Optimal Project Assignment Using Kuhn's Algorithm for Maximum Bipartite Matching

Adril Putra Merin - 13522068
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522068@std.stei.itb.ac.id

***Abstract*— This paper explores the application of Kuhn's algorithm to solve the project assignment problem. The project assignment problem aims to optimally allocate a set of people in a group to a set of projects based on their skills and project requirements. By modeling the problem as a bipartite graph, where one set represents people and the other set represents projects, we can leverage Kuhn's algorithm to find the maximum matching that maximizes the overall efficiency of assignments. This approach ensures that each person is assigned to zero or one project, and each project receives zero or one person. The efficiency of the algorithm is demonstrated through various experiments, showcasing its potential to significantly enhance project management processes. This also indicates that Kuhn's algorithm provides a robust and efficient solution, outperforming brute force assignment methods.**

Keywords—Depth First Search; DFS; Maximum Bipartite Matching; Project Assignment; Kuhn's Algorithm; Graph

I. Introduction

Effective task assignment is one the main challenges in project management and resource allocation within an organization. Ensuring that each person is assigned to a project that matches their skills and capabilities is crucial for achieving maximum efficiency and optimal outcomes. However, this process is often complex and requires robust algorithms to optimize the assignments.

In the context of the task assignment problem, we can model it as a bipartite graph, where one set represents people, and the other set represents tasks. Each person and project can be represented as a vertex and be connected if a person's skill meets the project requirement. The objective of maximum bipartite matching is to find a matching that maximizes the number of assignments, ensuring that each person is assigned to zero or one project, and each project receives zero or one person.

One of the most straightforward methods to solve maximum bipartite matching is by using the brute force algorithm. However, the following sections of this paper will show that this method is not the best way to solve this problem because, as the name suggests, the brute force algorithm does not consider the heuristics and details of the problem; instead, it tries all possible answers to get the optimal solution.

Another way to get the optimal project assignment is by using Kuhn's algorithm which offers an efficient solution for the maximum bipartite matching. The subsequent of this paper will discuss the theoretical background of Kuhn's algorithm and show various experiments to demonstrate the efficiency of this approach.

II. Basic Theory

A. Brute Force

The brute force algorithm is one of the simplest and most straightforward techniques in computer science for solving problems. As the name suggests, it involves systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's requirement without considering the heuristics or details of the problem.

While brute force algorithm is complete – will always find the solution if exists – and easy to implement, the computational costs are proportional to the number of candidate solutions which tends to grow very quickly as the size of the problem increases. Thus, this algorithm is not particularly efficient because it is possible to eliminate many candidate solutions through clever algorithms.

Brute force algorithm is usually used in a problem where the size is limited or in a condition where the simplicity of the implementation is more important than the computational speed. This algorithm can also be used as baseline method when benchmarking other algorithms. The general steps of this algorithm can be defined as follows.

1. Generate All Possible Candidates: List all possible solutions to the problem. This may involve generating permutations, combinations, or other relevant structures.
2. Evaluate Each Candidate: Check each candidate to see if it meets the problem's requirements. This often involves a straightforward comparison or evaluation function.
3. Select the Best Solution: Out of all candidates that meet the requirements, select the one that optimizes the

objective, for example minimum cost, maximum profit, etc.

Because of its generality, the brute force algorithm can be applied to a wide range of problems without many modifications. For example, Travelling Salesman Problem, Password Cracking, String Matching, Subset Sum, Bubble Sort, Selection Sort, etc.

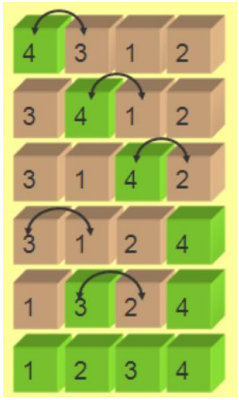

Figure 1. An example of brute force used in bubble sort

B. Graph

Graphs are fundamental data structures used in both mathematics and computer science to represent relationships between objects. They consist of vertices, which represent the objects being modeled, and edges, which represent the relationship between objects.

Based on edge orientation, a graph can be directed or undirected. In a directed graph, edges have a direction, indicating a one-way relationship between vertices. In an undirected graph, edges have no direction, signifying a mutual relationship between vertices.

