AMERICAN UNIVERSITY OF BEIRUT

# LEARNING BRANCHING STRATEGIES FOR PARAMETERIZED VERTEX COVER

by

RAMI ALI HOTEIT

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
April 2024

AMERICAN UNIVERSITY OF BEIRUT

# LEARNING BRANCHING STRATEGIES FOR PARAMETERIZED VERTEX COVER

by

RAMI ALI HOTEIT

Approved by:

| | |
|---|---|
| Dr. Amer E. Mouawad, Assistant Professor<br>Department of Computer Science | Advisor |
| Dr. Fatima K. Abu Salem, Associate Professor<br>Department of Computer Science | Member of Committee |
| Dr. Shady Elbassuoni, Associate Professor<br>Department of Computer Science | Member of Committee |

Date of thesis defense: April 18, 2024

# ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my mentor and advisor Dr. Amer Abdo Mouawad for his great contribution to making this work possible. Without his perseverance and insight this work would not have been possible.

I would also like to thank Dr. Fatima Abu Salem for her efforts and contributions to the work as well. I am also grateful for the support and backing of my amazing parents Ali and Salam as well as my sister Mira. I am also very thankful for the unwavering support and motivation provided by Rima W.

# ABSTRACT
# OF THE THESIS OF

Rami Ali Hoteit for Master of Science
Major: Computer Science

Title: Learning Branching Strategies for Parameterized Vertex Cover

The Parameterized Vertex Cover (PVC) problem is a central problem on graphs, where given a graph G and a positive integer k the goal is to decide whether the graph contains a set of at most k vertices whose deletion destroys all edges of the graph. In other words, a set of vertices is called a vertex cover of a graph if after deleting those vertices we obtain an edgeless graph. The problem is one of Karp's 21 NP-complete problems (Erickson, 2010)(Karp, 1972), meaning that the problem is computationally hard, takes exponential time to solve, and is not expected to be solvable in polynomial time unless P = NP. For most practical purposes, heuristics, approximation, or parameterized algorithms are the only reasonable way to solve large instances of the problem in a reasonable amount of time. We are interested in solving the problem exactly and one method for solving PVC is the Branch & Reduce paradigm. In a Branch & Reduce algorithm, we construct a search tree to solve a given instance by either branching on which vertices to include into a solution or applying reduction rules to reduce the search space. At a high level, the typical algorithm for PVC selects a vertex of highest degree and branches on either including said vertex in a solution or including all of its neighbors. Several reduction rules are also applied whenever possible. In this work, we investigate a new approach based on machine learning for optimizing vertex selection while solving PVC instances. We constructed a system that uses graph features as inputs to make inferences about the best weighting strategies to be applied on the different node features in order to select the best vertex to branch on. In our approach we utilize reinforcement learning technology to train our model. Our results show that we were able to outperform the high degree strategy in 85% of instances.

## TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................ 1

ABSTRACT........................................................................ 2

ILLUSTRATIONS ............................................................... 6

TABLES ............................................................................ 7

ABBREVIATIONS............................................................... 8

INTRODUCTION................................................................ 9

RELATED WORK............................................................. 14

A. Solving Parameterized Vertex Cover ..........................................14

1. Kernelization ..........................................................................14

2. Crown Decomposition..............................................................16

3. Bounded Search Tree ...............................................................17

4. Iterative Compression..............................................................18

5. Color Coding ..........................................................................19

6. Branch & Reduce ....................................................................20

7. Linear Programming................................................................22

8. Machine Learning....................................................................24

B. Branching Rules ........................................................................29

C. Variable Selection .....................................................................30

METHODOLOGY ............................................................ 35

A. Definition of problem ..................................................................35

B. Overview of the Solution..............................................................37

C. Features for nodes.......................................................................40

D. Features for graphs (Input for Model) .......... 43

E. Branching Simulator .......... 47

F. Reduction rules .......... 48

G. Reinforcement Learning .......... 50

1. State .......... 52

2. Action .......... 52

3. Reward .......... 53

4. Training Algorithm: Proximal Policy Optimization .......... 54

5. Training .......... 57

H. Training Specifics .......... 60

1. Graph Generation .......... 60

2. Libraries .......... 61

3. Parameters .......... 63

I. Limitations .......... 63

RESULTS & DISCUSSION .......... 65

A. Experimental Setup .......... 65

B. Training Results .......... 66

C. Testing Pipeline .......... 68

D. Main Model Results .......... 69

E. Experiments .......... 70

1. Training Algorithm .......... 71

2. Rewards .......... 72

3. Graph Generation .......... 73

4. Reduction Rules .......... 75

5. Varying K .......... 76

6. Varying N ....................................................................................................77

F. Node Feature Analysis ..........................................................................................77

1. Density:..........................................................................................................78

2. Average Clustering ..........................................................................................80

3. Normalized K ...................................................................................................82

4. Average Neighbor Degree .................................................................................84

5. Average Betweenness Centrality .......................................................................86

CONCLUSION AND FUTURE WORK ...................................... 88

REFERENCES ......................................................................... 91

# ILLUSTRATIONS

Figure

1. Branch and Reduce .......... 37
2. Overview of Model Inference .......... 38
3. Branching Simulator .......... 47
4. Reinforcement Learning Overview .......... 51
5. Proximal Policy Optimization Overview .......... 57
6. Training Overview (Method 1) .......... 57
7. Training Overview (Method 2) .......... 59
8. Graph Generation .......... 61
9. Training Results .......... 67
10. Testing Pipeline Overview .......... 68
11. Density Feature Analysis .......... 78
12. Average Clustering Feature Analysis .......... 80
13. Normalized K Feature Analysis .......... 82
14. Average Neighbor Feature Analysis .......... 84
15. Average Betweenness Centrality Feature Analysis .......... 86

# TABLES

Table

1. Main Model Results ........ 69
2. Varying Training Algorithm ........ 72
3. Varying Reward Experiment ........ 73
4. Varying Graph Generation Experiment ........ 74
5. Varying Reduction Rules Experiment ........ 76
6. Varying K Experiment ........ 76
7. Varying Graph Size Experiment ........ 77

# ABBREVIATIONS

VC – Vertex

Cover

PVC – Parameterized Vertex Cover

MILP – Mixed Integer Linear Programming

BVS – Branching Variable Selection

B&B – Branch & Bound

RL – Reinforcement Learning

ILP – Integer Linear Programming

SVM – Support Vector Machine

CNN – Convolutional Neural Network

RNN – Recurrent Neural Network

TSP – Traveling Salesman Problem

GNN – Graph Neural Network

PPO – Proximal Policy Optimization

MSE – Mean Squared Error

DDPG – Deep Deterministic Policy Gradient

# CHAPTER I

# INTRODUCTION

Graph theory is a branch of discrete mathematics that deals with the study of graphs. Graphs are a discrete data structure that consists of nodes connected by edges.(Gross et al., 2018) These graphs are used in various fields including computer science, molecular science, and circuit design.(“Graph Theory in Chemistry: A Brief Review,” 2022; Riaz & Ali, 2011; Toscano et al., 2015) Graphs are often used to model computational problems specifically in the field of computer science. One such problem is vertex cover. The vertex cover problem asks the question that given a graph G find the minimum set of vertices such that removing those vertices and the edges incident to them produces an independent graph, that is a graph without any edges. In other words, vertex cover tries to find the minimum set of vertices such that each edge has at least one of its endpoints within that set.

More Formally The vertex cover problem can be formally defined in the context of a graph G = (V, E), where V represents the set of vertices and E represents the set of edges connecting these vertices. The goal is to find a subset V’ such that for every edge (u, v) in E, at least one of the vertices u or v is in V'. The set V' is referred to as a vertex cover of G. The objective of the vertex cover problem is to identify the smallest possible vertex cover, meaning the vertex cover V' with the minimum number of vertices. This problem is known to be NP-complete, indicating that no known polynomial-time algorithm can solve all instances of the vertex cover problem efficiently for large graphs. (Tyagi & Batra, 2016)

Vertex cover has many variations but the one we consider in our study is parametrized vertex cover. Parameterized vertex cover is a variation of the classical vertex cover problem studied from the lens of parameterized complexity. Parameterized complexity is a subfield of complexity theory which is a field in computer science and mathematics that studies the computational resources required to solve problems. In the realm of parameterized complexity, these problems are analyzed with an additional consideration other than the size of the input and that is an additional parameter chosen for the specific problem. Parameterized complexity tries to reduce the computational running time of problems by introducing this parameter. The goal is to identify problems that are fixed-parameter tractable (FPT), meaning that they can be solved in time, that is polynomial in the size of the input and some fixed parameter. (Downey & Fellows, 2013)

Choosing the right parameter for the problem can be a hard decision to make, especially since this choice affects the running time of the parameterized problem. In the case of vertex cover one commonly chosen parameter is the maximum allowed size of the vertex cover denoted by K. The problem considered then becomes: Does there exist a set $V' \subseteq V$, with $|V'| \leq K$, such that each edge in E is incident to at least one vertex in V'?

Vertex cover is an NP-complete problem meaning a problem that is both in the complexity class NP and is also NP-hard. A problem is in NP if a given solution to that problem can be verified in polynomial time, and it is NP-hard if every problem in the NP class can be reduced (mapped) to that problem using a polynomial-time algorithm. (Niedermeier, 2006; Trevisan, 2011)

Being NP-complete means that solving vertex cover requires an exponential time algorithm if we assume that P!=NP while the parameterized version of vertex cover is fixed-parameter tractable which means that it can be solved in polynomial time in the size of the input and a fixed parameter. Solving such problems can be a burden which is why a lot of research has gone into algorithms to help optimize the running time of solving these problems. One of the most prominent algorithms is known as Branch & Reduce.(Downey & Fellows, 2013)

This algorithm has two main components, the first being the branching element and the second being the reduction. Branching is the process of using a search tree to find the answer to a certain problem. It involves dividing the problem into smaller subproblems until a base case or a leaf is reached. The second component, which is the reduction component, is the process of using problem-specific rules to reduce the size of the current problem. (Yamout et al., 2022) The two components work hand in hand to construct a search tree that branches on a decision at every level and then reduces the resulting instance from the decision until the optimal solution is found or proven to be unreachable.

In the case of Parameterized Vertex Cover, Branch & Reduce can be used to construct a search tree that divides the search space into smaller subproblems and systematically explores each subproblem in a recursive manner, which allows for a more efficient solution compared to brute-force methods in terms of running time. Yet the running time of Branch & Reduce can be improved further by utilizing many refinements. One of the most important factors affecting the size of search trees is vertex selection. Vertex selection is the process of choosing which vertex to branch on next. Branching on Vertex Cover consists of choosing a vertex where the left branch will be picking that vertex and including it in our vertex cover and the right branch will be choosing its neighbors and including them in the vertex cover instead. Whenever a vertex is included in a solution, we assume that it is deleted from the graph along with the edges incident on it resulting in smaller instances. In most cases, the vertex picked is the one with the highest degree or the vertex with the greatest number of neighbors. This is an intuitive choice that makes sense from a greedy perspective as eliminating this vertex would cover the greatest number of edges.

The main motivation behind the choice of the vertex and the application of reduction rules is to reduce the size of the search tree as that would in turn greatly reduce the running time of the Branch & Reduce algorithm which is why this dissertation is dedicated to studying the choice of the vertex selected to branch and making sure that this selection reduces the size of the search tree relative to other approaches most notably the high degree branching strategy. To optimize our decision of vertex we decided to employ the use of reinforcement learning which is a type of machine learning that involves an agent learning to make decisions in an environment to maximize a reward signal. (Sutton & Barto, 2018) Unlike supervised learning, reinforcement learning does not require labeled input/output pairs and instead focuses on finding a balance between exploration and exploitation of current knowledge. This paradigm makes the most sense as it is suited to optimize a decision-making process and because it does not need any data which in our case would prove to be almost impossible considering that obtaining enough data to make optimal decisions would require exponential time.

The main idea behind our approach is to train a reinforcement learning agent that observes specific features of the graph in question and then produces a weighting scheme for its vertex features that would enable us to make the decision on which vertex is best to branch on for the given graph in question. This approach would enable us to not only produce an agent that learns which features of a vertex are important to consider when branching but also explain which of these vertex features are the most important when considering this question.

In the sections to follow we will first look at the current literature on the problem in the related work section then we will discuss the methodology used to tackle the problem in question after which we will preview and discuss the results of our research and finally we will conclude our work while highlighting future areas this research could head in.

# CHAPTER II

# RELATED WORK

In this chapter, we will provide a thorough overview of the existing research surrounding vertex selection for branching. We will provide insight into methods relating to solving PVC and other parameterized combinatorial optimization problems. We will also explore the current selection strategies in the literature, which is predominantly max degree vertex selection, and finally, we will survey work relating to branching variable selection (BVS) when solving Mixed Integer Linear Programming using Branch & Bound which is where most of the work related to this topic exists. We will provide an overview of the techniques, findings, and limitations of the existing work and lay the groundwork for our method, introduced in the subsequent chapter.

## A. Solving Parameterized Vertex Cover

Vertex cover is one of the most studied problems in the field of parameterized complexity which means many algorithms have been devised to optimize its running time. In this section, we will go over the most famous algorithms and techniques commonly used to solve PVC and other parameterized combinatorial optimization problems.

### *1. Kernelization*

In the realm of parameterized algorithms, kernelization stands out as a cornerstone technique, especially in addressing PVC. Kernelization is a framework that provides a systemic way to reduce the size of the problem being solved to a smaller and more manageable version without changing the correctness and solvability of that problem.

The condensed version of the problem that is reached is called the “kernel”. In essence, this kernel is the crux of the problem or the part of the problem that is hard to solve.(Downey & Fellows, 2013; Fomin et al., 2019) Usually, after we get to the kernel, we can solve it exhaustively since we would have reduced the initial problem enough to enable us to do so without exploding the running time.

Kernelization revolves around the principle of preprocessing the input graph denoting the initial state of the problem until the process yields a kernel which is a reduced instance of that problem whose size is a function of the parameter k, rather than the overall size of the input. This is done through the application of a series of reduction rules. Each of these rules systematically identifies and removes and sometimes modifies parts of the graph in such a way that each decision made is one that is found in the optimal set of decisions. This technique significantly reduces the size of the search space for the solution and in turn reduces the running time of running an exhaustive method on the kernel. The most important part about kernelization is that we are guaranteed a solution for the original problem instance if the kernelized version has a solution. (Lokshtanov et al., 2012)

More formally kernelization is a technique used in parameterized complexity theory that aims to reduce the size of the problem instance while preserving the answer to the decision problem. It is a polynomial time algorithm for a parameterized problem P that transforms it from (I, K) [where I is the problem instance and K is the parameter that we parameterize the instance by] to (I’, K’) where I’ is the kernel and K’ is the new parameter. For kernelization to be valid:

1. The instance (I’, K’) is a yes instance of P if and only if (I, K) is a yes instance

2. The size of I’ and K’ are both bounded by a function of f(K)
3. The running time of the kernelization algorithm is polynomial time

The efficiency of the kernelization algorithm is measured by the size of the kernel it produces. This is obvious as smaller kernels generally mean that the exhaustive search algorithm is much more efficient. Advances in kernelization have led to kernels that are not only functions of k but are sometimes linear or polynomial. (Hespe et al., 2019)

Kernelization is a pivotal pre-processing technique in parameterized complexity that helps reduce the size of the problem significantly making the problem much easier to solve. It provides a structured framework that can be applied to a variety of problems. However, kernelization is not without its drawbacks as it is highly dependent on the size of the kernel that it produces. Furthermore, some kernelization algorithms can introduce their own significant computational overhead to the problem making its use less effective in reducing the overall running time.

## 2. *Crown Decomposition*

Crown Decomposition is a powerful technique used in combinatorial optimization, especially in the context of vertex cover. The main idea is to try to identify a particular structure in the graph known as a “crown”, and then to use that structure to simplify the graph and make the problem easier to solve.

A crown decomposition partitions the graph into three parts: a central subgraph C, a periphery subgraph H, and a reminder subgraph R. The periphery subgraph H is an independent set of vertices and there are no edges between R and H. The central subgraph C has a matching with the periphery subgraph H and that matching covers H. (Thomassé, 2009)

A crown decomposition is a useful tool that can be used when trying to find kernels for the graph. It is especially useful when trying to solve PVC where we can prove that by removing the central subgraph C the remaining subgraph R will essentially act as our kernel.

## ***3. Bounded Search Tree***

Bounded search tree is a technique often used while solving parameterized combinatorial optimization problems efficiently. The approach is based on the recursive division of the problem into smaller subproblems in a tree structure commonly known as a search tree. The idea is to branch on a decision for every node of the search tree in such a way that the original problem would be reduced for every branch. (Cygan et al., 2015)

More formally, a bounded search tree is a variant of exhaustive search that is used in solving combinatorial optimization problems by exploring their solution space through a systemic and controlled search process. Starting from the root, which is the original instance of the problem, bounded search tree subsequently branches on a decision to be made for the problem with each resulting child node representing a partial solution for the problem instance after applying the decision on it with the edges representing the transition from one state to the other based on the decision that was made. In the case of vertex cover this can be translated to a decision of choosing a vertex to be included in the vertex cover and the child node of the decision is the original graph with the vertex removed along with all its edges. The term “bounded” in this method refers to imposing a limit on the depth of the tree which is often directly correlated to the complexity of the problem. The parameter that bounds the tree is chosen based on the specific problem being solved. For vertex cover, that bound might be the size of the cover set.

