EFFICIENT ALGORITHMS FOR THE MINIMUM COST PERFECT MATCHING PROBLEM ON GENERAL GRAPHS

A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Alper Atamtürk
December, 1993

EFFICIENT ALGORITHMS FOR THE MINIMUM COST PERFECT MATCHING PROBLEM ON GENERAL GRAPHS

A THESIS

SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL

ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Alper Atamtürk

December, 1993

Alper Atamtürk
tarafından bağışlanmıştır

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Mustafa Akgül(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Osman Oğuz

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Barbaros Tansel

Approved for the Institute of Engineering and Sciences:

Prof. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

EFFICIENT ALGORITHMS FOR THE MINIMUM COST PERFECT MATCHING PROBLEM ON GENERAL GRAPHS

Alper Atamtürk
M.S. in Industrial Engineering
Supervisor: Assoc. Prof. Mustafa Akgül
December, 1993

The minimum cost perfect matching problem is one of the rare combinatorial optimization problems for which polynomial time algorithms exist. Matching algorithms find applications in *Postman Problem*, *Planar Multicommodity Flow Problem*, in heuristics to the well known *Traveling Salesman Problem*, *Vehicle Scheduling Problem*, *Graph Partitioning Problem*, *Set Partitioning Problem*, in *VLSI*, et cetera. In this thesis, reviewing the existing primal-dual approaches in the literature, we present two efficient algorithms for the minimum cost perfect matching problem on general graphs. In both of the algorithms, we achieved drastic reductions in the total number of time consuming operations such as scanning, updating dual variables and reduced costs. Detailed computational analysis on randomly generated graphs has shown the proposed algorithms to be several times faster than other algorithms in the literature. Hence, we conjecture that employment of the new algorithms in the solution methods of above stated important problems would speed them up significantly.

Key words: Minimum Cost Perfect Matching Problem, Primal-dual Algorithms, Blossom Algorithm, Fibonacci Heaps.

ÖZET

GENEL ÇİZGELERDE EN KÜÇÜK MALİYETLİ TAM EŞLEME PROBLEMİ İÇİN ETKİN ALGORİTMALAR

Alper Atamtürk
Endüstri Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Doç. Dr. Mustafa Akgül
Aralık, 1993

En küçük maliyetli tam eşleme problemi, çözümü için polinom zamanlı algoritmaların bulunduğu ender kombinatoryal en iyileme problemlerinden biridir. Eşleme algoritmaları *Postacı Problemi*, *Yüzeysel Çoklu Mal Akış Problemi* ile iyi bilinen *Gezgin Satıcı Problemi*, *Taşıt Çizelgeleme Problemi*, *Çizge Parçalama Problemi*, *Küme Parçalama Problemi* ve diğerleri için sezgisel yordamlarda kullanılır. Bu tez çalışmasında, literatürdeki primal-dual yaklaşımları gözden geçirdikten sonra, en küçük maliyetli tam eşleme probleminin çözümü için iki etkin algoritma sunuyoruz. Her iki algoritmada da tarama, ikil değişken ve indirgenmiş maliyet güncellemesi gibi zaman alıcı işlemlerin sayısında büyük ölçüde indirime gidilmiştir. Detaylı sayısal analizler önerilen algoritmaların rassal olarak üretilen çizgelerde literatürdeki diğer algoritmalardan birçok kat daha hızlı olduklarını göstermiştir. Sonuç olarak, yeni algoritmaların yukarıda değinilen önemli problemlerin çözüm metodlarında kullanıldığında, bunlarda da kayda değer hızlanmaların olabileceğini söyleyebiliriz.

Anahtar Kelimeler: En Küçük Maliyetli Tam Eşleme Problemi, Primal-dual Algoritmalar, Gonca Algoritması, Fibonacci Öbekleri.

To my father and mother

ACKNOWLEDGEMENT

I am indebted to Associate Professor Mustafa Akgül for his invaluable guidance, encouragement not only throughout this study but also during my undergraduate years. I thank to Associate Professor Osman Oğuz, Associate Professor Barbaros Tansel and Associate Professor Ömer Benli for careful reading of the thesis.

I wish to express my deepest gratitude to my family without whom this study would have not been possible.

I would also like to extend my sincere thanks to Levent Kandiller, for the enthusiasm he inspired on me for the last three years.

My special thanks are to Nilgün Tene for her love, moral support and encouragement.

Contents

1 INTRODUCTION **1**

1.1 Preliminaries 2

1.2 Matching Polytope 3

1.3 Combinatorial Background 7

2 LITERATURE REVIEW **12**

2.1 Edmonds' Blossom Algorithm 13

2.2 Ball and Derigs' Algorithm 21

3 MULTIPLE AUGMENTATION ALGORITHM **28**

4 SINGLE STAGE ALGORITHM **34**

5 COMPUTATIONAL STUDIES **41**

6 CONCLUSION **50**

List of Figures

1.1 A matching example 2

1.2 A fractional matching 4

1.3 A hypomatchable and nonseparable subset of N 5

1.4 Matching on P before augmentation 7

1.5 Matching on P after augmentation 8

1.6 Shrinking the odd cycle r,x,y,z,s 9

2.1 Fibonacci Heaps 15

2.2 Grow 17

2.3 Shrink 18

2.4 Expand 19

2.5 Augment 19

3.1 Marking trees, where augmenting paths are found 29

4.1 Heap Elements after an Augmentation 36

5.1 Number of SCAN Operations versus Number of Nodes 44

5.2 Number of UPDATE Operations versus Number of Nodes . . . 45

5.3 Number of FINDMIN Operations versus Number of Nodes . . . 46

5.4 Comparison of CPU times 47

5.5 Comparison of CPU times (log-log scale) 47

5.6 Augmentations per Stage versus Ratio of SCAN and UPDATE Operations performed by SA to by MA 48

List of Tables

2.1 Worst case complexity of weighted matching algorithms 13

5.1 Comparison of SCAN operation by algorithms on random graphs with varying node size, (20% edge density) 43

5.2 Comparison of UPDATE operation by algorithms on random graphs with varying node size, (20% density) 44

5.3 Comparison of FINDMIN operation by algorithms on random graphs with varying node size, (20% density) 45

5.4 Comparison of CPU times required by the algorithms on random graphs with varying node size, (20% density) 46

5.5 Comparison of the number of UPDATE operations performed by the algorithms on random graphs with varying edge density, (500 nodes) 48

5.6 Comparison of the number of UPDATE operations performed by the algorithms on random graphs with varying edge density, (500 nodes) 48

5.7 Comparison of the number of FINDMIN operations performed by the algorithms on random graphs with varying edge density, (500 nodes) 49

5.8 Comparison of CPU times required by the algorithms on random graphs with varying edge density, (500 nodes) 49

Chapter 1

INTRODUCTION

The minimum cost matching problem is one of the rare combinatorial optimization problems for which polynomial time algorithms exist. Matching algorithms find applications in *Postman Problem* [10], *Planar Multicommodity Flow Problem* [20], in heuristics to the well known *Traveling Salesman Problem* [7], *Vehicle Scheduling Problem* [4], *Graph Partitioning Problem* [6], *Set Partitioning Problem* [21], in *VLSI* [19], et cetera. In this thesis, we aimed at finding exact fast algorithms for the problem. We present two such algorithms in the following chapters.

The thesis is organized in six chapters. In the first chapter, we introduce the problem, give the definition of the matching polytope and present augmenting paths for finding matchings on graphs. Second chapter reviews two important primal-dual algorithms for the problem. In chapters three and four, we give two efficient algorithms, respectively. A series of detailed computational studies is summarized in chapter five. Finally, we conclude the presentation with chapter six.

1.1 Preliminaries

Let $G = (N, E)$ be an undirected graph, where N is the set of nodes, E is the set of edges and c be a real valued cost vector associated with the edges. We will use n to denote the number of nodes and m for the number of edges of graph G. A *matching* M on G is a subset of edges no two of which are incident to a common node. An edge in M is called a *matching edge*; conversely, every edge not in M is a *free edge*. A node is said to be a *matched node* if it is incident to a matching edge and a *free node* otherwise. The size of M is the number of edges it contains and the cost of M is the sum of its edge costs. A *perfect matching* has size $n/2$. The problem we want to solve is that of finding a minimum cost matching among the perfect matchings on G.

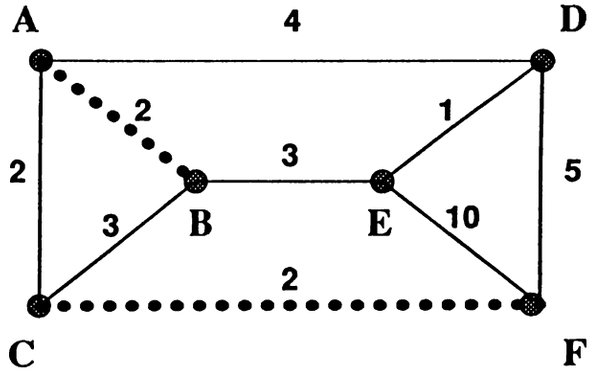

Figure 1.1: A matching example

In Figure 1.1 a matching on a small graph is illustrated. Dotted edges, (A,B) and (C,F) represent the matching edges. Rest of the edges are free. Only nodes E and D are free, others are matched nodes. Obviously, matching M is not perfect.

1.2 Matching Polytope

Let $Y \subseteq N$,

$$\Delta(Y) = \{(i,j) \in E : i \in Y, j \notin Y\}$$

$$\Gamma(Y) = \{(i,j) \in E : i \in Y, j \in Y\}$$

We may simply refer to $\Delta(Y)$ as the *cut edges* of Y. $\delta(Y) \in R^m$ is a vector associated with the edge set $\Delta(Y)$, where k^{th} component of $\delta(Y)$ is 1 if k^{th} edge is in $\Delta(Y)$, 0 otherwise. We also define $\gamma(Y) \in R^m$ similarly for the set $\Gamma(Y)$. If Y is a singleton $\{i\}$, we simply write $\delta(i)$ instead of $\delta(\{i\})$ and $\Delta(i)$ for $\Delta(\{i\})$. Furthermore let us define the *incidence vector* $\chi^F \in R^m$ of $F \subseteq E$, where k^{th} component of χ^F is 1 if k^{th} edge is in F, 0 otherwise.

Let $\mathcal{P}_{\mathcal{M}}(G)$ be the *perfect matching polytope* of G, which is the convex hull of the incidence vectors of perfect matchings in G, i.e.

$$\mathcal{P}_{\mathcal{M}}(G) \equiv \text{conv}\{\chi^M \in R^m : M \text{ is a perfect matching in } G\}$$

Using combinatorial properties of the problem, $\mathcal{P}_{\mathcal{M}}(G)$ is described with a set of linear equalities and inequalities.

First, let x be an integer vector to describe a matching, then, the *minimum cost perfect matching problem* can be formulated as:

$$\begin{array}{lll} & \min & c^T x \\ & \text{s.t.:} & \\ \text{P1)} & & x^T \delta(i) = 1 \quad , \forall\, i \in N \\ & & x \in \{0,1\}^{mx1} \end{array}$$

The equalities used in P1 are called the *assignment equalities* and when x is a binary vector, they indicate that for every node in the graph only a single edge incident to it should be in the matching. However, with this integer formulation it is very unlikely to solve the problem in polynomial time, unless P = NP. On the other hand, linear programming (LP) relaxation of P1 does not guarantee an integer solution for the problem. Mere the assignment equalities in this case may yield a fractional solution. This is easy to see in Figure 1.2.

x^T = (0,0,0,0.5,0.5,0.5,0.5,0.5,0.5) is an extreme solution of the LP relaxation of P1.

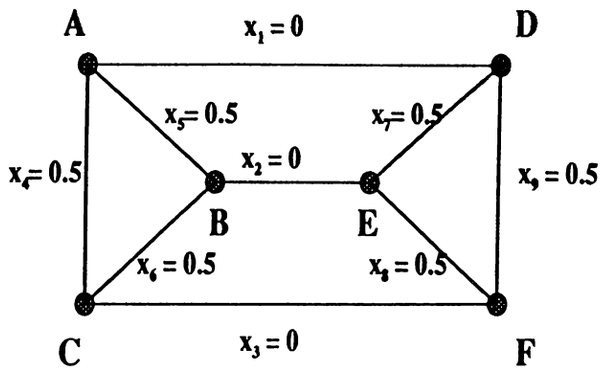

Figure 1.2: A fractional matching

Definition 1.1 *For $G = (N, E)$ and $W \subset N$ we define $G[W] \equiv (W, \Gamma(W))$ and call it subgraph induced by W.*

Definition 1.2 *A cutnode of a graph $G = (N, E)$ is a node $w \in N$ such that $G[N \setminus \{w\}]$ has more connected components than G has.*

Definition 1.3 *A graph G is called nonseparable if it does not contain any cutnode.*

Definition 1.4 *For an odd cardinality $W \subset G$ with $|W| \geq 3$, $G[W]$ is called hypomatchable if for every $w \in W$ there exists a perfect matching with respect to $G[W \setminus \{w\}]$.*

For $x \in R_+^m$ and every $W \subset N$ with $|W| \geq 3$ and odd, the following inequality is valid for $\mathcal{P}_{\mathcal{M}}$(G) :

$$x^T \gamma(W) \leq \lfloor 1/2 \mid W \mid \rfloor$$

This type of inequalities are called *blossom inequalities* and are due to Edmonds [9]. Edmonds has shown that together with the assignment inequalities, they describe $\mathcal{P}_{\mathcal{M}}$. Even though sufficient, they are far from being minimal.

