graphillion - 莫大な数の部分グラフを扱う Python ライブラリ

奈良先端科学技術大学院大学
川原　純

謝辞
本スライドで使用している図の一部と全適用事例は発売予定の書籍
「超高速グラフ列挙アルゴリズム」(仮題)
のドラフト版から引用しています。図の引用は「ZDD書籍から引用」と明示しています。
graphillion ライブラリの作者　井上 武 さん, 岩下 洋哲 さん
書籍の著者　湊 真一 先生, 斎藤 寿樹 先生, 安田 宜仁さん, 井上 武 さん,
羽室 行信 先生, 前川 浩基 さん, 丸橋 弘明 さん, 堀山 貴史 先生, その他の著者の皆様,
JST ERATO 湊離散構造処理系プロジェクトメンバーの皆様に感謝いたします。

今日の内容

- graphillion ライブラリの紹介
  - graphillion ライブラリでできること
  - graphillion の内部データ構造
    - ZDD
  - graphillion が使用しているアルゴリズム
    - フロンティア法
  - 適用事例紹介 (1), (2), (3)

graphillion とは

http://graphillion.org
NTT研究所　井上 武 氏 作

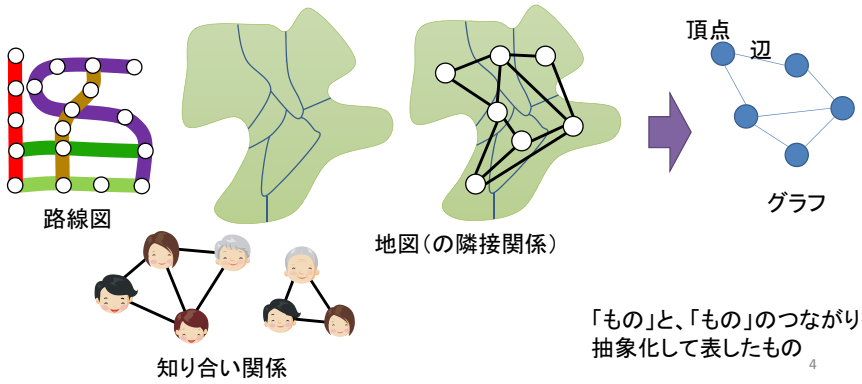
- 部分グラフ列挙ツール
- グラフに関する様々な最適化問題を解く

graphillion とは

http://graphillion.org
NTT研究所　井上 武 氏 作

- 部分グラフ列挙ツール
- グラフに関する様々な最適化問題を解く

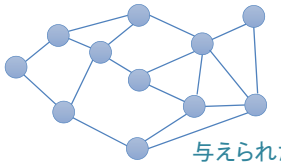
グラフとは



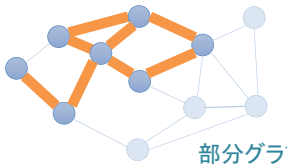
「もの」と、「もの」のつながりを
抽象化して表したもの

部分グラフ

- 与えられたグラフの一部分からなるグラフ

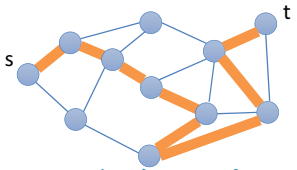


与えられたグラフ

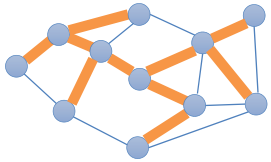


部分グラフの例

- 経路（パス）や全域木などは部分グラフ



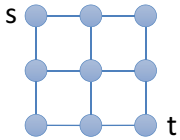
s-t 経路（s-t パス）



全域木

部分グラフ列挙

- 与えられた条件を満たす部分グラフをすべて求める
- 例:

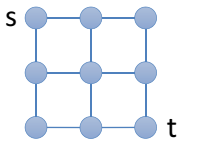


s から t までの、
同じ頂点を通らない経路は？

部分グラフ列挙

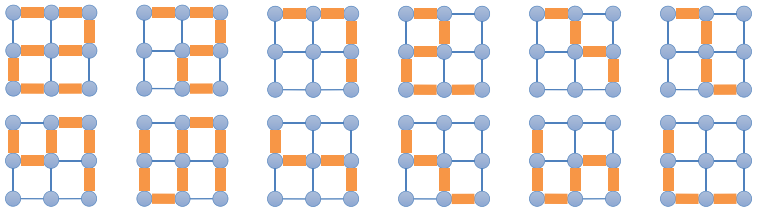
- 与えられた条件を満たす部分グラフをすべて求める
- 例:

全体グラフ (universe)



s から t までの、
同じ頂点を通らない経路は？

部分グラフ

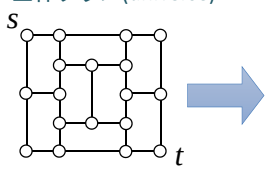


部分グラフ列挙

列挙して何が嬉しいのか？
➡ 後述

- 与えられた条件を満たす部分グラフをすべて求める
- 例:

全体グラフ (universe)



graphillion を使ってみよう(1)

- インストール (Mac, Linux)

```
$ easy_install graphillion
```

または、
http://graphillion.org
からダウンロード、ビルド

- python インタプリタを起動

```
$ python
```

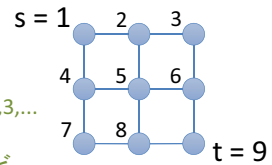
- ライブラリのインポート

```
>>> from graphillion import GraphSet
```


graphillion を使ってみよう(2)

- グラフの表現
  - networkx ライブラリと同じ

全体グラフ (universe)



頂点: オブジェクト（何でもよい） 本講演では1,2,3,...

辺: 頂点のタプル (1, 2) や (2, 3) など

グラフ: 辺のリスト [(1, 2), (1, 4), (2, 3), (2, 5), (3, 6), (4, 5), (4, 7), (5, 6), (5, 8), (6, 9), (7, 8), (8, 9)]

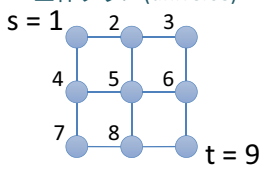
- 全体グラフの設定

```
>>> GraphSet.set_universe( [(1, 2), (1, 4), (2, 3), (2, 5),
(3, 6), (4, 5), (4, 7),(5, 6), (5, 8), (6, 9), (7, 8), (8, 9)] )
```


graphillion を使ってみよう(3)

- s-t 経路を列挙する

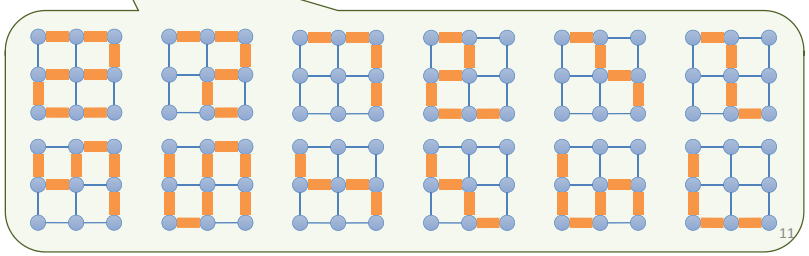
全体グラフ (universe)




```
>>> paths = GraphSet.paths(1, 9)
```

引数に始点と終点

paths 変数に、部分グラフの集合が格納される



graphillion でできること(1)

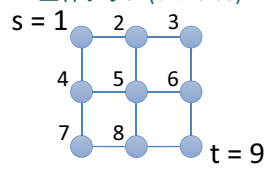
- 数のカウント

```
>>> print paths.len()
12
```

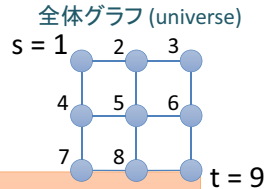
- 各部分グラフに対する処理

```
>>> for p in paths:
...     print p
```