The method is efficient for small parameter sizes, making it more effective than brute force approaches. Furthermore, the method is both optimal and complete as it can be used to find the best possible solution and because it can enumerate every possible solution within the bounds as well. Yet this method is far from perfect as it suffers from exponential growth of the running time as the number of branches increases. Efficiency is also highly dependent on the parameter chosen which means that for poorly chosen parameters the complexity of the method varies greatly. As a conclusion, this method offers a compromise between exhaustive search and computational feasibility making it a valuable tool in algorithmic problem solving for parameterized problem instances with a well-chosen parameter. (Downey & Fellows, 2013)

### 4. Iterative Compression

Iterative Compression is a technique used in the design of algorithms that are used to solve optimization problems. The technique is effective for problems where the solution can be built incrementally such as parametrized vertex cover. Iterative Compression makes use of the idea that it is easier to modify an existing solution that might be “too large” instead of trying to construct an optimal solution from scratch. (Downey & Fellows, 2013; Fomin et al., 2010)

The process of iterative compression is separated into several steps. We first start with an initial feasible solution. This solution might not necessarily meet the parameterized constraint. For example, we could start with a vertex cover of K+1 where K is the size of the maximum vertex cover allowed as a solution. The next step is the compression step, in this step, we take the current solution and try to compress its size, so if the solution is of size K+10 we apply the compression step which in the case of parameterized vertex cover could be removing the redundant vertices from the solution (vertices whose removal does not increase the number of uncovered edges). The compression step is then applied iteratively until a satisfactory solution is found.

Iterative compression offers a systemic technique to build a fixed-parameter tractable algorithm for problems that might seem intractable otherwise. It also uses a simple yet effective idea to achieve very efficient running times which is that oftentimes it is easier to build a solution iteratively rather than trying to find it from scratch. As with other techniques, Iterative compression has some drawbacks, mainly with the assumption that finding an initial solution is easily done which might not always be the case. Furthermore, the compression step itself might introduce a lot of complexity to the problem and might require very sophisticated algorithms to be able to compress the problem effectively.

### *5. Color Coding*

Color coding is a randomized algorithmic technique used to solve certain combinatorial optimization. It has been applied to a variety of problems including subgraph isomorphism, cycle detection, and path problems in graphs. The idea behind this method is to randomly assign colors to elements of the problem; in the case of parameterized vertex cover these elements are vertices of the graph. After the color assignment, a search for a "colorful" solution is initiated where each part of the solution has a distinct color. The essence of the color-coding technique is the observation that, if we color some universe with k colors uniformly at random, then a given k-element subset is colored with distinct colors with sufficient probability. (AlonNoga et al., 1995)

The process of color coding goes as follows: First, we begin with some hard problem P that we want to solve on instance I (a graph in this case). Second, we transform instance I to instance I' by coloring each vertex randomly. Third, we define an easier problem P' to solve on instance I' and we show that this problem can be solved quickly. Finally, we show that by repeating the problem a sufficient number of times we can find I' with a high enough probability. (Bannach et al., 2015)

Color coding is useful for tackling NP-hard problems that have been parameterized. It also does well at harnessing the power of parallelization as each attempt for coloring the graph can be done independently from one another. The disadvantages of using color coding and other randomized techniques lie in their random nature as having such an aspect does not guarantee a solution in any single run instead it provides a correct solution with a high probability after a sufficient number of iterations, however, it can be problematic as there will always be a non-zero probability of error, meaning that the algorithm might fail to find a solution ever if it did exist.

## *6. Branch & Reduce*

Branch and reduce is a fundamental algorithmic technique in parameterized complexity that combines ideas from Bounded Search Trees and Kernelization. The idea is to systemically explore the solution space by alternating between two key steps. The first is branching, where the problem is split into smaller subproblems based on a certain decision, and the second is reducing where the problem is simplified based on specific rules. (Ryoo & Sahinidis, 1996)

In the context of parameterized vertex cover the branching step involves selecting a node and creating two subproblems: one where we include the vertex in our vertex cover but not its neighbors and one where we include its neighbors but not the vertex itself. The first branch decreases the K by 1 while the second branch decreases the K by |Nv| (the size of the neighborhood of vertex v). After each branch a set of reduction rules is applied to each new sub-problem, these rules aim to simplify the graph without altering the existence of a K-vertex cover. Common reduction rules include removing isolated vertices and including vertices with a degree greater than K in the vertex cover.

The interaction between both the branching and the reduction is crucial for the efficiency of the algorithm. Well-designed branching rules create a balanced search tree which reduces the running time significantly while effective reduction rules can significantly decrease the size of the problem which can effectively prune large parts of the search space.

The development of efficient branch and reduce algorithms has been a major focus in the last couple of years in the area of parameterized complexity. For parameterized vertex cover, the work of Buss and Goldsmith introduced a simple branching algorithm with a running time of $O(2^K.n)$ , where n is the number of vertices(Buss & Goldsmith, 2006). Improvements by Niedermeier and Rossmanith, and by Chen, Kanj, and Xia, led to the current best known FPT algorithm for PVC , running in time $O(1.2738^K + K.|N|)$ (Chen et al., 2010; Niedermeier & Rossmanith, 2003). Beyond parameterized vertex cover, branch and reduce has been applied to many parameterized problems in different domains most notably Feedback Vertex Set, Cluster Editing, and Dominating Set, among others. (Gaspers & Liedloff, 2006)

Branch and reduce offers significant advantages. Firstly, branch and reduce is very versatile which means it can be applied to a wide range of problems. Secondly, unlike approximation algorithms, branch and reduce finds exact solutions for the problem. Finally, since it is closely related to bounded search trees this technique shares many of the same advantages. This technique is not without its drawbacks as finding the right branching and reduction rules can be a complex task. It also suffers from high memory and computational usage.

Throughout this dissertation, our focus will be on optimizing this technique. Specifically, our focus will be on designing an intelligent branching rule based on reinforcement learning techniques that would help reduce the size of the search tree significantly.

## 7. *Linear Programming*

Aside from algorithmically solving parameterized vertex cover, linear programming is a promising approach to achieving that goal. Integer Linear Programing (ILP) provides a robust framework for formulating and solving discrete optimization problems including combinatorial optimization. Employing ILP to solve parameterized vertex cover requires us to construct a linear model that reflects the constraints and objectives of the problem (Lancia & Serafini, 2018). The formulation is detailed below:

Objective Function

$Minimize \sum_{i=1}^{|v|} x_i$

Subject to the constraints:

$x; + x_j \geq 1 \quad \forall (v_j, v_i) \in E$

$x_i \in \{0,1\} \quad \forall i \in \{1,2, \dots |v|\}$

$\sum_{i=1}^{|v|} x_i \leq k$

Where:

- |V| is the number of vertices.
- vi is vertex i
- xi is a binary variable associated with every vertex
- K is the parameter of the problem
- E is the edge set

In this formulation (Chekuri, 2009),  we are trying to the number of binary variables xi that are set to 1. If a binary variable xi is set to one it corresponds to including vertex vi into the vertex cover. The first constraint details the fact that we must include at least one of every edge in our vertex solution so that our choice could cover that edge. The second constraint just forces the variable xi to be a binary variable. Finally, the last constraint ensures that the size of the vertex cover is less than K.

Using this formulation, we can solve parameterized vertex cover using a commercial ILP solver like Gurobi (Gurobi, 2024) or CPLEX (IBM, 2024). These solvers take in an ILP formulated problem and output a valid solution that meets all the constraints. These solvers work by utilizing techniques used for optimization problems such as the branch-and-cut method (which is similar to branch and reduce) to efficiently explore the feasible region of the problem by branching on variables and trying to find the optimal solution that matches the objective function while adhering to the constraints.

While ILP presents a theoretically robust solution for parameterized vertex cover, its performance however heavily relies on the size of the graph and the parameter k. As this increases, the complexity of the solution space can escalate and go beyond practical reach. Despite this drawback, ILP still remains a valuable tool in the exact algorithm toolkit that offers practical applicability for solving parameterized vertex cover.

## 8. *Machine Learning*

The focus of research in this area has been towards solving the unparameterized version of vertex cover among other combinatorial optimization problems like the traveling salesman problem and feedback vertex set. Machine learning has been an increasingly attractive option as the complexity associated with solving these problems algorithmically is huge with them being NP-Hard problems. As such these problems present an ideal domain for the application of machine learning techniques that can approximate solutions to a very high level of accuracy in only a fraction of the time. We will explore some of the techniques that have been used to solve these problems using machine learning.

### a. <u>Supervised Learning</u>

Supervised learning is a machine learning method that requires a model to learn on a labeled set of data. The method seeks to learn a mapping from the input features to the output labels. In the context of vertex cover, the input features can be derived from the graph properties such as node degrees, centrality measures, and local graph structures, while the output labels can indicate whether a vertex is part of the minimum vertex cover or not.(Bengio et al., 2021; Nasteski, 2017)

One approach to applying supervised learning on vertex cover is to train a decision tree. Decision trees are a traditional machine learning technique that learns the hierarchy of if-then rules by on the input features. By training these trees on a graph dataset with labels for their minimal vertex covers, the model could learn to predict which nodes should be included in the vertex cover based on their features. This approach has been explored in studies addressing similar NP-hard problems, such as the maximum independent set problem and the minimum dominating set problem,(Dai et al., 2017) providing a precedent for its application to vertex cover.

Another supervised method that is employed is SVM or support vector machines. SVMs are a powerful binary classifier that seeks to find a hyperplane in the feature space that best separates two classes. For this to work on vertex cover we can label the vertices into “in vertex cover” and “out of vertex cover” and make the SVM learn the mapping between the vertex features and their label. SVMs have been successfully applied to various graph-related problems,(Bagattini et al., 2018) such as graph classification and link prediction, demonstrating their potential for addressing the vertex cover problem.

### b. Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make a sequence of actions through interacting with an environment. The agent interacts with the environment which is represented by state S at every timestep by taking an action A and as such receiving a reward R (determined by the reward function) while moving to state S’ (determined by the transition function). The objective of this agent is to learn a policy that would maximize the cumulative reward over time. Reinforcement learning has been successfully applied to a wide range of sequential decision-making problems including game-playing, robotics (Polydoros & Nalpantidis, 2017) , and resource management. (Mazyavkina et al., 2020)

Vertex cover is a very suitable problem for reinforcement learning as it can be used to develop intelligent strategies for incrementally constructing a vertex cover. The environment can be modeled as a graph with each state being a graph G and a partial vertex cover VC while the agent's actions correspond to selecting a vertex to include in the vertex cover. The rewards could be designed to encourage the agent to find the minimal vertex cover. For instance, the agent could receive a positive reward for each edge that it covers with a negative reward for every additional vertex that it includes.

Several RL algorithms, such as Q-learning and policy gradient can be employed to learn effective strategies for solving the vertex cover problem. Q-learning is a value-based method that learns a function called action-value or Q. This function estimates the expected cumulative reward for taking a particular action in a given state. Policy gradient methods on the other hand seek to directly optimize a policy that maps states to actions using gradient ascent to maximize the expected reward. (Sutton & Barto, 2018)

The use of RL in solving combinatorial optimization has shown promising results. For example,(Bello et al., 2016) applied policy gradient methods to the traveling salesman problem, learning policies that outperformed traditional heuristics. Similarly, (Dai et al., 2017) used Q-learning to address the maximum independent set problem, demonstrating the effectiveness of RL for solving NP-hard graph problems. These successes suggest that RL could be a valuable approach for tackling the vertex cover problem as well.

### c. Deep Learning

Deep learning has revolutionized many fields including computer vision, natural language processing, and robotics. It has done that by enabling complex neural networks to learn high-level embeddings from raw data. In the context of combinatorial optimization, deep learning has been adapted to learn meaningful representations of problem instances that helped guide the search for optimal solutions. (K. Li et al., 2021)

In the context of vertex cover deep learning has been applied successfully many times. One approach used convolutional neural networks or CNNs to extract relevant features from the graphs to classify the nodes of the graph. However, CNNs perform well on structured data which requires us to transform the graph to a grid-like format such as an adjacency matrix which can then be fed into the CNN to make a prediction. This technique was used by (Gu & Yu, 2014) who proposed a CNN-based approach for solving the maximum clique problem.

Another deep learning technique that has proved its merit is recurrent neural networks or RNNs. RNNs are neural networks designed to process sequential data like the movement of something over time. This makes them suitable for problems that require making a series of sequential decisions much like choosing the minimal vertex cover. In this context, RNNs can be used to generate a sequence of vertices to include in the vertex cover based on the graph structure of the previous graphs. (Bello et al., 2016) employed a combination of RNNs and reinforcement learning to solve the traveling salesman problem or TSP where they introduced an RNN-based policy network that generates a sequence of cities to visit and another critic network that evaluates the quality of the generated solution. This method is referred to as an actor-critic method which is a method often used in reinforcement learning and which we use in this dissertation as well. A similar approach can be used for vertex cover where the RNN would generate a sequence of vertices to include in the vertex cover and the a critic network would evaluate each choice being made.

d. Graph Neural Networks

A subfield of deep learning that is very relevant to our particular use case is graph neural networks or GNNs. GNNs are a class of deep learning models that are specifically designed to operate on graph-structured data. In recent years the use of GNNs has skyrocketed due to their ability to learn rich representation from graph data that capture both global and local information. GNNs have been successfully applied to a wide range of graph-related problems such as node classification, link prediction, and graph classification. (Wu et al., 2021)

The key idea behind GNNs is to iteratively update the representation of each node by aggregating information from its neighbors in a process referred to as message passing. GNNs are composed of two main components: a message-passing component and an update component. In the message-passing phase, each node sends its current representation combined with its features as a message to all neighboring nodes. This representation is often captured by passing the node features in a fully connected layer that aims to learn valid embeddings from the vertex features. These messages are then aggregated using a permutation invariant function like average or sum. In the update phase, each node updates its representation based on the aggregated messages received from its neighbors and its previous representation. This updated representation captures local the context of the node in the graph. The message-passing and update phases are repeated for a fixed number of iterations or until a desired level of abstraction is reached. By stacking multiple GNN layers, the model can learn hierarchical representations that capture increasingly global information about the graph structure. The final node representations, often referred to as node embeddings, can be used for various downstream tasks, such as node classification, link prediction, or, in the case of the vertex cover problem, predicting the likelihood of a node being part of the minimum vertex cover. (Scarselli et al., 2009)

Various GNN architectures, such as Graph Convolutional Networks (GCNs) (Kipf & Welling, 2016), GraphSAGE (Hamilton et al., 2017), and Graph Attention Networks (GATs) (Veličković et al., 2017), have been proposed to handle different types of graph structures and learning tasks. For instance, GCNs use a message-passing scheme to aggregate information from neighboring nodes, while GATs employ attention mechanisms to weigh the importance of different neighbors during the aggregation process.

This method can be applied to the vertex cover problem by learning sufficient node embeddings to determine if a node should be included in the vertex cover or not. In fact, GNNs have been proven to be effective in solving combinatorial optimization problems by (Z. Li et al., 2018) used a combination of GCNs and reinforcement learning to address the minimum vertex cover problem, learning policies that outperformed traditional heuristics.

## B. Branching Rules

Solving Parameterized Vertex Cover using branch and reduce requires one to pick a branching rule to branch on every decision. This decision often significantly impacts the efficiency of the algorithm. Branching rules determine how the problem will be split into smaller subproblems at each branching step. In the literature, a handful of these rules have been proposed each comes with its own set of pros and cons, and in this section, we will discuss some of these rules.

The simplest and most used of these branching rules is the high-degree branching rule. This rule selects the vertex with the highest degree to branch and subsequently creates two branches in the tree. The first branch includes the vertex itself as part of the vertex cover and one where it is excluded but instead, its neighbors are included as part of the vertex cover. This is because every vertex cover will include this node or all of its neighbors as it would have to cover every edge in the graph including the edges associated with this node. The intuition behind this rule is that the high-degree vertices are more likely to be part of the minimum vertex cover as they would naturally cover more edges than other vertices. This is a greedy approach to solving this problem but has been widely used in practice and has proven its efficacy over time. (Chang et al., 2016; Lokshtanov et al., n.d.; Wang et al., 2019)

Another popular branching rule is the edge branching rule that selects an edge (u, v) and branches on it, creating two subproblems: one where u is included in the vertex cover and one where v is included instead. This rule has proven to be effective for dense graphs where the average degree is high and thus the high degree branching algorithm would not yield as good of results. By branching on edges instead of vertices, the edge branching rule can lead to more balanced subproblems and faster convergence for vertex cover.

Additionally, other more sophisticated techniques have been proposed to improve the efficiency of the branching strategies like those in (Harris & Narayanaswamy, 2022)

## C. Variable Selection

Variable Selection, also known as branching variable selection or node selection is an important component in branch and bound algorithms used in solving integer linear programming problems. The process involves choosing which variables to branch on at each step of the algorithm to efficiently navigate the solution space and to converge efficiently to the optimal solution. Variable selection works as follows: if we have a problem where we want to maximize an objective function that includes a variable x, we could create two branches for x, one where x <= 4 and one where x >= 5. This means that for the first branch, we include an extra constraint for our ILP problem which is that the decision variable x has to be equal or less than 4. This would help reduce the search space significantly for that branch and thus reduce the complexity of the problem significantly. The question would then be what variables should we branch on every branching step to create an optimal search tree in terms of computational complexity?

