最短経路問題 & 最小全域木

情報知識ネットワーク研究室 B4 谷陽太

2016/05/30 HCPC勉強会 (2017/05/22修正)

目次

目次

例題

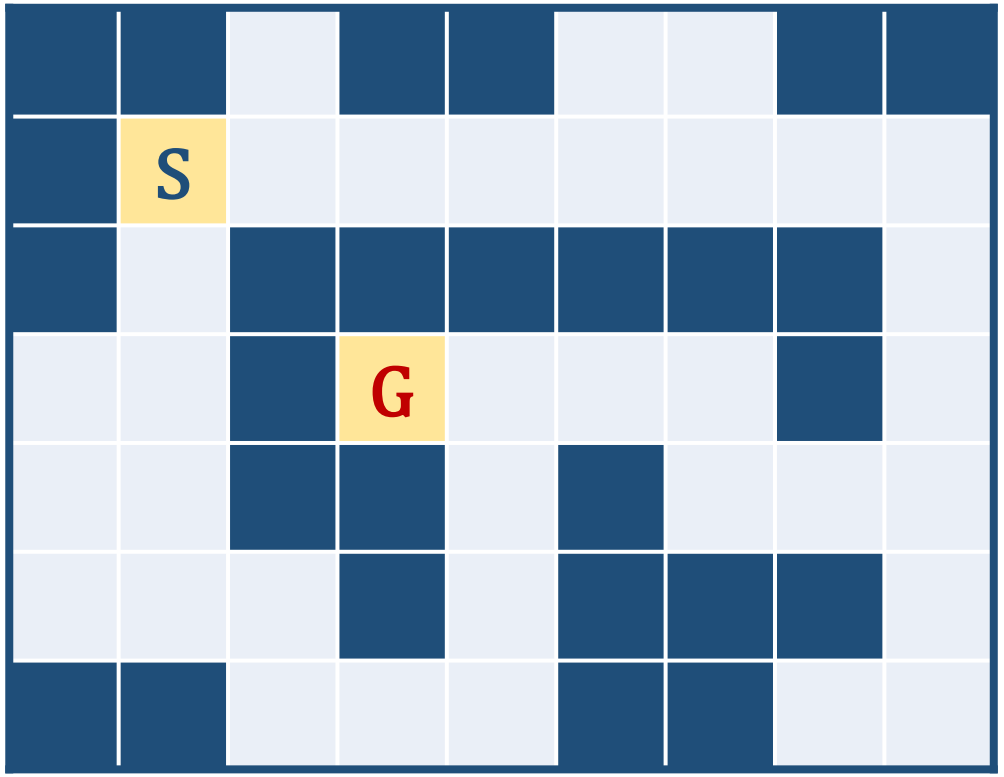
迷路が描かれたR×Cの盤面が与えられるので、
スタートからゴールまでの最少手数を求めよう！

- **Input**

  1行目 R C
  2行目 スタート地点の座標 Sy Sx
  3行目 ゴール地点の座標 Gy Gx
  4行目 盤面の情報 $c_{(1,1)}c_{(1,2)}\cdots c_{(1,c)}$
  ：
  r+3行目 盤面の情報 $c_{(r,1)}c_{(r,2)}\cdots c_{(r,c)}$

- **Constraints**

  $1 \leq \mathrm{R}, \mathrm{C} \leq 1000$



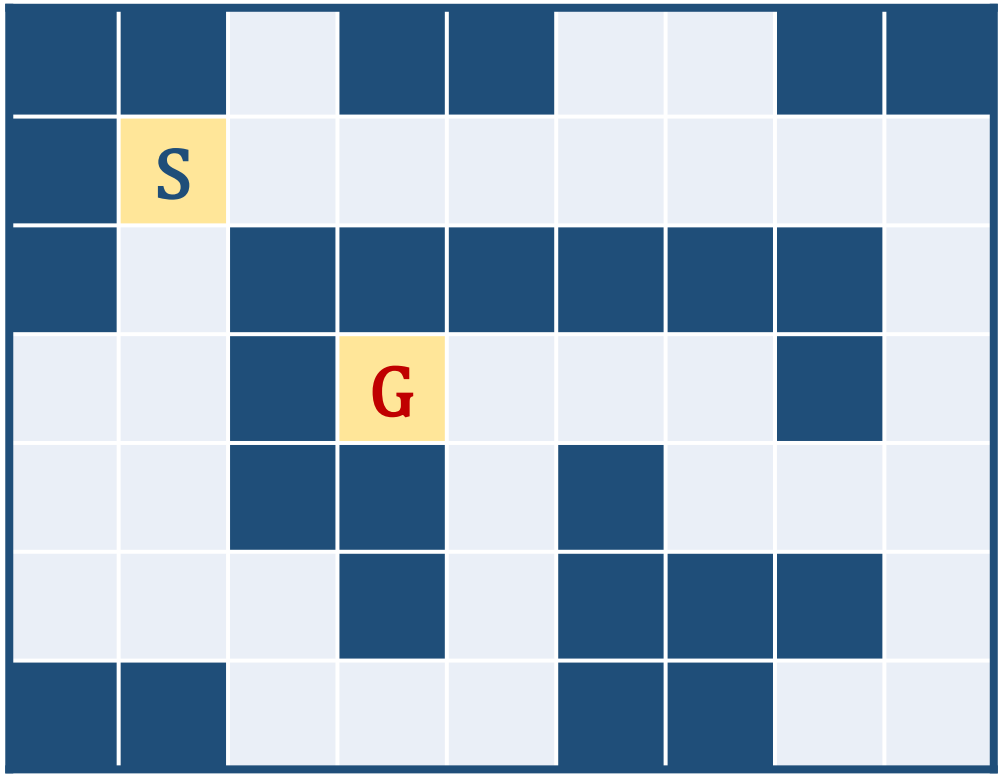
例題

迷路が描かれたR×Cの盤面が与えられるので、
スタートからゴールまでの最少手数を求めよう！

- **Sample Input**

```
7 9
2 2
4 4
##.##..##
#........
#.#######.
..#....#.
..##.#...
...#.###.
##...##..
```

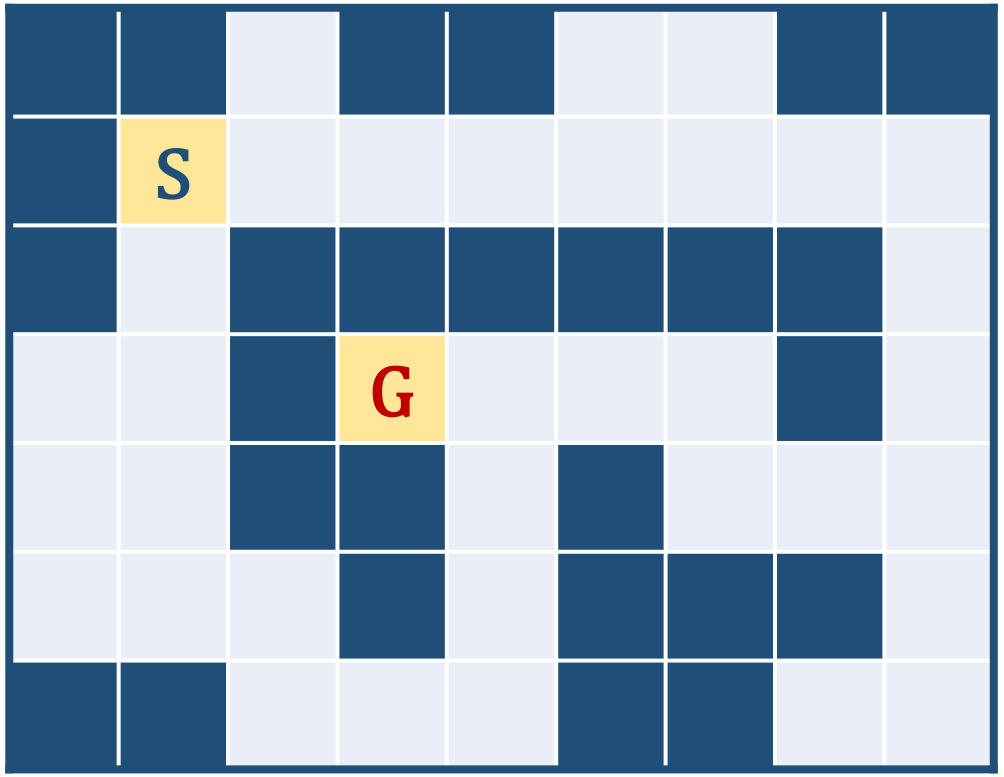
※ ‘.‘ は通路、
‘#’ は壁を表す



ちょっと考察 その1

- 例えば、G以外のすべての地点について、Sからの最少手数が求まっているとする

  このとき、Gまでの最少手数はどうやって求める？

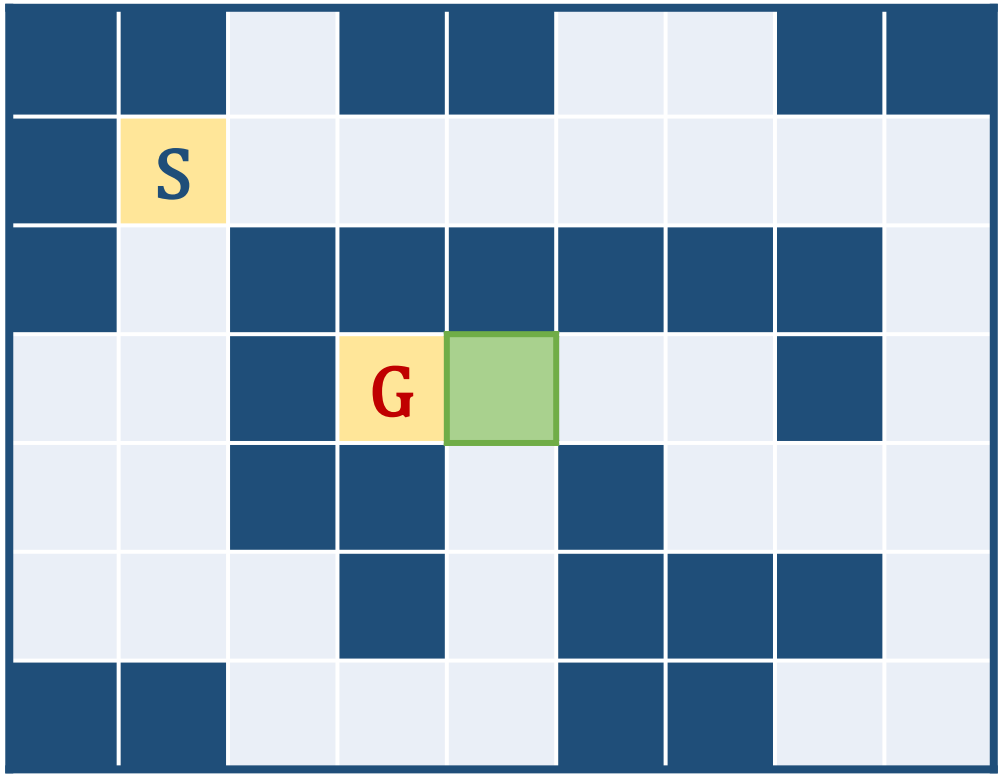


ちょっと考察 その1

- 例えば、G以外のすべての地点について、Sからの最少手数が求まっているとする

  このとき、Gまでの最少手数はどうやって求める？

  緑のマスまでの最少手数 + 1



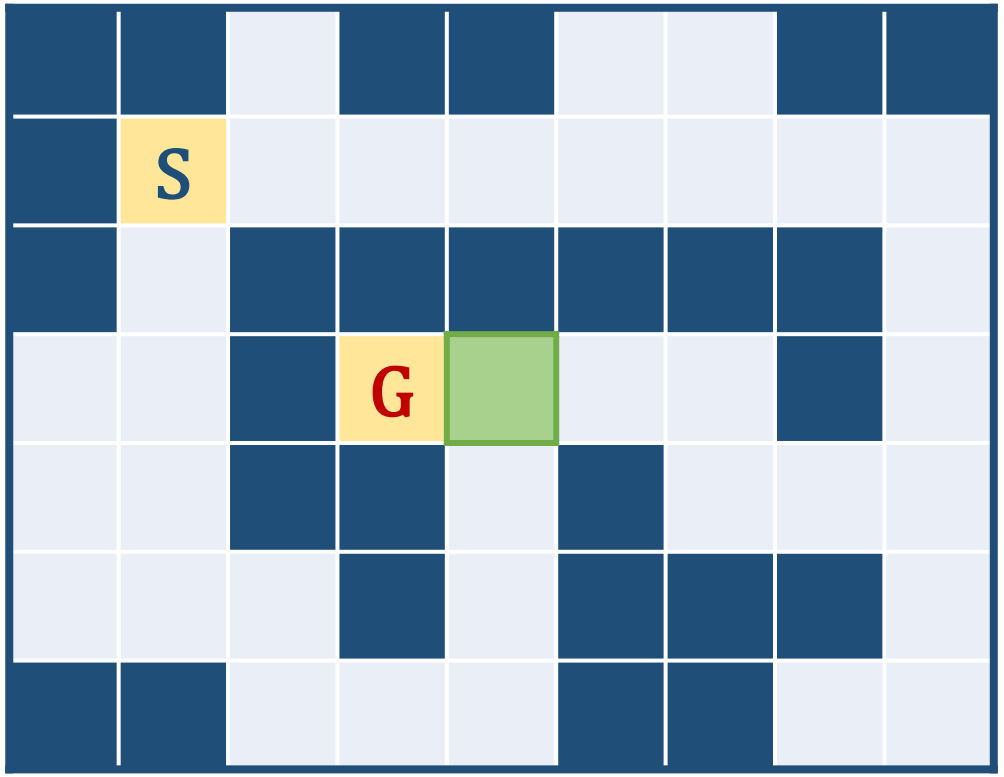
ちょっと考察 その1

- 例えば、G以外のすべての地点について、Sからの最少手数が求まっているとする

  このとき、Gまでの最少手数はどうやって求める？

  緑のマスまでの最少手数 + 1

- 緑までの最少も不明だったらそれはどう計算する？



ちょっと考察 その1

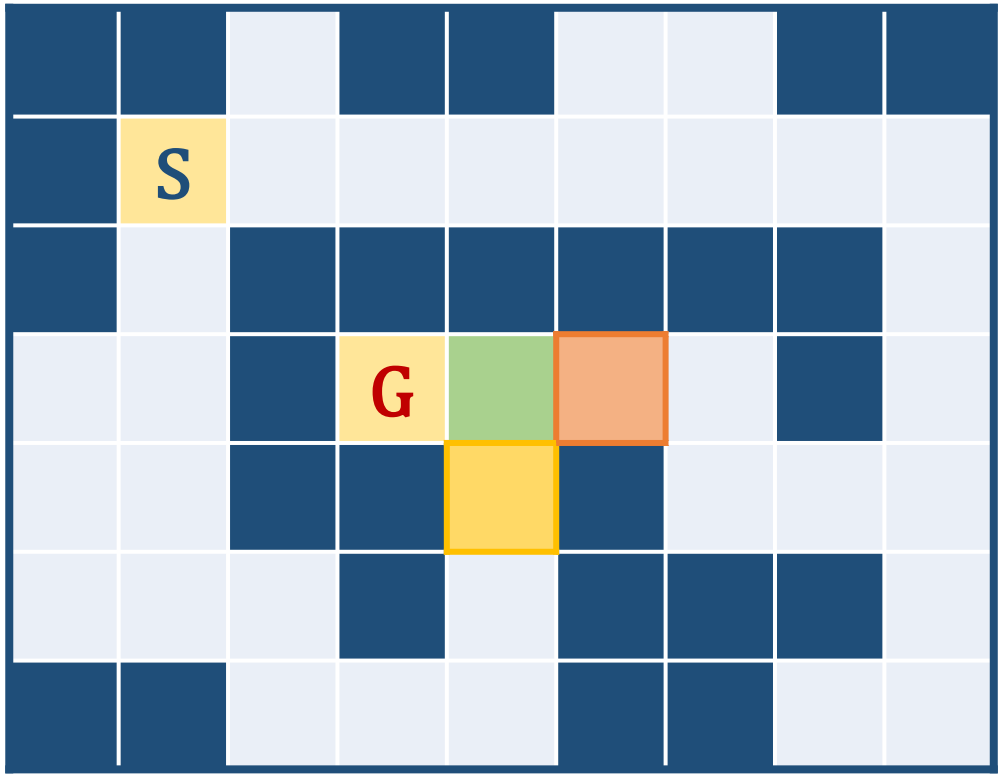
- 例えば、G以外のすべての地点について、Sからの最少手数が求まっているとする

  このとき、Gまでの最少手数はどうやって求める？

  緑のマスまでの最少手数 + 1

- 緑までの最少も不明だったらそれはどう計算する？

  min(黄色までの最少 + 1,
  オレンジまでの最少 + 1)



ちょっと考察 その1

- 例えば、G以外のすべての地点について、
  Sからの最少手数が求まっているとする

  このとき
  どうやっ

  緑のマス

- 緑までの
  それはどう計算する？

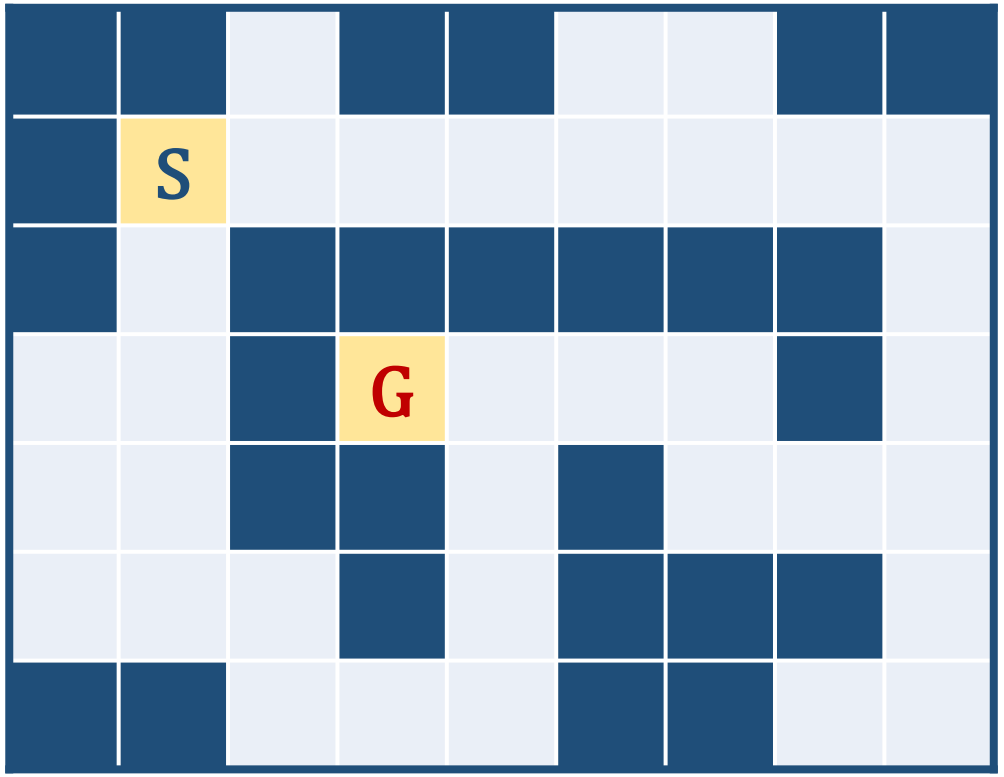
  min(黄色までの最少 + 1,
  オレンジまでの最少 + 1)

SからGまでの経路上の全地点について
最少手数が求まってる必要がありそう

ちょっと考察 その2

- さっきとは逆に、まだどの地点についても、Sからの最少手数が求まっていないとする

  このとき、即座に最少手数を求められる地点はどこ？

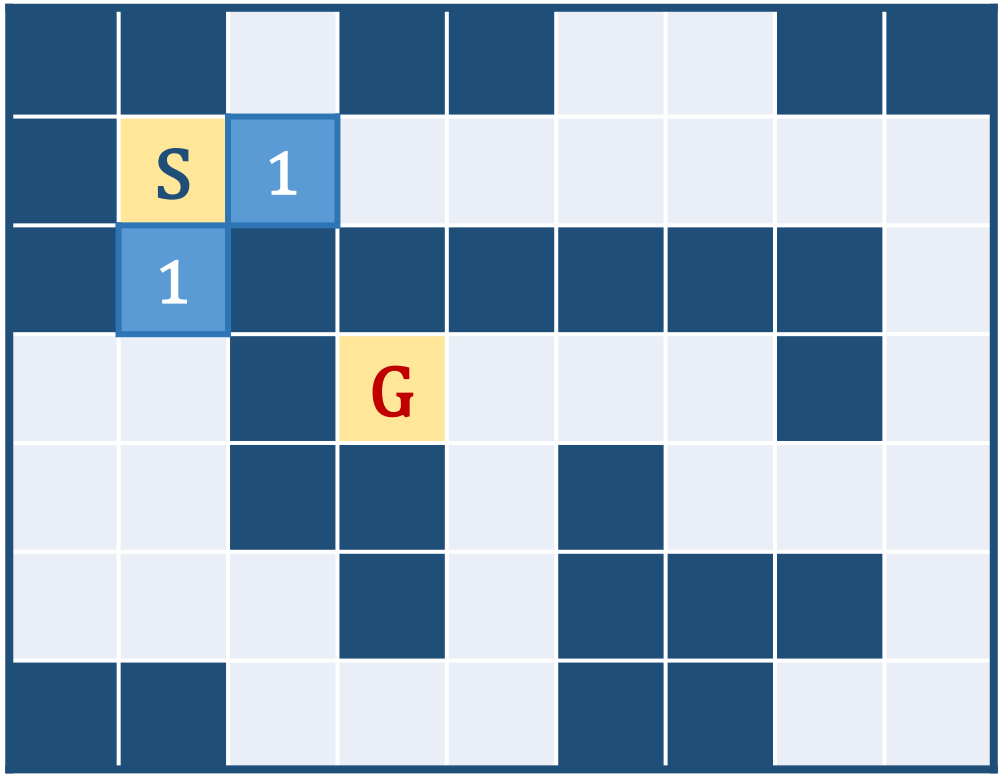


ちょっと考察 その2

- さっきとは逆に、まだどの地点についても、Sからの最少手数が求まっていないとする

  このとき、即座に最少手数を求められる地点はどこ？

  Sの隣の地点



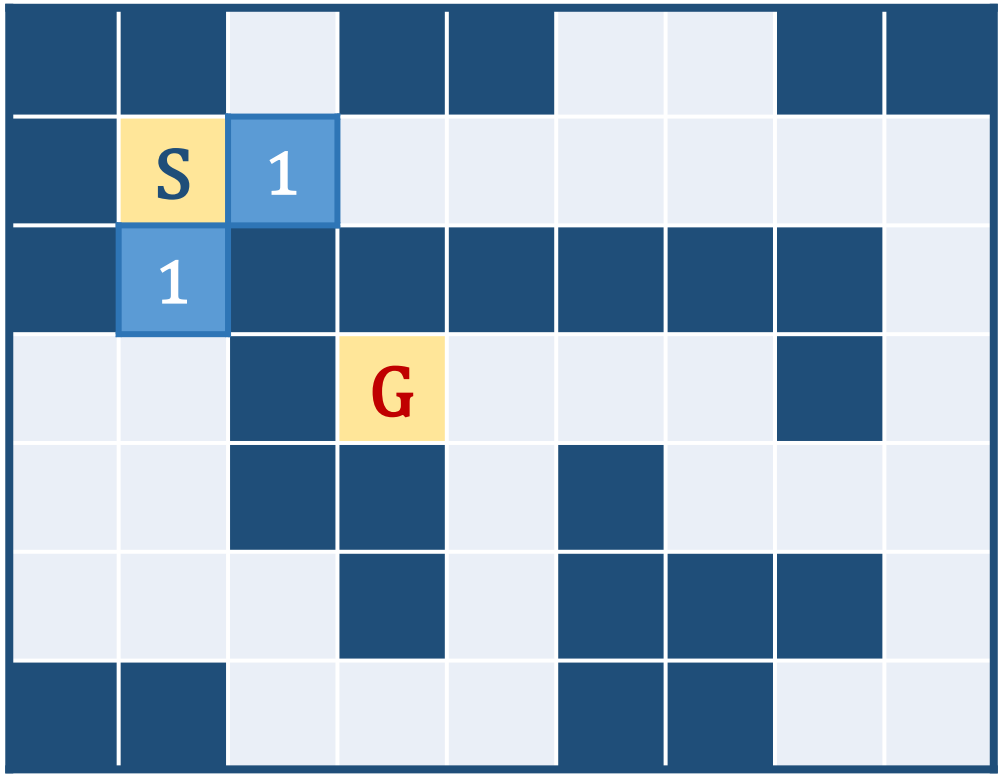
ちょっと考察 その2

- さっきとは逆に、まだどの地点についても、Sからの最少手数が求まっていないとする

  このとき、即座に最少手数を求められる地点はどこ？

  Sの隣の地点

- Sの隣までの最少手数が分かると何が嬉しい？



ちょっと考察 その2

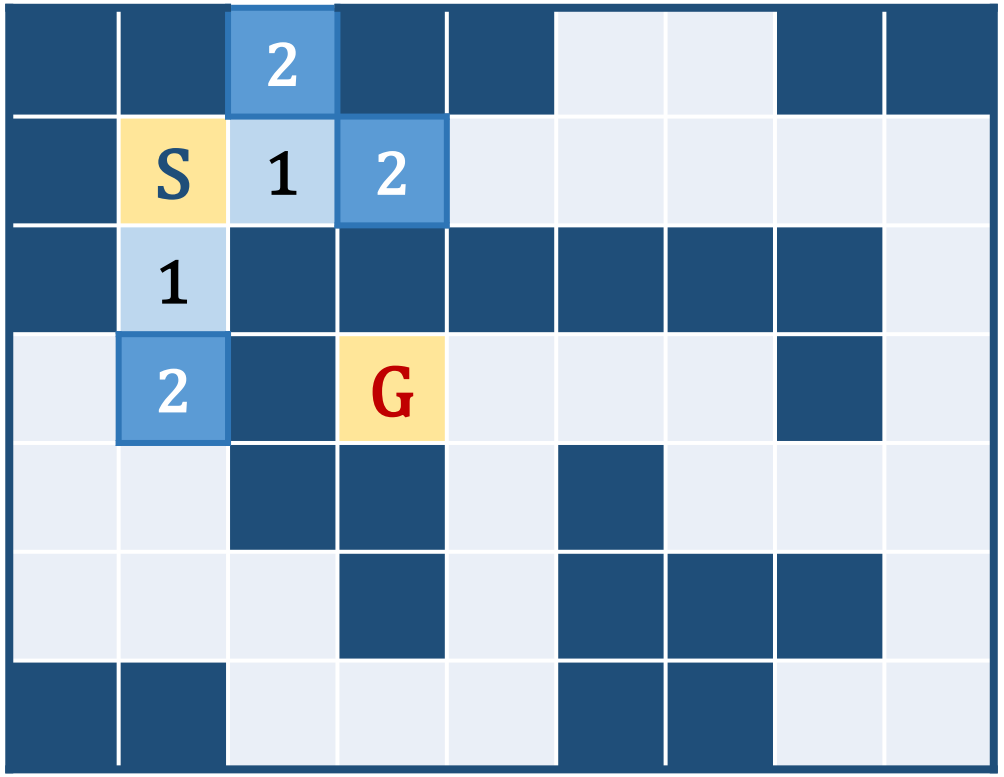
- さっきとは逆に、まだどの地点についても、Sからの最少手数が求まっていないとする

  このとき、即座に最少手数を求められる地点はどこ？

  Sの隣の地点

- Sの隣までの最少手数が分かると何が嬉しい？

  そのさらに隣も分かる！



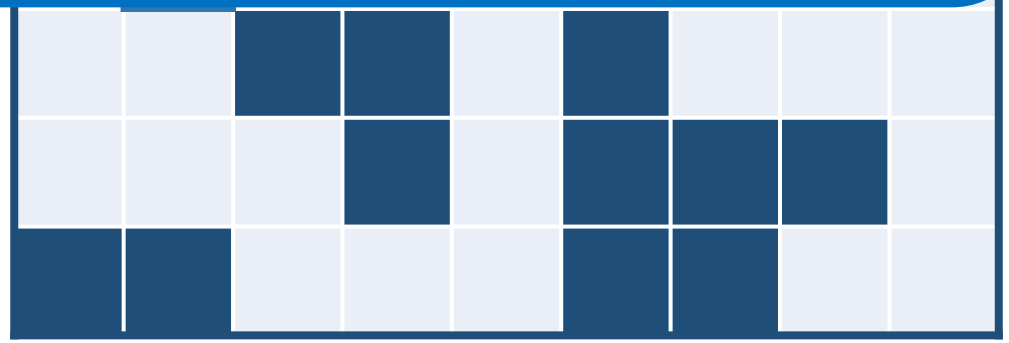
ちょっと考察 その2

- さっきとは逆に、まだどの地点についても、
  Sからの最少手数が求まっていないとする

この調子で、Gまで順に求めていけばいいのでは

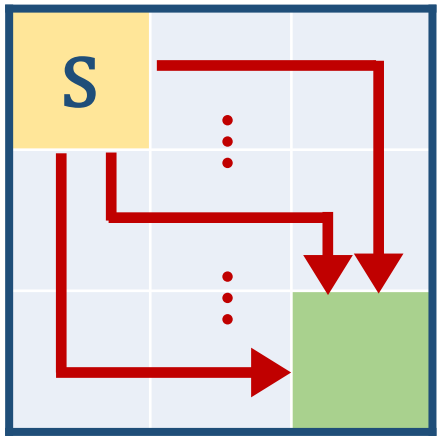
何が嬉しい？

そのさらに隣も分かる！

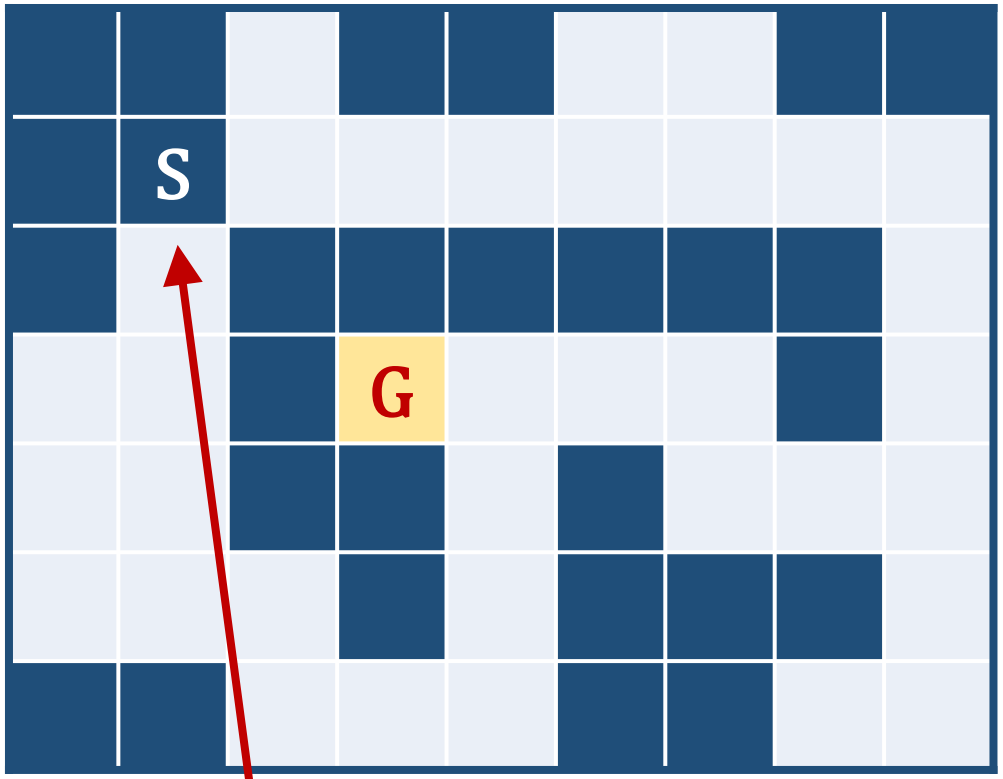


幅優先探索 (Breadth First Search)

- **方針: スタート地点から近い順に探索していこう！**
  ただし、一度探索した地点は
  二度と探索することのないように注意するよ！
  - 同じ地点について、何度も最少手数を計算し直すのは時間の無駄



幅優先探索の動き



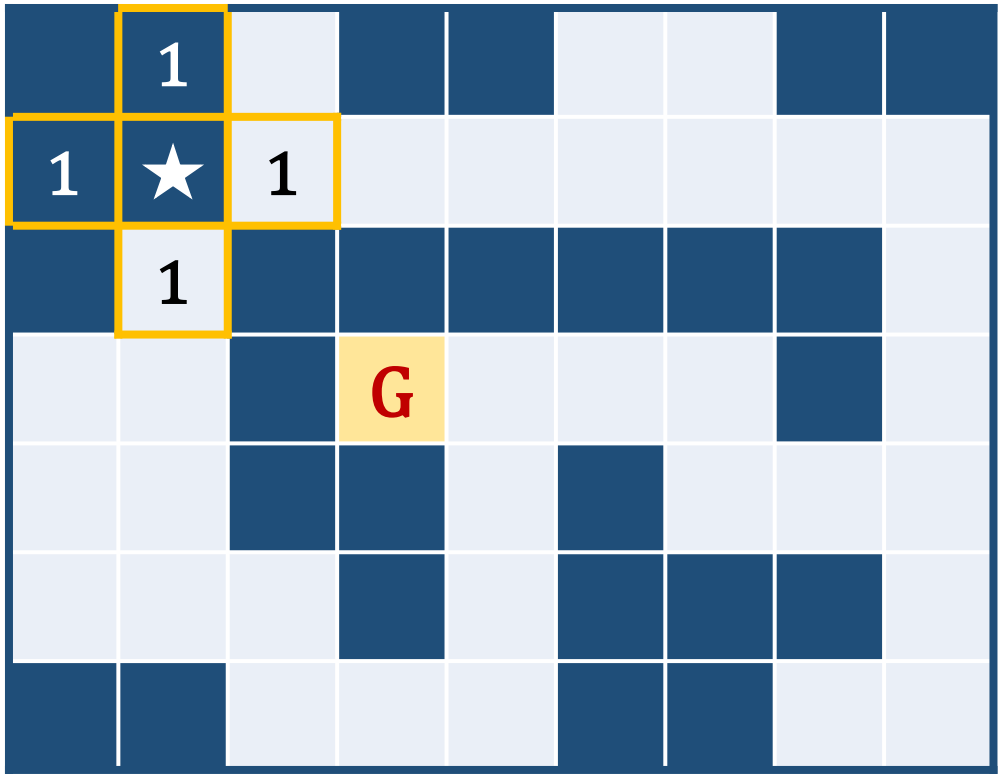
さらに、盤面上でSを埋めてしまう

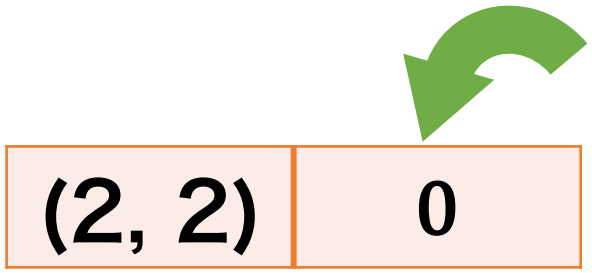
待機列 (登録順)

(2, 2)	0

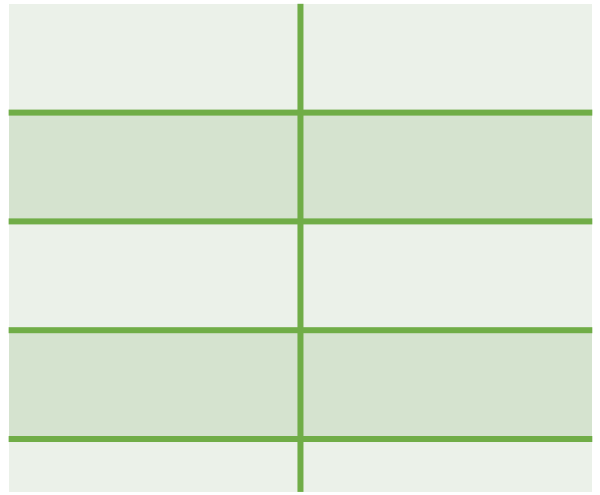
まず「列のようなもの」を用意して、
Sの座標 & SからSまでの最少手数を登録

幅優先探索の動き



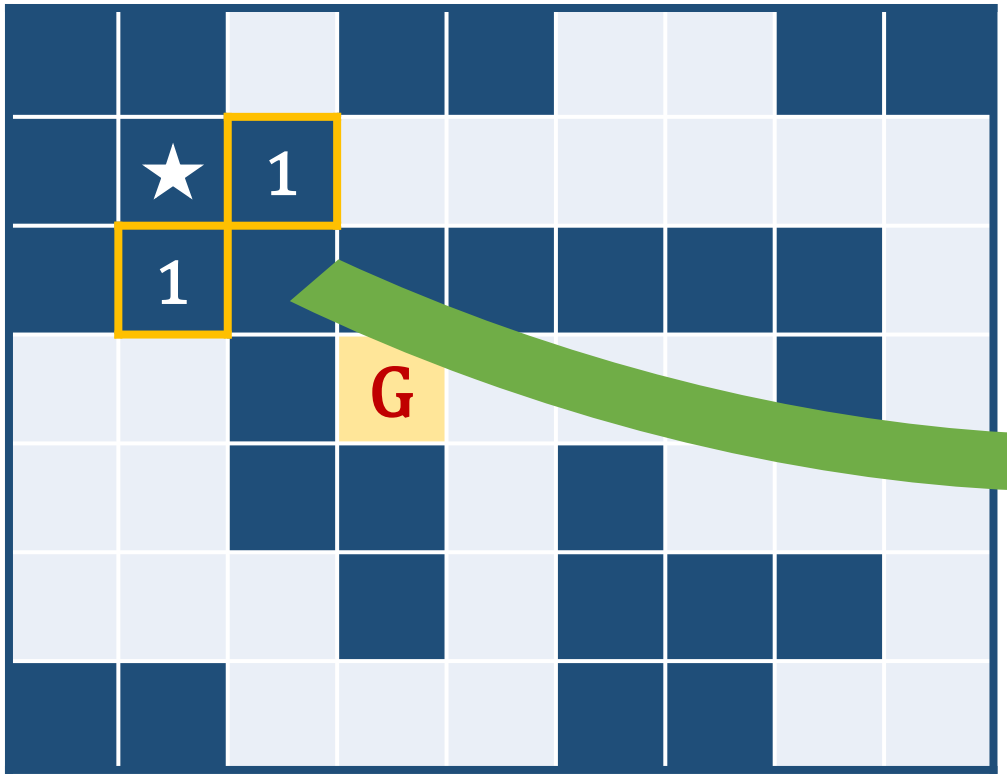


待機列 (登録順)



待機列の先頭を取り出してきて、
その隣接地点を確認

幅優先探索の動き



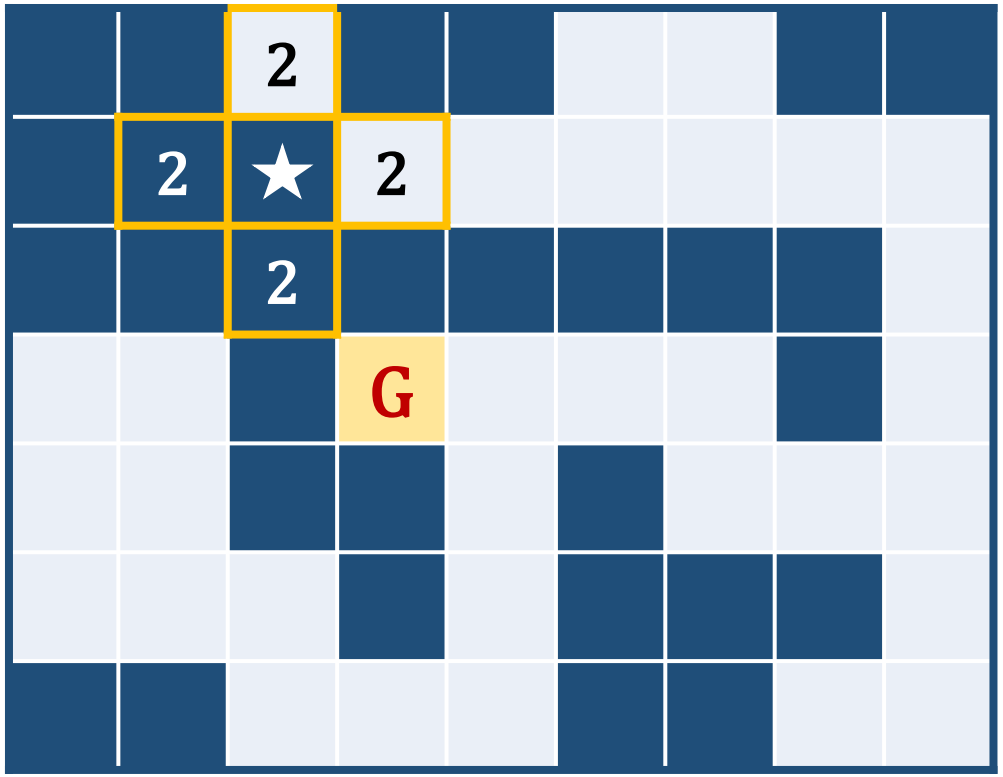
登録！

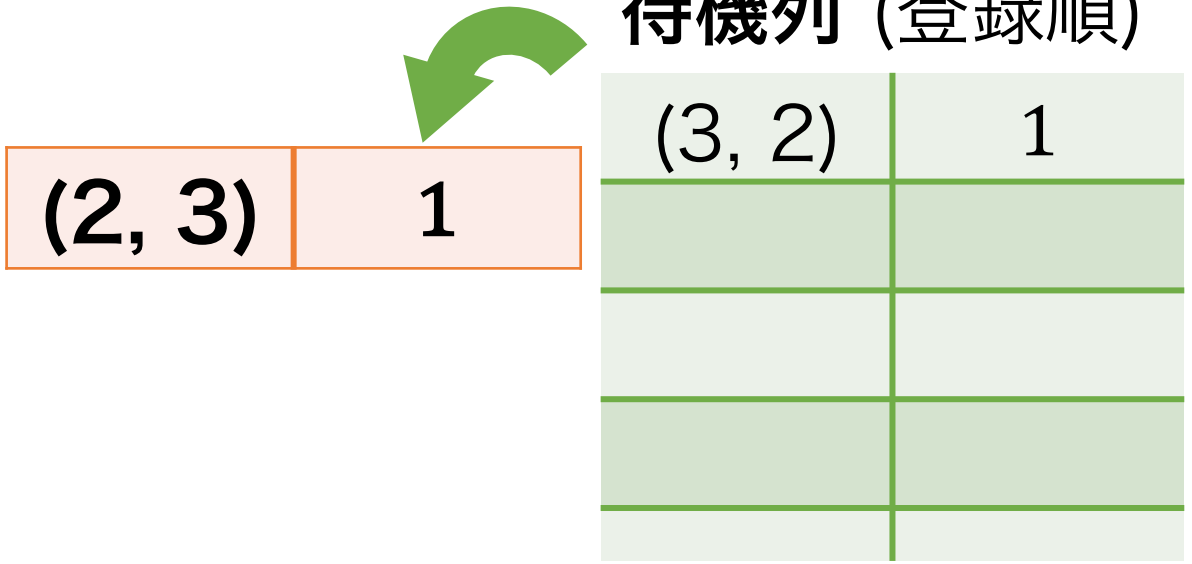
待機列 (登録順)

(2, 3)	1
(3, 2)	1

移動可能な地点を待機列に登録し、
盤面を埋める

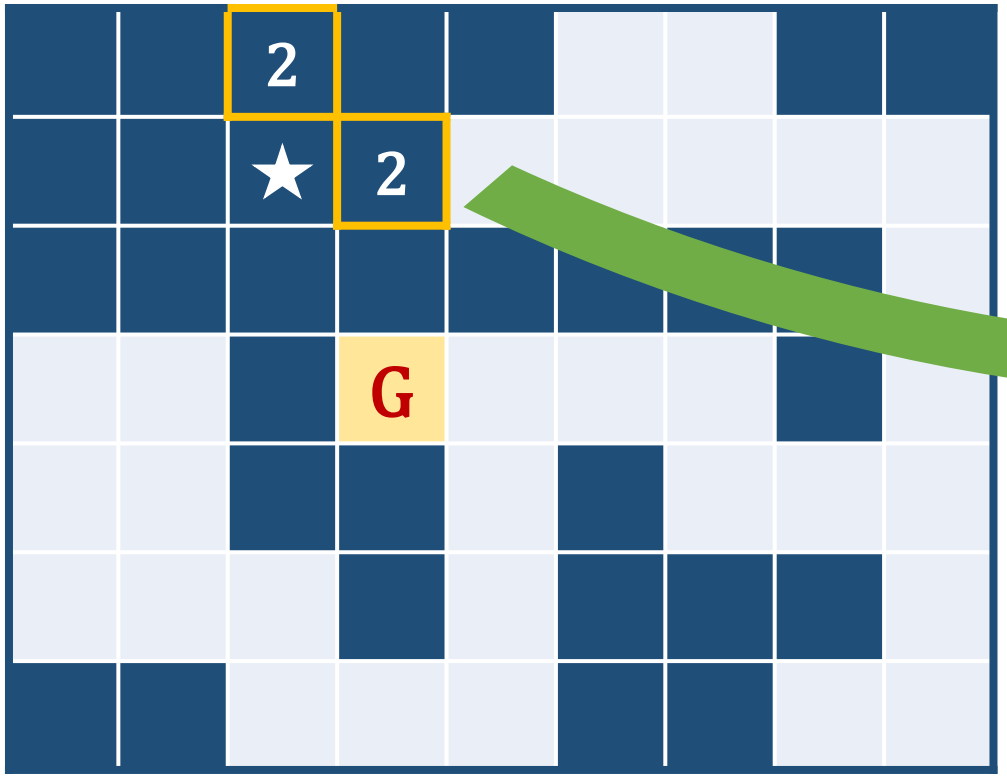
幅優先探索の動き





再び列の先頭を取り出して、
その周辺4つを確認

幅優先探索の動き

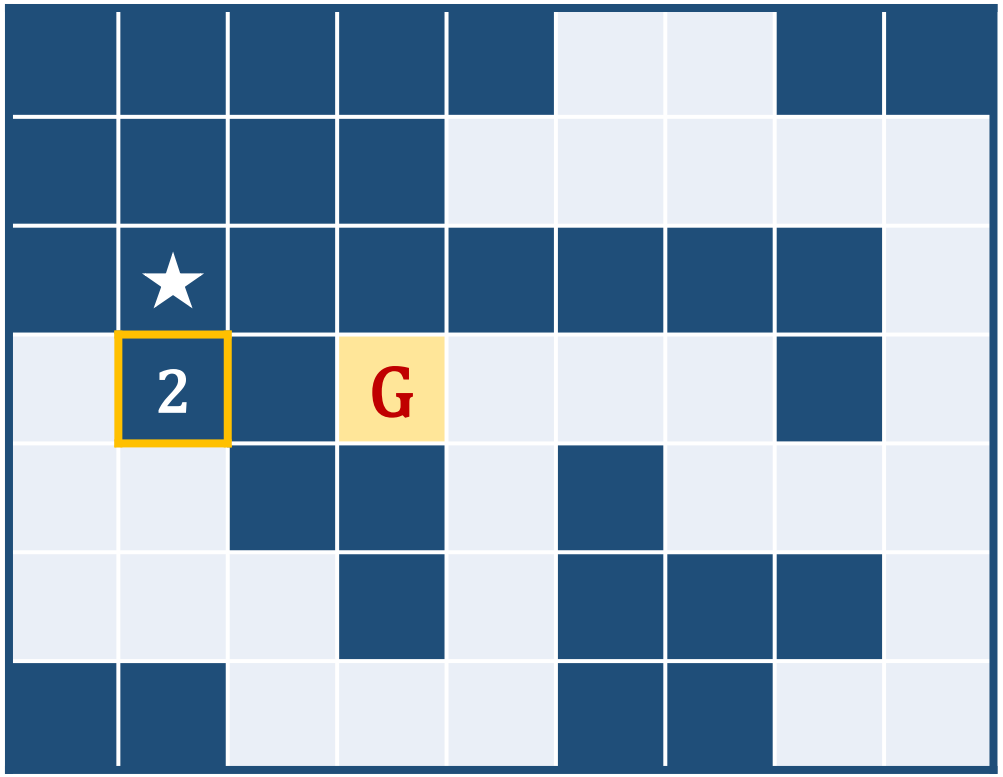


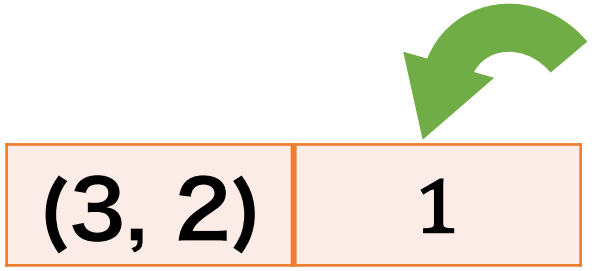
登録！

待機列 (登録順)

(3, 2)	1
(1, 3)	2
(2, 4)	2

幅優先探索の動き

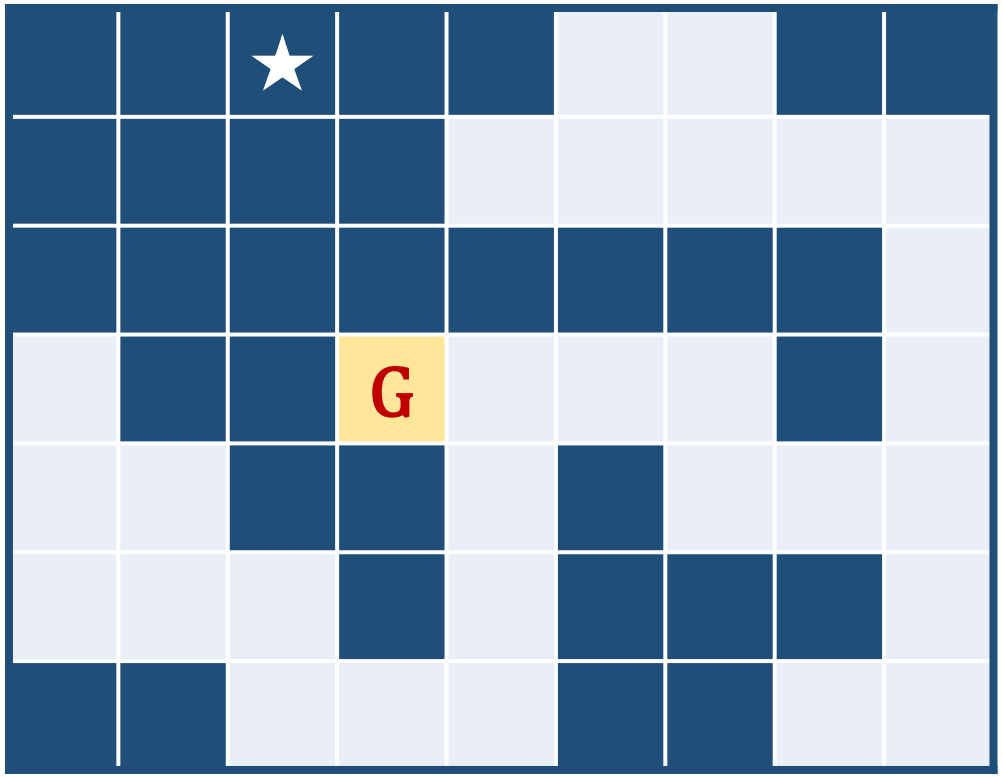




待機列 (登録順)

(1, 3)	2
(2, 4)	2
(4, 2)	2

幅優先探索の動き



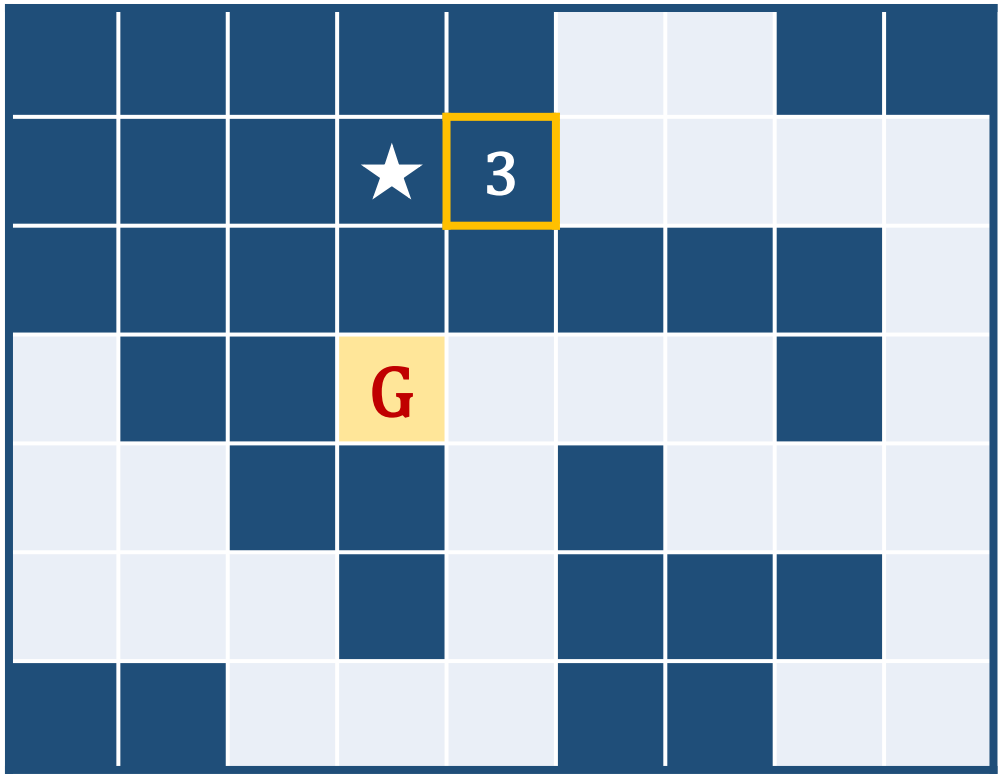
(1, 3)	2

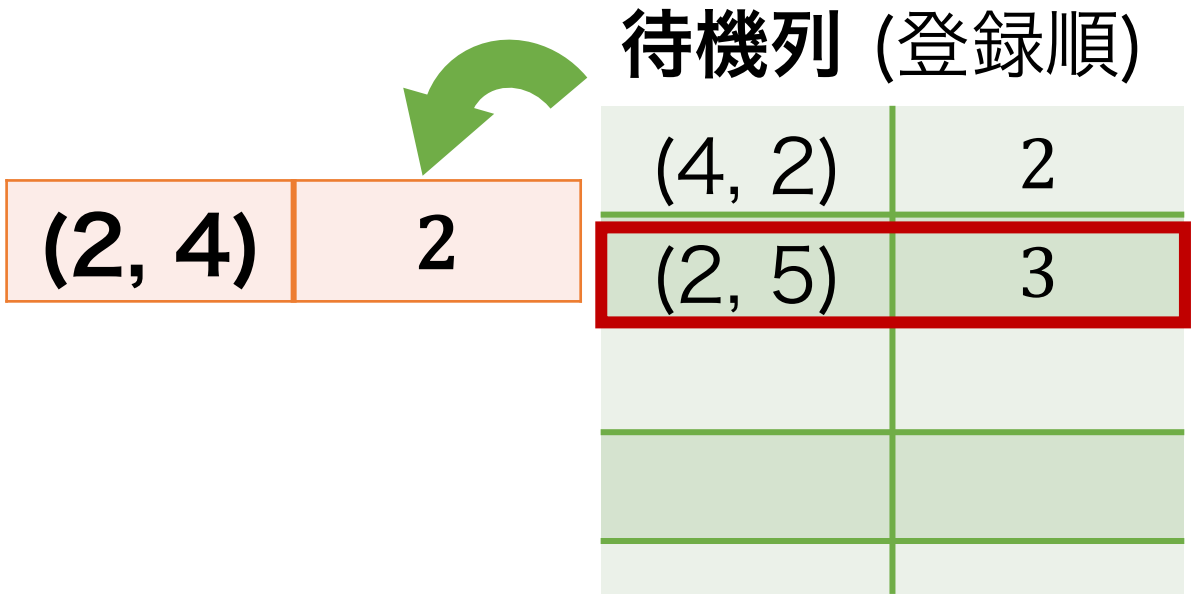
待機列 (登録順)

(2, 4)	2
(4, 2)	2

移動可能な隣接地点がない場合もある
(そのときは当然何もしない)

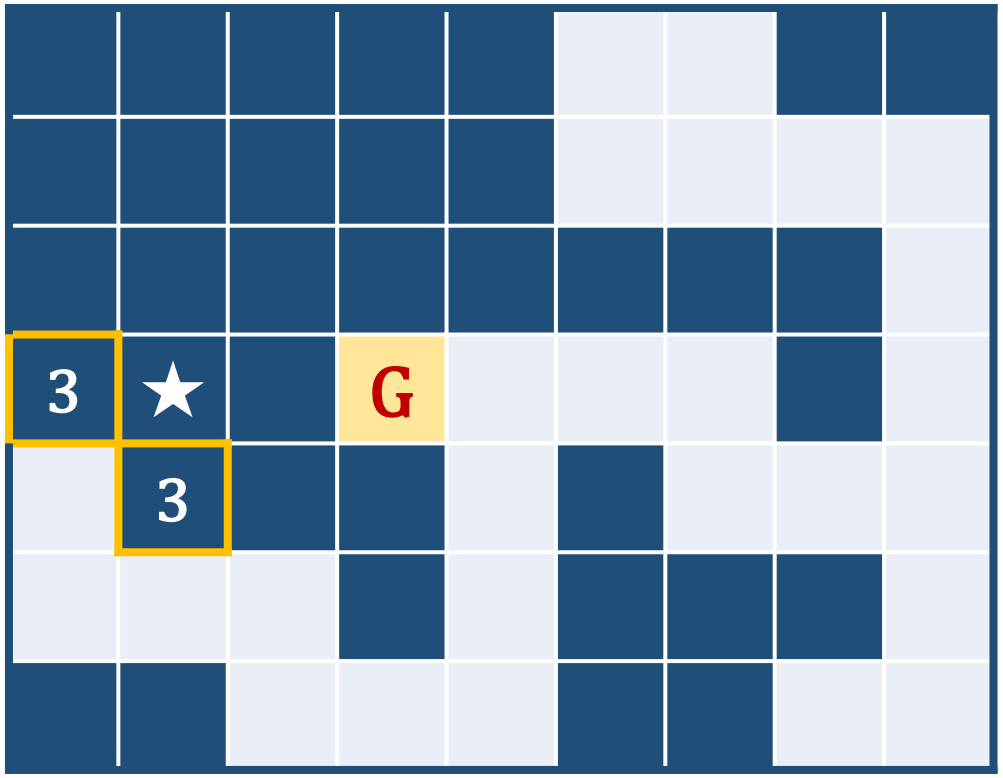
幅優先探索の動き

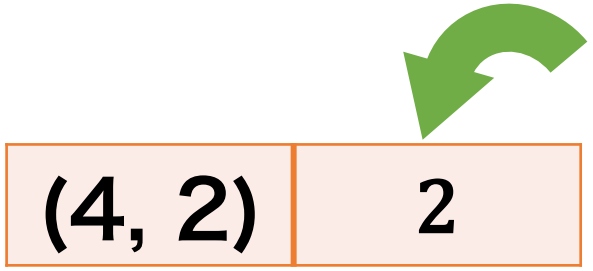




以下同様に……

幅優先探索の動き

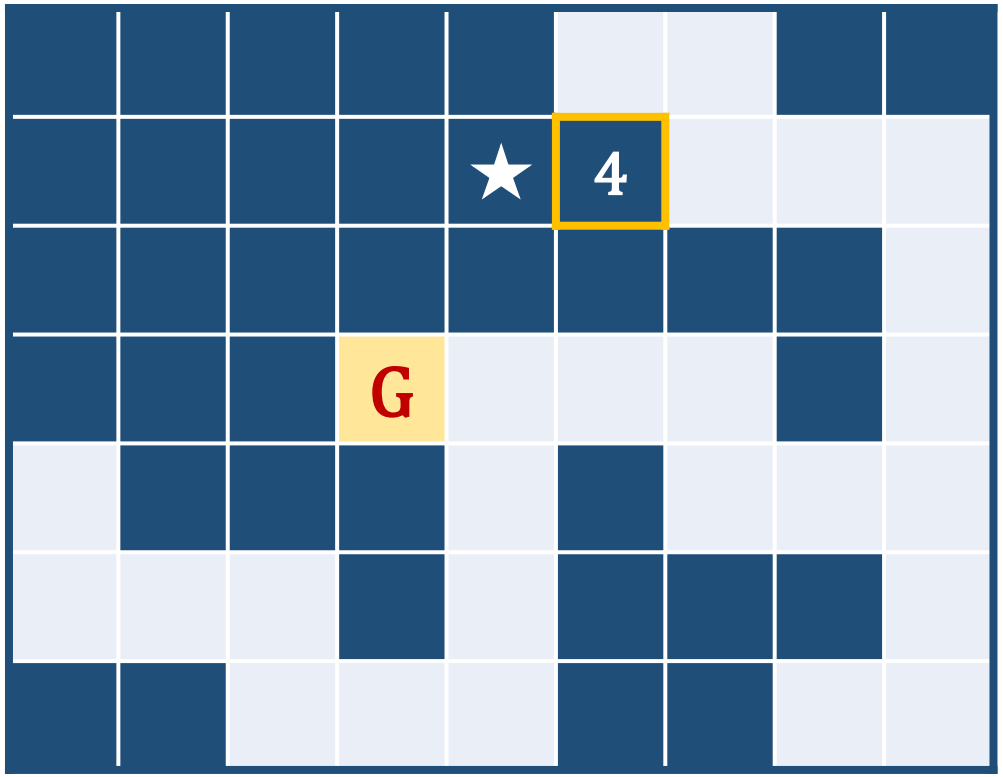


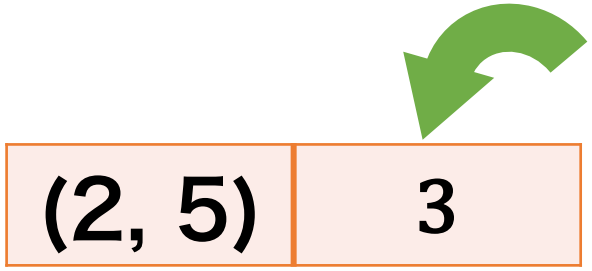


待機列 (登録順)

(2, 5)	3
(4, 1)	3
(5, 2)	3

幅優先探索の動き

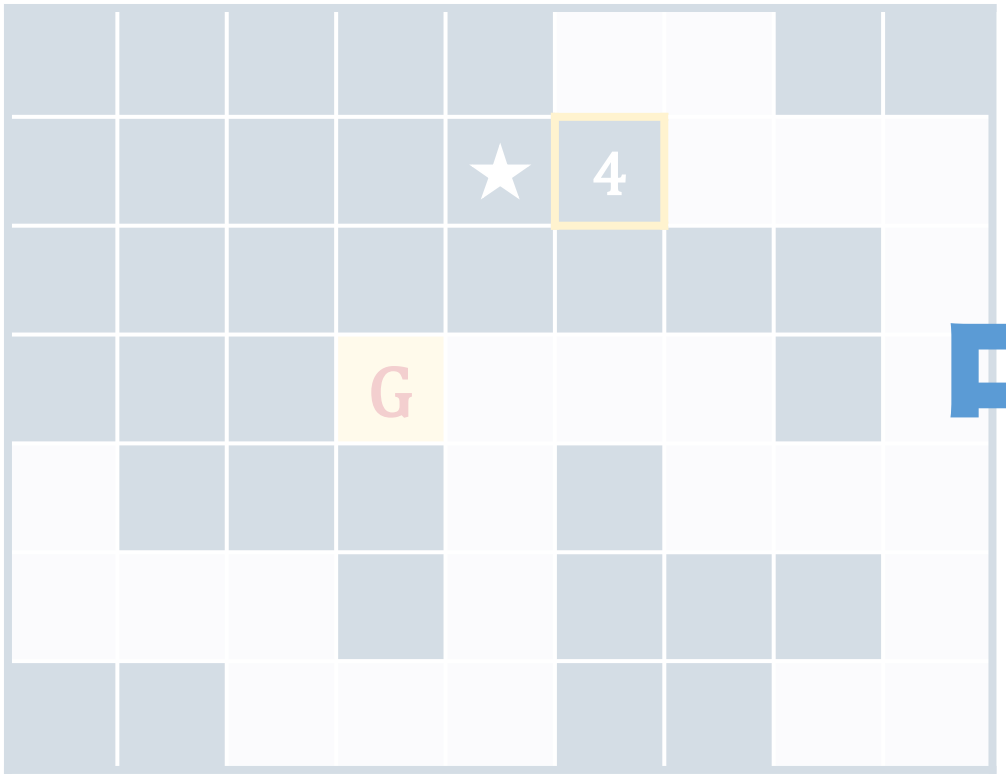




待機列 (登録順)

