Takizawa, A., Takechi, Y., Ohta, A., Katoh, N., Inoue, T., Horiyama, T., Kawahara, J. and Minato, S.:

Enumeration on region partitioning for evacuation planning based on ZDD

In Proceedings of the International Symposium on Operations Research and its Applications (ISORA 2013), pp.64-71, 2013.

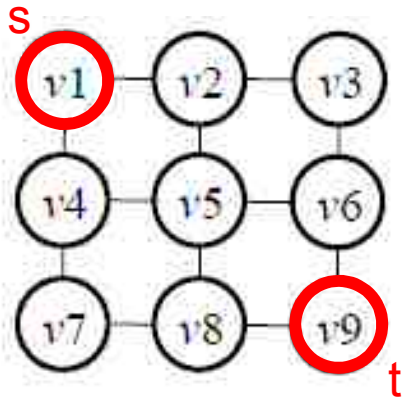
理論談話会#8

吉野大介

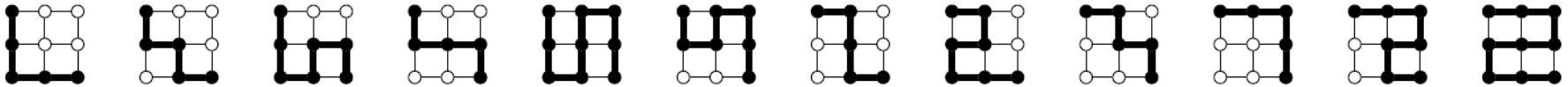
グラフの列挙と組み合わせの圧縮

■2×2の格子状グラフの問題

- 頂点sから頂点tまで同じところを2度通らない経路の組み合わせ
- 遠回りを許す場合、総数を簡潔に表す公式や漸化式などは見つかっていない



■総当たりで列挙すると、12通りということが分かる



■格子が増えると組み合わせ爆発

- 3×3だと184、4×4だと8,512、5×5だと1,262,816…
- 続きは「おねえさんの問題」にて　→https://youtu.be/Q4gTV4r0zRs

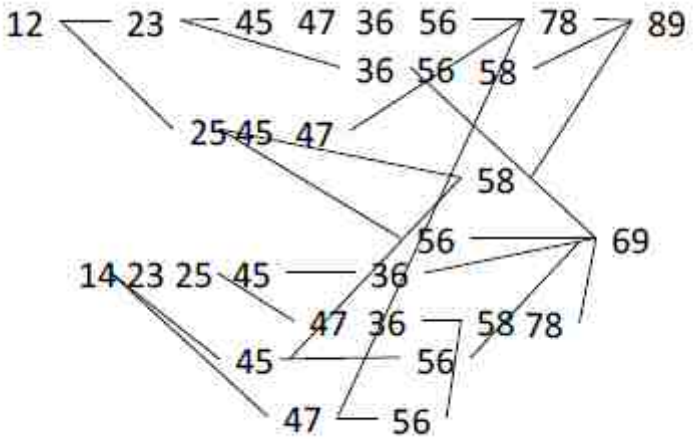
グラフの列挙と組み合わせの圧縮

■全経路を単純に列挙する
⇒グリッドが増えると組み合わせ爆発

1: 12 23 45 47 36 56 78 89
2: 12 23 36 56 58 89
3: 12 23 36 69
4: 12 25 45 47 78 89
5: 12 25 56 69
6: 12 25 58 89
7: 14 23 25 45 36 69
8: 14 23 25 47 36 58 78 69
9: 14 45 56 69
10: 14 45 58 89
11: 14 47 56 58 78 69
12: 14 47 78 89