The reason why variable selection is important in our context is because of two reasons: the first of which is that it has been widely studied in the literature over the years using a huge variety of methods, and second is because it is very similar to the problem we are tackling in this dissertation which is choosing a vertex to branch on in a branch and reduce algorithm for parameterized vertex cover. These reasons make studying this problem important to guide this dissertation.

One of the seminal works in the area was by (Khalil et al., 2016) where the authors proposed using imitation learning to learn branching policies. This means that they trained a model to learn to imitate a technique called strong branching to pick the best variable. Strong branching is a sophisticated technique used in branch and bound algorithms that tentatively branches on several candidate variables and calculates the potential impact of each branch. The impact is a measure of how much the objective function improves or how much the size of the tree is expected to be reduced. The node with the most promising score is then picked. The issue with strong branching is that it is computationally expensive which is why in this work Khalil et al. tried to cut down the running time by essentially training a model to be able to closely approximate the strong branching outcome which would cut the running time significantly. To do so they employed a learning-to-rank approach to imitate the strong branching strategy and demonstrated how it led to speed up over the SCIP solver on benchmark instances.

Building upon this work was (Alvarez et al., 2017) who proposed an alternative approach of approximating strong branching scores using a linear regression model. Instead of learning on a per-instance basis, they trained the model offline to be able to predict strong branching scores of nodes. This approach embedded each subproblem by a set of features and then used these features to predict the outcome of the strong branching algorithm.

Another advancement came shortly after with the work (Gasse Mila et al., 2019),who introduced a graph convolutional network (GCN) to help learn branching strategies. The idea was to represent the ILP problem as a bipartite graph where the left side nodes were the constraints and the right side where the variables and there would be an edge between both sides if the variable was included in the constraint. Once the problem was represented by a bipartite graph a GCN would be employed to be able to approximate a variable selection policy based on the bipartite graph input. The data required for training this supervised method would be generated using strong branching and stored for use during training. This method achieved state-of-the-art results, especially showcasing an ability to generalize to bigger graph sizes than those seen during training which is a core consideration when training such models.

In addition (Gupta et al., 2020) were able to propose a more efficient hybrid architecture that combines both Graph Neural Networks and Multi-layer Perceptrons to learn a branching heuristic for MILP. The key idea was to use the expressive but computationally expensive power of the GNN at the root not of the branch and bound tree to extract relevant information (since the structure doesn’t usually change that much along the subsequent nodes), and then to use the less computationally expensive MLP to extract information about the subsequent nodes to make branching decisions.

Finally, (Lin et al., 2022)proposed a novel tree aware branching transformer (T-BranT) framework that also learns variable selection policies for heterogeneous mixed integer linear programming problems. The idea was to input 3 sets of information to the model: first the local candidate features that capture each candidate’s importance, second local tree features that capture the structure of the decision tree so far this could give the model insight into how far along the problem it is in, and third past branching node features that show the model the decisions and features at previous nodes; this would provide the model with the necessary temporal information to make an informed decision not only based on the current features but also on the features of previous timesteps. To do this (Lin et al., 2022)train a novel transformer and GAT architecture that integrates local, global, and temporal features of the problem. T-BranT generalizes well across widely heterogeneous problems and outperforms previous tree-parameterized approaches in terms of solution and branching decision quality, especially on harder instances. This work demonstrates the potential of advanced neural architectures and global tree representations for learning effective and generalizable branching policies.

In summary, these papers showcase the progression and potential of applying machine learning techniques, particularly imitation learning, to enhance variable selection strategies in MIP solvers. The approaches range from learning branching rules offline on similar instances to learning policies on a per-instance basis using graph representations or search tree features. The representation of subproblems using search tree states has emerged as a promising direction for achieving generalization across heterogeneous instances. Overall, this line of research has gained significant attention and has shown impressive results in improving the efficiency of MIP solvers.

# CHAPTER III

# METHODOLOGY

In this chapter, we present our methodology for learning efficient branching strategies in the branch-and-reduce paradigm for solving the parameterized vertex cover problem. We will begin by giving a formal definition of the problem, followed by a definition of the branching environment including the reduction rules employed. We will then provide an overview of the method used to tackle the problem which includes a trained model that produces weights for node features given graph features. After that, we will describe the state representation used in our RL formulation which consists of the graph features of the current problem instance. We will also discuss the node features used for every node upon which we will apply the weighting produced by the trained model. Finally, we will discuss details in general about reinforcement learning and detail our use of the Proximal Policy Optimization (PPO) algorithm for training, and the two training paradigms used to train our model.

## A. Definition of problem

Parameterized Vertex Cover (PVC) is a combinatorial optimization problem formally defined as the following: Given an undirected graph G = (V,E), where V is the set of vertices that belong to graph and E is the set of edges of the graph, along with a parameter K, the objective is to determine whether there exists a vertex cover of size at most K. (Downey & Fellows, 2013) A vertex cover is a subset of vertices $S \subseteq V$ such that for every edge $e \in E$ at least one of its endpoints is part of the vertex cover. In other words, the task of Parameterized Vertex Cover is to find a subset of vertices with size at most K such that removing this set of vertices would produce a graph without any edges or an independent graph. Parameterized Vertex Cover is a well-known problem with many applications in a variety of domains, including network security, bioinformatics, and social network analysis.

As discussed in the previous section, Parameterized Vertex Cover is a well-studied problem with many different methods of solving it. One of the methods which we will focus on in this dissertation is the Branch-and-Reduce method. This method as indicated previously is an algorithmic paradigm used to solve PVC and other parameterized problems. This method includes two main components: The first is branching, where the problem is split into smaller subproblems based on a certain decision, and the second is reducing where the problem is simplified based on specific rules. The algorithm starts by applying reduction rules on the graph which would simplify the problem instance, this could be something like removing the vertices with degree greater than K and including them in vertex cover. The algorithm then proceeds to recursively branch on a selected vertex which would create two subproblems one that includes the selected vertex in the vertex cover while removing it from G and one that includes all its neighbors in the vertex cover while removing them from G, while applying reduction rules after every branching step.

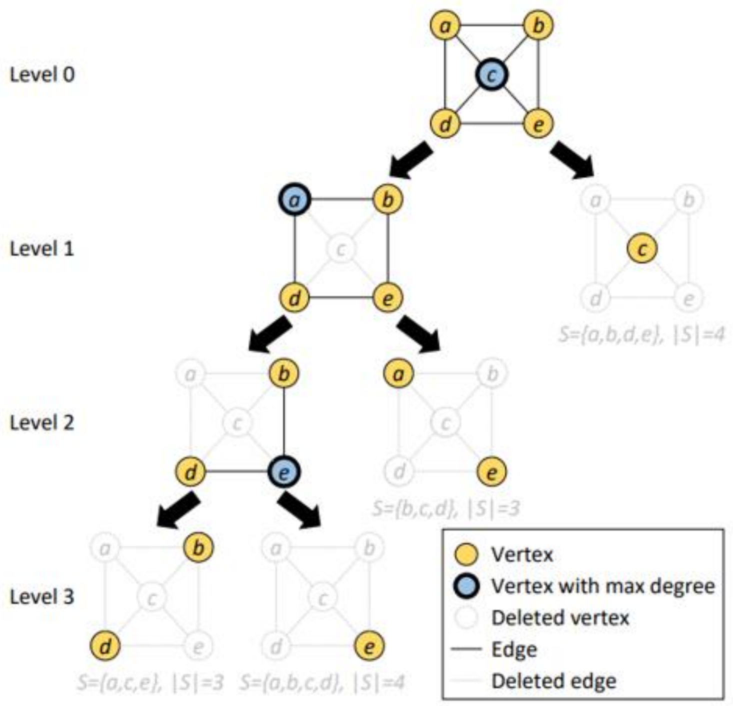


Figure 1. Branch and Reduce

The performance of Branch-and-Reduce heavily depends on the choice of the vertex to branch on at each step. Particularly, well-chosen vertices would lead to a smaller branching tree size which would minimize the running time of the algorithm. As discussed previously, the most used branching strategy in practice is branching on the vertex with the highest degree but it is not proven to be optimal. This is why in this work we aim to learn an effective branching strategy for solving parameterized vertex cover using branch and reduce by employing reinforcement learning to do so. By training a policy network to select the most promising vertex to branch on.

## B. Overview of the Solution

To solve the problem of finding the best vertex to branch on for Branch-and-Reduce we propose a novel reinforcement learning approach to effectively pick a vertex based on the structure of the graph. Below we detail how the trained model would behave:

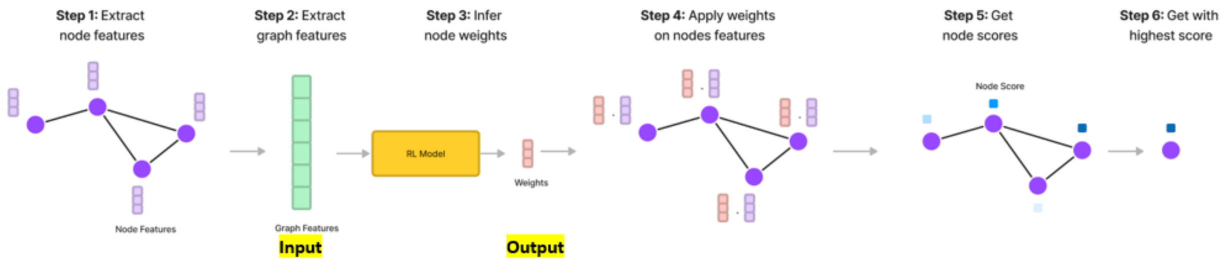

Step 1: Extract node features
Step 2: Extract graph features
Step 3: Infer node weights
Step 4: Apply weights on nodes features
Step 5: Get node scores
Step 6: Get with highest score
Node Features
Graph Features
Input
RL Model
Weights
Output
Node Score

Figure 2. Overview of Model Inference

The method depicted in the figure is an RL-based approach for making branching decisions for Branch-and-Reduce applied on parameterized vertex cover. The general idea is for the model to predict a set of node feature weights. These weights are inferred based on the graph structure which is implied from a set of curated graph features. Once the weights are inferred they would be used to determine which node features are the most important when considering which node Branch-and-Reduce should branch on. This is done by calculating a score based on those node weight features. We detail the exact steps of the method:

1. **Extract Node Features:** Extract a vector of node features for every node in the graph we want to solve parameterized vertex cover on. This step is essential as it captures the attributes of each node in the graph such as the degree, centrality measures, and other metrics. These features help us distinguish each node from one another and give each one a profile which the reinforcement learning model would choose to branch on.
2. **Extract Graph Features:** Extract a set of features for the graph that offers a holistic view and describes the overall structure of the graph. These features include

the density, average degree, and even temporal features based on the branching tree constructed so far. This set of features will be the input to our reinforcement learning agent and will allow it to make informed decisions by considering the structure of the graph.

3. **Infer Node Weights:** Infer using the trained reinforcement learning agent a set of weights to be applied to the node features. This effectively adjusts the importance of each node feature to the branching decision based on the graph's global features. This step essentially infers which node profile is ideal to branch on based on what the graph looks like.
4. **Apply Weights on Node Features & Get Node Scores:** This step applies the inferred weights to the node features. What this means is that the inferred weights are taken and the dot product between these weights and the node features is calculated for each node. This will provide us with a final score for each node which indicates how good it is to branch on each of these nodes.
5. **Take the Node with the Highest Score:** Once the score for every node is calculated the node with the highest score is selected and branched on. This entails creating two branches, one where the selected node is included in the vertex cover and removed from the graph, and another where its neighbors are included in the vertex cover and removed from the graph. Each branch's graph is then reduced by applying the reduction rules to each of them. This step is handled in our branching environment.

Overall, this process systemically transforms raw graph data into a sophisticated decision-making framework that leverages this raw data to give insight into which node features are most important during the branching process based on the current graph's structure. The next section will go into more detail on every component of the approach.

## C. Features for nodes

Node-level features are crucial in guiding the branching process in our formulation. Particularly these node features will be used to capture the local properties of individual vertices. Our model aims to produce proper weightings for these features to indicate at every step which features are the most significant to focus on while branching. These weights are multiplied by the node feature they correspond to, and all the values are added to get the final score for the vertex. All features were normalized and set to values between 0 and 1. The following node features are considered in this work:

- **Degree:** The number of edges incident to the vertex. The degree indicates the local connectivity and the number of neighboring vertices it has. Vertices with higher degrees are more likely to be part of the vertex cover as they would cover the greatest number of edges. In the typical high-degree branching rule where we branch on the vertex with the highest degree, the degree would have a weight of 1 while all other node features would have a degree of 0. This means that while considering which node to branch on we only consider the degree, in that sense our approach is a generalization of this method that tries to find the optimal weights for every graph.
- **Clustering Coefficient:** The clustering coefficient measures the proportion of the vertex's neighbors that are also neighbors to each other. It is calculated using this formula $\frac{2T(v)}{\deg(v)*\deg(v)-1}$ where T(v) is the number of triangles that vertex v is part of and deg(v) is the degree or number of neighbors of that vertex. This coefficient is important because it captures the

local density of the vertex's neighborhood. Vertices with higher clustering coefficients are part of tightly connected subgraphs which could influence the branching choice.

- **Coreness:** The coreness of a vertex is the maximum k-core value it belongs to, where a k-core is a maximal subgraph in which every vertex has a degree of at least k. The coreness of a vertex v is calculated by iteratively removing vertices with degrees less than k until no more vertices can be removed. The coreness of v is the highest value of k for which it remains in the graph. The coreness provides a measure of the centrality and importance of a vertex within the graph. Vertices with higher coreness values are more deeply embedded in the graph structure and may have a greater impact on the vertex cover.
- **PageRank:** PageRank computes a ranking of the nodes in the graph G based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages. PageRank is computed iteratively using the formula:

  $$PR(v) = \frac{(1-d)}{N} + d * \sum \frac{PR(u)}{\deg(u)}$$

  PageRank captures the global importance and influence of a vertex based on the structure of the graph. Vertices with higher PageRank scores are more likely to be part of the vertex cover and may have a greater impact on the overall solution.
- **Second Degree Neighbor:** The second-degree neighbors of a vertex are the vertices that are two hops away from it in the graph. The number of second-degree neighbors provides information about the broader neighborhood of a

vertex and its potential impact on the graph structure.

- **Closeness Centrality:** Closeness centrality measures the average shortest path distance from a vertex to all other vertices in the graph. It is calculated using this formula: $\frac{N-1}{\sum_y d(y,v)}$ where d(y,v) is the distance from y to v. Closeness centrality captures the notion of how close a vertex is to all other vertices in the graph. Vertices with high closeness centrality are more central and can reach other vertices quickly, which may be relevant for the vertex cover problem.
- **Betweenness Centrality:** Betweenness centrality measures the extent to which a vertex lies on the shortest paths between other pairs of vertices. It is calculated as follows: $\sum_{s,t \in V} \frac{SP(s,t,v)}{SP(s,t)}$ where SP(s,t) is the number of shortest paths from vertex s to vertex t, and SP(s,t,v) is the number of those paths that pass through vertex v. Betweenness centrality identifies vertices that act as bridges or bottlenecks in the graph. Vertices with high betweenness centrality may have a significant impact on the connectivity and structure of the graph, making them potential candidates for branching.
- **Eccentricity:** The eccentricity of a vertex is the maximum shortest path distance from that vertex to any other vertex in the graph. Eccentricity provides a measure of how far a vertex is from the farthest vertex in the graph.
- **Density, Average Clustering Removed:** 2 features that indicate the density and average clustering coefficient of the graph after removing a vertex and its incident edges. These features capture the change in the graph when a

vertex is removed. Vertices whose removal leads to a significant decrease in density and average clustering coefficient may be important for the vertex cover problem.

- **Density, Average Clustering Neighbors Removed:** 2 features that indicate the density and average clustering coefficient of the graph after removing the neighbors of the vertex.
- **Reducible Feature:** Indicates the number of vertices that would be removed due to the reduction rules if the vertex was removed from the graph. This provides insight on how removing the vertex might affect the size and complexity of the graph instance which would affect the branching decision process.

## D. Features for graphs (Input for Model)

In this section, we present a detailed description of the graph features used as input to the reinforcement learning agent used to make branching decisions in the branch-and-reduce algorithm. The graph features are also normalized to be between 0 and 1. We will define each feature formally as well as its calculation, and its significance in the context of the problem:

- **Density:** The ratio of the actual number of edges to the maximum possible number of edges in the graph. We calculate the density using this formula $\frac{2*|E|}{|V|*|V-1|}$ . The density is an important metric that helps us measure the overall connectivity and completeness of the graph.
- **Clustering Coefficient Measures (Average, Standard deviation, Variance):** The measure of the average clustering coefficient as well as its

standard deviation and variance for all nodes in the graph. The clustering coefficient for 1 vertex is calculated as follows: $\frac{2T(v)}{\deg(v)*\deg(v)-1}$ where T(v) is the number of triangles that vertex v is part of and deg(v) is the degree or number of neighbors of that vertex. These measures are important to show the tendency of vertices to cluster together and form tightly connected graphs.

