# ヒープ・ZDDに関して Vol.3

## 2021年夏学期ゼミ

交通研B4 村橋拓真

# 目次

- ヒープ/ZDDの必要性
- ヒープとZDDの有効性
- Dijkstra法の確認
- ヒープのルールと操作①
- ヒープのルールと操作②
- ヒープのルールと操作③
- ZDDの効率性
- ZDDの長所/実践

# ヒープ/ZDDの必要性

- 例として自分の(将来的な)研究における活用例を考えてみる

研究の概要

解きたい問題

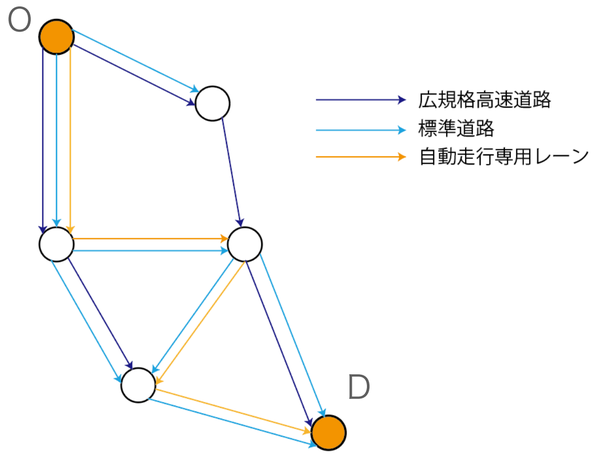
(高速道路上に自動走行レーンが作られることを想定した)道路空間/レーンの配分問題
オークション市場を配分方法に導入し，社会的最適と利用者均衡状態が一致するように考える.

想定

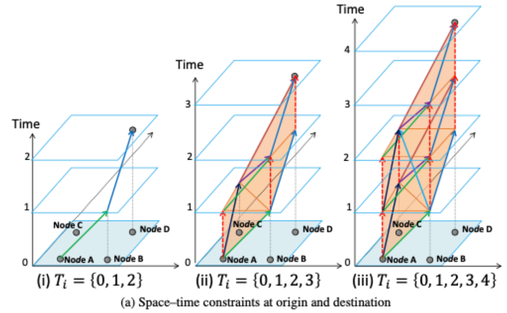
- ノードを結ぶリンクを車線ごとに捉える：大規模ネットワーク
- レーンの利用に関して，逐次的に入れ替わるので，次空間ネットワークを想定する

↑

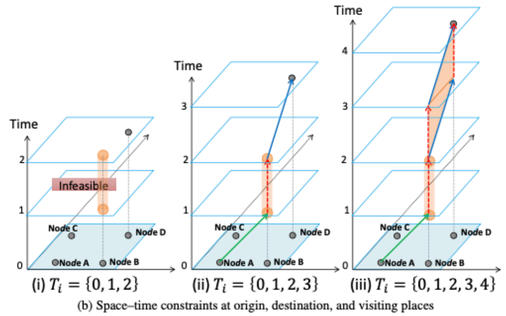
- 大規模な経路ネットワークを**ZDD**を用いて構築
- 利用者均衡状態の計算のためには，最短経路探索問題（またはRL）が入ってくる（**ヒープ**の活用）




ノード間のリンクを車線ごとに捉えたイメージ




(a) Space–time constraints at origin and destination




(b) Space–time constraints at origin, destination, and visiting places

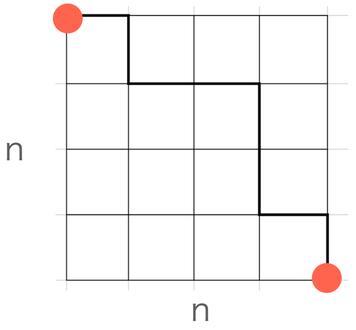
次空間ネットワークの例

Hayakawa, Hato(2018)より

# ヒープとZDDの有効性

- ヒープとZDDでできることを軽くまとめてみる

n×nの格子グラフの中での対角間での経路だけで
実現可能な経路の数はこんなに膨大になってしまう…




| n | 経路の数 |
|---|---|
| 1 | 2 |
| 2 | 12 |
| 3 | 184 |
| | |
| 8 | 3,266,598,486,981,640 |
| | |
| 11 | 182,413,291,514,248,000,000,000,000,000 |

