

A Comparative Analysis of Algorithms for the Vertex Cover Problem

Adiel Rum - 10123004

Program Studi Matematika

Fakultas Matematika dan Ilmu Pengetahuan Alam

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: 10123004@mahasiswa.itb.ac.id

Abstract—The Vertex Cover problem is a fundamental NP-hard challenge in graph theory with significant practical applications. This paper presents a comparative study of three distinct algorithms for solving this problem: an exhaustive Brute-Force search, a high-degree Greedy heuristic, and a guaranteed 2-Approximation algorithm. Through implementation and testing on various graph structures, this work empirically analyzes the inherent trade-off between solution optimality and computational efficiency. The results demonstrate that while the Brute-Force method guarantees the minimum vertex cover, its exponential time complexity makes it infeasible for non-trivial graphs. Conversely, the Greedy and 2-Approximation algorithms offer polynomial-time solutions but with different performance characteristics. The Greedy algorithm often finds optimal or near-optimal solutions quickly but lacks a reliable performance guarantee, whereas the 2-Approximation algorithm, while sometimes less optimal, consistently provides a solution size provably within twice the minimum. This investigation concludes that the selection of an appropriate algorithm is not absolute but is contingent on the specific application's requirements, balancing the need for computational speed against the tolerance for sub-optimality.

Keywords—Vertex Cover; Graphs; NP-Complete; Brute-Force; Greedy; 2-Approximation

I. INTRODUCTION

In the fields of computer science and discrete mathematics, graph theory stands as a cornerstone for modeling and analyzing complex networks and relationships. A graph, in its simplest form, is a collection of vertices (or nodes) connected by edges, representing a vast array of real-world scenarios, from social networks and transportation systems to computer networks and molecular biology. Within this domain lies a fundamental optimization problem known as the Vertex Cover problem. A vertex cover is defined as a subset of a graph's vertices such that every edge in the graph is incident to at least one vertex within this subset. This concept is not merely a theoretical curiosity; it has profound practical applications, such as placing the minimum number of security cameras to cover all hallways in a building or identifying a minimal set of key individuals in a network to monitor for information dissemination.

The primary challenge, however, is not just finding *any* vertex cover, but finding a *minimum vertex cover*, that is, a vertex cover with the smallest possible number of vertices.

Achieving this optimization is critical for efficiency and resource conservation. The difficulty of this task is formally captured by its classification as an NP-hard problem. This means that as the size of the graph grows, the time required to find the guaranteed optimal solution using any known method increases exponentially. For large, real-world graphs, finding the minimum vertex cover through exhaustive search becomes computationally infeasible, pushing the limits of even the most powerful computers.

This inherent complexity necessitates a trade-off between optimality and efficiency, leading to the development of various algorithmic strategies. This project, "Explorations of algorithms to find the vertex cover of a graph," delves into this challenge by implementing and visualizing a spectrum of these algorithms. We will explore three distinct approaches: a Brute-Force algorithm that guarantees optimality by exhaustively checking every possible subset of vertices; a Greedy algorithm that uses a simple heuristic of repeatedly selecting the vertex with the highest degree; and a 2-Approximation algorithm that provides a provable guarantee that its solution will be no more than twice the size of the true minimum. To bridge the gap between abstract theory and practical understanding, these algorithms are integrated into an interactive visualization tool, allowing for a step-by-step observation of how each method traverses the graph and constructs its solution.

Through this comparative exploration and visualization, this project aims to provide clear insights into the behavior, performance, and trade-offs associated with different approaches to solving the vertex cover problem. By observing these algorithms in action, we can better appreciate the intricate balance between computational cost and the quality of a solution, a central theme in the study of algorithm design and combinatorial optimization.

II. LITERATURE REVIEW

A. Vertex Cover

The Vertex Cover problem is a central and well-studied problem in the fields of graph theory and computational complexity. Formally, let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. A vertex cover of G is a subset of vertices $V' \subseteq V$ such that for every

edge $(u, v) \in E$, at least one of its endpoints is included in the subset, i.e., $\{u, v\} \cap V' \neq \emptyset$. While any graph has several possible vertex covers (with the set of all vertices, V , always being a trivial one), the optimization challenge lies in finding a *minimum vertex cover*. This is a vertex cover that has the smallest possible cardinality, denoted by $\tau(G)$. The associated decision problem asks, for a given graph G and an integer k , whether there exists a vertex cover of size at most k .

