Effective Preprocessing and Dynamic Programming Algorithms using Zero-suppressed Binary Decision Diagrams

Keita Maeda*‡ Ryuma Noma* Toshiki Saitoh* Takumi Shiota*†
Shinryu Tachibana* Naoya Taguch*‡ Soma Takao*

October 16, 2024

1 Introduction

Finding the optimal route from a starting point to a destination is essential in logistics, shuttle services, and navigation systems. Providing alternative routes that adapt to conditions like traffic jams or road closures is highly beneficial. Counting the number of routes from a starting point to a destination can be seen as finding all possible paths in a graph G. However, in transportation networks, the number of paths in the graph G is typically huge, and computing these paths is a #P-complete problem [Val79]. Therefore, efficient methods are needed to count these paths. In this paper, we address the following two problems:

Problem 1 (Undirected Graph) Given an undirected graph $G = (V, E)$, a pair of terminals (s, t), and an integer ℓ, count the number of $s - t$ paths with length at most ℓ.

Problem 2 (Directed Graph) Given a directed graph $G = (V, E)$, a pair of terminals (s, t), and an integer ℓ, count the number of $s - t$ paths with length at most ℓ.

2 Preliminaries

2.1 Graph

Let $G = (V, E)$ be a *graph*, where V is a set of vertices. If $E = \{(u, v) \mid u, v \in V, u \neq v\}$ is a set of ordered pairs, then the edges are called *directed edges*, and G is called a *directed graph*. If $E = \{\{u, v\} \mid u, v \in V, u \neq v\}$ is a set of unordered pairs, then the edges are called *undirected edges*, and G is called an *undirected graph*. The graph G is called a *weighted graph* if a *weight function* $w : E \to \mathbb{N}$ assigns a *weight* to each edge. A *path* is a sequence of distinct vertices $(v_1, \ldots, v_k)$ such that, for all $i \in \{1, \ldots, k-1\}$, either $(v_i, v_{i+1}) \in E$ (for directed graphs) or $\{v_i, v_{i+1}\} \in E$ (for undirected graphs). The vertices v_1 and v_k are called the *endpoints* of the path, and in both undirected and directed graphs, we call the path a $v_1 - v_k$ *path*. For the path P, the sum of the weights of edges $\sum_{i \in \{1,\ldots,k\}} w(v_i, v_{i+1})$ is the *length* of the path. A path is a *Hamiltonian path* if it visits every vertex in G exactly once. A graph is *connected* if there exists a path between any two vertices in G. A *cycle* is a path where the starting vertex and the ending vertex are the same. If a graph is both connected and acyclic, it is called a *tree*. A *directed acyclic graph* (DAG) is a directed graph that contains no cycles. For an undirected graph, the *neighbor set* of a vertex v_i is defined as $N(v_i) = \{v_j \mid \{v_i, v_j\} \in E\}$ and the *degree* of v_i is $\deg(v_i) = |N(v_i)|$. For a directed graph, the *out-neighbor set* and the *in-neighbor set* of a vertex v_i is $N_{\text{out}}(v_i) = \{v_j \mid (v_i, v_j) \in E\}$ and $N_{\text{in}}(v_i) = \{v_j \mid (v_j, v_i) \in E\}$, respectively. The *out-degree* and *in-degree* of a vertex v_i is $\deg_{\text{out}}(v_i) = |N_{\text{out}}(v_i)|$ and $\deg_{\text{in}}(v_i) = |N_{\text{in}}(v_i)|$, respectively. For a subset of vertices $W \subseteq V$ in the graph G, the graph $G[W] = (W, \{e \in E \mid e = (p, q) \text{ or } e = \{p, q\}, \text{ with } p, q \in W\})$ is called the *subgraph induced by* W. If removing a vertex disconnects the graph, that vertex is called a *cut vertex*. A *biconnected component* is a maximal subgraph that contains no cut vertices. A *block-cut tree* (BC-tree) is a tree formed by contracting each biconnected component into a single vertex and connecting these contracted components with cut vertices in an alternating manner. A directed graph is called *strongly connected* if, for any two vertices u and v, there is a path from u to v and a path from v to u. A *strongly connected component* (SCC) is a maximal subgraph that is strongly connected. The *decomposition of strongly connected components* (SCC decomposition) is the process of partitioning a directed graph into a collection of strongly connected components that do not overlap. After decomposing the graph into SCCs, contracting each SCC into a single vertex results in a DAG.

*Kyushu Institute of Technology
†Research Fellow of Japan Society for the Promotion of Science
‡Corresponding author ({taguchi.naoya675, maeda.keita600}@mail.kyutech.jp)

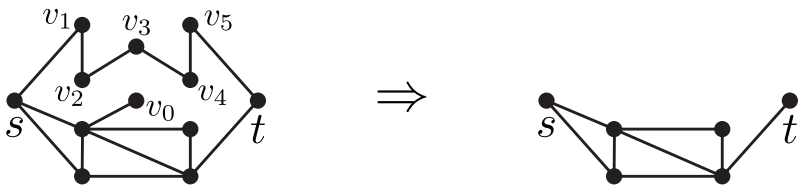

Figure 1: Example of vertex deletion for $\ell = 5$. Vertex v_0 can be removed because $\deg(v_0) = 1$. The path length through $s, v_1, v_2, \cdots, v_5, t$ is 6, exceeding ℓ, so v_1 to v_5 can be deleted.

2.2 Flow networks

For a directed graph $G = (V, E)$, (G, c, s, t) is a *flow network* where $c : E \to \mathcal{R}^+, s, t \in V$. A *flow* from the *source* to *sink* is defined as a function $f : E \to \mathcal{R}^+$ satisfying the *capacity* and *flow conservation* constraints. For a detailed definition of flow networks, refer to [CLRS09]. The *max-flow min-cut theorem* states that the value of the maximum flow in a network is equal to the total capacity of the minimum cut [FF56]. To find the maximum flow, a common technique is to use a *residual network*. A residual network represents the remaining capacity on each edge for additional flow and shows how much additional flow can be sent through the network. The *residual capacity* $c_f(u, v)$ is defined as follows:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v), & \text{if } (u,v) \in E, \\ f(v,u), & \text{if } (v,u) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

2.3 Zero-suppressed binary decision diagrams

A zero-suppressed binary decision diagram, or ZDD for short, is a data structure for a family of sets. For a detailed definition of ZDDs, refer to [Knu11]. The important features of a ZDD are that it can compactly represent a family of sets, and there are many efficient algebraic operations available on these families. After constructing a ZDD, we can efficiently count the number of objects represented by the ZDD. A ZDD is a directed acyclic graph consisting of a specific node, called the *root node*, and two terminal nodes, called the 0-*terminal* and 1-*terminal*, which have no outgoing edges. Each non-terminal node has two outgoing edges, called the *0-edge* and *1-edge*, with labels assigned to represent elements of the set. Thus, the method of constructing the ZDD is an important consideration. One approach is a frontier-based search [KIIM17], which constructs the ZDD directly by sharing and pruning nodes based on a state. Another approach involves repeatedly applying ZDD operations [Min93].

