Implementation of $\frac{1}{2}$-Approximation Path Cover Algorithm and Its Empirical Analysis

Junyuan Lin* Guangpeng Ren†

Abstract - In this paper, we demonstrate a deterministic algorithm that approximates the optimal path cover on various graphs and networks derived from a wide range of real-world problems. Based on the $\frac{1}{2}$-Approximation Path Cover Algorithm by Moran et al., we first organize the original algorithm into two versions - one with redundant edge removal and one without. To compare the two versions of algorithms, we prove the number of redundant edges for any general graphs to analyze the effects of edge removal. We also analyze theoretical guarantees of the two algorithms. To test the time complexity and performance, we conduct numerical tests on graphs with various structures and random weights, from structured ring graphs to random graphs, such as Erdős-Rényi graphs. The tests demonstrate the advantage in memory saving of the algorithm that does not remove any redundant edges and time saving of the other one, especially on large and high density graphs. We also perform tests on various graphs and networks derived from a wide range of real-world problems to suggest the effectiveness and applicability of both algorithms.

Keywords : path cover; graph algorithms; Watts-Strogatz graph; Erdős-Rényi graph; greedy algorithm; social network

Mathematics Subject Classification (2020) **:** 65K05

1 Introduction

In graph theory, a path cover of a graph refers to a set of paths that cover all vertices in the graph, where every vertex belongs to only one path. The optimal path cover on a weighted graph is a path cover that includes edges that have maximum (or minimum) weight sum. For any weighted graphs with unique weight on each edge, the optimal path cover is unique according to [1]. However, finding the optimal path cover in a graph is an NP-complete problem as shown in [15]. Therefore, we hope to apply approximation algorithms to estimate an optimal path cover. In [1], Moran et al. introduce three fundamental approximation algorithms for the maximum weighted path cover of a graph. $\frac{1}{2}$-Approximation Covering Algorithm is the most straight-forward one in the paper; it is a greedy algorithm that obtains an approximated maximum path cover with at least

*Department of Mathematics, Statistics and Data Sciences, Loyola Marymount University, Los Angeles, CA, USA. Email: junyuan.lin@lmu.edu

†Institute of Mathematical Sciences, Claremont Graduate University, Claremount, CA, USA. Email: guangpeng.ren@cgu.edu

$\frac{1}{2}$ of the optimal path cover's weight. There are two other path cover approximation algorithms introduced in [1] that are linear in programming. They both guarantee $\frac{2}{3}$ weight in undirected graphs and directed graphs. These two algorithms use the same method of finding a degree-constrained subgraph of the primary graph that has maximum weight. The degree of nodes in the subgraph is constrained to be less or equal to 2, and there are only paths and cycles left in the subgraph. Each cycle has at least three edges, and there is at least one edge in this cycle whose weight is less than $\frac{1}{3}$ of the total weight in this cycle. Therefore, these results give the $\frac{2}{3}$ theoretical bound of the algorithm.

In recent years, there have been works done on approximation algorithms that take different approaches. In [11], Tu and Zhou use primal-dual method and design a $\frac{1}{2}$-approximation algorithm called VCP_3 (Vertex Cover P_3) where P_3 is a path with 3 vertices. The main idea of VCP_3 is to remove redundant vertices as the algorithm goes. Its time complexity takes $O(M \cdot N)$ time, where M is the number of edges and N is the number of vertices in the graph covered by VCP_3. In [12], Zhang et al. provide an algorithm specifically for $MWVCC_k$ (Minimum Weight Connected k-Subgraph Cover). This algorithm is an optimization of the algorithm in [11], where it only considers when $k = 2$, yet the algorithm in [12] provides a $\frac{1}{k-1}$-approximation for the $MWVCC_k$ problem. Its time complexity is $O(N^2 \cdot M)$.

Finding approximated optimal path cover arises in many application problems. Researchers have been investigating path cover applications on various types of graphs, such as cocomparability graphs, cographs, interval graphs, block graphs, and permutation graphs in [5, 4, 6, 7]. For instance, finding the shortest distance or the route that costs the minimum time on a map originates from Dijkstra in [8] can be related to finding the mininal path cover. Based on the $\frac{1}{2}$-Approximation Covering Algorithm, Hu et al. approximate the level sets of the smooth error and further build adaptive multilevel structures that increase the accuracy and robustness of solving linear systems in weighted graph Laplacians. In [2], Ehikioya and Lu provide a path analysis of site visiting to help understand the website traffic and improve marketing strategies. In [9], Bertolino and Marre show a generalized algorithm specifically working towards the flow graph, which is a presentation of all possible paths in a program. In [3], Dwarakanath and Jankiti show that the path cover can provide the minimum number of test paths to cover different structural coverage criteria.

Nowadays, graphs and networks have become larger and more complex, calling for more robust path cover approximating algorithms. In this paper, we realize the $\frac{1}{2}$-Approximation Covering Algorithm in Python and derive a memory saving modification. The algorithm has several advantages: 1) It is a greedy algorithm and straightforward to implement. 2) Every step of the algorithm is deterministic, and there is no randomness involved. 3) There is a theoretical guarantee of time complexity. On top of these advantages, we compare the memory storage of the original algorithm to one that does not remove any redundant edges as it progresses. The motivation is that we observe that checking and

marking the redundant edges can acquire large memory storage especially as the graph gets larger and denser. Without removing redundant edges, one can significantly reduce the memory storage while still guaranteeing the $\frac{1}{2}$ weight lower bound. In Section 2, we review the $\frac{1}{2}$-Approximation Covering Algorithm by introducing necessary preliminaries, reorganizing the pseudocode and revisiting the theoretical analysis of the algorithm. In Section 3, we focus on the memory efficient modification of the algorithm and derive the theoretical guarantees of the modified version. In Section 4, we provide the numerical results and visualizations of both algorithms on Watts-Strogatz graph, a deterministic graph in the setting of this paper, and Erdős-Rényi graph, a random graph, as well as graphs from real-world situations to show the effectiveness and advantages of both algorithms.

