Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-014-0352-z

REGULAR PAPER

# Graphillion: software library for very large sets of labeled graphs

**Takeru Inoue · Hiroaki Iwashita · Jun Kawahara · Shin-ichi Minato**

**Abstract** Several graph libraries have been developed in the past few decades, and they were basically designed to work with a few graphs. However, there are many problems in which we have to consider all subgraphs satisfying certain constraints on a given graph. Since the number of subgraphs can increase exponentially with the graph size, explicitly representing these *sets* is infeasible. Hence, libraries concerned with efficiently representing a *single* graph instance are not suitable for such problems. In this paper, we develop Graphillion, a software library for very large sets of (vertex-) labeled graphs, based on zero-suppressed binary decision diagrams. Graphillion is not based 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 enable 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 overheads. 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 performed over very large sets of constrained graphs (i.e., cycles or forests with complicated conditions). The results show that Graphillion allows us to manage a huge number of graphs with very low development effort.

**Keywords** Graph · Set · Software library · Family algebra · Frontier-based search · Binary decision diagram

T. Inoue (✉) · H. Iwashita · S. Minato
ERATO Minato Discrete Structure Manipulation System Project, Japan Science and Technology Agency, Hokkaido 060–0814, Japan
e-mail: takeru.inoue@ieee.org

J. Kawahara
Graduate School of Information Science, Nara Institute of Science and Technology, Nara 630-0192, Japan

H. Iwashita · S. Minato
Graduate School of Information Science and Technology, Hokkaido University, Hokkaido 060–0814, Japan

## 1 Introduction

A graph is a representation of a set of edges, each of which connects a pair of vertices. Graphs are often used as a mathematical model for a variety of problems. Researchers have developed many sophisticated graph libraries, but the design focuses on handling a small number of graphs. Thus they cannot work with very large sets of graphs, even though the set can grow exponentially with graph size since a graph with $N$ edges induces up to $2^N$ 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 large sets 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 leads to poor scalability, while Graphillion handles a set of graphs collectively without considering graphs individually. Graphillion concentrates on edge-induced subgraphs of a given (vertex-)labeled graph $G = (V, E)$, and a set of graphs is reduced to 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 (i.e., finding minimum or maximum weighted graphs), 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 [4,5,10,13]. 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 using C or C++ for high-performance numerical computation, and it is well-suited to scientific and engineering code [12]. However, Python objects must be reinterpreted in every extended function call (e.g., Python's built-in `set` object can reinterpret all elements in some function calls), and this overhead would be unacceptable if a very large graph set were involved. Graphillion, in contrast, deals with a whole graph set directly without considering individual graphs, and so only 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 for graph sets via Python's extension mechanism.

We evaluate the performance and productivity of Graphillion in experiments. We first measure the performance on 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 s (just one second for some operations). We then present two case studies, a puzzle solver and a power network optimizer, and reveal that Graphillion reduces the lines of code by 90 % with an 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 graph in just 1 min.

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 its implementation, and Sect. 6 reports the experiments and results. Section 7 summarizes related work, and Sect. 8 concludes the paper.

## 2 Overview

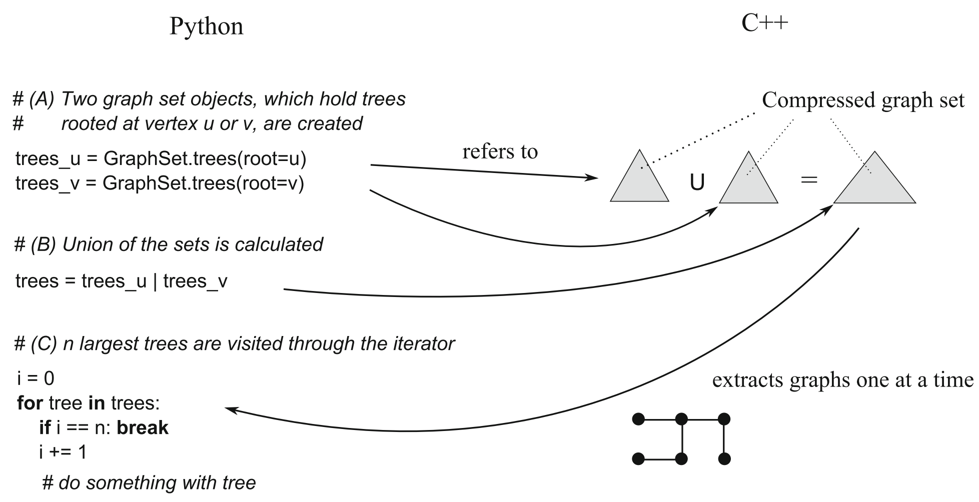
We describe below a design overview of Graphillion (Fig. 1), along with our goals, high performance and high productivity.

