Evaluation of algorithms for solving the edge coloring problem in bipartite graphs

Komnen Knežević[1], Predrag Obradović[1] and Marko Mišić[1][0000-0002-7369-4010]

[1] University of Belgrade, School of Electrical Engineering, 11120 Belgrade, Serbia
marko.misic@etf.bg.ac.rs

Abstract. The graph coloring problem has a wide range of applications in different areas of science and engineering. Graph coloring is a problem where it is necessary to assign colors to graph elements (vertices or edges). The graph coloring problem can be represented in the way that it is necessary to color the vertices of the graph, or to color the edges of the graph under certain conditions. It belongs to the Non polynomial (NP) class of problems. This paper presents an analysis of algorithms for solving the edge coloring problem and their performance for bipartite graphs. For edge coloring, it is necessary to examine whether it is possible to color the edges of the graph with certain number of colors so that any two edges incident to the same vertex are colored with a different color. Numerous approaches have been developed to solve this problem, which have advantages and disadvantages. In this study algorithms for edge coloring were analyzed, such as brute force algorithm based on the maximum degree of the nodes, an algorithm that uses Dinitz's method, and an algorithm that uses Euler's partition. The solutions were carefully evaluated using four different datasets. The results show that brute force algorithm is superior for smaller graphs with up to 10^4 vertices, when it reaches memory limits. The algorithm that uses the Euler's partition is the fastest for graphs with large number of vertices.

Keywords: bipartite graphs, edge coloring, Dinitz's algorithm, Euler's partition, algorithms analysis.

1 Introduction

A graph represents a non-linear data structure consisting of a set of vertices (nodes) and a set of edges (links) [1]. Edges connect nodes that are in a defined kind of relationship. Graphs are used in numerous fields of science and technology, such as computer science, mathematics, biology, economics, etc. The reason for such a wide usage is that in many fields there are problems that can be modeled by a certain interaction between some objects. A deeper analysis of such interactions is more easily performed through graphs.

Graph coloring is a problem where it is necessary to assign colors to graph elements which are vertices and edges [2]. To solve the problem of graph node coloring, for example for an undirected graph and an integer k, it is necessary to examine whether it is possible to color the vertices of the graph with k colors so that any two adjacent

vertices are colored with a different color. Also, this problem can be formulated as edge coloring problem where it is necessary to examine whether it is possible to color the edges of the graph with *k* colors so that any two edges incident to the same node are colored with a different color. Thus, the graph coloring problem can be presented both in the context of node coloring or edge coloring under certain conditions.

The graph coloring problem has a wide range of applications. Most of them are related to various scheduling problems [3]. Some of the applications are in communication systems, such as allocation of frequency range to radio stations. The other applications are present in educational settings, such as making timetables in schools, timetables for examinations during exam periods, but also for timetables for matches at sports events, timetables for airplanes taking off from airports, etc. Graph coloring allocation is the predominant approach to solve register allocation by compilers [4]. In all these situations, the problem can be represented as a graph and reduced to a graph coloring problem [2].

The graph coloring problem belongs to the group of *Non-Polynomial* problems which means that it could not be solved in polynomial time complexity [5]. The motivation of this paper is to present and evaluate several algorithms for solving the graph coloring problem with a minimum number of different colors, where the colors are assigned to the edges of the graph, but specifically for bipartite graphs.

Numerous approaches have been developed to solve this problem, as graph coloring problem has several variations. We analyzed several algorithms for edge coloring in bipartite graphs in terms of their theoretical foundations, design, time and space complexity, scalability, and applicability. Some solutions to the problem are theoretically sound, but applicable only in limited real-life scenarios. In this paper, three algorithms were implemented and evaluated using four different datasets: brute force algorithm based on the maximum degree of the nodes, an algorithm that uses Dinitz's method, and an algorithm that uses Euler's partition.

The rest of the paper is organized as follows. In the second section, elements of the graph theory relevant for the problem are presented. The third section describes used algorithms. In the fourth section, characteristic test examples are presented on which the described algorithms were tested. An analysis of the obtained results was performed. A short conclusion of the paper and the directions for further work are given in the final section.

2 Graph Coloring Problem in Bipartite Graphs

A bipartite graph is an undirected graph where the set of nodes *V* can be divided into two subsets *V1* and *V2* so that each edge *{x, y}* corresponds to one node from *V1* and one node from *V2*. A graph is bipartite if and only if there is no cycle of odd length in the graph.