The computational difficulty of the vertex cover problem is one of its most defining characteristics. It was famously included in Richard Karp's list of 21 NP-complete problems in his seminal 1972 paper, "Reducibility Among Combinatorial Problems." The classification of a problem as NP-complete signifies that there is no known algorithm that can solve it in polynomial time for all inputs. Furthermore, if such an algorithm were ever discovered for vertex cover, it would imply that $P=NP$, which would resolve one of the most profound open questions in computer science. This inherent hardness makes finding the exact minimum vertex cover for large graphs computationally intractable, as any known exact algorithm, such as brute-force, has a runtime that grows exponentially with the number of vertices. Consequently, the study of vertex cover has largely focused on developing approximation algorithms and heuristics that can find near-optimal solutions in a reasonable amount of time

B. Brute Force Algorithm

The most direct approach to solving the vertex cover problem is through a brute-force algorithm. This method is a straightforward, exhaustive search paradigm that systematically enumerates every possible candidate for a solution and checks whether each candidate satisfies the problem's statement. For the vertex cover problem, the candidates are all possible subsets of the graph's vertex set, V . The total number of such subsets is $2^{|V|}$, where $|V|$ is the number of vertices. A brute-force implementation would generate each of these subsets and, for each one, verify if it constitutes a valid vertex cover by checking if every edge in the graph is covered. The algorithm would keep track of the smallest valid cover found during this exhaustive process.

To find the minimum vertex cover specifically, the brute-force strategy can be refined. Instead of generating all $2^{|V|}$ subsets at once, the algorithm can iterate through possible cover sizes, k , from 0 to $|V|$. For each k , it generates all vertex subsets of size k (i.e., all combinations of k vertices) and checks if any of them form a valid vertex cover. The first value of k for which a valid cover is found will correspond to the size of the minimum vertex cover, and the corresponding subset will be an optimal solution. While this method guarantees optimality, its runtime complexity of $O(2^{|V|} \cdot |E|)$ makes it practical only for very small graphs, serving primarily as a baseline for understanding the problem's difficulty and for verifying the correctness of more sophisticated algorithms on small test cases.

C. Greedy Algorithm

Given the inefficiency of brute-force methods, heuristic approaches are often employed to find good, albeit not

necessarily optimal, solutions quickly. The Greedy algorithm for vertex cover is a prime example of such a heuristic. Its strategy is intuitively simple: at each step, select the vertex that covers the most uncovered edges. This means the algorithm calculates the degree (the number of incident edges) of every vertex in the current graph state and adds the vertex with the highest degree to the vertex cover. After a vertex is chosen, it and all its incident edges are removed from the graph. This process is repeated until no edges remain.

While this greedy approach is fast and often produces reasonably small vertex covers, it provides no guarantee of optimality. It is possible to construct graphs where this strategy yields a solution that is significantly larger than the minimum vertex cover. The ratio between the size of the cover found by the greedy algorithm and the size of the optimal cover can be as large as $O(\log |V|)$. Unlike the 2-approximation algorithm, it does not have a constant approximation ratio, meaning its performance relative to the optimal solution can degrade as the graph size increases. Nevertheless, its simplicity and efficiency make it a valuable tool for obtaining a quick estimate or an initial solution.

D. 2-Approximation Algorithm

In contrast to heuristics with no performance guarantees, approximation algorithms offer a provable bound on the quality of their solution relative to the optimal one. The 2-Approximation algorithm for vertex cover is a classic and elegant example. The algorithm operates on a simple iterative process: as long as there are edges remaining in the graph, it picks an arbitrary edge, say (u, v) , adds both of its endpoints, u and v , to the vertex cover, and then removes both vertices and all their incident edges from the graph. This loop continues until all edges have been covered.

The power of this algorithm lies in its approximation ratio of 2. This means the size of the vertex cover it produces is guaranteed to be no more than twice the size of the true minimum vertex cover. This guarantee arises from a simple observation: for every edge (u, v) chosen by the algorithm, at least one of its endpoints must be in any valid vertex cover, including the minimum one. Since the algorithm adds both endpoints, it adds at most twice as many vertices as would be required for an optimal cover of those same chosen edges. With a time complexity of $O(|E|)$, it provides a robust and efficient method for finding a good-quality solution with a predictable upper bound on its error.