- **High performance** Graphillion processes very large sets of graphs efficiently in terms of both space and time. It is implemented as a Python module with C++ extensions. A set of graphs is represented in a compressed form of a C++ object, which is created by frontier search (Fig. 1a) and is manipulated by family algebra (Fig. 1b). Since only the reference to 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. 1c).
- **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. 1b, c), and so it is very easy 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.

[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.



```
# (A) Two graph set objects, which hold trees
#       rooted at vertex u or v, are created
trees_u = GraphSet.trees(root=u)
trees_v = GraphSet.trees(root=v)

# (B) Union of the sets is calculated
trees = trees_u | trees_v

# (C) n largest trees are visited through the iterator
i = 0
for tree in trees:
   if i == n: break
   i += 1
   # do something with tree
```

**Fig. 1** Overview of Graphillion 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

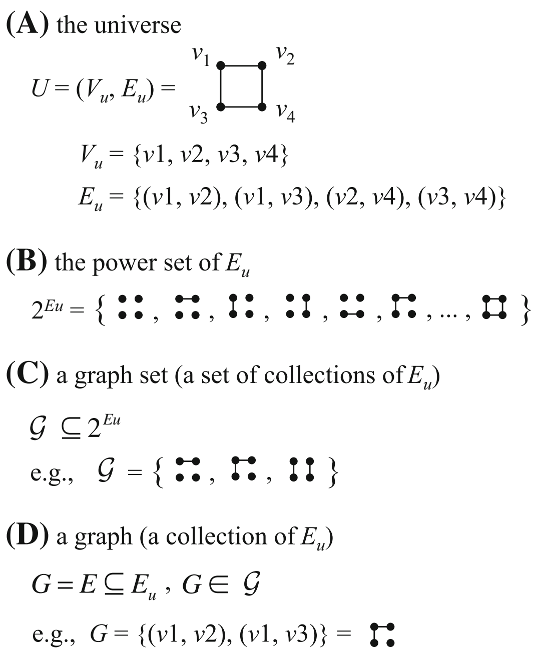

(A) the universe
$U = (V_u, E_u) =$
$v_1$
$v_2$
$v_3$
$v_4$
$V_u = \{v1, v2, v3, v4\}$
$E_u = \{(v1, v2), (v1, v3), (v2, v4), (v3, v4)\}$
(B) the power set of $E_u$
$2^{Eu} = \{ , , , , , , ..., \}$
(C) a graph set (a set of collections of $E_u$)
$\mathcal{G} \subseteq 2^{Eu}$
e.g., $\mathcal{G} = \{ , , \}$
(D) a graph (a collection of $E_u$)
$G = E \subseteq E_u$, $G \in \mathcal{G}$
e.g., $G = \{(v1, v2), (v1, v3)\} =$

**Fig. 2** Examples of our graph representation

## 3.1 Representation of a graph

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

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

We can set an arbitrary labeled graph as the universe. A graph $G$ used in Graphillion must be an edge-induced subgraph of the universe (Fig. 2d),