エッジ数＝64

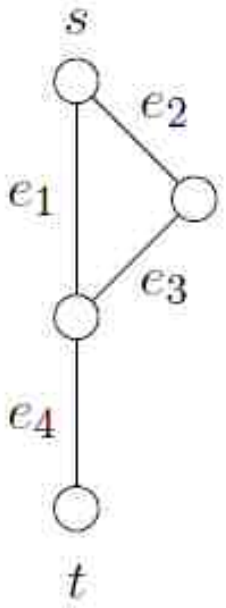
■圧縮表現による列挙
⇒情報を圧縮できる



エッジ数＝30

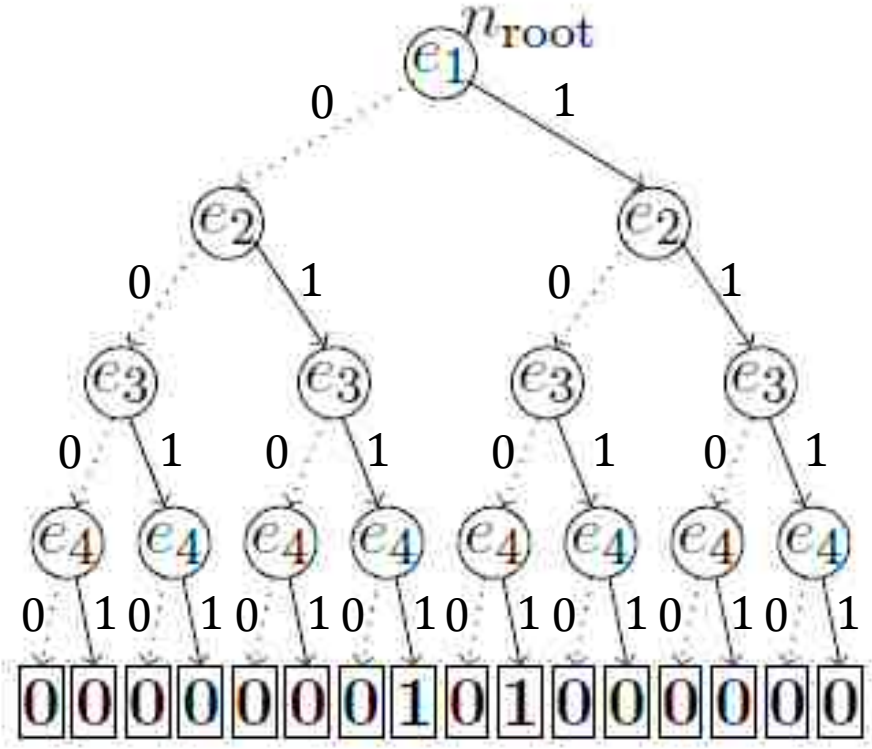
※12は$\{v_1, v_2\}$のパスを意味する

ZDD（ゼロサプレス型二分決定グラフ）



s-t経路
「e1-e4」もしくは「e2-e3-e4」の2通り

場合分け二分木（binary decision tree）による圧縮表現

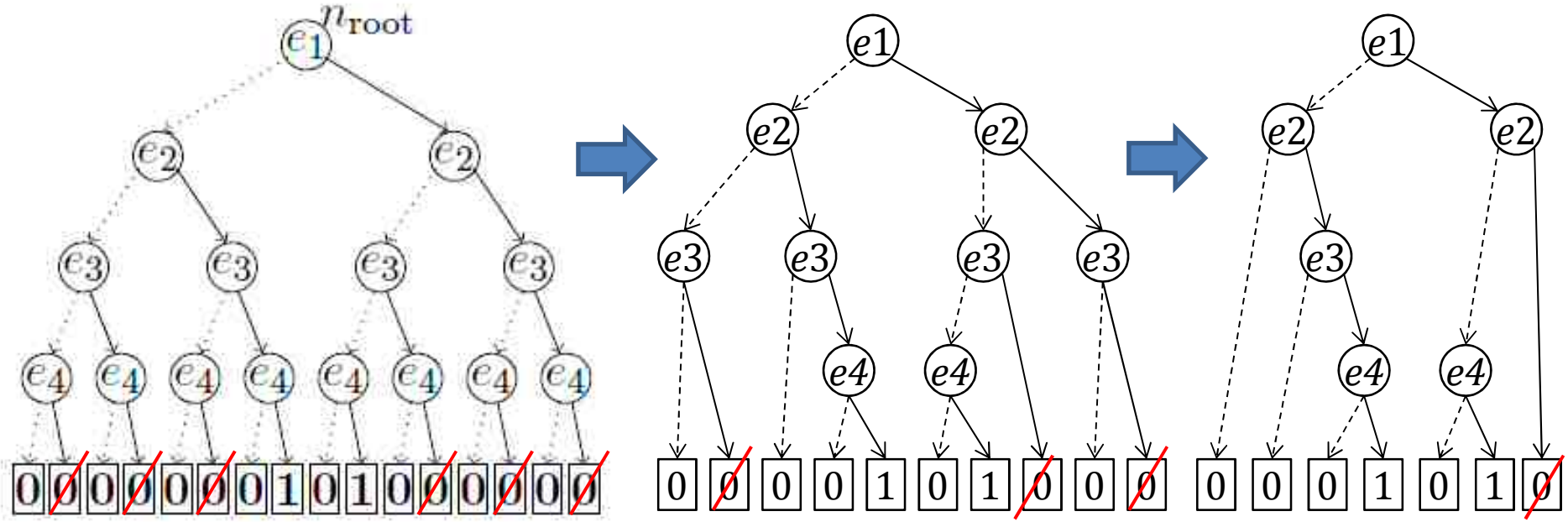


{e2, e3, e4}　{e1, e4}

ZDD（ゼロサプレス型二分決定グラフ）

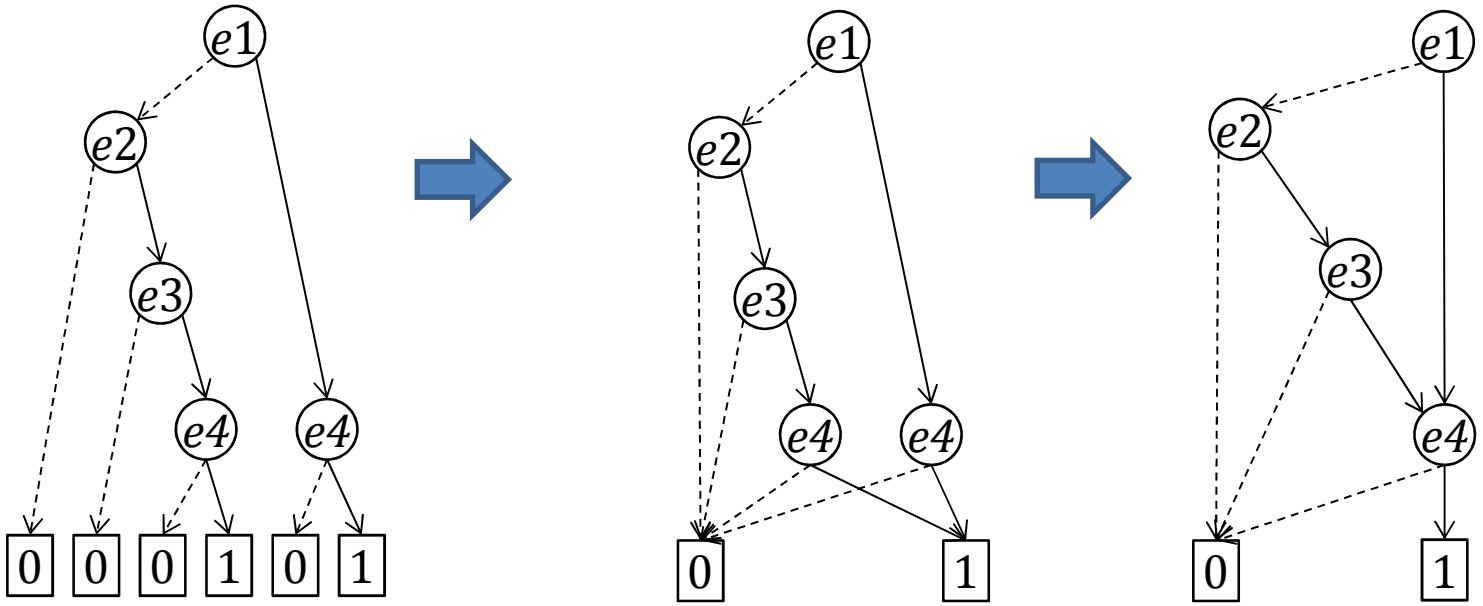
■冗長接点の削除

1-枝が0の値をもつ葉を有しているとき、この接点を取り除き、0-枝の行き先に直結させる



ZDD（ゼロサプレス型二分決定グラフ）

■等価接点の共有
葉を0/1それぞれ1つずつまでそぎ落とし、等価な接点（リンクが同じで、0-枝同士、1-枝同士の行先が同じ）を共有する



s-t経路集合を表すZDDの構築

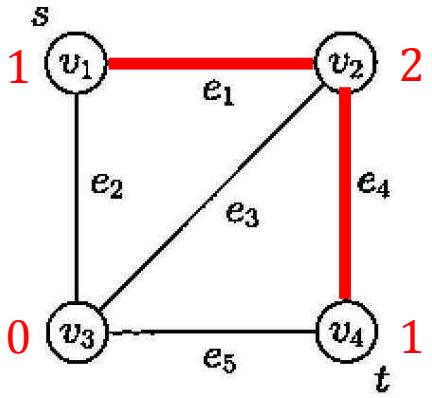
■s-t経路の性質

あるグラフ上の、あるs-t経路をPとしたとき、sとtはPの起終点となる

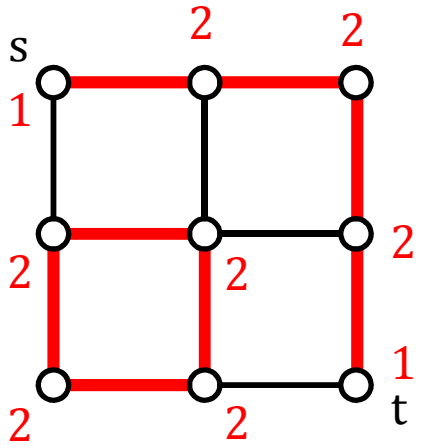
⇒sとtのPにおける次数は1

⇒Pの中間における頂点の次数は2

⇒Pに含まれない頂点の数は0



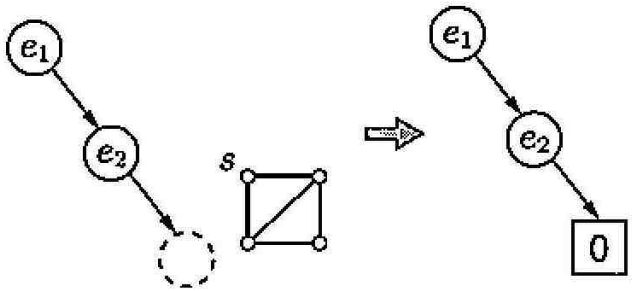
では逆に「sとtの次数は1、それ以外の頂点の次数は0または2」を満たす部分グラフは必ずs-t経路になるのか？・・・・そんなことはない



s-t経路集合を表すZDDの構築

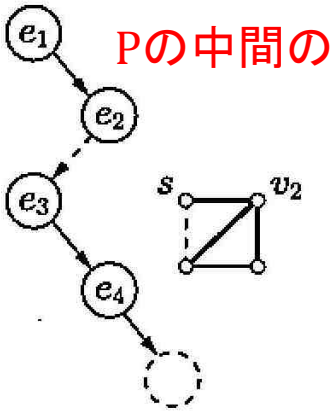
■ノードの作成と枝刈りによりs-t経路を表現

sまたはtの次数が2以上



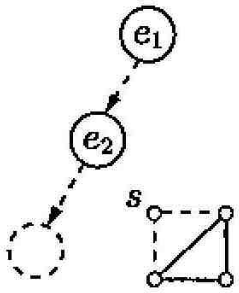
(1)

Pの中間の次数が3以上



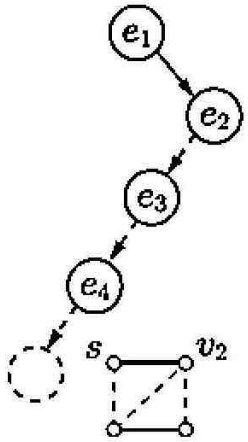
(2)

sまたはtの次数が0



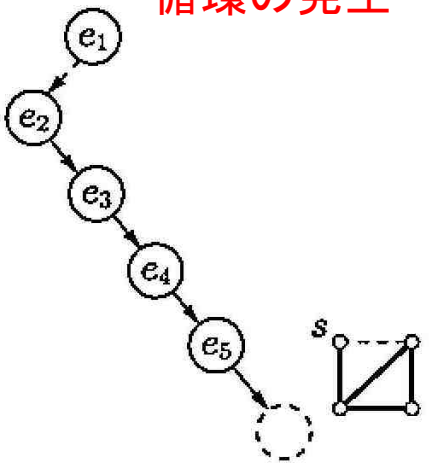
(3)

s-tの途中で分断



(4)

循環の発生

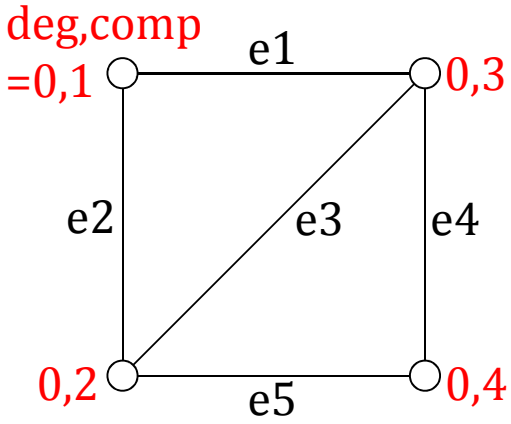


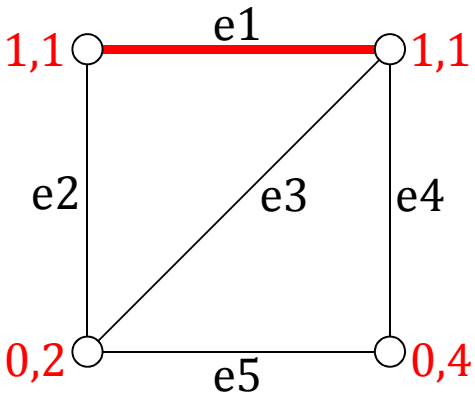
(5)

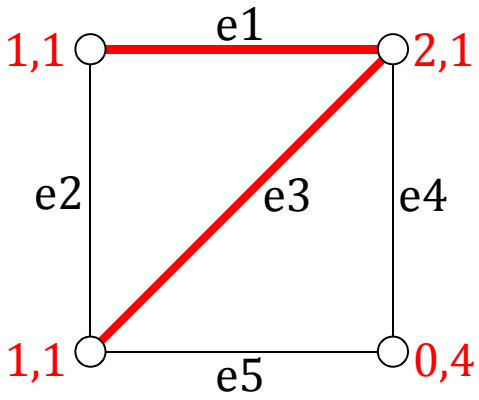
s-t経路集合を表すZDDの構築

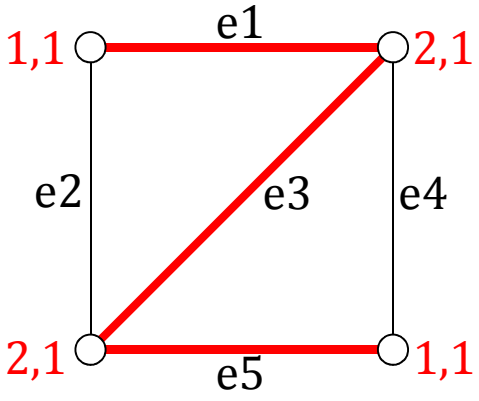
(1)~(5)をどう判断する？⇒各ノードに配列変数degとcompを持たせ、値を記憶する

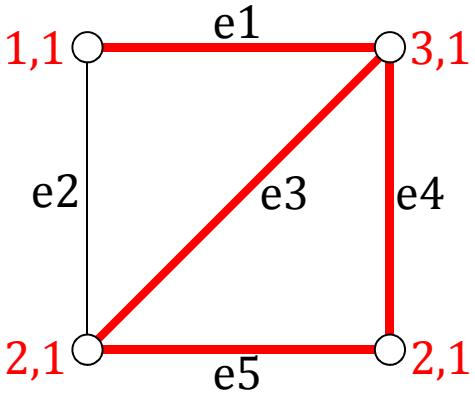
- 頂点vの次数をdeg[v]で表し、ノードnが持つ配列をn.degで表す
- 頂点vとwが同じ連結成分に含まれるならcomp[v]=comp[w]





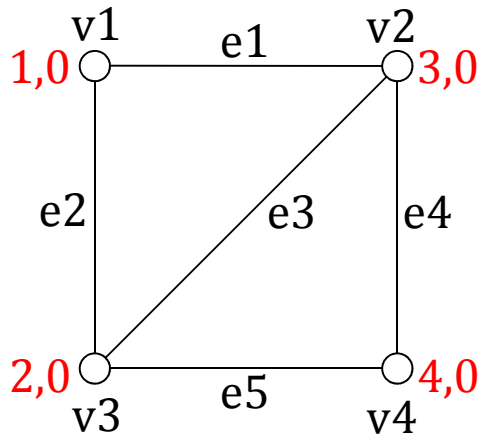


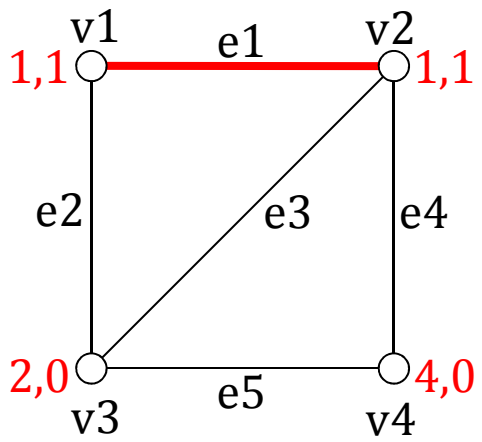


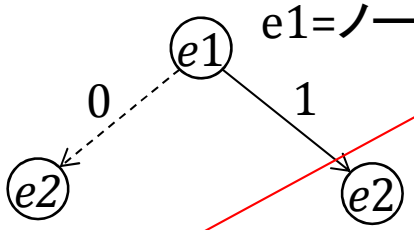


comp[v]=comp[w]である
v,w間を連結するとサイクル
が生じる

s-t経路集合を表すZDDの構築







e1=ノードn'