2 $\frac{1}{2}$-Approximation Covering Algorithm

We first review the preliminaries that help explain the $\frac{1}{2}$-Approximation Covering Algorithm, denoted as Algorithm 1, and the complexity of the algorithm. Then we rewrite the pseudocode and review the theoretical results of the $\frac{1}{2}$-Approximation Covering Algorithm introduced in [1].

2.1 Preliminaries

Consider an undirected weighted graph G, which contains vertex set V, edge set E, and associating weights W. We denote N to be the number of vertices in V and M to be the number of edges in E. An edge $e = \{u, v\}$ in E contains two end points u and v.$w(e)$ is the weight of e from W. We denote OPT as the optimal (maximal) path cover of G and `cover` as the approximated path cover that estimates OPT. In addition, we denote H to be the number of edges in the path cover and K to be the number of paths in the path cover.

2.2 Pseudocode

We rewrite $\frac{1}{2}$-approximation path cover algorithm in [1] in Algorithm 1. Each step is deterministic and easy to follow. While the advantages are obvious, Algorithm 1 can take considerable memory space as the graph gets larger.

Algorithm 1 $\frac{1}{2}$ Approximation Path Cover

```
1: procedure [ cover ] = PathCover((A))
Require: A——an undirected positive weighted graph G=(V,E,W)
Ensure: cover——a path cover of graph G
2:     sorted_edges ← Sort the edges in descending order based on W
3:     for e={u, v} ∈ sorted_edges do
4:         if neither u nor v is in any paths in cover then
5:             Add {u, v} as a new path in cover
6:         else if u is the end point of a path in cover and v is not in any paths then
7:             Append {v} to cover {path that contains u}
8:             Remove {adj(u), u} from sorted_edges
9:         else if v is the end point of a path in cover and u is not in any paths then
10:            Append {u} to cover {path that contains v}
11:            Remove {adj(v), v} from sorted_edges
12:        else if u and v are the end points of different paths in cover then
13:            Merge two paths
14:            Remove {adj(v), v} and {adj(u), u} from sorted_edges
15:        end if
16:    end for
17: end procedure
```

Here, we denote $adj(u)$ as the adjacent nodes of u in G but not in `cover`. We can observe that Algorithm 1 is essentially a greedy algorithm by checking each edge based on the weight in descending order to build the path cover. A review of the theoretical guarantees for this algorithm in [1] is included in Section 2.3 for completeness.

2.3 Algorithm 1 Analysis

When using Algorithm 1 to find an approximated path cover, the following theorem stated in [1] gives the theoretical bounds on accuracy and complexity:

Theorem 2.1 *On a weighted graph G, assume that OPT is its optimal path cover and* `cover` *is an approximation of OPT obtained by Algorithm 1, the weight of* `cover` *is at least $\frac{1}{2}$ of the weight of OPT*

$$w(\texttt{cover}) \geq \frac{1}{2} \cdot w(OPT).$$

The time complexity for finding is `cover` *$O(M \cdot LogM)$, where M is the number of edges in graph G.*

Algorithm 1 produces `cover` whose weight is greater or equal to $\frac{1}{2}$ of $w(OPT)$ which is the weight sum of OPT, hence its name. Its time complexity is $O(M \cdot LogM)$, which follows from the fact that initialization, the execution time of the loop, the removal of

redundant edges, and the output are in $O(M)$, but the sorting time is in $O(M \cdot LogM)$. Thus, the entire test is eventually finished in $O(M \cdot LogM)$.

3 Memory Efficient Algorithm

To make Algorithm 1 more memory efficient, we derive an approach that takes less memory space while preserving the theoretical results by maintaining the redundant edges.

3.1 Motivations

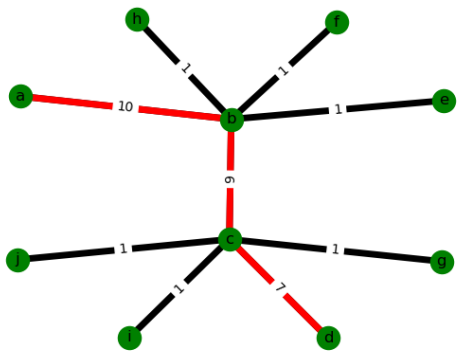

Figure 1: Redundancy

We use the following example to illustrate a big picture of the memory efficient algorithm. In Figure 1, edges in red are in path cover, and the edges in black are incident to the path. Algorithm 1 ranks the edge weights in descending order and picks edges $\{a, b\}$ and $\{b, c\}$. However, by the definition of path, only edges incident to end points (i.e. a and c) can be added to the path. Thus, checking and marking the redundant edges incident to b (the middle node of the path) takes extra memory space as it requires a variable to store. Since the program is going in descending order based on the edge weights, the primary algorithm keeps the redundant edges that are not necessary to process and achieves the goal of time saving. In the modified version, we maintain these redundant edges in the original set instead of creating extra storage space. The advantage of the memory saved becomes more significant, as the average degree of the graph and the number of nodes get higher. Meanwhile, the time complexity and $\frac{1}{2}$ weight bound are still guaranteed (Proof in Section 3.3).