(4, 1)	3
(5, 2)	3
(2, 6)	4

幅優先探索の動き



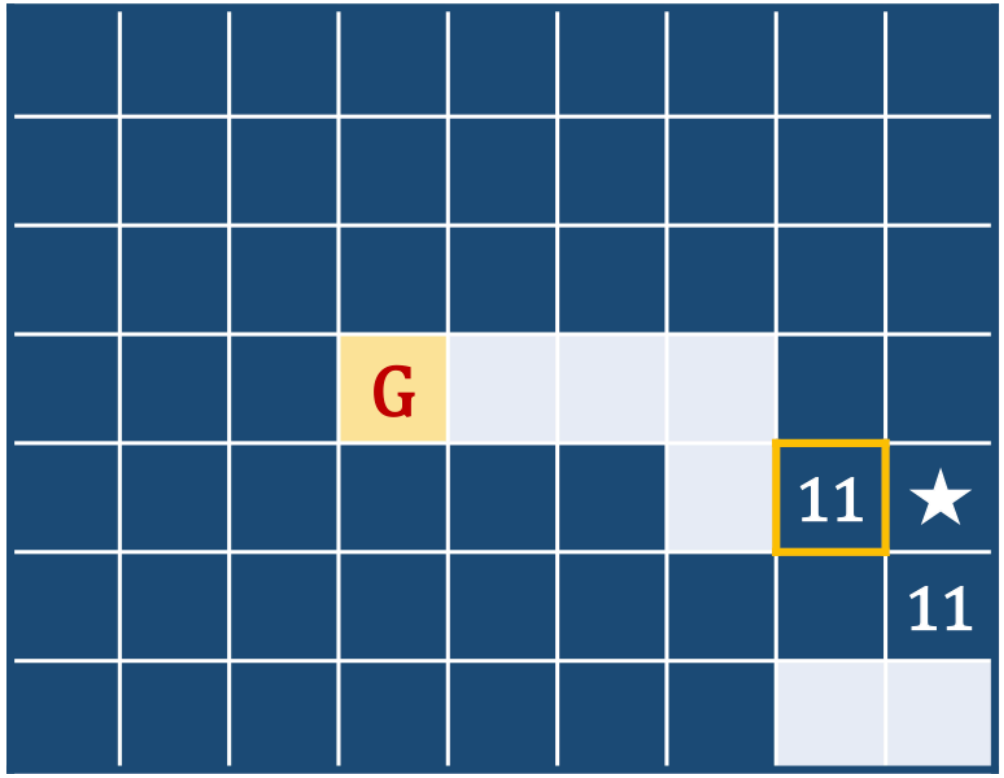
(2, 5)	3

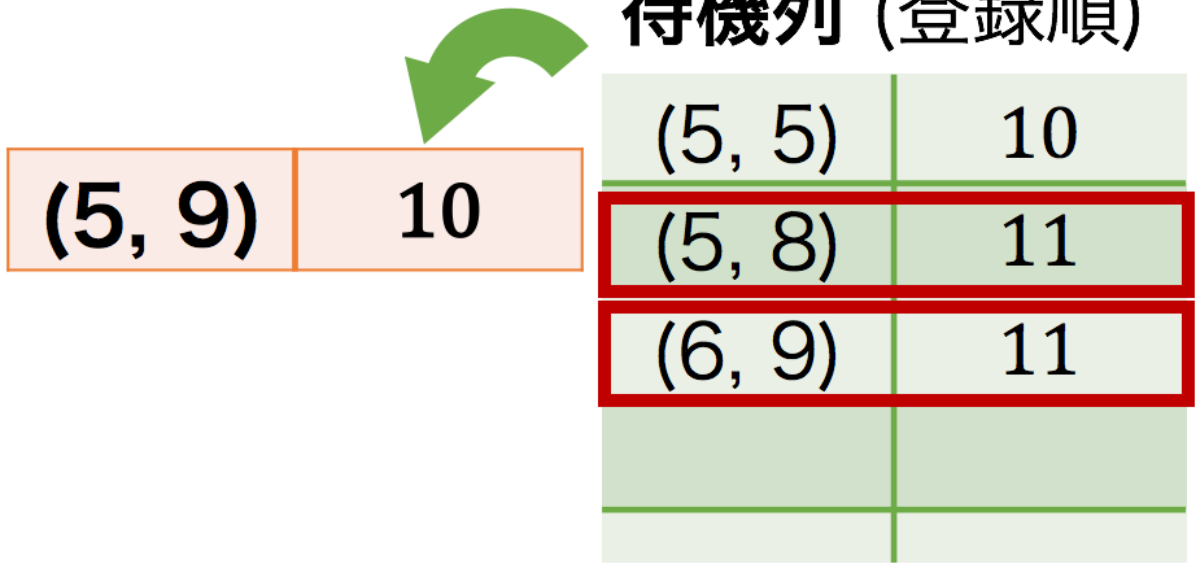
中略

待機列 (登録順)

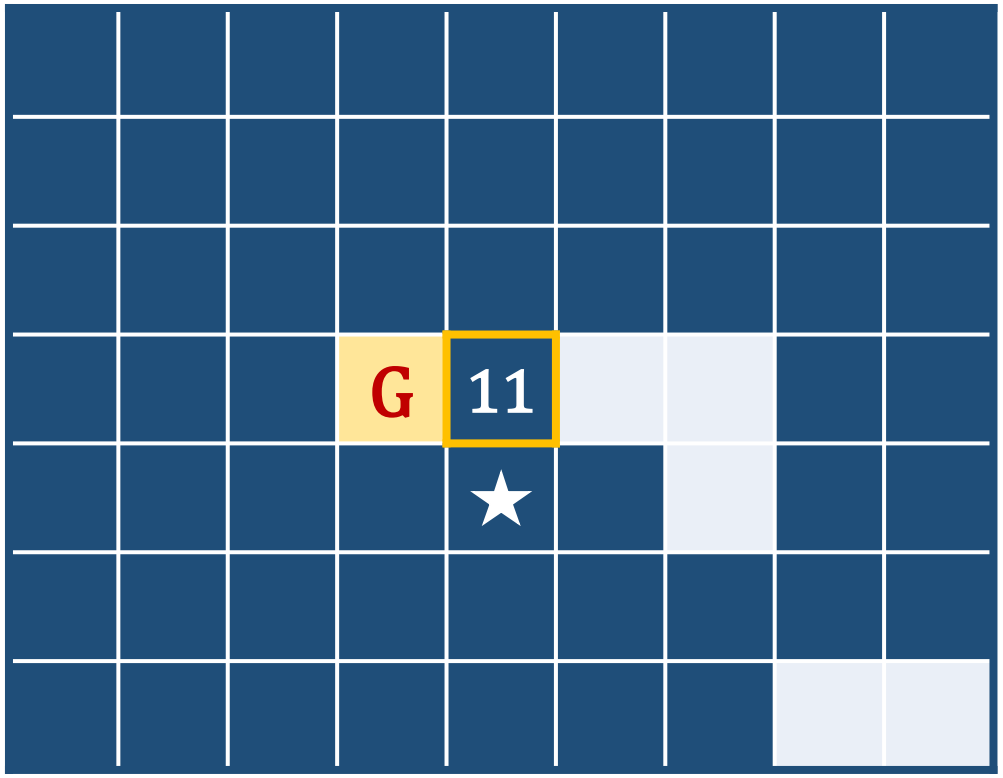
(4, 1)	3
(5, 2)	3
(2, 6)	4

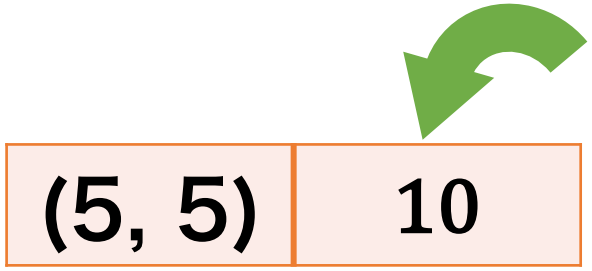
幅優先探索の動き





幅優先探索の動き

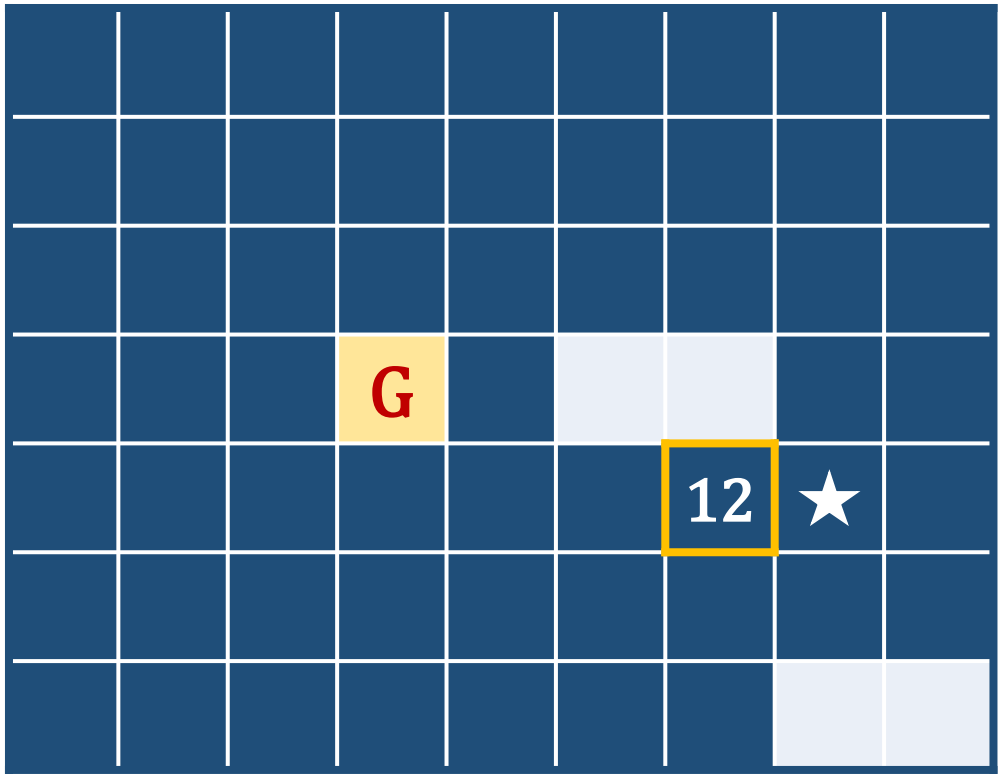


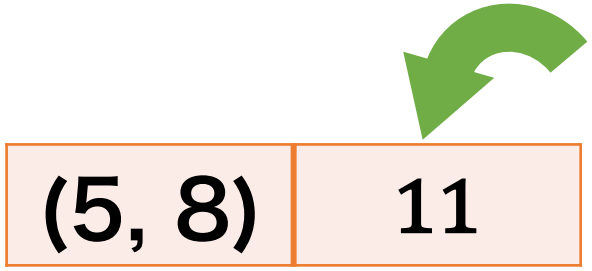


待機列 (登録順)

(5, 8)	11
(6, 9)	11
(4, 5)	11

幅優先探索の動き

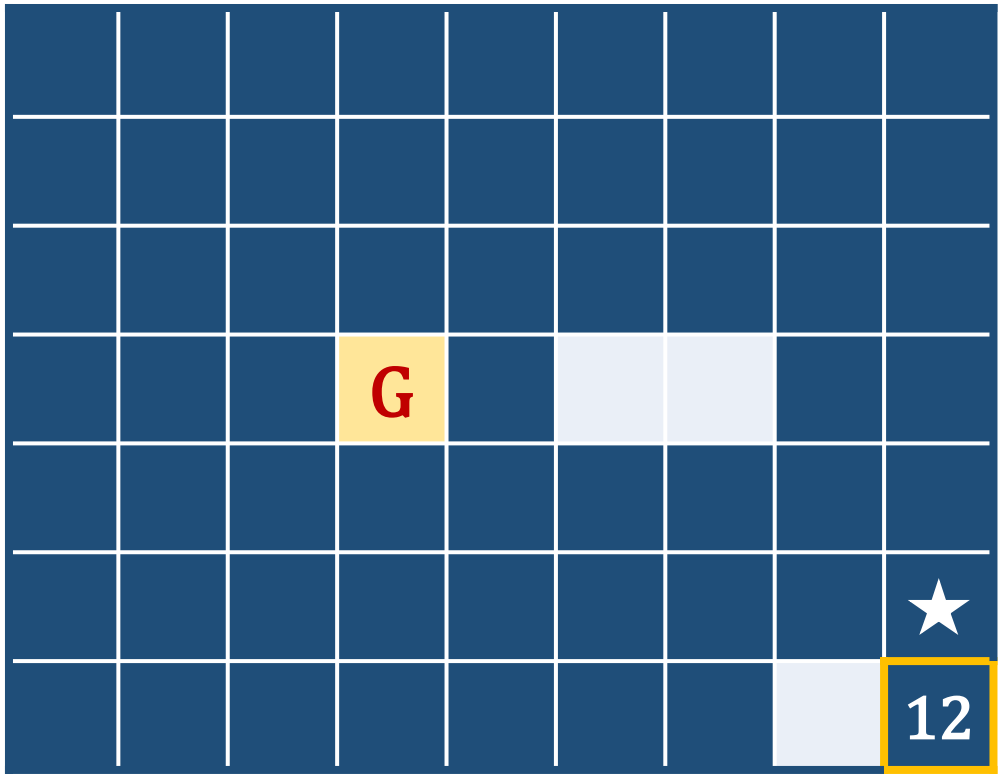




待機列 (登録順)

(6, 9)	11
(4, 5)	11
(5, 7)	12

幅優先探索の動き

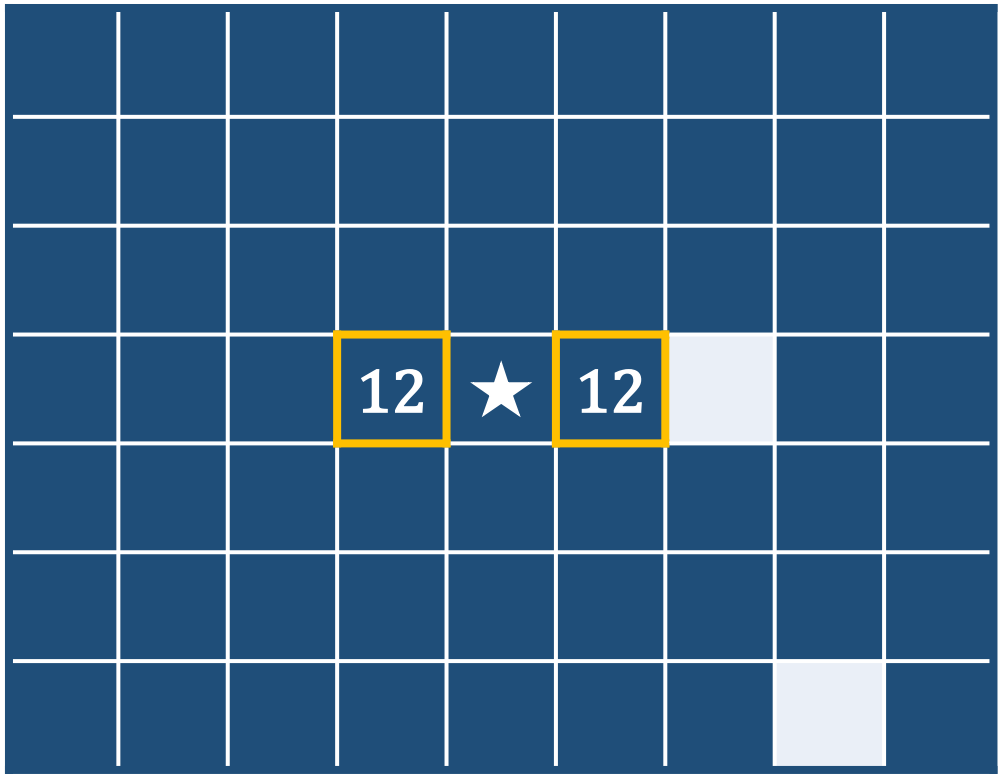


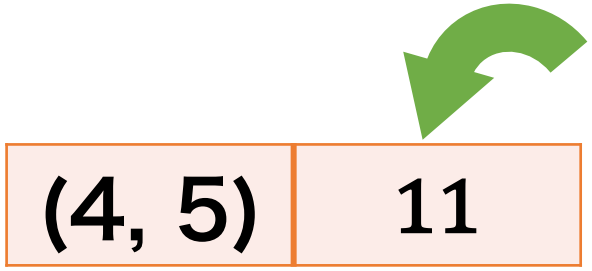
(6, 9)	11

待機列 (登録順)

(4, 5)	11
(5, 7)	12
(7, 9)	12

幅優先探索の動き

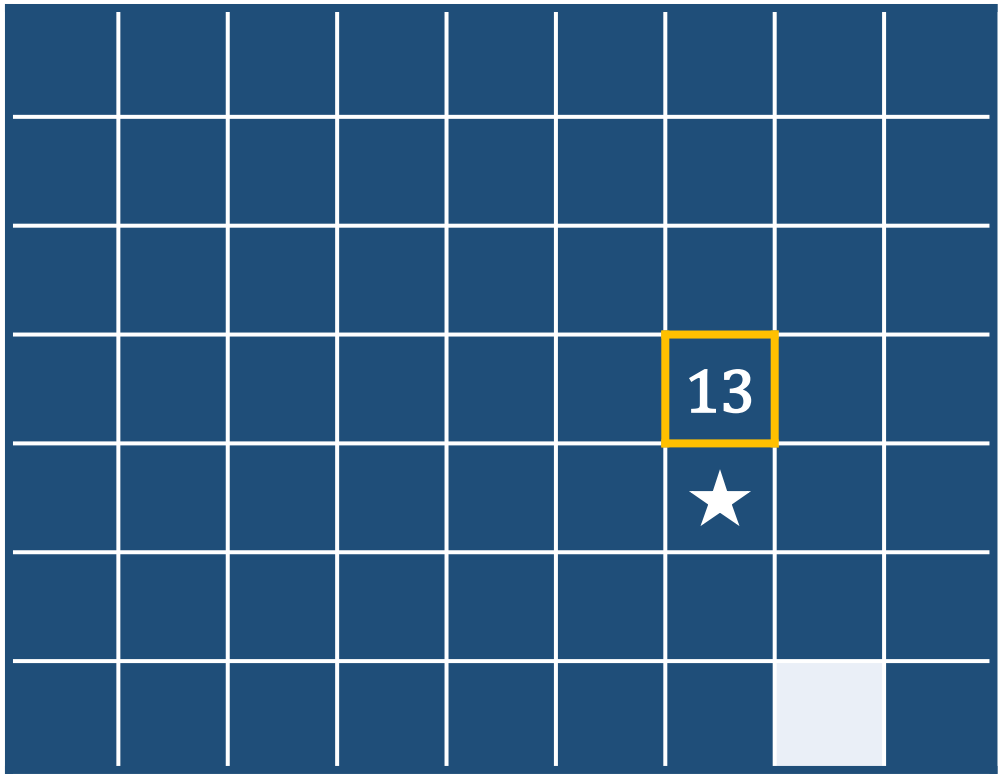




待機列 (登録順)

(5, 7)	12
(7, 9)	12
G	12
(4, 6)	12

幅優先探索の動き

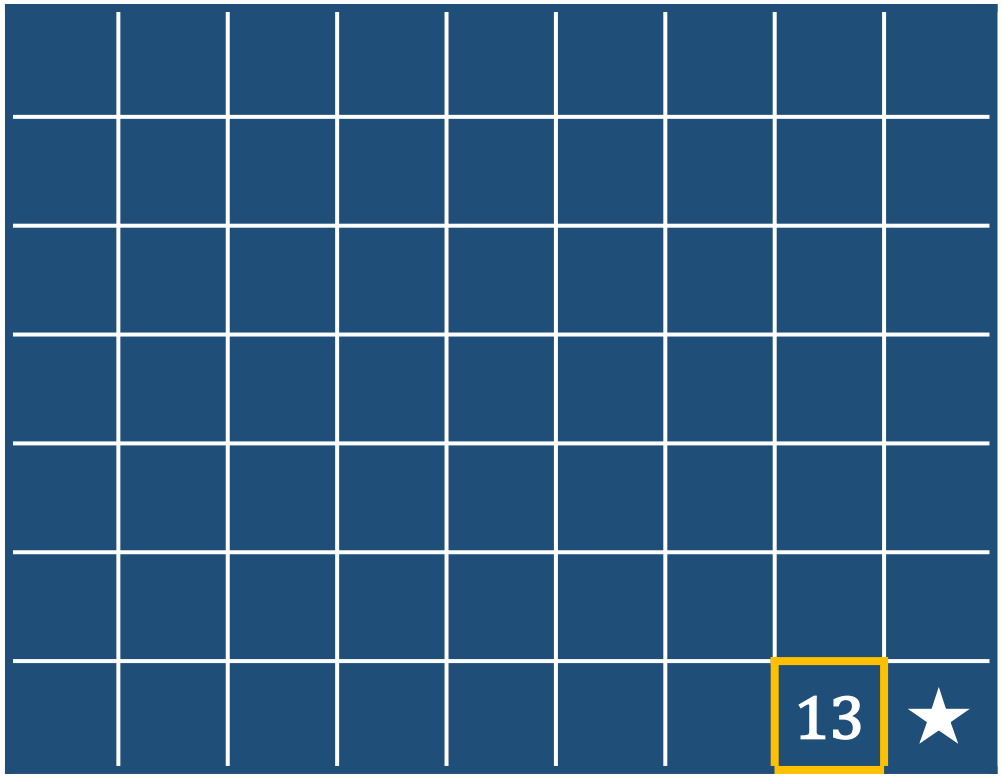


(5, 7)	12

待機列 (登録順)

(7, 9)	12
G	12
(4, 6)	12
(4, 7)	13

幅優先探索の動き

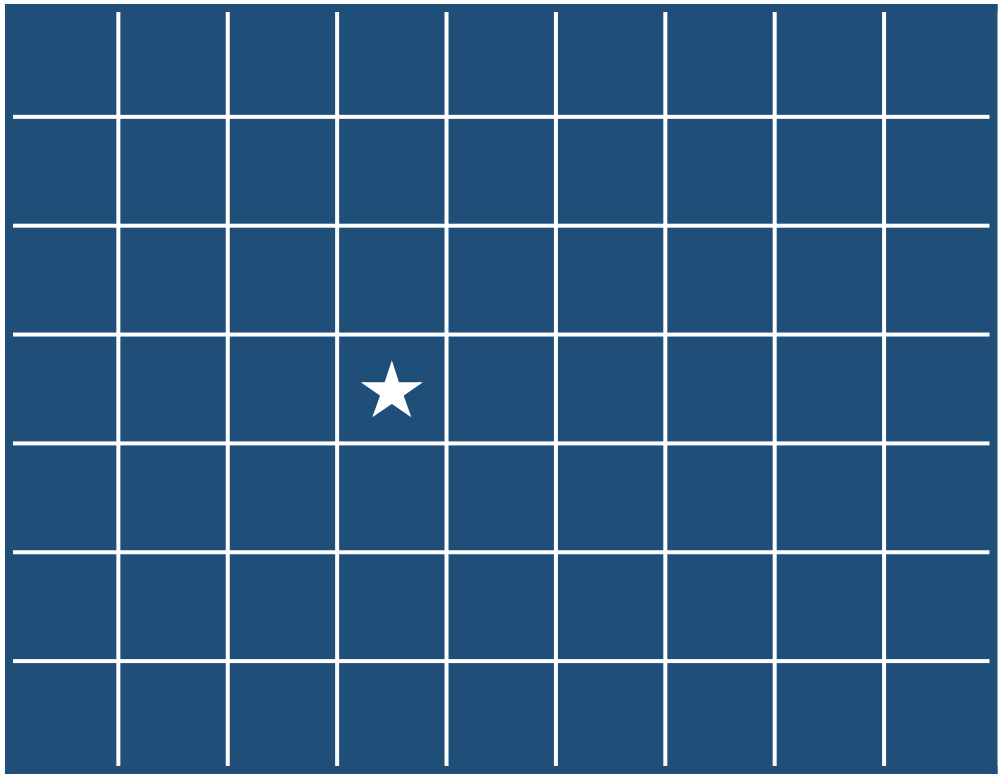


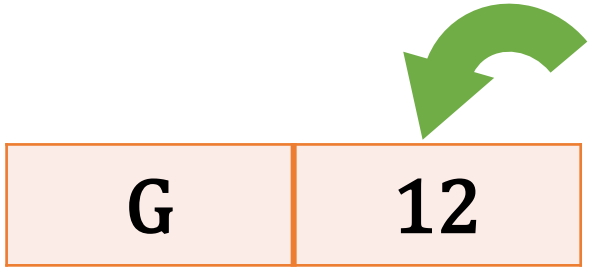
(7, 9)	12

待機列 (登録順)

G	12
(4, 6)	12
(4, 7)	13
(7, 8)	13

幅優先探索の動き





待機列 (登録順)

(4, 6)	12
(4, 7)	13
(7, 8)	13

幅優先探索の動き

G	12

待機列 (登録順)

(4, 6)	12
(4, 7)	13
(7, 8)	13

Gが出てきたら or
待機列が空になったら終了！

幅優先で使う武器: キュー (Queue)

- 待機列のような「登録した順に中身が出てくる構造」は、一般に**キュー**と呼ばれている
  - 先に入れたものが先に出てくるという動作原理を、英語でかっこよく First In, First Out という

キューの実装方法

Q. どうやって実装するの？

キューの実装方法

Q. どうやって実装するの？　A. もうあるので、それを使おう！

	C++での書き方	計算量
ライブラリ	#include <queue>	
宣言	queue<型> q;	
要素の挿入	q.push(x);	O(1) 時間
先頭を見る	q.front();	O(1) 時間
先頭を削除	q.pop();	O(1) 時間
空か確認	q.empty();	O(1) 時間

幅優先探索の解析

幅優先探索って、どれくらいの計算時間がかかるの？

- 1つの地点ごとにやっていること:
  隣接地点を1つひとつ見て、
  移動可能であるかの確認と、
  移動可能であればキューへの登録＆埋め立てをやっている
  → O(1) **時間**

- **これを各地点について高々1回やっているため、**

全体として O(RC) 時間　(R: 盤面の縦の長さ, C: 盤面の横の長さ)

実装例

```cpp
#include <iostream>
#include <utility>
#include <vector>
#include <queue>
using namespace std;

int main(void){
  // 入力を受け取る
  int r, c, sY, sX, gY, gX;
  cin >> r >> c;
  cin >> sY >> sX;
  cin >> gY >> gX;

  // canGo[i][j]: (i,j)が壁ならばfalse, 通路ならばtrue
  vector<vector<bool> > canGo(r + 2, vector<bool>(c + 2, false));
  string input;
  for(int i = 1; i <= r; i++){
    cin >> input;
    for(int j = 1; j <= c; j++){
      if(input[j - 1] == '.'){
        canGo[i][j] = true;
      }
    }
  }

  // まず最初に、Sの情報をキューに追加して、盤面を埋める
  queue<pair<int, pair<int, int> > > wait;
  wait.push(make_pair(0, make_pair(sY, sX)));
  canGo[sY][sX] = false;

  // 幅優先探索スタート
  int ans = -1;
  while(!wait.empty()){
    // キューの先頭要素を取り出す
    int nowcost = wait.front().first;
    int nowY = wait.front().second.first;
    int nowX = wait.front().second.second;
    wait.pop();

    // もしゴールに到達したら、そのコストが答え
    if(gY == nowY && gX == nowX){
      ans = nowcost;
      break;
    }

    // 今自分がいる地点からの4方向について
    int needleY[4] = {0, -1, 0, 1};
    int needleX[4] = {-1, 0, 1, 0};
    for(int i = 0; i < 4; i++){
      int nextY = nowY + needleY[i];
      int nextX = nowX + needleX[i];
      // その方向に進めるならば
      if(canGo[nextY][nextX]){
        // そのマスをキューに追加し、
        // 再度探索することのないように壁で埋める
        wait.push(make_pair(nowcost + 1,
                            make_pair(nextY, nextX)));
        canGo[nextY][nextX] = false;
      }
    }
  }

  // 出力
  cout << ans << endl;
  return 0;
}
```

目次

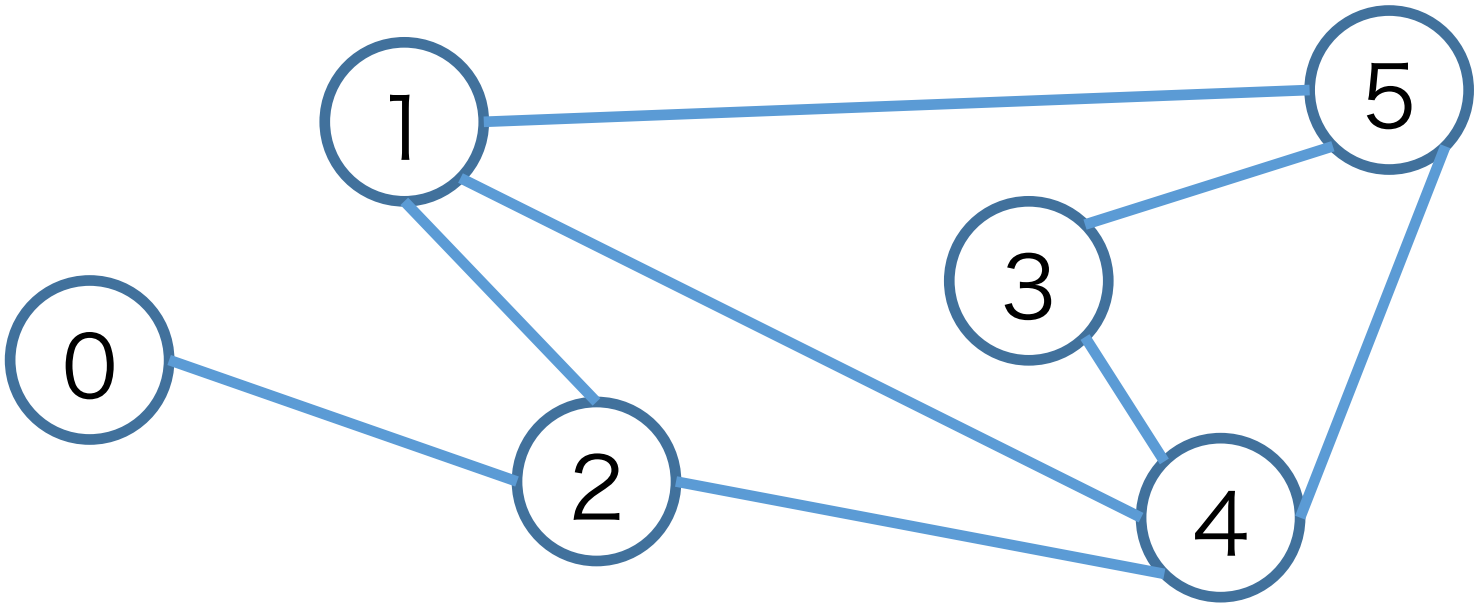
- 最短経路問題 その1
  - 幅優先探索

- **最短経路問題 その2**
  - **準備: グラフとは**
  - **ベルマンフォード法**
  - **ダイクストラ法**
  - **ワーシャルフロイド法**

- 最小全域木
  - 準備: 木とは
  - プリム法
  - クラスカル法

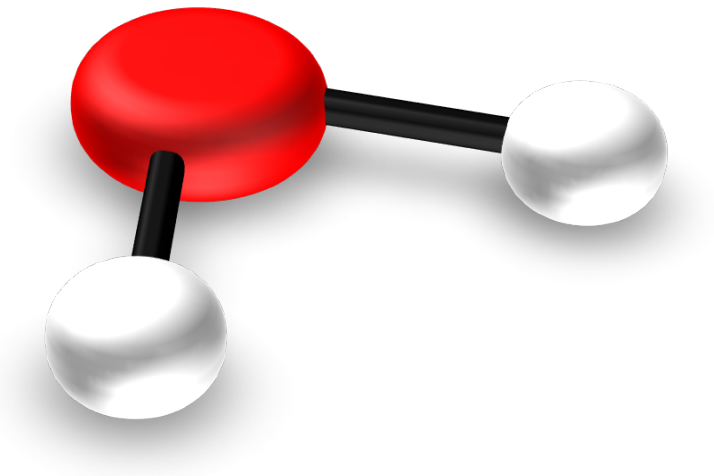
グラフとは？

- **物事とその関係を、点と線で表したもの！**
  - 点のことを**「頂点」**とか**「ノード」**とか呼ぶ
  - 線のことを**「辺」**とか**「枝」**とか呼ぶ



日常にみるグラフの例

- **鉄道網**
  頂点: 駅　　辺: 路線
- **人間関係**
  頂点: 人間　　辺: 思惑・利害・血縁など
- **分子**
  頂点: 原子　　辺: 結合
- **宇宙**
  頂点: 星　　辺: 重力
- **人生**
  頂点: できごと　　辺: 人類の可能性



日常にみるグラフの例

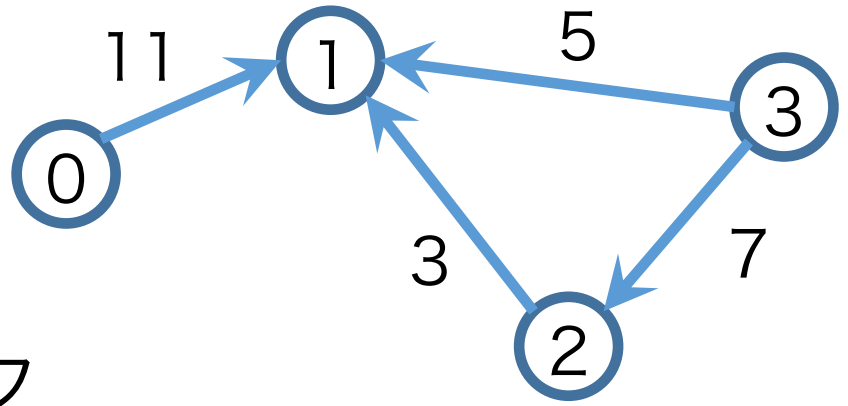
- 鉄道網
- 人間
- 分子
- 宇宙
- 人生

頂点: できごと　　辺: 人類の可能性

この世のすべては
グラフである

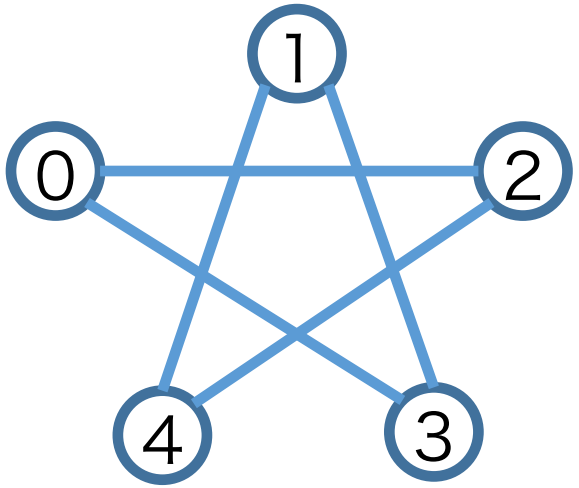
グラフの用語いろいろ

- **有向グラフ:** 辺に向きがあるグラフ
  無向グラフ: 辺に向きがないグラフ



- **重みつきグラフ:** 辺にコストがあるグラフ

- **閉路:** 同じ辺を二度通らずに
  元の頂点に戻ってこられる路



- **次数:** その頂点に繋がっている枝の本数

※ 以降、頂点の数をV、枝の本数をEで表すことにする

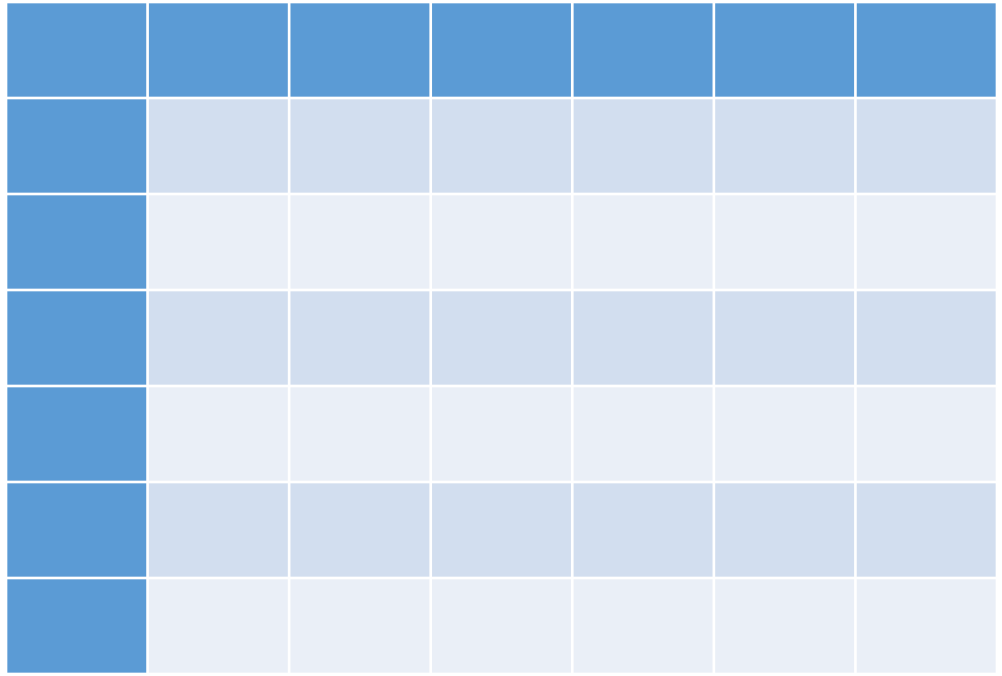
グラフの表現方法

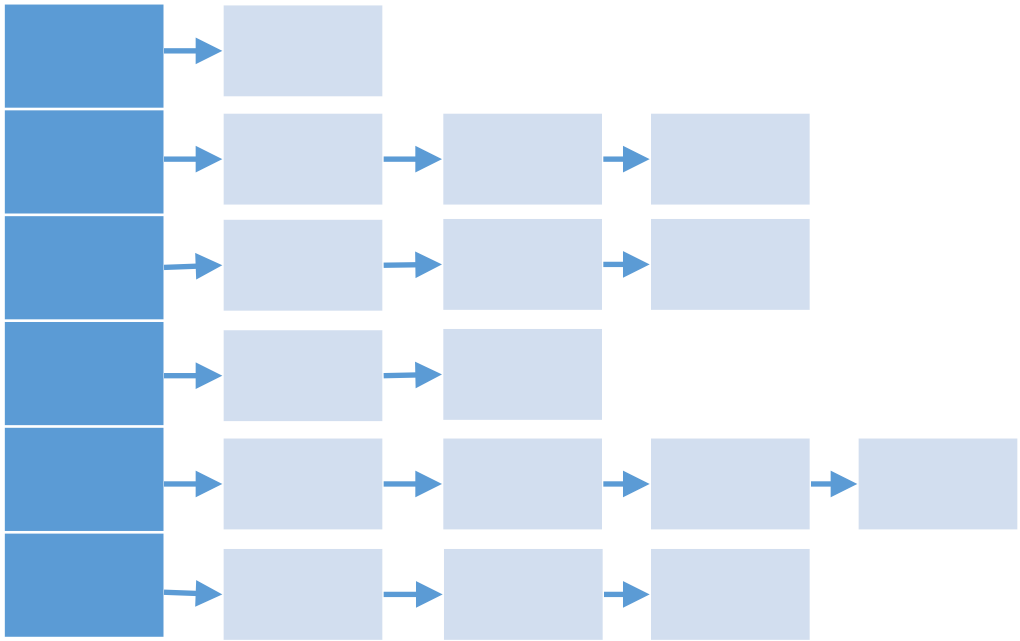
Q. プログラム上で、グラフはどう表現するの？

グラフの表現方法

Q. プログラム上で、グラフはどう表現するの？

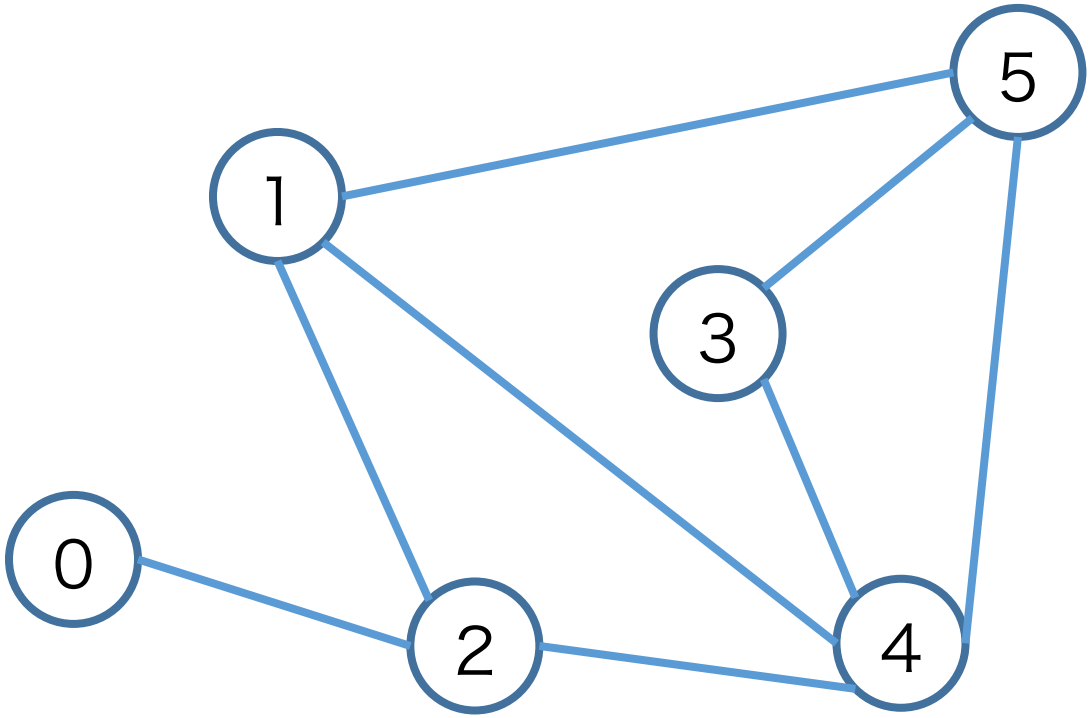
A. 隣接行列か隣接リストを使うよ！





隣接行列

- 二次元配列を使って、グラフの隣接関係を表す

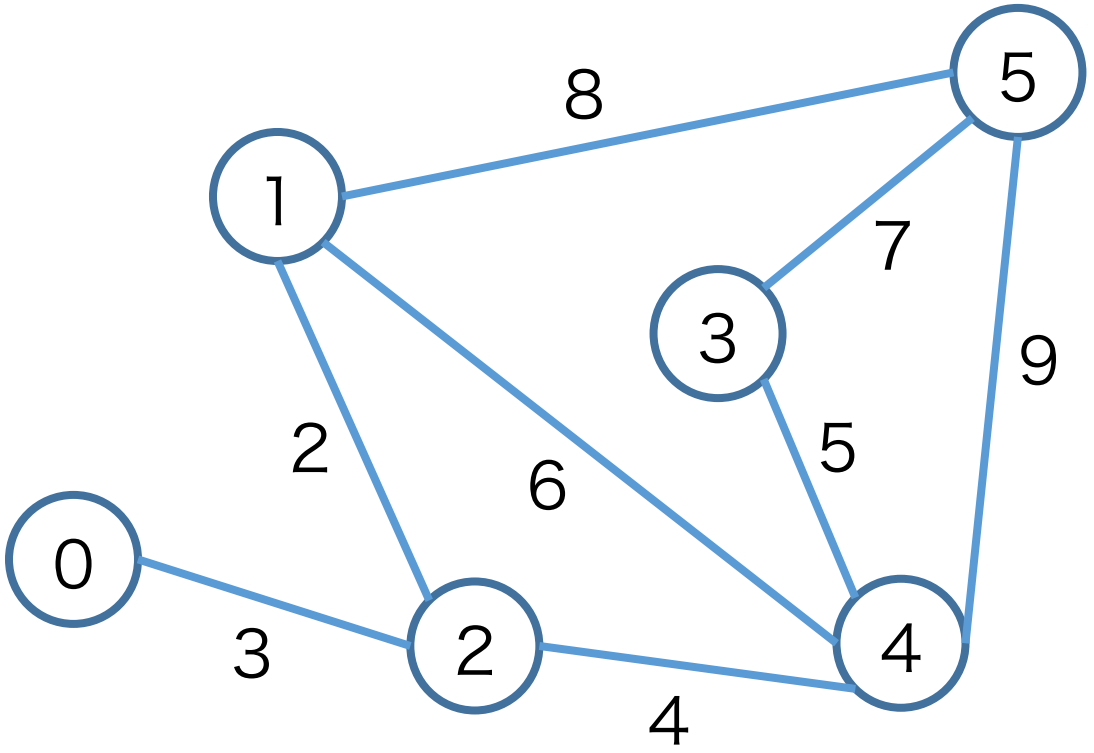


重みなしの場合: 隣接しているかどうか

	0	1	2	3	4	5
0	0	0	1	0	0	0
1	0	0	1	0	1	1
2	1	1	0	0	1	0
3	0	0	0	0	1	1
4	0	1	1	1	0	1
5	0	1	0	1	1	0

隣接行列

- 二次元配列を使って、グラフの隣接関係を表す



重みつきの場合: 隣までの辺のコスト

	0	1	2	3	4	5
0	INF	INF	3	INF	INF	INF
1	INF	INF	2	INF	6	8
2	3	2	INF	INF	4	INF
3	INF	INF	INF	INF	5	7
4	INF	6	4	5	INF	9
5	INF	8	INF	7	9	INF

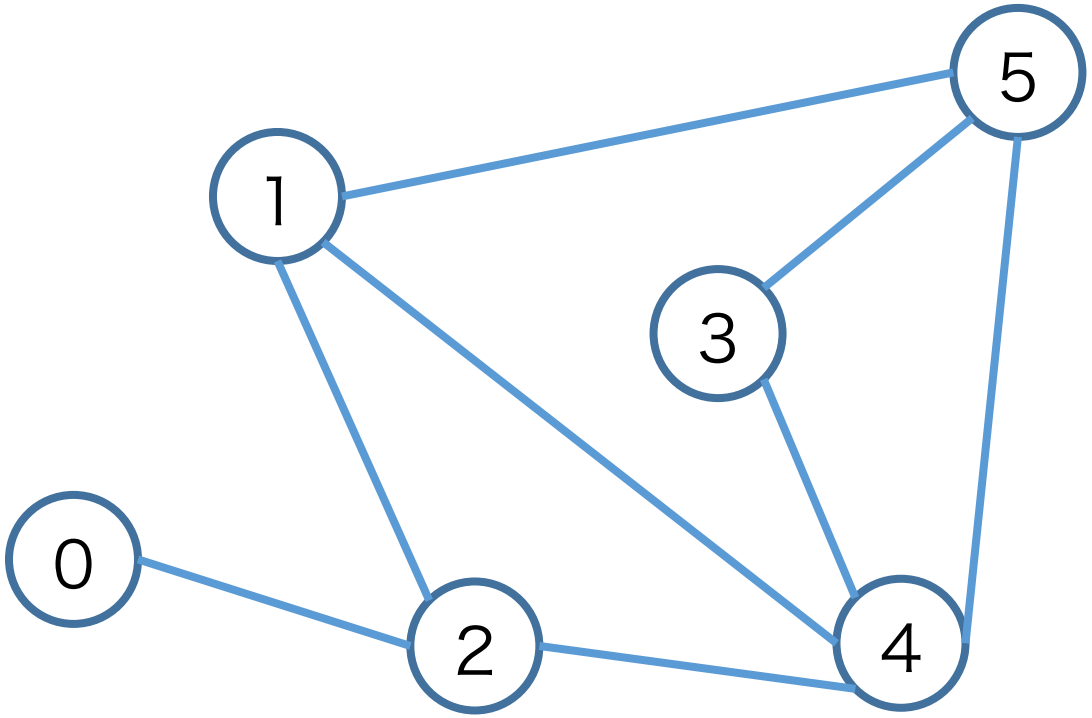
INF: 無限大 (適当にでかい数をぶち込む)

隣接行列の作り方 (C)

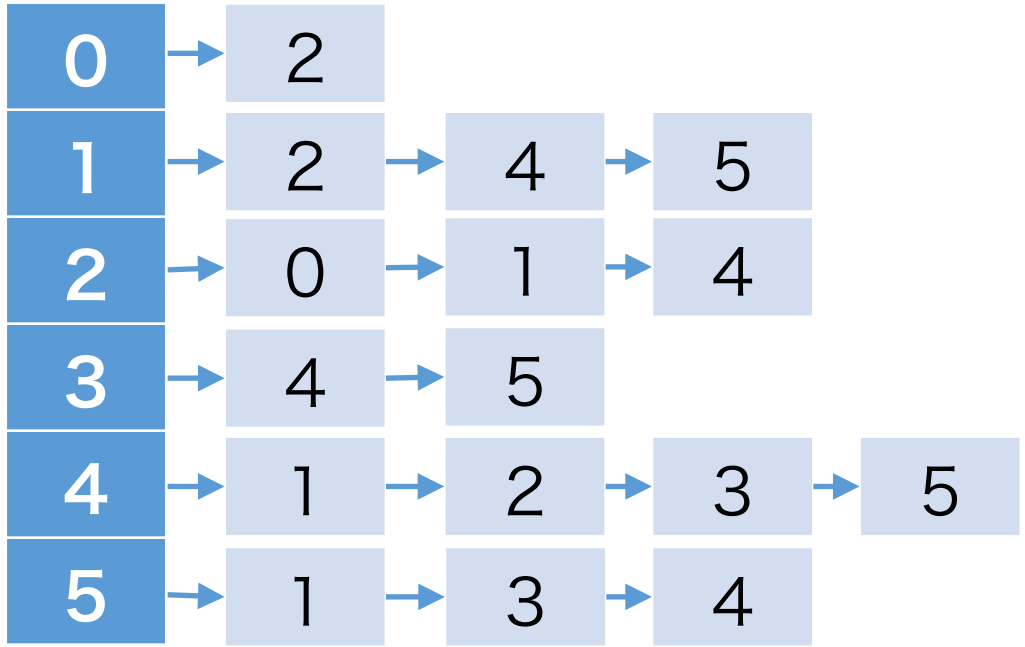
```c
// 二次元配列を宣言
int adjmatrix[V][V];

// とりあえずすべて無限大で初期化
for(i = 0; i < V; i++)
  for(j = 0; j < V; j++)
    adjmatrix[i][j] = INF;

// 入力「aからbにコストcの辺があるよ」
for(i = 0; i < E; i++){
  scanf("%d %d %d", &a, &b, &c);
  adjmatrix[a][b] = c;
}
```

隣接リスト

- リストを使って、グラフの隣接関係を表す

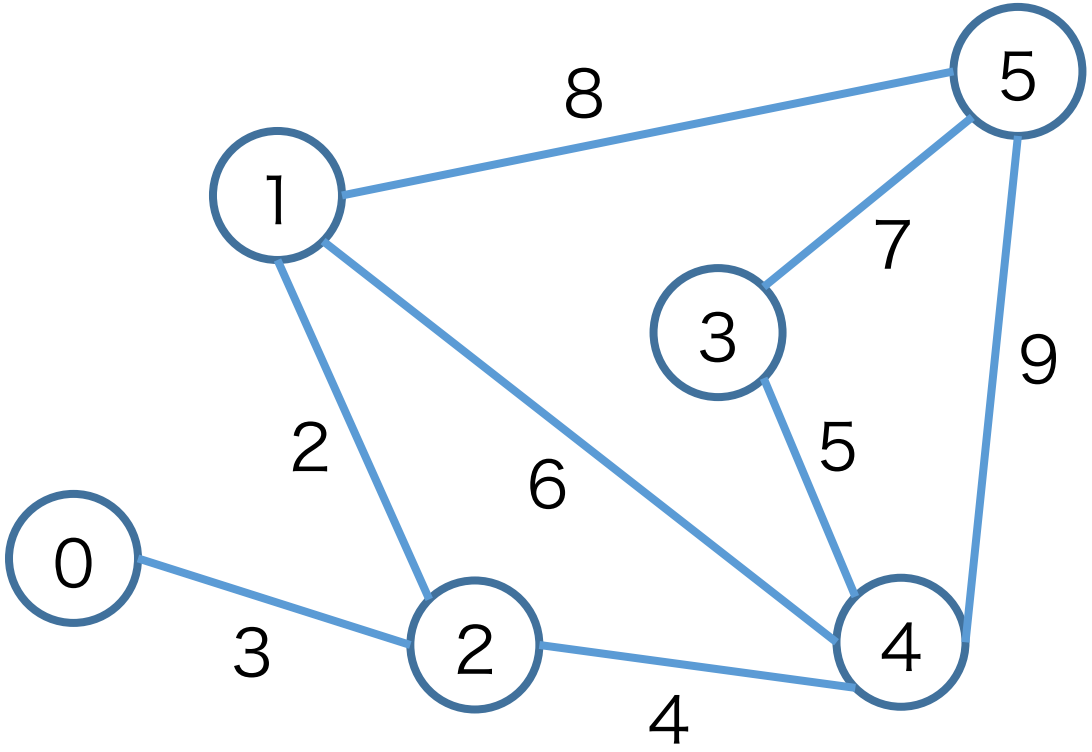


重みなしの場合: 隣接している頂点の番号



隣接リスト

- リストを使って、グラフの隣接関係を表す



重みつきの場合: (コスト, 隣の番号)のペア

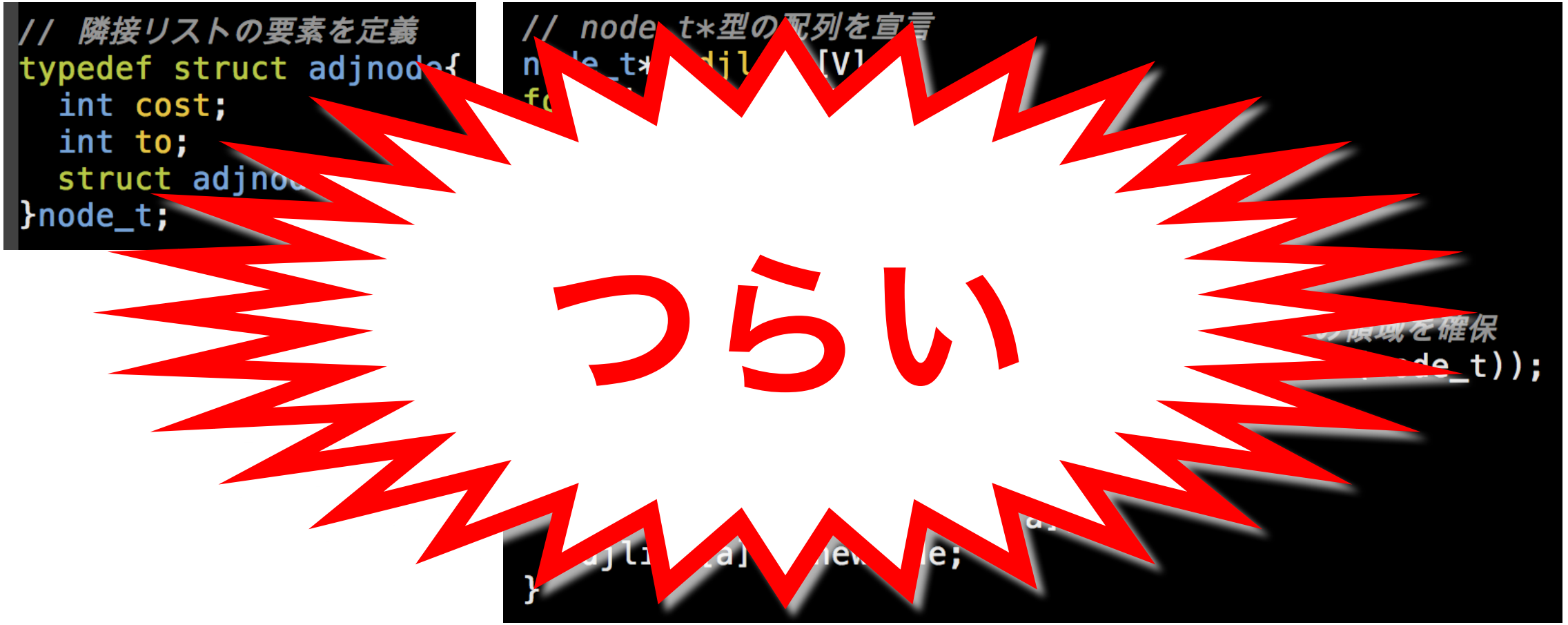
0	3,2			
1	2,2	6,4	8,5	
2	3,0	2,1	4,4	
3	5,4	7,5		
4	6,1	4,2	5,3	9,5
5	8,1	7,3	9,4	

隣接リストの作り方 (C)

```c
// 隣接リストの要素を定義
typedef struct adjnode{
  int cost;
  int to;
  struct adjnode* next;
}node_t;
```

```c
// node_t*型の配列を宣言
node_t* adjlist[V];
for (i = 0; i < V; i++)
  adjlist[i] = NULL;

// 入力「aからbにコストcの辺があるよ」
for (i = 0; i < E; i++) {
  scanf("%d %d %d", &a, &b, &c);
  // リストに新しい要素を追加するため、node_t*型の領域を確保
  node_t* newnode = (node_t *)malloc(sizeof(node_t));
  // 入力データを格納し、リストの先頭に挿入
  newnode->cost = c;
  newnode->to = b;
  newnode->next = adjlist[a];
  adjlist[a] = newnode;
}
```

隣接リストの作り方 (C)


// 隣接リストの要素を定義
typedef struct adjnode{
int cost;
int to;
}node_t;
// node_t*型の配列を宣言
[V]
域を確保
de_t));
e;
}
つらい

隣接リストの作り方 (C++)

```cpp
// 二次元vectorを宣言(vector型の配列でも可)
vector<vector<pair<int, int> > > adjlist(V);

// 入力「aからbにコストcの辺があるよ」
for (int i = 0; i < E; i++) {
  cin >> a >> b >> c;
  pair<int, int> p(c, b);
  adjlist[a].push_back(p);
}
```

C++を使おう

隣接行列 vs 隣接リスト

	隣接行列	隣接リスト
メモリ使用量	$O(V^2)$ 空間	$O(V + E)$ 空間
Q. この頂点とこの頂点って隣接してる？	$O(1)$ 時間	$O(\Delta)$ 時間
Q. この頂点に隣接してる頂点全部教えてよ	$O(V)$ 時間	$O(\Delta)$ 時間
実装難易度	簡単	隣接行列より面倒

(Δ: 最大次数)

隣接行列 vs 隣接リスト

	隣接行列	隣接リスト
メモリ使用量	$O(V^2)$ 空間	**$O(V+E)$ 空間**
Q. この頂点とこの頂点って隣接してる？	**$O(1)$ 時間**	$O(\Delta)$ 時間
Q. この頂点に隣接してる頂点全部教えてよ	$O(V)$ 時間	**$O(\Delta)$ 時間**
実装難易度	**簡単**	隣接行列より面倒

(Δ: 最大次数)

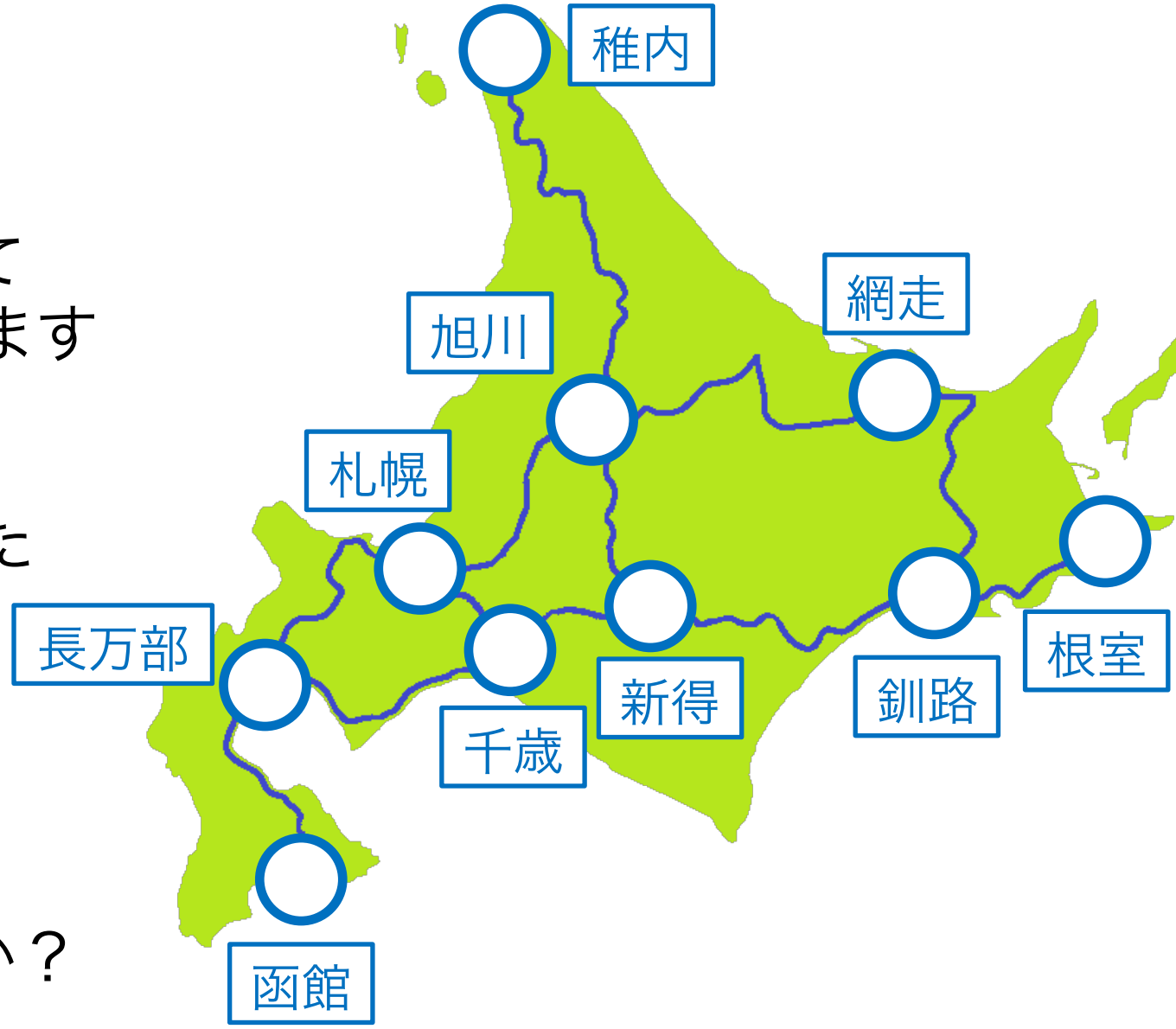
状況に応じて、使い分けていこう

例題

北海道に住む高橋くんは、
最寄りのｓ駅から列車に乗って
道内旅行をしたいと思っています

そこで、友達であるあなたは、
高橋くんからリクエストされた
ｇ駅までの所要時間を
教えてあげることにしました

なるべく早く到着するように
路線を選んだ場合、
ｇ駅まで何分かかるでしょうか？



例題

地点が v 個、路線が e 個与えられます

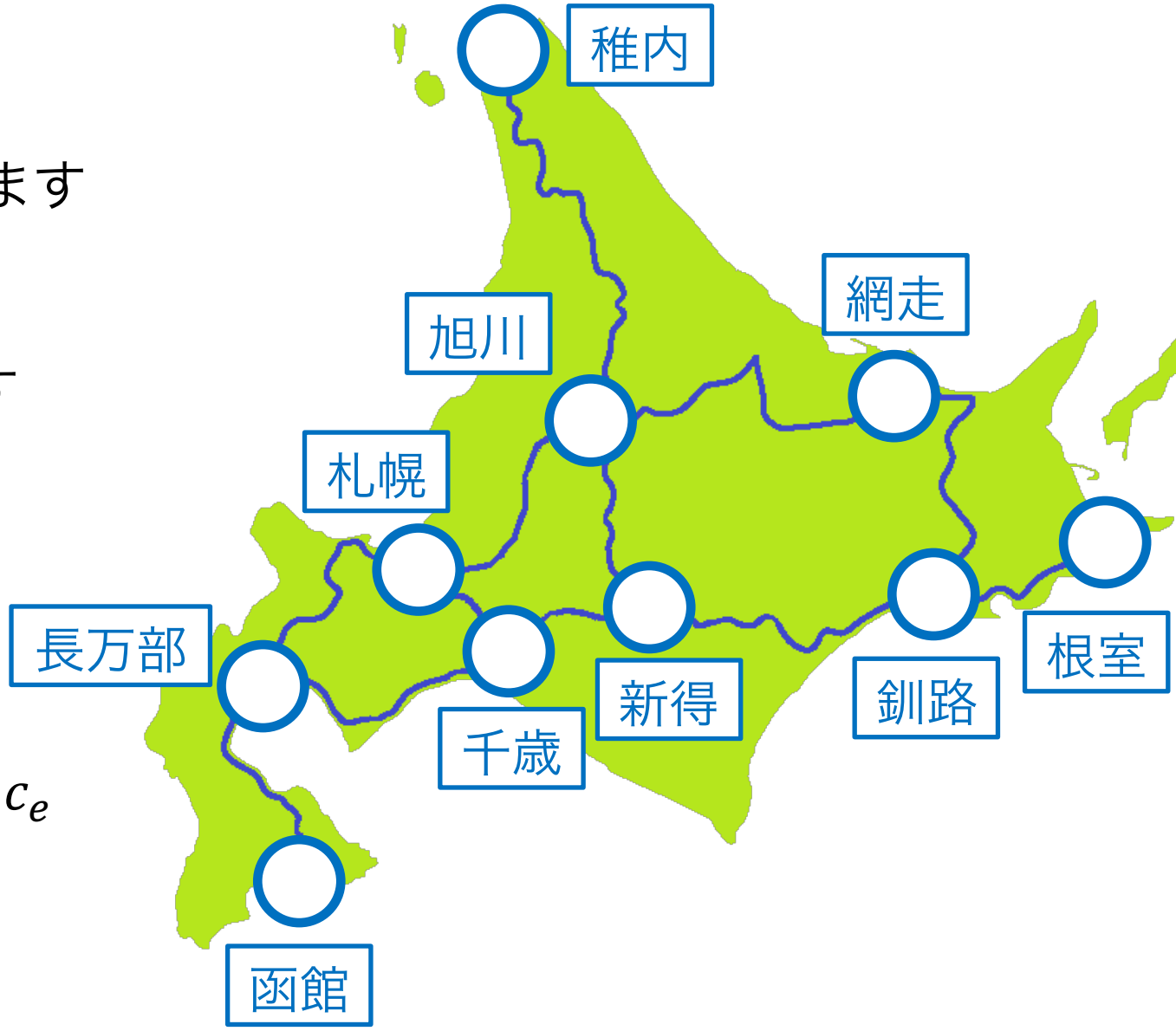
路線の情報は、
その路線が結ぶ2つの地点 a と b
及びその所要時間 c で与えられます

