

On Efficient Computation of DiRe Committees

Kunal Relia*

January 12, 2024

Abstract

Consider a committee election consisting of (i) a set of candidates who are divided into arbitrary groups each of size *exactly* two and a diversity constraint that stipulates the selection of *at least* one candidate from each group and (ii) a set of voters who are divided into arbitrary populations each approving *exactly* two candidates and a representation constraint that stipulates the selection of *at least* one candidate from each population's set of approved candidates.

The DiRe (Diverse + Representative) committee feasibility problem (a.k.a. the minimum vertex cover problem on unweighted undirected graphs) concerns the determination of the smallest size committee that satisfies the given constraints.

Here, for this problem, we discover an unconditional deterministic polynomial-time algorithm that is an amalgamation of maximum matching, breadth-first search, maximal matching, and local minimization.

*This work, while the author was a student at New York University, was generously supported in part by Julia Stoyanovich's NSF grants No. 1916647, 1934464, and 1916505. Independent Researcher. Correspondence: kunal.relia91@gmail.com or krelia@nyu.edu.

Contents

1	Introduction	3
2	Notation and Preliminaries	4
3	Algorithm Overview	4
3.1	Maximum Matching	4
3.2	Breadth-first Search	5
3.3	Maximal Matching	5
3.4	Local Minimization	7
3.5	Summary	7
4	Algorithm	8
5	Proof of Correctness	11
6	Time Complexity Analysis	16
7	Conclusion	19
A	Implementation of Algorithm	21

Preface: The DiRe committee feasibility problem (stated in the abstract) and the vertex cover problem on unweighted undirected graphs are equivalent (vertices = candidates; edges = candidate groups / voter populations’ approved candidates). Hence, for technical simplicity, we henceforth focus the discussion on the latter problem.

1 Introduction

Given an unweighted undirected graph (specifically, a 2-uniform hypergraph), the vertex cover of the graph is a set of vertices that includes at least one endpoint of every edge of the graph. Formally, given a graph $G = (V, E)$ consisting of a set of vertices V and a collection E of 2-element subsets of V called edges, the vertex cover of the graph G is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge of the graph, i.e., for all $e \in E$, $e \cap S \neq \emptyset$. The corresponding computational problem of finding the minimum-size vertex cover (MVC) is NP-complete¹ [Coo71, Lev73, Kar72], which means that there is no *known* deterministic polynomial-time algorithm to solve MVC. Here, we present an unconditional deterministic polynomial-time algorithm for MVC on unweighted simple connected graphs².

We sparingly use “Non-technical Comment” boxes in this paper. These comments are not a part of the paper in a technical sense but they provide important answers to some non-technical but important “whys” and “so whats” of the paper. It may help a reader relate to the journey of working on the paper.

Non-technical Comment: A chance re-encounter with one of Aesop’s fables, “The Fox and the Grapes”, from my childhood days was a motivation to begin thinking about this paper. By calling the DiRe committee feasibility problem “hard” (NP-hard), was I being the fox who found the grapes sour instead of my inability to find an efficient algorithm to reduce inequality?

¹Strictly speaking, the decision version of the vertex cover problem is NP-complete whereas MVC itself (search version) is NP-hard. See Section 2.1 of [Kho19] for a lucid explanation delineating (a) search and decision problems and (b) NP-hardness and NP-completeness.

²We subtly yet drastically switch the discussion from unweighted undirected graphs to unweighted simple connected graphs. For simplicity, we want to avoid having loops and/or unconnected components in the graph. In the context of this paper, this switch has no impact on the NP-completeness of the problem. Notwithstanding, in the case of the presence of loops, our algorithm will work (with minor modifications) if each loop is replaced by adding a dummy vertex and a corresponding edge. In the case of unconnected components, we can run the algorithm for each connected component independently and take a union of each of the minimum vertex covers to get the final minimum vertex cover.

2 Notation and Preliminaries

We now formally define the computation problems related to finding the vertex cover of a given graph. First, we define the search/optimization problem:

Definition 1 (Minimum Vertex Cover Problem (MVC)). *Given a graph G , what is the smallest non-negative integer k such that the graph G has a vertex cover S of size k ?*

Next, we restate the above as a decision problem:

Definition 2 (Vertex Cover Problem (VC)). *Given a graph G and a non-negative integer k , does the graph G have a vertex cover S of size at most k ?*

Unless stated otherwise, we henceforth discuss solving VC (i.e. Definition 2), which is actually NP-complete.

3 Algorithm Overview

The algorithm is broadly divided into four phases. The first three phases are (slightly adapted versions of) algorithms for three known problems, namely maximum matching, breadth-first search, and maximal matching. The last phase is a technique we call local minimization. We now discuss these phases and give an overview of the algorithm.

Definition 3 (Matching). *Given a graph G , a matching M is a subset of the edges E such that no vertex $v \in V$ is incident to more than one edge in M .*

Alternatively, we can say that given a graph G , no two edges in a matching M have a common vertex.

3.1 Maximum Matching

Phase 1 of the algorithm finds maximum matching of the input graph:

Definition 4 (Maximum Matching). *Given a graph G , a matching M is said to be maximum if for all other matching M' , $|M| \geq |M'|$.*

Equivalently, the size of the maximum matching M is the (co-)largest among all the matching. Next, there is a known relationship between the size of maximum matching and the size of minimum vertex cover:

Lemma 1. *In a given graph G , if M is a maximum matching and S is a minimum vertex cover, then $|S| \geq |M|$.*

Lemma 1 means that the largest number of edges in a matching does not exceed the smallest number of vertices in a cover. We use this fact to set a lower bound on the size of the minimum vertex cover and terminate the algorithm early if the integer k is less than $|M|$.

3.2 Breadth-first Search

Phase 2 of the algorithm stores the vertices at each level of the tree derived using breadth-first search (BFS):

Definition 5 (Breadth-First Search). *Given a graph G , a Breadth-first Search (BFS) algorithm seeds on a root vertex $v \in V$ and visits all vertices at the current depth level of one. Then, it visits all the nodes at the next depth level. This is repeated until all vertices are visited.*

While the BFS algorithm is canonically a search algorithm, we use it here to derive a tree. This tree itself is not needed. Only the information of the level at which each vertex is in the tree is stored for use during the third phase.

3.3 Maximal Matching

Phase 3 of the algorithm entails the use of maximal matching.

Definition 6 (Maximal Matching). *Given a graph G , a matching M is said to be maximal if for all other matching M' , $M \not\subseteq M'$.*

In other words, a matching M is maximal if we cannot add any new edge $e \in E$ to the existing matching. During this maximal matching phase, the edges are selected using a specific procedure that uses information stored (i) regarding the edges that are a part of the maximum matching and (ii) about the vertices present at each level of the tree derived using BFS. Additionally, during each iteration of maximal matching, the algorithm stores the *current* neighboring vertices of each endpoint. We call this as an endpoint vertex *representing* its neighboring vertex.

Definition 7 (Represents³). *Given a graph G , a vertex $u \in V$ is said to **represent** a vertex $v \in V$ when vertex v is currently connected to vertex u by an edge $e \in E$. Conversely, vertex v is **represented by** vertex u .*

Observe that when some vertex u *currently* represents a vertex v , the algorithm is essentially storing information about the presence of an edge connecting the two vertices. There is stress on the word *currently* as for a given iteration, an edge should not have been removed. The information is stored in *represents table* that consist of *represents lists*.

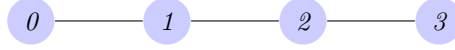
Definition 8 (Represents Table). *A represents table R is a 2-column table that stores the endpoints of edges selected during maximal matching and the vertices each endpoint represents.*

Definition 9 (Represents List). *Given a represents table R , a vertex $u \in V$ that is represented by a vertex $v \in V$ is said to be in the represents list of v .*

³The term is inspired by a type of multiwinner election where the aim is to elect the smallest committee that represents every voter. In our context, we want to select the smallest set of vertices that covers (represents) each edge.

Finally, in the last step of an iteration of the maximal matching phase, the algorithm removes the edge that connects (i) the two endpoints and (ii) endpoints and their respective neighbors.

Example 1. Consider the following graph G :



During maximal matching, assume that the algorithm first selects the edge connecting vertex 0 and vertex 1. Then, the endpoints of the selected edge are 0 and 1. For each endpoint, the algorithm stores the information of the vertices it represents. Here, vertex 0 represents $\{1\}$ and vertex 1 represents $\{0, 2\}$. All the edges connected to the two endpoints in any way are removed.



