TCS-TR-A-**13-65**

TCS Technical Report

Graphillion: Software Library Designed for Very Large Sets of Graphs in Python

by

Takeru Inoue, Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato

Division of Computer Science

Report Series A

June 11, 2013

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682
Fax: +81-011-706-7682

Graphillion: Software Library Designed for Very Large Sets of Graphs in Python

Takeru Inoue*† Hiroaki Iwashita †‡ Jun Kawahara§
Shin-ichi Minato‡†

June 11, 2013

Abstract

Several graph libraries have been developed in the past few decades, but they were designed to work with a few graphs even though the number of subgraphs exponentially increases with graph size. In this paper, we develop Graphillion, a software library for very large sets of graphs. Graphillion is not established on a traditional representation of graphs. Instead, a graph set is simply regarded as a "set of edge sets" ignoring vertices, which allows us to employ powerful tools of a "family of sets" (a set of sets) and permits large graph sets to be handled efficiently. We also utilize advanced graph enumeration algorithms, which enables the simple family tools to understand the graph structure. Graphillion is implemented as a Python library to encourage easy development of its applications, without introducing significant performance overhead. In experiments, we consider two case studies, a puzzle solver and a power network optimizer, in which several operations and heavy optimization have to be done over very large sets of constrained graphs (i.e., cycles or forests with complicated conditions). The results show that Graphillion allows us to manage an astronomical number of graphs with very low development effort.

1 Introduction

A graph is a representation of a set of edges, each of which connects a pair of vertices. It is often used as a mathematical model for a variety of problems. Researchers have developed many sophisticated graph libraries, but the design was focused on handling a small number of graphs. Thus they cannot work with

*takeru.inoue@ieee.org
†ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency, Sapporo, Japan.
‡Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan.
§Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Nara, Japan.

very large sets of graphs, even though the set can grow exponentially with graph size since a graph with N edges induces 2^N subgraphs. A graph library that could efficiently manage very large and complex sets of graphs within a small amount of memory would provide a novel way for powerful graph operations; e.g., an optimizer that efficiently finds the best graph from a non-convex graph set, and a graph database that can select all matched graphs from a very large set. To the best of our knowledge, there is no library that has been designed to handle such a large set of graphs.

In this paper, we introduce Graphillion, a software library optimized for very large sets of graphs. Traditional graph libraries maintain each graph individually, which causes poor scalability, while Graphillion handles a set of graphs collectively without considering individual graph. Graphillion concentrates on edge-induced subgraphs of a given graph $G = (V, E)$, and a set of graphs is reduced into a set of edge collections[1], or *a family of sets of edges* more formally; i.e., a set of graphs, $\{G_1 = (V, E_1), G_2 = (V, E_2)\}$, is regarded as a set of edge collections, $\{G_1 = E_1, G_2 = E_2\}$. This reduction loses the properties of each vertex, but allows programmers to apply a powerful theory on the family [8]. A set of collections can be represented in a compressed form by sharing common parts of similar collections, so a huge number of graphs can be stored in a small amount of memory. We also employ efficient algebra called family algebra [6], in order to perform optimization, selection, and modification on very large graph sets; the efficiency is due to the fact that they can be executed without decompressing the data.

This family theory, of course, is unconcerned about graph structure like a tree or a path, since it considers a graph to be just an edge collection with no structure. We rectify this omission by employing the graph enumeration algorithm called frontier-based search [13, 5, 4]. The algorithm lists all graphs that have a specified structure, and then the listed graphs (edge collections) are handled by family algebra. The number of graphs listed, of course, can be very enormous, but a recent development in enumeration algorithms allows us to output the graphs in compressed form without enumerating them one by one. This compressed form is easily converted into the compressed form of the family theory [15], and so there is no difficulty to adopt family algebra.

Graphillion is implemented in Python language because of its high productivity. Python is a high-level programming language with a rich set of libraries (or "modules" in the Python terminology) including NumPy/SciPy (mathematical computation) [17] and NetworkX (network analysis) [2]. Moreover, Python can be extended by C or C++ for high-performance numerical computation, and it is suited for scientific and engineering code [12]. However, Python objects must be reinterpreted in every extended function call, and this overhead would be unacceptable if a very large graph set were involved in the function call (e.g., Python's built-in `set` object can reinterpret all elements in some function calls). Graphillion,

[1] In order to describe a set of sets without confusion, the word *collection* is used to indicate an "inner" set like an edge set, while *set* is used for an "outer" set like a graph set.