- **Input**

  1行目 v e s g
  2行目 第1の路線の情報 a_1 b_1 c_1
  ：
  e+1行目 第eの路線の情報 a_e b_e c_e

- **Constraints**

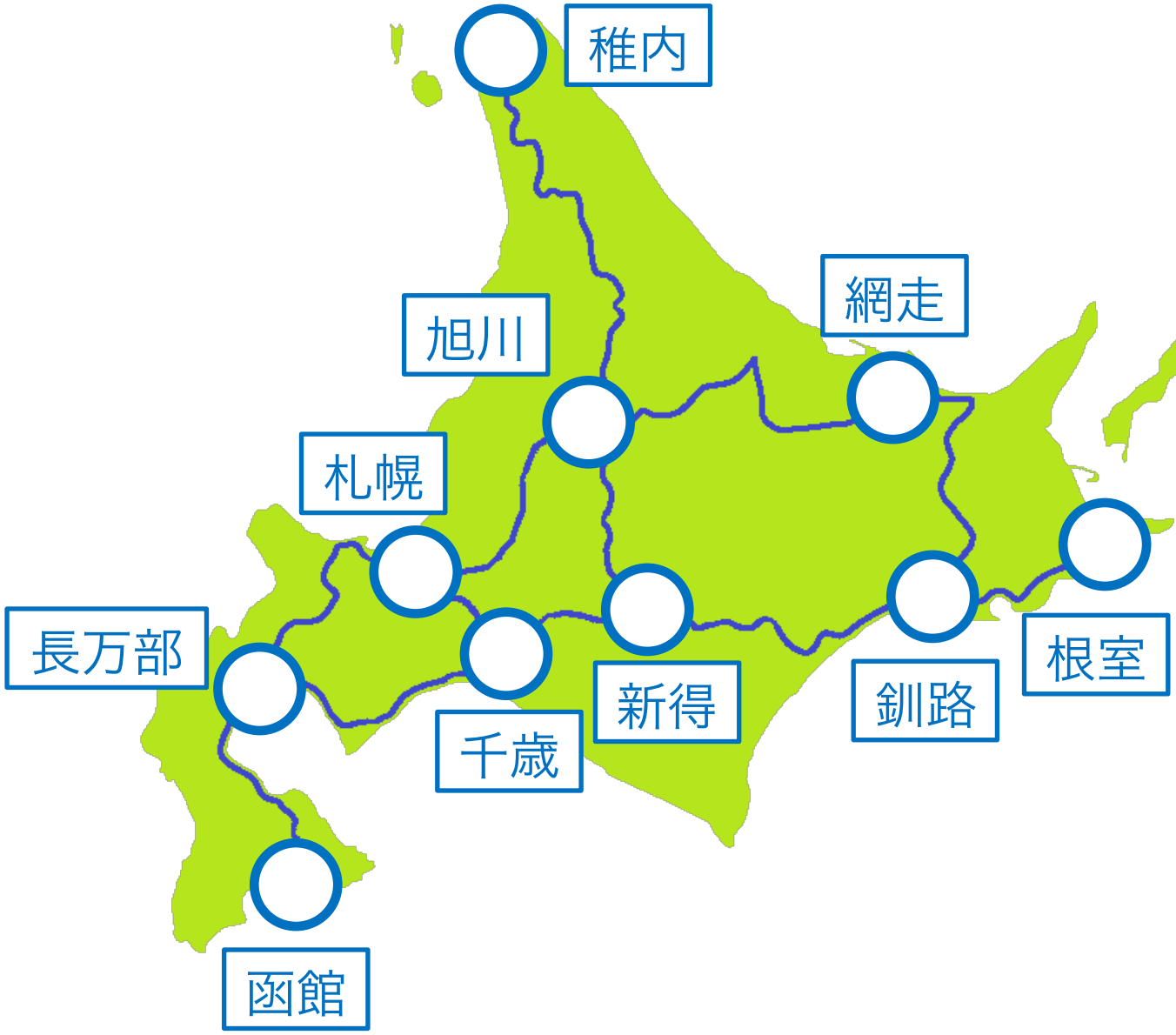
  $1 \leq v \leq 100$
  $1 \leq e, c \leq 10000$



例題

- **Sample Input**

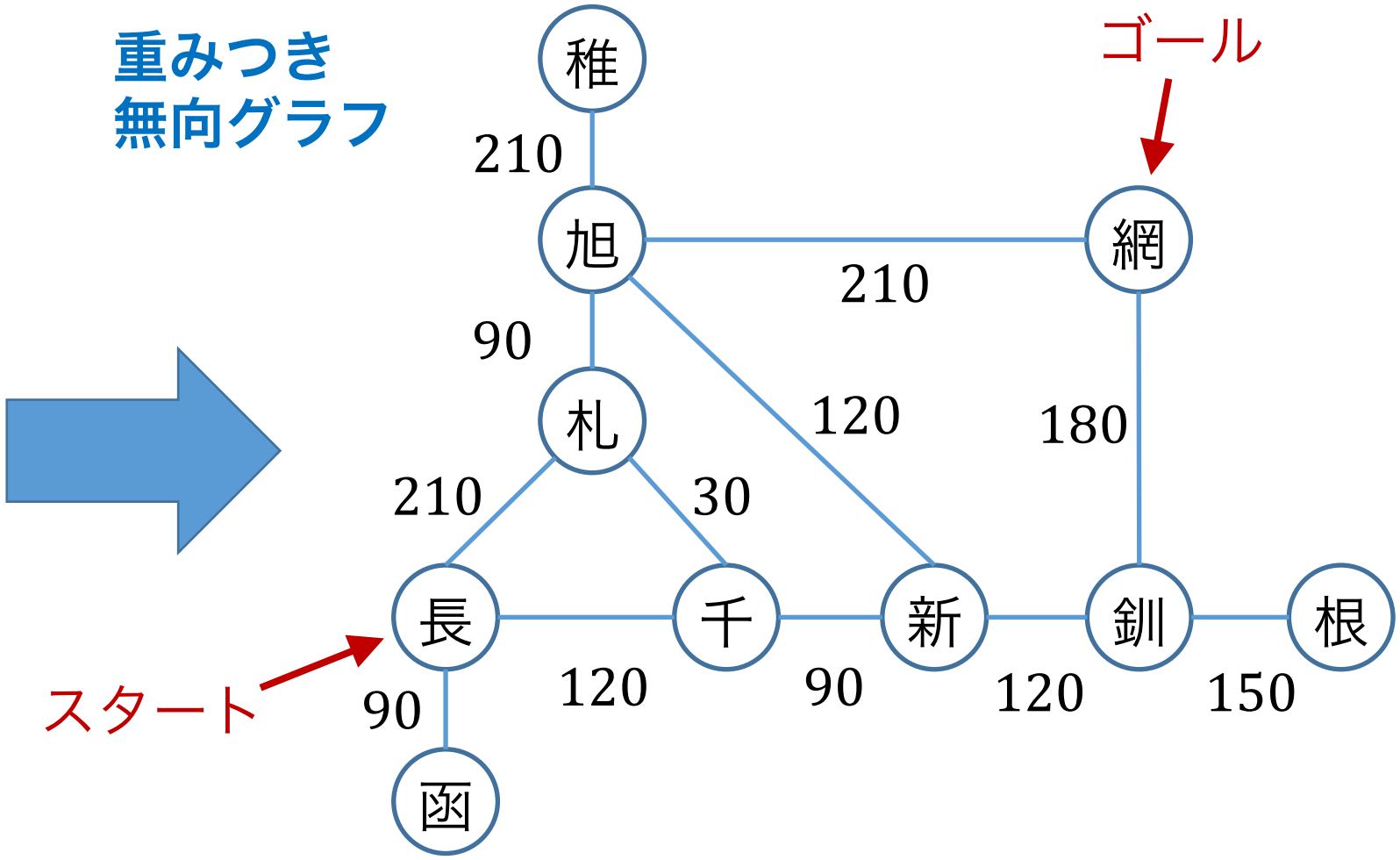
10　12　長万部　網走
長万部　函館　90
札幌　長万部　210
札幌　旭川　90
旭川　稚内　210
長万部　千歳　120
千歳　札幌　30
千歳　新得　90
新得　旭川　120
釧路　新得　120
釧路　根室　150
網走　旭川　210
網走　釧路　180



例題

- Sample Input

```
10 12 長万部 網走
長万部 函館 90
札幌 長万部 210
札幌 旭川 90
旭川 稚内 210
長万部 千歳 120
千歳 札幌 30
千歳 新得 90
新得 旭川 120
釧路 新得 120
釧路 根室 150
網走 旭川 210
網走 釧路 180
```



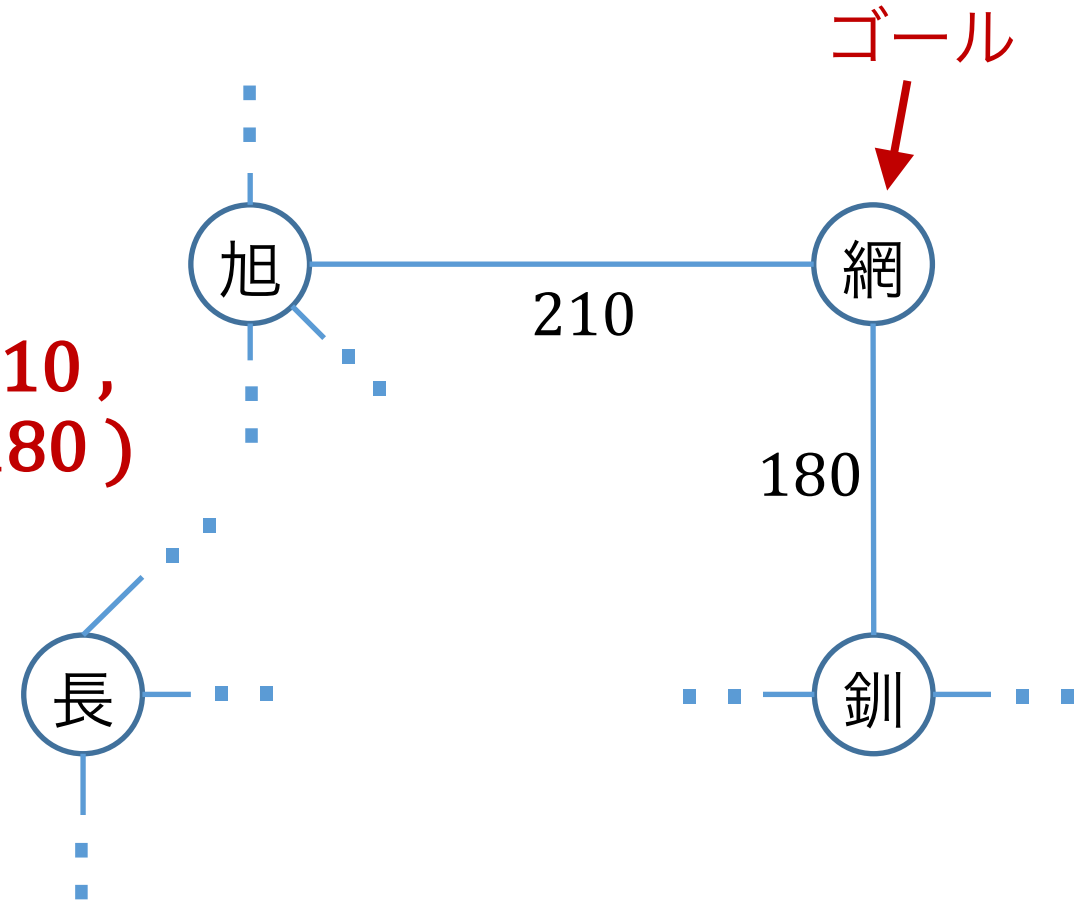
※ 実際の問題では、地点名は日本語ではなく
頂点番号 (整数) で与えられることがほとんどなので、
「文字列処理どうしよう」とかは考えなくていい

最短経路の性質

- グラフの途中がどうなっていようと

網走までの最小コスト
＝ min (旭川までの最小コスト ＋ 210 ,
釧路までの最小コスト ＋ 180)

- 網走に到達するためには
旭川か釧路を経由するしかないので
そこまでだって最小コストで行きたい

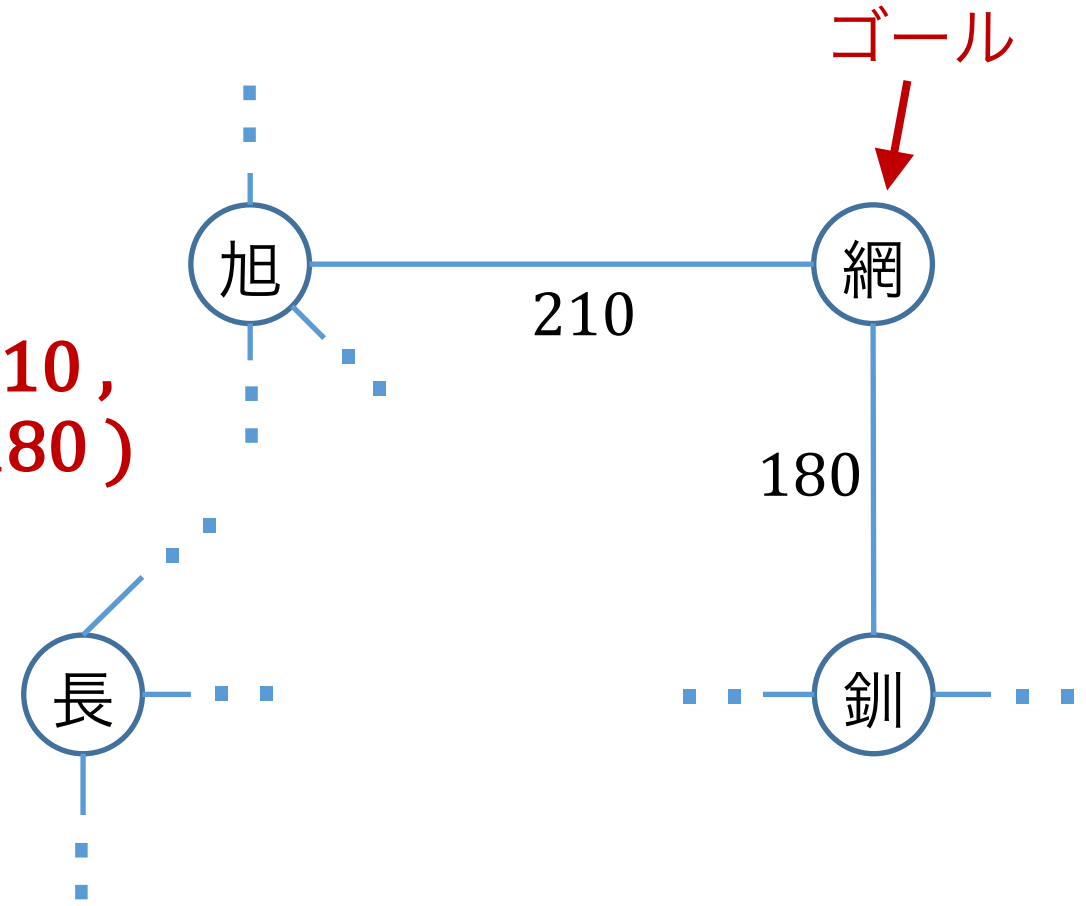


最短経路の性質

- グラフの途中がどうなっていようと

網走までの最小コスト
＝min (旭川までの最小コスト ＋210 ,
釧路までの最小コスト ＋180)

- 網走に到達するためには
旭川か釧路を経由するしかないので
そこまでだって最小コストで行きたい



- これは再帰的に成り立つため

頂点uまでの最小コスト
＝min (頂点uの隣までの最小コスト＋そこからuまでのコスト)

ベルマンフォード法 (Bellman-Ford algorithm)

- まず、すべての頂点からそれぞれ隣をみて、
  より小さい値で遷移できそうだったら、**コストを更新する**

ベルマンフォード法 (Bellman-Ford algorithm)

- まず、すべての頂点からそれぞれ隣をみて、
  より小さい値で遷移できそうだったら、**コストを更新する**

- 次に、すべての頂点からそれぞれ隣をみて、
  より小さい値で遷移できそうだったら、**コストを更新する！**

ベルマンフォード法 (Bellman-Ford algorithm)

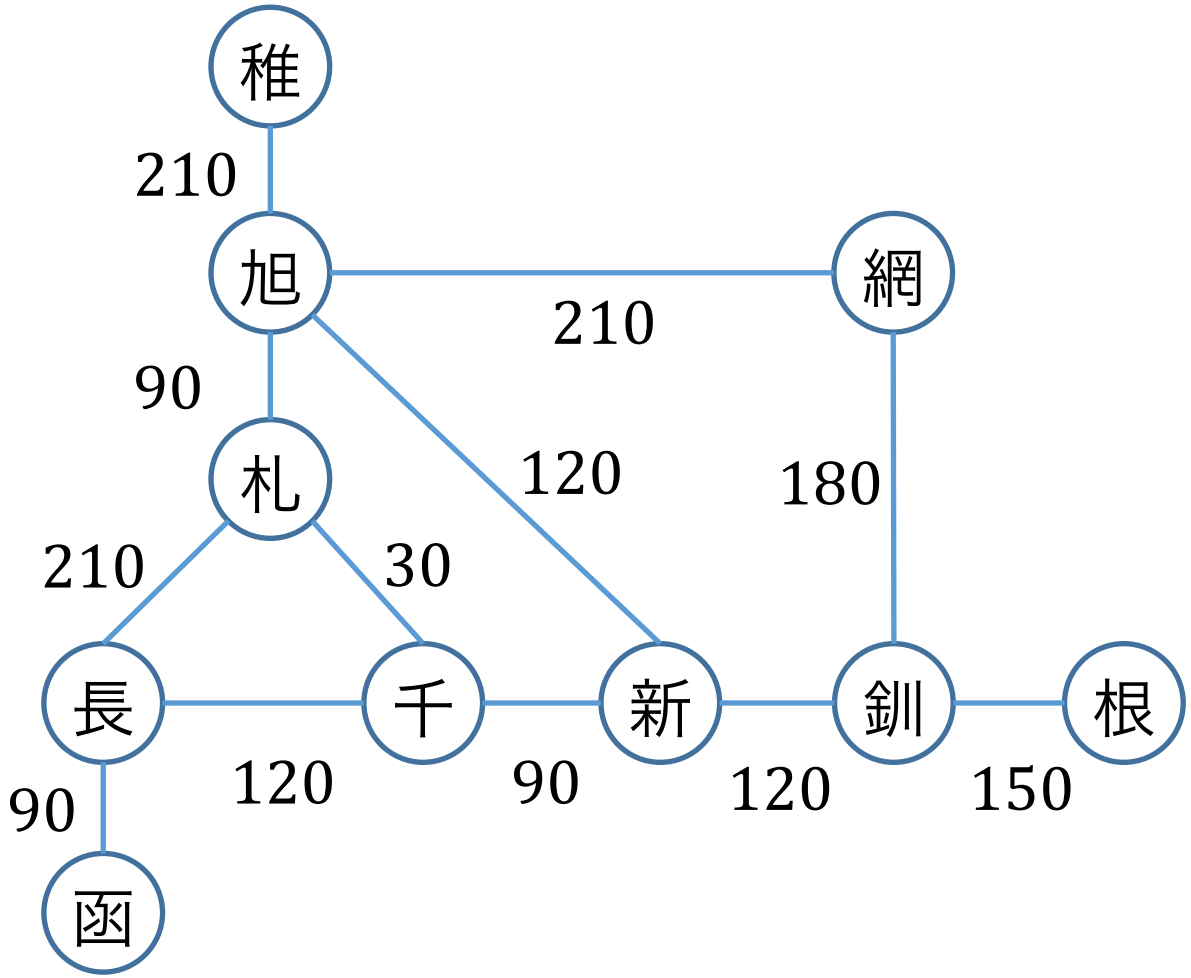
- まず、すべての頂点からそれぞれ隣をみて、
  より小さい値で遷移できそうだったら、**コストを更新する**

- 次に、すべての頂点からそれぞれ隣をみて、
  より小さい値で遷移できそうだったら、**コストを更新する！**

- 更に、すべての頂点からそれぞれ隣をみて、
  より小さい値で遷移できそうだったら、**コストを更新する！！**

⋮

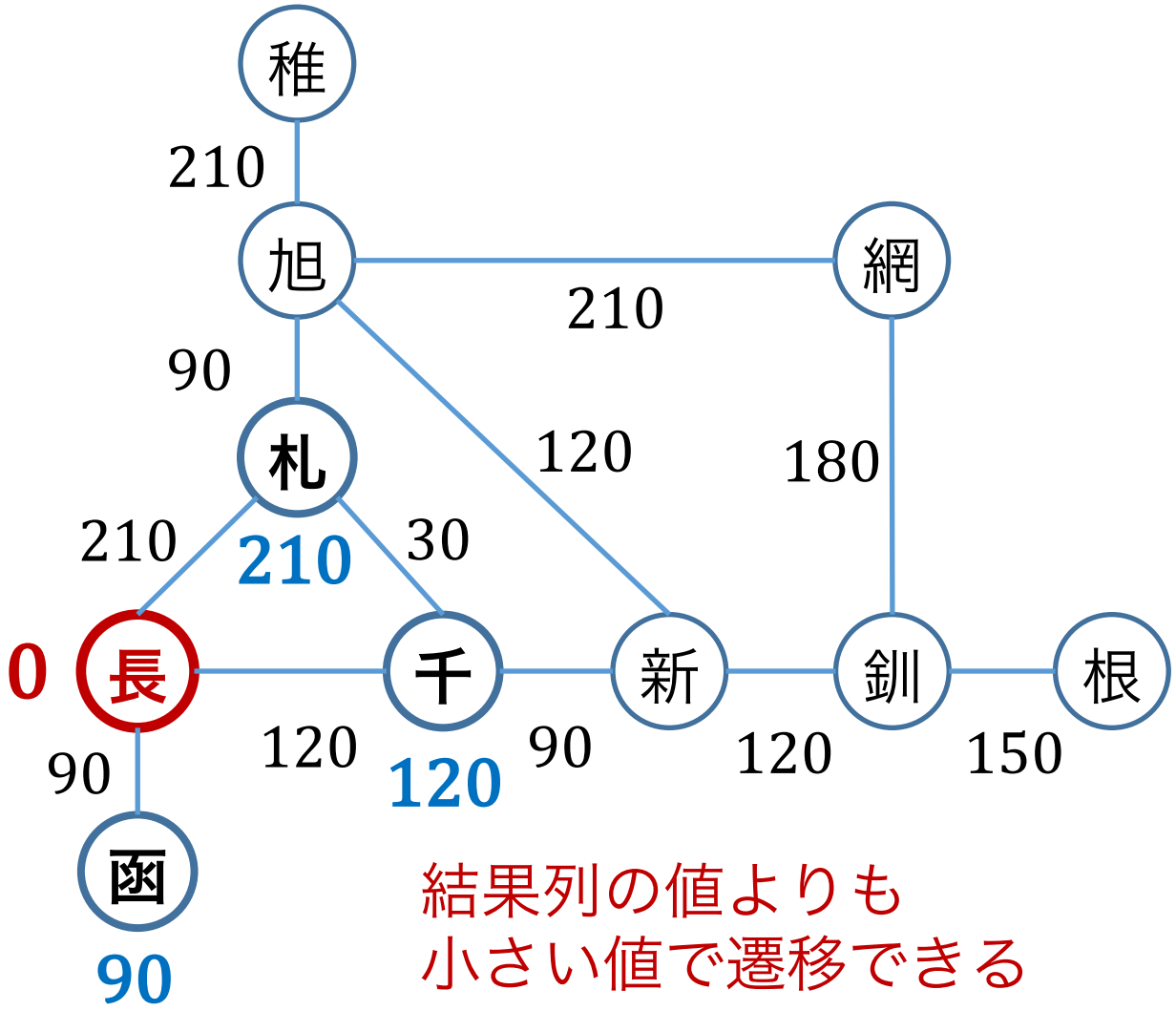
コストの更新



結果列

長万部	0
函館	INF
札幌	INF
旭川	INF
稚内	INF
千歳	INF
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新

