On Efficient Computation of DiRe Committees

Kunal Relia*

October 01, 2025

Abstract

Consider a committee election consisting of (i) a set of candidates who are divided into arbitrary groups each of size *at most* 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 *at most* two candidates and a representation constraint that stipulates the selection of *at least* one candidate from each population who has a non-null 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 propose an algorithm that is an amalgamation of maximum matching, breadth-first search, maximal matching, and local minimization. We prove the algorithm terminates in polynomial-time. We conjecture the algorithm is an unconditional deterministic polynomial-time algorithm.

*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

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; for details, see Appendix A). 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 $e \in E$, $e \cap S \neq \phi$. The corresponding computational problem of finding the minimum-size vertex cover (MVC) is NP-complete[1] [Coo71, Lev73, Kar72], which means that there is no *known* deterministic polynomial-time algorithm to solve MVC. Here, we conjecture an unconditional deterministic polynomial-time algorithm for MVC on unweighted simple connected graphs[2].

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 ?*

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.

[1] Strictly 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.

[2] We 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 (Appendix C). 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.

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 that 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', $M \not\subset M'$.*

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 (Represents[3])**.** *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.*

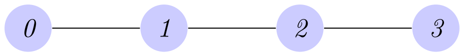
Definition 8 (Represents Table)**.** *A represents table R is a 2-column table that stores the endpoints of edges selected during maximal matching and the vertices each endpoint represents.*

[3]The 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.

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.*

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 may use Lemma 3 in the proof of correctness 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 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, we conjecture that 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

V_s = lexicographically sorted vertices
E_M = maximum matching found using the Blossom Algorithm [Edm65]
if $k < |E_M|$ **then**
  return "NO"
end
for each $v \in V_s$ **do**
  BFS_{level} = an array of arrays storing sorted vertices at each level of breadth-first search tree seeded on v
  R = `Maximal_Matching`$(G,\ E_M,\ BFS_{level})$
  S = `Local_Minimization`(R)
  if $|S| \leq k$ **then**
    return "YES"
  end
end
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

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
for each *level* in BFS_{level} **do**
    while *there is an unvisited vertex in level* **do**
        if *there exists an edge that connects two vertices on the same level and is in* E_M **then**
            select the edge
        else if *there exists an edge that connects two vertices on the same level and is not in* E_M **then**
            select the edge
        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**
            select the edge
        else
            select the edge that connects one vertex on the current level with another vertex on the next level and is not in E_M
        end
        Mark the two endpoints of the selected edge as visited in BFS_{level}
        Append after the last row of R the two endpoints of the selected edge and the respective vertices each endpoint represents
        Remove from graph G the selected edge and all the edges that are connected to the two endpoints
        If any vertex becomes edgeless in G, mark the vertex as visited in BFS_{level}
    end
end
return R

Algorithm 3: `Local_Minimization`(R)