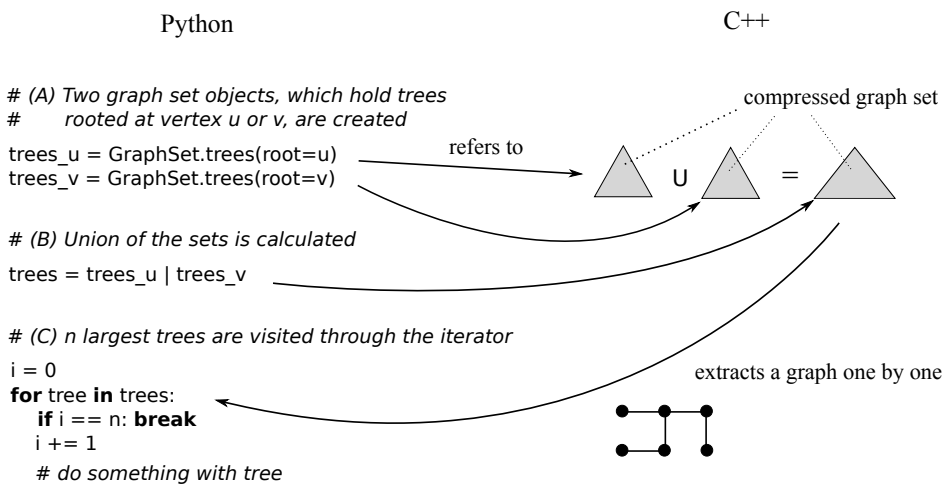

Figure 1: Graphillion's overview with a code example. The compressed graph set objects are maintained in the C++ world, while only minimum necessary objects are exposed to the Python world to reduce the overhead.

in contrast, deals with a whole graph set directly without considering individual graphs, and so just a reference to the set is reinterpreted regardless of the number of graphs in it. In this way, our graph set representation allows us to establish an efficient computation scheme of graph sets via Python's extension mechanism.

We evaluate the performance and productivity of Graphillion in experiments. We first measure the performance of simple operations. The results show that Graphillion needs only 500 MB of memory to process a very large set of 10^{37} trees in 10 seconds (just one second for some operations). We then present two case studies, a puzzle solver and a power network optimizer, and reveal that Graphillion cuts the lines of code by 90 % with acceptable performance overhead. In the power network optimization, our optimizer, which only needs a thousand lines of code, searches a non-convex set of 10^{58} feasible graphs and finds the optimal one just in one minute.

The rest of this paper is organized as follows. Section 2 gives an overview of Graphillion. Sections 3 and 4 discuss the theoretical aspects of Graphillion. Section 5 describes implementation, and Section 6 reports the experiments and results. Section 7 summarizes related work, and Section 8 concludes the paper.

2 Overview

We describe below a design overview of Graphillion (Fig. 1), along with our goals, high performance and high productivity.

- **High performance.** Graphillion processes very large sets of graphs efficiently in terms of time and space. It is implemented as a Python module

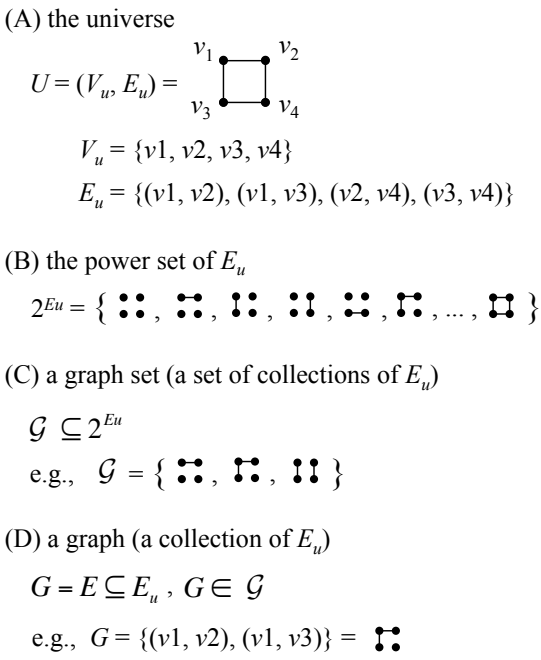

Figure 2: Examples of our graph representation.

with C++ extensions. A set of graphs is represented in a compressed form of a C++ object, which is created by frontier search (Fig. 1 (A)) and is manipulated by family algebra (Fig. 1 (B)). Since only the reference of the set is exposed to the Python world, the function call overhead is very small and its impact is independent of the size of the C++ object. Only minimum necessary graphs are extracted from the set through iterators, so there is no need to restore all the graphs in the object (Fig. 1 (C)).

- **High productivity.** Graphillion makes it easy to develop applications that deal with very large graph sets. Graphillion follows the programming interface of the built-in `set` class in Python (Fig. 1 (B, C)), and so it is very easily for Python programmers to use. Since we redesign family algebra to suit graph sets, it is tractable to write complicated operations over graph sets, such as optimization, selection, and modification. Since Python is a general-purpose programming language with a rich set of modules, programmers can implement their tasks just using Python and they are freed from the need to coordinate multiple programs in different languages. We evaluate the productivity by the number of code lines in this paper.

3 Representations of a Graph and the Set

This section formulates a graph set as a set of edge collections. Figure 2 shows an example of the representation used in this section.