$$G = E \subseteq E_u,$$

where edge collection $E$ alone defines the graph, and 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, and so this limitation is 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. 2b, 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]. A ZDD greatly compresses a very large set of collections without information loss, 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 of a grid graph versus the amount of memory needed to store them in a 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 be self-loops. Multiple edges can be placed between a same pair of vertices if they are distinguishable. Edges can be hyper edges, which can 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] |
|---|---|---|
| $2 \times 2$ | 10 | 990 |
| $3 \times 3$ | 750 | 9,870 |
| $4 \times 4$ | 7,37,354 | 61,830 |
| $5 \times 5$ | 8,96,59,81,766 | 3,35,190 |
| $6 \times 6$ | 1,33,41,22,53,35,91,284 | 23,64,750 |
| $7 \times 7$ | 2,41,75,10,62,60,51,12, 71,73,092 | 1,81,68,510 |
| $8 \times 8$ | 53,14,03,15,31,28,26,65,03, 00,53,06,20,174 | 5,63,21,790 |
| $9 \times 9$ | 1,41,30,43,45,22,30,40,66, 55,78,92,21,37,31,29,70, 09,012 | 20,71,15,950 |

## 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] Frontier search finds all graphs that have a specified structure.

It outputs the enumerated graphs in a compressed form that is easily converted into a ZDD [15]. The time complexity is determined by the size of the compressed form (slightly larger than that of corresponding ZDD), not by the number of graphs being output.

We briefly describe frontier search. Consider a tree that represents a set of graphs. On the tree, a node of depth $i$ corresponds to $i$-th edge of universe ($e_i \in E_u$), and a branch incident from the node is labeled to indicate whether the $i$-th edge is included to the collection ($e_i \in E$). A path from the root to a leaf corresponds to an edge collection ($E \subseteq E_u$), and a leaf indicates whether the path is included to the set. Two tree nodes can be shared if their subtrees are identical, which compresses the tree into a directed acyclic graph. Frontier search constructs such a directed acyclic graph by examining the universe graph without backtracking. A branch is pruned if all the paths through the branch cannot lead to the specified structure.

Frontier search was originally limited to simple structures like trees, but it has been generalized to support various structures [5,10]. Table 2 shows the structures supported by Graphillion. The search space can be restricted to a given graph set; graphs not included in the given set are not enumerated by frontier search [4].

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

**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, Hamiltonian or not |
| Cycle | Hamiltonian or not |
| Clique | Size |
| Connected component | Vertices to be connected |

**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 | $\mathcal{G}_1 \curvearrowleft \mathcal{G}_2 = \{G_1 \in \mathcal{G}_1 \mid \exists G_2 \in \mathcal{G}_2 (G_1 \subseteq G_2)\}$ |
| Supergraphs | $\mathcal{G}_1 \curvearrowright \mathcal{G}_2 = \{G_1 \in \mathcal{G}_1 \mid \exists G_2 \in \mathcal{G}_2 (G_1 \supseteq G_2)\}$ |
| 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 | $\mathcal{G}^{\downarrow} = \{G_1 \in \mathcal{G} \mid G_2 \in \mathcal{G} \wedge G_1 \supseteq G_2 \rightarrow G_1 = G_2\}$ |

Some simple sets of graphs can be created by ZDD's primitives without frontier search: e.g., the empty set and the 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 directly on the compressed data, 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 makes sense for graph sets without change. The first four operations in Table 3 are ordinary set operations. Each graph in a 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 $\mathcal{G}$ (Fig. 3a), check,

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

The other four operations in the table select graphs based on their structures. They do what their names suggest (they

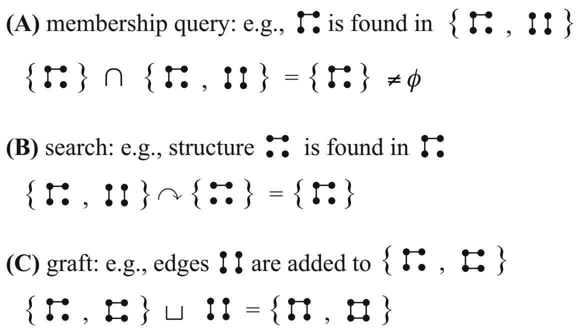

(A) membership query: e.g., is found in
(B) search: e.g., structure is found in
(C) graft: e.g., edges are added to

**Fig. 3** Examples of graph set manipulation via family algebra

**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 $\boxplus$) | $\mathcal{G} \boxplus \{E\} = \{G \oplus E \mid G \in \mathcal{G}\}$ |

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 $G$ (Fig. 3b), look at,

$$\mathcal{G} \curvearrowright \{G\}.$$

We now move on 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 the *join* operation defined in the family algebra, as shown in Table 4 (Fig. 3c). Similarly, edge(s) $E$ can be removed by performing the *meet* operation against the complement edge set $E^c = E_u \backslash E$ (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 the 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 $\mathcal{G}$
  $G$ = 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 here due to limited space.