III. ALGORITHM IMPLEMENTATION

A. Problem Statement

The Minimum Vertex Cover problem states: Given a graph $G = (V, E)$, find the smallest subset of vertices $C \subset V$ such that every edge in E has at least one endpoint in C . The program will take in an undirected graph $G = (V, E)$. Then the program will output a list of edges, $C \subset V$ that is the vertex cover of the graph. That is, for every (u, v) in E , then either u or v (or both) is in C .

Generally across the algorithms, to verify if a given C is a solution to the Vertex Cover problem. That is C meets the criteria of a vertex cover, we use the following procedure.

```

FUNCTION IsVertexCover(Graph G, Subset S)
INPUT: A graph G with edges E, and a subset of vertices S
OUTPUT: true if S is a vertex cover, false otherwise
FOR EACH edge (u, v) IN E:
  IF (u is NOT IN S) AND (v is NOT IN S)
  THEN
    RETURN false
  END IF
END FOR
RETURN true
  
```

The complexity of this function is $O(m)$ where m denotes the number of edges in the graph.

B. Brute Force Implementation

1.) Problem Mapping

a. Solution space

For a graph with $|V|$ vertices, the total number of possible subsets is $2^{|V|}$. All possible subset is a candidate for vertex cover.

b. Generating Function

The generating function generates all possible subsets of the vertices. The complexity for generating all subsets is $O(2^{|V|})$ where $|V|$ is the number of vertices

c. Validation Function

For every subset of vertices that is generated, The *IsVertexCover* function is used to verify if the given subset is a vertex cover.

2.) Complexity Analysis

Since the algorithm generates $2^{|V|}$ possible solutions, and every solution is verified in $O(|E|)$ time. The total complexity for the brute force algorithm is $O(2^{|V|}|E|)$.

3.) Implementation

The following is the pseudocode for the implementation of the brute force algorithm.

```

FUNCTION BruteForceVertexCover(Graph G):
INPUT: A graph G with vertices V and edges E
OUTPUT: A minimum vertex cover V_cover
FOR k FROM 0 TO size(V) :
  all_subsets_of_size_k =
  generate_combinations(V, k)
  FOR EACH subset S IN
  all_subsets_of_size_k:
    IF IsVertexCover(G, S) THEN
      RETURN S
    END IF
  END FOR
END FOR
RETURN an empty set
  
```

C. Greedy Implementation

1.) Problem mapping

a. Greedy Heuristic

The greedy algorithm selects the vertex with the highest degree to cover the maximum number of edges.

b. Solution Construction

It iteratively builds a solution by adding the highest-degree vertex to the cover, then removing that vertex and all its incident edges from consideration.

c. Termination and Validity

The process repeats until no edges remain, guaranteeing that the final set of chosen vertices forms a valid vertex cover for the entire graph/

2.) Complexity Analysis

The algorithm's main loop continues as long as there are edges in the graph. In the worst case, this loop can run $|V|$ times. Within each iteration, the most computationally expensive task is to find the vertex with the highest degree. This requires calculating the degrees of all vertices by iterating through the remaining edges, which takes $O(|E|)$ time. Therefore, the total time complexity for this implementation of the greedy algorithm is $O(|V| \cdot |E|)$.

3.) Implementation

The following is the pseudocode for the greedy algorithm.

```

FUNCTION GreedyVertexCover(Graph G):
INPUT: A graph G with vertices V and edges E
OUTPUT: A vertex cover V_cover
V_cover = an empty set
E_remaining = a copy of E
WHILE E_remaining is not empty:
  let v_max_degree = the vertex in
  V with the highest degree in the
  subgraph formed by E_remaining
  add v_max_degree to V_cover
  edges_to_remove = an empty set
  FOR EACH edge (u, v) IN
  E_remaining:
    IF u == v_max_degree OR v ==
    v_max_degree THEN
      add edge (u, v) to
      edges_to_remove
    END IF
  END FOR
  E_remaining = E_remaining -
  edges_to_remove
END WHILE
RETURN V_cover
  
```

D. 2-Approximation Implementation

1.) Problem Mapping

a. Core Idea

The algorithm is based on the principle that for any edge (u, v) , a valid vertex cover must contain either u or v (or both). This algorithm conservatively includes both endpoints of a selected edge to guarantee coverage.