3.2 Pseudocode

Here, we lay out the procedure of the memory efficient algorithm in Algorithm 2. In Algorithm 1, we need to store redundant edges with extra space to skip and save time

as the program goes through edges based on descending order of weight. Corresponding steps 8, 11, and 14 from Algorithm 1 are removed, thus, we can save memory space of storing redundant edges as the algorithm progresses. More numerical details on the efficiency increase will be discussed later in Section 4.

Algorithm 2 Memory Efficient Path Cover Approximating Algorithm

```
1: procedure [ cover ] = PathCover(A)
Require: A——an undirected positive weighted graph G=(V,E,W)
Ensure: cover——a path cover of graph G
2:     sorted_edges ← Sort the edges in descending order based on W
3:     for e={u, v} ∈ sorted_edges do
4:         if neither u nor v is in any paths in cover then
5:             Add {u, v} as a new path in cover
6:         else if u is the end point of a path in cover and v is not in any paths then
7:             Append {v} to cover {path that contains u}
8:         else if v is the end point of a path in cover and u is not in any paths then
9:             Append {u} to cover {path that contains v}
10:        else if u and v are the end points of different paths in cover then
11:            Merge two paths
12:        end if
13:    end for
14: end procedure
```

Notice that the algorithm still picks the same set of edges for `cover`, since the redundant edges do not make into the approximated cover and it does not matter whether we remove them or not.

3.3 Algorithm 2 Analysis

For the derived Algorithm 2, we claim the following theorem:

Theorem 3.1 *On a weighted graph G, assuming that OPT is the optimal path cover, and* `cover` *is the approximation of OPT obtained by Algorithm 2, we have*

$$w(\texttt{cover}) \geq \frac{1}{2} \cdot w(OPT).$$

The time complexity of finding `cover` *by Algorithm 2 is $O(M \cdot logM)$ where M is the number of edges in G.*

Proof.

As mentioned in Sections 3.1 and 3.2, the removed steps do not change the resulting approximated path cover. So according to Theorem 2.1, we have

$$w(\texttt{cover}) \geq \frac{1}{2} \cdot w(OPT),$$

where `cover` is generated by Algorithm 2.

The initialization, the execution time of the loop, and the output, are in $O(M)$. The sorting time is in $O(M \cdot LogM)$. Thus, the entire test has computational complexity $O(M \cdot LogM)$. □

Since now we understand that the advantage of Algorithm 2 comes from the memory saved of redundant edges, we claim the following theorem:

Theorem 3.2 *Let $\tilde{H}$ be the set of redundant edges and M be the edges in path cover, the memory ratio of Algorithm 1 and 2 can be written as the following:*

$$\frac{H}{H + \tilde{H}}$$

The ratio is bounded by the following best and worst case:

$$\frac{H}{2H} < \frac{H}{H + \tilde{H}} \leq \frac{H}{H}$$

The memory complexity is therefore represented as $O(\frac{H}{H+\tilde{H}})$

3.4 Removed Edges

We further theoretically analyze the gain in computation by removing redundant edges. First, let us visualize the process of removing redundant edges in Algorithm 1. Here, we take the Watts-Strogatz graph in the setting of structured ring graphs as an example shown in Figure 2.

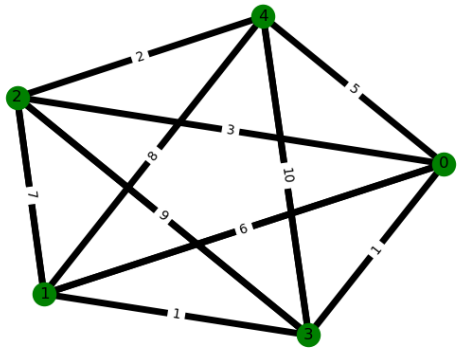

Figure 2: Watts-Strogatz Graph

Algorithm 1 sorts the edges in descending order of weights, i.e. {0, 1}, {1, 4}, {3, 4}, {2, 3}, {0, 2}, {0, 4}, {0, 3}, {1, 2}, {2, 4}, and {1, 3} based on Figure 2. Processing to Figure 3 on the left, first, it takes {0, 1}; there are no redundant edges yet since every

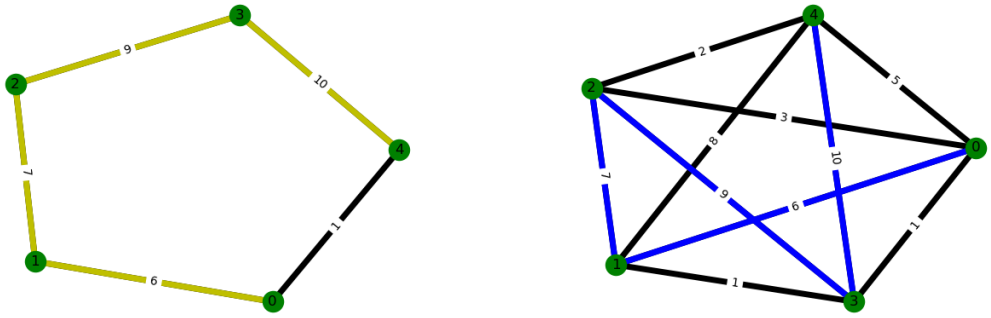

Figure 3: Results from Algorithm 1 (left) and Algorithm 2 (right)

other edge can be potentially chosen. Then, it takes {1, 4}. All edges incident to node 1 cannot be taken because the selected edges must make a path. It takes {3, 4} next, then {0, 4} and {2, 4} are eliminated for the same reason. Lastly, it removes {0, 3} after taking {2, 3}. {0, 2} is now the only remaining edge, and it cannot be taken. Therefore, Algorithm 1 terminates, and the green edges are selected into the path cover.

