Novel and Efficient Exact Approaches to Solve the Minimum Vertex Cover Problem

Novel and Efficient Exact Approaches to Solve the Minimum Vertex Cover Problem

Hong Huang
Insight Centre for Data Analytics,
University College Cork, Ireland
E-mail: hong.huang@insight-centre.org

***Abstract*—Since the $\mathcal{P}$ versus $\mathcal{NP}$ problem was first proposed in the last century, it has always been one of the most concerning problems in computer science. Decades have passed, and many research and studies have tried to prove $\mathcal{P}$ versus $\mathcal{NP}$ problems or find ways to solve $\mathcal{NP}$ problems efficiently. However, the $\mathcal{P}$ versus $\mathcal{NP}$ problem remains unsolved so far. In this paper, we try to provide some ideas for proving the $\mathcal{P}$ versus $\mathcal{NP}$ problem from a mathematical perspective and propose a method of proof. But more importantly, we propose three novel exact methods inspired by the proof and using the neighbour relation of nodes in the graph to solve the minimum vertex cover problem (which is one of $\mathcal{NP}$-hard problem) efficiently, that is in polynomial time complexity.**

1. Introduction

The famous $\mathcal{P}$ versus $\mathcal{NP}$ problem in the field of computer science first proposed in 1971 [1]. $\mathcal{P}$ or Class $\mathcal{P}$, in simple terms, is a type of problem where some algorithms can give a solution in polynomial time. On the other hand, $\mathcal{NP}$ or Class $\mathcal{NP}$, in simple terms, is a kind of problem for which no fast and effective algorithm can give a solution in polynomial time, but it is possible to verify the solution. The $\mathcal{P}$ versus $\mathcal{NP}$ problem is considered to be one of the most important open problems in computer science, and the $\mathcal{NP}$ problem is closely related to many other fields and even real-life problems [2], [3]. The minimum vertex cover (MVC) problem is an optimization problem and one of the classic $\mathcal{NP}$-hard problems, and its decision version is one of Karp's 21 $\mathcal{NP}$-complete problems [4].

The definition of the vertex cover problem has been proposed many years ago and can be defined in the following way [5]. For an undirected graph $G = (V, E)$, $\exists V' \subset V$ such that $\forall uv \in E \rightarrow u \in V' \vee v \in V'$, then we say that V' is a vertex cover of G. That is, every edge in the graph has at least one endpoint in the vertex cover of the graph. In simple terms, MVC problem is to find the smallest possible vertex cover for the graph.

Decades have passed, and many studies have tried to prove the relationship between $\mathcal{P}$ and $\mathcal{NP}$. More studies are trying to find better approximation algorithms or analysis complexity for some of the $\mathcal{NP}$ and above problems [6], [7], [8]. However, little research has tried to find exact algorithms and studied the principle behind it.

In this paper, we try to propose an idea to prove $\mathcal{P} = \mathcal{NP}$ mathematically and try to prove this hypothesis through this approach. We propose three novel algorithms inspired by the proof to solve MVC problem by using the neighbour relation of nodes in the graph, and we proved the correctness and completeness of the algorithms.

2. Preliminary

Before presenting the proof and algorithm, we need to define or redefine some terms for better explanation and expression. Some of these terms are common, and some are for MVC problem.

2.1. Common Use

Definition 1 (Problem)**.** *A problem is a solution in different forms or representation. It usually refers to the form of direct pattern that the subject of observation does not master. The problem itself is the solution.*

Definition 2 (Solution)**.** *A solution is a problem in different forms or representation. It usually refers to the form that is the simplest and most suitable for the observation subject; in other words, the form that the observation subject has mastered the direct pattern. The solution itself is the problem, the overlap of countless problems.*

Definition 3 (Solving)**.** *Solving is the process of transforming the form of the problem or solution.*

Definition 4 (Optimization)**.** *In terms of solution, optimization is the process of finding solutions that are more in line with the additional conditions of the observation subject; in terms of search or solving, optimization is the process of finding the transformation path of the least redundant steps between the observation object from the initial form to the target form.*

Definition 5 (Lower-problem)**.** *If problem P_1 and P_2 are similar problems, the complexity class of P_1 is lower than that of P_2, then we call P_1 is a lower-problem of P_2.*

Definition 6 (Sub-problem)**.** *If problem P_1 and P_2 are of the same type and complexity class, for solution S_1 of P_1*

and the solution S_2 of P_2: $S_1 \subset S_2$, then we call P_1 is a sub-problem of P_2.

Definition 7 (Efficient)**.** *There is no other forms in the path between the initial form to the target form.*

Definition 8 ($\mathcal{P}$)**.** *We follow the commonly used definition of $\mathcal{P}$, but we will briefly rewrite it here for a better understanding below. If the problem P_1 can be solved in polynomial time, then we call P_1 belong to the $\mathcal{P}$ class.*

Definition 9 ($\mathcal{NP}$)**.** *We follow the commonly used definition of $\mathcal{NP}$, but we will briefly rewrite it here for a better understanding below. If problem P_1 cannot be solved in polynomial time, but unidentified solution S to P_1 can be verified in polynomial time, then we call P_1 belong to $\mathcal{NP}$ class.*

2.2. For Minimum Vertex Cover

Definition 10 (Redundant)**.** *For the vertex cover V'_G, if $u \in V'_G \wedge N_G(u) \subset V'_G$, then we call u redundancy (of V'_G).*

Definition 11 (True-leaf)**.** *For an undirected graph $G(V, E)$, if there are multiple MVC V' and $\forall V' \rightarrow u \in V \wedge u \notin V'$, then we call u a true-leaf.*

Definition 12 (Leaf)**.** *For an undirected graph $G(V, E)$, if there is only one MVC V' and $u \in V \wedge u \notin V'$, then we call u a leaf.*

Definition 13 (Not-leaf (Or root))**.** *For an undirected graph $G(V, E)$, if there are multiple MVC V' and $\forall V' \rightarrow u \in V'$, then we call u a not-leaf or root.*

Definition 14 (Half-leaf)**.** *For an undirected graph $G(V, E)$, if there are multiple MVC V', if $u \in V$ and u is neither root nor true-leaf, then we call u half-leaf.*

Definition 15 (Ring)**.** *For an undirected graph $G(V, E)$, $uv \in E \rightarrow u' \in N(u) \wedge v' \in N(v) \wedge u'v' \in E$, then we call the relation between u, v, u', v' is a ring and u' and v' are in this ring. We can know that at least four vertexes are required to determine a ring.*

Definition 16 (Complete-ring)**.** *For an undirected graph $G(V, E)$, $uv \in E \rightarrow N(u) = N(v)$, then we call the relation between u, v, $N(u)$ is a complete-ring and $N(u)$ are in this ring. We can know that at least three vertexes are required to determine a complete-ring.*

Definition 17 (Isomeric-graph)**.** *For two different graphs, if they have the same number of nodes, and they have the same number of nodes in various degrees, but their structures are different, then we call the two graphs each other's isomeric-graph.*

Definition 18 (Hardness)**.** *For an undirected graph $G(V, E)$, the hardness of G is the difficulty of finding its MVC set regardless of the size of the graph. For example, another undirected graph $G'(V', E')$, $|V'| >= |V|$ and $|E'| >= |E|$, if the difficulty of finding MVC set in G is greater than G', then we say that the degree of hardness of G is higher than G'.*

3. Proof, Algorithm and Evaluation

In this section, we propose an idea that proves $\mathcal{P}$ versus $\mathcal{NP}$; then we try to prove the problem. Also, we propose several algorithms that can solve MVC problem quickly and effectively (in polynomial time). After each algorithm is presented, we prove the correctness and completeness of the algorithm; and we also conduct a brief evaluation of the algorithm.