3.1 Representation of a Graph

We first introduce a special graph that defines our universe (Fig. 2 (A)),

$$U = (V_u, E_u).$$

A graph G used in Graphillion must be an edge-induced subgraph of the universe (Fig. 2 (D)),

$$G = E \subseteq E_u,$$

where only edge collection E defines the graph, while vertices V are ignored. This simplification puts a limitation on vertices; vertices without edges cannot be recognized. However, graphs are mainly characterized by edge structures in many applications, making this limitation not a serious concern in most cases.

Our graph model puts no restriction on edge type [2], but this paper treats only simple undirected edges with no self-loops for simplicity. Edges can be weighted.

3.2 Representation of a Set of Graphs

A set of graphs, $\mathcal{G}$, is represented by a set of collections of E_u (Fig. 2 (B, C)),

$$\mathcal{G} \subseteq 2^{E_u},$$

where 2^{E_u} is the power set of E_u. A graph used in Graphillion is defined by $G \in 2^{E_u}$.

The maximum size of a graph set, $2^{|E_u|}$, increases exponentially with universe size. In order to represent a graph set efficiently, we utilize a compressed form of a set of collections, which is named the zero-suppressed binary decision diagram, or ZDD [8]. ZDD greatly compresses a very large set of collections by sharing the common parts of similar collections. We show an example of the great compression capability yielded by ZDD in Table 1, which presents the number of trees rooted at a corner on a grid graph versus the amount of memory needed to store them in ZDD (theoretical value ignoring implementation overhead). The amount of memory increases much more slowly than the number of trees[3].

[2] Edges can be either directed or undirected. They can also have self-loops. Multiple edges can be placed between a same pair of vertices if they are distinguishable. Edges can be hyper edges, which include any number of vertices.

[3] There is no rigorous theory that can estimate the compression ratio of binary decision diagrams, but it is believed that they will work well in most practical data applications [18].

Table 1: Number of trees versus memory needed by ZDD

Grid size	Number of trees	Memory of ZDD [Byte]
2×2	10	990
3×3	750	9870
4×4	737354	61830
5×5	8965981766	335190
6×6	1334122533591284	2364750
7×7	2417510626051127173092	18168510
8×8	531403153128266503005306201 74	56321790
9×9	1413043452230406655789221373129700 9012	207115950

Table 2: Creation methods for graph sets

Structure	Parameters
tree	a root vertex, spanning or not
forest	root vertices, spanning or not
path	terminal vertices, hamilton or not
cycle	hamilton or not
clique	size
connected component	vertices to be connected

4 Creation and Manipulation of a Set of Graphs

This section describes the creation of a graph set using frontier search and the use of family algebra to manipulate set contents.

4.1 Creation of a Set of Graphs

We build a ZDD representing a set of graphs by using a graph enumeration algorithm called frontier-based search [13] [4], which integrates several advanced techniques. Frontier search finds all graphs that have a specified structure based on dynamic programming. It outputs the enumerated graphs in a compressed form that is easily converted into a ZDD [15]. The time complexity is ruled by the size of the compressed form (slightly larger than that of ZDD), not the number of graphs being output.

Frontier search was originally limited to trivial structures like trees, but it has been generalized to support various structures [5]. Table 2 shows the structures supported by Graphillion.

[4] While this enumeration algorithm had no name originally, it was assigned the name in [10]

Table 3: Selection operations for graph sets

Operation	Definition
union	$\mathcal{G}_1 \cup \mathcal{G}_2 = \{G \mid G \in \mathcal{G}_1 \vee G \in \mathcal{G}_2\}$
intersection	$\mathcal{G}_1 \cap \mathcal{G}_2 = \{G \mid G \in \mathcal{G}_1 \wedge G \in \mathcal{G}_2\}$
difference	$\mathcal{G}_1 \setminus \mathcal{G}_2 = \{G \mid G \in \mathcal{G}_1 \wedge G \notin \mathcal{G}_2\}$
symmetric difference	$\mathcal{G}_1 \oplus \mathcal{G}_2 = (\mathcal{G}_1 \setminus \mathcal{G}_2) \cup (\mathcal{G}_2 \setminus \mathcal{G}_1)$
subgraphs	$\mathcal{G}_1 \sqsubseteq \mathcal{G}_2 = \{G_1 \in \mathcal{G}_1 \mid \exists G_2 \in \mathcal{G}_2 (G_1 \subseteq G_2)\}$
supergraphs	$\mathcal{G}_1 \sqsupseteq \mathcal{G}_2 = \{G_1 \in \mathcal{G}_1 \mid \exists G_2 \in \mathcal{G}_2 (G_1 \supseteq G_2)\}$
maximal graphs	$\mathcal{G}^{\uparrow} = \{G_1 \in \mathcal{G} \mid G_2 \in \mathcal{G} \wedge G_1 \subseteq G_2 \rightarrow G_1 = G_2\}$
minimal graphs	$\mathcal{G}^{\downarrow} = \{G_1 \in \mathcal{G} \mid G_2 \in \mathcal{G} \wedge G_1 \supseteq G_2 \rightarrow G_1 = G_2\}$