Moreover, a graph can be weighted or unweighted. In a weighted graph, each edge has a numerical weight or cost associated with it, representing cost of the relationship between vertices. Unweighted graphs have no such weights.

Visually, a graph is usually represented as a collection of dots or circles to denote the vertices, connected by lines or curves to denote the edges. In their implementations, Graphs can be represented in various ways, such as adjacency list and adjacency matrix.

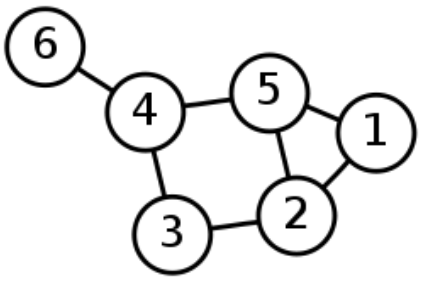

Figure 2. Visual representation of a graph

An adjacency matrix is a 2D array where each cell represents the presence or absence of an edge between two vertices in graph. If there is an edge from vertex i to vertex j, the cell (i, j) contains a non-zero or non-infinity value (usually 1 for an unweighted graph). If there is no edge, the cell contains a zero or infinity (in weighted graphs). For weighted graphs, the cell may contain the weight of the corresponding edges.

An adjacency list is a collection of lists or arrays, where each vertex in the graph has a list of its adjacent vertices. This representation is typically more memory-efficient for sparse graphs since it only stores information about existing edges.

Graphs have plenty of applications such as computer networking, social networks, transportation systems, recommendations system, etc.

C. Depth First Search

Depth First Search is an algorithm for traversing or searching a tree or graph data structure. DFS traverses a graph by starting at a chosen vertex (the start vertex) and exploring as far as possible along each branch before backtracking. It prioritizes exploring the deepest unexplored vertices first. When a dead-end is reached (namely there are no more unvisited vertices adjacent to the current vertex), DFS backtracks to the most recently visited vertex that still has unvisited neighbors and continues exploring from there. The general steps for this algorithm can be described as follows.

1. Begin the traversal by selecting a root vertex in the graph.
2. Visit the root vertex and mark it as visited to avoid revisiting it during traversal.
3. Visit each neighbor of the current vertex that has not been visited yet. If a vertex has multiple unvisited neighbors, choose one and proceed deeper into that neighbor before visiting other neighbors.
4. If all neighbors of the current vertex have been visited, backtrack to the most recently visited vertex with unvisited neighbors and continue the exploration from there.
5. Repeat steps 3 and 4 until all vertices in the graph have been visited.

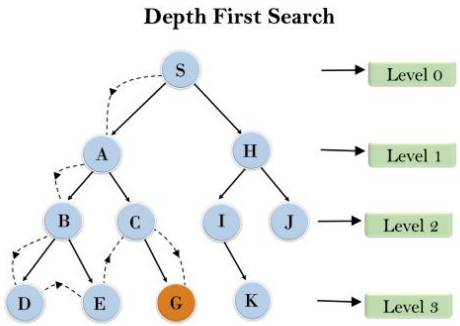

Figure 3. Example of DFS

The time complexity of DFS if $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because DFS visits each vertex and edge at most once.

D. Bipartite Graph

A bipartite graph is a type of graph whose vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent to each other. Formally, a graph $G = (V, E)$ is bipartite if its vertex set V can be partitioned into two sets V_1 and V_2 such that every edge in E connects a vertex in V_1 to a V_2.

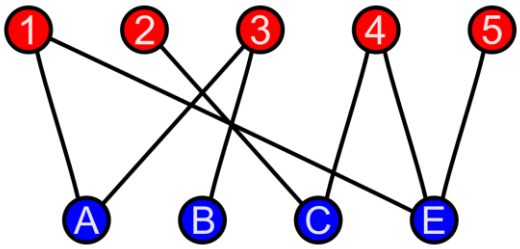

Figure 3. Visual example of a bipartite graph

To put it simply, a bipartite graph is a graph that can be colored using only two colors in such a way that no two adjacent vertices have the same color. This property makes bipartite graphs particularly interesting and useful in various applications.

E. Maximum Bipartite Matching

Maximum bipartite matching is a fundamental problem in graph theory and combinatorial optimization that involves finding the largest possible set of non-intersecting edges in a bipartite graph.

