 1, 2, 2, 5, 0, 3)$ by the maximum step length $\theta = 4$ as illustrated in Figure 3.15(f).

After removing the saturated arc $(3, 4)$ from A', the algorithm continues to calculate $\delta = \left(\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, 0, 0, \frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}\right)$ in Figure 3.15(g) and the maximum step length $\theta = 4$ is calculated to improve flow $x = (6, 6, 0, 5, 1, 2, 4, 7, 0, 5)$ in Figure 3.15(h). The saturated arc $(1, 3)$ is removed from A' which results in the number of incoming admissible arcs of node 3 becoming zero. Therefore node 3, and then node 5, is removed from N'. Repeat these procedures to calculate $\delta = (1, 0, 0, 1, 0, 0, 0, 1, 0, 0)$ and $\theta = 2$ to obtain a better feasible flow $x = (8, 6, 0, 7, 1, 2, 4, 9, 0, 5)$ as illustrated in Figure 3.15(i) and (j). Arc $(2, 4)$ and $(4, 6)$ are saturated and removed from A'. Node 6 is removed from N' and no more flow can be sent from s to t. We recalculate the distance labels by BFS and construct a new layered network as illustrate in Figure 3.15(k).

Continuing to solve for $b = (1, 1, 1, 0, 1, 1)$ and $\delta = (1, 0, 1, 0, 0, 0, 1, 0, 0, 1)$ and $\theta = 1$, we calculate a better feasible flow $x = (9, 6, 1, 7, 1, 2, 5, 9, 0, 6)$ as illustrate in Figure 3.15(l), and then node 1 is removed which disconnects the current layered network. We then recalculate the distance labels and observe that $d(t) = \infty$. Thus we terminate FSA and the optimality is attained.

3.7. Summary

Two new maximum flow algorithms have been introduced in this chapter. The main difference between our algorithms with most previous ones is that our algorithms can augment flow along all arcs in a layered network, based on the relative proportions of the residual capacities between each arcs from the same node at each iteration. The complexity of PAA is polynomial-time which is same as Dinic's Algorithm [**18**]. and the distance-directed algorithms in [**3**]. Though PAA may obtain fractional optimal flow, we can use the flow decomposition method to convert fractional flow to integral flow in polynomial-time. We also proposed some speed-up techniques such as changing data structure to skip exhausted nodes and using scaling phase technique to avoid small augmentations trying to improve the efficiency of PAA. Finally, we proposed another algorithm called FSA algorithm with the similar idea of calculating augmenting-flow vector δ for each arc to improve flow.

CHAPTER 4

MODIFIED LEAST-SQUARES DUAL-PRIMAL METHOD

In this chapter, we propose a modification version of the LSDP algorithm proposed by Wang et al. [**51**], called MLSDP, which modifies the LSDP method by constructing layered networks and solving each layered network by the LSDP method. In Section 4.1 we introduce the MLSDP algorithm for the maximum flow problem. We prove the MLSDP algorithm is a polynomial time algorithm in section 4.2. Section 4.3 gives an small example to show how MLSDP solves the maximum flow problem. We propose another idea for solving layered networks by using LSDP in Section 4.4. We also use a sparse linear solver to speed-up the efficiency of MLSDP in Section 4.5. Section 4.6 gives a summary of MLSDP algorithm.

4.1. Steps of MLSDP

The LSDP algorithm iteratively identifies an augmenting-flow vector δ over a subnetwork by solving for electron flow over an electrical network mapped from the original subnetwork. In particular, the electrical network can be constructed by adding an artificial arc (t, s) to the original layered network, where each admissible arc is associated with 1Ω of resistance and 1 voltage is added on (t, s). Based on the idea of augmenting flows over a layered network like PAA and FSA at each iteration, our proposed modified LSDP algorithm, denoted by MLSDP, uses LSDP to solve for an augmenting-flow vector δ which is different from the one by PAA and FSA over a layered network.

Detailed procedures of MLSDP are illustrated in Figure 4.6, which also involves three major procedures: *Construct-layered-network-LSDP*, *Calculate-electron-flow*, and

Augment-flow-LSDP, as illustrated in Figure 4.1, 4.3, and 4.5, respectively. Procedure *Construct-layered-network-LSDP* is modified from procedure *Construct-layered-network* used in chapter 3 to build a BFS tree T and mark the admissible arcs on T to be 1 and non-tree admissible arcs to be 0. The non-tree arcs will be used to find the cycle to formulate the KVL equations. If some nodes and arcs are not lying on any $s-t$ path in a layered network, we add these nodes to DEL_LIST and perform *delete-node-LSDP* procedure to remove these nodes and associated arcs from N' and A'. Here we show the *construct-layered-network-LSDP* procedure in Figure 4.1 and the *delete-node-LSDP* procedure in Figure 4.2.

```
Procedure 12 Construct-layered-network-LSDP($G(x), out, in, d, spanarc$)
$LIST := \{s\}; N' := \{s\}; DEL\_LIST := \varnothing; d(s) := 0; N' := \varnothing; A' := \varnothing;$
While $LIST \neq \varnothing$ do
    select the first node $i$ in $LIST$;
    If node $j$ is unmarked and $r_{ij} > 0$ then
        mark node $j$;
        $A' := A' \cup \{(i,j)\}; d(j) := d(i) + 1; spanarc_{ij} := 1;$
        $out(i) := out(i) + 1; in(j) := in(j) + 1; LIST := LIST \cup j;$
    Else if node $j$ is marked, $d(j) = d(i) + 1$, and $r_{ij} > 0$ then
        $A' := A' \cup \{(i,j)\}; spanarc_{ij} := 0; out(i) := out(i) + 1; in(j) := in(j) + 1;$
    If $out(i) = 0$ then $DEL\_LIST := DEL\_LIST \cup i;$
    Else then $N' := N' \cup i;$
    $LIST := LIST \setminus i;$
End while;
$A' := A' \cup \{(t,s)\}; spanarc_{ts} := 0;$
If $DEL\_LIST \neq \varnothing$ then
    Delete-node-LSDP($DEL\_LIST, G'(x), out, in, spanarc, T$);
Return $G' = (N', A')$;
```

Figure 4.1. Pseudo-code for constructing a layered network using for LSDP

First we perform *construct-layered-network-LSDP* step to construct a layered network $G' = (N', A', s, t)$ that the sets of admissible arcs A' with $|A'| = m'$ included the artificial arc $(t.s)$ and the spanning tree T. Then we transform current layered network to the electrical network by setting a unit voltage battery to arc (t, s) and a unit resister of each admissible arc.

```
**Procedure 13** *Delete-node-LSDP*$(DEL\_LIST, G'(x), out, in, spanarc, T)$
**While** $DEL\_LIST \neq \varnothing$ **do**
    select the first node $k$ in $DEL\_LIST$;
    **For** each arc $(i,k) \in A'$ **do**
        **If** arc $(i,k) \in A$ **then**
            $out\,(i) := out\,(i) - 1; A' := A' \setminus \{(i.k)\}\,; spanarc_{ik} := -1;$
        **Else then**
            $in\,(i) := in\,(i) - 1; A' := A' \setminus \{(i.k)\}\,; spanarc_{ik} := -1;$
            **If** $in\,(i) \neq 0$ **then**
                **For** each arc $(l,i) \in A'$ **do**
                    **If** $spanarc_{li} = 0$ **then** $spanarc_{li} := 1$;**break**;
        **If** $out\,(i) = 0$ or $in(i) = 0$ **then**
            $DEL\_LIST := DEL\_LIST \cup i; N' := N' \setminus i; T := T \setminus i;$
    **For** each arc $(k,j) \in A'$ **do**
        **If** arc $(k,j) \in A$ **then**
            $in\,(j) := in\,(j) - 1; A' := A' \setminus \{(k.j)\}\,; spanarc_{kj} := -1;$
            **If** $in\,(j) \neq 0$ **then**
                **For** each arc $(l,j) \in A'$ **do**
                    **If** $spanarc_{lj} = 0$ **then** $spanarc_{lj} := 1$;**break**;
        **Else then**
            $out\,(j) := out\,(j) - 1; A' := A' \setminus \{(k.j)\}\,; spanarc_{kj} := -1;$
        **If** $out\,(j) = 0$ or $in(j) = 0$ **then**
            $DEL\_LIST := DEL\_LIST \cup j; N' := N' \setminus j; T := T \setminus j;$
**End while;**
```

Figure 4.2. Pseudo-code for deleting nodes using for LSDP

```
**Procedure 14** *Calculate-electron-flow*$(G'(x), T, out, in, spanarc)$
$checkflow := 0;$
**While** $checkflow \neq 1$ **do**
    $\delta :=$ solve $m' = |A'|$ Kirchoff's law's equations by Gauss-Jordan elimination;
    $negflow := 0; deletearc := 0;$
    **For** each arc $(i,j) \in A'$ **do**
        **If** $\delta_{ij} < 0$ and $\delta_{ij} < negflow$ **then**
            $negflow := \delta_{ij}; deletearc :=$ arc $(i,j)\,; checkflow := 1;$
    **If** $deletearc \neq 0$ **then**
        *update-tree*$(G'(x), T, spanarc, out, in, deletearc)$;
**End while**;
**Return** $\delta$;
```

Figure 4.3. Pseudo-code for calculating nonnegative electron flow

Procedure *Calculate-electron-flow,* as illustrated in Figure 4.3, calculates nonnegative electron flow, which corresponds to the augmenting-flow vector δ. In order to calculate the electron flows over m' admissible arcs by Kirchhoff's circuit laws, MLSDP records a BFS spanning tree and selects $m' - n + 1$ non-tree admissible arcs to form

$m' - n + 1$ cycles for formulating the $m' - n + 1$ KVL equations, which define the voltage difference over $m' - n + 1$ cycles. These equations, together with $n - 1$ KCL equations regarding the flow balance constraints for nodes, form an $m' \times m'$ sparse system of linear equations, and is solved by Gauss-Jordan elimination.

If the calculated electron flow $\delta_{ij} < 0$ over some arc (i, j), it means the optimal δ_{ij} calculated by Kirchhoff's circuit laws moves the electron flow along the opposite of its original orientation. Such a δ_{ij} violates the feasible condition (i.e. $\delta_{ij} > 0$) and thus has to be removed from the layered network. Similar to LSDP, here MLSDP selects the arc with the most negative δ, denoted by *deletearc*, and removes it. Note that when an admissible arc is removed (i.e. we have one fewer variable), we need to remove a KVL equation. If the arc to be removed is a non-tree arc, we do nothing since its removal will not induce any KVL equation. Otherwise (i.e. it is a BFS tree arc), we need to find another non-tree arc to substitute the removed tree arc. Note that in the latter case, such a non-tree arc always exists since we have updated the tree before this procedure which removes those isolated nodes previously connected by a removed tree arc. These operations to update tree arcs are described in procedure *update-tree*, as illustrated in Figure 4.4. Removing an arc decreases the degrees of some nodes, which may further disconnect more nodes and arcs from any *s*-*t* path. Such removal of nodes and arcs can be done by procedure *delete-node-LSDP*. This procedure iteratively calculates δ, removes those arcs of negative δ, updates tree arcs, until the resultant δ are nonnegative.

Repeating to solve the electron flows and disconnect the arc with the most negative electron flow until all electron flows of admissible arcs are positive and we regard the current δ as our improving direction.

After obtaining a feasible improving direction δ, MLSDP calculates the maximum step length θ by a minimum ratio test that seeks a bottleneck arc $(i^*, j^*) = \arg \min_{(i,j) \in A'} \left\{ \frac{r_{ij}}{\delta_{ij}} \right\}$, removes the bottleneck arc, updates the flow x to be $x + \theta \times \delta$, and

```
Procedure 15 Update-tree($G'(x), T, spanarc, out, in, deletearc$)
$A' := A' \setminus deletearc; DEL\_LIST := \varnothing; T := T \setminus deletearc;$
If $deletearc(i,j) \in A$ then
    $out(i) := out(i) - 1; in(j) := in(j) - 1;$
    If $out(i) = 0$ then $DEL\_LIST \cup i; T := T \setminus i;$
    Else if $in(j) = 0$ then $DEL\_LIST \cup j; T := T \setminus j;$
Eles then
    $out(j) := out(j) - 1; in(i) := in(i) - 1;$
    If $out(j) = 0$ then $DEL\_LIST \cup j; T := T \setminus j;$
    Else if $in(i) = 0$ then $DEL\_LIST \cup i; T := T \setminus i;$
If $DEL\_LIST \neq \varnothing$ then
    delete-node-LSDP($DEL\_LIST, G'(x), out, in, spanarc, T$);
Else then
    If $deletearc(i,j) \in A$ and $spanarc_{ij} = 1$ then
        $spanarc_{ij} := -1;$
        For each arc $(l,j) \in A'$ do
            If $spanarc_{lj} = 0$ then $spanarc_{lj} := 1;$break;
    Else if $deletearc(i,j) \notin A$ and $spanarc_{ij} = 1$ then
        $spanarc_{ij} := -1;$
        For each arc $(l,i) \in A'$ do
            If $spanarc_{li} = 0$ then $spanarc_{li} := 1;$break;
```