3.1. Theoretically Proof

The idea of the proof is inspired by the definitions of $\mathcal{P}$ and $\mathcal{NP}$. One of the reasons why the $\mathcal{P}$ versus $\mathcal{NP}$ problem is difficult to prove is that it is difficult to find an algorithm that can solve the specific $\mathcal{NP}$ problem in polynomial time from the perspective of factual argumentation. It is difficult to prove the relationship between $\mathcal{NP}$ and time, especially polynomial time, and the relationship between $\mathcal{P}$ and $\mathcal{NP}$ starting from mathematical proofs and logical arguments. One of the critical points in the definition of the $\mathcal{NP}$ problem is that it cannot be solved in polynomial time. The transformation of mathematics ignores time, which can tackle the proof issue related to time. So, in other words, we only need to prove that the solution of the $\mathcal{NP}$ problem in a specific form is observable, then we can prove that the proposition is not valid. At the same time, if $\mathcal{P}$ does not exist, then the proposition will not be established, then the relationship between $\mathcal{P}$ and $\mathcal{NP}$ can naturally be proved.

Problem $\mathcal{P}$ versus $\mathcal{NP}$. Proposition . Prove that: $\exists \mathcal{P} \wedge \mathcal{P} \neq \mathcal{NP}$

Proof. Assuming the proposition is true.
Let problem $P \in \mathcal{P}$, problem $\mathcal{P} \in \mathcal{NP}$, conversion function for combining $\mathcal{F}$, $\{s_1, s_2, ..., s_m\}$ be the set of solution to $\mathcal{P}$.
There must exist $\{P_0, P_1, ..., P_n\}$ which is the lower-problem set of $\mathcal{P}$ such that:
$\theta = \sum_{i=0}^{n} \mathcal{F}(P_i) = \mathcal{F}(P_0) + \mathcal{F}(P_1) + ... + \mathcal{F}(P_n) = \mathcal{P}$.
Or $\mathcal{P}$ is an entirely independent question, which is impossible. Consider P and $\mathcal{F}(P)$ as a whole or number $\rightarrow \alpha + \beta + ... + \nu = \mathcal{P}$.
Consider $\{\alpha, \beta, ..., \nu\}$ as a list of sequence $\rightarrow (a_i)_{i=0}^{n} = (\alpha, \beta, ..., \nu)$.
By using generating function and polynomial interpolation [9], [10], [11], we have: $a_i = \alpha + \beta x + ... + \nu x^n = G(x)$
Then, $\mathcal{P} - \theta = \sum_{i=0}^{n} G_i(x) - \theta = (x \pm u_1) * (x \pm u_2) * ... * (x \pm u_m) = 0 \rightarrow \{u_1, u_2, ..., u_m\} = \{s_1, s_2, ..., s_m\}$.

From the definition 8 of $\mathcal{P}$, we know that $\mathcal{P}$ can be solved in polynomial time, so can the value of u. Moreover, the above derivation 3.1 shows that the solutions can be directly observed or obtained from the problem itself, which is contrary to the definition 9 of $\mathcal{NP}$. It can also be seen from the derivation that the value range can be obtained by deriving the function n times. n is related

to the type and complexity of problem $\mathcal{P}$, which is also contrary to the definition of definition 9 of $\mathcal{NP}$.

$\therefore$ In summary, the hypothesis is not correct, and the proposition is false. $\nexists \mathcal{P} \vee \mathcal{P} = \mathcal{NP}$. In both cases, $\mathcal{P} = \mathcal{NP}$.

■

In the above proof, we first use the $\mathcal{P}$ class problem to construct a similar $\mathcal{NP}$ problem. After the construction is completed, we regard the lower-problem and its conversion as a whole and sequence and then use generating function and polynomial interpolation to obtain the general term formula. Then we use the general term formula to replace the original question. Then through form conversion, the original problem is transformed into factorization, and then it is derived from the rules, which has been mastered, that the problem $\mathcal{P}$ can be directly solved (or can be directly observed and solved). It can be concluded that the problem of $\mathcal{NP}$ class can be in polynomial time, or the problem of $\mathcal{P}$ class does not exist, which is contrary to the proposition and definition. Finally, $\mathcal{P} = \mathcal{NP}$ is derived from the violation of the definition of $\mathcal{NP}$.

3.2. Deco-Algorithm

Deco-algorithm is inspired by the above mathematical proof to solve MVC problem. One of the essential things for an algorithm to solve MVC problem is the node selection criterion. In our deco-algorithm, the criterion is straightforward and accurate and complete, that is, one by one to select and remove the neighbour of the node with the smallest degree in the current graph. Every time by selecting and removing a neighbour, a new graph is generated; therefore, an MVC set is obtained by repeating the process, that is selecting and removing the neighbour of the node with the smallest degree, in the newly generated graph. The selected node will be removed from the graph and the same for the nodes whose degree becomes 0 caused by that. It should be noted that if the node with the smallest degree in the previous graph has not been removed from the current (newly generated) graph, then it will still have the smallest degree in the newly generated graph.

This algorithm requires two lists: a list *DegreeList* takes the degree as the internal interval and places the nodes by their degree in the corresponding interval in ascending order; The other list *G* stores the degree of nodes and the relation between nodes (can be understood as edges), and the function of this list can be achieved by the graph. After the graph data is entered and the graph is constructed, these two lists have also been constructed. The nodes fall into the 0-zone (the interval of degree 0) of *DegreeList* will be removed from *G* and *DegreeList*. Similarly, nodes selected into *ResultList* (minimum vertex cover set) also will be removed from the graph. Edges that lack endpoints also will be removed from the graph.

Function *remove()* takes a node or a set of nodes as a parameter input and then removes them. Function *dropAremove()* takes a node or a set of nodes as a parameter input and then drops them in *DegreeList* to an interval that is one degree lower than the current one; If a node hits 0-zone after falling, it is removed directly. Function *neighbour ()* takes a node as a parameter input and then returns the list of its current neighbour in G.

The algorithm starts from the non 0-zone of *DegreeList* until the *DegreeList* is empty. When the algorithm terminates, MVC is built.

Algorithm 1: Deco-MinimumVertexCover

Data: *Graph(V,E)*.
Result: *ResultList* (a set of minimum vertex cover).
initialization;
while n ← DegreeList **do**
  $v \leftarrow$ `neighbour`(n)[0];
  ResultList ← *ResultList* ∪ {v};
  `dropAremove`(`neighbour`(v));
  `remove`(v);
return *ResultList*;

The core of this algorithm is to add the neighbour of the current node with the smallest degree as the target to MVC set, then reduce the degree of the neighbour of the target node because the target node is removed, and finally clear the node in the 0-zone. List G helps to quickly find the neighbours of the target nodes and locate the neighbour's degree interval.

The algorithm does not need to detect 0-zone every time because only the neighbours of the target node are affected. In the process of decomposition (the flow of this algorithm), the graph is constantly shrinking, the number of neighbours that need to be manipulated will continue to decrease. In the worst case, the algorithm needs to run $|V|-1$ iterations, and the number of nodes needed to operate each time is $(|V|-1)!$. The time complexity of the algorithm is $\mathcal{O}(nlogn)$, but we will not analyze it here. It should be noted that what we mentioned here is the time complexity of the algorithm, which does not include the construction and formatting time of the graph. For our algorithm, the structure of the data is by default, in line with the algorithm requirements. In some cases, if necessary, structuring (or say formatting) the graph data requires $\mathcal{O}(n^2logn)$ time complexity. So, in this case, it takes $\mathcal{O}(n^2logn)$ total time complexity from constructing the graph data to finding an exact solution.

In the process of selecting the target nodes, multiple feasible possible nodes may appear. By choosing different possible nodes, we can obtain all potential MVC set, in other words, the whole set of MVC set.