全体グラフ (universe)



graphillion でできること(1)



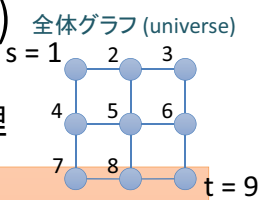
- 数のカウント

```
>>> print paths.len()
12
```

- 各部分グラフに対する処理

```
>>> for p in paths:
...     print p
...
[(1, 2), (2, 3), (3, 6), (4, 5), (4, 7), (5, 6), (7, 8), (8, 9)]
[(1, 2), (2, 3), (3, 6), (5, 6), (5, 8), (8, 9)]
[(1, 2), (2, 3), (3, 6), (6, 9)]
[(1, 2), (2, 5), (4, 5), (4, 7), (7, 8), (8, 9)]
[(1, 2), (2, 5), (5, 6), (6, 9)]
[(1, 2), (2, 5), (5, 8), (8, 9)]
[(1, 4), (2, 3), (2, 5), (3, 6), (4, 5), (6, 9)]
[(1, 4), (2, 3), (2, 5), (3, 6), (4, 7), (5, 8), (6, 9), (7, 8)]
[(1, 4), (4, 5), (5, 6), (6, 9)]
[(1, 4), (4, 5), (5, 8), (8, 9)]
[(1, 4), (4, 7), (5, 6), (5, 8), (6, 9), (7, 8)]
[(1, 4), (4, 7), (7, 8), (8, 9)]
```

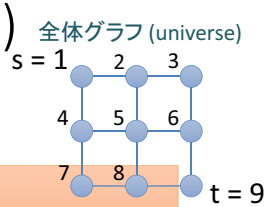

graphillion でできること(2)



- 経路の短い順に、各部分グラフを処理

```
>>> for p in paths.min_iter():
...     print p
...
[(1, 4), (4, 7), (7, 8), (8, 9)]
[(1, 4), (4, 5), (5, 8), (8, 9)]
[(1, 4), (4, 5), (5, 6), (6, 9)]
[(1, 2), (2, 5), (5, 8), (8, 9)]
[(1, 2), (2, 5), (5, 6), (6, 9)]
[(1, 2), (2, 3), (3, 6), (6, 9)]
[(1, 4), (4, 7), (5, 6), (5, 8), (6, 9), (7, 8)]
[(1, 4), (2, 3), (2, 5), (3, 6), (4, 5), (6, 9)]
[(1, 2), (2, 5), (4, 5), (4, 7), (7, 8), (8, 9)]
[(1, 2), (2, 3), (3, 6), (5, 6), (5, 8), (8, 9)]
[(1, 4), (2, 3), (2, 5), (3, 6), (4, 7), (5, 8), (6, 9), (7, 8)]
[(1, 2), (2, 3), (3, 6), (4, 5), (4, 7), (5, 6), (7, 8), (8, 9)]
```


graphillion でできること(3)

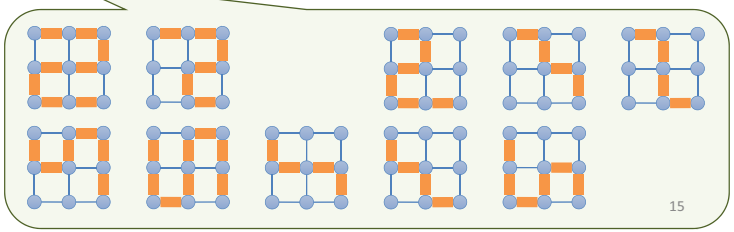


- (一様)ランダムに1つ抽出

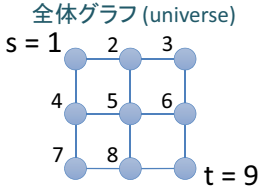
```
>>> p = paths.choice()
>>> print p
[(1, 2), (2, 3), (3, 6), (4, 5), (4, 7), (5, 6), (7, 8), (8, 9)]
```

- 頂点5を通る経路のみを抽出

```
>>> paths2 = paths.including(5)
```

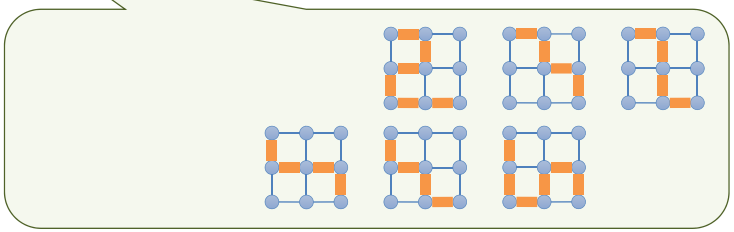


graphillion でできること(4)



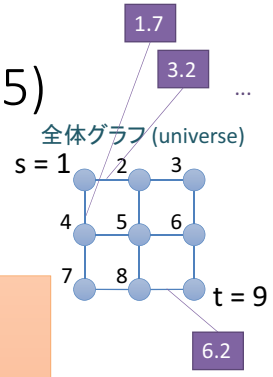
- さらに、頂点3を通らない経路のみを抽出

```
>>> paths3 = paths2.excluding(3)
```



graphillion でできること(5)

- 辺に重みをつけて、重みの和が最大・最小の部分グラフを求める




```
>>> weights = {}
>>> weights[(1, 2)] = 3.2
>>> weights[(1, 4)] = 1.7
# （略）
>>> weights[(8, 9)] = 6.2
>>>
>>> iter = paths3.max_iter(weights)
>>> p1 = iter.next()
>>> p2 = iter.next()
```

一番長い経路
二番目に長い経路

辺の重みは、辺をキー、重みをバリューとするディクショナリで与える

graphillion の威力(1)

データ取得: 駅データ.jp
http://ekidata.jp

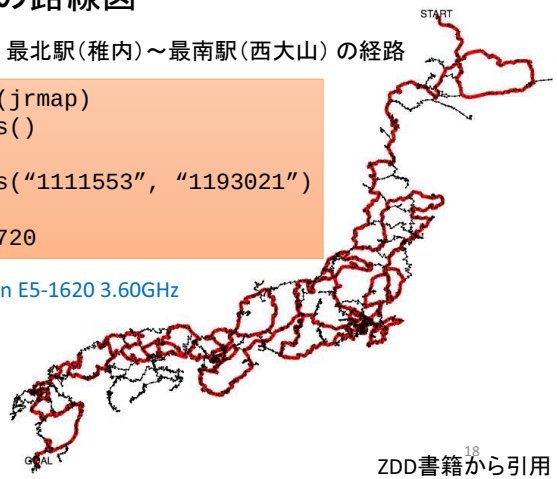
- JR（Japan Railway）の路線図

頂点数: 4534, 辺数: 4646　　最北駅（稚内）～最南駅（西大山）の経路

```
>>> GraphSet.set_universe(jrmap)
>>> weights = read_weights()
>>>
>>> paths = GraphSet.paths("1111553", "1193021")
>>> paths.len()
111287053969250364961151872O
```

計算時間: 258.16 秒　　Intel Xeon E5-1620 3.60GHz
使用メモリ: 12.4 GB

経路は圧縮して保持されている



ZDD書籍から引用

graphillion の威力(2)

- JR（Japan Railway）の路線図

頂点数: 4534, 辺数: 4646　　最北駅（稚内）～最南駅（西大山）の経路

最長経路を求める

```
>>> max_path = paths.max_iter(weights).next()
>>> print len(max_path)
2650
>>> sum = 0.0
>>> for edge in max_path:
...     sum += weights[edge]
...
>>> print sum
10337.3
```

駅数 − 1

総距離

最短経路も求まるが、ダイクストラ法などの既存アルゴリズムの方が速い

ZDD書籍から引用

graphillion で扱える部分グラフの種類

- 経路
  - s-t 経路
  - ハミルトン経路（すべての頂点を通る）
  - サイクル（1周して戻る）
- 木
  - 森、全域森
  - 木、全域木
  - 連結成分
- マッチング
- クリーク（頂点同士がすべて結ばれている頂点集合）
- その他、様々な制約を課した部分グラフ