- **Degree Measures (Average, Standard Deviation, Variance):** These are 3 features that quantify the degrees in the graph. The degree is the measure of how many neighbors a vertex has. We take the average, standard deviation, and variance for all vertices in the graph. The average degree provides an understanding of the typical connectivity, while its dispersion metrics inform on the uniformity or concentration of edges across the graph.
- **Betweenness Centrality Measures (Average, Standard Deviation, Variance):** Involves 3 measures for the betweenness centrality which is also a node feature that we get the average, standard deviation and variance for all vertices in the graph. The betweenness centrality measures the extent to which nodes act as bridges along the shortest path between other nodes. It is calculated as follows: $\sum_{s,t\in V}\frac{SP(s,t,v)}{SP(s,t)}$ where SP(s,t) is the number of shortest paths from vertex s to vertex t, and SP(s,t,v) is the number of those paths that pass through vertex v. This measure is very important to get an idea of how many bridge like nodes exist in the graph.
- **Closeness Centrality Measures (Average, Standard Deviation, Variance):** 3 descriptive statistics for the closeness centrality of the vertices of the graph that include their average, standard deviation, and variance. The

closeness centrality is the reciprocal of the sum of the shortest path distances from a vertex to all other vertices. It is calculated as follows for 1 vertex: $\frac{N-1}{\sum_y d(y,v)}$ where d(y,v) is the distance from y to v. This measure gives a sense of the proximity of the vertices to other vertices in the graph.

- **Eccentricity Measures (Average, Standard Deviation, Variance):** 3 descriptive statistics for the vertices of the graph that include their average, standard deviation, and variance. The eccentricity of a vertex is the maximum shortest path from the vertex to any other vertex in the graph. These measures indicate how far apart vertices in the graph are which could show how stretched a graph is.
- **Transitivity:** The ratio of the number of the number of triangles to the number of connected triples in the graph. A triangle is a set of three vertices that are all connected to one another. A connected triple is a set of three vertices where at least one vertex is connected to the other two. The transitivity is computed as $\frac{3*num\_triangles}{num\_connected\_triples}$ . This measure is significant as it helps quantify the presences of cliques or tightly connected subgraphs in the graphs.
- **Radius:** The minimum eccentricity of any vertex in the graph. This measure indicates the minimum distance to reach all vertices from any single vertex in graph and it shows how far apart or close vertices in the graph are.
- **Number of Connected Components:** The count of connected components in the graph. It is obtained by performing a depth-first search or breadth-first search traversal on the graph and counting the number of disjoint components. This measure is significant because it shows the connectivity

and fragmentation of the graph.

- **Normalized Average Shortest Path:** The average shortest path length between all pairs of vertices, normalized by the maximum possible path length. It is computed as the average of the shortest path distances between all pairs of vertices, divided by the maximum possible distance (i.e., the number of vertices minus one). It provides a normalized measure of the average distance between vertices in the graph.
- **Normalized K:** With K being the parameter of the problem. This measure is computed as the current value of K divided by the initial value of K. It indicates the proportion of the remaining budget relative to the initial budget, providing a measure of the problem's progress.
- **Degree Bucketed:** This feature is a binary bucketing mechanism for the degree based on the average value across the graph, it divides the vertices into two categories: those below the average and those at or above the average.  The significance of this approach lies in its ability to reduce complexity and potentially improve the interpretability and efficiency of the degree.
- **Temporal Features:** This is composed of 5 features for the parent node of the current graph in the search tree. These features include density, normalized k, and the three-degree measures (average, standard deviation, and variance of the degrees in the graph). In other words, these features are essentially some graph features of the predecessor graph in the search tree. These features help us get a sense of the temporal evolution of the graph across the search tree.

## E. Branching Simulator

In this section, we will discuss the Branch-and-Reduce simulator we implemented to apply parameterized vertex cover. The functionality of this simulator is to produce a branching tree given a starting graph G and a set of vertex decisions on which branching will occur.

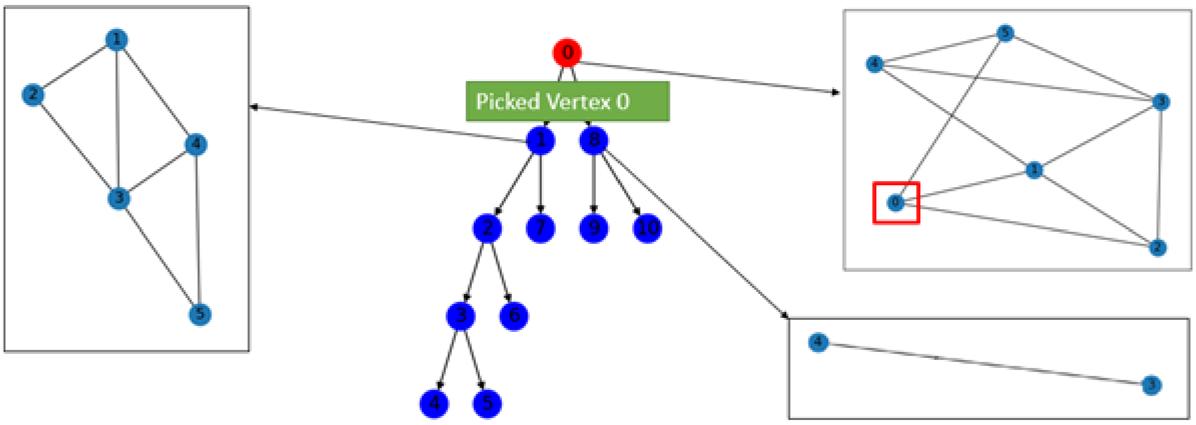


Figure 3. Branching Simulator

As shown in the figure, the simulator is a decision tree where each node is a graph. Starting from the root with the original graph we want to solve parameterized vertex cover on. The simulator also includes the parameter K which limits the size of the vertex cover. It works as follows:

- Starting from the root a vertex is given to the simulator.
- The simulator creates two branches one where it includes the vertex chosen in the vertex cover and decrements k by 1 and one where the neighbors of the vertex are chosen and included into the vertex cover and k is decremented by |N| (the length of the of set of neighbors of the chosen vertex). This step is valid based on the logical premise that for any edge in the graph, at least one of its

endpoints must be included in the vertex cover to satisfy the problem's constraints. By exploring both possibilities—either the vertex itself or its neighbors being in the cover—we ensure that all potential configurations are considered, and the solution space is thoroughly searched for an optimal or near-optimal solution.

- After creating each branch, the reduction rules are applied to each branch’s graph respectively and K is decremented based on how many vertices are removed during the reduction process
- This process is repeated every time we give the simulator a new vertex to branch on in a depth-first manner until the search tree is exhausted as we are studying the worst-case scenario performance.

## F. Reduction rules

As mentioned before applying reduction rules is one of the two main components of the Branch-and-Reduce algorithm. Particularly, reduction rules are applied after every branching decision to simplify the graph structure and reduce the search space (Yamout et al., 2022). This allows for more efficient exploration of the solution space. In this section, we present the four reduction rules we used in this dissertation for the branch and reduce algorithm.

- **Removing Vertices of Degree 0:** The first reduction rule focuses on vertices with a degree of 0, also known as isolated vertices. If a vertex has no neighbors, it cannot be part of any vertex cover since it does not cover any edges. Therefore, such vertices can be safely removed from the graph without affecting the optimal solution.

- **Removing the Neighbor of Vertices of Degree 1:** The second reduction rule deals with vertices of degree 1, also known as pendant vertices. If a vertex has only one neighbor, that neighbor or the vertex must be included in any vertex cover to cover the edge between them. Since the vertex only covers 1 edge but the neighbor might cover more than 1 edge it is always better to include the neighbors in the vertex cover and then remove both vertices. After applying this rule, the value of K (the parameter in the problem instance) is reduced by 1 to indicate that we included the neighbor of the degree 1 vertex in the solution.
- **Degree 2 Neighbors Rule:** The third reduction rule addresses the case where two vertices of degree 2 are neighbors of each other. In this scenario, if both vertices are not already removed, they can be safely removed from the graph and added to the vertex cover. This is because any vertex cover that includes one of these vertices must also include the other to cover the edge between them. After applying this rule, the value of K (the parameter in the problem instance) is reduced by 2 to indicate that we included the 2 vertices in the solution.
- **Removing Vertices of Degree Greater than K:** The fourth and final reduction rule focuses on vertices with a degree greater than the parameter K, which represents the maximum size of the vertex cover. If a vertex has more than K neighbors, it must be included in the vertex cover since at least one endpoint of each incident edge needs to be covered, which means that we have to choose from including the vertex or all of its neighbors, but since including all of the neighbors will produce an infeasible solution the

decision to include the vertex itself becomes a necessary one. After applying this rule, the value of K (the parameter in the problem instance) is reduced by 1 to indicate that we included the vertex in the solution.

These rules are applied in a loop to the graph instance until no more rules can be applied which would indicate that the graph can no longer be simplified using reduction rules. The reduction rules are important for the efficiency and correctness of the Branch-and-Reduce algorithm in solving the Parameterized Vertex Cover problem. By removing vertices and edges that can be safely excluded or included in the solution, these rules help in simplifying the problem and reducing the search space, ultimately leading to faster convergence towards the optimal vertex cover.

## G. Reinforcement Learning

Reinforcement Learning (RL) is a powerful machine learning paradigm that enables an agent to learn optimal decision-making strategies through interaction with an environment to achieve some goal. The agent learns from the outcomes of its actions, rather than from being taught explicitly. This learning involves discovering which actions yield the most reward by trying them out and assessing the results. (Sutton & Barto, 2018)

Reinforcement Learning consists of several key components that work together to enable an agent to learn and make informed decisions. These components include:

- **Agent:** The decision-maker, which interacts with the environment.
- **Environment:** The world through which the agent moves, providing specific states to the agent.
- **State:** A configuration or condition that the environment can be in at any

given time.

- **Action:** All possible moves that the agent can make.
- **Reward:** Feedback from the environment in response to the actions taken by the agent. It’s a scalar value that the agent tries to maximize over time.
- **Policy:** A strategy that the agent employs to determine the next action based on the current state.
- **Value Function**: A prediction of future rewards the agent expects to receive, used to evaluate which states are most beneficial.

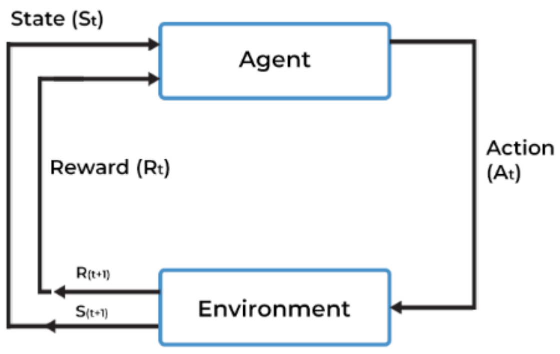


Figure 4. Reinforcement Learning Overview

The typical reinforcement learning pipeline is as follows: the agent receives the current state $S_t$ from the environment and acts based on its policy. After the action is taken, the environment transitions to a new state $S_{t+1}$ and emits a reward $R_t$, signaling the effectiveness of the action. This reward feeds into the agent's learning process, helping it refine its policy to maximize cumulative rewards over time. The loop represents the ongoing nature of this process, highlighting the sequential decision-making characteristic of RL, where each action influences future states and rewards, forming a continuous cycle of interaction and adaptation.

In this section, we will detail how we formulated the problem of choosing the optimal node feature weights to decide which vertex to branch on for parameterized vertex cover as a reinforcement learning problem discussing each component in the formulation.

### *1. State*

The state is a snapshot of the environment at a certain timestep. Part of this state will be used as input for the model to make inferences and output an action. In our formulation, the state consists of 2 components: the first are two branching simulators, there will be two branching simulators (the branching simulator is detailed in a previous section) one dedicated to our model and another dedicated to the opponent which we will be trying to outperform. This opponent will either be using high-degree branching or the best-performing model so far (more on that in the training section). The second component is the graph features for the current graph in the branching simulator of the model. These features capture the relevant properties of the graph at each step of the branch-and-reduce algorithm, such as the degree distribution, clustering coefficient, centrality measures, and other attributes. These features will be the input that the agent will see to make informed decisions and produce optimal weights for the node features.

### *2. Action*

The action taken by the agent at every step is a decision that this agent makes based on its policy. The policy is determined by the underlying model of the agent be it a neural network or something else. In our formulation, the action will be a set of weights with the same size as the node features.  These weights will determine how important each node feature is in the calculation of the final score. In particular, the final score of each vertex is calculated by taking the dot product between these weights and the node features of the vertex in question.

## *3. Reward*

The reward is a scalar value that indicates how good the action that the agent took was. This reward feeds into the learning algorithm of the agent to help train it to perform better. The reward system is crucial as it provides the feedback signal that the agent uses to learn from its interactions with the environment. The goal of the agent is to maximize the cumulative reward over time, which it does by adjusting its policy. In our formulation, we tested 4 different reward formulations as follows:

- **Versus Reward:** This is a competitive reward mechanism where the agent is rewarded or penalized based on its performance relative to an opponent (either a previous version of the agent or a heuristic that chooses vertices with high degrees). If the RL agent's selection leads to a better outcome compared to the opponent, it receives a positive reward (+1). Conversely, if the opponent performs better, the agent is penalized (-1). The agent beats the opponent in case the branching simulator of the agent produces a smaller final search tree than the opponent's branching simulator. This reward is adjusted by a factor that decreases over tree size, calculated as $\frac{Base\ Reward}{Tree\ Size}$, encouraging the agent to improve rapidly.
- **Reduction Reward:** The agent is rewarded based on the proportion of nodes it can remove from the graph at each step using the reduction rules. This is a direct incentive for reducing the graph's size using these rules. The reward is calculated as a ratio of the total reduction in nodes to the size of the graph,

multiplied by a factor to scale the reward appropriately as follows:

$\frac{Total\ Nodes\ Reduced}{Total\ Number\ of\ Nodes} * C$ where C is the scaling constant.

- **Leaf Reward:** The agent receives a reward for each leaf node found in the search tree, which encourages the discovery of terminal states where no further branching is necessary. This aligns with the goal of finding a vertex cover as efficiently as possible. The leaf reward is scaled based on the number of steps taken to reach the leaf, which incentivizes the agent to find leaves quickly and potentially favors solutions that lead to smaller search trees. The formula for this reward is: $\frac{Leaf\ Reward\ Constant}{Steps\ to\ reach\ Leaf}$.
- **Step Reward:** The agent receives a constant negative reward for each step it takes. This is a common technique to encourage efficiency; by penalizing the agent for each action, it learns to reach the goal in fewer steps, leading to solutions that require smaller search trees.

## *4. Training Algorithm: Proximal Policy Optimization*

To train our agent we need a reinforcement learning training algorithm that modifies the agent policy based on the rewards it receives from interacting with the environment. Proximal Policy Optimization (PPO) is a popular and effective reinforcement learning algorithm that has gained significant attention in recent years.(Schulman et al., 2017) PPO is an on-policy algorithm that belongs to the class of policy gradient methods. It aims to learn an optimal policy by directly optimizing the policy parameters to maximize the expected cumulative reward. The crux of PPO lies in updating policies incrementally to avoid large, destabilizing updates while maintaining enough flexibility to explore and adapt. PPO is made up of several components:

- **Policy Network:** The policy network in PPO is a neural network that directly defines the policy function $\pi(a|s,\theta)$, mapping states s to a probability distribution over actions a, with parameters $\theta$. This network is responsible for deciding which actions the agent should take given a particular state. During training, the policy network's parameters are updated to maximize expected rewards. The network's architecture can vary widely depending on the complexity of the state space and the required granularity of the action space.
- **Value Network:** The value network is a separate neural network that estimates the value of states, V(s). It helps in evaluating how good it is for the agent to be in a certain state and plays a crucial role in calculating the advantage function. Unlike the policy network, which outputs a probability distribution, the value network outputs a single scalar representing the expected sum of future rewards from the state.
- **Advantage Function:** The advantage function, denoted as A(s,a), measures the relative benefit of taking a specific action a in state s, compared to the average action at that state. It is often computed as $V(s_{t+1})+r_t-V(s_t)$, where V(s) is the value function representing the expected return of being in state s. The advantage function is pivotal for the policy gradient estimation as it tells the agent which actions are better or worse than the current policy's average.
- **Clipped Surrogate Objective:** PPO introduces a novel objective function that adds a clipping mechanism to the policy gradient estimator. This clipping limit the size of the policy update by modifying the objective to penalize changes that move the probability ratio $\frac{\pi(a|s,\theta)}{\pi(a|s,\theta old)}$ away from 1 by

more than a specified clipping range. This clipped objective function helps in maintaining the updates within a trust region, reducing the likelihood of making a harmful large update that could decrease performance.

PPO operates by collecting data from the current policy's interactions with the environment, estimating the expected returns, and calculating the advantage of the actions taken. The core component of PPO is the objective function it maximizes, which incorporates a novel clipping mechanism that prevents the new policy from deviating too far from the old one. This function allows for policy updates that are "proximal" (close) to the previous policy, hence the name.

The PPO algorithm iterates through the following steps:

1. **Collect Data:** For each iteration, the algorithm collects data in the form of $(S_t, Rt, S_{t+1}, A_t)$ where they represent the state, reward, next state, and action respectively. The agent acts according to the policy network.
2. **Calculate Advantage:** The advantage is calculated for all the actions collected according to the formula: $V(s_{t+1}) + R_t - V(s_t)$ where we use the value network to derive the V(s) values.
3. **Update the Policy Network:** we update the policy network by minimizing the following loss function:

$$L^{CLIP}(\theta) = E_t[\min(R_t.A_t , \text{clip}(R_t, 1- \epsilon, 1+ \epsilon). A_t)]$$

4. **Update the Value Network Parameters:** we update the value network by minimizing the mean squared error (MSE) between the predicted value V(s) and the actual cumulative reward from the s until the end of the episode: $(V(s) - R(s))^2$

5. **Repeat**: the cycle with the updated policy.