Data: Represents Table R

Result: returns S - the smallest vertex cover

```
S = ϕ
P = set of endpoints in R selected during maximal matching
for each endpoint vertex v in R do
    if v represents at least one vertex not in P then
        //freeze vertex v but do not remove any vertex from R
        R, S = Freeze_and_Remove(R, S, v, ϕ)
    end
end
// The following for loop will traverse through the table R top-down
for each row in R do
    if if any one endpoint in row is either frozen or removed then
        continue
    else if one endpoint u in row only represents another endpoint vertex v in row and v
      represents more than one vertex then
        if u is not represented by any endpoint in R other than v then
            R, S = Freeze_and_Remove(R, S, v, u)
        end
    end
end
// The following for loop will traverse through the table R bottom-up
for each row in R do
    if (if both endpoints are frozen) or (one endpoint is frozen and one is removed) then
        continue
    else if endpoint u remains and endpoint v is removed then
        R, S = Freeze_and_Remove(R, S, u, ϕ)
    else
        //at this point, both endpoints u and v in row represent exactly one vertex, namely
          each other
        if u is represented by more endpoints in R than v then
            R, S = Freeze_and_Remove(R, S, u, v)
        else if v is represented by more endpoints in R than u then
            R, S = Freeze_and_Remove(R, S, v, u)
        else
            R, S = Freeze_and_Remove(R, S, u, v)
        end
    end
end
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 = `Freeze_and_Remove`(R, S, *endpoint*, ϕ)
8: **end**
9: **for each** non-frozen and unremoved *endpoint* in R that does not represent any vertex **do**
10: | R, S = `Freeze_and_Remove`(R, S, ϕ, *endpoint*)
11: **end**
12: return R, S

We conjecture the following:

Conjecture 1. *Algorithm 1 returns "Yes" if and only if a given instance of VC is a "Yes" instance.*

5 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 (▶). In the case of nested loops, an additional pointer (▷) 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 at most m times only. This is because each time it is called, at least one vertex is either removed or frozen. Hence, after at most m calls, no unfrozen or unremoved vertex will exist. Each call takes $\mathcal{O}(m^2)$ time. Overall, in the worst case, the height of the 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 1. *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 $\mathcal{O}(m^3n^2)$. □

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 the time complexity of all lines. In the case of a sparse graph, it either dominates or is equivalent to the time complexity of other lines. Hence, the time complexity stated in Theorem 1 holds.

6 Conclusion

We conjecture that the VC problem can be solved efficiently. It implies that DiRe committees can be computed efficiently. Hence, achieving diversity and representation may be more *efficient* than initially expected.

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	$\mathcal{O}(m^2)$	▶ $\mathcal{O}(m^3)$	$\mathcal{O}(m^3)$
5	-	-	$\mathcal{O}(m^3)$
6	$\mathcal{O}(m^3)$ [Table 5]	▶ $\mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
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	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
12	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
13	$\mathcal{O}(m)$	▶ $\mathcal{O}(m^2)$	$\mathcal{O}(m^4)$