Figure 4.4. Pseudo-code for updating tree

updates the layered network by removing those nodes and arcs not lying on any *s*-*t* path in the layered network. Details of these operations are described in procedure *Augment-flow-LSDP* as illustrated in Figure 4.5.

```
Procedure 16 Augment-flow-LSDP($G'(x), \delta, T, spanarc, out, in$)
$\theta := \infty; REMOVEARC\_LIST := \varnothing; deletearc := 0;$
For each arc $(i,j) \in A'$ do
    $\theta_{ij} := \frac{r_{ij}}{\delta_{ij}};$
    If $\theta_{ij} < \theta$ then $\theta := \theta_{ij};$
For each arc $(i,j) \in A'$ do
    If arc $(i,j) \in A$ then $x_{ij} := x_{ij} + \delta_{ij} \times \theta;$
        If $x_{ij} = u_{ij}$ then add arc $(i,j)$ to $REMOVEARC\_LIST;$
    Else then $x_{ij} := x_{ij} - \delta_{ij} \times \theta;$
        If $x_{ij} = u_{ij}$ then add arc $(i,j)$ to $REMOVEARC\_LIST;$
While $REMOVEARC\_LIST \neq \varnothing$ do
    $deletearc :=$ the first arc in $REMOVEARC\_LIST;$
    update-tree($G'(x), T, spanarc, out, in, deletearc$);
    $REMOVEARC\_LIST := REMOVEARC\_LIST \setminus deletearc;$
End while;
```

Figure 4.5. Pseudo-code for augmenting flow using for LSDP

Based on similar framework of shipping flow on arcs of layered networks, MLSDP iteratively constructs a layered network, calculates a nonnegative electron flow δ and associated maximum step length θ to augment flow, until the layered network is saturated, and then a new layered network is constructed. MLSDP terminates when no more layered network can be constructed to connect s to t. We conclude our MLSDP algorithm in Figure 4.6.

```
Algorithm 17 $MLSDP(G(x))$
Initialization:
    For each node $i \in N$ do
        $out(i) := 0; in(i) := 0; d(i) := \infty;$
    For each arc $(i,j) \in A$ do
        $spanarc_{ij} := -1;$
$G'(x) :=$ Construct-layered-network-LSDP$(G(x), out, in, d, spanarc)$;
While $d(t) \neq \infty$ do
    While $s \in N'$ and $t \in N'$ do
        $\delta :=$ Calculate-electron-flow$(G'(x), T, out, in, spanarc)$;
        Augment-flow-LSDP$(G'(x), \delta.T, spanarc, out, in)$;
    End while;
    $G'(x) :=$ Construct-layered-network-LSDP$(G(x), out, in, d, spanarc)$;
End while;
```

Figure 4.6. MLSDP algorithm

4.2. Complexity Analysis for MLSDP

We show that our MLSDP algorithm runs in $O(nm^5)$ time. We first prove the *calculate-electron-flow* solving the KCL/KVL linear equations in $O(m^3)$ time to augment flow and the total running time in a given layered network to augment flow is $O(m^4)$ time.

Lemma 10. *For a given layered network, the total running time to augment flow by MLSDP is* $O(m^4)$

Proof. Note that we need $O(n+m)$ time to construct an m' linear equations for solving the electron flow δ. The system of linear equations by Kirchhoff's circuit laws can be solved in $O((m')^3) = O(m^3)$ time. If some of the electron flows violate the orientation, we select the most negative one to disconnect from A' and this step needs $O(m)$ time. Then performing *update-tree* step to update spanning tree T and removing unreachable nodes if necessary need $O(2m)$ time. If all electron flows are nonnegative, we perform *augment-flow-LSDP* step to improve flow in $G'(x)$. Searching for the maximum step length needs $O(m)$ time and improving flow and add the saturated arc to $REMOVEARC\_LIST$ also needs $O(m)$ because we need to scan all admissible arcs in A'. Then we remove the saturated arcs step by step by performing *update-tree* procedure and the running time is $O(m)$.

Concluding the above steps, the *calculate-electron-flow* step needs $O(n + m + m^3 + 2m)$ to obtain electron flows of admissible arcs. Then *augment-flow-LSDP* needs $O(m + m + 2m)$ to improve flow, remove saturated arcs, and update tree T. Therefore, we perform $O(m(n + 7m + m^3)) = O(m^4)$ time in doing *calculate-electron-flow* and *augment-flow-LSDP* until the flows passing through current layered network is optimal. $\square$

Then we show MLSDP will construct at most nm layered network to augment flow.

Lemma 11. *Until flow x is max-flow, we will perform the construct-layered-network-LSDP at most nm times*

Proof. Note that we may construct a new layered network in which the distance label associated with t remains the same as previous iteration. Now we show such layered networks will be created for at most m times. Suppose $d(t) = \alpha$ at some intermediate iterations, and the layered network has β admissible arcs. We know that s and t can be connected by at least one path of α admissible arcs in current layered network. So, if we remove more than $\beta - \alpha$ admissible arcs, all the shortest paths of β admissible arcs in residual networks will be disconnected, which forces the new layered network to be constructed, if there is any, by shortest paths of $\beta + 1$ admissible arcs. In other words, there are at most $\beta - \alpha \leq m$ different layered networks of the same value of $d(t)$ that can be constructed. Since $d(t)$ will increase at most n times, we will construct at most nm layered networks. $\square$

Based on Lemma 10 and 11, we can conclude that the MLSDP algorithm runs in $O(nm(m^4)) = O(nm^5)$. We have thus established the following theorem

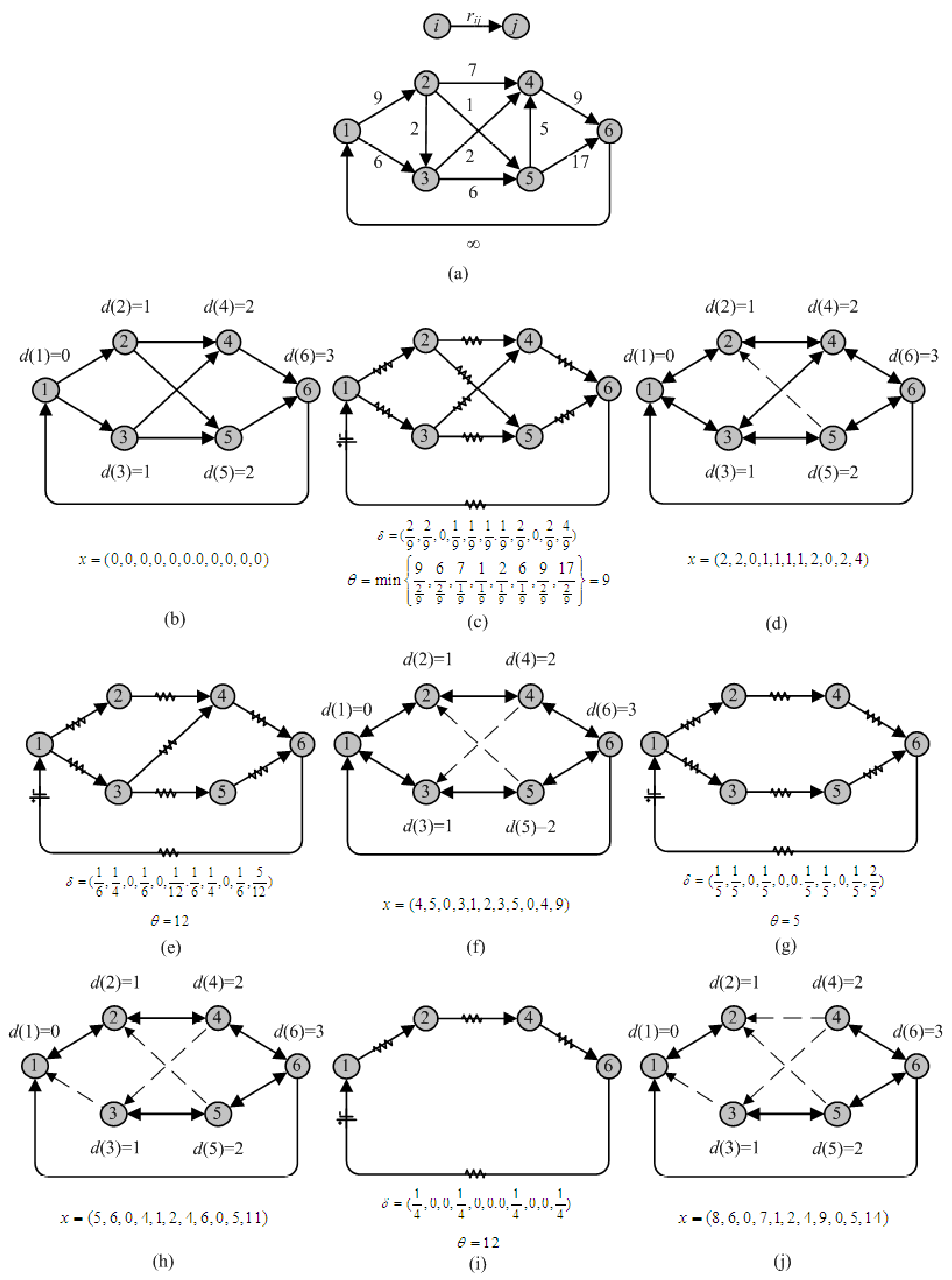
Theorem 12. *The MLSDP algorithm runs in $O(nm^5)$ time*

4.3. An Illustrative Example for MLSDP

Here we give a small maximum flow problem to illustrate how MLSDP algorithm works. In Figure 4.7(a), we would like to ship as much flow as possible from node 1 to 6. $N = \{1, 2, 3, 4, 5, 6\}$, $A = \{(1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (3,5), (4,6), (5,4), (5,6)\}$, $s = 1$, $t = 6$ and $u = (9, 6, 2, 7, 1, 2, 6, 9, 5, 17)$. The artificial arc $(6,1)$ is added with infinite capacity.

We perform a BFS search to construct the sets of admissible node $N' = \{1, 2, 3, 4, 5, 6\}$ and arc $A' = \{(1,2), (1,3), (2,4), (2,5), (3,4), (3,5), (4,6), (5,6), (6,1)\}$ which forms a layered network as illustrated in Figure 4.7(b). We then transform this layered network to an electrical network as illustrated in Figure 4.7(c), and calculate the electron flows $\delta = \left(\frac{2}{9}, \frac{2}{9}, 0, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{2}{9}, 0, \frac{2}{9}, \frac{4}{9}\right)$ to improve x to be $(2, 2, 0, 1, 1, 1, 1, 2, 0, 2, 4)$ with $\theta = \min\left\{\frac{9}{\frac{2}{9}}, \frac{6}{\frac{2}{9}}, \frac{7}{\frac{1}{9}}, \frac{1}{\frac{1}{9}}, \frac{2}{\frac{1}{9}}, \frac{6}{\frac{1}{9}}, \frac{9}{\frac{2}{9}}, \frac{17}{\frac{2}{9}}\right\} = 9$. Since arc $(2,5)$ is saturated, we remove this arc from A' in the beginning of the second inner-iteration. The algorithm continues to solve for electron flows $\delta = \left(\frac{1}{6}, \frac{1}{4}, 0, \frac{1}{6}, 0, \frac{1}{12}, \frac{1}{6}, \frac{1}{4}, 0, \frac{1}{6}, \frac{5}{12}\right)$ in Figure 4.7(e) with the maximum step length $\theta = 12$ to improve x to be $(4, 5, 0, 3, 1, 2, 3, 5, 0, 4, 9)$, where the saturated arc $(3,4)$ is removed from A'.

In the third inner-iteration, the electron flows $\delta = \left(\frac{1}{5}, \frac{1}{5}, 0, \frac{1}{5}, 0, 0, \frac{1}{5}, \frac{1}{5}, 0, \frac{1}{5}, \frac{2}{5}\right)$ and the maximum step length $\theta = 5$ is calculated in Figure 4.7(g). A better feasible flow $x = (5, 6, 0, 4, 1, 2, 4, 6, 0, 5, 11)$ is obtained and arc $(1,3)$ is saturated. We remove arc $(1,3)$ from A' and decrease the number of incoming admissible arcs of



(a)
x = (0,0,0,0,0,0,0,0,0,0,0,0)
(b)
θ = 9
(c)
x = (2,2,0,1,1,1,1,2,0,2,4)
(d)
θ = 12
(e)
x = (4,5,0,3,1,2,3,5,0,4,9)
(f)
θ = 5
(g)
x = (5,6,0,4,1,2,4,6,0,5,11)
(h)
θ = 12
(i)
x = (8,6,0,7,1,2,4,9,0,5,14)
(j)

Figure 4.7. An example of MLSDP for max-flow problem-Iteration1

node 3 to zero. Therefore node 3 is removed from N' and node 5 also needs to be removed at the same time. We repeat these steps to calculate the electron flows $\delta = \left(\frac{1}{4}, 0, 0, \frac{1}{4}, 0, 0, 0, \frac{1}{4}, 0, 0, \frac{1}{4}\right)$ and the maximum step length $\theta = 12$ is calculated in Figure 4.7(i). After obtaining a better feasible flow $x = (8, 6, 0, 7, 1, 2, 4, 9, 0, 5, 14)$, arc $(2, 4)$ is saturated and removed from A'. The number of outgoing admissible arcs of node 2 decreases to zero, so we need be removed node 2 from N'. After finishing this step, no more flow can be sent in current layered network. We recalculate the distance labels by BFS and construct a new layered network as illustrate in Figure 4.8(a).