2.4 Path decompositions

Let $G = (V, E)$ be a graph and $P = (X_1, X_2, \ldots, X_r)$ be a sequence of vertex subsets, called bags, where each $X_i \subseteq V$ for $i \in \{1, 2, \ldots, r\}$. The sequence P is a *path decomposition* of G if it satisfies the following three conditions:

1. Each vertex $v \in V$ appears in at least one bag X_i in P.
2. For every edge $\{u, v\} \in E$, there exists a bag X_i in P that contains both u and v.
3. For every vertex $v \in V$, if v is contained in two bags X_i and X_j with $i \leq j$, then v must also be contained in every bag X_k for all $k \in \{i, \ldots, j\}$.

The *pathwidth* of a graph G is the minimum of the maximum bag size minus one among all path decompositions of G.

3 Preprocessing

In a given graph G with terminals s and t, and a positive integer ℓ, edges and vertices that are not contained any path of length ℓ can be removed, as shown in Figure 1. We propose some preprocessing to remove such vertices and edges in this section. In Section 3.1, we first describe preprocessing methods based on simple ideas. Next, in Section 3.2, we present preprocessing methods based on biconnected components decomposition. Finally, in Section 3.3, we describe preprocessing methods for directed graphs based on flow networks.

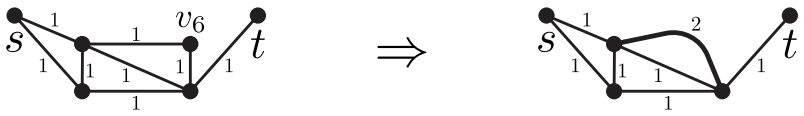

Figure 2: Example of vertex contraction. Vertex v_6 can be contracted because $\deg(v_6) = 2$. The edges connected to v_6 each have a weight of 1, so they are combined into a new edge with a weight of 2.

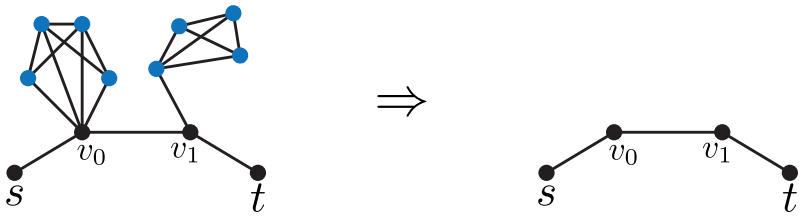

Figure 3: An example of vertices that cannot be removed using the method from Section 3.1. The blue vertices on the left do not contribute to the $s-t$ path counting, so they can be removed (right).

3.1 Simple preprocessing methods

In this section, we describe three preprocessing methods that can be applied to both directed and undirected graphs.

Method 1 (Removing vertices with degree 1)
If a vertex v (where $v \neq s$ and $v \neq t$) has a degree of 1, v can be removed from G. (This applies to v_0 in Figure 1.)

Method 2 (Removing vertices based on path length)
Let ℓ_{uv} be the shortest path length between vertices $u(\in V)$ and $v(\in V)$. If v satisfies $\ell_{sv} + \ell_{vt} > \ell$, remove v from G, as any $s-t$ path that includes v will have a length greater than ℓ. (This applies to v_1 through v_5 in Figure 1.)

Method 3 (Contracting vertices with degree 2)
Let the degree of a vertex v (where $v \neq s$ and $v \neq t$) be 2, and x, y be the neighbors of v. We remove v and its incident edges and add an edge x, y with weight $w(x, v) + w(v, y)$. (See the example in Figure 2.)

An unweighted graph can be seen as a weighted graph by assigning weights 1 to all edges. Using Method 3, we consider the graph as a weighted multiple graph.

3.2 Preprocessing using biconnected components decomposition

We here present a preprocessing that removes some vertices of components by extending Method 1. The path passing through the blue vertices in Figure 3 cannot form an $s-t$ path without passing through vertices v_0 and v_1 twice, so these vertices can be removed. In this section, we describe a method based on [MKRS95] to remove such vertices that do not contribute to the $s-t$ path counting by using the BC-tree. The method is as follows:

Step 1. Apply biconnected component decomposition and construct the BC-tree
Decompose the undirected graph $G = (V, E)$ into biconnected components and construct the BC-tree T_B.

Step 2. Identification of biconnected components on the path
In the BC-tree T_B, let B_s be the biconnected component containing s, and B_t be the one containing t. We use Breadth-First Search (BFS) to search for the biconnected components on the path from B_s to B_t in T_B. Each component is denoted as B_i $(1 \leq i \leq k)$, where k is the number of components.

Step 3. Identification of vertices that do not contribute to $s-t$ path counting
Each biconnected component contains several vertices. Let $V_{\bar{B}}$ be the set of vertices in G that are not part of B_s, B_t, or any of the B_i $(1 \leq i \leq k)$. Here, the set of vertices $V_{\bar{B}}$ does not contribute to the $s-t$ path counting in G.

Step 4. Deletion of vertices that do not contribute to $s-t$ path counting
Delete the vertices in $V_{\bar{B}}$ obtained in Step 3 and their adjacent edges from the graph G.

For directed graphs, we can use this preprocessing method by treating the edges as undirected.

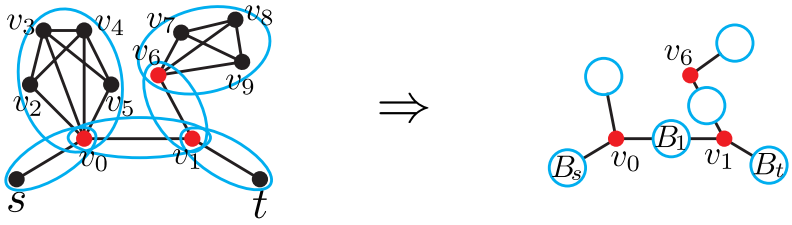

Figure 4: An example of biconnected component decomposition (left) and the resulting BC-tree (right). The red circles represent the cut vertices, and the blue circles represent the biconnected components. The biconnected component B_s contains s and v_0, B_1 contains v_0 and v_1, and B_t contains v_1 and t.

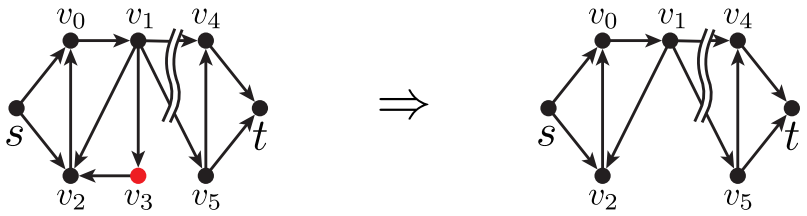

Figure 5: An example of a vertex that cannot be removed using the methods from Sections 3.1 and 3.2. The red vertex on the left does not contribute to the $s-t$ path counting, so it can be removed.