In the next iteration of maximal matching, the algorithm selects the edge connecting vertex 2 and vertex 3. The two endpoints represent each other only. Specifically, vertex 2 represents $\{3\}$ and vertex 3 represents $\{2\}$. All the edges connected to the two endpoints in any way are removed.



Finally, the following information is stored by the algorithm:

Node 1	Node 2
0 - $\{1\}$	1 - $\{0, 2\}$
2 - $\{3\}$	3 - $\{2\}$

Table 1: Information stored in a “**Represents Table**” R after the end of maximal matching phase.

The information contained in row 1 under “Node 2” of Table 1 is: vertex 1 is an endpoint vertex that represents vertices 0 and 2. Conversely, vertices 0 and 2 are represented by endpoint vertex 1. Also, vertices 0 and 2 are in the **represents list** of endpoint vertex 1.

Two known facts related to maximal matching will be useful later:

Lemma 2. The endpoints of a maximal matching form a vertex cover.

Lemma 3. In a graph G , if a matching M is maximum, it implies the matching M is also maximal. The converse does not hold.

We specifically use Lemma 3 in Section 5 and explain why the third phase is called maximal matching and not maximum matching.

3.4 Local Minimization

The last Phase, local minimization, is a new technique. It is not adapted from any known techniques to the best of our knowledge. Also, note that our version of local minimization is not related to the local search used in heuristic algorithms. We use the term *local* in local minimization because the vertex cover we get at the end of this phase is the “smallest” and not necessarily minimum. Specifically, the vertex cover we get is dependent on the endpoints of the edges selected during the maximal matching phase. Hence, from a given set of vertices, local minimization phase uses three stages to select a vertex cover of the smallest possible size, which may not be the minimum vertex cover:

1. **Freeze “necessary” vertices:** Freeze each endpoint v in the represents table R that represents a vertex u that is not an endpoint in R . Vertex u can not be in the vertex cover S as it is not an endpoint of any edge selected during maximal matching. Hence, vertex v necessarily needs to be a part of the vertex cover to cover the edge connecting u and v .
2. **Top-down removal of “terminal” vertices:** Remove each endpoint with degree one in graph G . The other endpoint is simultaneously frozen.
3. **Bottom-up freeze and remove:** Freeze and remove “necessary” and “terminal” vertices, respectively, based on the *current* state of table R .

Definition 10 (Local Minimization). *Given a graph G , a subset of vertices $V' \subseteq V$ that covers all edges and for each vertex $v \in V'$ the list of vertices it represents, the local minimization selects the smallest sized subset of vertices $S' \subseteq V'$ such that each edge is covered.*

3.5 Summary

The algorithm we discovered is an amalgamation of the above-discussed phases. The sequential implementation of these phases ensures we get a minimum vertex cover. At a high-level, this is because: (i) Maximum matching and breadth-first search ensures that the edges selecting during the maximal matching phase follows a procedure as opposed to vanilla maximal matching where edges are selected randomly. (ii) Maximal matching implies we get a vertex cover. (iii) Local minimization ensures we get the smallest vertex cover. Overall, the combination of all these implies we get the minimum vertex cover.

Non-technical Comment: *As discussed, the flow of the algorithm is as follows: maximum matching \rightarrow breadth-first search \rightarrow maximal matching \rightarrow local minimization. However, the evolution of the algorithm happened in the following order: maximal matching \rightarrow local minimization \rightarrow breadth-first search \rightarrow maximum matching. Indeed, eventually “prefixing” the algorithm with maximum matching helped us deal with the messy cycles, especially odd cycles. Recall that Blossom algorithm [Edm65] had to do “extra work” just to deal with odd cycles.*

4 Algorithm

We now present the core contribution of this paper, an algorithm to solve the VC problem. In the algorithm, all ties are broken and all ordering (sorting) of vertices is done based on lexicographic ordering unless noted otherwise. The ordering does not impact the correctness but ensures that for same input, the output remains the same.

Algorithm 1: Vertex_Cover(G, k)

Data: Graph $G = (V, E)$, non-negative integer k

Result: returns “YES” if there is a vertex cover S of size at most k ,
“NO” otherwise

```

1:  $V_s$  = lexicographically sorted vertices
2:  $E_M$  = maximum matching found using the Blossom Algorithm [Edm65]
3: if  $k < |E_M|$  then
4:   | return “NO”
5: end
6: for each  $v \in V_s$  do
7:   |  $BFS_{level}$  = an array of arrays storing sorted vertices at each level of
   |   breadth-first search tree seeded on  $v$ 
8:   |  $R = \text{MaximalMatching}(G, E_M, BFS_{level})$ 
9:   |  $S = \text{LocalMinimization}(R)$ 
10:  | if  $|S| \leq k$  then
11:    | return “YES”
12:  | end
13: end
14: return “NO”

```

Non-technical Comment: The technical discussion for each of the phases of the algorithm will follow in the succeeding sections. Here, we share our non-technical motivation for including maximum matching and BFS phases in the algorithm. Our guiding question was “Is it possible that we have missed out on considering all the factors that decide the vertices being selected to form the minimum-size vertex cover?” Such factors may not be given to us in the traditional sense and hence, may not be “visible”. We may have to infer them to use them. We do so in this paper. Given an unweighted undirected graph for VC problem, maximum matching and BFS lend inherent edge weights and directions, respectively. After traversing through the algorithm, it will be intuitively evident that during maximal matching, each edge carries certain “weight” and the edge selections happen in particular “direction”. Thus, identifying and including such factors was another motivation for this paper.

Algorithm 2: Maximal_Matching(G, E_M, BFS_{level})

Data: Graph $G = (V, E)$, Edges in maximum matching E_M , Levels at which each vertex is present after BFS BFS_{level}

Result: returns R - Represents Table

```

1:  $R$  = a two-column table, Represents Table, that stores the endpoints of
   an edge selected during maximal matching and the corresponding
   vertices each endpoint represents
2: for each  $level$  in  $BFS_{level}$  do
3:   while there is an unvisited vertex in level do
4:     if there exists an edge that connects two vertices on the same
       level and is in  $E_M$  then
5:       | select the edge
6:     else if there exists an edge that connects two vertices on the
       same level and is not in  $E_M$  then
7:       | select the edge
8:     else if there exists an edge that connects one vertex on the
       current level with another vertex on the next level and is in  $E_M$ 
       then
9:       | select the edge
10:    else
11:      | select the edge that connects one vertex on the current level
        | with another vertex on the next level and is not in  $E_M$ 
12:    end
13:    Mark the two endpoints of the selected edge as visited in
        $BFS_{level}$ 
14:    Append after the last row of  $R$  the two endpoints of the selected
       edge and the respective vertices each endpoint represents
15:    Remove from graph  $G$  the selected edge and all the edges that
       are connected to the two endpoints
16:    If any vertex becomes edgeless in  $G$ , mark the vertex as visited
       in  $BFS_{level}$ 
17:  end
18: end
19: return  $R$ 

```

Algorithm 3: Local_Minimization(R)

Data: Represents Table R

Result: returns S - the smallest vertex cover

```
1:  $S = \phi$ 
2:  $P$  = set of endpoints in  $R$  selected during maximal matching
3: for each endpoint vertex  $v$  in  $R$  do
4:   if  $v$  represents at least one vertex not in  $P$  then
5:     //freeze vertex  $v$  but do not remove any vertex from  $R$ 
6:      $R, S = \text{Freeze\_and\_Remove}(R, S, v, \phi)$ 
7:   end
8: end
9: // The following for loop will traverse through the table  $R$  top-down
10: for each row in  $R$  do
11:   if if any one endpoint in row is either frozen or removed then
12:     continue
13:   else if one endpoint  $u$  in row only represents another endpoint
        vertex  $v$  in row and  $v$  represents more than one vertex then
14:     if  $u$  is not represented by any endpoint in  $R$  other than  $v$  then
15:        $R, S = \text{Freeze\_and\_Remove}(R, S, v, u)$ 
16:     end
17:   end
18: end
19: // The following for loop will traverse through the table  $R$  bottom-up
20: for each row in  $R$  do
21:   if (if both endpoints are frozen) or (one endpoint is frozen and one
        is removed) then
22:     continue
23:   else if endpoint  $u$  remains and endpoint  $v$  is removed then
24:      $R, S = \text{Freeze\_and\_Remove}(R, S, u, \phi)$ 
25:   else
26:     //at this point, both endpoints  $u$  and  $v$  in row represent exactly
        one vertex, namely each other
27:     if  $u$  is represented by more endpoints in  $R$  than  $v$  then
28:        $R, S = \text{Freeze\_and\_Remove}(R, S, u, v)$ 
29:     else if  $v$  is represented by more endpoints in  $R$  than  $u$  then
30:        $R, S = \text{Freeze\_and\_Remove}(R, S, v, u)$ 
31:     else
32:        $R, S = \text{Freeze\_and\_Remove}(R, S, u, v)$ 
33:     end
34:   end
35: end
36: return  $S$ 
```