Theorem 1.1 (Edmonds and Pulleyblank[10]) *Hyperplanes represented by the blossom inequalities for odd cardinality $W \subset N$ with $\mid W \mid \geq 3$ are the facets of $\mathcal{P}_{\mathcal{M}}$ if and only if $G[W]$ are hypomatchable and nonseparable (Figure 1.3).*

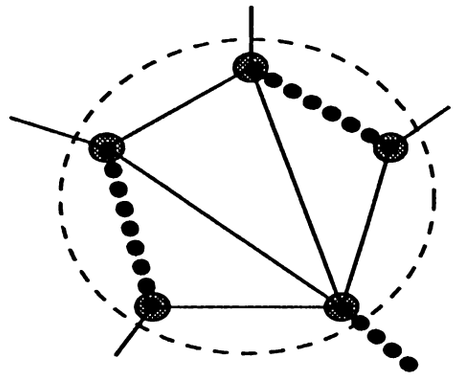

Figure 1.3: A hypomatchable and nonseparable subset of N

Let us use Q to denote the set of all such facet defining subsets of N. Then, facet defining blossom inequalities with the assignment equalities, lead to the below given *blossom characterization* of the *minimum cost perfect matching problem*,

$$
\begin{array}{lll}
 & \min & c^T x \\
 & \text{s.t.:} & \\
\text{P2)} & x^T \delta(i) = 1 & \forall\, i \in N \\
 & x^T \gamma(W) \leq \lfloor 1/2 \mid W \mid \rfloor & \forall\, W \in Q \\
 & x \geq 0 &
\end{array}
$$

Similarly, for $W \in Q$, we also have the following *cut inequalities*,

$$x^T \delta(W) \geq 1$$

to represent facets of $\mathcal{P}_{\mathcal{M}}$. Using the cut inequalities we can re-formulate the problem.

$$\begin{array}{lll} & \min & c^T x \\ & \text{s.t.:} & \\ \text{P3)} & x^T\delta(i) = 1 & \forall\, i \in N \\ & x^T\delta(W) \geq 1 & \forall\, W \in Q \\ & x \geq 0 & \end{array}$$

The above formulation is called the *cut characterization.* Actually there is a bijective linear transformation between the two characterizations.

The dual problem of P3 is,

$$\begin{array}{lll} & \max & \sum_{i\in N} y_i + \sum_{W\in Q} y_W \\ & \text{s.t.:} & \\ \text{D3)} & y_i + y_j + \sum\{y_W : (i,j) \in \Delta(W)\} \leq c_{ij} & \forall\, (i,j) \in E \\ & y_W \geq 0 & \forall\ W \in Q \end{array}$$

Let $\bar{c}_{ij}$ denote the reduced cost of an edge $(i,j) \in E$, i.e.

$$\bar{c}_{ij} = c_{ij} - y_i - y_j - \sum\{y_W : (i,j) \in \Delta(W)\}$$

Then the complementary slackness conditions of the linear program for a given M can be stated as,

$$(i,j) \in M \Rightarrow \bar{c}_{ij} = 0$$

$$y_W > 0 \Rightarrow |M \cap \Delta(W)| = 1$$

Even though linear characterization of the problem bears exponentially many inequalities in the number of nodes, importance of it comes from the fact that resulting complementary slackness conditions become very powerful tools in designing efficient algorithms for the problem. Also the combinatorial properties of the problem provides enough information so that only $O(n^2)$ of the inequalities are encountered in solving the problem.

1.3 Combinatorial Background

Definition 1.5 *A walk on G is a finite sequence of nodes and edges, where the elements of the sequence are alternatingly a node and an edge and where the starting and ending elements are nodes of G. A path on G is defined as a walk in which all nodes are distinct.*

Even though a path is a sequence, we do not distinguish the sequence and the set of elements in the sequence since it will not cause ambiguity in our context.

Definition 1.6 *An alternating path is a path where edges are alternatingly matching and free.*

Definition 1.7 *An augmenting path is an alternating path whose both ends are free nodes.*

Augmenting paths play a crucial role in matchings. An augmenting path is found by growing alternating paths; whenever one is found, the cardinality of the matching can be increased simply by swapping the free edges and the matched edges on the augmenting path. This is said to be an *augmentation*. In figure 1.4 an example augmenting path is given. The new matching on the same path after an augmentation is shown in figure 1.5. Thus if M is augmented by the augmenting path P, the new matching $\overline{M}$ is

$$\overline{M} = M \triangle P$$

where $M \triangle P$ denotes the symmetric difference operation on sets M and P.

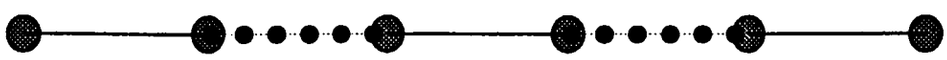

Figure 1.4: Matching on P before augmentation

Next theorem is used as a stopping criterion in finding maximum cardinality matching.

Figure 1.5: Matching on P after augmentation

Theorem 1.2 (Berge[5]) *A matching M in G contains the maximum number of edges if and only if it admits no augmenting path.*

Definition 1.8 *Let M be a matching in G and let P be an alternating path with respect to M. We define the length of P as:*

$$c(P) = \sum_{\{i,j\} \in P \setminus M} c_{ij} - \sum_{\{i,j\} \in P \cap M} c_{ij}$$

Similarly reduced cost length of P is defined as:

$$\overline{c}(P) = \sum_{\{i,j\} \in P \setminus M} \overline{c}_{ij} - \sum_{\{i,j\} \in P \cap M} \overline{c}_{ij}$$

Lemma 1.1 *Let P denote an augmenting path with respect to M in G, then cost of the augmented matching $\overline{M}$, accounts to*

$$\sum_{\{i,j\} \in \overline{M}} c_{ij} = \sum_{\{i,j\} \in M} c_{ij} + c(P)$$

Proof : This holds true, by definitions of $C(P)$ and $\overline{M}$.

Hence, cost of the matching is increased by the length of the augmenting path after an augmentation. Now suppose we have have a matching $M^\star$, with $\mid M^\star \mid = k$ and $c(M^\star)$ is minimum among all matchings of cardinality k. Then from *Lemma 1.1* a minimum length augmenting path with respect to $M^\star$ will lead to $\overline{M}^\star$ with $\mid \overline{M}^\star \mid = k+1$ and $c(\overline{M}^\star)$ minimum among all matchings of cardinality k. Lemma is also valid for the case $M = \emptyset$ for which minimum length augmenting path is simply an edge with minimum cost. So the problem boils down to successively finding minimum length (shortest) augmenting paths between pairs of free nodes until every node is matched. This solution method is called the *shortest augmenting path method.*

This method of growing alternating trees is performed with ease if the graph is bipartite. However there is a complication in non-bipartite graphs : odd cycles. In figure 1.6 a walk on nodes p, q, r, s, t does not lead to an augmenting path, however there is actually an augmenting path between p and t, which is the walk on p, q, r, x, y, z, s, t. Existence of odd cycles such as r, x, y, z, s bring about a major difficulty in the identification of augmenting paths. This difficulty is overcome by *shrinking* the odd cycle into a *pseudonode* say B. In the same figure it is clear that after the shrinking operation the augmenting path can easily be found.

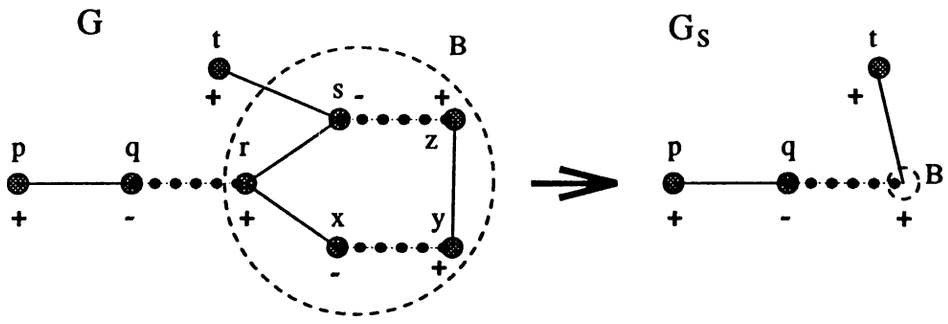

Figure 1.6: Shrinking the odd cycle r,x,y,z,s

Definition 1.9 *Given $\widehat{G} = (\widehat{N}, \widehat{E})$ let W be a subset of $\widehat{N}$ forming an odd cycle. $G_S \equiv ((\widehat{N} \setminus W) \cup B, (\widehat{E} \setminus \Gamma(W)))$, where B is a pseudonode (blossom) obtained by shrinking W, is called the surface graph.*

Note that blossoms can be nested, that is a pseudonode and some other (pseudo)nodes in G_S can further be shrunk to form another blossom. In this case, the new surface graph is obtained similarly. If we call the elements that are most recently shrunk in a nested blossom B as *maximal*, through *expansion* of B one can retrieve those maximal elements of the nested blossom B. Each one of the maximal elements may be a blossom or a node in G. Let Ω denote those maximal elements of B. If $G_S = (\widetilde{N}, \widetilde{E})$ is the surface graph before the expansion of B, the new surface graph obtained after expansion will be

$$G'_S \equiv ((\widetilde{N} \setminus B) \cup \Omega, (\widetilde{E} \cup \Gamma(\Omega))$$

Since minimum number of maximal elements in a blossom is three, maximum number of blossom formation is $n/2 - 1$. This observation will justify that a polynomial number of the blossom (or cut) inequalities in P2(P3) become tight. Shrinking and expanding blossoms will be explained in detail in the following chapter.

Lemma 1.2 *Let P_S be an augmenting path found in G_S. P_S induces an augmenting path P in G.*

Proof : The lemma directly falls from the fact that the blossoms on P_S are hypomatchable and nonseparable. Recursive expansion of the maximal elements of the blossoms on P_S leads to P.

At this point we will relate the augmenting paths to the generic primal dual algorithm. It is possible to view shortest augmenting method as in instance of the generic primal dual algorithm. Given a feasible dual solution y (possibly $y = 0$) to D3 let us call $EG(y) = (N, E(y))$ *equality subgraph*, where

$$E(y) \equiv \{e \in E : \bar{c}_e = 0\}$$

Furthermore let matching M in $EG(y)$ that satisfying the complementary slackness conditions be a *compatible pair* of y. If M is perfect, that is the primal problem also feasible, then it is optimal. However if M is not perfect, from theorem of alternatives, there exist a feasible direction d, strictly speaking a ray in the dual cone for $EG(y)$, that is the graph restricted to the edges in $E(y)$. So we can improve the dual solution in the direction of the ray. The amount of improvement is however dictated by an edge in $E \setminus E(y)$. Let θ be the maximum improvement in the direction d without violating dual feasibility for the edges in $E \setminus E(y)$, then

$$\theta = \min\{\bar{c}_e : e \in E \setminus E(y)\}$$

Now an edge e' with $\bar{c}_{e'} = \theta$ will be in the new equality subgraph $EG(y') = (N, E(y'))$, where $y' = y + \theta d$. Hence maintaining the complementary slackness conditions, at each iteration we can either find a perfect matching which is

optimal or improve the dual objective function. Equality subgraph helps us to maintain the basic invariant of the generic primal dual algorithm, that is the satisfaction of the complementary slackness conditions.

In the shortest augmenting path method, edges on the alternating paths are part of the equality subgraph. Search for an edge with minimum reduced cost to grow a path is nothing but a ratio test of reduced cost among the edges not in the equality subgraph. Growing an alternating path amounts to improving the dual solution. Hence the method is just a combinatorial instance of the generic primal dual algorithm.

Expansion of blossoms turns out to be one of the most cumbersome operations in matching algorithms for non-bipartite graphs impeding both theoretical and practical computational efficiency. In bipartite graphs, expansion and shrinking are not needed since from an combinatorial point of view, odd cycles are not encountered and from a linear programming point of view, all the variables of the dual problem are free.

Chapter 2

LITERATURE REVIEW

Edmonds[9] presented the first efficient algorithm of $O(n^4)$ for the matching problem, which can easily be implemented with $O(n^2m)$ complexity. Afterwards, there came several other primal-dual algorithms, that are improvements of Edmonds' blossom algorithm. Gabow[13] and Lawler[18] have independently shown that blossom algorithm can be implemented with $O(n^3)$ complexity. Galil, Micali and Gabow[16] gave an $O(nm \log n)$ algorithm using *splittable heaps*. Ball and Derigs[3] presented $O(n^3)$ and $O(nm \log n)$ algorithms that are modifications of the previous ones. Essentially, decrease to $O(nm \log n)$ is achieved through utilization of elegant data structures, rather than a change in the main algorithm. Gabow has recently given an $O(nm + n^2 \log n)$ algorithm for the problem, which also uses quite complicated data structures [14]. There exist other solution methods for the problem such as, primal [8], scaling [15] and cutting plane [17] algorithms; however, here we will concentrate on primal-dual algorithms. Table 1 lists the primal dual algorithms known to us for the minimum cost perfect matching problem.

In the following two sections we will describe our implementations of *Edmonds' Blossom Algorithm* and *Ball & Derigs' Algorithm*, respectively. These implementations are somewhat different than they were originally described. Firstly, our implementations allow growing of many alternating trees simultaneously, rather than one at a time. Secondly, we employ state-of-the-art data

Year	Author	Complexity
1965	*Edmonds*	$O(n^4)$
1974	*Gabow*	$O(n^3)$
1976	*Lawler*	$O(n^3)$
1983	*Ball &Derigs*	$O(n^3)$
1986	*Galil, Micali &Gabow*	$O(nm \log n)$
1990	*Gabow*	$O(nm + n^2 \log n)$

Table 2.1: Worst case complexity of weighted matching algorithms

structures such as Fibonacci Heaps which were not known at the time these algorithms were first posed.