3.3 Removing non-contributed vertices

We present a preprocessing that removes vertices not contained in any $s-t$ paths for directed graphs. For a directed graph G, a vertex v is $s-t$ *cut vertex* if it contains any $s-t$ paths in G, that is, there is no $s-t$ path if we remove the cut vertex. For a vertex v, there is no $v-t$ paths after removing vertices $s-v$ cut vertices, the vertex v is *non-contributed*. This implies that any walk from s to t through the non-contributed vertex v contains a cut vertex at least twice. Thus, there is no $s-t$ path through v. For example, in Figure 5, there is no $s-t$ path through the red vertex v_3 because there are $s-v_3$ and v_3-t cut vertices v_0 and v_1. Therefore, the obtained graph after removing v_3 is equivalent.

Here, we describe a method to find all non-contributed vertices in directed graphs $G=(V,E)$ and the vertices s and t using the flow networks. Let v be a vertex in a directed graph $G=(V,E)$. To check v non-contributed, we construct the flow network (G,s,v,c), where the function c is a vertex capacity and each vertex capacity is one. It is well-known that we can compute the maximum flow of a vertex capacity flow network by transforming an edge capacity flow network and then using Ford-Fulkerson's algorithm. By computing the maximum flow on (G,s,v,c), we can find a set of $s-v$ cut-vertices C_s using the residual network. After removing C_s vertices and their incident edges, we check whether there exists a $v-t$ path in $G[V\setminus C_s]$.

4 Paths counting algorithms

We propose three algorithms using ZDDs for the path counting problems. The first algorithm is based on the frontier-based search. The second is a dynamic programming approach for computing Hamiltonian paths. The third algorithm combines the dynamic programming for strongly connected components (SCC) with the second algorithm for directed graphs.

4.1 Frontier-based search algorithm

This section presents an algorithm based on frontier-based search, which applies dynamic programming to *path decompositions*. To perform frontier-based search efficiently, it is important to find a "good" path decomposition of the input graph G. However, finding an optimal path decomposition is known to be NP-hard [ACP87]. Inoue and Minato proposed a heuristic approach using beam search [IM16] to find a "good" path decomposition. Here, we introduce a method for computing the path decomposition of G by applying this heuristic with a new evaluation function and adjusting the beam width. Once the path decomposition is obtained, we explain how to construct a ZDD that represents all $s-t$ paths in G using frontier-based search.

4.1.1 Path decomposition using beam search

To efficiently perform frontier-based search, we consider "good" path decompositions using a new heuristic approach based on beam search [IM16]. We achieved this through the following process.

It is known that a vertex ordering corresponds to a path decomposition [Kin92]. Thus, we can see that the problem to find a "good" path decomposition is equal to finding a "good" vertex ordering. First, we defined an evaluation function, $eval(\pi)$, to measure the efficiency of the frontier-based search for a given vertex ordering $\pi = (v_1, v_2, \ldots, v_n)$. The time complexity of the frontier-based search is $O\left(\sum_{i=1}^{n} f(F_i)\right)$, where F_i represents the size of the frontier at i-th vertex. Since $f(F_i)$ can grow exponentially with the size of the frontier, it is important to minimize not only the maximum frontier size but also the cumulative size of the frontiers throughout the search. Thus, we use the following evaluation function:

$$eval(\pi) = \log\left(\sum_{k=1}^{n} e^{F_k}\right),$$

where smaller values of $eval(\pi)$ indicate more efficient searches. By employing beam search with this evaluation function, we find an efficient vertex ordering for the path decomposition.

Next, we describe the process of generating a path decomposition from the optimal vertex sequence π obtained through beam search and our introduced evaluation function. The path decomposition is constructed as follows:

Step 1. Initialize an empty set S and an empty sequence L.

Step 2. For each vertex v_i in π:

1. Add v_i to S.
2. Include any vertices v_j such that $j > i$ and $(v_i, v_j) \in E$ in S.
3. If new vertices were added, append S to L.
4. Remove v_i from S and append the updated S to L.

This process results in a sequence L that represents the path decomposition of the graph.

4.1.2 ZDD constraction for $s-t$ paths

We construct a ZDD representing all $s-t$ paths in G by *frontier-based search* [KIIM17]. The frontier-based search constructs a ZDD representing all subgraphs satisfying some conditions in a top-down manner. The basic idea of our algorithm is the same as that in [KIIM17]. We apply two constraints, degree and connectivity constraints, to obtain $s-t$ paths. The degree constraint for $s-t$ paths is that both degrees of terminals s and t are one, and the degrees of the other vertices are zero or two. The connectivity constraint is that the subgraph is connected. It is well known that any subgraph of G satisfies the two conditions if and only if it is an $s-t$ path.

To apply these constraints, we employ a *subsetting* method by TdZdd[1] which is a C++ library to construct a ZDD in a top-down manner. Given a ZDD Z and a constraint C, the subsetting method obtains a new ZDD by extracting the subgraphs satisfying the condition C from the subgraphs represented by Z. Our algorithm executes the subsetting four times to adapt the problems with length constraints and to compute the degree constraint efficiently. We implement the following subsetting steps.

Undirected graph

Step 1. Length constraint: The number of edges is at most ℓ. We only maintain the number of edges as a state and prune the search if the number of selected edges exceeds ℓ.

Step 2. Relaxation of the degree constraint: The degrees of terminals are odd, and those of the other vertices are even. We maintain a bit vector as a state for a bag of the path decomposition, and it represents that the degrees of the vertices in the bag are odd or even. We prune the search if a terminal's degree is even or a vertex's degree except for terminals is odd.

Step 3. Degree constraint: degrees of terminals are one, and those of the other vertices are zero or two. An integer array represents the degrees of the vertices in a bag. We prune the search if the degree becomes larger than three.

Step 4. Connectivity constraint: The subgraphs are connected. We maintain a set of paths by a mate array [KIIM17]. We prune the search if the subgraph contains a cycle.

[1] `https://github.com/kunisura/TdZdd`

Directed graph

Step 1. Length constraint: This is the same as for undirected graphs.

Step 2. Relaxation of the degree constraint: The sum of the in-degree and out-degree at the terminal vertices is odd, while for all other vertices, the sum of the in-degree and out-degree is even. The rest is the same as in the undirected graph.

Step 3. Degree constraint: The in-degree of terminal s is 0 and its out-degree is 1, while the in-degree of terminal t is 1 and its out-degree is 0. For all other vertices, both the in-degree and out-degree are 1. The rest is the same as in the undirected graph.

Step 4. Connectivity constraint: This is the same as for undirected graphs.