## 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 (e.g., `union`). 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++ (see 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 at a time 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 between 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., a 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 consider 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.

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

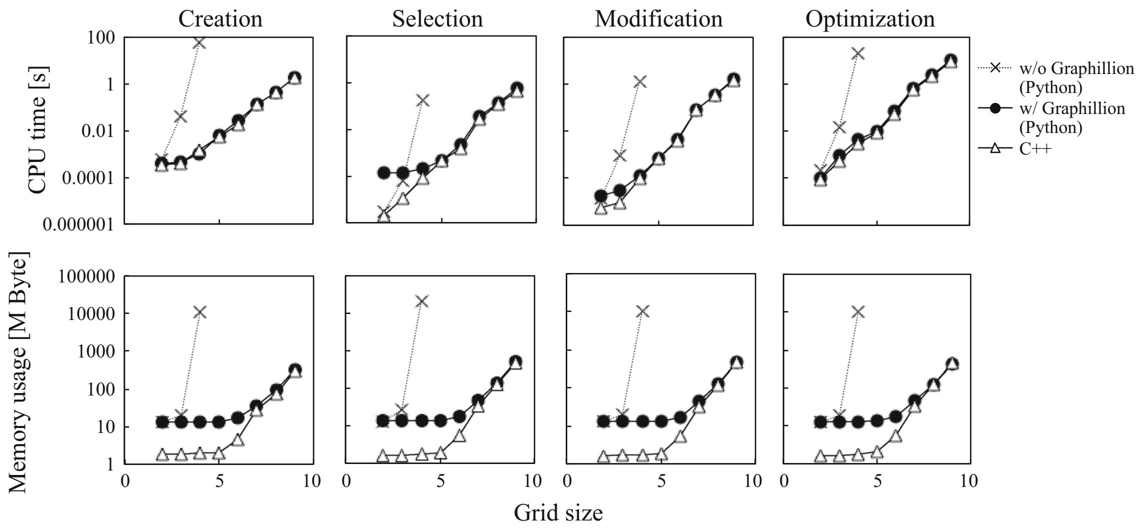


**Fig. 4** CPU time and memory usage for the basic operations with and without Graphillion. The operations are executed over trees rooted at a corner on a grid graph. The grid size on the horizontal axis indicates $N$ of the $N \times N$ grid

## 6.1 Basic performance

We evaluate the performance by using a set of trees rooted at a corner on a grid graph. The set size is shown in Table 1. Creation performance is measured by building a set of the trees. 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. Modification performance is evaluated by grafting an edge to all trees. Finally, optimization performance is measured by finding the top-3 weighted trees with the maximization 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 a 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 $5 \times 5$ grid within an hour due to the very large number of trees. Graphillion performs somewhat poorly on the small grids due to the overhead of object mapping and conversion, but the overhead is negligible for the larger grids. Graphillion finished all operations in less than 10 seconds with 500 MB of memory even for the $9 \times 9$ grid, which has $10^{37}$ trees. Examining the performance of Graphillion's operations in detail, we found that ZDDs accounted for most of the memory usage in the larger grids, while Python's initial memory becomes dominant for the smaller grids (this makes the memory usage nearly constant among the smaller grids). Selection required the largest memory capacity since it involves two sets of trees,[6] Creation consumes the least memory; this is because frontier search builds a ZDD directly, while the other operations have to rely on interim results of family algebra. Optimization took the longest CPU time, because it calls the "difference" operation at every iteration as discussed in Sect. 4.2. CPU times of other operations are ruled by ZDD size, though they differ slightly according to the operation's complexities; e.g., selection is the simplest and fastest, while creation requires a little more time due to frontier search.