- b. Solution Construction
It iteratively picks an arbitrary uncovered edge, adds both of its vertices to the cover, and then removes all edges incident to either of these two vertices.
- c. Termination and Validity
The process repeats until no edges remain, which ensures a valid cover. This method is a 2-Approximation algorithm, meaning the size of the cover it produces is provably no more than twice the size of the optimal minimum vertex cover.

2.) Complexity Analysis

The algorithm processes each edge in the graph at most once. The main loop continues as long as there are uncovered edges. In each step, at least one edge is selected and removed, along with other incident edges. With an efficient implementation (e.g., using adjacency lists), the total time complexity is linear in the size of the graph, which is $O(|V| + |E|)$.

3.) Implementation

The following is the pseudocode for the implementation of the 2-Approximation algorithm.

FUNCTION TwoApproxVertexCover(Graph G):

INPUT: A graph G with vertices V and edges E

OUTPUT: A vertex cover V_cover

```

V_cover = an empty set
E_remaining = a copy of E
WHILE E_remaining is not empty:
  let (u, v) be an edge in E_remaining
  add u to V_cover
  add v to V_cover
  edges_to_remove = an empty set
  FOR EACH edge (x, y) IN E_remaining:
    IF x == u OR x == v OR y == u
    OR y == v THEN
      add edge (x, y) to edges_to_remove
  END IF
  END FOR
  E_remaining = E_remaining - edges_to_remove
END WHILE
RETURN V_cover

```

IV. TESTING

A. Test Case 1

The following is the graph input for the first test case

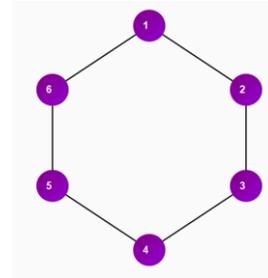


Fig. 1. First Test Case

Brute Force Result:

	Vertex Cover Size: 3 Total Steps: 32 Time Taken: 0.0002 seconds Vertices: [1, 3, 5]
--	--

Greedy Result:

	Vertex Cover Size: 3 Total Steps: 3 Time Taken: 0.0001 seconds Vertices: [2, 4, 6]
--	---

2-Approximation Result:

	Vertex Cover Size: 6 Total Steps: 3 Time Taken: 0.0001 seconds Vertices: [1, 2, 3, 4, 5, 6]
--	--

B. Test Case 2

The following is the graph input for the second test case.

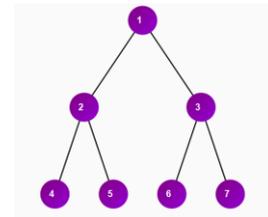


Fig. 2. Second Test Case

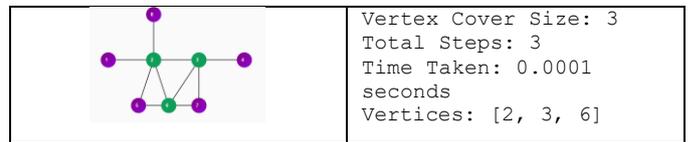
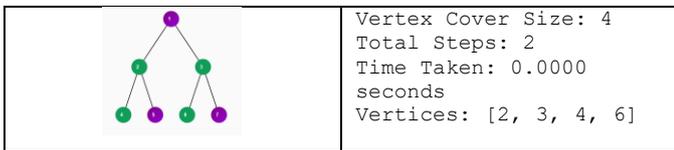
Brute Force Result:

	Vertex Cover Size: 2 Total Steps: 18 Time Taken: 0.0001 seconds Vertices: [2, 3]
--	---

Greedy Result:

	Vertex Cover Size: 2 Total Steps: 2 Time Taken: 0.0001 seconds Vertices: [2, 3]
--	--

2-Approximation Result:



C. Test Case 3

The following is the graph input for the third test case.