The flow graph is a weighted directed graph, where the weight of the edges represents their throughput capacity. The condition that the amount of flow entering the node is equal to the amount of flow leaving the node must be satisfied. However, this condition does not include a source node that has only output flow, and a destination node

that has only input flow. Some of the systems that can be modeled with such a graph are: liquid flow through the pipe distribution system, current flow in the electrical network, information flow in the communication network, transport problems in the road and railway network [6]. The optimal flow represents the maximum output flow through the source node, that is, the maximum input flow into the destination node, which can be transmitted through the flow graph to satisfy the above conditions. The problem can be solved by various algorithms such as MPM [7], Ford-Fulkerson [8], *push-relabel* [9], and Dinitz [10] algorithms. Those algorithms can be used in solving graph coloring problems.

For bipartite graphs, a matching is a set of the edges connecting nodes from *V1* and nodes from *V2* such that no edge shares a common node. A maximum matching is a matching that contains maximum possible number of edges. This problem can be solved by converting it into a flow graph and using one of the aforementioned algorithms such as Ford-Fulkerson to find the maximum matching.

An Euler path is a path in a graph that includes all edges exactly once. An Euler cycle is an Euler path that starts and ends at the same node. An Euler graph is a graph that contains an Euler cycle. Euler's theorem states that if we consider a connected graph *G*, the graph *G* is an Euler graph if and only if each node of the graph is of even degree [11]. In a graph *G* in which there are exactly two nodes of odd degree *u* and *v*, there exists an Euler path that starts at one of the nodes *u* or *v* and ends at the other node of odd degree. There is no Euler cycle in such a graph. An Euler partition represents a partition of the edges of the graph into open and closed paths in such a way that every vertex of odd degree is the end of exactly one open path, and every vertex of even degree is the end of no open paths.

A Hamiltonian path is a path in a graph in which each node of the graph is visited exactly once. A Hamiltonian cycle represents a Hamiltonian path that starts and ends at the same node. A Hamiltonian graph is a graph in which there is a Hamiltonian cycle. Determining whether a Hamiltonian path exists in a graph is an NP-complete problem. Some solutions to graph coloring problems in bipartite graphs use Euler partitions.

For general graphs, the *chromatic index* of the graph, which is denoted by $\boldsymbol{\chi'(G)}$ represents the minimum number of different colors used to color the graph G. Let's denote by $\Delta(G)$ the degree of the graph which represent the maximum degree of all vertices of the graph G. It must be true that $\chi'(G) \geq \Delta(G)$. According to *König's* theorem, every bipartite graph satisfies $\chi'(G) = \Delta(G)$, so we know what is the number of colors that we need.

In graph theory, graph coloring represents the problem of assigning certain colors to the elements of the observed graph. One example is the coloring of nodes, where it is necessary to assign colors to the nodes of the graph so that there are no two nodes that are neighbors and colored the same color. Another example is that for a given graph *G* it is necessary to color the graph with a minimum number of different colors, so that colors are assigned to each edge with the condition that there are no two incident edges for the same node that are colored the same color. That problem will be discussed in more detail in this paper with the additional condition that *G* is a bipartite graph.

For each graph edge coloring problem, there is an equivalent graph node coloring problem of the corresponding line graph [12]. For a given graph *G*, a line graph *L(G)*

is a graph such that each node in the graph *L(G)* represents an edge in the graph *G*. Nodes in the graph *L(G)* are adjacent if and only if the corresponding edges in the graph *G* are adjacent, that is, they have a common node. Thus, a certain coloring of the nodes of the line graph *L(G)* represents the coloring of the edges of the graph *G* with the same number of different colors.

3 Algorithms

In this section, we present three different solutions for the edge coloring problem. Each solution is represented with the pseudocode.

