View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Jurnal Online Informatika

JURNAL ONLINE INFORMATIKA
JOIN (Jurnal Online Informatika)
Volume 6 No.1 | June 2021: 118-127
DOI: 10.15575/join.v6i1.692
p-ISSN: 2528-1682
e-ISSN: 2527-9165

# A Fast Dynamic Assignment Algorithm for Solving Resource Allocation Problems

**Ivanda Zevi Amalia[1], Ahmad Saikhu[2], Rully Soelaiman[3]**
[1,2,3]Department of Informatics Engineering, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia

**Article Info**

***Article history:***

Received January 04, 2020
Revised February 05, 2021
Accepted February 22, 2021
Published June 17, 2021

***Keywords:***

Bipartite graph
Dynamic assignment algorithm
Feasible node-weighting
Hungarian algorithm
Resource allocation problem

**ABSTRACT**

The assignment problem is one of the fundamental problems in the field of combinatorial optimization. The Hungarian algorithm can be developed to solve various assignment problems according to each criterion. The assignment problem that is solved in this paper is a dynamic assignment to find the maximum weight on the resource allocation problems. The dynamic characteristic lies in the weight change that can occur after the optimal solution is obtained. The Hungarian algorithm can be used directly, but the initialization process must be done from the beginning every time a change occurs. The solution becomes ineffective because it takes up a lot of time and memory. This paper proposed a fast dynamic assignment algorithm based on the Hungarian algorithm. The proposed algorithm is able to obtain an optimal solution without performing the initialization process from the beginning. Based on the test results, the proposed algorithm has an average time of 0.146 s and an average memory of 4.62 M. While the Hungarian algorithm has an average time of 2.806 s and an average memory of 4.65 M. The fast dynamic assignment algorithm is influenced linearly by the number of change operations and quadratically by the number of vertices.

***Corresponding Author:***

Ivanda Zevi Amalia,
Department of Informatics Engineering,
Institut Teknologi Sepuluh Nopember,
Jl. Teknik Kimia - Gedung Departemen Teknik Informatika, Surabaya, Indonesia
Email: zevi16@mhs.if.its.ac.id

## 1. INTRODUCTION

The assignment problem is one of the fundamental problems in the field of combinatorial optimization [1–3]. The assignment problem is also commonly known as the classic scheduling problem, where there are $n$ workers assigned to complete $m$ jobs [4, 5]. Many problems in the real world applied the assignment problem, ranging from personnel scheduling [6–8], train scheduling [5], workforce planning [4], smart parking system [9], robotic assignment system [1, 10, 11], cloud computing [12], weapon target assignment [13, 14] and many others. The main objective of the assignment problem is to find an optimal and efficient pair of workers and jobs [15]. Because each worker has different efficiencies or abilities in completing work, it is necessary to obtain the most optimal and efficient assignment plan [4].

Vinchoo et al. (2017) conducted a comparative analysis of the five approaches used to solve the assignment problem. Results from this comparison is Hungarian algorithm is the best approach to solve the assignment problem [16]. There are several previous studies that modified the Hungarian algorithm to solve assignment problems according to their respective criteria. Li et al. (2016) develop Hungarian algorithm to solve serial-parallel system assignment problems [4].

Rabbani et al. (2019), Iampang et al. (2010), dan Wang et al. (2018) modified the Hungarian algorithm to solve the unbalanced assignment problem. Rabbani et al. (2019) proposed a modification of the Hungarian algorithm to solve the unbalanced assignment problem without assigning some work to the dummy machine [17]. Iampang et al. (2010) also conducts research stages similar to Rabbani et al. (2019), except that the proposed modification Hungarian algorithm still uses a dummy machine [15]. Wang et al. (2018) developed a Hungarian algorithm to dynamically assign positions to robots and to form the desired formation [1]. Based on several previous studies that discuss assignment problems, it can be concluded that the Hungarian algorithm can be modified to solve various assignment problems according to each criterion.

The assignment problem that is solved in this research is a dynamic assignment to find the maximum weight on resource allocation problems. The dynamic characteristic lies in the change in weight or assignment cost that can occur after the optimal solution is obtained. One possible solution is to perform calculations repeatedly every time the weight changes by using the Hungarian algorithm. This will cause problems if the weight changes are done too often. So that the calculation process will also be repeated and become ineffective because it will take a lot of time. Therefore, we need an algorithm that can solve dynamic assignment problems without doing calculations from scratch.

This paper proposed a fast dynamic assignment algorithm based on the Hungarian algorithm. The proposed dynamic assignment algorithm represents the assignment problem in the form of a bipartite graph. The basic idea of the proposed dynamic assignment algorithm is to maintain the feasible node-weighting value of the previous calculations when the weight changes, both the calculation results of the Hungarian algorithm and the results of the calculation of the proposed dynamic assignment algorithm. The proposed dynamic assignment algorithm can solve the dynamic assignment problems in this study optimally both in terms of time and memory used because there is no need to perform calculations from the beginning.