3.2.1. Deco-Algorithm Proof. Here we will prove that deco-algorithm is correct and complete. The proof will start by considering the conditions that make the problem more difficult. Since any situation is equally difficult for our deco-algorithm, the proof and discussion here are based on standard concepts (such as the greedy algorithm perspective).

Proof-1 For a graph $G(V, E)$ with $|V|$ nodes, $|E|_{max} = |V| + |V|(|V| - 3)/2$, $|E|_{min} = |V| - 1$ if all nodes are connected together. Adding nodes and edges to the graph may increase the size of MVC set, but it does not necessarily

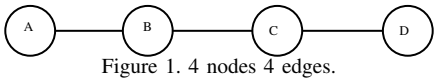


Figure 1. 4 nodes 4 edges.

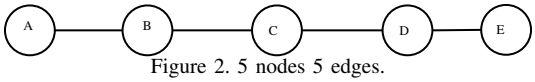


Figure 2. 5 nodes 5 edges.

increase the degree of hardness. Please observe Figure 1, 2, 3.

Figure 2 has more nodes and edges than Figure 1, but Figure 2 has the same degree of hardness as Figure 1. In Figure 3, there are more edges than Figure 2 and more nodes and edges than Figure 1, but the degree of hardness of Figure 3 is lower than that of Figure 1 and Figure 2. Because in Figure 3, searching for MVC set only needs to select any four nodes.

Please observe Figures 4, 5, and 6. We add a connection node and edge to Figures 1 and 2 at different places to become Figures 4, 5, and 6. Comparing Figure 4 and Figure 2, we find that they are the same. Then, changing from Figure 1 to Figure 2 does not increase the degree of hardness. Figure 5 and Figure 1 are the same types of graphs; they both have an even number of nodes and $|V|-1$ edges. Changing from Figure 2 to Figure 5 is equivalent to changing Figure 2 to Figure 1 with more nodes and edges. So changing from Figure 2 to Figure 5 does not increase the degree of hardness. Changing from Figure 2 to Figure 6 only adds node F and edge FB, that is, connect F to B. In Figure 2, MVC set is $\{B, D\}$. Changing from Figure 2 to Figure 6 does not change the minimum cover set and the method of finding MVC. So changing from Figure 2 to Figure 6 does not increase the degree of hardness. And we can have:

Lemma 1. *Adding an edge to a half-leaf makes that half-leaf mandatory.*

Lemma 2. *Adding edges to a leaf makes the leaf selectable, that is, a half-leaf.*

Please observe Figures 7, 8, and 9. From Figure 2 to Figure 7, more than one node and edge have been added. However, the structure of Figure 6 and Figure 7 is not different. In Figure 2, Figure 6, Figure 7, MVC set and the search process and degree of hardness are the same. Changing from Figure 2 to Figure 8 is connecting two nodes to leaf, which makes leaf C mandatory. In Figure 2, nodes B and D are mandatory. For Figure 8, a necessary node C is added to MVC of Figure 2. Therefore, the change from Figure 2 to Figure 8 does not increase the degree of hardness. Similarly, changing from Figure 8 to Figure 9 does not increase the degree of hardness. And we can have:

Lemma 3. *Adding nodes and edges to the graph may increase the size of MVC set, but it does not necessarily increase the degree of hardness.*

Please observe Figures 10. From Figure 3 to Figure 10, a node and an edge are added, and node F is connected

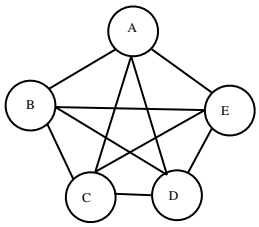


Figure 3. 5 nodes 10 edges.

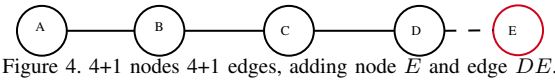


Figure 4. 4+1 nodes 4+1 edges, adding node E and edge DE.

to E. The edge FE makes the node E mandatory so that MVC changes from any four nodes to any three nodes plus the mandatory node E. So Figure 10 is more stringent than Figure 3, thus changing from Figure 3 to Figure 10 increases the degree of hardness. And we have:

Lemma 4. *Two MVC sets of the same size in different graphs, the one with the higher number of nodes that can be selected arbitrarily is in a lower degree of hardness.*

Please observe Figures 11, Figures 3.4, Figures 13. Considering the changes from Figure 1 and Figure 2 to Figure 11, Figure 3 to Figure 3.4, and Figure 8 to Figure 13, the degree of hardness has increased. In Figure 3.4, one side is reduced, making the node E from optional to non-selectable. Therefore, reducing the edge did not relax the problem, but increased the difficulty. Figure 1, Figure 2, Figure 7, Figure 8 and Figure 11 all have similar numbers of nodes and edges. However, the degrees of the nodes in Figure 11 are relatively similar, and there is a ring. Combined with the previous results, we can deduce:

Lemma 5. *For MVC problem, there is a highest degree of hardness.*

Lemma 6. *The highest degree of hardness occurs when the degree of each nodes in the graph is the same, and there are multiple rings.*

Under our derivation, Figure 14 is an example of the graph in the highest degree of hardness. The highest degree of hardness and the lowest degree of hardness can be gradually interchanged. Therefore, if our deco-algorithm can find a solution to MVC problem in the highest degree of hardness and the lowest degree of hardness universally, then it can achieve the same purpose under any circumstances.

Obviously, in Figure 1, Figure 2, Figure 3, Figure 14, our deco-algorithm can successfully find MVC set.

Proof-2 Our deco-algorithm breaks the difficulty by disassembling the problem when it encounters the highest degree of hardness situation. The algorithm always tries to decompose big problems into small ones, decompose graphs with more nodes into graphs with fewer nodes, and decompose high degree of hardness into a low degree of hardness. So if our method is correct, then it must be complete.

Since the edge has two endpoints, this means that for each edge (or every two connected nodes), at least one of

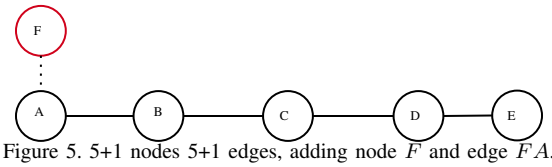


Figure 5. 5+1 nodes 5+1 edges, adding node F and edge FA.

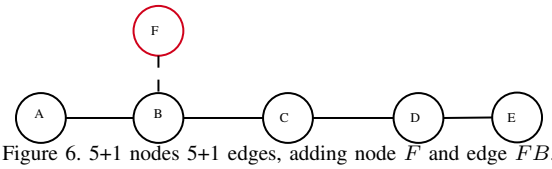


Figure 6. 5+1 nodes 5+1 edges, adding node F and edge FB.

the endpoints must in MVC set. Our algorithm tackles the relation between two connected nodes. If the degrees of the two nodes are equal, then we set aside this relation and start with other relations. If all nodes have the same degree (regular graph), according to our deco-algorithm, choose any point to destroy the current steady state.

For two connected nodes, the possible situation is a repeatable combination of half-leaf, root or (true-leaf, root). In the former case, no matter which node is added to MVC set, it will not be wrong. For the latter case, there are three possibilities: one is that the neighbours of one node are also the neighbours of another node (Figure , $N(A) \subset N(B)$); the other is that the neighbours of the two nodes are not the same (Figure , $N(A) \cap N(B) \neq \emptyset \wedge N(A) \not\subset N(B)$); The neighbours of the two points are completely different (Figure , $n \in N(A) \wedge n \notin N(B)$).

