ASSIGNMENT PROBLEM AND ITS VARIATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHMET GÜLEK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING

DECEMBER 2007

Approval of the thesis

"**ASSIGNMENT PROBLEM AND ITS VARIATIONS**"

submitted by **Mehmet Gülek** in partial fullfillment of the requirements for the degree of **Master of Science in Computer Engineering** by,

Prof. Dr. Canan Özgen _____
Dean, **Graduate School of Natural and Applied Sciences**

Prof. Dr. Volkan Atalay _____
Head of Department, **Computer Engineering**

Prof. Dr. İsmail Hakkı Toroslu _____
Supervisor, **Computer Engineering, METU**

Examining Committee Members:

Prof. Dr. Faruk Polat _____
Computer Engineering, METU

Prof. Dr. İsmail Hakkı Toroslu _____
Computer Engineering, METU

Assoc. Prof. Dr. Göktürk Üçoluk _____
Computer Engineering, METU

Assoc. Prof. Dr. Halit Oğuztüzün _____
Computer Engineering, METU

Yılmaz Arslanoğlu _____
Computer Engineering, METU

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Mehmet Gülek

Signature :

ABSTRACT

ASSIGNMENT PROBLEM AND ITS VARIATIONS

Gülek, Mehmet

M.Sc., Department of Computer Engineering

Supervisor: Prof. Dr. İsmail Hakkı Toroslu

December 2007, 41 pages

We investigate the assignment problem, which is the problem of matching two sets with each other, optimizing a given function on the possible matchings. Among different definitions, a graph theoretical definition of the linear sum assignment problem is as follows: Given a weighted complete bipartite graph, find a maximum (or minimum) one-to-one matching between the two equal-size sets of the graph, where the score of a matching is the total weight of the matched edges. We investigate extensions and variations like the incremental assignment problem, maximum subset matching problem, maximum-weighted tree matching problem. We present a genetic algorithm scheme for maximum-weighted tree matching problem, and experimental results of our implementation.

Keywords: assignment, one-to-one, Kuhn-Munkres, genetic, algorithm

ÖZ

EŞLEŞTİRME PROBLEMİ VE ÇEŞİTLEMELERİ

Gülek, Mehmet

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. İsmail Hakkı Toroslu

Aralık 2007, 41 sayfa

Bu çalışmada eşleştirme problemi incelenmiştir. Eşleştirme problemi, verilen iki kümeyi, olası eşleştirmeler üzerinde tanımlanmış bir fonksiyonu en çoklayan ya da en azlayan şekilde eşleştirme problemidir. Doğrusal eşleştirme probleminin değişik tanımlarından biri şu şekildedir: Verilen, iki eşit parçalı, olası her ayrıtı içeren bir çizge için, parçalar arasındaki en az (veya en çok) puanlı birebir eşleştirmeyi bulma. Bir eşleştirmenin puanı, içerilen ayrıtların puanlarının toplamı olarak tanımlanmaktadır. Artırmalı eşleştirme problemi, alt küme eşleştirme problemi ve en çok puanlı ağaç eşleştirme problemi gibi, klasik eşleştirme probleminin değişik türleri ve uzantıları incelenmiştir. En çok puanlı ağaç eşleştirme problemi için bir genetik yöntem tanımlanmış ve yapılan kodlamadan elde edilen deneysel sonuçlar verilmiştir.

Anahtar Kelimeler: eşleştirme, birebir, Kuhn-Munkres, genetik, algoritma

ACKNOWLEDGMENTS

I would like to thank İ.H. Toroslu, Halit Oğuztüzün, and my brother Hüseyin Gülek.

To My Family

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

An assignment is a one-to-one mapping between two finite sets of equal sizes. There are different ways to express this formally. One way is to represent the mapping by a permutation. A permutation is a one-to-one matching from a set into itself. Another way is to use an adjacency matrix, where exactly one 1 occurs at every row, exactly one 1 occurs at every column, and the rest of the items are 0. The 1's correspond to matched items, and the 0's correspond to non-matched items.

A common example to explain the assignment problem is as follows: Assume you have n workers and n jobs. Any worker is able to do any job. Every worker-job pair has a score. One may think that, the score is somehow the measure of how properly that worker may perform that job. You want

1. all the jobs be performed,
2. all the workers work
3. the total score be maximized (or minimized).

This problem is called the "**Linear Sum Assignment Problem**", also known as the "**Maximum-Weighted Bipartite Matching Problem**". Surveys on Linear Assignment Problem can be found in [3] and [1].

Let $G = (U, V, E)$ be a complete bipartite graph, where $|U| = |V| = n$, and $E = UxV$. Let $U = U_1, U_2, ..., U_n$ and $V = V_1, V_2, ..., V_n$. Let W be an $n \times n$ weight matrix. W_{ij} is a real number which is the weight of the edge (U_i, V_j). Let P be a permutation from $\{1, 2, ..., n\}$ to itself. Let P also denote, at the same time, the corresponding one-to-one function from U to V (Thus $P(U_i) = V_{P(i)}$). With these notations which already ensure the constraints (1) (since P is a function) and (2) (since P is one-to-one) above, the constraint (3) can formally be expressed as below:

$\sum_{i=1}^{n} W_{i,P(i)}$ is maximized (minimized).

One can easily see that, including either maximization or minimization of the sum in the definition does not matter. They correspond to the same problem. One may alter the signs of the numbers W_{ij}, and switch from maximization problem to minimization problem, or vice versa.

One may also easily see that, a possible constraint of non-negativity of the edge weights (that is, all the edge weights must be non-negative) does not produce a different problem. This follows easily from the following lemma:

Lemma 1.1 *Using the above notations and considering the maximization of the sum, let P be a permutation corresponding to an optimal solution. Let W' be a weight matrix, generated from W by adding the constant k to the r^{th} row. Then the permutation P still specifies an optimal solution for the new problem.*

Proof By contradiction. Observe that, for any permutation P, the score of the permutation for the new problem is equal to that for the original problem, added k; since the difference of the sums is only due to the difference in the items $W_{r,P(r)}$ and $W'_{r,P(r)}$, where $W'_{r,P(r)} = W_{r,P(r)} + k$. So, occurence of a better score for the new problem simply suggests a better solution for the original problem, which is a contradiction. ∎

The similar argument is also true, which considers adding a constant to some column of the weight matrix.

CHAPTER 2

A POLYNOMIAL-TIME ALGORITHM: KUHN-MUNKRES ALGORITHM

2.1 Introduction

In this section, we describe the Kuhn-Munkres algorithm, which is a well-known algorithm to solve the Linear Sum Assignment Problem. This algorithm was described by [7] and [9]. The algorithm finds an optimal solution to the problem in only $O(n^3)$ time, where n is the number of vertices, see [8]. The method is also known as the Hungarian method. Before proceeding, we make some assumptions and definitions.

We assume that all edge weights are non-negative. $W_{i,j} \geq 0$, for $i, j = 1, 2, ..., n$.

We define a *feasible vertex labeling* as a real-valued function l defined both on U and V. This function must satisfy the constraint

$$l(U_i) + l(V_j) \geq W_{i,j}, \text{ for } i, j = 1, 2, ..., n.$$

The number $l(v)$ is called the *label* of the vertex v. For any graph, a feasible vertex labeling can be found as below:

$$l(U_i) = max\{W_{i,j} | j = 1, 2, ..., n\},\ l(V_i) = 0, \text{ for } i = 1, 2, ..., n.$$

We define the *Equality Subgraph*, G_l as the minimal subgraph of G, including all the vertices, but only the edges (U_i, V_j) such that

$$W_{i,j} = l(U_i) + l(V_j).$$

Throughout this chapter, what is meant by a *match* is a one-to-one, but probably partial function from U to V. If a match is total, then we call it a *perfect matching*.

A vertex is said to be *matched*, according to a match, if some edge in that match touches that vertex. If a vertex is not matched, then it is said to be *free*. An edge is said to *matched*, if it is included in the match. Otherwise it is said to be *free*. An *alternating path* is a path beginning at a free vertex, and alternating between free and matched edges. An *augmenting path* is an alternating path ending at a free vertex. One may observe that, if M is a match and P is an augmenting path with respect to M, then, $((M \cup P) - (M \cap P))$ is a matching of size $|M| + 1$, where the size of a matching is the number of edges included, and $|M|$ denotes the size of M. We will show the total weight of a match M by $w(M)$. Then,

$$w(M) = \sum_{(i,j) \in M} W_{i,j}.$$

2.2 The Algorithm

The below theorem is a basis for an elegant algorithm to find a maximum matching.

Theorem 2.2.1 *If the Equality Subgraph G_l, has a perfect matching M, then M is a maximum-weighted matching in G.*

Proof Let M be a perfect matching in G_l. We have, by definition,

$$\begin{aligned} w(M) &= \sum_{(i,j) \in M} W_{i,j}. \\ &= \sum_{x \in U \cup V} l(x). \end{aligned}$$

Let M' be any perfect matching in G. Then

$$w(M') = \sum_{(i,j) \in M'} W_{i,j} \leq \sum_{x \in U \cup V} l(x) = w(M).$$

Hence,

$$w(M') \leq w(M). \qquad \blacksquare$$

The algorithm starts with a feasible labeling. Then one computes the Equality Subgraph and finds a maximum matching. If it is perfect, then we are done. Else, one adds more edges to the Equality Subgraph by updating the vertex labels. This process does not cause any matched edge to leave the Equality Subgraph. After adding some number of edges to the Equality Subgraph, one eventually finds an augmenting path. Using the augmenting path,

one updates the matching. By the above observation, the new matching is larger than the old one. The algorithm eventually finds a perfect matching. By the above theorem, it is a maximum-weighted matching. We give a more formal definition of the algoritm:

Start with an arbitrary feasible vertex labeling l, determine the Equality Subgraph G_l.