4.2 Counting Hamiltonian paths using ZDDs

In this section, we explain a dynamic programming algorithm for finding Hamiltonian paths, known as the algorithm for the traveling salesperson problem. We extend this algorithm to the counting problems and implement it by ZDDs.

For a given graph $G = (V, E)$, a subset X of vertices includes vertices s and t. We define a function $f(s, t, X)$ as the number of Hamiltonian paths from s to t in $G[X]$. The function $f(s, t, X)$ can be computed by the following recursive formula (for more details, see [MFI+24]):

Undirected graph

$$f(s,t,X) = \begin{cases} \left(\sum_{\substack{v \in N(t), v \in X, \\ X' = X \setminus \{t\}}} f(s,v,X') \right) + e(s,t,|X|) & s,t \in X \text{ and } |X| > 1 \\ 1 & s = t \\ 0 & \text{Otherwise} \end{cases}$$

where

$$e(s,t,\ell) = \begin{cases} \#\{\{s,t\},\ell\} & \{s,t\} \in E(G) \text{ and } w(\{s,t\}) = \ell \\ 0 & \{s,t\} \notin E(G) \end{cases}$$

Directed graph

$$f(s,t,X) = \begin{cases} \left(\sum_{\substack{v \in N_{\text{in}}(t), v \in X, \\ X' = X \setminus \{t\}}} f(s,v,X') \right) + e(s,t,|X|) & s,t \in X \text{ and } |X| > 1 \\ 1 & s = t \\ 0 & \text{Otherwise} \end{cases}$$

where

$$e(s,t,\ell) = \begin{cases} \#\{(s,t),\ell\} & (s,t) \in E(G) \text{ and } w((s,t)) = \ell \\ 0 & (s,t) \notin E(G) \end{cases}$$

Here, we treat an unweighted graph as a weighted graph by assigning a weight of 1 to each edge. A path of length w can then be contracted into a single edge with weight w. This contraction allows multiple paths with the same endpoints and the same length ℓ to be represented as a single edge. In the formulas, $\#\{\{s,t\},\ell\}$ and $\#\{(s,t),\ell\}$ indicate the number of distinct paths with the same endpoints and the same length.

The total number of $s-t$ paths of length ℓ is given by:

$$\sum_{s,t \in X,\ X \subseteq V \text{ and } |X|-1 \leq \ell} f(s,t,X).$$

This recursive function works by first extracting sets from the ZDDs that represent $s-v$ paths of length $\ell - w(\{v,t\})$, excluding vertex t. By adding vertex t to these sets, we construct ZDDs that represent the $s-t$ paths of length ℓ.

4.3 Path counting using decomposution of SCCs

In this section, we present an algorithm for directed graphs that combines the decomposition of SCC with the algorithm described in Section 4.2. This approach uses the structure of SCCs to count $s-t$ paths more efficiently. The algorithm follows these three steps:

Step 1. Decomposition of SCCs:
First, we perform SCC decomposition on the given directed graph G. In this process, the graph is partitioned into SCCs, where each SCC is a maximal subgraph that is strongly connected. After decomposition, we contract each SCC into a single vertex, resulting in a DAG. This step is done using depth-first search (DFS), which takes $O(|V|+|E|)$ time.

Step 2. Counting paths in each SCC:
For each SCC C_i in the decomposition, we apply the algorithm from Section 4.2 (for directed graph) to count the number of $s-t$ paths within the SCC. Since each SCC is strongly connected, the paths are counted independently for each component. The result of this step gives the number of paths between certain vertices in each SCC.

Step 3. Combining results using dynamic programming:
Once the paths in each SCC are counted, we use dynamic programming (DP) to combine the results. Since the contracted SCCs form a DAG, we can compute the total number of $s-t$ paths in the original graph by traversing the DAG. We multiply the number of paths from s to the start vertices of each SCC, from the start vertices to the end vertices within each SCC, and from the end vertices of each SCC to t. In the DP approach, paths between different SCCs do not overlap, and the total count is obtained by summing up the results.

By following these steps, the algorithm efficiently counts $s-t$ paths in directed graphs using the structure provided by SCC decomposition and dynamic programming.

5 Experiments

In order to ascertain the effectiveness of the preprocessing described in Section 3, as well as the three algorithms described in Section 4, we apply them to benchmark instances provided by AFSA ICGCA 2024[2]. The public benchmark includes 50 instances for Problem 1 (undirected graph) and 50 instances for Problem 2 (directed graph). We denote the algorithms proposed in Section 4.1 as **FBS**, in Section 4.2 as **HAMDP**, and in Section 4.3 as **SCC**. We implement the algorithms in C++ and use the TdZdd[1] library for **FBS** and SAPPOROBDD[3] for **HAMDP** and **SCC**. We run the programs on a machine with Linux CentOS 7.9, an Intel Xeon CPU E5-2643 v4 (3.40 GHz, 24 cores), and 256 GB of memory. To match the competition evaluation environment, we set a timeout of 10 minutes per instance. We apply **FBS** to both undirected and directed graphs. However, since **SCC** consistently performs faster than **HAMDP** on directed graphs, we use **SCC** for directed graphs and **HAMDP** for undirected graphs in our experiments. For **FBS**, we compute a path decomposition of the input graph within 10 seconds.

Table 1 and 2 show the results of preprocessing applied to Problem 1 (undirected graph) and Problem 2 (directed graph), respectively. In the case of undirected graphs, we reduce a large number of vertices and edges in instances starting with "stanford-Oregon" (see Figure 6) and "stanford-as". For directed graphs, we reduce a number of vertices and edges in instances such as "cit-HepPh-30-raw" (see Figure 7) and "p2p-Gnutella24-10-118-raw". Additionally, for Problem 2, there are five instances where the algorithm presented in Section 3.3 does not finish within 10 minutes.

Table 3 shows experimental results for Problem 1 (undirected graph). **FBS** and **HAMDP** solved 35 and 5 instances from 50 instances, respectively. For undirected graphs, **FBS** solves instances faster than **HAMDP**. Table 4 shows experimental results for Problem 1 (directed graph). **FBS** and **SCC** solved 23 and 13 instances from 50 instances, respectively. Similar to undirected graphs, in most cases, **FBS** solves instances faster than **SCC**. However, for the instance "airlines-migration-airtraffic-airlines-8" (Figure 8), **SCC** solves the problem faster than **FBS**. We consider this is because the pathwidth (PW) is large at 44, while the value of ℓ is small at 7.

Based on these results, we switch the algorithm depending on the pathwidth and length: for instances with a pathwidth of 20 or less and a length greater than or equal to 10, we use **FBS**; otherwise, we use **HAMDP** or **SCC**.