2.1 Edmonds' Blossom Algorithm

We grow alternating trees rooted at each free node to find augmenting paths between pairs of free nodes. The collection of such alternating trees will be called the *Planted Forest*, PF. Initially, PF consists of trivial trees of single free nodes with $\oplus$ labels. As the trees are grown larger, nodes will assume labels $\oplus/\ominus$ alternatingly. After an augmentation, trees rooted at newly matched nodes are moved to the *Matched Forest*, MF, where they are labelled 0. In the case an odd cycle is encountered, it is shrunk to form a *pseudonode*. As defined in the previous chapter, this new graph induced by the shrinking operation is called the *Surface Graph*, G_S. From now on, every node in G_S will be called a *blossom*. Thus some blossoms are trivial and exist in $G = (N, E)$, as well. Let us use $b(j)$ to denote the blossom in G_S that node j is in. In referring to all the nodes $j \in N$ that are included in $b(j)$, we will say *real nodes* of $b(j)$ and denote them as *Real(b(j))*.

The Blossom Algorithm aims to find an augmenting path between a pair of free nodes at each iteration. The search for a minimum cost augmenting path is realized by growing minimum reduced cost alternating paths on G_S. Actually, all of the trees rooted at a free node are grown simultaneously, so the

augmenting path can be found between any of the trees. When an augmenting path lying on two such trees is found, that is when a minimum cost augmenting path between any two of the roots of the trees in PF is detected, the path is augmented and both of the trees on which the augmenting path lies are carried to MF. After this, a new iteration starts to search for another augmenting path on the remaining alternating trees in PF.

Within an iteration, search for an augmenting path is pursued by seven basic operations. These are *Scan*, *Update*, *Findmin*, *Grow*, *Shrink*, *Expand* and *Augment*. These operations basically have the same function in all of the algorithms that will be presented here, however there exist slight modifications from algorithm to algorithm. At this point, we present the pseudocode of the main algorithm. Then we will describe each operation in detail.

```
Edmonds' Blossom Algorithm
while PF ≠ ∅ do
begin
  Scan(k), ∀k ∈ PF;
  Update(k); ∀k ∈ PF
  (ϵ, i, j) ← Findmin();
  if (ϵ = ϵ_A) Grow(j, i);
  if (ϵ = ϵ_B)
  begin
    if (i and j belong to the same tree)
        Shrink(i, j);
    else Augment(i, j);
  end
  if (ϵ = ϵ_C) Expand(i);
end
Recover(MF);
```

We utilize three *Fibonacci Heaps* [12] to facilitate fast *Findmin* operation. Findmin outputs the minimum key (ϵ) over all the three heaps. $FibA$ is for the edges which have one end in MF and one in a $\oplus$ labelled blossom. $FibB$ is for edges with both ends in different $\oplus$ labelled blossoms and $FibC$ is for keeping the dual variables of $\ominus$ labelled non-trivial blossoms in PF. For each blossom i in MF we have an element in $FibA$ that keeps the minimum reduced edge from i to a $\oplus$ labelled blossom in PF. $FibA_i$ denotes the key in $FibA$ for

the minimum reduced cost cut edge (q, r) of i in figure 2.1, such that $b(q) = i$, $b(r) = j$ and $lb_i = 0, lb_j = \oplus$, where lb_i denotes the label of i. In order to store this edge, we have two entries associated with the heap element, *node* to keep q and nb (neighbor) to keep r. Key is the reduced cost of edge (q, r) denoted as $\bar{c}_{qr}$. $FibB_j$ is the key in $FibB$ for the minimum reduced cost edge from $\oplus$ labelled blossom j to another $\oplus$ labelled blossom. $FibB_j$ is half of $\bar{c}_{kl}$ of the co-boundary edge (k, l) of j in the same figure, such that $b(k) = j$, $b(l) = p$ and $lb_j = lb_p = \oplus$. It is possible that both j and p have distinct elements in $FibB$ for the same edge. If such an edge causes an augmentation, we simply delete both of the relevant elements from $FibB$. Finally, each non-trivial $\ominus$ labelled blossom in PF has an element in $FibC$ with a key $FibC_i$ that is equal to dual variable of i. In $FibC$, nb entries are empty since it is for dual variable information of blossoms, rather than for reduced cost of edges.

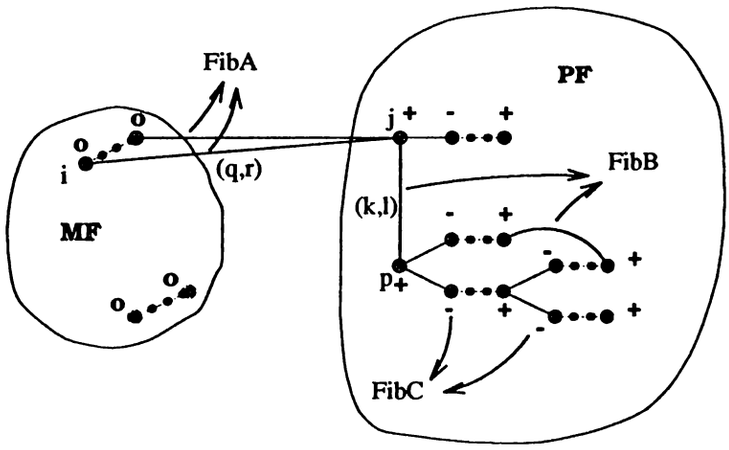

Figure 2.1: Fibonacci Heaps

In Edmonds' algorithm *Scan* is called at every iteration. By scanning each element in the cut edges of $\oplus$ labelled blossoms in PF, the minimum reduced cost edge (kept in Fib_A with the key being the reduced cost) from each 0 labelled blossom to a $\oplus$ labelled blossom and the minimum reduced cost edge (kept in Fib_B with the key being half of the reduced cost) from each $\oplus$ labelled blossom to another $\oplus$ labelled blossom are determined. Dual variables of $\ominus$ labelled blossoms are stored in Fib_C.

```
Scan(i)
begin
  if (lb_i = ⊕) then
  for ∀(k,j) ∈ Δ(i) : b(k) = i do
  begin
    if (lb_{b(j)} = 0) then
      FibA_{b(j)} ← min{FibA_{b(j)}, c̄_{kj}};
    if (lb_{b(j)} = ⊕) and (b(j) ≠ i) then
      FibB_{b(j)} ← min{FibB_{b(j)}, 0.5c̄_{kj}};
  end
  else if (lb_i = ⊖) and (i ∉ N) then /* i is a pseudonode */
    FibC_i ← y_i;
end /* Scan */


Findmin()
begin
  ε_A ← min_i{FibA_i};
  ε_B ← min_i{FibB_i};
  ε_C ← min_i{FibC_i};
   ε ← min{ε_A, ε_B, ε_C};
  q ← node achieving this minimum;
  i ← b(q);
  j ← b(nb_i);
end /* Findmin */
```

After *Scan*, *Findmin* is called to find the minimum key over all of the three heaps. ϵ is this minimum key and q is the real node achieving this minimum.

Now that ϵ is known, *Update* is called to change the dual variables of the blossoms and the reduced costs of edges in G_S. For each $\oplus$ labelled blossom i, y_i is increased by ϵ while reduced costs of the cut edges of i are decreased by ϵ. Conversely, for each $\ominus$ labelled blossom i, y_i is reduced by ϵ while reduced costs of the cut edges are increased by ϵ. Observe that, with this kind of an *Update* operation dual feasibility is maintained for every edge and reduced costs of the forest edges (edges in the equality subgraph) are kept zero. Reason for keeping only half of the reduced cost of edges between different $\oplus$ labelled blossoms in *FibB* should be clear now. If $\epsilon = \epsilon_B$ with $\epsilon_B = 0.5\bar{c}_e$, after the update $\bar{c}_e$ becomes zero since ϵ is deducted for both of the $\oplus$ labelled blossoms the end nodes of e are in. Also note that at each iteration dual objective function

```
Update(i)
begin
  if (lb_i = ⊕) then
  begin
    y_i ← y_i + ε;
    for (k, j) ∈ Δ(i) : b(k) = i do
      c̄_kj ← c̄_kj − ε;
  end
  else if (lb_i = ⊖) then
  begin
    y_i ← y_i − ε;
    for (k, j) ∈ Δ(i) : b(k) = i do
      c̄_kj ← c̄_kj + ε;
  end
end /* Update */
```

improves as much as $(\mid N \mid -2 \mid M \mid)\ \epsilon$. This claim follows from the fact that there are $(\mid N \mid -2 \mid M \mid)$ more $\oplus$ labelled blossoms than $\ominus$ labelled ones in PF.

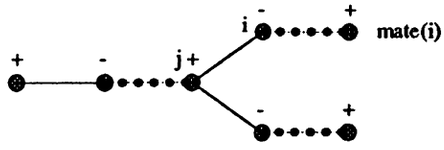

Figure 2.2: Grow

```
Grow(j, i)
begin
  parent_i ← j;
  parent_mate(i) ← i;
  lb_i ← ⊖;
  lb_mate(i) ← ⊕;
end /* Grow */
```

Grow is called if *Findmin* outputs an edge between a $\oplus$ labelled blossom j and a 0 labelled blossom i. In this case, i is made a child of j and moved to PF from MF together with its matched blossom, denoted as $mate(i)$. Please refer to fig 2.2.

Shrink is called if *Findmin* outputs an edge between two $\oplus$ labelled blossoms

i and j on the same alternating tree. This is the case, when an odd cycle is found. The odd cycle is shrunk to form a new $\oplus$ labelled blossom. *Nearest common ancestor* of i and j, denoted as $nca(i,j)$, is the first common blossom of paths from i and j to the root of the tree. In figure 2.3 let P_i be the path from i to $nca(i,j)$ and P_j be the path from j to $nca(i,j)$.

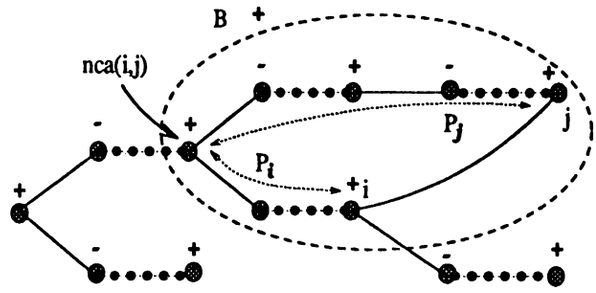

Figure 2.3: Shrink

Shrink(i,j)
begin
  Shrink $P_i \cup P_j \cup (i,j)$ into B;
  Update G_S;
  $y_B \leftarrow 0$;
  $lb_B \leftarrow \oplus$;
end /* Shrink */

Expand is called if *Findmin* outputs a non-trivial $\ominus$ labelled blossom i. In this case, i is expanded and G_S is updated to include the maximal elements in i. Please refer to figure 2.4 for this operation. Let j be the maximal element in i where the unmatched edge in $\Delta(i)$ emanates from, and let k be the maximal element in i where the matched edge in $\Delta(i)$ emanates from. Furthermore define P_O as the path between j and k with odd number of blossoms, including j and k. and P_E as the path between the remaining blossoms of i. Existence of P_O and P_E is guaranteed since there are odd number of maximal elements in i. Note that primal feasibility may be violated for the edges on the alternating path between maximal elements. This can be reconstructed simply by swapping the matching edges and free edges on one of the paths. Since swapping takes place in the equality subgraph the objective function is not affected.

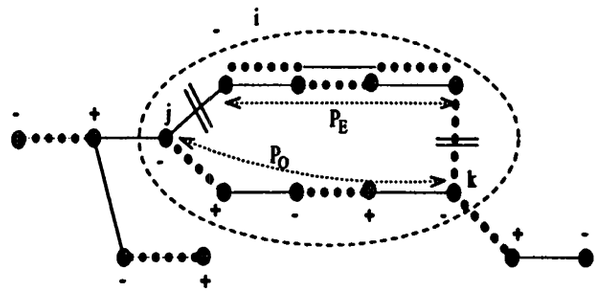

Figure 2.4: Expand

Expand(i)
begin
Unshrink i;
Update G_S;
If necessary, swap matched and unmatched edges on P_O;
If necessary, swap matched and unmatched edges on P_E;
Label the blossoms on P_O alternatingly by $\ominus$ and $\oplus$;
Move P_E to MF;
end /* Expand */

Augment is called if *Findmin* outputs an edge between two $\oplus$ labelled blossoms i and j on different alternating trees, denoted as T_i and T_j respectively in figure 2.5. This is the case, when an augmenting path between the roots of the trees is found. Now, define P_i as the path from i to the root of T_i and P_j from j to the root of T_j. Then the augmenting path is $P = P_i \cup (i,j) \cup P_j$.

After all the nodes are matched *Recover* is called to expand all the nested blossoms. One may refer to figure 2.4 again for this operation. Let j be the

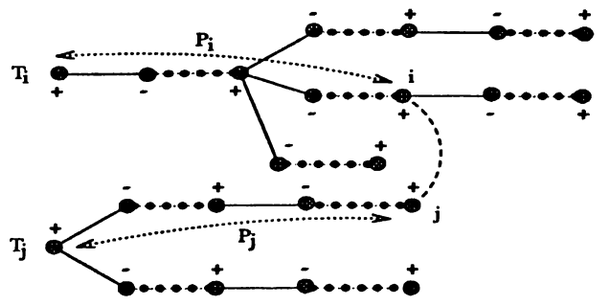

Figure 2.5: Augment

```
Augment(i, j)
  Swap matched and unmatched edges on P;
  T_i' ← T_i \ P_i;
  T_j' ← T_j \ P_j;
  Move P, T_i' and T_j' to MF;
end /* Augment */
```

blossom in i where the unmatched edge in $\Delta(i)$ emanates from, and let k be the blossom in i where the matched edge in $\Delta(i)$ emanates from. Furthermore define P_O as the path between j and k with odd number of blossoms, including j and k, and P_E as the path between the remaining blossoms of i. Finally, let L be the list of maximal elements in i. Recover is very similar to expand; it may be viewed as recursive expand operation.

```
Recover(i)
  if i ∈ N return;
  else
  begin
    Unshrink i;
    Update G_S;
    Swap matched and unmatched edges on P_O such that
      j is adjacent to a matched edge on P_O;
    Swap matched and unmatched edges on P_E such that
      blossom adjacent to j is adjacent to a matched edge on P_E;
    Recover(l) ,∀l ∈ L;
  end
end /* Recover */
```

The algorithm performs *Scan* and *Update* operations before one of the *Grow, Shrink, Augment* or *Expand* operations is called. Scanning and updating before such a call can be done in $O(m)$ time. The other operations can easily be performed in $O(n)$. Between two augmentations there are $O(n)$ iterations, so the work needed to be done between augmentations is $O(nm)$. Since there is a total of $n/2$ augmentations, we have $O(n^2m)$ complexity for the blossom algorithm.