14	$\mathcal{O}(m^2)$	▶ $\mathcal{O}(m^3)$	$\mathcal{O}(m^4)$
15	$\mathcal{O}(m^3)$ [Table 5]	▶ $\mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
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	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
22	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
23	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
24	$\mathcal{O}(m^3)$ [Table 5]	▶ $\mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
25	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
26	-	-	$\mathcal{O}(m^4)$
27	$\mathcal{O}(m^2)$	▶ $\mathcal{O}(m^3)$	$\mathcal{O}(m^4)$
28	$\mathcal{O}(m^3)$ [Table 5]	▶ $\mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
29	$\mathcal{O}(m^2)$	▶ $\mathcal{O}(m^3)$	$\mathcal{O}(m^4)$
30	$\mathcal{O}(m^3)$ [Table 5]	▶ $\mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
31	$\mathcal{O}(1)$	▶ $\mathcal{O}(m)$	$\mathcal{O}(m^4)$
32	$\mathcal{O}(m^3)$ [Table 5]	▶ $\mathcal{O}(m^4)$	$\mathcal{O}(m^4)$
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 (▶) 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	$\mathcal{O}(m^2)$	►$\mathcal{O}(m^3)$	$\mathcal{O}(m^3)$
8	-	-	$\mathcal{O}(m^3)$
9	$\mathcal{O}(m^2)$	$\mathcal{O}(m)$	$\mathcal{O}(m^3)$
10	$\mathcal{O}(m^2)$	►$\mathcal{O}(m^3)$	$\mathcal{O}(m^3)$
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 (►) denotes that a line is within a loop.

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 DiRe Committee and Vertex Cover

We formally show the equivalence between the DiRe Committee Feasibility problem and the Vertex Cover problem on unweighted, undirected graphs.

Definition 11 (DiRe Committee Feasibility Problem (DiReCF))**.** *We are given an instance of a committee election consisting of (i) a set of candidates C who are divided into arbitrary groups $R \in \mathcal{R}$ each of size at most two and a diversity constraint l_R that stipulates the selection of at least one candidate from each non-empty group ($l_R = 1$ for all $R \in \mathcal{R}$ where $|R| > 0$, $l_R = 0$ otherwise) and (ii) a set of voters O who are divided into arbitrary populations $P \in \mathcal{P}$ each approving at most two candidates W_P and a representation constraint l_P that stipulates the selection of at least one candidate from each population who has a non-empty set of approved candidates ($l_P = 1$ for all $P \in \mathcal{P}$ where $|W_P| > 0$, $l_P = 0$ otherwise).*

Given a committee size k that is a non-negative integer, the goal of DiReCF is to determine whether there is a committee W of size at most k that satisfies the given constraints such that $|R \cap W| \geq l_R$ for all $R \in \mathcal{R}$ and $|W_P \cap W| \geq l_P$ for all $P \in \mathcal{P}$?

To keep this section standalone, we again define the vertex cover problem:

Definition 12 (Vertex Cover Problem (VC))**.** *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 $e \in E$, $e \cap S \neq \phi$.*

Given a non-negative integer k, the goal of VC is to determine whether the graph G has a vertex cover S of size at most k?

We now show that DiReCF and VC on unweighted, undirected graphs are equivalent by (i) reducing VC to DiReCF and (ii) reducing DiReCF to VC.

Theorem 2. *DiReCF and VC are equivalent.*

Proof. We first give a polynomial-time reduction from VC to DiReCF.