[2] `https://afsa.jp/icgca2024/`
[3] `https://github.com/Shin-ichi-Minato/SAPPOROBDD`

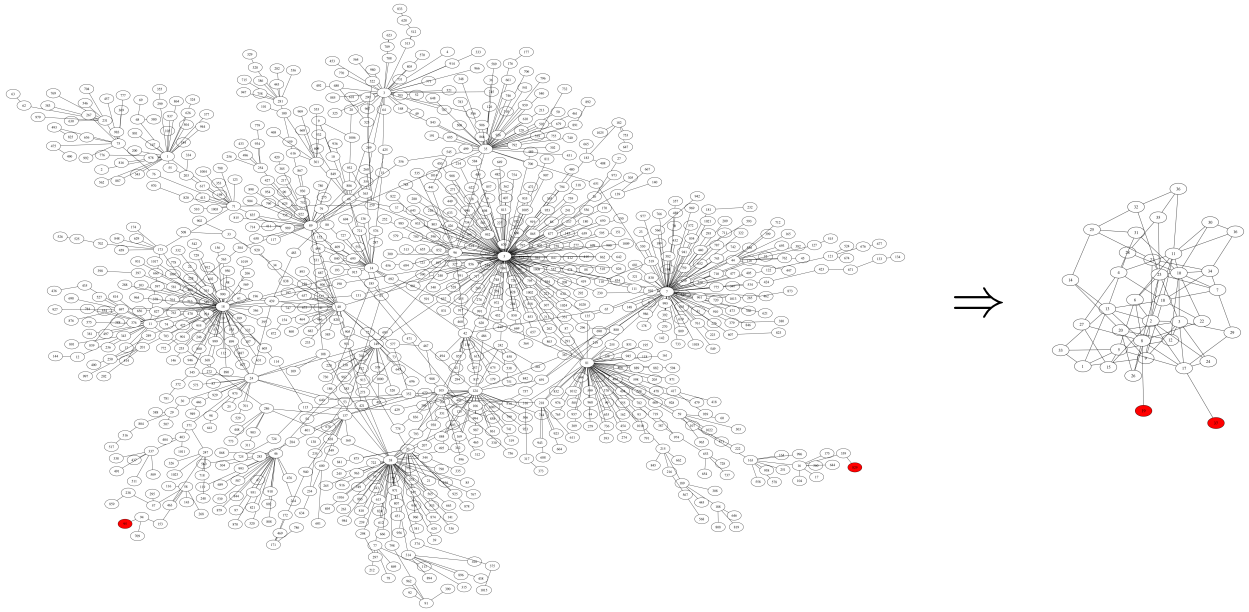

Figure 6: If we apply preprocessing to the undirected graph stanford-Oregon-1-1085-384 (left), we obtain the smaller graph (right). The red vertices represent s and t.

References

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[IM16] Yuma Inoue and Shin-ichi Minato. Acceleration of zdd construction for subgraph enumeration via path-width optimization. *TCS-TR-A-16-80. Hokkaido University*, 2016.

[KIIM17] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-Based Search for Enumerating All Constrained Subgraphs with Compressed Representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(9):1773–1784, 2017.

[Kin92] Nancy G. Kinnersley. The vertex separation number of a graph equals its path-width. *Information Processing Letters*, 42(6):345–350, 1992.

[Knu11] Donald E Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Pearson Education India, 2011.

[MFI+24] Keita Maeda, Yuta Fujioka, Takumi Iwasaki, Takumi Shiota, and Toshiki Saitoh. Divide-and-Conquer Algorithms for Counting Paths using Zero-Suppressed Binary Decision Diagrams. In *Proceedings of the 24th Korea–Japan Joint Workshop on Algorithms and Computation*, 2024.

[Min93] Shin-ichi Minato. Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*, pages 272–277. ACM Press, 1993.

[MKRS95] K. Madhukar, D.Pavan Kumar, C.Pandu Rangan, and R. Sundar. Systematic design of an algorithm for biconnected components. *Science of Computer Programming*, 25(1):63–77, 1995.

[Val79] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

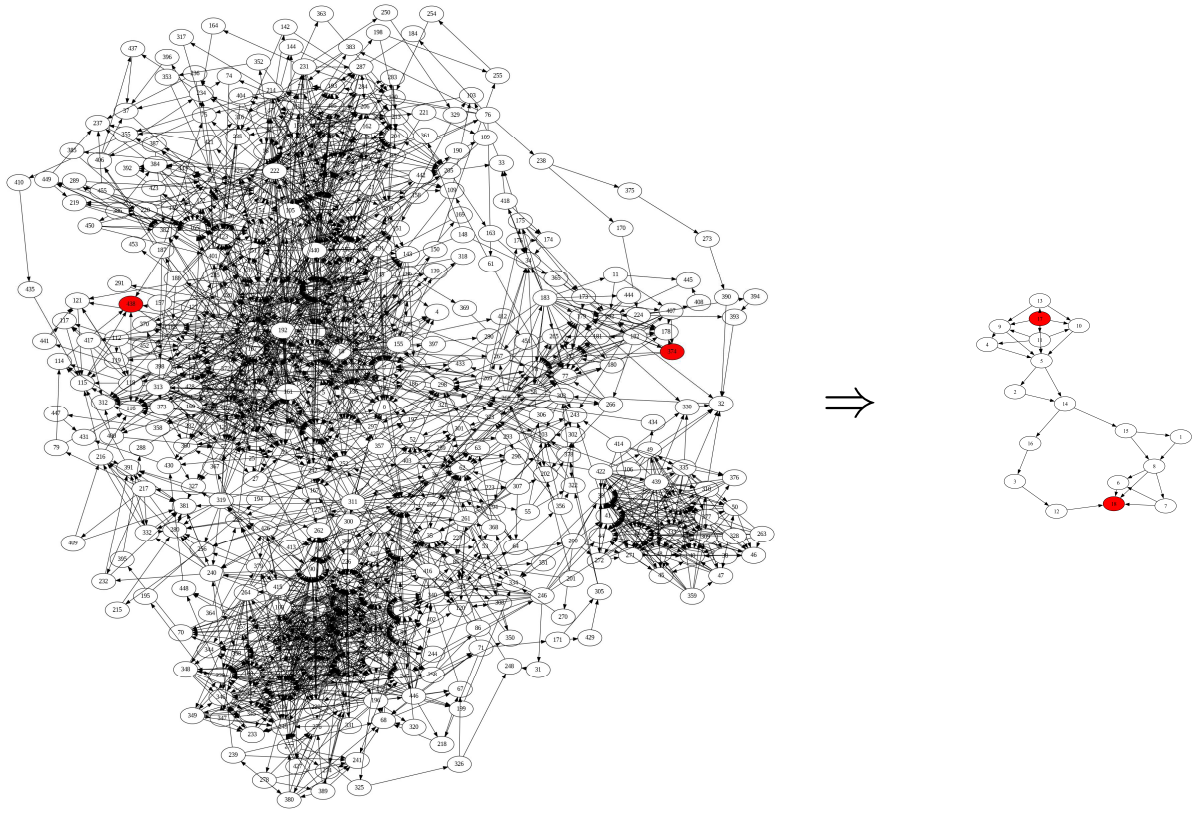

Figure 7: If we apply preprocessing to the directed graph cit-HepPh-30-raw (left), we obtain the smaller graph (right).