For the first two possibilities, if the node with a lower degree (node A in the figure) is root, then its neighbour with a higher degree (node B in the figure) must also be root because they have familiar neighbours. The relation of (root, root) violates the prerequisites, so under these two possibilities, the lower degree is true-leaf. For the third possibility, if the node where the degree is not low is true-leaf, then its neighbours are root. Moreover, the neighbours of its neighbours must also be true-leaf with a degree of not low; otherwise, the algorithm cannot directly enter the inner relation. The scale needs to be repeated in the next neighbours unless the graph is growing dynamically, or the graph is in a steady state. This derivation is contrary to our premise, so the point with a lower degree is true-leaf. Now we can have:

Lemma 7. *In the relation (edge) of the node with the lowest degree in the current graph, it never is wrong to add the node with the higher degree to MVC set.*

$\therefore$ Combining the above two proofs, our deco-algorithm is correct and complete.

Since the degree of hardness of each graph is the same for our algorithm, the highest degree and the lowest degree are the same. Then we have:

Lemma 8. *For MVC problem, the degree of hardness is the same in each graph.*

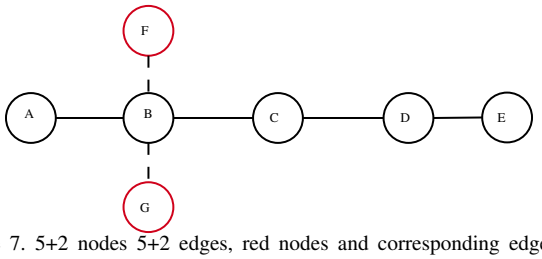


Figure 7. 5+2 nodes 5+2 edges, red nodes and corresponding edges are added.

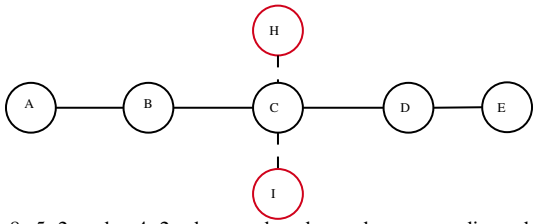


Figure 8. 5+2 nodes 4+2 edges, red nodes and corresponding edges are added.

3.3. Sub-Algorithm

Combining mathematical proof and deco-algorithm, we can get the second algorithm to solve MVC problem. In this section, we will present the sub-algorithm.

The criterion of our sub-algorithm algorithm is simple, accurate and complete. The idea of this algorithm is to start with a whole graph, then delete redundant nodes from it, and then substitute two nodes with one node while maintaining the features of the vertex cover. This process is repeated until it cannot be executed, and then a MVC set is completed.

This algorithm requires multiple lists: List *G* stores the relation between nodes in the original graph and the current statistics of each edge; The list *currentList* stores the nodes and relationships in the current graph, and is also output as a result at the end, which is the same as the input graph during initialization; The list *absentList* stores nodes that are not in the current graph, which is the complement of currentList in the graph; The list *targetList* stores the nodes that have not yet been sure to add to MVC set or not; The *redundantList* temporarily stores the redundant nodes found.

The function *check()* takes a list as a parameter input, detects which nodes in this list are redundant in currentList, and returns a list of these nodes. The function *currentAdd()* takes a node as a parameter input and adds it to currentList in contrast to G. The function *neighbourG()* takes a node as a parameter input and returns a list of its neighbors in G. The function *neighbourC()* takes a node as a parameter input and returns a list of its neighbors in currentList. TheThe function *findSub()* takes a list as an input parameter and returns nodes that can be used as an update.

At the time of initialization, the algorithm needs to perform a redundancy check on the entire graph to obtain a vertex cover set that is not guaranteed to be minimum. Redundancy detection does not need to be performed on each node. The neighbours of the removed redundant node must not be redundant, because at least one endpoint of each

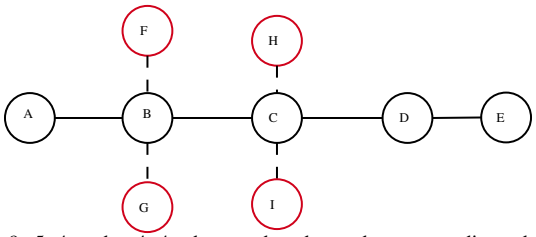


Figure 9. 5+4 nodes 4+4 edges, red nodes and corresponding edges are added.

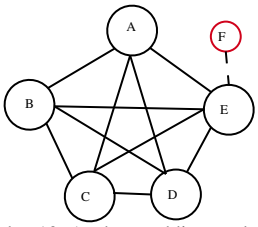


Figure 10. 5+1 nodes 10+1 edges, adding node F and edge FE.

edge is in the vertex cover set. Each time the algorithm performs a substitute, it only needs to perform redundant detection on the neighbours of the update node, because other nodes have been confirmed in the previous operation. There are two ways to detect redundancy: one is if the neighbours of the node are all in the graph; the other is that the statistics of the edges involved at the node are greater than 1. The two methods are similar.

The algorithm will skip the updated node and the neighbours of the updated node when looking for the substitute node pair because these nodes cannot be further substituted. The substitute node pair must have a common neighbour, and the update node must not be in the current graph. The neighbours of the substitute node pair other than the update point must still be in the graph. Therefore, the algorithm finds substitute node pairs through the nodes that have not been determined and the nodes that are not currently in the graph.

Algorithm 2: Deco-MinimumVertexCover

Data: *Graph(V,E).*
Result: *currentList* (a set of minimum vertex cover).

initialization;
redundantList ← `check`(*currentList*);
currentList ← *currentList* \ *redundantList*;
absentList ← *absentList* ∪ *redundantList*;
while n ← $findSub$(targetList ∩ absentList) **do**
    targetList ← *targetList* \ ({n} ∪ `neighbourG`(n));
    `currentAdd`(n);
    redundantList ← `check`(`neighbourC`(n));
    currentList ← *currentList* \ *redundantList*;
    absentList ← *absentList* ∪ *redundantList*;
return *currentList*;

The detection of redundancy and substitutable nodes required by the algorithm continues to decrease with the process. Each detection and substitute will involve the edges or neighbours involved in the node, but as the process needs

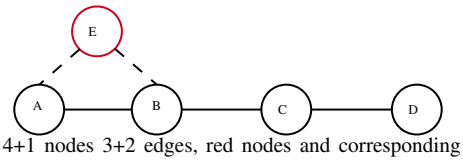


Figure 11. 4+1 nodes 3+2 edges, red nodes and corresponding edges are added.

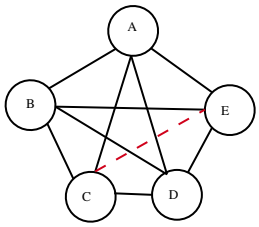


Figure 12. 5 nodes 10-1 edges, removing edge CE.

to involve less. The time complexity of the algorithm is $\mathcal{O}(nlogn)$, but we will not analyze it here.

Similar to deco-algorithm, in the process of selecting the update node, multiple feasible possible nodes may appear. By choosing different possible nodes, we can obtain all potential MVC set, in other words, the whole set of MVC set.

3.3.1. Sub-Algorithm Proof. In this section, we will prove the accuracy and completeness of the sub-algorithm. Since the correctness of removing redundancy has been proved above, we only need to prove that substituting two nodes with one node is correct in any case.