Algorithm 4: Freeze_and_Remove($R, S, freeze, remove$)

Data: Represents Table R , Vertex Cover S , vertex to be frozen $freeze$,
vertex to be removed $remove$

Result: returns Represents Table R , Vertex Cover S

```
1: Remove vertex  $remove$  and its represents list from  $R$ 
2: Freeze vertex  $freeze$  in  $R$ 
3: Append vertex  $freeze$  to  $S$ 
4: Remove vertex  $freeze$  from every represents list in  $R$ 
5: Remove the represents list of vertex  $freeze$  in  $R$ 
6: for each non-frozen and unremoved  $endpoint$  in  $R$  that represents
    $remove$  do
7:   |  $R, S = \text{Freeze\_and\_Remove}(R, S, endpoint, \phi)$ 
8: end
9: for each non-frozen and unremoved  $endpoint$  in  $R$  that does not
   represent any vertex do
10:  |  $R, S = \text{Freeze\_and\_Remove}(R, S, \phi, endpoint)$ 
11: end
12: return  $R, S$ 
```

5 Proof of Correctness

In this section, we show the correctness of the algorithm by proving the following theorem:

Theorem 1. *Algorithm 1 returns “Yes” if and only if a given instance of VC is a “Yes” instance.*

We prove the theorem through a sequence of lemmas. Foremost, in the forward direction, we have the following lemma:

Lemma 4. *If a given instance of VC is a “Yes” instance, then the Algorithm 1 returns “Yes”.*

Proof. If the given instance of VC is a “Yes” instance, then Line 4 of Algorithm 1 can never return “No” as $k \geq |E_M|$ (Lemma 1). Also, the execution will never reach Line 14 of Algorithm 1 as Line 11 will return “Yes” during one of the m iterations when Algorithm 3 finds the minimum vertex cover because $k \geq |S|$. \square

Next, in the reverse direction, we have the following lemma:

Lemma 5. *If the Algorithm 1 returns “Yes”, then the given instance of VC is a “Yes” instance.*

We prove Lemma 5 in the following sequence: (i) we prove that variable S in Line 9 of Algorithm 1 is always a vertex cover (Lemma 6), (ii) we prove that this vertex cover is the smallest vertex cover based on the endpoints selected during

maximal matching (Lemma 7), and (iii) there exists at least one minimum vertex cover among the m smallest vertex covers (Lemma 8).

Lemma 6. *Variable S in Line 9 of Algorithm 1 is a vertex cover.*

Proof. Algorithm 2 is an algorithm for maximal matching. The endpoints of the edges selected during maximal matching form a vertex cover (Lemma 2). However, Algorithm 3 removes some of these endpoints. But an endpoint is removed only if every endpoint that represents the removed endpoint is (i) already frozen⁴ or (ii) is immediately frozen. This is equivalent to ensuring that each edge has at least one endpoint in the set of vertices. This implies that the set of vertices returned by Algorithm 3 is a vertex cover. Hence, variable S in Line 9 of Algorithm 1 is a vertex cover. \square

Lemma 7. *Given a set of endpoints $V' \subseteq V$ of edges selected during maximal matching, variable S in Line 9 of Algorithm 1 is the smallest vertex cover such that for all vertex covers $S' \subseteq V'$, $|S| \leq |S'|$.*

Proof. Based on Lemma 6, S is guaranteed to be a vertex cover. Hence, it remains to be proven that S is the smallest vertex cover derivable from the endpoints selected during maximal matching (Algorithm 2). To do so, we use a couple of observations:

Observation 1. *Given a represents table R consisting of r rows, an endpoint vertex v in row i cannot represent an endpoint vertex u in row j , for all $j < i$ where $i, j \in \mathbb{N}$ and $1 \leq i, j \leq r$.*

When an edge is selected during maximal matching, all the edges covered by two of its endpoints are removed. Simultaneously, the represents table R is updated to reflect the two endpoints and the vertices each of the endpoints represent (Line 14 - Algorithm 2). Hence, the succeeding entry in R can not represent any of the endpoints already present in the table R .

Next observation is related to the distance between (i) an endpoint and the vertices it represents and (ii) the endpoint and the endpoints it is represented by. Such distance is at most one level in the BFS_{level} .

Observation 2. *Given a represents table R and a BFS level table BFS_{level} , an endpoint vertex v in R at level i in BFS_{level} can represent or can be represented by a vertex u that is at level $i - 1$, i or $i + 1$ in BFS_{level} .*

Finally, we proceed to show that variable S in Line 9 of Algorithm 1 is the smallest vertex cover such that for all vertex covers $S' \subseteq V'$, $|S| \leq |S'|$. To do so, we prove that Algorithm 3 returns the smallest vertex cover derivable from the endpoints selected during maximal matching (Algorithm 2).

- *Lines 2 to 8 of Algorithm 3 freezes “necessary” vertices:* Variable P (Line 2 - Algorithm 3) consists of the endpoints of the edges selected by Algorithm 2. Any endpoint v in represents table R that represents a vertex

⁴Recall that a frozen vertex is implicitly always added to the set of vertices S .

not in P needs to be frozen and added to the vertex cover S . If a vertex u is not in P , it implies that, by design, it will not be in the vertex cover. Hence, any vertex that is connected to u via an edge will be an endpoint in R and in turn, it needs to be in the vertex cover.

- *Lines 10 to 18 of Algorithm 3 removes “terminal” vertices:* A terminal vertex u in a connected graph does not bring value to the vertex cover because (i) it represents one vertex v and (ii) it is represented by one vertex v . On the other hand, given that the graph G is connected and consists of more than two vertices⁵, vertex v either (i) represents more than one vertex or (ii) is represented by more than one vertex or (iii) both. Hence, removing vertex u and freezing v is appropriate. Vertex v is added to the vertex cover S . Importantly, due to the presence of recursive calls in Algorithm 4, the loop in Line 10 of Algorithm 3 is executed top-down. At this stage, a top down execution will facilitate removal or freezing of more vertices by the recursive calls of Algorithm 4 as compared to a bottom-up execution (Observation 1).
- *Lines 20 to 35 of Algorithm 3 freezes and removes the **current** “necessary” and “terminal” vertices, respectively:* The execution of the loop in Line 20 will happen bottom-up. During i^{th} iteration of the loop, if both the endpoints are neither frozen nor removed, then it implies that the endpoints represent each other only. Hence, we freeze the endpoint that is represented by more number of endpoints in R and remove the other. Due to Observation 2, the decision to freeze the endpoint that is represented more in R is valid. In case both the endpoints are represented by the same number of endpoints, tie is broken based on lexicographical ordering (as discussed at the beginning of Section 4). Note that represents table R is continuously updated after each freeze or remove operation. Consequently, the represents list of each endpoint gets updated and it may become a necessary or a terminal vertex⁶. This is handled by Line 7 and 10 of Algorithm 4, respectively. Specifically, Line 7 of Algorithm 4 freezes an endpoint that represents a removed vertex. By induction, this is like removing terminal vertices and freezing its neighboring endpoints, but dependent on the current state of the represents table R . Similarly, Line 10 of Algorithm 4 removes an endpoint that does not represent any vertex based on the current state of the represents table R . This is a valid removal as the vertex had not been frozen yet and it does not represent any vertex in the latest iteration.

The above discussed sequential implementation of freezing “necessary” vertices, removing “terminal” vertices, and freezing and removing the **current** “necessary” and “terminal” vertices, respectively, ensures that the frozen vertices (equivalently, the vertices in vertex cover S) form the smallest vertex cover.