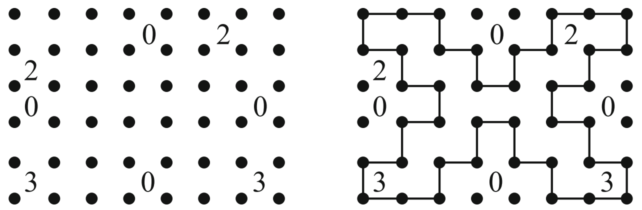


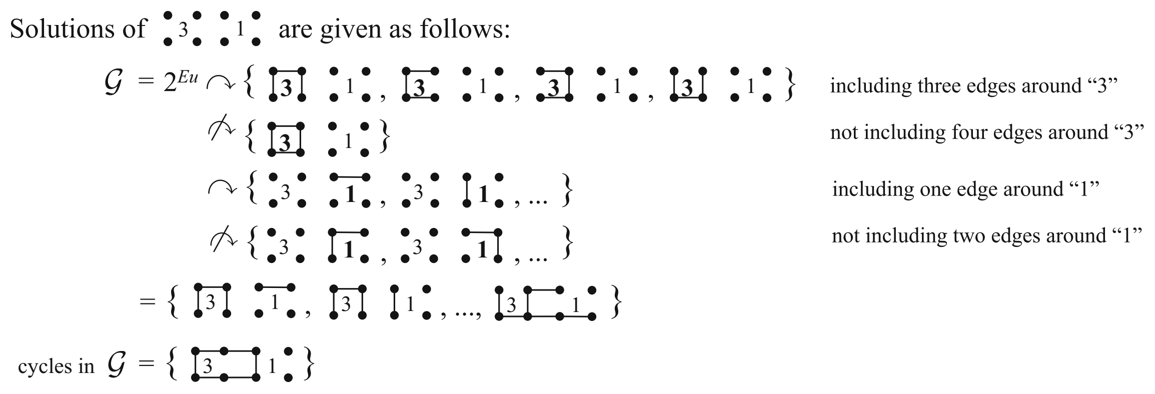
**Fig. 5** An example of the Slitherlink problem (*left*) and its solution (*right*) on $6 \times 8$ grid; adjacent dots are connected with *vertical* or *horizontal lines*, and a cycle is formed satisfying given hints, which indicate the number of lines surrounding it while empty cells may be surrounded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

## 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 have previously developed a Slitherlink solver [19]; it was the fastest solver that could list *all* solution cycles. The

[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.

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



Solutions of 3 1 are given as follows:
including three edges around "3"
not including four edges around "3"
including one edge around "1"
not including two edges around "1"
cycles in

**Fig. 6** An example of Slitherlink solution by Graphillion. Here, we define $\mathcal{G}_1 \not\curvearrowright \mathcal{G}_2 = \mathcal{G}_1 \backslash (\mathcal{G}_1 \curvearrowright \mathcal{G}_2)$. 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)

**Table 5** Lines of code for Slitherlink solvers

| Implementation | C++ | Python |
|---|---|---|
| w/o Graphillion | 2,116 | 0 |
| w/ Graphillion | 0 | 153 |

solver employs frontier search redesigned for Slitherlink and has special algorithms to process hints. The solver consists of 2,116 lines of C++ code.

We developed another solver with Graphillion but without a dedicated frontier search 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 consists of just 153 lines of Python. This is a 93 % reduction in the number of code lines, and it is, in addition, written in easy Python, rather than in complicated C++ (Table 5).[8]

We measured the CPU time and memory usage for 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 were 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 terms of CPU time due to the special algorithms in the dedicated solver, but the tradeoff between performance and productivity is acceptable.

[8] We should be careful when comparing the numbers of lines between Python and C++, since they have different grammars and syntaxes: e.g., Python does not require opening and closing brackets while C++ does. However, in our experiments, the average line length in C++ is not significantly shorter than that in Python (26.0 characters per line in C++ compared with 32.2 characters in Python). Moreover, they both use the object-oriented programming model. We, therefore, believe that our evaluation roughly measures the productivity.