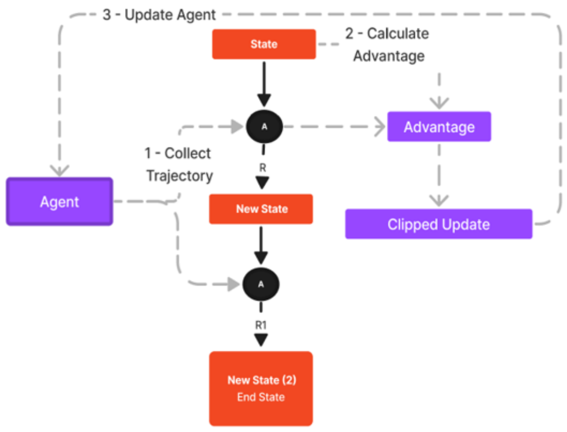
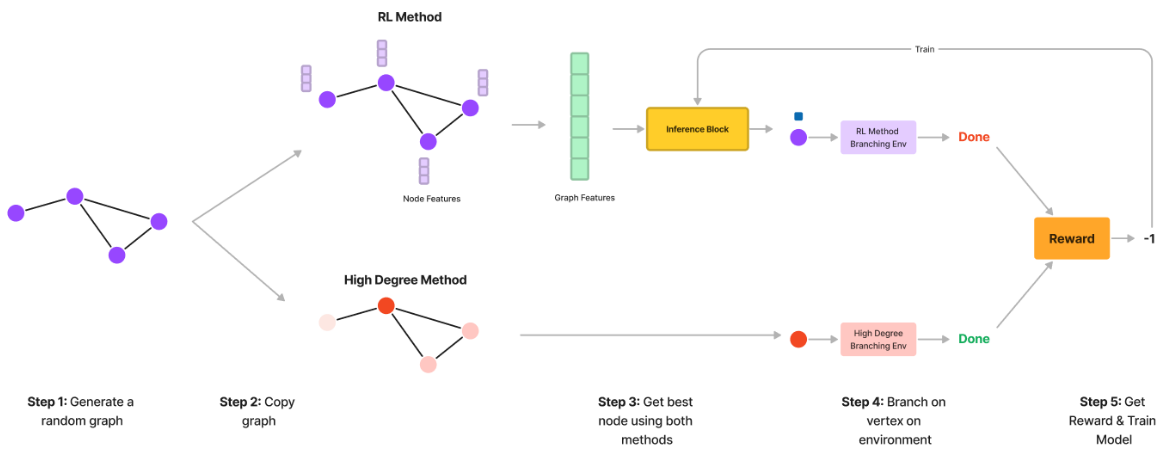


Figure 5. Proximal Policy Optimization Overview

## 5. *Training*

Training a reinforcement learning (RL) agent is an iterative process that fine-tunes the agent's policy so it can perform a task more effectively. By interacting with its environment, the agent learns to associate certain states with actions that maximize cumulative rewards. This learning phase is crucial for the agent to make intelligent, goal-oriented choices. In the context of our formulation, the agent learns to identify key



RL Method
Train
Node Features
Graph Features
Inference Block
RL Method Branching Env
Done
Reward
-1
High Degree Method
High Degree Branching Env
Done
Step 1: Generate a random graph
Step 2: Copy graph
Step 3: Get best node using both methods
Step 4: Branch on vertex on environment
Step 5: Get Reward & Train Model

Figure 6. Training Overview (Method 1)

features for the vertices to help guide the vertex selection process while branching. Training is done guided by a tailored reward system that reinforces beneficial actions.

A common approach to training an RL model is adversarial training, a technique that pits two opponents against each other in a competitive environment. This method is similar to the concept of iron sharpening iron; as one agent learns from the environment, its counterpart adapts to the changing strategies, creating a dynamic learning process that can lead to more robust and sophisticated policies. The adversarial nature of this training method helps prevent overfitting and encourages the discovery of strategies that are not immediately apparent. By continuously challenging the agents to outperform their adversaries, they explore and learn a broader range of strategies and, in turn, develop a deeper understanding of the environment and the task at hand. Adversarial training is particularly effective in complex environments where the optimal strategies are not known a priori. In our case we train our agent on two different opponents: First, we train the agent to beat high degree branching strategy to help build a solid base of the agent reasoning then we train the agent on the best previous version of itself helping it patch up any weaknesses it had before.

The pipeline for the training on the high degree opponent is as follows:

1. **Graph Generation:** The loop begins with the generation of a random graph. This graph represents the initial instance of the PVC problem that the RL agent needs to solve using branching.
2. **Graph Duplication:** The generated graph is duplicated to allow for parallel comparison between the RL method and an opponent. This ensures that both methods operate on identical instances of the problem.
3. **Best Node Identification:** For the RL method, node features are extracted,

and an inference model is used to select the best node for branching by calculating the score based on the inferred node feature weights from the model. In parallel, a high-degree heuristic method identifies the node with the highest degree and selects it.

4. **Branching Decision:** Each method's selected node is then used to branch in their respective branching simulators. The RL method uses its current policy to decide which vertex to branch on, while the high-degree method simply selects the vertex with the highest degree.
5. **Reward and Training:** The results of each branching decision are then evaluated. If the RL branching simulator indicates that the branching tree is finished before the high degree branching simulator then the model is given a positive reward. Conversely, if the high degree branching simulator finishes first the model receives a negative reward. This reward signal is used to train the model, adjusting the policy to improve future decision-making.

For the second level of training a similar pipeline is used where we just replace the high degree inference mechanism with a previous agent as follows:

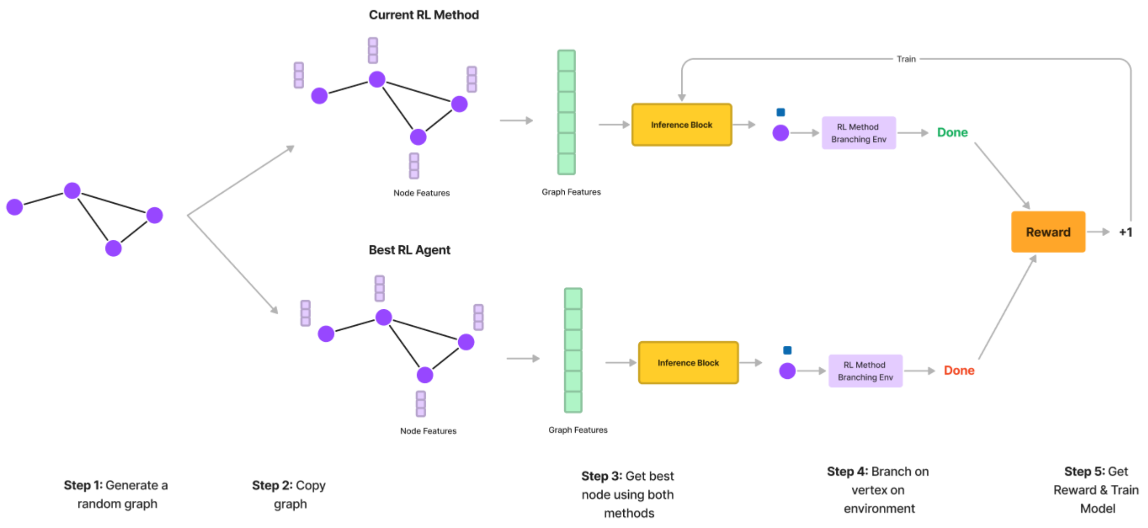


Figure 7. Training Overview (Method 2)

## H. Training Specifics

In this section we will detail some of the technical details behind the training of our model.

### *1. Graph Generation*

When training reinforcement learning models in the context of graph-related problems, it's vital to expose the model to a variety of graph structures. This diversity ensures that the learned policies are generalizable and robust to different kinds of network topologies. Here is an overview of graph generation methods used in our training:

1. **Gnm Random Graph:** This method creates a graph by randomly adding edges until a total of 'm' edges are reached. Each potential edge has an equal probability of being chosen in each step of the process.
2. **Random Density Graph:** Similar to the previous method, graphs are generated by continually adding edges until a specified density is achieved. Density, defined as the ratio of the number of edges to the number of possible edges, is used here as a target metric to dictate the stopping point for edge addition/
3. **Erdős-Rényi:** Named after the mathematicians Paul Erdős and Alfréd Rényi, this model creates graphs by including each possible edge with a fixed probability. This process results in networks where the degree distribution follows a binomial distribution.
4. **Barabási-Albert:** The Barabási-Albert model generates scale-free networks through a preferential attachment mechanism, where new nodes are more

likely to connect to existing nodes that already have a higher degree. This model mimics the power-law distribution observed in many real-world networks, including the Internet and social networks, where few nodes (hubs) accumulate a large number of connections.

5. **Watts-Strogatz:** Starting with a regular ring lattice, the Watts-Strogatz model introduces randomness by rewiring edges with a certain probability.

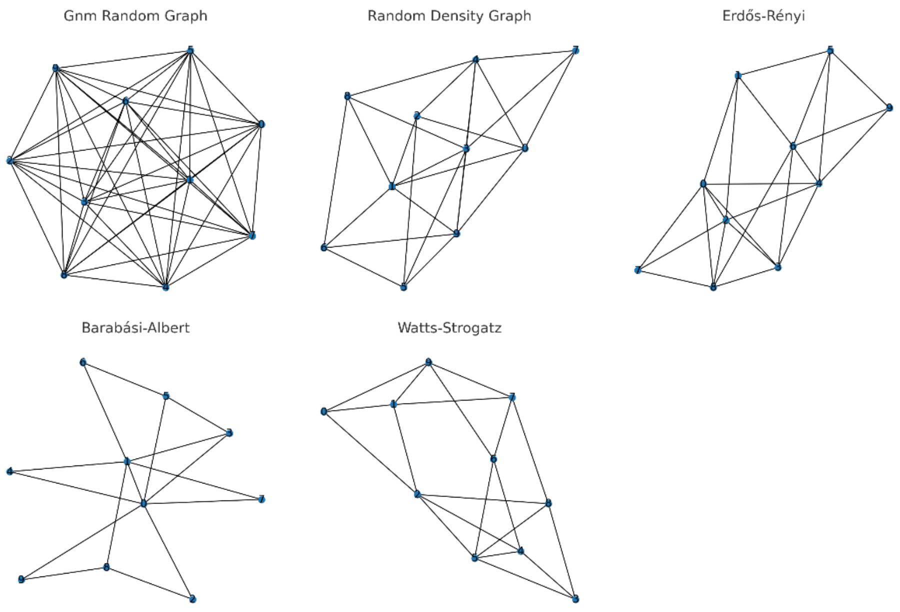


Figure 8. Graph Generation

This method is known for producing 'small-world' networks that exhibit high clustering like regular lattices and short average path lengths similar to random graphs.

## 2. *Libraries*

To build the code base for our project we used the programming language python. We also utilized several state-of-the-art libraries to aid in the development of the project. Notably:

- **OpenAI Gymnasium:** offers a standardized interface for a diverse suite of environments, enabling the development and benchmarking of reinforcement learning algorithms. It is instrumental for defining and managing the environment where the RL agent operates, providing a controlled and replicable setup for training and evaluating the agent's performance. We used OpenAI Gym to create the environment in which our model was trained on.

- **Stable Baselines 3**: is a set of reliable implementations of reinforcement learning algorithms in PyTorch. It facilitates the training process by providing a collection of pre-coded RL algorithms, including Proximal Policy Optimization (PPO), which can be easily deployed and customized. This library is crucial for streamlining the RL training process, reducing development time, and ensuring the agent's training is grounded in well-established methods. We used StableBaselines to train our agent using PPO.
- **NetworkX**: is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. With its comprehensive suite of graph generation and analysis tools, NetworkX is ideal for simulating the varied graph structures that the RL agent must learn to navigate. Its versatility in handling graphs makes it an indispensable tool for generating training and evaluation datasets and for implementing the graph-related functions required for problem-solving in the project. We used NetworkX to generate the graphs used in our training as well as for other graph related operations.

### ***3. Parameters***

In the training regimen for the reinforcement learning model, a specific set of parameters was methodically chosen to shape the learning environment and define the scope of the training process.

1. **The size of the graphs:** used during training ranged from 20 to 45 nodes, providing a spectrum of complexity while keeping computational demands within reasonable limits.
2. **The parameter K**, defining the upper bound of the vertex cover size, varied adaptively between half to an eighth of the number of nodes N/2 to N/8, ensuring a diversity of problem constraints.
3. **Timesteps:** The model underwent an extensive training phase comprising 500,000 timesteps, each timestep representing a complete sequence of interactions with the environment.

This comprehensive training setup was designed to equip the RL model with the experience needed to navigate the complexity of vertex covers across a broad spectrum of potential graph types.

## I. Limitations

There were many limitations faced during the training process most notably:

- **Variable Size Features:** In graph-based machine learning, handling variable-size features is a common challenge. Traditional machine learning models require fixed-size input, but graphs can vary widely in the number of nodes and edges. Techniques to handle this, like padding, can introduce noise or computational inefficiency. Graph Neural

Networks (GNNs) address this to some extent, but their integration in all frameworks is not yet standard.

- **Lack of Support for GNNs:** While GNNs are gaining traction for their ability to directly process graphs, their adoption is not yet universal across all machine learning frameworks and libraries especially in RL libraries. This limits the choice of tools and can hinder performance optimization.
- **Exponential Data Growth for Supervised methods:** Supervised learning methods can struggle with combinatorial problems like the vertex cover problem due to the exponential growth of the data. As the size of the graph increases, the number of potential solutions and, therefore, the amount of training data required can grow exponentially, making it computationally infeasible to generate and process.

# CHAPTER IV
# RESULTS & DISCUSSION

In this section, we present a comprehensive analysis of the results produced from all the experiments conducted in this dissertation. The main goal is to evaluate the efficacy of using reinforcement learning to optimize the vertex selection process in branch-and-reduce. We will first provide an overview of the experimental process detailing the various configurations of each experiment as well as the hardware used during training. We will also discuss the testing pipeline used during these experiments. We will then display the results of our experiments in a clear tabular format while providing a clear analysis of each result. Furthermore, we will provide a detailed discussion of the node feature importance analysis. We end this section by mentioning the limitations we encountered during our various experiments.

## A. Experimental Setup

We conduct our training and testing on an ASUS TUF A15 laptop equipped with an AMD Ryzen 7000 series 9, NVIDIA GTX 4050, and a 1000 GB SSD, the experiments aimed to refine an RL agent's proficiency in selecting optimal node feature weights for effective branching within a branch-and-reduce framework. Our experimental framework was designed to span a variety of graph sizes and complexities. The training was executed over 500,000 timesteps, with each session extending approximately 12 hours, ensuring thorough exposure to the problem space. The graphs for training were generated within the size range of 20 to 45 nodes, simulating a controlled yet challenging environment for the agent to learn. Testing, however, expanded this scope significantly, involving graphs ranging from 40 to 130 nodes, to evaluate the model's generalization capabilities and scalability. The parameter K, dictating the vertex cover's upper size limit varied between half to an eighth of the graph's node count (N/2 to  N/8), presenting a dynamic range of problem difficulties.

## B. Training Results

This section presents the results of the two different training methodologies applied to the reinforcement learning agent tasked with solving the parameterized vertex cover problem. The first approach referred to as Base Training, involves training the agent against a heuristic strategy that branches on vertices with the highest degree. The second approach, versus (VS.) Training, entails training the agent against the best-performing version of itself in a form of self-play, where the agent attempts to outperform its previous iterations.

The Base Training graph shows the evolution of average rewards over episodes, with rewards being grouped per 199 episodes and smoothed with a moving average (window=7). The graph illustrates a trend where the agent initially experiences a steeper learning curve, indicated by a gradual increase in rewards, followed by a period of fluctuation. This suggests that the agent is learning to improve its policy over time by exploring and exploiting the decision space defined by the vertex cover problem. However, the consistency of the agent's performance is subject to variability, possibly due to the exploratory nature of its policy in the complex environment.

Conversely, the VS. Training graph, which groups rewards per 351 episodes, exhibits a noticeably different trend. Here, the moving average indicates a steadier ascent towards higher rewards. This improvement pattern suggests that the agent benefits from the adversarial training approach, potentially due to the constant pressure of competing against a more capable opponent – itself. The self-play paradigm appears to provide the agent with more robust and challenging scenarios, fostering a more refined policy over time.

The effectiveness of reinforcement learning strategies for combinatorial optimization problems is influenced by the dichotomy in training approaches. Base Training, while foundational, may limit the agent's exposure to sophisticated strategies, as it relies on a static heuristic for branching decisions. In contrast, VS. Training encourages the agent to continuously adapt and overcome increasingly challenging benchmarks set by its prior achievements. This dynamic method seems to result in an agent that not only performs better but may also generalize more effectively across different instances of the problem.