ヒープ（最短経路問題での場合）

リストの形式ではなく，**データ構造内に一定の法則を持たせた状態**（min/maxヒープ）で，
各ノードの一時コストを保存することで，**計算量を省略可能**.
最短経路問題だけではなく，**再帰的に最小値（最大値）を取り出すアルゴリズムであれば適用が可能**.

例えば，Dijkstra法ならば計算量を，$O(|V|^2+E)$ から $O((|V|+|E|)log|V|)$ に落とし込むことが可能.
※V：ノード数，E：リンク数

ZDD（都市/交通分野で応用を考えた場合）

**経路の列挙はとても大変**である（図）．同時に経路データ $\{e_1e_3e_4,\ e_2e_4e_5,\ \cdots\}$ を，**効率的に保存する方法も難しい**.

各リンクを経路として選択するかどうかの二通りの決定の連続として，場合わけ2分木があるが，
その**2分木に対して省略や接点の共有を用いることで，ZDDではネットワークを圧縮して表現することが可能**である.

# Dijkstra法の確認

- ヒープの話に入る前にDijkstraの一連の流れを確認

## 1. インプットするデータ構造

Shibuya_node

| nodeID | latitude | longitude |
|---|---|---|
| 1 | 35.6644694701132 | 139.691300818354 |
| 2 | 35.664375324321 | 139.692157808500 |
| 3 | 35.6642892678908 | 139.692730495390 |
| 4 | 35.6641289072508 | 139.693888803312 |
| 5 | 35.664460937215 | 139.69499778337 |
| 6 | 35.6639184488099 | 139.695420878018 |

Shibuya_link

| LinkID | n1 | n2 | LinkCost |
|---|---|---|---|
| 0 | 1 | 2 | 78.28223494 |
| 1 | 1 | 11 | 109.1107196 |
| 2 | 1 | 1521 | 28.12928997 |
| 3 | 2 | 1 | 78.28223494 |
| 4 | 2 | 3 | 52.72191064 |
| 5 | 2 | 7 | 37.86278976 |

ノードデータとリンクデータ

## 2. アルゴリズム

V：全ノード，E：全リンク，S：グラフの頂点の中で最短経路が確定したもの，Q：頂点の中でS以外
s：始点，d(v)：Sの点のみを経由した時のsからvへの経路長，p(v)：現段階のvの一つ前のノード

Dijkstra法のアルゴリズム（リストを使用）

初期処理

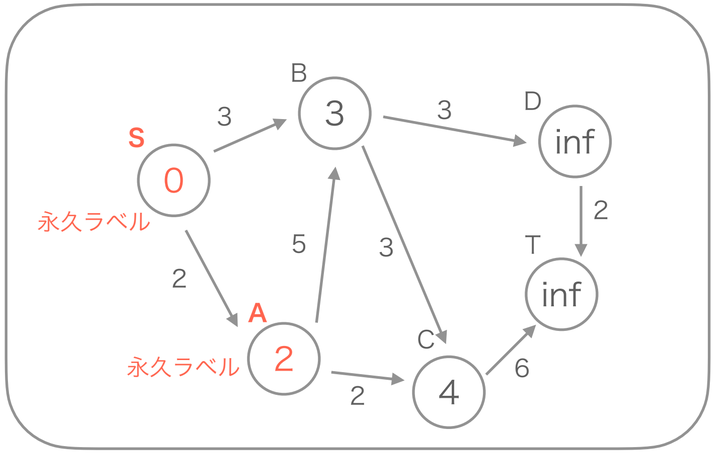
```
d(s)←0，S←s，Q←V - {s}
for 各頂点v∈Qについて，d(v) <- inf
for 各sの連接点uについてd(u) <- w(s,u) #w(s,u)はノードsからuへのリンクコスト
while Q ≠ φ do
    Qの中でd(x)が最小の頂点xを取り出す
    Sにxを追加
    for 各xの隣接点yについて，
        if d(y) > d(x) + w(x, y)
            then d(y) ←d(x) + w(x, y), p(y)←x
```