今日の内容

- graphillion ライブラリの紹介
  - graphillion ライブラリでできること
  - graphillion の内部データ構造
    - ZDD
  - graphillion が使用しているアルゴリズム
    - フロンティア法
  - 適用事例紹介 (1), (2), (3)

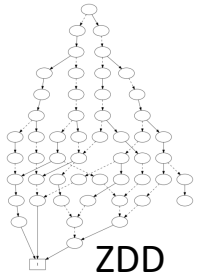
graphillion で用いられるデータ構造

ゼロサプレス型二分決定グラフ

- Zero-suppressed Binary Decision Diagram (ZDD) [S.Minato 93]
- 集合族（集合の集合）をコンパクトに効率良く記憶

{e1, e2, e5}, {e1, e3, e4, e7}, {e1, e5, e8, e12},
{e3, e7, e9}, {e2, e6, e7, e8, e9, e10}, {e4},
{e1, e4, e6, e8}, {e1, e2, e5, e8}, {e1, e9, e10},
{e4, e5, e9}, {e1, e3, e4, e5, e9, e11}, {e6, e7},
{e1, e4, e5}, {e1, e5, e8, e10}, {e2, e5, e11},
{e5, e8, e9}, {e2, e3, e7, e8}, {e10, e11, e12},
…

特殊な形をしたグラフ

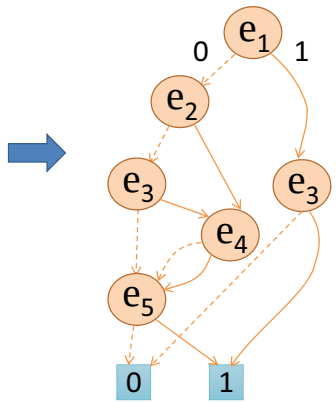


ZDD

ZDD の基本

$\{ \{e_1 e_3\}, \{e_2 e_5\}, \{e_2 e_4 e_5\}, \{e_3 e_4 e_5\}, \{e_3 e_5\}, \{e_5\} \}$

集合族

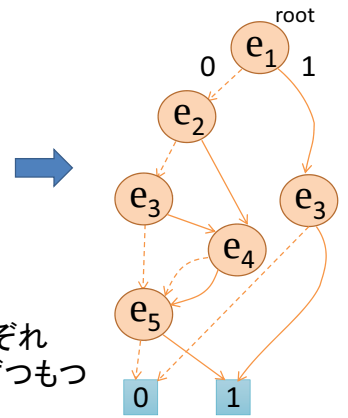


ZDD

要素は $\{e_1, e_2, e_3, e_4, e_5\}$ の5種類で固定
（事前に固定する必要あり）

ZDD の基本

$\{ \{e_1 e_3\}, \{e_2 e_5\}, \{e_2 e_4 e_5\}, \{e_3 e_4 e_5\}, \{e_3 e_5\}, \{e_5\} \}$

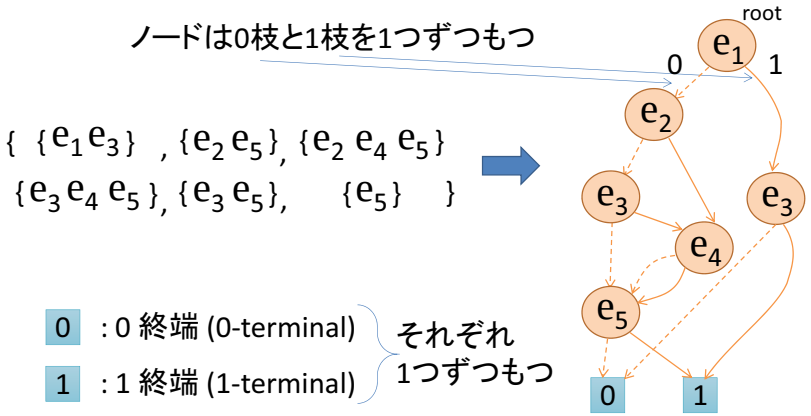


0 : 0 終端 (0-terminal)
1 : 1 終端 (1-terminal)
それぞれ1つずつもつ

e_i : ノード　$e_1 \sim e_5$ いずれかのラベル
$e_1, e_2, \ldots, e_5$ の順序

ZDD

ノードは0枝と1枝を1つずつもつ



$\{ \{e_1 e_3\}, \{e_2 e_5\}, \{e_2 e_4 e_5\}, \{e_3 e_4 e_5\}, \{e_3 e_5\}, \{e_5\} \}$

0 : 0 終端 (0-terminal)

1 : 1 終端 (1-terminal)

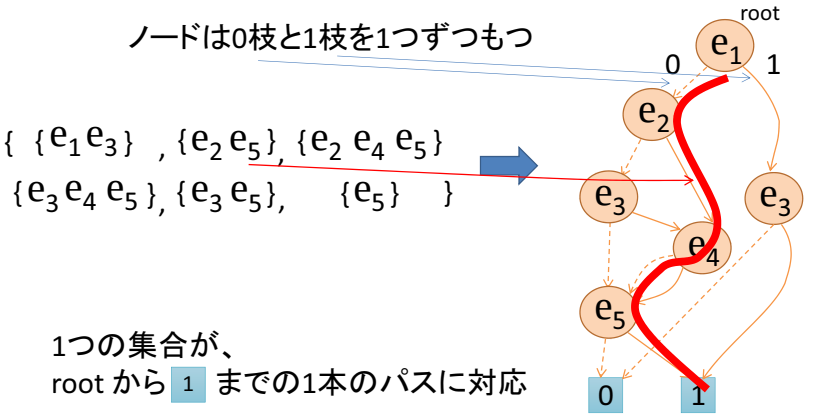
それぞれ1つずつもつ

e_i : ノード　$e_1 \sim e_5$ いずれかのラベル

$e_1, e_2, \ldots, e_5$ の順序

ZDD

ノードは0枝と1枝を1つずつもつ



$\{ \{e_1 e_3\}, \{e_2 e_5\}, \{e_2 e_4 e_5\}, \{e_3 e_4 e_5\}, \{e_3 e_5\}, \{e_5\} \}$

1つの集合が、
root から 1 までの1本のパスに対応

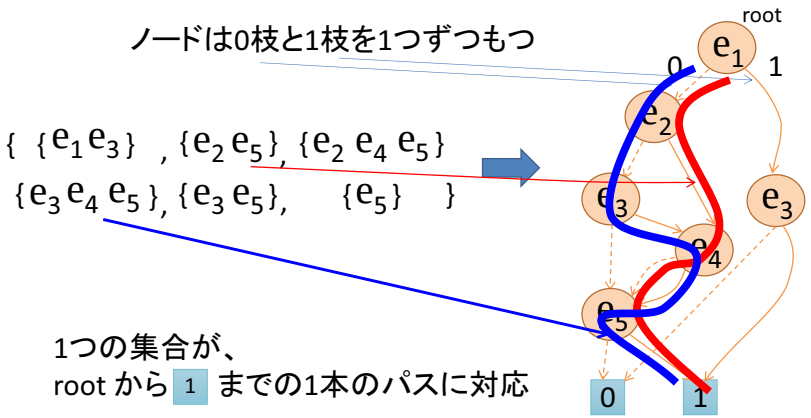
ZDD

0 : 0-terminal

1 : 1-terminal

e_i : node with label $e_1 \sim e_5$

ノードは0枝と1枝を1つずつもつ



$\{ \{e_1 e_3\}, \{e_2 e_5\}, \{e_2 e_4 e_5\}, \{e_3 e_4 e_5\}, \{e_3 e_5\}, \{e_5\} \}$