Given a bipartite graph $G = (V, E)$, with vertex sets V_1 and V_2, a matching M is a subset of edges from E such that no two edges in M share a common endpoint. A maximum bipartite matching is a matching with the maximum possible number of edges.

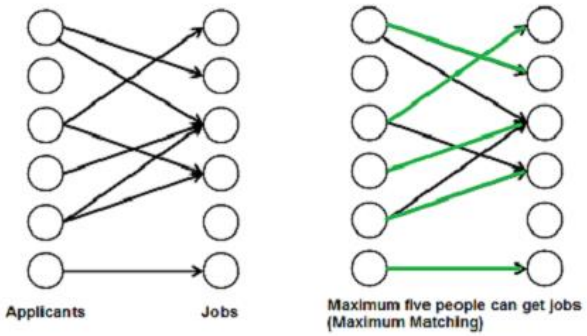

Figure 4. An example of maximum bipartite (source: https://www.geeksforgeeks.org/maximum-bipartite-matching/)

Before proceeding to the next section, these are some important definitions of some terms that will used frequently in following sections.

- **Cardinality of a matching**: number of edges in a matching
- **Saturated vertex**: a vertex that have an adjacent edge from the matching M, namely which have degree exactly one in the subgraph formed by M.
- **Maximal matching**: a matching M of graph G that is not a subset of any other matching
- **Path**: a simple path that does not contain any repeated vertices or edges.
- **Alternating path**: a path in which the edges alternately belong or do not belong to a matching M.
- **Augmenting path**: alternating path whose initial and final vertices are unsaturated, namely they do not belong in the matching M.
- **Symmetric difference of sets A and B**: set of all elements that belong to either A or B, but not both. Mathematically, this expression can be written as follows.

$$A \oplus B = (A - B) \cup (B - A)$$

F. Berge's Lemma

Berge's Lemma originally observed by the Danish mathematician Julius Petersen in 1891 and later by Hungarian mathematician Denés Kőnig in 1931. This lemma was formally proven by the French mathematician Claude Berge in 1957.

This lemma states that a matching M is maximum if and only if there is no augmenting path relative to the matching M. We will prove both directions of this bi-implication by using contradiction.

1. A matching M is maximum ⇒ There is no augmenting path relative to M.

   Assume there is an augmenting path P relative to the given maximum matching M. This augmenting path P must be of odd length, containing one more edge not in M than the number of edges it has that are also in M. We can construct a new matching M' by including all edges in the original matching M except those also in P, and including the edges in P that are not in M. This new matching M' is valid because the starting and ending vertices of P are unsaturated by M and the remaining vertices are only saturated by the matching $P \cap M$. As a result, the new matching M' will have one more edge than M. Thus, contradicting the assumption that M was maximum. Formally, given an augmenting path P with respect to some maximum matching M, the matching $M' = P \oplus M$ has $|M'| = |M| + 1$, which is a contradiction.

2. There is no augmenting path relative to M ⇒ A matching M is maximum

   Let there be a matching M' of greater cardinality than M. Consider the symmetric difference $Q = M \oplus M'$. The subgraph Q is no longer necessarily a matching. Each vertex in Q has a maximum degree of 2, meaning all connected components in Q are one of three types:

   - An isolated vertex

- A simple path with edges alternating between M and M′

- a cycle of even length whose edges are alternately from M to M′.

Since M′ has more edges than M, Q contains more edges from M′ than M. By the pigeonhole principle, at least one connected component must be a path with more edges from M′ than M. Any such alternating path will have initial and final vertices unsaturated by M, forming an augmenting path for M, which contradicts the initial assumption.

G. Kuhn's Algorithm

Kuhn's algorithm is one of the most efficient algorithms to solve maximum bipartite matching problem. This algorithm is a direct application of Berge's lemma. First, we start with an empty matching. The algorithm then repeatedly searches for an augmenting path. Whenever it finds one, it updates the matching by alternating along the path. This process is repeated until no more augmenting paths can be found, at which point the current matching is considered maximum.

For convenience, let's assume that the input graph is already divided into two parts, namely V_1 and V_2 and the traverse algorithm used in this case is DFS. General steps for this algorithm can be described as follows.