⁵In case the graph G has two vertices, it is a trivial case and any one vertex will form the minimum vertex cover. It is handled by Line 32 of Algorithm 3.

⁶We stress on the word “current” due to the continuous updates to represents table R .

In summary, Algorithm 3 returns the smallest possible vertex cover derivable from the endpoints in R given as input. Formally, variable S in Line 9 of Algorithm 1 is the smallest vertex cover such that for all vertex covers $S' \subseteq V'$, $|S| \leq |S'|$ where V' are the endpoints in R . \square

Lemma 8. *Given m sets of endpoints $V' \subseteq V$ of edges selected during maximal matching, there is at least one set V' that is a super-set of the set of vertices in the minimum vertex cover.*

Proof. Algorithm 3 (local minimization) finds the smallest vertex cover from a given set of vertices (Lemma 7). We now show that the input to Algorithm 3 consists of the following cases, each of which ensures that for every graph G , Algorithm 3 will find its minimum vertex cover:

- *perfect matching⁷ (all vertices):* When Algorithm 2 (maximal matching) finds a perfect matching, all vertices will be added as endpoints in the represents table R . Hence, the smallest vertex cover that Algorithm 3 returns is indeed the minimum vertex cover.
- *maximum matching:* There can be multiple maximum matching in a graph. Algorithm 1 uses one maximum matching. Hence, there are four possibilities:
 1. *endpoints of the maximum matching E_M is a super set of the minimum vertex cover and Algorithm 2 traverses through E_M :* This is a trivial case and Algorithm 3 returns the minimum vertex cover.
 2. *endpoints of the maximum matching E_M is not a super set of the minimum vertex cover and Algorithm 2 traverses through E_M :* This case implies that there is some other maximum matching E'_M whose endpoints are a super set of the minimum vertex cover. In such cases, Algorithm 2 may traverse through E_M during an initial iteration. However, there will always be an iteration of BFS seeded on a vertex not an endpoint in E_M that will eventually ensure that Algorithm 2 traverses through E'_M , which implies that Algorithm 3 returns the minimum vertex cover.
 3. *endpoints of the maximum matching E_M is a super set of the minimum vertex cover and Algorithm 2 does not traverse through E_M :* This case may occur when the seed for BFS is not an endpoint of E_M . There are two sub cases here: (i) the graph consists of odd cycles and hence there is another maximum matching or maximal matching that Algorithm 2 traverses through and whose endpoints are a super set of minimum vertex cover. Here, Algorithm 3 returns the minimum matching cover. (ii) the another maximum matching that Algorithm 2 traverses through is *not* a super set of minimum vertex

⁷A perfect matching M_P matches all the vertices of a graph. Hence, $|M_P| = \frac{m}{2}$.

cover and hence Algorithm 3 does not return the minimum vertex cover. In such a case, the loop in Algorithm 1 will continue to iterate over different seeds of BFS and there always exists one seed that is an endpoint of an edge in E_M . This implies that Algorithm 2 will eventually traverse through E_M and Algorithm 3 will return the minimum vertex cover.

4. *endpoints of the maximum matching E_M is not a super set of the minimum vertex cover and Algorithm 2 does not traverse through E_M* : This case, in principle, is equivalent to point 2 but the ordering of the seeds selected for BFS are reversed. During the initial iterations, Algorithm 2 not traversing through E_M implies it traverses through another maximum matching or maximal matching whose endpoints is a super set of minimum vertex cover. In turn, this implies that Algorithm 3 returns the minimum vertex cover. For example, this happens when the input graph is a wheel graph. The maximum matching E_M may only consist of edges on the boundary. However, there is another maximum matching E'_M that consists of an edge whose one endpoint is the center vertex of the wheel graph and hence, collectively, whose endpoints are a super set of the minimum vertex cover. Note that, by design, Algorithm 2 in wheel graphs will never traverse through E_M .

- *maximal matching*: Depending on the seed of BFS selected in Algorithm 1, an iteration may result into Algorithm 2 selecting edges that form a maximal matching that is not necessarily a maximum matching (Lemma 3). This is why the third phase is called *maximal* matching. During such an iteration, there are two possibilities:

1. *endpoints of the maximal matching in Algorithm 2 is a super set of the minimum vertex cover*: This is a trivial case and Algorithm 3 returns the minimum vertex cover.
2. *endpoints of the maximal matching in Algorithm 2 is a not super set of the minimum vertex cover*: This case may occur even when the seed of BFS is a vertex that is an endpoint of an edge in maximum matching E_M . However, this seed is a terminal vertex (i.e. a vertex with degree = 1 in graph G). Hence, during one of the iterations of loop in Algorithm 1 when the the BFS is seeded on a non-terminal vertex that is an endpoint in E_M , one of the cases discussed in “maximum matching” will occur and Algorithm 3 will return the minimum vertex cover.

These cases complete the proof for this lemma. □

Lemma 9. *Algorithm 3 returns a minimum vertex cover.*

Proof. The proof follows due to a combination of Lemma 6, Lemma 7 and Lemma 8. Specifically, Lemma 6 proved that S is a vertex cover, Lemma 7

proved that S is the smallest vertex cover and Lemma 8 proved that there is at least one iteration where the input to Algorithm 3 consists of a super set of the minimum vertex cover and it returns a minimum vertex cover. Hence, as a combination of these lemmas, S is indeed the minimum vertex cover. \square

Overall, Lemma 9 means that if Algorithm 1 returns “Yes”, then the given instance of VC is a “Yes” instance. This completes the proof in the reverse direction. In turn, it completes the proof of Theorem 1.

6 Time Complexity Analysis

In this section, we discuss the time complexity of the algorithm (Table 2, Table 3, Table 4, Table 5). m denotes the number of vertices V and $n (\leq m^2)$ denotes the number of edges E .

In each table, we give the complexity of each line (each operation), the complexity of the loop (complexity of line multiplied by the number of loop iterations) and the dominant complexity. For convenience, the beginning of a loop, specifically the number of loop iterations, is highlighted (e.g., [Line 6](#) in Table 2). Each statement within the loop is prefixed with a pointer (\blacktriangleright). In case of nested loops, an additional pointer (\triangleright) is used.

Time complexity of Algorithm 4: We elaborate upon the time complexity of Algorithm 4 because the time complexity of the remainder of the algorithms is self-explanatory from the respective tables. In Algorithm 4, we have recursive calls (line 7 and line 10). However, by design, Algorithm 4 can be called m times only. This is because each time it is called, at least one vertex is either removed or frozen. Hence, after m calls, no unfrozen or unremoved vertex will exist. Each call takes $\mathcal{O}(m^2)$ time. Overall, in the worst case, height of recursion tree is m and each level has one subproblem taking $\mathcal{O}(m^2)$. Thus, total complexity is $\mathcal{O}(m) \cdot \mathcal{O}(m^2) = \mathcal{O}(m^3)$.

Theorem 2. *The asymptotic running time of Algorithm 1 is $\mathcal{O}(m^3n^2)$.*

Proof. Line 8 in Algorithm 1 dominates the complexity of all other lines as shown in Table 2. This dominant complexity is $\mathcal{O}(m^3n^2)$. Hence, the time complexity of the entire algorithm is $\mathcal{O}(m^3n^2)$. \square

On one hand, asymptotically, $\mathcal{O}(n) = \mathcal{O}(m^2)$. This is because the maximum number of edges (n) possible in a simple graph is $\frac{m \cdot (m-1)}{2}$, which is less than m^2 . On the other hand, asymptotically, $\mathcal{O}(n) = \mathcal{O}(m)$. This is because the minimum number of edges (n) needed in a connected graph is $m - 1$. In either case, the dominating time complexity discussed in Table 2 remains the same. In the worst case, it dominates time complexity of all lines. In the case of a sparse graph, it either dominates or is equivalent to time complexity of other lines. Hence, the time complexity stated in Theorem 2 holds.