Proof If three nodes can be used to substitute one node, then two of these three nodes must be used to substitute the same node. Because it means that the update node is the neighbour of these three nodes, the update node must also be the neighbour of two of them. If three nodes can be used to substitute two nodes, then two of the three nodes must be used to substitute the same two nodes. The reason is the same as above. We can derive from this:

Lemma 9. *For $k > m > 1$, if k nodes can be used to substitute m nodes, then $k-1$ nodes must be used to substitute the same nodes.*

We assume that the updated nodes must be substituted again to obtain the minimum cover set. Since the updated node is substituted by two nodes, and each edge must have at least one endpoint in the vertex cover set when the updated node is substituted, at least one more node must be added to the vertex cover set. This is contrary to the definition of MVC set. So the assumption is not correct, the updated node does not need to be substituted, and the updated node is mandatory. At the same time, this also makes the neighbours of the update node after removing redundancy not to be substitutable. Because the update node has neighbours after the redundant nodes are removed from the graph, then the neighbours are mandatory. Since the update node is also mandatory, if the neighbour is substituted, the feature of the vertex cover will be lost.

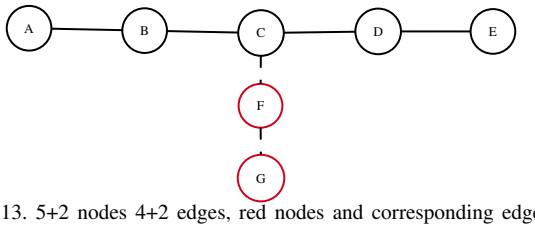


Figure 13. 5+2 nodes 4+2 edges, red nodes and corresponding edges are added.

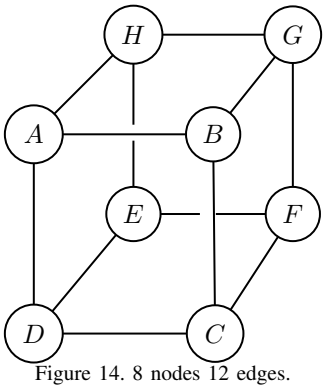


Figure 14. 8 nodes 12 edges.

∴ In summary, so substituting one node with two nodes is correct in any situation.

So our sub-algorithm is complete and accurate. All situations are similar to Figure 18 and are an extension of Figure 18. At the same time, it can be seen that it is actually feasible to use a node to try to substitute its neighbours directly, but the overall process will lack directionality. It is judged whether the current vertex cover set is minimum by the redundant nodes after the replacement, and then MVC set can be obtained by continuously substituting and removing redundant nodes. It can be seen that we can detect whether a vertex cover is the smallest. When substituting a single node (1 node for 1 node), a node cannot be substituted by a neighbour with a lower degree. Because if they have different neighbours, it is apparent that they cannot be interchanged; if their neighbours are in the relationship within the parent set and the subset, then the node with a low degree cannot cover all the edges where the node with a high degree is located, it also verifies that our deco-algorithm above is correct. We can have:

Lemma 10. *It can check whether the vertex cover set is minimum by substituting neighbours.*

Lemma 11. *To ensure the characteristics of the minimum, for two adjacent nodes, the nodes with a higher degree cannot be substituted by a node with a lower degree.*

We use two nodes to substitute because two is the smallest executable number to substitute one, which also makes it is easier to find. At the same time, we use two nodes to substitute one node as the driving force for the reduction of the graph.

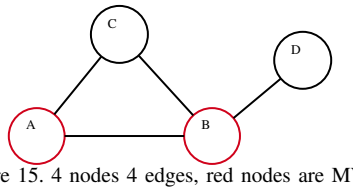


Figure 15. 4 nodes 4 edges, red nodes are MVC.

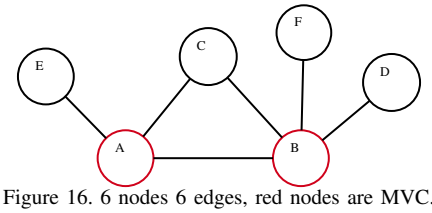


Figure 16. 6 nodes 6 edges, red nodes are MVC.

3.4. Cons-Algorithm

Combining mathematical proof and deco-algorithm and sub-algorithm, we can get the third algorithm to find all potential MVCs in a graph, that is, the entire set of MVC sets. In this section, we will present the cons-algorithm.

The criterion of our cons-algorithm is to construct our goal graph from the empty graph. The method of construction is to continuously connect nodes to the existing graph or add edges between two existing nodes each time to become the next graph. During the construction process, MVC set of each map is stored for reuse in further construction. If MVC set is known to identify isomorphic graphs or sub-problems effectively, then the algorithm is carried out with breadth-first, that is, adding edges to nodes with low degrees first, which can reduce the number of times that graphs for sub-problems need to be constructed.

Before we start presenting our algorithm, we need to introduce four lemmas.

Lemma 12. *Among all MVC sets of the current graph, the one that contains the endpoint of the newly added edge is MVC set of the new graph.*

Lemma 13. *Suppose MVC set of the current graph does not contain the endpoints of the newly added edge. In that case, the formula for MVC set of the new graph is generally MVC set of a group N that takes the two endpoints of the newly added edge as a whole, that is, one of the two endpoints is combined with the set of minimum vertex sets of the remaining part after extracting N under certain conditions.*

The specific condition is: for the edge connecting N and other parts, there must be at least one endpoint in the result of the composition. It can be simply understood as removing the impossible, that is, not the vertex cover, among all the combined results, and what is left is the sets of MVC set of the new graph. Because under this premise, in order to cover the new edge, one more node must be added to the current MVC set. This node can be one of the two endpoints of the edge or two in exceptional cases. An exceptional case is to add an edge between two leaf nodes that already exist in the graph, which may cause the size of MVC set of the remaining part after N extraction to be lower than before.

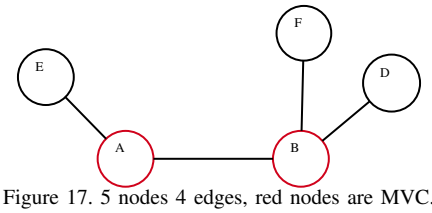


Figure 17. 5 nodes 4 edges, red nodes are MVC.

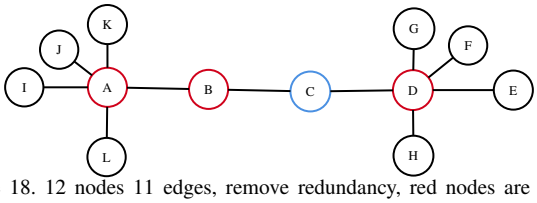


Figure 18. 12 nodes 11 edges, remove redundancy, red nodes are MVC, and blue node is redundant.

Lemma 14. *In exceptional cases, the entire set of MVC set of the new graph is the combination of one of the endpoints of the new edge and MVC set of the old graph, plus the combination of two endpoints of the new edge and the sets of MVC set of the remaining part after extraction.*

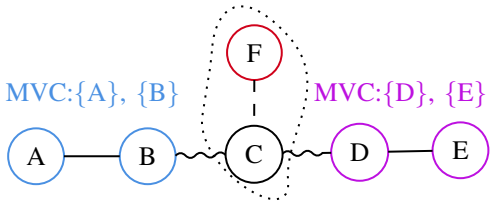
For example, in Figure , the removed edge CE is considered as an addition, then MVC set before the addition is $\{A, B, D\}$. After removing the CE, the part $\{A, B, D\}$ is left, and the set of MVC set for this part is $\{\{A, B\}, \{A, D\}, \{B, D\}\}$. Then the complete set of MVC set of the new graph is $\{\{A, B, D, E\}, \{A, B, D, C\}\}$ plus $\{\{A, B, C, E\}, \{A, D, C, E\}, \{B, D, C, E\}\}$.

The exceptional cases all appear when the two endpoints of the new edge are extracted as a group N, the remaining part is the complete graph and N has edges connected to all the endpoints of the complete graph. Because if it is not under such a premise, either one of the endpoints of N is half-leaf or pulling off N will not affect the size of MVC set of the rest.