← |V|回繰り返し

← |Q|回の探索

← 全工程の合計で|E|回繰り返し

Dijkstra法の図（途中経過）




## 3. アウトプット

s，t間の最短経路とそのコスト

Dijkstra法
1323 -> 893
cost = 1809.3821705829998
path : [1323, 1324, 1322, 1321, 1320, 719, 453, 452, 451, 449, 448, 446, 445, 444, 1037, 473, 780, 924, 514, 518, 538, 922, 915, 861, 550, 551, 552, 613, 614, 889, 906, 907, 616, 891, 893]
calculaton time = 0.12610578536987305

# ヒープのルールと操作①

- ヒープにはルールがあるデータ構造であり，またデータの削除や追加をした時にはその形式を保つために操作が必要になる．

## 1．ヒープのルール

例として書いた図1.1のネットワークに対して，Dijkstra法を適用することを考える．「一時ラベルの中で値が最小なノードがAで，隣接ノードのラベルを書き換え終えた時点での状態」に関しての**一時ラベルをヒープを用いて表記**すると図1.2のようになる．ヒープを配列に落とし込むと図1.3のようになる．
図1.2のヒープよりも少し大きなヒープを図1.4に示す．図1.4のようなmin-heap（rootの値が最小）では，「親の値は必ず子の値よりも小さくなっている」ことがルールとなっている．ヒープを実際にコーディングする際には配列に落とし込むと図1.5のようになる．**法則性のあるデータ構造をうまく配列化できるのがヒープの強み**である．また，ヒープの階層数を深さと言い，また頂点を根，末端側を葉と呼ぶ．ヒープの深さは，**ヒープ内部のノード数を$N$とした時，$\log N$ となる**．

バイナリヒープ（２分木構造）が一般的で，今回もヒープと表記する場合はこれを指す．**バイナリヒープの場合は，親のインデックスをnとすると，子のインデックスは2n, 2n+1となる**ため，配列にアクセスする際はこの法則を用いれば良い．

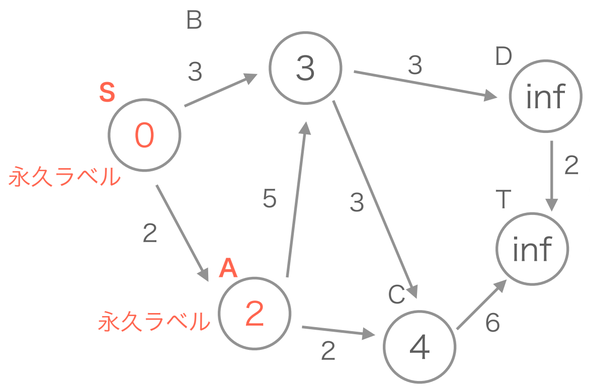



図1.1 ネットワークの例．始点：S，終点：Tで探索を行う．

```
while Q ≠ φ do
    Qの中でd(x)が最小の頂点x(x=A)を取り出す
    Sにxを追加
    for 各xの隣接点yについて,
        if d(y) > d(x) + w(x, y)
            then d(y) ←d(x) + w(x, y), p(y)←x
```