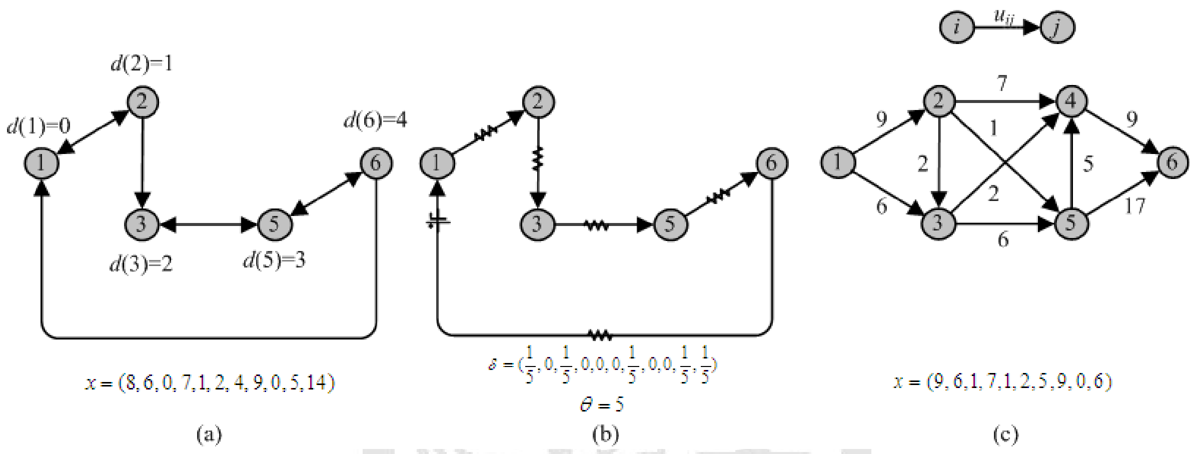

d(2)=1
d(1)=0
d(6)=4
d(3)=2
d(5)=3
$x = (8, 6, 0, 7, 1, 2, 4, 9, 0, 5, 14)$
(a)
$\delta = (\frac{1}{5}, 0, \frac{1}{5}, 0, 0, 0, \frac{1}{5}, 0, 0, \frac{1}{5}, \frac{1}{5})$
$\theta = 5$
(b)
u_{ij}
$x = (9, 6, 1, 7, 1, 2, 5, 9, 0, 6)$
(c)

Figure 4.8. An example of MLSDP for max-flow problem-Iteration2

Now $N' = \{1, 2, 3, 5, 6\}$ and $A' = \{(1, 2), (2, 3), (3, 5), (5, 6), (6, 1)\}$. Transforming this layered network to an electrical network, we calculate the electron flows $\delta = \left(\frac{1}{5}, 0, \frac{1}{5}, 0, 0, 0, \frac{1}{5}, 0, 0, \frac{1}{5}, \frac{1}{5}\right)$ and the maximum step length $\theta = 5$ to improve the flow x to be $(9, 6, 1, 7, 1, 2, 5, 9, 0, 6, 15)$. After removing the saturated arc $(1, 2)$ from A' and node 1 from N', no more flow can be sent again since $d(t) = \infty$, which means there exists no $s - t$ augmenting path. We then terminate the algorithm and the optimality is attained.

Note that MLSDP also gives fractional maximum flow as PAA or FSA. Thus we can also apply the flow decomposition techniques proposed in Section 3.4.

4.4. Another MLSDP Implementation

In the original LSDP algorithm proposed by Wang et al. [**51**], there are two types of arcs that will be disconnected in the procedure: (1) those arcs reaching its residual capacity (i.e. disconnected due to θ), and (2) those arcs containing the most negative electron flows. The original LSDP algorithm has to check whether those disconnected arcs of the second type have to be reconnect or not, depending on whether the resultant electrical potentials on the end nodes for an arc consist with its orientation. In MLSDP, we do not conduct such checks to reconnect those arcs. As a result, when MLSDP disconnect a layered network, the next layered network may still have the same $s-t$ path length. This means MLSDP may construct more than $n-1$ layered networks, although we have already shown that at most nm layered networks may be constructed by Lemma 11. Here we explain how to use another MLSDP implementation, denoted by Algorithm MLSDPI based on the original LSDP algorithm, to solve the maximum flow for a layered network, as illustrated in Figure 4.10, and why it also has the same complexity $O(nm^5)$ as MLSDP.

If we check each disconnected arc of the second type for whether it has to be reconnect by procedure *reconnect-arc* as illustrated in Figure 4.9 or not by the electrical potentials on its end nodes whenever we calculate a feasible δ, each augmentation still disconnects an arc, which leads to an $O(m)$ iteration to disconnect a layered network. Within each such iteration, it may calculate δ for $O(m)$ times since each calculation may give at most one arc that contains the most negative electron flow and such an arc has to remain disconnected (i.e. belonging to type 1) until a feasible δ is calculated. In other words, it takes $O(m^5)$ time to calculate the maximum flow for a layered network since each calculation of δ takes $O(m^3)$ time, MLSDP at most calculates m times of feasible δ and there are at most m times of infeasible δ calculation between consecutive times of feasible δ. Since there are at most $n-1$ layered networks, this MLSDP implementation takes $O(nm^5)$ time.

Although MLSDPI has the same complexity as MLSDP, it may solve larger system of linear equations than MLSDP since it reconnects some arcs that are disconnected in MLSDP. As a result, MLSDPI tends to take more time in solving for δ, which is the bottleneck operation for MLSDP-like algorithms. Therefore we would still use MLSDP, instead of MLSDPI, in our computational experiments.

```
Procedure 18 Reconnect-arc($N'', d, G'(x)$)
$index := 1; i = N''(index)$;
While $i \neq t$ do
    For each arc $(i, j) \in A$ do
        If $d(j) = d(i) + 1$ and $r_{ij} > 0$ then
            If $j \notin N'$ then
                $N' \cup j; spanarc_{ij} := 1; A' := A' \cup \{(i, j)\}$ ;
            Else then $spanarc_{ij} := 0$;
            $out(i) := out(i) + 1; in(j) := in(j) + 1$;
        Else if $d(i) = d(j) + 1$ and $r_{ij} > 0$ then
            If $i \notin N'$ then
                $N' \cup i; spanarc_{ji} := 1; A' := A' \cup \{(j, i)\}$ ;
            Else then $spanarc_{ji} := 0$;
            $out(j) := out(j) + 1; in(i) := in(i) + 1$;
Return $G'(x)$;
```

Figure 4.9. Pseudo-code for reconnecting arc

```
Algorithm 19 MLSDPI($G(x)$)
Initialization:
    For each node $i \in N$ do
        $out(i) := 0; in(i) := 0; d(i) := \infty$;
    For each arc $(i, j) \in A$ do
        $spanarc_{ij} := -1$;
$G'(x) :=$ Construct-layered-network-LSDP($G(x), out, in, d, spanarc$);
While $d(t) \neq \infty$ do
    While $s \in N'$ and $t \in N'$ do
        $\delta :=$ Calculate-electron-flow($G'(x), T, out, in, spanarc$);
        Augment-flow-LSDP($G'(x), \delta.T, spanarc, out, in$);
        Reconnect-arc($N'', d, G'(x)$);
    End while;
    $G'(x) :=$ Construct-layered-network-LSDP($G(x), out, in, d, spanarc$);
End while;
```

Figure 4.10. MLSDPI algorithm

4.5. Speed-up Techniques

Since solving the Kirchhoff's equations takes $O(m^3)$ time, which is the bottleneck operation in the entire algorithm that takes $O(nm^5)$. Any techniques to solve the Kirchhoff's equations in less time would shorten the empirical running time of MLSDP. Here we will use the UMFPACK sparse linear solver, instead of the Gauss-Jordan elimination, to solve the m' linear equations of KCL/KVL. The UMFPACK, downloadable from Davis's website [**17**], is a library for solving the unsymmetrical sparse linear systems, $Ax = b$. Details of UMFPACK can be found in [**13, 14, 15, 16**]. The main idea of UMFPACK for solving $Ax = b$ is to factorize the sparse matrix A into $PRAQ = LU$, where P and Q are permutation matrices, R is a diagonal matrix, L is a lower triangular matrix with a diagonal of 1, and U is an upper triangular matrix. The first solution is computed by backward substitutions and then improved by iterative refinement.

The input matrices for UMFPACK have to be the compressed sparse column (CSC) format, which stores the nonzero elements of matrix A into three separate arrays: the first index array contains the start of the columns, the second index array contains the row indices, and the third array contains the numerical values. For example, the matrix A

$$\begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ -1 & 1 & -1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

has the following column major coordinate data:

column index array:	0 4 7 10 12 14
row index array:	0 1 3 4 0 2 3 1 3 4 2 3 0 4
values	1 −1 −1 1 1 −1 1 1 −1 1 1 1 −1 1

The solution procedure for a sparse linear system by UMFPACK contains three steps: the first step is a symbolic analysis which pre-orders the columns of A to reduce fill-ins, finds the supernodal column elimination tree, post-orders the tree, and then returns the symbolic object. The second step is a numerical factorization which performs the numerical LU factorization for the coefficient matrix using the symbolic object and returns the numeric object. Finally, The last step solves the sparse linear system using the numeric object.

In our computational tests (see Section 5.3.3 for details), the UMFPACK implementation of MLSDP performs much better than the Gauss-Jordan implementation of MLSDP.

4.6. Summary

Instead of directly solving for the electron flows on the original network as the LSDP algorithm proposed by Wang et al. [**51**], our proposed MLSDP iteratively solves for the electron flows on layered networks. The original LSDP algorithm does not guarantee to calculate the maximum flow in polynomial time. On the other hand, our MLSDP algorithm can terminate in $O(nm^5)$ time. We also suggest to solve the Kirchhoff's circuit laws by efficient sparse linear system solver to improve the efficiency of MLSDP implementation.

CHAPTER 5

COMPUTATIONAL EXPERIMENTS AND ANALYSES

In this chapter we list and summarize the results of our computational experiments for solving randomly generated maximum flow problems by our proposed algorithms and several state-of-the-art algorithms. In particular, Section 5.1 introduces the settings of our computational experiments; Section 5.2 introduces the problem families and how we use the generators to generate problem instances; Section 5.3 compares the performance for our proposed algorithms. Then, we select our best implementations to compare with other state-of-the-art maximum flow algorithms in Section 5.4. Finally, we give a summary in Section 5.5.

5.1. Computational Setup

The testing environment was set using cygwin 1.5.25-15 on a ASUS M5100 PC with an Inter Core 2 Duo, CPU 1.86 GHz, 3.00 gigabytes of memory and Microsoft Windows XP SP3 as its OS. All the programs are written in C language and complied with the *gcc* using the -O3 optimization option. Twelve maximum flow algorithms are tested: (1) PAA without doubly linked list implementation, denoted by PAA1; (2) PAA with doubly linked list implementation, denoted by PAA2; (3) PAA with doubly linked list and pushing and pulling implementation, denoted by PAAFB; (4) PSA, the scaling version of PAA2; (5) FSA; (6) the MLSDP implementation using Gauss-Jordan elimination for Kirchhoff's circuit laws, denoted by MLSDP1; (7) the MLSDP implementation using UMFPACK implementation, denoted by MLSDP2; (8) Dinic's algorithm [**18**] implemented by [**46**], denoted by DINIC; (9) Dinic's algorithm implemented by [**11**], denoted by; (10) Goldberg and Tarjan's preflow-push algorithm [**35**] using highest-label-first implementation, denoted by HI_PR; (11) Fujishige and

Isotani's [**26**] scaling version of MA ordering, denoted by FS1; (12) Matsuoka and Fujishige's algorithm [**41**] using MA ordering and preflows, denoted by FMAP. Program DINIC is available from DIMACS ftpsite [**52**], but it can not process a network of more than 20000 nodes. Program DF and HI_PR are available from Goldberg's website [**32**].

We recorded the running time of all programs in seconds, without considering the time spent for processing the input and output.

5.2. Problem Instances

We tested five problem families (GENRMF-LONG, GENRMF-WIDE, WAS-10, AK, and ACU) created by four generators: GENRMF, WASHINGTON, AK, and ACYCLIC-DENSE. These generators can be downloaded from DIMACS ftpsite [**52**] or Goldberg personal website [**33**].

The GENRMF network families have b grid frames where each frame is composed by an $a \times a$ grid. Each node in a frame connects to its horizontal and vertical neighbors of the same frame as illustrated in Figure 5.1(a). Arcs between frames are randomly generated. Figure 5.1(b) shows each GENRMF network has a^2b nodes and $5a^2b - 4ab - a^2$ arcs. The capacities for arcs inside each frame arc uniformly chosen from the range $[c_1, c_2]$, and capacities for arcs between frames are set to be $c_2 \times a \times a$. The source and sink nodes are located in the first and last frame, respectively.

Depending on different settings of a and b, we have tested two types of GENRMF network families: GENRMF-LONG and GENRMF-WIDE, where $a = 2^{k_1/4}$, $b = 2^{k_1/2}$, $c_1 = 1$, $c_2 = 100$ for the GENRMF-LONG family and $a = 2^{2k_1/5}$, $b = 2^{k_1/5}$, $c_1 = 1$, $c_2 = 100$ for the GENRMF-WIDE family, with different k_1's.