2.2 Ball and Derigs' Algorithm

In this section, we define a new terminology called *stage*. A stage is the period between two augmentations, so it consists of a series of iterations, which ends with an *Augment* operation. Lawler showed the possibility of implementing the blossom algorithm in such a way that an edge is scanned at most two times during a stage instead twice at each iteration. The basic observation here is that as long as label of a blossom remains the same, its dual variable will continue either to increase or to decrease depending on its label. Change in the reduced cost of the cut edges of that blossom will be in the reverse direction. Within a stage, if the aggregate amount of change that should be done in the dual variable of blossoms and in the reduced costs of the cut edges can be maintained in an efficient way, it may be possible to use this information to arrive at the actual dual variables and reduced costs without the need to calculate them at each iteration. Hence, computational complexity will be reduced to $O(n^3)$ and one will achieve considerable saving in the total number of the *Scan* and *Update* operations between augmentations.

In this algorithm scanning of a cut edge is done only when blossom of one of the end nodes of the edge assumes a $\oplus$ label in a stage. Observe that once the blossom of a node assumes $\oplus$ label in PF it will remain $\oplus$ till the end of the stage. Thus an edge is scanned at most twice throughout a stage. However there exists a bit difficulty with $\ominus$ labelled blossoms, since blossom of a real node that was previously $\ominus$ labelled may assume 0 and $\ominus$ labels through successive expand and grow operations. Tackling $\ominus$ labelled blossoms will be explained further in detail.

The algorithm of Ball & Derigs[3] which is an improvement of Lawler's original algorithm performs the same *Findmin*, *Recover* and similar *Grow*, *Shrink*, *Augment*, and *Expand* operations as Edmonds'. Improvement is achieved through modification of the main algorithm and of *Scan* and *Update* operations. The algorithm has two loops, inner and outer. Each call of the outer loop is a stage and there is a total of $n/2$ stages throughout the algorithm.

```
Ball & Derigs' Algorithm
while PF ≠ ∅ do
begin
  Scan(k), ∀k ∈ PF;
  repeat
    (ϵ, i, j) ← Findmin();
    if (ϵ = ϵ_A) Grow(i, j);
    if (ϵ = ϵ_B)
    begin
      if (i and j belong to the same tree)
          Shrink(i, j);
      else Augment(i, j);
    end
    if (ϵ = ϵ_C) Expand(i);
  until (there is an augmentation)
  Update(k); ∀k ∈ PF
  Full_Update();
end
Recover(MF);
```

For delaying the *Scan* and *Update* operations, we have a variable d_i for each blossom i in PF. d_i keeps the value of ϵ when blossom i is moved to the PF during a stage. Rather than updating dual variables and reduced costs at each iteration of the algorithm, amount of update needed is stored in such a way that actual update can be performed at the end of a stage. The amount of update to be done for the dual variable of a blossom i in PF that existed in G_S at the beginning of a stage and for the reduced cost of the edges in $\Delta(i)$ is equal to $\epsilon - d_i$ at any time in the stage. This amount may be viewed as a measure of presence of blossom i in PF during the stage.

```
Grow(i, j)
begin
  parent_i ← j;
  parent_mate(i) ← i;
  lb_i ← ⊖;
  lb_mate(i) ← ⊕;
  d_i ← ϵ;
  d_mate(i) ← ϵ;
end /* Grow */
```

As mentioned before we encounter a slight difficulty in delaying reduced

cost updates of the cut edges of $\ominus$ labelled blossoms. When a $\ominus$ labelled pseudonode is expanded, reduced cost update for some cut edges will be in the reverse direction since the real nodes these edges emanate will now have a $\oplus$ labelled blossom in the new surface graph instead of a $\ominus$ labelled one. We will decrease the reduced cost of these edges, which we used to increase before the expansion. Updating the reduced cost of these edges during expand will give an $O(nm)$ complexity for a stage, since we may have to update same edges over an over again as a nested $\ominus$ labelled blossom is expanded recursively during a stage. To overcome this difficulty, relevant updates are performed on the realnodes. For this, we utilize w array of size $O(n)$. Amount of update needed to be done on the reduced costs of these edges are kept in the entries of w for the real nodes that the edges emanate from. w array will also be utilized in the *Shrink* operation. Necessary reduced cost updates for the cut edges of blossoms that are shrunk into a new pseudonode will be kept in the entries of w. Total work for updating elements of w is of $O(n^2)$ complexity throughout a stage, since both of the shrink and expand can be called $O(n)$ times per stage.

Expand(i)
begin
Unshrink i;
Update G_S;
If necessary, swap matched and unmatched edges on P_O;
If necessary, swap matched and unmatched edges on P_E;
Label the blossoms on P_O alternatingly by $\ominus$ and $\oplus$;
$d_j \leftarrow \epsilon \ \ \forall j$ on P_O;
$w_r \leftarrow w_r + (\epsilon - d_i) \ \ \forall r \in Real(i)$;
Move P_E to MF;
end /* Expand */

Recall that during an expansion matched pairs of blossoms may be carried to MF. This brings new edges between PF and MF which are not scanned before. If we were to scan these edges and put the minimum reduced cost edge from a new blossom in MF to a $\oplus$ labelled blossom, in $FibA$, complexity of a stage becomes $O(nm)$ again. This is due to the fact that we may have to scan some of these edges over and over again (at most $O(n)$ times) as nested $\ominus$ labelled blossoms are expanded recursively during a stage. To solve this problem, Lawler's idea was to keep the reduced costs of edges between $\oplus$

```
Shrink(i, j)
begin
  for ∀k on P_i ∪ P_j do
  begin
    if lb_j = ⊕ then
      y_k ← y_k + (ε − d_k);
      for ∀r ∈ Real(k)  w_r ← w_r − (ε − d_k);
    else if lb_j = ⊖ then
      y_k ← y_k − (ε + d_k);
      for ∀r ∈ Real(k)  w_r ← w_r + (ε − d_k);
  end
  for ∀k on P_i ∪ P_j do
    if lb_k = ⊖ then Scan(k);
  Shrink P_i ∪ P_j ∪ (i, j) ∈ E into B;
  Update G_S;
  y_B ← 0;
  lb_B ← ⊕;
end  /* Shrink */
```

labelled blossoms and $0/\ominus$ labelled blossoms in an array. Let p and v be two $O(n)$ arrays and j^{th} entry of p, p_j be the other end of an edge from the real node j to a $\oplus$ labelled blossom with minimum reduced cost. v_j is the reduced cost of this edge as at the beginning of a stage. When a blossom b_0 is moved to MF after an *Expand* operation, a search is done on v_j for every real node of b_0 and a new key is inserted to $FibA$ for a minimum reduced cost edge from b_0 to a $\oplus$ labelled blossom. This is just an O(n) work for each expand. Utilization of these two arrays makes it possible to scan a $\oplus$ labelled blossom only once throughout a stage, hence brings about considerable saving in the number of *Scan* operation.

The *Scan* operation in this algorithm not only computes the keys of the Fibonacci Heaps but also the entries of the v and p arrays. A blossom $b_\oplus$ is scanned only when it first time assumes a $\oplus$ label in a stage, so $d_{b_\oplus} = \epsilon$. We call the cut edges of $b_\oplus$ to 0 and $\ominus$ labelled blossoms as *candidate* edges. Let us first consider the reduced cost of an edge (j, i) between a 0 labelled blossom b_0 and $\oplus$ labelled blossom $b_\oplus$. Observe that at any time during the stage updated amount of $\bar{c}_{ji}$, which we will denote as $\tilde{c}_{ji}$ equals to $\bar{c}_{ji} - (\epsilon - d_{b_\oplus}) + w_j + w_i$, since $(\epsilon - d_{b_\oplus}) + w_j + w_i$ is the amount of reduction we have delayed to perform on $\bar{c}_{ji}$.

In this sum $(\epsilon - d_{b_\oplus})$ term reflects the delayed update after $b_\oplus$ first appeared in PF, while $w_j + w_i$ is for any update at a shrink and/or expand operation before the appearance of $b_\oplus$. Recall that w array is for complementing reduced cost updates for which (j, i) used to be a co-boundary edge of a blossom disappeared from the surface graph through a shrink or an expand operation. For the real nodes of j of b_0 a candidate edge has $\tilde{c}_{jk} = \bar{c}_{jk} + w_j + w_k$, so it is valid to compare the value $\bar{c}_{jk} + w_j$ for candidate edge (j, k) with $v_j - (\epsilon - d_{b_\oplus}) + w_{p_j}$ since v_j assumes the value of $\bar{c}_{jp_j}$ at the beginning of stage, where (j, p_j) is the edge with minimum reduced cost value from j to a $\oplus$ labelled blossom.

Now, let us look at the situation for reduced cost of an edge (j, k) between a $\ominus$ labelled blossom $b_\ominus$ and a $\oplus$ labelled blossom $b_\oplus$. Updated reduced cost $\tilde{c}_{jk}$ is equal to $\bar{c}_{jk} - (\epsilon - d_{b_\oplus}) + (\epsilon - d_{b_\ominus}) + w_j + w_k$. For real nodes of j of $b_\ominus$, a candidate edge (j, k) has $\tilde{c}_{jk}$ equal to $\bar{c}_{jk} + (\epsilon - d_{b_\ominus}) + w_j + w_k$. That is the justifying reason of comparing $v_j - (\epsilon - d_{b_\oplus}) + w_{p_j}$ with $\bar{c}_{jk} + w_k$. In order to maintain the order relationship among the elements of heaps, ϵ is added to its actual key value of a new element when it is inserted to one of them. This is necessary since keys in the heaps are not updated at each iteration.

```
Scan(i)
begin
  if (lb_i = ⊕) then
  for ∀(k, j) ∈ Δ(i) : b(k) = i do
  begin
    if (lb_{b(j)} = 0) then
      FibA_{b(j)} ← min{FibA_{b(j)}, c̄_kj + w_k + w_j + ε};
    if (lb_{b(j)} = ⊕) and (b(j) ≠ i) then
      FibB_{b(j)} ← min{FibB_{b(j)}, 0.5(c̄_kj − (ε − d_j) + w_k + w_j) + ε};
    if ( c̄_kj + w_k < v_j − (ε − d_{b(p_j)}) + w_{p_j}) then
      v_j ← c̄_kj;
      p_j ← k;
  end
  else if (lb_i = ⊖) and (i ∉ N) then
    FibC_i ← y_i + ε;
end /* Scan */
```

As stated above, amount of update to be done for dual variable of blossom i in PF and for reduced cost of $\Delta(i)$ is $\epsilon - d_i$ at any time. *Update* in this

```
Update(i)
begin
  if (lb_i = ⊕) then
  begin
    y_i ← y_i + (ε − d_i);
    for (k, j) ∈ Δ(i) : b(k) = i do
      c̄_kj ← c̄_kj − (ε − d_i);
  end
  else if (lb_i = ⊖) then
  begin
    y_i ← y_i − (ε − d_i);
    for (k, j) ∈ Δ(i) : b(k) = i do
      c̄_kj ← c̄_kj + (ε − d_i);
  end
end /* Update */
```

algorithm is modified to allow a cumulative increase or decrease depending on the label of blossom i.

At the end of a stage *Full_Update* is called to complement the updates on the reduced cost of the edges which we have delayed at *Shrink* and *Expand* operations throughout the stage. With this operation all of the delayed updates on the reduced cost of every edge are completed for the stage.

```
Full_Update()
begin
  c̄_ij ← c̄_ij + w_i + w_j  ∀(i, j) ∈ E;
end /* Full_update */
```

During a stage an edge is scanned at most two times, that is when blossom of its end nodes assume $\oplus$ label, which gives $O(m)$ for total scanning operations. All of the other operations in an iteration is bounded with $O(n)$ time . Since each of the operations can at most be called $O(n)$ times between augmentations, complexity of a stage becomes $O(n^2)$. Thus, the overall complexity of the algorithm is $O(n^3)$. Instead of v array, if we were to keep a *splittable heap* for each 0 and $\ominus$ labelled blossom, we could complete an expand operation in $O(\log n)$ time. When a $\ominus$ labelled blossom is expanded, its heap, which stores the minimum reduced cost to a $\oplus$ labelled blossom, is split to form other heaps for each of the maximal elements of that blossom that assumes a 0 or $\ominus$ label

at the new surface graph $O(\log n)$ time. However in this case scanning of an edge also takes $O(\log n)$ time rather than O(1). With the use of *splittable heaps* scanning becomes the dominating operation in a stage. With a little more care in the implementation of blossoms a stage can be done in $O(m \log n)$ time [3]. Hence the complexity of the algorithm is $O(nm \log n)$.