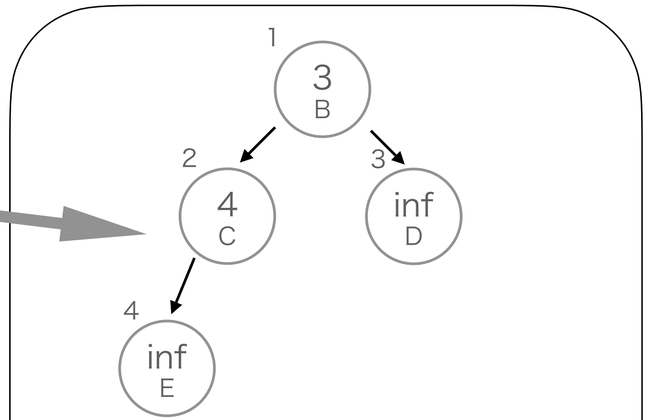



図1.2 図1.1の状態の一時ラベルをヒープ表現したもの

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 3 (B) | 4 (C) | inf (D) | inf (E) |

図1.3 配列化した図1.2のヒープ

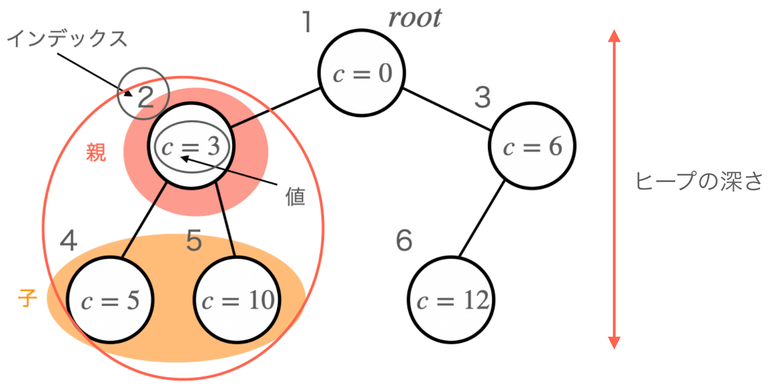



図1.4 min-ヒープの例（バイナリヒープ）

| 1 | 2 | 3 | 4 | 5 | 6 | ← インデックス |
|---|---|---|---|---|---|---|
| 0 | 3 | 5 | 8 | 10 | 12 | ← 値 |

図1.5 配列化した図1.4のヒープ

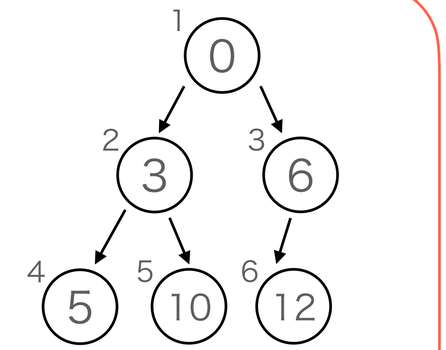
# ヒープのルールと操作②

- ヒープにはルールがあるデータ構造であり，またデータの削除や追加をした時にはその形式を保つために操作が必要になる．

## 2.1．ヒープの操作①

ヒープに対しての操作は
①データの追加：insert(val)
②データの削除：delete(val)
③デー他の値の変更：changekey(i, val)　の三つである．これらの計算量は，ヒープの深さに相当し，ノードの数を$N$とした時，$\log N$となる．

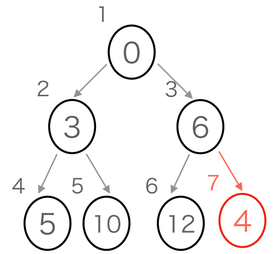
全ページの図1.4のmin-ヒープを例に，それぞれinsert，dekete，changekeyの処理方法を示す．

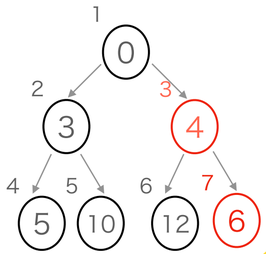


元のヒープ（全スライドの図1.4のもの）

### ① insert

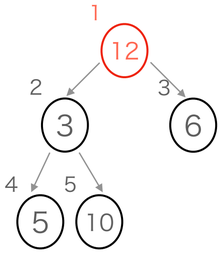
今回は，値が4のノードをヒープに加える．
処理の手順は
① 今回加えるノードをヒープの最下段に加える．
② ルールを満たすように上から親 - 子の間でノードを入れ替える．