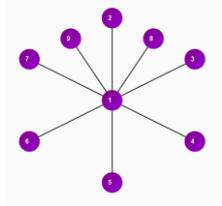
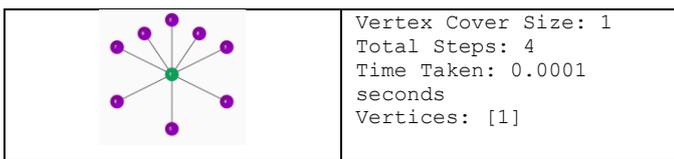
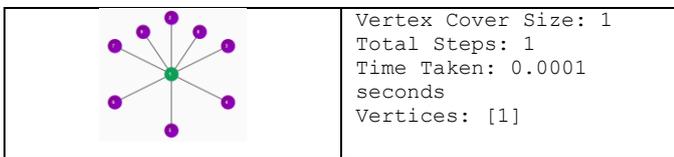


Fig. 3. Third Test Case

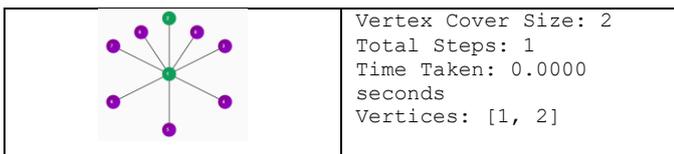
Brute Force Result:



Greedy Result:



2-Approximation Result:



D. Test Case 4

The following is the graph input for the fourth test case.

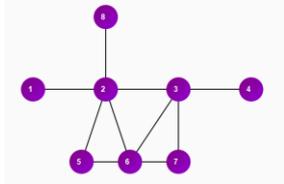
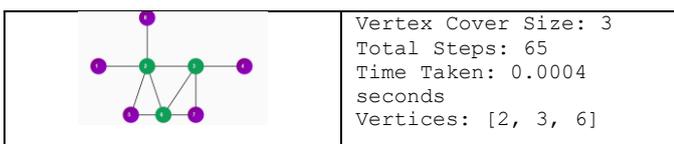


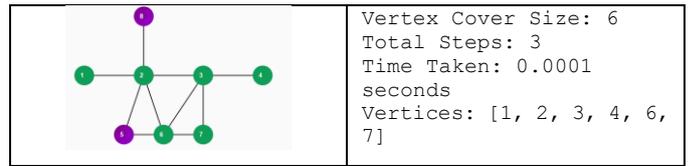
Fig. 4. Fourth Test Case

Brute Force Result:



Greedy Result:

2-Approximation Result:



E. Test Case 5

The following is the graph input for the fifth test case.

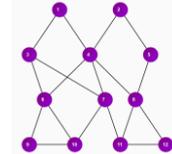
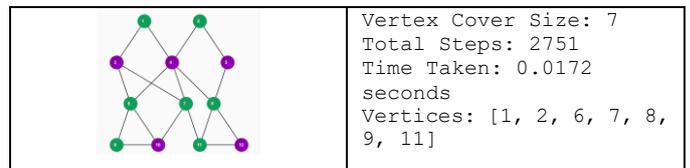
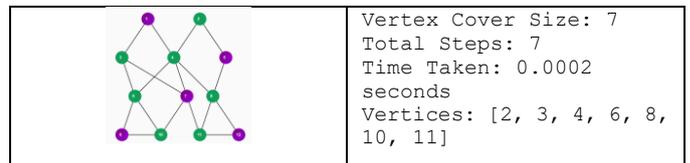


Fig. 5. Fifth Test Case

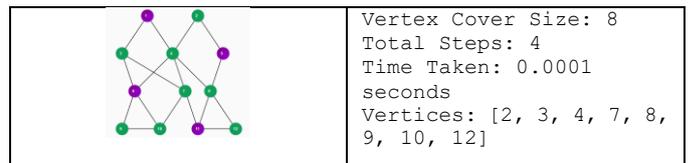
Brute Force Result:



Greedy Result:



2-Approximation Result:



F. Test Case 6

The following is the graph input for the fifth test case

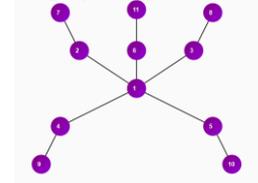
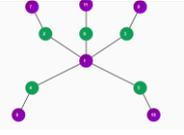
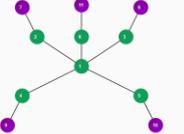