## 2. METHOD

The following are some of the theories and methodologies used in this research:

### 2.1. Assignment Problem

In general, the assignment problem can be stated in the following points [18, 19]:

(1) There are $n$ workers and $n$ jobs,
(2) Each worker will do one job and each job will be done by one worker,
(3) There is a weight for each worker in doing each job,
(4) The goal is how to get $n$ optimal work arrangements.

The assignment problem can be represented as a linear programming model. Suppose for assigning worker $i$ to job $j$ need a weight $w_{ij}$, and defined

$$x_{ij} = \begin{cases} 1, & \text{if worker } i \text{ is assigned to job } j \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The modeling of the assignment problem will be as follows

$$Maximize\ z = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}\, x_{ij} \quad (2)$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1, i = 1, 2, \dots, n \quad (3)$$

$$\sum_{i=1}^{n} x_{ij} = 1, j = 1, 2, \dots, n \quad (4)$$

$$x_{ij} = 1\ or\ 0 \quad (5)$$

The optimal value in this case can be the maximum value or the minimum value depending on the needs. However, in this study the maximum value is taken. This problem can also be modeled into a bipartite graph $G = (S \cup T, E)$ [7]. The set of vertices $S$ can be assumed as workers and $T$ as jobs. Each edge $(i, j)$ on graph $G$ states that worker $i$ from set $S$ can do job $j$ from set $T$ with weight $W_{ij}$ which represents the weight of worker $i$ in doing work $j$. An example of a bipartite graph that represents an assignment problem can be seen in Figure 1. From Figure 1 it can be seen if the structure of the assignment problem graph is the same as the complete bipartite graph structure shown in Figure 2b.

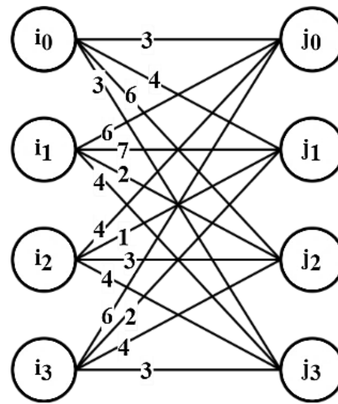


Figure 1. Example of a graph representation of an assignment problem

## 2.2. Bipartite Graph

A graph is called a bipartite if the set of vertices can be divided into two subsets S and T so that each edge has one endpoint at S and one endpoint at T. So that in each set there are no neighboring vertices [20]. Meanwhile, a complete bipartite graph is a bipartite graph where each vertex in one set is adjacent to all vertices in the other set [21]. An example of a bipartite graph can be seen in Figure 2.

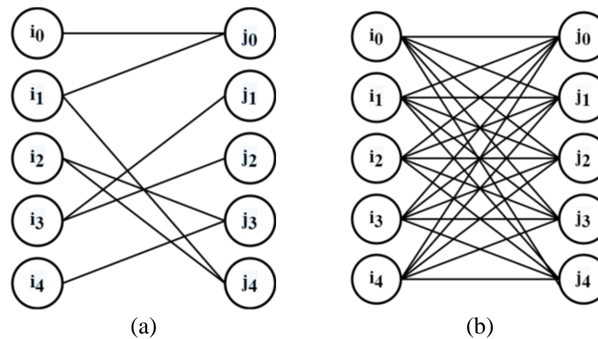


Figure 2. Example of a bipartite graph: (a) standard bipartite graph and (b) complete bipartite graph

## 2.3. Matching

If there is a graph $G = (V, E)$ which has a set of vertices $V$ and a set of edges $E$. The graph is an undirected graph and each edge $e \in E$ has a weight $w_e$. A matching on graph $G = (V, E)$ is a set of edges $M \subseteq E$ where no edges touch each other in $M$. Meanwhile, perfect matching on graph $G = (V, E)$ is a matching $M$ where each vertex in $V$ is incident to exactly one edge on $M$. The augmenting path is an alternating path in matching $M$ that starts and ends from an unmatched vertex. Maximum-size matching on graph $G$ is matching $M$ which has the biggest $|M|$. Maximum-weight matching on graph $G$ is matching $M$ which has the largest total weight [22]. There are three pictures in Figure 3 where the red lines indicate that the edges are included in the matching.

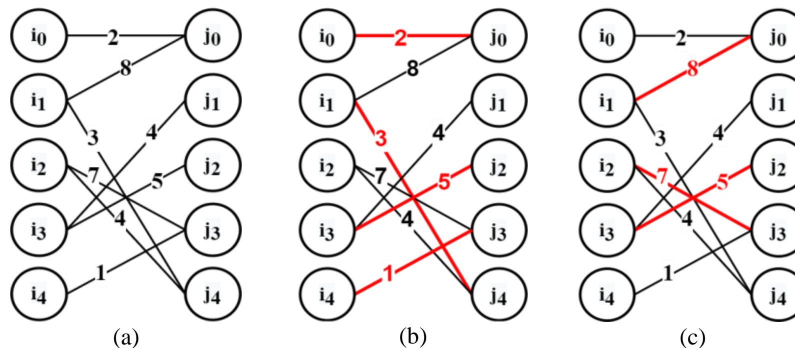


Figure 3. Difference between maximum-size matching and maximum-weight matching: (a) weighted bipartite graph (b) max-size matching (c) max-weight matching

## 2.4. Bipartite Matching

Bipartite matching is a matching that is contained in a bipartite graph. Complete matching of graph $G = (S \cup T, E)$ is a matching which has cardinality of $min\{|S|, |T|\}$. A regular bipartite graph is a graph $G =$

$(S \cup T, E)$ where each vertex on the graph has the same degree. $k$-regular bipartite graph means that each vertex on the graph has $k$ degrees [21].