Figure 3 on the right suggests the end results of Algorithm 2. It takes {0, 1} as the first edge and adds {1, 4}, {3, 4}, and {2, 3} to the path cover consecutively. When checking other edges, we find that adding any one of them would make the selected edges no longer a path. Therefore, Algorithm 2 terminates. The edges marked in blue in Figure 3 on the right are the selected as the approximated path cover.

We can observe that two algorithms select the exact same set of edges into the approximated path cover as mentioned in our previous theoretical analysis. Algorithm 2 checks all ten edges before it terminates, where Algorithm 1 only checks five edges since redundant edges are removed along the way. We generalize our findings on the number of edges removed in the following claims:

Lemma 3.3 *In a connected weighted graph G, let H be the number of edges in the path cover and N be the number of vertices in the graph. If G is covered by one single path, we have:*

$$H = N - 1$$

Proof.

By induction:

Base Case: Suppose we have a connected weighted graph G which is an edge connected by 2 vertices, G itself would be the one single path. In this case, $H = 1$ and $N = 2$. Then we have:

$$H = N - 1$$

Induction Step: Let $k \in \mathbb{Z}^+$ be given, and suppose that it is true for the case $N = k$:

$$H = k - 1 = N - 1$$

We add one more point to G, which makes number of node $N = k + 1$. There has to be one edge that connects the new node to the rest to cover the full node set of G. So we have:

$$\begin{aligned} H &= k - 1 + 1 \\ &= (k + 1) - 1 \\ &= N - 1 \end{aligned}$$

It holds for $N = k + 1$, and the proof of induction is complete. □
The following Corollary 3.2 can be derived from Lemma 3.3:

Corollary 3.4 *In a fully connected weighted graph G, denote H as the number of edges in the path cover and N as the number of vertices in the graph, we know that*

$$H = N - 1$$

Proof.

Since fully connected graphs are guaranteed to be covered by a single path, it can be easily proved by Lemma 3.3. □
However, this is not guaranteed for any connected but not fully connected graphs.

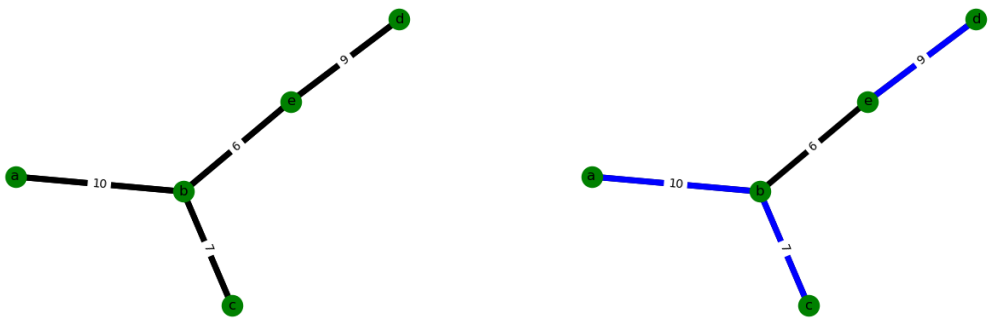

Figure 4: Original Graph (left) and Algorithm 2 (No edges removed) (right)

For example, Figure 4 on the left is a connected graph but not a fully connected graph. We can see by launching Algorithm 2, that there are two paths in the path cover (see Figure 4 on the right). In this case, H does not equal $N - 1$. For general cases where graphs are less connected, we claim the following theorem to conclude the number of removed edges.

Theorem 3.5 *In a weighted undirected graph, let H be the number of edges in path cover, N be the number of vertices in the graph, K be the number of paths in path cover. H can be expressed as the following:*

$$H = N - K$$

Proof. To show $H = N - K$

By Lemma 3.3,

$$H = \sum_{i=1}^{K} H_i = \sum_{i=1}^{K} (N_i - 1) = \sum_{i=1}^{K} (N_i) - K = N - K$$

where H_i is the number of edges and N_i is the number of nodes in the $i - th$ path in the path cover. Since all i paths cover the entire vertex set, $\sum_{i=1}^{K}(N_i) = N$, Theorem 3.5 is therefore proved. □

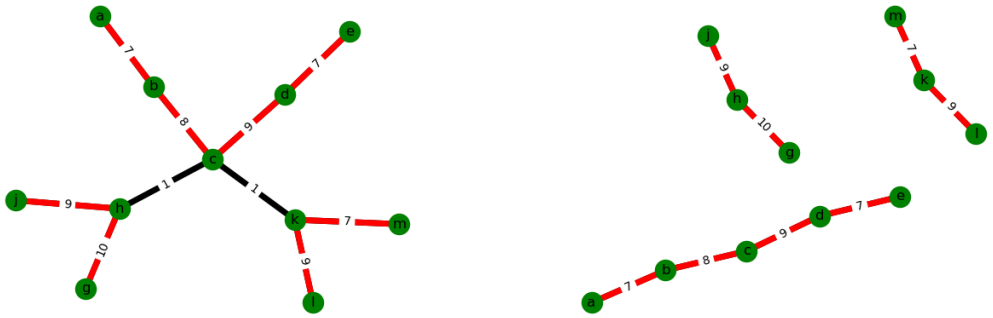

Figure 5: Theorem 3.5 Visualization 1 (left) and Theorem 3.5 Visualization 2 (right)

As an example, Figure 5 on the left displays a graph containing three paths without removing the redundant edges, and Figure 5 on the right visualizes the graph that removes the redundant edges. We observe that $N = 14$ and $K = 3$. Thus $H = 11$, and it verifies the formula given by Theorem 3.5.