Table 1. Pseudocode of the edge coloring based on the maximum degree of a node in graph.

```
Input: Bipartite graph G
Output: Colored edges of bipartite graph G
```

```
recolorEdge(u, c1, c2)
v = EdgeOfNodeWithColor(u, c1).Node
idOfEdge = EdgeOfNodeWithColor(u, c1).Edge
colorEdge(idOfEdge, c2)
if existsNeighbourWithColorOfEdge(v, c2) == false then
  return
else
  recolorEdge(v, c2, c1)
end_if
return
ColorAllEdges()
for i = 1 to m do
  u = edge[i].firstNode
  v = edge[i].secondNode
  if existsSameFreeColorForBoth(u,v) then
    c = findSameFreeColorForBoth(u,v)
    colorEdge(i, c)
  else
    c1 = findFreeColor(u)
    c2 = findFreeColor(v)
    colorEdge(i, c1)
    if existsNeighbourWithColorOfEdge(v, c2) then
      recolorEdge(v, c1, c2)
    end_if
  end_if
end_for
```

Brute force edge coloring algorithm (BFEC) is based on the maximum degree of the nodes. The pseudocode is given in Table 1. Every color has the index that satisfies equation $1 \leq Color_{index} \leq \Delta(G)$. We traverse through the set of edges. For every edge (x, y) we try to find the existence of a color C such that neither vertex x nor vertex y of edge (x, y) has utilized color C in coloring any of the edges incident to them. If such a color C exists, then we color that edge with the found color. Otherwise, we find the color that vertex x did not use (e.g. $Color_{index} = 1$), and find the color that vertex y did not use (e.g. $Color_{index} = 2$). After that we color the edge (x, y) with $Color_{index} = 1$, and have to recolor the edge (y, z) with $Color_{index} = 2$ because now vertex y has two edges colored with the same color. After that we must check if the vertex z satisfies the conditions and so on. Time complexity of this algorithm is $O(|E||V|)$.

Table 2. Pseudocode of the Dinitz's algorithm.

```
Input: Graph G = ((V,E),c,s,t)
Output: Maximum flow between source s and destination t

G_f  ← ConstructResidualGraph(G)
f ← 0
while existsPathBetweenSourceAndTarget do
  G_L = BFS(G_f)
  while existsPathInLayeredGraph do
  findSomePath()
  k = c_f
  f+=k
  removeZeroResidualEdges()
  removeSinkNodes()
end_while
end_while
f_max  ← f
```

Table 3. Pseudocode of the edge coloring based on Dinitz's algorithm.

```
Input: Bipartite graph G
Output: Colored edges of bipartite graph G

D = getMaxDegree(G)
G1 = makeRegularGraph(G)
for k = 1 to D do
  edgesToColor = Dinitz(G1)
for every edge e in edgesToColor do
    color(e) = k
    delete e from G1;
  end_for
end_for
```

Edge coloring using Dinitz's algorithm (DEC) is based on making the regular bipartite graph with degree of $\Delta(G)$ based on the input bipartite graph G. In this algorithm, we need to be sure that every vertex has the same degree. After that, we use Dinitz's algorithm $\Delta(G)$ times to color matched edges with color of current index. It is based on the Hall's theorem which states that in a regular bipartite graph, there exists a perfect matching meaning that every vertex will have its matched pair. Implementations are usually based on Dinitz's algorithm, which computes the maximum flow in a flow graph, and which is used to find the matching set of edges of a bipartite graph. Time complexity of the algorithm is $O(\Delta|E||V|^{0.5})$. The pseudocodes are given in Table 2 and Table 3.

Table 4. Pseudocode of Euler's partition algorithm.

```
Input: Bipartite graph G
Output: List of all paths that represent egde partitions of G

Paths ← emptyList
StartNodes ← emptyQueue
PutAllNodesWithOddDegree(StartNodes)
PutAllNodesWithNonZeroEvenDegree(StartNodes)
while StartNodes is not empty do
  start = getFirstFromQueue(StartNodes)
  removeFirstFromQueue(StartNodes)
  if isNonZeroDegree(start) = true then
    currPath = makeNewPathEmpty()
    currNode = start
    while isNonZeroDegree(currNode) = true do
      edge = getEdgeFromCurrNode(currNode)
      newNode = getNeighbourFromEdge(edge)
      deleteEdge(edge)
      PutEdgeInCurrPath(currPath, edge)
      currNode = newNode
    end_while
    PutCurrPathInPaths(currPath, Paths)
    if isNonZeroDegree(start) = true then
      putNodeInStartNodes(start, StartNodes)
    end_if
  end_if
end_while
return Paths
```