VC $\leq_P$ DiReCF: We reduce an instance of vertex cover (VC) problem to an instance of DiReCF. We have one candidate $c_i \in C$ for each vertex $v_i \in V$. We have one candidate group $R \in \mathcal{R}$ consisting of two candidates c_i and c_j for each edge $e \in E$ that connects vertices v_i and v_j. For each candidate group $R \in \mathcal{R}$, we set the diversity constraint l_R to one. Additionally, for each edge $e \in E$ that connects vertices v_i and v_j, we have a population of voters $P \in \mathcal{P}$ who approve of two candidates c_i and c_j in W_P. For each voter population $P \in \mathcal{P}$, we set the representation constraint l_P to one. Finally, we set the target committee size to be k.

We have a vertex cover of size at most k if and only if there is a committee of size at most k that satisfies all the constraints.

($\Rightarrow$) If an instance of the vertex cover problem is a yes instance, then the corresponding instance of DiReCF is a yes instance. This is because if there is a vertex cover S of size k, then for each vertex $v_i \in S$, we have a candidate c_i in committee W who is in one or more candidate groups and is among the approved candidates for one or more populations. As each edge is covered by the vertex cover S, at least one candidate from each candidate group and from each voter populations' approved candidates is present in the committee W of size k.

($\Leftarrow$) If there is a committee W of size k that satisfies all the constraints, then there is a vertex cover S of size k. This is because for each $c_i \in W$, there is a vertex $v_i \in S$. Given that all constraints are satisfied by W, it implies all edges are covered by the vertex cover.

DiReCF $\leq_P$ VC: We reduce an instance of DiReCF problem to an instance of the vertex cover (VC) problem. We have one vertex $v_i \in V$ for each candidate $c_i \in C$. Next, we have an edge $e \in E$ for the following scenarios:

- for each candidate group $R \in \mathcal{R}$ that has candidates c_i and c_j, we have an edge that connects v_i and v_j.
- for each candidate group $R \in \mathcal{R}$ that has only one candidate c_i, we have an edge that connects v_i with v_i. Basically, we have a loop.
- for each voter population $P \in \mathcal{P}$ that approves of candidates c_i and c_j, we have an edge that connects v_i and v_j.

- for each voter population $P \in \mathcal{P}$ that approves only one candidate c_i, we have an edge that connects v_i with v_i. We again have a loop.

For the cases described above, we have the diversity constraint $l_R = 1$ for all candidate groups $R \in \mathcal{R}$ where $|R| > 0$. We have the representation constraint $l_P = 1$ for all voter populations $P \in \mathcal{P}$ where $|W_P| > 0$. The constraints correspond to the requirement that each edge must be covered ($e \cap S \neq \phi$). We do nothing for candidate groups of size zero and for voter populations who do not approve of any candidates. The corresponding constraints are set to zero and are henceforth ignored. Finally, we set the target committee size and the size of the vertex cover to k.

We have a committee of size at most k that satisfies all the constraints if and only if there is a vertex cover of size at most k.

($\Rightarrow$) If there is a committee W of size k that satisfies all the constraints, then for each candidate $c_i \in W$, there is a vertex v_i in the vertex cover S of size k. This is because we know that $|R \cap W| \geq l_R$ for all candidate groups $R \in \mathcal{R}$ and $|W_P \cap W| \geq l_P$ for all voter populations $P \in \mathcal{P}$. It implies that $|e \cap S| \geq 1$ for all edges $e \in E$, which means $e \cap S \neq \phi$.