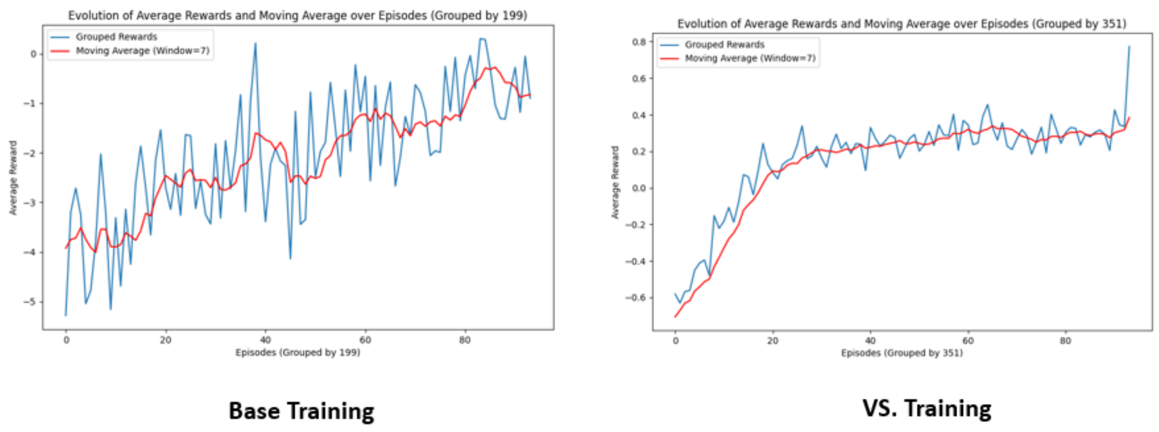
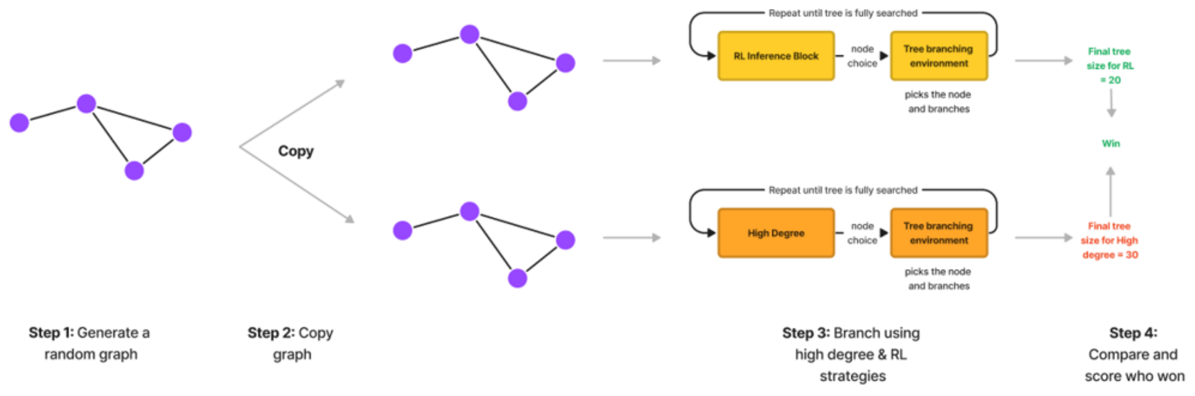


Figure 9. Training Results

The upward trends in the VS. Training graph compared to the oscillations in the Base Training graph underscore the potential of iterative self-improvement methodologies in machine learning. These results suggest that pitting the agent against itself in an ever-escalating series of challenges can lead to more consistent and higher-quality performance in complex tasks such as the vertex cover problem addressed in this work.

## C. Testing Pipeline

To test our approach, we compare the agent's performance against the high degree branching strategy. We compare the size of the resulting search tree from the agent and high degree strategy to see which produces a smaller search tree. Ideally, we want our agent to consistently produce smaller search trees than the high-degree branching strategy.



Copy
Repeat until tree is fully searched
RL Inference Block
node choice
Tree branching environment
picks the node and branches
Final tree size for RL = 20
Win
Repeat until tree is fully searched
High Degree
node choice
Tree branching environment
picks the node and branches
Final tree size for High degree = 30
Step 1: Generate a random graph
Step 2: Copy graph
Step 3: Branch using high degree & RL strategies
Step 4: Compare and score who won

Figure 10. Testing Pipeline Overview

The diagram details the testing pipeline used to evaluate the performance of the trained agent. These are the steps:

- First, we generate a random graph using one of the generation methods discussed in the methodology section.
- Second, we create to copies of the graph to test on high degree and on the agent.
- Third, we branch on the two graphs using two separate branching

simulators and following the two approaches: high degree and the agent’s policy.

- Finally, we compare the sizes of the search trees produced by both branching simulators to determine which one performed better.

## D. Main Model Results

Two primary models emerged from our experiments resulting from both training strategies we used. The first model (**Model Base)** was pitted against a heuristic strategy favoring vertices of high degrees. Over 500 iterations, the RL model demonstrated superior performance in 64% of the cases, tied in 10% of the instances. This result underscores the model's capacity to learn and adopt strategies that outperform basic heuristics.

The second model (**Model Plus)** was Model Base trained against the best version of itself, a scenario designed to assess incremental learning and self-improvement. The model outperformed the high degree heuristic in 88% of the trials, with only 1% ties, and a 6% average difference in tree size (with a standard deviation of ±5%), across 500 iterations. This not only attests to the model's effectiveness but also its ability to refine and enhance its policy over time.

Table 1. Main Model Results

| **Model Name** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| Model Base | 64% | 4% | 8% | 500 |
| Model Plus | 88% | 6% | 5% | 500 |

The performance of the reinforcement learning agent is evaluated using several metrics:

- **Wins Against High Degree:** This metric represents the percentage of times the learned branching strategy outperforms the high-degree heuristic, which selects the node with the highest degree for branching.
- **Average Tree Size Difference:** This metric measures the difference in the size of the search tree between the learned branching strategy and the high-degree heuristic. (A positive score indicates that the model produced a smaller tree size than high degree branching)
- **Size Standard Deviation:** This metric captures the variability in the size of the search tree explored by the learned branching strategy. A lower standard deviation suggests more consistent performance across different test instances.
- **# Test Iterations:** This represents the number of test instances used to evaluate the performance of the learned branching strategy for each graph generation method

These results provide compelling evidence of the model's capability to navigate the complexities of the vertex cover problem efficiently. Through rigorous training and testing phases, the RL agent demonstrated a significant understanding of strategic vertex selection, optimizing the balance between exploration and exploitation to achieve commendable performance metrics.

## E. Experiments

We conducted a variety of experiments to find the best training configurations to train our model. The experiments consisted of varying several different aspects and comparing the resulting models. The variables included: the training algorithm, reward scheme, graph generation methods, reduction rules applied, as well as the size of the graph and the parameter K.

## *1. Training Algorithm*

In this experiment we vary the training algorithm used to train the agent between three different choices:

- **Deep Deterministic Policy Gradient (DDPG):** is a model-free, online, off-policy reinforcement learning method. A DDPG agent is an actor-critic reinforcement learning agent that searches for an optimal policy that maximizes the expected cumulative long-term reward.
- **Twin Delayed Deep Deterministic policy gradient (TD3):** is a model-free, off-policy reinforcement learning method that builds upon the Deep Deterministic Policy Gradient (DDPG) by introducing twin Q-networks and delayed policy updates, aiming to reduce overestimation bias and improve learning stability.
- **Proximal Policy Optimization (PPO):** is a model-free, on-policy reinforcement learning algorithm that optimizes policy gradients by maintaining a balance between exploration and exploitation with a clipped objective function to ensure stable and efficient learning.

Table 2. Varying Training Algorithm

| **Training Algorithm** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| **PPO** | **70%** | 3% | 15% | 500 |
| DDPG | 62% | 2% | 10 | 500 |
| TD3 | 56% | 1% | 11% | 500 |

These results suggest that while PPO achieves the highest improvement over the heuristic strategy, its variability is higher, which might indicate occasional overfitting or exploratory decisions that do not always result in the most optimal tree size reduction. DDPG, while not outperforming PPO, offers a balance between performance and consistency. TD3's more conservative improvement could either point to a need for further tuning or inherent conservative decision-making in the algorithm's design.

### ***2. Rewards***

In this experiment, we vary the reward schema used in the training. We use the 4 rewards schemas found in the methodology section alone to test how each one performs. The rewards include:

- **Versus Reward:** Gains a point for outperforming an opponent's tree size, loses a point for underperforming.
- **Reduction Reward:** Earns points based on the percentage of nodes removed from the graph using the reduction rules
- **Leaf Reward:** Receives points for reaching leaf nodes quickly in the search tree.
- **Step Reward:** Incurs a penalty for each step taken to encourage more direct solutions.

Table 3. Varying Reward Experiment

| **Reward Applied** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| **Versus Reward** | **57%** | 6% | 8% | 500 |
| Reduction Reward | 46% | 4% | 9% | 500 |
| Leaf Reward | 24% | -6% | 9% | 500 |
| Step Reward | 19% | -8% | 10% | 500 |

The Versus Reward, where the RL model competes against a high-degree strategy, shows a promising 57% win rate, indicating that the RL model often outperforms the high-degree baseline with a notable average tree size reduction of 6%. However, the Reduction Reward seems less effective, with a lower 46% win rate and a smaller average tree size difference, suggesting that focusing solely on the reduction may not be as advantageous. The Leaf and Step Rewards appear to disincentivize the model, with win rates dropping to 24% and 19%, respectively, and the average tree size difference turning negative. This points to a potential over-penalization for steps taken, leading to suboptimal paths and decisions that increase the search tree size, as reflected by the size standard deviations which indicate a consistent variance across all reward types. These outcomes highlight the delicate balance required in reward system design to steer the agent toward the most efficient strategies.

### *3. Graph Generation*

In this experiment we vary the graph generation method used to train our model. We use only a single graph generation method to train the agent and test it on all the rest. The graph generation methods used are as follows:

- **Gnm Random Graph:** Generates graphs by adding edges randomly until a pre-defined total is met, ensuring each edge has an equal chance of selection.

- **Random Density Graph:** Produces graphs by adding edges to achieve a specific edge-to-possible-edge ratio, dictating graph density.
- **Erdős-Rényi:** Forms graphs by including each potential edge with a constant probability, leading to a binomial degree distribution.
- **Barabási-Albert:** Creates scale-free networks using preferential attachment, where nodes tend to connect to already well-connected hubs.
- **Watts-Strogatz:** Begins with a ring lattice and introduces randomness by rewiring edges, generating networks with high clustering and short path lengths.

Table 4. Varying Graph Generation Experiment

| **Graph Generation Method** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| Gnm Random Graph | 53% | -3% | 23% | 500 |
| Random Density Graph | 57% | 2% | 15% | 500 |
| Erdős-Rényi | 20% | -10% | 33% | 500 |
| Barabási-Albert | 33% | -11% | 20% | 500 |
| **Watts-Strogatz** | **62%** | **3%** | **10%** | **500** |

The results indicate that the agent performs best when trained on the Watts-Strogatz graph generation method, winning against the high-degree heuristic in 62% of the test instances and achieving a 3% reduction in the average search tree size. On the other hand, training on the Erdős-Rényi and Barabási-Albert methods leads to lower performance, with the agent winning in only 20% and 33% of the test instances, respectively.

These results suggest that the choice of graph generation method used for training the reinforcement learning agent has a significant impact on its performance and generalization ability. Training on graphs with certain structural properties, such as high clustering and short path lengths (Watts-Strogatz), may lead to more effective branching strategies that can generalize well to other graph types. However, training on graphs with more extreme degree distributions (Erdős-Rényi and Barabási-Albert) may result in less robust branching strategies.

### *4. Reduction Rules*

In this experiment we vary the reduction rules applied in training. We test all possible combinations of applying the reduction rules to see which ones had the biggest impact on the algorithm. The 4 reduction rules are as follows:

- **Removing Vertices of Degree 0:** Eliminates isolated vertices as they don't contribute to covering any edges.
- **Removing the Neighbor of Vertices of Degree 1:** Includes the single neighbor of pendant vertices in the cover, then removes both, optimizing the cover by leveraging vertices that cover more edges.
- **Degree 2 Neighbors Rule:** If two degree 2 vertices are neighbors, both are added to the cover and removed from the graph, ensuring the covered edge between them doesn't necessitate additional vertices.
- **Removing Vertices of Degree Greater than K:** Includes vertices with connections exceeding the cover size limit (k) directly into the cover, as excluding them would lead to an infeasible solution.

We denote the reductions used in the training as a binary vector as follows:

[degree 0 rule, degree 1 rule, degree 2 rule, degree k rule]

[1,0,0,0] indicates only degree 0 rule was applied during training.

Table 5  Varying Reduction Rules Experiment

| **Reduction Rules** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| [1,1,1,1] | **65%** | 6% | 4% | 500 |
| [1,0,0,0] | 49% | 4% | 8% | 500 |
| [1,1,0,0] | 7% | -7% | 20% | 500 |
| [1,0,1,0] | 19% | -3% | 15% | 500 |
| [1,1,1,0] | 5% | -10% | 13% | 500 |
| [1,0,0,1] | 10% | -8% | 15% | 500 |
| [1,1,0,1] | 4% | -8% | 20% | 500 |
| [1,0,1,1] | 9% | -5% | 17% | 500 |

We applied the reduction rules during training but when we ran the testing pipeline we applied all of them. This shows that the model learned to integrate the reductions rules into its strategy which explains the relatively low results across the rows.

### *5. Varying K*

In this experiment we vary the parameter in PVC while we train to see the impact it has on the final model.

Table 6.  Varying K Experiment

| **K Range** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| N/6 – N/10 | 46% | -1% | 6% | 500 |
| **N/4 – N/8** | **57%** | 4% | 8% | 500 |
| N/2 – N/6 | 56% | 6% | 5% | 500 |

The experiments show no big impact for this change but it is noticeable that the

N/6 – N/10 range performs slightly worse than the other ranges. This can be attributed to the problem being harder and unsolvable as the K would have decreased significantly.

### *6. Varying N*

In this experiment we vary the N or the number of nodes in the graph during training.

Table 7. Varying Graph Size Experiment

| **Graph Size Range** | **Wins Against High Degree** | **Average tree size difference** | **Size standard deviation** | **# Test Iterations** |
|---|---|---|---|---|
| 40 – 70 | 45% | 2% | 7% | 500 |
| 20 – 45 | 57% | 4% | 6% | 500 |
| **15 – 25** | **60%** | 6% | 6% | 500 |

The results show that graph sizes do not make a very big difference as well. The drop in the higher range can be attributed to the higher complexity of the problem and the insufficient training time since we applied the same amount of training across all 3 cases which makes sense since harder problems will take more time to converge.

## F. Node Feature Analysis

In this section, we conduct a node feature analysis to add explainability to the trained model's results. We generate different graphs and for each graph feature, we plot the relationship between that graph feature and the weight predictions of a model for various node-level features. Each scatter plot corresponds to a different node feature. We include the full plot for 5 graph features: Density, Average Clustering, Average Betweenness Centrality, Normalized K, and Average Neighbor Degree. Each plot will provide us insight how the decision for each node feature weight is affected by the variation of each graph feature.

## 1. *Density*

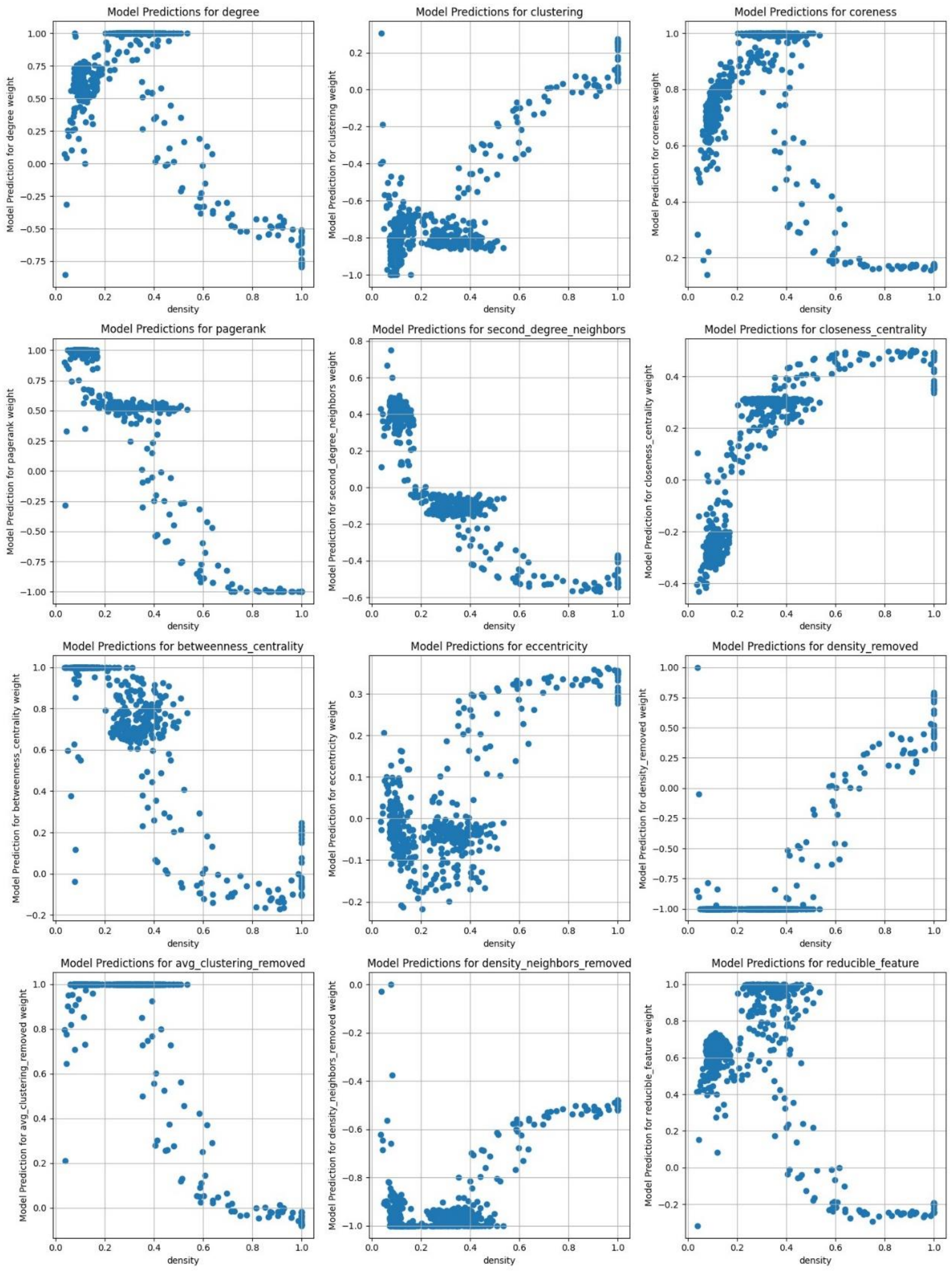


Figure 11. Density Feature Analysis

As density increases, certain node features may gain prominence in the model's predictive framework up to a saturation point. This can happen with features like node degree, where initially, a denser graph means more connections per node, increasing the feature's importance. However, once a graph becomes sufficiently dense, adding more edges doesn't significantly change the model's perception of a node's degree since many nodes are already highly connected. This is interesting to see as it would suggest that picking the highest degree is non-optimal and could even be a bad choice in some cases which would suggest that the high degree branching heuristic can be outdone by a trained agent like our model.