Table 5. Pseudocode of the edge coloring based on Euler's partition.

```
Input: Bipartite graph G
Output: Colored edges of bipartite graph G

RecursiveEdgeColoring(D)
if D is odd then
  if D = 1 then
    M = G
  else
    M = Dinitz(G)
    c = getNewColor()
    for every edge e in M do
      color(e) = c
      delete e from G;
    end_for
  end_if
end_if
Paths = EulerPartition(G)
if Paths is not empty then
  List1 <- makeEmptyList()
  List2 <- makeEmptyList()
  for each path p in Paths do
    for i = 1 to r do
      if i is odd then
        put ei in List1
      else
        put ei in List2
      end_if
  end_for
  for i = 1 to 2 do
      RecursiveEdgeColoring(⌊D/2⌋)
  end_for
end_if
EdgeColoring(G)
DeleteAllNodesWithZeroDegree(G)
D <- maxDegreeOfVertexInGraph(G)
RecursiveEdgeColoring(D)
```

Edge coloring using Euler's partition (EPEC) is an elegant way to improve the previous algorithm with the divide-and-conquer technique and the use of Euler partition. The idea [13] is to utilize Euler partition to divide a regular bipartite graph with degree $\Delta(G)$ into two subgraphs, *G1* and *G2*, such that the maximum degree of all vertices in both subgraphs is equal to $\left\lfloor \frac{\Delta(G)}{2} \right\rfloor$ or $\left\lceil \frac{\Delta(G)}{2} \right\rceil$. After the division of the regular bipartite

graph into two subgraphs *G1* and *G2*, the objective is to independently color each of the subgraphs *G1* and *G2*. If $\Delta(G)$ is an odd number, then both subgraphs could have the maximum degree of all vertices equal to $\left\lceil \frac{\Delta(G)}{2} \right\rceil$, and if we color separately subgraphs *G1* and *G2* then we would color the graph *G* with $\Delta(G) + 1$ colors, which is not the minimal number of colors. So, in this case we would use Dinitz's algorithm to find a matching set of edges, color them, and in that way, we made a graph *G* with degree $\Delta(G) - 1$, which is an even number and can continue to divide the graph into two subgraphs. We repeat this process until we have the graph G' that has $\Delta(G') = 1$. Time complexity of this algorithm is $O(|E||V|^{0.5} log\Delta)$. An interesting case emerges when $\Delta(G) = 2^n$, where the time complexity of this algorithm becomes $O(|E| \log|V| + |V|)$. The pseudocodes are given in Table 4 and Table 5.

4 Evaluation and Discussion

The three described algorithms have been implemented in C++ programming language. The method of loading the number of nodes and edges of the graph was based on a Codeforces online competition problem [14]. The implementation of the Dinitz algorithm was based on [15][16], while the implementation method for the algorithm using the Euler partition was based on the solution implemented in [16]. Their execution time has been evaluated on a local machine using four different datasets with various forms of bipartite graphs.

The first set of test examples consisted of several graphs: a complete graph, a sparse graph, as well as an example of a graph that was created so that the maximum degree of all vertices was of the form $\Delta(G) = 2^{\mathrm{n}}$. In this set of test examples, the total number of vertices in the graph was up to 1000 vertices, as shown in Table 6.

The second set of tests consists of several test examples, for which the total number of vertices in the graph is 10^4, with the maximum number of edges that goes up to $5 * 10^6$ as presented in Table 7. The third set of tests consisted of tests in which there were a total of $2 * 10^4$ vertices, with the maximum number of edges that goes to 10^7. The overview of the third dataset is shown in Table 8. The fourth set of tests was created so that there were $2 * 10^5$ vertices in total, where maximum number of edges goes to $5 * 10^7$ (Table 9). For the second and third set of tests the maximum value for $\Delta(G)$ is 1000, and for the fourth set of tests the maximum value for $\Delta(G)$ is 500.

Table 6. Overview of dataset 1 used for testing.

	No. V1	No. V2	No. edges	$\Delta(G)$
Complete graph	400	400	$16 * 10^4$	400
Sparse graph	600	600	1800	3
Graph $\Delta(G) = 2^{\mathrm{n}}$	1000	1000	263120	512

Table 7. Overview of dataset 2 used for testing.

No. nodes	No. V1	No. V2	No. edges	$\Delta(G)$
10^4	5000	5000	25000	5
			$5 * 10^4$	10
			10^5	20
			$5 * 10^5$	100
			$25 * 10^5$	500
			$5 * 10^6$	1000

Table 8. Overview of dataset 3 used for testing.

No. nodes	No. V1	No. V2	No. edges	$\Delta(G)$
$2 * 10^4$	10^4	10^4	10^5	10
			$2 * 10^5$	20
			10^6	100
			$5 * 10^6$	500
			10^7	1000

Table 9. Overview of dataset 4 used for testing.