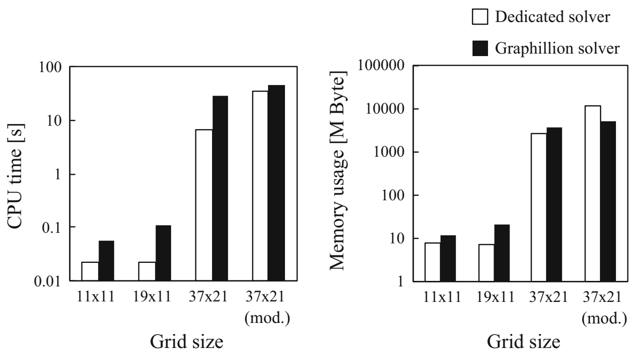


**Fig. 7** CPU time and memory usage of the dedicated solver and the Graphillion solver on Slitherlink problems

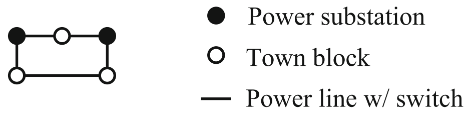


**Fig. 8** An example of power distribution network, which is represented by a graph; the power flow can be configured by the switches

We can obtain the top-$k$ longest or shortest cycles with Graphillion's iterators, when the problem has multiple solution cycles. It took just another 0.24 s to find the three longest cycles from among the 117059496 solutions to 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 and an edge corresponds to a power line with a switch (Fig. 8). The power flow is configured by changing the open/closed

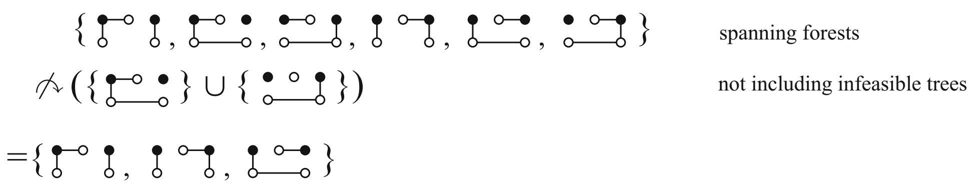


**Fig. 9** An example of optimization algorithms for power networks in Fig. 8; feasible solutions are obtained as the spanning forests with no infeasible trees, and then the optimal one is determined (not shown in the figure)

status 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 previous 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-infeasible 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 (the one in 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, consisted of 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 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 consists 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 the number of code lines (Table 6).

The two optimizers were compared for the 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 terms of 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 to its 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.

Additionally, Graphillion can be used as a graph database of feasible forests. We issued queries specifying an open or closed switch to select all the forests matching the queries, as shown in Fig. 3b. Graphillion processed the queries within just 1.5 s for a closed switch and within 0.5 s for an open switch in the largest network.

**Table 6** Lines of code for power network optimizers

| Implementation | C++ | Python |
|---|---|---|
| w/o Graphillion | 6,856 | 1,221 |
| w/ Graphillion | 0 | 1,164 |

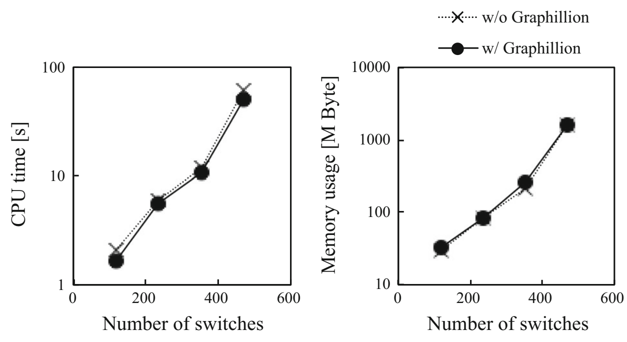


**Fig. 10** CPU time and memory usage of power network optimizers with and without Graphillion

## 7 Related work

There are several existing graph libraries, including NetworkX [2] and Boost Graph Library [14], which are widely used for graph analysis. They are, however, designed for a small number of graphs or a simple power set of edges: that is, they can find a shortest path from just a power set of edges without constraints. In contrast, Graphillion can find shortest paths from a large and complex set of graphs;

given a constrained set of graphs, which could be created by Graphillion operations, we first select paths from the constrained set (see Sect. 4.1) and then find minimum weighted paths from them (see Sect. 4.2).