The AK generator generates a network of $4k_2 + 6$ nodes and $6k_2 + 7$ arcs. An AK network consists two subnetworks, denoted by $N(1)$ and $N(2)$, which are connected in parallel where the source node connects to the source nodes of subnetworks $N(1)$ and $N(2)$ by arcs of very large capacities and the sink node connects to the sink nodes of

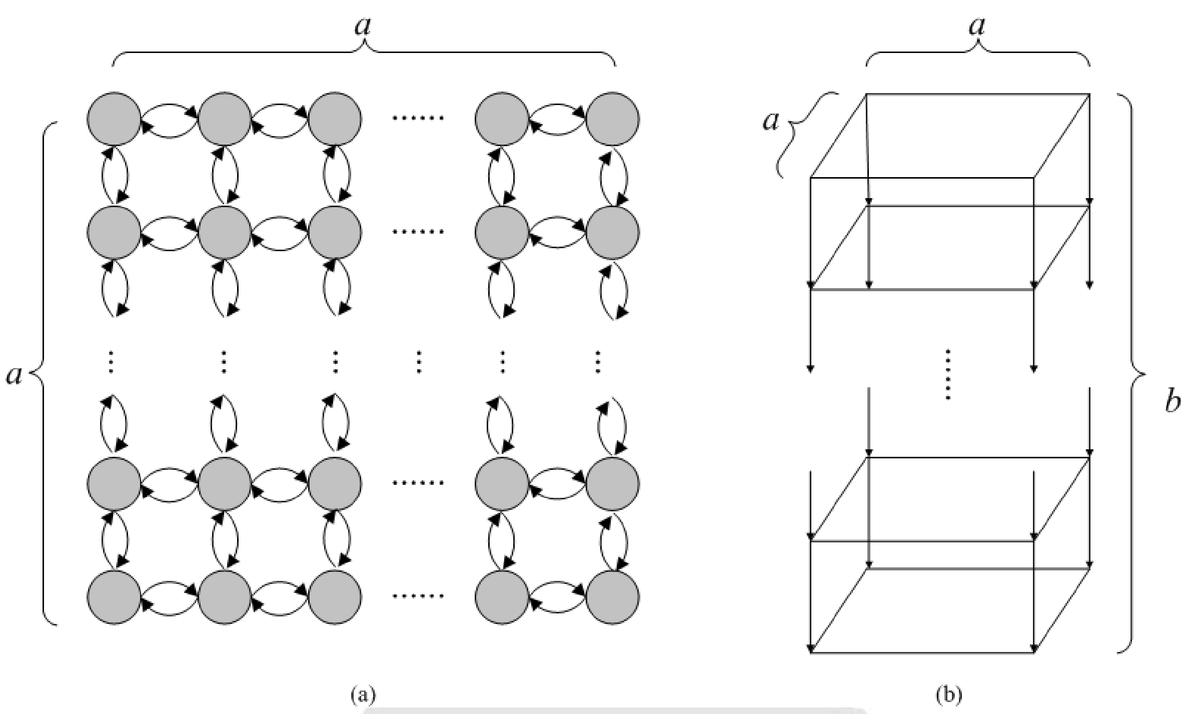


Figure 5.1. Genrmf network

$N(1)$ and $N(2)$ by arcs with very large capacities. $N(1)$ is illustrated in Figure 5.2, which contains $2k_2 + 2$ nodes and $3k_2 + 2$ arcs.

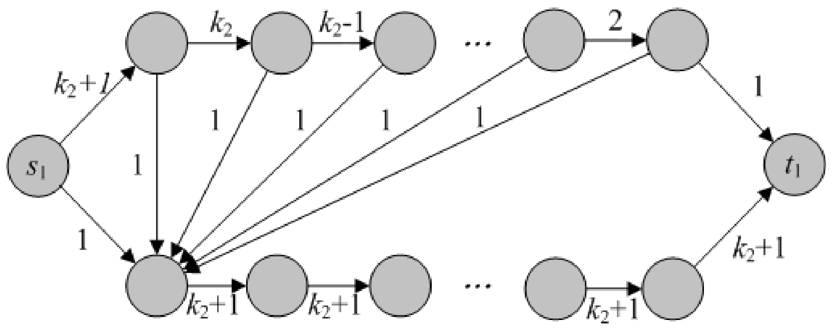


Figure 5.2. The subnetwork $N(1)$ of AK generator

Subnetwork $N(2)$ contains $2k_2 + 2$ nodes and $3k_2$ arcs which is illustrated in Figure 5.3. In general, an AK layered network maybe composed by one, two, or three simple augmenting paths, where the length of the augmenting path follows the sequence: 3, 5, 7,.., $k_2 + 3$, $k_2 + 4$, $k_2 + 5$,..., $2k_2 + 3$. As a result, there would be $1.5k_2 + 1$ layered networks. In general, augmenting path based algorithms, even using the layered network frameworks, may not take too much advantage for this network since there are many layered networks to be constructed in this network. The details of the AK network can be found in [**11**].

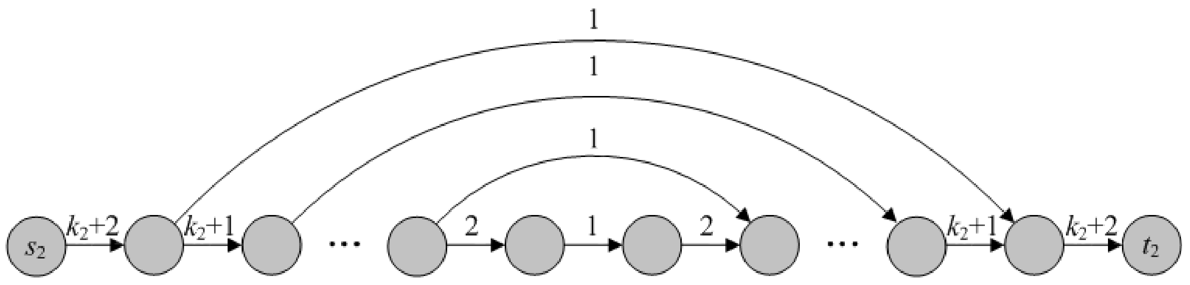


Figure 5.3. The subnetwork $N(2)$ of AK generator

We set the functional parameter to 10 for the WASHINGTON network family and denote it by WAS-10. The WASHINGTON network structure is illustrated in Figure 5.4, which contains $3k_3 + 3$ nodes and $4k_3 + 1$ arcs. The network contains k_3 paths disjointed with each other. Note that each augmenting path has the same number of arcs and will saturate an arc of unit capacity. In general, augmenting path based algorithms may not take too much advantage for this network since there are many augmenting paths, although there is only one layered network. Note that PAA, FSA, and MLSDP algorithms should take more advantages in this network since it is symmetric and all the bottleneck arcs will be saturated at one time. Details of the WASHINGTON network can be found in [**39**].

The last generator we used is ACYCLE-DENSE, which contains 2^{k_4} nodes where each node connects to all other nodes of larger indices by equal-capacity arcs as illustrated in Figure 5.5. We denote these networks by ACU. ACU contains only two layered networks. The second layered network (of length 2) contains many augmenting paths of the same capacity, which can be saturated by PAA, FSA, and MLSDP in one augmentation. On the other hand, Dinic's algorithm will require many augmentations.

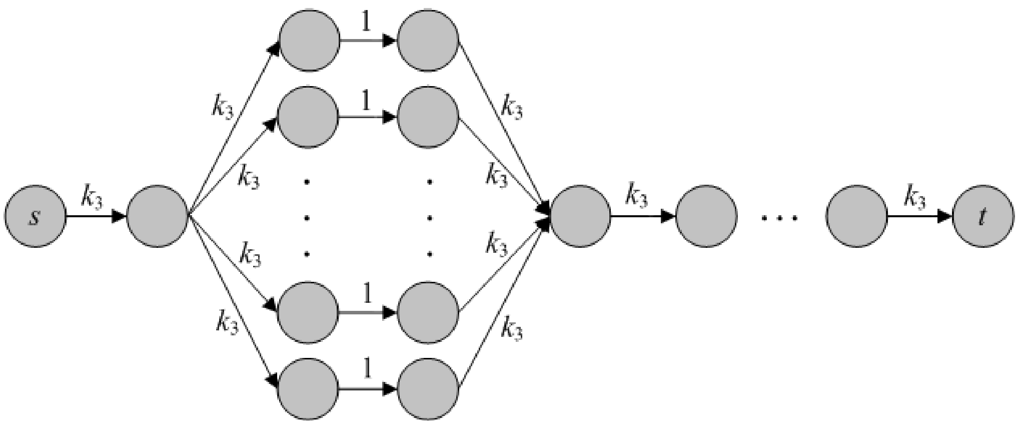


Figure 5.4. Washington-10 network

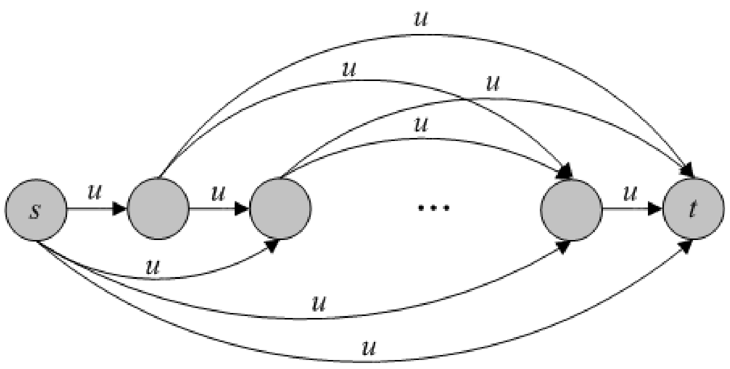


Figure 5.5. Acyclic-dense with equal capacity arcs network

5.3. Testings among Proposed Algorithms

First we generate three groups respectively for the GENRMF-LONG and GENRMF-WIDE network families, where each group contains 5 random cases of similar sizes but different topologies. In particular, besides the major grid and frame structure, random arcs may be generated to connect nodes between frames. On the other hand, the AK, WAS-10, and ACU networks will not generate random arcs but those arcs of fixed topology and capacities depending on some given parameters. Thus we only generate 3 groups, where there is only one case for each group, to represent small, medium, and large networks, for the AK, WAS-10, and ACU network families. For each group of networks, we solve the maximum flow by some tested maximum flow algorithms, and record the average running time and number of iterations in subsection 5.3.1, 5.3.2, and 5.3.3. We then choose efficient algorithms to compare with Dinic's, Goldberg and Tarjan's, Fujishige and Isotani's, and Matsuoka and Fujishige's algorithms and discuss the behavior of our proposed algorithms in subsection 5.4.

5.3.1. Comparing Different PAA Implementations

Here we compared the generic PAA implementation, denoted by PAA1, with three of its speed-up versions, denoted by PAA2, PAAFB, and PSA, respectively. The results are listed in Table 5.1. With the doubly linked list implementation, PAA2 performs better than PAA1 for solving GENRMF, AK, and ACU families. Both PAAFB and PSA are implemented based on PAA2. The testing results show that PAAFB performs better than PAA2 for solving GENRMF-LONG family, and performs similarly to PAA2 for other cases. PSA performs better than PAA2 in most cases except GENRMF-WIDE family. We also count the total number of augmentations (i.e. iterations) for PAA2, PAAFB, and PSA, which are listed in Table 5.2. Table 5.2 shows that PAAFB performs similarly to PAA2 in terms of number of augmentations, and PSA has the fewest augmentations, compared with PAA2 and PAAFB. Due to the special topologies of the WAS-10 and ACU networks, all PAA2, PAAFB, and PSA have the same number of augmentations for calculating the maximum flows on these networks. The results show that the doubly linked list and scaling phase techniques do help reduce the empirical running time of PAA, thus we will use PAA2 and PSA to compare with other maximum flow algorithms in the following subsections.

5.3.2. Comparing PAA2 with FSA

To have a fair comparison, we also exploit the doubly linked list to implement FSA, and compare it with PAA2 in both running time and number of augmentations, as shown in Table 5.3. FSA only performs better than PAA2 due to fewer flow augmentations for solving AK networks, but performs worse than PAA2 for solving GENRMF-LONG,

Table 5.1. Comparing running time of PAA with its implementation algorithms

Network		**Running time (second)**			
Family	(n,m)	**PAA1**	**PAA2**	**PAAFB**	**PSA**
GENRMF -LONG	$(65536, 311040)$	535.891	324.74	294.05	170.113
	$(130682, 625537)$	2152.531	1421.287	1205.865	706.066
	$(270848, 1306607)$	10240.866	6066.44	4948.694	3136.519
GENRMF -WIDE	$(65025, 314840)$	306.703	130.566	128.644	303.728
	$(123210, 599289)$	1283.378	549.044	557.084	1099.791
	$(259308, 1267875)$	5976.716	2464.14	2672.694	4612.066
AK	$(65542, 98311)$	480.375	329.5776	311.962	83.172
	$(131078, 196615)$	2117.578	1386.224	1331.585	405.329
	$(262150, 393223)$	8897.843	5730.197	5557.406	1582.547
WAS-10	$(49155, 65537)$	0.016	0.0155	0.031	0.0282
	$(98307, 131073)$	0.031	0.031	0.062	0.084
	$(196611, 262145)$	0.078	0.07	0.14	0.1872
ACU	$(1024, 523776)$	0.125	0.063	0.078	0.047
	$(2048, 2096128)$	0.532	0.313	0.328	0.234
	$(4096, 8386560)$	2.485	1.375	1.391	1.188

Table 5.2. Comparing total number of augmentations of PAA with its implementation algorithms

Network		**# of augmentations**		
Family	(n,m)	**PAA2**	**PAAFB**	**PSA**
GENRMF -LONG	$(65536, 311040)$	51243.6	51239.8	23928
	$(130682, 625537)$	105204.6	105200.2	46416.8
	$(270848, 1306607)$	223705.8	223644.6	111360.2
GENRMF -WIDE	$(65025, 314840)$	57566.4	57592.8	44323.8
	$(123210, 599289)$	110899.2	110953.6	90539.6
	$(259308, 1267875)$	240152	240278.2	209048.2
AK	$(65542, 98311)$	32770	32770	10928
	$(131078, 196615)$	65538	65538	21850
	$(262150, 393223)$	131074	131074	43696
WAS-10	$(49155, 65537)$	1	1	1
	$(98307, 131073)$	1	1	1
	$(196611, 262145)$	1	1	1
ACU	$(1024, 523776)$	2	2	2
	$(2048, 2096128)$	2	2	2
	$(4096, 8386560)$	2	2	2

GENRMF-WIDE, and WAS-10 families. For ACU networks, PAA2 performs a little better than FSA, although they have the same number of flow augmentations.