1. Choose an arbitrary maximum matching M in G_l. If M is perfect for G, then M is optimal. Stop. Otherwise, there is some unmatched $x \in U$. Set $S = \{x\}$ and $T = \emptyset$.

2. If $J_{G_l}(S) \neq T$, go to step 3. Otherwise, $J_{G_l}(S) = T$. Find

$$\alpha_l = \min_{x \in S, y \in T^c} \{l(x) + l(y) - w(xy)\}$$

where T^c denotes the complement of T in V , and construct a new labeling l' by

$$l'(v) = \begin{cases} (v) - \alpha_1, v \in S \\ l(v) + \alpha_1, v \in T \\ l(v), otherwise \end{cases}$$

Note that $\alpha_1 > 0$ and $J_{G'_l}(S) \neq T$. Replace l by l' and G_l by $G_{l'}$.

3. Choose a vertex y in $J_{G_l}(S)$, not in T. If y is matched in M, say with $z \in U$, replace S by $S \cup \{z\}$ and T by $T \cup \{y\}$, and go to step 2. Otherwise, there will be an M-alternating path from x to y, and we may use this path to and a larger matching M' in G_l. Replace M by M' and go to step 1.

We will state the following theorem, without giving a proof:

Theorem 2.2.2 *Kuhn-Munkres algorithm finds a maximum-weighted matching of a given weighted complete bipartite graph.*

2.3 The Time Complexity Of The Algorithm

Theorem 2.3.1 *The time complexity of the Kuhn-Munkres algorithm is* $O(n^3)$.

Proof One may observe that, starting with any feasible vertex labeling (for example, as shown above) and any maximum matching, the algorithm takes at most n main steps, where a *main step* is defined as a series of steps after which the size of the match is incremented. A main step takes at most n loops, where a *loop* is a series of steps after which S is incremented

by one element. Clearly, a main step can not take more than n loops. Since a loop takes finite number of steps (2-4), the algorithm takes not more than $O(n^2)$ basic steps. A basic step does not take more than $O(n)$ time if implemented properly (For details see [8]. So, the algorithm takes $O(n^3)$ time. ∎

In the next section, we show a sample run of the algorithm.

2.4 A Sample Run Of The Algorithm

Figure 2.1 shows the weight matrix for a complete bipartite graph with $4 + 4$ vertices.

	V_1	V_2	V_3	V_4
U_1	9	2	8	1
U_2	2	5	2	6
U_3	2	1	5	3
U_4	6	1	1	1

Figure 2.1: Sample graph

We label the vertices as suggested before. See Figure 2.2.

	V_1	V_2	V_3	V_4	$l(U_i)$
U_1	9	2	8	1	9
U_2	2	5	2	6	6
U_3	2	1	5	3	5
U_4	6	1	1	1	6
$l(V_i)$	0	0	0	0	

Figure 2.2: Sample graph with vertex labels

We find the maximum match $M = \{(U_1, V_1), (U_2, V_4), (U_3, V_3)\}$. The labels and the

match are shown in Figure 2.3. The gray items are the matched edges, and the dashed ones are those in the Equality Subgraph but not in the match.

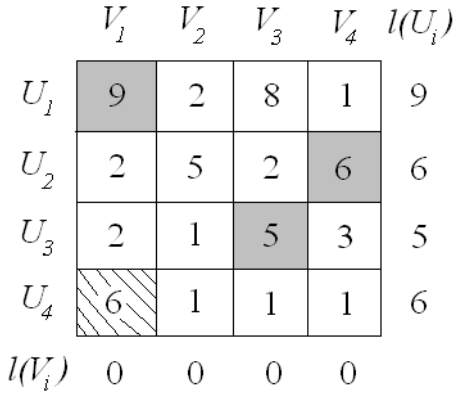

Figure 2.3: Sample run of the algorithm

Step 1. Since U_4 is not matched, we set $S = \{U_4\}, T = \emptyset$.

Step 2. We compute $J_{G_l}(S) = \{V_1\}$. Since $J_{G_l}(S) \neq T$, goto step 3.

Step 3. We choose $y = V_1$ in $J_{G_l}(S) - T$. Since V_1 is matched with U_1 in the match M, we add U_1 to S and V_1 to T. Now $S = \{U_1, U_4\}, T = \{V_1\}$. Goto step 2.

Step 2. We compute $J_{G_l}(S) = \{V_1\}$. Since $J_{G_l}(S) = T$, we find

$$\begin{aligned} \alpha_l &= \min_{x \in \{U_1, U_4\}, y \in \{V_2, V_3, V_4\}} \{l(x) + l(y) - W(xy)\} \\ &= l(U_1) + l(V_3) - W_{13} \\ &= 1. \end{aligned}$$

We decrement the labels of the vertices U_1 and U_4 by one, while we increment that of V_1 by one. Note that, this operation does not cause (U_1, V_1) and (U_4, V_1) to leave the Equality Subgraph. We update the labels as shown in Figure 2.4. Note that, the number of blue items increased by one. We compute $J_{G_l}(S) = \{V_1, V_3\}$.

Step 3. We choose $y = V_3$ in $J_{G_l}(S) - T$. Since V_3 is matched with U_3 in the match M, we add U_3 to S and V_3 to T. Now $S = \{U_1, U_3, U_4\}, T = \{V_1, V_3\}$. Goto step 2.

Step 2. We compute $J_{G_l}(S) = \{V_1, V_3\}$. Since $J_{G_l}(S) = T$, we find

$$\alpha_l = \min_{x \in \{U_1, U_3, U_4\}, y \in \{V_2, V_4\}} \{l(x) + l(y) - W(xy)\}$$

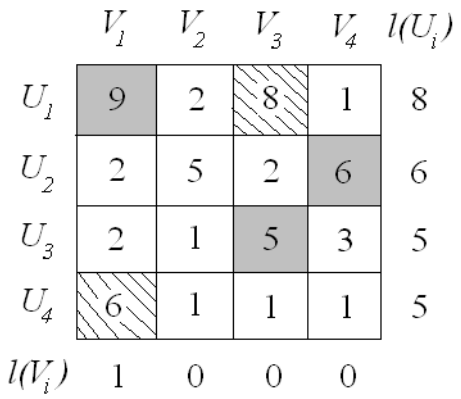

Figure 2.4: Sample run of the algorithm(cont'd)

$$\begin{aligned} &= l(U_3) + l(V_4) - W_{34} \\ &= 2. \end{aligned}$$

We decrement the labels of the vertices U_1, U_3 and U_4 by two, while we increment that of V_1 and V_3 by two. We update the labels as shown in Figure 2.5. We compute $J_{G_l}(S) = \{V_1, V_3, V_4\}$.

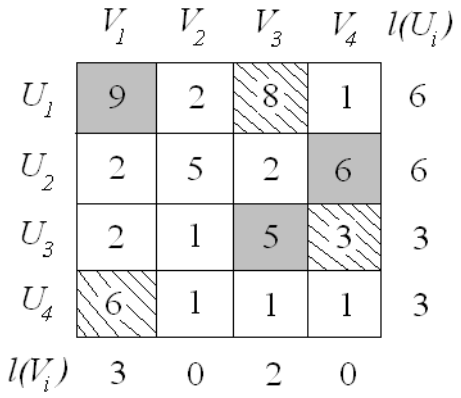

Figure 2.5: Sample run of the algorithm(cont'd)

Step 3. We choose $y = V_4$ in $J_{G_l}(S) - T$. Since V_4 is matched with U_2 in the match M, we add U_2 to S and V_4 to T. Now $S = \{U_1, U_2, U_3, U_4\}, T = \{V_1, V_3, V_4\}$. Goto step 2.

Step 2. We compute $J_{G_l}(S) = \{V_1, V_3, V_4\}$. Since $J_{G_l}(S) = T$, we find

$$\begin{aligned} \alpha_l &= \min_{x \in \{U_1, U_2, U_3, U_4\}, y \in \{V_2\}} \{l(x) + l(y) - W(xy)\} \\ &= l(U_2) + l(V_2) - W_{22} \end{aligned}$$

$= \quad l.$

We decrement the labels of the vertices U_1, U_2, U_3 and U_4 by one, while we increment that of V_1, V_3 and V_4 by one. We update the labels as shown in Figure 2.6. We compute $J_{G_l}(S) = \{V_1, V_2, V_3, V_4\}$.

	V_1	V_2	V_3	V_4	$l(U_i)$
U_1	9	2	8	1	5
U_2	2	5	2	6	5
U_3	2	1	5	3	2
U_4	6	1	1	1	2
$l(V_i)$	4	0	3	1	

Figure 2.6: Sample run of the algorithm(cont'd)

Step 3. We choose $y = V_2$ in $J_{G_l}(S) - T$. We observe that V_2 is not matched in M. So there must be an augmenting path from U_4 to V_2. We grow the Hungarian tree starting from U_4 to find this path, as shown in Figure 2.7.