1つの集合が、
root から 1 までの1本のパスに対応

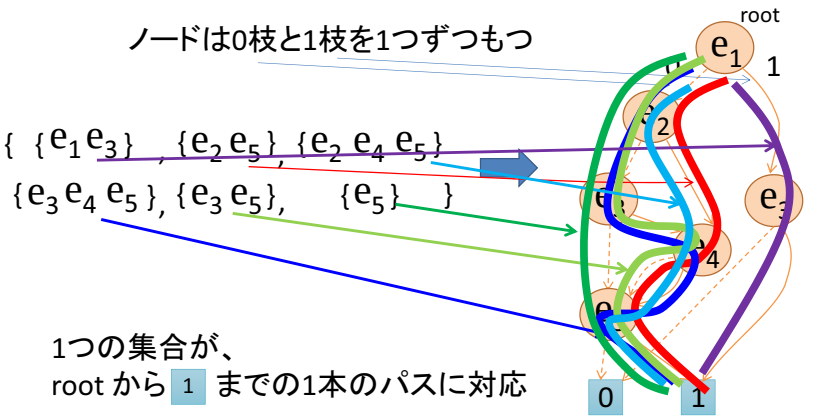
ZDD

0 : 0-terminal

1 : 1-terminal

e_i : node with label $e_1 \sim e_5$

ノードは0枝と1枝を1つずつもつ



$\{ \{e_1 e_3\}, \{e_2 e_5\}, \{e_2 e_4 e_5\}, \{e_3 e_4 e_5\}, \{e_3 e_5\}, \{e_5\} \}$

1つの集合が、
root から 1 までの1本のパスに対応

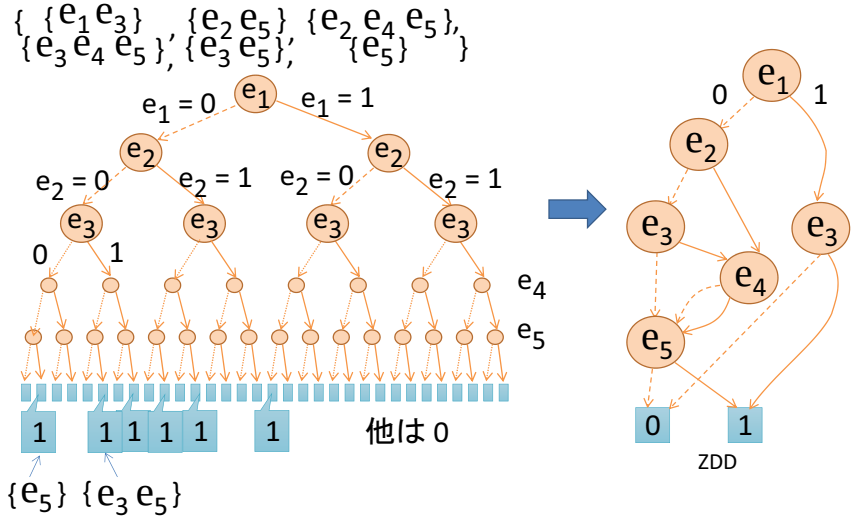
ZDD

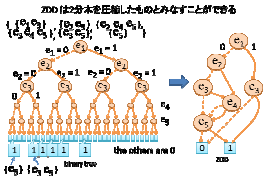
0 : 0-terminal

1 : 1-terminal

e_i : node with label $e_1 \sim e_5$

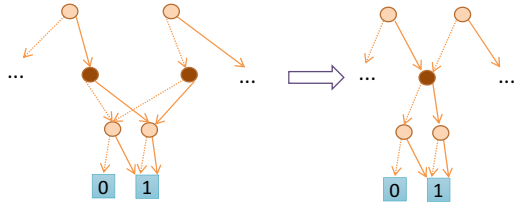
ZDD は2分木を圧縮したものとみなすことができる





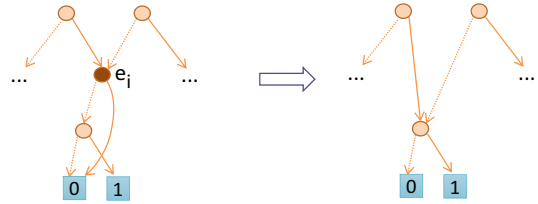
2種類のルールを
繰り返し適用して圧縮

ノード共有ルール



2つの「等価な」ノードは
共有される

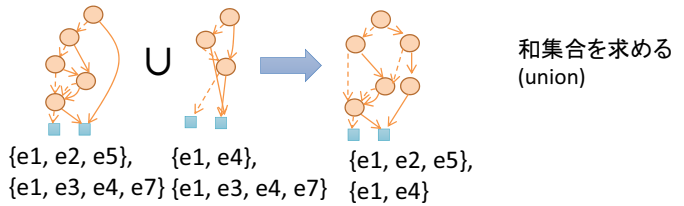
ゼロサプレスルール



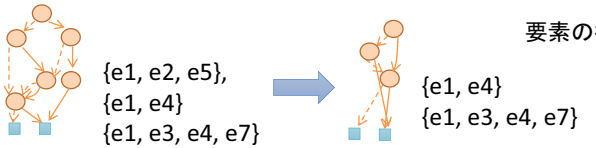
1-枝が0-Terminalを指す
ノードは削除される

ZDDの演算

- ZDD を用いると、集合演算や、要素の検索等、様々な演算が効率良く行える



和集合を求める
(union)



要素の抽出

e4 を含む
集合を取り出す

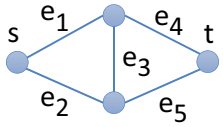
アルゴリズムは省略

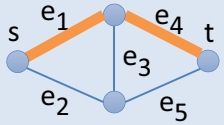
今日の内容

- graphillion ライブラリの紹介
  - graphillion ライブラリでできること
  - graphillion の内部データ構造
    - ZDD
  - graphillion が使用しているアルゴリズム
    - フロンティア法
  - 適用事例紹介 (1), (2), (3)

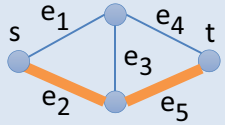
s-t 経路を例に説明

s-t 経路は辺の集合で表現できる

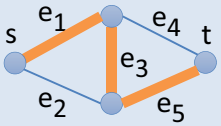




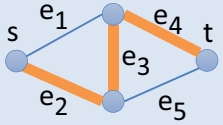
{e1, e4}



{e2, e5}

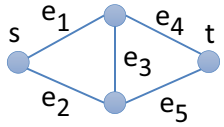


{e1, e3 ,e5}

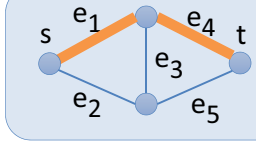


{e2, e3 ,e4}

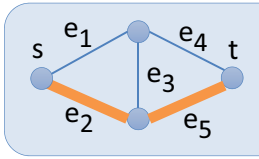
s-t 経路は辺の集合で表現できる



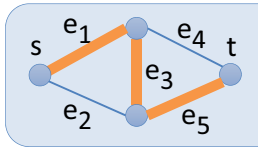
全ての s-t 経路を列挙して、
辺集合の集合で表す



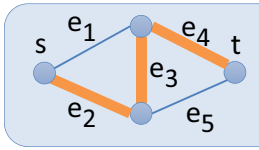
{e1, e4}



{e2, e5}

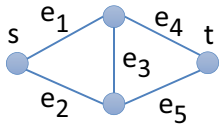


{e1, e3 ,e5}



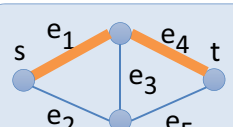
{e2, e3 ,e4}

s-t 経路は辺の集合で表現できる

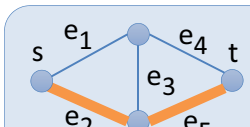


全ての s-t 経路を列挙して、
辺集合の集合で表す

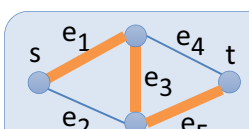
全 s-t 経路の集合 = {{e1, e4},{e2, e5},{e1, e3 ,e5},{e2, e3 ,e4}}