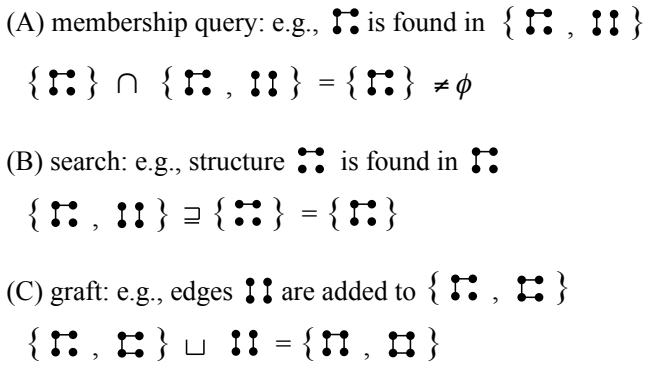

Figure 3: Examples of graph set manipulation via family algebra.

The search space can be limited within a given graph set; graphs not included in the given set are not enumerated by frontier search [4].

Simple graph sets can be created by ZDD's primitives without frontier search; an empty set and a power set are given by the ZDD's primitives, and small graph sets can be created by explicitly specifying the graphs (edge collections).

4.2 Manipulation of a Set of Graphs

Family algebra defines several operations on sets of collections, and the operations can be efficiently performed over ZDDs [6]. Surprisingly, these operations can be executed on the compressed data without decompression, so they are highly efficient. In this subsection, we describe the operations for optimization, selection, and modification, in the context of graph sets.

We begin with selection operations. Several selection operations are defined for a set of collections, and their semantics make sense for graph sets without change. The first four operations in Table 3 are ordinary set operations. Each graph in a

Table 4: Modification operations for graph sets

Operation	Definition
graft (join $\sqcup$)	$\mathcal{G} \sqcup \{E\} = \{G \cup E \mid G \in \mathcal{G}\}$
remove (meet $\sqcap$)	$\mathcal{G} \sqcap \{E^c\} = \{G \cap E^c \mid G \in \mathcal{G}\}$
flip (delta $\boxplus$)	$\mathcal{G} \boxplus \{E\} = \{G \oplus E \mid G \in \mathcal{G}\}$

set is regarded as an opaque element without inner structure, and the operations are performed over the sets. It is worth noting that *intersection* can be used for a membership query; to test if graph G is in set $\mathcal{G}$ by checking (Fig. 3 (A)),

$$\{G\} \cap \mathcal{G} \neq \emptyset.$$

The other four operations in the table select graphs based on their structures. They do what their names suggest (they are originally called subsets or maximal sets in family algebra). The *supergraphs* operation can be used for search; to explore $\mathcal{G}$ for graphs that include given structure G by (Fig. 3 (B)),

$$\mathcal{G} \sqsupseteq \{G\}.$$

We move to modification. All graphs in a set can be modified at once by slightly modifying family algebra. Table 4 shows the modification operations (original operation names in family algebra are shown in parentheses for reference). To graft edge(s) E to all graphs in set $\mathcal{G}$, we utilize *join* operation defined in the family algebra, as shown in Table 4 (Fig. 3 (C)). Similarly, edge(s) E can be removed by performing *meet* operation against the complement edge set $E^c = E_u \setminus E$ (i.e., E^c are edges not to be removed in this context). The *flip* operation flips edge status in all graphs.

Optimization is provided by a search algorithm of family algebra that finds a maximum or minimum weighted edge collection (graph) in the set. Since this search algorithm returns just a single best graph, we employ the *difference* operation to obtain multiple graphs in descending (or ascending) order of weight; the search algorithm is applied repeatedly while removing the previous best graph from the set by the difference operation as follows.

for $i = 1 \to k$ **do** {find top-k graphs from $\mathcal{G}$}
  $G = \text{find\_max}(\mathcal{G})$ {get best G from $\mathcal{G}$}
  {do something with G}
  $\mathcal{G} = \mathcal{G} \setminus \{G\}$ {remove G for the next iteration}
end for

Graphillion defines other operations like hitting sets [16], random sampling, and counting graphs in a set, but we do not describe them due to space limits.

5 Implementation

This section describes the implementation of Graphillion. Frontier search and family algebra are implemented in C++, while the programming interface is written in Python. This interface is based on Python's `set`; e.g., the size query (`len` function in Python), membership query (`in` operation), iterators (`for` operation), and general set operations (`union`, etc.). We add graph-specific operations to this interface like `supergraphs`, `graft`, and the graph-weight optimizers. Our implementation requires 14,965 lines of code in C++ and 2,251 lines in Python.

A graph set object in Python maintains a reference to the corresponding ZDD object of C++ (Fig. 1). The graph set object is very lightweight, since it has no attribute other than the reference. The selection methods return a new graph set object that refers to the associated ZDD object. The modification methods just replace their reference with a new reference to the new ZDD object. The optimizers are implemented as a Python iterator, which runs a loop step by step and yields the best graphs one by one instead of extracting all of them at once.