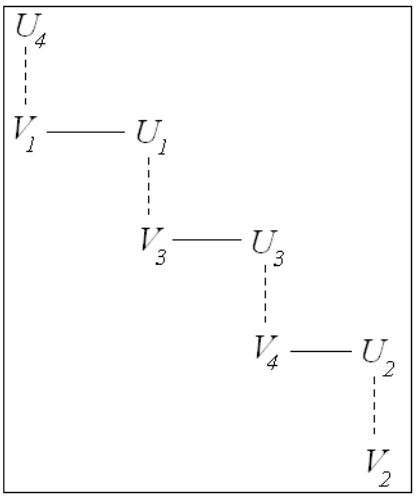

Figure 2.7: The Hungarian tree

In Figure 2.7, the vertical dashed lines show non-matched edges, while horizontal lines

show matched edges. As shown above, the path is

$P = \{(U_4, V_1), (V_1, U_1), (U_1, V_3), (V_3, U_3), (U_3, V_4), (V_4, U_2), (U_2, V_2)\}$.

We construct the new match M' by

$$\begin{aligned} M' &= (M \cup P) - (M \cap P) \\ &= \{(U_4, V_1), (U_1, V_3), (U_3, V_4), (U_2, V_2)\} \end{aligned}$$

Graphically, the above operation corresponds to adding the vertical lines (edges) to and deleting horizontal lines from the original match. Note also that the above operation incremented the size of the match by one. We set $M = M'$ and goto step 1. The new matching is shown in Figure 2.8.

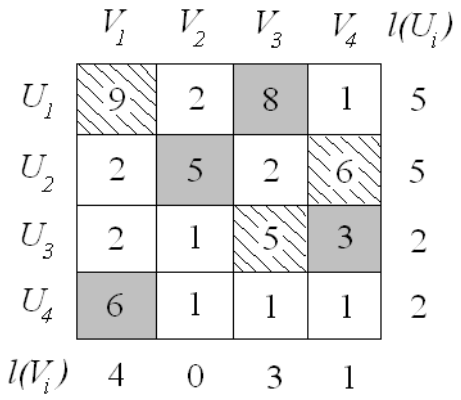

Figure 2.8: Sample run of the algorithm(cont'd)

Step 1. Since M is complete, it is a maximum-weighted matching, with weight 22. **STOP**.

CHAPTER 3

INCREMENTAL ASSIGNMENT PROBLEM

3.1 Introduction

Incremental assignment problem was proposed in [10]. The problem is defined as: Given a weighted bipartite graph and its maximum-weighted matching, determine the maximum-weighted matching of the graph extended with a new pair of vertices, one on each partition, and weighted edges connecting these new vertices to all the vertices on their opposite partitions [10]. An algorithm was proposed to the problem which finds an optimal solution in $O(n^2)$ time. We do not step into details. The algorithm requires that, together with the maximum-weighted matching, a feasible vertex labeling is provided which includes the maximum-matching in the Equality Subgraph.

We do not step into details. Simply stating, the algorithm starts with a feasible vertex labeling for the new graph. The labels of the old vertices are the same with their old labels. The labels of the new vertices (say U_{n+1} and V_{n+1}) are found by some formulas. The algorithm grows the Hungarian tree starting with the new vertex U_{n+1}, and tries to find an augmenting path. When that is found, the match is updated, and the procedure terminates. Else, the labels of the vertices are updated to add more edges to the Equality Subgraph.

The above method is efficient when a problem instance is updated by adding a pair of vertices. Unfortunately, the method requires the feasible vertex labeling of the original graph. We propose a method to find a feasible vertex labeling for a given maximum-weighted matching.

3.2 Finding Feasible Vertex Labeling

Let $G = (U, V, E)$ be a complete bipartite graph, where $|U| = |V| = n$, and $E = UxV.LetU = U_1, U_2, ..., U_n$ and $V = V_1, V_2, ..., V_n$. Let W be an nxn weight matrix. W_{ij} is a real number which is the weight of the edge (U_i, V_j). We may alternatively use $W(U_i, V_j)$ to show the same value.

Now we define the problem of finding a feasible vertex labeling. Let M be a maximum-weighted matching of G. Without loss of generality we assume $M = \{(U_1, V_1), (U_2, V_2), ..., (U_n, V_n)\}$. The problem is: Find a vertex labeling function l defined both on U and V, such that

$$l(U_i) + l(V_j) \geq W_{ij}, i = 1, 2, ..., n. \quad (3.1)$$

$$l(U_i) + l(V_i) = W_{ii}, i = 1, 2, ..., n. \quad (3.2)$$

Figure 3.1 shows an example graph with $n = 4$. The gray items are the matched edges.

	V_1	V_2	V_3	V_4
U_1	8	2	1	9
U_2	2	5	6	2
U_3	5	1	3	2
U_4	1	1	1	6

Figure 3.1: An example graph

In the next section, we state an algorithm to find a feasible vertex labeling.

3.3 An Algorithm To Find A Feasible Vertex Labeling

Lemma 3.3.1 $l(U_i) = W_{ii} - l(V_i), i = 1, 2, ..., n.$

Proof Direct consequence of (3.2). ∎

The above lemma shows that, finding the values $l(V_i)$ is sufficient. Then, one can easily obtain the values $l(U_i)$. In Figure 3.2, x_i is used instead of $l(V_i)$.

	V_1	V_2	V_3	V_4	$l(U_i)$
U_1	8	2	1	9	$8 - x_1$
U_2	2	5	6	2	$5 - x_2$
U_3	5	1	3	2	$3 - x_3$
U_4	1	1	1	6	$6 - x_4$
$l(V_i)$	x_1	x_2	x_3	x_4	

Figure 3.2: Example graph with vertex labels

The constraint $x_3+(5-x_2) \geq 6$ can be obtained from the item W_{23} of the weight matrix. With a little bit modification, it can be expressed as $x_2 - x_3 \leq -1$. Similarly, the item W_{42} suggests that $x_4 - x_2 \leq 5$. Generally, the constraint suggested by the $(i,j)^{th}$ item of the weight matrix is

$$x_i - x_j \leq W_{ii} - W_{ij}$$

Let us define A_{ij} by $A_{ij} = W_{ii} - W_{ij}$. Then what we have is a set of inequations

$$x_i - x_j \leq A_{ij} for i, j = 1, 2, ..., n.$$

This is a well-known problem. These constraints are named *difference constraints*. A well-known algorithm to find some solution is given:

Step 1. Convert the system of linear inequalities into a directed weighted graph G' by

- The inequality $x_i - x_j \leq A_{ij}$ is represented as vertices x_i and x_j, and a directed edge connecting them. The direction is from x_j to x_i. The weight of the edge is $W'(x_j, x_i) = A_{ij}$.
- Introduce vertex v_0, such that $W'(x_0, x_i) = 0$ for all i.

Step 2. Find shortest paths from v_0 to all other vertices. In this step, a single-source shortest path algorithm must be used, which handles negative edge-weights. Bellman-Ford algorithm can be used [2].

If G' contains no negative weight cycles, then the shortest path solution, starting from v_0, is the feasible solution to the system. It is not hard to see that, in our problem setting, G' can not contain any negative weight cycles. Because otherwise would suggest a matching

with bigger weight (constructed by adding the edges in the cycle to and deleting the necessary edges from the matching), contradicting the assumption that the given matching is maximum-weighted.

The time-complexity of the above procedure is that of Step 2, which is $O(n^3)$ if Bellman-Ford algorithm is used.

CHAPTER 4

MAXIMUM-WEIGHTED TREE MATCHING PROBLEM

4.1 Introduction

In this section, we define the **Maximum-Weighted Tree Matching Problem**, which is a variant of **Maximum-Weighted Bipartite Matching Problem** already introduced.

4.2 Notation

In this section, we set some notation. An *undirected graph* is pair (V, E), where V is a finite set of *vertices*, and $E \subseteq V \times V$ is the set of *edges*, where $(x, y) \in E$ implies $(y, x) \in E$ and $(x, x) \notin E$ for all $x, y \in V$. A *path* in a graph is a tuple of vertices $(v_1, v_2, ..., v_n)$ where $n \geq 2$ and $(v_i, v_{i+1}) \in E$ for $i = 1, 2, ..., n-1$. A *cycle* is a path starting and ending at the same vertex. A graph is acyclic if it has no cycles. A *tree* is an acyclic connected undirected graph. A vertex of that graph is called a *node*. A *rooted tree* is a nonempty tree which has a distinguished node called the *root*. A *forest* is a collection of trees. A node p is a *child* of another node q if p and q are connected with an edge, and q is closer to the root. In this case, q is said to be the *parent* of p. The *ancestor* relation is the transitive closure of the *parent* relation. The *descendent* relation is that of the *child* relation. A node is said to be a *leaf* if it has no children.

4.3 Maximum-Weighted Tree Matching Problem

Definition Let F be a forest of rooted trees. Let also $k \in N$ be given. For each node d in F, the function w is defined. $w(d)$ is a k-tuple of natural numbers. So, if D is the set of

nodes in F, $w : D \rightarrow N^k$. We will simply show the t^{th} elementh of $w(d)$ by $w(d,t)$. The problem is to find a k-tuple $K = (d_1, d_2, ..., d_k)$ of k distinct nodes of F such that

$$\sum_{i=1}^{k} w(K(i), i)$$

is maximal among all such tuples, subject to the **independency** constraint, which states that any two distinct nodes $x, y \in K$ should be independent, where independency is defined as below:

Definition Two distinct nodes x and y are said to be **dependent** if they simultaneously occur in some shortest path from a root to a leaf. Otherwise they are said to be **independent**. Simply we will say that a set A of nodes is independent if any two nodes in A are so. Also we will say that the sets A and B are independent if for all pairs $(x \in A, y \in B)$, x and y are independent.