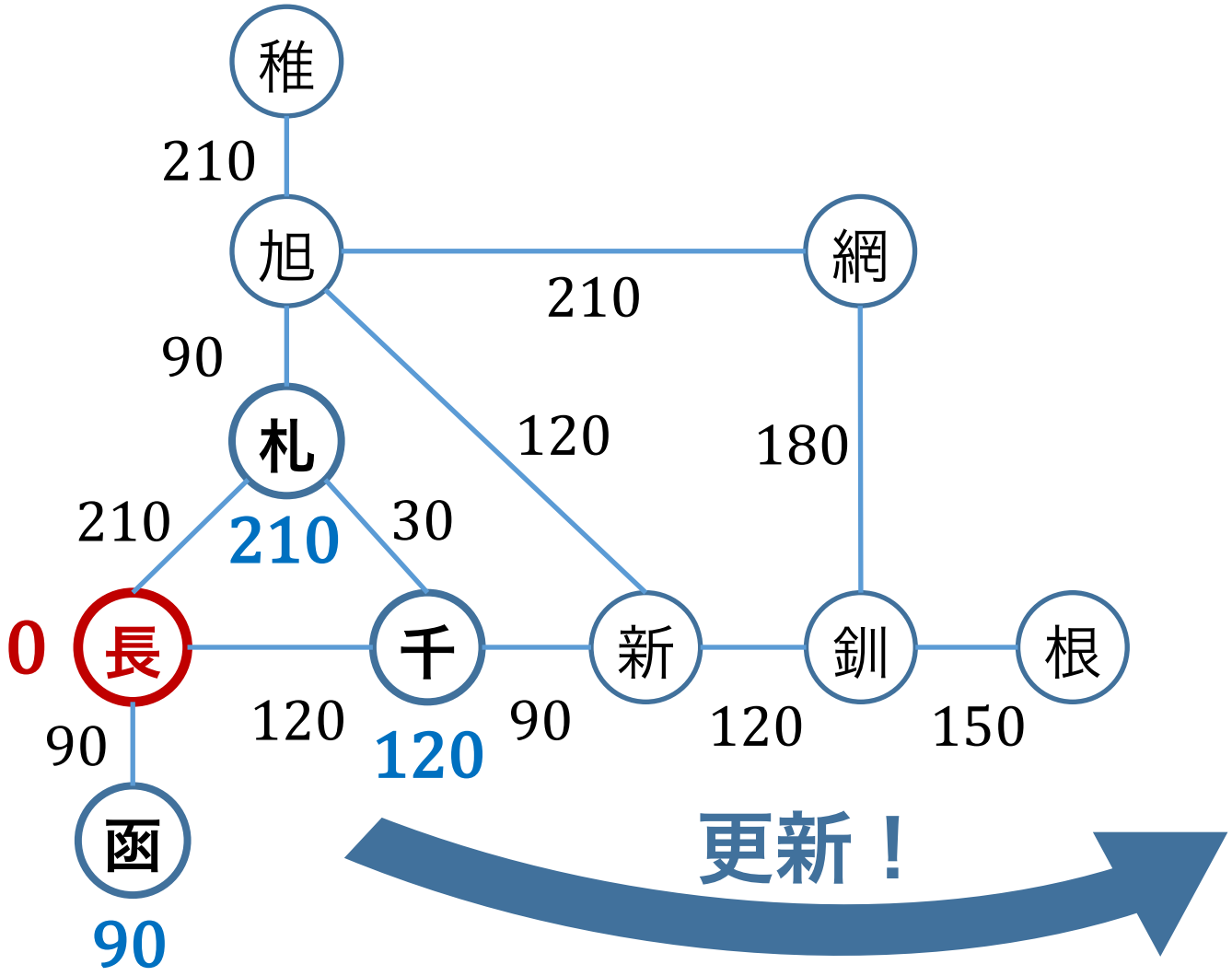


結果列の値よりも
小さい値で遷移できる

結果列

長万部	0
函館	INF
札幌	INF
旭川	INF
稚内	INF
千歳	INF
新得	INF
釧路	INF
根室	INF
網走	INF

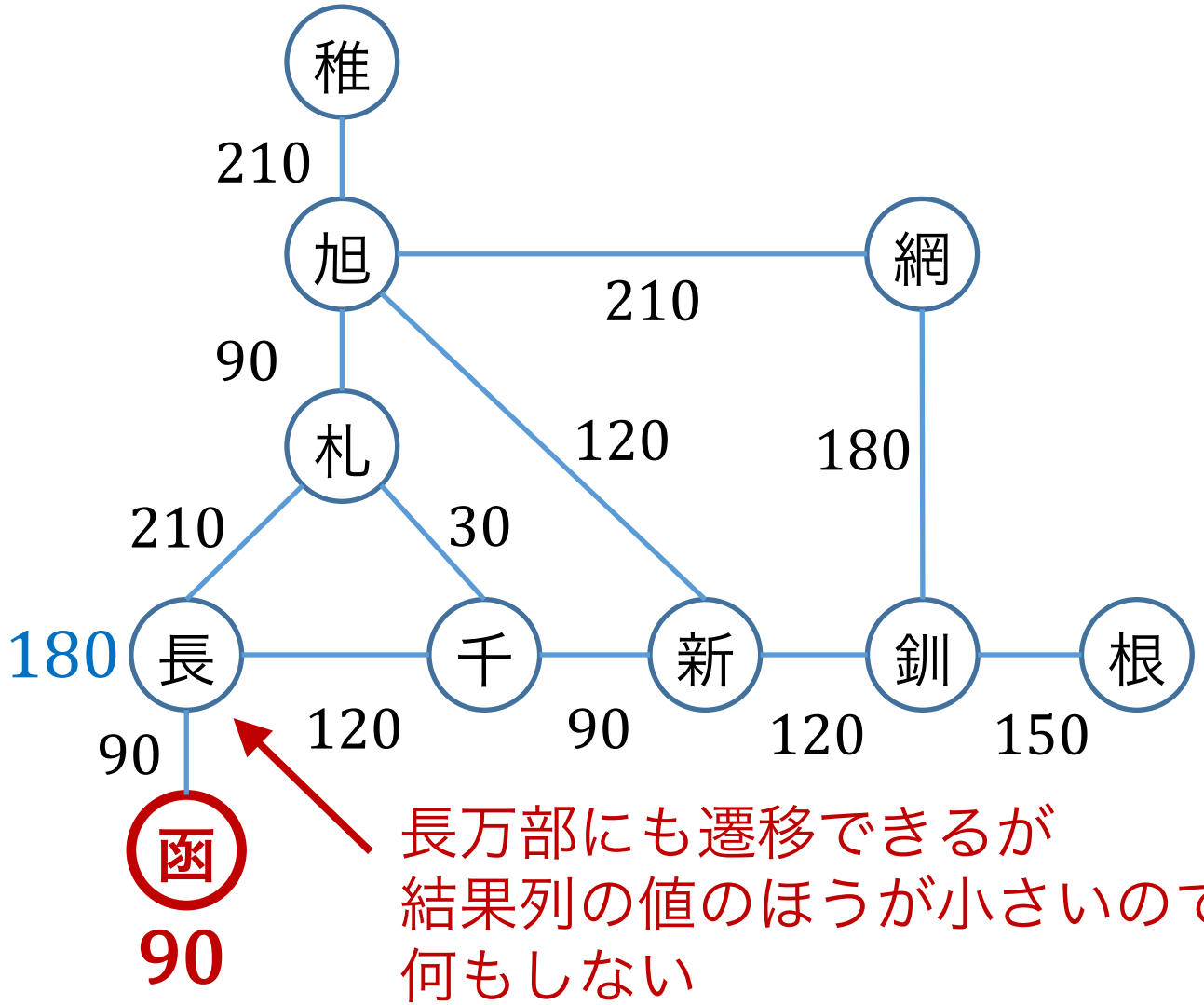
コストの更新



結果列

長万部	0
函館	90
札幌	210
旭川	INF
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

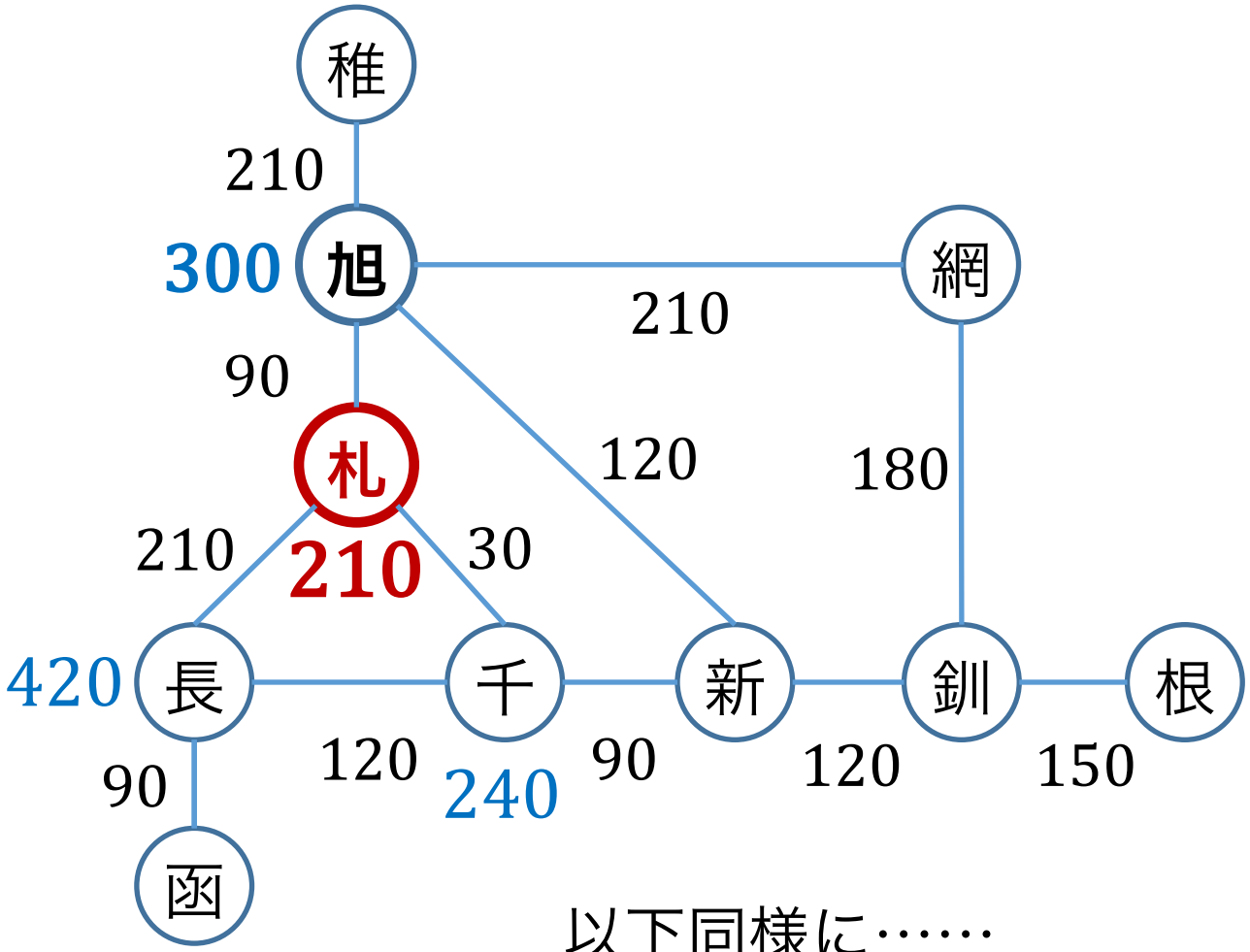
コストの更新



結果列

長万部	0
函館	90
札幌	210
旭川	INF
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新

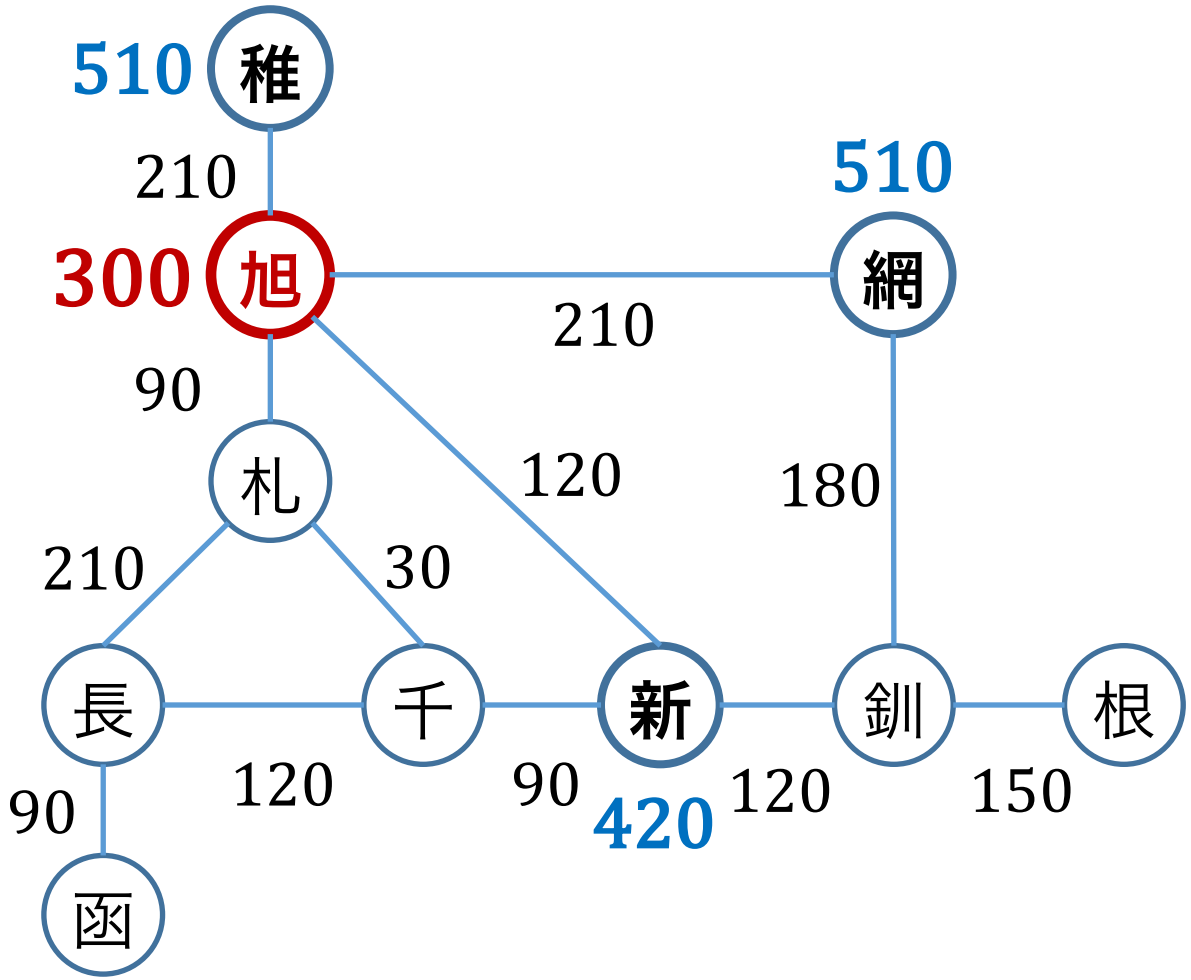


以下同様に……

結果列

長万部	0
函館	90
札幌	210
旭川	300
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

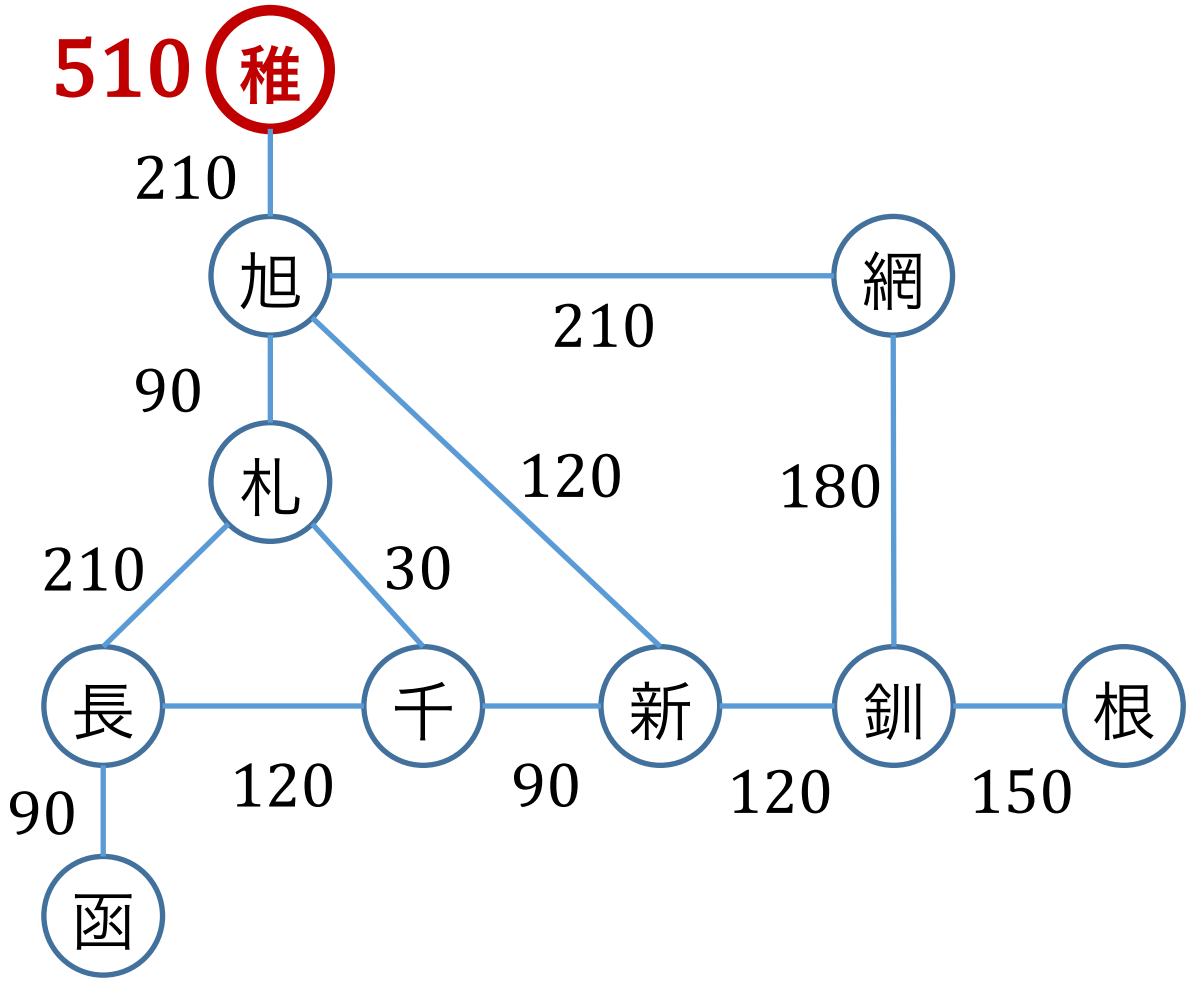
コストの更新



結果列

長万部	0
函館	90
札幌	210
旭川	300
稚内	510
千歳	120
新得	420
釧路	INF
根室	INF
網走	510

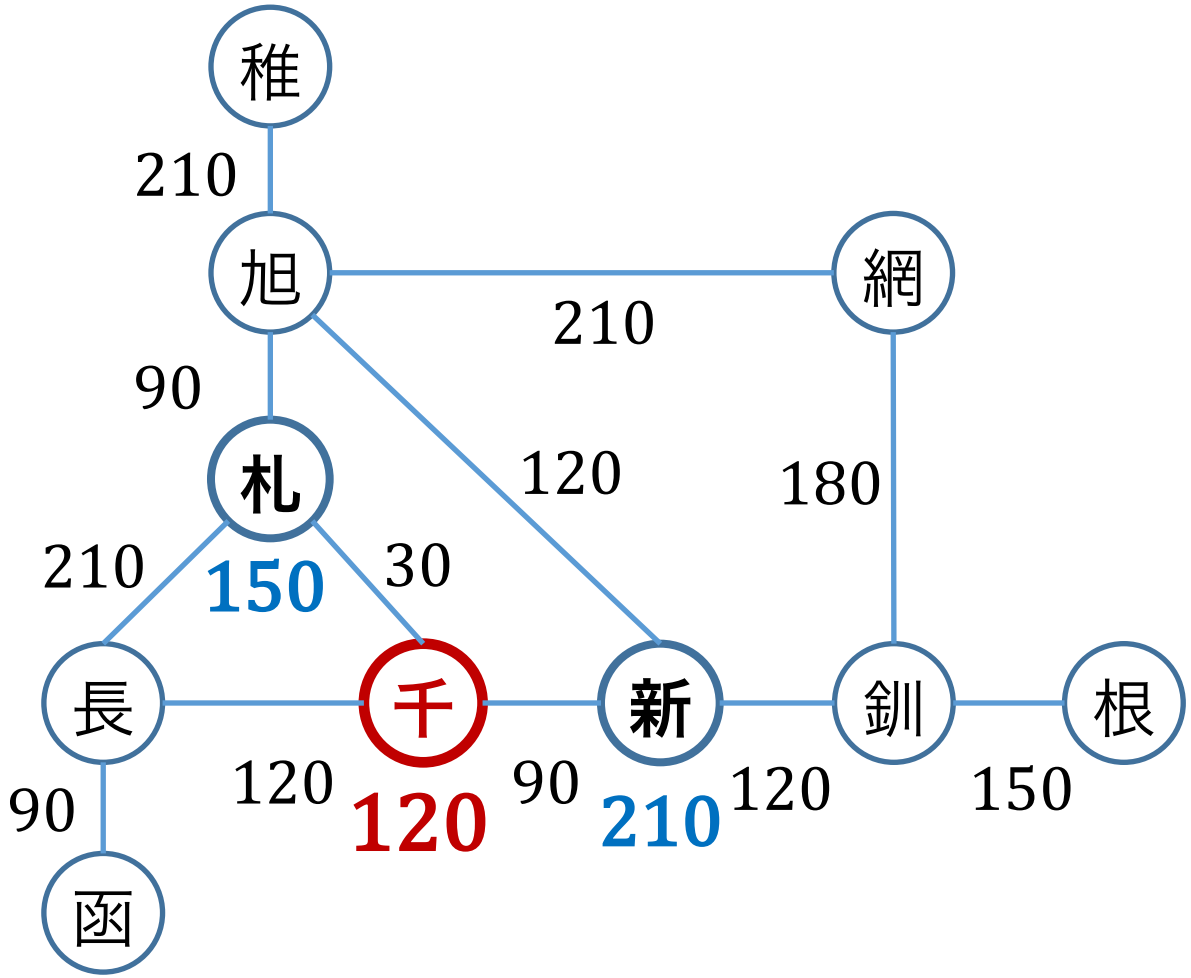
コストの更新



結果列

長万部	0
函館	90
札幌	210
旭川	300
稚内	510
千歳	120
新得	420
釧路	INF
根室	INF
網走	510

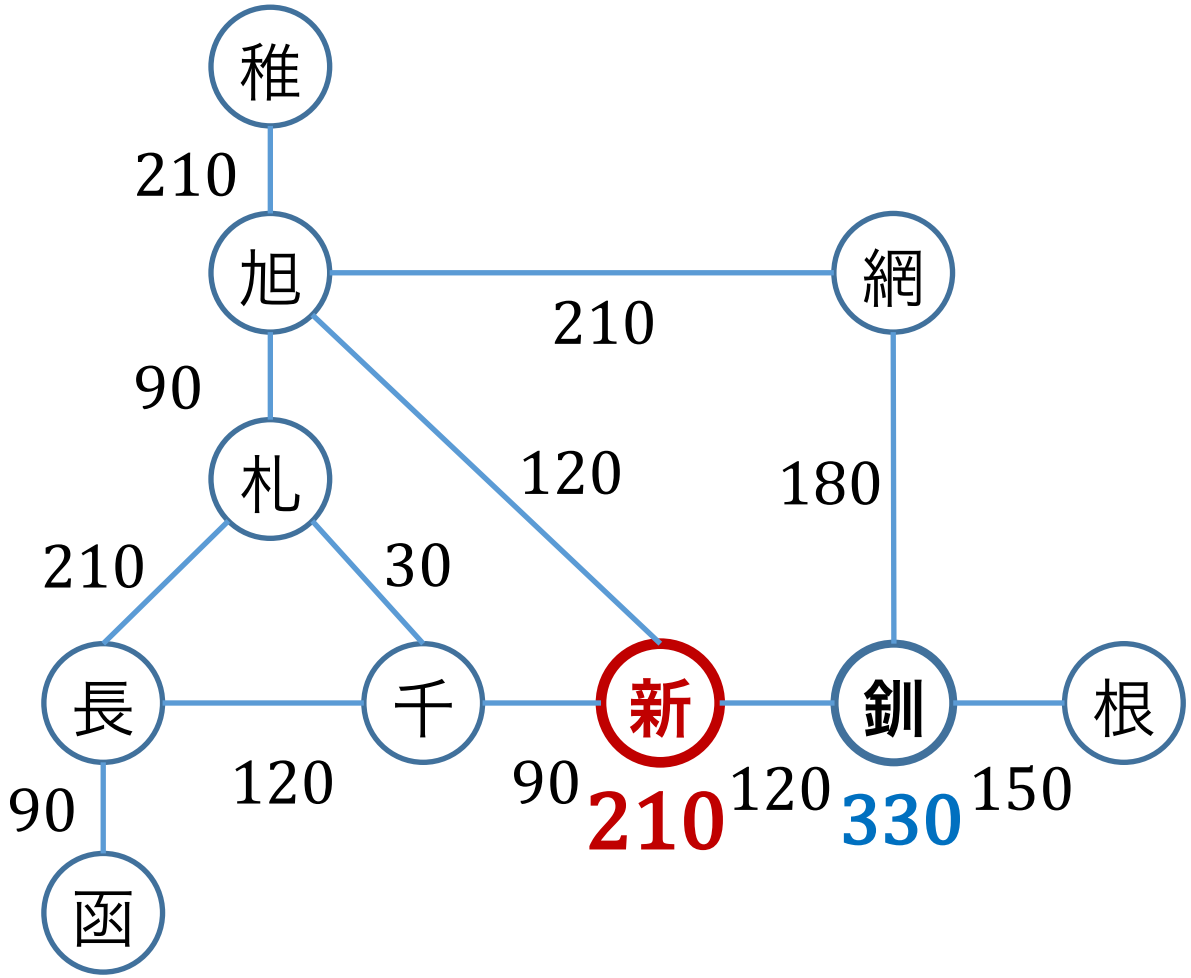
コストの更新



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	INF
根室	INF
網走	510

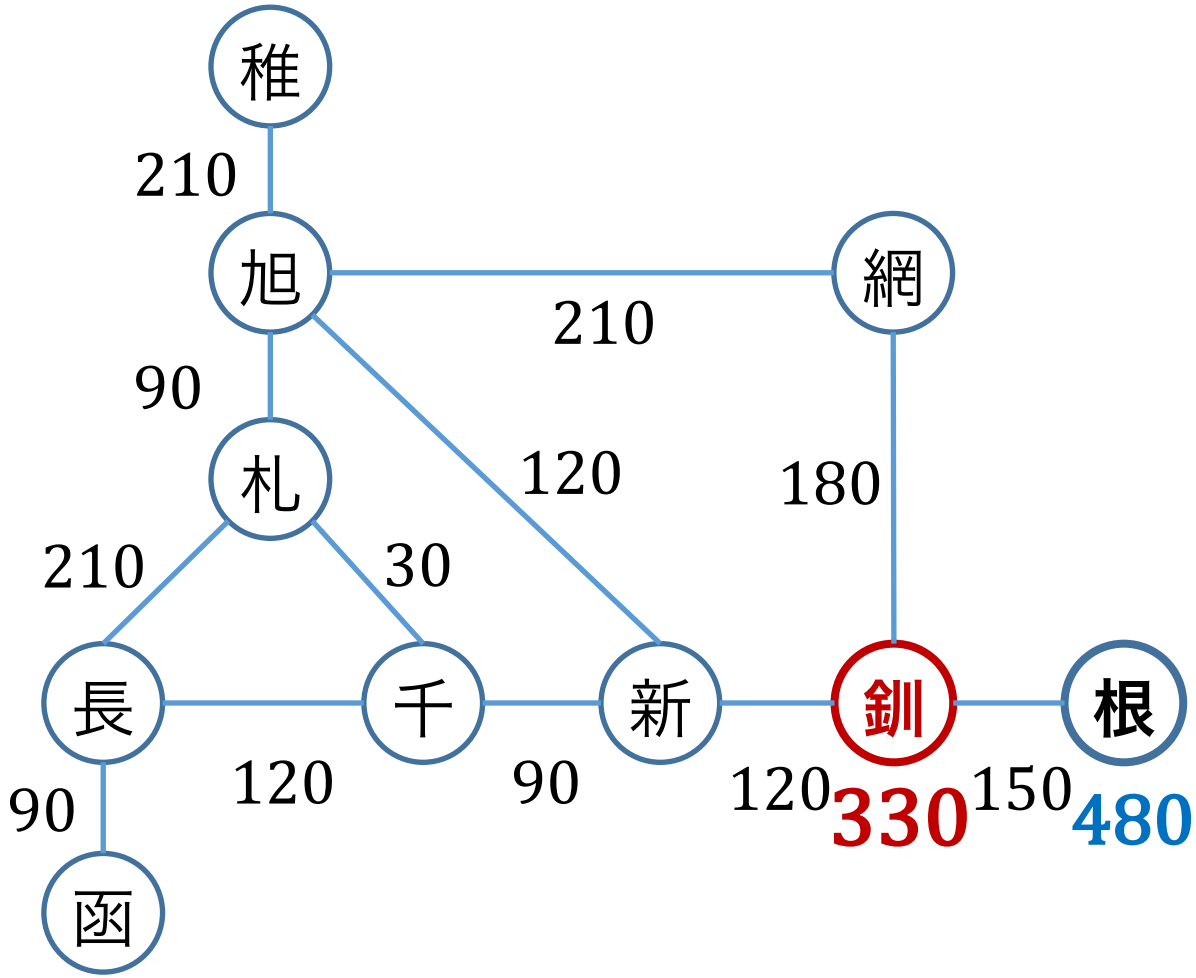
コストの更新



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	INF
網走	510

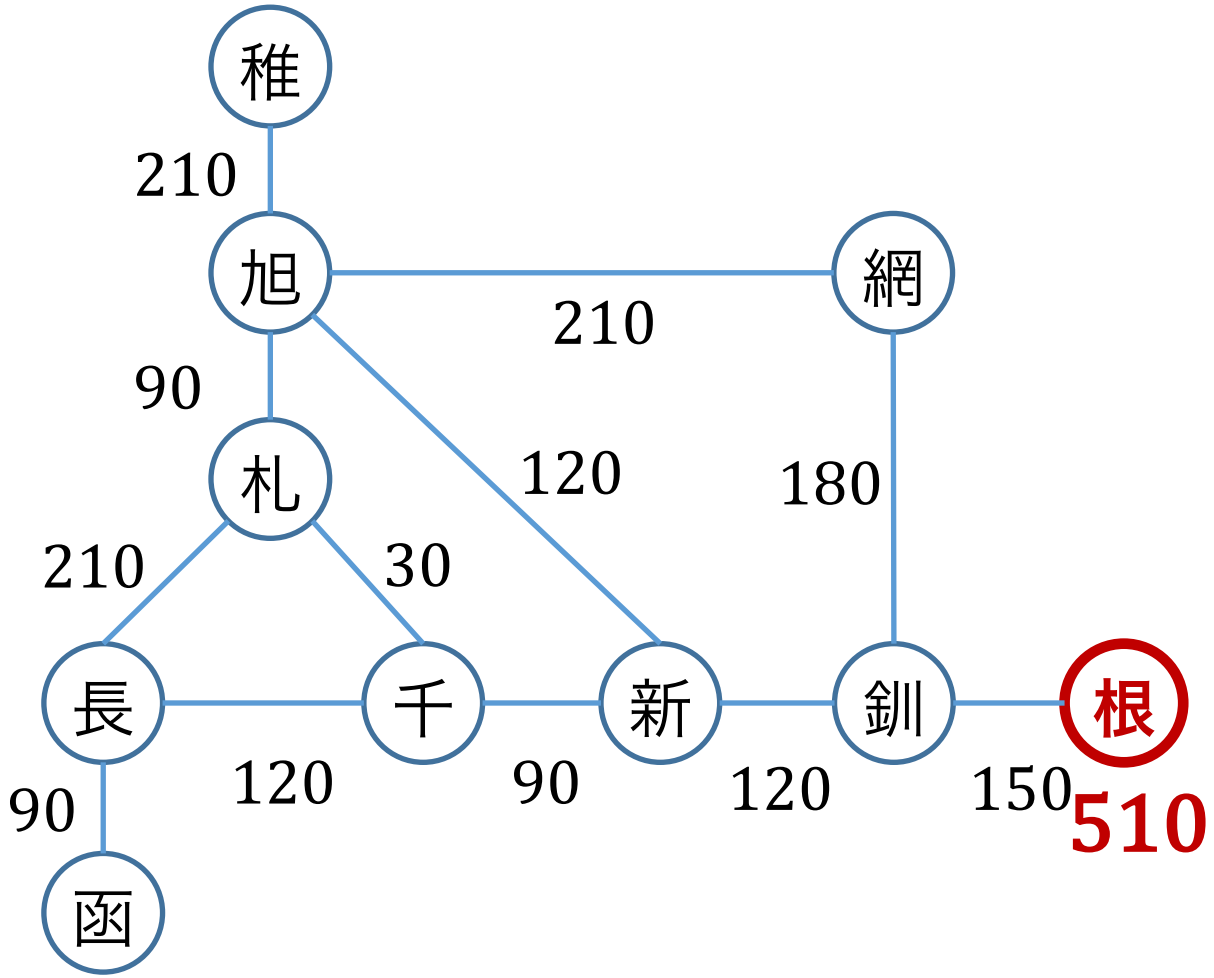
コストの更新



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

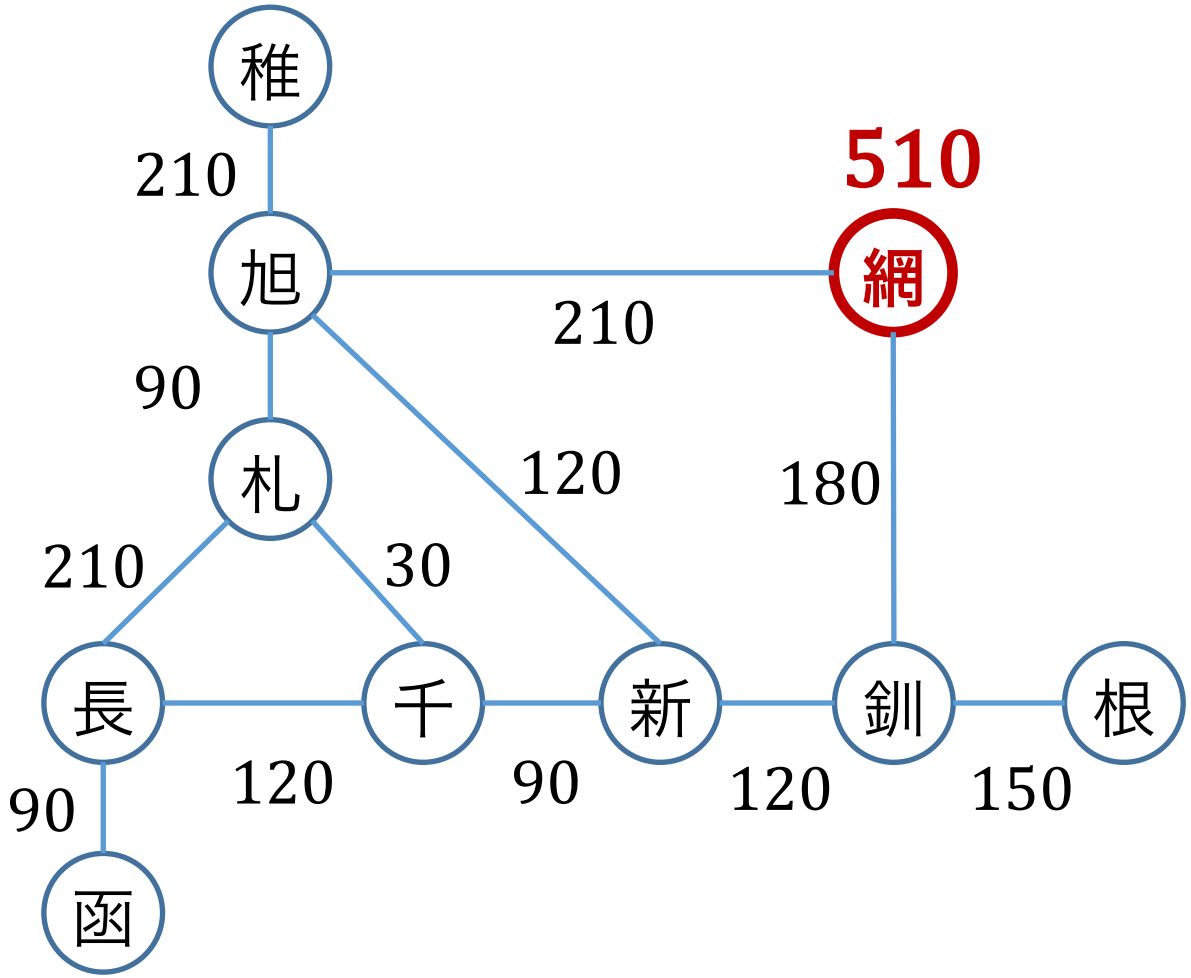
コストの更新



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

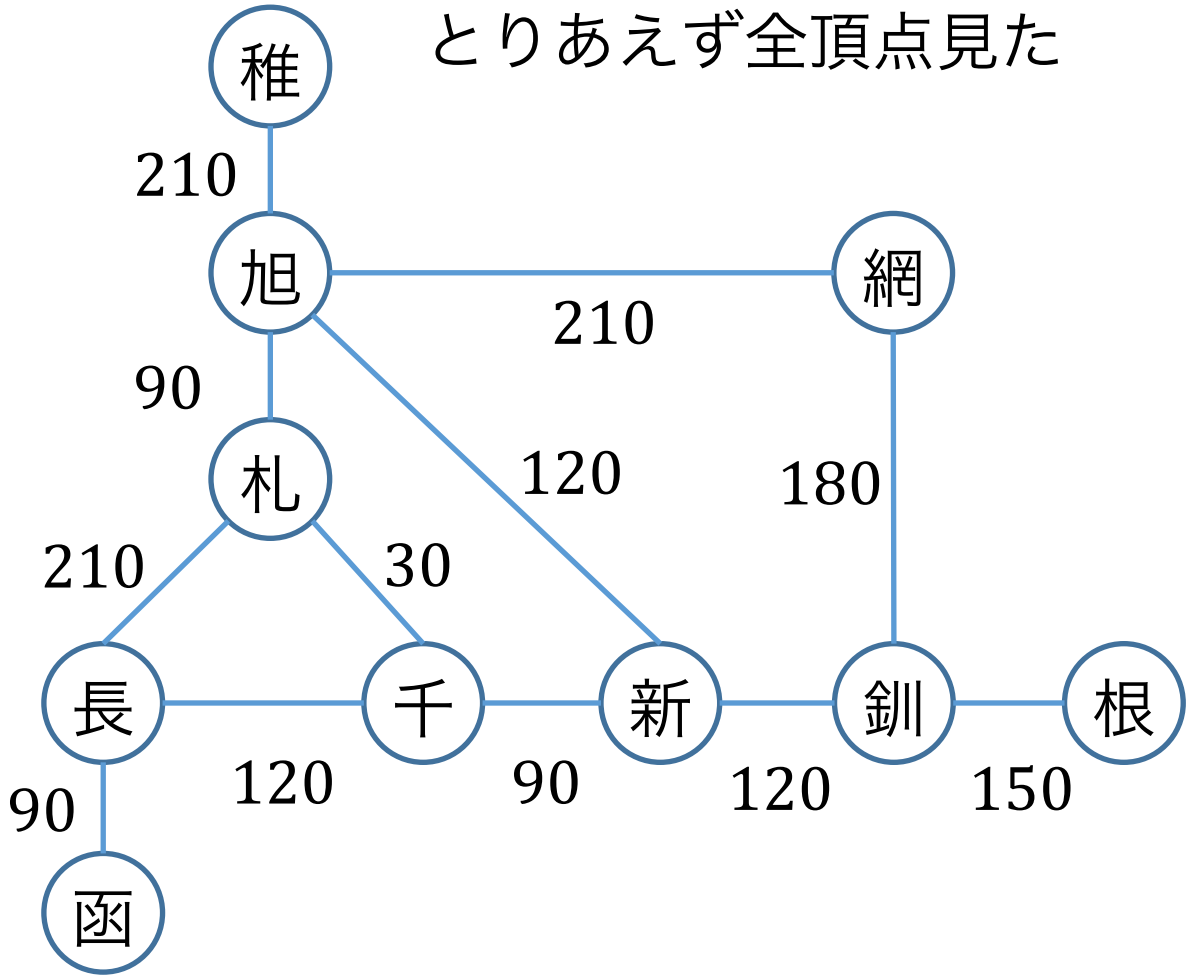
コストの更新



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

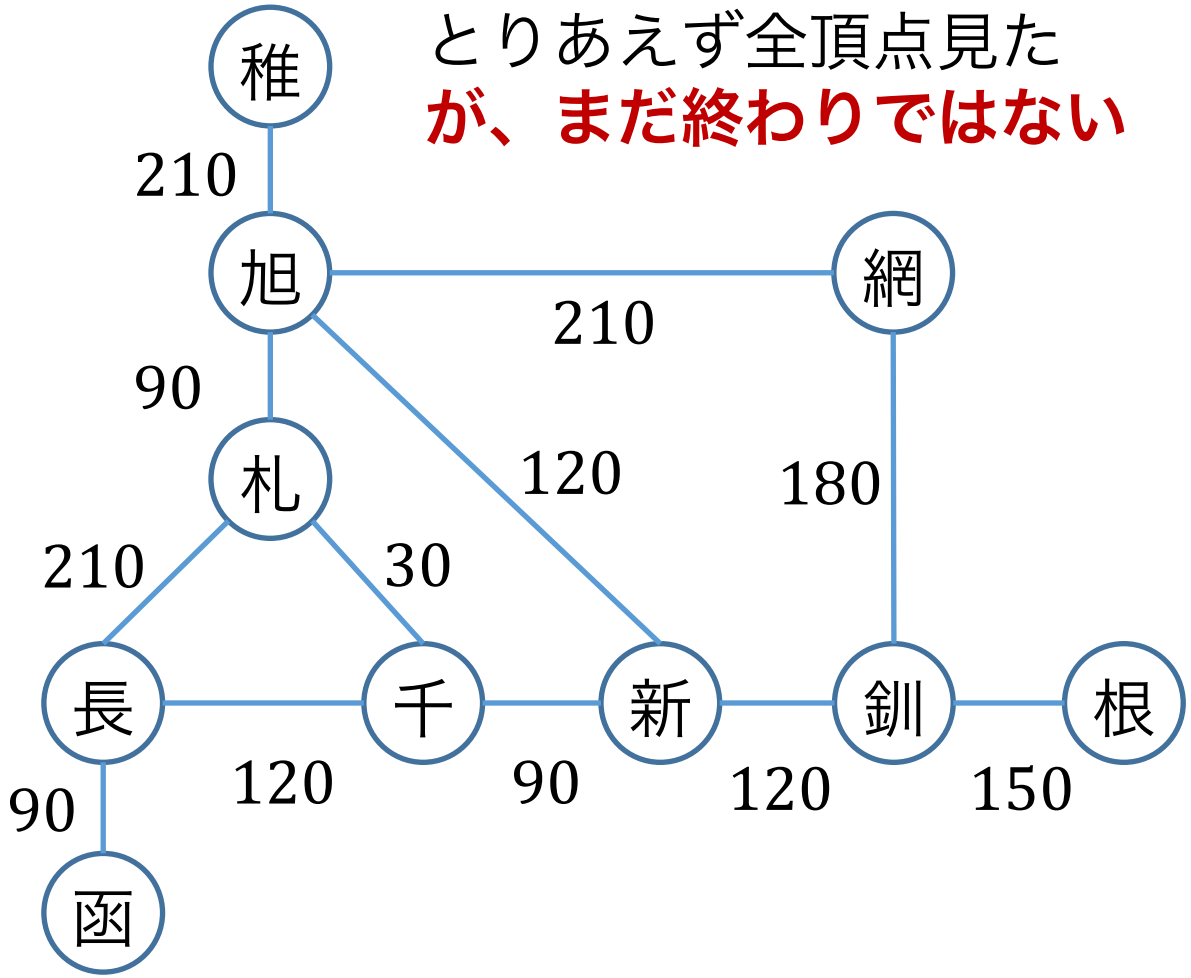
コストの更新



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

コストの更新

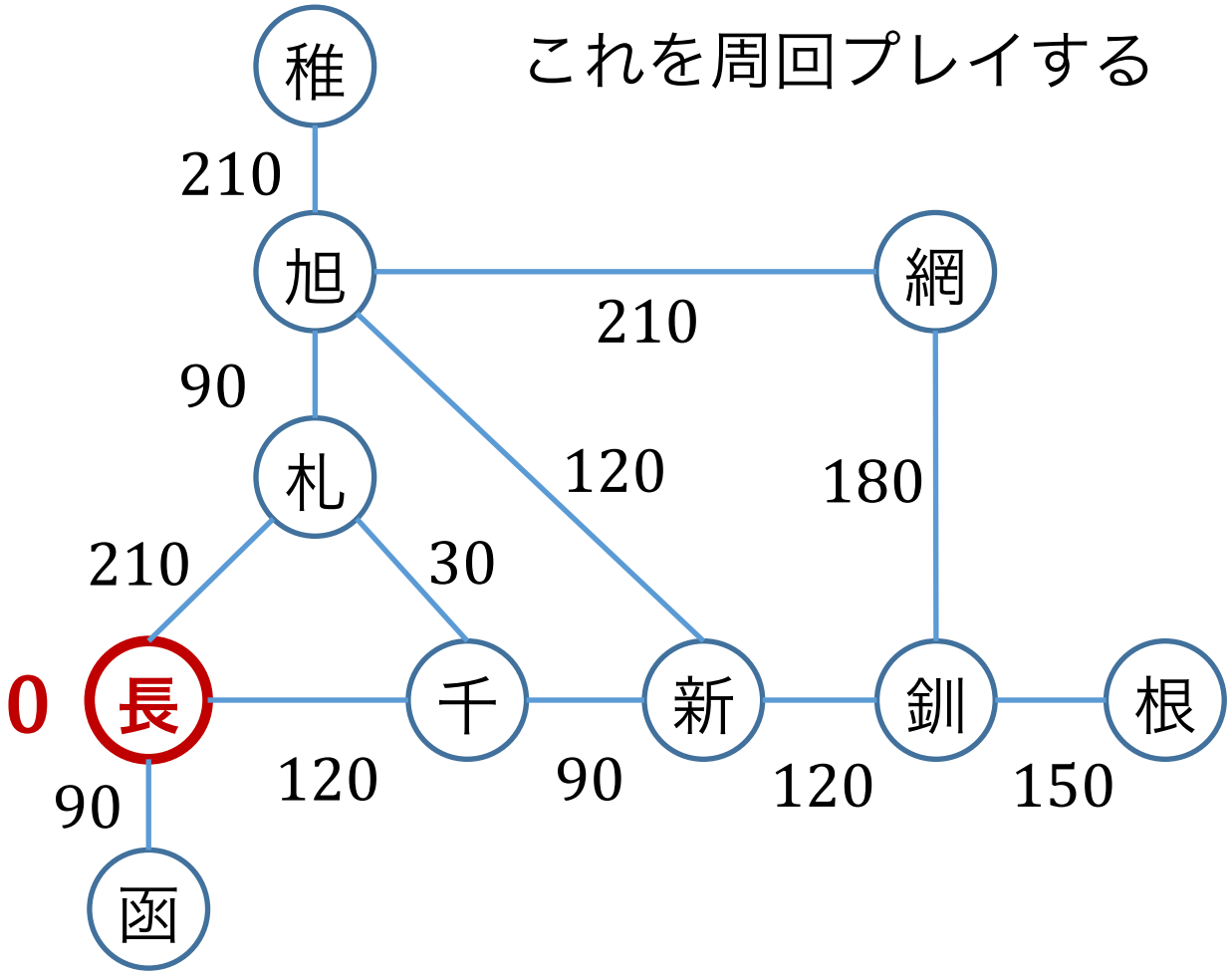


とりあえず全頂点見た
が、まだ終わりではない

結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

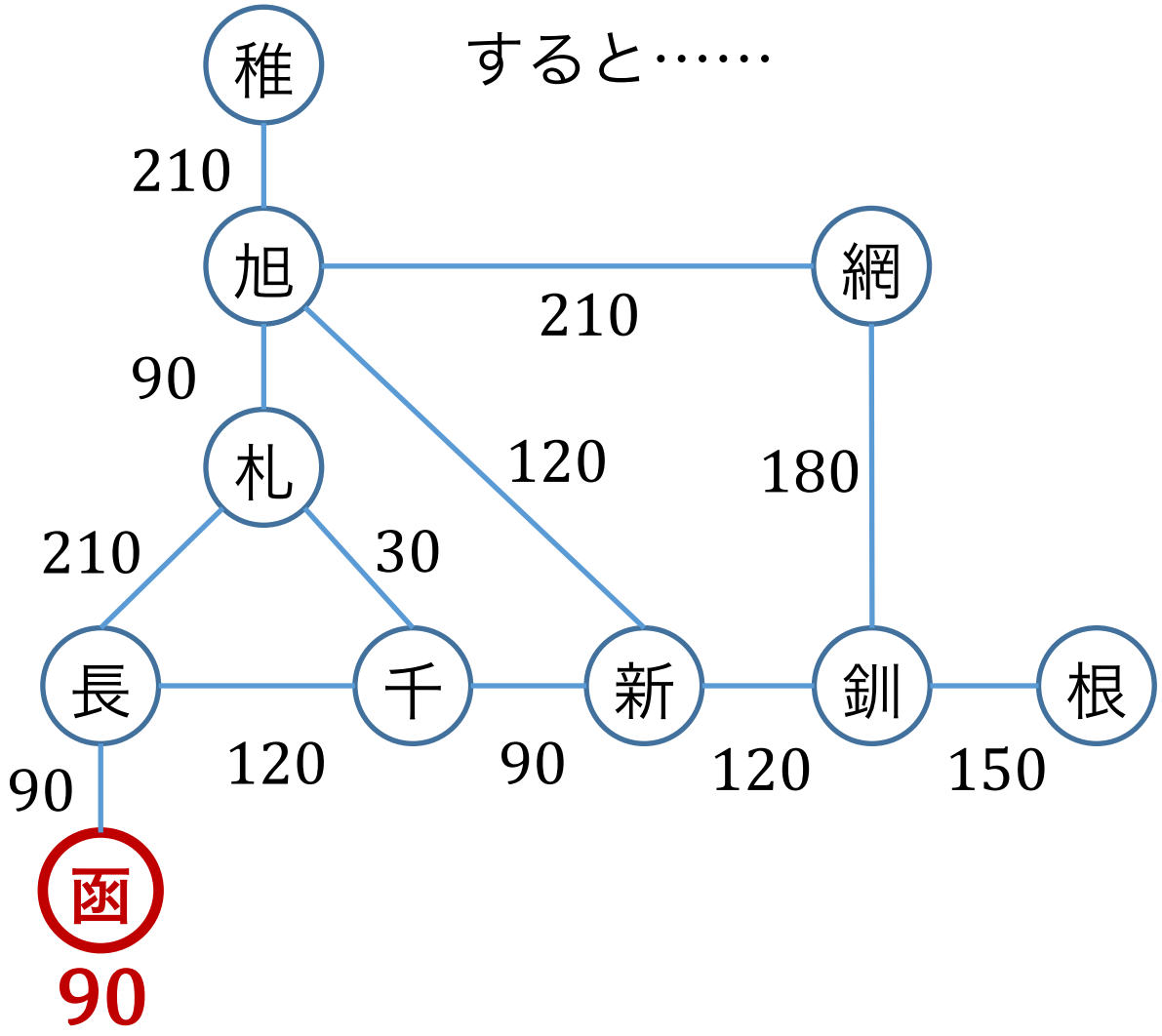
コストの更新 2周目



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

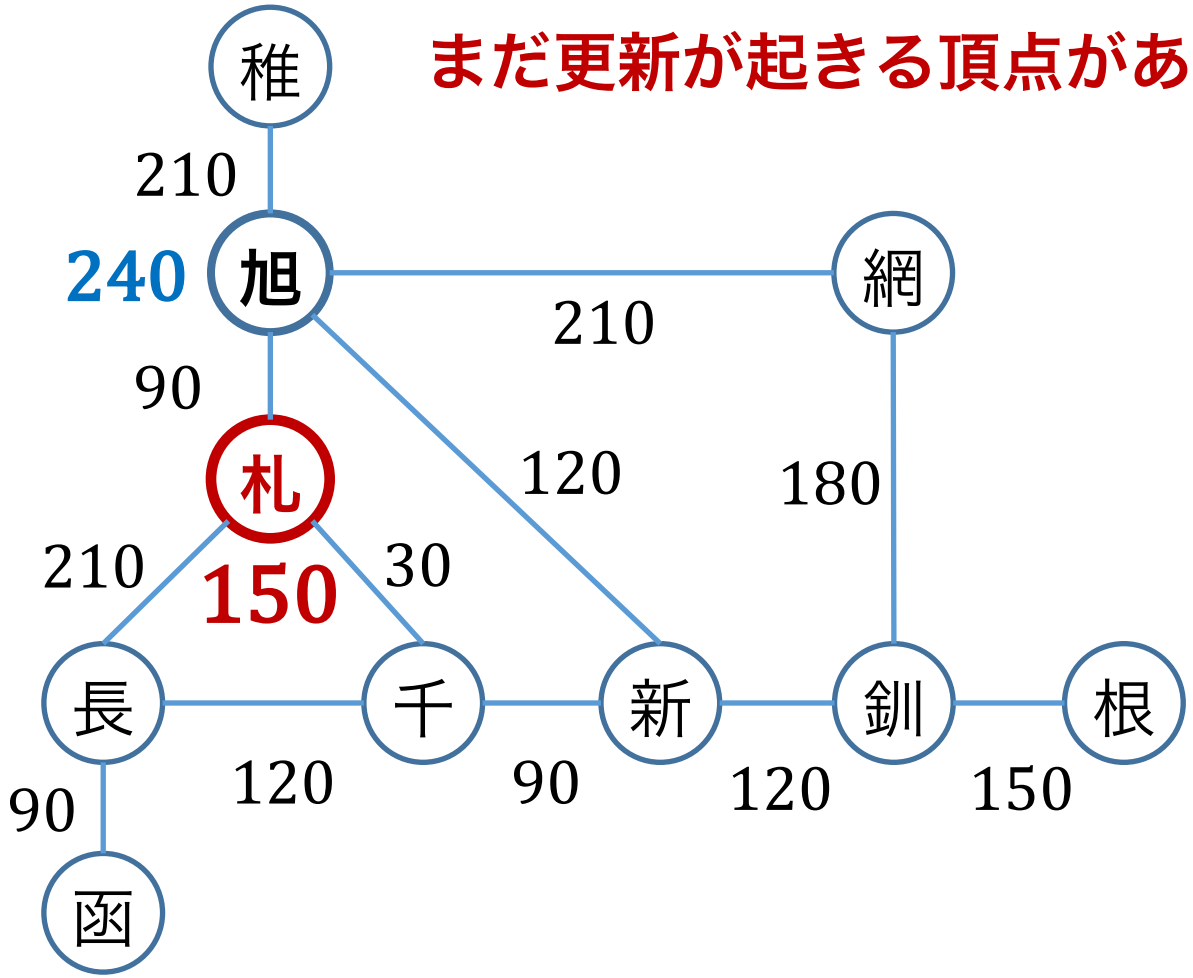
コストの更新 2周目



結果列

長万部	0
函館	90
札幌	150
旭川	300
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

コストの更新 2周目

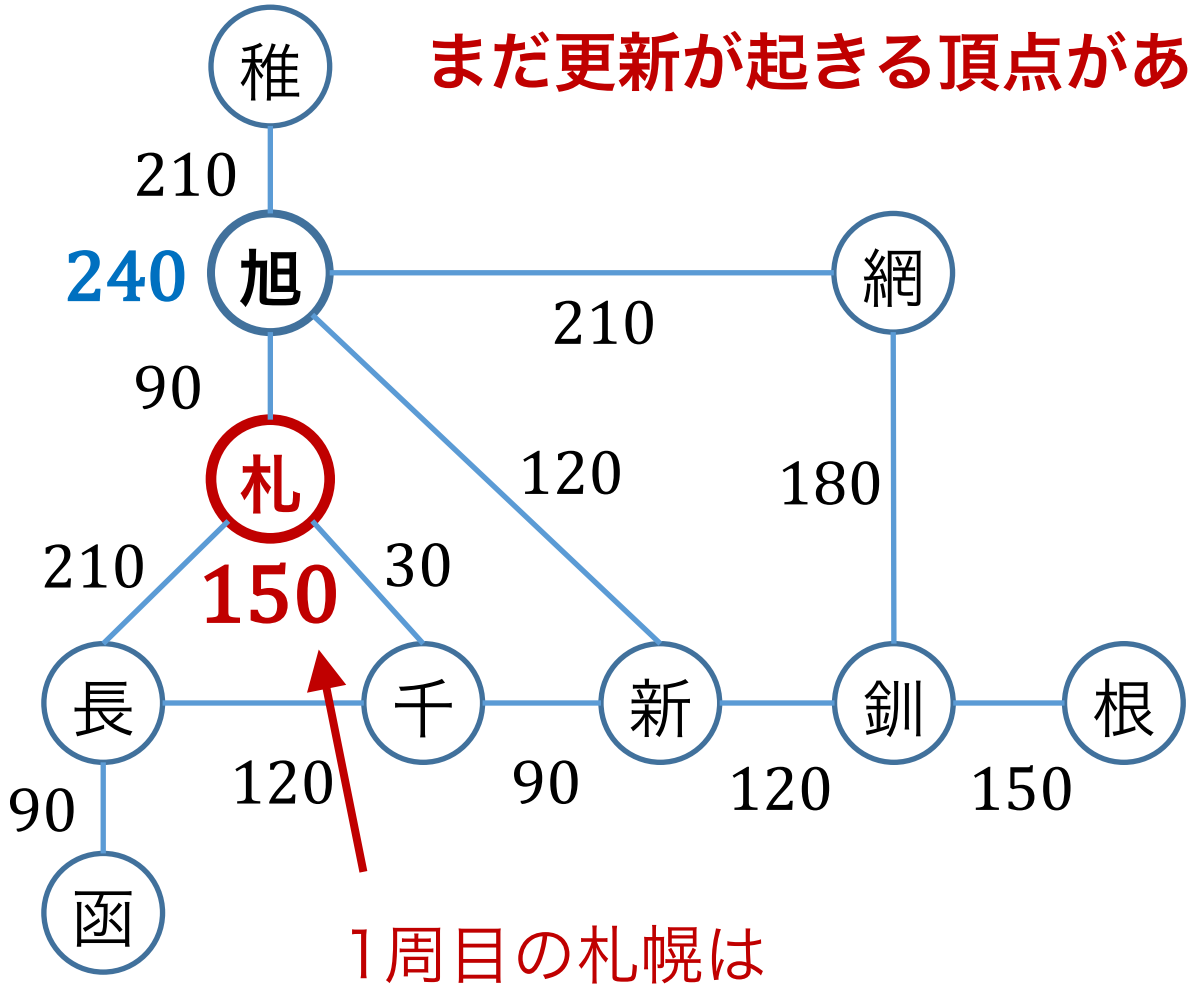


まだ更新が起きる頂点がある!!

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

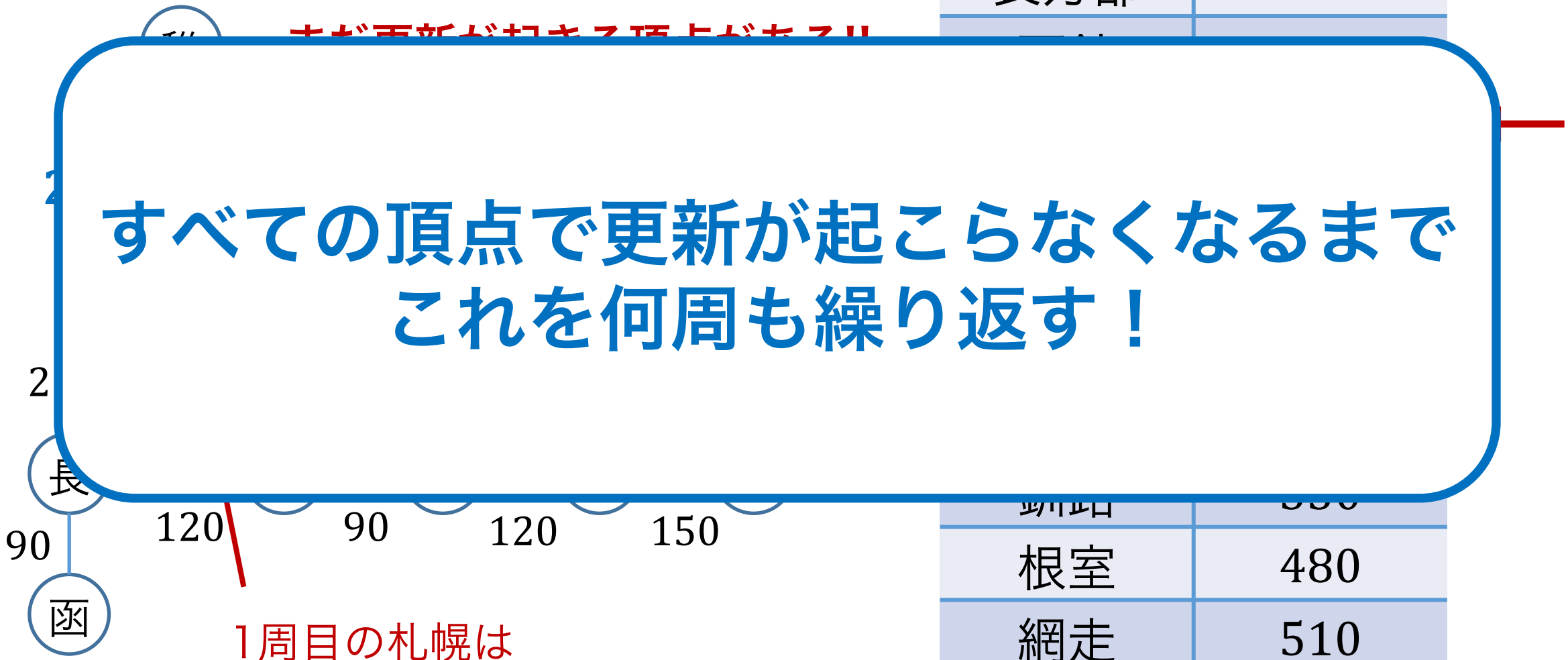
コストの更新 2周目



結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	510
千歳	120
新得	210
釧路	330
根室	480
網走	510

コストの更新 2周目



結果列

長万部	0
釧路	390
根室	480
網走	510

すべての頂点で更新が起こらなくなるまで
これを何周も繰り返す！

ベルマンフォード法の解析

- 1周ごとにやっていること:
  全頂点から、それぞれ隣接している頂点すべてを見ている

ベルマンフォード法の解析

- 1周ごとにやっていること:
  全頂点から、それぞれ隣接している頂点すべてを見ている
  つまり、**すべての枝をなめている**
  → O(E) **時間**

ベルマンフォード法の解析

- 1周ごとにやっていること:
  全頂点から、それぞれ隣接している頂点すべてを見ている
  つまり、**すべての枝をなめている**
  → $O(E)$ **時間**

- **じゃぁ、何周すればいいの？**

ベルマンフォード法の解析

- 1周ごとにやっていること:
  全頂点から、それぞれ隣接している頂点すべてを見ている
  つまり、**すべての枝をなめている**
  → **O(E) 時間**

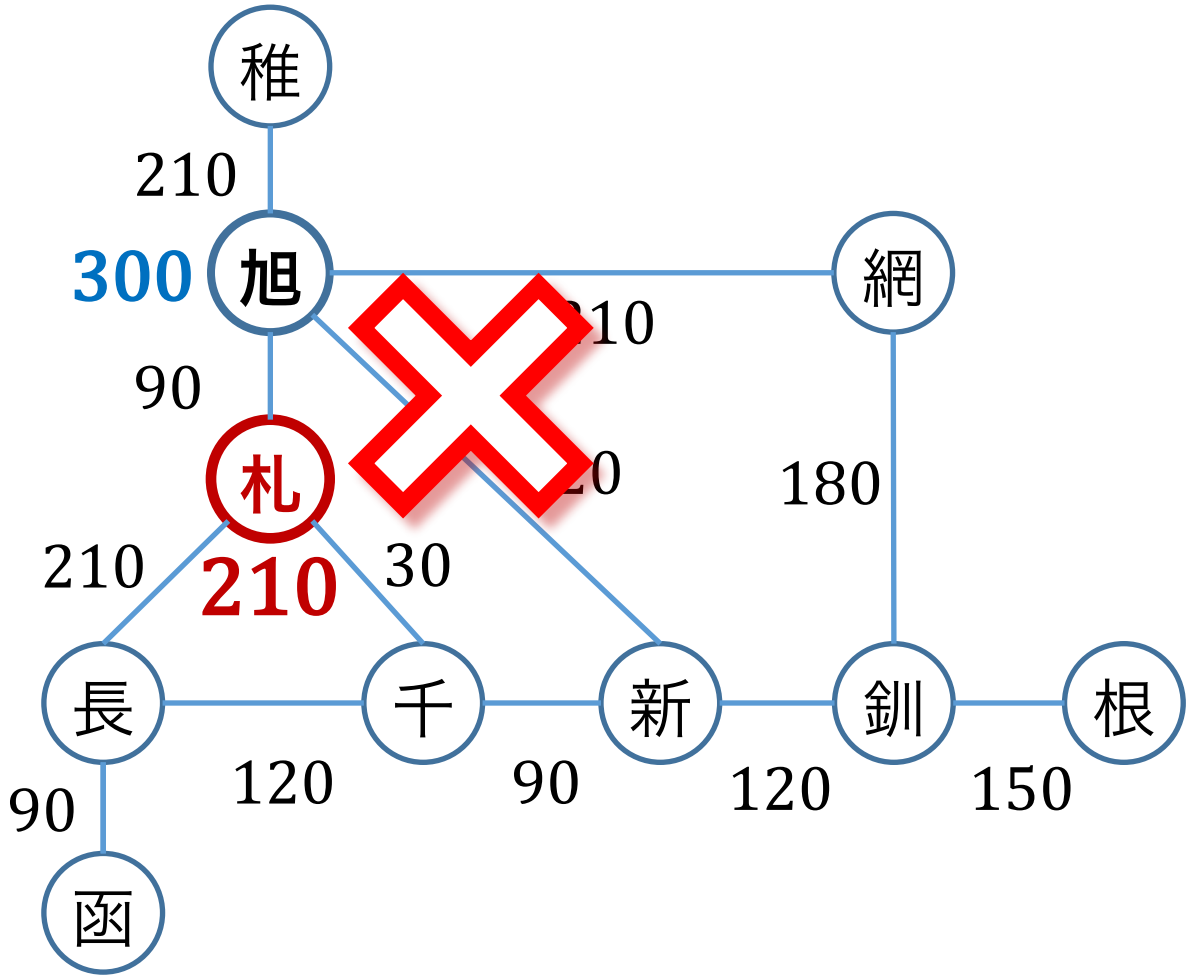
- **じゃぁ、何周すればいいの？**
  1周ごとに、少なくとも1つの頂点の最短コストが決定する
  つまり、**最大V回のループすれば更新が終わる**

全体として O(VE) 時間

実装例

```
 8 // ベルマンフォード
 9 // s: 開始ノード, v: 頂点数, adjlist: 隣接リスト (first: コスト second: 行き先)
10 vector<int> bellman(int s, int v, vector<vector<pair<int, int> > > adjlist){
11   bool update;
12   vector<int> result(v, INF);
13   result[s] = 0;
14
15   do{
16     update = false; // とりあえず更新フラグを折っておく
17
18     // 隣接リストをすべてなめる
19     for (int i = 0; i < v; i++) {
20?      for (int j = 0; j ≤ adjlist[i].size(); j++) {
21         // 隣接している点のコストを更新できるか確認する
22         int adjpoint = adjlist[i][j].second;
23         int adjcost = adjlist[i][j].first;
24         if (result[adjpoint] > result[i] + adjcost) {
25           result[adjpoint] = result[i] + adjcost;
26           update = true; // 更新できた場合、フラグを立てる
27         }
28       }
29     }
30     // フラグが立っている限りループ
31   }while(update);
32
33   // 返り値: 結果列
34   return result;
35 }
```

ダイクストラ法 (Dijkstra's algorithm)

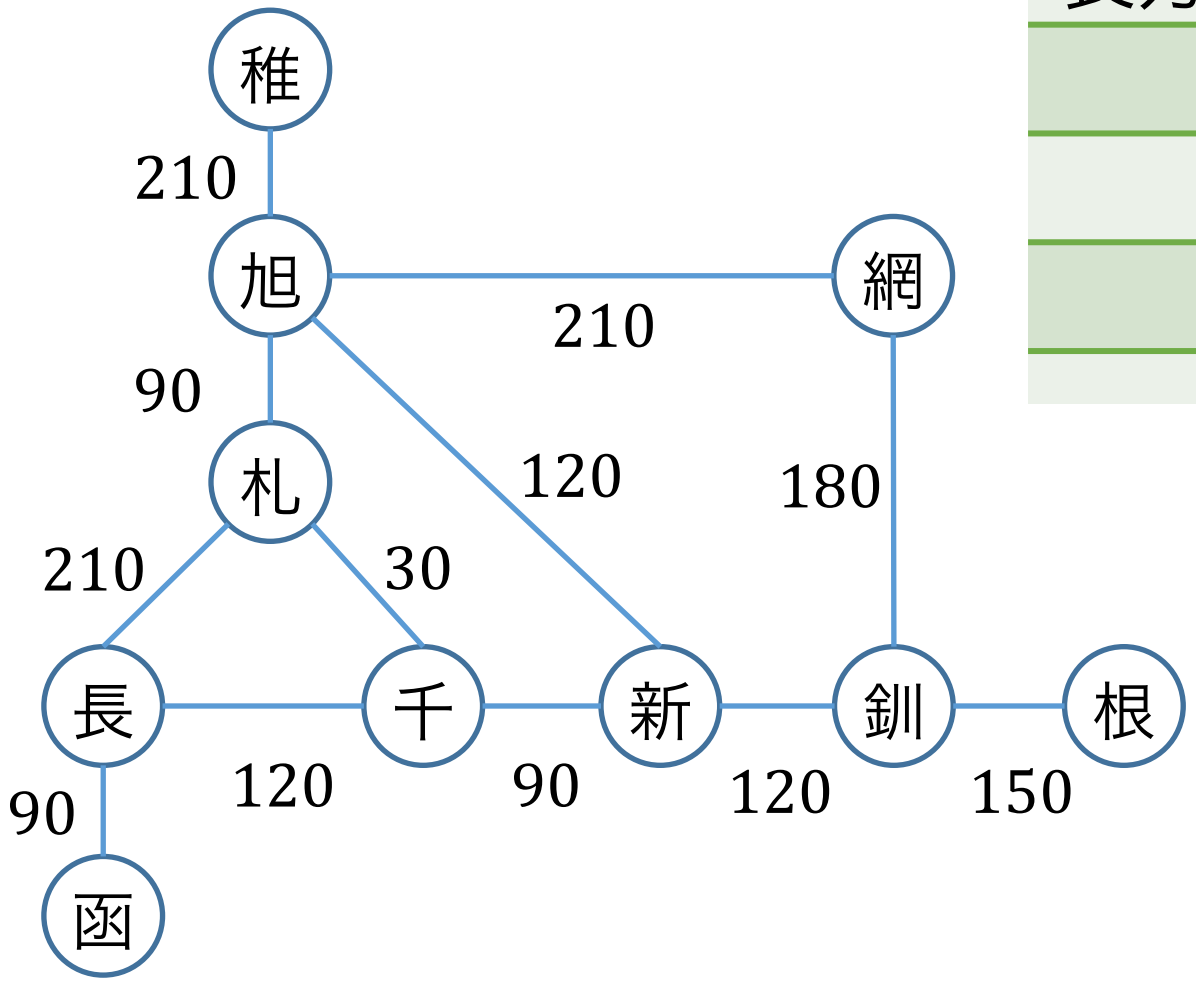


- ベルマンフォードでは
  札幌までの最短コストが
  確定する前に、
  旭川に手を出していた

  → **無駄な更新**

この無駄を解消したのが
ダイクストラ法！

コストの更新



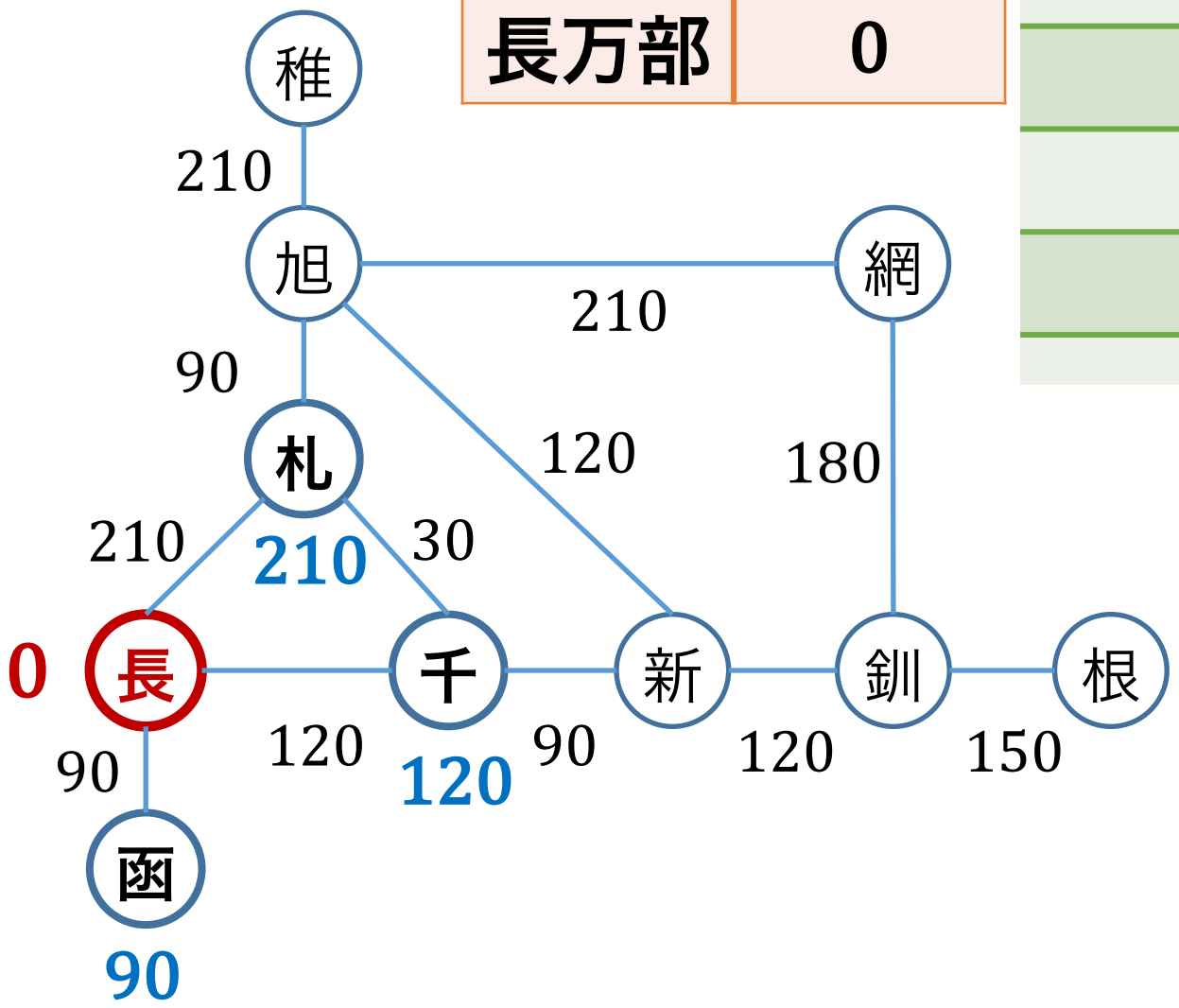
待機列 (昇順)

長万部	0

結果列

長万部	0
函館	INF
札幌	INF
旭川	INF
稚内	INF
千歳	INF
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



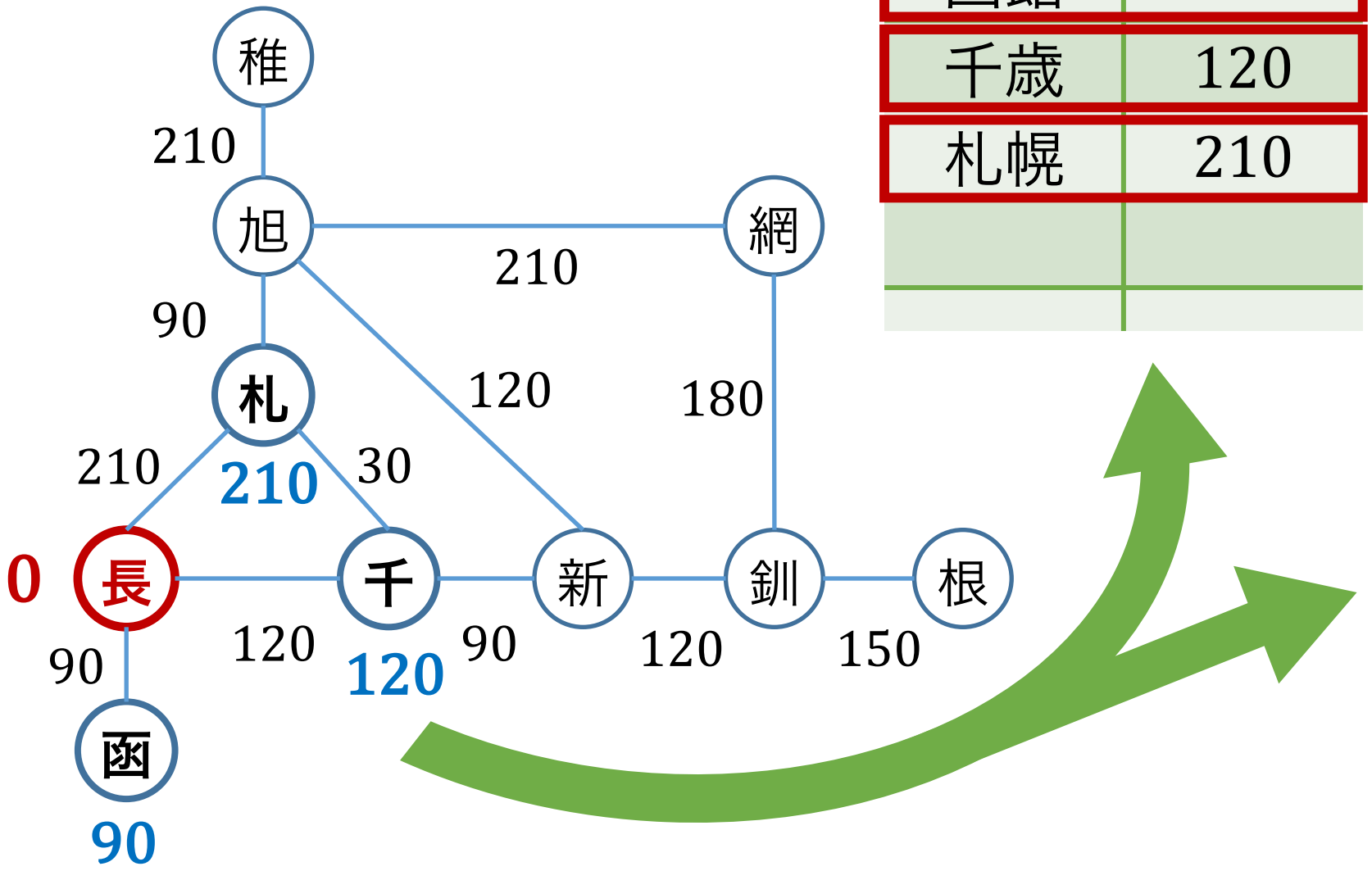
長万部	0

待機列 (昇順)

結果列

長万部	0
函館	INF
札幌	INF
旭川	INF
稚内	INF
千歳	INF
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



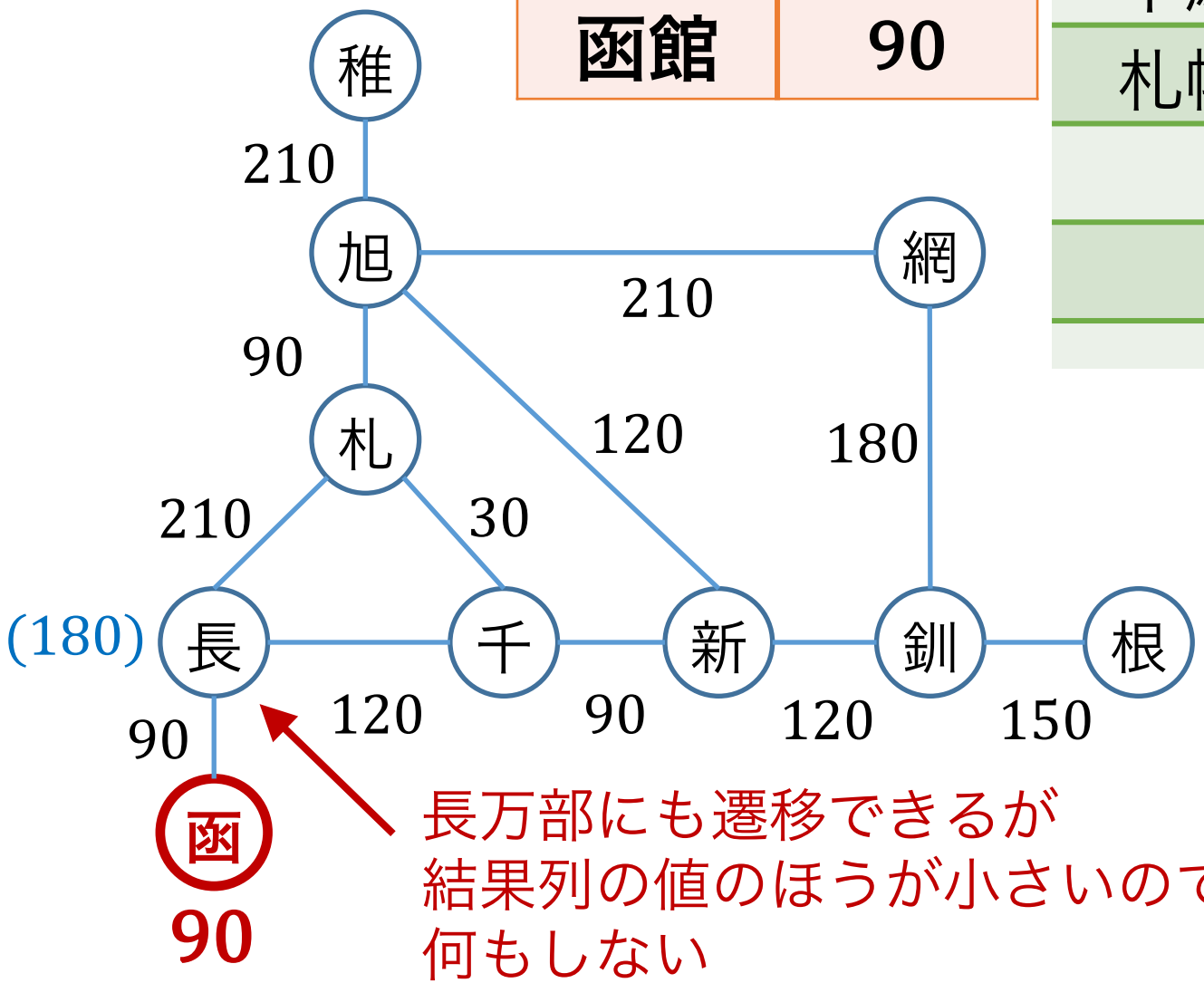
待機列 (昇順)

函館	90
千歳	120
札幌	210

結果列

長万部	0
函館	90
札幌	210
旭川	INF
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



函館	90

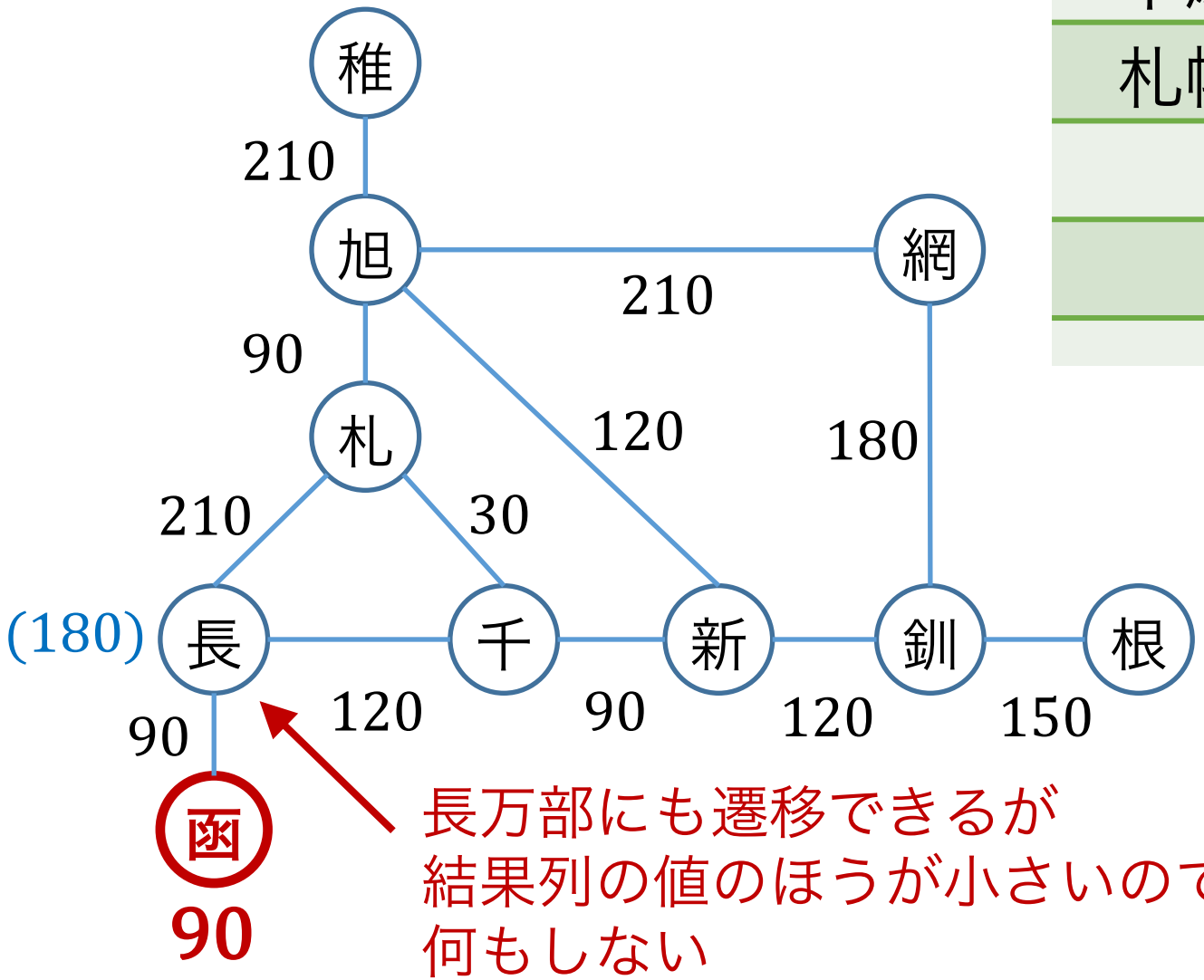
待機列 (昇順)

千歳	120
札幌	210

結果列

長万部	0
函館	90
札幌	210
旭川	INF
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



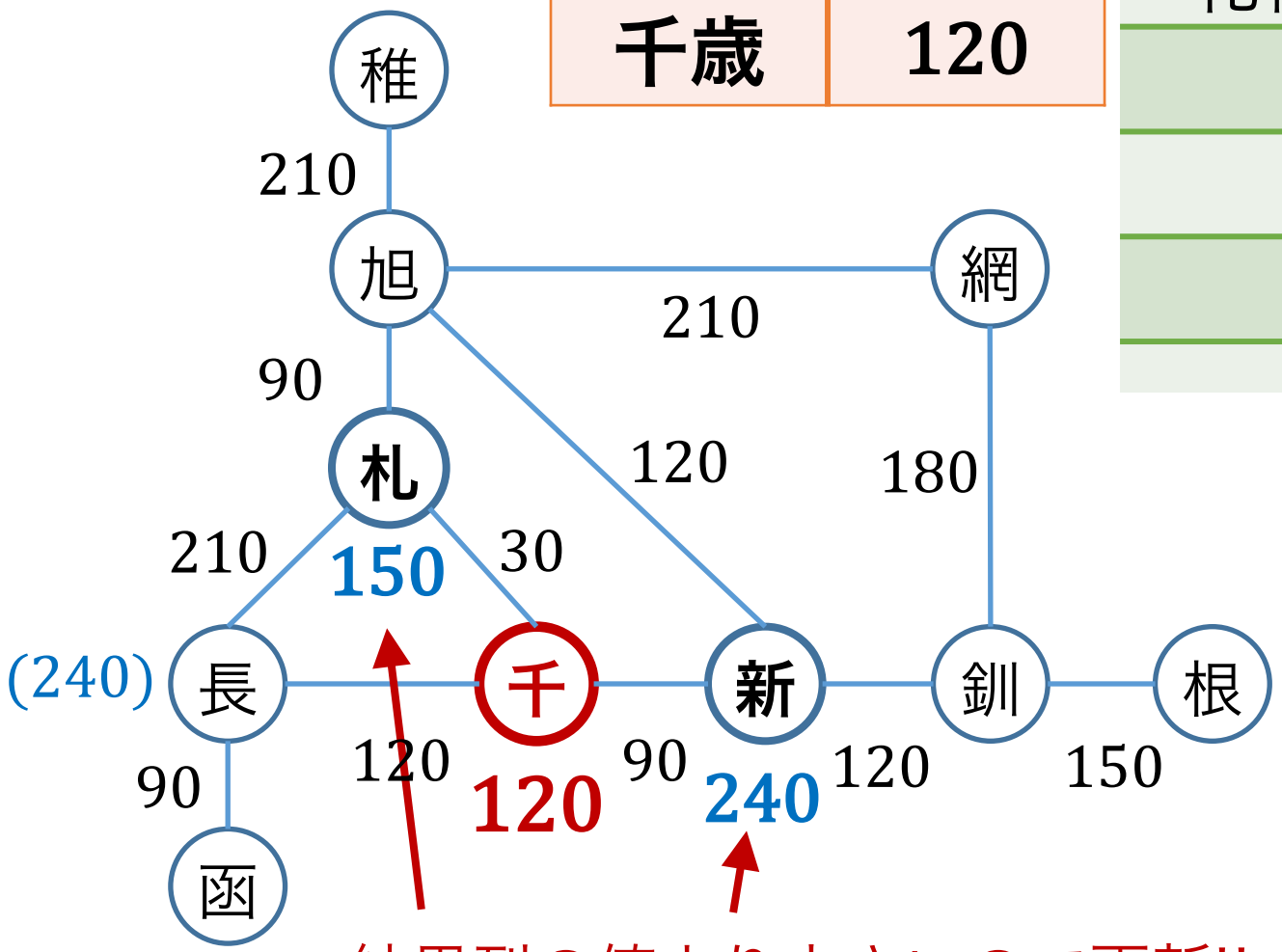
待機列 (昇順)

千歳	120
札幌	210

結果列

長万部	0
函館	90
札幌	210
旭川	INF
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



千歳	120

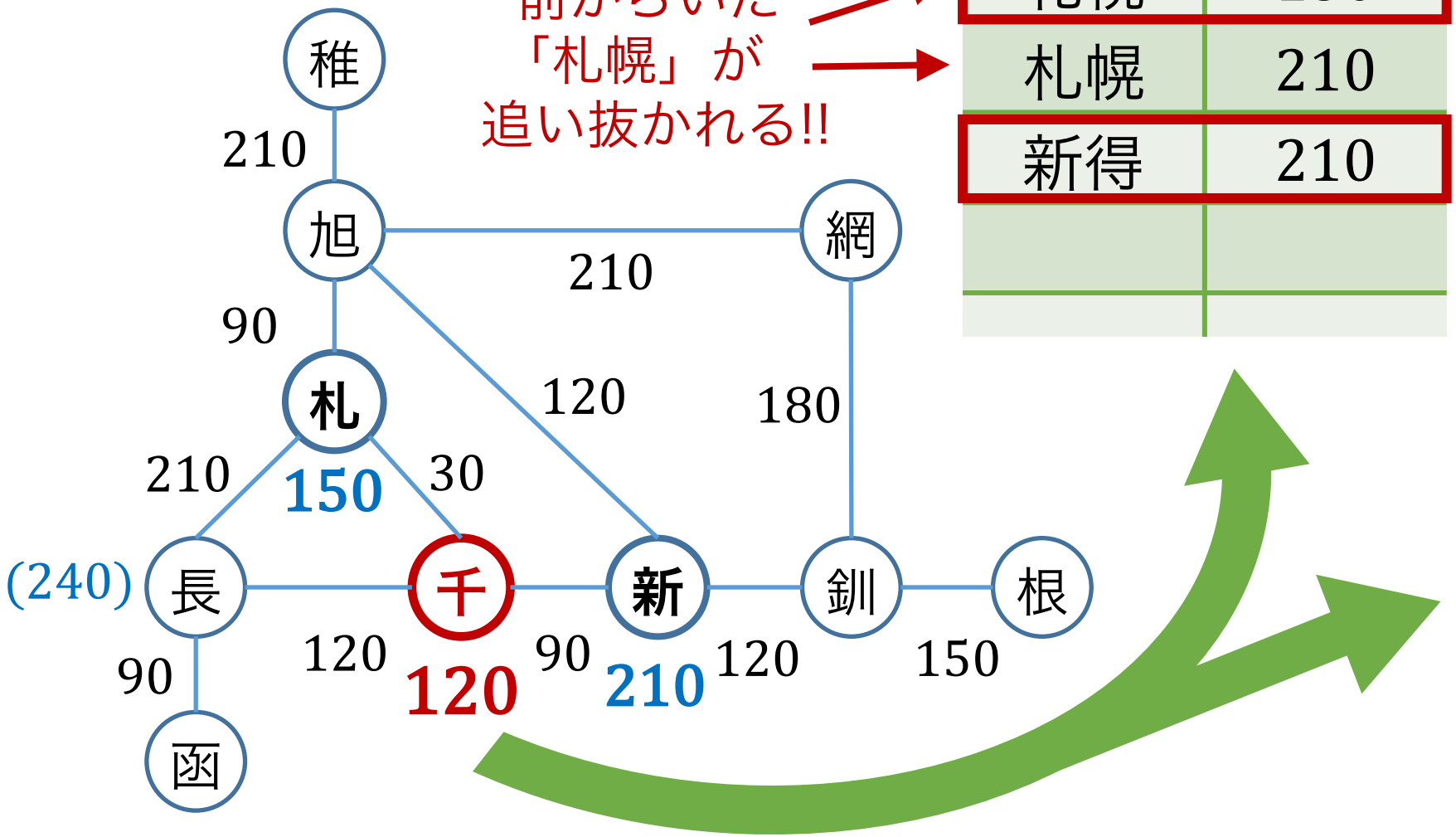
待機列 (昇順)

札幌	210

結果列

長万部	0
函館	90
札幌	210
旭川	INF
稚内	INF
千歳	120
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



前からいた
「札幌」が
追い抜かれる!!