Chapter 3

MULTIPLE AUGMENTATION ALGORITHM

The multiple augmentation algorithm [1] is another improvement of the blossom algorithm in reducing the number of necessary scan and dual variable/reduced cost update operations. Different from the other primal-dual/successive shortest path algorithms, we may carry out more than one augmentation in a stage, which explains the term *multiple augmentation.*

In order to facilitate augmentation on more than a couple of alternating trees, we keep a *root* field for each blossom, which identifies the root of the alternating tree the blossom is on. Since at the beginning each tree consists of a single blossom, initially $root_j = j, \forall j \in G_S$. Whenever we grow a pair of matched blossoms to a tree, we make root fields of this pair equal to their parent's root field. In this way one can find the root of any blossom on an alternating tree in $O(1)$ time. When an augmenting path between a pair of alternating trees is found, in order to differentiate them from the rest of PF, we mark the roots of the trees involved, by changing the signs of their root fields to negative. We will call such trees as *marked trees.* The pair of blossoms

causing the augmentation is put into the pair list PL so that the augmenting paths may be traced back towards the roots later on at the end of the stage. In Figure 3.1 dotted edges are the matching edges, whereas the dashed ones are augmenting. The roots of trees on which augmenting paths are already found are marked, denoted with the letter M in the same figure. Inscribed in squares are those blossom pairs that are listed in PL.

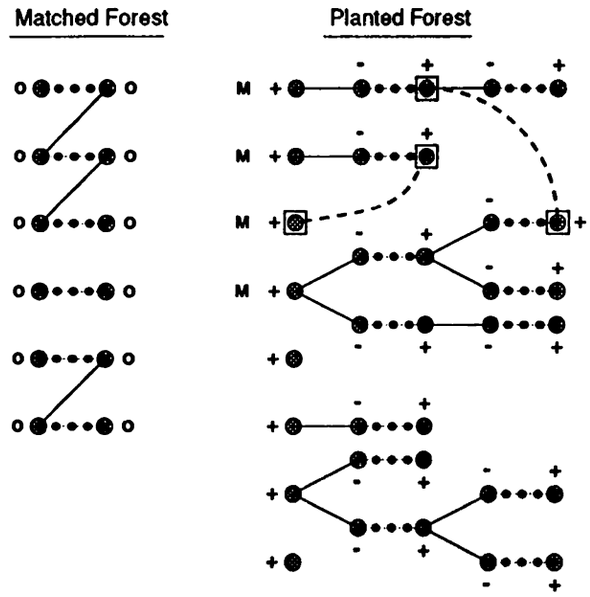

Figure 3.1: Marking trees, where augmenting paths are found

Marked trees are not carried to MF immediately. When a tree is marked, current minimum reduced costs of the edges between blossoms on the marked tree and $\oplus$ labelled blossoms of unmarked trees in PF are inserted to $FibA$. Thus for the rest of the nodes in PF, nodes in marked trees behave as if they are in MF; conversely, since at the beginning of the stage reduced cost of the edges between 0 labelled blossoms and those $\oplus$ labelled blossoms on marked trees were put in $FibA$, for the nodes in MF blossoms in marked trees behave as they are in PF. We may still grow marked trees until the end of stage. However as for the other blossoms on marked trees, reduced cost of the cut edges from those grown pairs to $\oplus$ labelled blossoms of unmarked trees are kept in $FibA$ as well. Note also that we will not perform any dual variable or reduced cost update for the cut edges of those pairs at the end of a stage. The

```
The Multiple Augmentation Algorithm :
while PF ≠ ∅ do
begin
  S ← ∅;
  Scan(k), ∀k ∈ PF;
  while (1) do
  begin
    (ϵ, i, j) ← Findmin();
    if (ϵ = ϵ_A)
      if root_i > 0  Grow(i, j);
      else  break;
    if (ϵ = ϵ_B)
    begin
      if (i and j belong to the same tree)  Shrink(i, j);
      else  Mark(i, j);
    end
    if (ϵ = ϵ_C)  Expand(i);
  end
  Update(k); ∀k ∈ PF : root_k > 0
  Full_Update();
  Multiple_augment(PL);
end
Recover(MF);
```

stage will continue until there is a grow from a marked tree to an unmarked tree in PF; that is until the first time two augmenting paths intersect. We detect this case easily, since the root of a pair to be grown from a marked tree has a negative value in its root field.

Our *Scan* operation differs from that of Ball & Derigs in allowing to scan the edges between marked trees and $\oplus$ labelled blossoms on unmarked trees. When a 0 labelled blossom i is grown to a blossom j on a marked tree with its mate, heap entry for i is deleted from $FibA$. However, we do need the reduced costs of edges between marked trees and unmarked trees in PF be kept in $FibA$. For this reason it is necessary that we scan i to put the relevant edge with minimum reduced cost to $FibA$. Note that the same blossom will not be scanned again during a stage.

In the *Mark* operation we only mark the trees where the augmenting path is found. Those trees are not moved to MF. The pair of blossoms causing

```
Scan(i)
begin
  if (lb_i = ⊕) then
  for ∀(k,j) ∈ Δ(i) : b(k) = i do
  begin
    if (lb_{b(j)} = 0) or (root_{b(j)} marked) then
      FibA_{b(j)} ← min{FibA_{b(j)}, c̄_kj + w_k + w_j + ε};
    if (lb_{b(j)} = ⊕) and (b(j) ≠ i) and (root_{b(j)} unmarked) then
      FibB_{b(j)} ← min{FibB_{b(j)}, 0.5(c̄_kj − (ε − d_{b(j)}) + w_k + w_j) + ε};
    if ( c̄_kj + w_k < v_j − (ε − d_{b(p_j)}) + w_{p_j}) then
      v_j ← c̄_kj;
      p_j ← k;
  end
  else
  if (lb_i = ⊖) and (root_i unmarked) then
    FibC_i ← y_i + ε;
  if (root_i marked) then
  for ∀(k,j) ∈ Δ(i) : b(k) = i do
  begin
    if (lb_{b(j)} = ⊕) and (root_{b(j)} unmarked) then
      FibA_i ← min{FibA_i, c̄_kj − (ε − d_{b(j)}) + w_k + w_j + ε};
    if (c̄_kj + w_j < v_k − (ε − d_{b(p_k)}) + w_{b(p_k)}) then
      v_k ← c̄_kj;
      p_k ← j;
  end
end /* Scan */
```

the augmentation is put into the pair list PL. We said that when a tree is marked, current minimum reduced costs of the edges between blossoms on the marked tree and $\oplus$ labelled blossoms of unmarked trees in PF are inserted to $FibA$. Reason for this is detecting an intersection of augmenting paths. This is the case when minimum comes from an edge between a marked tree and a $\oplus$ labelled blossom on an unmarked tree. In order to do that without scanning the blossoms on marked trees, we extend Lawler's idea for keeping reduced costs of edges between $\oplus$ labelled blossoms and $0/\ominus$ labelled blossoms in an array, say v. In our algorithm, we additionally keep the minimum reduced cost of edges between two $\oplus$ labelled blossoms in v array. Hence, we use the at hand information in v array, which is obtained through scanning of new $\oplus$ labelled blossoms in PF and avoid rescanning blossoms in marked trees during a stage. Reason for that is detecting an intersection of augmenting

paths. This is the case when minimum comes from an edge between a marked tree and a $\oplus$ labelled blossom on an unmarked tree. At any time during a stage, if $\bar{c}_{ij}$ is the reduced cost of edge (i,j) between two $\oplus$ labelled blossoms $b_{\oplus i}$ and $b_{\oplus j}$ at the beginning of the stage, updated reduced cost $\tilde{c}_{ij}$ is equal to $\bar{c}_{ij} - (\epsilon - d_{b_{\oplus i}}) - (\epsilon - d_{b_{\oplus j}}) + w_i + w_j$. Hence, for $q \in Real(k^{\oplus})$ and a $\oplus$ labelled blossom $b(p_q)^{\oplus}$, $\tilde{c}_{qp_q}$ is $v_q - (\epsilon - d_{k\oplus}) - (\epsilon - d_{b(p_q)^{\oplus}}) + w_q + w_{p_q}$, where (q,p_q) is the edge with minimum reduced cost value from q to a $\oplus$ labelled blossom because v_q assumes the value of $\bar{c}_{qp_q}$ at the beginning of the stage. With a similar argument, updated reduced cost of an edge (q,p_q), between a $\ominus$ labelled blossom $k^{\ominus}$ and a $\oplus$ labelled blossom $b(p_q)^{\oplus}$ equals to $v_q + (\epsilon - d_{k\ominus}) - (\epsilon - d_{b(p_q)^{\oplus}}) + w_q + w_{p_q}$. Below, T_i denotes the alternating tree rooted at blossom i. Even though updating dual variables of blossoms on marked trees and the reduced cost of their cut edges may be delayed to the end of the stage, we find it convenient to update them while marking.

```
Mark(i,j)
begin
  s ← root_i;
  t ← root_j;
  root_s ← -s;
  root_t ← -t;
  PL ← PL + [i,j];
  for ∀k ∈ T_s ∪ T_t do
  begin
    if (lb_k = ⊕) then
      FibA_k ← min{v_q - (ε - d_k) - (ε - d_{b(p_q)}) + w_q + w_{p_q} + ε : q ∈ Real(k)};
    else
    if (lb_k = ⊖) then
      FibA_k ← min{v_q + (ε - d_k) - (ε - d_{b(p_q)}) + w_q + w_{p_q} + ε : q ∈ Real(k)};
  end
  Update(k) ∀k ∈ T_s ∪ T_t;
end /* Mark */
```

At the end of a stage, dual variables of blossoms on unmarked trees and the reduced costs of their cut edges are updated. Note that for edges between marked and unmarked trees reduced costs are partially updated in *Mark*. Now that all the augmenting paths found throughout the stage are stored via the pair list PL, augmentation on the marked trees can be done. *Multiple_augment*

operation swaps the matched and unmatched edges on augmenting paths and moves all of the marked trees to MF at once. Obviously, efficiency of the algorithm increases as the number of trees marked per stage gets larger.

```
Multiple_augment(PL)
begin
  Augment(i,j) , ∀ [i,j] ∈ PL;
end  /* Multiple_augment */
```

The complexity of the algorithm is $O(n^3)$ when implemented as described above. There are at most $n/2$ stages ($n/2$ is achieved if only a single augmenting path is found in every stage). At each stage an edge is scanned only either when blossom of one of its end nodes is labelled $\oplus$ or when grown to a marked tree with $\ominus$ label. Since such blossoms remain in PF until the end of stage, scanning of an edge occurs at most two times in a stage. Dual variable updates and all other operations including marking are done at most in $O(n)$ time. Observe that reduced cost updates of edges are performed at the end of a stage, except for in *Mark*. Partial update on reduced costs for edges that have one end in marked trees are performed while marking, but there is no second update for those at the end of the stage. Hence overall reduced cost update in a stage takes $O(m)$. Since any of the $O(n)$ operations can be called at most $O(n)$ times, each stage takes $O(n^2)$. For $O(nm \log n)$ complexity we utilize *splittable heaps*. Although, update after expand can be done in $O(\log n)$, this time scanning of each edge takes $O(\log n)$ time rather than O(1). With a little more care in the implementation of the blossoms a stage can be done in $O(m \log n)$. Hence, we achieve an $O(nm \log n)$ complexity for the algorithm.

Chapter 4

SINGLE STAGE ALGORITHM

Here we present a single stage primal-dual algorithm [2] for the minimum cost perfect matching problem. That is, contrary to other approaches we do not stop a stage to initialize shortest paths until the algorithm finds the optimal solution. By initialization of shortest paths we mean to scan all of the blossoms in PF and building up the three *Fibonacci Heaps* that will be used throughout a stage. Dual feasibility is achieved by re-scanning some nodes, only when the need arises. Information on the need for rescanning is kept in one additional array. As a result, time consuming shortest path initializations at the beginning of each stage is totally eliminated. Elimination of initialization process at an expense of re-scanning some blossoms throughout the algorithm has experimentally proved to be very effective.

The single stage algorithms performs all operations in one stage. Here, we totally eliminate the initialization of shortest paths, that is scanning of blossoms in PF at the beginning of stages. Instead, we only scan necessary blossoms in *Pivot* operation. In the following we will describe the reasons for re-scanning when initialization of shortest paths is eliminated, and show the possibility of maintaining dual feasibility without initialization. Then we will

The Single Stage Primal-Dual Algorithm :

```
Scan(k), ∀k ∈ PF;
while PF ≠ ∅ do
begin
  (ϵ, i, j, q) ← Findmin();
  if ( !Pivot(ϵ, q) ) continue;
  if (ϵ = ϵ_A) Grow(i, j);
  if (ϵ = ϵ_B)
  begin
    if (i and j belong to the same tree) Shrink(i, j);
    else Augment_and_Update(i, j);
  end
  if (ϵ = ϵ_C) Expand(i);
end
Update(k); ∀k ∈ PF
Full_Update();
Recover(MF);
```

present an efficient method to allow necessary re-scanning and heap correction.

Recall that in the previous algorithms we arranged the Fibonacci Heaps as follows: Minimum reduced cost of edges between MF and PF were kept in $FibA$, half of the minimum reduced cost of edges between $\oplus$ labelled blossoms in PF were in $FibB$, and finally dual variables of the $\ominus$ labelled non-trivial blossoms in PF were stored in $FibC$. This was true since when starting a new stage after finding an augmenting path, the heaps were re-built from scratch each time. However, since initialization is not the case in this algorithm, $FibA$ and $FibB$ may store some irrelevant information after two trees on which an augmented path lies are moved from PF to MF. This is encountered specifically in two ways:

(a) Some nodes of $FibA$ may keep irrelevant reduced cost; i.e. reduced cost of an edge whose both ends belong to MF.

(b) One end of an edge that was once scanned and put into $FibB$ may no longer hold a $\oplus$ label; i.e. the edge has one end in PF, the other in MF.

In figure 4.1 node $b(j)$ is moved to MF after an augmentation. Case (a) refers

to the heap key for edge (i,j), $FibA_{b(i)}$, whereas case (b) refers to the heap key for edge (k,j), $FibB_{b(k)}$. We overcome these difficulties by re-scanning blossoms $b(i)$ and $b(k)$ in order to correct the key of the relevant heap node. An important point to mention here is that these blossoms are scanned only if their keys in the heaps are found to be minimum by *Findmin*.