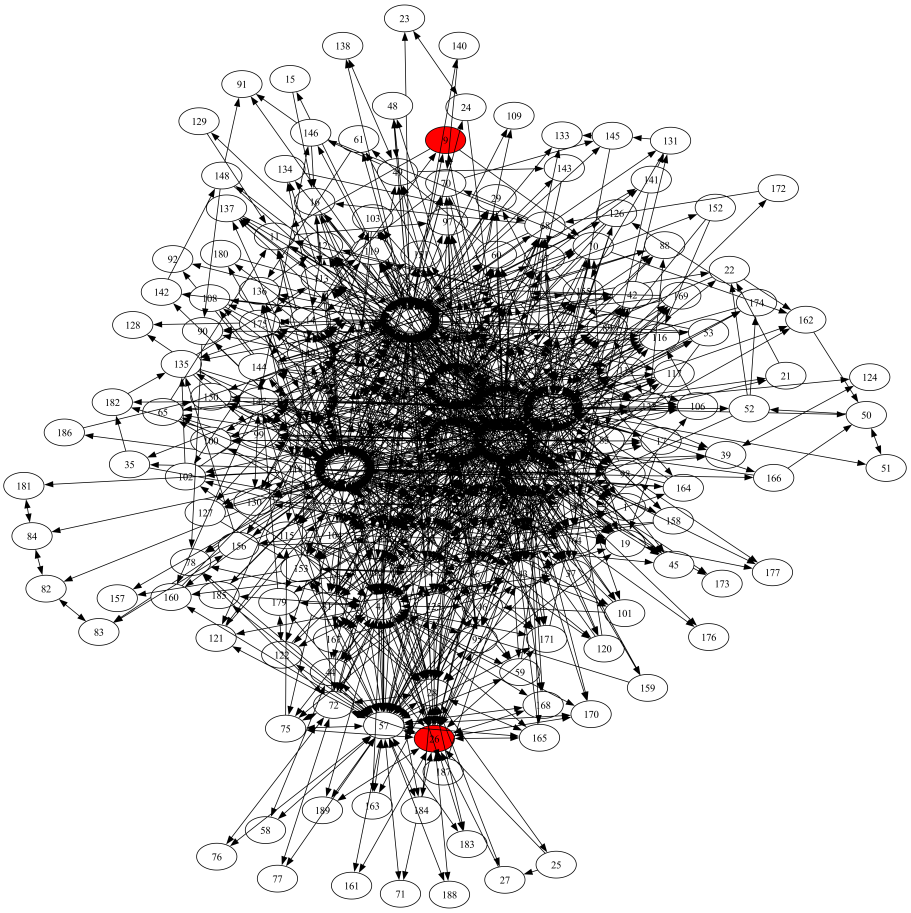

Figure 8: airlines-migration-airtraffic-airlines-8 (after preprocessing)

	Original Graph			Preprocessed graph		
	$\|V\|$	$\|E\|$	$\|\ell\|$	$\|V\|$	$\|E\|$	$\|\ell\|$
matpower-case145-105	145	422	105	101	376	102
matpower-case145-34	145	422	34	101	376	31
matpower-case1888rte-1420	1888	2308	1420	360	736	778
matpower-case1888rte-154	1888	2308	154	360	736	147
matpower-case1888rte-75	1888	2308	75	360	736	68
matpower-case300-123	300	409	123	124	228	120
matpower-case89pegase-37	89	206	37	49	166	34
matpower-case_ACTIVSg200-140	200	245	140	58	103	128
matpower-case_ACTIVSg2000-1300	2000	2667	1300	896	1563	1299
matpower-case_ACTIVSg2000-901	2000	2667	901	896	1563	900
matpower-case_ACTIVSg500-127	500	584	127	96	180	125
matpower-case_ACTIVSg500-392	500	584	392	96	180	239
rocketfuel-k_o-133	318	758	133	132	572	129
rocketfuel-k_o-136	172	381	136	113	320	135
rocketfuel-k_o-375	631	2078	375	424	1871	372
rocketfuel-k_o-550	960	2821	550	537	2393	548
rocketfuel-k_o-735	960	2821	735	537	2393	678
rocketfuel-k_o-99	240	404	99	83	245	97
rsndlib–36	100	154	36	57	111	34
rsndlib–62	98	152	62	57	111	61
stanford-Oregon-1-1085-384	1025	1085	384	35	94	82
stanford-Oregon-1-1557-543	1437	1557	543	64	183	166
stanford-Oregon-1-1557-629	1437	1557	629	64	183	166
stanford-Oregon-1-1557-839	1437	1557	839	64	183	166
stanford-as-733-1003-776	916	1003	776	34	97	96
stanford-as-733-1492-1156	1306	1492	1156	57	183	170
stanford-as-733-1492-140	1306	1492	140	57	183	130
stanford-as-733-610-465	580	610	465	14	36	54
stanford-ca-CondMat-423-206	417	423	206	12	18	55
stanford-ca-GrQc-357-132	266	357	132	48	137	72
stanford-ca-GrQc-695-334	496	695	334	81	222	142
stanford-ca-GrQc-695-453	496	695	453	81	222	142
stanford-ca-GrQc-695-483	496	695	483	81	222	142
stanford-ego-Facebook-1166-162	821	1166	162	309	653	155
stanford-ego-Facebook-235-167	203	235	167	42	73	92
stanford-ego-Facebook-235-177	203	235	177	42	73	92
stanford-ego-Facebook-425-131	336	425	131	99	188	128
stanford-ego-Facebook-425-159	336	425	159	99	188	156
stanford-feather-lastfm-social-234-114	227	234	114	10	15	25
stanford-feather-lastfm-social-508-178	450	508	178	55	113	129
stanford-musae-wiki_chameleon-245-92	200	245	92	46	91	85
stanford-musae-wiki_chameleon-570-325	482	570	325	65	153	157
stanford-musae-wiki_crocodile-1311-1200	1221	1311	1200	72	160	225
stanford-musae-wiki_crocodile-1311-636	1221	1311	636	72	160	225
stanford-musae-wiki_squirrel-1253-485	941	1253	485	254	565	483
stanford-musae-wiki_squirrel-759-252	672	759	252	104	191	242
topologyzoo-k_o-12	20	26	12	10	16	12
topologyzoo-k_o-132	145	186	132	55	94	131
topologyzoo-k_o-169	754	895	169	202	341	169
topologyzoo-k_o-366	754	895	366	202	341	366