4 Numerical Tests

The numerical tests are conducted with a 3.30 GHz Intel Core i7-11370H CPU, a quad-core processor, and 32 GB of RAM. The test program is written in Python, importing the packages `NetworkX` and `NumPy`. The dictionary is the principle data structure throughout the program, since it is implemented as hash tables, and its average time complexity is $O(1)$. In the end, packages of `line-profiler` and `sys` are imported to record the computational time.

Tables 1-7 and Figures 6-8 provide the numerical results from the tests and visualizations on both Watts-Strogatz graphs and Erdős-Rényi graphs. We set 4, 6 and 8 as the degree of nodes in Watts-Strogatz and $\frac{2 \cdot ln(M)}{M}$, $\frac{2.5 \cdot ln(M)}{M}$ and $\frac{3 \cdot ln(M)}{M}$ as the probability of generating edges in Erdős-Rényi, where M is the number of edges, to see how it would affect the results as the density gets higher. We denote the average degree of nodes as D_{avg}.

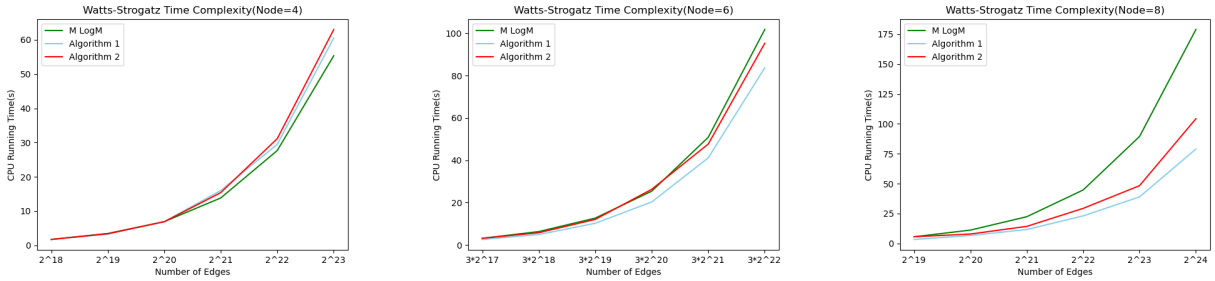

Figure 6: Watts-Strogatz Graph Node=4 (left), Node=6 (middle), Node=8 (right)

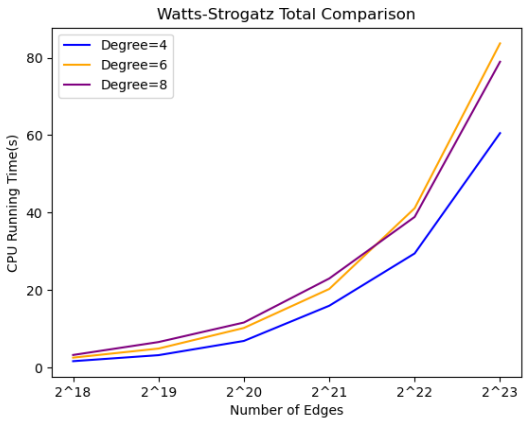

Figure 7: Watts-Strogatz Total Comparison

Table 1: Watts-Strogatz Graph (Degree of nodes = 4)

N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
131,072	262,144	4	0.27s	0.19s	1.42
262,144	524,288	4	0.38s	0.38s	1.00
524,288	1,048,576	4	0.83s	0.69s	1.20
1,048,576	2,097,152	4	1.61s	1.70s	0.95
2,097,152	4,194,304	4	3.09s	3.14s	0.98
4,194,304	8,388,608	4	6.46s	6.43s	1.00

Table 2: Watts-Strogatz Graph (Degree of nodes = 6)

N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
131,072	393,216	6	0.20s	0.27s	0.74
262,144	786,432	6	0.40s	0.48s	0.83
524,288	1,572,864	6	0.78s	1.03s	0.76
1,048,576	3,145,728	6	1.74s	1.98s	0.88
2,097,152	6,291,456	6	3.46s	3.88s	0.89
4,194,304	12,582,912	6	7.02s	8.40s	0.83

Table 3: Watts-Strogatz Graph (Degree of nodes = 8)

N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
131,072	524,288	8	0.22s	0.28s	0.78
262,144	1,048,576	8	0.42s	0.58s	0.72
524,288	2,097,152	8	0.88s	1.19s	0.74
1,048,576	4,194,304	8	1.80s	2.34s	0.77
2,097,152	8,388,608	8	3.59s	4.49s	0.80
4,194,304	16,777,216	8	7.31s	10.29s	0.71

Tables 1-3 and Figures 6-7 are the numerical results and visualizations of the Watts-Strogatz graph. The setting of Watts-Strogatz in this paper is a deterministic structure graph, as we let the degree of nodes be fixed number 4, 6, and 8, and only the weights on edges are assigned randomly from 1 to 10. The starting point of $O(M \cdot LogM)$ is rescaled to the starting point of Algorithm 2, to observe if Algorithm 2 meets the time complexity bound. Algorithm 1 and Algorithm 2 are plotted based on the values from the tables to compare the computational time. We can observe that Algorithms 1 and 2 are slightly above the theoretical bound of $O(M \cdot LogM)$, when the degree of nodes is 4. This is due to the overhead of algorithms, but they have better performance as the degree of nodes increases. When the degree of nodes is 4, the graph is relatively sparse, and it is difficult to claim that Algorithm 1 performs better than Algorithm 2. However, as the degree of nodes increases, Algorithm 1 saves more time. When the degree of nodes is 6,

the time ratio $\frac{Algo1}{Algo2}$ stabilizes around 0.82. When the degree of nodes is 8, the time ratio stabilizes around 0.75. Therefore, in a deterministic graph, we can claim that Algorithm 1 is expected to obtain better results on computational time as the graph and its average degree of nodes gets larger.