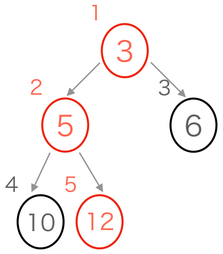



### ② delete

今回は，一番上のノードを削除する．
処理の手順は
① 一番上のノードを削除し，そこに元々最下段のノードの値を入れる．
② ルールを満たすように上から親 - 子の間でノードを入れ替える．




### ③ changekey

処理の手順は
① 値を書き換える．
② 値を書き換えたところから，ルールを満たすように上から親 - 子の間でノードを入れ替える．

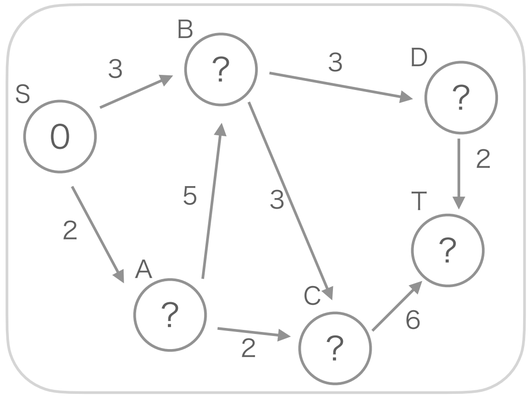
# ヒープのルールと操作③

- ヒープにはルールがあるデータ構造であり，またデータの削除や追加をした時にはその形式を保つために操作が必要になる.