Line Number	Line complexity	Loop complexity	Dominant complexity
1	$\mathcal{O}(m \cdot \log m)$	-	$\mathcal{O}(m \cdot \log m)$
2	$\mathcal{O}(m^2 n)$	-	$\mathcal{O}(m^2 n)$
3	$\mathcal{O}(1)$	-	$\mathcal{O}(m^2 n)$
4	$\mathcal{O}(1)$	-	$\mathcal{O}(m^2 n)$
5	-	-	$\mathcal{O}(m^2 n)$
6	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m^2 n)$
7	$\mathcal{O}(m + n)$	► $\mathcal{O}(m^2 + mn)$	$\mathcal{O}(m^2 n)$
8	$\mathcal{O}(m^2 n^2)$ [Table 3]	► $\mathcal{O}(m^3 n^2)$	$\mathcal{O}(m^3 n^2) = \mathcal{O}(m^7)$
9	$\mathcal{O}(m^4)$ [Table 4]	► $\mathcal{O}(m^5)$	$\mathcal{O}(m^3 n^2)$
10	$\mathcal{O}(1)$	► $\mathcal{O}(m)$	$\mathcal{O}(m^3 n^2)$
11	$\mathcal{O}(1)$	► $\mathcal{O}(m)$	$\mathcal{O}(m^3 n^2)$
12	-	-	$\mathcal{O}(m^3 n^2)$
13	-	-	$\mathcal{O}(m^3 n^2)$
14	$\mathcal{O}(1)$	-	$\mathcal{O}(m^3 n^2)$

Table 2: Line wise time complexity of Algorithm 1. A highlight denotes the number of loop iterations. A pointer (►) denotes that a line is within the loop. Without loss of generality, we assume the average length of vertex names is a constant and hence, ignore it in time complexity analysis of Line 1.

Line Number	Line complexity	Loop complexity	Dominant complexity
1	$\mathcal{O}(1)$	-	$\mathcal{O}(1)$
2	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$
3	$\mathcal{O}(1)$	► $\mathcal{O}(m^2)$	$\mathcal{O}(m^2)$
4	$\mathcal{O}(n^2)$	►▷ $\mathcal{O}(m^2 n^2)$	$\mathcal{O}(m^2 n^2)$
5	$\mathcal{O}(1)$	►▷ $\mathcal{O}(m^2)$	$\mathcal{O}(m^2 n^2)$
6	$\mathcal{O}(n^2)$	►▷ $\mathcal{O}(m^2 n^2)$	$\mathcal{O}(m^2 n^2)$
7	$\mathcal{O}(1)$	►▷ $\mathcal{O}(m^2)$	$\mathcal{O}(m^2 n^2)$
8	$\mathcal{O}(n^2)$	►▷ $\mathcal{O}(m^2 n^2)$	$\mathcal{O}(m^2 n^2)$
9	$\mathcal{O}(1)$	►▷ $\mathcal{O}(m^2)$	$\mathcal{O}(m^2 n^2)$
10	$\mathcal{O}(1)$	►▷ $\mathcal{O}(m^2)$	$\mathcal{O}(m^2 n^2)$
11	$\mathcal{O}(n^2)$	►▷ $\mathcal{O}(m^2 n^2)$	$\mathcal{O}(m^2 n^2)$
12	-	-	$\mathcal{O}(m^2 n^2)$
13	$\mathcal{O}(m)$	►▷ $\mathcal{O}(m^3)$	$\mathcal{O}(m^2 n^2)$
14	$\mathcal{O}(m + m^2)$	►▷ $\mathcal{O}(m^3 + m^4)$	$\mathcal{O}(m^2 n^2)$
15	$\mathcal{O}(n)$	►▷ $\mathcal{O}(m^2 n)$	$\mathcal{O}(m^2 n^2)$
16	$\mathcal{O}(m)$	►▷ $\mathcal{O}(m^3)$	$\mathcal{O}(m^2 n^2)$
17	-	-	$\mathcal{O}(m^2 n^2)$
18	-	-	$\mathcal{O}(m^2 n^2)$
19	$\mathcal{O}(1)$	-	$\mathcal{O}(m^2 n^2)$

Table 3: Line wise time complexity of Algorithm 2. A highlight denotes the number of loop iterations. A pointer (►) denotes that a line is within a loop. An additional pointer (▷) denotes a nested loop.

Line Number	Line complexity	Loop complexity	Dominant complexity
1	$\mathcal{O}(1)$	-	$\mathcal{O}(1)$
2	$\mathcal{O}(m)$	-	$\mathcal{O}(m)$
3	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$
4	$\mathcal{O}(m^2)$	$\blacktriangleright \mathcal{O}(m^3)$	$\mathcal{O}(m^3)$
5	-	-	$\mathcal{O}(m^3)$
6	$\mathcal{O}(m^3)$ [Table 5]	$\blacktriangleright \mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
7	-	-	$\mathcal{O}(m^4)$
8	-	-	$\mathcal{O}(m^4)$
9	-	-	$\mathcal{O}(m^4)$
10	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m^4)$
11	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
12	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
13	$\mathcal{O}(m)$	$\blacktriangleright \mathcal{O}(m^2)$	$\mathcal{O}(m^4)$
14	$\mathcal{O}(m^2)$	$\blacktriangleright \mathcal{O}(m^3)$	$\mathcal{O}(m^4)$
15	$\mathcal{O}(m^3)$ [Table 5]	$\blacktriangleright \mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
16	-	-	$\mathcal{O}(m^4)$
17	-	-	$\mathcal{O}(m^4)$
18	-	-	$\mathcal{O}(m^4)$
19	-	-	$\mathcal{O}(m^4)$
20	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m^4)$
21	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
22	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
23	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
24	$\mathcal{O}(m^3)$ [Table 5]	$\blacktriangleright \mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
25	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
26	-	-	$\mathcal{O}(m^4)$
27	$\mathcal{O}(m^2)$	$\blacktriangleright \mathcal{O}(m^3)$	$\mathcal{O}(m^4)$
28	$\mathcal{O}(m^3)$ [Table 5]	$\blacktriangleright \mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
29	$\mathcal{O}(m^2)$	$\blacktriangleright \mathcal{O}(m^3)$	$\mathcal{O}(m^4)$
30	$\mathcal{O}(m^3)$ [Table 5]	$\blacktriangleright \mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
31	$\mathcal{O}(1)$	$\blacktriangleright \mathcal{O}(m)$	$\mathcal{O}(m^4)$
32	$\mathcal{O}(m^3)$ [Table 5]	$\blacktriangleright \mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
33	-	-	$\mathcal{O}(m^4)$
34	-	-	$\mathcal{O}(m^4)$
35	-	-	$\mathcal{O}(m^4)$
36	$\mathcal{O}(1)$	-	$\mathcal{O}(m^4)$

Table 4: Line wise time complexity of Algorithm 3. A highlight denotes the number of loop iterations. A pointer (\blacktriangleright) denotes that a line is within a loop.

Line Number	Line complexity	Loop complexity	Dominant complexity
1	$\mathcal{O}(m + m)$	-	$\mathcal{O}(m)$
2	$\mathcal{O}(m)$	-	$\mathcal{O}(m)$
3	$\mathcal{O}(1)$	-	$\mathcal{O}(m)$
4	$\mathcal{O}(m^2)$	-	$\mathcal{O}(m^2)$
5	$\mathcal{O}(m + m)$	-	$\mathcal{O}(m^2)$
6	$\mathcal{O}(m^2)$	$\mathcal{O}(m)$	$\mathcal{O}(m^2)$
7	$\mathcal{O}(m^2)$	$\blacktriangleright \mathcal{O}(m^3)$	$\mathcal{O}(m^3)$
8	-	-	$\mathcal{O}(m^3)$
9	$\mathcal{O}(m^2)$	$\mathcal{O}(m)$	$\mathcal{O}(m^3)$
10	$\mathcal{O}(m^2)$	$\blacktriangleright \mathcal{O}(m^3)$	$\mathcal{O}(m^3)$
11	-	-	$\mathcal{O}(m^3)$
12	$\mathcal{O}(1)$	-	$\mathcal{O}(m^3)$

Table 5: Line wise time complexity of Algorithm 4. A highlight denotes the number of loop iterations. A pointer (\blacktriangleright) denotes that a line is within a loop.

7 Conclusion

We show that the VC problem can be solved efficiently. It implies that DiRe committees can be computed efficiently. Hence, achieving diversity and representation is more *efficient* than initially expected. Also, indeed, $P = NP$.

Broader Impact: We do not expect major, *immediate*, positive or negative, *practical* implications of this work. It is primarily because extrapolating our algorithm to elections where candidates are divided into arbitrarily sized arbitrary groups itself seems non-trivial (a.k.a. extrapolating our algorithm to hypergraphs itself seems non-trivial).

Acknowledgement

Blank for now.