n'.deg = 0 0 0 0
n'.comp = 1 2 3 4

e1が選択されない状態=ノードn0

n0.deg = 0 0 0 0
n0.comp = 1 2 3 4

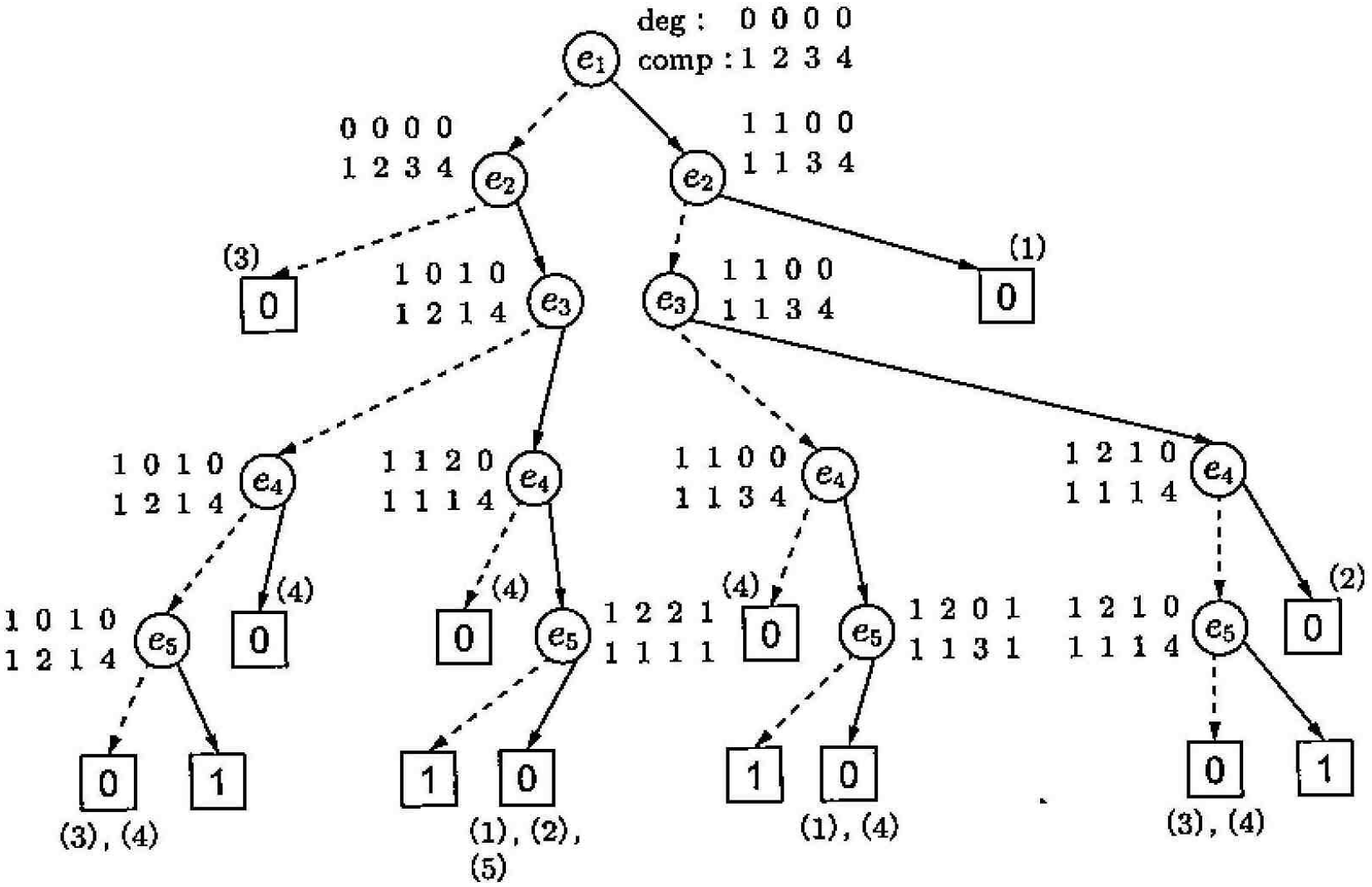
n'.degとn'.compをそのままコピー

e1が選択された状態の=ノードn1

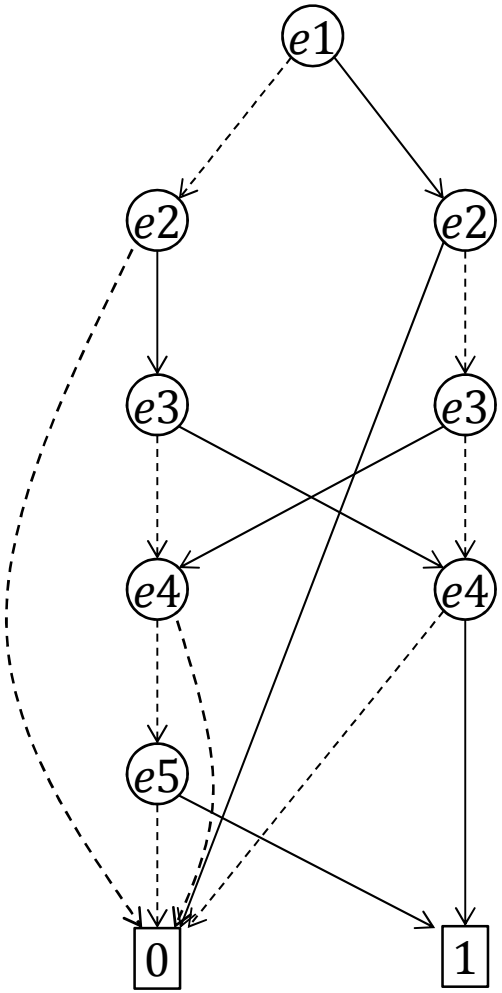
n1.deg = 1 1 0 0
n1.comp = 1 1 3 4

- v1とv2の次数が1つ増える
- V1とv2の連結成分が結合されるため、連結成分番号の大きい方の値を小さい方に書き換える

s-t経路集合を表すZDDの構築



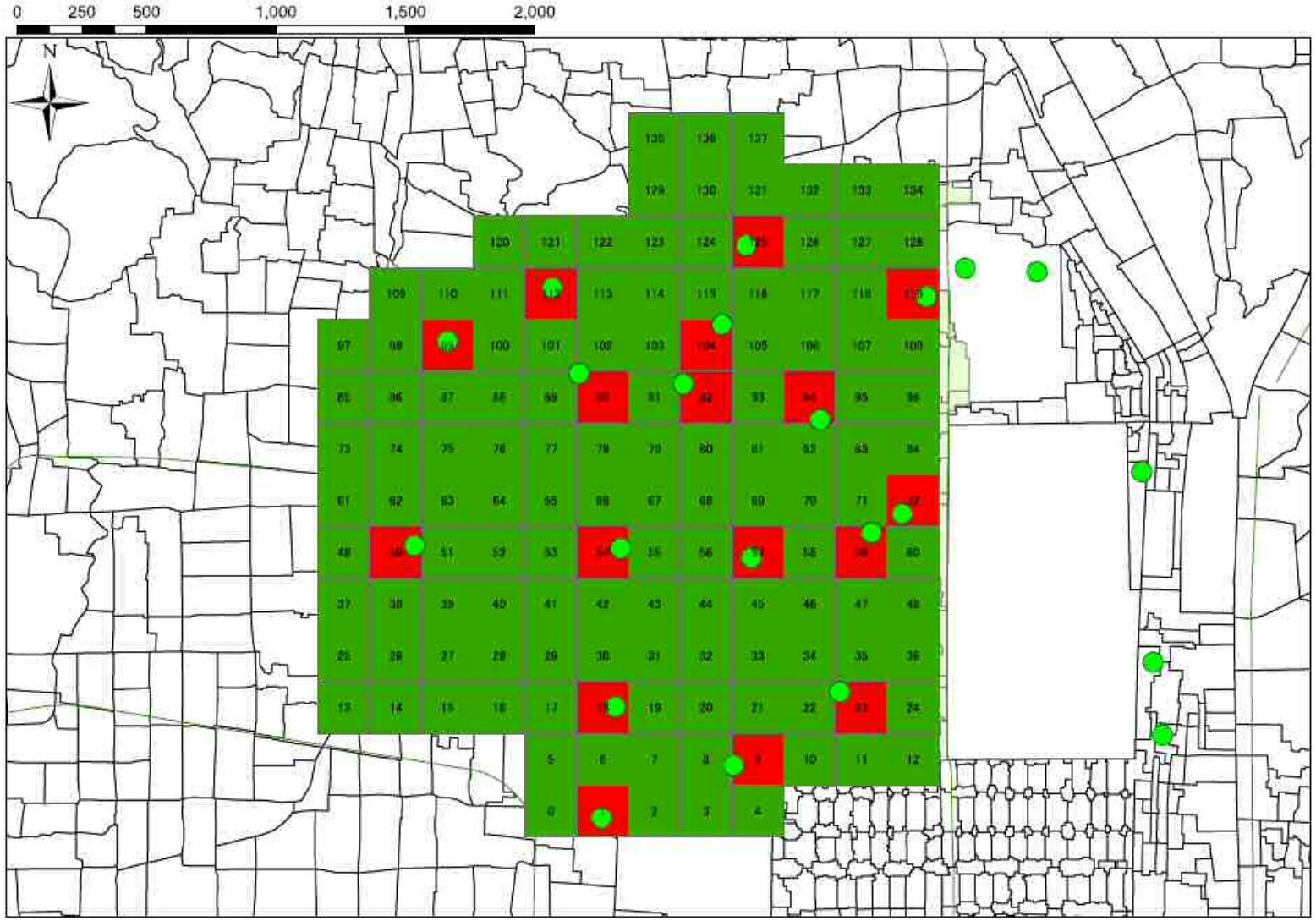
s-t経路集合を表すZDDの構築



話は戻って、論文の概要

- ZDDを用いて各地区における避難所への割当てパタンを列挙する
  - 集合分割問題であり、厳密解を求めるのが困難
- 特定の避難所に避難者が集中しないような平準化が避難計画立案においては重要な課題
  - 単純に学校区で一律に設定すると収容人数がばらつく
  - 重みづけボロノイによる解法が提案されているが最適解が保証されていないし、解のバリエーションが無い
- ZDDによって、制約（容量や歩行可能距離）を満たす割り当てを効率的に全列挙する方法を提案