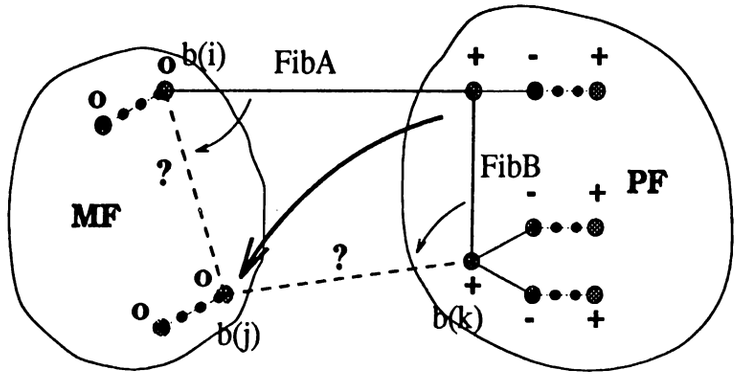

Figure 4.1: Heap Elements after an Augmentation

Lemma 4.1 *Let $FibA_{b(i)}$ be the minimum key of FibA and j the other end of edge (i,j) associated with that key, where $lb_{b(i)} = 0$, $lb_{b(j)} = \oplus$. If j was moved to MF at least once after the scanning of edge (i,j), by rescanning $b(i)$, $FibA_{b(i)}$ can be corrected.*

Proof: Let b^j be the blossom of the real node j when it is moved to PF before scanning of edge (i,j) and let $\bar{c}_{ij}$ be the reduced cost at that time. If j were to remain in PF, for some ϵ, updated reduced cost $\tilde{c}_{ij}$ would be $\bar{c}_{ij} - (\epsilon - d_{bj})$. But if j is moved before, say at $\epsilon' \leq \epsilon$, then $\bar{c}_{ij} - (\epsilon - d_{bj}) \leq \bar{c}_{ij} - (\epsilon' - d_{bj})$. Even if j is later on moved to PF with $\oplus$ or $\ominus$ label, $FibA_{b(i)}$ would be less than or equal to its correct value. Scanning operation supports the required correction,

$$FibA_{b(i)} \leftarrow min\{FibA_{b(i)}, \bar{c}_{il} - (\epsilon - d_{b(l)}) + w_i + w_l + \epsilon\};$$

where $b(l)$ is a $\oplus$ labelled blossom.

Lemma 4.2 *Let $FibB_{b(k)}$ be the minimum key of $FibB$ and j the other end of edge (k,j) associated with that key, where $lb_{b(k)} = lb_{b(j)} = \oplus$. If j has moved to MF at least once after the scanning of edge (k,j), by rescanning $b(k)$, $FibB_{b(k)}$ can be corrected.*

Proof: Let b^j and b^k be the blossoms real nodes j and k were in when they were moved to PF before the scanning of edge (j,k). Similarly, if j were to remain in PF, for some ϵ updated reduced cost $\tilde{c}_{kj}$ would be $\bar{c}_{kj} - (\epsilon - d_{b^k}) - (\epsilon - d_{b^j})$. However if j is moved to MF by augmentation before, say at $\epsilon' \leq \epsilon$, then $\bar{c}_{kj} - (\epsilon - d_{b(k)}) - (\epsilon - d_{b(j)}) \leq \bar{c}_{kj} - (\epsilon - d_{b(k)}) - (\epsilon' - d_{b(j)})$. Since the decrease in the reduced cost of edge (k,j) will not be lower than in the case when both ends having $\oplus$ label in PF, $FibB_{b(k)}$ is less than or equal to its correct value. The scanning operation we propose allows us to accommodate reductions necessitated by both end blossoms of an edge. Correction will be made in the following way:

$$FibB_{b(k)} \leftarrow min\{FibB_{b(k)}, 0.5(\bar{c}_{kl} - (\epsilon - d_{b(k)}) - (\epsilon - d_{b(l)}) + w_k + w_l) + \epsilon\}$$

where $b(l)$ is a $\oplus$ labelled blossom.

Lemma 4.3 *In both cases, total number of necessary rescanning during the algorithm is not more than $n/2$ for a blossom.*

Proof: In either of the heaps correction of a key for blossom k becomes necessary only when nb_k is moved to the MF by an augmentation. Since there are $n/2$ augmentations, the result follows.

Now that we showed the heap elements can be corrected with some modification in *Scan* operation, it remains to describe how to decide the need for re-scanning. Now it is clear that when end points of an edge changes their places between forests, we need to update associated values in heaps. One naive way of doing this is checking nb's of other ends of edges that emanate from blossoms being moved from PF to MF. So, we could know whether nb of a heap entry is also moved or not. If nb for a blossom is no longer in PF, then we should re-scan that blossom to correct its key value in the respective

```
Scan(i)
begin
  if (lb_i = 0) then for ∀(k,j) ∈ Δ(i) : b_k = i do
  begin
    if (lb_{b_j} = ⊕) then
      FibA_i ← min{FibA_i, c̄_kj − (ε − d_{b(j)}) + w_k + w_j + ε};
    if (c̄_kj + w_j < v_k − (ε − d_{b(p_k)}) + w_{b(p_k)}) then
      v_k ← c̄_kj;
      p_k ← j;
      date_nb[k]← ε;
  end
  else if (lb_i = ⊕) then for ∀(k,j) ∈ Δ(i) : b(k) = i do
  begin
    if (lb_{b(j)} = 0) then
      FibA_{b(j)} ← min{FibA_{b(j)}, c̄_kj − (ε − d_i) + ε};
    if (lb_{b(j)} = ⊕) and (b(j) ≠ i) then
      FibB_{b(j)} ← min{FibB_{b(j)}, 0.5(c̄_kj − (ε − d_{b(j)}) − (ε − d_i)) + ε};
    if (c̄_kj + w_k < v_j − (ε − d_{b(p_j)}) + w_{b(p_j)}) then
      v_j ← c̄_kj;
      p_j ← k;
      date_nb[j]← ε;
  end
  else if (lb_i = ⊖) then
    FibC_j ← y_i + ε;
end /* Scan */
```

heap. However by doing that we would be performing a search on edges which we were actually avoiding. In terms of computation cost of the extra work we are to do, should be less than our gains.

For this purpose, we developed a concept of dating. Date of an operation is the most recent ϵ at that time. Note once more that heap key for node i is erroneous if $b(nb_i)$ is moved to MF at least one time. Now assume that *Findmin* outputs edge (i, nb_i) with minimum key from either $FibA$ or $FibB$. To simplify notation let j be nb_i. If it is from $FibA$, then at the time of scanning edge (i,j) $lb_{b(i)}$ was 0 and $lb_{b(j)}$ was $\oplus$. Or if it is from $FibB$, then at the time of scanning edge (i,j) both of $lb_{b(i)}$ and $lb_{b(j)}$ were $\oplus$. Within the curse of the algorithm $b(j)$ may be moved to MF at least once. The question is how to conclude that move to MF has been the case. At the time *Findmin* outputs edge (i,j), if $lb_{b(j)}$ is not $\oplus$, the conclusion is immediate. However after staying

some time in MF $b(j)$ may assume $\oplus$ label again. In order to overcome the difficulty in detecting whether the heap key is correct or not we use the dating concept and utilize an $O(n)$ array. This array is called as *date_nb*. In *date_nb*[i], we keep the date at the time when edge (i,j) is scanned. If $b(j)$ is is moved to MF and comes back to PF with a $\oplus$ label after putting the relevant key to a heap for edge (i,j), then $d_{b(i)}$ should be larger that *date_nb*[i] unless ϵ remains constant in the intermediate iterations. But in the latter case, key value for edge (i,j) would be the same, thus correct, even with the change of $lb_{b(j)}$. Hence dating becomes an efficient way of detecting the need for rescanning a blossom. All these are done in *pivot* operation.

```
Pivot(ε, q)
begin
  if (ε = ε_C) return 1;
  if ((lb_{b(nb_q)} ≠ +) or (date_nb[q] < d_{b(nb_q)}))
    scan(b_q);
    return 0;
end /* Pivot */
```

A final point is incorporating the blossoms that are moved to MF after an expand operation into the new algorithm. Recall that Lawler's idea was to keep the reduced cost of edges between $0/\ominus$ labelled and $\oplus$ labelled blossoms in an $O(n)$ v array with indices being the real nodes of $0/\ominus$ labelled blossoms. As it has been in the case of heap keys, entries of v array may contain irrelevant information since the other end of the edge for which the reduced cost is stored in v may have been removed from PF at least once. However this difficulty can also be overcome with the dating concept. Furthermore one can utilize the same *date_nb* array for this purpose, since indices of *date_nb* are the real nodes. All one needs to do is to update *date_nb*[i] in *Scan* if p_i is modified.

When an augmenting path is detected, *Augment_and_Update* is called. The path is immediately augmented and minimum reduced cost edges between $\oplus$ labelled blossoms and blossoms on the trees T_i and T_j, where the path is founded are put into *FibA* as we did in the multiple augmentation algorithm. Note that some of the newly inserted heap elements may be erroneous due to un-updated v array. We prefer to enter them wrong and correct them only if

```
Augment_and_Update(i, j)
  for ∀k ∈ T_i ∪ T_j do
  begin
    if (lb_k = ⊕) then
      FibA_k ← min{v_q − (ε − d_k) − (ε − d_{b(p_q)}) + w_q + w_{p_q} + ε : q ∈ Real(k)};
    else
    if (lb_k = ⊖) then
      FibA_k ← min{v_q + (ε − d_k) − (ε − d_{b(p_q)}) + w_q + w_{p_q} + ε : q ∈ Real(k)};
  end
  Update(k)  ∀k ∈ T_i ∪ T_j;
  Augment(i, j);
end  /* Augment_and_Update */
```

they appear as minimizing elements.

Now, we will show that the single stage algorithm has an $O(n^3)$ complexity. Each of the grow and shrink operations between augmentations is called $O(n)$ times and can be implemented in $O(n)$ [18]. An edge is scanned when blossom of one end node is moved to PF with a $\oplus$ label, and to MF after an augmentation. It may also be re-scanned at most once between two augmentations. Work for scanning edges is still $O(m)$ complexity per stage. As in the previous algorithms, each of the $O(n)$ operations can at most be called $O(n)$ times in a stage. With $n/2$ augmentations throughout the algorithm the complexity is $O(n^3)$. Utilization of *Splittable Heaps* reduces the work for a stage to $O(m \log n)$, thereby leads to an $O(nm \log n)$ complexity for the algorithm.

Chapter 5

COMPUTATIONAL STUDIES

Throughout the study we have aimed at reducing the total number of time consuming scanning and updating operations. Both of the algorithms we have presented carried out this motivation and in the preceding chapters we have outlined how the number of necessary computations were cut down. In this chapter we will summarize the results of our computational experiences with the proposed algorithms for the minimum cost perfect matching problem.

In order to compare the efficiency of our algorithms with the other primal-dual algorithms in the literature, we have coded forest versions of the $O(n^3)$ single augmentation algorithm of [3] (SA), Edmonds' original $O(n^2m)$ blossom [9] (E), the multiple augmentation algorithm [1] (MA) and the single stage [2] (SS) algorithms in C language. Three *Fibonacci Heaps* are used to store respective minimums. Shrinking and expanding blossoms are done explicitly with dynamic data structures. All of the algorithms use common *Grow, Shrink, Expand* and *Recover* functions. We have not included a greedy initialization procedure in any of the algorithms. The programs are complied with Gnu C compiler with -O2 option and run on a SPARC Station 2 under SunOS 4.1.3. Random input graphs are generated by the code *random.c* under pub/netflow/generators/ matching directory at dimacs.rutgers.edu. The algorithms are tested on 20 random graphs generated for several node and edge configurations; average of their CPU usage and of number of basic operation

calls are noted.

We present results for three basic operations for which number of calls vary considerably from algorithm to algorithm. These are *Scan, Update* and *Findmin* operations. Here we will present our results in two parts. In the first part, experiments run for random graphs of varying node size with fixed edge density are outlined. In the second part, we will mention the results for graphs of varying edge density with fixed node size. After presenting the savings in the number of these operations by our algorithms, we will note the consequent speed-up in the solution times.

In table 5.1 a comparison of number of the *Scan* operations the algorithms performed on random graphs with node size varying from 100 to 1000 and with 20% edge density is presented. The number of *Scan* operation calls are given in log-log scale in figure 5.1. Similarly, table 5.2 and figure 5.2 are for *Update* operation, whereas table 5.3 and figure 5.3 are for *Findmin* operation. As the number of nodes get larger, we observe a drastic decrease in the number of *Scan* and *Update* calls for both of the algorithms posed. Note that the savings are exponential in the number of nodes. In figures 5.1 and 5.2 our algorithms give flatter functions than the ones compared with. There is a slight increase in the number of *Findmin* calls which is due to erroneous heap elements. In MA augmentations per stage is more in larger graphs, hence increased savings in the number of these operations is achieved. This is easily seen in figure 5.6.

These figures provide a meaningful basis for healthy comparison of algorithmic efficiency, since they are independent of hardware, compilation and data structures used in the codes. Computational results are even more encouraging for larger graphs. Table 5.4 shows average run times of the algorithms versus number of nodes for random graphs with 20% edge density. We show CPU times required by the algorithms in the experiments graphically both in normal and in log-log scale with figures 5.4 and 5.5 respectively.