Regarding the two endpoints of the newly added edge as a group N, then no MVC set contains N, which means that the neighbours of N are in all MVC sets. Therefore, when N is removed from the graph, MVC set of other parts will not become larger. Moreover, this also leads to the fact that in general, when the graph is divided into multiple parts by extracting the newly added edges (2 if constructing with breadth-first), the size of MVC set of each part is the same to the number of their nodes occupied in the previous MVC set before extracting.

For example, in Figure , since node F is connected to leaf node C, FC needs to be taken out of consideration. Before extraction, MVC set of the original graph is $\{B, D\}$. After removing the FC, the graph is divided into two parts. Furthermore, these two parts originally had one node in MVC set, namely $\{B\}$ and $\{D\}$, so MVC size of these two parts is 1.

It should be noted that the two endpoints of the new edge and the neighbours of the endpoints are both not in any MVC of the current graph. Because if this happens, the graph before adding the new edge does not have the feature



original-MVC:{B, D}
Figure 19. 5+1 nodes 4+1 edges, the size of MVC of parts is the same as the number of nodes in the original MVC.

of vertex cover. In the best case, adding an edge will not increase the size of MVC set of the current graph, but it will certainly not decrease.

Many sub-problems that appear in the construction process are isomorphic graphs of the graphs that have stored the set of MVC sets. If the solution of the corresponding isomorphic graph is not stored, there are two ways to handle it. One is to rebuild the part from scratch. However, if the sub-problems can be effectively identified under the premise of knowing MVC set, then this part does not need to be rebuilt. This part may have a sub-problem that has the same solution as the graph that has already been solved. We can start from the sub-problem or directly obtain the solution of this part, that is another method. The minimum sub-problem with the same solution here is a sub-problem with the same MVC set as the original problem with the least number of points and degrees. Moreover, according to Proof 3.2.1 we can get:

Lemma 15. *MVC set of a graph is its skeleton. Two different graphs, if they have only the same MVC set, that is, the position of the vertex cover in the graph corresponds to each other (adjacent or not), then the size relationship between their MVC and other neighbours are also the same, and they have the same minimum homologous subproblems.*

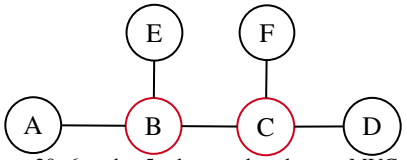


Figure 20. 6 nodes 5 edges, red nodes are MVC set.

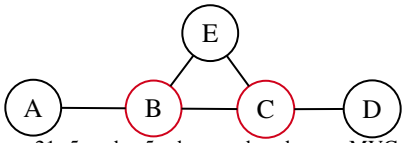


Figure 21. 5 nodes 5 edges, red nodes are MVC set.

For example, in Figure 16 and Figure 20, their set of MVC sets are $\{\{B, C\}\}$; their nodes B and C are adjacent, and the degrees of B and C are higher than their neighbours except each other; their minimum sub-problems with the same solution are the same, as shown in Figure 21.

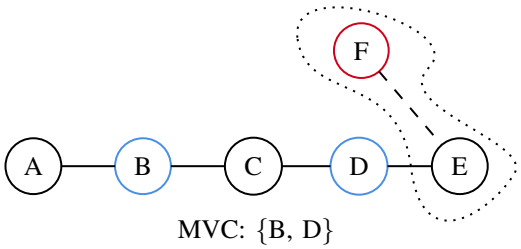


Figure 22. 5+1 nodes 4+1 edges, connecting F to E, blue nodes are MVC.

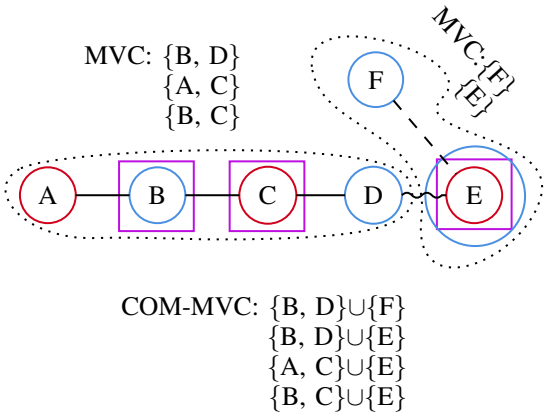


Figure 23. 4+2 nodes 4+1 edges, MVC combination with Condition, nodes in the same color are in the same MVC.

Figure 22 and Figure 23 show the process of extraction and combination in general. The set of MVC sets for $\{F, E\}$ is $\{\{F\}, \{E\}\}$. The set of MVC sets for the remaining part after extraction is $\{\{B, D\}, \{A, C\}, \{B, C\}\}$. The direct combination of the two sets of solutions is $\{\{B, D, E\}, \{B, D, F\}, \{A, C, E\}, \{A, C, F\}, \{B, C, E\}, \{B, C, F\}\}$. Then removing the infeasible $\{B, C, F\}$ and $\{A, C, F\}$ is the set of MVC sets of the new figure. Alternatively, when combining, if the endpoints of the connecting edge between the extracted part and the remaining part are not in MVC, skip it.

Cons-algorithm requires: A graph *G(V', E')* initially has all the nodes of *Graph* but no edges, and finally becomes the same as *Graph*; degree statistics can be realized by *G* and *Graph*. The particular storage structure *S* stores the solutions (MVC) of all the graphs that appear during the construction process, and is classified according to the number of nodes and the degree of the nodes and the size of MVC set; the whole algorithm shares one *S*. The list notleafList stores all roots and half-leafs in *G*, and is updated with *G*. If the endpoint of the new edge is not in this list, it means that the steps of segmentation and finding MVC set of the sub-problem are required. The list *tempGraList* temporarily stores the resulting graph. List *tempSolList* temporarily stores the resulting sets of MVC sets. The list *result* temporarily stores the resulting set of MVC sets.

The function *conCombine()* takes sets of MVC sets and graph *G* as input, combines them according to the conditions of graph *G*, and returns the combined set of MVC set. The function *segment()* takes an edge and a graph as input parameters, it extracts the two endpoints of the edge as a group from the graph and returns the resulting graph. The function *retrieve()* takes a graph as a parameter input and returns the set of MVC sets of the graph stored in *S*, or returns null if there is no stored solution. The function *recursiveCons()* takes a graph as a parameter input, which is a recursive call of cons-algorithm itself.

Algorithm 3: Cons-SetsMinimumVertexCover

```
Data: Graph(V,E), S.
Result: result (set of minimum vertex cover sets); S.
initialization;
while e ← E do
    tempGraList ← segment(e, G);
    tempSolList ← ∅;
    while g ← tempGraList do
        result ← retrieve(g);
        if result = ∅ then
            result ← recursiveCons(g);
        tempSolList ← tempSolList ∪ {result};
        result ← conCombine(tempSolList, G);
        S ← S ∪ result;
return result;
```

If the algorithm only needs to construct the graph in a particular direction or order, then the sub-problems that need to be reused will have been stored in the path of the graph construction process. If there is a need to divide the graph into two sub-problems during construction, then one of the sub-problems may not be stored in the experienced path. In the worst case of this premise, if the algorithm cannot identify isomorphic graphs and sub-problems, then the algorithm needs to construct the entire graph from the reverse direction when constructing it.

Overall, in the best case, there are $|V|$ nodes and $|E| = |V| - 1$ edges in the graph, then the algorithm needs to construct $|E|+1$ times and store the entire set of MVC sets for $|E|+1$ graph. In the worst case, if the algorithm cannot identify isomorphic graphs and sub-problems with the same MVC set under the premise of knowing MVC set, if the graph is a complete graph (with $|V|$ nodes and $|E| = |V| + |V|(|V|-3)/2 = |V|(|V|-1)/2$ edges), then the algorithm needs to construct $C^0_{(|V|(|V|-1)/2)} + C^1_{(|V|(|V|-1)/2} + ... + C^{|V|(|V|-1)/2}_{(|V|(|V|-1)/2} = 2^{(|V|(|V|-1)/2)}$ times and store the set of MVC sets for the same number of graphs.