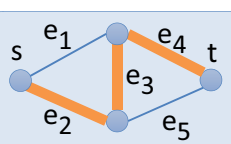
{e1, e4}



{e2, e5}

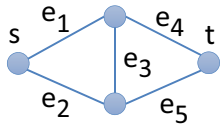


{e1, e3 ,e5}



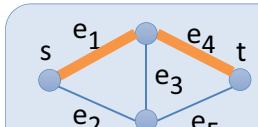
{e2, e3 ,e4}

s-t 経路は辺の集合で表現できる

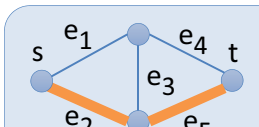


全ての s-t 経路を列挙して、
辺集合の集合で表す
これをZDDで表現する

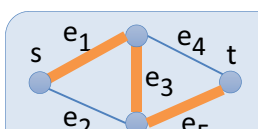
全 s-t 経路の集合 = {{e1, e4},{e2, e5},{e1, e3 ,e5},{e2, e3 ,e4}}



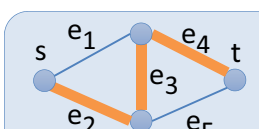
{e1, e4}



{e2, e5}

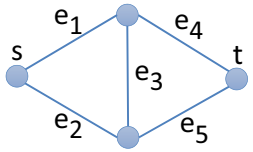


{e1, e3 ,e5}



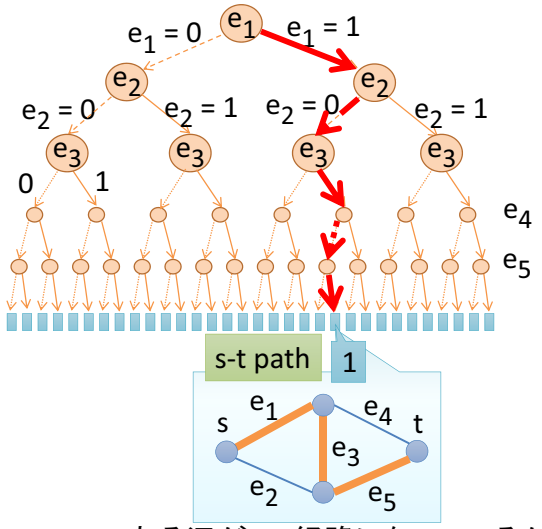
{e2, e3 ,e4}

経路列挙（ZDD構築）アルゴリズム [Knuth 08]



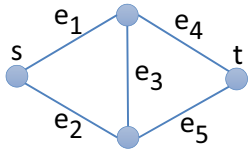
1. 辺に順番を付ける
（例えば、s から幅優先）
（もっと良い方法もあり）

2. ZDDを構築
辺 e_1, e_2,… の順に処理
各辺変数 e_i に対し、
$e_i = 0$ or 1 を決めていく



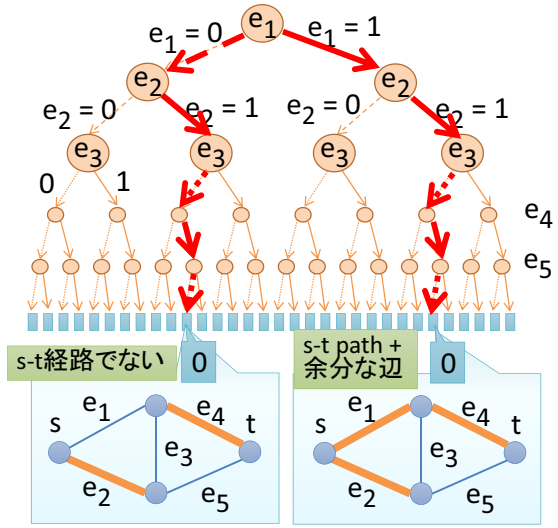
$e_i = 1$ である辺が s-t 経路になっているか

経路列挙（ZDD構築）アルゴリズム [Knuth 08]



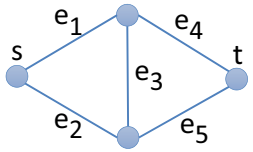
1. 辺に順番を付ける
（例えば、s から幅優先）
（もっと良い方法もあり）

2. ZDDを構築
辺 e_1, e_2,… の順に処理
各辺変数 e_i に対し、
$e_i = 0$ or 1 を決めていく



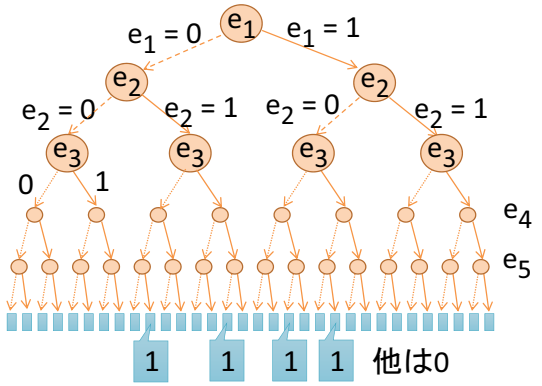
$e_i = 1$ である辺が s-t 経路になっているか？

経路列挙（ZDD構築）アルゴリズム [Knuth 08]