References

- [Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.
- [Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

- [Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.
- [Kho19] Subhash Khot. On the proof of the 2-to-2 games conjecture. *Current Developments in Mathematics*, 2019(1):43–94, 2019.
- [Lev73] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

A Implementation of Algorithm

We give an example to explain the implementation of the entire algorithm. Additional examples can be found [here](#) and [here](#) (link will open to Google Slides).

Example 2. Consider the graph G shown in Figure 1. An instance of the VC problem consists of the graph G and an integer $k = 4$. The algorithm traverses through the graph as depicted from Figure 2 to Figure 25. The algorithm returns “YES” as the minimum size vertex cover shown in Figure 25 is of size 4.

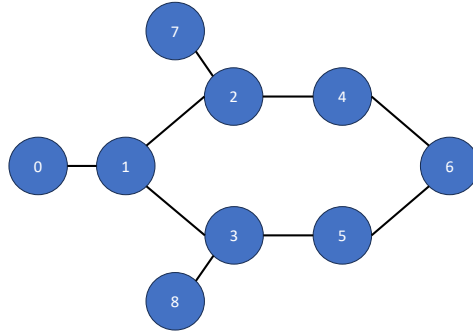


Figure 1: Example Graph G .

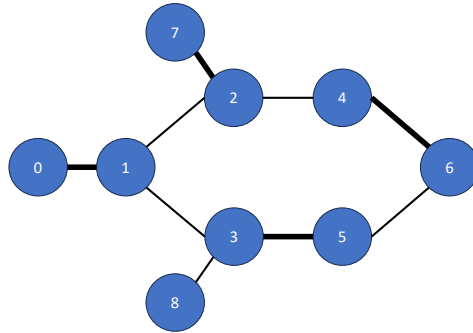


Figure 2: Bold edges $\{(0, 1), (2, 7), (3, 5), (4, 6)\}$ form a maximum matching of graph G .

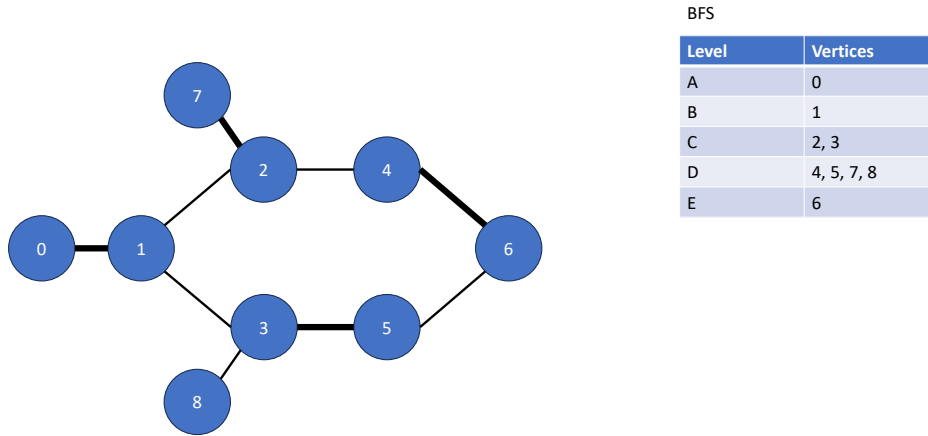


Figure 3: The “BFS” table lists the vertices at each level of the BFS (seeded on vertex ‘0’).

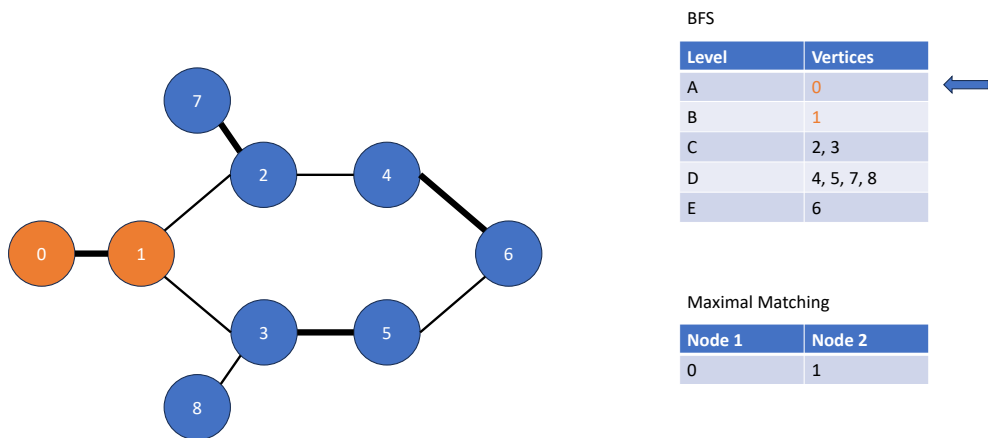


Figure 4: “Maximal Matching” table lists vertices 0 and 1 (orange vertices in graph G), which are the endpoints of the first edge selected during maximal matching. Each endpoint is marked as visited (orange font; BFS table).

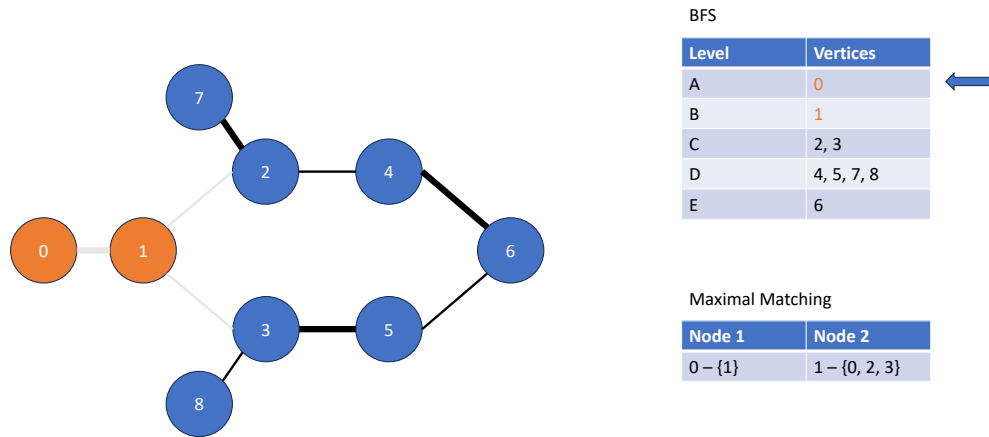


Figure 5: For each of the endpoints, namely 0 and 1, the respective curly brackets ($\{\}$) enlists the vertices connected to the corresponding vertex. Here, 0 is connected to $\{1\}$ and 1 is connected to $\{0, 2, 3\}$. In graph G , the grayed out edges represent the removed edges.

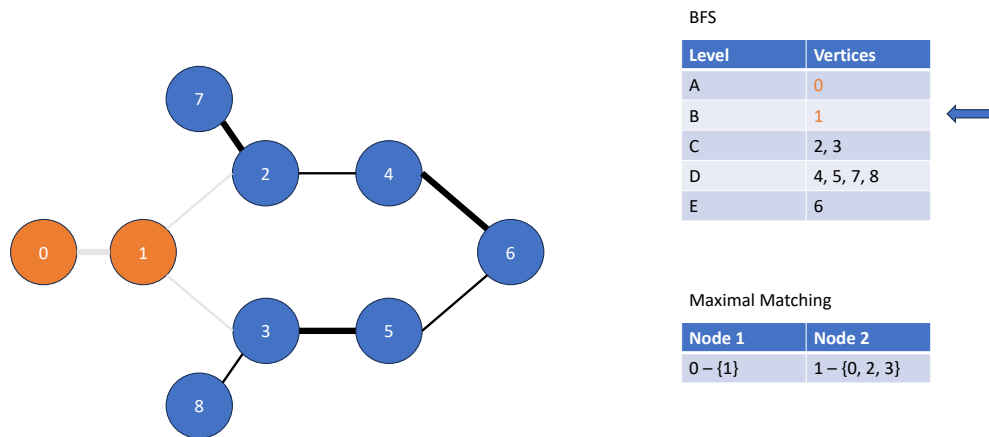


Figure 6: As all vertices on Level A of BFS table is visited, the pointer now is on Level B.