Vertices and edges are simply indexed by integers in C++ to improve the efficiency, while any *hashable* object can be used as a vertex in Python for better productivity[5] (an edge is just a tuple of two vertex objects). Graphillion provides a transparent mechanism to convert integers and objects by maintaining the mapping. The mapping is created automatically at universe registration, which must be done at the beginning of the code. If edges not found in the universe are used, an exception is raised.

In order to enhance productivity further, any type of graph object (e.g., NetworkX graph) can be used in Graphillion. A graph object is transparently converted into the Graphillion's internal representation (an edge collection) by user-defined converters. Programmers can use Graphillion as an enhancement tool for their favorite graph modules simply by registering the converters.

6 Experiments

In this section, we first show the performance of Graphillion's operations. We then discuss two case studies, a puzzle solver and a power network optimizer, to examine the tradeoff between performance and productivity. All experiments were conducted with Python 2.7 and GCC 4.7 on Linux 2.6 using a single core in Intel Xeon E31290 (3.60 GHz) with 32 GB of RAM.

6.1 Basic Performance

We evaluate the performance using a set of trees rooted at a corner on a grid graph. The set size is shown in Table 1. The performance of creation is measured

[5] This is analogous to Python's built-in `set`, which accepts any hashable object as an element

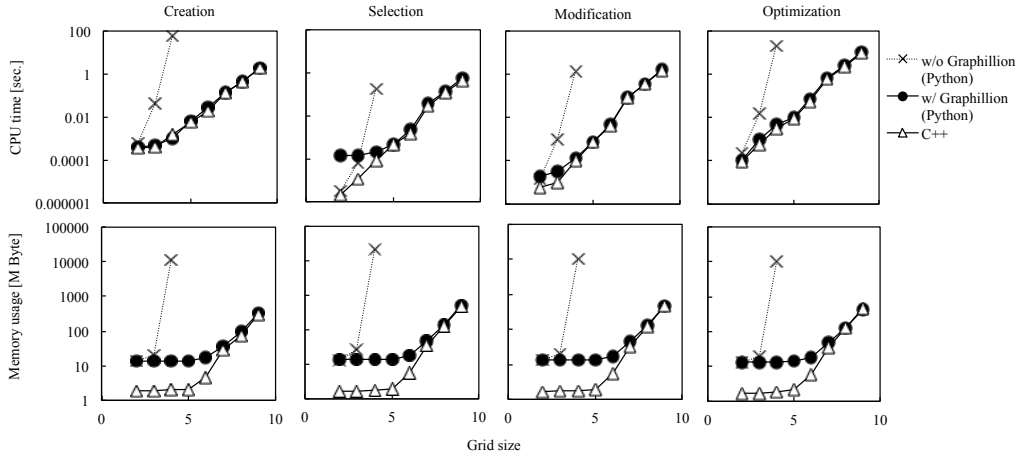

Figure 4: CPU time and memory usage for the basic operations with and without Graphillion. The operations are executed over trees rooted at a corner on a grid graph. The grid size on the horizontal axis indicates N of the $N \times N$ grid.

by building a set of the trees. The selection performance is evaluated by calculating the union of two sets of trees; trees in one set are rooted at a corner while those in the other set are rooted at the diagonally opposite corner. The modification performance is evaluated by grafting an edge to all trees. The optimization performance is measured by finding the top-3 weighted trees with the maximizing operation.

We measured the CPU time and the memory usage of these operations with and without Graphillion. In the implementation without Graphillion, graphs are created as NetworkX objects, and are stored in Python's built-in `set` object (the union operation is provided by the built-in `set`, but the other operations were added by us). In order to evaluate Python's overhead, we developed pure C++ implementation of the operations just for the experiments.

The results are shown in Fig. 4. The implementation without Graphillion could not finish any operation for a 5×5 grid within an hour due to the very large number of trees. Graphillion performs a little poorly on the small grids due to the overhead of object mapping and conversion, but the overhead is negligible in the larger grids. It finished all operations in less than 10 seconds with 500 MB of memory even for the 9×9 grid, which has 10^{37} trees. Creation and optimization are slower than the other operations, because they involve complicated search algorithms. Selection requires twice as much memory than the others since it uses two sets of trees[6], but it is the fastest due to its simple operation.

[6]Selection requires 500 MB of memory, which is slightly larger than double the theoretical value (207 MB), shown in Table 1, because of the unused slots in the hash table used to maintain ZDDs. The flat regions seen in the memory usage for smaller grids are also due to the pre-allocated slots of the hash table.

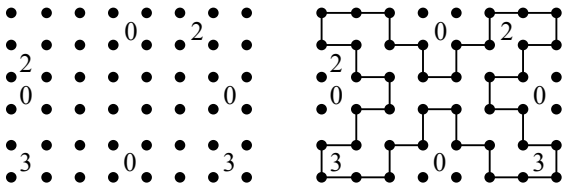

Figure 5: An example of the Slitherlink problem (left) and its solution (right) on 6×8 grid; adjacent dots are connected with vertical or horizontal lines, and a cycle is formed satisfying given hints, which indicate the number of lines surrounding it while empty cells may be surrounded by any number of lines.