1. The algorithm checks all vertices v in the first part of the graph (V_1): $v = 1 \ldots n_1$. If a vertex v is already saturated by the current matching, it skips this vertex. Otherwise, it attempts to saturate this vertex by starting a DFS search for an augmenting path from it.
2. The algorithm examines all edges from vertex v.
3. If an edge (v, u) leads to a vertex u that is not yet saturated by the matching, then an augmenting path has been found. This edge is then included in the matching and the search for the augmenting path from v stops.
4. If an edge (v, u) leads to a vertex u that has been saturated by an edge (u, p), the algorithm continues along this edge. The traversal moves to vertex p and continues searching for an augmenting path from there (repeat from step (2)).
5. The traversal starting from vertex v either finds augmenting path, thereby saturating v, or fails to find such a path, leaving v unsaturated.

III. Analysis And Implementation

In this section, we will discuss two possible solutions to get the optimal project assignment, namely brute force algorithm and Kuhn's algorithm. First, let us define some constraints that will be used to analyze and implement the solutions.

Let n be the number of people in a certain organization and m be the number of projects that need to be done. A person's skills may meet some projects' requirements. However, each person can only be assigned to one project or none. Consequently, each project can only receive zero or one person. Our objective is to find an assignment that maximizes the number of people legally assigned to some projects.

It is easy to see that this problem can be transformed into maximum bipartite matching problem. For convenience, let's assume that the input graph is already split into two parts, namely V_1 (set of all people) and V_2 (set of all projects). Each vertex v in V_1 is connected to vertex u in V_2 if the person denoted by v meets the requirement of the project in u. Thus, we can use both the brute force algorithm and Kuhn's algorithm to tackle this problem.

A. Brute Force Algorithm

One of the most straightforward and simple solutions to this problem is to enumerate all the legal matchings and select the one with the highest cardinality. The enumeration can be done by iterating over V_1 and for each vertex, try to connect to each of its neighbors in V_2, while keeping track the previously selected vertices. As stated before, this algorithm is complete (we will always find the solution), yet extremely slow and we will see why.

In general, the steps for the brute force algorithm to solve the project assignment problem recursively can be described as follows.

1. Initialize two empty lists of pair of integers *maxMatching* and *currentMatching*. The array *maxMatching* will store the maximum matching found, while *currentMatching* will store the current matching being checked.
2. Initialize another array of Boolean *visited* with the size of n_2 and value of FALSE. If vertex i in V_2 is already assigned to a vertex in V_1 then $\text{visited}[i] = \text{TRUE}$, otherwise $\text{visited}[i] = \text{FALSE}$.
3. Generate the matching recursively started from $v = 1$ until $v = n_1$ for all possibility.
4. Let v be the current vertex being processed. If $v > n_1$ which indicates that all vertices in V_1 have been processed, then check if the size of currentMatching is greater than the size of *maxMatching*. If that is true, set *maxMatching* to *currentMatching*.
5. If $v \leq n_1$, then for each adjacent vertex w of V_1, if $\text{visited}[w] = \text{FALSE}$, then set $\text{visited}[w] = \text{TRUE}$, add the pair {v, w} to currentMacthing set, and proceed to the next vertex in V_1. In this case, we can also choose not to connect any neighbor to v and proceed to the next vertex.

Below is the pseudocode implementation of the algorithm.

Algorithm 1. Brute Force Algorithm

```
Procedure generateMatching(v: int, maxMatching: list[pair of int], currentMatching: list[pair of int], visited: list[bool])

if v > n1 then
   if size(maxMatching) < size(currentMatching) then
      maxMatching ← currentMatching
else
   for neighbor w of v do
```

```
    if visited[w] = FALSE then
        visited[w] ← TRUE
        currentMatching.add({v, w})
        generateMatching(v + 1, maxMatching,
currentMatching, visited)
        currentMatching.pop()
        visited[w] ← FALSE
    generateMatching(v + 1, maxMatching, currentMatching,
visited)
```

This algorithm traverses through all vertices in V_1. Note that for each vertex v in V_1, generating possible matching involves iterating through all adjacent vertices to v. Suppose that the number of neighbors of each vertex in V_1 is $\leq$ b. Thus, the time complexity of the brute force algorithm is

$$O(b^{n_1})$$

This makes the brute force approach impractical for large graphs due to its exponential growth in complexity. Even if b is relatively small, if n is large, the computational cost will still be expensive.