Table 4: Watts-Strogatz Graph GigaByte Memory Storage

N	M	D_{avg}	Algo 1	Algo 2	Memory Ratio
131,072	262,144	4	0.0237	0.0188	1.26
131,072	393,216	6	0.0503	0.0405	1.24
131,072	524,288	8	0.1034	0.0834	1.24
4,194,304	8,388,608	4	0.2106	0.1697	1.24
4,194,304	12,582,912	6	0.4223	0.3453	1.22
4,194,304	16,777,216	8	0.8495	0.6941	1.22

Next, we want to observe how these two algorithms vary on memory storage. Watts-Strogatz graph has fixed number of edges, so it is a good way to compare the memory storage difference. In Table 4, as the number of edges and average degree increase, Algorithm 2 has great advantages on memory storage on both GB number and ratio compare to Algorithm 1. Throughout all 3 tables, Algorithm 1 has around extra 24% memory stored compared to Algorithm 2.

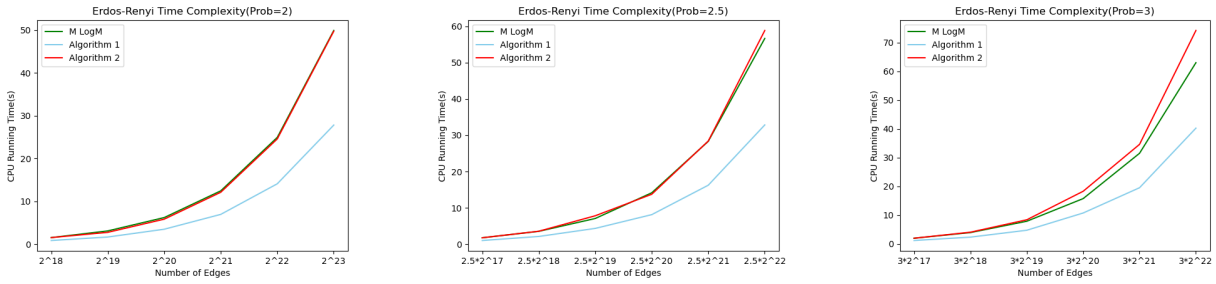

Figure 8: Erdős-Rényi Graph Probability=2 (left), Probability=2.5 (middle), Probability=3 (right)

Table 5: Erdős-Rényi Graph (Probability = $\frac{2\cdot(ln(M))}{M}$)

N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
25,806	261,786	20.28	0.08s	0.16s	0.50
48,585	524,926	21.60	0.13s	0.25s	0.52
91,763	1,048,862	22.86	0.29s	0.53s	0.54
173,811	2,097,921	24.14	0.59s	1.55s	0.38
330,076	4,195,049	25.42	1.16s	3.05s	0.38

Table 6: Erdős-Rényi Graph (Probability = $\frac{2.5\cdot(ln(M))}{M}$)

N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
25,806	327,924	25.42	0.07s	0.18s	0.39
48,585	655,398	26.98	0.16s	0.30s	0.53
91,763	1,310,775	28.56	0.31s	0.80s	0.39
173,811	2,623,118	30.62	0.61s	1.48s	0.41
330,076	5,242,429	31.76	1.24s	3.54s	0.35
628,322	10,485,256	33.38	32.82s	58.83s	0.56

Table 7: Erdős-Rényi Graph (Probability = $\frac{3\cdot(ln(M))}{M}$)

N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
25,806	393,057	30.46	1.18s	1.97s	0.60
48,585	786,773	32.38	2.35s	4.07s	0.58
91,763	1,572,307	34.26	4.75s	8.39s	0.57
173,811	3,147,013	36.22	10.75s	18.31s	0.59
330,076	6,288,240	38.10	19.54s	34.63s	0.56
628,322	12,580,367	40.04	40.26s	74.24s	0.54

While Algorithm 2 has performed well on a deterministic structure graph, we would also like to see its performance on a random graph such as Erdős-Rényi graph. Erdős-Rényi graph generates an edge between a pair of nodes based on the assigned probability. In our experiments, we use $\frac{2\ln(M)}{M}$, $\frac{2.5\ln(M)}{M}$ and $\frac{3\ln(M)}{M}$ respectively to ensure that the graphs are connected, and we assign the weights on edges randomly from 1 to 10. In Figure 8, Algorithm 2 stays close to the theoretical bound, while Algorithm 1 takes less computational time on all sizes of graphs and the computational time increases slower compared to Algorithm 2. In Tables 5-7, the time ratio of Algorithm 1 and 2 is stable around 0.4. We can claim from the results that as the average degree gets larger, Algorithm 1 shows better computational performance.