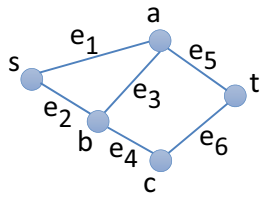


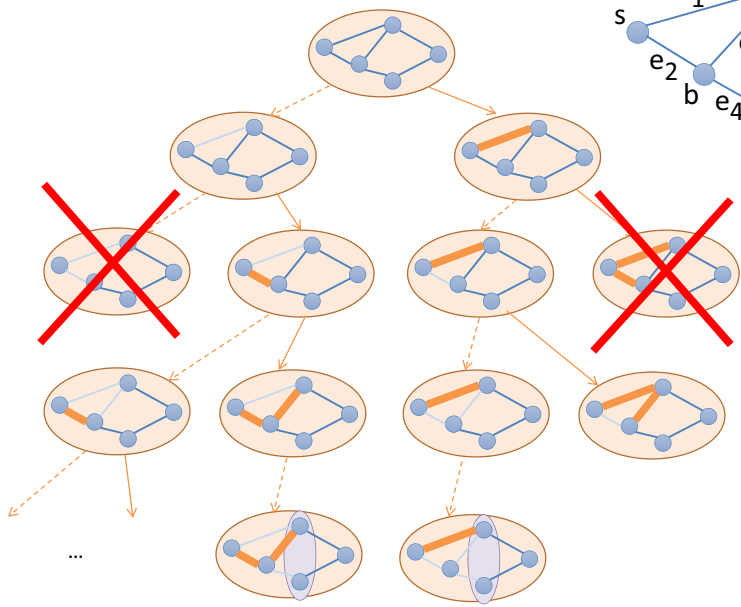
1. 辺に順番を付ける
（例えば、s から幅優先）
（もっと良い方法もあり）

2. ZDDを構築
辺 e_1, e_2,… の順に処理
各辺変数 e_i に対し、
$e_i = 0$ or 1 を決めていく

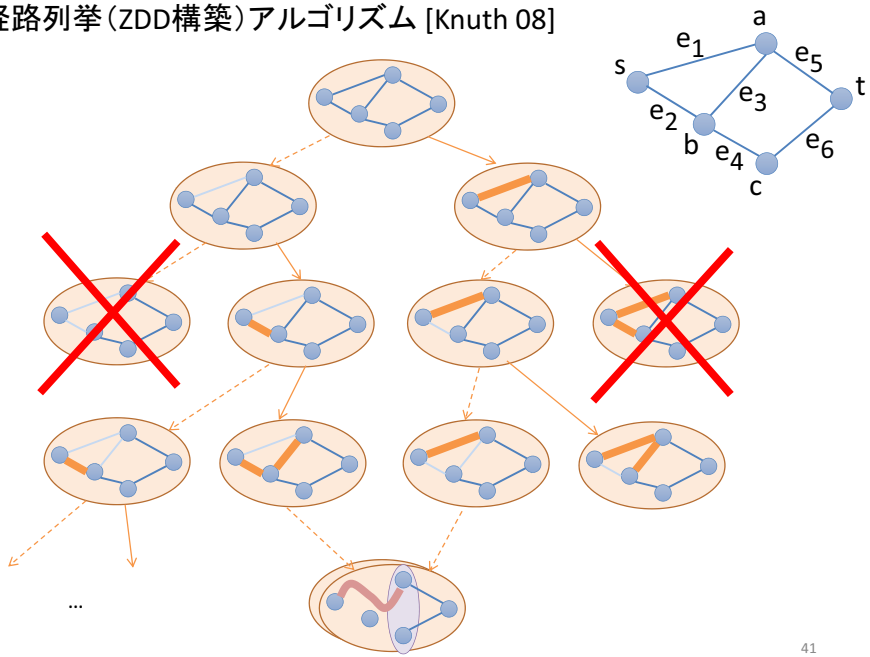


経路列挙（ZDD構築）アルゴリズム [Knuth 08]

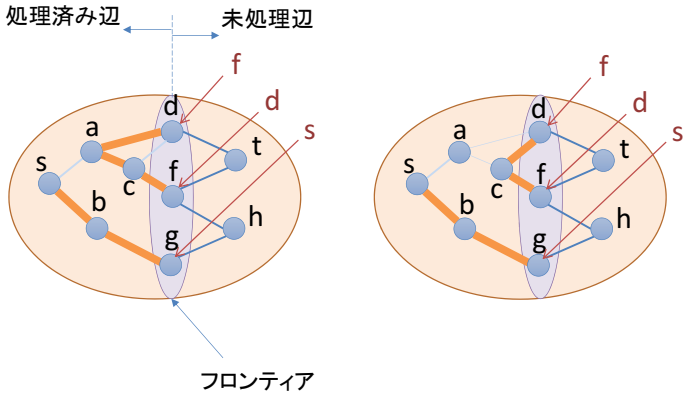




経路列挙（ZDD構築）アルゴリズム [Knuth 08]



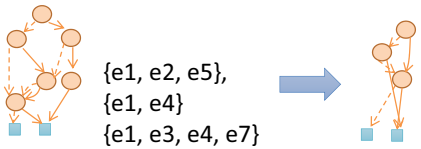
経路列挙（ZDD構築）アルゴリズム [Knuth 08]



部分経路の反対側の端点を記憶しておき、
フロンティア上ですべて一致するなら、
2つは同じ状態とみなす。

ZDDの形式で保存

- ZDDが構築できれば、経路の全列挙は簡単
  - top から 1-終端までたどればよい
- 構築したZDDはコンパクトなので、そのまま保持可能
- ZDD演算を行うことで、経路の条件付き検索が可能

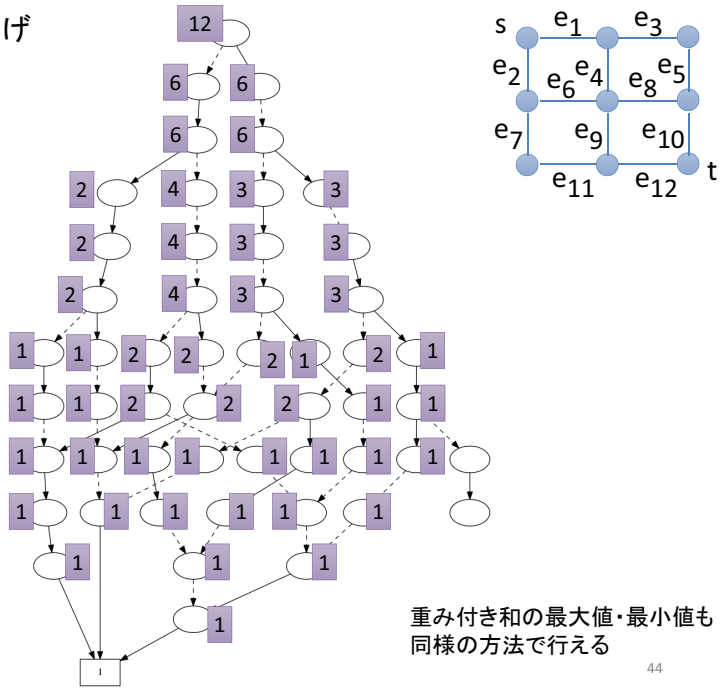


{e1, e2, e5},
{e1, e4}
{e1, e3, e4, e7}

{e1, e4}
{e1,e3, e4, e7}

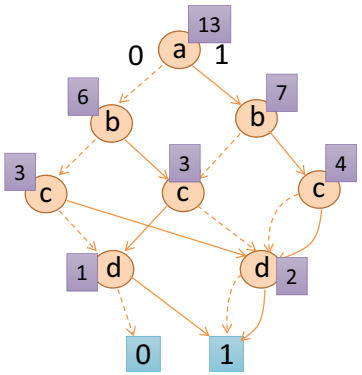
e4 を通る
経路のみを
取り出す

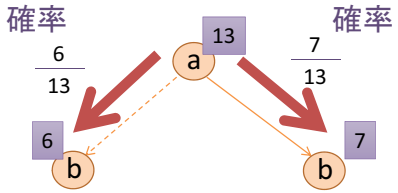
経路の数え上げ



重み付き和の最大値・最小値も
同様の方法で行える

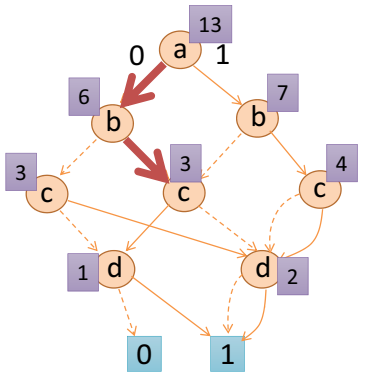
一様ランダムサンプリング

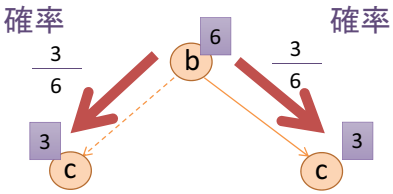




{d}, {c}, {c, d},
{b}, {b, d}, {b, c, d},
{a}, {a, d}, {a, c, d},
{a, b}, {a, b, c}, {a, b, d},
{a, b, c, d}

一様ランダムサンプリング

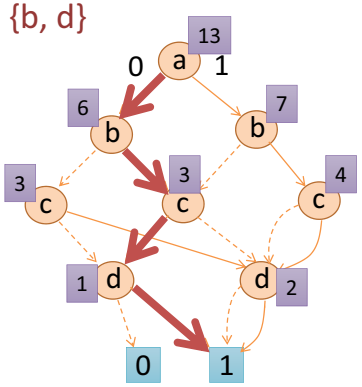




{d}, {c}, {c, d},
{b}, {b, d}, {b, c, d},
{a}, {a, d}, {a, c, d},
{a, b}, {a, b, c}, {a, b, d},
{a, b, c, d}

一様ランダムサンプリング

{b, d}



{d}, {c}, {c, d},
{b}, {b, d}, {b, c, d},
{a}, {a, d}, {a, c, d},
{a, b}, {a, b, c}, {a, b, d},
{a, b, c, d}