We often use general optimizers like CPLEX[9] 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 the top-$k$ solutions.

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

VSOP [9] employs family algebra like 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 utilize Python's rich collection of 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 showed that programmers can handle very large graph sets with just a small number of lines of code. Graphillion can also be used for railway analysis.[10]

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.

Since we would like to discover more applications for which Graphillion works well, we make it publicly available online at Graphillion's page[11] and PyPI (the Python Package Index).[12]

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

[10] http://www.nysol.jp/en/home/apps/ekillion.

## References

1. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1:1–1:39 (2008)
2. Hagberg, A., Swart, P., S Chult, D.: Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy 2008), pp. 11–16 (2008)
3. Inoue, T., Takano, K., Watanabe, T., Kawahara, J., Yoshinaka, R., Kishimoto, A., Tsuda, K., Minato, S., Hayashi, Y.: Distribution loss minimization with guaranteed error bound. IEEE Trans. Smart Grid **5**(1), 102–111 (2014)
4. Iwashita, H., Minato, S.: Efficient top-down ZDD construction techniques using recursive specifications. Tech. rep., Hokkaido University, Division of Computer Science, TCS Technical Reports, TCS-TR-A-13-69 (2013). http://www-alg.ist.hokudai.ac.jp/tra.html
5. Kawahara, J., Inoue, T., Iwashita, H., ichi Minato, S.: Frontier-based search for enumerating all constrained subgraphs with compressed representation. Tech. rep., Hokkaido University, Division of Computer Science, TCS Technical Reports, TCS-TR-A-14-76 (2014). http://www-alg.ist.hokudai.ac.jp/tra.html
6. Knuth, D.E.: 7.1.4 Binary Decision Diagrams, vol. 4A. Addison-Wesley, USA (2011)
7. Lavaei, J., Rantzer, A., Low, S.: Power flow optimization using positive quadratic programming. In: Proceedings of the 18th IFAC World Congress (2011)
8. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of Conference on Design Automation, pp. 272–277 (1993)
9. Minato, S.: VSOP (valued-sum-of-products) calculator for knowledge processing based on zero-suppressed BDDs. Federation over the Web. Lecture Notes in Computer Science, vol. 3847, pp. 40–58. Springer, Berlin, Heidelberg (2006)
10. Minato, S.: Techniques of BDD/ZDD: brief history and recent activity. IEICE Trans. Inf. & Syst. E96-D(7) (2013)
11. Nikoli: Slitherlink 1 (1992)
12. Oliphant, T.E.: Python for scientific computing. Comput. Sci. Eng. **9**(3), 10–20 (2007)
13. Sekine, K., Imai, H., Tani, S.: Computing the Tutte polynomial of a graph of moderate size. In: Algorithms and Computations, Lecture Notes in Computer Science, vol. 1004, pp. 224–233. Springer (1995)
14. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Professional, USA (2001)

[11] http://graphillion.org/.

[12] http://pypi.python.org/.

15. Sieling, D., Wegener, I.: Reduction of OBDDs in linear time. Inform. Process. Lett. **48**(3), 139–144 (1993)
16. Toda, T.: Hypergraph transversal computation with binary decision diagrams. In: Experimental Algorithms, Lecture Notes in Computer Science, vol. 7933, pp. 91–102. Springer (2013)
17. Walt, S.V.D., Colbert, S., Varoquaux, G.: The NumPy array: a structure for efficient numerical computation. Comput. Sci. Eng. **13**(2), 22–30 (2011)
18. Yoshinaka, R., Kawahara, J., Denzumi, S., Arimura, H., Minato, S.: Counterexamples to the long-standing conjecture on the complexity of BDD binary operations. Inform. Process. Lett. **112**(16), 636–640 (2012)
19. Yoshinaka, R., Saitoh, T., Kawahara, J., Tsuruma, K., Iwashita, H., Minato, S.: Finding all solutions and instances of numberlink and slitherlink by ZDDs. Algorithms **5**(2), 176–213 (2012)