Fig. 6. Sixth Test Case

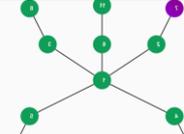
Brute Force Result:

	Vertex Cover Size: 5 Total Steps: 779 Time Taken: 0.0046 seconds Vertices: [2, 3, 4, 5, 6]
---	---

Greedy Result:

	Vertex Cover Size: 6 Total Steps: 6 Time Taken: 0.0002 seconds Vertices: [1, 2, 3, 4, 5, 6]
---	--

2-Approximation Result:

	Vertex Cover Size: 10 Total Steps: 5 Time Taken: 0.0001 seconds Vertices: [1, 2, 3, 4, 5, 6, 8, 9, 10, 11]
---	---

V. RESULTS DISCUSSION

TABLE I. VERTEX COVER SIZE

Test Case	Algorithm		
	Brute-Force	Greedy	2-Approximation
1	3	3	6
2	2	2	4
3	1	1	2
4	3	3	6
5	7	7	8
6	5	6	10

This table clearly illustrates the trade-off in solution quality. The Brute-Force algorithm provides the benchmark for the smallest possible cover size. The Greedy algorithm performs exceptionally well, finding the optimal solution in 5 out of 6 cases and being off by only one vertex in the other two. In contrast, the 2-Approximation algorithm consistently produces larger covers, sometimes reaching its worst-case bound of being twice the size of the optimal solution (as seen in Test Cases 1, 2, 3, and 6), but never exceeding it.

TABLE II. TOTAL STEPS

Test Case	Algorithm		
	Brute-Force	Greedy	2-Approximation
1	32	3	3
2	18	2	2
3	4	1	1
4	65	3	3
5	2751	7	4
6	779	6	5

TABLE III. TIME TAKEN (SECONDS)

Test Case	Algorithm		
	Brute-Force	Greedy	2-Approximation
1	0.0002	0.0001	0.0001
2	0.0001	0.0001	0.0000
3	0.0001	0.0001	0.0000
4	0.0004	0.0001	0.0001
5	0.0172	0.0002	0.0001
6	0.0046	0.0002	0.0001

The computational cost difference is starkly evident here. The "Total Steps" for the Brute-Force algorithm grows exponentially, as seen in the jump from 65 steps in Test Case 4 to 2751 steps in Test Case 5. This highlights its inefficiency. The Greedy and 2-Approximation algorithms remain highly efficient, with their step counts staying low and scaling much more gracefully with the size and complexity of the graph.

Time is the real-world manifestation of computational steps. The execution time for the Brute-Force algorithm, while small here, clearly increases at a much faster rate than the other two algorithms. The Greedy and 2-Approximation algorithms are nearly instantaneous for graphs of this size, reinforcing their practicality for larger, more complex problems where the Brute-Force method would take an unacceptably long time.

TABLE IV. VERTICES IN FINAL COVER

Test Case	Algorithm		
	Brute-Force	Greedy	2-Approximation
1	[1, 3, 5]	[2, 4, 6]	[1, 2, 3, 4, 5, 6]
2	[2, 3]	[2, 3]	[2, 3, 4, 6]
3	[1]	[1]	[1, 2]
4	[2, 3, 6]	[2, 3, 6]	[1, 2, 3, 4, 6, 7]
5	[1, 2, 6, 7, 8, 9, 11]	[2, 3, 4, 6, 10, 11]	[2, 3, 4, 7, 8, 9, 10, 12]
6	[2, 3, 4, 5, 6]	[1, 2, 3, 4, 5, 6]	[1, 2, 3, 4, 5, 6, 8, 9, 10]

This table provides the qualitative data behind the numbers. It allows us to see the different "strategies" each algorithm took. In Test Case 1, we see that both [1, 3, 5] and [2, 4, 6] are valid optimal solutions. In Test Case 6, we can trace the Greedy algorithm's suboptimal choice of vertex 1, which led to a larger final cover compared to the Brute-Force solution. This level of detail is crucial for understanding *how* an algorithm arrives at its solution and why its performance varies depending on the graph's structure.