Now we take a closer look at AK networks to explain why FSA performs better than PAA2 in such networks. Note that AK networks are carefully designed so that

Table 5.3. Comparing PAA with FSA

Network		Running time (second)		# of augmentations	
Family	(n, m)	**PAA2**	**FSA**	**PAA2**	**FSA**
GENRMF -LONG	$(65536, 311040)$	324.74	780.406	51243.6	107550.4
	$(130682, 625537)$	1421.287	3245.1	105204.6	215173
	$(270848, 1306607)$	6066.44	11794.694	223705.8	446722.8
GENRMF -WIDE	$(65025, 314840)$	130.566	325.516	57566.4	128798.6
	$(123210, 599289)$	549.044	1423.503	110899.2	248974.4
	$(259308, 1267875)$	2464.14	6415.369	240152	559278.6
AK	$(65542, 98311)$	329.5776	285.919	32770	24578
	$(131078, 196615)$	1386.224	1196.606	65538	49154
	$(262150, 393223)$	5730.197	4987.265	131074	98306
WAS-10	$(49155, 65537)$	0.0155	0.031	1	1
	$(98307, 131073)$	0.031	0.078	1	1
	$(196611, 262145)$	0.07	0.152	1	1
ACU	$(1024, 523776)$	0.063	0.078	2	2
	$(2048, 2096128)$	0.313	0.297	2	2
	$(4096, 8386560)$	1.375	1.375	2	2

each layered network is composed of either one or two $s-t$ paths. When an AK layered network is a single $s-t$ path, FSA performs the same as PAA2, which should also be the same as conventional augmenting path algorithm. On the other hand, when an AK layered network contains two disjoint $s-t$ paths, FSA can saturate both paths in one iteration due to the special structure of AK networks, which has to be done by two augmentations using PAA2. As a result, FSA will have fewer flow augmentations than PAA2 for solving AK networks, which in turn reduces its running time.

To be more specific, we use the example in Figure 5.6 to explain this situation. In this example, there are two shortest augmenting paths in this AK layered network. When performing PAA, we first distribute $\frac{10^6-2}{2\times10^6-3}$ unit augmenting-flow to the path in the upper subnetwork $N(1)$, and $\frac{10^6-1}{2\times10^6-3}$ unit augmenting-flow to the path in the lower subnetwork $N(2)$. Obviously, the bottleneck node will occur in subnetwork $N(2)$ and the maximum step length $\theta = \frac{2\times10^6-3}{10^6-1}$. After augmenting 1 unit of flow, subnetwork $N(2)$ is saturated and the remaining capacity of subnetwork $N(1)$ is $1 - \frac{10^6-2}{10^6-1}$. Therefore, PAA2 will need another augmentation to saturate subnetwork $N(1)$. However, when performing FSA, the unit augmenting flows distributed to subnetwork $N(1)$ and

$N(2)$ are equal, which saturates both the bottleneck arcs of $N(1)$ and $N(2)$ (since they happen to be equal). As a result, FSA only augment once which saturates both $N(1)$ and $N(2)$ at the same time. This explains why FSA may perform better than PAA2 for calculating maximum flows in AK networks.

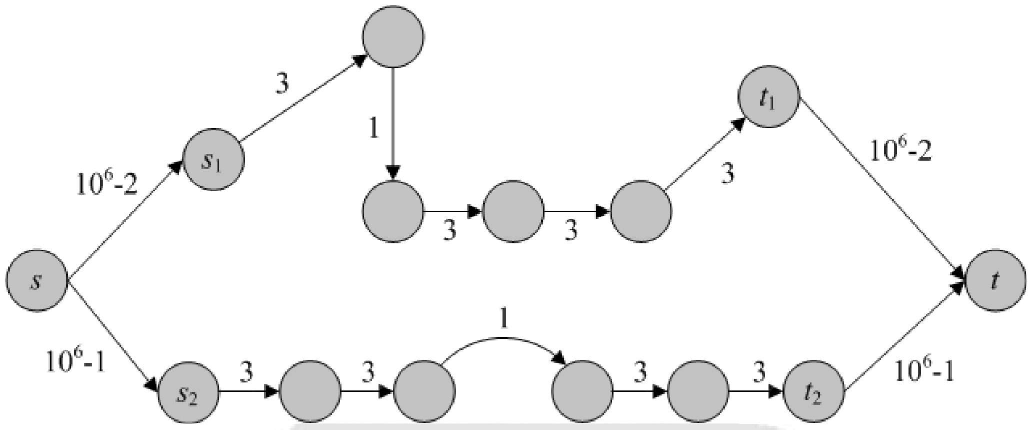


Figure 5.6. An example of layered network constructed from AK family

5.3.3. Comparing PAA with MLSDP

We compared PAA2 with MLSDP1 and MLSDP2 on groups of small networks generated by GENRMF-LONG, GENRMF-WIDE, AK, WAS-10, and ACU. The results are shown in Table 5.4. It is clear that the Gauss-Jordan method used in MLSDP1 takes much more time than the UMFPACK solver used in MLSDP2 for all cases. In terms of number of augmentations, MLSDP2 has fewer iterations than PAA2 for AK networks. This is because when a layered network is constructed with two disjoint paths, the electron flows passing through these two disjoint paths have to be equal since they contain the same number of arcs (i.e. resistances), and then one augmentation can saturate the entire layered network, similar to FSA. On the other hand, MLSDP2 has more augmentations than PAA2 for GENRMF-LONG and GENRMF-WIDE networks. From these results, we also learn that the nondegenerate pivoting based algorithm such as MLSDP may not take too much advantages in maximum flow problems.

Table 5.4. Comparing PAA with MLSDP

Network		**Running time (second)**			**# of augmentations**	
Family	(n, m)	**PAA2**	**MLSDP1**	**MLSDP2**	**PAA2**	**MLSDP2**
GENRMF -LONG	$(256, 1008)$	0.0053	14.516	1.099	141.4	163
	$(756, 3240)$	0.026	1503.651	30.041	486	758
	$(1225, 5376)$	0.0677	10418.302	106.161	862.4	1289.8
GENRMF -WIDE	$(363, 1562)$	0.0053	63.693	2.77	236	345
	$(676, 3003)$	0.0203	547.099	10.716	462.8	660.2
	$(1024, 4608)$	0.0623	4018.271	61.473	737.2	1612.4
AK	$(1030, 1543)$	0.062	19.813	2.703	514	386
	$(2054, 3079)$	0.281	263.703	20.968	1026	770
	$(4102, 6151)$	1.094	3947.484	165.327	2050	1538
WAS-10	$(771, 1025)$	0.0023	0.062	0.031	1	1
	$(1539, 2049)$	0.0078	0.282	0.093	1	1
	$(3075, 4097)$	0.016	1.171	0.407	1	1
ACU	$(256, 32640)$	0.005	0.312	0.015	2	2
	$(512, 130816)$	0.016	2.579	0.047	2	2
	$(1024, 523776)$	0.079	20.344	0.234	2	2

5.4. Testings on Proposed Algorithms and Other Algorithms

Concluding subsection 5.3.1, 5.3.2, and 5.3.3, PAA2 and PSA are the most efficient implementations among all of our proposed maximum flow algorithms. Therefore, we only compare these two algorithms with five other state-of-the-art maximum flow algorithms: DINIC, DF, HI_PR, FS1, and FMAP.

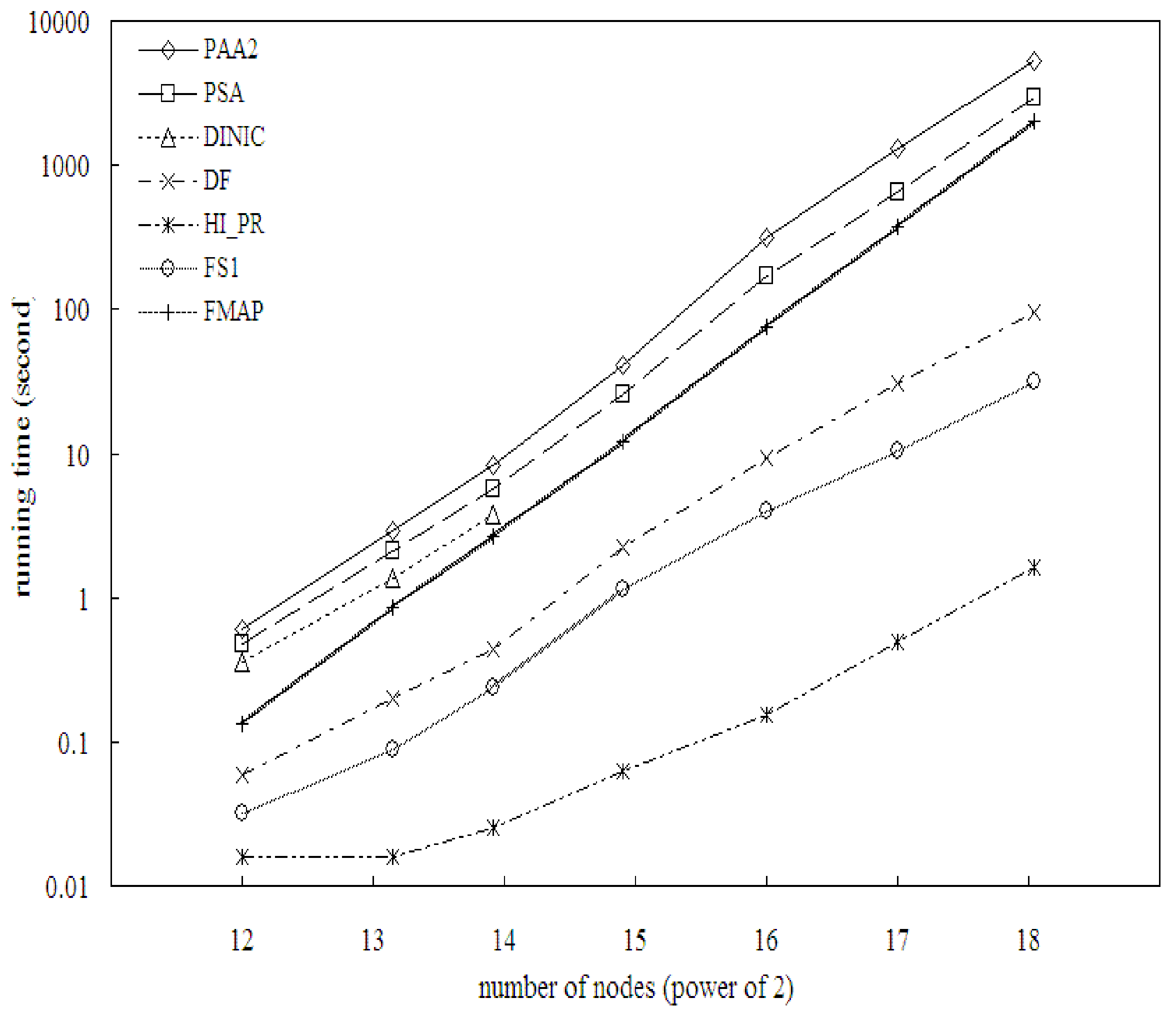
The results of our computational testings are illustrated in Figure 5.7, 5.8, 5.9, 5.10, and 5.11 for network families of GENRMF-LONG, GENRMF-WIDE, AK, WAS-10, and ACU, respectively. For the GENRMF-LONG and GENRMF-WIDE families, we generate seven groups of 10 random cases, where each group represent cases of similar sizes. For the AK, WAS-10, and ACU families, we generator seven groups of one case and run 10 times to calculate the average time. For each case, we solve its maximum flow by each of the seven algorithms (i.e. PAA2, PSA, DINIC, DF, HI_PR, FS1, and FMAP). Then we record the average running time for each group and each algorithm, and plot the results in the figures.

Figure 5.7 and 5.8 indicate that HI_PR and DF are more efficient, PAA2 and PSA takes much more time for most cases except large cases of GENRMF-WIDE where FMAP performs the worst.

Figure 5.9 shows that FMAP is the most efficient algorithm and DINIC takes much more time than others for AK networks. The performance of PAA2 in AK networks is already better than its performance in solving GENRMF networks and the performance of PSA is better than DF.