3.4.1. Cons-Algorithm Proof. Since our cons-algorithm is based on the step-by-step construction of graphs and MVC set, we only prove that isomorphic graphs can be effectively identified on the premise of knowing the number and degree of nodes and MVC set of the graphs.

Generally, determining MVC set needs to contain the information of the entire image, that is, the degree of the nodes and their neighbours. However, in the following discussion on identifying isomorphic images, the information

of the entire graph has been given before MVC set. In the following proofs, when we say that we know or determine MVC set, we refer to the degree of the nodes in MVC set. ***Proof*** In some cases, if the number of points and the degree of each point are determined, the graph is determined; in other words, the graph is unique under this premise. Then we can locate and identify the graph by the number of nodes and the degree of each node. But in some cases, the graph may have isomeric-graph. For example, Figure 24 and Figure 25 are isomeric-graphs of each other.

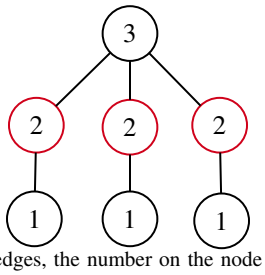


Figure 24. 7 nodes 6 edges, the number on the node is its degree, and red nodes are MVC set to the graph.

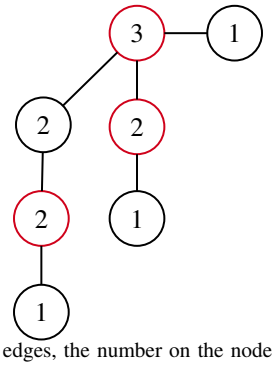


Figure 25. 7 nodes 6 edges, the number on the node is its degree, and red nodes are MVC set to the graph.

For the case of the graph with only one MVC set: The isomeric-graph of a graph is to adjust the node pairs based on the original graph. After adjusting the neighbours of a node, it may cause the half-leaf node to become a leaf node, which will cause more changes. If there is only one MVC set in a graph, all nodes except the nodes in MVC set must be leaf nodes. According to Proof 3.2.1, for such a graph, the size relation between the nodes in MVC set and their neighbours is fixed, that is to say, there must be fixed discards in the process of searching for MVC set. So the neighbours of these nodes are fixed. We can get:

Lemma 16. *Under the premise that there is only one minimum vertex cove set, the figure with a specific number of points and the degree of each point and MVC set is unique. We can use these parameters to identify and locate a graph effectively.*

For the case of the graph with multiple MVC sets: If a graph has multiple MVC sets, part of the neighbours of the half-leaf node in one of MVC are also half-leaf nodes. From Proof 3.2.1, we can know that there is a specific order to gradually select this known MVC set from the graph. Suppose the half-leaf neighbour of MVC set is adjusted in the graph and changing the selection order of the nodes in MVC set at the same time. In that case, since the degree of each point and the total number of points remain unchanged, the final result will have the qualities of a vertex cover only when the adjusted half-leaf neighbour is equal to the previous degree. So we can get:

Lemma 17. *For two isomeric-graphs with multiple MVC sets, if they have the same MVC set, then their other MVC sets are also the same.*

Combining Lemma 16 and lemma 17:

Lemma 18. *There is no isomeric-graph with the same MVC set for a graph. The graph with a specific number of nodes and the degree of each node and one of MVC set is unique.*

For a graph with multiple MVC sets, if an additional node is connected to one of the nodes in one of MVC sets, it is equivalent to selecting that MVC set. In other words, after the additional nodes are connected, there is only a MVC in the new graph. According to Lemma 16, we know that such a graph is unique. Then the graph before the new node is connected is also unique. This also proves Lemma 17 and Lemma 18.

In exceptional cases, for our cons-algorithm, the remaining part of the graph after extracting the two endpoints of the new edge is a complete graph. Because all the set of MVC sets of the complete graph are any combination of $|V|-1$ nodes, it can be quickly calculated even if the corresponding solution is not stored. $\therefore$ In summary:

Lemma 19. *It can effectively identify isomorphic graphs on the premise of knowing the number of node and the degree of points and the same MVC set.*

Only a graph with the same degree of all the nodes can guarantee that the neighbours after the exchange have the same degree as before when exchanging the neighbours of two half-leaf nodes. However, such a graph does not have an isomeric-graph, which also verifies lemma 17 from the side.

Using the generating function and the polynomial interpolation can also prove lemma 3. A graph is determined by nodes and edges and the connection relationship between nodes. The degree of a node means the number of endpoints of edges it is, and the number of other nodes it connects. Knowing a MVC set of the graph means knowing the minimum number of nodes needed to cover all edges in the graph and the degree of these nodes. If we also know the number of nodes in the graph, and the size of MVC set is fixed. Then we can regard the nodes in MVC set as mathematical values, and then use polynomial interpolation to find the curve of these nodes. According to the axioms, such a curve is unique. The resulting curve is the representation of the graph. So in fact, knowing the total number of points of the picture and a MVC set of the picture can determine the only graph.

3.5. Another Theoretically Proof

After proposing cons-algorithm, in this section, we will try to use a hypothetical oracle machine learning algorithm to prove the relation between $\mathcal{P}$ and $\mathcal{NP}$.

Proof-1 Suppose there is an oracle machine learning algorithm L that can learn by the description of the problem type and some specific examples and basic data and solutions of the problem. In this case, the algorithm can learn a universal rule of the same type that is not higher than the size and complexity of the given instance and can use this rule to solve problems of the same size and difficulty or below.

Then if there is a kind of $\mathcal{NP}$ problem under common sense, suppose it has only two specific instances for each specific size parameter n, and its time complexity is $\mathcal{O}(2^n)$ or simply 2^n. Then suppose there is the hardest problem instance $P_{n,2}$ for n, so the problem instances whose size parameter is not higher than n are all its sub-problems. For $P_{n,2}$, L must use all other problem instances and their solutions as the learning set in order to be able to find the solution to $P_{n,2}$ and find the universal rule.

The foundation of L can learn the rule and solve $P_{n,2}$ is that $P_{n,2}$ is not independent of other problem instances (its subproblems), and there are enough independent solutions of subproblems for L to learn. There are two possible situations where $P_{n,2}$ is independent of the sub-problems. One is that $P_{n,2}$ is independent of all problem instances, which means that P cannot be derived from other problem instances, which is impossible. The other is that $P_{n,2}$ depends on larger and more complex problem instances rather than smaller and simpler ones, and to solve $P_{n,2}$ these problem instances must be solved first, which is impossible from common sense. So L can learn the rule and solve $P_{n,2}$. Perhaps it takes a massive amount of time to learn such rule. However, after acquiring the rule, then solving the same type of problems below $P_{n,2}$ is in the P class under common sense, just like using mathematical formulas, theorems and even definitions to acquiring solutions. So is this problem and this approach in the $\mathcal{P}$ class or $\mathcal{NP}$ class?