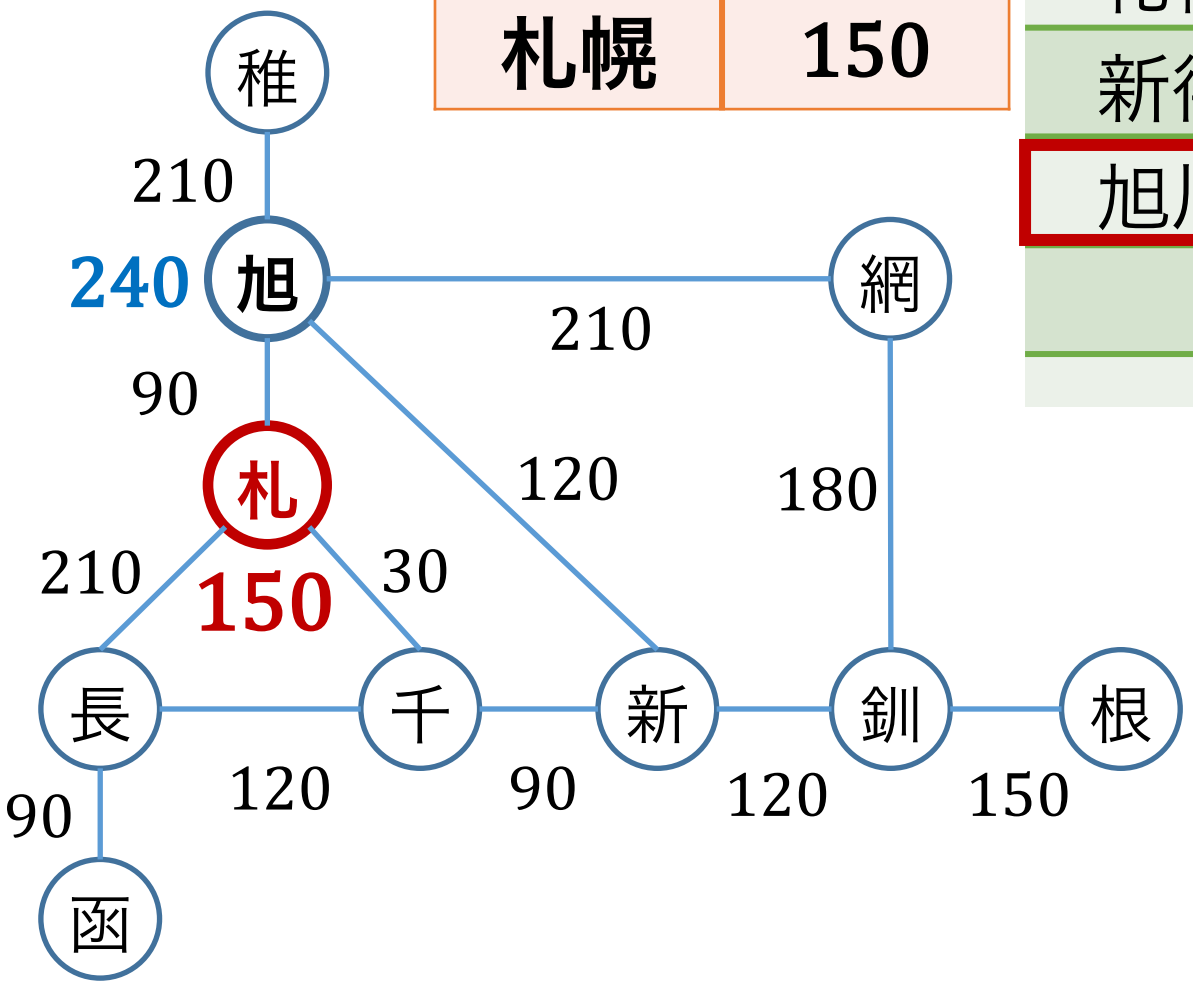
待機列 (昇順)

札幌	150
札幌	210
新得	210

結果列

長万部	0
函館	90
札幌	150
旭川	INF
稚内	INF
千歳	120
新得	210
釧路	INF
根室	INF
網走	INF

コストの更新



札幌	150

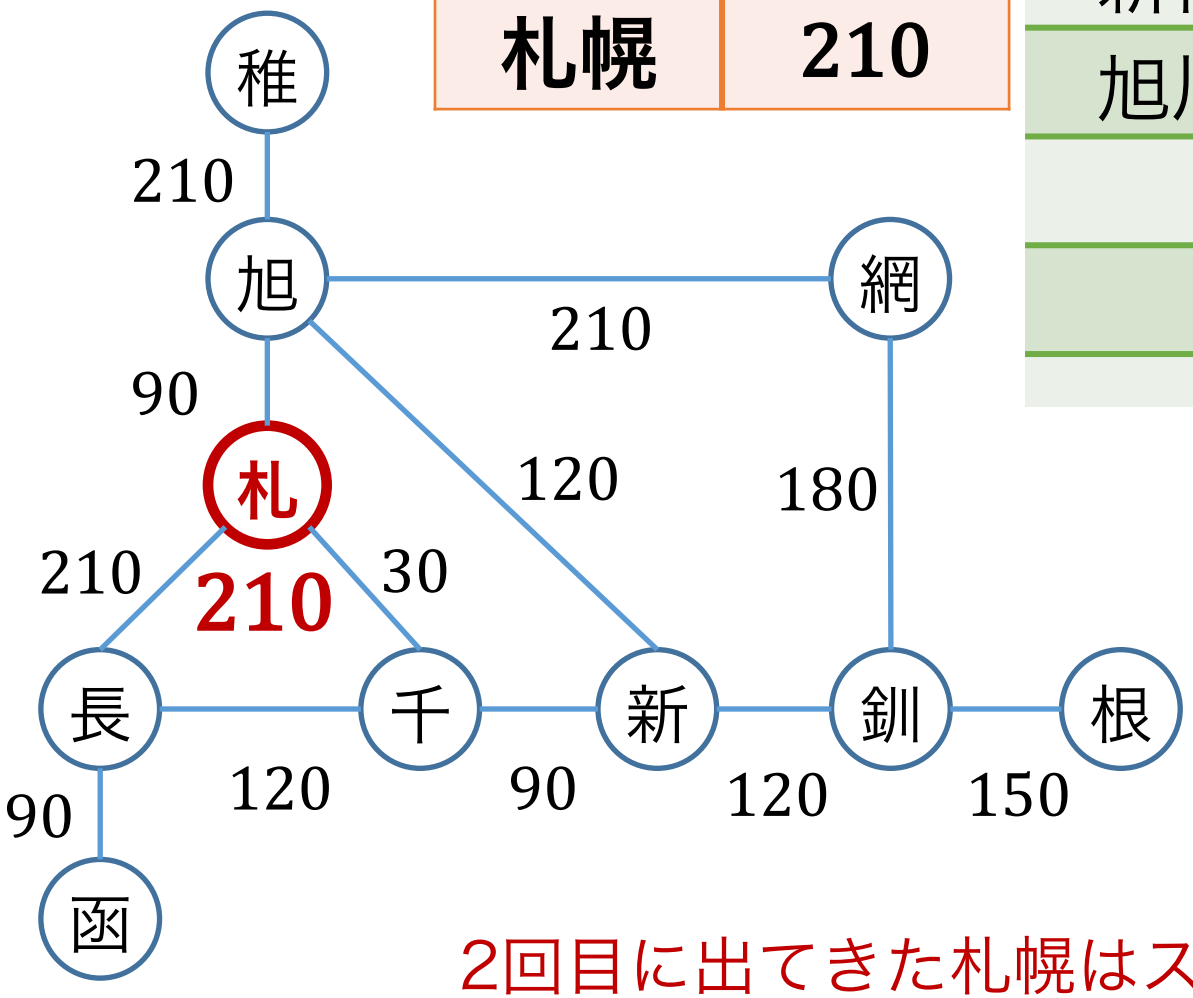
待機列 (昇順)

札幌	210
新得	210
旭川	240

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	INF
千歳	120
新得	210
釧路	INF
根室	INF
網走	INF

コストの更新



札幌	210

待機列 (昇順)

新得	210
旭川	240

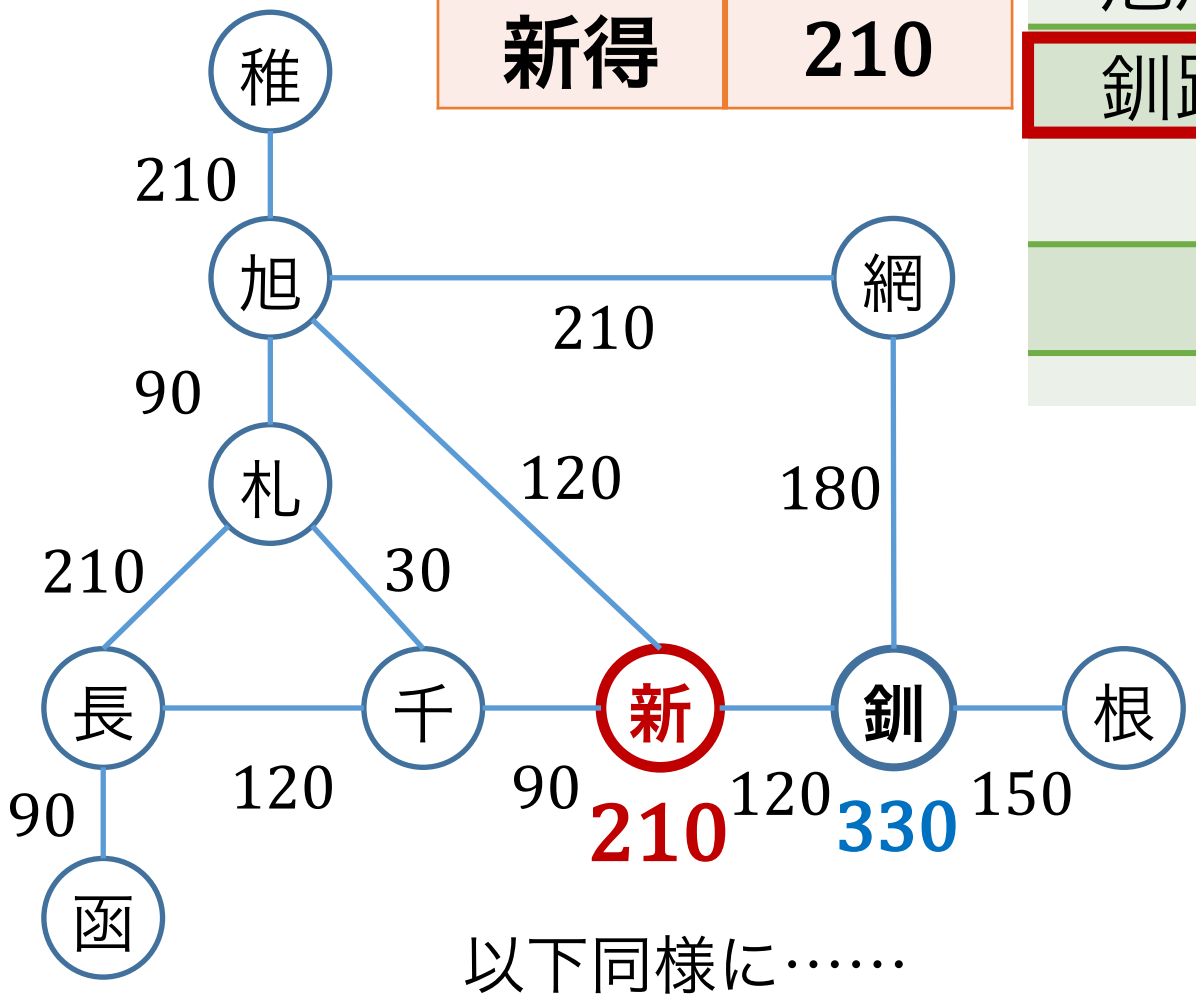
結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	INF
千歳	120
新得	210
釧路	INF
根室	INF
網走	INF

2回目に出てきた札幌はスルー

コストの更新

新得	210



以下同様に……

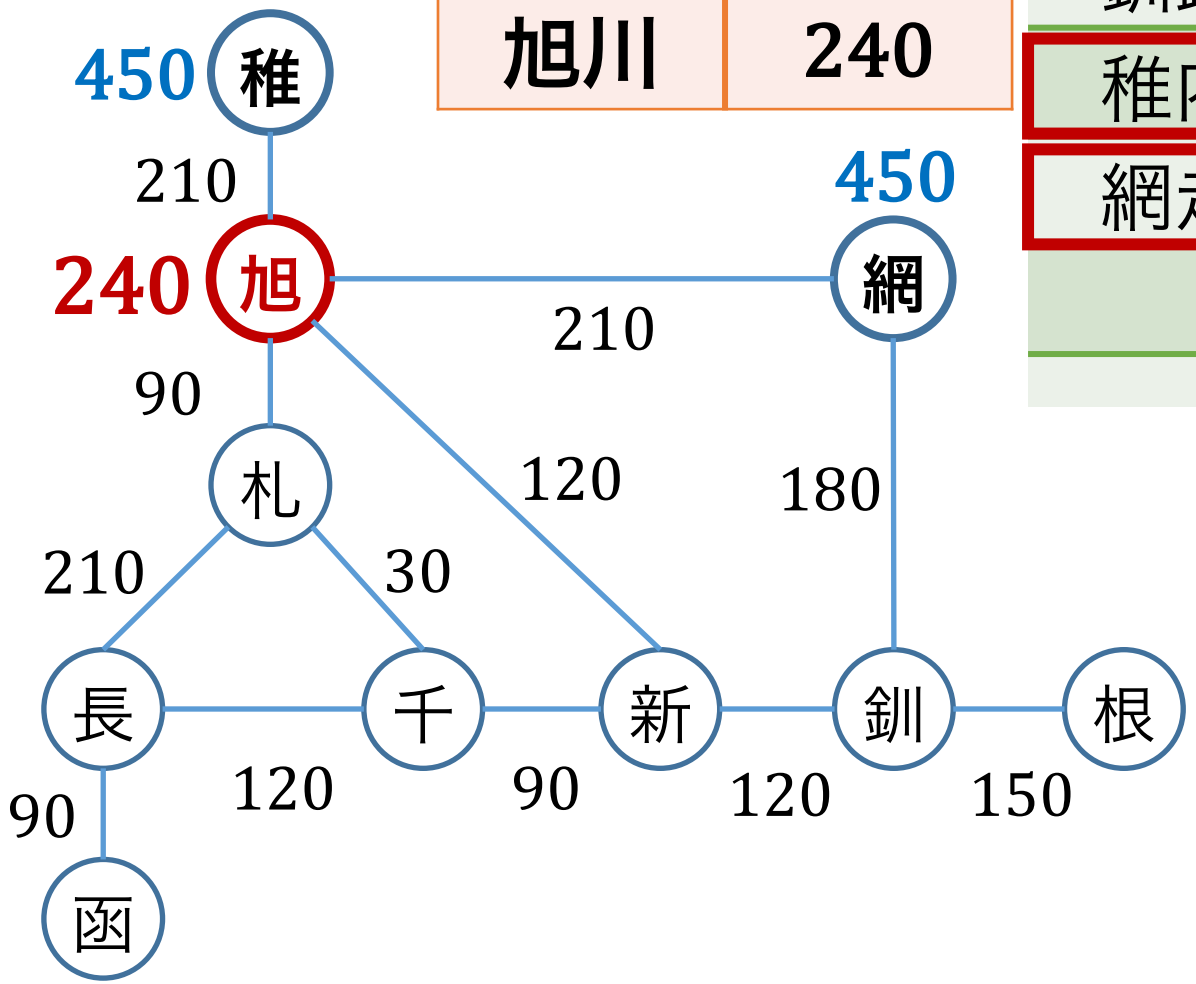
待機列 (昇順)

旭川	240
釧路	330

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	INF
千歳	120
新得	210
釧路	330
根室	INF
網走	INF

コストの更新



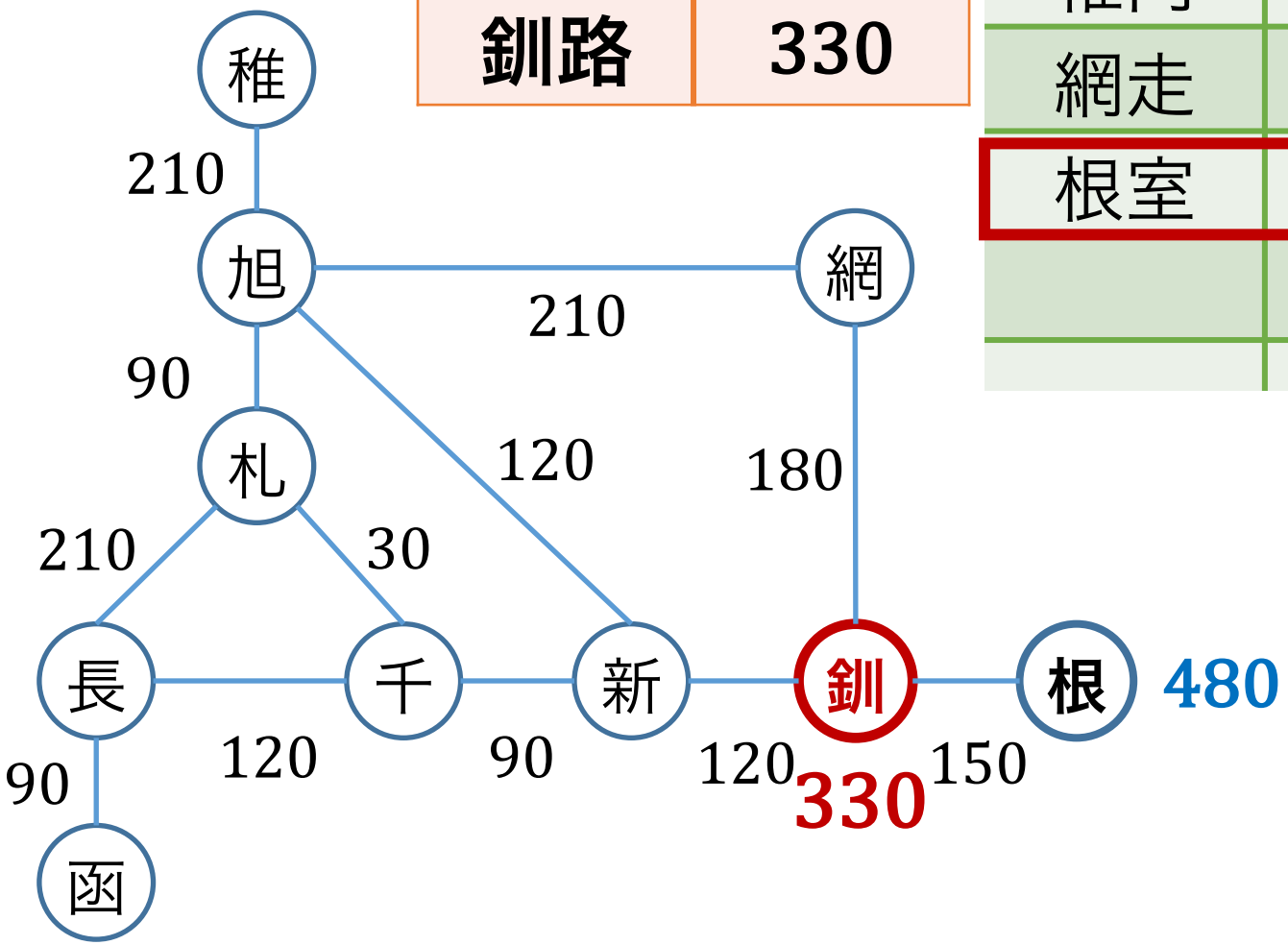
待機列 (昇順)

釧路	330
稚内	450
網走	450

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	450
千歳	120
新得	210
釧路	330
根室	INF
網走	450

コストの更新



釧路	330

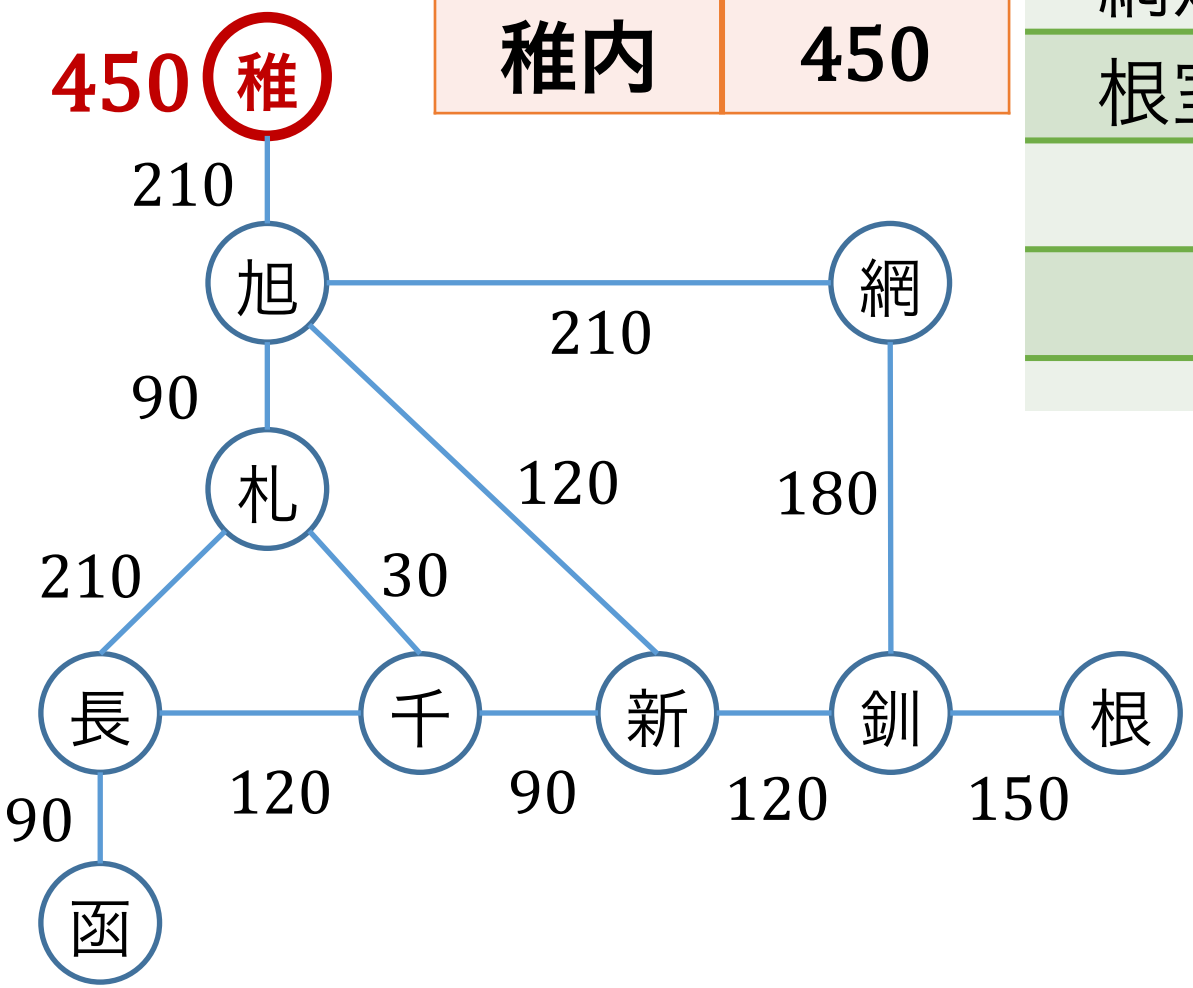
待機列 (昇順)

稚内	450
網走	450
根室	480

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	450
千歳	120
新得	210
釧路	330
根室	480
網走	450

コストの更新



稚内	450

待機列 (昇順)

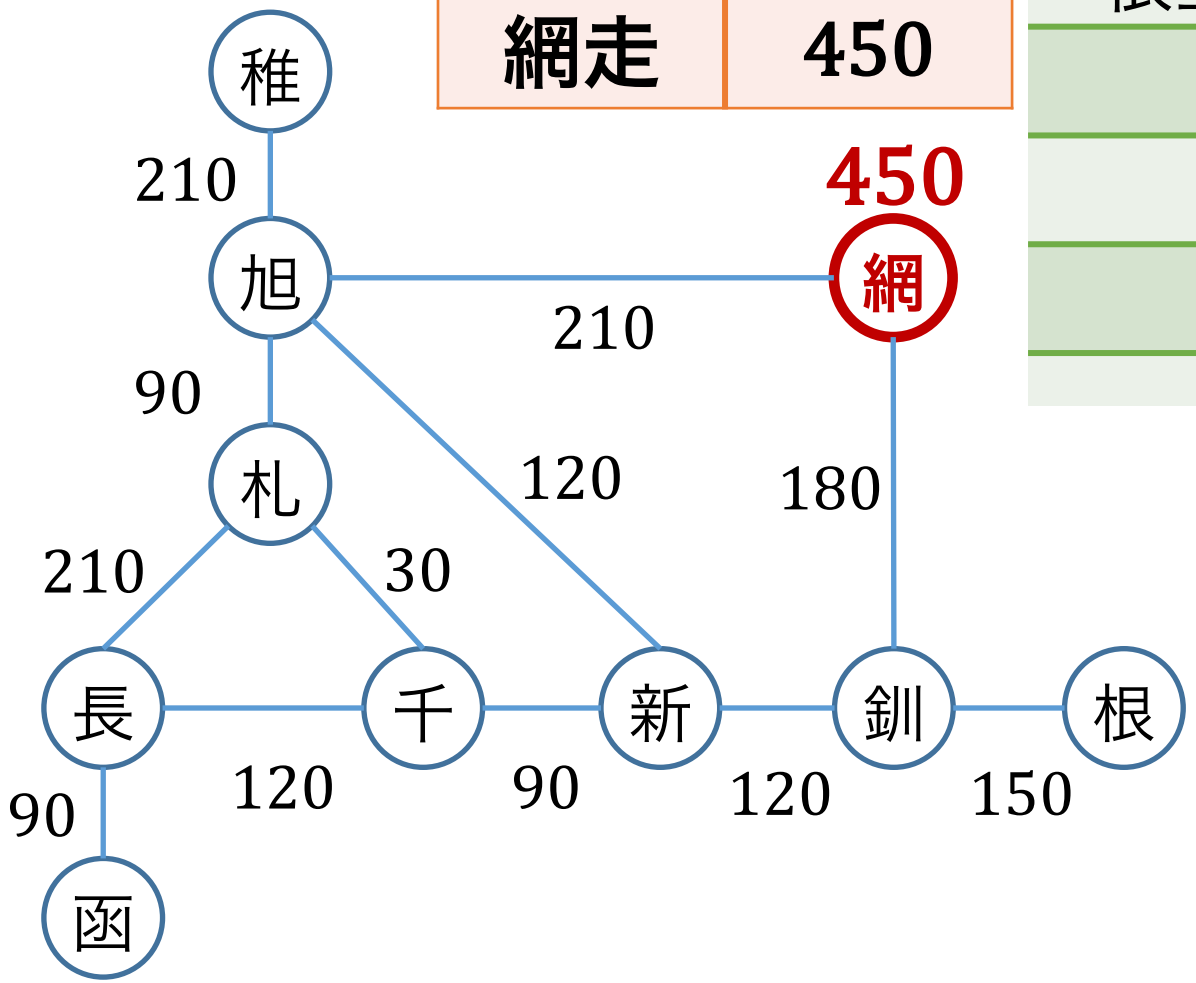
網走	450
根室	480

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	450
千歳	120
新得	210
釧路	330
根室	480
網走	450

コストの更新

網走	450



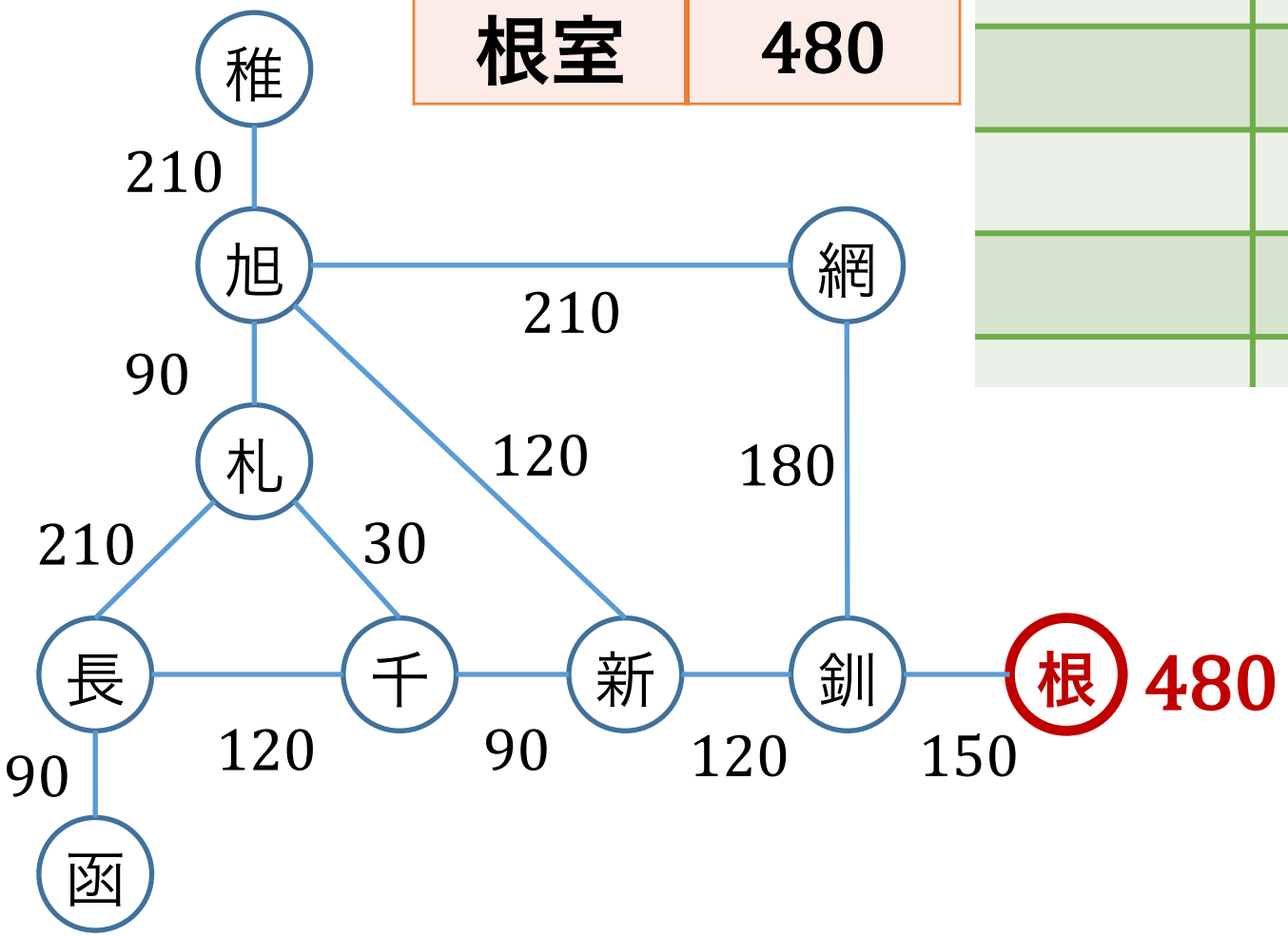
待機列 (昇順)

根室	480

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	450
千歳	120
新得	210
釧路	330
根室	480
網走	450

コストの更新



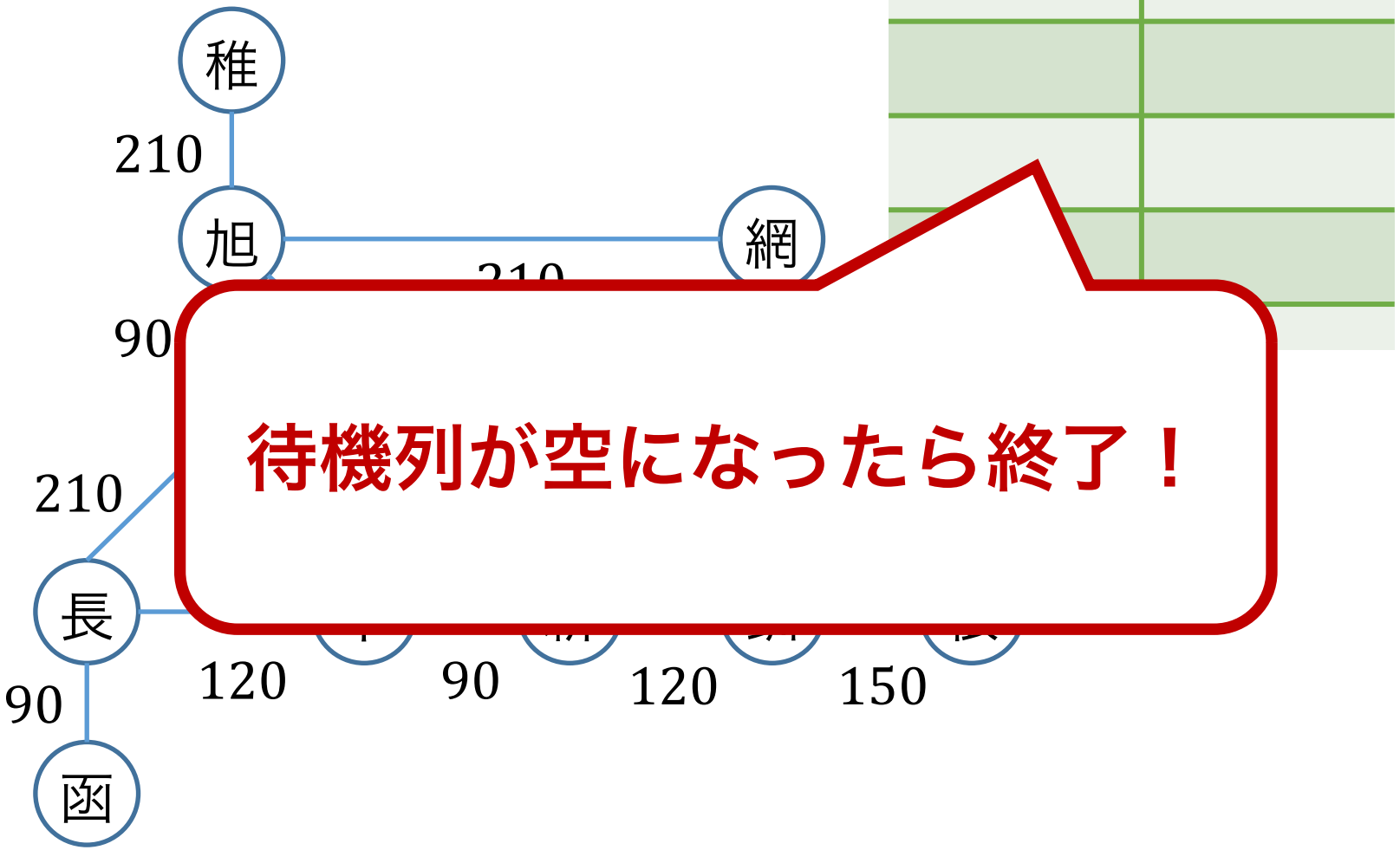
根室	480

待機列 (昇順)

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	450
千歳	120
新得	210
釧路	330
根室	480
網走	450

完了！



待機列が空になったら終了！

待機列 (昇順)

結果列

長万部	0
函館	90
札幌	150
旭川	240
稚内	450
千歳	120
新得	210
釧路	330
根室	480
網走	450

ダイクストラ法の解析

- 待機列から出てきた頂点の値が、後から書き換わることはない
  つまり、**最小コストが確定した頂点だけが出てくる！**

ダイクストラ法の解析

- 待機列から出てきた頂点の値が、後から書き換わることはない
  つまり、**最小コストが確定した頂点だけが出てくる！**

- 待機列に追加される頂点数:
  待機列から初めて出てきた頂点と隣接する頂点のみ
  つまり、**すべての枝をなめている**
  → $O(E)$ **個**

ダイクストラ法の解析

- 待機列から出てきた頂点の値が、後から書き換わることはない
  つまり、**最小コストが確定した頂点だけが出てくる！**

- 待機列に追加される頂点数:
  待機列から初めて出てきた頂点と隣接する頂点のみ
  つまり、**すべての枝をなめている**
  → O(E) 個

- **じゃぁ、待機列への操作ってどれくらい時間かかるの？**
  というか、最小値を取り出すってどうすればいいの？
  例えば、追加のたびにsortを実行していたら遅すぎる

優先度つきキュー (Priority Queue)

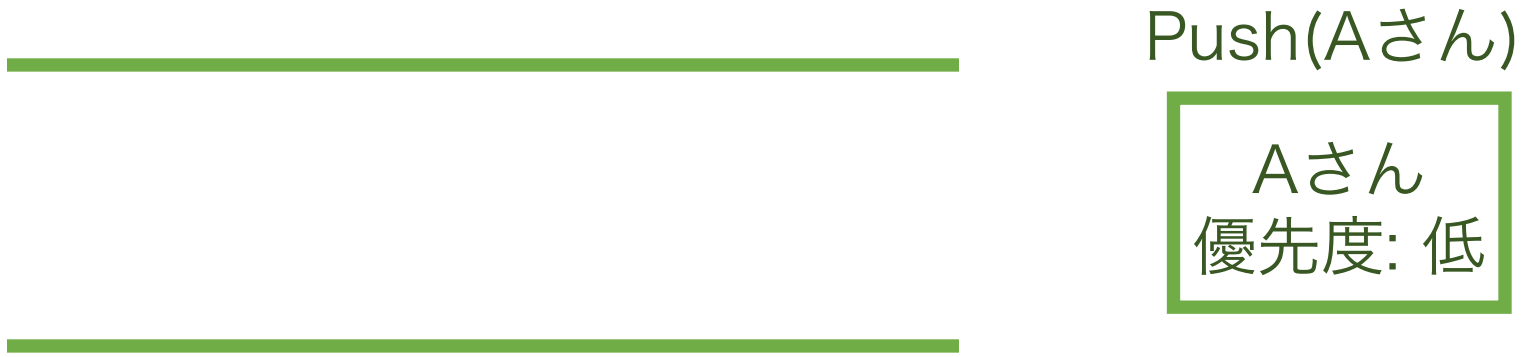
- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる

優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる



優先度つきキュー (Priority Queue)

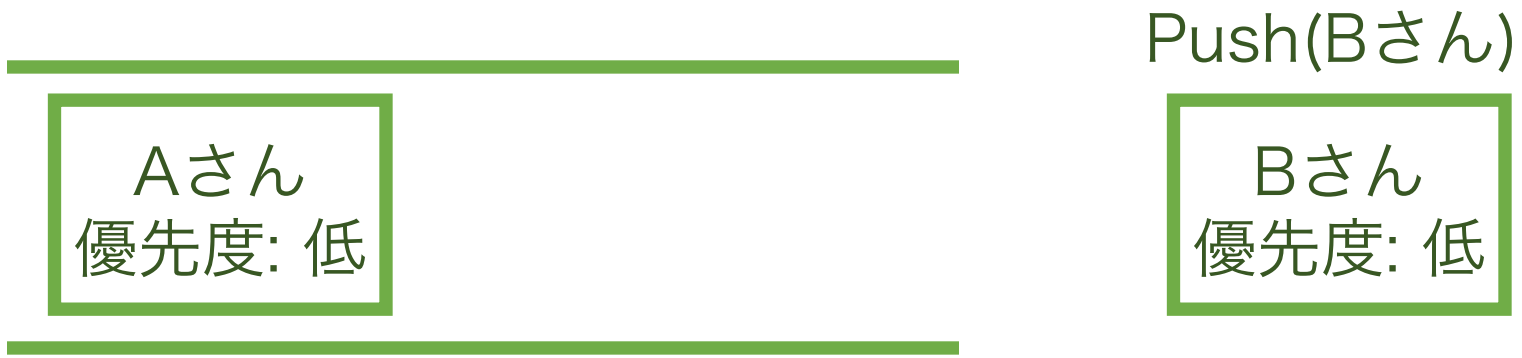
- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる

優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる



優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる

優先度つきキュー (Priority Queue)

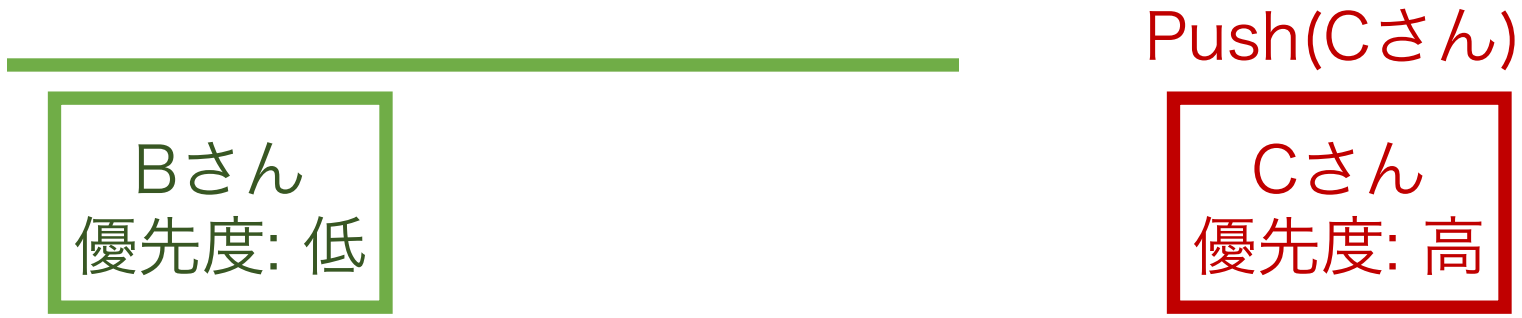
- ここで着目するのが、**優先度つきキュー**というデータ構造

これは、横入りを許可したキューで、
優先度が高いものから順番に出てくる

優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる



優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

  これは、横入りを許可したキューで、
  優先度が高いものから順番に出てくる

横入り発生

優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

これは、横入りを許可したキューで、
優先度が高いものから順番に出てくる

優先度つきキュー (Priority Queue)

- ここで着目するのが、**優先度つきキュー**というデータ構造

これは、横入りを許可したキューで、
優先度が高いものから順番に出てくる

- つまり、First In, First Out とは限らない
  ~~「それってキューなの？」というのは禁句~~

優先度つきキューの実装方法

Q. どうやって実装するの？

優先度つきキューの実装方法

Q. どうやって実装するの？　A. もうあるので、それを使おう！

	C++での書き方	計算量
ライブラリ	#include <queue>	
宣言	priority_queue<型> pq;	
要素の挿入	pq.push(x);	O(log n) 時間
先頭を見る	pq.top();	O(1) 時間
先頭を削除	pq.pop();	O(log n) 時間
空か確認	pq.empty();	O(1) 時間

(n: pqの要素数)

ダイクストラ法の解析に戻ると……

- 待機列に追加される頂点数:
  待機列から出てきた頂点と隣接する頂点のみ
  (同じ頂点が複数回出てきた場合、2回目からはスルーする)
  つまり、**すべての枝をなめている**
  → $O(E)$ **個**

- **じゃぁ、待機列への操作ってどれくらい時間かかるの？**

ダイクストラ法の解析に戻ると……

- 待機列に追加される頂点数:
  待機列から出てきた頂点と隣接する頂点のみ
  (同じ頂点が複数回出てきた場合、2回目からはスルーする)
  つまり、**すべての枝をなめている**
  → $O(E)$ 個

- **じゃぁ、待機列への操作ってどれくらい時間かかるの？**
  → **挿入・削除ともに $O(\log E)$ 時間**

全体として $O(E \log E)$ 時間

優先度つきキュー使用上の注意

- C++の優先度つきキューは、
  数値が大きいものから順に出てくる（やりたいことと逆）

優先度つきキュー使用上の注意

- C++の優先度つきキューは、
  数値が大きいものから順に出てくる（やりたいことと逆）

  → 正負を逆転してから挿入しよう！

- JavaやPythonの優先度つきキューは、
  デフォルトだと小さい順に出てくるらしい

実装例

```
// ダイクストラ
// s: 開始ノード, v: 頂点数, adjlist: 隣接リスト (first: コスト second: 行き先)
vector<int> dijk(int s, int v, vector<vector<pair<int, int> > > adjlist){
  priority_queue<pair<int, int > > wait;
  vector<int> result(v, 1e9);

  // スタート地点を追加
  result[s] = 0;
  wait.push(make_pair(0, s));

  // ダイクストラ本体
  while (!wait.empty()) {
    // 動く
    int nowpoint = wait.top().second;
    int nowcost = -wait.top().first;
    wait.pop();
    // 今のコストが探索済みのコストより大きい場合、読み飛ばし
    if (result[nowpoint] < nowcost) { continue; }

    // 今いる頂点と隣接しているすべての頂点をなめる
    for (int i = 0; i ≤ adjlist[nowpoint].size(); i++) {
      int nextpoint = adjlist[nowpoint][i].second;
      int nextcost = nowcost + adjlist[nowpoint][i].first;
      // 探索結果より安く到達できそうであれば、結果を更新して優先度つきキューに格納
      if(result[nextpoint] > nextcost){
        result[nextpoint] = nextcost;
        wait.push(make_pair(-nextcost, nextpoint));
      }
    }
  }

  return result; // 返り値: 結果列
}
```

ベルマンフォード vs ダイクストラ

- **基本的にはダイクストラ法のほうが良い**

ベルマンフォード法: $O(VE)$ 時間
ダイクストラ法: $O(E \log E)$ 時間　← **速い！**

ベルマンフォード vs ダイクストラ

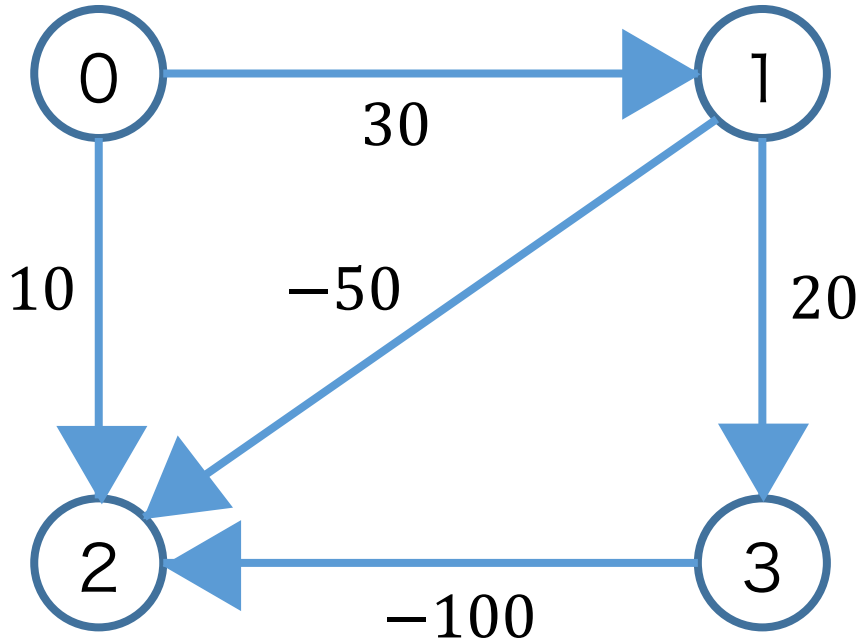
- **基本的にはダイクストラ法のほうが良い**

  ベルマンフォード法: $O(VE)$ 時間
  ダイクストラ法: $O(E \log E)$ 時間　← **速い！**

- **しかし、グラフ中に負のコストが存在した瞬間、ダイクストラの栄光は崩れ去る**

  - 実は今までの議論は、
    「コストがすべて正である」という仮定のもと行われていた

負のコストがあるダイクストラ



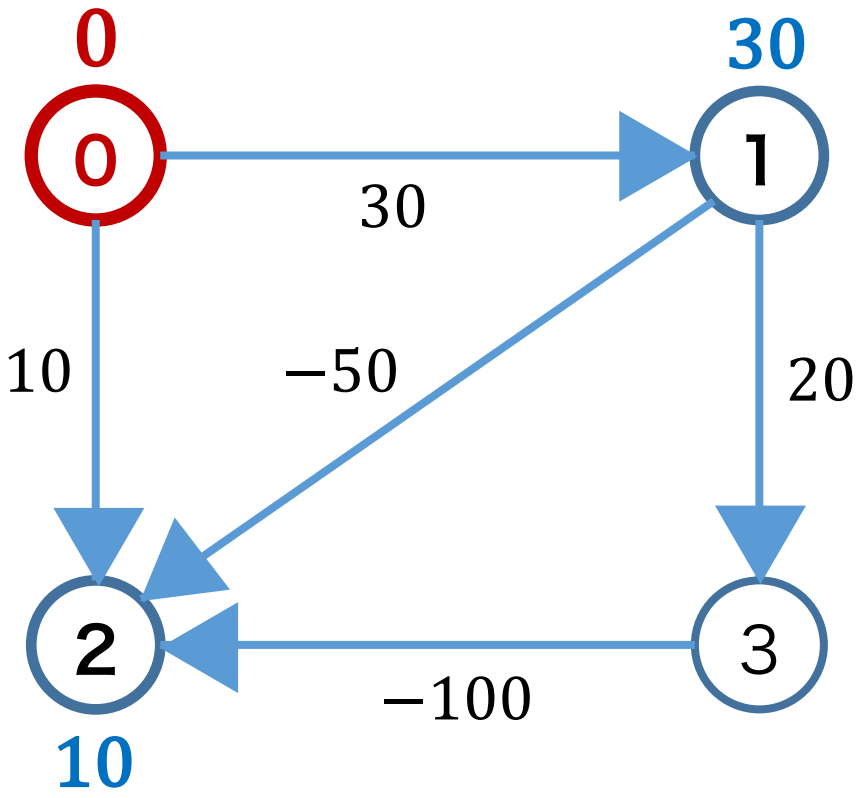
待機列 (昇順)

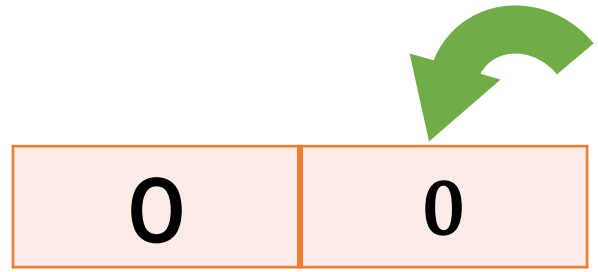
0	0

結果列

0	0
1	INF
2	INF
3	INF

負のコストがあるダイクストラ





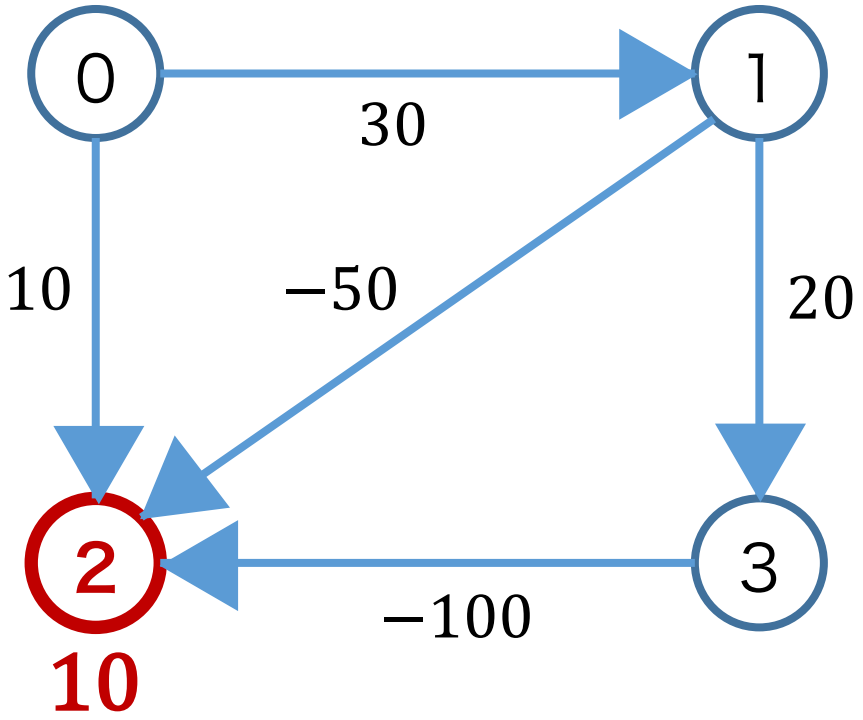
待機列 (昇順)

2	10
1	30

結果列

0	0
1	30
2	10
3	INF

負のコストがあるダイクストラ



待機列 (昇順)

2	10

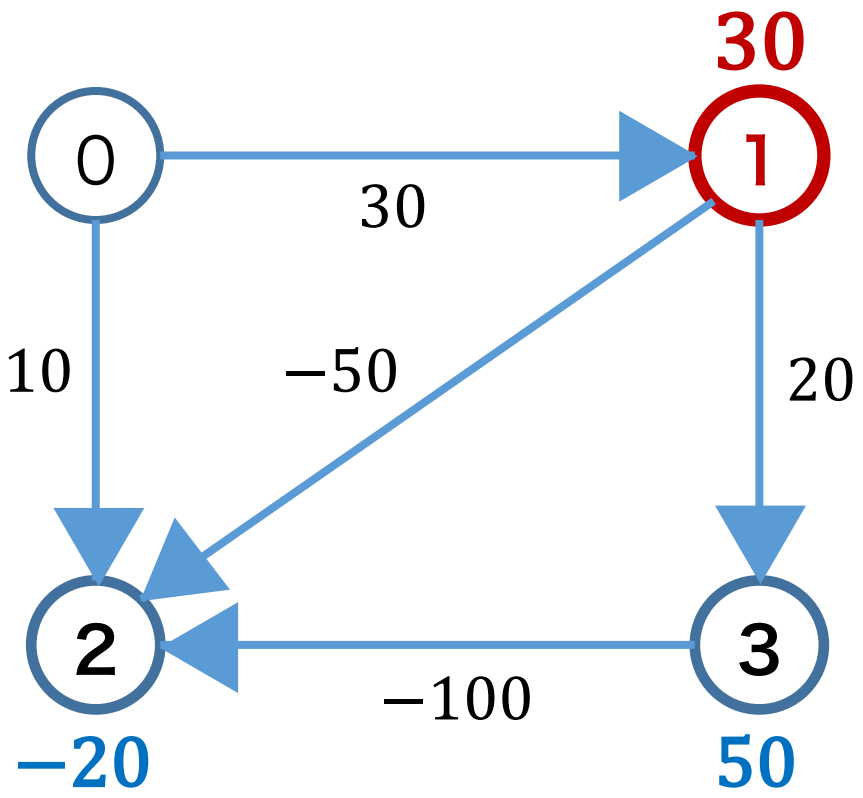
1	30

結果列

0	0
1	30
2	10
3	INF

頂点2がpopされる

負のコストがあるダイクストラ



待機列 (昇順)

1	30

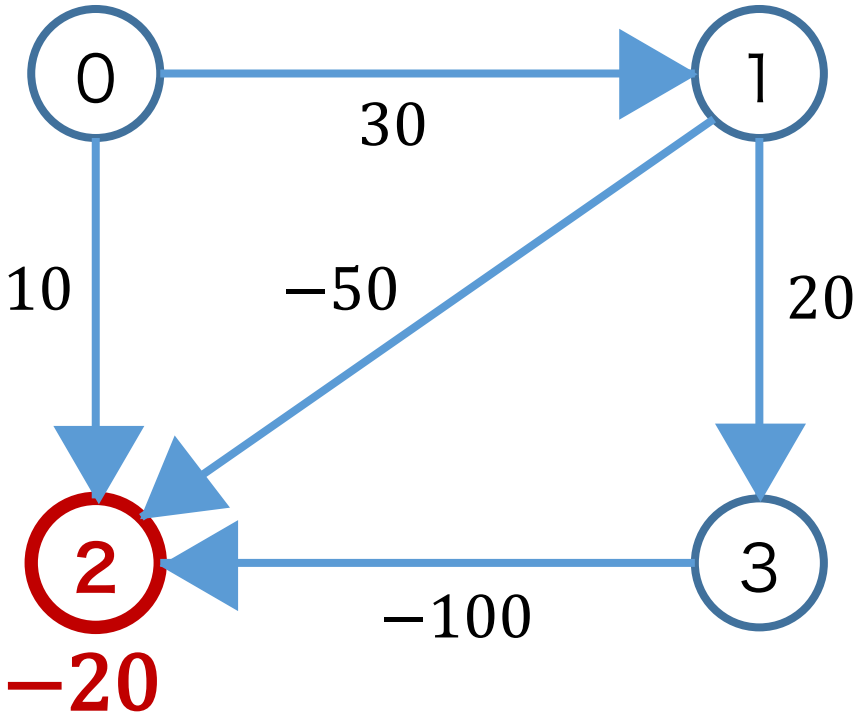
2	−20
3	50

結果列

0	0
1	30
2	−20
3	50

…… にも関わらず、
再び頂点2が更新される！

負のコストがあるダイクストラ



2	−20

待機列 (昇順)

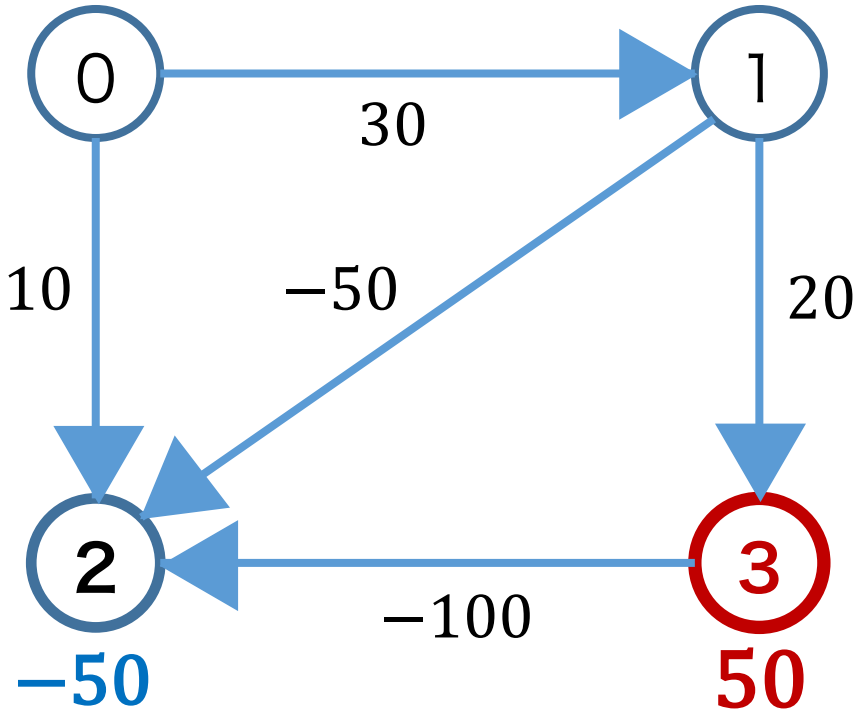
3	50

結果列

0	0
1	30
2	−20
3	50

そして頂点2がpopされる

負のコストがあるダイクストラ



3	50

待機列 (昇順)

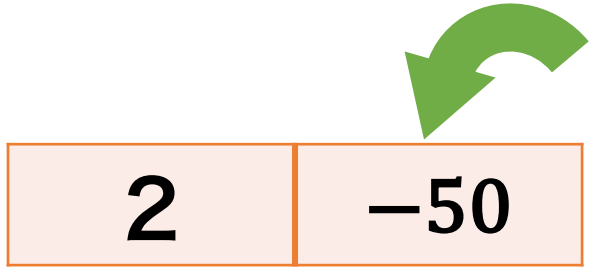
2	−50

結果列

0	0
1	30
2	−50
3	50

…… にも関わらず (ry

負のコストがあるダイクストラ

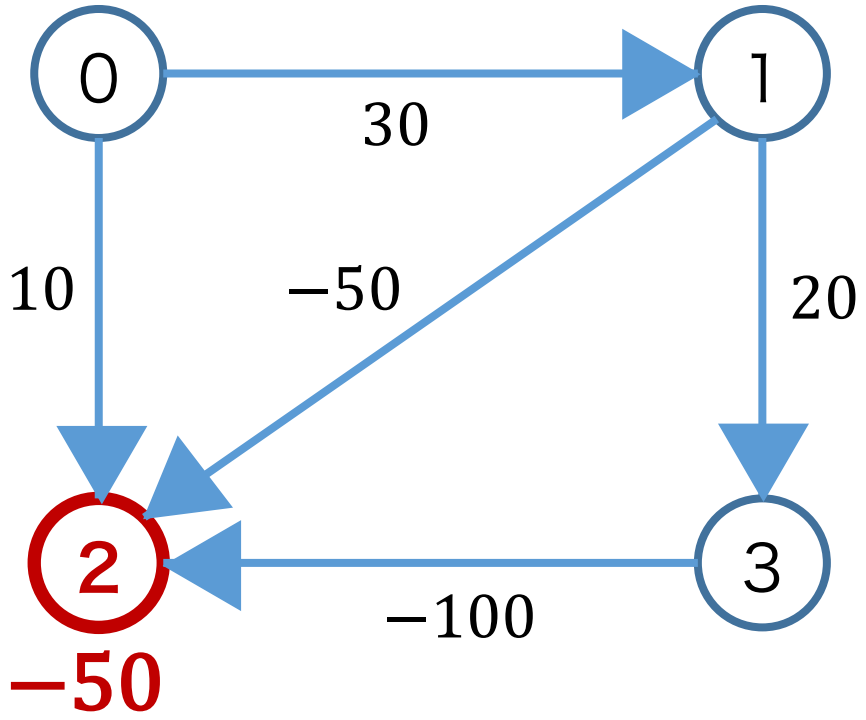


2	−50

待機列 (昇順)

結果列

0	0
1	30
2	−50
3	50



と、こんな感じに良くない

ベルマンフォード vs ダイクストラ

- 負のコストが存在する場合のダイクストラは、
  「無駄な更新を省く」が機能しなくなるので、
  効率はベルマンフォートと同じくらいに落ちる
  → **優先度つきキューを使っている分だけ遅い**

- 負の閉路がある場合、ダイクストラは無限ループに陥る

ベルマンフォード vs ダイクストラ

- 負のコストが存在する場合のダイクストラは、
  「無駄な更新を省く」が機能しなくなるので、
  効率はベルマンフォートと同じくらいに落ちる
  → **優先度つきキューを使っている分だけ遅い**

- 負の閉路がある場合、ダイクストラは無限ループに陥る

- ベルマンフォードは、負の閉路が存在しなければ
  「最大V回ループすれば終わる」という性質があるので、
  ループ回数をカウントするだけで
  負の閉路を検出することができる

ベルマンフォード vs ダイクストラ

- 負のコストが存在する場合のダイクストラは、
  「無駄な更新を省く」が機能しなくなるので、
  効率

- 負

- ベ
  「最大V回ループすれば終わる」という性質があるので、
  ループ回数をカウントするだけで
  負の閉路を検出することができる

負のコストがあるならベルマンフォードを、
コストがすべて正ならダイクストラを使おう！

ワーシャルフロイド法 (Warshall–Floyd algorithm)

- 今までの2つと違い、**全点対**の最短経路が求まる

dp	長	函	札	旭	稚	千	新	釧	根	網
長										

↑ベルマンフォードやダイクストラでは、こうだった

ワーシャルフロイド法 (Warshall–Floyd algorithm)

- 今までの2つと違い、**全点対**の最短経路が求まる

dp	長	函	札	旭	稚	千	新	釧	根	網
長										
函										
札										
旭										
稚										
千										
新										
釧										
根										
網										

ワーシャルフロイドでは、こうなる

コストの更新

- まず、隣接行列を作る

(空欄はINF)

dp	長	函	札	旭	稚	千	新	釧	根	網
長		90	210			120				
函	90									
札	210			90		30				
旭			90		210		120			210
稚				210						
千	120		30				90			
新				120		90		120		
釧							120		150	180
根								150		
網				210				180		

コストの更新

- 次に、自分自身へのコスト (対角成分) を0にする

(空欄はINF)

dp	長	函	札	旭	稚	千	新	釧	根	網
長	0	90	210			120				
函	90	0								
札	210		0	90		30				
旭			90	0	210		120			210
稚				210	0					
千	120		30			0	90			
新				120		90	0	120		
釧							120	0	150	180
根								150	0	
網				210				180		0

コストの更新

- これに魔法をかける

(空欄はINF)

dp	長	函	札	旭	稚	千	新	釧	根	網
長	0	90	210			120				
函	90	0								
札	210		0	90		30				
旭			90	0	210		120			210
稚				210	0					
千	120	30				0	90			
新				120	90	90	0	120		
釧							120	0	150	180
根								150	0	
網				210				180		0

```
for(int k = 0; k < V; k++)
  for(int i = 0; i < V; i++)
    for(int j = 0; j < V; j++)
      dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
```

完了！

- ね、簡単でしょ？

dp	長	函	札	旭	稚	千	新	釧	根	網
長	0	90	150	240	450	120	210	330	480	450
函	90	0	240	330	540	210	300	420	570	540
札	150	240	0	90	300	30	120	240	390	300
旭	240	330	90	0	210	120	120	240	390	300
稚	450	540	300	210	0	330	330	450	600	420
千	120	210	30	120	330	0	90	210	360	330
新	210	300	120	120	330	90	0	120	270	300
釧	330	420	240	240	450	210	120	0	150	180
根	480	570	390	390	600	360	270	150	0	300
網	450	540	300	210	420	330	300	180	330	0

ワーシャルフロイド法の解析

```
for(int k = 0; k < V; k++)
  for(int i = 0; i < V; i++)
    for(int j = 0; j < V; j++)
      dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
```

書くのが楽！

- 動きのイメージ
  「i から j に行きたいんだけど、
  k を経由するのとしないの、どっちが早い？」

- 計算量は自明に $O(V^3)$ 時間

負のコストがあるワーシャルフロイド

```
for(int k = 0; k < V; k++)
    for(int i = 0; i < V; i++)
        for(int j = 0; j < V; j++)
            if(dp[i][k] != INF && dp[k][j] != INF)
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
```

- **ワーシャルフロイド法は、負のコストがあっても問題なく動く**
  けど、その場合は上記のif文を入れるようにしよう

- 負の閉路があった場合、対角成分に負の値が出てくるので、
  それを監視することで検出可能

余談

- ダイクストラは、英文字で書くとDijkstra

余談

- ダイクストラは、英文字で書くとDijkstra

まるで3重forループみたいな名前してるね！

余談

- ダイクストラは、英文字で書くとDijkstra

まるで3重forループみたいな名前してるね！

でも、ベルマンフォード法・ワーシャルフロイド法と異なり、

ダイクストラ法だけは3重ループをしないぞ！

自分でコードを書いて、確認してみよう！

目次

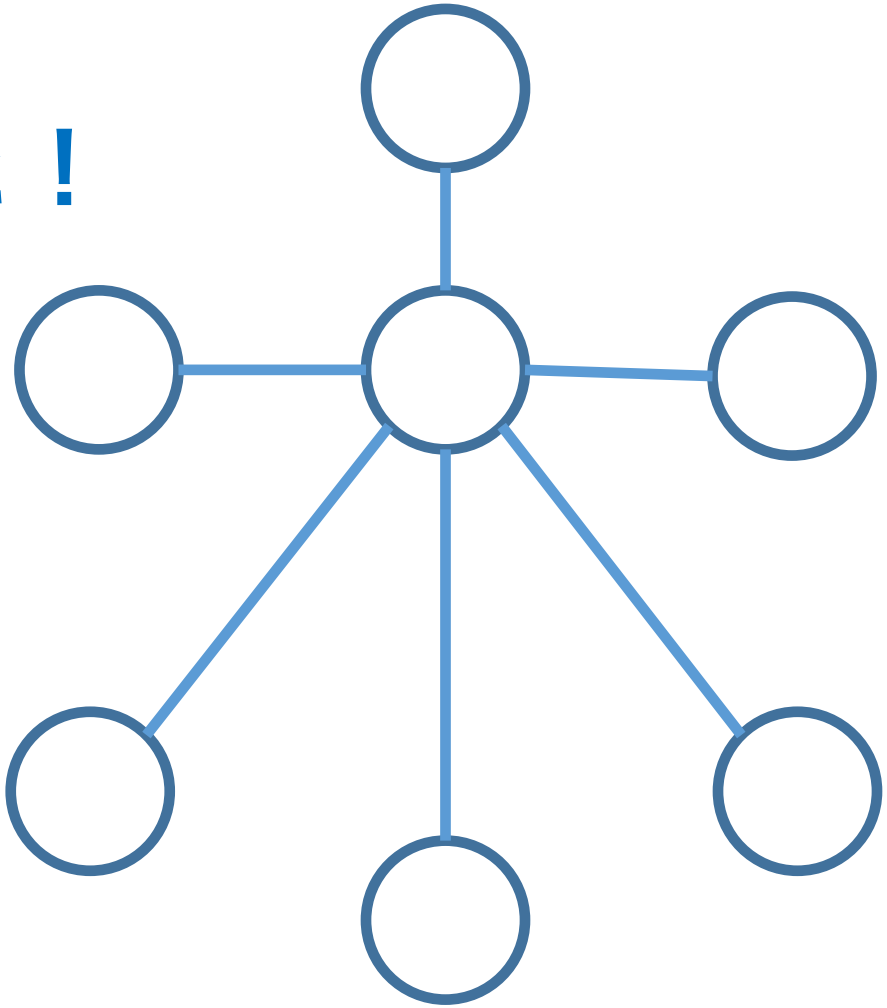
- 最短経路問題 その1
  - 幅優先探索

- 最短経路問題 その2
  - 準備: グラフとは
  - ベルマンフォード法
  - ダイクストラ法
  - ワーシャルフロイド法

- **最小全域木**
  - **準備: 木とは**
  - **プリム法**
  - **クラスカル法**

木とは？

- **閉路がなく連結なグラフのこと！**



  - **閉路:** 同じ辺を2度通らずに
    元の頂点に戻ってこれる路

  - **連結:** 任意の頂点同士が
    互いに行き来できること

    （今回は特に向きはないとする）

- **森:** 閉路がないグラフのこと (非連結でもOK)

木とは？

- **閉路がなく連結なグラフのこと！**

  - **閉路:** 同じ辺を2度通らずに
    元の頂点に戻ってこれる路

  - **連結:** 任意の頂点同士が
    互いに行き来できること

    （今回は特に向きはないとする）

- **森:** 閉路がないグラフのこと (非連結でもOK)

木とは？

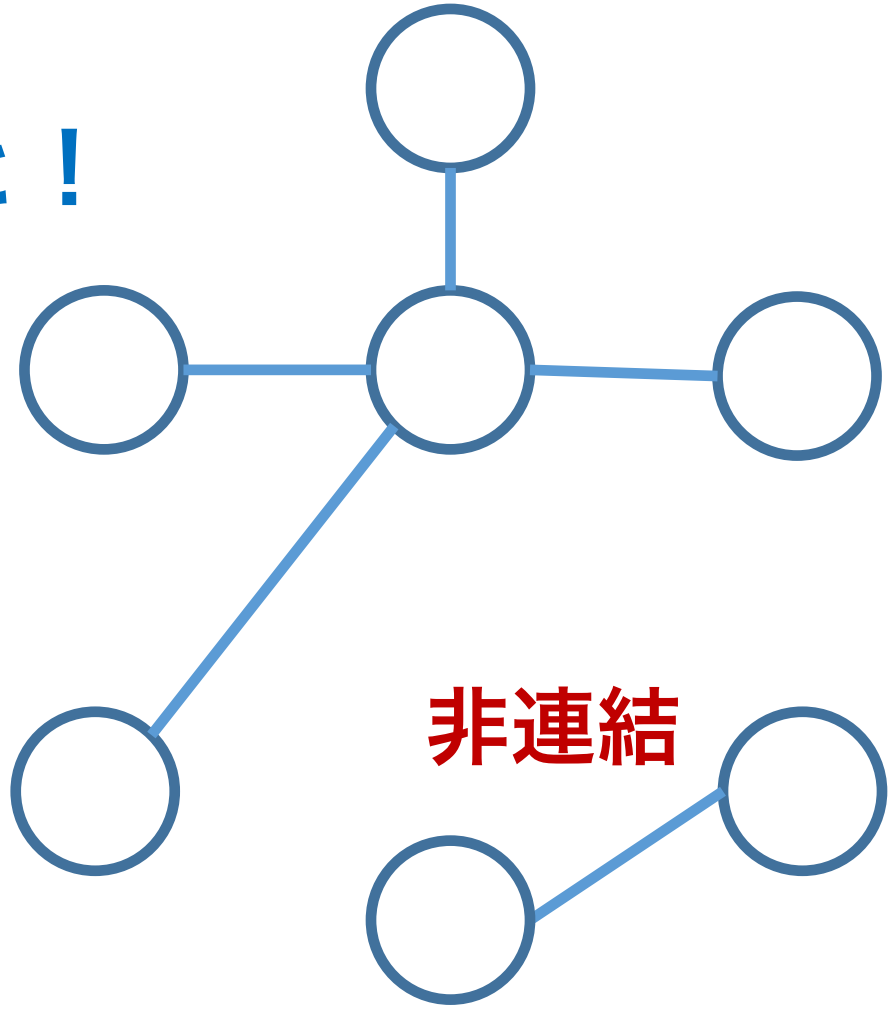
- **閉路がなく連結なグラフのこと！**

  - **閉路:** 同じ辺を2度通らずに
    元の頂点に戻ってこれる路

  - **連結:** 任意の頂点同士が
    互いに行き来できること

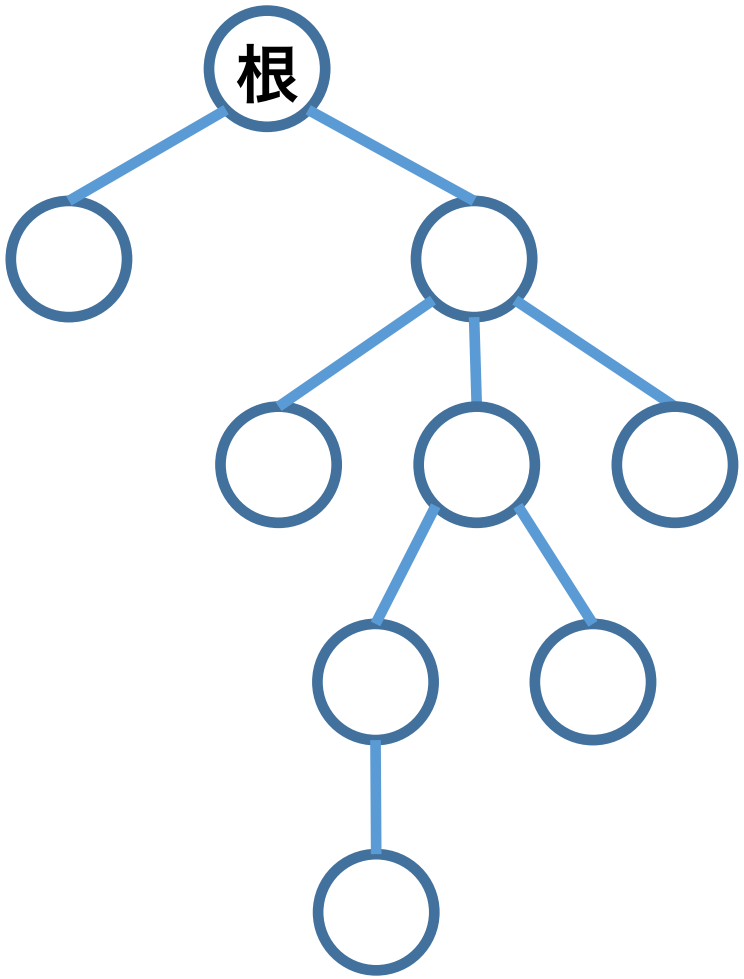
    （今回は特に向きはないとする）

- **森:** 閉路がないグラフのこと (非連結でもOK)