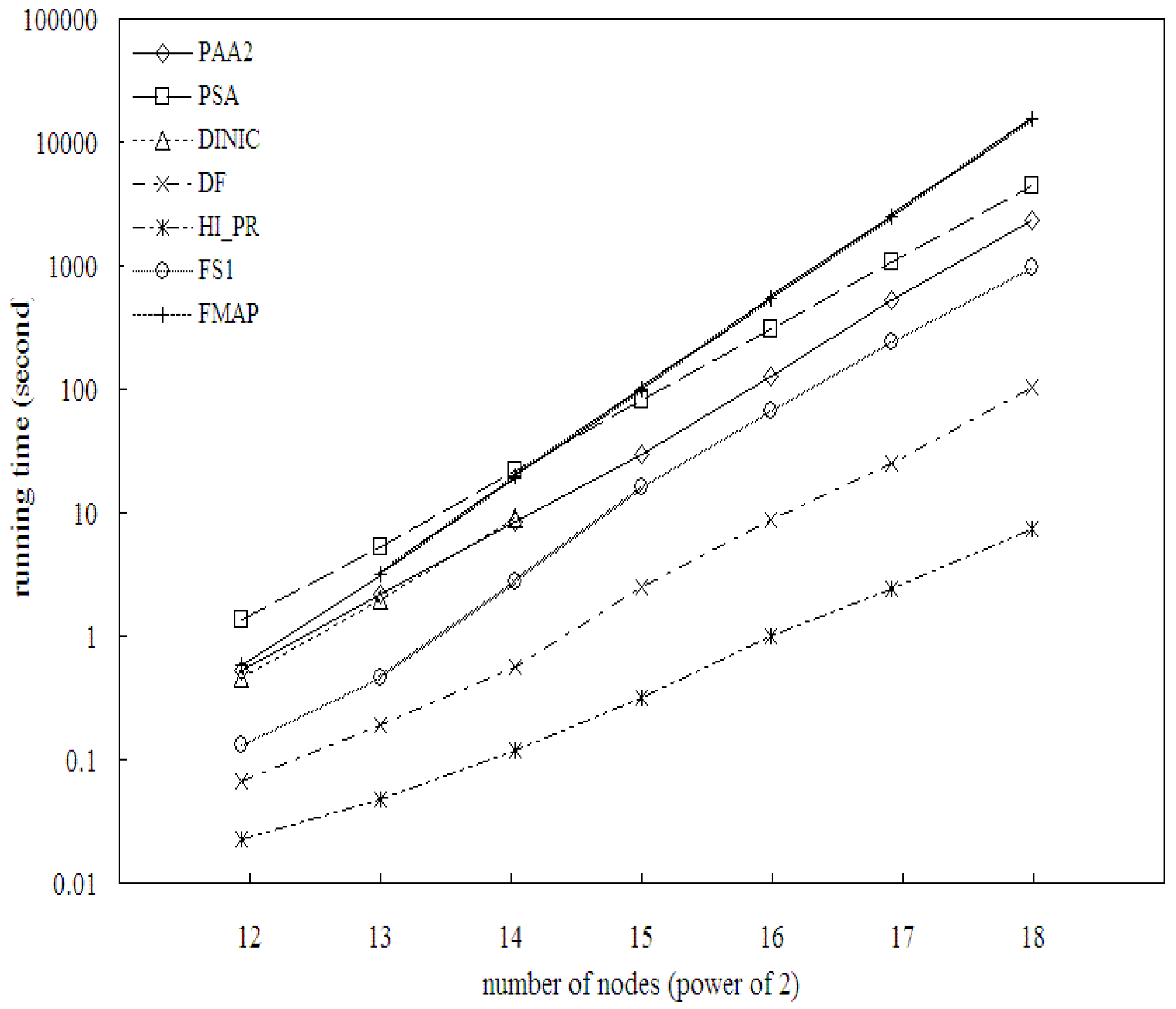
Figure 5.10 shows results for the WAS-10 family. PAA2 and PSA outperform DINIC, FS1, and FMAP and PAA performs almost the same as HI_PR. We investigated the steps of PAA2 in this special structure networks and found that PAA2 only augmented one time to obtain the maximum flow. With this special structure networks, PAA2 terminates within one iteration. In particular, it only constructs a single layered network in $O(m)$ time, augments flows once which takes $O(2n + 3m)$ time, and then terminates. Thus the theoretical complexity for PAA2 in solving WAS-10 networks can be reduced to $O(m)$ time. This explains why PAA2 almost has the best running time (similar to HI_PR) for solving WAS-10 networks. On the other hand, FMAP performs the worst in all cases of this network family.

Figure 5.11 shows results for the ACU in each case. PAA2 and PSA outperform DINIC and perform a little faster than FMAP. Note that for this specific network family, PAA2 also takes $O(m)$ time since it suffices to perform two iterations of flow augmentation to calculate the maximum flow.



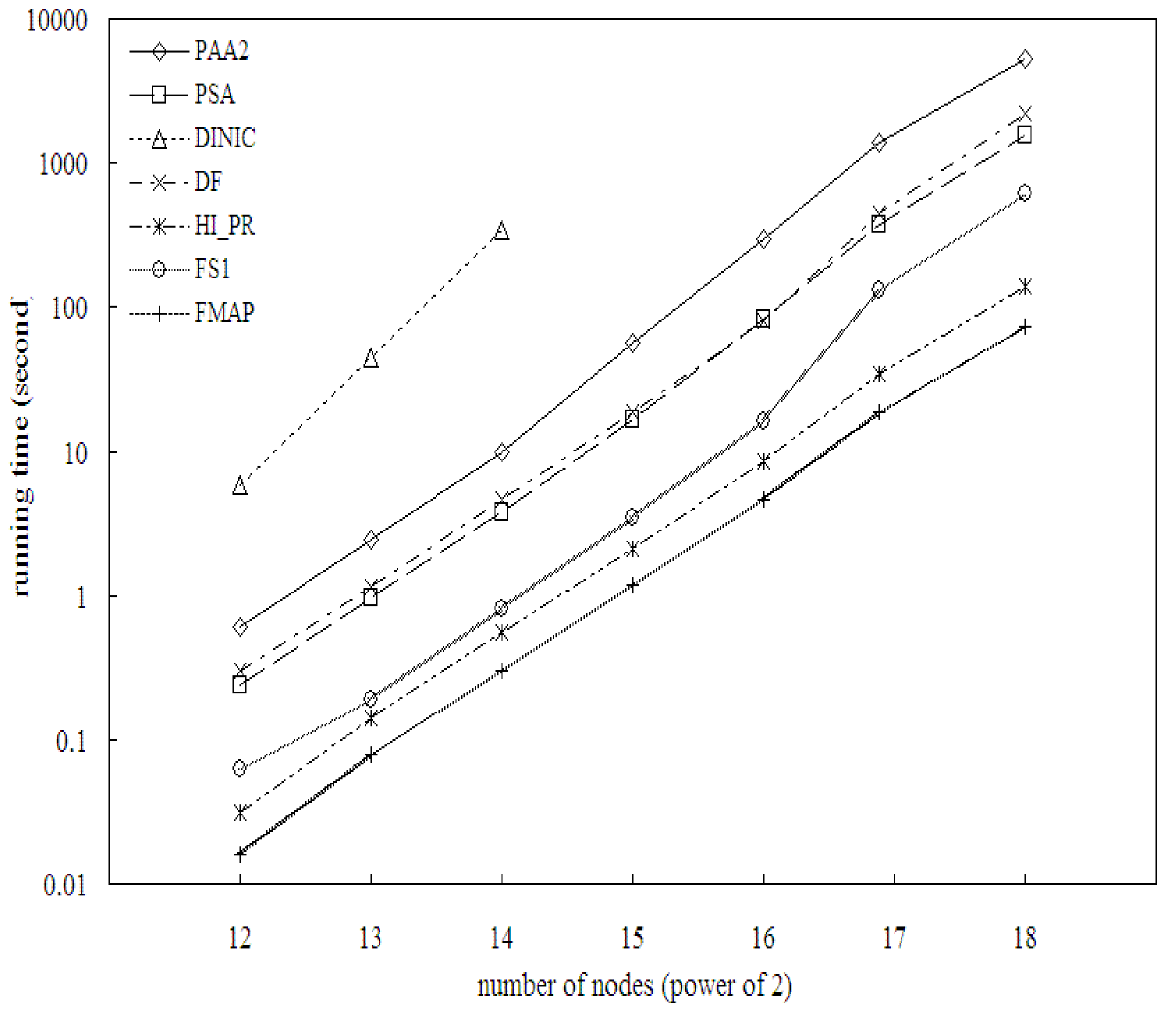
GENRMF-LONG		Running time (second)						
n	**m**	**PAA2**	**PSA**	**DINIC**	**DF**	**HI_PR**	**FS1**	**FMAP**
4096	**18368**	0.599	0.472	0.354	0.059	0.016	0.032	0.131
9100	**41760**	2.903	2.1	1.362	0.2	0.016	0.087	0.859
15488	**71687**	8.268	5.737	3.782	0.444	0.025	0.241	2.641
30589	**143364**	41.503	26.081	*	2.244	0.063	1.15	12.144
65536	**311040**	312.381	172.022	*	9.244	0.153	4.022	74.99
130682	**625537**	1299.275	657.95	*	30.447	0.494	10.506	369.141
270848	**1306607**	5231.609	2973.041	*	95.653	1.625	31.397	2011.353

Figure 5.7. Computational results on GENRMF-LONG family data



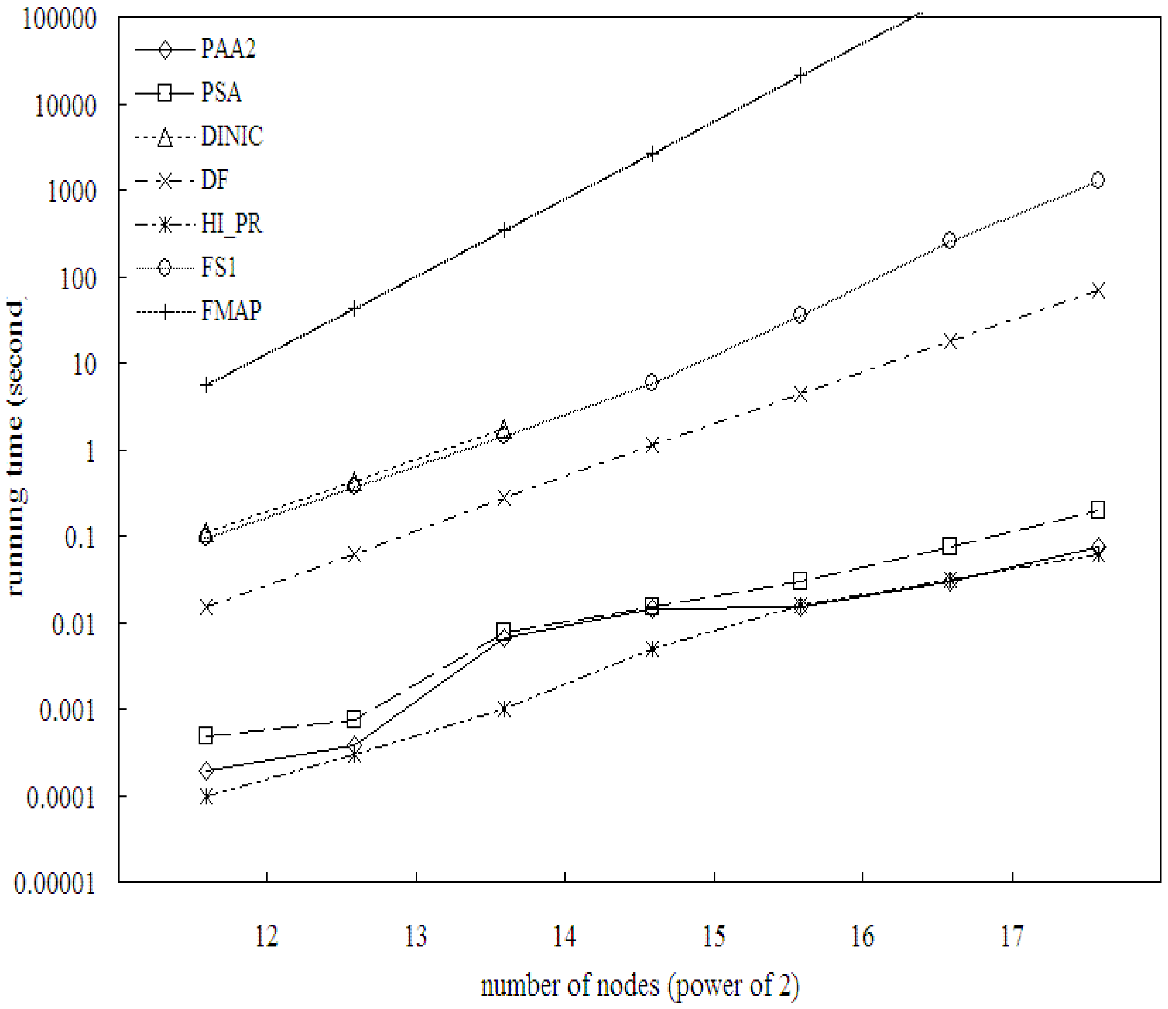
GENRMF-WIDE		Running time (second)						
n	**m**	**PAA2**	**PSA**	**DINIC**	**DF**	**HI_PR**	**FS1**	**FMAP**
3920	**18256**	0.525	1.378	0.464	0.066	0.022	0.131	0.582
8214	**38813**	2.171	5.184	1.938	0.187	0.047	0.456	3.184
16807	**80262**	8.312	22.028	8.908	0.572	0.119	2.762	19.953
32768	**157696**	29.656	82.422	*	2.459	0.319	16.141	98.379
65025	**314840**	127.6	302.64	*	8.706	0.997	65.644	540.663
123210	**599289**	520.668	1086.39	*	25.175	2.397	240.575	2506.781
259308	**1267875**	2292.612	4427.694	*	103.291	7.409	970.697	15309.431