# of nodes	# of SCAN Operations				SCAN Ratio				
	E	SA	MA	SS	E/MA	SA/MA	E/SS	SA/SS	MA/SS
100	8024	3283	1291	882	6.21	2.54	9.82	4.02	1.58
200	33950	12696	3438	2141	9.87	3.69	15.85	6.59	1.78
300	72395	27924	5902	3226	12.26	4.73	22.44	9.63	2.03
400	130419	48896	8850	4713	14.73	5.52	27.67	11.58	2.09
500	213570	76696	12399	6387	17.22	6.18	33.44	13.37	2.16
600	287866	108970	14549	7335	19.78	7.49	39.25	16.40	2.18
700	388063	147446	17252	8707	22.49	8.54	44.56	18.96	2.22
800	496813	191354	19933	10016	24.92	9.60	49.60	21.37	2.23
900	651927	242900	22848	11698	28.53	10.63	55.72	23.24	2.19
1000	800464	300337	25210	13318	31.75	11.91	60.10	24.98	2.09

Table 5.1: Comparison of SCAN operation by algorithms on random graphs with varying node size, (20% edge density)

At this point we give similar comparative tables 5.5, 5.6, 5.7, 5.8 for the same operations but performed on random graphs with varying edge density and 500 nodes. Number of calls of the operations under varying edge density are not as sensitive as under varying number of nodes. However, we observe that the multiple augmentation algorithm improves as the density increases. This can be explained in the following way: as the number of edges increases possibility of finding disjoint augmenting paths in a stage gets larger. Performing more augmentations in a single stage, improves the efficiency of the algorithm.

In conclusion, experimental studies show that both the multiple augmentation algorithm and the single stage algorithm successfully reduce the number of necessary operations. Consequently, saving in the time consuming operations lead our algorithms to be several times faster than the other algorithms compared.

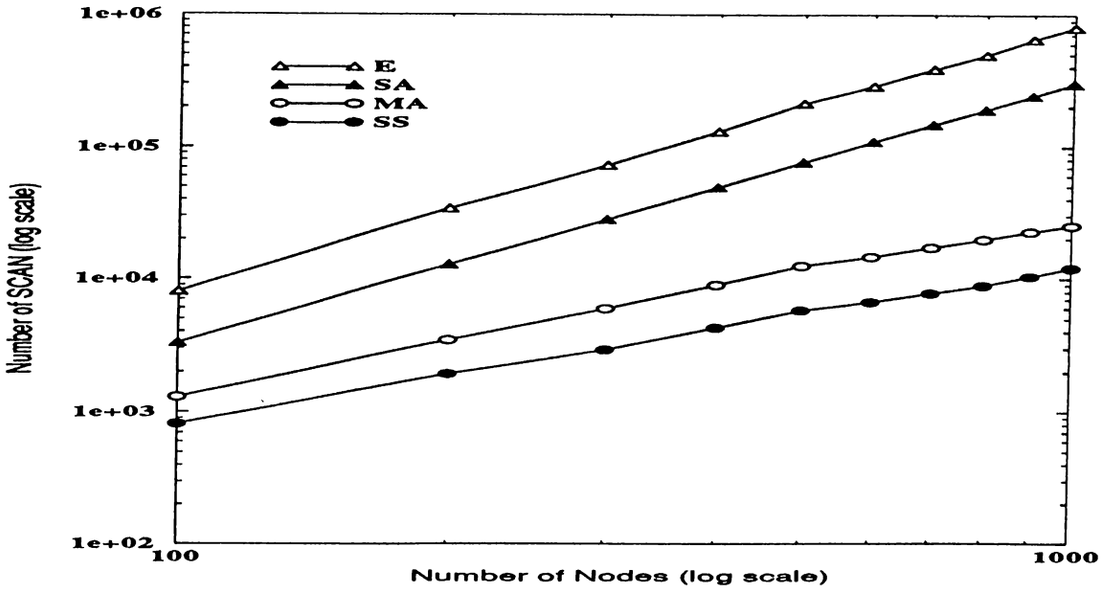

Figure 5.1: Number of SCAN Operations versus Number of Nodes

# of nodes	# of UPDATE Operations				UPDATE Ratio				
	E	SA	MA	SS	E/MA	SA/MA	E/SS	SA/SS	MA/SS
100	8024	3276	979	352	8.20	3.34	22.80	9.31	2.78
200	33950	12680	2710	819	12.52	4.68	41.57	15.48	3.31
300	72395	27894	4783	1214	15.13	5.83	59.63	22.98	3.94
400	130419	48833	7238	1768	18.02	6.74	73.82	27.62	4.09
500	213570	76604	10188	2399	20.96	7.52	89.07	31.93	4.24
600	287866	108853	11960	2732	24.07	9.10	105.44	39.84	4.38
700	388063	147342	14268	3240	27.20	10.33	118.84	45.47	4.40
800	496813	191254	16530	3675	30.05	11.57	135.26	52.04	4.49
900	651927	242754	18975	4354	34.36	12.79	149.73	55.75	4.36
1000	800464	300126	20793	4981	38.49	14.43	160.70	60.25	4.17

Table 5.2: Comparison of UPDATE operation by algorithms on random graphs with varying node size, (20% density)

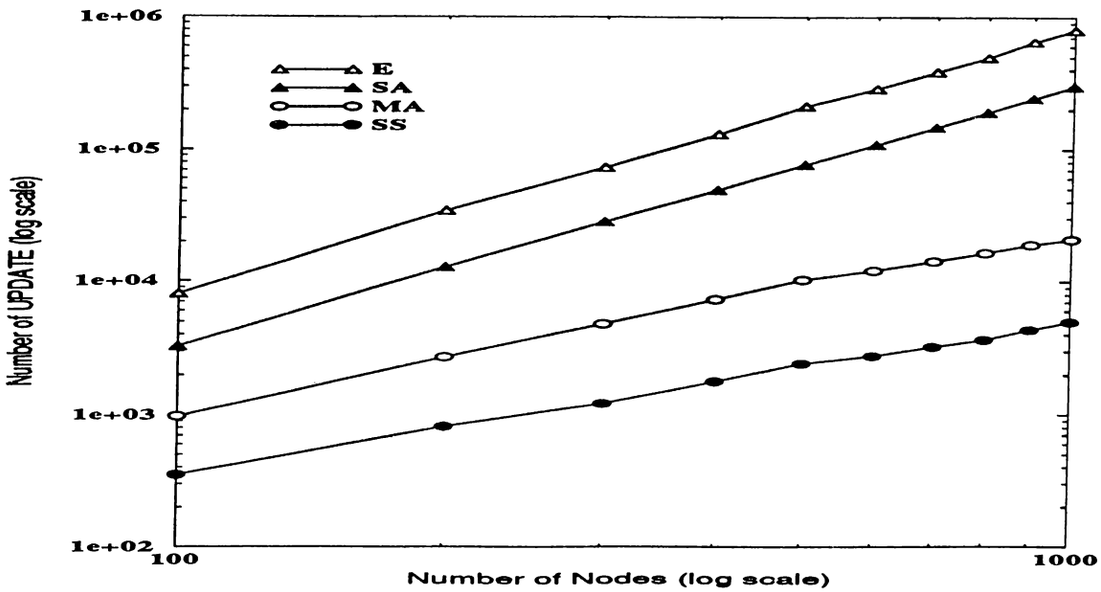

Figure 5.2: Number of UPDATE Operations versus Number of Nodes

# of	# of FINDMIN Operation				FINDMIN Ratio				
nodes	E	SA	MA	SS	E/MA	SA/MA	E/SS	SA/SS	MA/SS
100	180	179	214	417	0.84	0.84	0.43	0.43	0.51
200	417	415	466	1024	0.89	0.89	0.41	0.41	0.45
300	608	606	689	1527	0.88	0.88	0.40	0.40	0.45
400	895	896	994	2253	0.90	0.90	0.39	0.40	0.44
500	1203	1202	1314	3042	0.91	0.91	0.39	0.39	0.43
600	1385	1373	1517	3510	0.91	0.90	0.39	0.39	0.43
700	1639	1631	1774	4173	0.92	0.92	0.39	0.39	0.42
800	1865	1845	2006	4849	0.93	0.92	0.38	0.38	0.41
900	2225	2187	2380	5671	0.93	0.92	0.39	0.38	0.42
1000	2505	2486	2693	6429	0.93	0.92	0.39	0.38	0.42

Table 5.3: Comparison of FINDMIN operation by algorithms on random graphs with varying node size, (20% density)

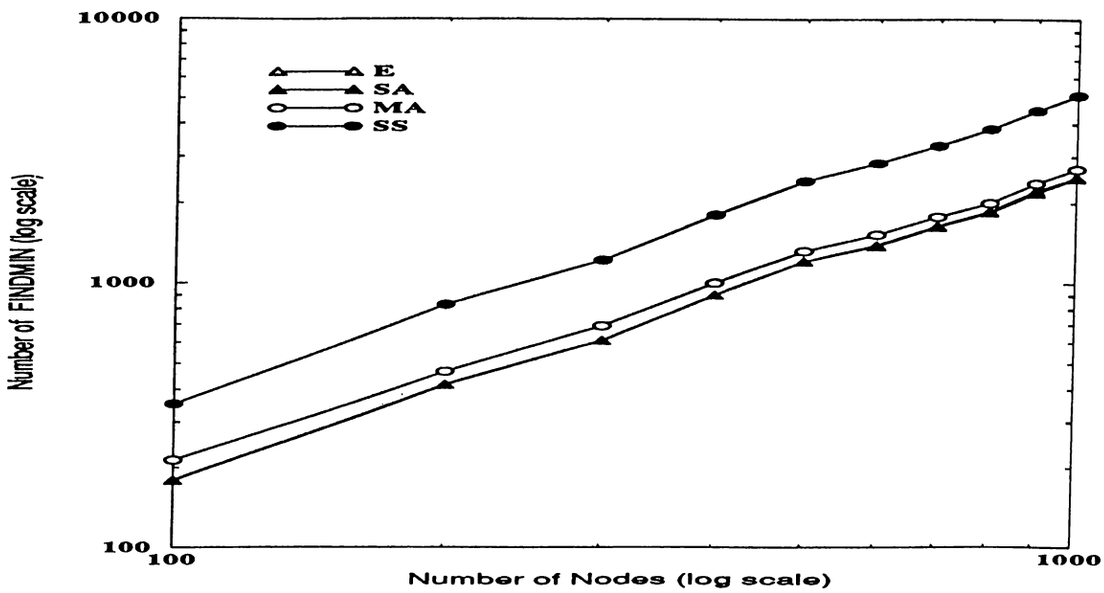

Figure 5.3: Number of FINDMIN Operations versus Number of Nodes

# of nodes	CPU Time (sec.) E	SA	MA	SS	CPU Time Ratio E/MA	SA/MA	E/SS	SA/SS	MA/SS
100	2.03	0.96	0.35	0.21	5.78	2.73	9.66	4.47	1.63
200	16.80	7.99	2.31	1.58	7.26	3.45	10.63	5.03	1.46
300	58.24	28.99	7.44	4.31	7.82	3.89	13.51	6.71	1.72
400	164.29	79.90	20.22	12.96	8.12	3.95	12.67	6.16	1.56
500	363.33	171.65	37.10	23.52	9.79	4.62	15.44	7.29	1.58
600	679.73	326.32	64.22	41.17	10.58	5.08	16.51	7.92	1.56
700	1250.53	619.93	144.52	94.49	8.65	4.28	12.75	5.93	1.53
800	2055.48	1022.98	240.08	167.17	8.56	4.26	12.30	6.11	1.43
900	3496.76	1762.03	472.84	346.85	7.39	3.72	10.08	5.09	1.36
1000	5378.31	2697.57	840.06	619.26	6.40	3.21	8.69	4.38	1.34

Table 5.4: Comparison of CPU times required by the algorithms on random graphs with varying node size, (20% density)

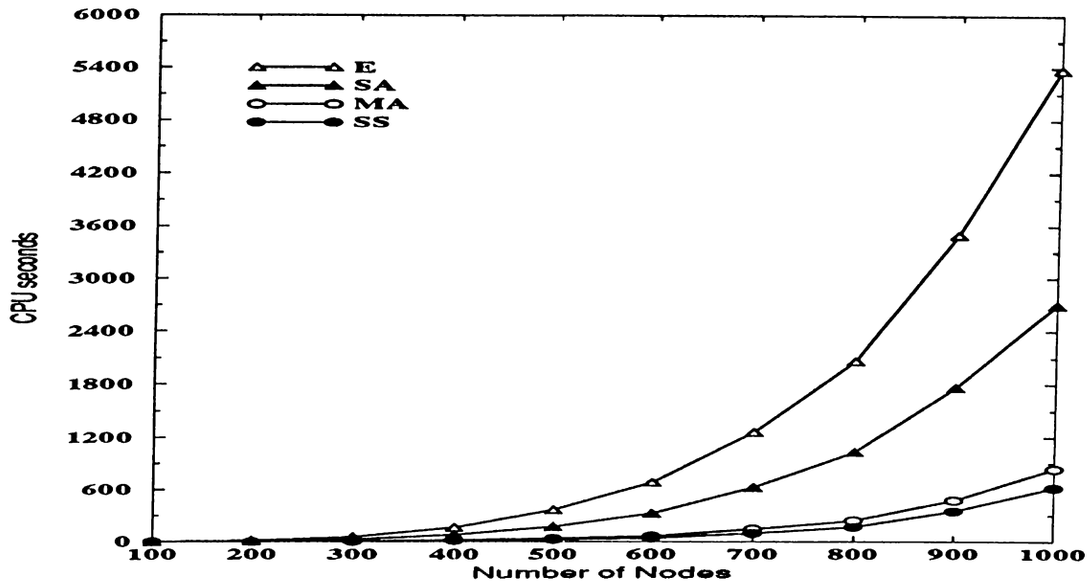

Figure 5.4: Comparison of CPU times

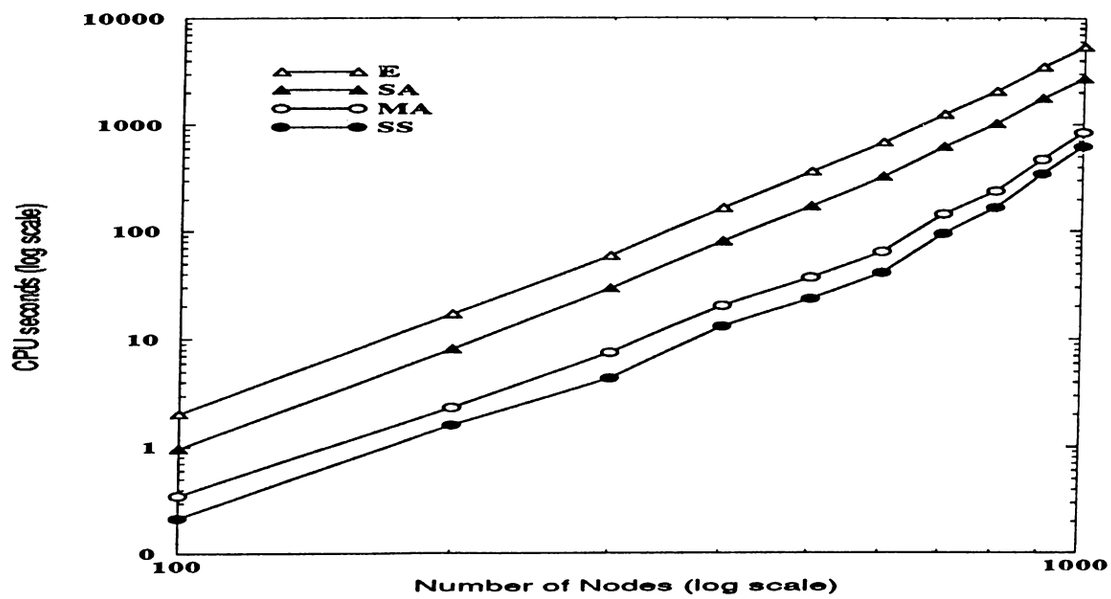

Figure 5.5: Comparison of CPU times (log-log scale)