The testing phase of this project provides a clear and practical demonstration of the theoretical concepts governing the vertex cover problem. By comparing the Brute-Force, Greedy, and 2-Approximation algorithms across a variety of graph structures, we can empirically observe the fundamental trade-off between computational cost and solution optimality. The results from the six test cases not only validate the

theoretical performance guarantees and complexities of each algorithm but also offer nuanced insights into how graph topology influences their behavior.

A. Brute Force

Across all test cases, the Brute-Force algorithm successfully identified the minimum vertex cover, serving as the essential benchmark against which the other algorithms are measured. Its strength lies in its exhaustive search, which guarantees optimality by systematically checking every possible subset of vertices. For instance, in Test Case 1 (a 6-cycle graph), it correctly found a minimum cover of size 3, and in Test Case 3 (a star graph), it identified the single-vertex optimal cover.

However, the empirical data starkly illustrates the algorithm's prohibitive computational cost, which is its defining weakness. The Time Taken and Total Steps metrics scale exponentially with the number of vertices ($|V|$). This is evident when comparing the simple 6-vertex graph in Test Case 1 (32 steps) to the more complex 12-vertex graph in Test Case 5, which required 2751 steps and a significantly longer computation time. While the times recorded are small on these limited test cases, they reflect the $O(2^{|V|}|E|)$ complexity. This exponential growth renders the Brute-Force approach computationally infeasible for all but the smallest or simplest of graphs, reinforcing its classification as an NP-hard problem and underscoring the necessity for more efficient heuristic and approximation methods in practical, real-world applications.

B. Greedy

The Greedy algorithm, which iteratively selects the vertex with the highest degree, demonstrates a fascinating and varied performance across the test cases. Its primary allure is its speed and simplicity, consistently outperforming the Brute-Force method in terms of steps and time. However, its effectiveness in finding a near-optimal solution is highly dependent on the structure of the input graph.

However The algorithm's performance was optimal in several instances. In Test Case 3, the star graph, the Greedy algorithm immediately identifies the central vertex (node 1) as the optimal one-vertex cover. This is a classic example where the greedy heuristic excels, as covering the single, high-degree central node is the most efficient solution. Similarly, in the tree structure of Test Case 2 and the more complex graph of Test Case 4, the Greedy algorithm found the optimal solution. In these cases, the highest-degree vertices happened to be part of an optimal vertex cover.

Conversely, Test Case 6 reveals the algorithm's potential for suboptimal choices. The graph is a "star-of-stars" or a multi-star graph. The central node (1) has the highest initial degree. The Greedy algorithm selects it first. However, after removing node 1, five disconnected edges remain, forcing the algorithm to select one endpoint from each, resulting in a total cover of size 6. The Brute-Force algorithm found a smaller cover of size 5 by selecting the "satellite" vertices (2, 3, 4, 5, 6), which perfectly cover all edges. This illustrates the myopic nature of the greedy choice; selecting the vertex with the highest degree is a locally optimal decision that does not guarantee a globally optimal result. This aligns with the theoretical understanding that the

Greedy algorithm's approximation ratio is $O(\log(|V|))$, meaning its solution can be significantly worse than the optimal one as the graph grows.

C. 2-Approximation

The 2-Approximation algorithm offers a formal contract: it will deliver a valid vertex cover that is no more than twice the size of the minimum cover. The test results consistently uphold this theoretical guarantee. In every single test case, a size of the vertex cover produced by this algorithm was less than or equal to two times the size of the optimal cover found by the Brute-Force method

Its mechanism—picking an arbitrary edge and adding both its vertices to the cover—is simple and fast, with a linear time complexity of $O(|V| + |E|)$. This efficiency is evident in the low number of steps and near-instantaneous execution times. However, the results also show that this guarantee often comes at the cost of solution quality when compared to the Greedy algorithm.

A "worst-case" scenario for the 2-Approximation algorithm is vividly demonstrated in Test Case 1, the 6-cycle graph. The optimal cover size is 3. The 2-Approximation algorithm, by picking three disjoint edges, selects both endpoints for each, resulting in a cover of size 6, exactly twice the size of the optimal solution. This occurs because for each edge (u, v) selected, an optimal cover might only need one of those vertices (e.g., vertex u), but the algorithm conservatively adds both. A similar result is seen in Test Case 3 (the star graph), where it produces a cover of size 2 while the optimal is size 1. By picking an edge like $(1, 2)$, it adds both nodes, where only node 1 was necessary.