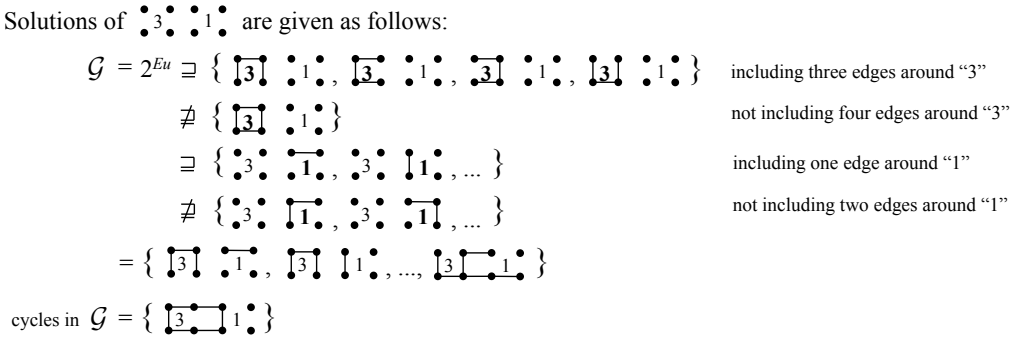

Figure 6: An example of Slitherlink solution by Graphillion. Here, we define $\mathcal{G}_1 \not\supseteq \mathcal{G}_2 = \mathcal{G}_1 \setminus (\mathcal{G}_1 \supseteq \mathcal{G}_2)$. For the hint of "3", the solutions must include (be supergraphs of) three edges around the hint (2nd line), but must *not* include more edges (3rd line). Similarly, the hint of "1" is processed (4th and 5th lines). Finally, cycles are found by frontier search (7th line).

6.2 Puzzle Solver

The first case study is the Slitherlink puzzle[7], which is a logic puzzle to find a cycle that satisfies given hints (Fig. 5). We developed a Slitherlink solver in our past work [19]; it was the fastest solver that could list *all* solution cycles. The solver employs frontier search redesigned for Slitherlink; it has special algorithms to process hints. The solver is written in 2,116 lines of C++ code.

We developed another solver with Graphillion, without frontier search dedicated for Slitherlink. This new solver, first, enumerates subgraphs that satisfy the hints (2nd to 5th lines of Fig. 6), and then runs frontier search over the hint-satisfying subgraphs to select solution cycles (7th line of Fig. 6). Thanks to the generality of Graphillion, the new solver is written just in 153 lines of Python. This is a 93 % reduction in code line number, and it is, in addition, written in easy Python, not in complicated C++ (Table 5).

[7] `http://www.nikoli.com/en/puzzles/slitherlink/`

Table 5: Lines of Code for Slitherlink Solvers

Implementation	C++	Python
w/o Graphillion	2116	0
w/ Graphillion	0	153

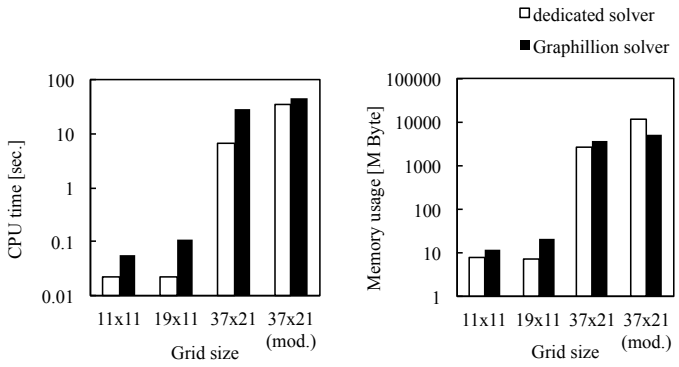

Figure 7: CPU time and memory usage of the dedicated solver and the Graphillion solver on Slitherlink problems.

We measure the CPU time and memory usage on three problems found in a Slitherlink book [11], all of which have just a single solution. We also conduct an experiment against a modified problem in which ten hints are randomly removed to permit multiple solutions. Figure 7 shows the results. Both solvers scaled similarly with problem size, and their memory usages were roughly comparable. The Graphillion solver is slightly outperformed in CPU time due to the special algorithms in the dedicated solver, but the tradeoff between performance and productivity is acceptable.

We can obtain top-k longest or shortest cycles with Graphillion's iterators, when the problem has multiple solution cycles. It took just another 0.24 seconds to find the three longest cycles from among the 117059496 solutions in the modified problem.