($\Leftarrow$) If there is a vertex cover S of size k, then there is a committee W of size k that satisfies all the constraints. Each edge covered by a vertex in S implies each constraint being satisfied by a candidate in W.

□

In summary, as VC $\leq_P$ DiReCF and DiReCF $\leq_P$ VC, the two problems are equivalent and can be used interchangeably. For technical simplicity, the paper uses VC instead of DiReCF.

B Related Work

All NP-complete problems are "equivalent" from the perspective of computational complexity theory. Hence, any progress toward finding an efficient algorithm for any one NP-complete problem will have an impact on each and every NP-complete problem. However, there are thousands of known NP-complete problems and a literature review on each one of them is beyond the scope of this paper. Therefore, we focus our discussion on the literature review of the vertex cover problem. Specifically, we elaborate upon how our algorithm is fundamentally different from previous work on the vertex cover problem.

B.1 Approximation Algorithms and Restricted Graphs

While there is an extremely rich line of work discussing the (i) hardness and hardness of approximation of the vertex cover problem, (ii) finding approximation algorithms[4] for restricted cases (e.g., graphs with bounded degree) and (iii) finding exact algorithms for restricted cases (e.g., bipartite graphs), there is no work initiated to find an exact algorithm for the vertex cover problem on graphs for which the problem is NP-complete[5]. Hence, to the best of our knowledge, there is no prior work relevant to our approach. Additionally, the paper builds upon the common fact that the endpoints of a maximal matching of a graph form a vertex cover.

B.2 Parameterized Complexity

Broadly speaking, parameterized complexity and in particular, fixed-parameter tractability is the study of the complexity of computational problems *conditioned* on one or more parameters. In contrast, our algorithm is *unconditional*. Moreover, our work does not build upon any known parameterized algorithms.

B.3 Blossom Algorithm

We used the Blossom algorithm for our implementation. Hence, we cite it and not papers that improve upon the Blossom algorithm (e.g., a faster algorithm for maximum matching due to Micali and Vazirani[6]). Moreover, the time complexity of the Blossom algorithm has no impact on the overall time complexity of the algorithm presented in our paper. Hence, implementing a faster algorithm for maximum matching is not needed.

[4] By "algorithms", we mean a polynomial-time (efficient) algorithm unless and until noted otherwise.

[5] Approximation algorithms are actually for NP-hard problems. In this discussion, we use NP-hardness and NP-completeness interchangeably.

[6] `https://ieeexplore.ieee.org/document/4567800` (last accessed: February 8, 2024)

C Vertex Cover on Unweighted Simple Connected Graphs

We now prove that the vertex cover (VC) problem on unweighted simple connected graphs is NP-complete.

Definition 13 (Simple Graph)**.** *A graph $G = (V, E)$ is said to be a simple graph if the graph (i) is undirected, (ii) has no loops, i.e., it has no edge that starts and ends at the same vertex and (iii) does not have more than one edge between any pair of vertices.*