This behavior highlights the trade-off inherent in this algorithm. While it avoids the catastrophic failures that a heuristic like Greedy *could* produce on certain graphs, it also lacks the "cleverness" to identify obviously better choices. Its strength is not in finding the best possible solution, but in providing a reliable, efficient, and mathematically-provable boundary on how far its solution can deviate from the true optimum.

In conclusion, the empirical results derived from the test cases align perfectly with the established theoretical foundations of algorithm analysis. They confirm the NP-hard nature of the vertex cover problem, demonstrate the practical limitations of brute-force solutions, and provide a tangible comparison of heuristic and approximation strategies. The visualization and testing have successfully bridged the gap between abstract complexity theory and the concrete performance of algorithms, offering clear insights into the crucial balance between speed, resource consumption, and the quality of a final solution.

VI. CONCLUSION

This investigation into the Vertex Cover problem through the implementation and testing of Brute-Force, Greedy, and 2-Approximation algorithms successfully demonstrated the fundamental trade-off between solution optimality and computational efficiency. The empirical results confirmed that while the Brute-Force method guarantees a minimum vertex cover, its exponential complexity renders it impractical for all

but the smallest graphs. In contrast, both the Greedy and 2-Approximation algorithms provide efficient, polynomial-time alternatives, though with differing strengths: the Greedy heuristic often yields optimal or near-optimal results but lacks a performance guarantee, whereas the 2-Approximation algorithm, while sometimes producing larger covers, provides a reliable and mathematically-proven upper bound on its solution size. Ultimately, the findings establish that the optimal choice of algorithm is not absolute but is contingent on the specific application's tolerance for sub-optimality versus its need for performance guarantees, clearly illustrating the practical implications of theoretical computer science principles.

VII. APPENDIX

VIDEO LINK AT YOUTUBE

The link to the YouTube video can be found in the following:

<https://www.youtube.com/watch?v=bvgVjbuS1gY>

ACKNOWLEDGMENT

The Author would formally thank:

1. Allah SWT. The almighty that has given me strength to finish this paper
2. Nur Ulva Maulidevi and Rinaldi Munir for being an amazing lecturer for the IF2211 Course
3. All other parties the author can't list one by one.

REFERENCES

The source code for the program referenced above can be found in the following GitHub Repository:

https://github.com/adielrum/Makalah_10123004.git

- [1] Munir, R. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/04-Algorithm-Greedy-\(2025\)-Bag1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/04-Algorithm-Greedy-(2025)-Bag1.pdf) . Diakses pada 24 Juni 2025.

- [2] Munir, R. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/05-Algorithm-Greedy-\(2025\)-Bag2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/05-Algorithm-Greedy-(2025)-Bag2.pdf) . Diakses pada 24 Juni 2025.
- [3] Munir, R. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/06-Algorithm-Greedy-\(2025\)-Bag3.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/06-Algorithm-Greedy-(2025)-Bag3.pdf) . Diakses pada 24 Juni 2025.
- [4] Munir, R. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/02-Algorithm-Brute-Force-\(2025\)-Bag1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/02-Algorithm-Brute-Force-(2025)-Bag1.pdf). Diakses pada 24 Juni 2025.
- [5] Munir, R. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/03-Algorithm-Brute-Force-\(2025\)-Bag2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/03-Algorithm-Brute-Force-(2025)-Bag2.pdf). Diakses pada 24 Juni 2025.
- [6] Chen, Jianer, Iyad A. Kanj, and Ge Xia. "Improved upper bounds for vertex cover." *Theoretical Computer Science* 411.40-42 (2010): 3736-3756.
- [7] Chen, Jianer, Iyad A. Kanj, and Weijia Jia. "Vertex cover: further observations and further improvements." *Journal of Algorithms* 41.2 (2001): 280-301.
- [8] Chen, Jianer, Iyad A. Kanj, and Ge Xia. "Improved upper bounds for vertex cover." *Theoretical Computer Science* 411.40-42 (2010): 3736-3756.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025



Adiel Rum - 10123004