木の用語いろいろ

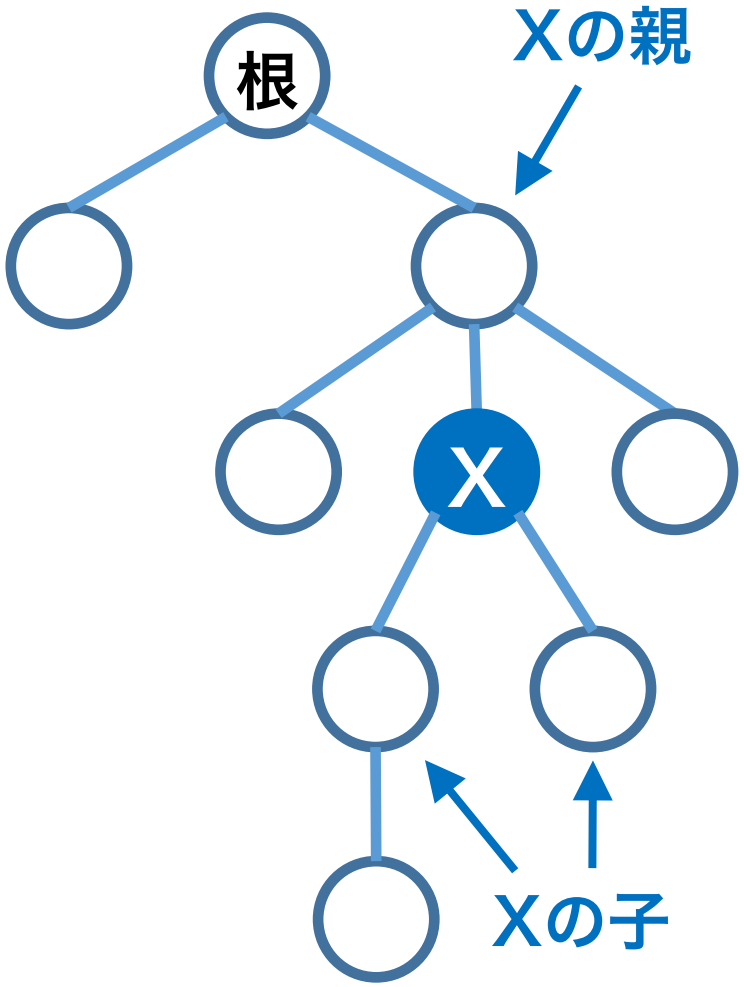
- 適当な頂点を1つ選んで
  「根」と名付けることがある



木の用語いろいろ

- 適当な頂点を1つ選んで
  「根」と名付けることがある

- ある頂点に対して、
  根に近いほうの隣接頂点を**「親」**
  根から遠いほうの隣接頂点を**「子」**と呼ぶ

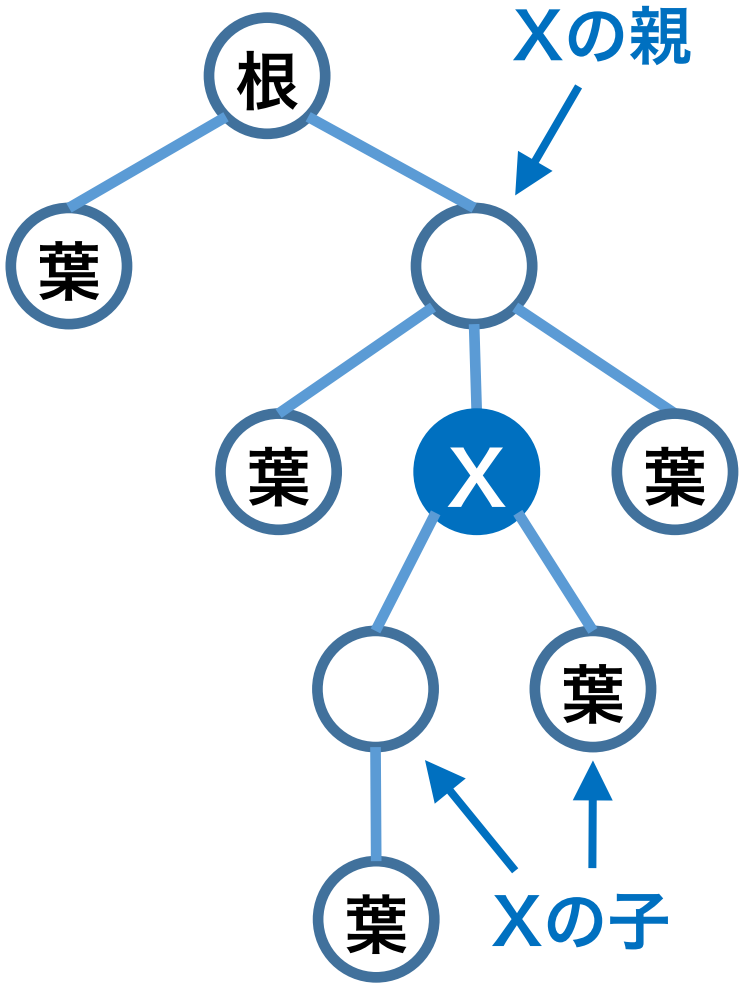


木の用語いろいろ

- 適当な頂点を1つ選んで
  「根」と名付けることがある

- ある頂点に対して、
  根に近いほうの隣接頂点を**「親」**
  根から遠いほうの隣接頂点を**「子」**と呼ぶ

- 子がいない頂点を**「葉」**と呼ぶ



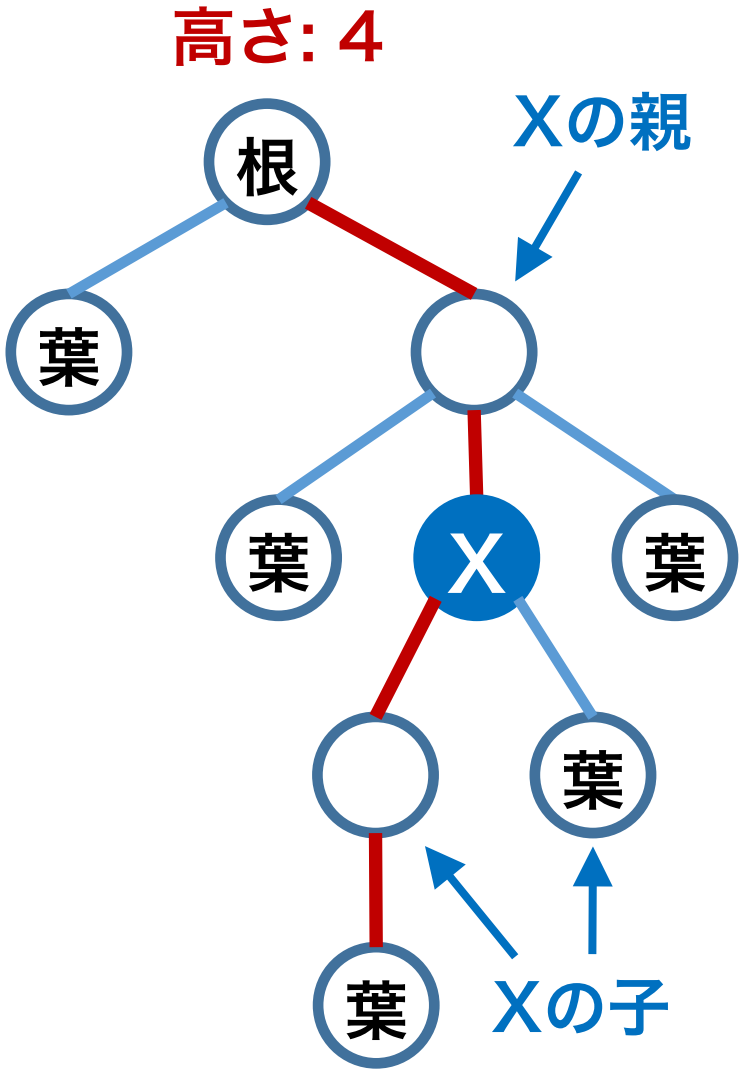
木の用語いろいろ

- 適当な頂点を1つ選んで
  「根」と名付けることがある

- ある頂点に対して、
  根に近いほうの隣接頂点を**「親」**
  根から遠いほうの隣接頂点を**「子」**と呼ぶ

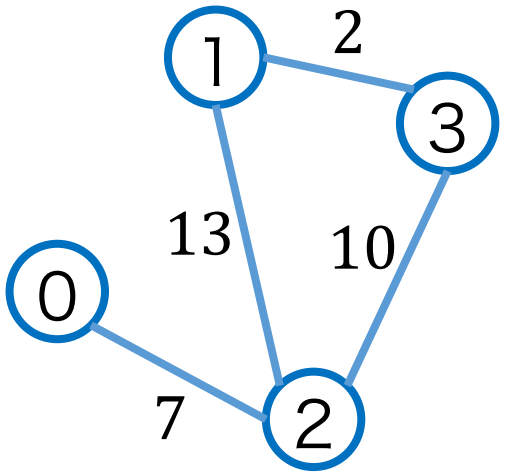
- 子がいない頂点を**「葉」**と呼ぶ

- 根から一番遠い葉までの間にある辺の数を
  「木の高さ」と呼ぶ

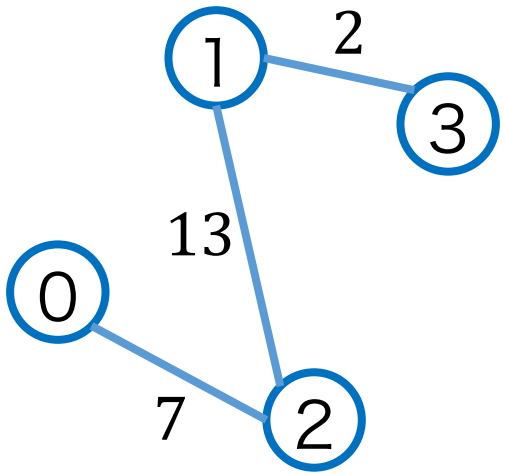


全域木 (Spanning Tree)

- **元のグラフの頂点すべてと、元のグラフの枝の一部 (or 全部) を使って作られた木のこと**

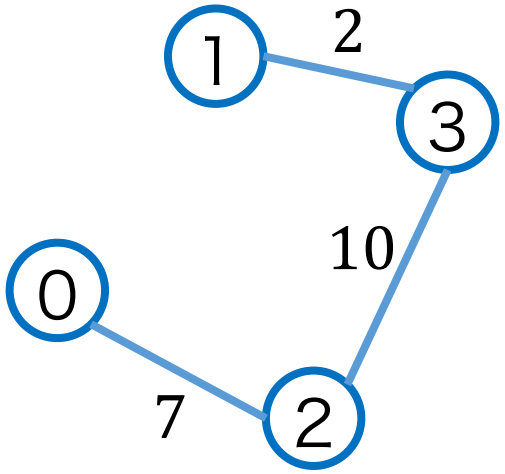


元のグラフ



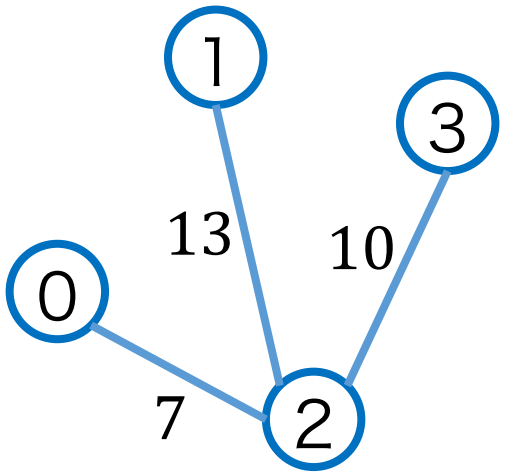
全域木(1)

コスト合計: 22



全域木(2)

コスト合計: 19



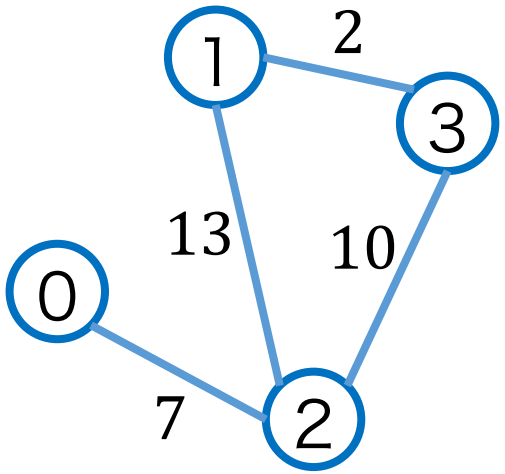
全域木(3)

コスト合計: 30

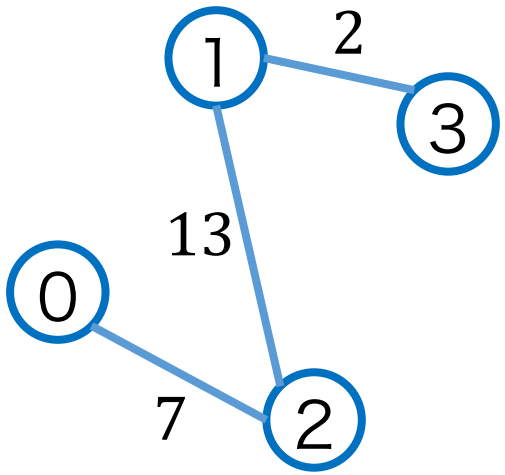
全域木 (Spanning Tree)

元のグラフがそもそも木だった場合のみ
すべての枝を使う必要がある
(というか元のグラフと同じものができる)

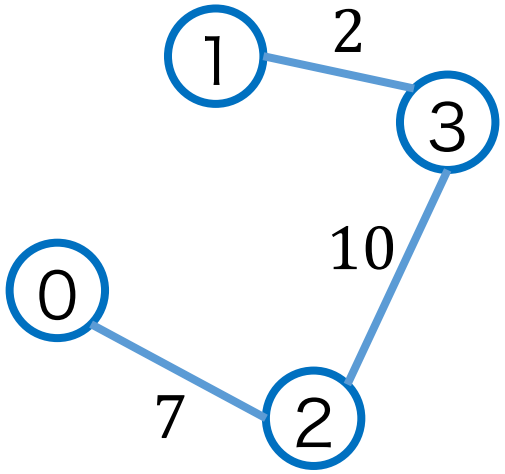
- **元のグラフの頂点すべてと、元のグラフの枝の一部 (or 全部) を使って作られた木のこと**



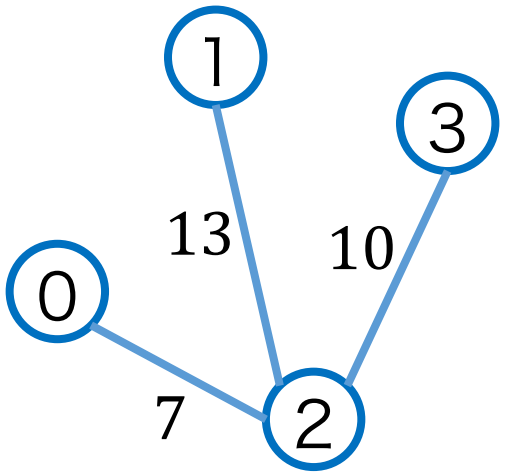
元のグラフ



全域木(1)
コスト合計: 22



全域木(2)
コスト合計: 19

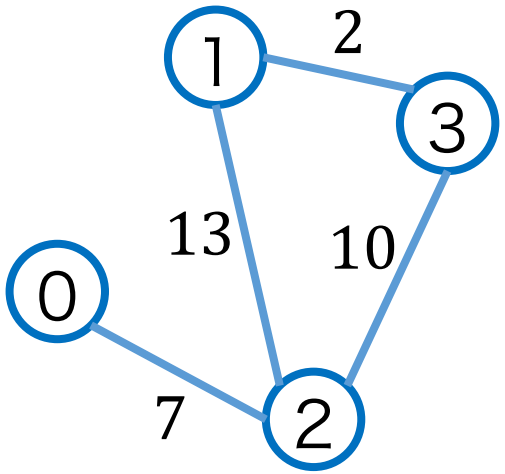


全域木(3)
コスト合計: 30

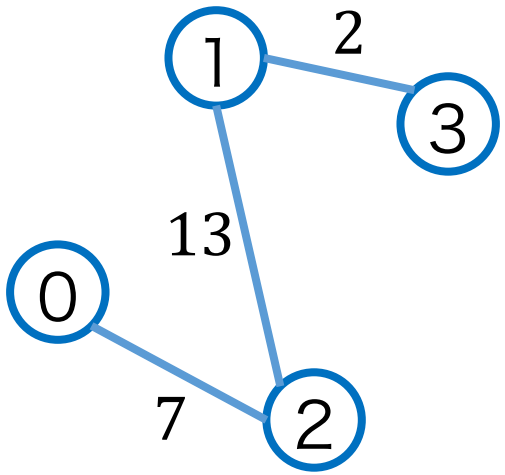
全域木 (Spanning Tree)

元のグラフがそもそも木だった場合のみ
すべての枝を使う必要がある
(というか元のグラフと同じものができる)

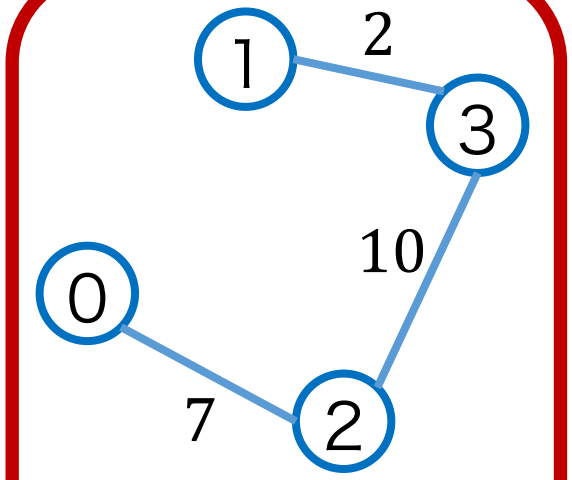
- **元のグラフの頂点すべてと、元のグラフの枝の一部 (or 全部) を使って作られた木のこと**



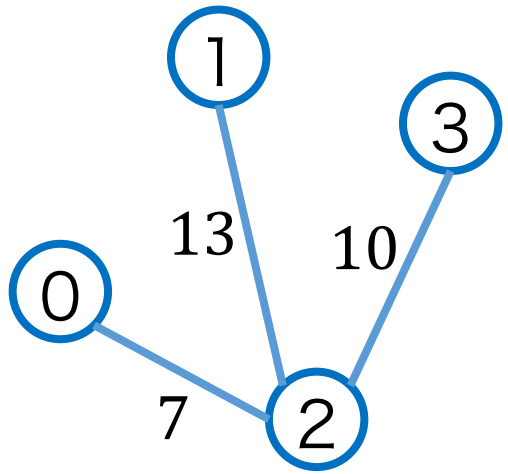
元のグラフ



全域木(1)
コスト合計: 22



全域木(2)
コスト合計: **19**



全域木(3)
コスト合計: 30

最小全域木

例題

除雪作業に耐えかねた高橋くんは、
道内を走る鉄道の真下に
地下道を掘ることにしました

しかし、高橋くんは疲れているため、
あまり長いトンネルを
掘りたくありません

任意の町同士が、地下道だけで
行き来可能になるためには、
少なくとも何キロのトンネルを
掘らなければいけないでしょうか？

例題

地点が v 個、路線が e 個与えられます

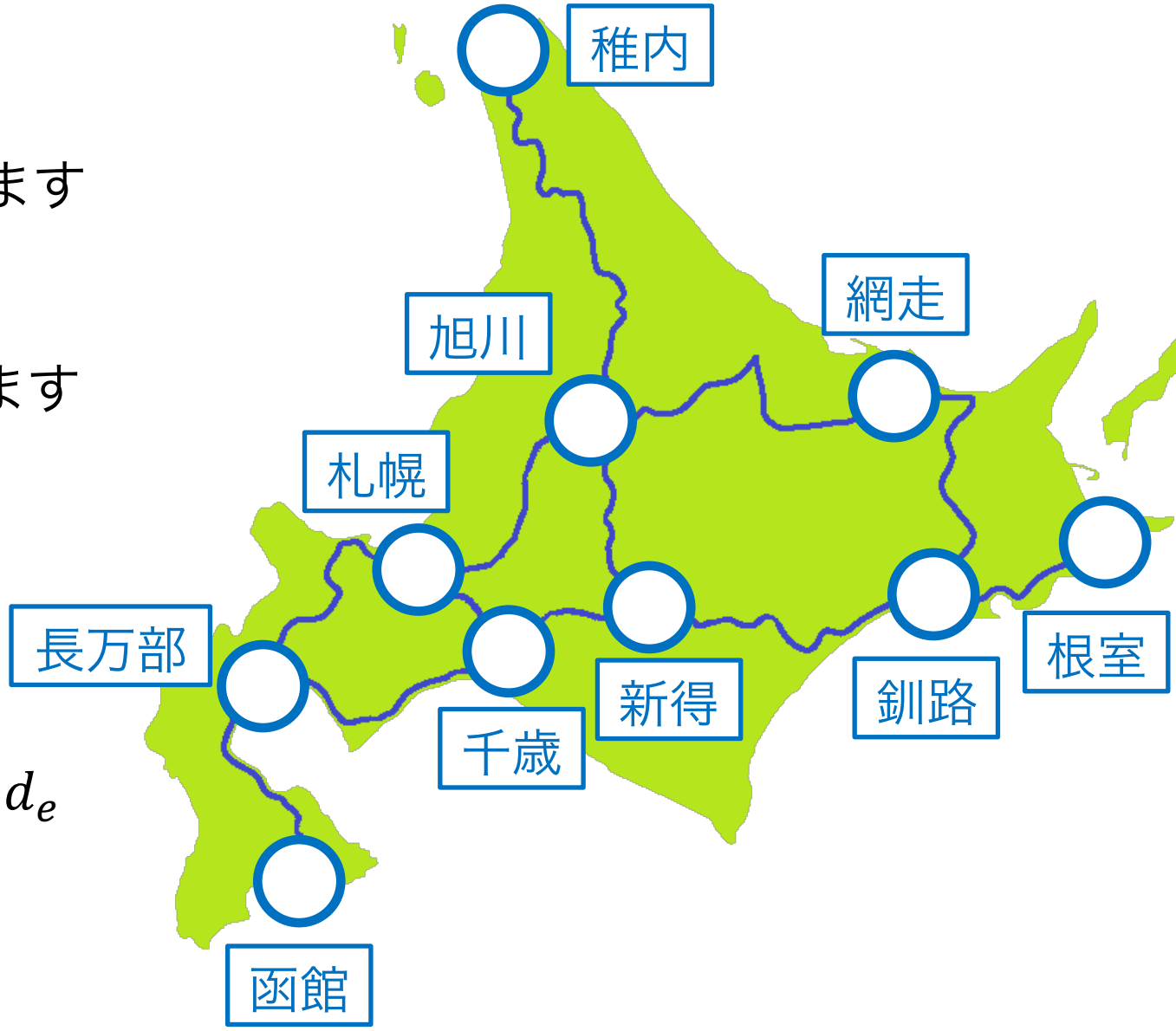
路線の情報は、
その路線が結ぶ2つの地点 a と b
及びその路線の距離 d で与えられます

- **Input**

  1行目 v e
  2行目 第1の路線の情報 a_1 b_1 d_1
  ：
  e+1行目 第eの路線の情報 a_e b_e d_e

- **Constraints**

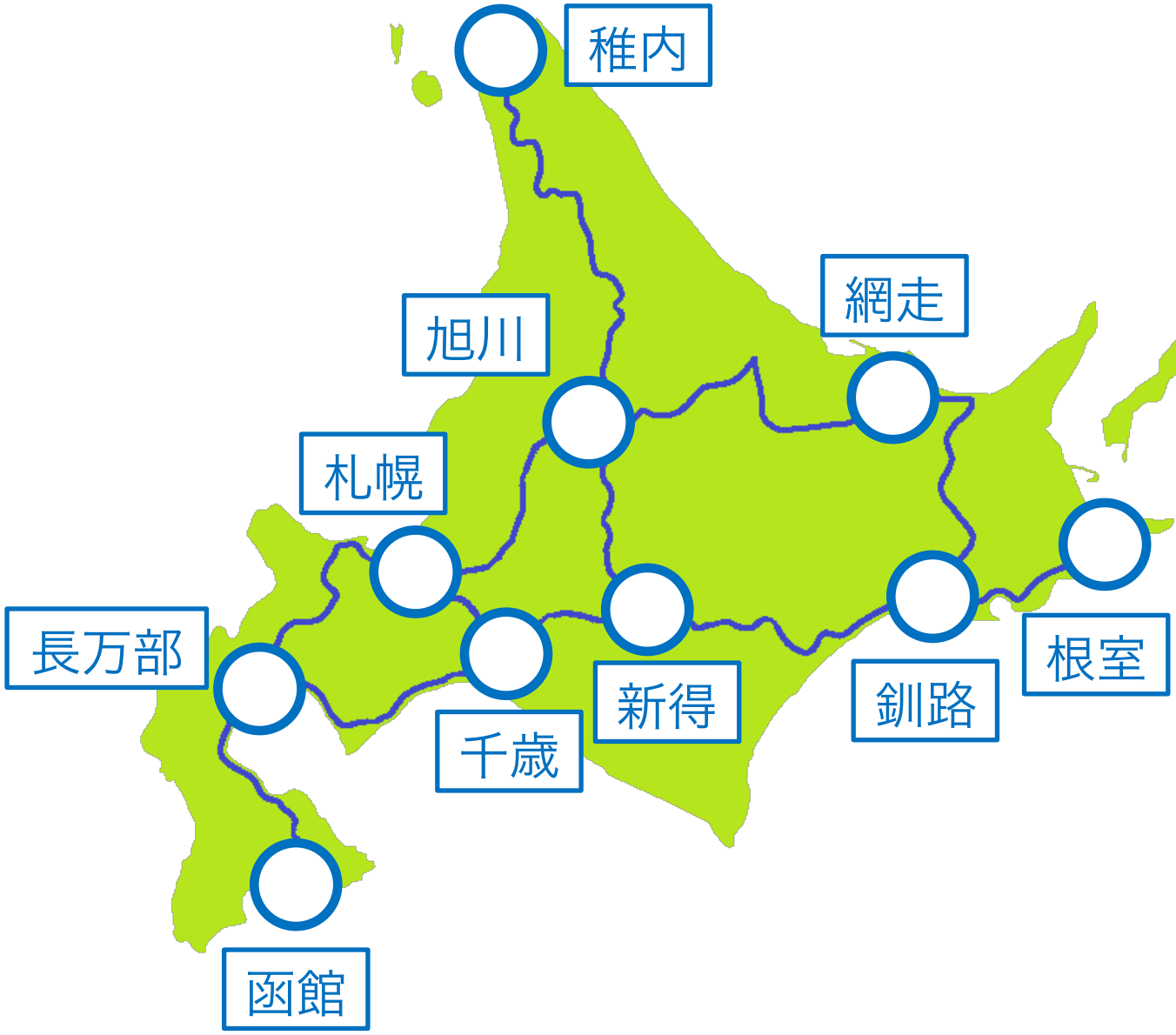
  $1 \leq v \leq 100$
  $1 \leq e, d \leq 10000$



例題

- **Sample Input**

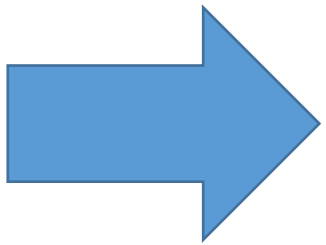
10　12
長万部　函館　110
札幌　長万部　175
札幌　旭川　135
旭川　稚内　260
長万部　千歳　160
千歳　札幌　45
千歳　新得　125
新得　旭川　130
釧路　新得　180
釧路　根室　135
網走　旭川　240
網走　釧路　170

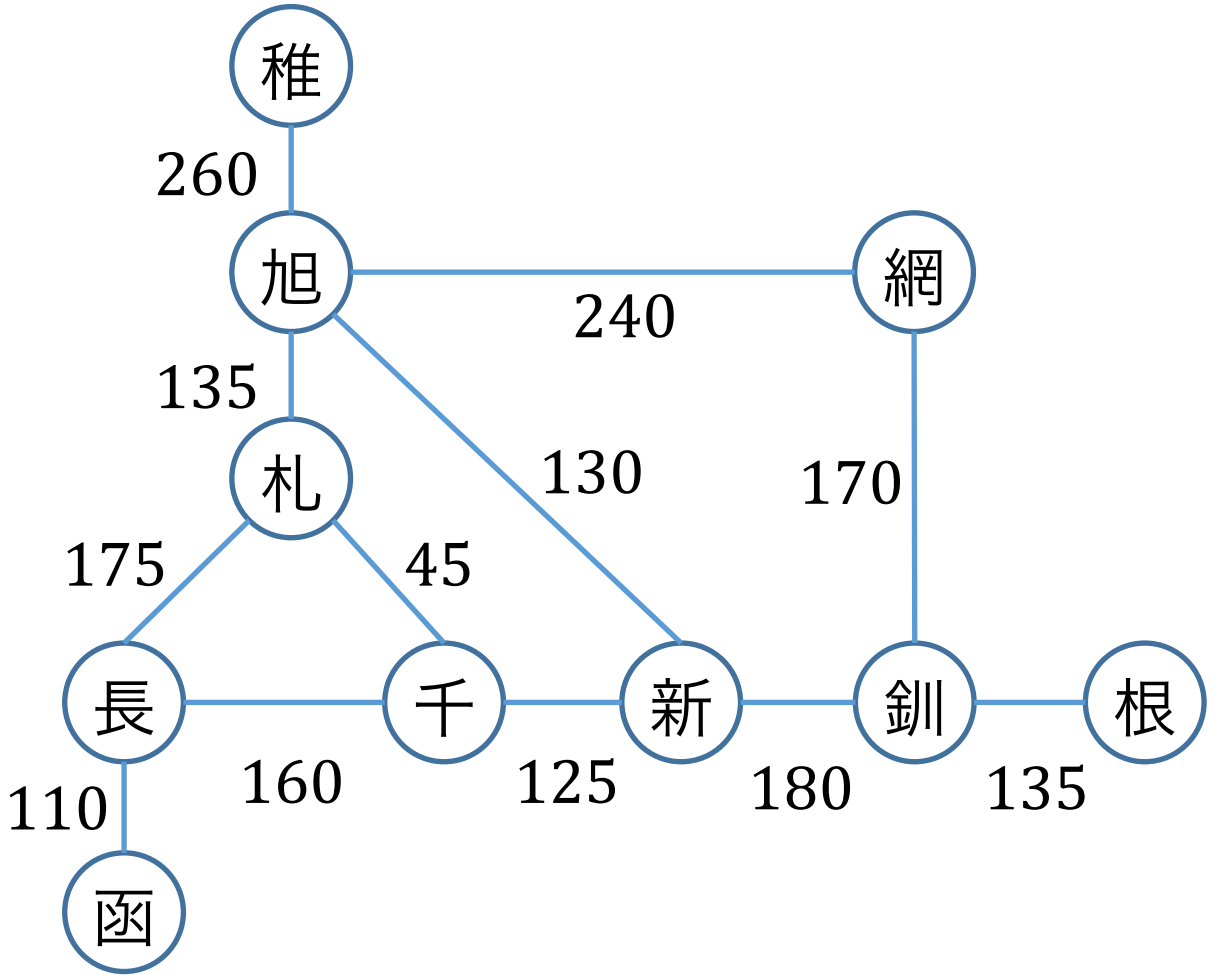


例題

- **Sample Input**

10　12
長万部　函館　110
札幌　長万部　175
札幌　旭川　135
旭川　稚内　260
長万部　千歳　160
千歳　札幌　45
千歳　新得　125
新得　旭川　130
釧路　新得　180
釧路　根室　135
網走　旭川　240
網走　釧路　170





プリム法 (Prim's MST algorithm)

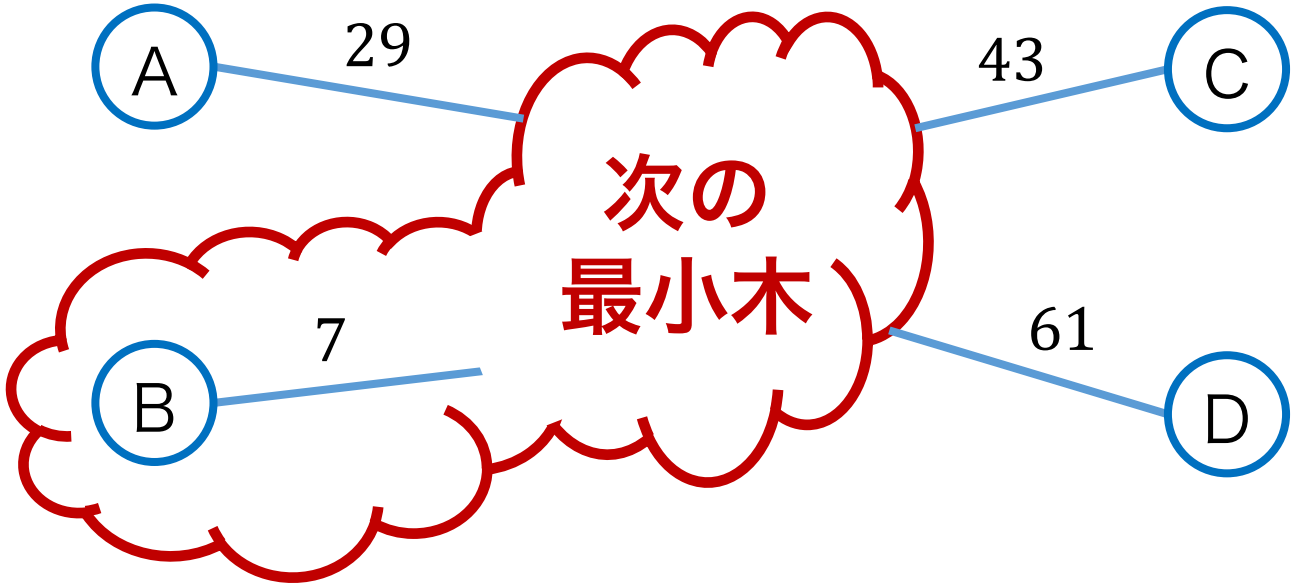
- **方針: 少ない頂点数で最小木を作って、それをじわじわ巨大化させていく！**

  - 現在の最小木から伸びてる辺のうち、**コストが一番小さい辺 (& その先にある頂点) を木に加える**

プリム法 (Prim's MST algorithm)

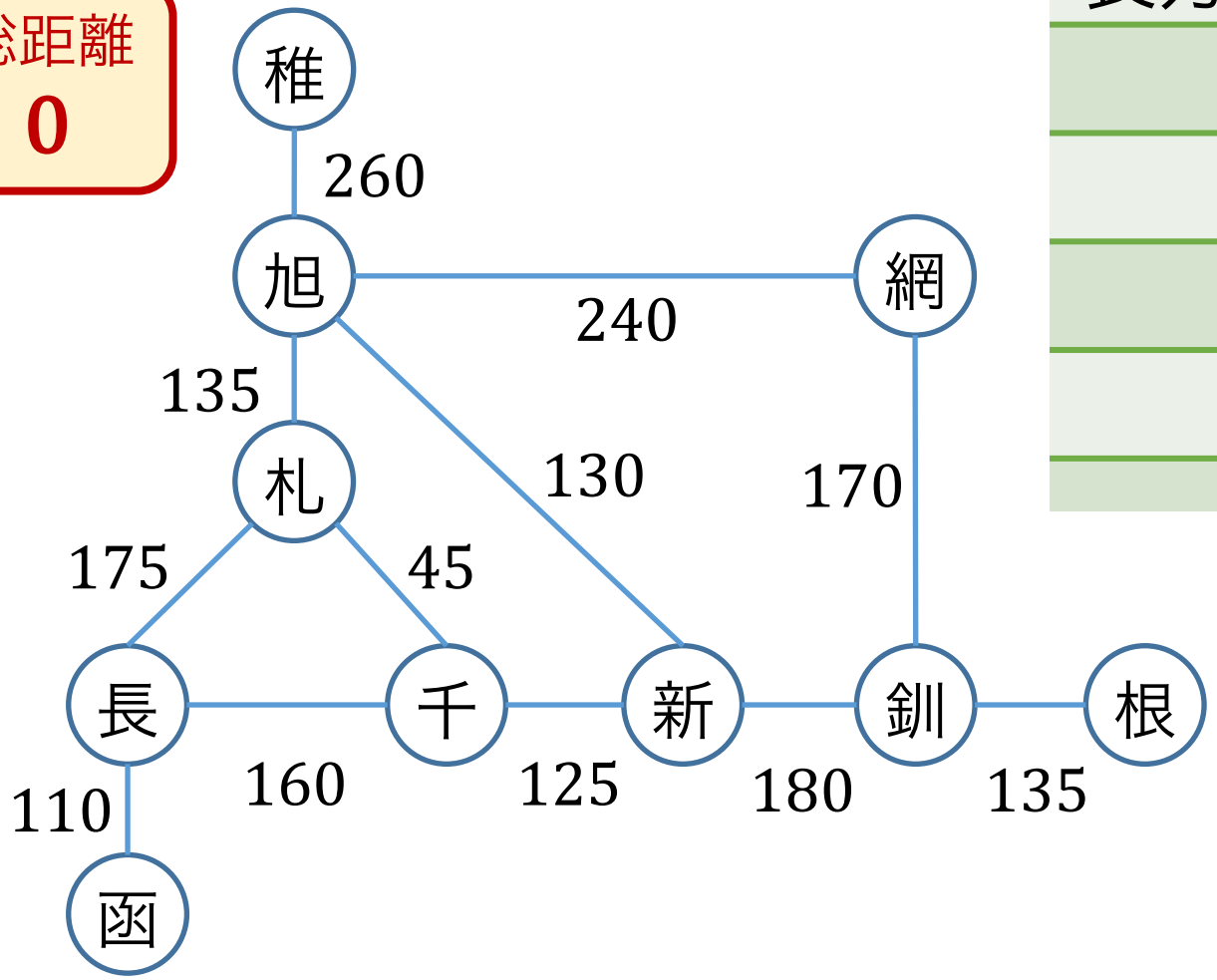
- **方針: 少ない頂点数で最小木を作って、それをじわじわ巨大化させていく！**

  - 現在の最小木から伸びてる辺のうち、**コストが一番小さい辺 (& その先にある頂点) を木に加える**



コストの更新

総距離
0



待機列 (Priority Q)

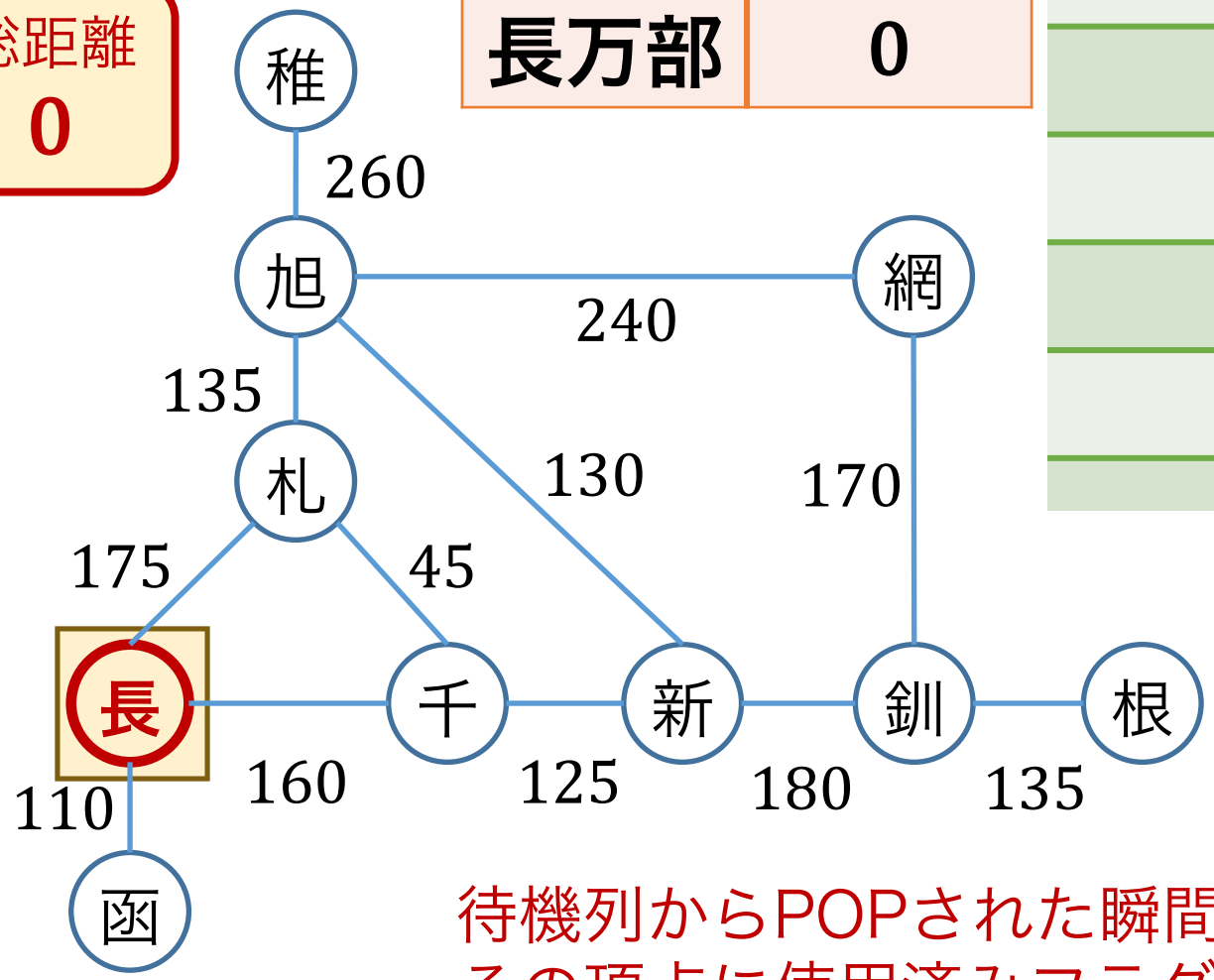
長万部	0

結果列

長万部	0
函館	INF
札幌	INF
旭川	INF
稚内	INF
千歳	INF
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新

総距離
0



長万部	0

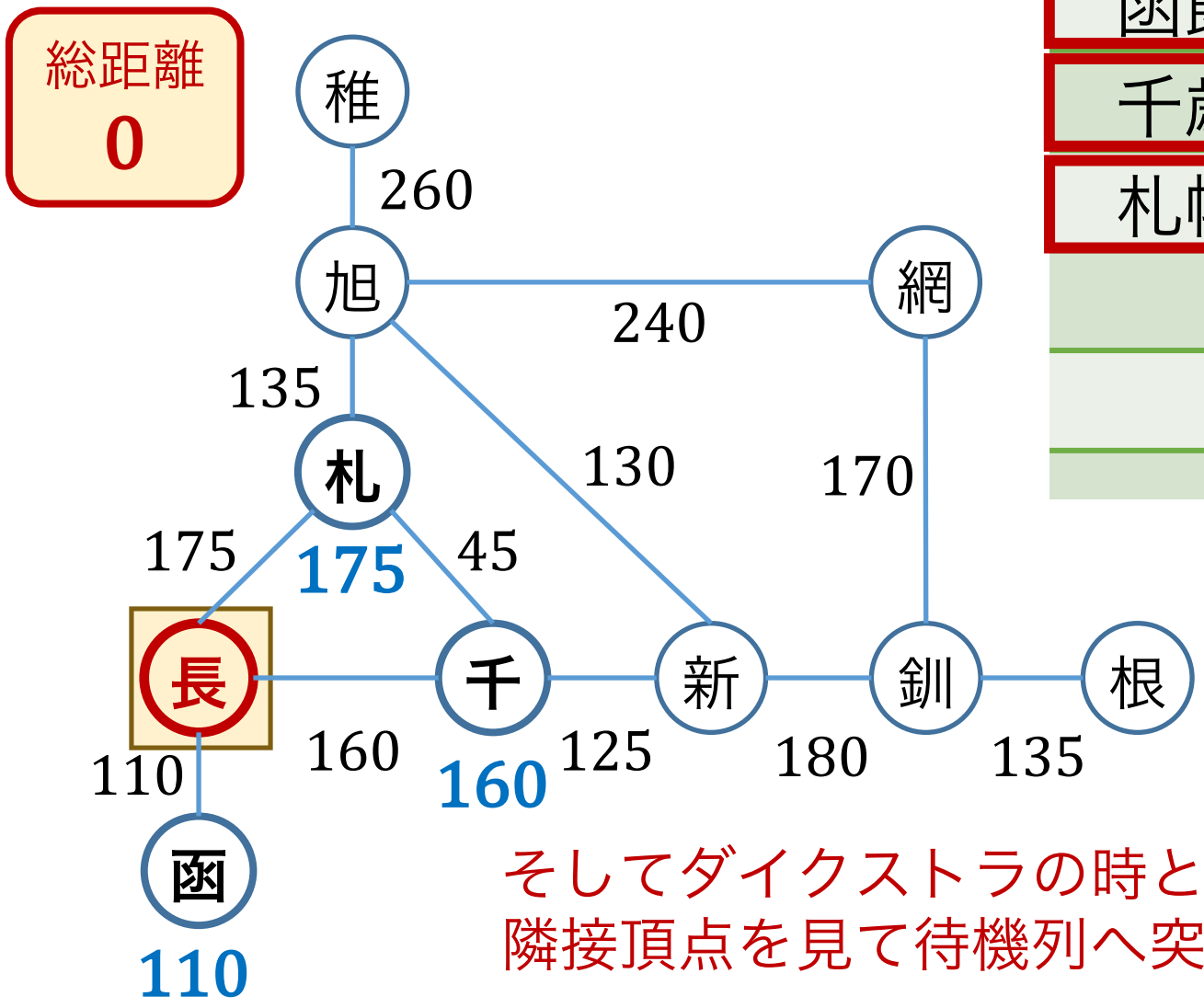
待機列 (Priority Q)

結果列

長万部	USED
函館	INF
札幌	INF
旭川	INF
稚内	INF
千歳	INF
新得	INF
釧路	INF
根室	INF
網走	INF

待機列からPOPされた瞬間、
その頂点に使用済みフラグを立てる

コストの更新



そしてダイクストラの時と同様、
隣接頂点を見て待機列へ突っ込む

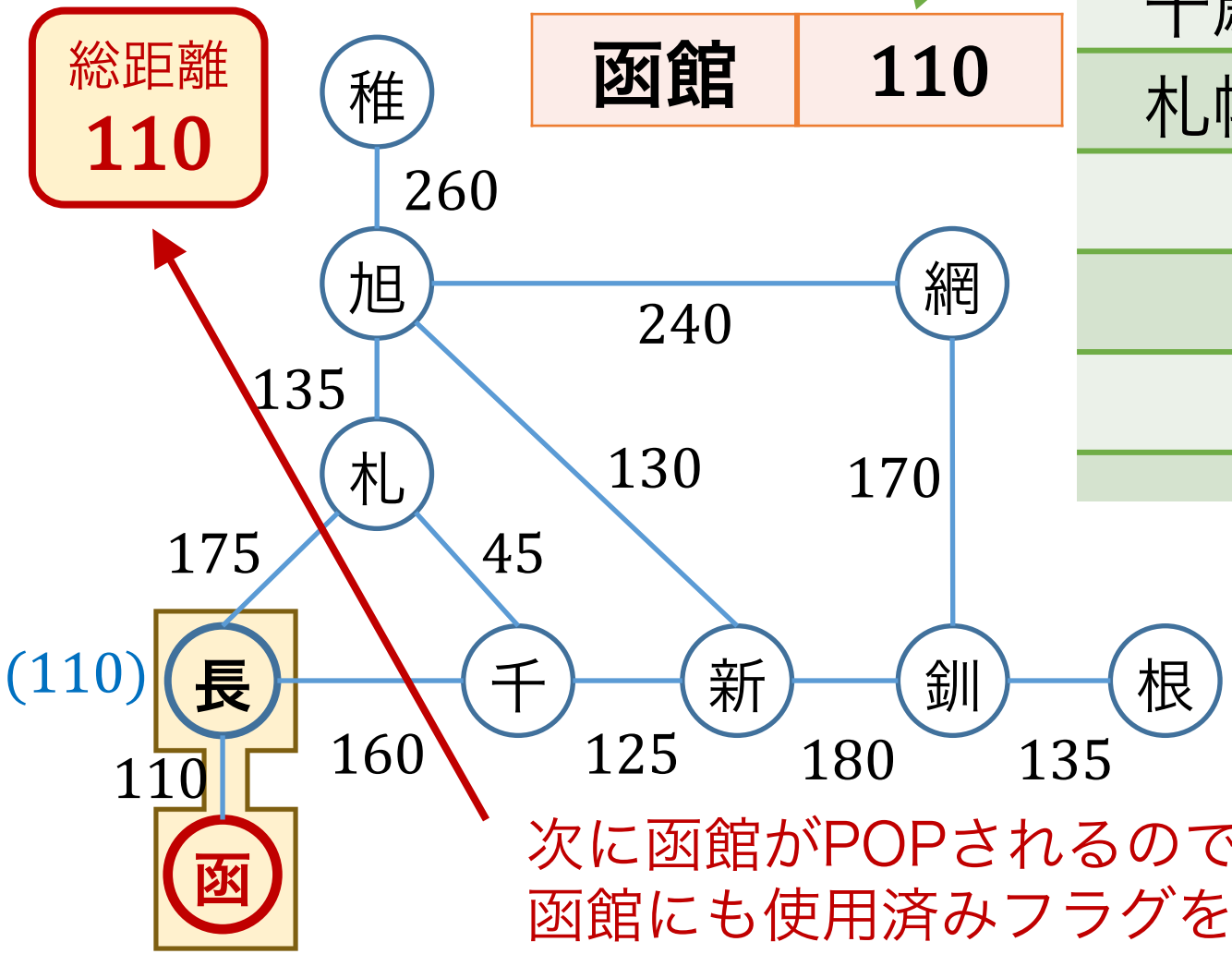
待機列 (Priority Q)

函館	110
千歳	160
札幌	175

結果列

長万部	USED
函館	110
札幌	175
旭川	INF
稚内	INF
千歳	160
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



函館	110

待機列 (Priority Q)

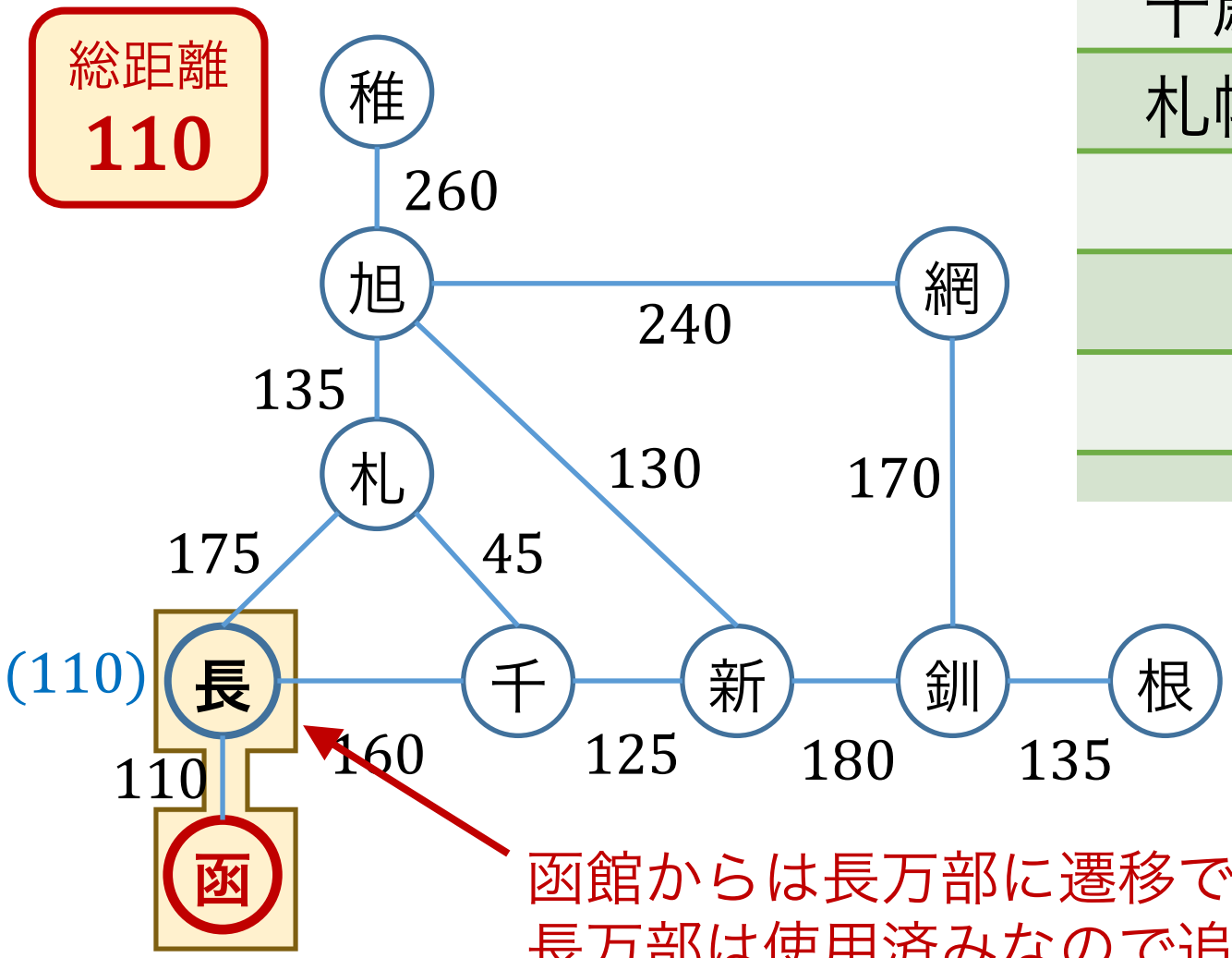
千歳	160
札幌	175

結果列

長万部	USED
函館	USED
札幌	175
旭川	INF
稚内	INF
千歳	160
新得	INF
釧路	INF
根室	INF
網走	INF

次に函館がPOPされるので、
函館にも使用済みフラグを立てる
同時に、出てきた110を総距離に加算

コストの更新



函館からは長万部に遷移できるが、
長万部は使用済みなので追加はしない

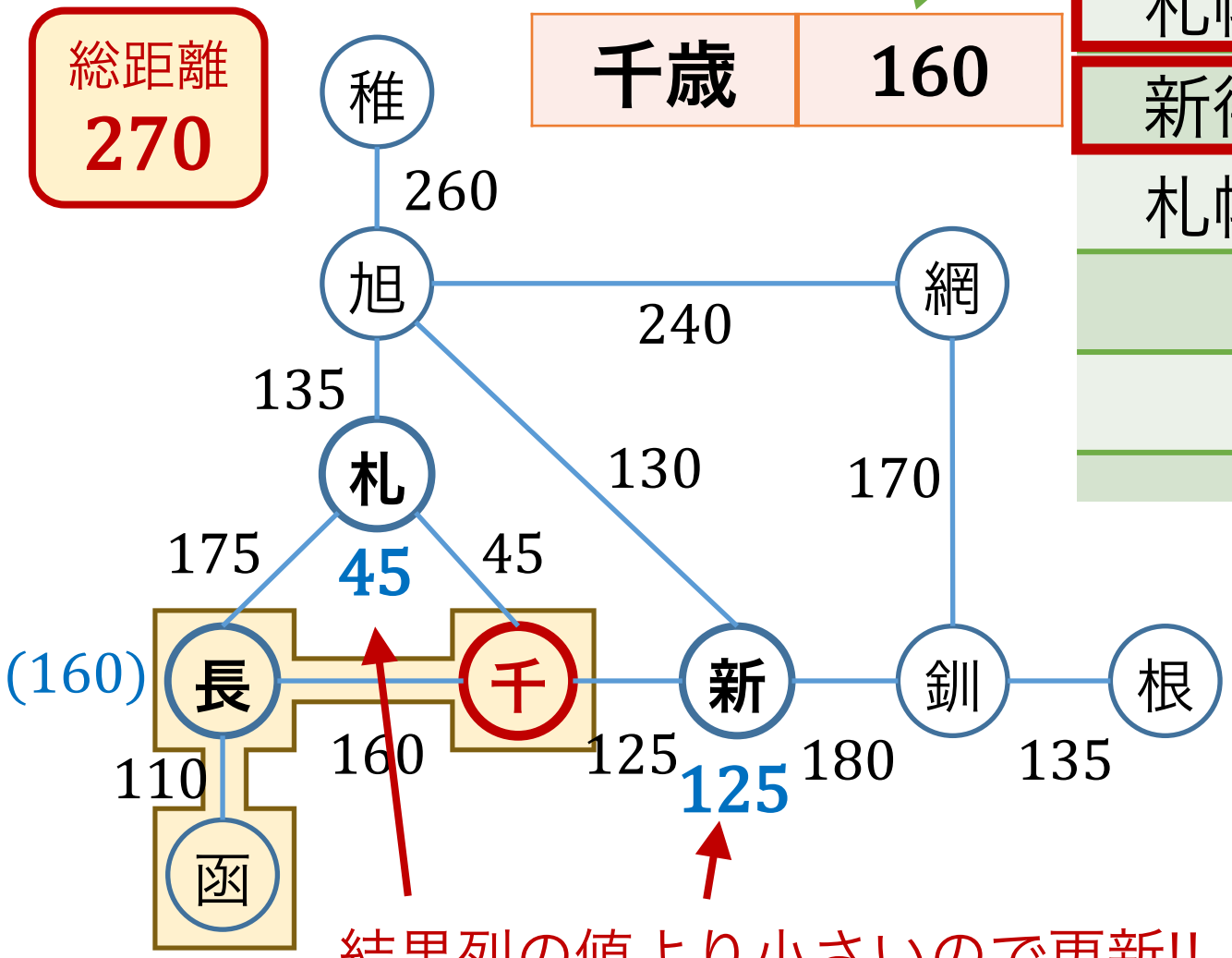
待機列 (Priority Q)

千歳	160
札幌	175

結果列

長万部	USED
函館	USED
札幌	175
旭川	INF
稚内	INF
千歳	160
新得	INF
釧路	INF
根室	INF
網走	INF

コストの更新



千歳	160

待機列 (Priority Q)

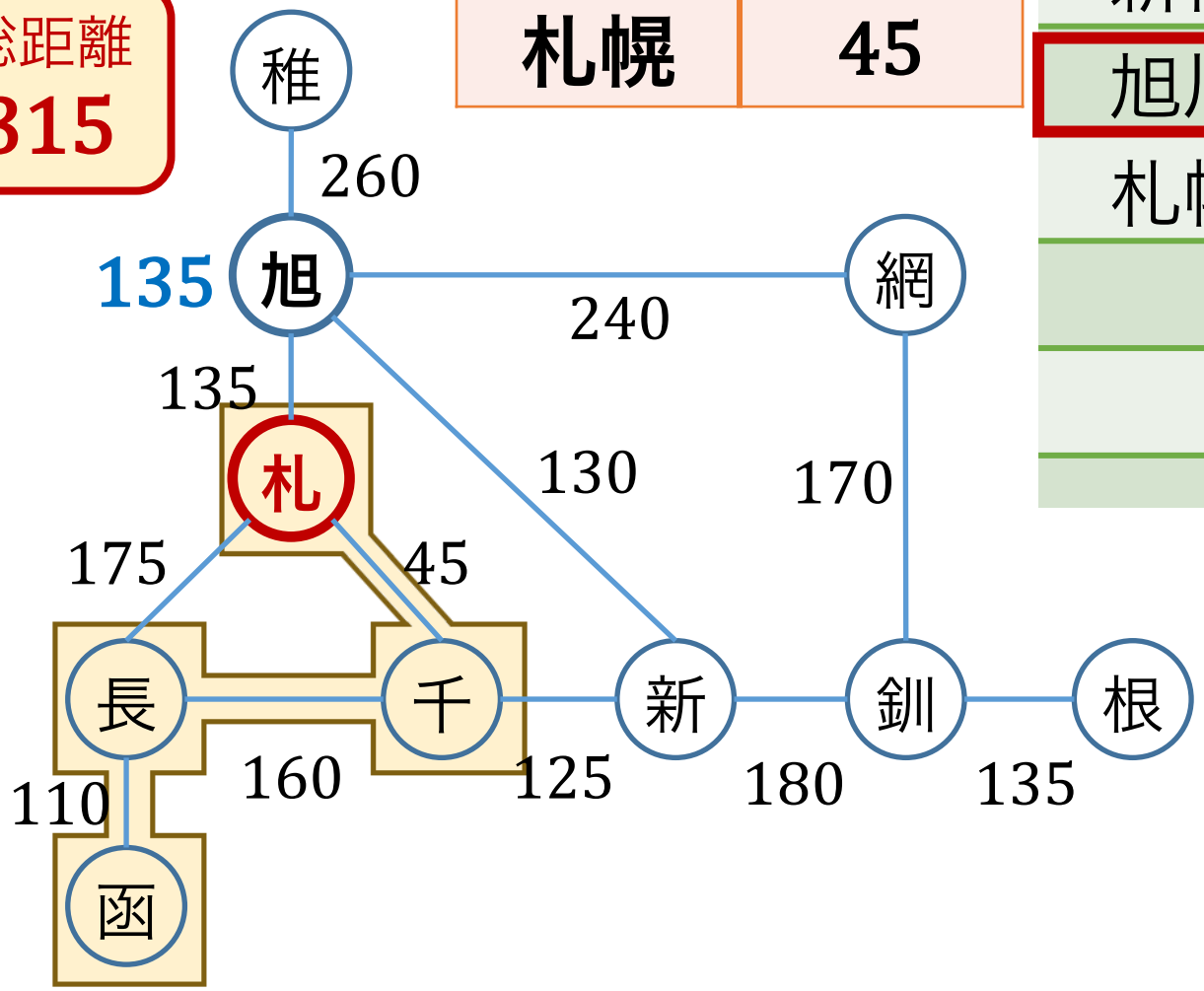
札幌	45
新得	125
札幌	175

結果列

長万部	USED
函館	USED
札幌	45
旭川	INF
稚内	INF
千歳	USED
新得	125
釧路	INF
根室	INF
網走	INF

コストの更新

総距離
315



札幌	45

待機列 (Priority Q)