For other node features, increased density might actually diminish their importance. For example, in a very dense graph, individual node centrality might become less distinctive since many nodes are central, diminishing the uniqueness of the feature across the graph.

In some cases, the relationship is not straightforward, suggesting that the model might interpret these features differently at various stages of graph density. This can lead to profiles where features are valued by the model only within specific density ranges, potentially reflecting structural changes that occur in graphs as they grow denser.

## *2. Average Clustering*

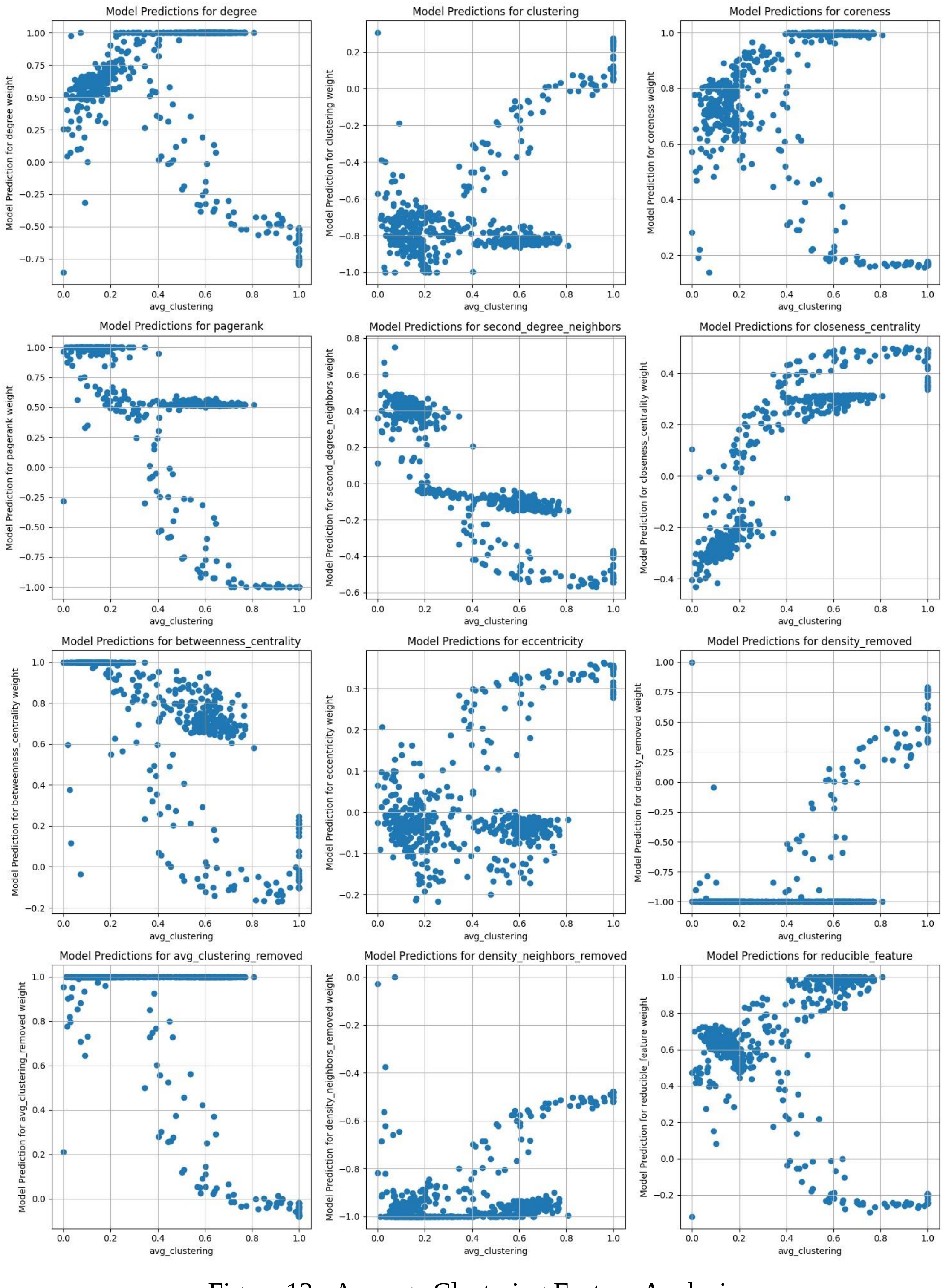


Figure 12. Average Clustering Feature Analysis

As the average clustering coefficient of a graph increases, the graph becomes more densely connected within local neighborhoods. In other words, nodes tend to form tightly-knit groups characterized by a higher number of triangles, where nodes are more likely to be connected to their neighbors' neighbors. This increased local connectivity leads to the formation of distinct clusters or communities within the graph, resulting in a more structured and organized network topology.

In a graph where average clustering is on the rise, the node feature degree may initially increase in importance, as the model recognizes the growing number of connections per node in emerging cliques. However, beyond a certain level of clustering, each additional connection may become less informative due to the high prevalence of connections overall, which could explain a plateau or even a decrease in the importance assigned to the degree.

Interestingly, for betweenness centrality, there is a marked decrease in weight as clustering increases, which implies that nodes that were once critical for connecting different parts of the graph become less so as more triangles form, providing alternative paths. This diminishing importance suggests that the model perceives such nodes to be less critical in facilitating communication or flow within the graph as clustering tightens.

For features related to reduction like reducible features, there's an uptrend in weight with increased clustering, possibly highlighting the model's strategy to focus on nodes that, if removed, would lead to a significant reduction in the overall graph size or help simplify the graph's structure efficiently. This trend suggests that the model's strategy evolves to prioritize the potential impact of a node's removal in highly interconnected networks.

## *3. Normalized K*

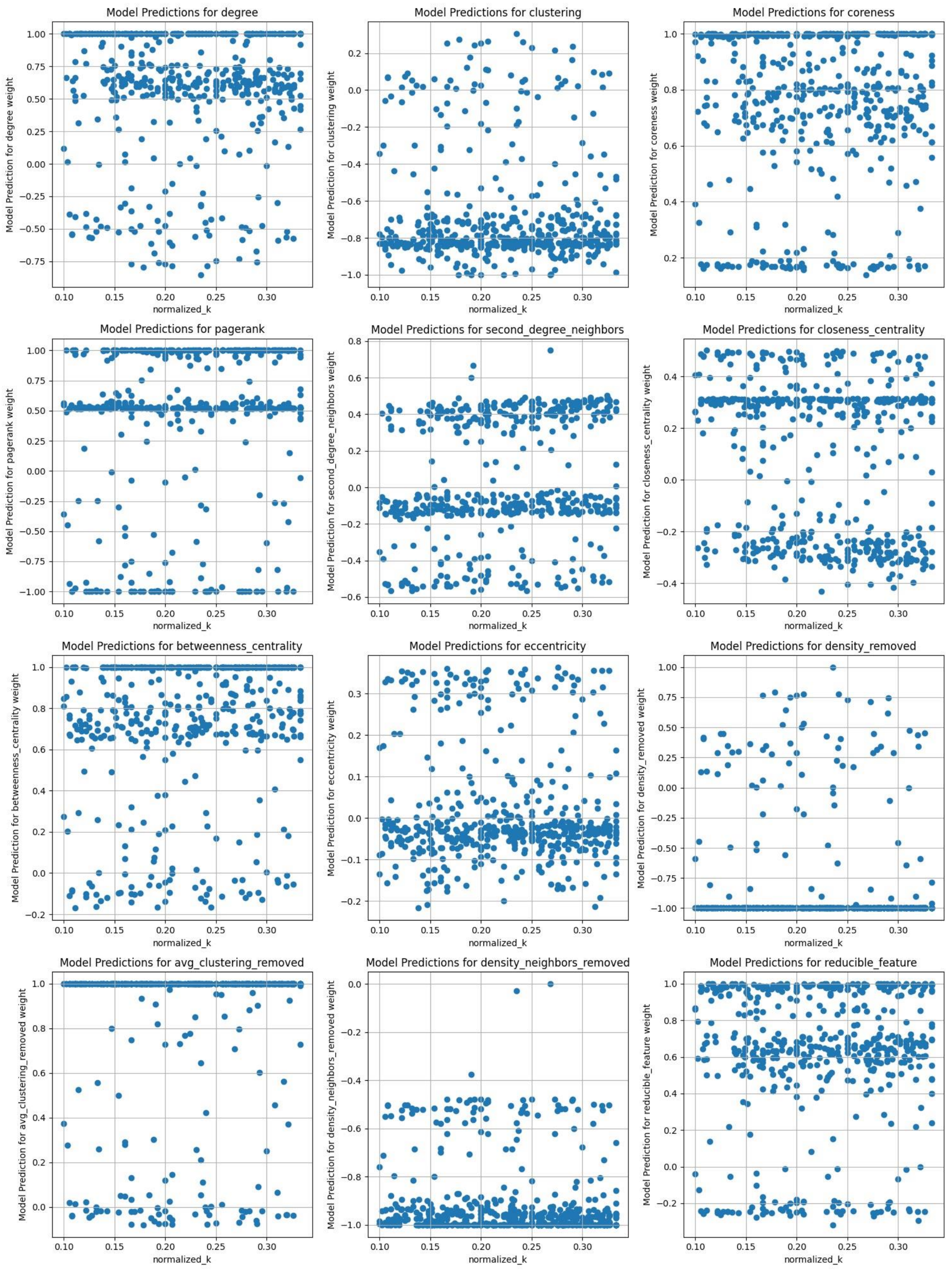


Figure 13. Normalized K Feature Analysis

In the context of a parameterized vertex cover problem, as the parameter K (which determines the maximum size of the vertex cover) increases, it affects both the complexity of the problem and the strategy for finding a solution. The vertex cover problem asks to find the smallest set of vertices such that every edge of the graph is incident to at least one vertex in the set. Increasing the parameter K means we are allowing for larger vertex covers, which may increase the solution space and potentially simplify finding a cover, but it could also lead to less efficient solutions since we're not constrained to find the smallest set.

As K increases, the weight assigned to the degree feature remains relatively high and stable, suggesting that the degree of a node (the number of connections it has) is consistently important in determining its likelihood of being part of the vertex cover, regardless of the allowed size of the cover. The weight for clustering decreases as k increases, which could imply that the tight-knit clusters of nodes become less critical to include in the vertex cover when we have more flexibility in the cover's size.

In contrast, the coreness feature weight remains high, highlighting the significance of core nodes that are part of the main structure of the graph rather than peripheral nodes. The consistent high weight for closeness centrality suggests that nodes central to the graph remain vital to include in the vertex cover because they can cover many edges due to their short distances to all other nodes in the graph.

## *4. Average Neighbor Degree*

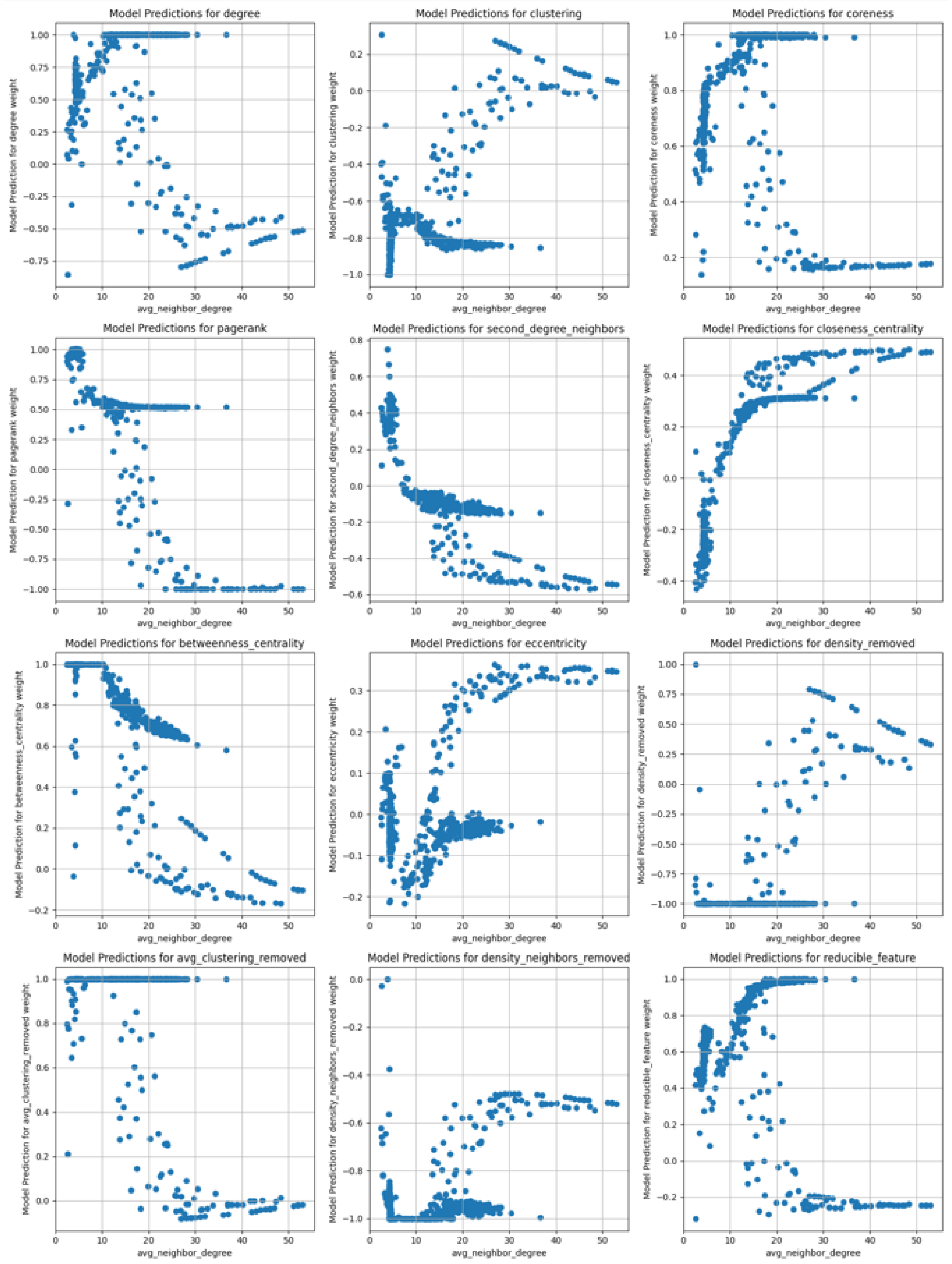


Figure 14. Average Neighbor Feature Analysis

As the average neighbor degree increases, reflecting the graph’s trend toward connectivity, the model’s response varies distinctively across different features. The weight predictions for degree initially decrease as the average neighbor degree increases, which suggests that in the context of nodes with highly connected neighbors, the model devalues the direct degree of a node, possibly due to the sufficiency of indirect connections for network robustness. This trend reverses at the higher end of neighbor degree averages, indicating an adaptation to the increased importance of direct connections in maintaining network integrity.

For closeness centrality, the weight predictions rise steadily with the average neighbor degree, aligning with the concept that as nodes are more closely linked, the centrality of nodes becomes increasingly relevant for efficient information dissemination. Interestingly, the predictions for betweenness centrality show a marked drop and then level off, which may imply that the nodes’ role as intermediaries becomes less critical as the average connectivity of their neighbors grows.

In the reducible feature, there's a sharp increase in weight as the neighbor degree increases. This might reflect a strategy where the model gives priority to reducing the graph complexity when nodes are highly connected, perhaps aiming to streamline the network's structure to more efficiently solve the vertex cover problem.