対象地域（Kamigyo ward of Kyoto city）



老朽家屋・細街路・袋小路が多い

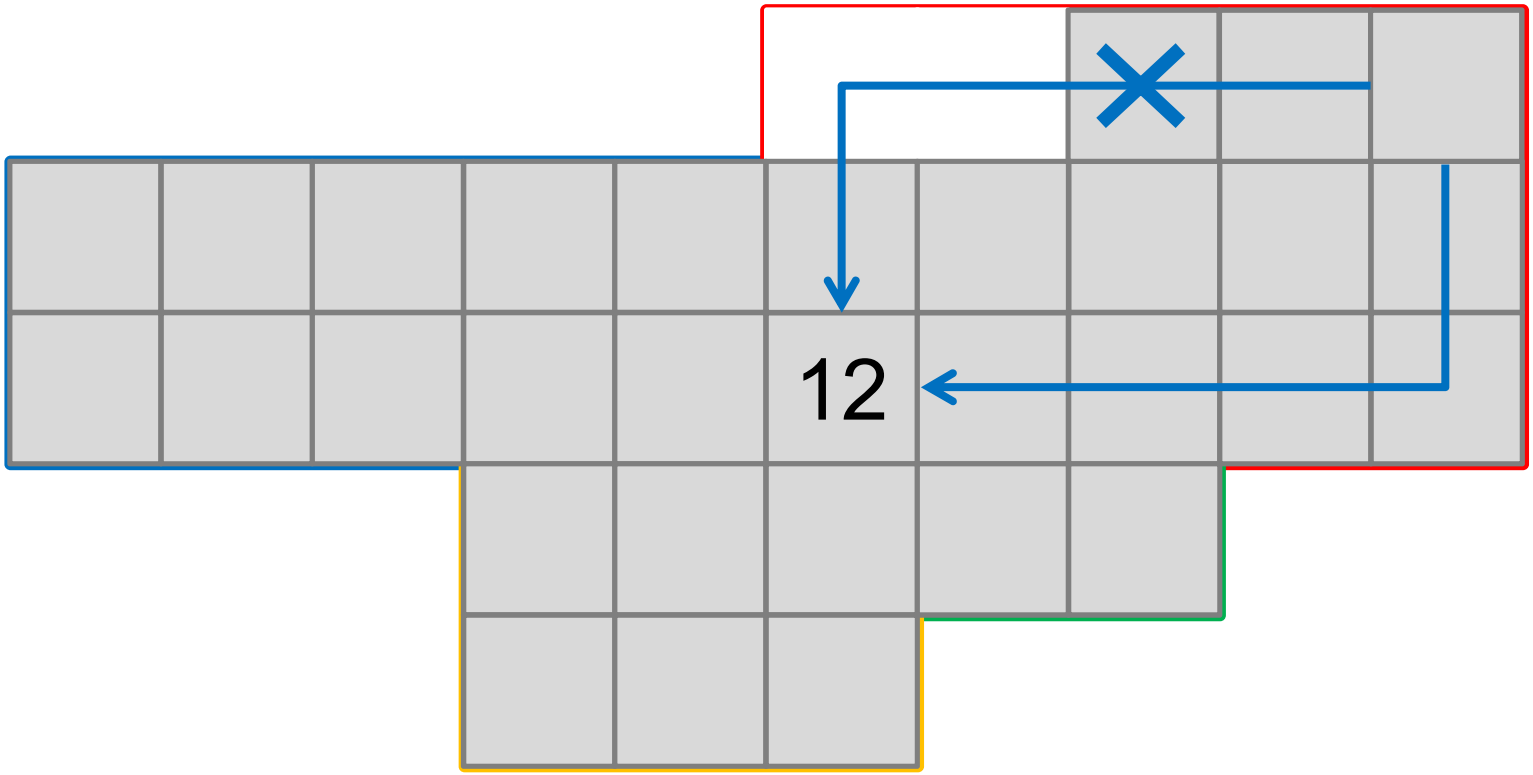
分析のおおまかな流れ

- 逆探索法を用いて各避難所における、凸型領域制約、距離制約、容量制約を満たす割り当てを列挙する
- 求めた避難所ごとの割り当ての列挙解から、各セルに「割り当てられる可能性がある避難所のインデックス」を保存する
- 避難所の割り当てパタンをZDDに変換し、各セルにおける避難所の割り当てパタンをトポロジー制約をかけながら1つずつ積の演算を用いて足し合わせ、実現可能な割り当てパタンを圧縮し全列挙する

制約条件

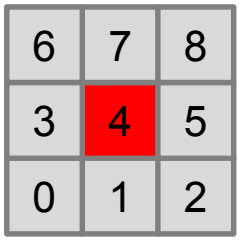
- 凸型領域制約
  - 避難所を中心とする領域が凸であれば、避難動線が他の領域と重ならず地域のまとまりも明瞭
  - 避難所セルに関する領域は、それぞれ避難所セルを含むような長方形領域の和として表現する
- 距離制約
  - 避難所セルの中心から遠い辺までの距離（maxdis）以内のセルを領域範囲とする
- 容量制約
  - 避難所間での収容率（避難者数/避難所の容量）のばらつきを小さく抑える
  - 避難想定者をエリア内の夜間人口に比例するものと定義し、各セルに夜間人口を按分、収容率の最大値（maxcap）を超えないようにする

凸型領域制約

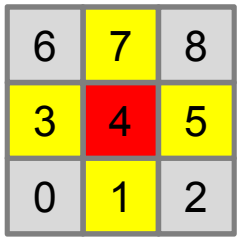


逆探索法によるセルの避難所への割り当て列挙

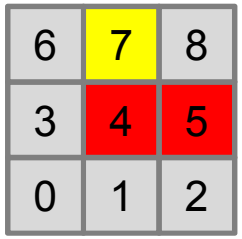
- 避難所を中心としたサイズに応じた固有のindexを定める
- 避難所を含むセル(4)だけからなる領域を「根」と定義