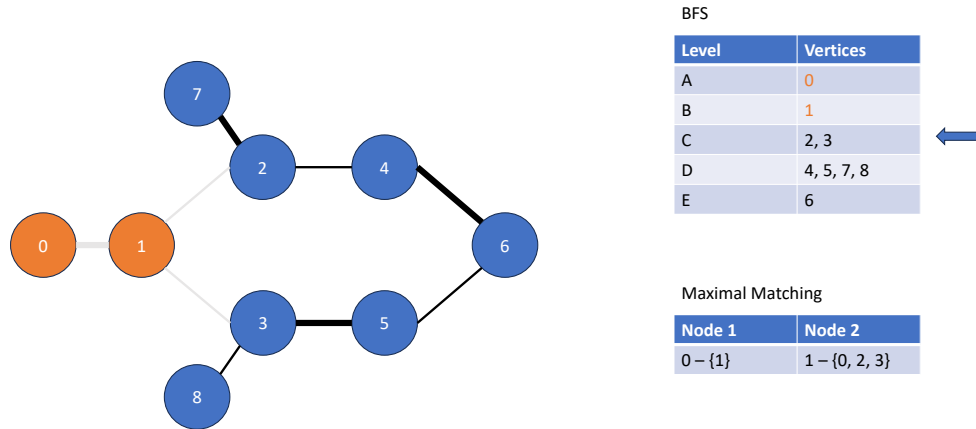


Figure 7: As all vertices on Level B of BFS table is visited, the pointer now is on Level C.

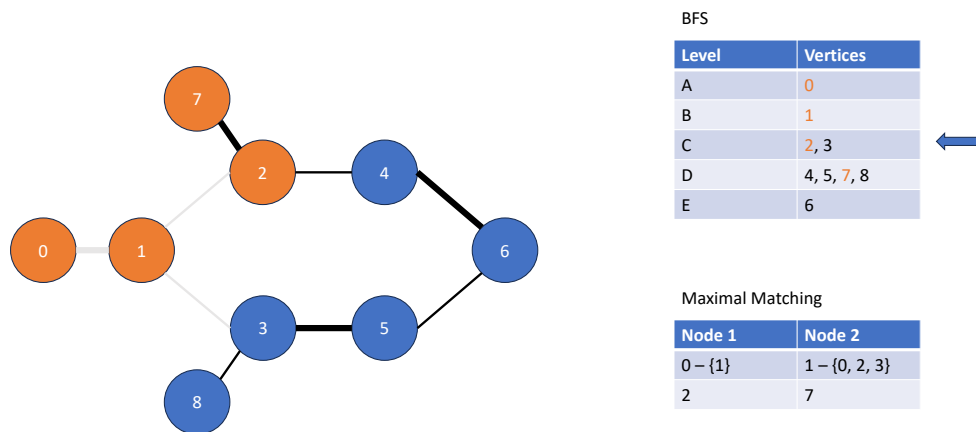


Figure 8: Vertex 2 comes before vertex 3 when sorted lexicographically. Hence, it is selected as one of the endpoints. As the edge connecting vertices 2 and 7 is part of maximum matching, it is preferred over edge connecting vertices 2 and 4. Hence, “Maximal Matching” table lists vertices 2 and 7, which are the endpoints of the second edge selected during maximal matching. Each endpoint is marked as visited (orange font; BFS table).

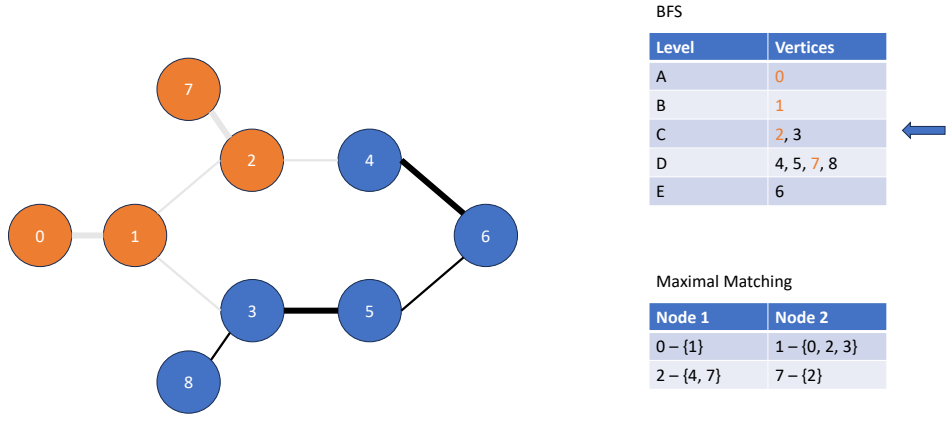


Figure 9: For each of the endpoints, namely 2 and 7, the respective curly brackets ($\{\}$) enlists the vertices connected to the corresponding vertex via an un-removed edge. Here, 2 is connected to $\{4, 7\}$ and 7 is connected to $\{2\}$. The corresponding edges are removed (grayed out).

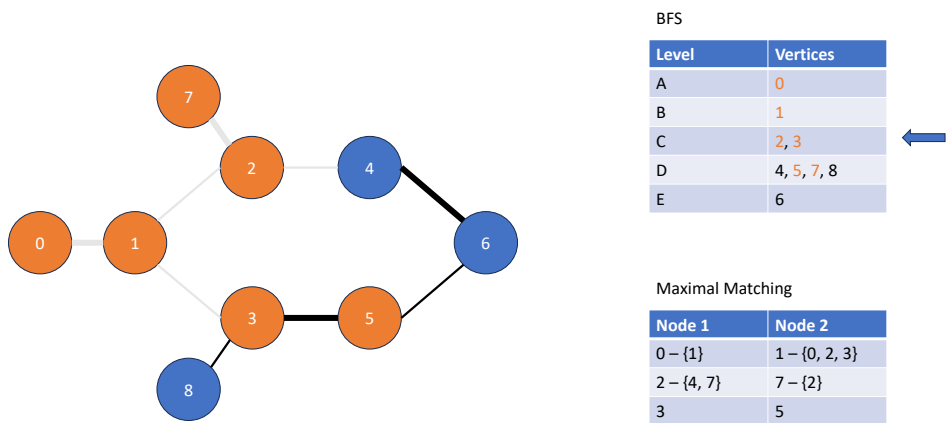


Figure 10: As the edge connecting vertices 3 and 5 is part of maximum matching, it is preferred over edge connecting vertices 3 and 8. Hence, “Maximal Matching” table lists vertices 3 and 5, which are the endpoints of the third edge selected during maximal matching. Each endpoint is marked as visited (orange font; BFS table).

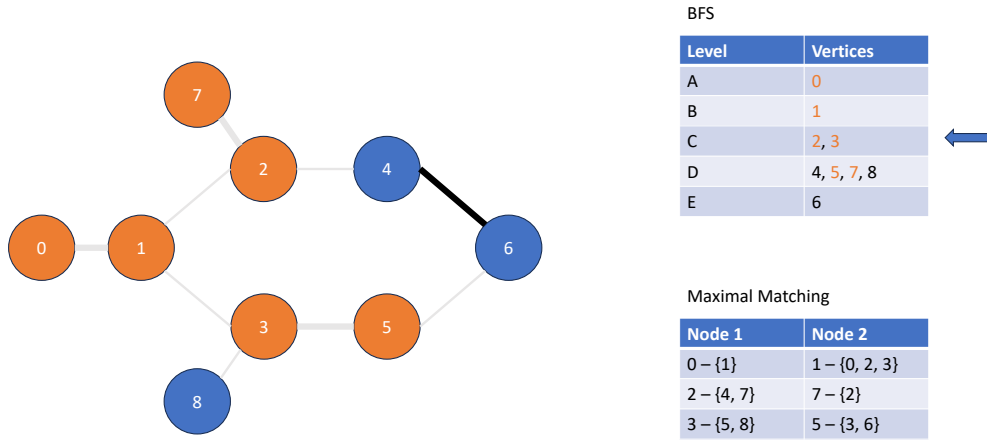


Figure 11: For each of the endpoints, namely 3 and 5, the respective curly brackets ($\{\}$) enlists the vertices connected to the corresponding vertex via an unremoved edge. Here, 3 is connected to $\{5, 8\}$ and 5 is connected to $\{3, 6\}$. The corresponding edges are removed (grayed out).