Definition 14 (Connected Graph)**.** *A graph $G = (V, E)$ is said to be a connected graph if, for each pair of vertices, there exists a path that connects the pair of vertices.*

Theorem 3. *The vertex cover (VC) problem on unweighted simple connected graphs is NP-complete.*

Proof. We first show the problem's membership in NP and then proceed to reduce from a known NP-hard problem.

Membership in NP: The vertex cover (VC) problem on unweighted simple connected graphs is in NP. Given a candidate solution and an integer k, we can easily verify if the solution is a vertex cover of size at most k.

NP-hardness: We reduce from a known NP-hard problem, namely the vertex cover (VC) problem on unweighted undirected graphs. Specifically, we reduce an instance of the vertex cover problem on unweighted undirected graphs (VC1) to an instance of the vertex cover problem on unweighted simple connected graphs (VC2)[7].

For each vertex $v_i \in V$ in VC1, there is a vertex $v'_i \in V'$ in VC2. Next, for the edges, we have the following scenarios:

- there is an edge $e \in E$ in VC1 that connects two distinct vertices v_i and v_j: there is a corresponding edge $e' \in E'$ in VC2 that connects two distinct vertices v'_i and v'_j.
- there are multiple edges in VC1 that connects two distinct vertices v_i and v_j: there is one edge $e' \in E'$ in VC2 that connects two distinct vertices v'_i and v'_j.
- there is an edge in VC1 that loops over the same vertex v_i: create a dummy vertex $d'_i \in D'$ and then, there is an edge $e' \in E'$ in VC2 that connects the vertex v'_i with the dummy vertex d'_i. Overall, for each loop in VC1, there is a dummy vertex created in VC2.

Next, there is one dummy vertex $u' \in U'$ in VC2 that is connected to each vertex $v' \in V'$ and dummy vertex $d' \in D'$. Specifically, for each pair of vertices consisting of u', there is a dummy edge $f' \in F'$ that connects the pair of vertices. In summary, the vertices in VC2 consist of a union of the following: $V' \cup D' \cup U'$. The edges in VC2 consist of a union of the following: $E' \cup F'$. Finally, we set the vertex cover size in VC2 to be at most $k + 1$.

It remains to be proven that there is a vertex cover on unweighted undirected graph of size at most k if and only if there is a vertex cover on unweighted simple connected graph of size at most $k + 1$.

($\Rightarrow$) If there is a vertex cover S of size k in an instance of VC1, then for each vertex $v_i \in S$, we have a vertex v'_i in the vertex cover S' of VC2. S' covers all edges $e' \in E'$ of VC2. Additionally, dummy vertex u' is always in the vertex cover S', which covers all the edges $f' \in F'$. Consequently, the size of the vertex cover of VC2 is $k + 1$.

($\Leftarrow$) The instance of the VC2 problem is a yes instance when each and every edge is covered. Then the corresponding instance of the VC1 problem is a yes instance as well. More specifically, there are the following cases when the instance of the VC2 problem can be a yes instance, i.e., it has a vertex cover S' of size $k + 1$:

1. vertex cover S' consists of zero dummy vertex from D', k vertices from V', one vertex u' - This is a trivial case and the instance of the VC1 problem will have vertex cover S consisting of vertex v_i for every vertex $v'_i \in S'$. This will be of size k.
2. vertex cover S' consists of x[8] dummy vertex from D', $k - x$ vertices from V', one vertex u' - For each dummy vertex $d' \in D'$ selected, the corresponding vertex $v'_i \in V'$ connected to the