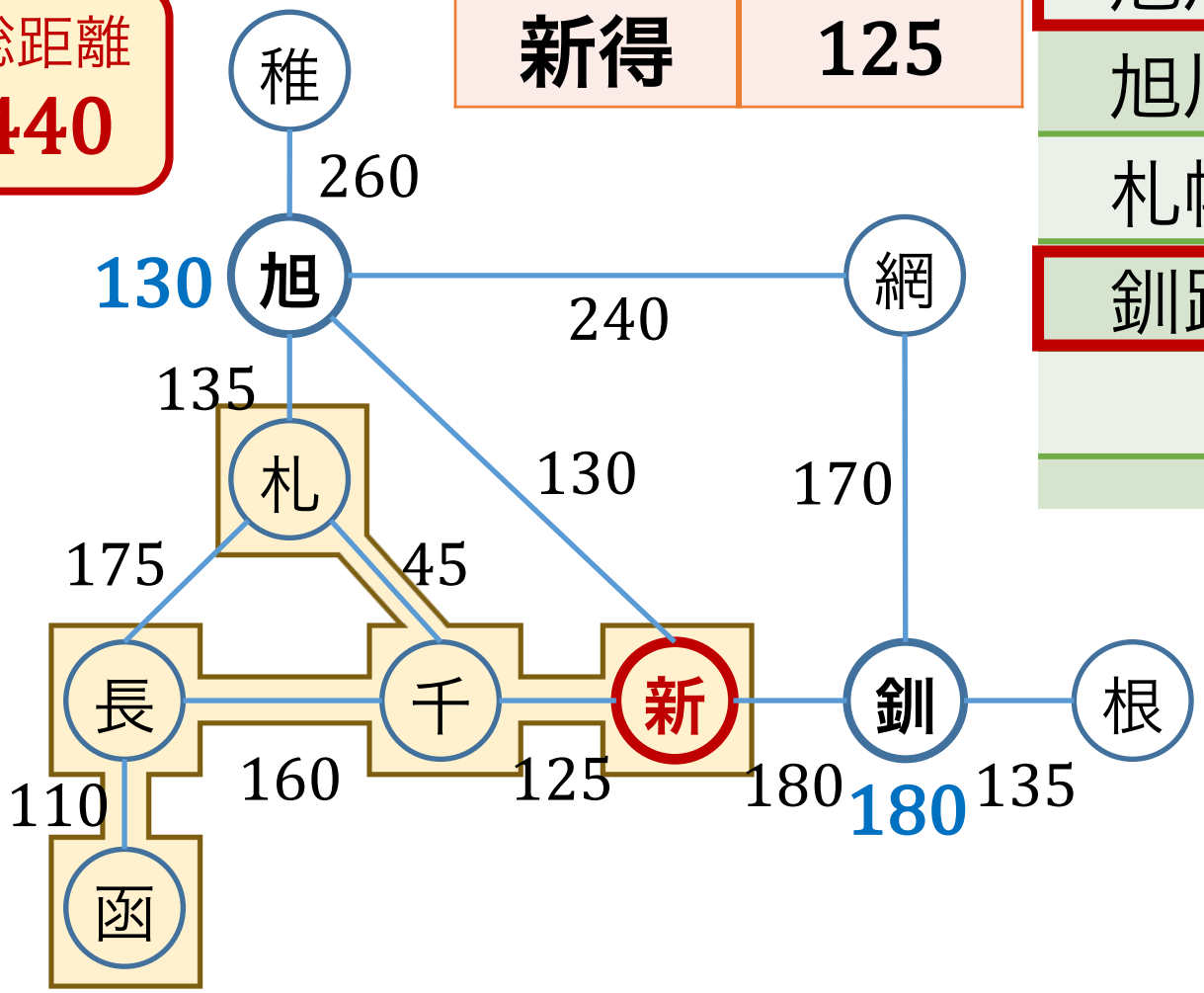
新得	125
旭川	135
札幌	175

結果列

長万部	USED
函館	USED
札幌	USED
旭川	135
稚内	INF
千歳	USED
新得	125
釧路	INF
根室	INF
網走	INF

コストの更新

総距離
440



新得	125

待機列 (Priority Q)

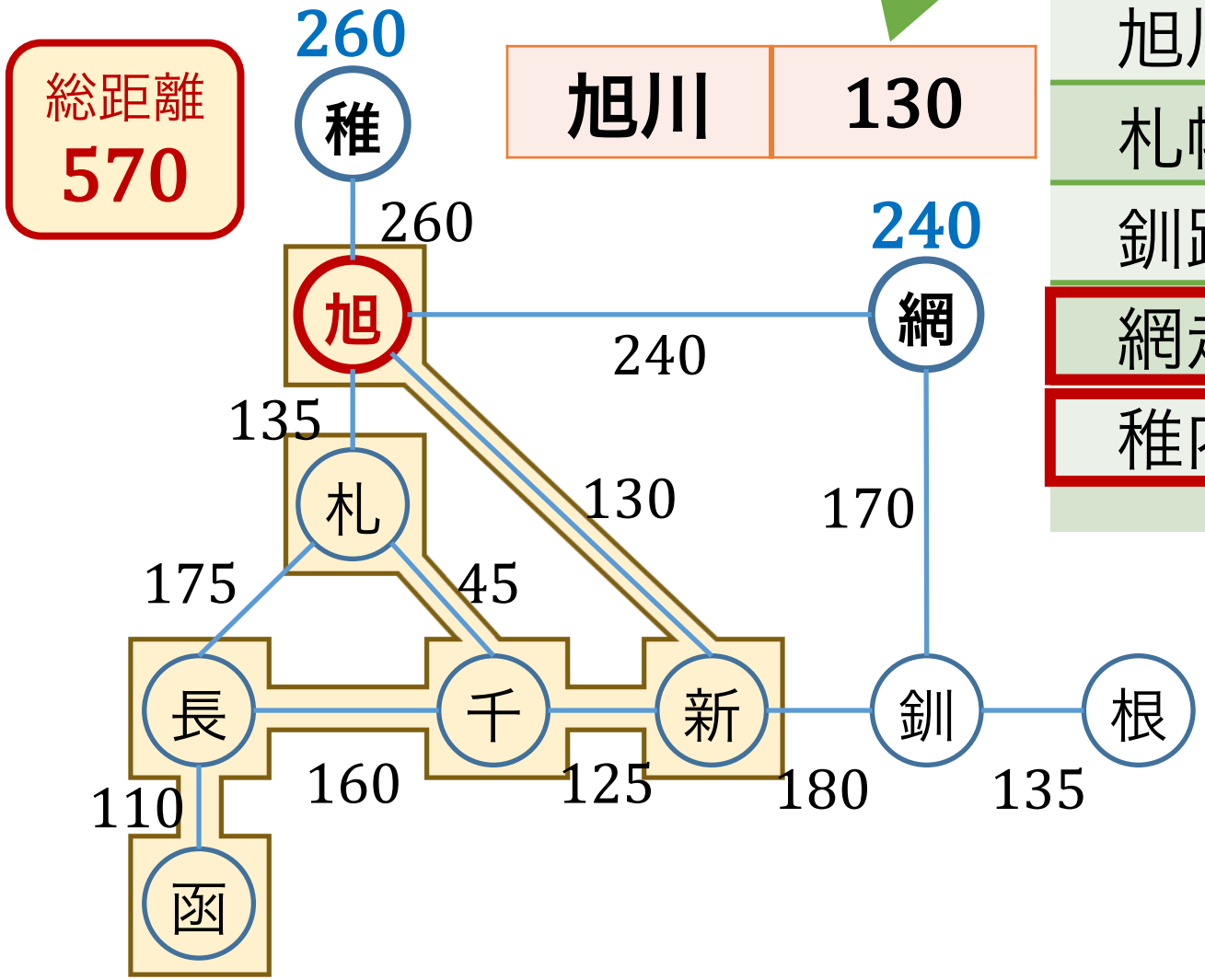
旭川	130
旭川	135
札幌	175
釧路	180

結果列

長万部	USED
函館	USED
札幌	USED
旭川	130
稚内	INF
千歳	USED
新得	USED
釧路	180
根室	INF
網走	INF

コストの更新

総距離
570



旭川	130

待機列 (Priority Q)

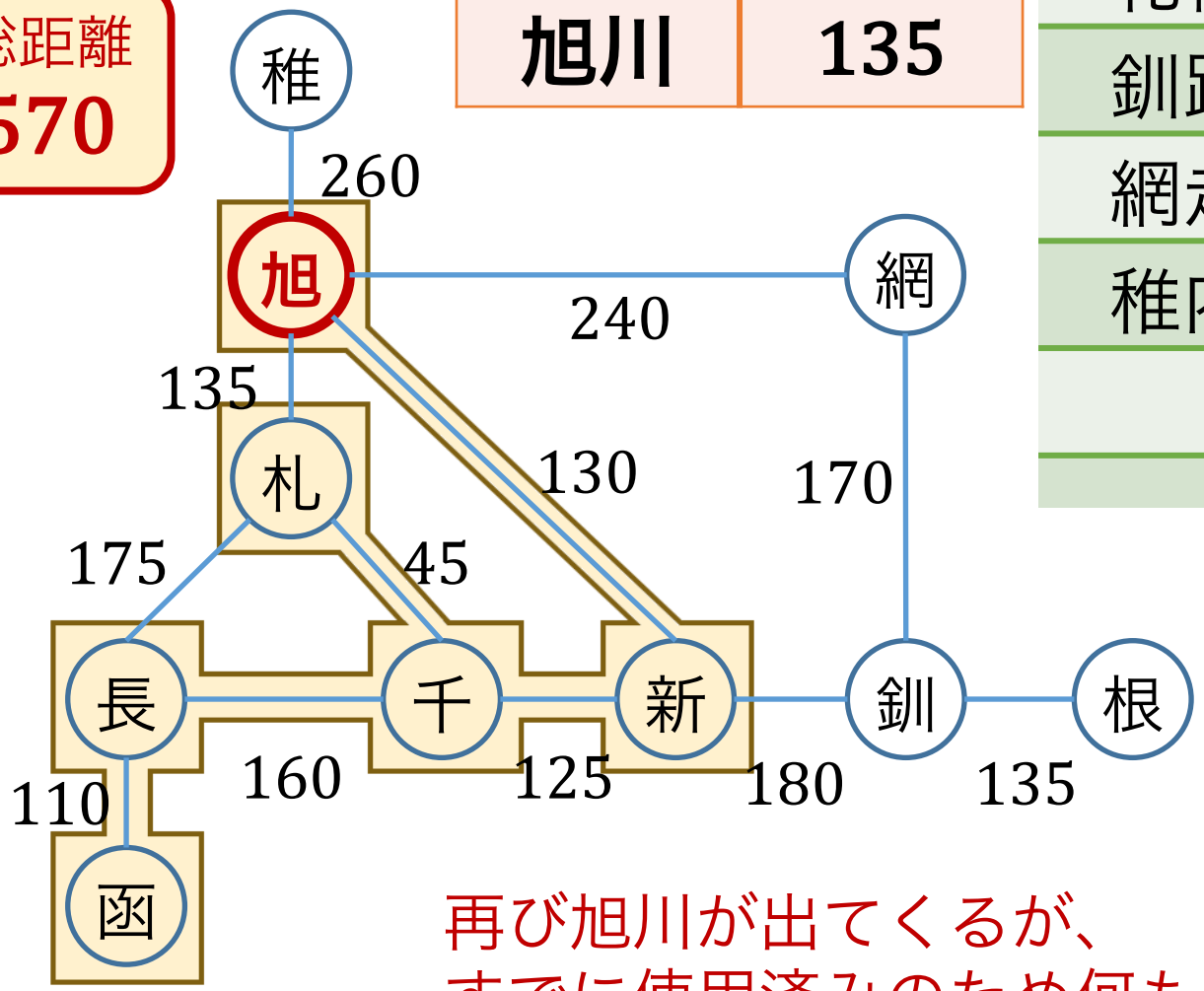
旭川	135
札幌	175
釧路	180
網走	240
稚内	260

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	180
根室	INF
網走	240

コストの更新

総距離
570



旭川	135

待機列 (Priority Q)

札幌	175
釧路	180
網走	240
稚内	260

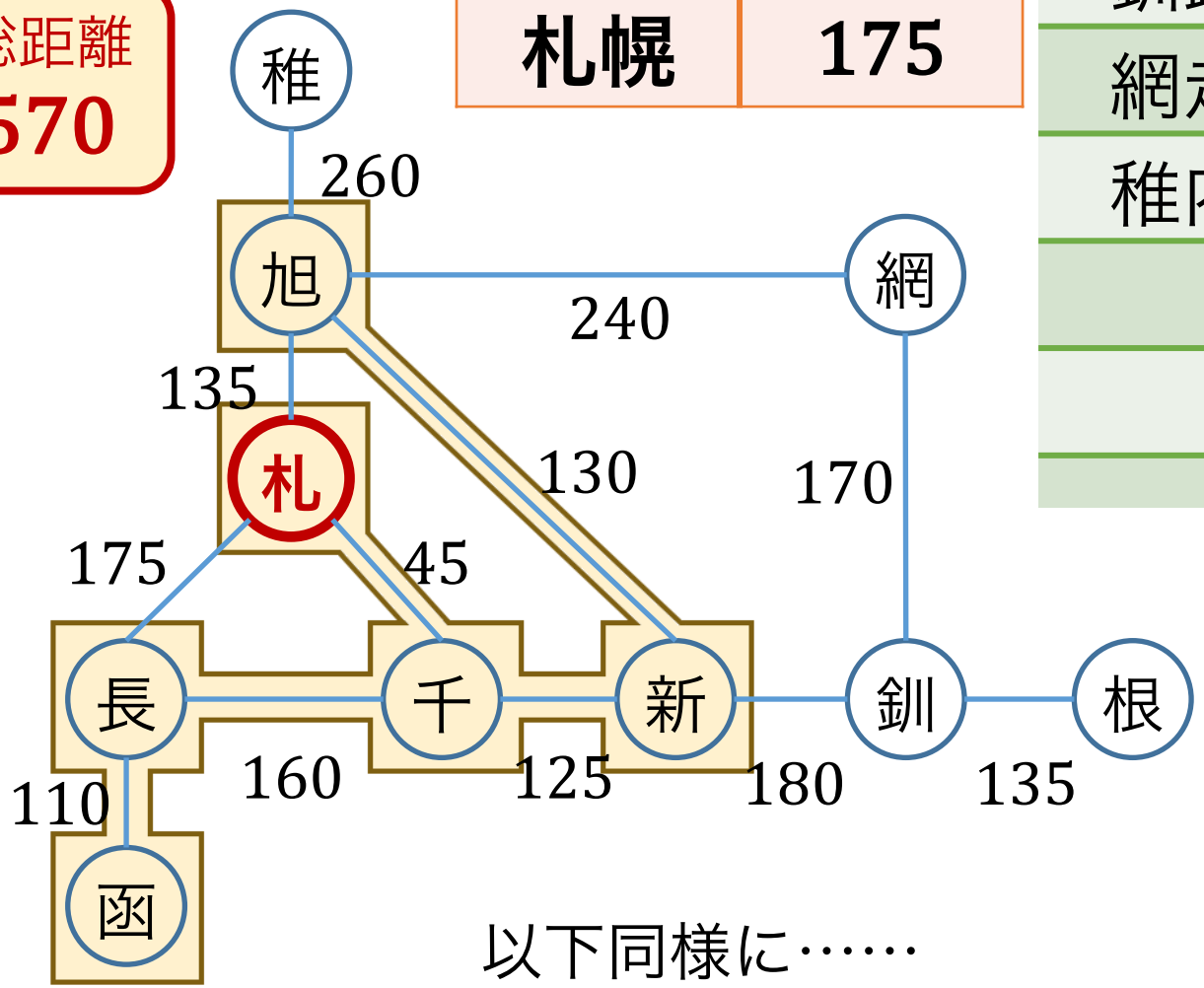
結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	180
根室	INF
網走	240

再び旭川が出てくるが、
すでに使用済みのため何もしない

コストの更新

総距離
570



札幌	175

以下同様に……

待機列 (Priority Q)

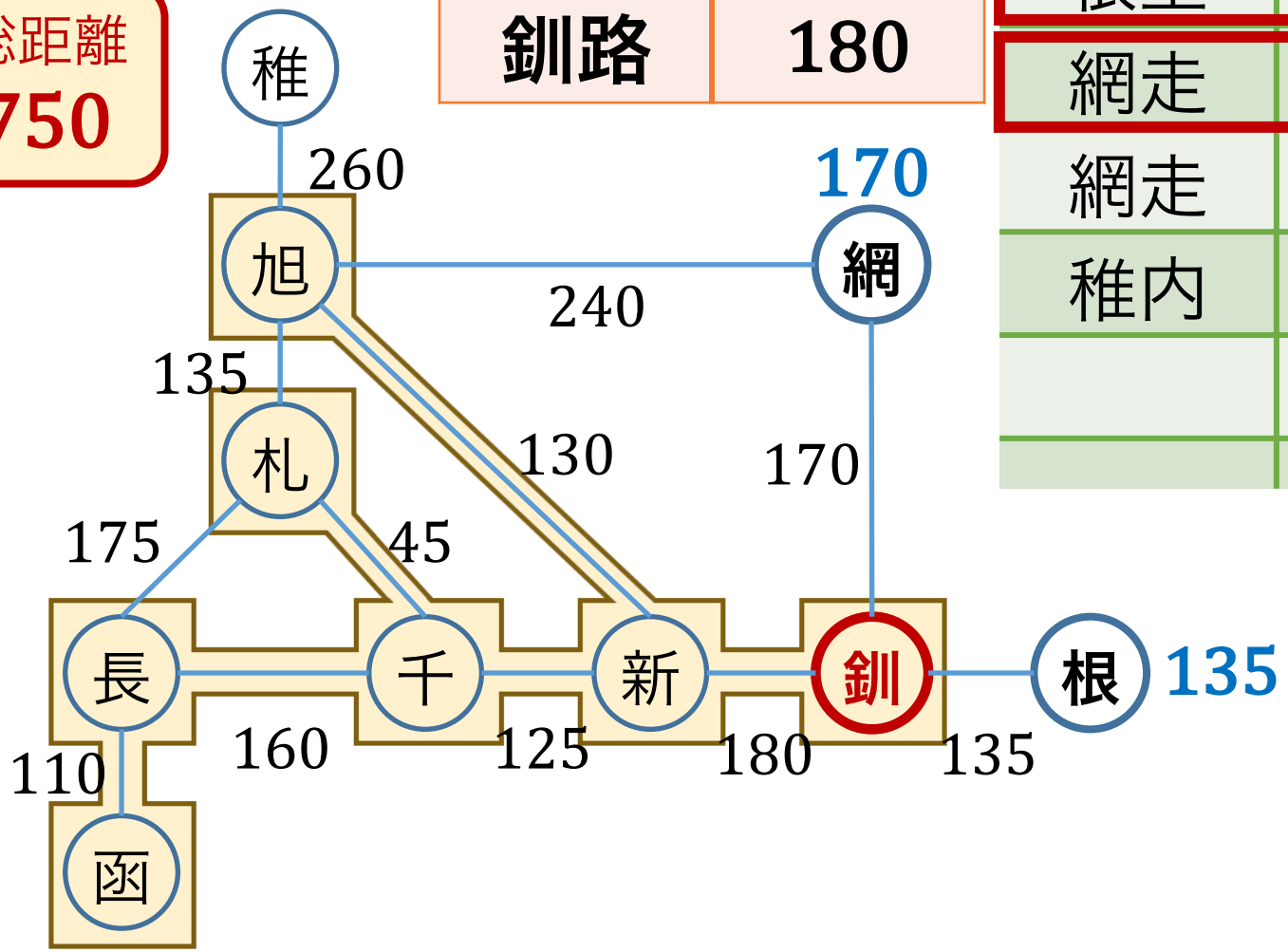
釧路	180
網走	240
稚内	260

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	180
根室	INF
網走	240

コストの更新

総距離
750



釧路	180

待機列 (Priority Q)

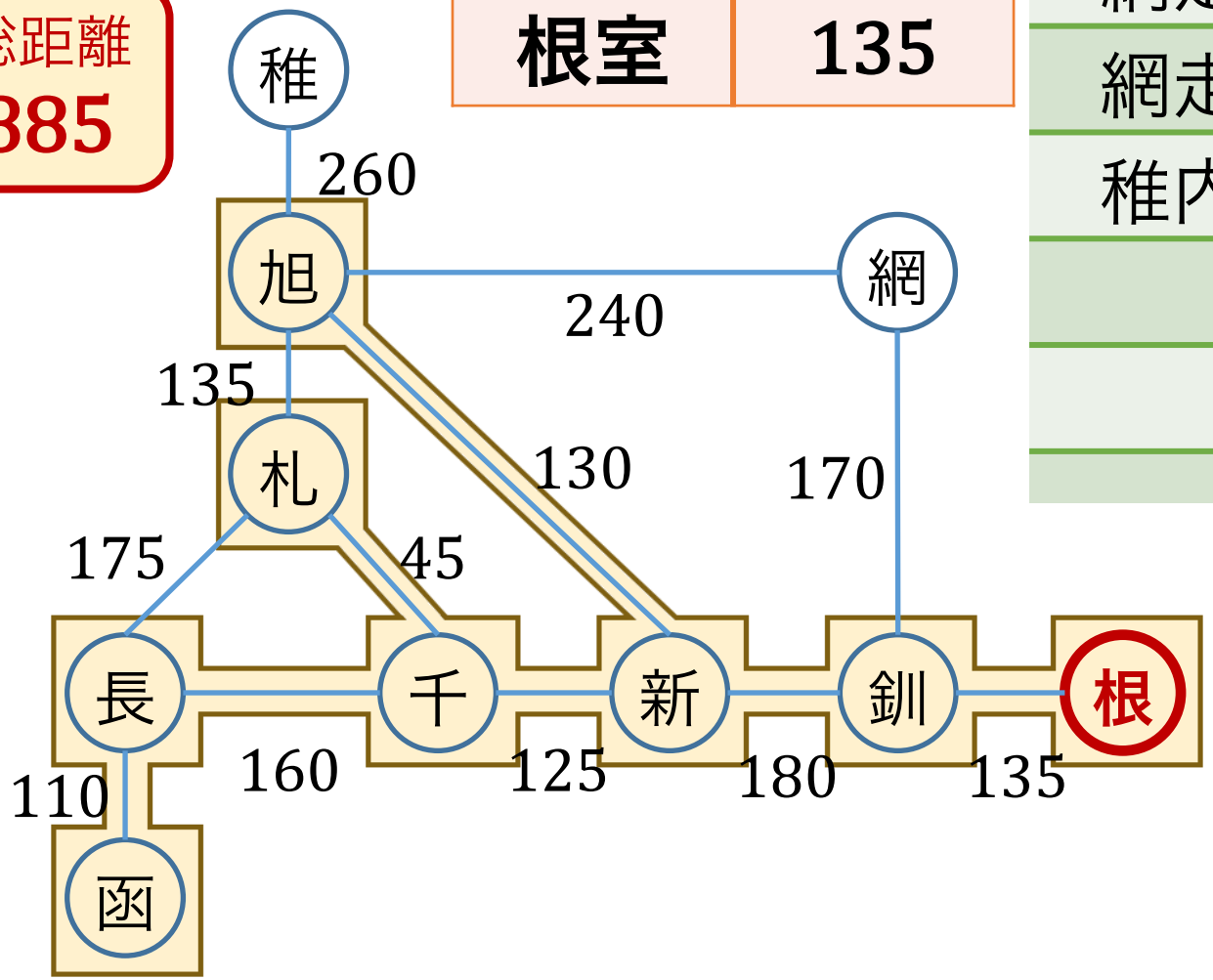
根室	135
網走	170
網走	240
稚内	260

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	USED
根室	135
網走	170

コストの更新

総距離
885



根室	135

待機列 (Priority Q)

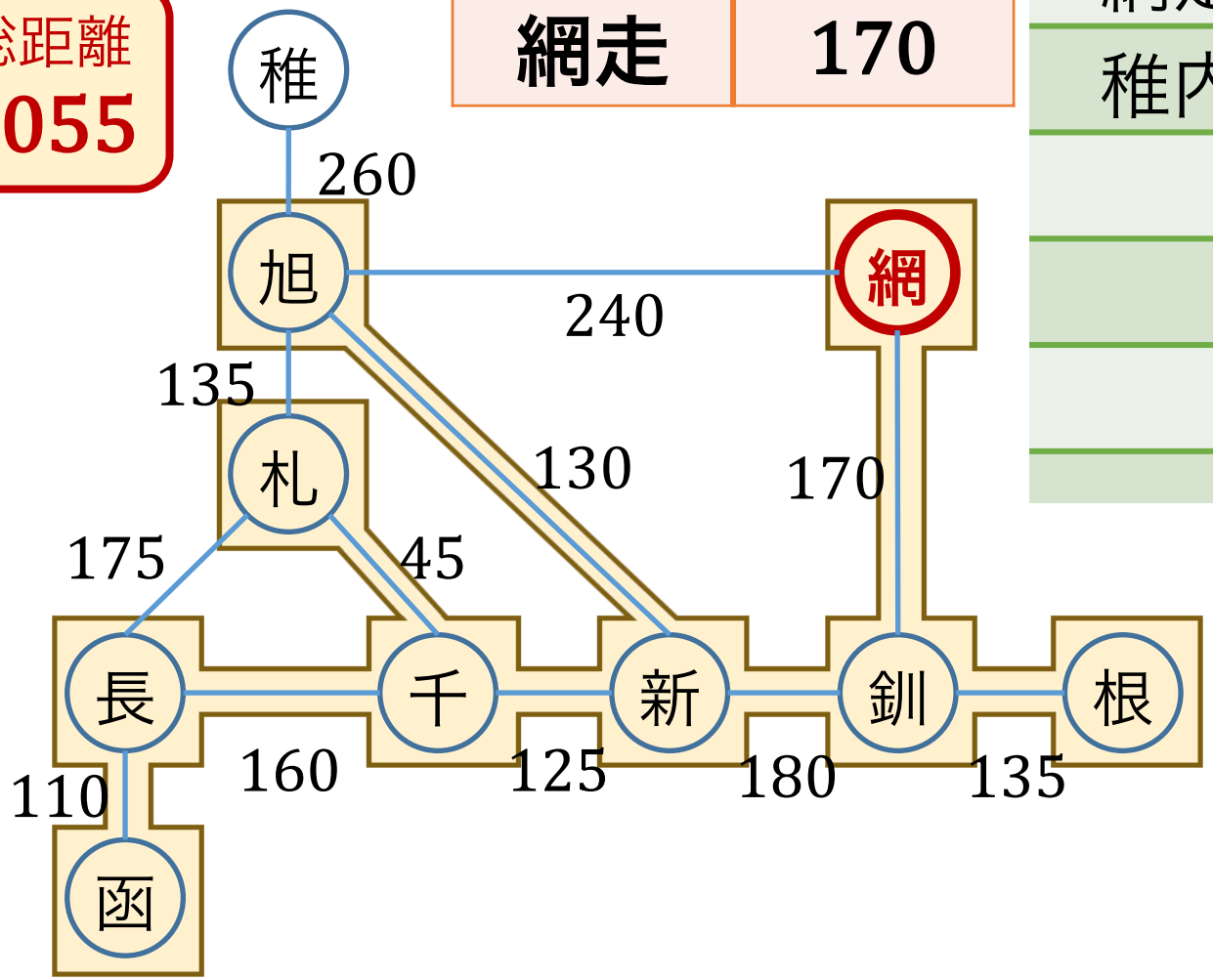
網走	170
網走	240
稚内	260

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	USED
根室	USED
網走	170

コストの更新

総距離
1055



網走	170

待機列 (Priority Q)

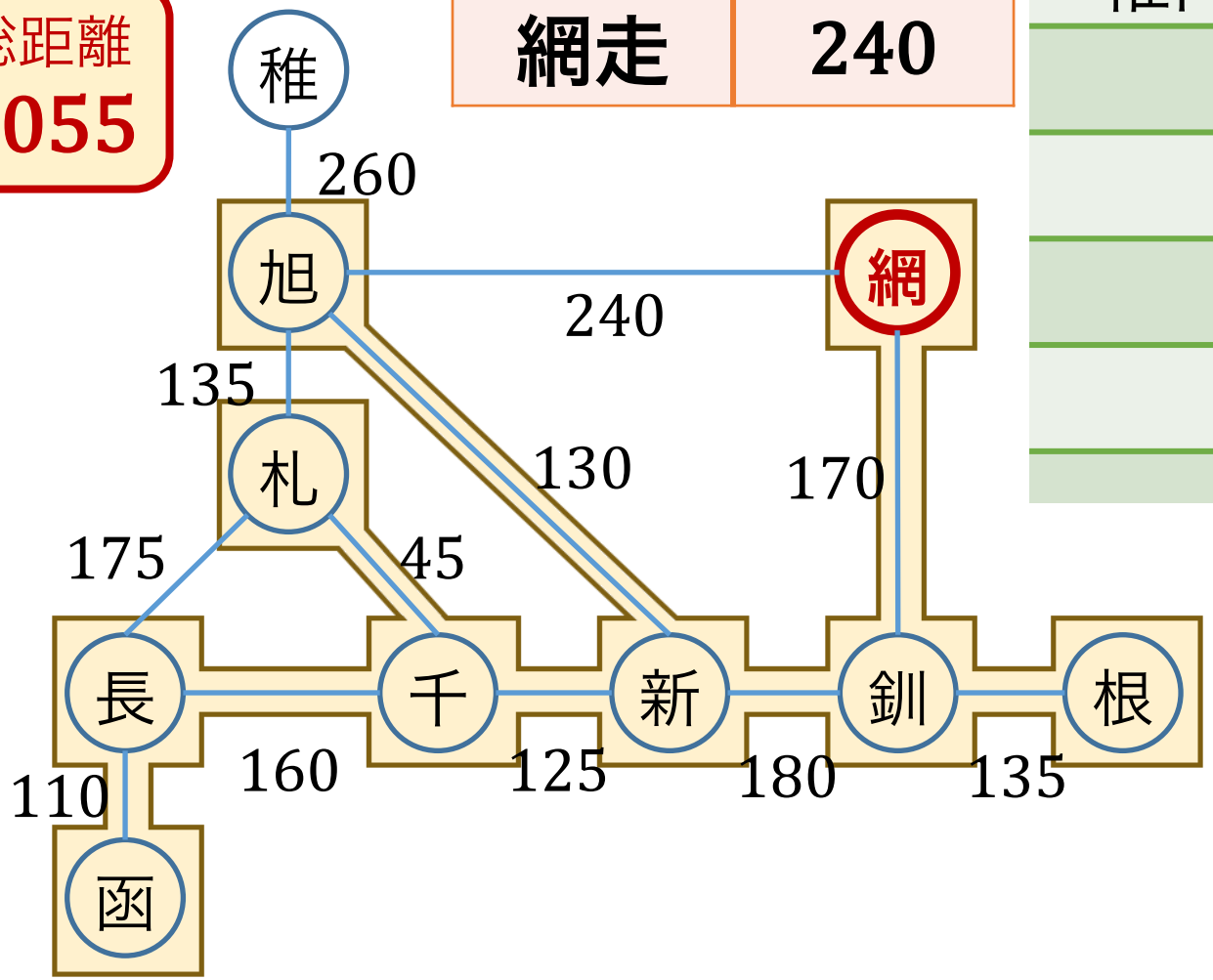
網走	240
稚内	260

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	USED
根室	USED
網走	USED

コストの更新

総距離
1055



網走	240

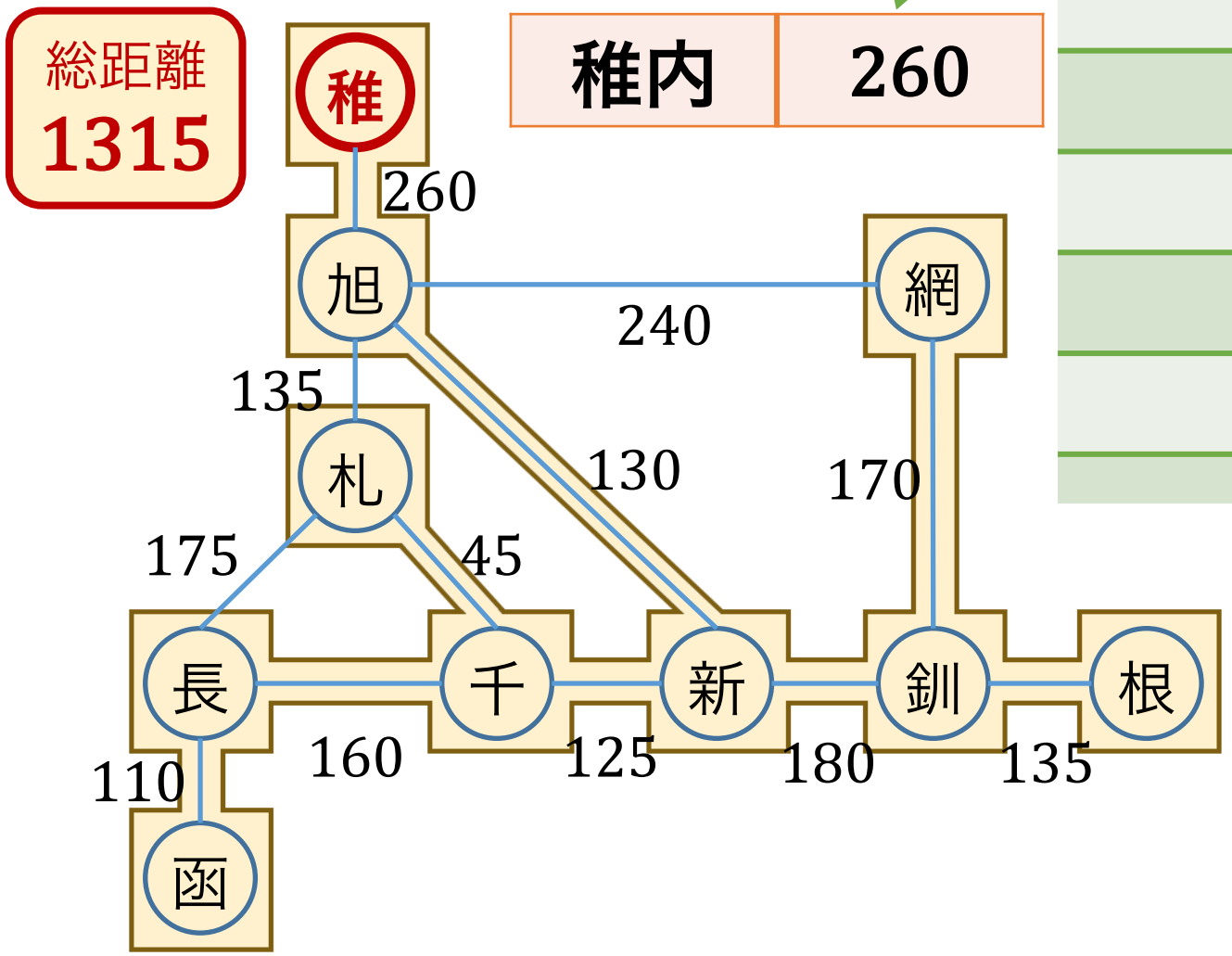
待機列 (Priority Q)

稚内	260

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	260
千歳	USED
新得	USED
釧路	USED
根室	USED
網走	USED

コストの更新



稚内	260

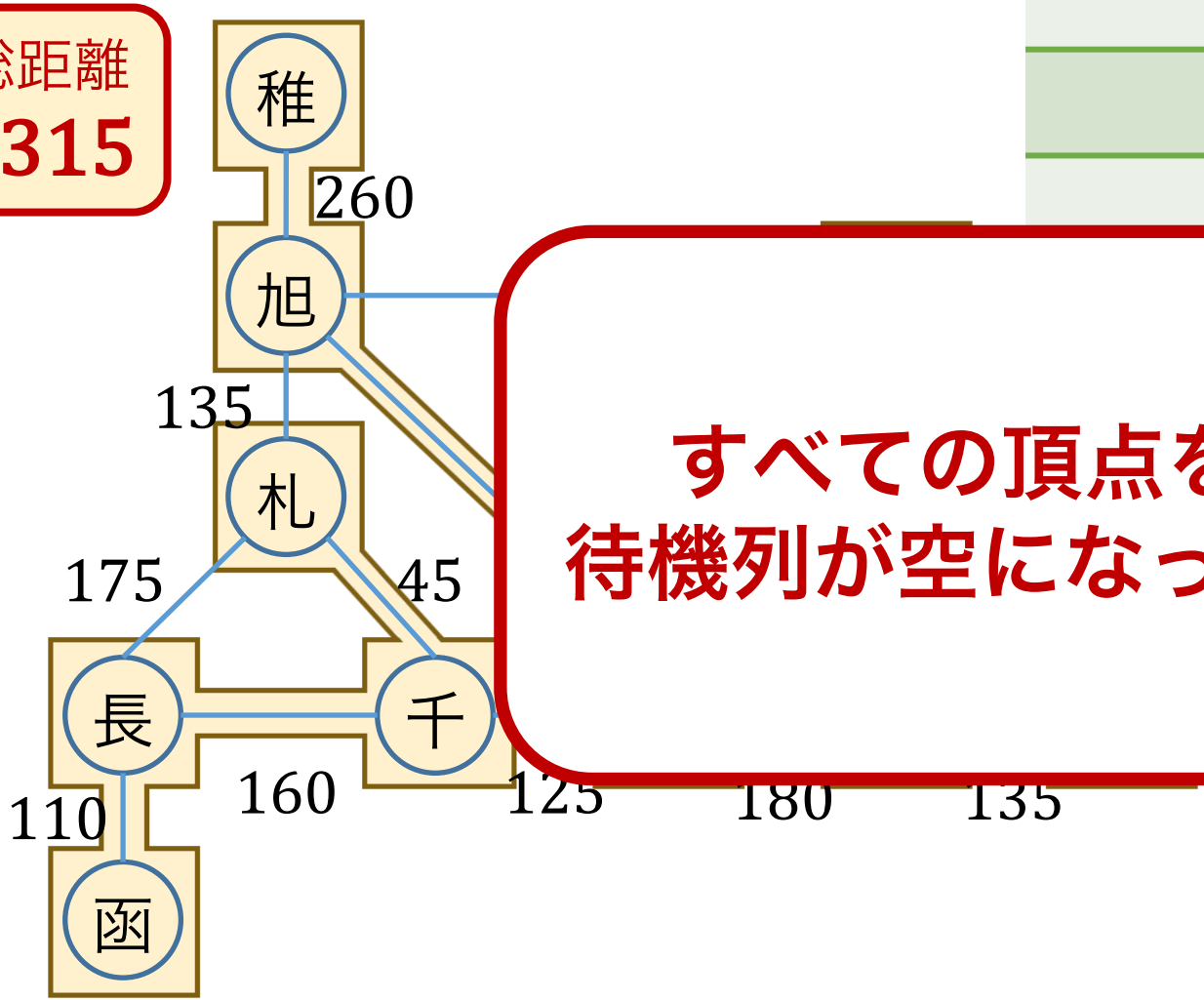
待機列 (Priority Q)

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	USED
千歳	USED
新得	USED
釧路	USED
根室	USED
網走	USED

コストの更新

総距離
1315



すべての頂点を使用 or
待機列が空になったら終了！

待機列 (Priority Q)

結果列

長万部	USED
函館	USED
札幌	USED
旭川	USED
稚内	USED
千歳	USED
新得	USED
釧路	USED
根室	USED
網走	USED

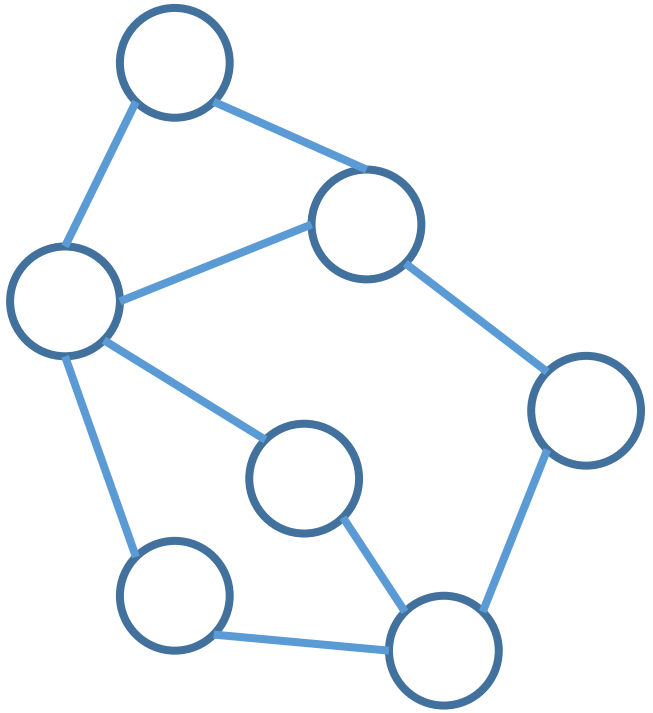
プリム法の正当性

元のグラフの全頂点の集合を U
作成途中の木に含まれている頂点の集合を X
X と U＼X を結ぶ最小コストの辺を e とおく

仮定: e を含まない最小全域木 T’ が存在する

1. T’ に e を加えると、必ず閉路ができる
2. その閉路に含まれる辺には、
   X と U＼X を結ぶ e 以外の辺 e’ が必ず存在する
3. T’ から e’ を取り除いて e を加えても全域木
4. e のコストは e’ 以下なので、e に置き換えても最小全域木

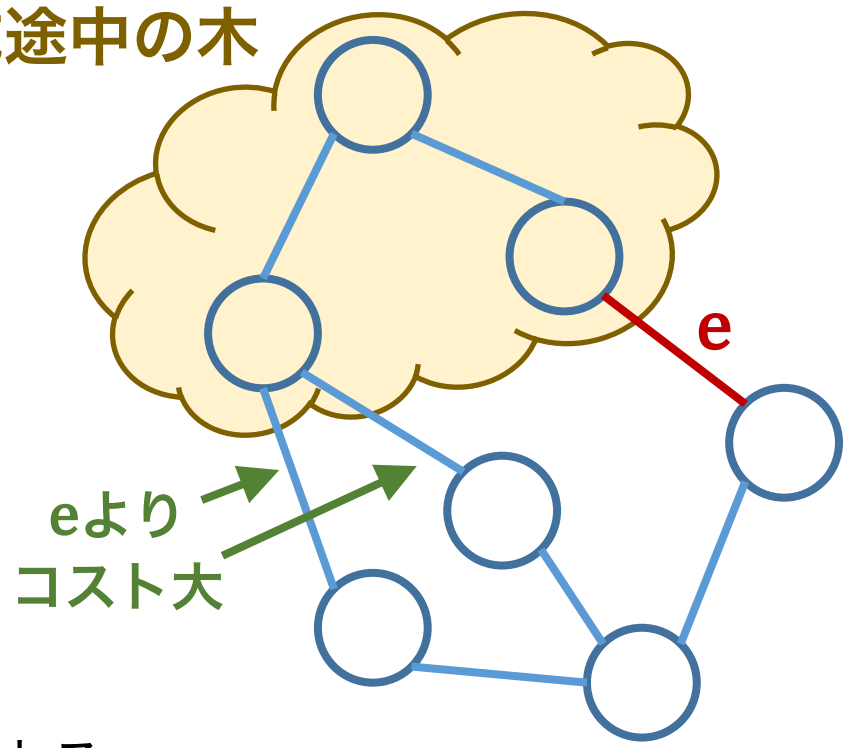
→ **e を選んでOK！**



元のグラフ

プリム法の正当性

元のグラフの全頂点の集合を U
作成途中の木に含まれている頂点の集合を X
X と U＼X を結ぶ最小コストの辺を e とおく



仮定: e を含まない最小全域木 T’ が存在する

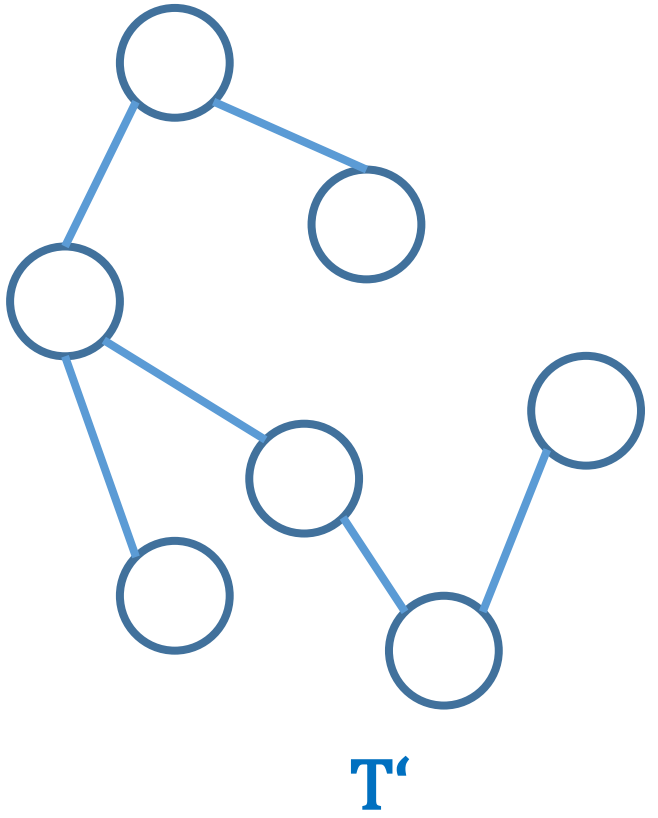
1. T’ に e を加えると、必ず閉路ができる
2. その閉路に含まれる辺には、
   X と U＼X を結ぶ e 以外の辺 e’ が必ず存在する
3. T’ から e’ を取り除いて e を加えても全域木
4. e のコストは e’ 以下なので、e に置き換えても最小全域木

→ **e を選んでOK！**

プリム法の正当性

元のグラフの全頂点の集合を U
作成途中の木に含まれている頂点の集合を X
X と U＼X を結ぶ最小コストの辺を e とおく

仮定: e を含まない最小全域木 T’ が存在する



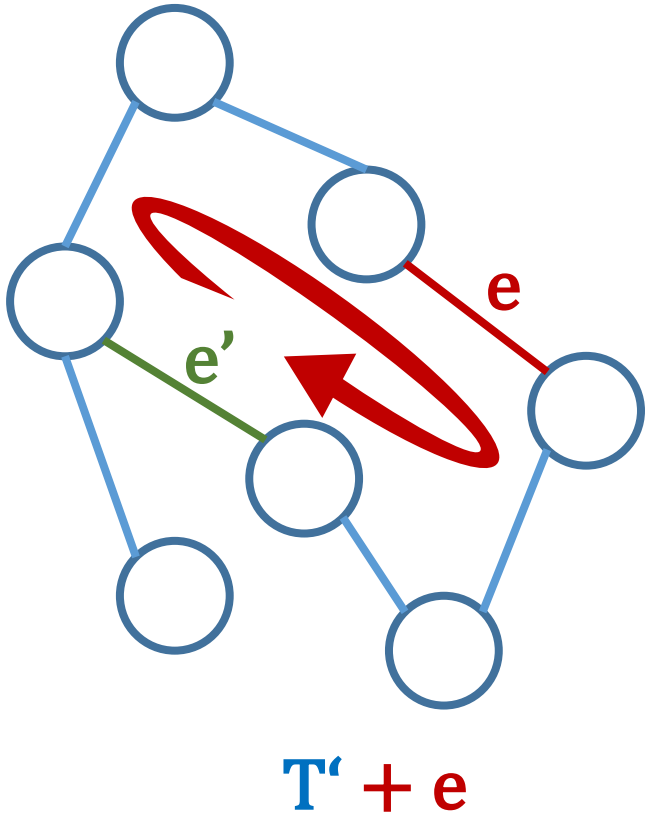
1. T’ に e を加えると、必ず閉路ができる
2. その閉路に含まれる辺には、
   X と U＼X を結ぶ e 以外の辺 e’ が必ず存在する
3. T’ から e’ を取り除いて e を加えても全域木
4. e のコストは e’ 以下なので、e に置き換えても最小全域木

→ **e を選んでOK！**

プリム法の正当性

元のグラフの全頂点の集合を U
作成途中の木に含まれている頂点の集合を X
X と U＼X を結ぶ最小コストの辺を e とおく

仮定: e を含まない最小全域木 T’ が存在する



1. T’ に e を加えると、必ず閉路ができる
2. その閉路に含まれる辺には、
   X と U＼X を結ぶ e 以外の辺 e’ が必ず存在する
3. T’ から e’ を取り除いて e を加えても全域木
4. e のコストは e’ 以下なので、e に置き換えても最小全域木

→ **e を選んでOK！**

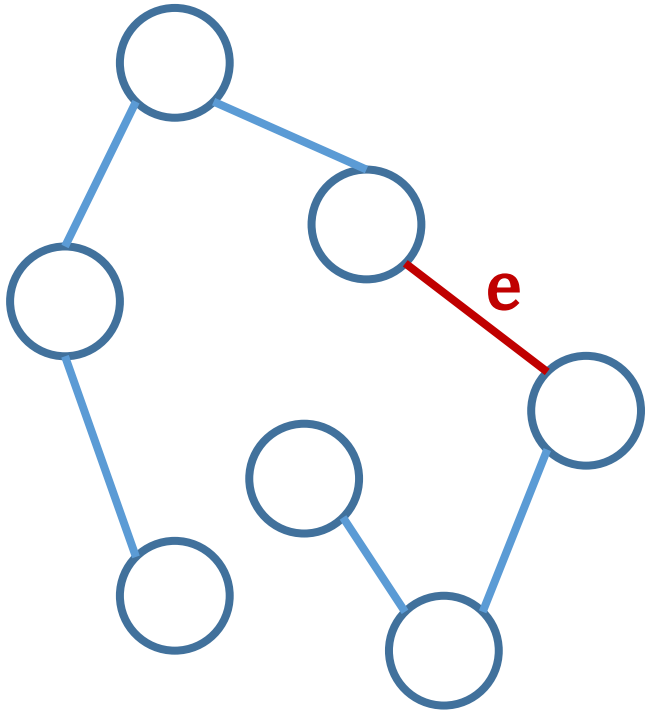
プリム法の正当性

元のグラフの全頂点の集合を U
作成途中の木に含まれている頂点の集合を X
X と U＼X を結ぶ最小コストの辺を e とおく

仮定: e を含まない最小全域木 T’ が存在する

1. T’ に e を加えると、必ず閉路ができる
2. その閉路に含まれる辺には、
   X と U＼X を結ぶ e 以外の辺 e’ が必ず存在する
3. T’ から e’ を取り除いて e を加えても全域木
4. e のコストは e’ 以下なので、e に置き換えても最小全域木

   → **e を選んでOK！**



プリム法の解析

- やっていることは、ダイクストラ法とほぼ同じ

  ダイクストラ法との相違点

  – ダイクストラ法でのコストの計算は
  「その頂点までの最小コスト＋そこから隣までのコスト」だったが、
  プリム法では単純に**「そこから隣までのコスト」**で計算する

  – 待機列から出てきた頂点を**USED**にする

  – 負の辺や負の閉路があっても問題なく動く

- 計算量はダイクストラ法と同じく **O(E log E) 時間**

実装例

```cpp
// プリム法
// v: 頂点数, adjlist: 隣接リスト(first: 辺のコスト, second: 行き先)
int prim(int v, vector<vector<pair<int, int> > > adjlist){
  priority_queue<pair<int, int> > wait;  // すでに作られている木までの最小コスト
  vector<int> result(v, INF);  // 頂点iと木の間の最小コスト
  int length = 0, size = 0;  // 木のコスト, 頂点数

  // とりあえず頂点を1つ待機列に追加
  result[0] = 0;
  wait.push(make_pair(0, 0));

  // 待機列が空でなく、まだすべての頂点を網羅していなければループ
  while(!wait.empty() && size < v){
    // 候補辺のうち最小のものを取り出す
    int newpoint = wait.top().second;
    int newcost = -wait.top().first;
    wait.pop();
    if(result[newpoint] < newcost){ continue; }

    // 出てきた頂点を木に追加
    result[newpoint] = -INF;
    length += newcost;
    size++;

    // 点newpointから接続されているすべて辺を舐める
    for(int i = 0; i ≤ adjlist[newpoint].size(); i++){
      int nextpoint =  adjlist[newpoint][i].second;
      int nextcost = adjlist[newpoint][i].first;
      // より木に近くなった頂点があれば、そこへの枝をqueueに追加
      if(result[nextpoint] > nextcost){
        result[nextpoint] = nextcost;
        wait.push(make_pair(-nextcost, nextpoint));
      }
    }
  }

  // 返り値: 木のコスト
  return length;
}
```

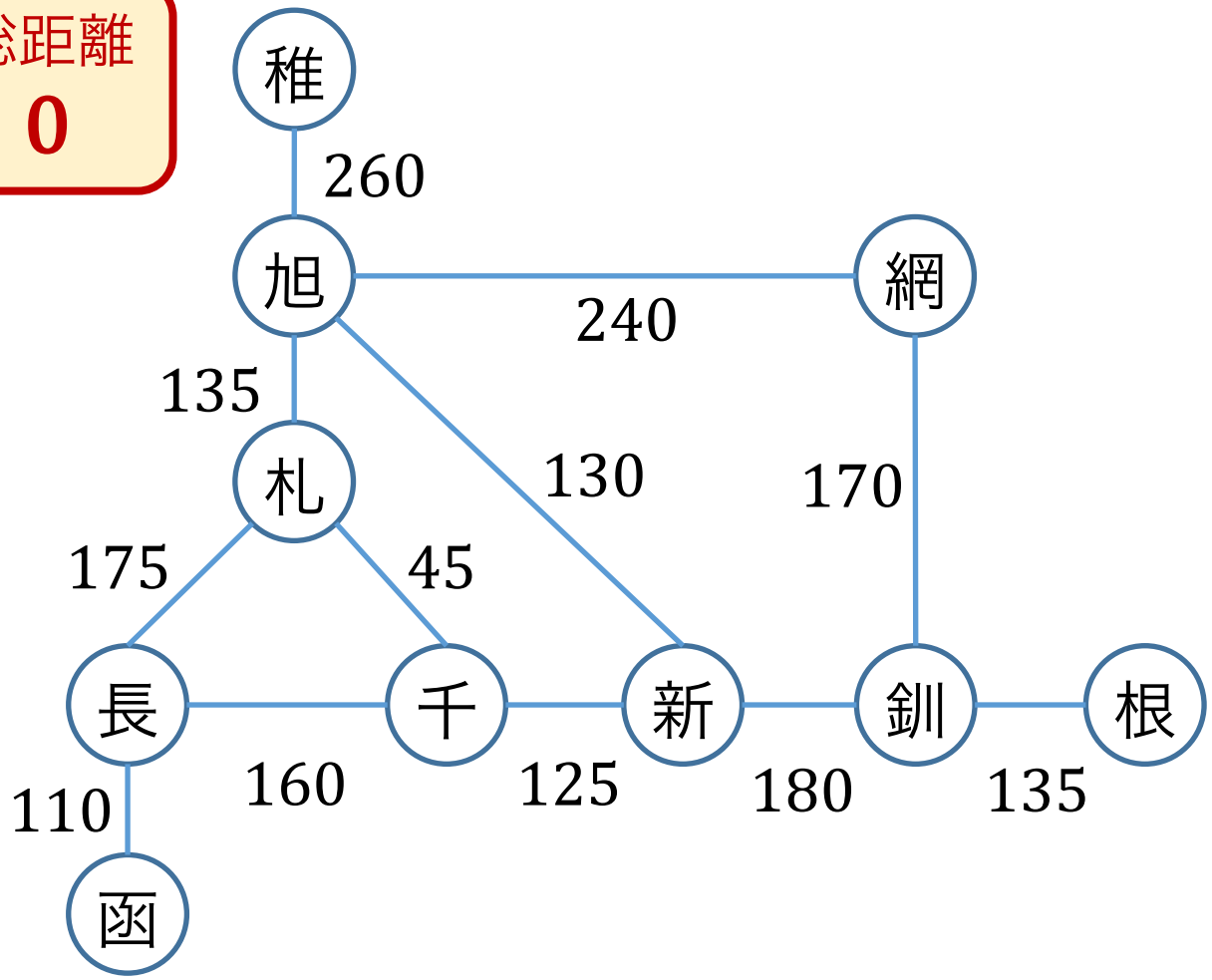
クラスカル法 (Kruskal's algorithm)

- **方針: 「そんな細かいこと考えないで、**
  コスト小さい順に辺を取ってくれば良いじゃん！
  閉路ができないよう気をつけるからさー」

  - グラフ全体の適当なところから辺を取ってくるので、
    途中経過は木とは限らない (森ではある)

コストの更新

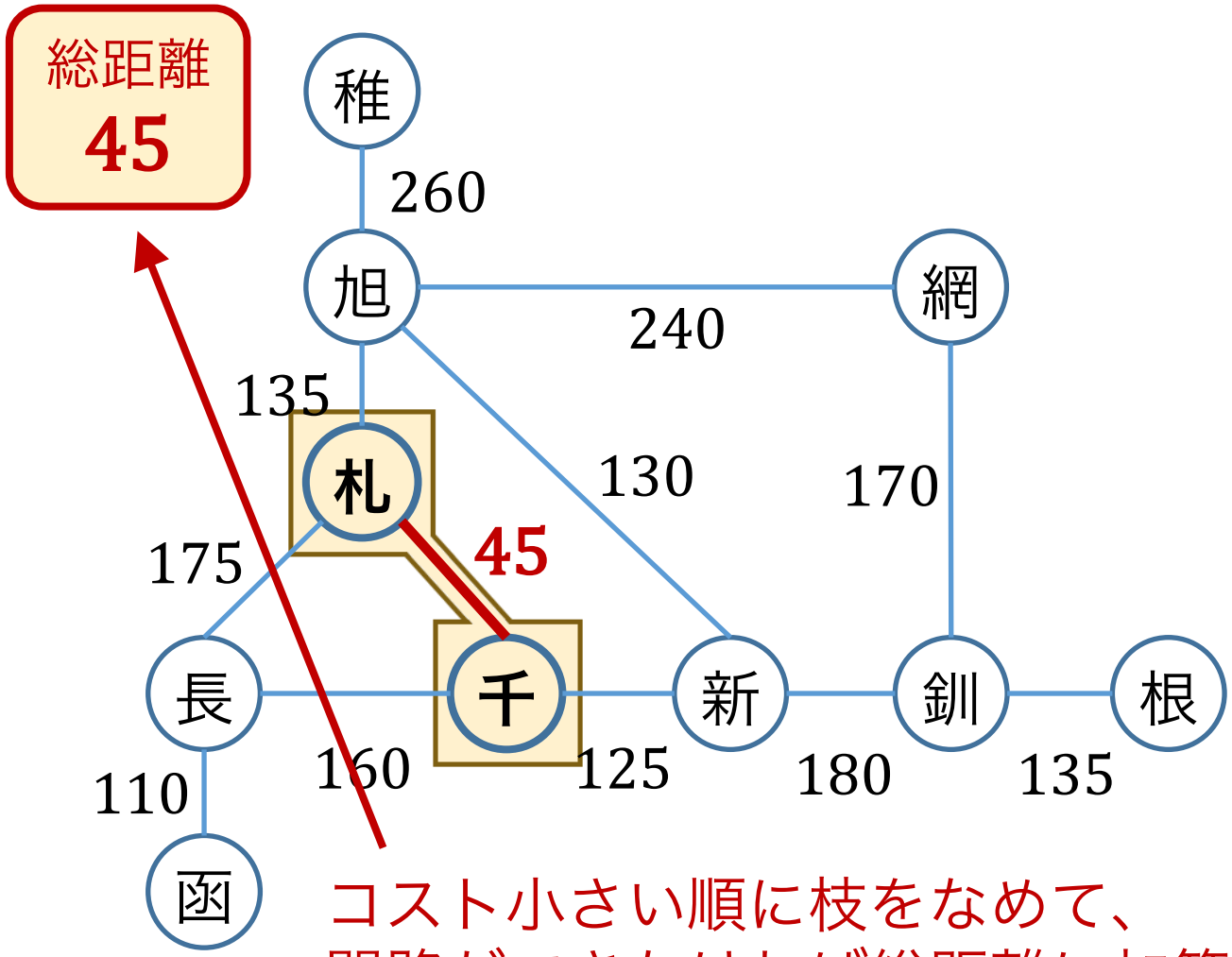
総距離
0



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

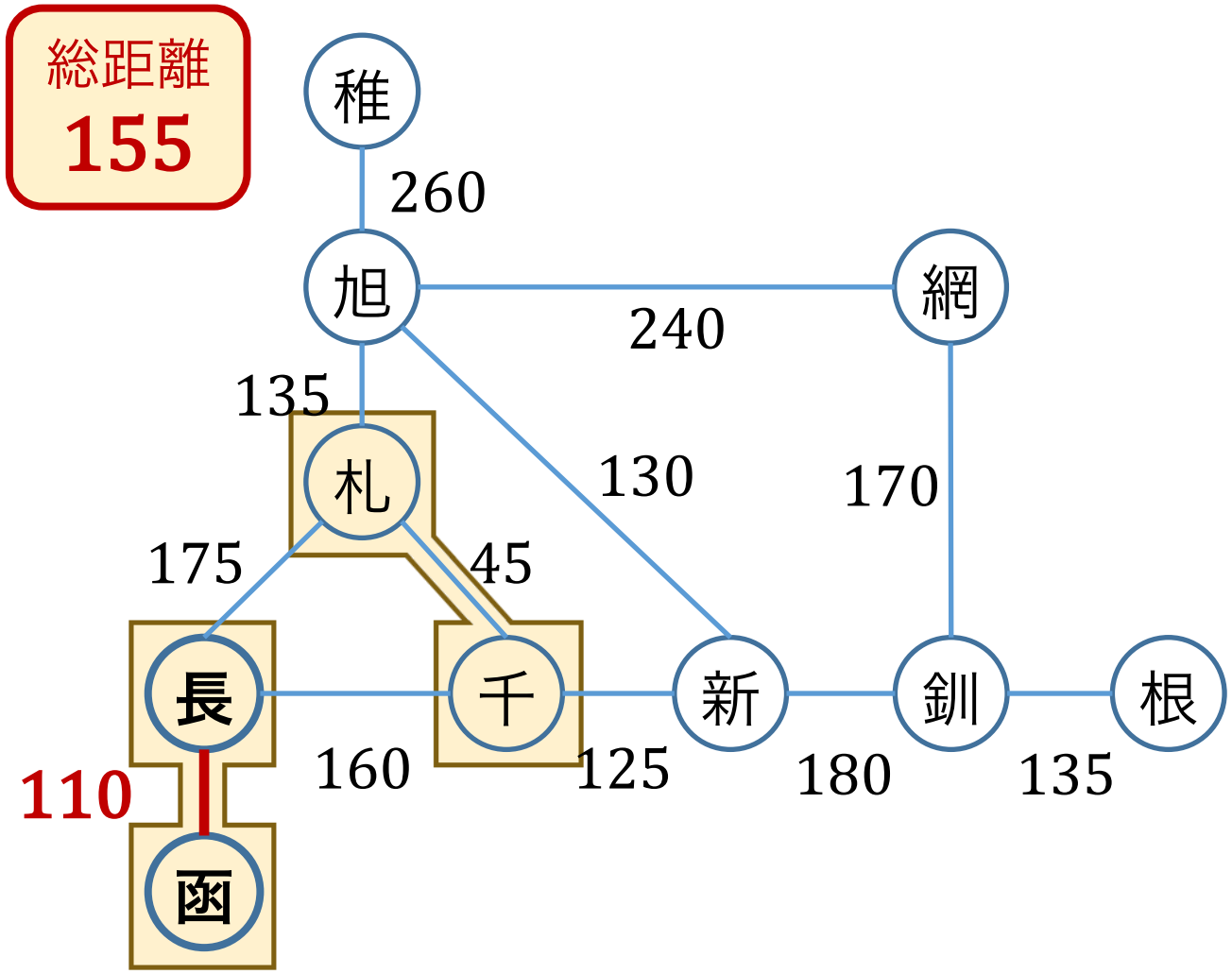
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

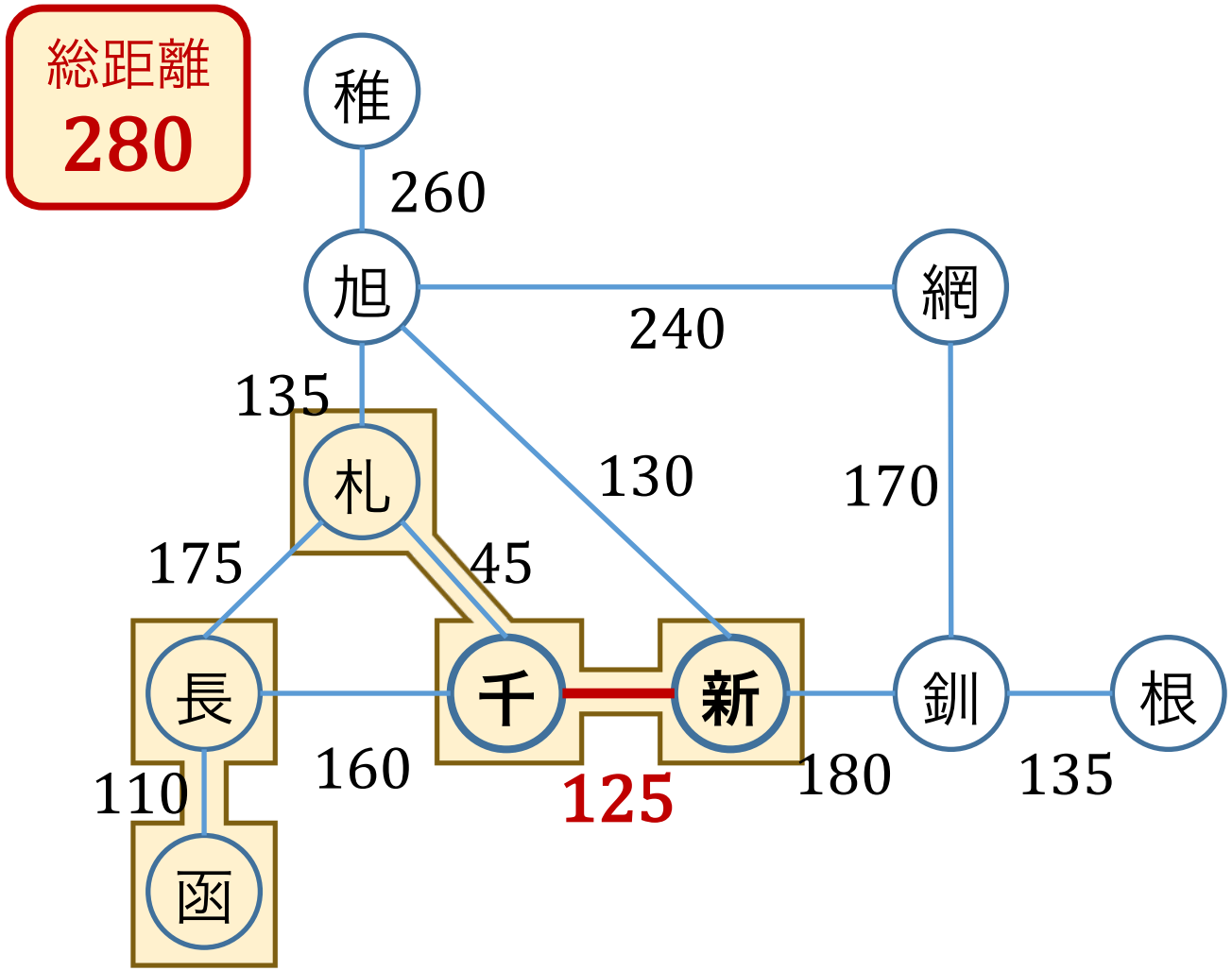
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