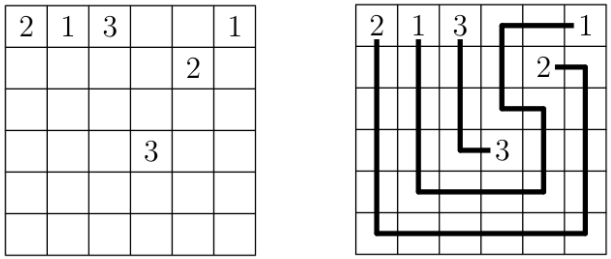
graphillion で扱える部分グラフの種類 （再掲）

- 経路
  - s-t 経路
  - ハミルトン経路（すべての頂点を通る）
  - サイクル（1周して戻る）
- 木
  - 森、全域森
  - 木、全域木
  - 連結成分
- マッチング
- クリーク（頂点同士がすべて結ばれている頂点集合）
- その他、様々な制約を課した部分グラフ

フロンティア法は、Knuth のアルゴリズム（s-t 経路）を、様々な部分グラフに適用できるように拡張したもの

適用事例1: パズルソルバー

- ナンバーリンク



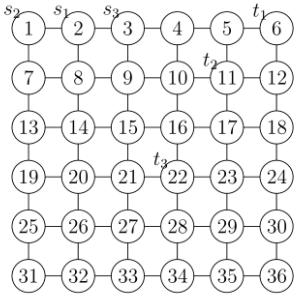
同じ数字同士を、線を交差させずに結ぶ

R. Yoshinaka, T. Saitoh, J. Kawahara, K. Tsuruma, H. Iwashita, S. Minato.
Finding all solutions and instances of numberlink and slitherlink by ZDDs.
Algorithms, 5, 176–213, 2012.

ZDD書籍から引用

適用事例1-1: ナンバーリンクソルバー 1/4

- グラフの問題として定式化



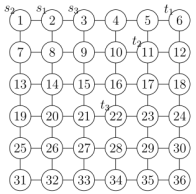
s1-t1 経路
s2-t2 経路
…
sp-tp 経路

（複数終端対経路）
を同時に求める問題

ZDD書籍から引用

適用事例1-1: ナンバーリンクソルバー 2/4

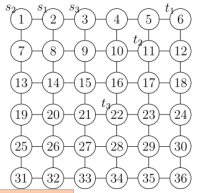
- graphillion には、直接、複数終端対経路を求めるメソッドはない
- 条件をつけた部分グラフを求めるメソッドを使う



ZDD書籍から引用

適用事例1-1: ナンバーリンクソルバー 3/4

- 次数の条件
  - 終端（si, ti）の次数は1
  - それ以外の次数は0 または 2



```
>>> universe = [...]
>>> GraphSet.set_universe(universe)
>>> point_pairs = [[2, 6], [1, 11], [3, 22]]
>>> points = sum(point_pairs, [])
>>>
>>> deg_cons = dict([(v, 1) for v in
...     set(range(1, 37)) if v in points])
>>> deg_cons.update(dict([(v, (0, 2)) for v in
...     set(range(1, 37)) if v not in points]))
```

[2, 6, 1, 11, 3, 22]

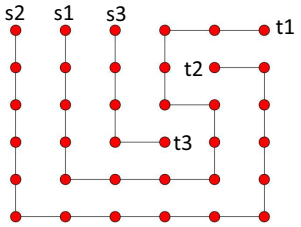
{1: 1, 2: 1, 3: 1, 4: (0, 2), 5: (0, 2), 6: 1,...

ZDD書籍から引用

適用事例1-1: ナンバーリンクソルバー 4/4

```
[[2, 6], [1, 11], [3, 22]]
```

```
>>> solutions = GraphSet.graphs(
...                     vertex_groups=point_pairs,
...                     degree_constraints=deg_cons,
...                     no_loop=True)
>>>
>>> print solutions.len()
1
```



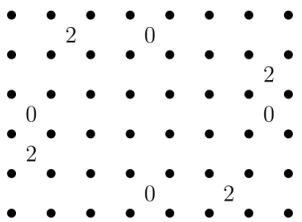
得られた解

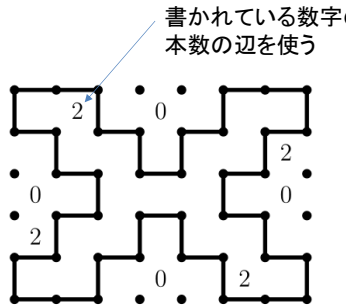
{1: 1, 2: 1, 3: 1, 4: (0, 2), 5: (0, 2), 6: 1,...

ZDD書籍から引用

適用事例1-2: スリザーリンクソルバー 1/4

- スリザーリンク





サイクルを1個作る

ZDD書籍から引用

適用事例1-2: スリザーリンクソルバー 2/4

- 初めに、マスの数字は無視して、サイクルをすべて求める

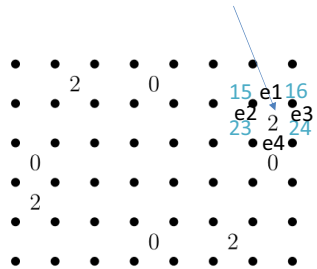
```
>>> universe = [...]
>>> GraphSet.set_universe(universe)
>>>
>>> cycles = GraphSet.cycles()
```


適用事例1-2: スリザーリンクソルバー 3/4

- 次に、ある1つのマスに着目
  そのマスの制約を満たしている部分グラフのみを抽出

```
>>> e1 = (15, 16)
>>> e2 = (15, 23)
>>> e3 = (16, 24)
>>> e4 = (23, 24)

>>> from itertools import
combinations
>>> g_set = list(combinations([e1,
e2, e3, e4], 2))
>>> in_gs = GraphSet(g_set)
>>> cycles = cycles.including(in_gs)
```

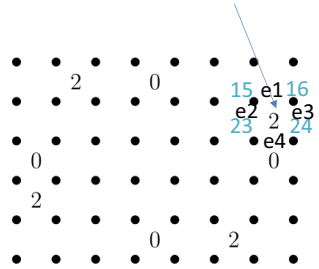


ZDD書籍から引用

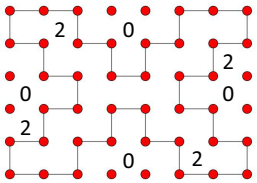
適用事例1-2: スリザーリンクソルバー 4/4

- 次に、ある1つのマスに着目
  そのマスの制約を満たしている部分グラフのみを抽出

```
>>> g_set2 = list(combinations([e1,
e2, e3, e4], 3))
>>> ex_gs = GraphSet(g_set2)
>>> cycles = cycles.excluding(ex_gs)
```



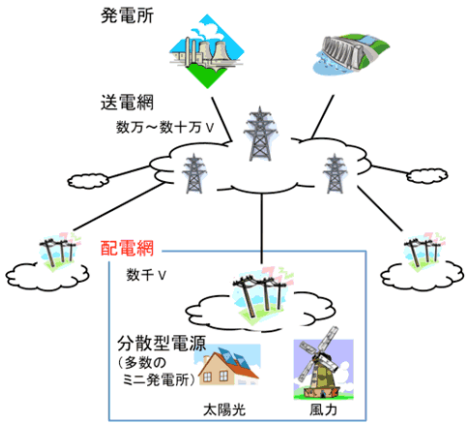
これをすべてのマスについて行う。
得られた部分グラフが解



得られた解

ZDD書籍から引用

適用事例2: 配電網のスイッチ構成 1/10

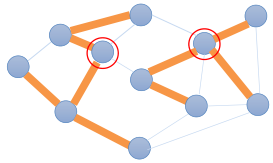


配電網

すべての家は
ちょうど1つの変電所に
つながっている必要がある

サイクルが生じてはいけない

変電所を根とする根付き全域森



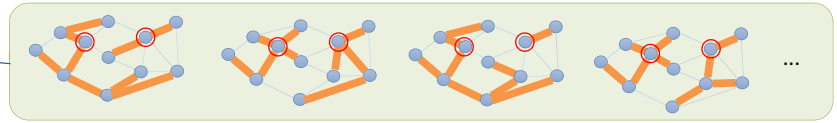
引用: http://www.jst.go.jp/pr/announce/20120223/index.html

T. Inoue et al., “Distribution Loss Minimization with Guaranteed Error Bound,”
IEEE Transactions on Smart Grid, Vol. 5, Issue 1, 2014, pp. 102-111.

