教科書輪講

13章　重み付きグラフ

石田研究室　M1　種部俊孝

重み付きグラフ

12章では、無向グラフと有向グラフについて扱った

13章では、重み付きグラフについて扱う

名前	特徴
無向グラフ	エッジに方向がないグラフ
有向グラフ	エッジに方向があるグラフ
重み付き無向グラフ	エッジに重み（値）があり、方向がないグラフ
重み付き有向グラフ	エッジに重み（値）があり、方向があるグラフ

目次

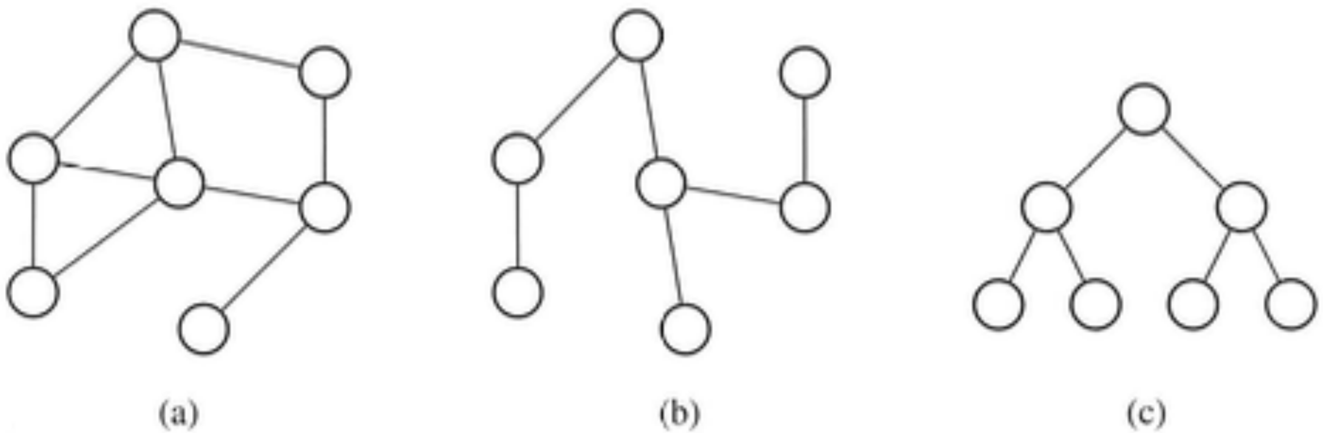
1. 最小全域木(重み付き無向グラフ)
2. 単一始点最短経路(重み付き有向グラフ)

目次

1. 最小全域木（重み付き無向グラフ）
2. 単一始点最短経路（重み付き有向グラフ）

木(tree)

木は閉路を持たないグラフのこと



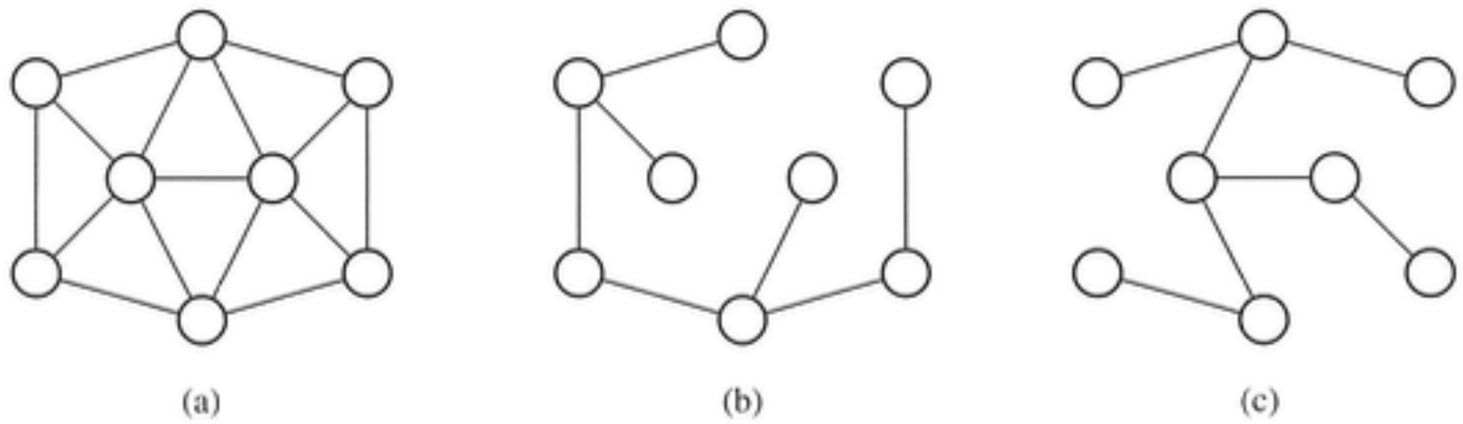
(a)は閉路を持つので木ではなく、(b)と(c)は閉路がないので木

木では、ある頂点rから頂点vまで、必ず1通りの経路が存在する

全域木

グラフG(V,E)の全域木G=(V’, E’)