### 2.5. Dynamic Assignment Algorithm

When optimal matching of a bipartite graph has been obtained, the problem does not stop at that moment. A new problem arises when there is a change in some assignment weights on a bipartite graph from modeling the assignment problem. In a possible system where changes often occur especially in a relatively short period of time, the calculation process must be carried out as quickly as possible. This problem is known as the dynamic assignment problem.

On the online assessment site SPOJ (Sphere Online Judge) there is a question that underlies this research. The problem's name is Dynamic Assignment Problem which has the question number 12749 and problem code DAP on the SPOJ [23]. On the problem given an integer $n$ which is the size of matrix $n \times n$. After that, the weights of the $n \times n$ matrix are given. Then given an integer $m$ which is the number of operations. After that it is given m operations which must be implemented as follows:

#### 2.5.1. Update row ($X\ i\ x_0\ x_1\ x_2\ \dots\ x_{N-1}$)

The first operation is that all the weights in the $i$-th row of the matrix are replaced with the input given, which is replaced by $x_0$ to $x_{N-1}$. This operation represents the type of update row.

#### 2.5.2. Update column ($Y\ i\ y_0\ y_1\ y_2\ \dots\ y_{N-1}$)

The second operation is that all weights in the $i$-th column of the matrix are replaced with the input given, which is replaced by $y_0$ to $y_{N-1}$. This operation represents the type of update column.

#### 2.5.3. Update cell ($C\ i\ j\ w$)

The third operation is that the weight value in the matrix $i$-th row, $j$-th column is changed to $w$. It can be said that this operation represents the type of update cell.

#### 2.5.4. Add vertex ($A$)

The fourth operation is that the size of the weight matrix is increased by 1 become $N + 1$. The weights of a $(N + 1)$-th row and a $(N + 1)$-th column of the matrix are all set to 0. It can be said that this operation represents changing the type of add vertex.

#### 2.5.5. Query operation ($Q$)

The program is asked to give the optimal matching weights from the current graph. It can be said that this operation represents the type of query operation.

### 2.6. Hungarian Algorithm

Hungarian algorithm is an algorithm that has been known to solve the problems of the assignment properly and efficiently [24, 25]. It is necessary to modify the Hungarian algorithm to solve the dynamic assignment problem. The dynamic assignment problem that solved in this research is to find the maximum-weight matching on the complete bipartite graph $G = (S \cup T, E)$ where $|S| = |T|$.

For example, there is a complete bipartite graph $G = (S \cup T, E)$ where $S = \{1,2,\dots,n\}$ and $T = \{1', 2', \dots, n'\}$ and the weight function $W = (w_{ij})$. $w_{ij}$ is the weight of the edge $(i, j')$ and it is assumed that $w_{ij}$ is always non-negative, then the Hungarian algorithm can find optimal matching on $G$ with complexity $O(n^3)$. In this case $n = |S| = |T|$. Two vectors $< u, v >$ where $u = (u_1, u_2, \dots u_n)$ and $v = (v_1, v_2, \dots v_n)$ are feasible node-weighting if they satisfy Eq. (6) [22]:

$$u_i + v_j \geq w_{ij} \text{ for all } i, j = 1 \dots n \tag{6}$$

For example, optimal matching on $G$ has a total weight of $D$, it can be seen that:

$$\mathrm{w(M)} \leq \mathrm{D} \leq \sum_{i=1}^{n} u_i + v_i \tag{7}$$

In Eq. (7), $w(M)$ is the sum of the weights of any perfect matching $M$ in $G$. If a perfect matching $M$ can be found where equality occurs in Eq. (7), then $M$ is optimal matching and the condition must be found.

### 2.7. Fast Dynamic Assignment Algorithm

The fast dynamic assignment algorithm is an algorithm proposed in this research to solve dynamic resource allocation problems. This algorithm represents the assignment problem into a bipartite graph. The algorithm can find the optimal matching value on a bipartite graph that changes in weight without doing any calculations from the beginning. The basic idea of the proposed dynamic assignment algorithm is to maintain

the feasible node-weighting value of the previous calculations when the weight changes, both the calculation results of the Hungarian algorithm and the results of the calculation of the proposed dynamic assignment algorithm. An explanation of the value of feasible node-weighting has been discussed in the Eq. (6).