6.3 Power Network Optimizer

The second case study is power loss minimization in a distribution network; this is a discrete non-convex optimization problem involving hundreds of variables [7]. A power distribution network can be represented by a graph in which a vertex corresponds to a town block or a power substation while an edge is a power line with a switch (Fig. 8). The power flow is configured by changing the open/closed status

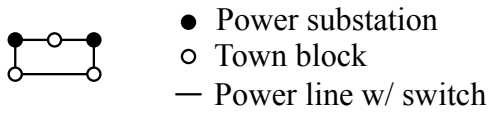

Figure 8: An example of power distribution network, which is represented by a graph; the power flow can be configured by the switches.

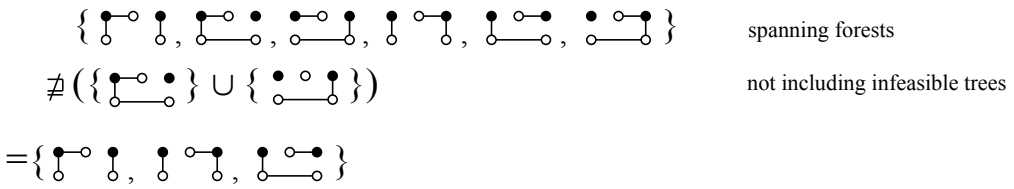

Figure 9: An example of optimization algorithms for power networks in Fig. 8; feasible solutions are obtained as the spanning forests with no infeasible trees, and then the optimal one is searched for (not shown in the figure).

of switches. It must be cycle-free to avoid short circuits, and must cover all blocks to avoid blackouts; the power flow, as a consequence, forms a spanning forest, in which each tree is rooted at a power substation. The flow also must satisfy complicated electrical constraints on line capacity and voltage drop; roughly speaking, very large or tall trees are forbidden. The network is operated to minimize resistive line losses while satisfying these constraints.

In our past work [3], we developed a power loss optimizer that utilized frontier search and family algebra in an ad-hoc manner without the unified concept discussed in this paper. The loss optimizer first enumerates all spanning forests rooted at substations by frontier search (1st line of Fig. 9). It then enumerates all electrically-*in*feasible trees for each substation by conducting complicated power calculations (2nd line of Fig. 9). Family algebra selects forests that do *not* include the infeasible trees (3rd line of Fig. 9). Finally, the minimum-loss forest is found from the selected feasible forests; since the search space consists of only the feasible forests, the search algorithm does not need to consider the complicated constraints. To handle the nonlinear nature of the power loss, a dedicated search algorithm had been developed (that of family algebra was not used). Our past work implemented a part of frontier search and of family algebra in 6,856 lines of C++ code, while the complicated power calculations, including nonlinear optimization, was written in 1,221 lines of Python code. Intermediate data are serialized into a file, which is exchanged between the C++ program and the Python program.

We developed another power loss optimizer that implements the same algorithms but employs Graphillion for frontier search and family algebra; we are

Table 6: Lines of Code for Power Network Optimizers

Implementation	C++	Python
w/o Graphillion	6856	1221
w/ Graphillion	0	1164

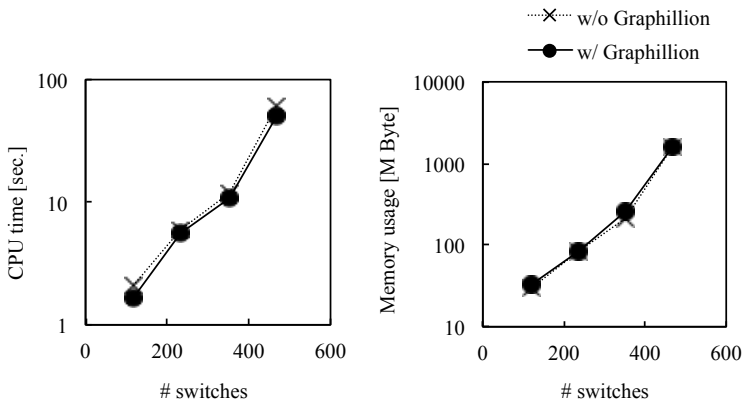

Figure 10: CPU time and memory usage of power network optimizers with and without Graphillion.

allowed to focus on the power calculation and the nonlinear optimization. Since this optimizer is implemented as a single program, it does not need to exchange intermediate data. It is written in 1,164 lines of Python code without C++. This Python code is shorter than the original, because it does not require serialization and object mapping. In total, we achieved a 86 % reduction in code line number, and it is, in addition, written in easy Python, not complicated C++ (Table 6).

The two optimizers are compared for power distribution networks used in [3]; the largest network has 432 blocks (vertices) and 468 power lines (edges). The results are shown in Fig. 10. Both implementations demonstrate comparable performance in the CPU time and memory usage (the memory usage includes both C++ and Python programs for the implementation without Graphillion). The Graphillion optimizer was slightly faster due toits omission of data exchange, while it required a bit more memory because of the full Python implementation. This memory overhead is negligible compared to the productivity improvement, which allows programmers to focus on their own problems without considering complicated graph operations. Surprisingly, more than 10^{58} feasible forests were handled with only 1.5 GB memory in the largest network. Graphillion needed just one thousand code lines to find an optimal solution from a non-convex set of 10^{58} graphs in just one minute.