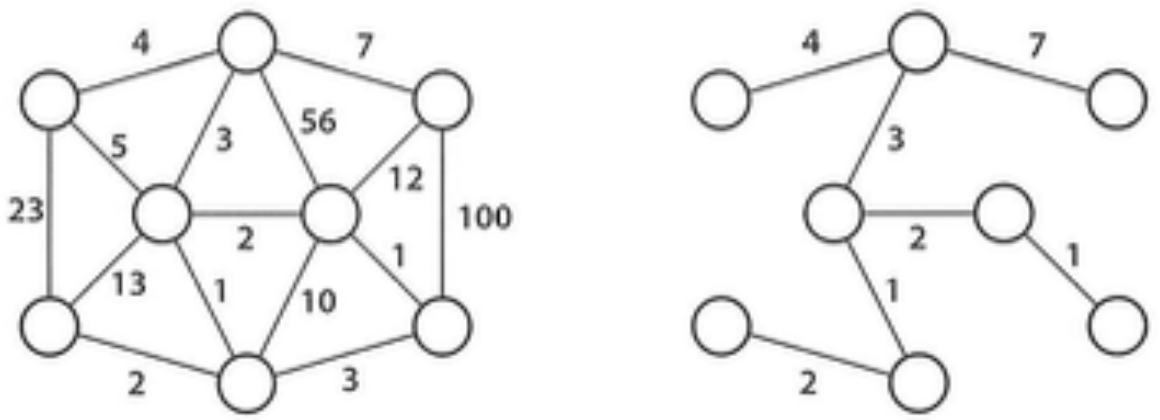
- グラフの全ての頂点Vを含む部分グラフ（V=V’)であり、できるだけ多くの辺を持つもの
- グラフの全域木は、深さ優先探索または幅優先探索で求めることができる



グラフの全域木は1つとは限らず、(a)のグラフは(b)や(c)のような複数の全域木を持つ

最小全域木

グラフの全域木の中で、辺の重みの総和が最小のもの

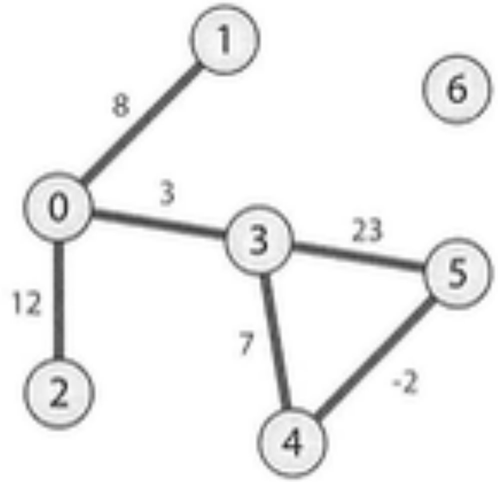


左の重み付きグラフの最小全域木は右のようになる

重み付き無向グラフの隣接行列

重み付き無向グラフの隣接行列では、

頂点iと頂点jの間に重さwの辺がある場合、M[i][j]とM[j][i]の値をwとする



j

i

	0	1	2	3	4	5	6
0	∞	8	12	3	∞	∞	∞
1	8	∞	∞	∞	∞	∞	∞
2	12	∞	∞	∞	∞	∞	∞
3	3	∞	∞	∞	7	23	∞
4	∞	∞	∞	7	∞	-2	∞
5	∞	∞	∞	23	-2	∞	∞
6	∞	∞	∞	∞	∞	∞	∞