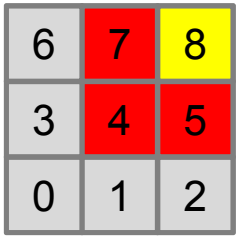
- 適合セルは4の周り全てであるため、(1,3,5,7)の4つ



- 仮にindex5に繋いだ場合、1と3は5より小さいので適合セルから排除される
- 8は4を含む長方形領域ではないため×
- 2は5より小さいので×



- 7に繋いだ場合、6は7より小さいので×
- 8が適合セルに入る

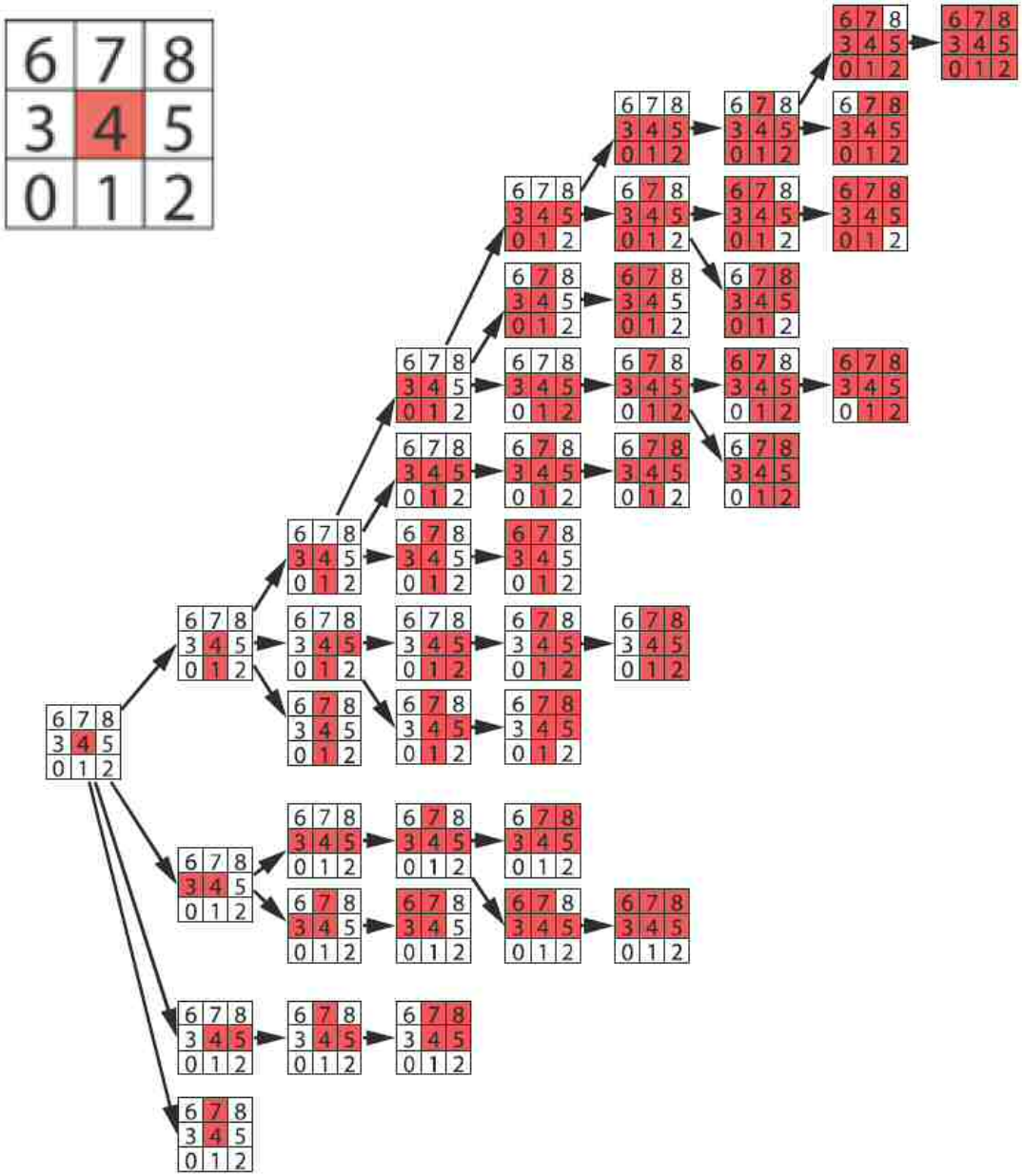


これを全ての組み合わせで列挙

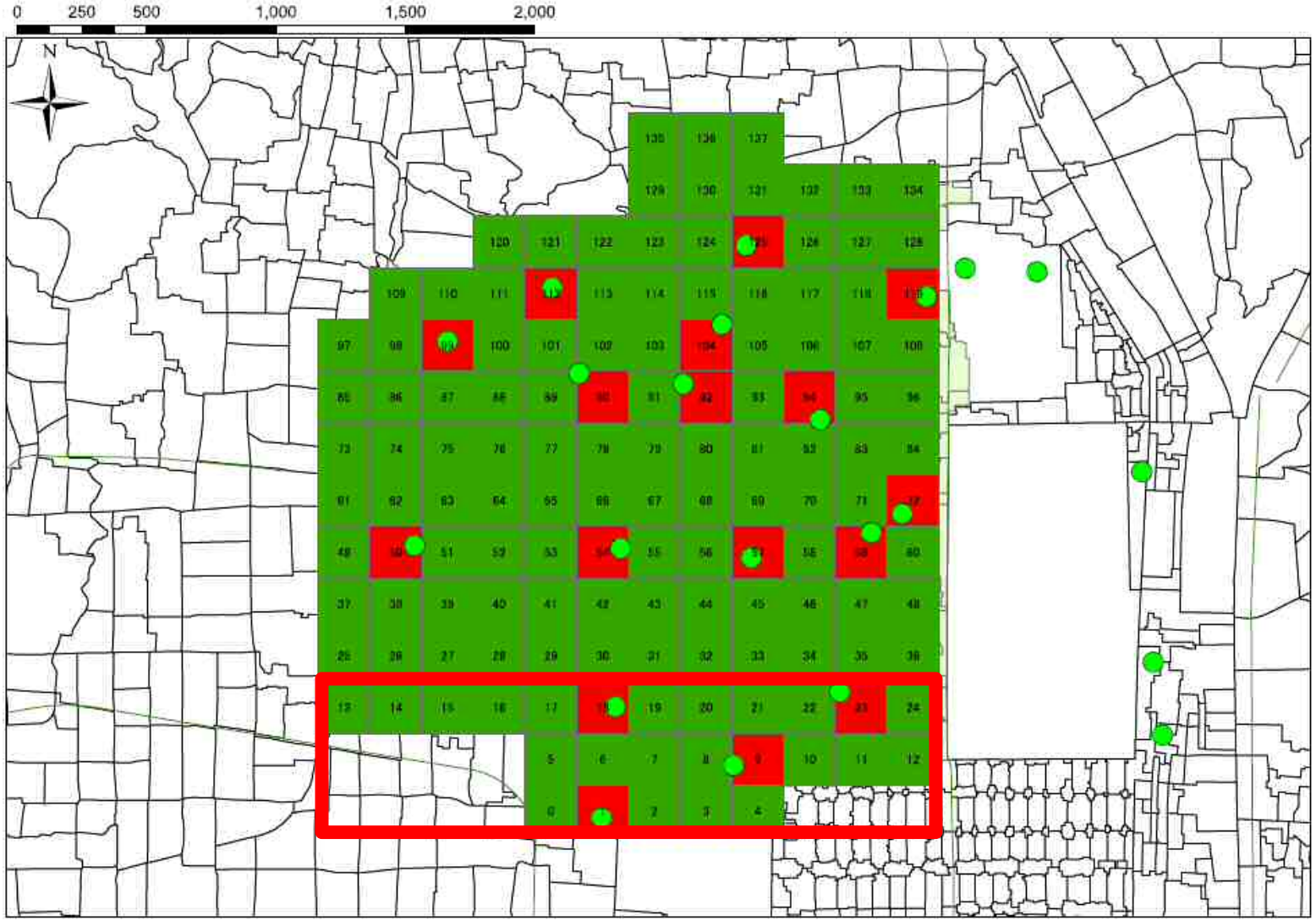
＜適合セル：避難所を選択する可能性があるセルの選択ルール＞
- 避難所と同じ軸上にある場合は適合
- 避難所と同じ軸上でなくても、隣接するセルが接続済みであり、なおかつ避難所を含む長方形領域に入っていれば適合
- ある適合セルが選択された際に、選択されたセルのindexを下回るindexのセルは適合セルから排除される

＜容量制約と距離制約＞
- この流れの中で、セルが接続されるたびに接続されたセル内の避難想定人数を足し合わせて、それが避難所における収容可能容量を超えた時点でそれ以上のセルを接続することをやめる（容量制約を満たすものに限定）
- 同じく、距離が一定以上となった場合もそれ以上のセルの接続をやめる（距離制約を満たすものに限定）

３×３グラフの場合の列挙



対象地域（Kamigyo ward of Kyoto city）



ZDDへの変換による分割の列挙

逆探索法による避難所の列挙解の中に一度でも現れたセルの範囲を「避難所のカバー圏」と定義

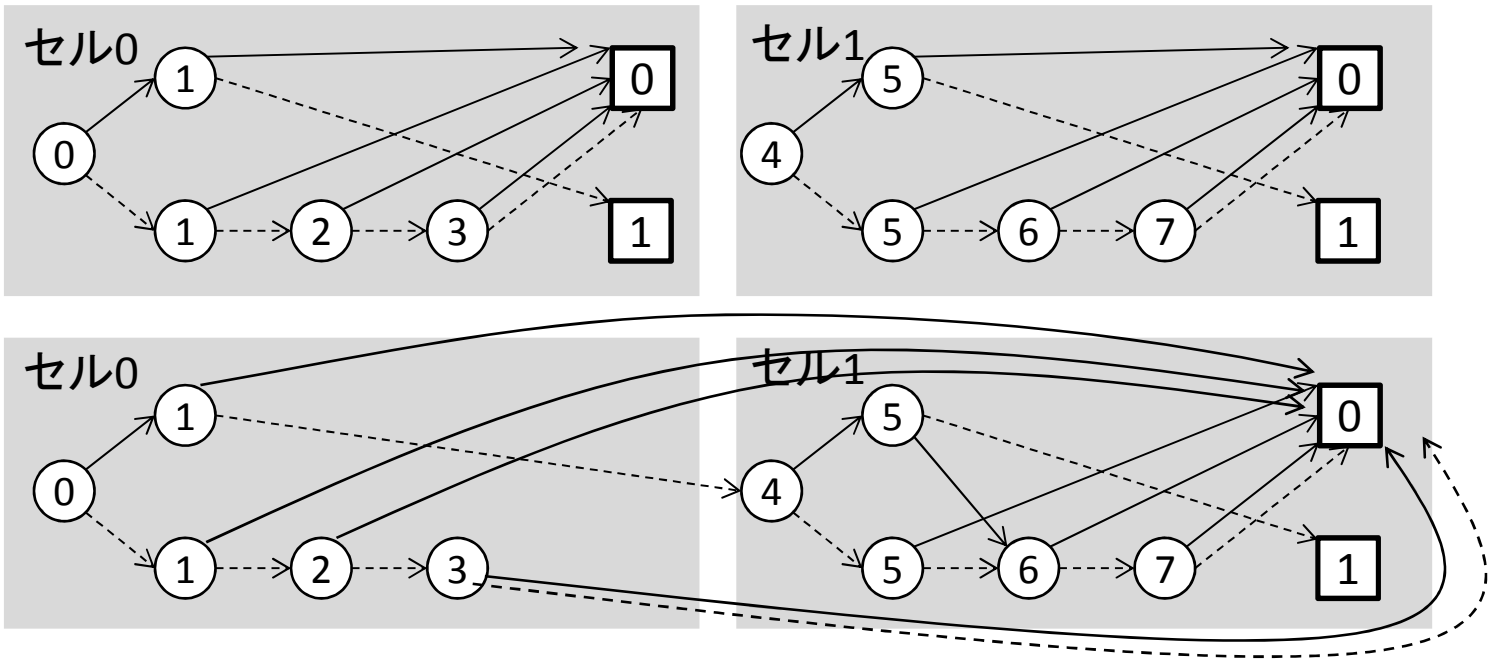
（セルのIndex:割り当てられる可能性のある避難所）の表現
(0:1),(1,1),(2:1,9),(3:1,9),(4:1,9),(5:1,18),(6:1,18),(7,1,9,18),(8:1,9,18),(9:9),(10:9,23)・・・・

ZDDへの変換
index=0のセル
　避難所セルに連番を振る⇒1=0,9=1,18=2,23=3⇒(0:0)
index=1のセル
　避難所セルに連番を振る⇒1=4,9=5,18=6,23=7⇒(1:4)
index=2のセル
　避難所セルに連番を振る⇒1=8,9=9,18=10,23=11⇒(2:8,9)
・・・・・
index=nの場合、避難所セルのindexは4n,…,4n+3で表現される

ZDDへの変換による分割の列挙

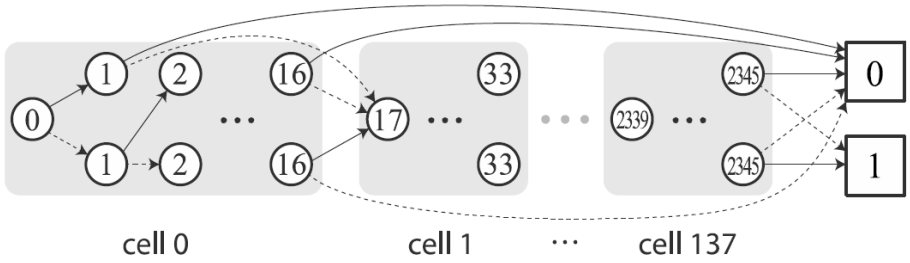
ZDDへの変換
index=0のセル
　避難所セルに連番を振る⇒1=0,9=1,18=2,23=3⇒(0:0)
index=1のセル
　避難所セルに連番を振る⇒1=4,9=5,18=6,23=7⇒(1:4)

セル0とセル1を足し合わせる（積の演算）

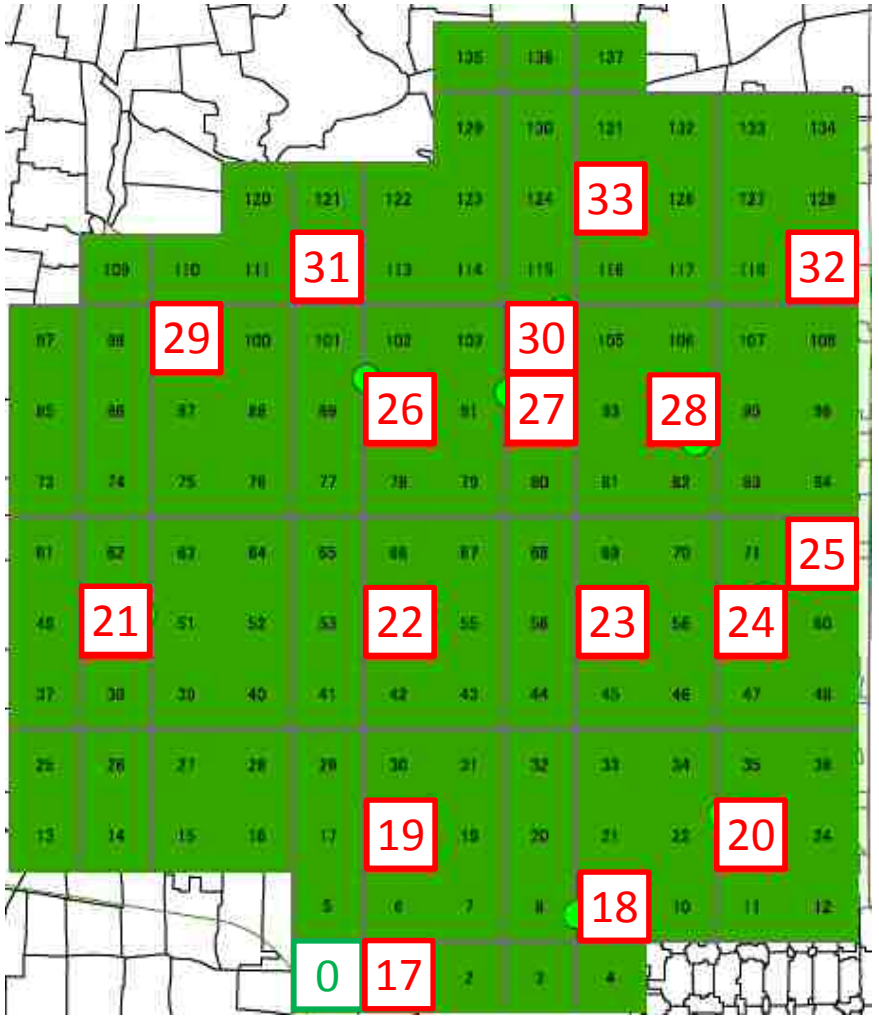


以降、全てのセルのZDDを1つずつ足し合わせて上京区全体の割当を求める

ZDDへの変換による分割の列挙



- 各〇：ZZDノード
- 番号0,1,・・・2345：各セルにおける避難所のインデックス（各セルのZZDノードnに対応する避難所のインデックス{1,・・・,17}は、$i=n \bmod 17+1$で表現される）
- 各セルごとにZDDを作成し、直積により全セルをまとめる。その際には、各セルのノードから1-枝が複数個出るものを排除するとともに、各セルの全ノードから0-枝が出ているものについても排除する。