No. nodes	No. V1	No. V2	No. edges	$\Delta(G)$
$2 * 10^5$	10^5	10^5	$5 * 10^5$	5
			10^6	10
			$2 * 10^6$	20
			10^7	100
			$5 * 10^7$	500

The results of the first set of tests (Table 10) showed us that the brute force algorithm has the shortest execution time, which is unexpected based on the time complexities of all algorithms. Comparing the algorithm that uses the Dinitz's algorithm and the algorithm that uses the Euler's partition, we noticed that in case of a sparse graph the execution time of the algorithm that uses the Dinitz's algorithm is less, which is also unexpected based on their time complexity. Regarding the example of a graph that has the maximum degree of all vertices of the form $\Delta(G) = 2^n$, we noticed the expected result, which is that the algorithm that uses Euler's partitions (5.772s) is much faster than the algorithm that uses Dinitz's algorithm (46.848s). We got similar results as in the example of a complete graph.

Table 11 presents the results of the second set of tests. It shows that the algorithm that uses Euler's partitions has better performance than the algorithm that uses Dinitz's algorithm as the value of the parameter $\Delta(G)$ increases. However, the parameters of this set of examples are not large enough to show that the algorithm with Euler partition runs faster than the brute force edge coloring algorithm.

Table 10. The results of the three implemented algorithms on dataset 1. The best results are given in bold. BFEC – brute force edge coloring algorithm, DEC – Dinitz algorithm-based edge coloring, EPEC – Euler partitions -based edge coloring algorithm.

	BFEC	DEC	EPEC
Complete graph	**0.229s**	25.795s	3.729s
Sparse graph	**0.291ms**	14.823ms	18.065ms
Graph $\Delta(G) = 2^n$	**0.325s**	46.848s	5.772s

Table 11. The results of the three implemented algorithms on dataset 2. The best results are given in bold. BFEC – brute force edge coloring algorithm, DEC – Dinitz algorithm-based edge coloring, EPEC – Euler partitions -based edge coloring algorithm.

$\Delta(G)$	BFEC	DEC	EPEC
5	**24.207ms**	41.856ms	55.678ms
10	**30.726ms**	109.223ms	126.630ms
20	**31.616ms**	306.135ms	288.818ms
100	**120.089ms**	4.288s	1.879s
500	**1.418s**	1.254min	13.257s
1000	**5.527s**	5.635min	52.342s

Table 12. The results of the three implemented algorithms on dataset 3. The best results are given in bold. BFEC – brute force edge coloring algorithm, DEC – Dinitz algorithm-based edge coloring, EPEC – Euler partitions -based edge coloring algorithm.

$\Delta(G)$	BFEC	DEC	EPEC
10	**48.526ms**	230.434ms	258.063ms
20	**58.024ms**	641.160ms	585.068ms
100	*Memory limit*	9.364s	**4.061s**
500	*Memory limit*	2.798min	**34.764s**
1000	*Memory limit*	*Infinity*	**1.163min**

Table 13. The results of the three implemented algorithms on dataset 5. The best results are given in bold. BFEC – brute force edge coloring algorithm, DEC – Dinitz algorithm-based edge coloring, EPEC – Euler partitions -based edge coloring algorithm.

$\Delta(G)$	BFEC	DEC	EPEC
5	*Memory limit*	**0.977s**	1.152s
10	*Memory limit*	**2.470s**	2.580s
20	*Memory limit*	6.664s	**5.956s**
100	*Memory limit*	1.424min	**45.715s**
500	*Memory limit*	*Infinity*	**5.489min**

The result of the third set of tests shows that if the value of parameter $\Delta(G)$ in the graph is 100 (with 10^6 edges) or more, the brute force algorithm does not execute due to the memory limits, as shown in Table 12. If we look at the test example for which $\Delta(G)$ = 500 (with $5 * 10^6$ edges), we can notice that the algorithm that uses the Euler's partition is executed almost 5 times faster than the algorithm that uses the Dinitz's algorithm. The example where $\Delta(G) = 1000$ (with 10^7 edges) has interesting results. The algorithm that colors edges using Dinitz's algorithm takes infinitely long to execute, while the algorithm that uses Euler's partitions executes in almost one minute.