適用事例2: 配電網のスイッチ構成 2/10

- 根付き全域森を列挙する

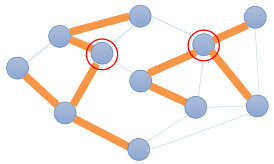
```
>>> universe = [...]
>>> GraphSet.set_universe(universe)
>>> forests = GraphSet.forests(roots=[1, 9, 73, 81],
                               is_spanning=True)
>>> print forests.len()
54060425088
```



適用事例2: 配電網のスイッチ構成 3/10

- 電気制約
  - 1つの変電所が給電可能な最大家庭数に制限がある

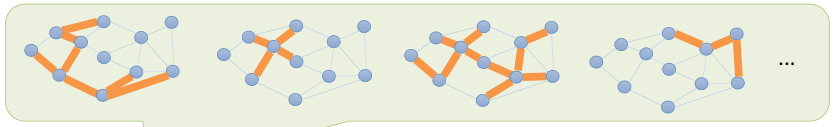
制限数 5



制限数オーバー

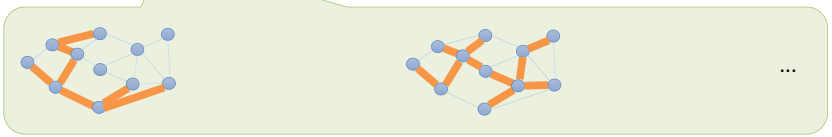
適用事例2: 配電網のスイッチ構成 4/10

- 電気制約を満たす根付き全域森を求める　制限数 6



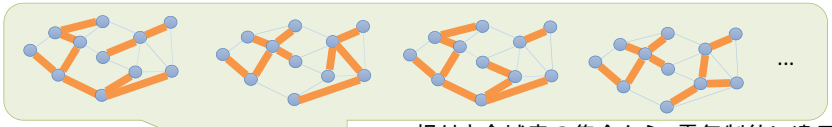
まずは、電気制約を満たさない大きな木を求める

```
>>> all_trees = GraphSet.trees()
>>> too_large_trees = all_trees.larger(6)
```



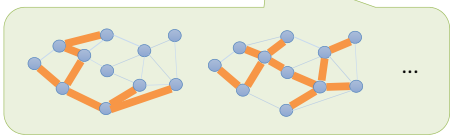
適用事例2: 配電網のスイッチ構成 5/10

- 電気制約を満たす根付き全域森を求める　制限数 6



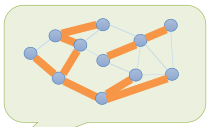
根付き全域森の集合から、電気制約に違反する木を含む根付き全域森を削除

```
>>> safe_forests = forests.excluding(too_large_trees)
```



適用事例2: 配電網のスイッチ構成 6/10

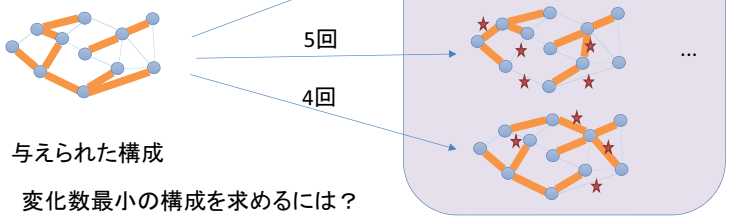
- 与えられた構成（根付き全域森）が、制約を満たすかどうか調べる



```
>>> unsafe_forest = [...]
>>> unsafe_forest in safe_forests
False
```


適用事例2: 配電網のスイッチ構成 7/10

- 与えられた構成から、スイッチON/OFFをいくつか変化させて、制約を満たす構成に変化させる



与えられた構成

変化数最小の構成を求めるには？

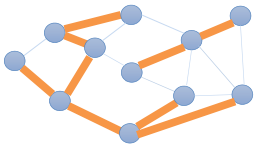
safe_forests（制約を満たす）

適用事例2: 配電網のスイッチ構成 8/10

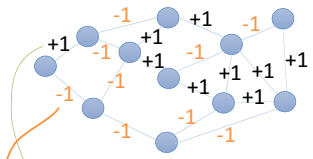
- 最適化問題として定式化

スイッチを変化させるごとにペナルティーが発生
ペナルティーを最小化する構成を求める

graphillion では、採用した辺にしか重みをつけることはできない



与えられた構成
（元グラフ）

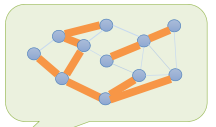


元グラフにない辺を採用すると
ペナルティー +1 点 （採用しないと 0点）

元グラフにある辺を採用すると
ペナルティー -1 点 （採用しないと 0点）

適用事例2: 配電網のスイッチ構成 9/10

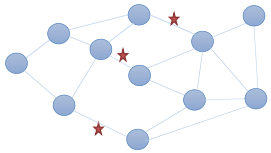
- 与えられた構成（根付き全域森）が、制約を満たすかどうか調べる



```
>>> unsafe_forest = [...]
>>>
>>> weights = {}
>>> for switch in universe:
...     weights[switch] = 1 if switch in unsafe_forest else -1
...
>>> for f in safe_forests.min_iter(weights):
...     print f
...     on_switches = [e for e in f if e not in unsafe_forest]
...     off_switches = [e for e in unsafe_forest if e not in f]
```


適用事例2: 配電網のスイッチ構成 10/10

- blocking set (hitting set)



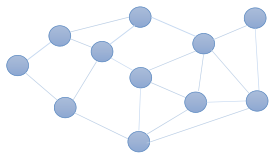
切断によって、構成（根付き全域森）がなくなるような辺の集合

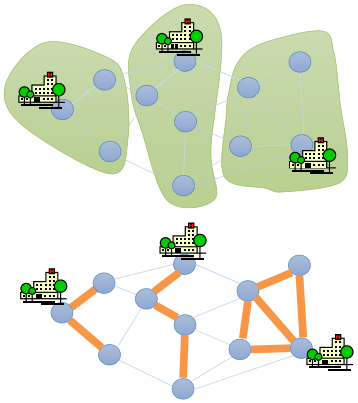
```
>>> failures = safe_forests.blocking()
>>> minimal_failures = failures.minimal()
```

極小のものだけを求めることができる

事例3: 避難所割り当て 1/2

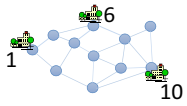
- グラフと避難所が与えられたとき、避難所ごとにグラフを分割





事例3: 避難所割り当て 2/2

- 割当をすべて列挙する



1, 6, 10 は同じ連結成分に含まれてはいけない

```
>>> universe = [...]
>>> GraphSet.set_universe(universe)
>>> assignments = GraphSet.graphs([[1], [6], [10]])
>>>
>>> maximal_assignments = assignments.maximal()
```

同じ連結成分内で辺が張れる場合は必ず張る

その他の適用事例

- 選挙区割りの列挙
- 多面体の展開図の列挙
- 住宅フロアプランの列挙
- …

まとめ

- graphillion の紹介
  - 内部データ構造とアルゴリズム
    - ZDD
    - フロンティア法
- 適用事例の紹介
  - パズルソルバー（ナンバーリンク、スリザーリンク）
  - 配電網のスイッチ構成
  - 避難所割り当て