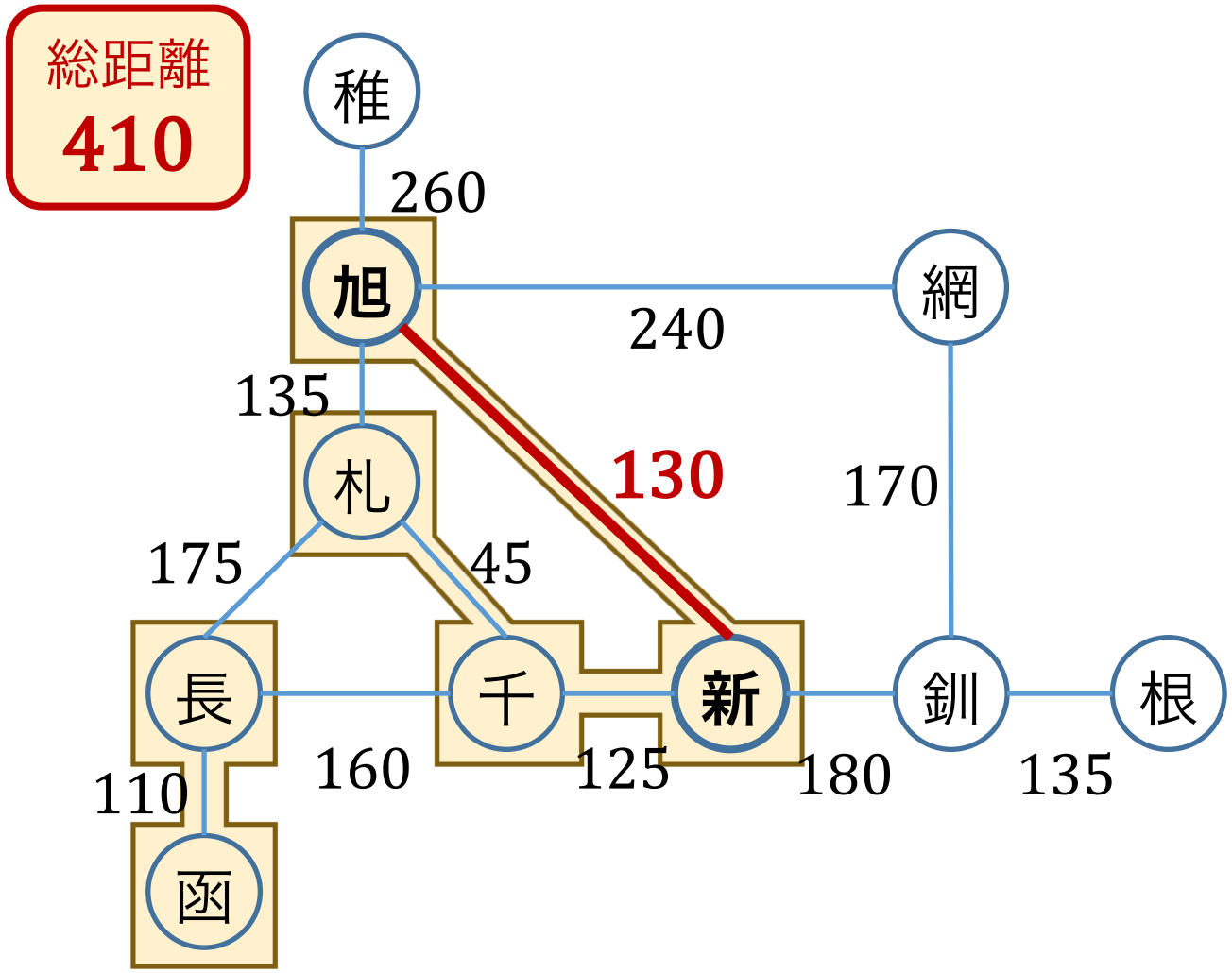
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

コストの更新

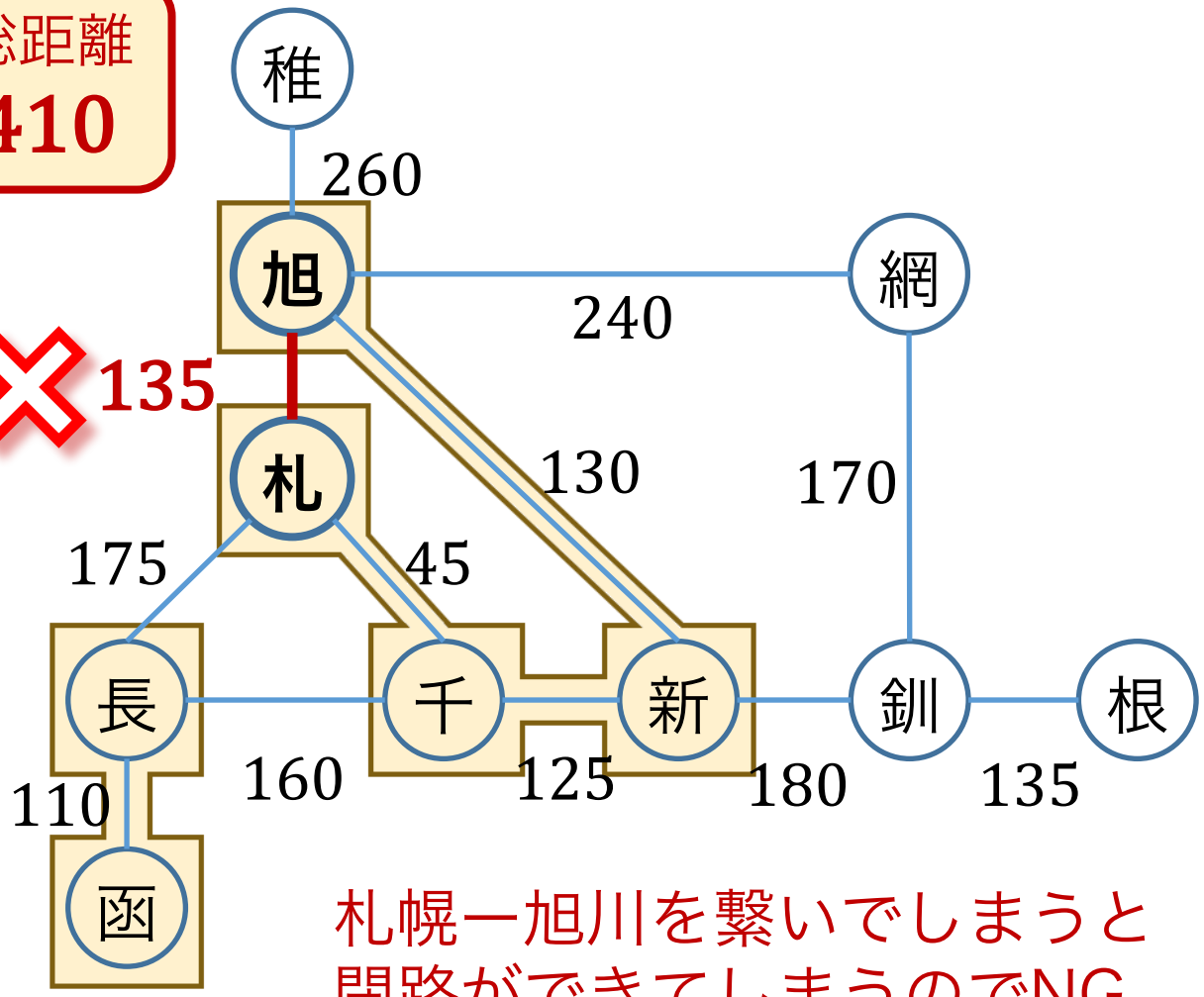


枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

コストの更新

総距離
410

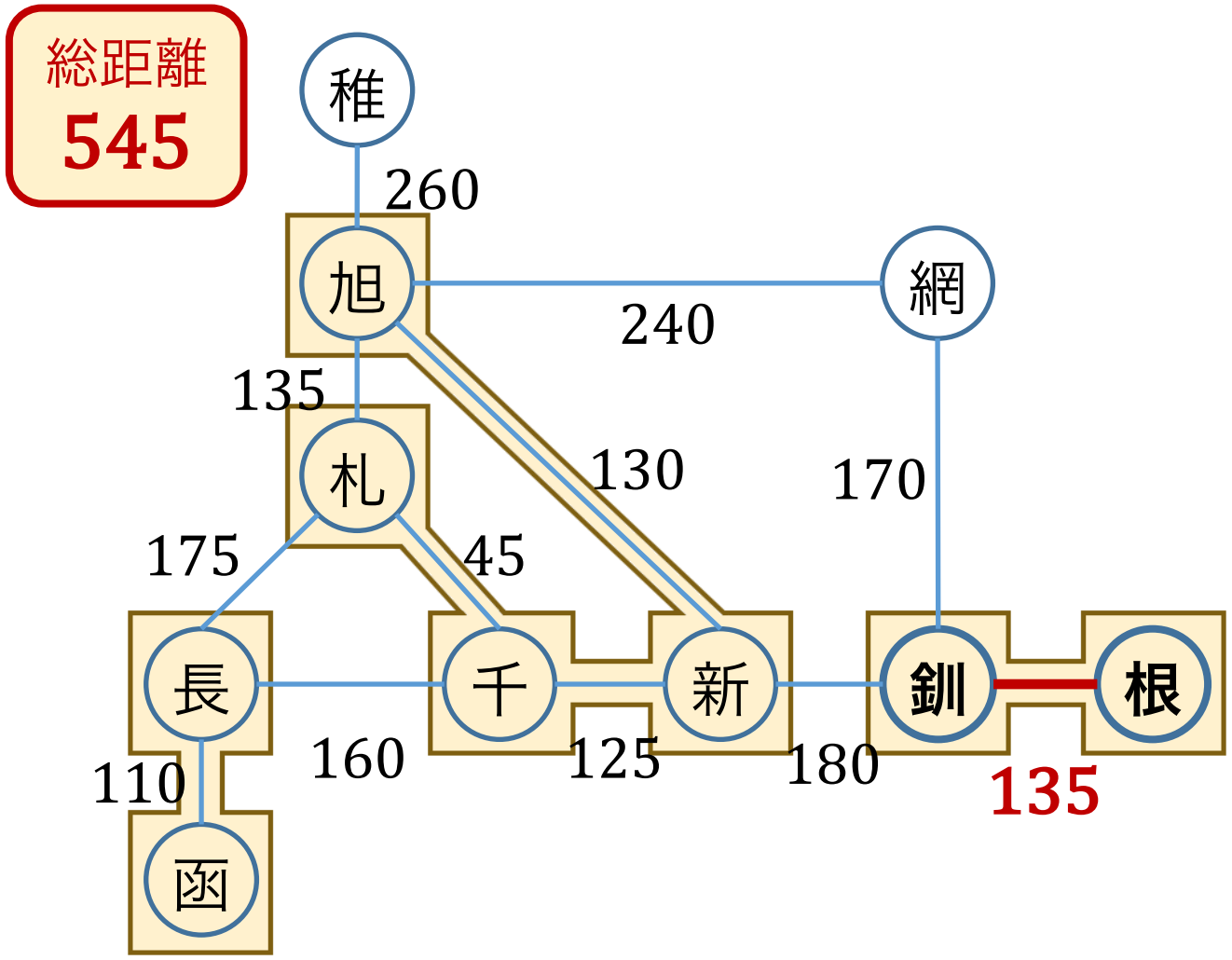


札幌ー旭川を繋いでしまうと
閉路ができてしまうのでNG

枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

コストの更新

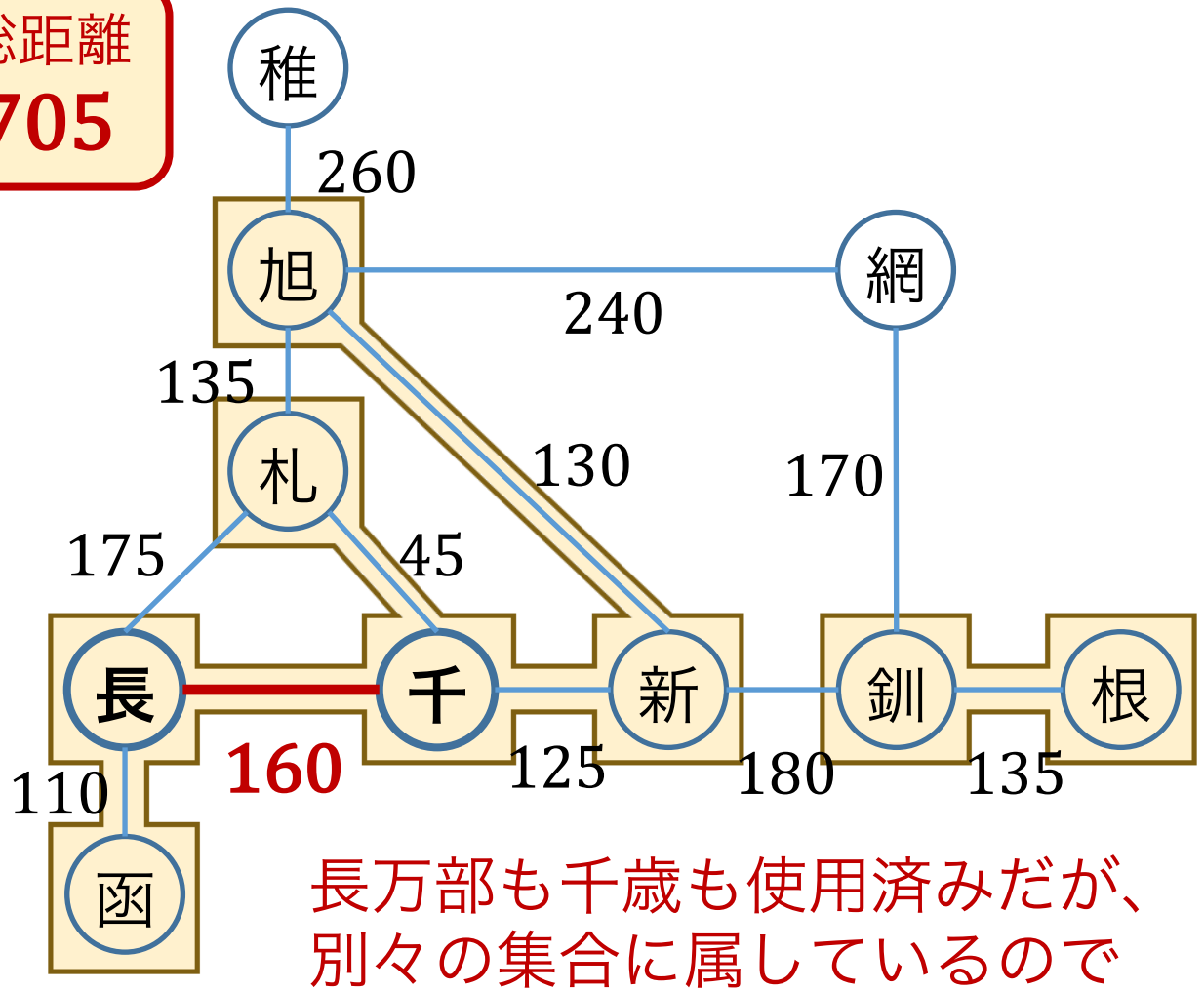


枝一覧 (昇順)

千歳 － 札幌	45
長万部 － 函館	110
千歳 － 新得	125
新得 － 旭川	130
札幌 － 旭川	135
釧路 － 根室	135
長万部 － 千歳	160
網走 － 釧路	170
札幌 － 長万部	175
釧路 － 新得	180
網走 － 旭川	240
旭川 － 稚内	260

コストの更新

総距離
705

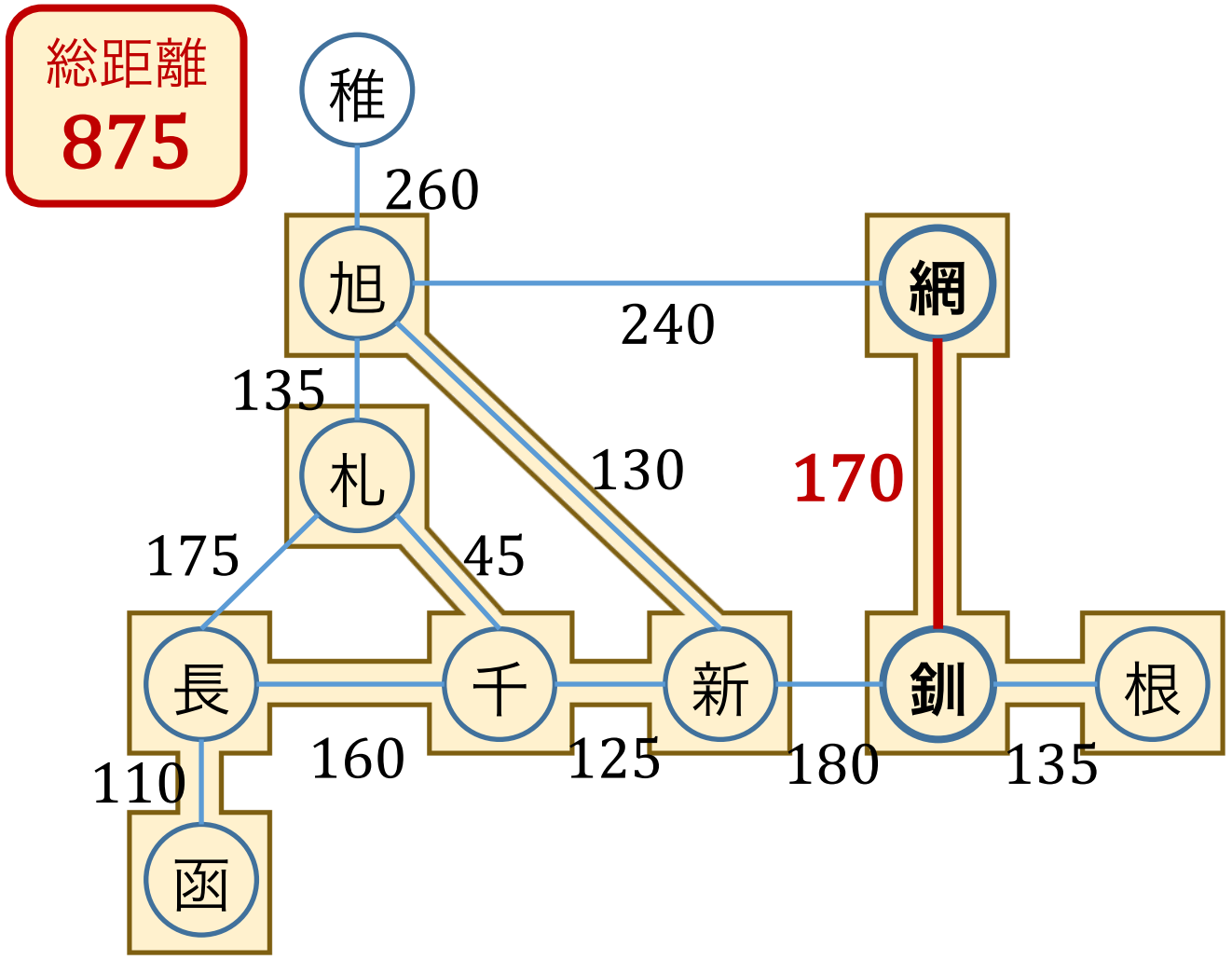


長万部も千歳も使用済みだが、
別々の集合に属しているので
閉路は発生しない → OK

枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

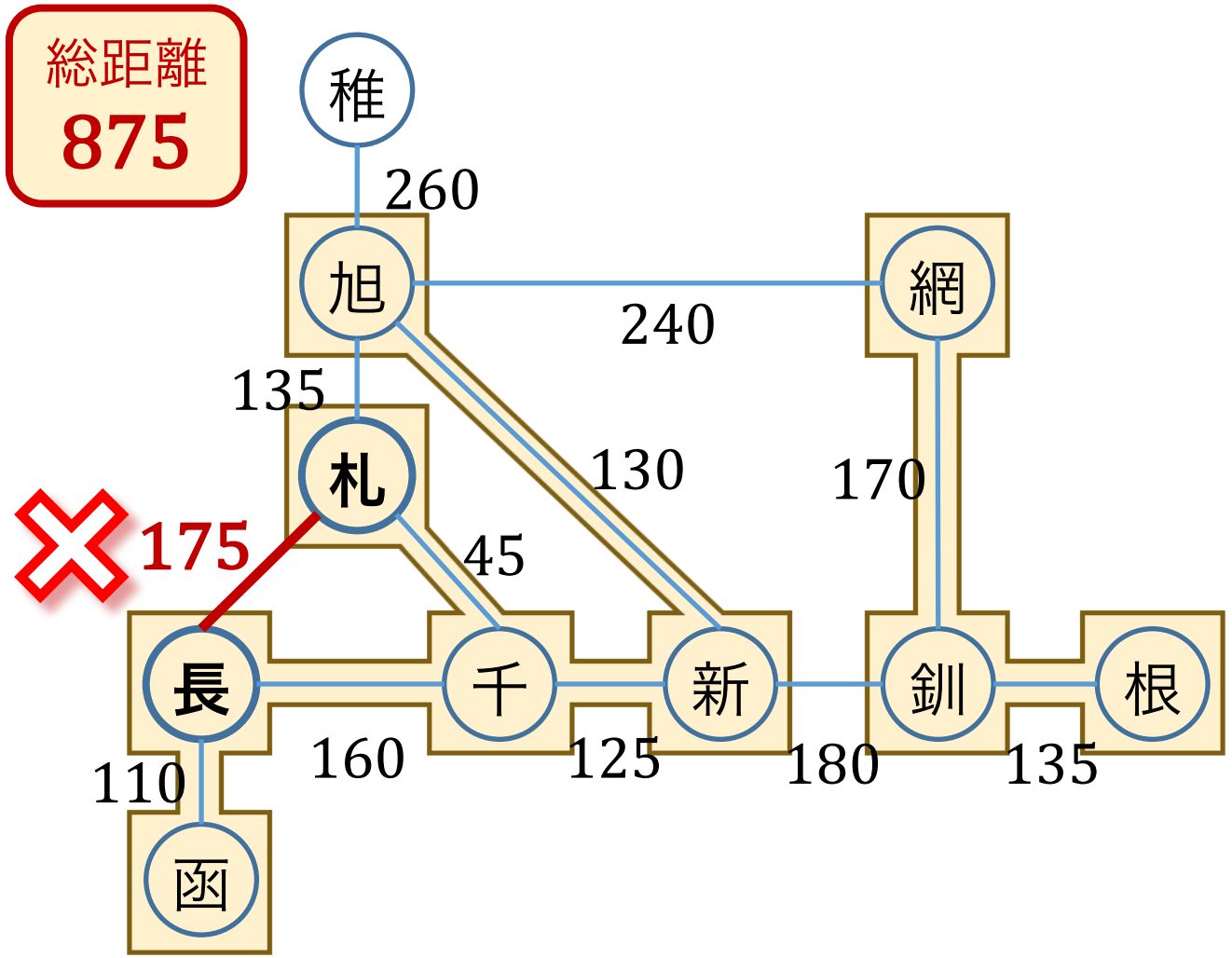
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

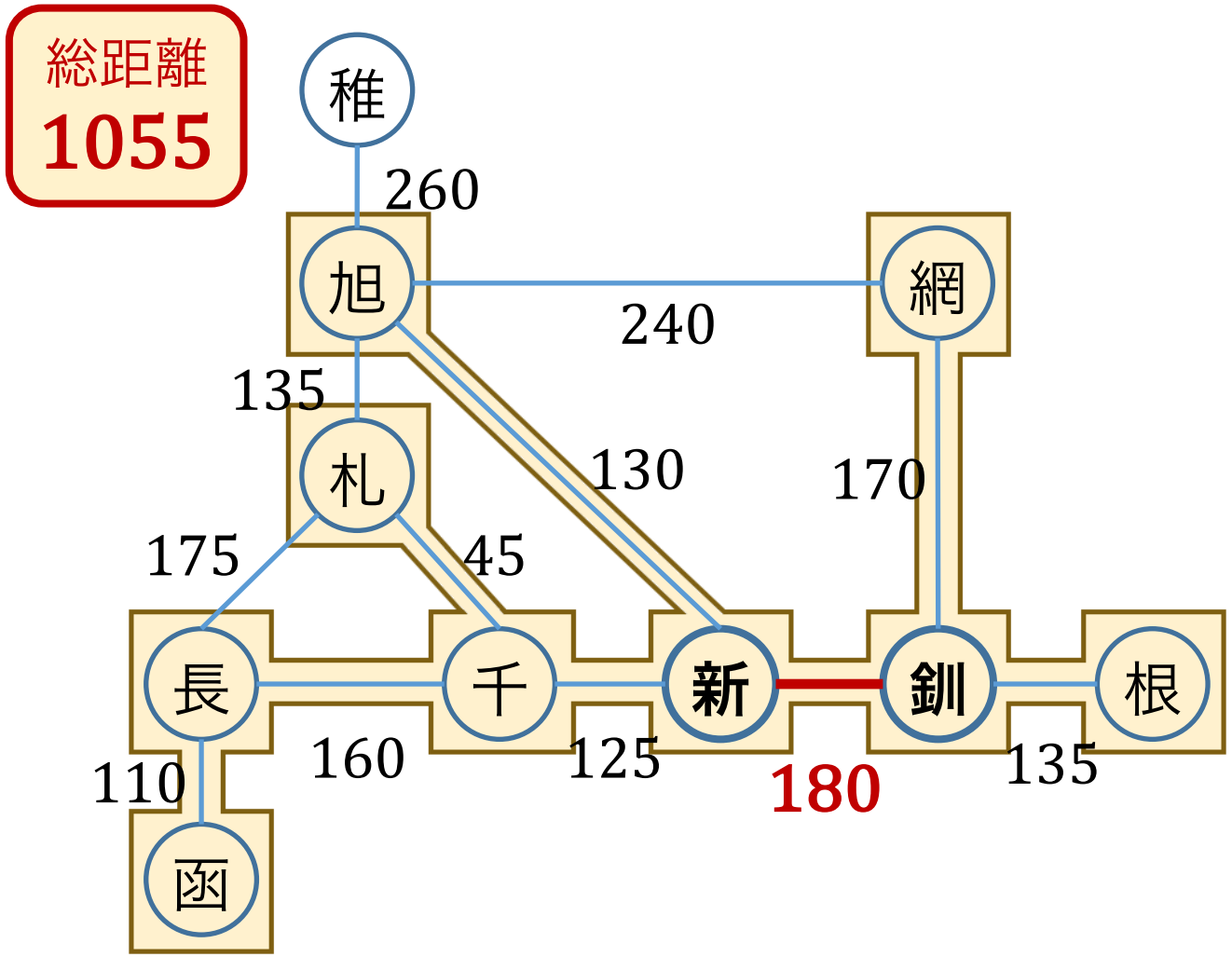
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

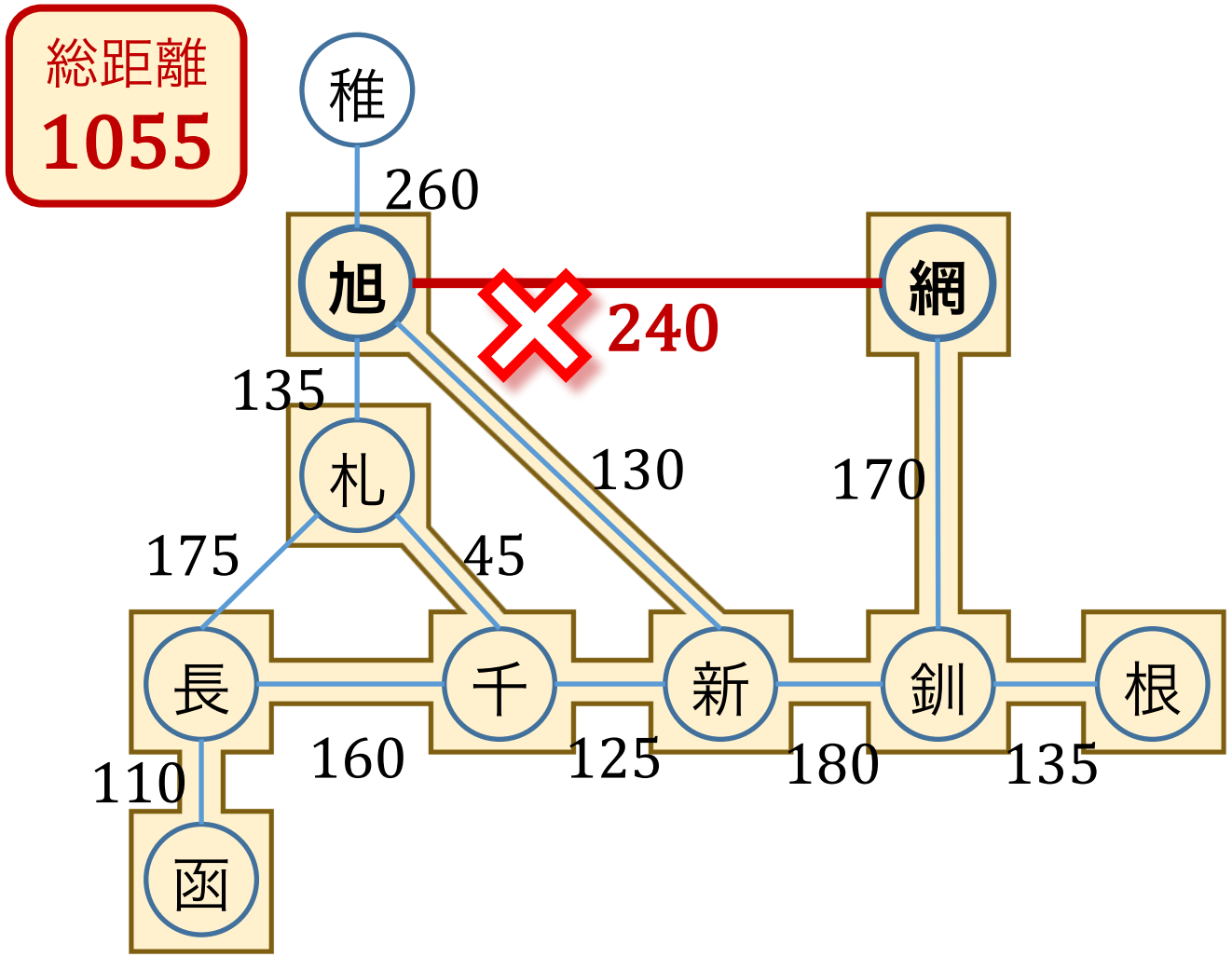
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

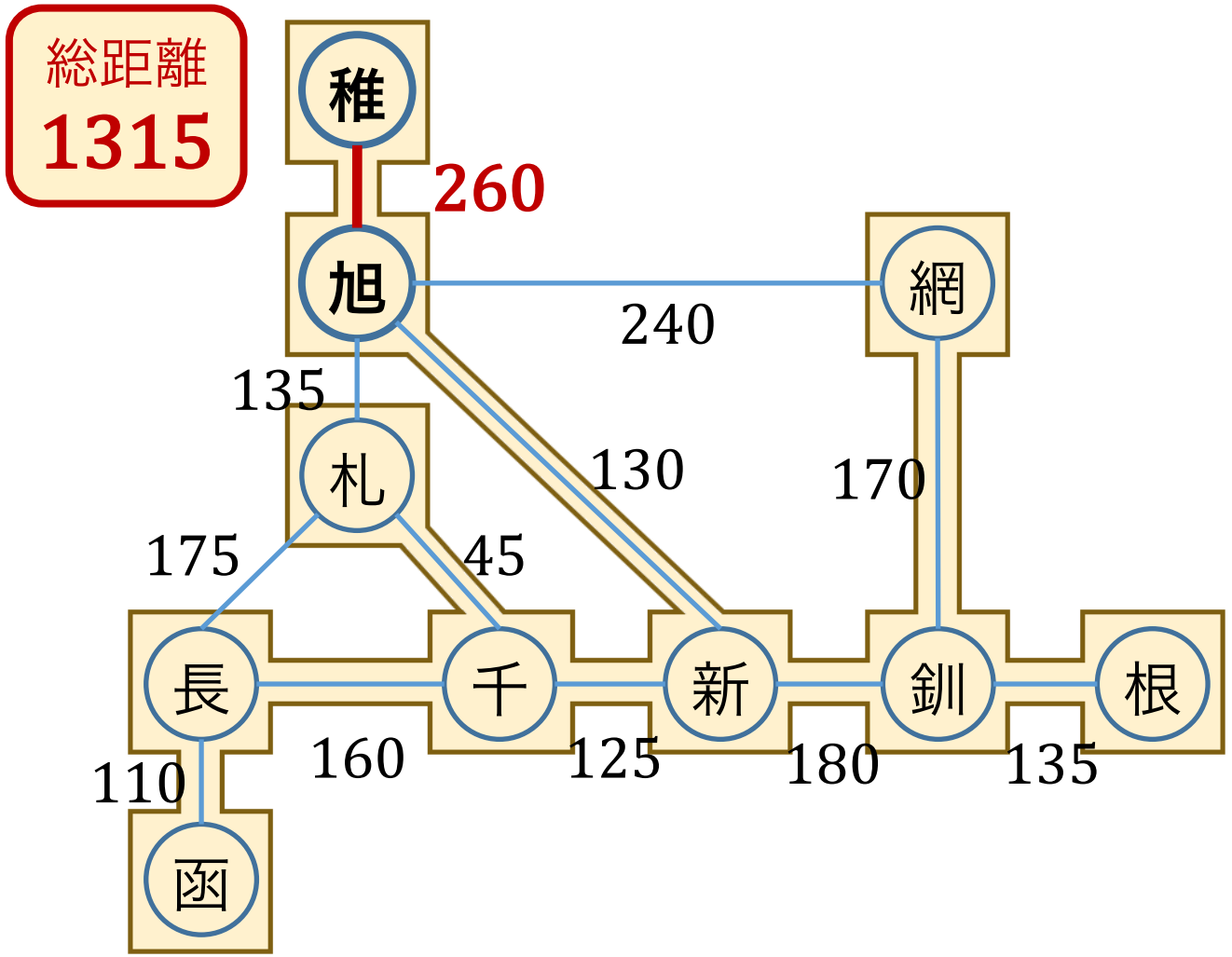
コストの更新



枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

コストの更新

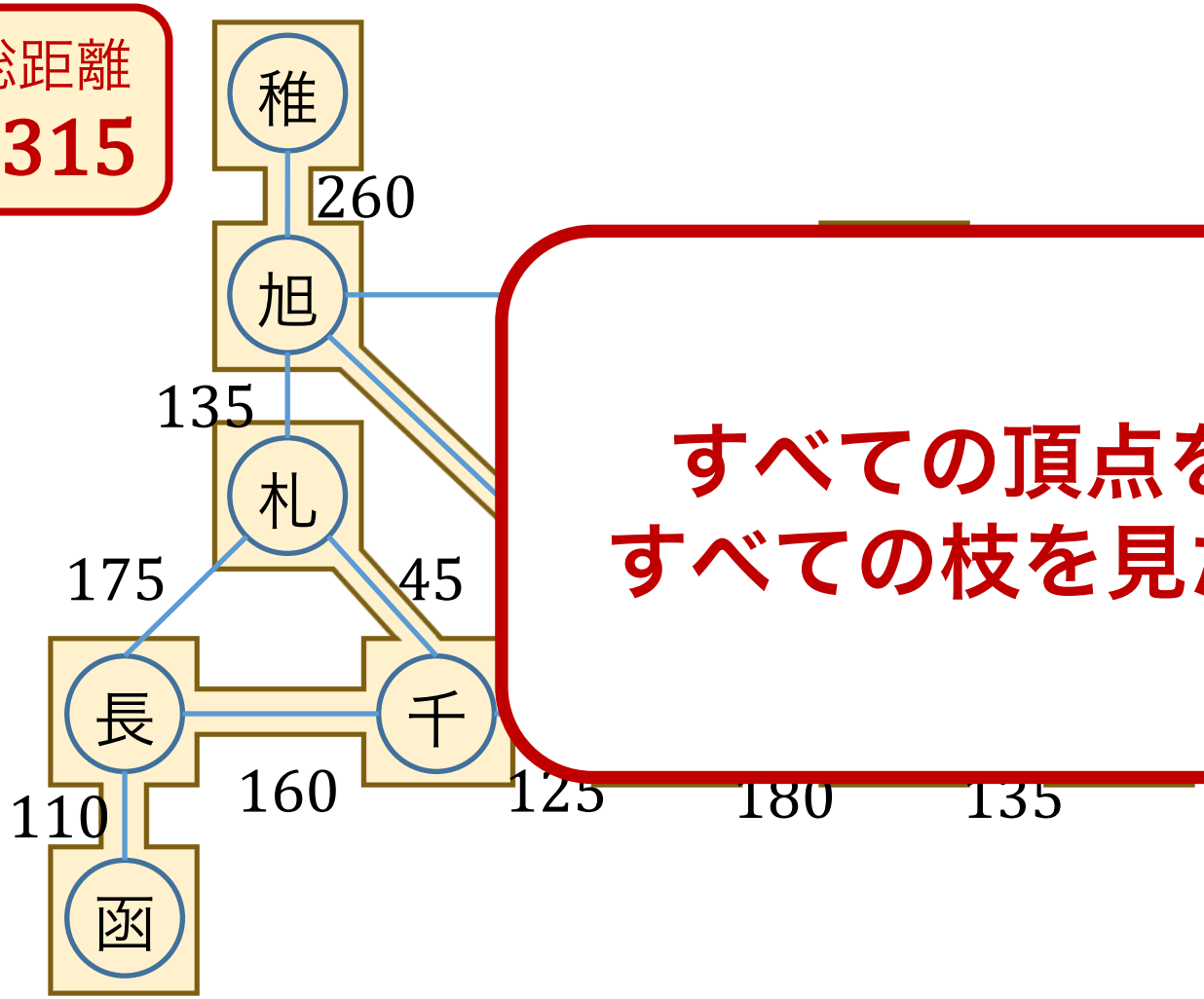


枝一覧 (昇順)

千歳	ー	札幌	45
長万部	ー	函館	110
千歳	ー	新得	125
新得	ー	旭川	130
札幌	ー	旭川	135
釧路	ー	根室	135
長万部	ー	千歳	160
網走	ー	釧路	170
札幌	ー	長万部	175
釧路	ー	新得	180
網走	ー	旭川	240
旭川	ー	稚内	260

コストの更新

総距離
1315



すべての頂点を連結 or
すべての枝を見たら終了！

枝一覧 (昇順)

千歳 － 札幌	45
長万部 － 函館	110
千歳 － 新得	125
新得 － 旭川	130
	135
	135
	160
	170
	175
釧路 － 新得	180
網走 － 旭川	240
旭川 － 稚内	260

クラスカル法の解析

- 前処理:
  すべての枝を昇順にソート
  → **$O(E \log E)$ 時間**

クラスカル法の解析

- 前処理:
  すべての枝を昇順にソート
  → **$O(E \log E)$ 時間**

- 本処理:
  枝を順々に見ていき、閉路ができなければ総距離を加算
  つまり、**閉路ができるかどうかの判定回数が $O(E)$ 回**

クラスカル法の解析

- 前処理:
  すべての枝を昇順にソート
  → **O(E log E) 時間**

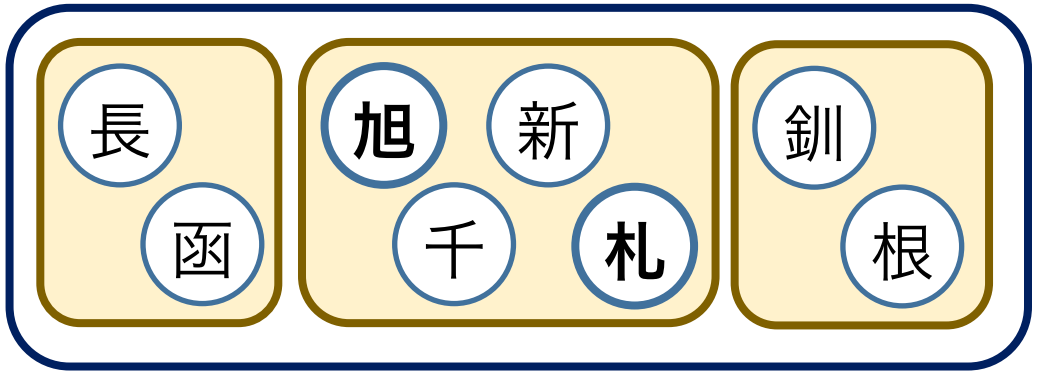
- 本処理:
  枝を順々に見ていき、閉路ができなければ総距離を加算
  つまり、**閉路ができるかどうかの判定回数が O(E) 回**

- **じゃぁ閉路の判定ってどのくらいかかるの？**
  というか、どうすれば判定できるの？

今、こんな感じのものが欲しい

例１

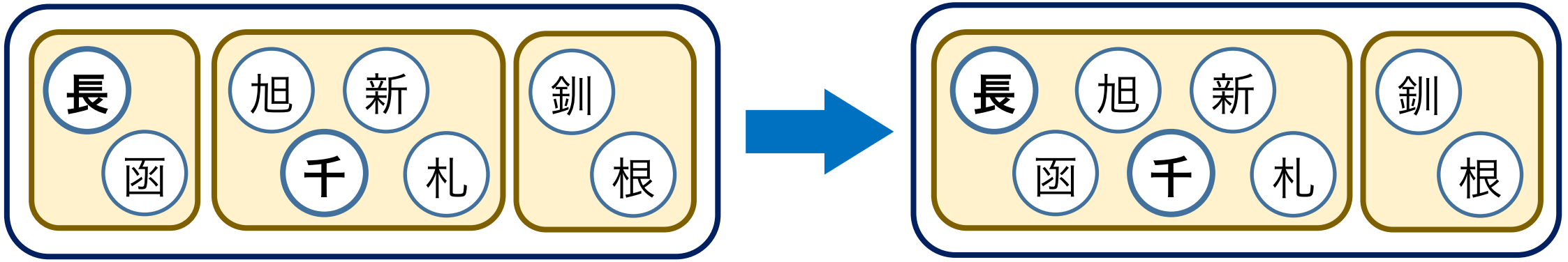
Q. 旭川と札幌は
同じ集合？



A. 同じだよー

例２

長万部たちと千歳たちを統合して！

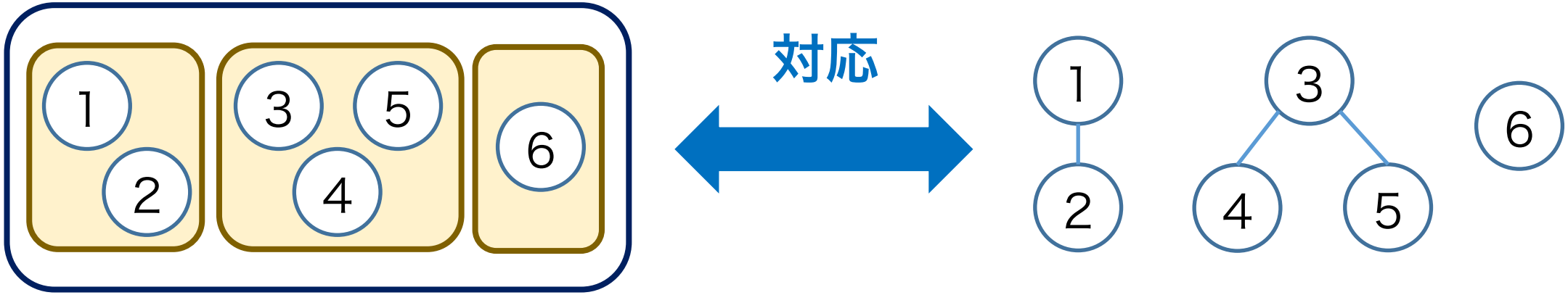


＼できたよー／

Union-Find Tree

- ここで着目するのが、**Union-Find Tree**というデータ構造
  素集合森 (disjoint-set forest) とも呼ばれている

  データをいくつかの木に分割して保持している！



Union-Find Treeの動き

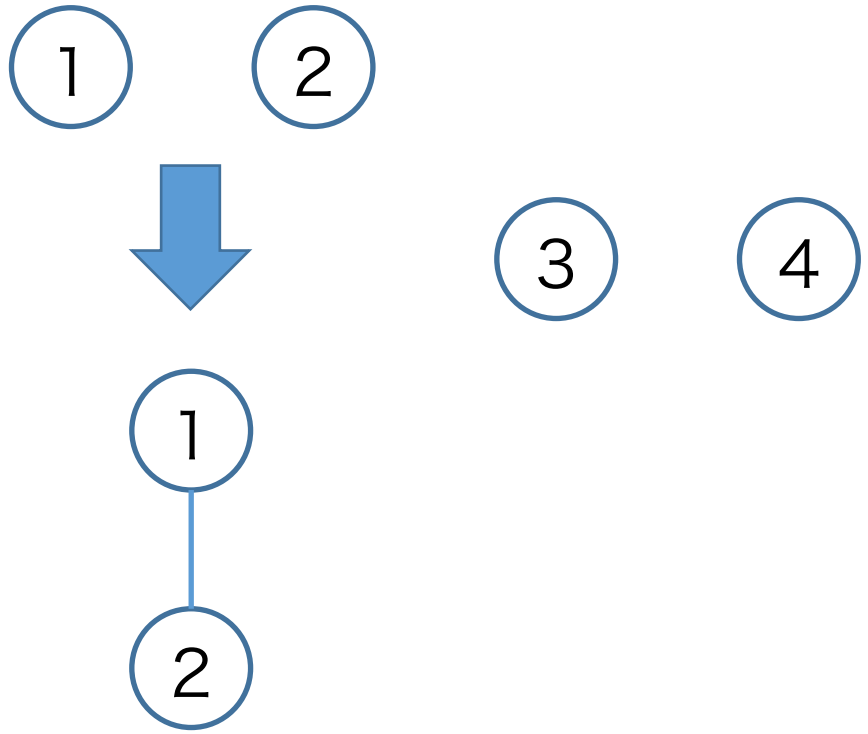
- **初期化**
  最初はバラバラ

Union-Find Treeの動き

- **初期化**
  最初はバラバラ

- **まとめる**
  片方の木の根から
  もう一方の木の根に
  辺を張る

Q. 1の木と2の木を統合して！

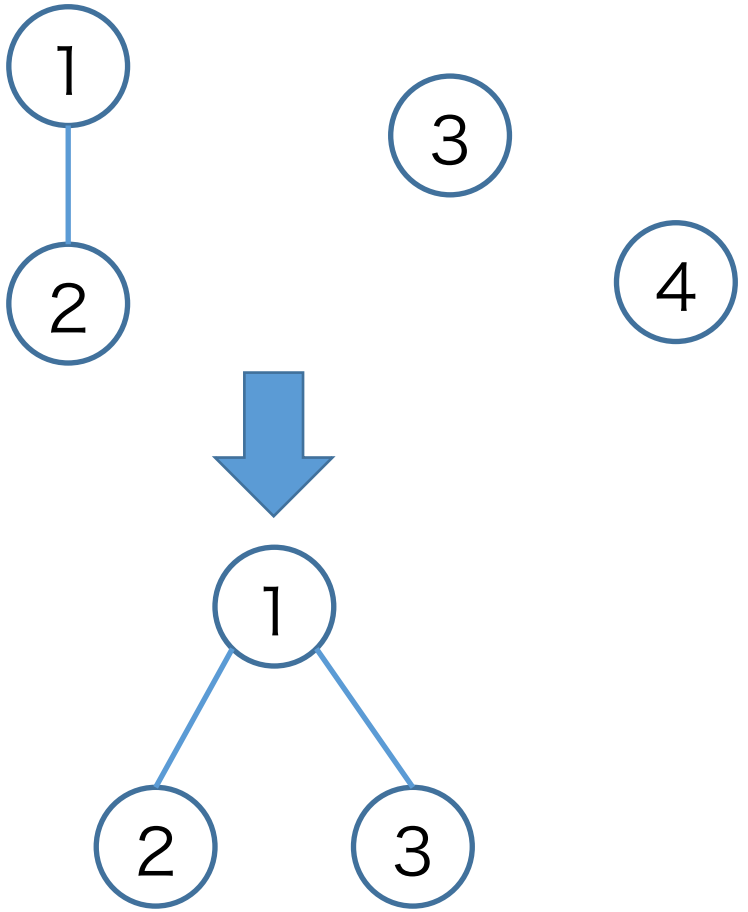


Union-Find Treeの動き

- **初期化**
  最初はバラバラ

- **まとめる**
  片方の木の根から
  もう一方の木の根に
  辺を張る

Q. 1の木と3の木を統合して！



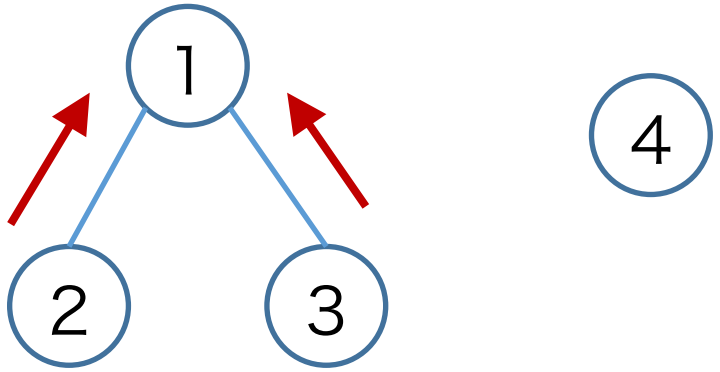
Union-Find Treeの動き

- **初期化**
  最初はバラバラ

- **まとめる**
  片方の木の根から
  もう一方の木の根に
  辺を張る

- **同じ木かどうか判定**
  根まで上っていって
  同じかどうか見る

Q. 2と3って同じ木？



A. 同じだよー

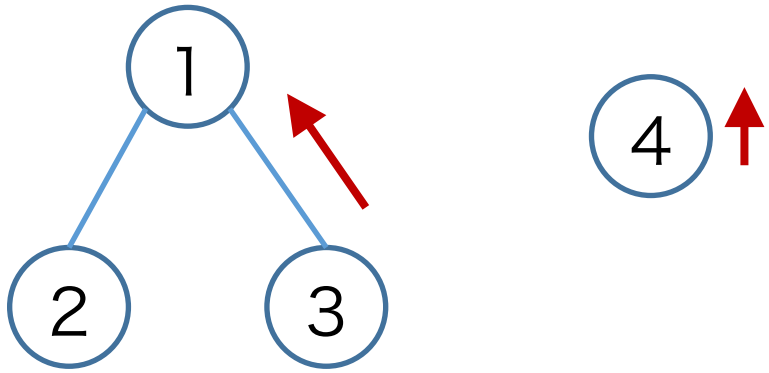
Union-Find Treeの動き

- **初期化**
  最初はバラバラ

- **まとめる**
  片方の木の根から
  もう一方の木の根に
  辺を張る

- **同じ木かどうか判定**
  根まで上っていって
  同じかどうか見る

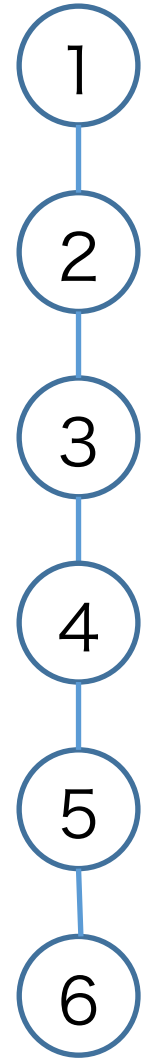
Q. 3と4って同じ木？



A. 違うよー

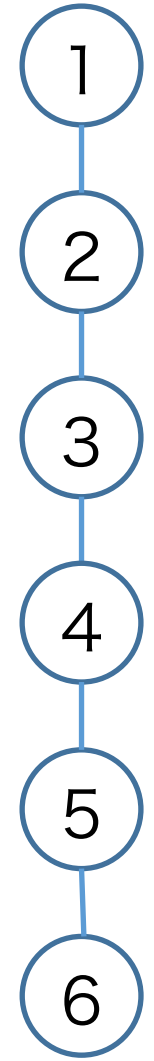
Union-Find Treeの計算量

この実装での最悪ケースを考えると、
こういう木ができる可能性がある



Union-Find Treeの計算量

この実装での最悪ケースを考えると、
こういう木ができる可能性がある

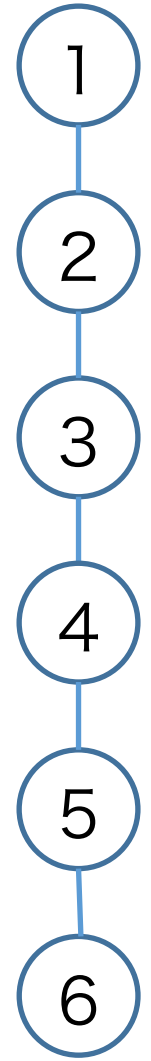


この場合、根まで辿り着くためには
最大で要素数回だけ上る必要がある

→ **各処理にかかる計算量は O(n) 時間** (n: 要素数)

Union-Find Treeの計算量

この実装での最悪ケースを考えると、
こういう木ができる可能性がある



この場合、根まで辿り着くためには
最大で要素数回だけ上る必要がある

→ **各処理にかかる計算量は $O(n)$ 時間**　(n: 要素数)

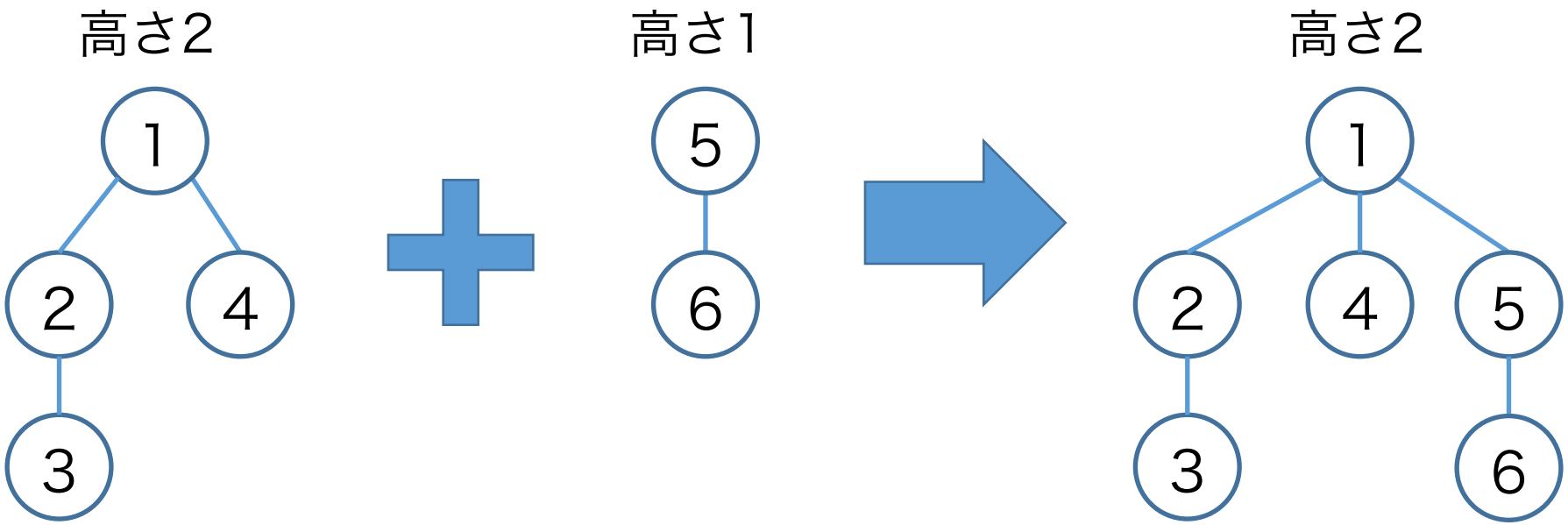
もっと早くできないかな？

まとめる時も判定する時も、**根さえ分かればいい**
→ **根まで高速に到達できるように工夫しよう！**

Union-Find Treeの2つの工夫

- **工夫その1: ランク**

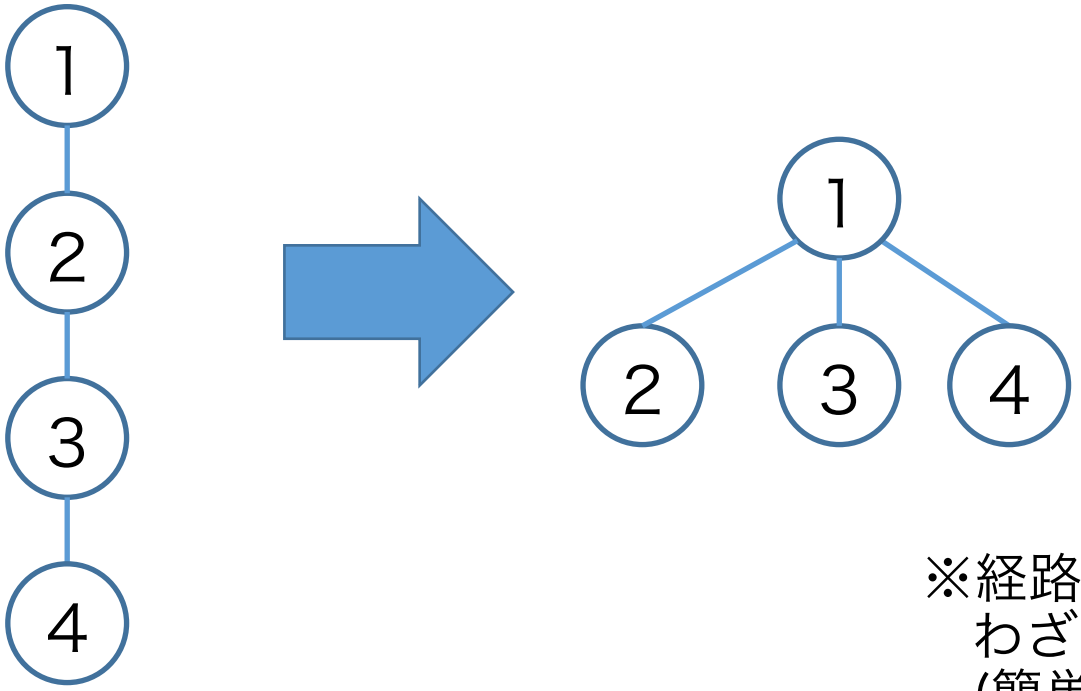
  木の高さの情報を保持しておいて、
  低い方を高い方の下にぶら下げることにする



Union-Find Treeの2つの工夫

- **工夫その2: 経路圧縮**

  再帰的に根を調べるたびに、**調べた辺を根に直接繋ぎ直す！**



※経路圧縮をすると高さが変化するが、
わざわざその情報を書き換えたりはしない
(簡単のため放っておく)

Union-Find Treeの計算量

- この2つの工夫をすると、
  各処理にかかる計算量は $O(\alpha(n))$ 時間　(n: 要素数)

Union-Find Treeの計算量

- この2つの工夫をすると、
  各処理にかかる計算量は $O(\alpha(n))$ 時間　(n: 要素数)

  - **$\alpha(n)$ って何？**
    「逆アッカーマン関数」というもので、**ほぼ定数**

例: $\alpha\left(2^{2^{2^{2^{2^{2}}}}} - 3\right) = 4$

Union-Find Treeの計算量

- この2つの工夫をすると、
  ~~**各処理にかかる計算量は $O(\alpha(n))$ 時間**~~ (n: 要素数)

  - **$\alpha(n)$ って何？**
    「逆アッカーマン関数」というもので、**ほぼ定数**

    例: $\alpha\left(2^{2^{2^{2^{2^{2}}}}} - 3\right) = 4$

→ **各処理にかかる計算量は ほぼ $O(1)$ 時間**

クラスカル法の解析に戻ると……

- 前処理:
  すべての枝を昇順にソート
  → **$O(E \log E)$ 時間**

- 本処理:
  枝を順々に見ていき、閉路ができなければ総距離を加算
  つまり、**閉路ができるかどうかの判定回数が $O(E)$ 回**

- **じゃぁ閉路の判定ってどのくらいかかるの？**

クラスカル法の解析に戻ると……

- 前処理:
  すべての枝を昇順にソート
  → **$O(E \log E)$ 時間**

- 本処理:
  枝を順々に見ていき、閉路ができなければ総距離を加算
  つまり、**閉路ができるかどうかの判定回数が $O(E)$ 回**

- **じゃぁ閉路の判定ってどのくらいかかるの？**
  → **$O(1)$ 時間**

全体として $O(E \log E) + O(E) = O(E \log E)$ 時間

```cpp
// UnionFind木 クラスの定義
class UnionFindTree{
public:
  UnionFindTree();
  UnionFindTree(int size);
  void init(int size);
  void unite(int a, int b);
  bool same(int a, int b);
  bool isConnected(void);
private:
  int m_size, unite_count;
  vector<int> parent;
  vector<int> rank;
  int find(int a);
};

// コンストラクタ
UnionFindTree::UnionFindTree(){};
UnionFindTree::UnionFindTree(int size){ init(size); };

// UF木の初期化
void UnionFindTree::init(int size){
  m_size = size;
  unite_count = 0;
  parent.clear();
  rank.clear();
  for(int i = 0; i < size; i++){
    parent.push_back(i);
    rank.push_back(0);
  }
  return;
}

// UF木で集合を連結する
void UnionFindTree::unite(int a, int b){
  int x = find(a);
  int y = find(b);
  // すでに繋がっていたら何もしない
  if(x == y){ return; }

  unite_count++;
  if(rank[x] < rank[y]){ parent[x] = y; }
  else{
    parent[y] = x;
    if(rank[x] == rank[y]){ rank[x]++; }
  }
  return;
}

// UF木で同じ集合かどうかを判定する
bool UnionFindTree::same(int a, int b){
  return find(a) == find(b);
}

// UF木ですべての頂点が同じ木かどうかを判定する
bool UnionFindTree::isConnected(void){
  return unite_count == m_size - 1;
}

// UF木の根を見つける
int UnionFindTree::find(int a){
  if(parent[a] != a){ parent[a] = find(parent[a]); }
  return parent[a];
}
```

```cpp
// クラスカル法 前処理
// v: 頂点数, adjlist: 隣接リスト(first: 辺のコスト, second: 行き先)
vector<pair<int, pair<int, int> > > kruskal_pre(int v, vector<vector<pair<int, int> > > adjlist){
  vector<pair<int, pair<int, int> > > edge; // first: 辺のコスト, second: 両端の頂点
  for(int i = 0; i < v; i++)
    for(int j = 0; j ≤ adjlist[i].size(); j++)
      edge.push_back(make_pair(adjlist[i][j].first, make_pair(i, adjlist[i][j].second)));
  sort(edge.begin(), edge.end());

  return edge; // 返り値: ソート済み枝リスト
}

// クラスカル法 本処理
// v: 頂点数, edge: ソート済み枝リスト(first: 辺のコスト, second: 両端の頂点)
int kruskal(int v, vector<pair<int, pair<int, int> > > edge){
  int length = 0; // 木のコスト
  UnionFindTree uf(v); // UF木の宣言と初期化

  // すべての辺を見終わるか、すべての頂点が連結されるまでループ
  for(int i = 0; i ≤ edge.size() && !uf.isConnected(); i++){
    // i番目に小さい枝をみる
    int cost = edge[i].first;
    int a = edge[i].second.first;
    int b = edge[i].second.second;
    // 枝の両端が同じ集合だったら、読み飛ばし
    if(uf.same(a, b)){ continue; }

    // UF木を連結して、コストを更新
    uf.unite(a, b);
    length += cost;
  }

  return length; // 返り値: 木のコスト
}
```

プリム vs クラスカル

- 普通に最小全域木を求めるだけなら、どちらを使ってもOK
  Union-Find Treeの実装がない分、**プリムの方が書くのは楽**

プリム vs クラスカル

- 普通に最小全域木を求めるだけなら、どちらを使ってもOK
  Union-Find Treeの実装がない分、**プリムの方が書くのは楽**

- ただし、何度も何度も最小全域木を求め直す場合や、
  そもそも元のグラフが非連結な場合などの特殊なケースでは、
  クラスカルの方が速かったり楽だったりすることがある

  - 何度も求め直す場合:
    クラスカルは、一度前処理さえしてしまえば、本処理は $O(E)$ で実行できる

  - 元のグラフが非連結な場合:
    プリムは工夫しないと途中で処理が止まってしまうが、
    クラスカルはそのままの実装で「最小全域森」を求められる

まとめ

最短経路と最小全域木の問題を倒すテクニックを理解できた！

めざせ、グラフマスター！

- あとは実際に問題を解いて、自分のものにするだけ！

- 付録として
  今日の内容に関連するプロコンの問題をまとめたので、
  たくさん解いて力をつけよう！

付録

- 最短経路問題 その1
  - http://abc007.contest.atcoder.jp/tasks/abc007_3
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0179
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0121
- 最短経路問題 その2
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0200
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0117
  - http://qupc2014.contest.atcoder.jp/tasks/qupc2014_d
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2585
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2334
- 最小全域木
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0180
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0072
  - http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=2511
  - http://judge.u-aizu.ac.jp/onlinejudge/cdescription.jsp?cid=ICPCOOC2014&pid=F