Figure 5.8. Computational results on GENRMF-WIDE family data



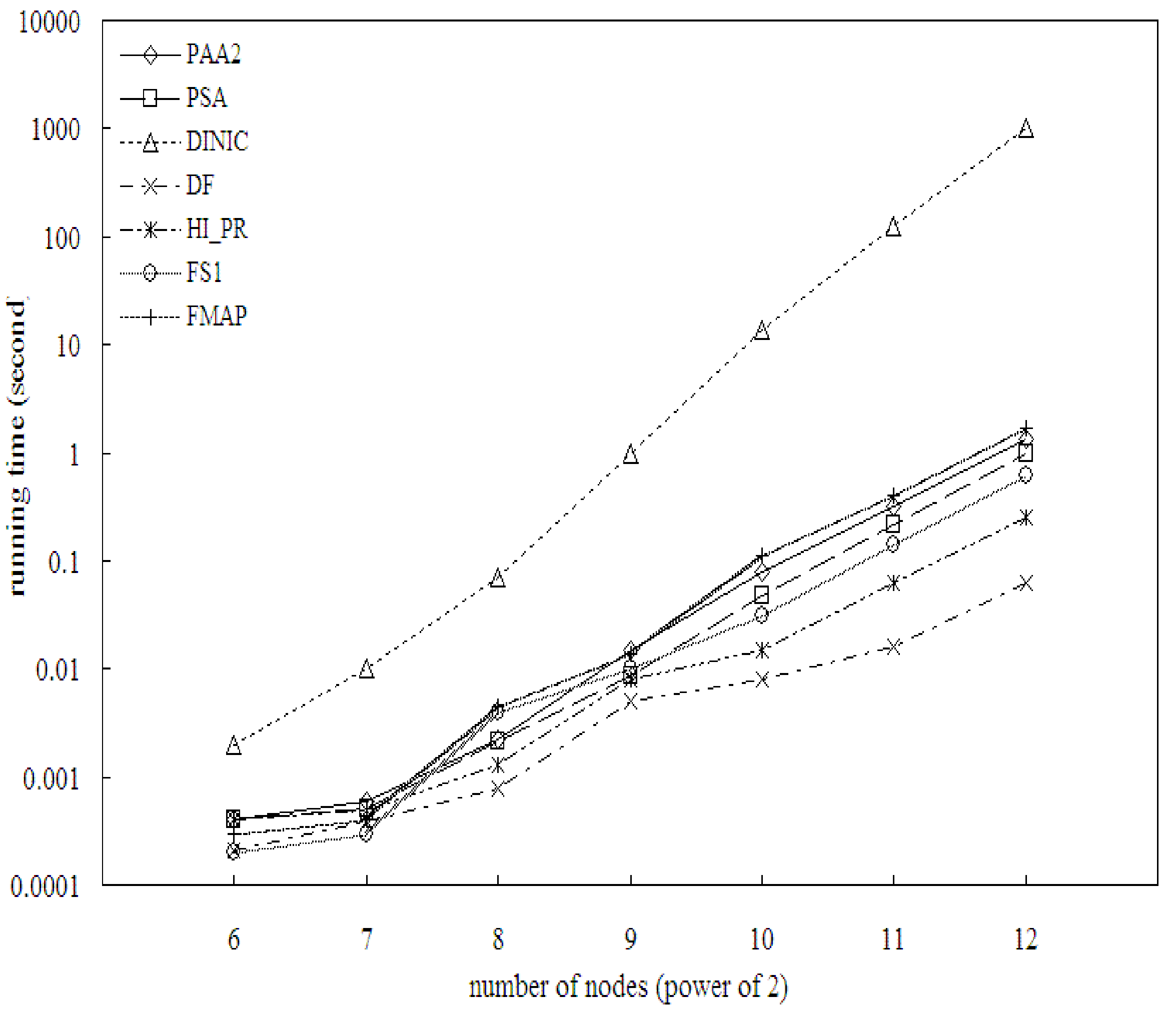
AK		Running time (second)						
n	**m**	**PAA2**	**PSA**	**DINIC**	**DF**	**HI_PR**	**FS1**	**FMAP**
4102	**6151**	0.606	0.235	5.92	0.297	0.031	0.063	0.016
8198	**12295**	2.451	0.953	44.78	1.141	0.14	0.187	0.078
16930	**24583**	9.931	3.797	346.95	4.593	0.547	0.797	0.297
32774	**49159**	56.914	16.485	*	18.562	2.14	3.453	1.172
65542	**98311**	296.872	82.938	*	80.438	8.609	16.188	4.672
131078	**196615**	1376.426	370.578	*	451.094	34.313	130.39	18.61
262150	**393223**	5338.706	1558.016	*	2223.891	138.453	616.844	74.39

Figure 5.9. Computational results on AK family data



WAS-10		Running time (second)						
n	**m**	**PAA2**	**PSA**	**DINIC**	**DF**	**HI_PR**	**FS1**	**FMAP**
3075	**4097**	0.0002	0.0005	0.11	0.015	0.0001	0.094	5.516
6147	**8193**	0.0004	0.0008	0.43	0.063	0.0003	0.375	42.922
12291	**16385**	0.0067	0.0076	1.71	0.282	0.001	1.453	339
24579	**32769**	0.014	0.015	*	1.109	0.005	5.843	2694.25
49155	**65537**	0.015	0.029	*	4.422	0.016	35.218	21443
98307	**131073**	0.03	0.074	*	17.734	0.031	263.297	170707.72
196611	**262145**	0.074	0.193	*	71.047	0.063	1282.812	1314449.47

Figure 5.10. Computational results on WAS-10 family data



ACU		Running time (second)						
n	**m**	**PAA2**	**PSA**	**DINIC**	**DF**	**HI_PR**	**FS1**	**FMAP**
64	**2016**	0.0004	0.0004	0.002	0.0002	0.0004	0.0002	0.0003
128	**8128**	0.0006	0.0005	0.01	0.0004	0.0005	0.0003	0.0004
256	**32640**	0.0022	0.0021	0.07	0.0008	0.0013	0.0004	0.0045
512	**130816**	0.015	0.0085	0.98	0.005	0.008	0.01	0.014
1024	**523776**	0.078	0.047	13.5	0.008	0.015	0.031	0.11
2048	**2096128**	0.313	0.218	124	0.016	0.063	0.141	0.406
4096	**8386560**	1.36	0.969	1027.06	0.063	0.25	0.61	1.688

Figure 5.11. Computational results on ACU family data

5.5. Summary

In this chapter we give the computational results of several PAA and MLSDP implementations, and compare them with other maximum flow algorithms. The introduction of speed-up techniques such as using doubly linked list, scaling phase, and sparse linear system solvers does effectively shorten the empirical running time for PAA and MLSDP. In general, MLSDP takes more time than PAA, either theoretically or empirically, due to the calculation of electron flows based on Kirchhoff's circuit laws.

In terms of empirical running time, PAA and MLSDP are not as competitive as conventional maximum flow algorithms. We suspect the cause of inefficiency may be resulted from the calculation of improving vectors δ. Although we expect the δ by PAA and MLSDP may reduce the number of flow augmentations since all the arcs in each layered network, rather than arcs on a single path, are exploited to augment flows, the reduction in iterations does not pay off the increase of running time in calculating δ. Besides, if constructed layered networks are all sparse networks, PAA may not perform efficiently even if PAA can saturate a node instead of an arc. As a result, the constants associated with $O(nm)$ and $O(m^5)$ to saturate a layered network respectively by PAA and by MLSDP may be larger than which by conventional augmenting-path based algorithms. However, for some networks of special topologies or configurations (e.g. WAS-10, AK, or ACU), PAA may have good performance since its theoretical time bound can reduce to $O(m)$ time.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

This chapter concludes the thesis by emphasizing our contributions in Section 6.1, and then suggesting some potential directions for future research to improve the efficiency for PAA and MLSDP algorithms.

6.1. Summary and Contributions

This thesis focuses on the issues in solving the maximum flow problem. The maximum flow problem has been studied for over five decades and most maximum flow algorithms can be classified into two major categories: augmenting path type and preflow-push type algorithms. Previous max-flow algorithms augment flows via only one path or one arc at each iteration. In this thesis, we first raise the following two questions: (1) why not augment flows via more than one path or one arc at the same time? and (2) how do we augment more flows at each iteration efficiently without violating the node flow balance and arc flow capacity constraints? To answer these questions, we propose two new max-flow algorithms, PAA and FSA, which identify admissible subnetworks that usually contain more admissible paths and arcs to ship flows. In particular, an augmenting-flow vector $\delta = [\delta_{ij}]$ for each admissible arc (i, j) can be efficiently calculated to resolve the problem of node flow balance. Then we can ship a total flow of θ units from the source, via all admissible arcs where each arc (i, j) contributes $\theta\delta_{ij}$ units, without violating the arc flow capacity constraints. Our algorithms are carefully designed to have good theoretical polynomial-time complexities, based on the Dinic's layered network framework [**18**]. Furthermore, we integrate the least squares dual-primal method (LSDP) with similar layered network framework to have a polynomial-time algorithm MLSDP, which resolve the complexity issue raised by Wang et al. [**51**]. Since LSDP needs to solve an NNLS subproblem to find a nondegenerate improving direction to improve flow, this may take a lot of time and cause LSDP algorithm uncompetitive. On the other hand, other maximum flow algorithms including network simplex methods can find a nondegenerate improving direction very efficiently. Therefore, we think the LSDP algorithm is not an appropriate method for the maximum flow problem. Comprehensive computational experiments have been conducted in Chapter 5, and the results provide more insights for different max-flow algorithms. Although in general our proposed algorithms are computationally inferior to the state-of-the-art max-flow algorithms, we hope some of our proposed new ideas and techniques could intrigue more researchers to continue their investigation.

To be more specific, we give the following summary on our findings and contributions:

(1) We are the first to suggest that the amount of flows to be shipped for a node could depend on the "room" of each admissible outgoing arcs. The intuition of PAA is to utilize the arc capacity proportionally, so that an arc that is more spacious can ship more flows, which in turn may reduce the number of iterations and hopefully to achieve the maximum flow in shorter time.

(2) PAA has a good theoretical complexity, $O(n^2m)$, which is the same as Dinic's algorithm [**18**]. Therefore, PAA is theoretically efficient.

(3) We are the first to suggest that the amount of flows to be shipped for a node could be split fairly to each of its admissible outgoing arcs. The intuition of FSA is to calculate another augmenting-flow vector $\delta = [\delta_{ij}]$ for each admissible arc (i, j) more efficiently than PAA, since it saves some overhead in calculating the flow ratios.

(4) Although the δ calculated by FSA may help to saturate at least one admissible arc (unlike PAA that saturates at least one node), calculating such a δ takes less computational time since it only depends on outgoing degrees. FSA has a

good theoretical complexity, $O(nm^2)$, which is the same as the algorithm by Edmonds and Karp [**20**].

(5) We propose the first polynomial-time LSDP algorithm, MLSDP, on solving maximum flow in $O(nm^5)$ time, which closes the open problem raised by Wang et al. [**51**].

(6) We have shown that any fractional maximum flows calculated by PAA, FSA, or MLSDP can be converted to integral maximum flows in $O(m^2)$ time, based on ideas of flow decomposition.

(7) We have conducted comprehensive computational experiments on comparing our algorithms with several popular and state-of-the-art max-flow algorithms that includes algorithms based on shortest augmenting paths (DINIC and DF, see [**18, 46, 11**]), preflow-push (HI_PR, see [**31, 35, 34, 32**]), and MA orderings (FS1 and FMAP, see [**25, 26, 41**]). Test networks of different topologies and sizes are randomly generated by four different network generators.

(8) We have proposed several implementations of PAA, including the generic one (PAA1), the one with more advanced data structures (PAA2), the one based on PAA2 but alternatively calculating the pushing δ and pulling δ (PAAFB), and the scaling version of PAA2 (PSA). The results show PSA is the most efficient implementation in practice.

(9) FSA is usually computationally inferior to PAA2, and thus inferior to PSA as well, except on AK networks where FSA performs fewer iterations than PAA2 (but still takes more time than PSA).

(10) We have also implemented two versions of MLSDP, including the generic one (MLSDP1), and the one exploiting advanced sparse matrix factorization subroutines (MLSDP2). The results show that efficient matrix operations could effectively boost the entire performance of MLSDP. However, MLSDP2 still

takes much more running time than PAA2, even if it may terminate in fewer iterations for some specific network families (e.g. AK).

(11) Comparing PAA with five other popular and state-of-the-art max-flow implementations, PAA may be suitable for networks of symmetric topology or arc capacities such as the WAS-10 or ACU networks, where PAA can terminate within very few iterations. In general, HI_PR is the most efficient implementation, while DINIC is the slowest implementation. FS1 performs better than FMAP, except on AK networks. The MA ordering based max-flow algorithms perform secondly on average, except on WAS-10 networks.

6.2. Suggestions for Future Research

In this Section, we propose some interesting directions for future research

6.2.1. Constructing Layered Networks More Efficiently

In our tests, we find the bottleneck procedure of PAA is the *constrcut-layered-network* procedure, especially when performing PAA in AK family. This is because constructing a layered network requires to scan all arcs of the original network, which may not be worthy, especially when the resultant layered networks are small or contain very few augmenting paths. Although our current implementations already exploit doubly linked lists to speed up operations on layered networks, instead of scanning through all arcs and nodes on the original network, this seems not good enough.