Someone might argue that the time to solve the sub-problems and the time to learn L also need to be included in the total time to solve $Pn, 2$. Then the total time used to solve $P_{n,2}$ is: the time to acquiring the solution of sub-problem + the learning time + the time to use the rule to solve $P_{n,2} = T_{P_{1,1}} + T_{P_{1,2}} + ... + T_{P_{n,1}} + T_L + T_{S_{P_{n,2}}} = 2^1 + 2^1 + 2^2 + ... + 2^n = 2^{n+1} + 2^n + T_L + T_{S_{P_{n,2}}} = T_{P_{n,2}}$. At the same time, we know that $P_{n,2}$ is an instance of this type of $\mathcal{NP}$ problem, so $T_{P_{n,2}} = 2^n$. Then $2^1 + 2^1 + ... + 2^n = 2^{n+1} + 2^n + T_L + T_{S_{P_{n,2}}} = 2^n$. However, this is not true. It means that either this type of $\mathcal{NP}$ problem under common sense does not belong to the NP class, and its solving time is not exponential; or for this type of $\mathcal{NP}$ problem under common sense, the most difficult instance $P_{n,2}$, L learning the rule and solving it does not need to use the solutions of all its sub-problems as the learning set. If L does not require the solutions of all sub-problems, then $\mathcal{P}$ has a direct relationship with some of the sub-problems, and there are some patterns can make these sub-problems and their solutions skip other sub-problems and become $P_{n,2}$ and the solutions. It holds for any $n \geq 1$, and it is valid for different problem instances with the same n. In other words, this type of problem does not belong to the $\mathcal{NP}$ class under common sense, and the time complexity of solving one of the problem instances is not exponential.

Here is a more practical example.

Proof-2 For MVC problem, consider the following problem that contains a sequence of problems: find the entire set of MVC sets of each graph in the list $\{G(1,0), G(2,1), ..., G(|V|, |E|)\}$. So here it is assumed that the actual time complexity of finding the entire set of MVC sets of a graph is exponential. Then the time required to solve this problem sequence is $T = sum_{i=1}^{m} T_{G_i}$. Suppose we use our cons-algorithm to find the entire set of MVC sets of a graph. Then we already know that in the worst case, for a graph, the algorithm needs to construct all its sub-problems first to obtain its solution. Moreover, such a graph can be a complete graph, and we know that MVC of any complete graph is any $|V| - 1$ nodes in the graph. So we can have $T = sum_{i=1}^{m} T_{G_i} = T_{G_i} = \mathcal{O}(1)$. However, this equation is impossible to hold. Unless for our cons-algorithm, or exist arbitrary algorithms, in the worst case, the problem can be solved without knowing the solutions of all sub-problems. Then the T_{G_i} of our cons-algorithm in the above equation is larger than the actual time required. The above equation becomes $T = T_{cons,G_i}$, then it can be established. At the same time, T_{G_i} must be non-exponential, or there is even a universal rule that can give a solution to any graph in linear time. Then the above equation can be fully established.

We believe that it can learn the rules from the data in the graph. The graph can be determined by the nodes and the degree of the nodes and the relation between nodes. For a $Graph(V, E)$, $N(n)$ is the neighbour of node n, $D(n)$ is the degree of node n, then we can have the following informal equations to express this graph:

When $|V|$ and $D(n)$ are fixed, most of the information on the graph has been determined, especially for the vertex cover problem.

$$\begin{cases} n_1 + n_2 + ... + n_j = N(n_k) \\ \vdots \\ |n_1| + |n_2| + ... + |n_j| = D(n_k) \\ \vdots \\ \sum_{i=1}^{k} |n_i| = |V| \\ \sum_{i=1}^{k} D(n_i) = |E| * 2 \end{cases}$$

However, some graphs will have isomeric-graph. The number of isomeric-graphs of a graph can be calculated by generating function and Polya enumeration theorem [12]. From this simple system of equations or previous proofs, we can know that possible structural changes are affected by $|V|$ and $D(n)$. For some isomeric-graphs, they may have the same size but different MVC set. We think there is another feature besides the number of nodes in the graph and the degree of nodes, by using these three features, we can learn the rules and solutions to the desired problems, although these features are still difficult to capture by existing knowledge. By using the number of nodes in the

graph, the degree of the nodes, different additional features and solutions to acquiring solutions to some problems, such as whether the graph has a vertex cover set of size k and a MVC, etc. For the vertex cover problem, these additional features may be the number of rings or complete-rings in the graph or more complex and difficult to grasp features.

For MVC problem, our cons-algorithm shows that in order to find the entire set of MVC sets of a graph, it is necessary to find the solution of the sub-problem first. That may be the reason why some methods are very slow in solving MVC problem. When these methods are searching for MVC set of a particular graph, they also find MVC set of other graphs.

4. Conclusion and Future Work

$\mathcal{P}$ versus $\mathcal{P}$ problem is one of the most concerning problems in the field of computer science. The key to this problem is whether there is a polynomial-time algorithm for solving $\mathcal{P}$ class problems.

MVC problem, as one of the well-known problems of $\mathcal{NP}$-hard level, has received widespread research attention. Moreover, MVC has a vital connection with real-life problems. If an effective and fast algorithm can be found to solve MVC problem, the $\mathcal{P}$ versus $\mathcal{P}$ problem will be solved in disguise, and it will have necessary enlightenment for solving complex problems in real-life.

In this paper, by proposing the new thinking angel and the new idea of proof for the $\mathcal{P}$ versus $\mathcal{P}$ problem, we also have proposed three novel algorithms that can effectively and exactly solve MVC problem, and proved the correctness and completeness of the algorithms. Then we discussed and considered proving $\mathcal{P}$ versus $\mathcal{P}$ from the perspective of machine learning.

As future work, we would like to consider migrating our algorithm to other $\mathcal{NP}$-hard problems, further optimizing our algorithms and propose other effective exact algorithms to solve MVC problem. Furthermore, we would like to consider using machine learning to solve MVC problem or even similar real-life problems exactly.

References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, 1971, pp. 151–158. [Online]. Available: https://doi.org/10.1145/800157.805047

[2] L. Fortnow, "The status of the P versus NP problem," *Commun. ACM*, vol. 52, no. 9, pp. 78–86, 2009. [Online]. Available: https://doi.org/10.1145/1562164.1562186

[3] ——, *The Golden Ticket - P, NP, and the Search for the Impossible*. Princeton University Press, 2013. [Online]. Available: http://press.princeton.edu/titles/9937.html

[4] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, 2010, pp. 219–241. [Online]. Available: https://doi.org/10.1007/978-3-540-68279-0_8

[5] R. Bar-Yehuda and S. Even, "A linear-time approximation algorithm for the weighted vertex cover problem," *J. Algorithms*, vol. 2, no. 2, pp. 198–203, 1981. [Online]. Available: https://doi.org/10.1016/0196-6774(81)90020-1

[6] D. S. Hochbaum, "Approximation algorithms for the set covering and vertex cover problems," *SIAM J. Comput.*, vol. 11, no. 3, pp. 555–556, 1982. [Online]. Available: https://doi.org/10.1137/0211045

[7] P. Sinha and A. A. Zoltners, "The multiple-choice knapsack problem," *Oper. Res.*, vol. 27, no. 3, pp. 503–515, 1979. [Online]. Available: https://doi.org/10.1287/opre.27.3.503

[8] G. J. Woeginger, "Exact algorithms for np-hard problems: A survey," in *Combinatorial Optimization - Eureka, You Shrink!, Papers Dedicated to Jack Edmonds, 5th International Workshop, Aussois, France, March 5-9, 2001, Revised Papers*, 2001, pp. 185–208. [Online]. Available: https://doi.org/10.1007/3-540-36478-1_17

[9] D. E. Knuth, *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997. [Online]. Available: https://www.worldcat.org/oclc/312910844

[10] E. Waring, "Vii. problems concerning interpolations," *Philosophical Transactions of the Royal Society of London*, pp. 59 – 67.

[11] W. Dunham, "Journey through genius: The great theorems of mathematics," *Convergence*, pp. 155 – 183, 01 1990.

[12] G. Pólya, "Kombinatorische anzahlbestimmungen für gruppen, graphen und chemische verbindungen," *Acta Math.*, vol. 68, pp. 145–254, 1937. [Online]. Available: https://doi.org/10.1007/BF02546665