ZDDへの変換による分割の列挙

ZDD：地域全体の割当の集合を保存

1. $ZDD \leftarrow \varnothing$, construct $OZDD$ and $SZDD$
2. **for** $i \leftarrow 1$ **to** 17
3. Apply Algorithm 1 for evacuation center c_i

and obtain the set of regions $\Re_i$

4. Construct ZDD_i for $\Re_i$
5. **if** i **is** 1
6. $ZDD \leftarrow ZDD_i$

i=1のとき、避難所i=1の割当の集合を保存するZDD1をZDDとして設定

7. **else**
8. $ZDD \leftarrow ZDD \times ZDD_i$

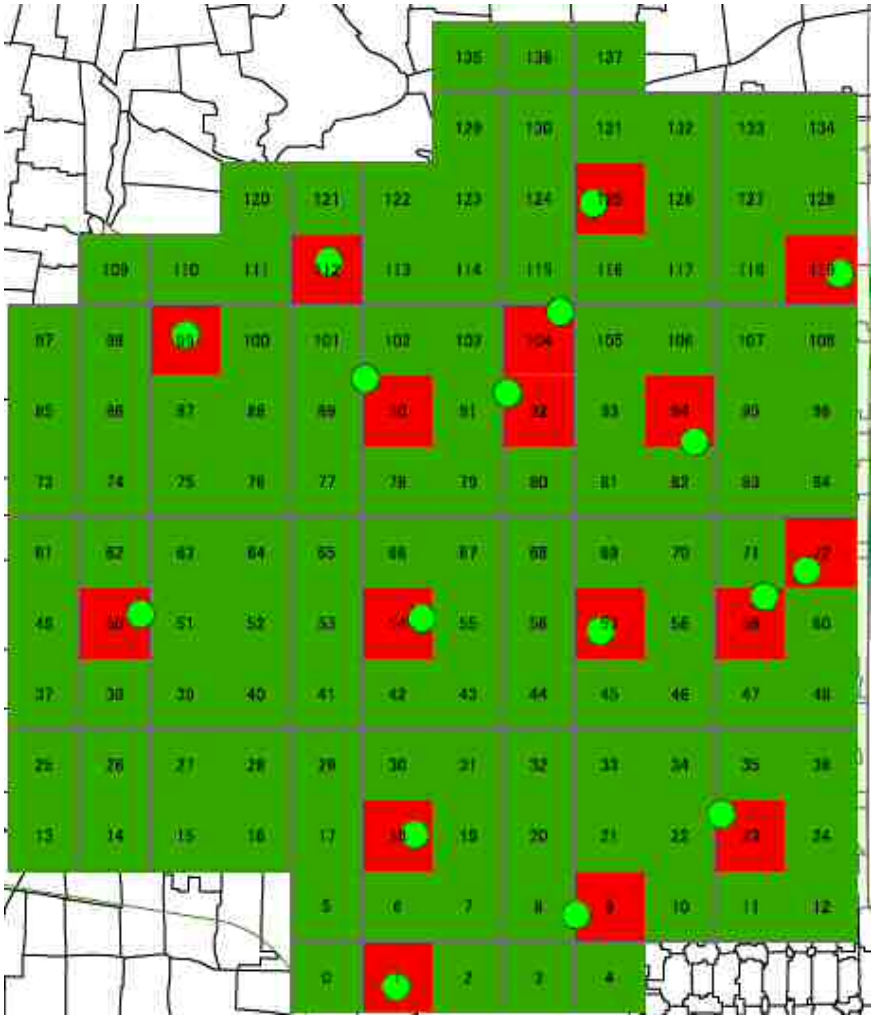
i>2以降は、ZDD×ZDDiによりZDDを更新

10. $ZDD \leftarrow ZDD \setminus ZDD.Restrict(OZDD)$
11. $ZDD \leftarrow ZDD.Restrict(SZDD)$
12. **end if**
13. **end for**

単純な直積演算だと制約を満たさない場合があるので、

- 各セルに異なる2つ以上の避難所が割り当てられないようにする
- 各セルに必ずどこか1つの避難所が割り当てられるようにする

京都市上京区でのケーススタディ



id	capacity	id	capacity
1	320	10	432
2	212	11	473
3	396	12	131
4	280	13	375
5	352	14	187
6	288	15	288
7	204	16	384
8	356	17	298
9	320		

Condition 1.

- *the distance constraint*: 1,100m
- *the capacity constraint*: 1,500%
- the number of patterns output: 0

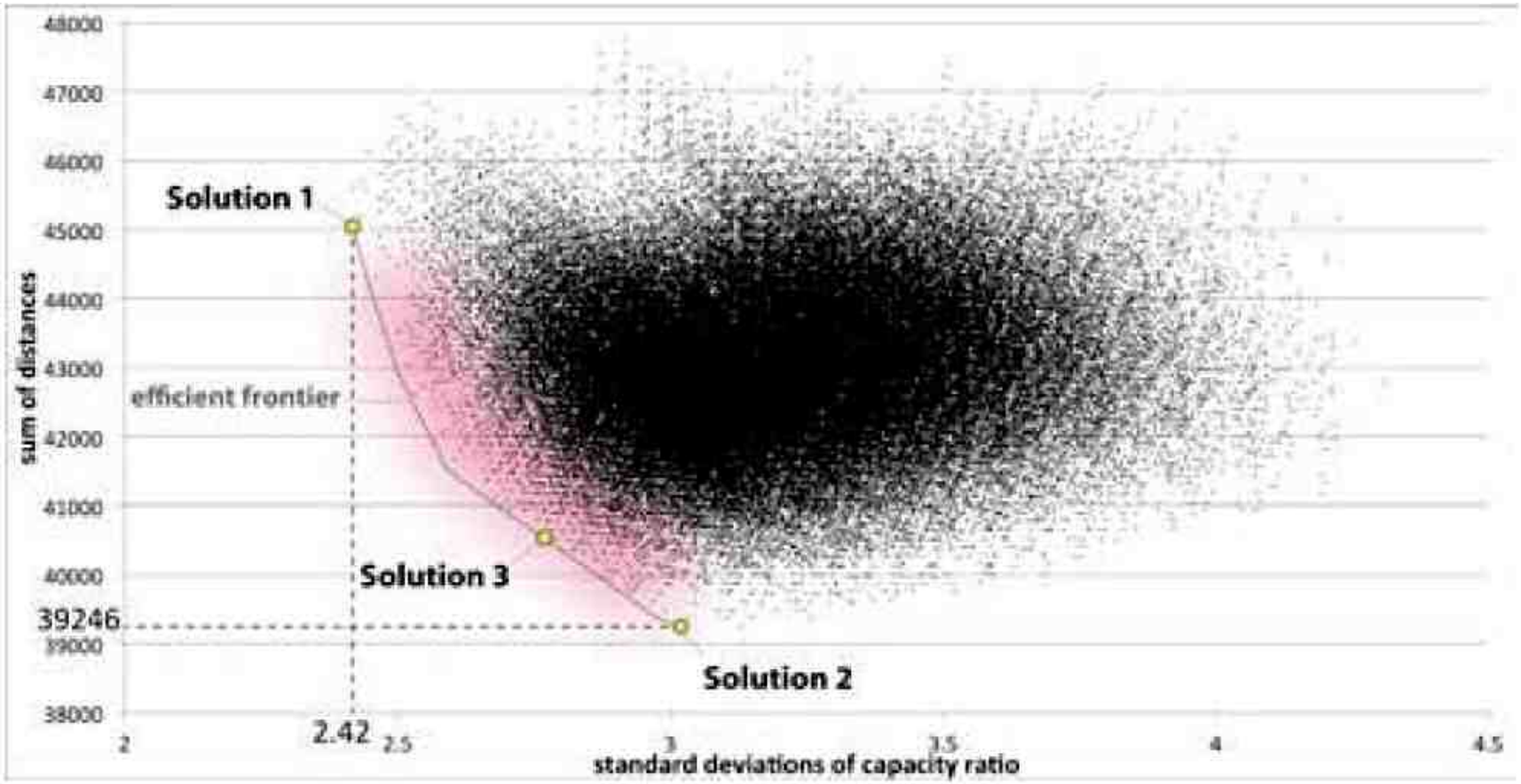
Condition 2.

- *the distance constraint*: 1,100m
- *the capacity constraint*: 1,600%
- the number of patterns output: 7,126,383

Condition 3.

- *the distance constraint*: 1,100m
- *the capacity constraint*: 1,545%
- the number of patterns output: 91,520