The results of the fourth set of tests are presented in Table 13. They have similar results as in the third set of tests, only for smaller values of parameter $\Delta(G)$, because the total number of vertices in the bipartite graph is 10 times higher in the tests of the fourth set. For all examples in this test set, the brute force algorithm does not execute due to memory limitations. For the example where the parameter $\Delta(G)$ has a value of 100 (with 10^7 edges), there is almost double the difference in execution time between the algorithm that use the Dinitz algorithm and the algorithm that use the Euler partition. In the last example of the fourth set of tests, for the value of the parameter $\Delta(G) = 500$ the algorithm that use Dinitz's algorithm takes infinitely long to execute while algorithm that use Euler partition executes in almost 5 minutes.

Based on the results from all sets of tests, we can conclude that for small graph sizes, the brute force algorithm is executed the fastest, due to the existence of a larger constant factor in the time complexity (which is not shown in the *O* notation) than in the case of algorithms that use Euler's partition and Dinitz's algorithm. On the other hand, in the case of graphs that have a large number of vertices, the algorithm that uses the Euler partition is the fastest.

5 Conclusion

This paper presents the graph coloring problem that belongs to the group of *NP-hard* problems. A definition of the edge coloring problem in bipartite graphs is given, in which the number of used colors needs to be minimized. Several algorithms that solve the presented problem with different approaches are described.

The algorithms were tested on several synthetic datasets. Each of the datasets consisted of some characteristic topologies of a bipartite graph, such as a complete graph, a sparse graph, a graph for which the maximum degree of a node of the form $\Delta(G) = 2^n$ is valid, as well as for other values of the parameter $\Delta(G)$. The results show that brute force algorithm is superior for smaller graphs with up to 10^4 vertices, when it reaches memory limits. The algorithm that uses the Euler's partition is the fastest for graphs with large number of vertices.

There are two main directions for future research work. The first direction is to improve the algorithms in terms of better performance than the described algorithms, as well as the implementation of new algorithms that bring improvement in terms of performance by means of optimization and parallelization. It is significant for reasons of faster execution in the case when there is a huge number of nodes. Another direction is the research of new fields where graph coloring can be applied in real-life problems.

References

1. Bender, E., Williamson, S., Lists, D.: Graphs With an Introduction to Probability. University of California at San Diego, 147-148 (2010).
2. Jensen, T. R., Toft, B.: Graph coloring problems. John Wiley & Sons (2011).
3. Marx, D.: Graph colouring problems and their applications in scheduling. Periodica Polytechnica Electrical Engineering (Archives), vol. 48, no. 1-2, pp. 11-16. (2004).
4. Eisl, J., Leopoldseder, D., Mössenböck, H.: Parallel trace register allocation. In: Proceedings of the 15th International Conference on Managed Languages & Runtimes, pp. 1-7. ACM, USA (2018).
5. Karp, R. M.: Reducibility among combinatorial problems. Springer Berlin Heidelberg (2010).
6. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to algorithms. MIT press (2022).
7. Malhotra, V. M., Kumar, M. P., Maheshwari, S. N.: An O (| V| 3) algorithm for finding maximum flows in networks. Information Processing Letters, vol. 7, no. 6, pp. 277-278. (1978).
8. Ford, L. R., & Fulkerson, D. R.: Maximal flow through a network. Canadian journal of Mathematics, vol. 8, pp. 399-404. (1956).
9. Goldberg, A. V., Tarjan, R. E.: A new approach to the maximum-flow problem. Journal of the ACM (JACM), vol. 35, no. 4, pp. 921-940. (1988).
10. Dinitz, Y.: Dinitz'algorithm: The original version and Even's version. Theoretical Computer Science: Essays in Memory of Shimon Even, pp. 218-240. (2006).
11. Beşeri, T.: Edge Coloring of a Graph. Izmir Institute of Technology (Turkey) (2004).
12. Deo, N.: Graph theory with applications to engineering and computer science. Courier Dover Publications (2017).
13. Gabow, H. N.: Using Euler partitions to edge color bipartite multigraphs. International Journal of Computer & Information Sciences, vol. 5, no. 4, pp. 345-355 (1976).
14. Codeforces - Problem F – Edge coloring of bipartite graph, https://codeforces.com/contest/600/problem/F, last accessed 2023/05/24.
15. Maximum flow – Dinic's algorithm, https://cp-algorithms.com/graph/dinic.html, last accessed 2023/05/24.
16. Sotanishy implementation of edge coloring of bipartite graph, https://codeforces.com/contest/600/submission/162194398, last accessed 2023/05/24.