From the performance of the two implementations of Dinic's algorithm (i.e. DINIC and DF), we find that the distance-directed algorithm proposed by Ahuja and Orlin [**3**] (i.e. DF) performs much better than the generic Dinic's algorithm (i.e. DINIC). We suspect this is because DF avoids constructing layered networks as DINIC. In particular, by exploiting the distance labels, DF can identify the same augmenting paths as DINIC while saving much time and storage space in constructing the layered network. We suggest to investigate similar techniques for improving the efficiency of PAA.

6.2.2. Augmenting Flows Along at most k Outgoing Arcs in PAA

Since PAA will distribute proportional flows to each admissible arcs based on the relative proportions of the residual capacities between each arcs emanating from the same node in layered networks, this may result in very small proportional flows to be distributed, if the difference between residual capacities of arcs are large. In other words, the augmenting-flow vector δ may be dominated by some specific arcs. In such cases, it seems calculating δ based on all outgoing arcs may not be so time-effective, and the resultant δ may have similar effect to the δ calculated by only part of the outgoing arcs. Thus we suggest investigating the techniques that augment flows along at most k, instead of all, outgoing arcs. Such techniques may help avoid shipping small proportional flows to some nodes, but the side effect becomes to saturate an admissible arc, instead of a node, at each augmentation. In which case, the theoretical complexity becomes worse (although it is still in polynomial time), but it may help the practical efficiency.

6.2.3. Running PAA for Fair Networks

In our computational results, we suspect that PAA may perform better in "fair" networks. Here the word "fair" means the difference of residual capacity on outgoing arcs for a node is small. Intuitively, the proportional flow of arcs from the same node in such networks will be similar, which may help to avoid shipping smaller flows to some arcs that are more spacious. Moreover, for a dense fair network, the performance of PAA may become much better than augmenting path based algorithms, since PAA can saturate a node in each iteration and the sum of augmented arc flows in a bottleneck node is much more than a single bottleneck arc flow along an augmenting path.

6.2.4. Additional Information to Identify Appropriate Nodes and Arcs to Augment Flow by PAA

PAA may augment very small flow in one iteration when a large proportional flow distributing to a node but the sum of residual capacity of outgoing arcs from this node is very small. If we can perform some preprocessing to identify some nodes of large throughput capacity, then PAA can send more flow along these nodes and their associated arcs. The challenge lies in how to derive such rules effectively and efficiently.

6.2.5. Efficient Pushing and Pulling Flow Augmentations

Although we have implemented both pushing and pulling flow augmentations in Section 3.5.2, we have to use some extra storage to speed up the calculation and updates for some parameters. How to find an efficient way with less storage to calculate the improving directions of pushing and pulling augmentation to augment flow is a potential research topic.

6.2.6. Integrating PAA and Preflow Push Algorithms

Since the preflow-push algorithms are much more efficient than augmenting path algorithms, it would be an interesting research topic to integrate the idea of proportional flow augmentation when we push flows in preflow-push algorithms. To do so, one may have to resolve the problem of many excess nodes since the excess flow may be pushed along all outgoing arcs, instead of only one arc as conventional preflow-push algorithms, unless the excess flow is pushed along several selected, instead of all, outgoing arcs proportionally.

6.2.7. Path Flow Augmentations in MLSDP

In a given layered network, if there exists m' admissible arcs to sent flow, we will use Kirchhoff's laws to construct an m' linear system of equations and solve it to obtain the electron flow for each arc in $O((m')^3)$ time, which is the bottleneck operation of MLSDP. Thus any technique to calculate the electron flow more efficiently will improve the complexity for MLSDP. To this end, here we propose an idea based on the concept of flow decomposition to find path flows. In particular, we try to find $m-n+1$ shortest paths from s to t and these paths can become cycles if the artificial arc (t, s) is added. These cycles will satisfy the KVL and the sum of electron flows along each cycle is 1. Since finding a path must satisfy flow balance for each node lying on that path, the KCL is implicitly holds in these paths. Then we can construct $m - n + 1$ linear equations to solve the electron flow of each path and then obtain the electron flow of each arc. Set $m - n + 1 = q$, which may be much smaller than m', so that solving the Kirchhoff's laws takes $O(q^3)$ time. Note that it is not clear how to find these q paths.

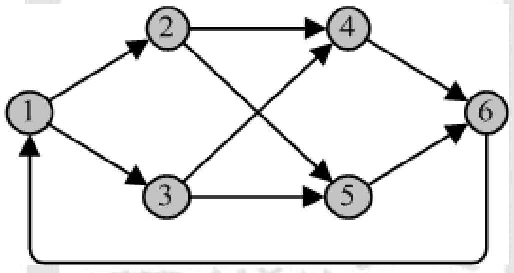


Figure 6.1. An example of using path flows to calculate electron flow of each arc

$$\begin{bmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{6.1}$$

Here we give a small example to show how to use shortest paths to calculate electron flow of each arc. Figure 6.1 is a layered network and $n = 6, m = 9$, and $A' = \{(1,2),(1,3),(2,4),(2,5),(3,4),(3,5),(4,6),(5,6)\}$. We try to find $m-n+1 = 4$ shortest paths to be KVL cycles. These four cycles are: $C_1 : 1-2-4-6-1; C_2 : 1-2-5-6-1; C_3 : 1-3-4-6-1; C_4 : 1-3-5-6-1$. Then we formulate a linear system as 6.1 to solve for augmenting-flow cycle vector δ_c along each cycle and obtain $\delta_c = \left(\frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}\right)$. Based on δ_c, we can obtain $\delta_{A'} = \left(\frac{2}{9}, \frac{2}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{2}{9}, \frac{2}{9}, \frac{4}{9}\right)$ which is the same as solving m' linear equations by original Kirchhoff's laws. However, whether there always exist exactly $m - n + 1$ cycles or not, and how to efficiently identify these cycles are still not clearly known. We suggest this to be an interesting and potential research topic.

References

[1] Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. *Network flows: theory, algorithms, and applications*, Prentice-Hall, New Jersey, 1993

[2] Ahuja, R. K. and Orlin, J. B. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37, 748-759, 1989

[3] Ahuja, R. K. and Orlin, J. B. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38, 413-430, 1991

[4] Ahuja, R. K., Orlin, J. B. and Tarjan, R. E. Improved time bounds for the maximum flow problem. *SIAM Journal on Scientific Computing.* 18, 939-954, 1989

[5] Armstrong, R. D., Chen, W., Goldfarb, D. and Jin, Z. Strongly polynomial dual simplex methods for the maximum flow problem. *Mathematical Programming*, 80, 17-33, 1998

[6] Barnes, E., Chen, V., Gopalakrishnan, B. and Johnson, E. L. A least-squares primal-dual algorithm for solving linear programming problems. *Operations research letters*, 30, 289-294, 2002

[7] Cerulli, R., Gentili, M. and Iossa, A. Efficient preflow-push algorithms. *Computers and Operations Research*, 35, 2694-2708, 2008

[8] Chang, J. H. 2006. *Solving the network flow problem by a least-squares primal-dual method.* Unpublished master's thesis, National Chen Kung University.

[9] Cheriyan, J., Hagerup, T. and Mehlhorn, K. An $O\left(n^3/\log n\right)$-time maximum-flow algorithm. *SIAM Journal on Scientific Computing.* 25, 1144-1170, 1996

[10] Cheriyan, J. and Maheshwari, S. N. Analysis of Preflow Push Algorithms for Maximum Network Flow. *SIAM Journal on Computing*, 6, 1057-1086, 1989

[11] Cherkassky, B.V. and Goldberg A.V. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19, 390-410, 1997

[12] Cheung, T. Y. Computational Comparison of Eight Methods for the Maximum Network Flow Problem. *ACM Transactions on Mathematical Software*, 6, 1-16, 1980

[13] Davis, T. A. and Duff, I. S. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18, 140-158, 1997

[14] Davis, T. A. and Duff, I. S. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25, 1-19,1999

[15] Davis, T. A. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30, 165-195, 2004

[16] Davis, T. A. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30, 196-199, 2004

[17] Davis, T. A. unsymmetric multifrontal sparse LU factorization package. available: http://www.cise.ufl.edu/research/sparse/umfpack

[18] Dinic, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11, 1277-1280, 1970

[19] Dinitz, Y. Dinitz' Algorithm: The Original Version and Even's Version. *Lecture Notes in Computer Science*, 3895, 218-240, 2006

[20] Edmonds, J. and Karp, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association for Computing Machinery*, 19, 248-264, 1972

[21] Even, S. *Graph algorithms*. Rockville, M.D.: Computer Science Press, 1979

[22] Ford, L. R. and Fulkerson, D. R. Maximal flow through a network. *Canadian Journal of Mathematics*, 8, 399-404, 1956

[23] Ford, L. R. and Fulkerson, D. R. *Flows in networks*. Princeton, N.J.: Princeton University Press, 1962

[24] Fulkerson, D. R. and Dantzig, G. B. Computation of maximum flow in networks. *Naval Research Logistics Quarterly*, 2, 277-283, 1955

[25] Fujishige, S. A maximum flow algorithm using MA ordering. *Operations Research Letters*, 31, 176-178, 2003

[26] Fujishige, S. and Isotani, S. New maximum flow algorithms by MA orderings and scaling, *Journal of the Operations Research*, 46, 243-250, 2003

[27] Gabow, H. N. Scaling Algorithms for Network Problems. *Journal of Computer and System Sciences*, 31, 148-168, 1985

[28] Galil, Z. An $O\left(V^{5/3}E^{2/3}\right)$ algorithm for the maximal flow problem, *Acta Informatica*, 14, 221-242, 1980

[29] Galil, Z. and Naamad, A. An $O(EV \log^2 V)$ Algorithm for the Maximal Flow Problem, *Journal of computer and system sciences*, 21, 203-217, 1980

[30] Gopalakrishnan, B., Barnes, E., Johnson, E. and Sokol, J. A least-squares network flow algorithm. Tech. Rep., Georgia Institute of Technology, 2004

[31] Goldberg, A. V. 1985. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., Cambridge, MA

[32] Goldberg, A.V. Network Optimization Library. available: http://www.avglab.com/andrew/soft.html

[33] Goldberg, A.V. Synthetic Maximum Flow Families. available: http://www.avglab.com/andrew/CATS/maxflow_synthetic.html

[34] Goldberg, A. V., Grigoriadis, M. D. and Tarjan, R. E. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Mathematical Programming*, 50, 277-290, 1991

[35] Goldberg, A. V. and Tarjan, R. E. A new approach to the maximum-flow problem. *Proceedings of the 18th Association for Computing Machinery Symposium on the Theory of Computing,* 136-146, 1986. Full paper in *Journal of the Association for Computing Machinery*, 35, 921-940, 1988

[36] Goldfarb, D. and Grigoriadis, M. D. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13, 83-123, 1988

[37] Goldfarb, D. and Hao, J. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Mathematical Programming*, 47, 353-365, 1990

[38] Hu, T. C. Minimum-cost flows in convex-cost networks. *Naval Research Logistics Quarterly*, 13, 1-9, 1966

[39] Johnson, D.S. and McGeoch, C.C. *Network Flows and Matching: First DIMACS Implementation Challenge*, American Mathematical Society, Providence, RI, 1993

[40] Karzanov, A. V. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15, 434-437, 1974

[41] Matsuoka, Y. and Fujishige, S. Practical efficiency of maximum flow algorithms using MA ordering and preflows. *Journal of the Operations Research Society of Japan*, 48, 297-307, 2005

[42] Malhotra, V., Kumar, M. P. and Maheshwari, S. N. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7, 277-278, 1978

[43] Nagamochi, H. and Ibaraki, T. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5, 54-66, 1992

[44] Nagamochi, H. and Ibaraki, T. Graph connectivity and its augmentation: applications of MA orderings. *Discrete Applied Mathematics*, 123, 447-472, 2002

[45] Sedeño-Noda, A. and González-Martín, C. A $O(nmlog(U/n))$ Time Maximum Flow Algorithm. *Naval Research Logistics*, 47, 511-520, 2000

[46] Setubal, J. S. *Implementations and variations of a maximum-flow algorithm.* Doctoral Thesis. UMI Order Number: UMI Order No. GAX93-12742., University of Washington, 1992

[47] Shioura, A. The MA-ordering max-flow algorithm is not strongly polynomial for directed networks. *Operations Research Letters*, 32, 31-35, 2004

[48] Sleator, D. D. and Tarjan, R. E. A data structure for dynamic trees. *Journal of Computer and system Sciences*, 24, 362-391, 1983

[49] Tarjan, R. E. A simple version of Karzanov's blocking flow algorithm. Operations Research Letters, 2, 265–268, 1984

[50] Wang, I.-L. On solving shortest paths with a least-squares primal-dual algorithm, Asia Pacific Operations Research, 2008 (to appear)

[51] Wang, I.-L., Chang, C.H., Chen, Y.Y, and Kang, Y.C. A least-squares dual-primal algorithm for the maximum flow problem, Proceedings of the 3rd Annual Conference of the Operations Research Society at Taiwan (ORSTW), Chun-li, Taiwan, 2006.

[52] The first DIMACS international algorithm implementation challenge: Network Flows and Matching, 1990, Available: ftp://dimacs.rutgers.edu/pub/netflow/.