Table 1: Results of preprocessing for Problem 1

	Original Graph			Preprocessed graph		
	$\|V\|$	$\|E\|$	$\|\ell\|$	$\|V\|$	$\|E\|$	$\|\ell\|$
airlines-migration-airrtraffic-airlines-8	235	2101	8	189	1998	7
cit-HepPh-10-30-raw	3454	22955	30	21	28	21
cit-HepPh-30-raw	10363	139155	120	18	34	18
email-EuAll-10-20-raw	26521	121365	20	6080	72461	16
email-EuAll-30-raw	79564	221146	45	12058	101733	42
matpower-case145-34	145	822	34	130	788	33
matpower-case1888rte-1420	1888	4500	1420	777	2243	777
matpower-case_ACTIVSg200-140	200	477	140	127	334	127
north-g.100.0	100	191	25	29	54	22
north-g.12.79	12	26	7	7	13	7
north-g.41.26	41	82	13	8	11	8
north-g.42.5	42	109	16	16	36	12
north-g.55.30	55	130	18	3	3	3
p2p-Gnutella04-10-76-raw	1087	2157	76	138	197	58
p2p-Gnutella04-30-raw	3262	10968	130	1596	4882	113
p2p-Gnutella06-10-200-raw	871	1952	200	73	178	73
p2p-Gnutella06-30-raw	2615	8707	100	1076	3408	92
p2p-Gnutella08-10-54-raw	630	1574	54	72	220	36
p2p-Gnutella08-30-raw	1890	6951	95	771	2998	85
p2p-Gnutella24-10-118-raw	2651	4640	118	99	112	91
p2p-Gnutella24-30-raw	7955	22152	75	3221	9033	70
p2p-Gnutella31-10-92-raw	6258	11759	92	435	681	58
p2p-Gnutella31-30-raw	18775	54012	200	7010	20554	187
rocketfuel-k_o-133	318	1478	133	178	1202	130
rocketfuel-k_o-735	960	5500	735	679	4943	679
rocketfuel-k_o-99	240	787	99	133	571	97
rsndlib–36	100	300	36	88	276	34
soc-Slashdot0811-10-18-raw	7736	261974	18	7247	252994	14
soc-Slashdot0811-100-raw	77360	905468	24	-	-	-
soc-pokec-relationships-10-raw	163280	2558494	34	-	-	-
soc-redditHyperlinks-body-10-22-raw	3577	49230	22	2483	43745	18
soc-redditHyperlinks-body-100-raw	35776	137821	26	10253	95414	20
soc-sign-bitcoinalpha-10-16-raw	378	4237	16	334	4111	14
soc-sign-bitcoinalpha-30-raw	1134	13226	14	953	12794	11
stanford-Oregon-1-1085-384	1025	2115	384	82	268	82
stanford-Oregon-1-1557-543	1437	3036	543	166	560	166
stanford-ego-Facebook-235-167	203	458	167	96	249	96
stanford-musae-wiki_chameleon-245-92	200	477	92	95	276	87
stanford-musae-wiki_crocodile-1311-1200	1221	2556	1200	234	628	234
stanford-musae-wiki_squirrel-759-252	672	1480	252	265	690	247
topologyzoo-k_o-132	145	362	132	124	314	124
topologyzoo-k_o-366	754	1745	366	660	1554	366
twitter_combined-10-28-raw	8130	251918	28	7651	243746	27
twitter_combined-30-raw	24391	829792	30	23258	807199	27
web-Google-10-100-raw	87571	662997	100	38910	342602	98
web-Google-30-raw	262713	2193922	68	-	-	-
web-Stanford-10-154-raw	28190	412543	154	12196	195257	151
web-Stanford-100-raw	281903	2312497	488	-	-	-
wiki-Vote-10-16-raw	711	16955	16	379	8878	14
wiki-Vote-30-raw	2134	68860	12	977	32293	11

Table 2: Results of preprocessing for Problem 2

	$\|V\|$	$\|E\|$	$\|\ell\|$	# paths	PW	**FBS** [s]	**HAMDP** [s]
matpower-case145-105	101	376	102	3.6×10^{36}	14	**21.02**	timeout
matpower-case145-34	101	376	31	1.8×10^{21}	14	**12.4**	timeout
matpower-case1888rte-1420	360	736	778	2.0×10^{73}	15	**127.21**	timeout
matpower-case1888rte-154	360	736	147	-	15	timeout	timeout
matpower-case1888rte-75	360	736	68	6.4×10^{18}	15	**92.65**	timeout
matpower-case300-123	124	228	120	1.2×10^{21}	10	**1.8**	timeout
matpower-case89pegase-37	49	166	34	8.4×10^{15}	13	**17.88**	timeout
matpower-case_ACTIVSg200-140	58	103	128	2.1×10^{10}	9	**0.38**	timeout
matpower-case_ACTIVSg2000-1300	896	1563	1299	-	37	timeout	timeout
matpower-case_ACTIVSg2000-901	896	1563	900	-	37	timeout	timeout
matpower-case_ACTIVSg500-127	96	180	125	2.3×10^{17}	9	**1.43**	timeout
matpower-case_ACTIVSg500-392	96	180	239	6.2×10^{17}	9	**1.04**	timeout
rocketfuel-k_o-133	132	572	129	3.3×10^{34}	11	**5.44**	timeout
rocketfuel-k_o-136	113	320	135	6.8×10^{34}	11	**2.77**	timeout
rocketfuel-k_o-375	424	1871	372	-	36	timeout	timeout
rocketfuel-k_o-550	537	2393	548	-	48	timeout	timeout
rocketfuel-k_o-735	537	2393	678	-	48	timeout	timeout
rocketfuel-k_o-99	83	245	97	2.5×10^{24}	14	**24.71**	timeout
rsndlib–36	57	111	34	3.5×10^{9}	15	**24.35**	timeout
rsndlib–62	57	111	61	2.8×10^{12}	14	**118.66**	timeout
stanford-Oregon-1-1085-384	35	94	82	1.9×10^{9}	11	**0.50**	timeout
stanford-Oregon-1-1557-543	64	183	166	4.0×10^{16}	15	**49.71**	timeout
stanford-Oregon-1-1557-629	64	183	166	4.0×10^{16}	15	**49.68**	timeout
stanford-Oregon-1-1557-839	64	183	166	4.0×10^{16}	15	**49.76**	timeout
stanford-as-733-1003-776	34	97	96	5.3×10^{8}	9	**0.16**	timeout
stanford-as-733-1492-1156	57	183	170	3.0×10^{15}	14	**31.75**	timeout
stanford-as-733-1492-140	57	183	130	3.0×10^{15}	14	**41.07**	timeout
stanford-as-733-610-465	14	36	54	1.1×10^{4}	7	**0.02**	0.05
stanford-ca-CondMat-423-206	12	18	55	4.8×10^{1}	5	**0.02**	**0.02**
stanford-ca-GrQc-357-132	48	137	72	-	18	timeout	timeout
stanford-ca-GrQc-695-334	81	222	142	-	20	timeout	timeout
stanford-ca-GrQc-695-453	81	222	142	-	20	timeout	timeout
stanford-ca-GrQc-695-483	81	222	142	-	20	timeout	timeout
stanford-ego-Facebook-1166-162	309	653	155	-	34	timeout	timeout
stanford-ego-Facebook-235-167	42	73	92	4.7×10^{7}	12	**0.31**	timeout
stanford-ego-Facebook-235-177	42	73	92	4.7×10^{7}	12	**0.29**	timeout
stanford-ego-Facebook-425-131	99	188	128	-	18	timeout	timeout
stanford-ego-Facebook-425-159	99	188	156	-	18	timeout	timeout
stanford-feather-lastfm-social-234-114	10	15	25	4.6×10^{1}	5	**0.00**	**0.00**
stanford-feather-lastfm-social-508-178	55	113	129	2.4×10^{11}	12	**0.53**	timeout
stanford-musae-wiki_chameleon-245-92	46	91	85	7.1×10^{9}	12	**0.48**	timeout
stanford-musae-wiki_chameleon-570-325	65	153	157	6.6×10^{13}	12	**1.15**	timeout
stanford-musae-wiki_crocodile-1311-1200	72	160	225	3.1×10^{16}	13	**2.09**	timeout
stanford-musae-wiki_crocodile-1311-636	72	160	225	3.1×10^{16}	13	**2.07**	timeout
stanford-musae-wiki_squirrel-1253-485	254	565	483	-	54	timeout	timeout
stanford-musae-wiki_squirrel-759-252	104	191	242	-	19	timeout	timeout
topologyzoo-k_o-12	10	16	12	3.0×10^{1}	5	**0.00**	0.01
topologyzoo-k_o-132	55	94	131	1.8×10^{9}	7	**0.34**	372.74
topologyzoo-k_o-169	202	341	169	1.6×10^{21}	9	**5.04**	timeout
topologyzoo-k_o-366	202	341	366	1.1×10^{31}	9	**6.02**	timeout