It is necessary to adjust the feasible node-weighting value to the weight change on a bipartite graph so that the Hungarian algorithm can then be run without having to find the feasible node-weighting from the beginning. It can be said that the proposed algorithm is the development of the Hungarian algorithm. This is because the way the proposed dynamic assignment algorithm works is to change some of the results of the Hungarian algorithm that have been run according to the weight changes that occur. This study will focus on discussing the assignment problem for dynamic resource allocation in obtaining the maximum-weight matching.

There are four kinds of weight changes. Complete bipartite graph $G = (S \cup T, E)$ is defined first. In this case $n = |S| = |T|$ where $S = \{1,2,\ldots,n\}$ and $T = \{1',2',\ldots,n'\}$ and the weight function $W = (w_{ij})$ where $w_{ij}$ is the weight of the edge $(i,j')$ where $i \in S$ and $j' \in T$ and it is assumed that $w_{ij}$ is always non-negative. There is also a feasible node-weighting $< u,v >$ , then a Hungarian algorithm is run which results in optimal matching $M$ and a new $< u,v >$ value is obtained. The four changes in this research are:

#### 2.7.1. Changes on the weight of edge $(i,j')$ for all $j' \in T$

In this case, there is a vertex i ∈ S where all edges that intersect with it changed the value of its weight. From $< u,v >$ the results of the previous Hungarian algorithm, the first adjustment that must be made is to remove edge $(i,j')$ from optimal matching $M$ because there must be a possibility that the optimal matching arrangement will change. The next adjustment is to change the $u_i$ value. The $u_i$ value is converted into the following Eq. (8):

$$u_i = max(u_i, (w_{ij} - v_j)) \text{ for all } j = 1,\ldots,n \tag{8}$$

This is done to maintain the feasible node-weighting value as described in Eq. (6) by finding the $u_i$ value so that $u_i + v_j \geq w_{ij}$ for all $j$. This is because $u_i \geq w_{ij} - v_j$ for all $j$ is in accordance with that obtained from Eq. (8).

#### 2.7.2. Changes on the weight of edge $(i,j')$ for all $i \in S$

In this case, there is a vertex j' ∈ T where all the edges that intersect with it are changed by weight. From the $< u,v >$ results of the previous Hungarian algorithm, the first adjustment that must be made is if vertex $j'$ is not pair of $i$ for optimal matching $M$, then there is a possibility that the optimal matching will change. Therefore, it is necessary to remove edge $(i,j')$ from optimal matching $M$. The next adjustment is to change the $v_j$ value. The value of $v_j$ is converted into the following Eq. (9):

$$v_j = max(v_j, (w_{ij} - u_i)) \text{ for all } i = 1,\ldots,n \tag{9}$$

This is done to maintain the feasible node-weighting value as described in Eq. (6) by finding the $v_j$ value so that $u_i + v_j \geq w_{ij}$ for all $i$. This is because $v_j \geq w_{ij} - u_i$ for all $i$ is in accordance with that obtained from Eq. (9).

#### 2.7.3. Changes in the weight value of an edge $(i,j')$

In this case, there is an edge $(i,j')$ i ∈ S and j' ∈ T whose weight is changed from $w_{ij}$ to $w'_{ij}$. The first adjustment that must be done is to remove the edge $(i,j')$ from the optimal matching $M$ because there must be a possibility that the optimal matching arrangement will change. After that, one of the $u_i$ or $v_j$ values need to be adjusted so that the feasible condition of the node-weighting is maintained. Adjust one the value of $u_i$ or $v_j$. If only the value of $u_i$ is adjusted, the value of $u_i$ will be like in Eq. (8). Or if only the value of $v_j$ is adjusted, the value of $v_j$ will be like in Eq. (9).

#### 2.7.4. The addition of two vertices, one vertex in $S$ and one vertex in $T$

In this case, two vertices are added to the graph $G$. One vertex is added to $S$ and one vertex in $T$. For example, the vertex added to $S$ is vertex $p$ and vertex $q$ on $T$, then we also add some edges connecting $p$ to all vertices in $T$ and some edges connecting $q$ to all vertices on $S$ and one edge connecting $p$ to $q$. In this case, the weights of all the newly added edges are assigned a value of 0. In addition, $u_p$ or $v_q$ are added to $< u,v >$ with the value $u_p$ equal to the following Eq. (10):

$$u_p = \max(u_p, (w_{pj} - v_j)) \text{ for all } j = 1\ldots n \tag{10}$$

and $v_q$ is equal to the following Eq. (11):

$$v_q = max(v_q, (w_{iq} - u_i)) \text{ for all } i = 1...n \quad (11)$$

The assignment of $u_p$ or $v_q$ values as written in Eq. (10) and Eq. (11) is to adjust the feasible node-weighting value. After the adjustment process is carried out according to the type of change that has occurred, it is necessary to run a Hungarian algorithm to obtain a new optimal matching. It should be noted that in this case the Hungarian algorithm is no longer running from scratch. The process of finding optimal matching with only one pair of vertices that have not entered into optimal matching must be much faster than the process of finding optimal matching from the initial state. Moreover, the feasible node-weighting has been adjusted so that the value is maintained in accordance with the matching $M$ currently formed.