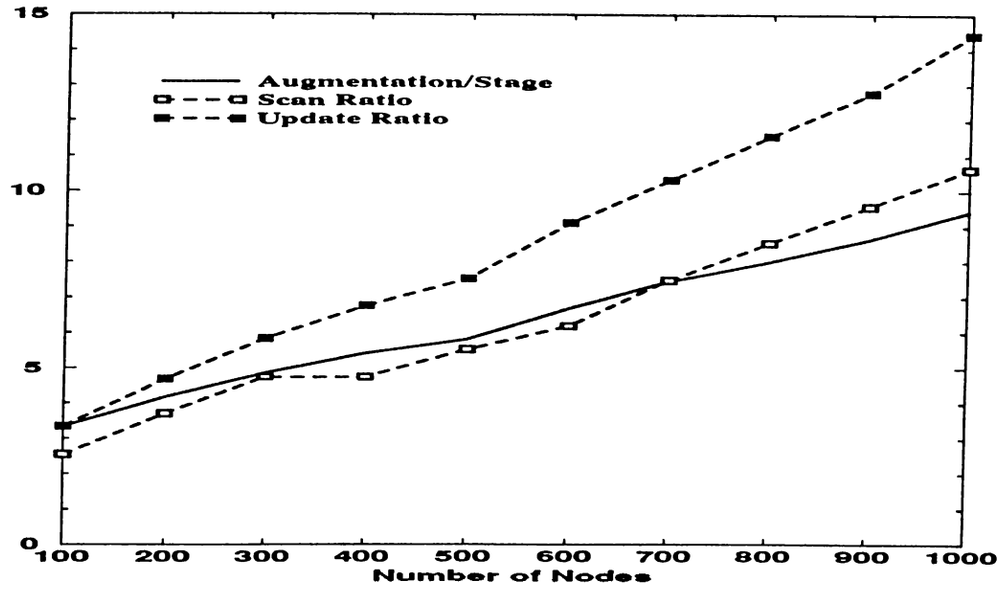

Figure 5.6: Augmentations per Stage versus Ratio of SCAN and UPDATE Operations performed by SA to by MA

edge density	# of SCAN Operation E	SA	MA	SS	FINDMIN Ratio E/MA	SA/MA	E/SS	SA/SS	MA/SS
10	207154	76741	12120	5418	17.09	6.33	38.23	14.16	2.24
20	213570	76696	12399	6387	17.22	6.18	33.44	13.37	2.16
40	194313	75569	10356	5331	18.76	7.30	36.45	14.17	1.94
80	172109	74277	8016	5411	21.47	13.73	31.81	13.73	1.48

Table 5.5: Comparison of the number of UPDATE operations performed by the algorithms on random graphs with varying edge density, (500 nodes)

edge density	# of UPDATE Operation E	SA	MA	SS	UPDATE Ratio E/MA	SA/MA	E/SS	SA/SS	MA/SS
10	207154	76672	10048	2272	20.62	7.63	91.18	76.30	4.42
20	213570	76604	10188	2399	20.96	7.52	89.07	31.93	4.24
40	194313	75475	8303	2215	23.40	9.09	87.72	34.07	3.75
80	172109	74223	6176	1958	27.87	12.02	87.90	37.91	3.15

Table 5.6: Comparison of the number of UPDATE operations performed by the algorithms on random graphs with varying edge density, (500 nodes)

edge density	# of FINDMIN Operation				FINDMIN Ratio				
	E	SA	MA	SS	E/MA	SA/MA	E/SS	SA/SS	MA/SS
10	1148	1142	1245	2286	0.92	0.92	0.50	0.50	0.54
20	1203	1202	1314	3042	0.91	0.91	0.39	0.39	0.43
40	1100	1111	1233	2267	0.89	0.90	0.48	0.49	0.54
80	982	989	1104	2080	0.89	0.89	0.47	0.47	0.53

Table 5.7: Comparison of the number of FINDMIN operations performed by the algorithms on random graphs with varying edge density, (500 nodes)

edge density	CPU Time (sec.)				CPU Time Ratio				
	E	SA	MA	SS	E/MA	SA/MA	E/SS	SA/SS	MA/SS
10	117.70	53.42	13.49	8.76	8.72	3.96	13.43	6.09	1.54
20	363.33	171.65	37.10	23.52	9.79	4.62	15.44	7.29	1.58
40	377.58	185.82	39.11	28.76	9.68	4.73	13.11	6.46	1.36
80	619.18	351.00	63.95	50.17	9.67	5.48	12.33	7.02	1.27

Table 5.8: Comparison of CPU times required by the algorithms on random graphs with varying edge density, (500 nodes)

Chapter 6

CONCLUSION

In this thesis we have proposed two efficient primal-dual algorithms for the minimum cost perfect matching problem. In both of the algorithms we allow growing of many alternating trees simultaneously. State-of-the-art data structures such as *Fibonacci Heaps* are utilized for keeping ordered sets in the implementation of the algorithms.

In the *multiple augmentation algorithm* when an augmenting path is found, we mark the roots of trees involved and continue searching disjoint augmenting paths in a stage until the first time any two of those paths intersect. Actual augmentation on all of the marked trees is realized simultaneously at the end of the stage. Since an edge is scanned at most two times per stage and dual variable/reduced cost updates are delayed further than the first detection of an augmenting path, we achieve considerable reductions in the total number of these operations. Naturally efficiency of the algorithm increases as the number of augmenting paths per stage gets larger.

The single stage algorithm is a result of the study to delay the scanning and updating operations even after the first intersection of the augmenting paths. Once the heaps are built at the beginning of the algorithm they are kept until the optimal solution is found. In other words, they are not re-constructed throughout the course of the algorithm. After an augmentation blossoms of the

trees on which the augmenting path is found are scanned to put the relevant reduced cost information of the cut edges in the heaps. Even though not initializing the heaps after an augmentation leads to some erroneous keys in the heaps, we showed that they can be corrected to maintain dual feasibility. Furthermore, we have given an efficient way of correcting the heaps elements, using a concept of *dating* the elements and re-scanning.

Both of the algorithms we have posed in the thesis have the same worst case complexity with the other primal-dual algorithms in the literature as implemented in the same way. Using simple arrays, we get an $O(n^3)$ complexity. Utilization of *splittable heaps* reduces the bound to $O(nm \log n)$.

From a practical point of view the algorithms turned out to be very efficient. Dramatic reductions in the total number of time consuming scanning and updating operations, have lead to very fast solution times on randomly generated graphs when compared to the other algorithms.

Even though computational studies show the effectiveness of our algorithms, more can be done in terms of decreasing the cpu time. Better labelling techniques existing in the literature for keeping the blossoms and use of greedy initialization before starting the algorithm could also reduce the computation time.

We have said that *the single stage algorithm* has come up as an improvement of *the multiple augmentation algorithm* in delaying operations and performing aggregate updates. Yet we conjecture that one can do even better than *the single stage algorithm.* Recall that after an augmentation we put minimum reduced cost edges between blossoms on the trees that are moved to MF and $\oplus$ labelled blossoms in $FibA$; and keep on with the stage. This was to reduce the Scan and Update operations. However, towards the end of the algorithms alternating trees as well as the augmenting paths get very large. Note that in *the single stage algorithm* even though number of Scan and Update operations is less than the same operations in *the multiple augmentation algorithm*, we face an increase in the number of Findmin operation. This is due to the erroneous heap keys. Possibility of encountering erroneous heap keys is larger

when alternating trees are bigger. Stopping the stage at some critical point and re-constructing heaps from scratch will lead to less erroneous heap keys, thus less Findmin calls, at an expense of increased scanning. While for dense graphs number of Scan operation is more critical, for sparse graphs increased Findmin operation may become an important factor affecting CPU time. Hence, there is a degree of freedom in the number of stages depending upon the edge density of the graph.

Note also that both multiple augmentation and dating/re-scanning (heap correcting) may be generalized to $b-$*matching* problems and specialized to linear assignment problem. Dating/re-scanning can also be used in other primal-dual algorithms to eliminate the initialization phase after every dual variable update.

Multiple augmentation can efficiently be parallelized in a synchronous way, that is each of the alternating trees may be grown by different CPU and a set of disjoint augmenting paths found by several CPU's may then be augmented. On the other hand, single stage algorithm can be developed into an asynchronous parallel algorithm, where CPU's do not have to wait idle until the intersection of paths. This would lead to more efficient parallel algorithm for the minimum cost perfect matching problem on non-bipartite graphs.

Finally as mentioned earlier employment of the proposed algorithms as subroutines in the solution methods for *The Postman Problem*, *Planar Multicommodity Flow Problem*, *Traveling Salesman Problem*, *Vehicle Scheduling Problem*, *Graph Partitioning Problem* and *Set Partitioning Problem* can speed them up significantly.

Bibliography

[1] M. Akgül and A. Atamtürk, *A Multiple Augmentation Algorithm for the Minimum Cost Perfect Matching Problem*, Technical Report IEOR 9308, Bilkent University, Ankara.

[2] A. Atamtürk and M. Akgül, *A Single Stage Algorithm for the Minimum Cost Perfect Matching Problem*, Technical Report IEOR 9309, Bilkent University, Ankara.

[3] M.O. Ball and U. Derigs, *An analysis of alternate strategies for implementing matching algorithms*, Networks 13 (1983) 517-549.

[4] M.O. Ball, L.D. Bodin and R. Dial, *A matching based heuristic for scheduling mass transit crews and vehicles*, Transportation Science 17 (1983) 4-31.

[5] C. Berge, *Two Theorems in Graph Theory*, Proc. Natl. Acad. Sci. USA, 43, (1957) 842-844.

[6] T.N. Bui, S. Chaudur, F.T. Leighton and M. Sipser, *Graph bisection algorithms with good average case behavior*, Combinatorica 7 (1987) 171-191.

[7] N. Christofides, *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*, Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University (1975).

[8] W.H. Cunningham and A.B. Marsh, *A primal algorithm for optimum matching*, Mathematical Programming Study 8 (1978) 50-72.

[9] J. Edmonds, *Maximum matching and a polyhedron with 0,1 vertices*, J. Res. Natl. Bur. Standards, 69B (1965) 125-130.

[10] J. Edmonds and E.L. Johnson, *Matching, Euler Tours and the Chinese Postman Problem*, Mathematical Programming 5, (1973) 88-124.

[11] J. Edmonds and W. Pulleyblank, *Facets of 1-matching polyhedra*, in Hypergraph Seminar, Lecture Notes in Mathematics No. 411 (Springer Verlag, Berlin, (1974) 214-242.

[12] M.L. Fredman and R.E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of ACM. **34** (1987), 596–615. Also in Proc. 25'th FOCS (1984) 338-346.

[13] H.N. Gabow, *Implementation of algorithms for maximum matching on non-bipartite graphs*, Ph.D. dissertation, Dept. of Computer Science, Stanford Univ., Stanford, California (1974).

[14] H.N. Gabow, *Data structures for weighted matching and nearest common ancestors with linking*, in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms. ACM, New York, (1990) 434-443.

[15] H.N. Gabow, *Faster Scaling Algorithms for General Graph-Matching Problems*, Journal of ACM 38 (1991) 815-853.

[16] Z. Galil, S. Micali and H. Gabow, *An O(EVlogV) algorithm for finding a maximal weighted matching in general graphs*, SIAM J. Comp. 15 (1986) 120-130.

[17] M. Grötschel and O. Holland, *Solving Matching Problems with Linear Programming*, Mathematical Programming 33 (1985) 243-259.

[18] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, (Holt, Rinehart and Winston, New York, 1976).

[19] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, (John Wiley & Sons, Chichester, 1990).

[20] K. Matsumoto, T. Nishizeki and N. Saito, *Planar Multicommodity Flows, Maximum Matchings and Negative Cycles*, SIAM J. Computing 15 (1986) 495-510.

[21] G. Nemhauser and G. Weber *Optimal set partitioning, matchings and Lagrangian duality*, Naval Res. Logist. Quart. 26 (1979) 553-563.