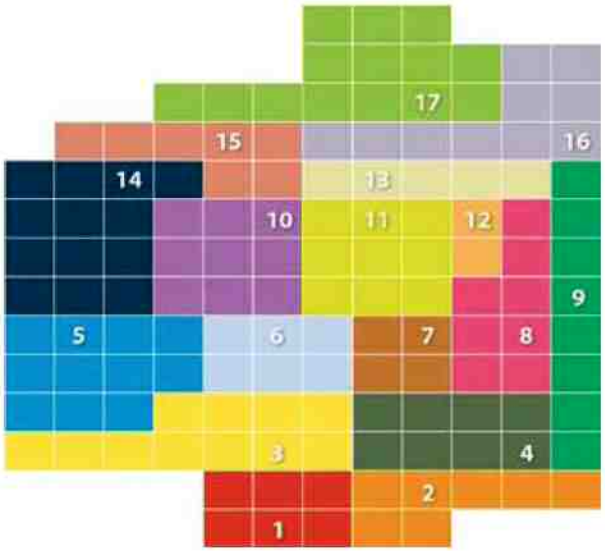
分割パタン91,520個の列挙結果の散布図



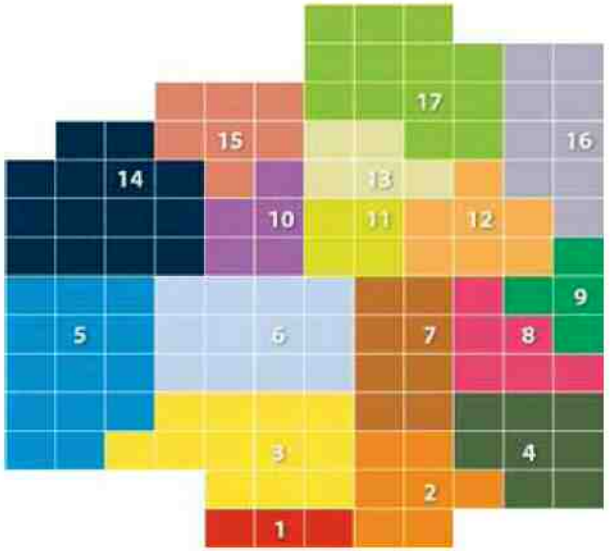
移動距離の和

収容率の標準偏差

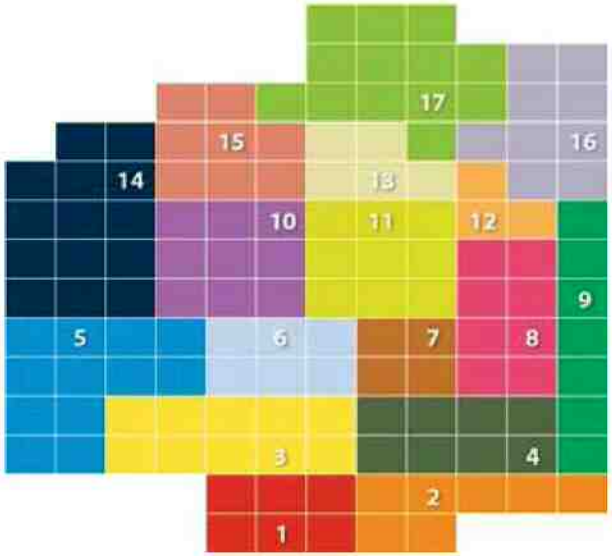
分割パタン計算結果



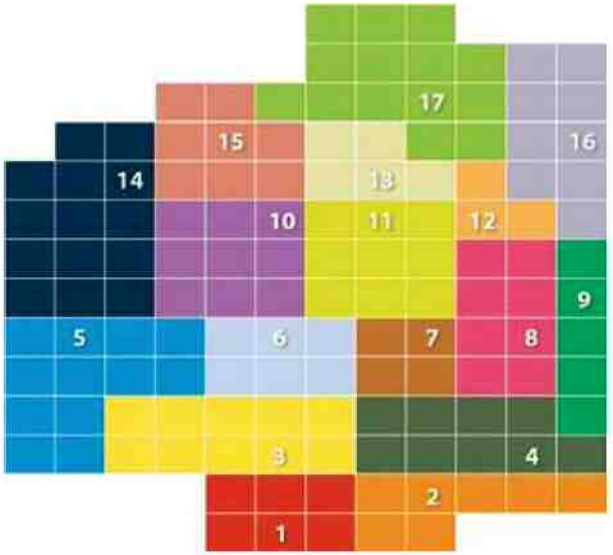
Solution 1（容量比標準偏差最小）



Solution 3（容量比標準偏差・距離総和がともに小）



Solution 2（避難距離総和最小）



制約条件なしSolution 2（最寄り避難所に避難）

所感

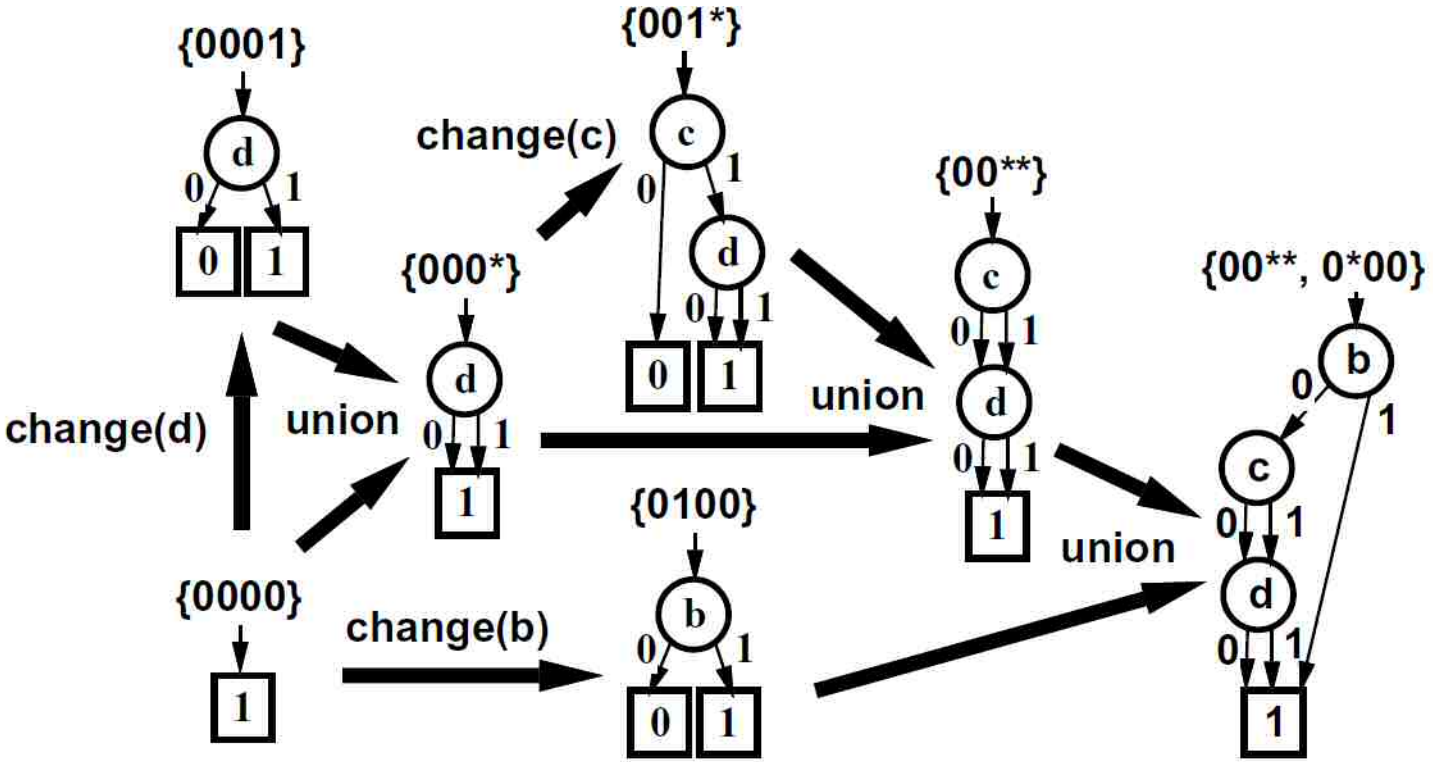
- ZDDは探索を繰り返す従来型のアルゴリズムではなく、複数の解をまとめて列挙し圧縮して出力する点が特徴
- ノードに保存する情報を加えることで、より複雑な制約にも対応可能（組み合わせ数の上限・下限、時間・距離制約、容量制約等）
- 制約を満たす組み合わせを効率的に全列挙する方法⇒活用の幅は広い（特にネットワークが疎な場合など、何らかの制約により実現不可能な組み合わせが相当数生じる場合に有効）
  - 避難所割り当て
  - 住宅レイアウト
  - 電力網・道路網・公共交通網　等

参考文献

- Kawahara, J., Inoue, T., Iwashita, H. and Minato, S.: Frontier-based search for enumerating all constrained subgraphs with compressed representation, Hokkaido University, Division of Computer Science, TCS Technical Reports, TCS-TA-A-14-76, 2014.
- Minato, S: Zero-suppressed BDDs and their applications, International Journal on Software Tools for Technology Transfer, Vol.3, No.2, pp.156-170, 2001.
- 湊真一: 超高速グラフ列挙アルゴリズム, 森北出版, 2015.
- 藤重悟: 離散構造とアルゴリズムII, 近代科学社, 1993.

ZDDの生成方法

- 二分木を構築して圧縮規制を適用⇒結局二分木のサイズが爆発して意味なし
- ZDDの生成方法
  - ① ZDD同士の集合演算に基づくZDDの生成方法
    - Change（追加）演算とunion（集合和）演算の組み合わせによりZDD成長
    - 計算が解けないこともある
  - ② フロンティア法



その他の部分グラフに対するZDDの構築

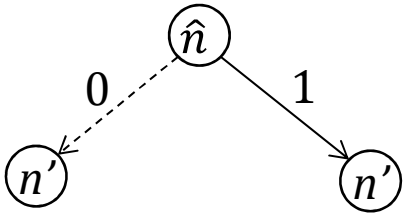
1. 全体グラフの全ての頂点を通る（ハミルトンs-t経路）
   - 全ての頂点の次数が0の場合、0-終端に繋ぐ
2. 採用した辺の個数
   - 採用した辺の個数をノード情報に格納する
3. サイクルの有無
   - サイクルが生じても0-終端に繋がない
   - サイクルが生じたことをノードに記憶させると、サイクルを含まなければならない部分グラフを列挙することも可
4. 確定した連結成分の個数（全域木）
   - 頂点がフロンティアからはずれる（連結成分が確定する）ごとに、連結成分をノードに記憶させる
5. 着目する頂点が同じ連結成分に含まれるか否か
   - 頂点対が同じ連結成分を持たない場合0-終端に繋ぐ

単一凸領域の列挙アルゴリズム

1. $R \leftarrow R_{root}$; $j \leftarrow 0$;　　ある避難所を含むセルだけからなる領域を根と定義する。

2. **repeat**
3. **while** $j < \delta(R)$ **do**　　適合セルの中で最も小さいindexを有するセルを選ぶ（繰り返し）
4. $j \leftarrow j+1$;　　もしなければAdj関数はnullを返し、あれば、R'を返す
5. **if** $Adj(R,j) \neq null$　　R'が制約条件を満たさない場合は選定しない
   and $f_{parent}(Adj(R,j)) = R$ **then**
6. $R' \leftarrow Adj(R,j)$;
7. **if** R' does not satisfy *the capacity constraint*
   or *the distance constraint* **then**
8. **go to** line 4;
9. **end if**
10. $R \leftarrow R'$; $j \leftarrow 0$;
11. *Output*(R);
12. **end if**
13. **end while**
14. **if** $R \neq R_{root}$ **then**
15. $R'' \leftarrow R$; $R \leftarrow f_{parent}(R)$;
16. $j \leftarrow 0$;
17. **repeat** $j \leftarrow j+1$ **until** $Adj(R,j) = R'$
18. **end if**
19. **until** $R = R_{root}$ **and** $j = \delta(R_{root})$

フロンティア法



Algorithm 1: ConstructDD

```
1  N_1 ← {n_root}. N_i ← ∅ for i = 2, ..., m + 1.
2  for i ← 1 to m do
3      foreach n̂ ∈ N_i do
4          foreach x ∈ {0, 1} do
5              n' ← CheckTerminal(n̂, i, x)
6              if n' = nil then
7                  Copy n̂ to n'.
8                  UpdateInfo(n', i, x)
9                  if there exists n'' ∈ N_i s.t. n'' is identical to n' then
10                     n' ← n''
11                 else
12                     N_{i+1} ← N_{i+1} ∪ {n'}
13             Create the x-arc of n̂ and make it point at n'.
```