## 3. RESULTS AND DISCUSSION

The following are the results of the implementation and experiments that have been carried out in this study:

### 3.1. Implementation of the Proposed Algorithm

There are four types of weight changes have been discussed. Each change has its own adjustments. The following is a discussion of the algorithms for each change:

#### 3.1.1. Update row

Changes of the weight on edge $(i, j')$ for all $j' \in T$ shortened to **update row**. Figure 4 is an algorithm of the UpdateRow function. The UpdateRow function will run when the program receives input according to the format described in section 2.5.1. The mathematical explanation and flow of the UpdateRow function algorithm follow the steps described in section 2.7.1. Line 1 of the program stores the input of the row index to be changed. Then line 2 until line 4 is the process of changing the weight. Line 5 until line 6 is the elimination of matching edges that intersect with the $r$-th vertex in $S$. Line 8 until line 10 is the process of feasible node-weighting adjustment of the $u$ value according to Eq. (8).

```
Input : row          : the index of the row to be changed
        nweight[] : new weight
Output : -
1.  r ← row
2.  for j = 0 to n do
3.      w[r][j] ← nweight[j]
4.  od
5.  mateT[ mateS[r] ] ← -1
6.  mateS[r] ← -1
7.  u[r] ← 0
8.  for j = 0 to n do
9.      u[r] ← max(u[r], w[r][j] – v[j])
10. od
```

Figure 4. Algorithm of UpdateRow function

#### 3.1.2. Update column

```
Input : col          : the index of the column to be changed
        nweight[] : new weight
Output : -
1.  c ← col
2.  for i = 1 to n do
3.      w[i][c] ← nweight[i]
4.  od
5.  if mateT[c] != -1
6.      mateS[ mateT[c] ] ← -1
7.      mateT[c] ← -1
8.  v[c] ← 0
9.  for i = 1 to n do
10.     v[c] ← max(v[c], w[i][c] – u[i])
11. od
```

Figure 5. Algorithm of UpdateColumn function

Changes in the value of the weight on edge $(i, j')$ for all $i \in S$ shortened to **update column**. Figure 5 is an algorithm of the UpdateColumn function. The UpdateColumn function will run when the program receives input according to the format described in section 2.5.2. The mathematical explanation and flow of the UpdateColumn function algorithm follow the steps described in section 2.7.2. Line 1 of the program stores the

input of the column index to be changed. Then line 2 until line 4 is the process of changing the weight. Line 5 until line 7 is the elimination of matching edges that intersect with the $c$-th vertex in $T$. Line 9 until line 11 is the process of feasible node-weighting adjustment of the $v$ value according to Eq. (9).

### 3.1.3. Update cell

Changes in the weight value of an edge $(i, j')$ are shortened to **update cell**. Figure 6 is an algorithm of the UpdateCell function. The UpdateCell function will run when the program receives input according to the format described in section 2.5.3. The mathematical explanation and flow of the UpdateCell function algorithm follow the steps described in section 2.7.3. Line 1 of the program stores the input of the row index to be changed. Line 2 of the program stores the input of the column index to be changed. Line 3 is the process of changing the weight. Line 4 until line 5 is the process of deleting edges when changes occur in the edges that are in optimal matching. Line 7 until line 9 is a process for adjusting the feasible node-weighting $u$ so that the $< u, v >$ values are maintained. Since the adjusted one is the feasible node-weighting $u$, the adjustment will follow Eq. (8).

```
Input : row     : the index of the row to be changed
        col     : the index of the column to be changed
        nweight : new weight of w[r][c]
Output : -
1. r ← row
2. c ← col
3. w[r][c] ← nweight
4. mateT[ mateS[r] ] ← -1
5. mateS[r] ← -1
6. u[r] ← 0
7. for i = 1 to n do
8.     u[r] ← max(u[r], w[r][i] – v[i])
9. od
```

Figure 6. Algorithm of UpdateCell function

### 3.1.4. Add vertex

The addition of two vertices each on $S$ and $T$ is shortened to **add vertex**. Figure 7 is an algorithm of the AddVertex function. The AddVertex function will run when the program receives input according to the format described in section 2.5.4. The mathematical explanation and flow of the AddVertex function algorithm follow the steps described in section 2.7.4. Line 1 of the program adds the number of vertices. Line 3 until line 5 is a feasible node-weighting adjustment process. As explained in Section 2.7.4, when the vertex is added, the value $u[n]$ or the value $v[n]$ must be adjusted. Since the adjusted one is the feasible node-weighting $v$, the adjustment will follow Eq. (9).

```
Input : -
Output : -
1. n ← n + 1
2. v[n] ← 0
3. for i = 1 to n do
4.     v[n] ← max(v[n], w[i][n] – u[i])
5. od
```