Table 8: Erdős-Rényi Graph Memory Storage(Probability = $\frac{2 \cdot (ln(M))}{M}$)

N	M	D_{avg}	Algo 1	Algo 2	Memory Ratio
25,806	261,786	20.28	0.0032	0.0021	1.52
48,585	524,926	21.60	0.0095	0.0063	1.50
91,763	1,048,862	22.86	0.0176	0.0138	1.28
173,811	2,097,921	24.14	0.0296	0.0243	1.22
330,076	4,195,049	25.42	0.0507	0.0404	1.25

Table 9: Erdős-Rényi Graph Memory Storage (Probability = $\frac{2.5 \cdot (ln(M))}{M}$)

N	M	D_{avg}	Algo 1	Algo 2	Memory Ratio
25,806	261,786	20.28	0.0039	0.0019	2.05
48,585	524,926	21.60	0.0104	0.0068	1.52
91,763	1,048,862	22.86	0.0233	0.0167	1.40
173,811	2,097,921	24.14	0.0259	0.0195	1.33
330,076	4,195,049	25.42	0.0516	0.0425	1.21

Table 10: Erdős-Rényi Graph Memory Storage (Probability = $\frac{3 \cdot (ln(M))}{M}$)

N	M	D_{avg}	Algo 1	Algo 2	Memory Ratio
25,806	261,786	20.28	0.0054	0.0037	1.46
48,585	524,926	21.60	0.0104	0.0070	1.49
91,763	1,048,862	22.86	0.0190	0.0131	1.45
173,811	2,097,921	24.14	0.0246	0.0200	1.23
330,076	4,195,049	25.42	0.0530	0.0436	1.22

Memory storage advantage of Algorithm 2 is also demonstrated in Erdős-Rényi graph. The memory ratio of $\frac{Algo1}{Algo2}$ is around 1.50.

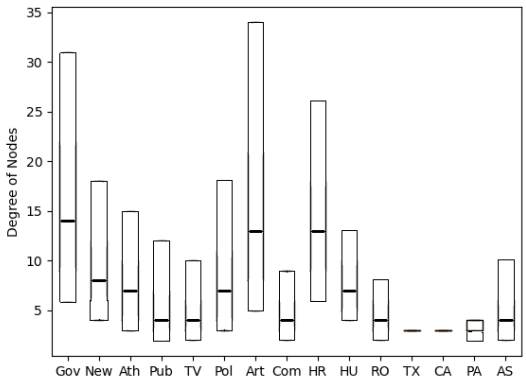

Figure 9: Real-World Graphs Boxplot

Table 11: gemsec-Facebook

Group	N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
Government	7,057	89,429	25.34	0.26s	0.45s	0.58
New Sites	27,917	205,964	14.76	0.66s	1.18s	0.56
Athletes	13,866	86,811	12.52	0.28s	0.44s	0.64
Public Figures	11,565	67,038	11.59	0.21s	0.36s	0.58
TV Shows	3,892	17,239	8.86	0.06s	0.10s	0.60
Politician	5,908	41,706	14.12	0.15s	0.28s	0.54
Artist	50,515	819,090	32.43	2.43s	4.37s	0.56
Company	14,113	52,126	7.39	0.19s	0.28s	0.68

Table 12: gemsec-Deezer

Group	N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
HR	54,573	498,202	18.26	1.66s	2.87s	0.58
HU	47,538	222,887	9.38	0.86s	1.26s	0.68
RO	41,773	125,826	6.02	0.53s	0.72s	0.74

Table 13: Road Network

Group	N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
TX	1,379,917	1,921,660	2.79	14.97s	15.04s	1.00
CA	1,965,206	2,766,607	2.82	21.49s	21.51s	1.00
PA	1,088,092	1,541,898	2.83	12.19s	11.85s	1.03

Table 14: Road Network Memory Usage

Group	N	M	D_{avg}	Algo 1	Algo 2	Memory Ratio
TX	1,379,917	1,921,660	2.79	0.2205	0.1815	1.21
CA	1,965,206	2,766,607	2.82	0.3669	0.3279	1.12
PA	1,088,092	1,541,898	2.83	0.2058	0.1656	1.24

Next, we test Algorithm 1 and 2 on real-world problems to see if the above advantages continue. We have selected two social network data sets i.e. gamesec-Facebook and gamesec-Deezer in [14] and one road network data set from Stanford Large Data Network Collection in [13].

Table 15: as-skitter

Group	N	M	D_{avg}	Algo 1	Algo 2	Time Ratio
as-skitter	1,696,415	11,095,298	6.54	37.14s	63.78s	0.58

Table 16: as-skitter Memory Storage

Group	N	M	D_{avg}	Algo 1	Algo 2	Memory Ratio
as-skitter	1,696,415	11,095,298	6.54	0.1822	0.1623	1.12

Since the degree distributions of these four real-world data sets are extremely right skewed, we showcase only 25% to 75% quantile of degree distribution in Figure 9 with the extreme values from both sides removed. Thus, it would be easier for us to observe the shape of the degree distributions. For all the numerical tests, Algorithms 1 and 2 are conducted on the original data sets, without removing any extreme values. In Tables 11-15, time ratio for gamesec-Facebook and gamesec-Deezer are relatively stable. While the nodes and edges vary widely, the average degree is high enough to ensure that Algorithm 1 outperforms Algorithm 2. When the average degree of graphs is low, such as in the road networks, the advantages of Algorithm 1 are not as pronounced. This can be explained by the types of these data sets. as-skitter is a large graph with high average degree, and the test shows the great advantage of Algorithm 1 on computational time. The first two tables are social networks that have clusters and a high degree of nodes that include more redundant edges for Algorithm 1 to remove and save time. However, in road network graphs, Algorithm 1 can only remove edges at cross-intersections and T-intersections. The property of the road network constrains the effectiveness of Algorithm 1. In Table 16, due to the high nodes and high density property of as-skitter, Algorithm 2 performs well on memory storage by only taking 85% if memory space comparing to Algorithm 1, which demonstrates its effectiveness on memory saving. Overall, on larger graphs with a higher average degree of nodes, we can conclude that Algorithm 1 performs increasingly

well on saving computational time, whereas Algorithm 2 gains more advantage in saving memory storage. The trade off between time and memory storage needs to be decided depending on the needs. The implementation of both algorithms is available on GitHub: $https://github.com/George-the-Ren/12_Optimization/tree/main$