## *5. Average Betweenness Centrality*

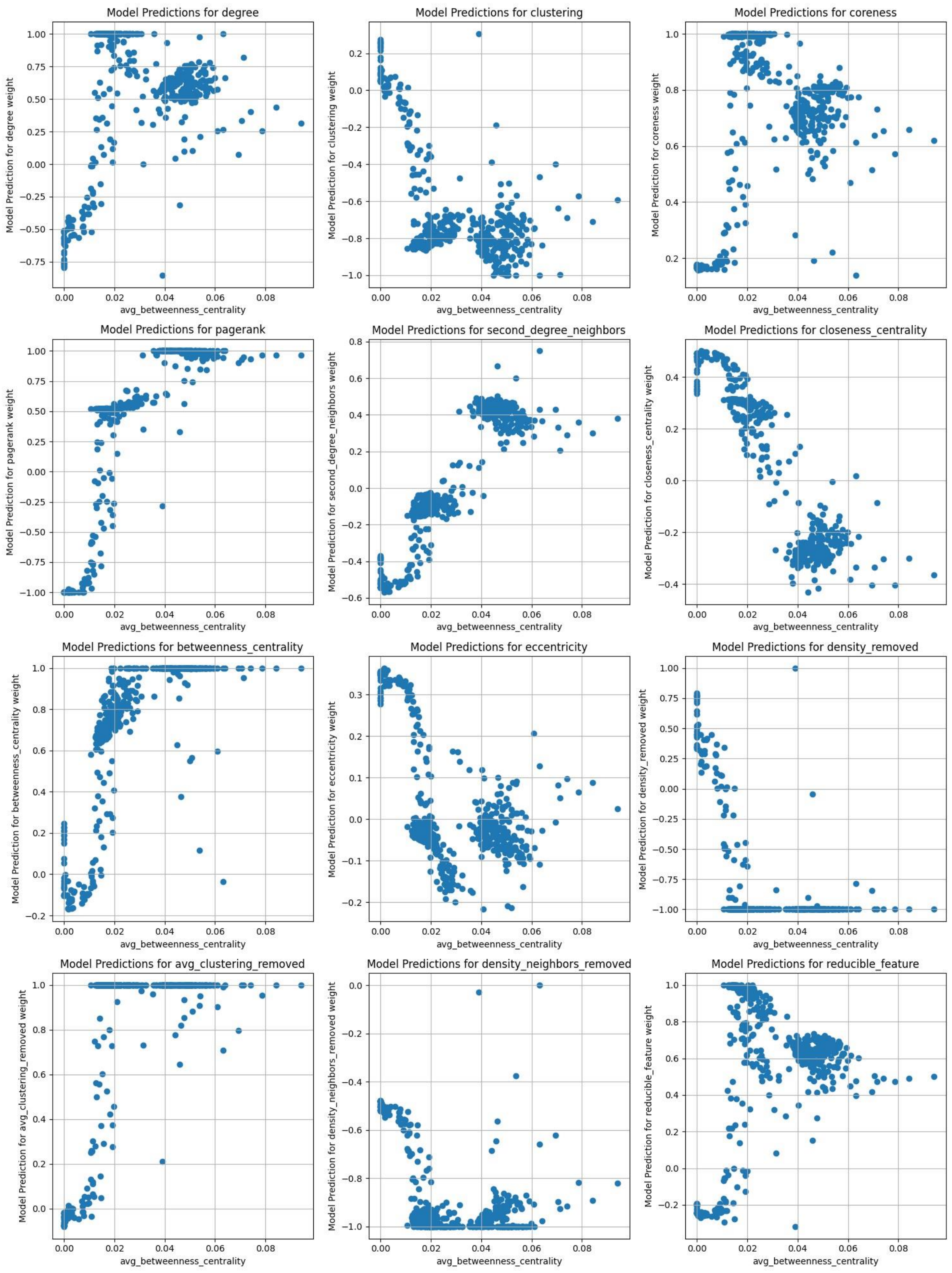


Figure 15. Average Betweenness Centrality Feature Analysis

Betweenness centrality is a measure of the number of times a node lies on the shortest path between other nodes. When the average betweenness centrality increases, it suggests that nodes are increasingly functioning as bridges within the network. This typically means there are more bottleneck points through which information or flow passes, indicating a less clustered and potentially more hierarchical or sparsely connected graph structure.

As average betweenness centrality increases, a decrease in the weight for closeness centrality might suggest that nodes central in terms of betweenness are not necessarily the quickest to reach all other nodes, indicating a more broker-like position rather than a hub. A decrease in clustering weight could imply that nodes that frequently act as bridges in the network (high betweenness centrality) are part of less clustered or tightly-knit groups, perhaps because they connect disparate parts of the graph.

The increase followed by a plateau and subsequent decrease in the weight assigned to degree might indicate that nodes with an initially higher degree benefit from their many connections but only up to a point. After this peak, the importance of having additional connections diminishes, possibly because the role of a node as a critical bridge is sufficiently established, and further connections may not significantly enhance that role.

Lastly, the decrease in weight for eccentricity as betweenness centrality increases could indicate that nodes that act as important connectors tend to be closer to the center of the graph (lower eccentricity). This makes sense as such nodes are strategically positioned to influence the flow of information or connectivity within the network, making them less peripheral (hence lower eccentricity) and more central in terms of control over network flow

# CHAPTER V

# CONCLUSION AND FUTURE WORK

In this dissertation, we explored a novel approach to optimizing the branch-and-reduce algorithm for solving the parameterized vertex cover problem using reinforcement learning. Our goal was to develop an intelligent agent capable of learning effective branching strategies by selecting the most promising vertex to branch on at each step based on the graph's structural features.

We formulated the problem as a Markov Decision Process and employed the Proximal Policy Optimization (PPO) algorithm to train our reinforcement learning agent. The state representation consisted of graph-level features capturing the overall structure and properties of the graph at each step. The agent's action space corresponded to predicting weights for node-level features, which were then used to calculate scores for each vertex, guiding the branching decision.

Through extensive experiments and ablation studies, we evaluated the performance of our trained agent against the traditional high-degree branching heuristic. Our best-performing model, trained using adversarial play against itself and against high degree branching with a carefully designed reward functions, consistently outperformed the high-degree strategy, achieving an 88% win rate and an average tree size reduction of 6%.

We also conducted a thorough analysis of the learned node feature weights, providing insights into the agent's decision-making process. The results showed that the agent adapted its strategy based on the graph's structural properties, assigning varying importance to different node features depending on the graph's density, clustering coefficient, and other characteristics.

Our work demonstrates the potential of reinforcement learning in tackling combinatorial optimization problems like parameterized vertex cover. By learning to make intelligent branching decisions based on the graph's structure, our approach offers a promising avenue for improving the efficiency of branch-and-reduce algorithms.

However, our work also has some limitations that open up opportunities for future research. One limitation is the scalability of our approach to very large graphs, as the computational complexity of extracting graph-level features and performing inference on the policy network can become prohibitive. Future work could explore more efficient graph representation techniques, such as graph embeddings or sampling strategies, to enable the application of our approach to larger-scale problems.

Another direction for future research is the integration of graph neural networks (GNNs) into our reinforcement learning framework. GNNs have shown great promise in learning powerful graph representations and capturing complex structural patterns. By incorporating GNNs into our state representation or action selection mechanism, we could potentially enhance the agent's ability to make more informed branching decisions based on the graph's local and global structure.

Furthermore, our approach could be extended to other combinatorial optimization problems on graphs, such as the maximum clique problem or the minimum dominating set problem. Adapting our reinforcement learning framework to these problems would require defining appropriate state representations, action spaces, and reward functions tailored to the specific problem constraints and objectives.

In conclusion, this dissertation presents a novel reinforcement learning approach to optimizing the branch-and-reduce algorithm for parameterized vertex cover. Our results demonstrate the effectiveness of learning intelligent branching strategies based on the graph's structural features, paving the way for further research at the intersection of reinforcement learning and combinatorial optimization on graphs.

# REFERENCES

AlonNoga, YusterRaphael, & ZwickUri. (1995). Color-coding. *Journal of the ACM (JACM)*, *42*(4), 844–856. https://doi.org/10.1145/210332.210337

Alvarez, A. M., Louveaux, Q., & Wehenkel, L. (2017). A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, *29*(1), 185–195. https://doi.org/10.1287/ijoc.2016.0723

Bagattini, F., Cappanera, P., & Schoen, F. (2018). Lagrangean-based combinatorial optimization for large-scale S3VMs. *IEEE Transactions on Neural Networks and Learning Systems*, *29*(9), 4426–4435. https://doi.org/10.1109/TNNLS.2017.2766704

Bannach, M., Stockhusen, C., & Tantau, T. (2015). Fast Parallel Fixed-Parameter Algorithms via Color Coding. *Leibniz International Proceedings in Informatics, LIPIcs*, *43*, 224–235. https://doi.org/10.4230/LIPIcs.IPEC.2015.224

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural Combinatorial Optimization with Reinforcement Learning. *5th International Conference on Learning Representations, ICLR 2017 - Workshop Track Proceedings*. https://arxiv.org/abs/1611.09940v3

Bengio, Y., Lodi, A., & Prouvost, A. (2021). Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, *290*(2), 405–421. https://doi.org/10.1016/J.EJOR.2020.07.063

Buss, J. F., & Goldsmith, J. (2006). Nondeterminism within $P^ *  $. *Https://Doi.Org/10.1137/0222038*, *22*(3), 560–572. https://doi.org/10.1137/0222038

Chang, M. S., Chen, L. H., Hung, L. J., Rossmanith, P., & Su, P. C. (2016). Fixed-parameter algorithms for vertex cover P3. *Discrete Optimization*, *19*, 12–22. https://doi.org/10.1016/j.disopt.2015.11.003

Chekuri, C. (2009). *CS 598CSC: Approximation Algorithms*.

Chen, J., Kanj, I. A., & Xia, G. (2010). Improved upper bounds for vertex cover. *Theoretical Computer Science*, *411*(40–42), 3736–3756. https://doi.org/10.1016/J.TCS.2010.06.026

Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., & Saurabh, S. (2015). *Parameterized Algorithms*. https://doi.org/10.1007/978-3-319-21275-3

Dai, H., Khalil †⇤, E. B., Zhang, Y., Dilkina, B., & Song, L. (2017). Learning Combinatorial Optimization Algorithms over Graphs. *Advances in Neural Information Processing Systems*, *30*.

Downey, R. G., & Fellows, M. R. (2013). *Fundamentals of Parameterized Complexity*. https://doi.org/10.1007/978-1-4471-5559-1

Erickson, J. (2010). *Algorithms Lecture 21: NP-Hard Problems*. http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/

Fomin, F. V., Gaspers, S., Kratsch, D., Liedloff, M., & Saurabh, S. (2010). Iterative compression and exact algorithms. *Theoretical Computer Science*, *411*(7–9), 1045–1053. https://doi.org/10.1016/J.TCS.2009.11.012

Fomin, F. V., Lokshtanov, D., Saurabh, S., & Zehavi, M. (2019). Kernelization: Theory of parameterized preprocessing. *Kernelization: Theory of Parameterized Preprocessing*, 1–516. https://doi.org/10.1017/9781107415157

Gaspers, S., & Liedloff, M. (2006). A branch-and-reduce algorithm for finding a minimum independent dominating set in graphs. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4271 LNCS*, 78–89. https://doi.org/10.1007/11917496_8/COVER

Gasse Mila, M., Montréal, P., Ferroni, N., Charlin Mila, L., Montréal, H., & Lodi Mila, A. (2019). *Exact Combinatorial Optimization with Graph Convolutional Neural Networks*. https://github.com/ds4dm/learn2branch.

Graph Theory in Chemistry: A Brief Review. (2022). *Journal of Scientific Enquiry*, *2*(1). https://doi.org/10.54280/JSE.222103

Gross, J. L., Yellen, J., & Anderson, M. (2018). Graph Theory and Its Applications. *Graph Theory and Its Applications*. https://doi.org/10.1201/9780429425134

Gu, S., & Yu, S. (2014). A chaotic neural network for the maximum clique problem. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3060*, 391–405. https://doi.org/10.1007/978-3-540-24840-8_28/COVER

Gupta, P., Gasse Mila, M., Montréal, P., Khalil, E. B., Pawan Kumar, M., Lodi, A., & Bengio, Y. (2020). *Hybrid Models for Learning to Branch*.

Gurobi. (2024). *Gurobi Optimization*. https://www.gurobi.com/

Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. *Advances in Neural Information Processing*

*Systems*, *2017-December*, 1025–1035. https://arxiv.org/abs/1706.02216v4

Harris, D. G., & Narayanaswamy, N. S. (2022). *A faster algorithm for Vertex Cover parameterized by solution size*. http://arxiv.org/abs/2205.08022

Hespe, D., Schulz, C., & Strash, D. (2019). Scalable Kernelization for Maximum Independent Sets. *Journal of Experimental Algorithmics (JEA)*, *24*(1). https://doi.org/10.1145/3355502

IBM. (2024). *IBM ILOG CPLEX Optimization Studio*. https://www.ibm.com/products/ilog-cplex-optimization-studio

Karp, R. (1972). *Reducibility among Combinatorial Problems*.

Khalil, E. B., Le Bodic, P., Song, L., Nemhauser, G., & Dilkina, B. (2016). *Learning to Branch in Mixed Integer Programming*. www.aaai.org

Kipf, T. N., & Welling, M. (2016). Semi-Supervised Classification with Graph Convolutional Networks. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*. https://arxiv.org/abs/1609.02907v4

Lancia, G., & Serafini, P. (2018). Integer Linear Programming. *EURO Advanced Tutorials on Operational Research*, 43–66. https://doi.org/10.1007/978-3-319-63976-5_4

Li, K., Zhang, T., Wang, R. W. Y., & Han, Y. (2021). Deep Reinforcement Learning for Combinatorial Optimization: Covering Salesman Problems. *IEEE Transactions on Cybernetics*, *52*(12), 13142–13155. https://doi.org/10.1109/TCYB.2021.3103811

Li, Z., Chen, Q., & Koltun, V. (2018). Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. *Advances in Neural Information Processing Systems*, *2018-December*, 539–548. https://arxiv.org/abs/1810.10659v1

Lin, J., Zhu, J., Wang, H., & Zhang, T. (2022). Learning to branch with Tree-aware Branching Transformers. *Knowledge-Based Systems*, *252*. https://doi.org/10.1016/j.knosys.2022.109455

Lokshtanov, D., Misra, N., & Saurabh, S. (2012). Kernelization - Preprocessing with a guarantee. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7370*, 129–161. https://doi.org/10.1007/978-3-642-30891-8_10/COVER

Lokshtanov, D., Narayanaswamy, N. S., Raman, V., Ramanujan, M. S., & Saurabh, S. (n.d.). *Faster Parameterized Algorithms using Linear*

*Programming* ⇤.

Mazyavkina, N., Sviridov, S., Ivanov, S., & Burnaev, E. (2020). Reinforcement Learning for Combinatorial Optimization: A Survey. *Computers and Operations Research*, *134*. https://doi.org/10.1016/j.cor.2021.105400

Nasteski, V. (2017). *An overview of the supervised machine learning methods*. https://doi.org/10.20544/HORIZONS.B.04.1.17.P05

Niedermeier, R. (2006). VERTEX COVER—AN ILLUSTRATIVE EXAMPLE. *Invitation to Fixed-Parameter Algorithms*, 31–40. https://doi.org/10.1093/ACPROF:OSO/9780198566076.003.0004

Niedermeier, R., & Rossmanith, P. (2003). On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms*, *47*(2), 63–77. https://doi.org/10.1016/S0196-6774(03)00005-1

Polydoros, A. S., & Nalpantidis, L. (2017). Survey of Model-Based Reinforcement Learning: Applications on Robotics. *Journal of Intelligent and Robotic Systems: Theory and Applications*, *86*(2), 153–173. https://doi.org/10.1007/S10846-017-0468-Y/METRICS

Riaz, F., & Ali, K. M. (2011). Applications of graph theory in computer science. *Proceedings - 3rd International Conference on Computational Intelligence, Communication Systems and Networks, CICSyN 2011*, 142–145. https://doi.org/10.1109/CICSYN.2011.40

Ryoo, H. S., & Sahinidis, N. V. (1996). A branch-and-reduce approach to global optimization. *Journal of Global Optimization 1996 8:2*, *8*(2), 107–138. https://doi.org/10.1007/BF00138689

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. http://arxiv.org/abs/1707.06347

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning : an introduction*.

Thomassé, S. (2009). *Kernelization via Combinatorial Optimization*. *Vol. 38*. https://hal-lirmm.ccsd.cnrs.fr/lirmm-00394592

Toscano, L., Stella, S., & Milotti, E. (2015). Using graph theory for automated electric circuit solving. *European Journal of Physics*, *36*(3), 035015. https://doi.org/10.1088/0143-0807/36/3/035015

Trevisan, L. (2011). *Combinatorial Optimization: Exact and Approximate*

*Algorithms*.

Tyagi, R., & Batra, M. (2016). Implementation and Comparison of Vertex Cover Problem using Various Techniques. *International Journal of Computer Applications*, *144*(10), 975–8887.

Veličković, P., Casanova, A., Liò, P., Cucurull, G., Romero, A., & Bengio, Y. (2017). Graph Attention Networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*. https://doi.org/10.1007/978-3-031-01587-8_7

Wang, L., Hu, S., Li, M., & Zhou, J. (2019). An exact algorithm for minimum vertex cover problem. *Mathematics, 7*(7). https://doi.org/10.3390/math7070603

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(1), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

Yamout, P., Barada, K., Jaljuli, A., Mouawad, A. E., & El Hajj, I. (2022). Parallel Vertex Cover Algorithms on GPUs. *Proceedings - 2022 IEEE 36th International Parallel and Distributed Processing Symposium, IPDPS 2022*, 201–211. https://doi.org/10.1109/IPDPS53621.2022.00028