Figure 7. Algorithm of AddVertex function

The four types of changes to the dynamic assignment problems that were resolved in this study have been described above. Now we will discuss the fast dynamic assignment algorithm. Figure 8 contains the general stages of the fast dynamic assignment algorithm. Line 2 until line 10 represents the four change scenarios that are resolved by the proposed fast dynamic assignment algorithm. Each change scenario has different stages of completion as described in Figure 4 until Figure 7. What needs to be considered from the running of this algorithm is the process that occurs in row 12. The Hungarian algorithm is run again without having to initialize the feasible node-weighting value and the matching arrangement because both were obtained from the previous Hungarian algorithm. Of course, the Hungarian algorithm running in row 12 runs $k$ phases where $k$ is the number of edges that have been removed from optimal matching due to previous changes.

e-ISSN 2528-1682
p-ISSN 2527-9165

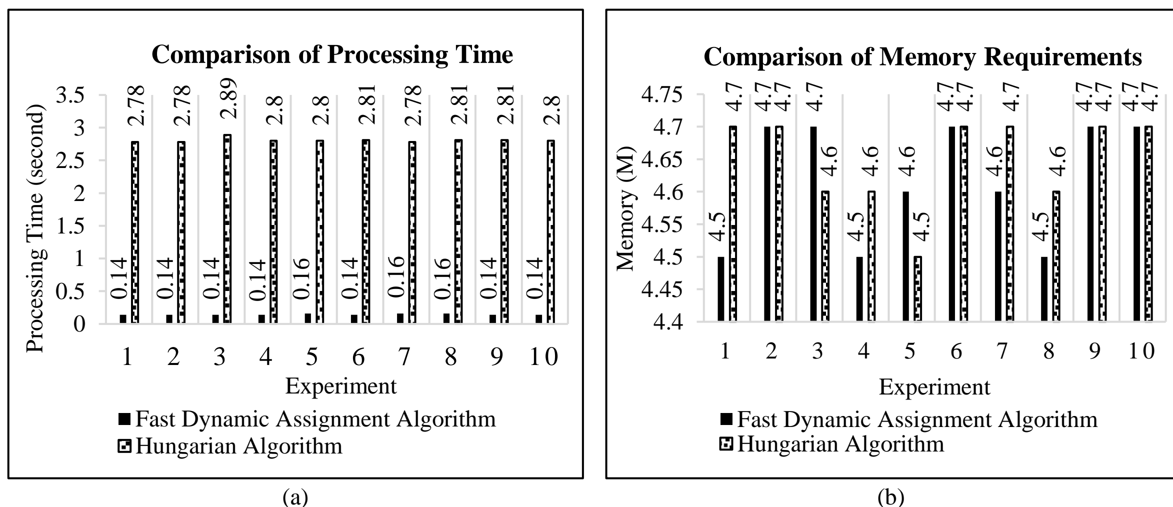
```
Input : n          : the number of vertices
        w[n][n] : weight of bipartite graph
Output :  total weight of optimal matching when prompted
1.   HUNGARIAN
2.   while there are changes do
3.       if kind of change = update row
4.           UpdateRow
5.       else if kind of change = update column
6.           UpdateColumn
7.       else if kind of change = update cell
8.           UpdateCell
9.       else if kind of change = add vertex
10.          AddVertex
11.      else asked the total weight of the current optimal
         matching
12.          Run HUNGARIAN function without doing the
             first line process
13.          Return the optimal matching weight for the above
             process
14.      fi
15.  od
```

Figure 8. Fast dynamic assignment algorithm

### 3.2. Experimental Results

There are several test scenarios carried out. The first scenario is to send the source code of the fast dynamic assignment algorithm to the SPOJ online assessment site on a question entitled Dynamic Assignment Problem [23]. Results of sending the source code to SPOJ produce an output "Accepted" which means the source code successfully completed all test cases provided. The time it takes for the source code to solve this problem is 0.14 s, and requires 4.5 M of memory. The source code that was created managed to get the first rank out of a total only 8 people who managed to solve this problem. The ranking can be seen in this link https://www.spoj.com/ranks/DAP/.

The second scenario is to measure the time and memory ratio between the use of the Hungarian algorithm and the fast dynamic assignment algorithm. To get the time and memory comparison, the source code of the Hungarian algorithm and the fast dynamic assignment algorithm were sent 10 times each on the SPOJ online assessment site on the question entitled Dynamic Assignment Problem. Then the average time and memory are calculated from the results of each algorithm. The comparison can be seen in graph Figure 9(a). While the comparison of the required memory can be seen in graph Figure 9(b).




(a) (b)

Figure 9. Comparison between the fast dynamic assignment algorithm and *Hungarian algorithm* : (a) Comparison of Processing Time (b) Comparison of Memory Requirements

From the graph in Figure 9(a), it can be seen that the average time needed for the fast dynamic assignment algorithm is 0.146 s, while the Hungarian algorithm is 2.806 s. The fast dynamic assignment algorithm has a much better processing time than the Hungarian algorithm. This is because the Hungarian algorithm has to repeat the initialization process from the beginning, while the fast dynamic assignment algorithm uses the

results from previous calculations by maintaining the feasible node-weighting value. From Figure 9(b) it can be seen that the average memory required by the fast dynamic assignment algorithm is slightly better than the Hungarian algorithm. The average memory required for the fast dynamic assignment algorithm is 4.62 M, and the Hungarian algorithm is 4.65 M.