Graphillion also can be used as a graph database of feasible forests. We issued queries specifying an open/closed switch to select all the forests matched to the queries, like Fig. 3 (B). Graphillion processed the queries within just 1.5 seconds for a closed switch and within 0.5 seconds for an open switch in the largest network.

7 Related Work

There are several graph libraries like NetworkX [2] and Boost Graph Library [14]. These libraries are widely used for graph analysis. They are, however, designed for a small number of graphs or a simple power set of edges; i.e., they can find a shortest path just from a power set of edges without constraints. In contrast, Graphillion can find shortest paths from large and complex sets of graphs, since it can maintain such sets explicitly but efficiently.

We often use general optimizers like CPLEX[8] for graph optimization. However, they require us to describe the constraints in simple formulae, but many practical problems are too complicated to permit this. The algebraic approach provided by Graphillion sometimes works well as shown by the power network optimization, which cannot be solved by general optimizers. In addition, general optimizers are not designed to search for multiple solutions, while Graphillion provides iterators that yield top-k solutions.

Graph databases [1] store multiple graphs and provide selection methods on graph structure. However, they cannot store as many graphs as Graphillion can, because they do not employ efficient graph set representation.

VSOP [9] employs family algebra as does Graphillion, but it provides an abstraction for combinatorial item sets, not graph sets. Frontier search is, of course, not implemented in VSOP, and so it does not create graph sets of a given structure efficiently. Since VSOP runs on its own interpreter, we cannot rely on Python's rich libraries.

8 Conclusions

In this paper, we have introduced Graphillion, which is a software library designed for very large sets of graphs. Our representation of a graph set allows us to utilize the theory of the "family of sets", which can compress graph sets and manipulate them efficiently. Graphillion is implemented in Python and provides a sophisticated but easy to use interface. Experiments showed the excellent performance of Graphillion. Two case studies revealed that programmers can handle very large graph sets with just a small number of lines of code.

Future work includes a plug-in mechanism for operation customization, generalized design for directed graphs and hyper graphs, and analysis of compression ratio on graph set characteristics.

[8] `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`

Since we would like to find out more applications for which Graphillion works well, we make it publicly available online at Graphillion's page[9] and PyPI (Python Package Index)[10].

References

[1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[2] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, pages 11–16, 2008.

[3] T. Inoue, K. Takano, T. Watanabe, J. Kawahara, R. Yoshinaka, A. Kishimoto, K. Tsuda, S. Minato, and Y. Hayashi. Distribution network reconfiguration for tightly bounded minimum loss by ZDDs. Technical report, Hokkaido University, Division of Computer Science, TCS Technical Reports, TCS-TR-A-12-58, 2012.

[4] H. Iwashita, J. Kawahara, T. Saitoh, R. Yoshinaka, and S. Minato. Top-down ZDD construction techniques for efficient graph enumeration and indexing. Technical report, Hokkaido University, Division of Computer Science, TCS Technical Reports. to appear.

[5] J. Kawahara et al. Frontier search for enumerating all constrained subgraphs with compressed representation. Technical report, Hokkaido University, Division of Computer Science, TCS Technical Reports. to appear.

[6] D. E. Knuth. *7.1.4 Binary Decision Diagrams*, volume 4A. Addison-Wesley, USA, 2011.

[7] J. Lavaei, A. Rantzer, and S. Low. Power flow optimization using positive quadratic programming. In *Proceedings of the 18th IFAC World Congress*, 2011.

[8] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of Conference on Design Automation*, pages 272–277, 1993.

[9] S. Minato. VSOP (valued-sum-of-products) calculator for knowledge processing based on zero-suppressed BDDs. In *Federation over the Web*, volume 3847 of *Lecture Notes in Computer Science*, pages 40–58. Springer Berlin Heidelberg, 2006.

[9] `http://graphillion.org/`
[10] `http://pypi.python.org/`

[10] S. Minato. Techniques of BDD/ZDD: Brief history and recent activity. *IEICE Trans. Inf. & Syst.*, E96-D(7), 2013.

[11] Nikoli. *Slitherlink 1.* 1992.

[12] T. E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[13] K. Sekine, H. Imai, and S. Tani. Computing the Tutte polynomial of a graph of moderate size. In *Algorithms and Computations*, volume 1004 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 1995.

[14] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Professional, 2001.

[15] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48(3):139–144, 1993.

[16] T. Toda. Hypergraph transversal computation with binary decision diagrams. In *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2013.

[17] S. van der Walt, S. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

[18] R. Yoshinaka, J. Kawahara, S. Denzumi, H. Arimura, and S. Minato. Counterexamples to the long-standing conjecture on the complexity of BDD binary operations. *Information Processing Letters*, 112(16):636–640, 2012.

[19] R. Yoshinaka, T. Saitoh, J. Kawahara, K. Tsuruma, H. Iwashita, and S. Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.