Note that, above we used $K(i)$ to show the i^{th} element of the k-tuple, treating the k-tuple as a function from $\{1, 2, ..., k\}$. We will call this problem as **Maximum-Weighted Tree Matching Problem**.

Assume that we have k jobs, and some number of workers. Any worker can do any job with a given profit. The workers may come together forming groups. Also any group can do any job, again with some given profit. The groups may also come together forming another group. This recursive structure is formulated above by using the notion of *tree*s. One should note that, this formulation does not allow all groupings between workers, the tree structure restricts the grouping. Indeed, this structure is similar to that in companies. The leaf nodes correspond to workers, and the other nodes to groups. The problem is to assign the jobs to workers/groups with maximum profit. The problem definition requires all jobs to be performed, since we have defined K as a k-tuple. An alternative definition might be done, defining K as a partial one-to-one mapping. The *dependency* constraint amounts to saying *"a worker can not be used simultaneously to perform two distinct jobs"*. One may have also noticed that, the problem has a solution if and ony if k does not exceed the number of leaves.

In the rest of this work, we investigate some different versions of the problem. In the next chapter, we show that a generalized version is *NP-Hard*. After that, we show that a much simplified version can be solved in polynomial time. Then, we present a genetic algorithm to solve the above problem.

CHAPTER 5

MAXIMUM-WEIGHTED SUBSET MATCHING PROBLEM

5.1 Introduction

We define the **Maximum-Weighted Subset Matching Problem** and show that it is NP-Hard, by proving NP-Completeness of the decision problem version.

5.2 Maximum-Weighted Subset Matching Problem

Definition Let U be a finite set of workers and J be a finite set of jobs. Let $W : (J \times 2^U) \rightarrow N$ be a partial weight function. The **Maximum-Weighted Subset Matching Problem** asks for a matching $M : J \rightarrow 2^U$ such that

$$\sum_{j \in J} W(j, M(j))$$

is maximal among all matchings, subject to the constraint

$$\text{If } x, y \in J \text{ with } x \neq y, \text{ then } M(x) \text{ and } M(y) \text{ are disjoint.}$$

The weight function should be defined on the $(job, subset)$ pairs in the matching.

Informally speaking, the above constraint says that *"A worker can not be used more than once"*. Note that it is probable that one can not find any matching to complete the jobs at all. The matching is a complete function from the set of jobs, which means that *"All jobs should be performed"*.

In the next section we prove that the decision problem version of the defined problem is NP-Complete.

5.3 NP-Completeness

We first define a decision problem related to the above problem:

Definition Let U be a finite set of workers and J be a finite set of jobs. Let $W : (J \times 2^U) \rightarrow N$ be a partial weight function. Let $k \in N$ be given. The question is whether there exists some matching $M : J \rightarrow 2^U$ such that

$$k \leq \sum_{j \in J} W(j, M(j))$$

subject to the constraint

$$\text{If } x, y \in J \text{ with } x \neq y, \text{ then } M(x) \text{ and } M(y) \text{ are disjoint.}$$

For simplicity we call this problem **MWSM-dec**. We will describe an instance of **MWSM-dec** by a quadruple (U, J, W, k).

Now we introduce the so-called **Set-Packing** problem.

Definition Suppose we have a finite set S and a list L of subsets of S. Then, the **Set-Packing** problem asks if some k subsets in L are pairwise disjoint (in other words, no two of them intersect). We let a triple (S, L, k) specify an instance of **Set-Packing**.

Theorem 5.3.1 ***Set-Packing*** *is NP-Complete.*

Proof We reduce the well-known **k-clique** problem, or simply **Clique** problem, which was shown to be NP-Complete in [6]. Let $G = (V, E)$ be an undirected graph, and k be a natural number. The **k-clique** problem asks whether G has a k-clique, that is, a complete subgraph with k vertices. Let $n = |V|$. Then we will construct a list L of n subsets. We set $S = V \times V$. We assume an ordering on the vertices V of G. We also assume a one-to-one correspondence between V and L. For every $(x, y) \in (V \times V - E)$ with $x < y$, we add the elements (x, y) to both subsets corresponding to x and y. Finally, two vertices are neighbours in G iff the corresponding subsets in L are disjoint. Thus, there is a k-clique in G iff there are some k subsets in L which are pairwise disjoint. Note that, the reduction is polynomial. We would similarly reduce the **Independent Set** Problem if we have simply replaced $(V \times V - E)$ by E in the above reduction. ∎

Below we show that, **MWSM-dec** is NP-Complete.

Theorem 5.3.2 ***MWSM-dec*** *is NP-Complete.*

Proof First we observe **MWSM-dec** $\in P$. Given a candidate matching $M : J \to 2^U$, one can verify the satisfaction of the two constraints in polynomial time. Second, we reduce the **Set-Packing** problem introduced above. Assume that we have a finite set S and a list L of subsets of S, and let $k \in N$ be given. We may assume $k \leq |L|$, since otherwise would directly indicate a **"NO"** answer. The constructed instance of **MWSM-dec** is as follows: The set of workers U is S. There are k jobs, that is $|J| = k$. For any pair $j \in J$ and $X \in L$, the weight function is defined as $W(j, X) = 1$. W is undefined on other pairs. Informally speaking, this means that *"Any set of workers in L can do any of the jobs, with constant profit"*. It is easy to see that

$$(S, L, k) \in \textbf{Set-Packing} \text{ iff } (U, J, W, k) \in \textbf{MWSM-dec}.$$

The size of the new problem instance is at most square of the original problem instance, since we assumed $k \leq |L|$, showing that the reduction is polynomial. ∎

5.4 Conclusion

The decision problem version of the **Maximum-Weighted Subset Matching Problem**, which we have called **MWSM-dec** is NP-Complete, as shown in the above work. This shows that the optimization problem version is NP-Hard, thus no exact polynomial-time algorithm exists unless P=NP.

CHAPTER 6

MAXIMAL K-NODES PROBLEM

6.1 Introduction

We define the **Maximal k-nodes Problem** and show that it can be solved efficiently, i.e., in polynomial time.

6.2 Maximal k-Nodes Problem

Definition Let F be a forest of rooted and weighted trees. The weights of the nodes are positive integers. For a node d, let $w(d)$ denote the weight of d. Let also $k \in N$ be given. The problem is to find a set $K = \{d_1, d_2, ..., d_k\}$ of k nodes in F such that, the sum of the weights of these nodes

$$\sum_{d \in K} w(d)$$

is maximal among all such sets, subject to the **independency** constraint, which states that any two distinct nodes $x, y \in K$ should be independent (or simply, K should be independent), where independency has been defined in Chapter 4.

We will call this problem as **Maximal k-nodes Problem**.

REMARK: One may formulate an instance of the above problem by using a weighted graph. The nodes of the graph are those of the forest F. The node weights are the same with the weights of those in F. Two nodes are combined with an edge if and only if they simultaneously occur in the same shortest path from a root to a leaf in F. Independency of two distinct nodes in a graph should be defined as "having no edges between them", in this case.

6.3 An Algorithm To Solve The Problem

In this section, we present an algorithm to solve the problem. Firstly, for simplicity we assume that F is actually a tree. We do not lose any generality doing so, because, otherwise, F can be converted to a tree by adding a duplicate of some node r as the root, and making all the other roots children of r. This will not effect the result.

The below function EvalMax() is the main function. For a given tree T, it returns a sequence $S = (S_0, S_1, ..., S_t)$, where t is the number of leaves in T, and S_k denotes the sum of maximal k-nodes, for $k = 0, 1, ..., t$. Thus, the algorithm computes the optimum solution for all possible k values. Although the algorithm does not keep which nodes are in the optimal selection, it is not hard to modify it to give not only the sums, but the sets of nodes also. It may effect the time complexity, without harming polynomiality.

The algorithm uses a bottom-up approach. We give a pseudo-code below:

Algorithm 6.3.1: EvalMax(T)

Let r be the (weight of) root of T

Let $c_1, c_2, ..., c_m$ be the children of r.

for $i \leftarrow 1$ **to** m

  do $maxseq_i \leftarrow$ EvalMax(c_i)

$combined \leftarrow$ Merge($maxseq_1, maxseq_2, ..., maxseq_m$)

if m=0

  then $\begin{cases} combined[0] \leftarrow 0 \\ combined[1] \leftarrow r \end{cases}$

  else if r>combined[1]

  then $combined[1] \leftarrow r$

return ($combined$)

The above function calls the Merge() function, pseudo-code of which is given below. Merge() merges m sequences iteratively. First it merges the first and the second sequences. Then it merges the result with the third one. Then the fourth one, ..., so on.

Algorithm 6.3.2: MERGE$(s_1, s_2, ..., s_m)$

combined $\leftarrow s_1$

for $i \leftarrow 2$ **to** m

  do *combined* $\leftarrow$ MERGETWO(*combined*, s_i)

return (*combined*)

The above function calls the MERGETWO() function, pseudo-code of which is given below. It merges two sequences s_1 and s_2 by examining all possible pairs $(s_1[i], s_2[j])$. So, for example, if the maximum sum of 7 nodes can be reached by selecting 3 nodes from the first set (not necessarily a tree) and 4 nodes from the second set (again not necessarily a tree), it will be detected by the function when $i = 3$ and $j = 4$.