B. Kuhn's Algorithm

Kuhn's algorithm is one of the most efficient algorithms to solve maximum bipartite matching problems. This algorithm operates by repeatedly searching for an augmenting path. Whenever it finds one, it updates the matching by alternating along the path. This process is repeated until no more augmenting paths can be found, at which point the current matching is considered maximum.

The general steps for this algorithm have been provided in the previous section. Therefore, I will only provide the pseudocode implementation of Kuhn's algorithm in this section. Below is the pseudocode implementation of Kuhn's algorithm.

Algorithm 2. Kuhn's Algorithm

```
Function try_kuhn(v: int, adj: list[list[int]], mt: list[int],
used: list[bool]) → bool

if used[v] then
    → FALSE
used[v] ←TRUE
for neighbor w of v do
    if mt[w] = -1 or try_kuhn(mt[w]) then
        mt[w] ← v
        → TRUE
→ FALSE

Procedure kuhn_algorithm(n: int, k:int, adj: list[list[int]])

mt ← [-1 for 1…k] // initialize mt
for v traverse [1…n] do
    used ← [FALSE for 1…n] // reassign used
    try_kuhn(v)
```

The pseudocode above is based on DFS, which accepts a bipartite graph explicitly divided into two parts. In this context, n represents the number of vertices in V_1, while k represents the number of vertices in V_2. The list adj[v] contains the list of edges originating from vertex v in V_1, listing the vertices to which these edges lead. The vertices in both parts are independently numbered; vertices in V_1 are numbered from 1 to n, and those in V_2 are numbered from 1 to k.

Two additonal lists are used, namely *mt* and *used*. The list *mt* stores information about the current matching, specifically for the vertices in V_2: mt[i] represents the vertex number in V_1 that connected by an edge to vertex i in V_2. The list *used* keeps track of visited vertices during the DFS traversal to make sure that no vertex is visited more than once.

The procedure *kuhn_algorithm* initializes current matching as empty and processes each vertex v in V_1 using *try_kuhn* after resetting the list *used*. Inside the function *try_kuhn*, all edges from vertex v in V_1 are examined. It checks if an edge leads to an unsaturated vertex w or if w is saturated but an augmenting path can be found by recursively starting from mt[w]. If an augmenting path is found, the function alternates the edge adjacent to w to vertex v before returning *TRUE*.

The time complexity of Kuhn's algorithm depends on which part of the graph is chosen as V_1 and which as V_2. In the described implementation, the traversal only begins from the vertices in V_1, making algorithm run in $O(n_1 m)$ where n_1 is the number of vertices in V_1 and m is the number of edges. In the worst-case scenario, this becomes $O(n_1^2 n_2)$ when each vertex in V_1 connected to every vertex in V_1 ($m = n_1 n_2$).

IV. Case Study

In this section, we will try to compare the time required by the brute force algorithm and Kuhn's algorithm to solve the project assignment problem.

A. Test Case 1

In this case, the number of vertices in V_1 is greater than the number of vertices in V_2. We will use n = 10 and k = 4 to test this problem. The dataset for this test case is given as follows.

1	1, 2
2	2, 3
3	3, 4
4	1, 2, 3, 4
5	1, 2, 3, 4
6	1, 4
7	2, 4
8	1
9	2
10	3

Table 1. Dataset for test case 1

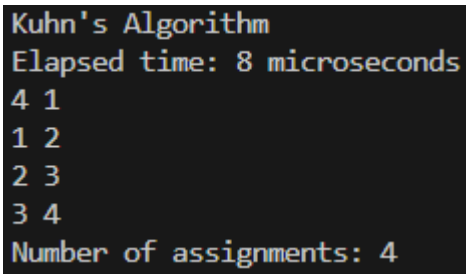

Figure 5. Test Case 1 using Kuhn's algorithm

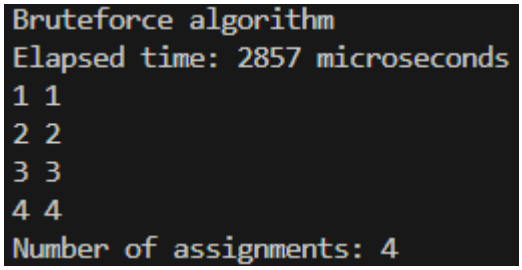

Figure 6. Test Case 1 using the brute force algorithm

The results presented above reveal a significant time difference between the brute force algorithm and Kuhn's algorithm. This disparity arises from the exponential growth of the brute force algorithm, in contrast to the polynomial time complexity of Kuhn's algorithm.