The third trial scenario is to use test case data from the output of the test case generator program. The test case generator program generates data randomly according to the given parameters and the data will be stored in a file. These parameters are the number of vertices, the number of operations, the interval for which the query occurs, and the interval for adding vertices. The program can also calculate the time required to complete each test case. Check and analyze whether the number of vertices and the number of operations affect the performance of the fast dynamic assignment algorithm.

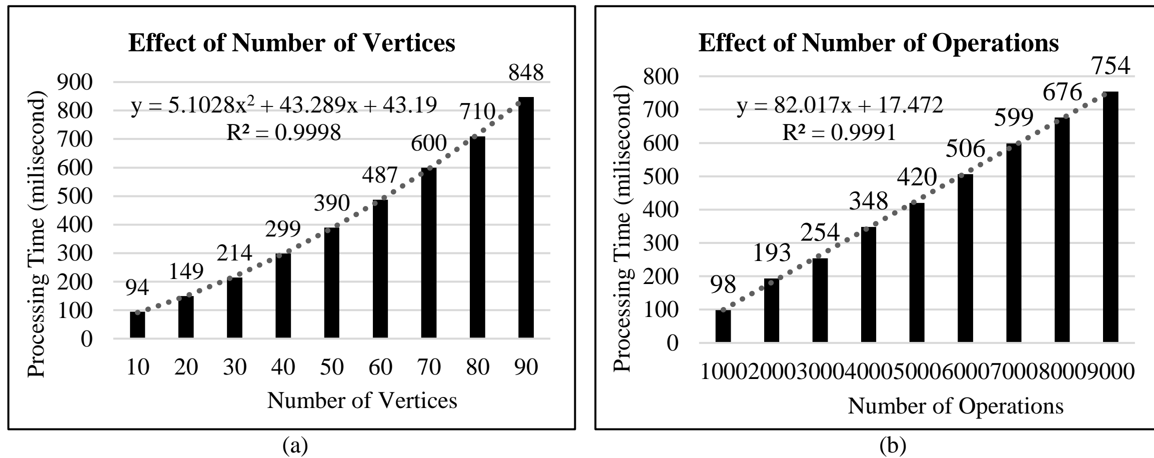



Figure 10. The effect of the number of vertices and the number of operations on the fast dynamic assignment algorithm performance

In this experiment, the number of vertices varied between 10 and 90 with a range of 10 vertices. The number of operations is defined as 10000 operations, where each operation multiples of 10 is a query operation. The 999 multiples operation is an add vertex operation. Then the program execution time is recorded for each data in milliseconds so that the effect of many vertices on program execution time can be observed. The trial results of these experiments can be seen in the graph in Figure 10(a). The graph in Figure 10(a) tends to approach the quadratic curve. This is consistent with the complexity of the fast dynamic assignment algorithm which is influenced by the number of vertices quadratically.

At the experiment of the effect of number of operations, the number of operations made varies between 1000 to 10000 with a range of 1000 operations. The number of vertices is determined to be 90 vertices. Also specified that each operation of 10 is a query operation. Every 999 multiple operations is an add vertex operation. Then the program execution time is recorded for each data in milliseconds so that the effect of many operations on program execution time can be observed. The trial results of these experiments can be seen in the graph in Figure 10(b). The graph in Figure 10(b) tends to approach a linear curve. This is consistent with the complexity of the fast dynamic assignment algorithm which is influenced by the number of operations linearly.

## 4. CONCLUSION

From the results of experiments that have been carried out, several things can be drawn from the performance of the fast dynamic assignment algorithm. The fast dynamic assignment algorithm has a much better time average than the Hungarian algorithm in solving dynamic assignment problems on resource allocation. From the test results, the proposed algorithm has an average time of 0.146 s, while the Hungarian algorithm is 2.806 s. While the average memory required by the fast dynamic assignment algorithm is slightly better than the Hungarian algorithm. From the test results, the proposed algorithm has an average memory of 4.62 M, and the Hungarian algorithm is 4.65 M. The performance of the fast dynamic assignment algorithm is influenced linearly by the number of change operations and quadratically by the number of vertices.

This research has been able to propose an algorithm that can reduce processing time and memory requirements to solve dynamic assignment problems. However, the memory requirements have not changed much so that further research can make an increase in terms of memory needs.

**ACKNOWLEDGEMENTS**

The authors would like to thank Institut Teknologi Sepuluh Nopember Surabaya and Indonesian Ministry of Research, Technology, and Higher Education for supporting this research. The author also appreciates comments or suggestions from reviewers.