## 2.2. ヒープの操作②

「ヒープのルールと操作①」のスライドで考えていた，Dijkstra法に関して実際にヒープを書きながら進めていく．（左下のアルゴリズムにつけた番号参照）

Dijkstra法を適用するネットワーク

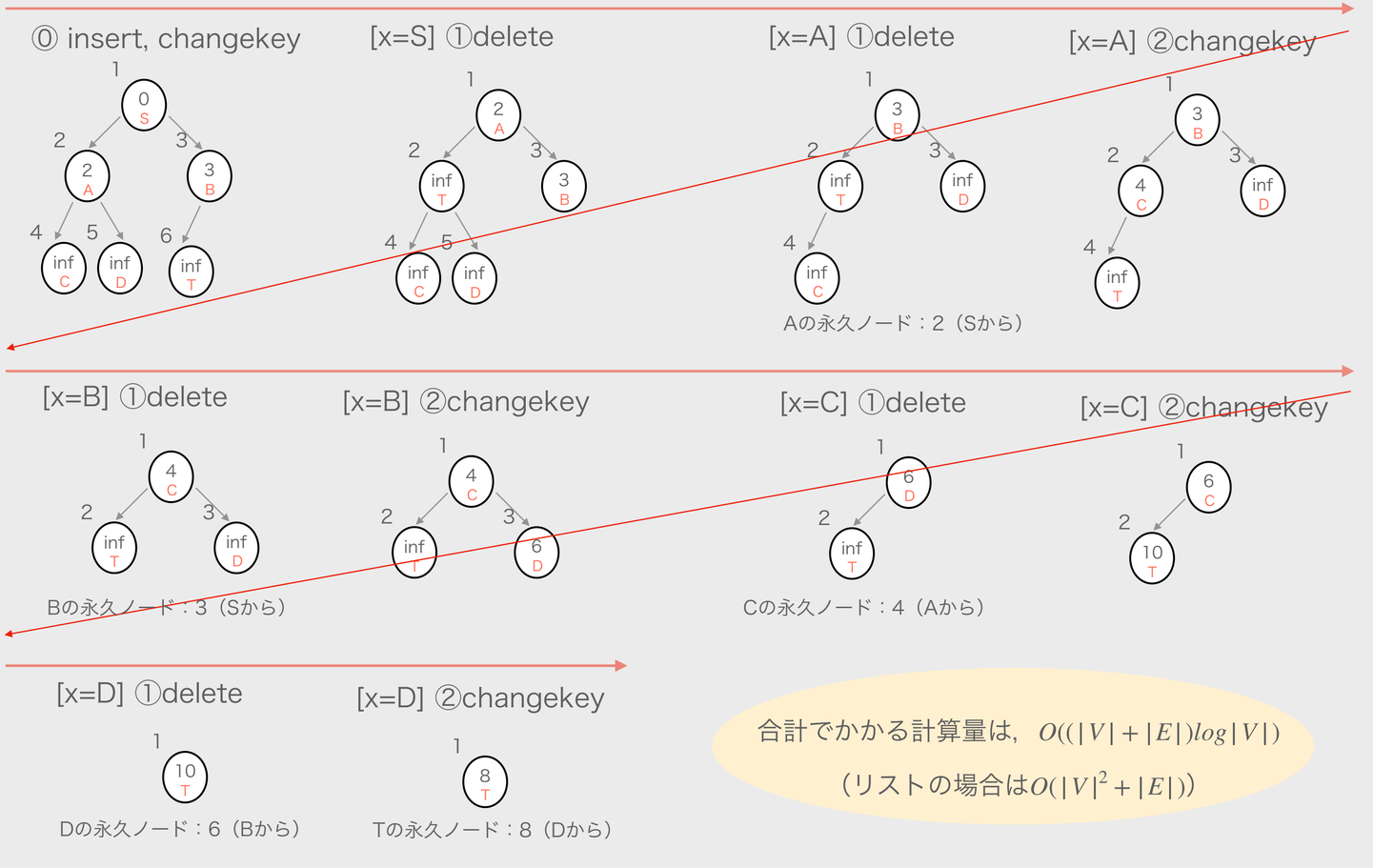



Dijkstra法のアルゴリズム

V：全ノード，E：全リンク，S：グラフの頂点の中で最短経路が確定したもの，Q：頂点の中でS以外
s：始点，d(v)：Sの点のみを経由した時のsからvへの経路長，p(v)：現段階のvの一つ前のノード

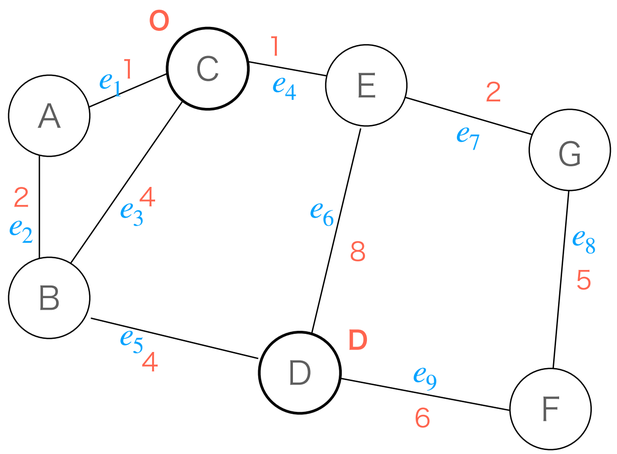
Dijkstra法のアルゴリズム（リストを使用） ⓪

```
d(s)←0，S←s，Q←V - {s}
for 各頂点v∈Qについて，d(v) <- inf
for 各sの連接点uについてd(u) <- w(s,u) #w(s,u)はノードsからuへのリンクコスト
while Q ≠ φ do
    Qの中でd(x)が最小の頂点xを取り出す   ① delete
    Sにxを追加
    for 各xの隣接点yについて，
        if d(y) > d(x) + w(x, y)
            then d(y) ←d(x) + w(x, y), p(y)←x   ② changekey
```



# ZDDの効率性

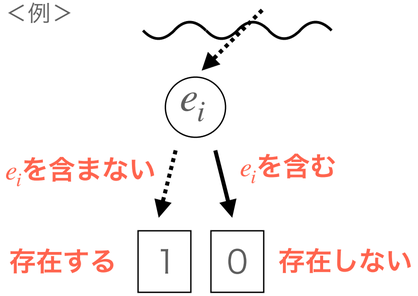
- 下のネットワークを例に，OD間の実際の経路の数と，その表現（場合分け2分木，ZDDをみる.