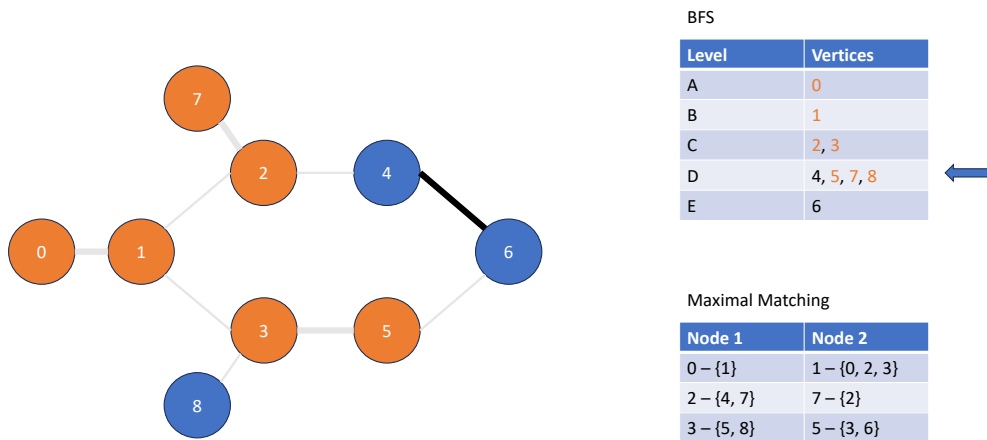


Figure 12: Vertex 8, which now has no unremoved edges, is marked as visited (orange font; BFS table). As all vertices on Level C of BFS table is visited, the pointer now is on Level D.

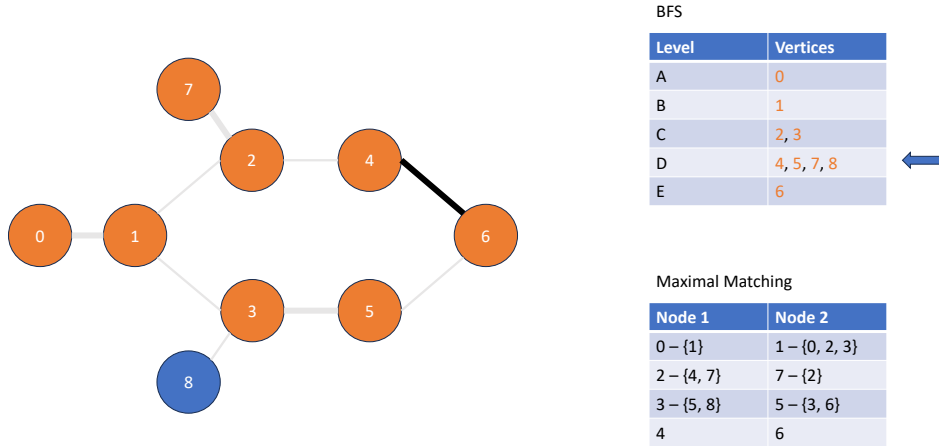


Figure 13: The edge connecting vertices 4 and 6, which is part of maximum matching, is the only remaining edge. Hence, “Maximal Matching” table lists vertices 4 and 6, which are the endpoints of the fourth and final edge selected during maximal matching. Each endpoint is marked as visited (orange font; BFS table).

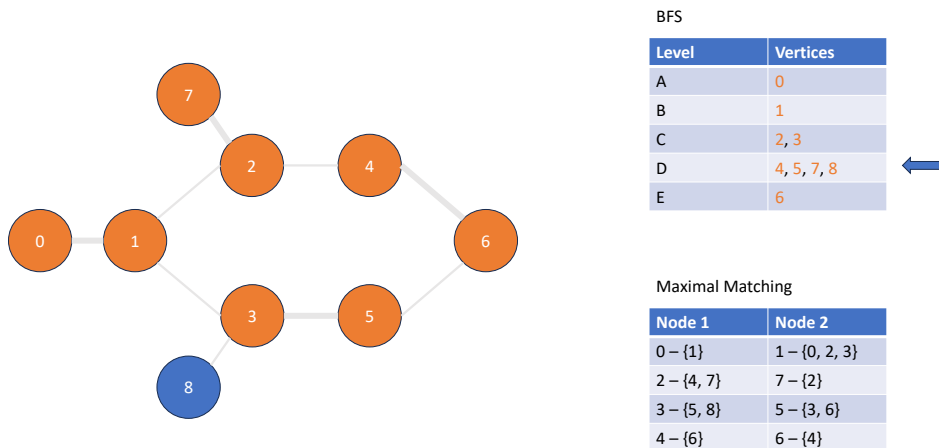


Figure 14: For each of the endpoints, namely 4 and 6, the respective curly brackets ({}) enlists the vertices connected to the corresponding vertex via an unremoved edge. Here, 4 is connected to {6} and 6 is connected to {4}. The corresponding edge is removed (grayed out).

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 15: The “Maximal Matching” table will be used for “Local Minimization” phase of the algorithm. Vertices {0, 1, 2, 3, 4, 5, 6, 7} are labeled as eponymous “endpoint” vertices.

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 16: Vertex 3 is frozen (highlighted yellow) as it represents vertex 8, which is not an “endpoint” vertex. By default, the represents list of a frozen vertex (here, vertex 3) is removed (not shown here for convenience).

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 17: Vertex 0 in row 1 is removed (grayed out) as it represents no other vertex except its same-row neighbor (namely vertex 1). Consequently, vertex 1 in row 1 is frozen (highlighted yellow) as it now represents a removed vertex (namely vertex 0).

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 18: Vertex 7 in row 2 is removed (grayed out) as it represents no other vertex except its same-row neighbor (namely vertex 2) and is not represented by any vertex in rows above it. Consequently, vertex 2 in row 2 is frozen (highlighted yellow) as it now represents a removed vertex (namely vertex 7).

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2 , 3 }
2 – {4, 7}	7 – { 2 }
3 – {5, 8}	5 – { , 6}
4 – {6}	6 – {4}

Figure 19: The frozen vertices 1, 2, and 3 are removed (grayed out) from “represents” list (curly brackets) of each vertex, wherever applicable. Here, vertices 2 and 3 are removed from represents list of vertex 1 and vertex 3 is removed from represents list of vertex 5.

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2 , 3 }
2 – {4, 7}	7 – { 2 }
3 – {5, 8}	5 – { , 6}
4 – {6}	6 – {4}




Figure 20: Arrow depicts the bottom-up elimination of vertices.

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – { , 6}
4 – {6}	6 – {4}

Figure 21: Start with the last row. Given that both the vertices represent only each other, we freeze one and remove the other. Specifically, vertex 4 is not represented by any vertex (or equivalently it is represented by frozen vertex 2) and vertex 6 is represented by non-frozen vertex 5. Hence, vertex 4 is removed (grayed out) and vertex 6 is frozen (yellow highlight).

Local Minimization

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 22: The frozen vertex 6 is removed (grayed out) from “represents” list of each vertex, wherever applicable. Here, it is removed from represents list of vertex 5. Consequently, vertex 5 does not represent any vertex. Hence, it is also removed (grayed out).

Smallest Vertex Cover = {1, 2, 3, 6}

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 23: Only frozen vertices remain in the table. The local minimization phase terminates. The frozen vertices form the smallest vertex cover for the iteration of the algorithm whose BFS is seeded on vertex ‘0’.

Minimum Vertex Cover = {1, 2, 3, 6}

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
3 – {5, 8}	5 – {3, 6}
4 – {6}	6 – {4}

Figure 24: The size of the smallest vertex cover (= 4) is equivalent to the size of maximum matching. Hence, the smallest vertex cover is indeed the minimum vertex cover and the algorithm terminates early.

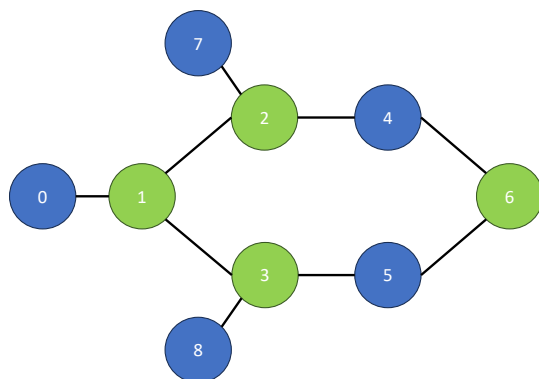


Figure 25: Vertices $\{1, 2, 3, 6\}$ form the minimum vertex cover of size 4.