## 5. REFERENCES

[1] H. Wang, D. Shi, and B. Song, "A dynamic role assignment formation control algorithm based on hungarian method," in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovations*, 2018, pp. 687–696.
[2] S. Ding and X. J. Zeng, "Uncertain random assignment problem," *Appl. Math. Model.*, Vol. 56, pp. 96–104, 2018.
[3] P. Kaur, A. Sharma, V. Verma, and K. Dahiya, "A priority based assignment problem," *Appl. Math. Model.*, Vol. 40, No. 17–18, pp. 7784–7795, 2016.
[4] T. Li, Y. Li, and Y. Qian, "Improved Hungarian algorithm for assignment problems of serial-parallel systems," *J. Syst. Eng. Electron.*, Vol. 27, No. 4, pp. 858–870, 2016.
[5] F. F. Pashchenko, N. A. Kuznetsov, N. G. Ryabykh, I. K. Minashina, E. M. Zakharova, and O. A. Tsvetkova, "Implementation of train scheduling system in rail transport using assignment problem solution," in *6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks, EUSPN-2015*, 2015, Vol. 63, pp. 154–158.
[6] T. Öncan, Z. Şuvak, M. H. Akyüz, and K. Altınel, "Assignment problem with conflicts," *Comput. Oper. Res.*, Vol. 111, pp. 214–229, 2019.
[7] I. H. Toroslu and Y. Arslanoglu, "Genetic algorithm for the personnel assignment problem with multiple objectives," *Inf. Sci. (Ny).*, Vol. 177, No. 3, pp. 787–803, 2007.
[8] S. Lan, W. Fan, T. Liu, and S. Yang, "A hybrid SCA–VNS meta-heuristic based on Iterated Hungarian algorithm for physicians and medical staff scheduling problem in outpatient department of large hospitals with multiple branches," *Appl. Soft Comput. J.*, Vol. 85, p. 105813, 2019.
[9] M. Ratli, A. A. El Cadi, B. Jarboui, and A. Artiba, "Dynamic assignment problem of parking slots," *Proc. 2019 Int. Conf. Ind. Eng. Syst. Manag. IESM 2019*, pp. 1–6, 2019.
[10] S. Chopra, G. Notarstefano, M. Rice, and M. Egerstedt, "A Distributed Version of the Hungarian Method for Multirobot Assignment," *IEEE Trans. Robot.*, Vol. 33, No. 4, pp. 932–947, 2017.
[11] S. Marangoz, M. F. Amasyalı, E. Uslu, F. Çakmak, N. Altuntaş, and S. Yavuz, "More scalable solution for multi-robot–multi-target assignment problem," *Rob. Auton. Syst.*, Vol. 113, pp. 174–185, 2019.
[12] R. R. Patel, T. T. Desai, and S. J. Patel, "Scheduling of jobs based on Hungarian method in cloud computing," in *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, 2017, pp. 6–9.
[13] A. Kline, D. Ahner, and R. Hill, "The Weapon-Target Assignment Problem," *Comput. Oper. Res.*, Vol. 105, pp. 226–236, 2019.
[14] C. Leboucher *et al.*, "Novel evolutionary game based multi-objective optimisation for dynamic weapon target assignment," in *Proceedings of the 19th World Congress The International Federation of Automatic Control*, 2014, Vol. 47, No. 3, pp. 3936–3941.
[15] A. Iampang, V. Boonjing, and P. Chanvarasuth, "A cost and space efficient method for unbalanced assignment problems," in *2010 IEEE International Conference on Industrial Engineering and Engineering Management*, 2010, pp. 985–988.
[16] R. V. Vinchoo, Mohit Manoj; Deolekar, "Comparative analysis of different approaches to solve the job assignment problem," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, 2017, pp. 129–134.
[17] Q. Rabbani, A. Khan, and A. Quddoos, "Modified Hungarian method for unbalanced assignment problem with multiple jobs," *Appl. Math. Comput.*, Vol. 361, pp. 493–498, 2019.
[18] H. A. Taha, *Operations Research an Introduction Eighth Edition*. United States of America: Pearson Education, 2007.
[19] D. Gurukumaresan, C. Duraisamy, R. Srinivasan, and V. Vijayan, "Optimal solution of fuzzy assignment problem with centroid methods," *Mater. Today Proc.*, pp. 5–7, 2020.
[20] A. Niv, M. MacCaig, and S. Sergeev, "Optimal assignments with supervisions," *Linear Algebra Appl.*, Vol. 595, pp. 72–100, 2020.
[21] J. A. . Bondy and U. S. R. . Murty, *Graph Theory, 3rd edition*. Berlin, Germany: Springer, 2008.
[22] D. Jungnickel, *Graphs, Networks and Algorithms, 4th edition*. New York: Springer Heidelberg, 2012.
[23] SPOJ, "Dynamic Assignment Problem," 2012. http://www.spoj.com/problems/DAP/ (accessed Aug. 23, 2020).
[24] E. Malaguti and R. Medina Durán, "Computing k different solutions to the assignment problem," *Comput. Ind. Eng.*, Vol. 135, pp. 528–536, 2019.
[25] K. Date and R. Nagi, "GPU-accelerated Hungarian algorithms for the Linear Assignment Problem," *Parallel Comput.*, Vol. 57, pp. 52–72, 2016.