Table 3: Experimental results for Problem 1

	$\|V\|$	$\|E\|$	$\|\ell\|$	# paths	PW	**FBS** [s]	**SCC** [s]
airlines-migration-airrtraffic-airlines-8	189	1998	7	7.2×10^6	44	205.89	**12.38**
cit-HepPh-10-30-raw	21	28	21	4.5×10^1	3	0.01	**0.00**
cit-HepPh-30-raw	18	34	18	3.6×10^2	6	0.08	**0.00**
email-EuAll-10-20-raw	6080	72461	16	-	1426	timeout	timeout
email-EuAll-30-raw	12058	101733	42	-	1805	timeout	timeout
matpower-case145-34	130	788	33	6.5×10^{20}	14	68.92	timeout
matpower-case1888rte-1420	777	2243	777	-	15	timeout	timeout
matpower-case_ACTIVSg200-140	127	334	127	1.6×10^9	9	0.78	timeout
north-g.100.0	29	54	22	9.4×10^2	6	0.02	**0.00**
north-g.12.79	7	13	7	1.3×10^1	4	**0.00**	**0.00**
north-g.41.26	8	11	8	9.0×10^0	3	**0.00**	**0.00**
north-g.42.5	16	36	12	1.2×10^2	7	0.01	**0.00**
north-g.55.30	3	3	3	2.0×10^0	3	0.01	**0.00**
p2p-Gnutella04-10-76-raw	138	197	58	5.3×10^3	10	0.11	**0.01**
p2p-Gnutella04-30-raw	1596	4882	113	-	310	timeout	timeout
p2p-Gnutella06-10-200-raw	73	178	73	1.9×10^7	13	**2.19**	18.65
p2p-Gnutella06-30-raw	1076	3408	92	-	193	timeout	timeout
p2p-Gnutella08-10-54-raw	72	220	36	1.0×10^9	18	**172.42**	timeout
p2p-Gnutella08-30-raw	771	2998	85	-	132	timeout	timeout
p2p-Gnutella24-10-118-raw	99	112	91	5.1×10^1	5	**0.00**	**0.00**
p2p-Gnutella24-30-raw	3221	9033	70	-	975	timeout	timeout
p2p-Gnutella31-10-92-raw	435	681	58	-	29	timeout	timeout
p2p-Gnutella31-30-raw	7010	20554	187	-	2095	timeout	timeout
rocketfuel-k_o-133	178	1202	130	2.2×10^{33}	11	**50.28**	timeout
rocketfuel-k_o-735	679	4943	679	-	47	timeout	timeout
rocketfuel-k_o-99	133	571	97	2.3×10^{23}	14	**412.52**	timeout
rsndlib–36	88	276	34	1.3×10^9	15	**240.09**	timeout
soc-Slashdot0811-10-18-raw	7247	252994	14	-	-	timeout	timeout
soc-Slashdot0811-100-raw	-	-	-	-	-	-	-
soc-pokec-relationships-10-raw	-	-	-	-	-	-	-
soc-redditHyperlinks-body-10-22-raw	2483	43745	18	-	641	timeout	timeout
soc-redditHyperlinks-body-100-raw	10253	95414	20	-	-	timeout	timeout
soc-sign-bitcoinalpha-10-16-raw	334	4111	14	-	78	timeout	timeout
soc-sign-bitcoinalpha-30-raw	953	12794	11	-	204	timeout	timeout
stanford-Oregon-1-1085-384	82	268	82	3.2×10^8	11	**4.36**	timeout
stanford-Oregon-1-1557-543	166	560	166	-	15	timeout	timeout
stanford-ego-Facebook-235-167	96	249	96	1.5×10^7	12	**7.65**	440.87
stanford-musae-wiki_chameleon-245-92	95	276	87	2.6×10^9	12	**6.32**	timeout
stanford-musae-wiki_crocodile-1311-1200	234	628	234	4.7×10^{15}	13	**55.9**	timeout
stanford-musae-wiki_squirrel-759-252	265	690	247	-	19	timeout	timeout
topologyzoo-k_o-132	124	314	124	1.1×10^8	7	**0.35**	13.80
topologyzoo-k_o-366	660	1554	366	1.2×10^{27}	9	**16.31**	timeout
twitter_combined-10-28-raw	7651	243746	27	-	-	timeout	timeout
twitter_combined-30-raw	23258	807199	27	-	-	timeout	timeout
web-Google-10-100-raw	38910	342602	98	-	-	timeout	timeout
web-Google-30-raw	-	-	-	-	-	-	-
web-Stanford-10-154-raw	12196	195257	151	-	-	timeout	timeout
web-Stanford-100-raw	-	-	-	-	-	-	-
wiki-Vote-10-16-raw	379	8878	14	-	145	timeout	timeout
wiki-Vote-30-raw	977	32293	11	-	355	timeout	timeout

Table 4: Experimental results for Problem 2