ある$\hat{n}$に着目し、0/1-枝を作成

n'が0/1-終端ではない場合、$\hat{n}$のdegとcompの値をn'にコピーして更新

n'と同様の属性を持つn"が存在する場合、ノードの共有ができる

そうでなければ、n'をノードとして追加する

$\hat{n}$の0/1-枝を作成し、n'に向かわせる

フロンティア法

Algorithm 2: UPDATEINFO$(\hat{n}, i, x)$ for SIMPATH

```
1  Let e_i = {v, w}.
2  foreach u ∈ {v, w} such that u ∉ F_{i-1} do
3      n̂.deg[u] ← 0
4      n̂.comp[u] ← j such that u = v_j
       the index of u.
5  if x = 1 then
       // Increment the degrees of vertices v and w.
6      n̂.deg[v] ← n̂.deg[v] + 1
7      n̂.deg[w] ← n̂.deg[w] + 1
       // The two components are connected.
8      c_min ← min{n̂.comp[v], n̂.comp[w]}
9      c_max ← max{n̂.comp[v], n̂.comp[w]}
10     foreach u ∈ F_i do
11         if n̂.comp[u] = c_max then
12             n̂.comp[u] ← c_min
13 foreach u ∈ {v, w} such that u ∉ F_i do
14     Forget n̂.deg[u] and n̂.comp[u].
15 Renumber n̂.comp so that c(1), c(2), ..., c(n) are sorted in the ascending order,
   where c(i) is the minimum vertex v' satisfying n̂.comp[v'] = i.
```

辺に接続していない頂点u＝ノード$\hat{n}$とする。deg=0, compは辺に接続されているノードと異なる値を順番に与える

1-枝の場合、ノード$\hat{n}$と接続されるノードの次数が1増える

compの最大値をcmax、最小値をcminとする

$\hat{n}$のcompがcmaxと等しい場合は、cminに置き換える

一連の処理が終わったら、uに関するdeg, comp情報は破棄する（フロンティアから外して次に行く）

フロンティア法

Algorithm 3: CHECKTERMINAL$(\hat{n}, i, x)$ for SIMPATH

```
1  Let e_i = {v, w}.
2  if x = 1 then
3      if n̂.comp[v] = n̂.comp[w] then
       component.
4          return 0
5  Copy n̂ to n'.
6  UPDATEINFO(n', i, x).
7  foreach u ∈ {v, w} do
8      if (u = s or u = t) and n'.deg[u] > 1 then
9          return 0
10     else if (u ≠ s and u ≠ t) and n'.deg[u] > 2 then
11         return 0
12 foreach u ∈ {v, w} such that u ∉ F_i do
13     if (u = s or u = t) and n'.deg[u] ≠ 1 then
14         return 0
15     else if (u ≠ s and u ≠ t) and n'.deg[u] ≠ 0 and n'.deg[u] ≠ 2 then
       (i-2).
16         return 0
17 if i = m then
18     return 1
19 return nil
```

(5) vとwのcompが等しい＝同じ連結成分の場合、サイクルが生じる
⇒0-枝を生成

(1) sとtの次数が1以上⇒0-枝を生成

(2) sとt以外の次数が2以上
⇒0-枝を生成

(3) sとtの次数が1ではない⇒0-枝を生成

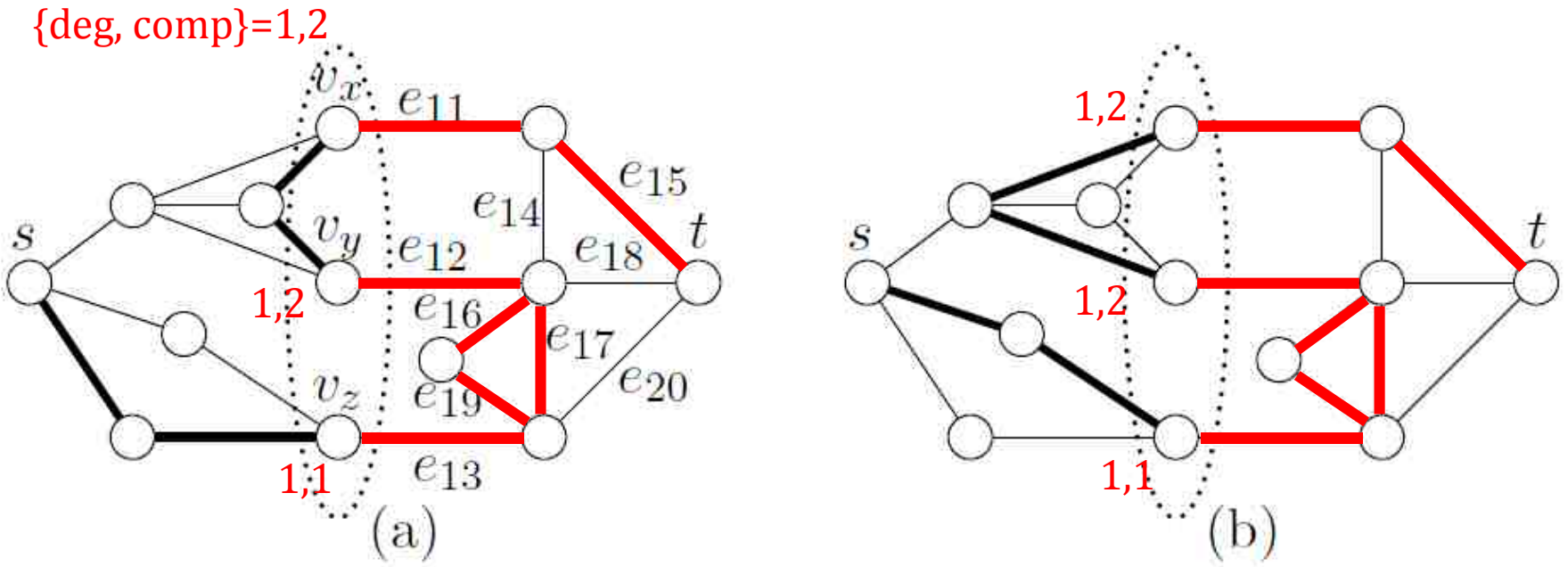
(4) 途中で分断⇒0-枝を生成

最後の辺の処理が終わるとs-t経路が完成⇒1-枝を生成

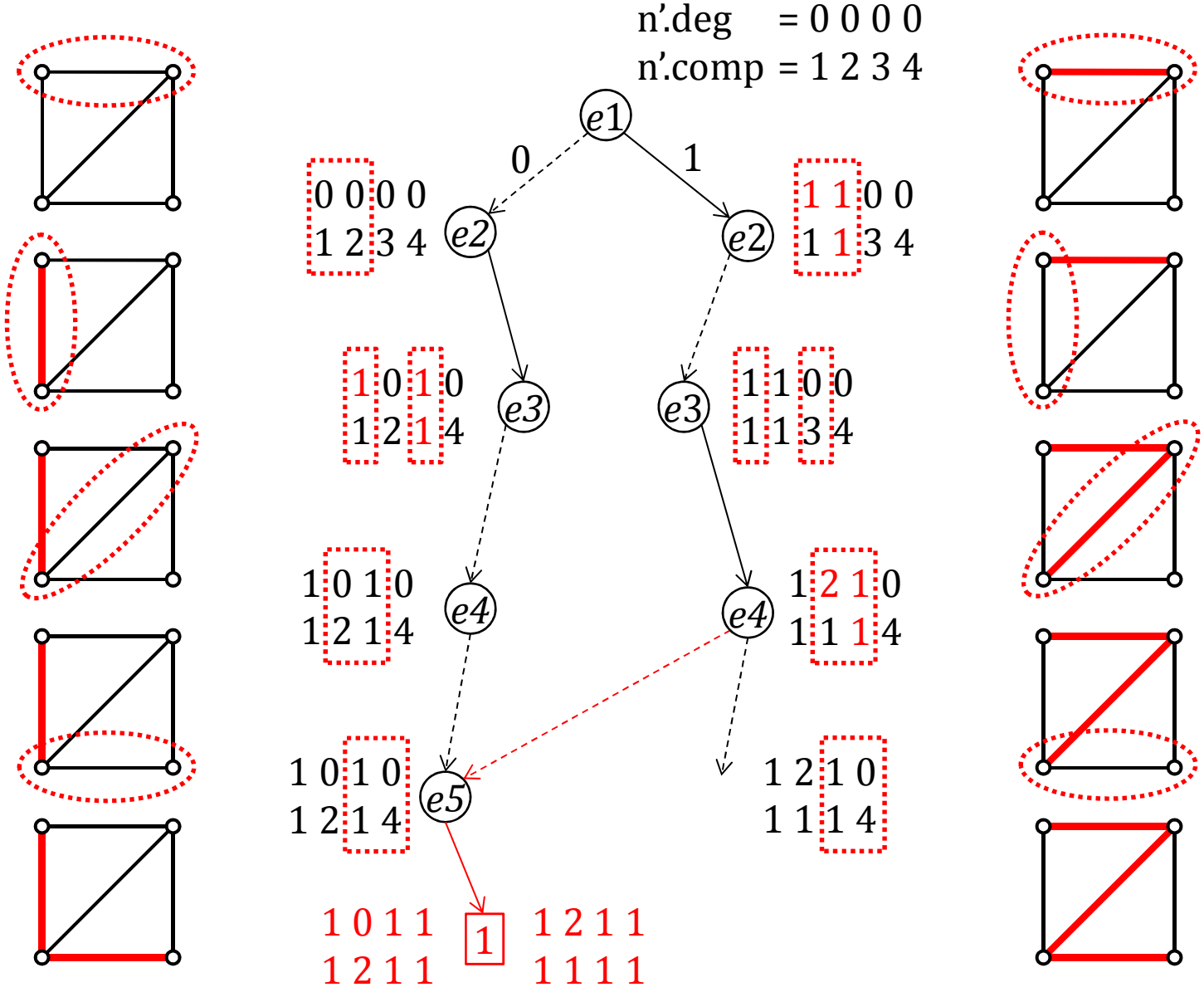
0/1-枝でなければ、nilを返す（次の辺の確認）

フロンティア法

- ノードの作成中にノードの共有ができるかどうか判定し、できるだけ共有する方法
- 破線内の頂点のdeg, comp値が同じであれば、左側の経路の完成形が同じになる
- このような楕円内の頂点集合をフロンティアと呼ぶ
- フロンティアの定義＝既に処理した辺と未処理の辺が両方接続している頂点の集合



フロンティア法


n'.deg = 0 0 0 0
n'.comp = 1 2 3 4
e1
0
1
0 0 0 0
1 2 3 4
e2
1 1 0 0
1 1 3 4
e2
1 0 1 0
1 2 1 4
e3
1 1 0 0
1 1 3 4
e3
1 0 1 0
1 2 1 4
e4
1 2 1 0
1 1 1 4
e4
1 0 1 0
1 2 1 4
e5
1 0 1 1
1 2 1 1
1
1 2 1 1
1 1 1 1
1 2 1 0
1 1 1 4