Algorithm 6.3.3: MERGETWO(s_1, s_2)

Let $len1$ be the length of s_1, and $len2$ be the length of s_2.

for $i \leftarrow 0$ **to** $(len1 + len2)$

  do *combined*$[i] \leftarrow 0$

for $i \leftarrow 0$ **to** $len1$

  do for $j \leftarrow 0$ **to** $len2$

  do if $s_1[i] + s_2[j] >$ *combined*$[i + j]$

  then *combined*$[i + j] \leftarrow s_1[i] + s_2[j]$

Note that $s_1[0]$ and $s_2[0]$ are always 0.

return (*combined*)

6.4 Correctness Of The Algorithm

Lemma 6.4.1 *Let N_1 and N_2 be two disjoint sets of nodes. Let s_1, s_2 be two finite sequences, where $s_1[j]$ denotes the maximum possible sum of j independent nodes in N_1, and similarly for s_2. Assume that N_1 and N_2 are independent. Then the output of* MERGETWO(s_1, s_2) *is the finite sequence corresponding to $N_1 \cup N_2$ in the same sense.*

Proof We let *length*(x) denote the size of a sequence or a set x. Let $NS = N_1 \cup N_2$, and let s be the sequence corresponding to N. We must show that s is actually the output of MERGETWO(s_1, s_2). Consider $s[t]$ for some $t \in \{0, 1, 2, ..., length(s_1)+length(s_2)\}$. It denotes the maximum sum of t independent nodes in NS. Clearly, if $A \subseteq NS$ is an independent set of size t, there exists two sets $A_1 \subseteq N_1$ and $A_2 \subseteq N_2$, where $A = A_1 \cup A_2$, A_1 is independent

and A_2 is also independent. Let $t_1 = length(A_1)$ and $t_2 = length(A_2)$. Then $t = t_1 + t_2$. We also have $0 \leq t_1 \leq length(s_1)$ and $0 \leq t_2 \leq length(s_2)$. So, when $i = t_1$ and $j = t_2$, the procedure computes $s_1[t_1] + s_2[t_2]$, and puts that value to $combined[t_1 + t_2]$ if it was not reached before. So we conclude that, the return value *combined* is equal to s. ∎

Lemma 6.4.2 *Let $N_1, N_2, ..., N_m$ be pairwise disjoint sets of nodes. Further let $N_1, N_2, ..., N_m$ be pairwise independent. Let s_i correspond to N_i. Then* MERGE$(s_1, s_2, ..., s_m)$ *returns the sequence corresponding to $N_1 \cup N_2 \cup ... \cup N_m$.*

Proof We will show by induction that, after the execution of the **for** loop, *combined* stores the desired sequence. For the case $m = 1$, nothing to show, since $combined = s_1$ in that case, as desired. If $m = 2$, the procedure returns MERGETWO(s_1, s_2). Since N_1 and N_2 are independent, this result is the desired sequence corresponding to $N_1 \cup N_2$. Let $m > 2$, and assume the correctness of the lemma for $m-1$. Then, before the last call MERGETWO$(combined, s_m)$, *combined* stores the sequence corresponding to $N_1 \cup N_2 \cup ... \cup N_{m-1}$. Since $N_1 \cup N_2 \cup ... \cup N_{m-1}$ and N_m are independent, MERGETWO$(combined, s_m)$ just returns the desired sequence for $N_1 \cup N_2 \cup ... \cup N_m$. ∎

Theorem 6.4.3 *Let T be a tree. Then* EVALMAX(T) *returns the sequence corresponding to T.*

Proof We will make induction on size of T, where we define the size of a tree as the number of nodes in it. The smallest tree contains just the root. For that case, as handled seperately in the procedure, the 0^{th} element of the return sequence is 0, and the 1^{st} one is the weight of the root, as desired. Assume that r has some children $c_1, c_2, ..., c_m$. Then the children are smaller, and by the inductive hypothesis, EVALMAX(c_i) returns the desired sequence corresponding to c_i. They are stored in $maxseq_i$. We note that, c_i are disjoint. We also note that they are pairwise independent. So, by the above lemma, MERGE$(maxseq_1, maxseq_2, ..., maxseq_m)$ returns the desired sequence corresponding to $c_1 \cup c_2 \cup ... \cup c_m$. So, after the call to MERGE(), *combined* stores the desired sequence only except the element $combined[1]$. The maximum sum of 1 element in the whole tree may be the weight of root. This case is handled seperately, possibly changing $combined[1]$. We note that, r can not be used together with any node in c_i, because it is dependent to any of them. ∎

6.5 An Example Run

In this section, we show an example run of the algorithm without giving much details. Figure 6.1 shows an example tree where the numbers in the rectangles show the weights of nodes.

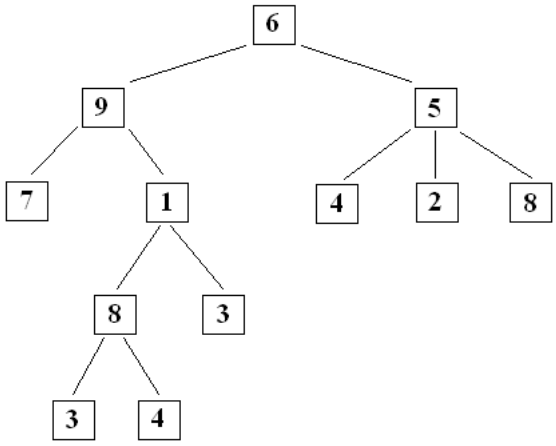

Figure 6.1: An example tree

Figure 6.2, Figure 6.3 and Figure 6.4 show how the algorithm works from bottom to up. The sequences appearing near the nodes show the output of EvalMax().

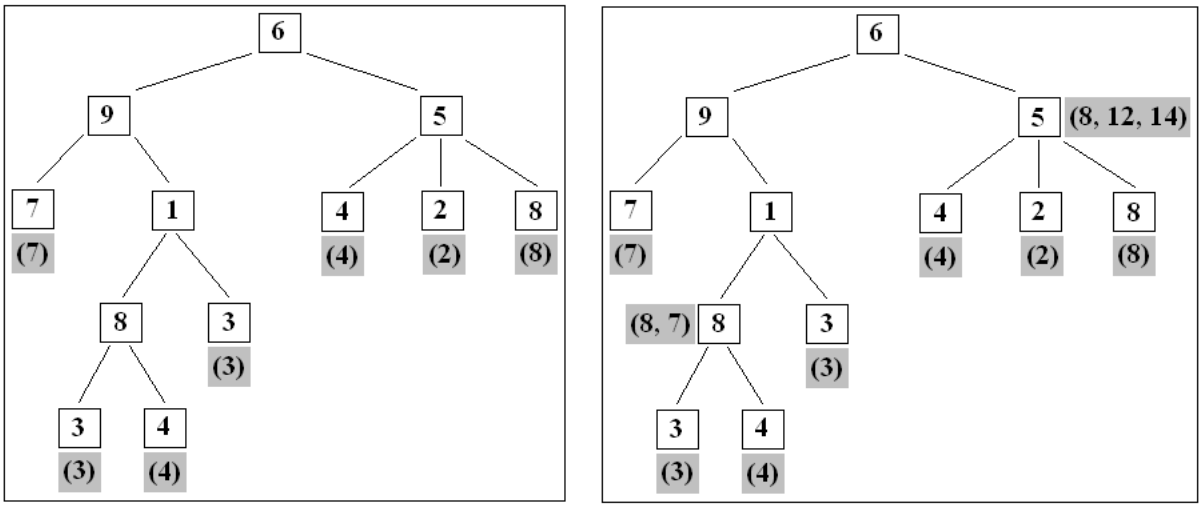

Figure 6.2: Example tree with the outputs of EvalMax()

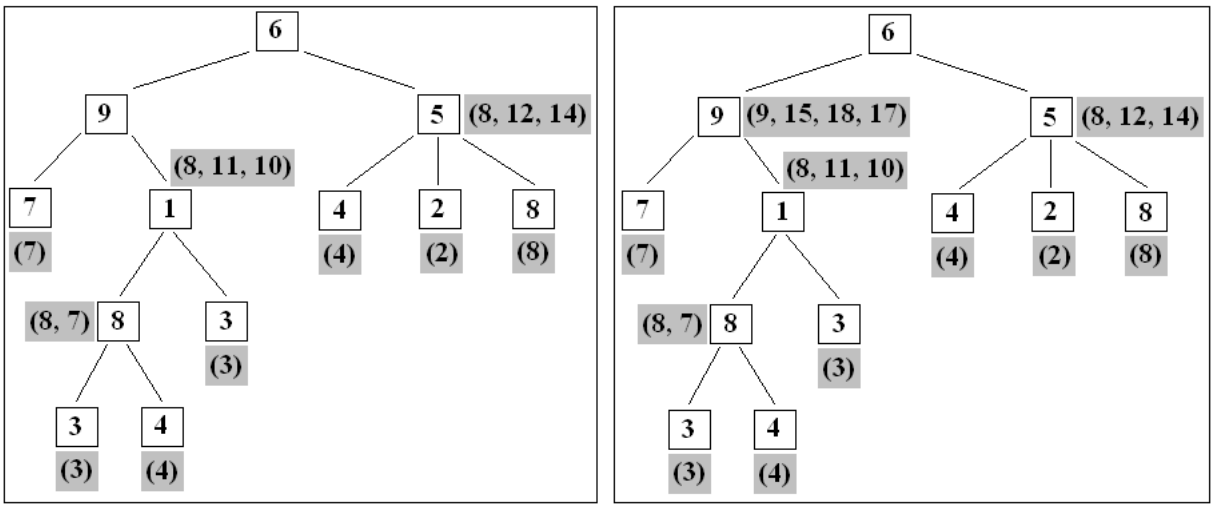

Figure 6.3: Example tree with the outputs of EvalMax() (cont'd)

The result is shown at Figure 6.4, near the root of the tree. The sequence $(9, 17, 23, 27, 30, 32, 31)$ (note that the 0^{th} elements of the sequences are not shown, since they are all 0) indicates, for example, the selection of 4 pairwise independent nodes from the tree gives the maximal sum of 27, and the number is 32 for 6 nodes. The reader may have noticed that, the sequence is not monotone increasing; in fact it doesn't have to be so.