今回OD間の経路を考えるネットワーク

上の経路に関してどのようにデータを書き込むかを考える.
経路は{ $e_1e_2e_5$, $e_3e_5$, $e_4e_6$, $e_4e_7e_8e_9$} の四通り.
方法論的には基本的には場合分け2分木とZDDがある.

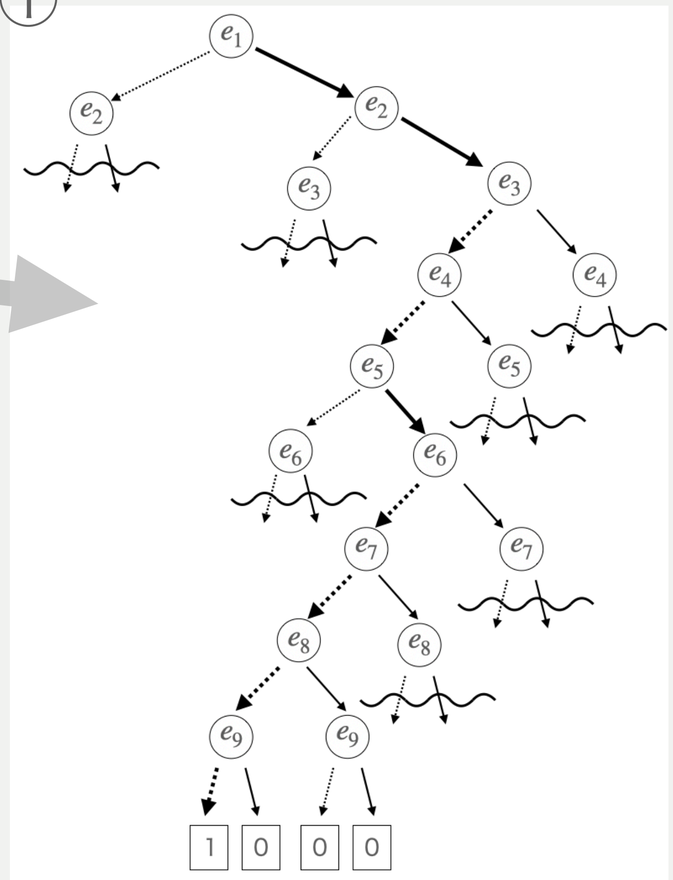
＜例＞




＜解説＞

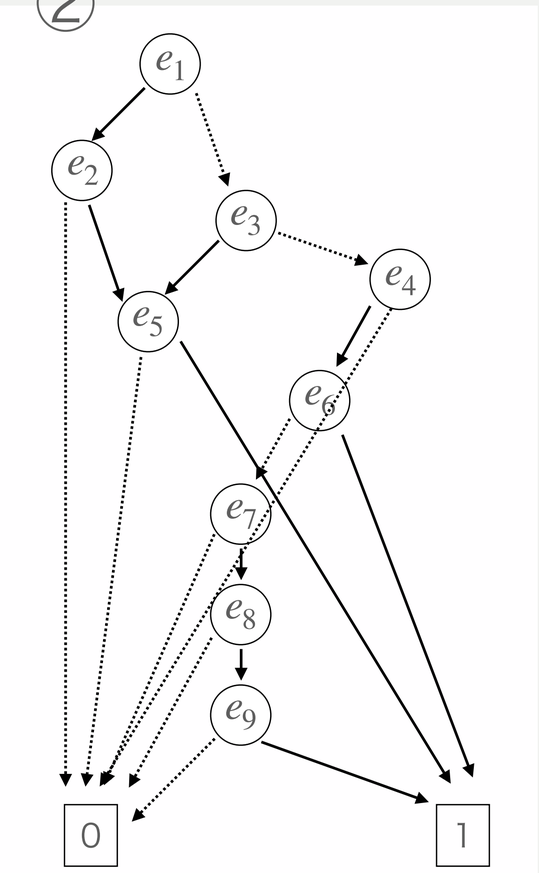
| | |
|---|---|
| $e_i$ ——→ | リンク$e_i$を含む場合 |
| $e_i$ ⋯→ | リンク$e_i$を含まない場合 |
| ——→ 1 | 辿ってきた経路が存在する |
| ——→ 0 | 辿ってきた経路が存在しない |