5 Conclusion and Future Work

Finding an optimal path cover in any graph is an NP-complete problem as described in [15], yet there have been many works done to utilize approximation algorithms to estimate the optimal path cover. In this paper, we first review the $\frac{1}{2}$-Approximation Covering Algorithm. It is a greedy path cover approximating algorithm. It guarantees a path cover that obtains at least $\frac{1}{2}$ weight of the optimal path cover, with time complexity of $O(M \cdot LogM)$. We make a derivation of $\frac{1}{2}$-Approximation Covering Algorithm by not removing redundant edges as the algorithm progresses to save memory storage. We review the theoretical results from [1] in Section 2.3 and show in Section 3.3 that the $\frac{1}{2}$ theoretical bound and time complexity still hold for Algorithm 2. In Section 3.4, we prove the number of edges remained in the approximated path cover for any weighted graphs, which suggests how many edges are marked for removal by Algorithm 1. We then discuss our test results on both of the deterministic ring structured graphs and Erdős-Rényi random graphs in Section 4. From both the time cost and scaling on graphs with various sizes and densities, we observe that Algorithm 1 outperforms Algorithm 2 on both types of graphs regarding to computational time, and Algorithm 2 shows great advantage on acquiring much less memory storage, especially when the graph is large and its density is high. Then we show that Algorithm 1 performs effectively on most real-world networks with different structures, especially on high-degree graphs, such as the social networks, and Algorithm 2 is memory efficient on large and high density graph. Overall, there is a trade off of computational time and memory storage between two algorithms. Readers can choose which one to use depending on their needs.

As the modern problems tend to involve large data sets, the derived algorithm shows competitiveness in applications to the real-world data sets, especially in the form of high degree networks, such as social activities and e-commerce. In the future, we would like to apply it to other graph theory problems, such as cocomparability graphs, cographs, interval graphs, block graphs, and permutation graphs. Each structure may require some changes to the algorithm, but it is a general approach in many cases.

We also want to explore if there are other potential data structures, functions, and programming tips of Python that we can implement to optimize our current program. Additionally, we want to rewrite the current program in other languages, such as MATLAB or JAVA, which are more compatible with the hardware. Furthermore, we plan to take a step ahead to parallel optimization as we can separate one graph into multiple graphs by removing the local clustering, and solve all the graphs at once to generate the path cover which would bring the efficiency to an even higher level.

References

[1] S. Moran, I. Newman, Y. Wolfstahl, Approximation algorithms for covering a graph by vertex-disjoint paths of maximum total weight, *Networks*, **20** (1990), 055–064.

[2] S. Ehikioya, S. Lu, A Path Analysis Model for Effective E-Commerce Transactions, *Afr. J. Comp. & ICT*, **12** (2019), 055–071.

[3] A. Dwarakanath, A. Jankiti, Minimum Number of Test Paths for Prime Path and Other Structural Coverage Criteria, *the 26th ICTSS*, **8763** (2014), 063–079.

[4] R. Lin, S. Olariu, G. Pruesse, An optimal path cover algorithm for cographs, *COMPUT. MATH. APPL.*, **30** (1995), 075–083.

[5] D.G. Corneil, B. Dalton, M. Habib, LDFS-Based Certifying Algorithm for the Minimum Path Cover Problem on Cocomparability Graphs, *SIAM J. Comput.*, **42** (2013), 792–807.

[6] S.R. Arikati, C.P. Rangan, Linear algorithm for optimal path cover problem on interval graphs, *Inf. Process. Lett.*, **35** (1990), 149–153.

[7] R. Srikant, R. Sundaram, K.S. Singh, C.P. Rangan, Optimal path cover problem on block graphs and bipartite permutation graphs, *Theor. Comput. Sci*, **115** (1993), 351–357.

[8] E.W. Dijkstra, A node on two problem in connexion with graphs, *Numer Math (Heidelb)*, **1** (1959), 269–271.

[9] A. Bertolino, M. Marre, Automatic generation of path covers based on the control flow analysis of computer programs, *IEEE Trans. Softw. Eng.*, **20** (1994), 885–899.

[10] X. Hu, J. Lin, L.T. Zikatanov, An Adaptive Multigrid Method Based on Path Cover, *SIAM J. Sci. Comput.*, **41** (2019), 220–241.

[11] J. Tu, W. Zhou, "A primal–dual approximation algorithm for the vertex cover P_3 problem, *Theor. Comput. Sci.*, **412** (2011), 7044–7048.

[12] Y. Zhang, Y. Shi, Z. Zhang, Approximation algorithm for the minimum weight connected k-subgraph cover problem, *Theor. Comput. Sci.*, **535** (2014), 054–058.

[13] J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney, Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters, *Internet Mathematics*, **6** (2009), 029–123.

[14] B. Rozemberczki, R. Davies, R. Sarkar, C. Sutton, GEMSEC: Graph Embedding with Self Clustering, *ASONAM*, (2019), 065–072.

[15] M. Garey, D. Johnson, R.E. Tarjan, The planar Hamiltonian circuit problem is NP-complete, *SIAM J. Comput.*, **5** (1976), 707–714.