B. Test Case 2

In this case, the number of vertices in V_1 is lesser than the number of vertices in V_2. We will use n = 5 and k = 8 to test this problem. The dataset for this test case is given as follows.

1	1, 2, 3, 4
2	2, 3, 4, 5
3	3, 4, 5, 6
4	4, 5, 6, 7
5	5, 6, 7, 8

Table 2. Dataset for test case 2

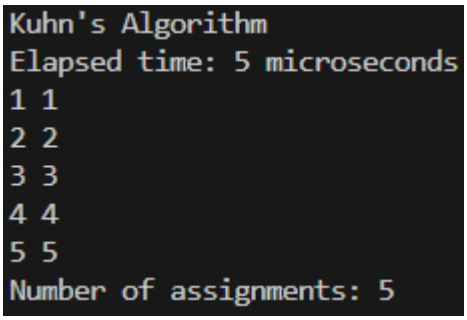

Figure 7. Test Case 2 using Kuhn's algorithm

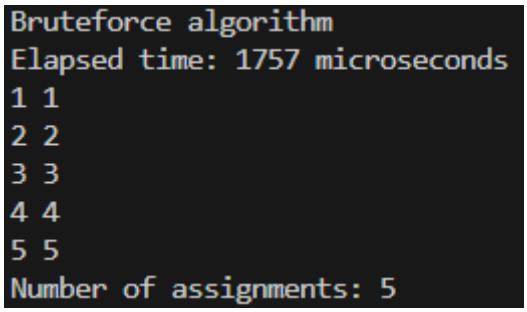

Figure 8. Test Case 2 using brute force algorithm

It is important to note the significant time difference between the brute force algorithm and Kuhn's algorithm in this test. The results of this test are consistent with our previous findings, demonstrating that Kuhn's algorithm is substantially more efficient than the brute force algorithm. These experimental results corroborate the theoretical complexity analysis presented in the previous section, which highlights the difference in time complexity between the two algorithms.

V. Conclusion

In this paper, we explored the application of Kuhn's algorithm in solving the project assignment problem. Through both theoretical and experimental validation, we demonstrated the efficiency of Kuhn's algorithm compared to the brute force approach.

Our theoretical analysis established that Kuhn's algorithm operates in polynomial time, specifically $O(n_1^2 n_2)$ in worst case which is significantly more efficient than the exponential growth of the brute force algorithm, i.e. $O(b^{n_1})$. Experimentally, we tested both algorithms on two datasets that confirm our theoretical findings. Kuhn's algorithm consistently outperformed the brute force algorithm in terms of execution time.

In conclusion, Kuhn's algorithm provides a robust solution for optimal project assignment in bipartite graphs. Its polynomial time complexity makes it a superior choice over the brute force algorithm, ensuring efficient performance even with large datasets.

Aknowledgement

I express my gratitude to the Almighty God for His grace, as I have been able to successfully complete the paper "Optimal Project Assignment Using Kuhn's Algorithm for Maximum Bipartite Matching". I would also like to express my gratitude to the lecturers of the Algorithm Strategy course who has guided me throughout the learning process in this course. Finally, I would like to thank all the sources referenced in this paper.

References

[1] "Depth First Search (DFS) – Algorithms for Competitive Programming." Accessed: June 12 2024. [Online]. Available: https://cp-algorithms.com/graph/depth-first-search.html

[2] Kingsfor, Card "CMSC 451: Maximum Bipartite Matching." Accessed: June 12 2024. [Online]. Available: https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/matching.pdf

[3] “Kuhn’s Algorithm for Maximum Bipartite Matching – Algorithms for Competitive Programming.” Accessed: June 12 2024. [Online]. Available: https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html

[4] Maulidevi, N. U., Munir, R. ”Breadth/Depth First Search (Bagian 1)”. 2021. Accessed: June 12 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf

[5] Munir, R. “Graf (Bagian 1).” Accessed: June 12 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf

[6] “Maximum Bipartite Matching – Geeks for Geeks”. Accessed: June 12 2024. [Online]. Available: https://www.geeksforgeeks.org/maximum-bipartite-matching

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

Adril Putra Merin 13522068