[7]The terms VC1 and VC2 are used in this reduction only.

[8]variable x is an integer such that $1 \leq x \leq k$.

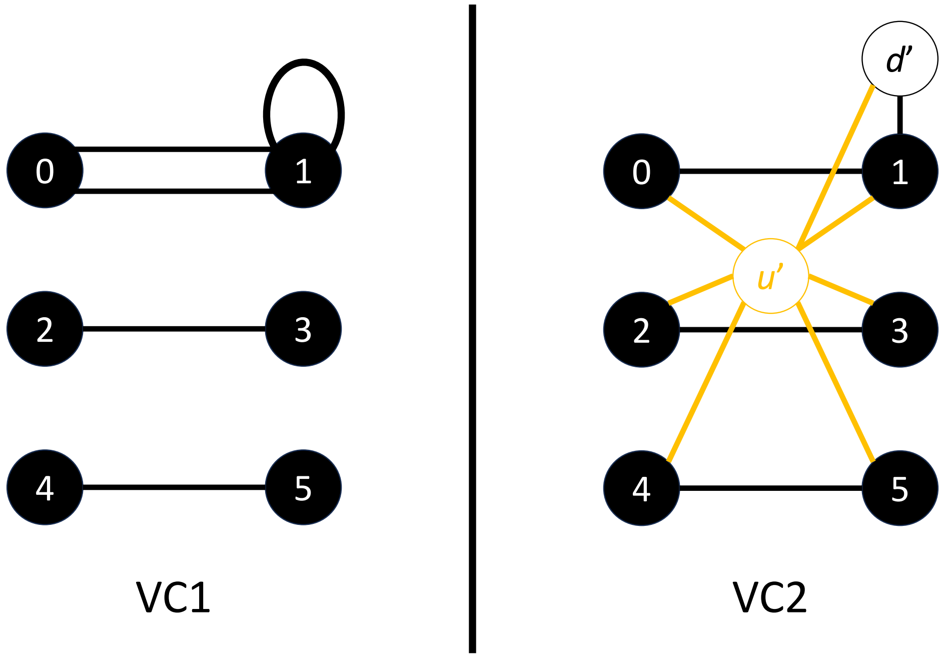


Figure 1: VC1 denotes an instance of the vertex cover problem on unweighted, undirected graph. VC2 denotes an instance of the vertex cover problem on unweighted simple connected graph. The multi-edges connecting vertices 0 and 1 in VC1 are removed in VC2. The loop connecting vertex 1 to itself is replaced by an edge in VC2 that connects vertex 1 to a dummy vertex d'. Another dummy vertex u' (yellow vertex) is added to VC2 and is connected to all existing vertices to make the graph connected.

dummy vertex is not selected. Hence, given that the dummy vertex is of degree one, it can be swapped with the vertex it is connected to. This won't have any effect on the validity of the vertex cover S'. In summary, x dummy vertices from D' in vertex cover S' are replaced by the corresponding x vertices from V'. Consequently, an instance of the VC1 problem will have vertex cover S consisting of vertex v_i for every vertex $v_i' \in S'$. This will be of size k.

3. vertex cover S' consists of zero dummy vertex from D', $k+1$ vertices from V' (vertex u' is not selected) - This case may arise when VC2 is a complete graph. Specifically, when VC2 is a complete graph, it does not consist of any vertex in D'. Moreover, the vertex cover S' of VC2 is equivalent to the vertex set V'. Hence, we can replace any vertex from S' with dummy vertex u' and the instance of VC2 still remains a yes instance. Formally, the new vertex cover S'' will consist of $\{S' \setminus \{v'\}\} \cup \{u'\}$ for some $v' \in V'$. Hence, for every $v' \in S''$ where $v' \in V'$, there is a corresponding $v \in S$ in VC1. The vertex cover S in VC1 is of size k as the new vertex cover S'' consists of k vertices from V'.

4. vertex cover S' consists of zero dummy vertex from D', zero vertices from V' and one vertex

Dummy vertices D'	Vertices V'	Dummy vertex U'	Case
✗	✗	✗	Not possible
✗	✓	✗	Case 3
✓	✗	✗	Not possible
✓	✓	✗	Not possible
✗	✗	✓	Case 4
✗	✓	✓	Case 1
✓	✗	✓	Case 2
✓	✓	✓	Case 2

Table 6: A summary of different possibilities of presence (✓) and absence (✗) of vertices from each set of vertices in the minimum vertex cover S' in an instance of VC2. Each Case corresponds to an instance of VC2 being a yes instance in the proof of correctness in the reverse direction for Theorem 3.

u' - In such a case, one endpoint of all edges in VC2 is u'. Hence, the corresponding instance of VC1 contains no edges and its vertex cover will be a null set.

Finally, note that no other cases can lead to a yes instance of VC2 (e.g., S' is not a vertex cover if S' consists of, for example, x dummy vertices from D', $k-x+1$ vertices from V' and zero vertex from U').

This completes the other direction of the proof of correctness. In turn, this completes the entire proof. □

D 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 2. 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 3 to Figure 26. The algorithm returns "YES" as the minimum size vertex cover shown in Figure 26 is of size 4.*

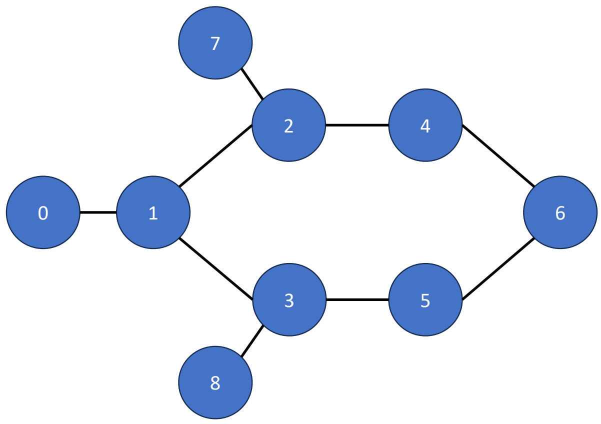


Figure 2: Example Graph G.

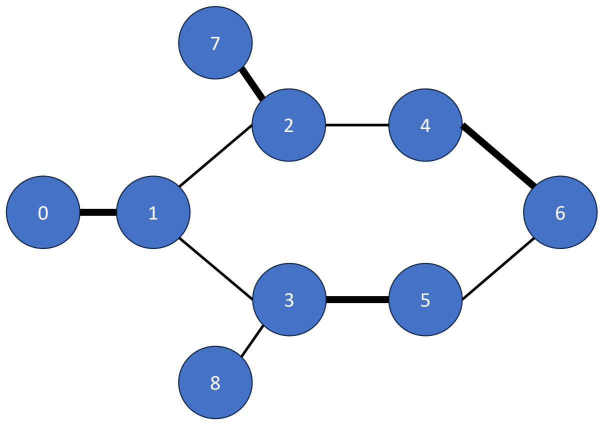
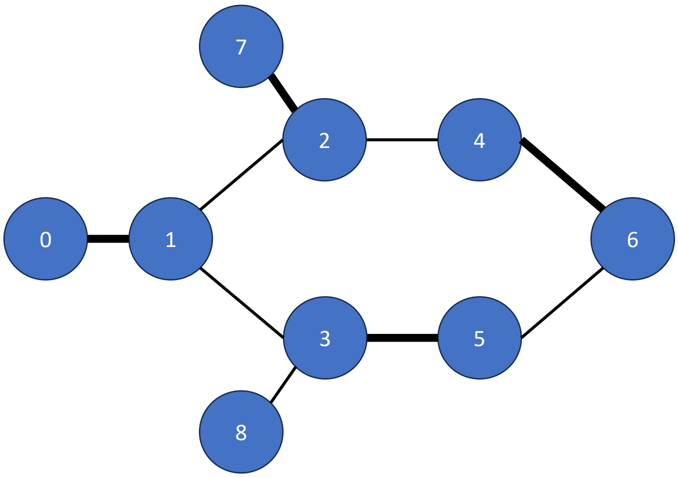


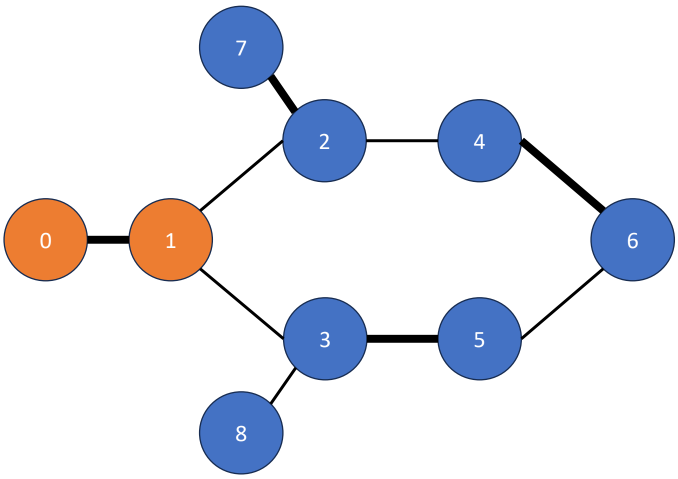
Figure 3: Bold edges $\{(0, 1), (2, 7), (3, 5), (4, 6)\}$ form a maximum matching of graph G.



BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Figure 4: The "BFS" table lists the vertices at each level of the BFS (seeded on vertex '0').



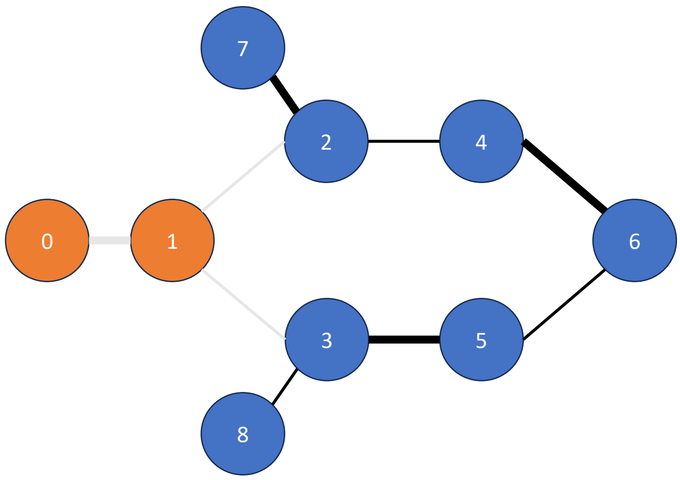
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

Node 1	Node 2
0	1

Figure 5: "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).



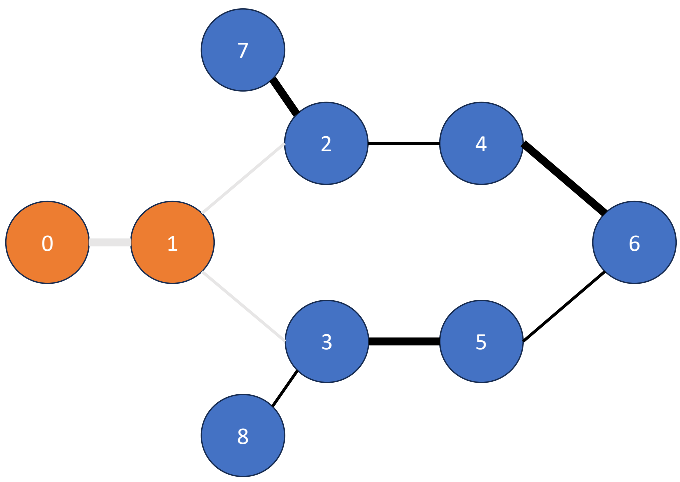
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}

Figure 6: 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.



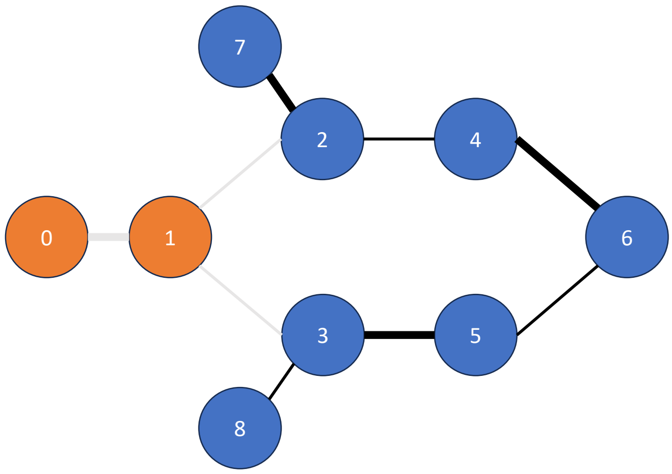
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}

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



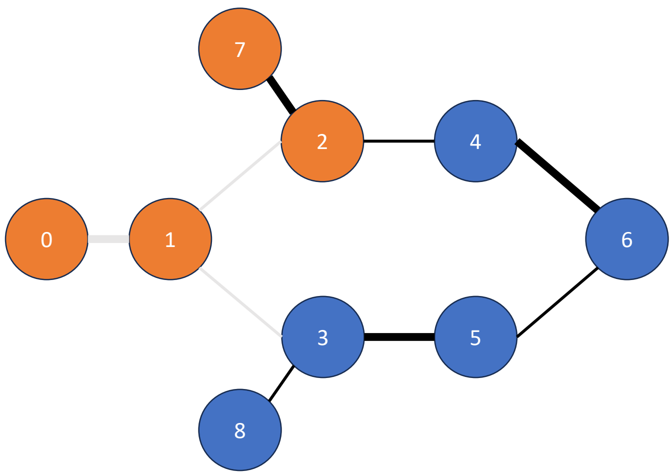
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}

Figure 8: As all vertices on Level B of BFS table is visited, the pointer now is on Level C.



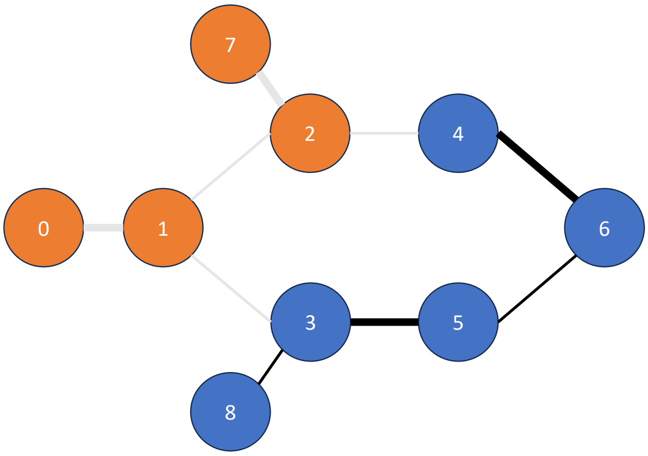
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2	7

Figure 9: 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).



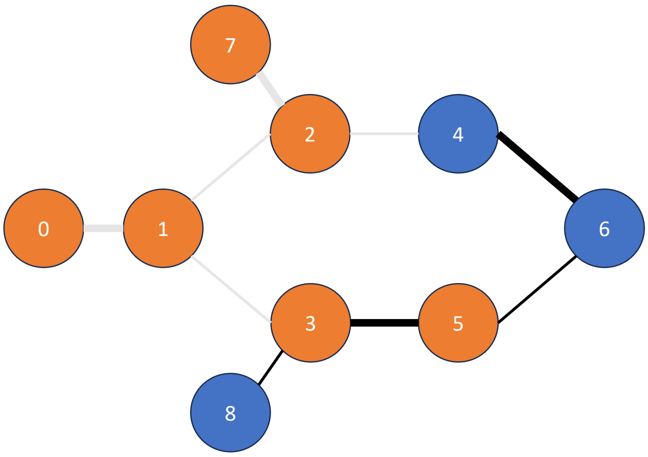
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

Node 1	Node 2
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}

Figure 10: For each of the endpoints, namely 2 and 7, the respective curly brackets ({}) enlists the vertices connected to the corresponding vertex via an unremoved edge. Here, 2 is connected to {4, 7} and 7 is connected to {2}. The corresponding edges are removed (grayed out).



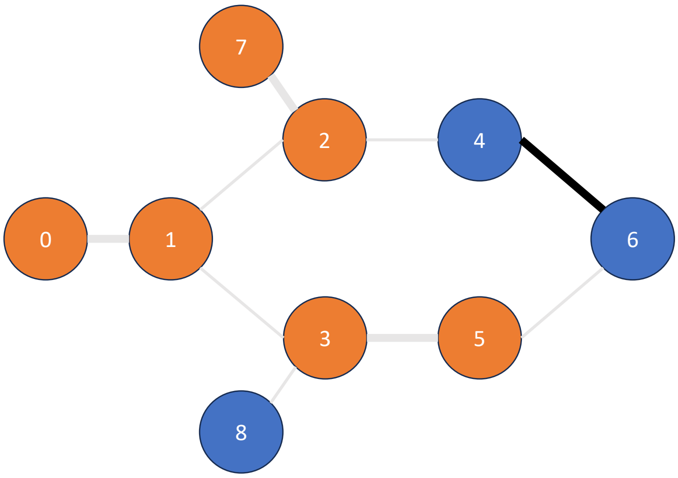
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

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

Figure 11: 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).



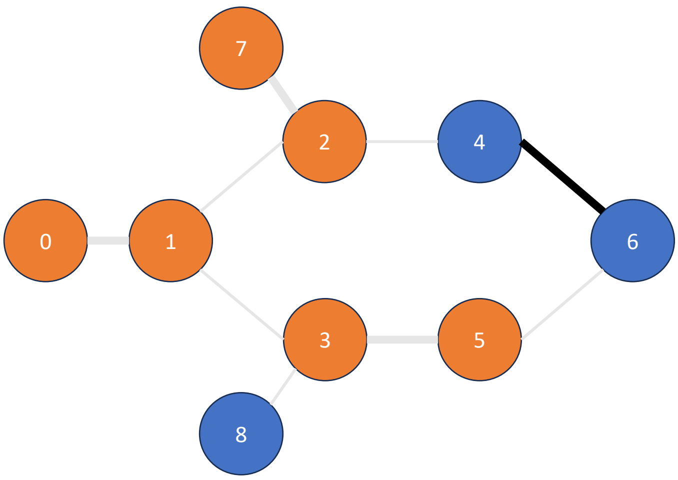
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

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

Figure 12: 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).



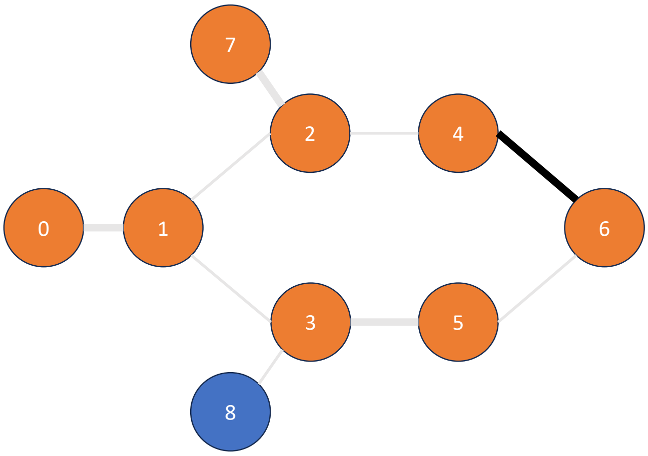
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

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

Figure 13: 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.



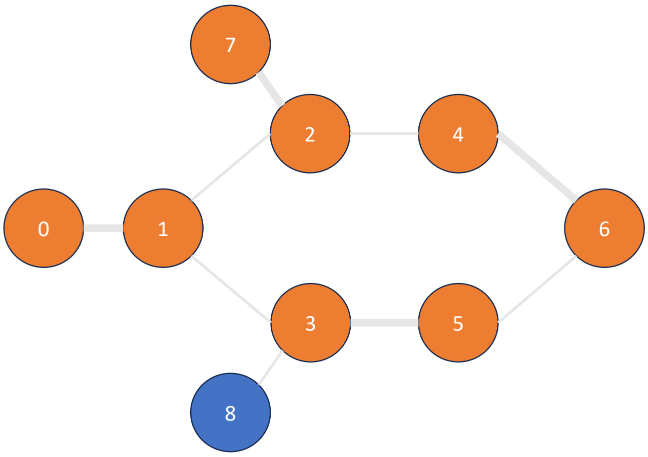
BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

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

Figure 14: 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).



BFS

Level	Vertices
A	0
B	1
C	2, 3
D	4, 5, 7, 8
E	6

Maximal Matching

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: 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 16: 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
3 – {5, 8}	5 – {3, 6}
0 – {1}	1 – {0, 2, 3}
2 – {4, 7}	7 – {2}
4 – {6}	6 – {4}

Figure 17: 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 18: 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 19: 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 – {3, 6}
4 – {6}	6 – {4}

Figure 20: 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 – {3, 6}
4 – {6}	6 – {4}

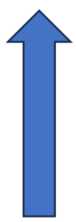


Figure 21: Arrow depicts the bottom-up elimination of vertices.

Local Minimization

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

Figure 22: 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 23: 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 24: 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 25: 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.

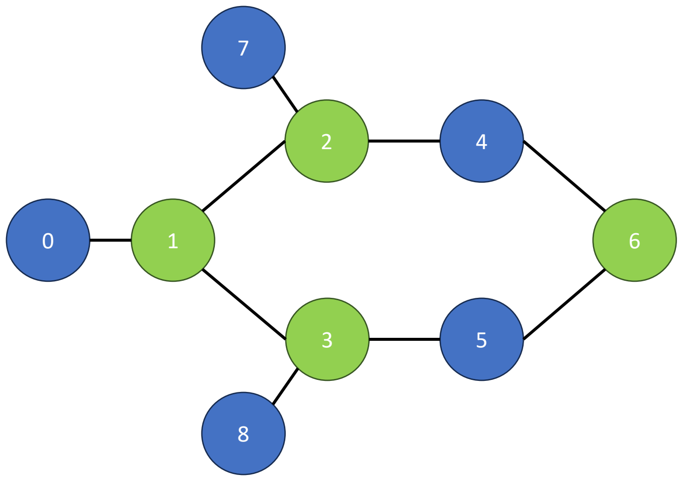


Figure 26: Vertices {1, 2, 3, 6} form the minimum vertex cover of size 4.