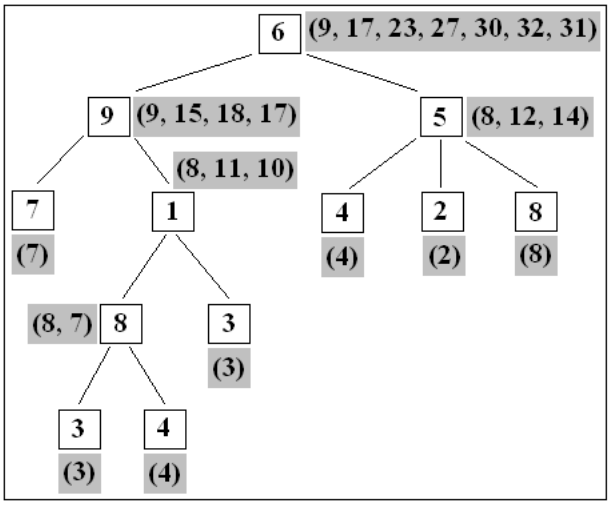

Figure 6.4: Example tree with the outputs of EvalMax() (cont'd)

6.6 The Complexity Of The Algorithm

In this section, we investigate the time complexity of the stated algorithm and prove that it runs in polynomial time.

Theorem 6.6.1 *Let T be a tree, and n be the number of nodes in T. The worst case running time of* EvalMax(T) *is* $O(n^3)$.

Proof The EvalMax() function is called for every node exactly once. It consumes constant time except for the recursive call to EvalMax() and the call to Merge(). The runing time of Merge() is determined by that of MergeTwo(). The running time of MergeTwo() is $O(len_1 * len_2)$, where len_1 and len_2 are the lengths of the sequences sent as parameters.

Now we investigate the running time of Merge($s_1, s_2, ..., s_m$). The length of the sequence *combined* increases from s_1 to $s_1 * s_2 * ... * s_m$ by calling MergeTwo() $m-1$ times. The calls to MergeTwo() consumes $s_1 * s_2$, $(s_1 + s_2) * s_3$, $(s_1 + s_2 + s_3) * s_4$, ..., $s_1 + s_2 + ... + s_{m-1}) * s_m$ time. If we let len_i denote the length of sequence s_i, we have the running time of Merge($s_1, s_2, ..., s_m$):

$$T_{Merge} = \sum_{i=1}^{m-1} \sum_{j=1}^{i} len_j * len_{i+1}$$

Now let

$$A = (\sum_{i=1}^{m} len_i) * (\sum_{i=1}^{m} len_i)$$

A little inspection shows that $T_{Merge} < A$, observing that all of the terms in T_{Merge} is included in the expansion of A, and some terms in that expansion do not appear in T_{Merge} . Since

$$\sum_{i=1}^{m} len_i \in O(n)$$

we have that

$$A \in O(n^2)$$

We conclude that $T_{Merge} \in O(n^2)$. Then running time of a call to EvalMax() is $O(n^2)$, if we do not count the recursive calls. To count the number of recursive calls, we note that, EvalMax() is called n times. So, summing up, the total running time of EvalMax(T) is $O(n^3)$. ∎

We note without proving that, if the tree is a balanced t-ary tree for some $t \geq 2$, the running time actually is $O(n^2)$, but there are cases where $O(n^3)$ is the tightest possible bound, as shown in Figure 6.5 (Note that, the depth of the tree is approximately half of the number of nodes).

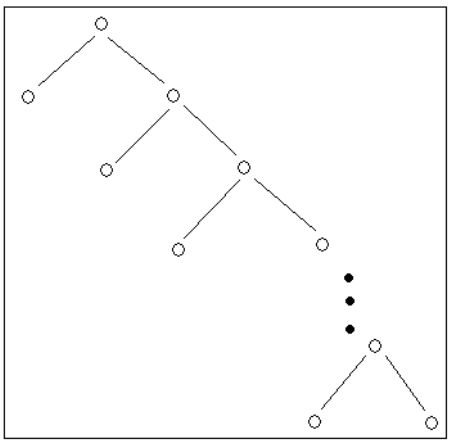

Figure 6.5: An example tree structure showing the worst case of the algorithm

6.7 Conclusion

We have defined the **Maximal k-nodes** problem and have shown that it can be solved in $O(n^3)$ time. Without proving, we have stated that it is the tightest possible bound for the

time complexity of the algorithm. We have not counted the extra cost for outputting the selected nodes, which may increase the complexity.

CHAPTER 7

IMPLEMENTATION OF A GENETIC ALGORITHM

7.1 Introduction

In this section, we introduce the genetic algorithms [5]. A survey on genetic algorithms can be found in [4]. An optimization problem, shortly, is a problem in which, one expects to find an element E in a search space S, such that $f(E)$ is a maximal element of the set $\{f(X) : X \in S\}$, where $f : S \to N$ is a profit function. In computer science, the optimization problems dealed with have easily computable profit functions, and easily expressable search spaces. Often, finding a maximal element in reasonable time is not obvious. In particular, if an optimization problem is *NP-Hard*, it is impossible unless $P = NP$. For such problems, we need efficient methods to find some element E in S, which we can use for practical purposes, expecting $f(E)$ to be as big as possible.

A genetic algorithm may provide an approximate solution for such a problem. The concept of a *genetic algorithm* has been inspired from the evolutionary process in biological organisms. Mutations and sexual reproduction in (some) organisms may increase the *quality* of the population, where the *quality* of a population depends on that of the organisms. *Chromosomes* play the important role in the process, since they are the main parts of the organisms which are mutated and crossed-over while sexual reproduction takes place. A genetic algorithm is roughly as below:

For a search space S, one finds an encoding function $encode : S \to T$, where T is a set of *strings*. The *strings* may be strings consisting of 0's and 1's, to represent an element of the search space. Then one generates a (generally random) pool of chromosomes, called a *generation*. Then, step by step, the generation is updated by means of *mutation*

and *cross-over* mechanisms. A *mutation* is a sudden change in some part of a chromosome. Assuming representation by bits, a *mutation* on the chromosome $(a_1, a_2, ..., a_n)$ may result with $(a_1, a_2, ..., a_{k-1}, a'_k, a_{k+1}, ..., a_n)$, if the k^{th} bit is mutated. A *cross-over* is a mechanism which generates two new chromosomes from two old chromosomes. A typical *cross-over* between the chromosomes $(a_1, a_2, ..., a_n)$ and $(b_1, b_2, ..., b_n)$ generates two new chromosomes $(a_1, a_2, ..., a_k, b_{k+1}, b_{k+2}, ..., b_n)$ and $(b_1, b_2, ..., b_k, a_{k+1}, a_{k+2}, ..., a_n)$. That is, the two old chromosomes are divided from the k^{th} position and the parts after that position are exchanged. After a new generation is generated, the process goes on using that one. One may decide a fixed *number of iterations*, or require a criterion to be met, to finalize the algorithm. The final generation is expected to contain *good* chromosomes corresponding to elements of S with high profit.

7.2 Implementation

In this section we describe our genetic algorithm implementation. The functions given below are not exactly the same as those in the actual implementation, we are simplifying them.

We use the notation in Chapter 4. First we state an assumption on the input, which have made the implementation little bit simpler, although it is not necessary.

Assumption F is a strictly binary rooted tree. By *"strictly binary"* we mean that, a node is either a leaf, or has exactly two children.

We are not going into formal details. The assumption of F being a tree has been made in Section 6.3 before. If a non-leaf node d has only one child d_1, then they can be combined into a single node d', for $i = 1, 2, ..., k$, setting $w(d', i) = max\{w(d, i), w(d_1, i)\}$. Observe that, d and d_1 are dependent to exactly same nodes besides being dependent to each other, and assigning a job to one of them is not reasonable if the other one is better at that job. If d has more than two children $d_1, d_2, ..., d_t$, then one can add a duplicate node d' (say, duplicate of d_2) as a child of d, keeping d_1 as a child of d, and making $d_2, d_3, ..., d_t$ children of d'. This procedure can be repeated until the tree is binary. Observe that, after this procedure, the number of nodes in the resulting tree is not more than twice the original one.

We assign an index to all nodes. The index $index(d)$ of a node d is a number between 1 and n, where n is the number of nodes. Tree is preorder traversed, the indexes of the nodes show the order they are visited at this traversed. Figure 7.1 shows an example indexing:

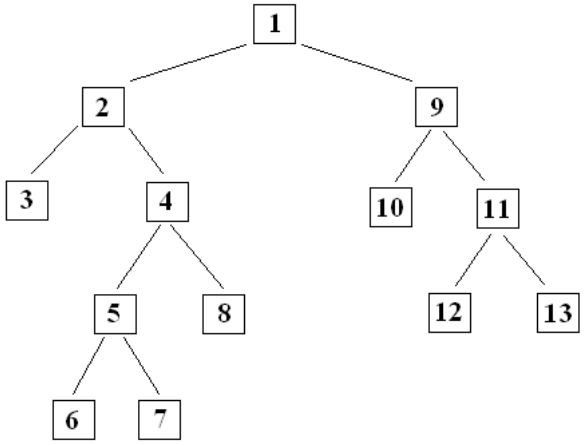

Figure 7.1: Indexes of nodes in an example tree

From now on, we will identify a node with its index.