①



2分木で経路$e_1e_2e_5$を表現したもの
非常に冗長で，無駄が多いことがわかる

②



ZDDで{ $e_1e_2e_5$, $e_3e_5$, $e_4e_6$, $e_4e_7e_8e_9$} を表現したもの

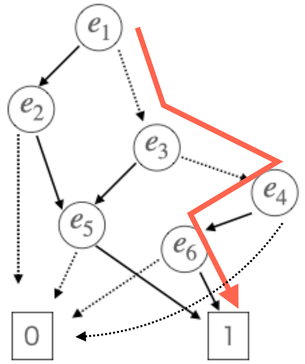
# ZDDの長所/実践

- ZDDの利便性，Graphillionを用いたpythonでのZDDの実装

## ZDDの利便性

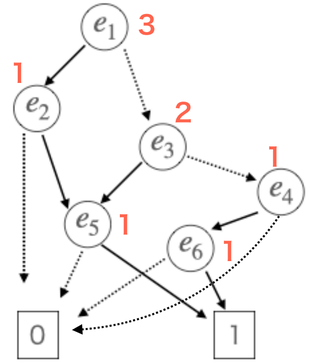
ZDDは索引化されており，情報の再確認が容易

・ある組み合わせがデータベース上にあるか？




**根節点から辿り，最後に1-終端節点に辿り着くか**確認すればよい．
今回は$e_4e_6$という経路が存在するか確認している．

・各接点以下の経路数を調べる




下から1-**終端接点を通る経路数を足し合わせていく**ことで，「**そのリンクからの組み合わせで経路が何通り存在するのか？**」を確認することが可能である．
左の図のように，「そのノードから出ている矢印2つから伸びているノードの”経路数”」を足し合わせていけば良い．

## Graphillion

グラフ集合を扱うためのライブラリで，バスや全域木，マッチングなど様々な対象を列挙できる．PythonやC++で使用できる．詳しくは下のリンクを参考に．（以下例）
https://qiita.com/cabernet_rock/items/50f955afc16287244154

```
# 必要なモジュールのインポート
from graphillion import GraphSet
import graphillion.tutorial as tl
```

事前にインストールしておけば，他のライブラリと同じように呼び出すことが可能．

```
import time          # 計算時間を調べる。
universe = tl.grid(8, 8) # 9×9のグリッド
GraphSet.set_universe(universe)
start = 1
goal = 81
s = time.time()      # 計算開始時刻
paths = GraphSet.paths(start, goal)
```

ここでは，9×9の格子グラフで対角間の経路数を数えている．0.4秒以下でなんと3000兆を超える列挙数を列挙している．

```
time.time() - s
```

```
0.35927605628967285
```

```
len(paths)
```

```
3266598486981642
```

```
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)] )
```

このように，リンクを設定してあげればZDDを自作できる．(1, 2)は「**ノード1とノード2を結ぶリンクが存在すること**」を意味している．

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

```
paths.len()
```

```
12
```

経路の種類は12通りであった．

```
for p in paths:
    print(p)
```

ここでは結果は省略しているが，その一覧も下の方法で全経路を列挙することも可能．