Now we explain the chromosome structure. A chromosome $p = (p_1, p_2, ..., p_k)$ is a k-tuple of nodes, where $p_1 < p_2 < ... < p_k$, and $p_1, p_2, ..., p_k$ are independent. This tuple actually can be seen as a set, since it does not have the meaning of assignment of jobs to nodes. It does just indicate a selection of independent nodes. This will make sense after we describe the scoring of chromosomes.

Cross-over between two chromosomes

$$p = (p_1, p_2, ..., p_k)$$

$$q = (q_1, q_2, ..., q_k)$$

results with two new chromosomes

$$p^* = (p_1, p_2, ..., p_t, q_{t+1}, q_{t+2}, ..., q_k)$$

$$q^* = (q_1, q_2, ..., q_t, p_{t+1}, p_{t+2}, ..., p_k)$$

if we divide them from the t^{th} position.

Referring to the example tree given in the above figure, assuming $k = 5$, crossing-over two genes

$$p = (3, 5, 8, 10, 12)$$

$$q = (3, 6, 7, 10, 11)$$

from the 2^{nd} position gives two new chromosomes

$$p^* = (3, 5, 7, 10, 11)$$

$$q^* = (3, 6, 8, 10, 12)$$

Note that, in the above example, p^* is not a valid chromosome, since 5 and 7 are dependent nodes. We provide a mechanism to correct the chromosomes after cross-overs. First we place 0's in the chromosome to indicate invalid nodes. The chromosome becomes

$$p^{**} = (3, 5, 0, 10, 11)$$

The below algorithm is used to correct p^{**} by the call CorrectChromosome(p^{**}).

Algorithm 7.2.1: CorrectChromosome(c)

Let *zeros* be number of 0's in c

for $i \leftarrow 1$ **to** *zeros*

do
- Let $frees$ be free nodes, i.e., nodes that can be safely added to c
- Let *nonleaves* be nonleaf nodes in c
- $r \leftarrow$ a random node in one of $frees$ and *nonleaves*
- **if** r is in $frees$
  - **then** Replace a 0 in c with r
  - **else if** r is in *nonleaves*
    - **then**
      - Let r_1 and r_2 be children of r
      - Replace r with 0 in c
      - Replace a 0 in c with r_1
      - Replace a 0 in c with r_2

CorrectChromosome(p^{**}) corrects the dependencies in p^{**}. It should be sorted again to be a valid chromosome, according to our definition.

We have not used any mutations. But the procedure of correcting a resulting chromosomes after a cross-over may be argued to include mutations, since it may randomly add nodes to a chromosome, and randomly delete nodes from that.

When the above function is called by CorrectChromosome(p^{**}) and the resulting chromosome is sorted, we may have valid chromosomes like below

$$p^{***} = (3, 5, 8, 10, 11), or$$

$$p^{***} = (3, 5, 10, 12, 13)$$

In the first case, node 8 has been randomly selected and added to c. In the second one, node 11 has been randomly selected and replaced with its two children 12 and 13.

The generation of the initial pool actually is done by the above algorithm, starting with an empty (i.e., all items are 0) chromosome, and correcting it.

The fitness of a chromosome is computed by the Kuhn-Munkres algorithm, which we call to optimally assign the k jobs to selected k nodes in a chromosome. As we have indicated before, a chromosome does not actually show a matching, instead it only shows a selection of nodes. One might follow an alternative approach, in which chromosomes show the matching. But this approach does not take advantage of the fact that, assigning k jobs to k nodes optimally is a task for which, a polynomial time solution already exists. On the other side, following our approach, one requires $O(k^3)$ time to compute the fitness of a chromosome, while the alternative approach reduces it to linear time.

7.3 Experimental Results

We have written ANSI-C code implementing the tasks briefly described above. For test purposes, we have written another C code to generate random inputs of different sizes, and compute the optimum scores by exhaustive search. In Table 7.1 we give the experimental results on randomly generated inputs. Success ratio is found by dividing the score of the best chromosome found in given number of iterations by the optimum score. We state some parameters used in the algorithm. The size of the pool is $10n$, where n is the number of nodes. At the end of each iteration, the chromosomes are sorted according to the fitness function. Best n chromosomes are directly copied to the new generation. So we never lose the best chromosome, guaranteeing the success ratio to be non-decreasing while number of iterations increases. $2n$ pairs among $5n$ best chromosomes are randomly selected and crossed-over. A chromosome is not used in more than one cross-over. Worst $4n$ chromosomes are deleted. Totally, the new generation consists of $4n$ new chromosomes, and $6n$ old chromosomes.

For bigger tree sizes, we do not have a fast way to obtain the optimum score. So, we use upper bounds to observe the performance of the genetic algorithm. An upper bound for a problem instance can be easily found by summing up the best weights for each job. Also the parameters used for big inputs are not the same. The size of the pool is 100, independent of the problem size. At the end of each iteration, the chromosomes are sorted according to the fitness function. Best 3 chromosomes are directly copied to the new generation. 40 pairs among 90 best chromosomes are randomly selected and crossed-over. A chromosome is not used in more than one cross-over. Worst 80 chromosomes are deleted. Totally, the new generation consists of 80 new chromosomes, and 20 old chromosomes.

In Figure 7.2, Figure 7.3, Figure 7.4, and Figure 7.5 we give the experimental results for four random inputs. For the first one, the number of nodes is 1,001, and the number of jobs is 40. For the second one, the number of nodes is 2,001, and the number of jobs is 50. For the third one, the number of nodes is 10,001, and the number of jobs is 60. For the fourth one, the number of nodes is 100,001, and the number of jobs is 100. The plots have been generated using ***gnuplot***.

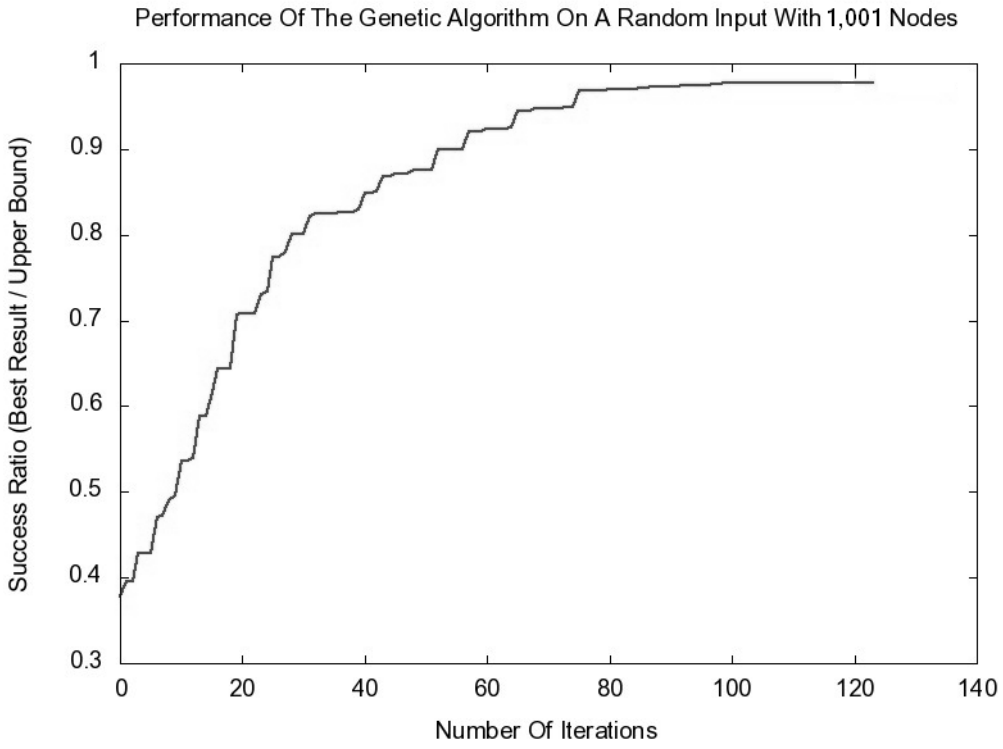

Figure 7.2: Performance of the genetic algorithm on random input with 1,001 nodes

Table 7.1: Performance Of The Genetic Algorithm On Small Random Inputs

Number of nodes	**Number of jobs**	**Number of iterations**	**Success Ratio**
7	3	0	1.000000
17	7	0	1.000000
21	8	0	0.994631
		1	1.000000
25	10	0	0.992129
		1	0.999845
		3	0.999971
		5	0.999972
		10	1.000000
29	12	0	0.996969
		1	0.997208
		3	0.999826
		5	1.000000
33	13	0	0.999918
		1	0.999918
		3	0.999989
		5	0.999989
		10	1.000000
49	20	0	0.955593
		1	0.982867
		3	0.996585
		5	0.998506
		10	0.999446
		245	0.999739

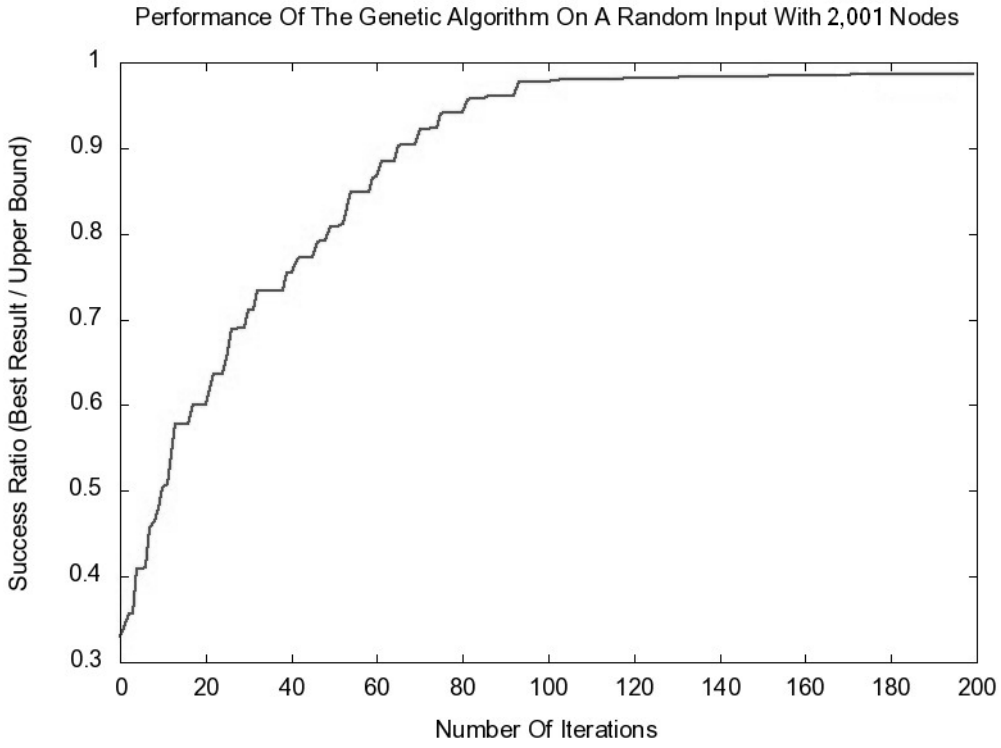

Figure 7.3: Performance of the genetic algorithm on random input with 2,001 nodes

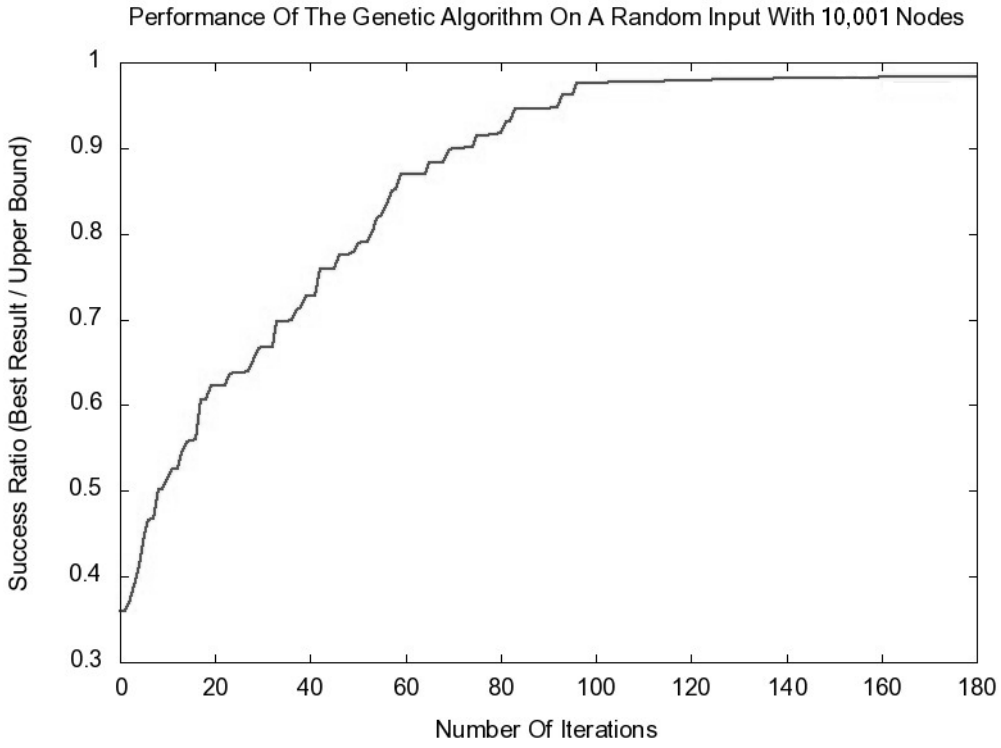

Figure 7.4: Performance of the genetic algorithm on random input with 10,001 nodes

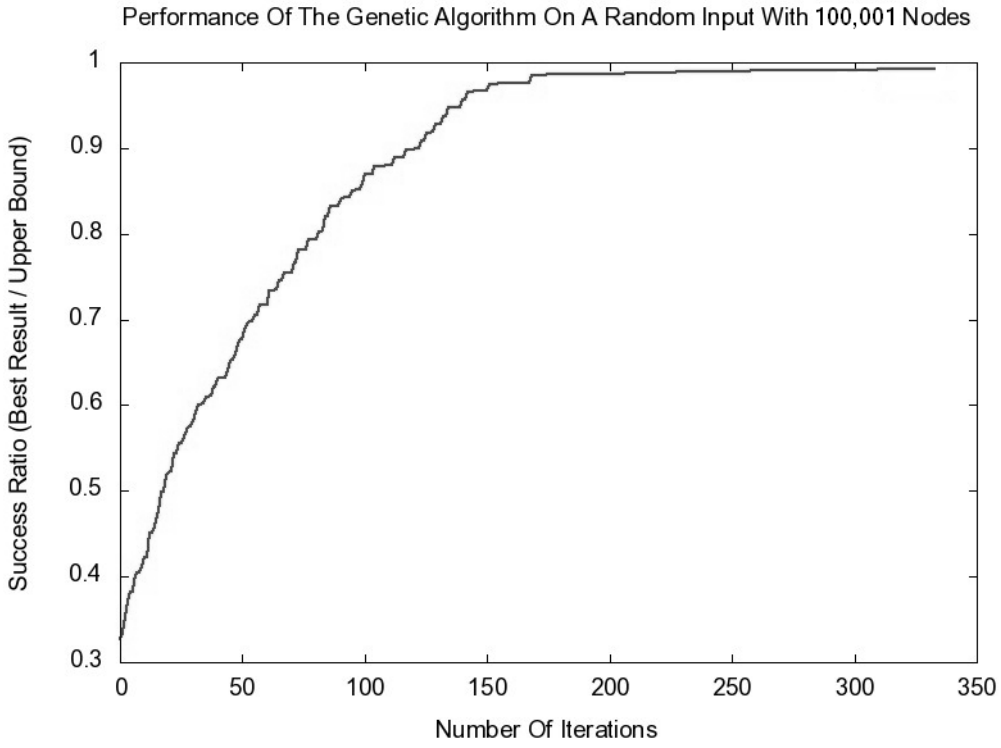

Figure 7.5: Performance of the genetic algorithm on random input with 100,001 nodes

7.4 Conclusion

The defined problem may be in P, may be NP-Complete. We do not have a proof for the both cases. The genetic algorithm we have implemented has given good results for small inputs. It is hard to examine for large inputs, since we should do exhaustive search on the input to find the optimum score for comparison. We argue that, including Kuhn-Munkres algorithm in the implementation makes it better, since Kuhn-Munkres algorithm finds the optimal ordering of a set of nodes.

CONCLUSION

In this work, the classical assignment problem (also called *Linear Sum Assignment Problem* or *Maximum-Weighted Bipartite Matching Problem*) has been investigated. The polynomial-time Kuhn-Munkres algorithm has been described. Some features of the so-called *Incremental Assignment Problem* has been investigated. An $O(n^3)$ procedure has been proposed for the problem of finding the feasible vertex labels, when the maximum-weighted matching of a bipartite graph is known. Unfortunately this complexity renders the procedure unsuitable for practical purposes. *Maximum-Weighted Tree Matching Problem* has been defined. A generalized version, which we have called *Maximum Subset Matching Problem*, has been shown to be NP-Hard. On the other hand, we have been presented an efficient algorithm to solve a simplified version which we have called *Maximal k-nodes Problem.* A genetic algorithm scheme has been presented to solve the *Maximum-Weighted Tree Matching Problem*, and experimental results has been given for random small inputs. A future work may be solving the *Maximum-Weighted Tree Matching Problem* in polynomial time, or showing it is NP-Hard.

REFERENCES

[1] M. Akgül. The linear assignment problem. *Combinatorial Optimization, M. Akgul and S. Tufecki, eds., Springer Verlag, Berlin*, pages 85–122, 1992.

[2] Paul E. Black. *"Bellman-Ford algorithm", in Dictionary of Algorithms and Data Structures [online].* U.S. National Institute of Standards and Technology, 2005.

[3] R. E. Burkard and E. Çela. Linear assignment problems and extensions. *Handbook of Combinatorial Optimization Vol.4 (D.-Z. Du and P.M. Pardalos, eds.) Dordrecht: Kluwer Academic Publishers*, 1999.

[4] David E. Goldberg. Simple genetic algorithms and the minimal, deceptive problem. In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 74–88. Morgan Kaufmann Publishers, 1987.

[5] John H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

[6] R.M. Karp. *Compexity of Computer Computations*, chapter Reducibility among combinatorial problems, pages 85–103. Miller, R.E. and Thatcher, J.W. (Eds.). Plenum Press, New York, 1972.

[7] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

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

[9] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[10] İsmail H. Toroslu and Göktürk Üçoluk. Incremental assignment problem. *Inf. Sci.*, 177(6):1523–1529, 2007.