辺がない状態として非常に大きな値を設定すると都合が良いことが多い

プリムのアルゴリズム

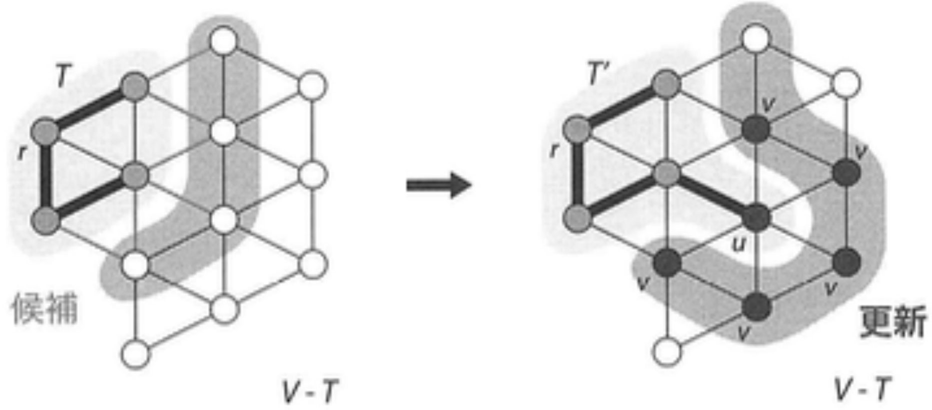
グラフG=(V,E)の最小全域木(MST)を求める代表的なアルゴリズムの1つ

グラフG=(V,E)の頂点全体の集合をV、MSTに属する頂点の集合をTとする

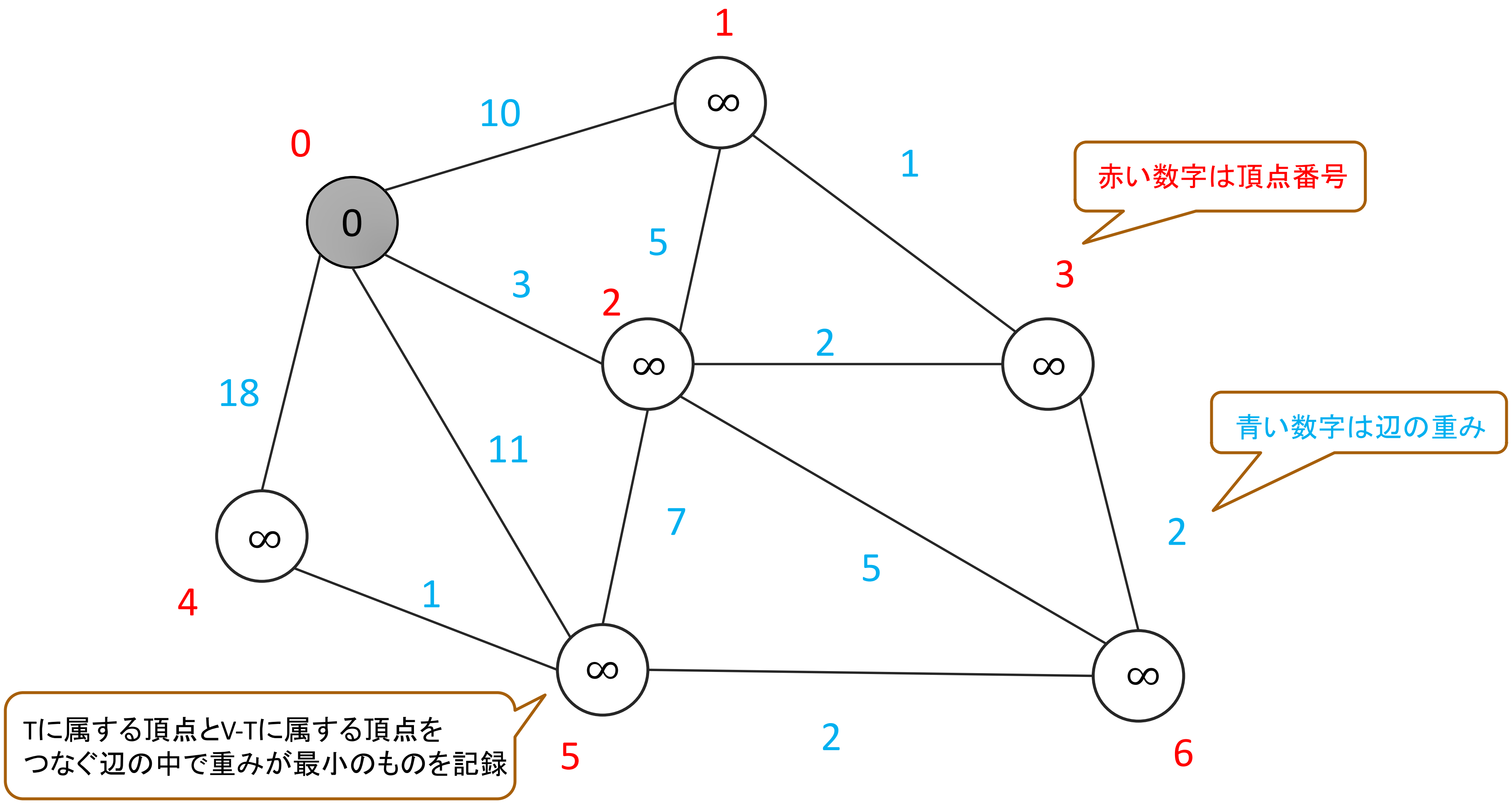
1. Gから任意の頂点rを選び、それをMSTのルートとして、Tに追加する
2. 次の処理をT=Vになるまで繰り返す

   Tに属する頂点とV-Tに属する頂点をつなぐ辺の中で、重みが最小である辺(p_u, u)を選び、

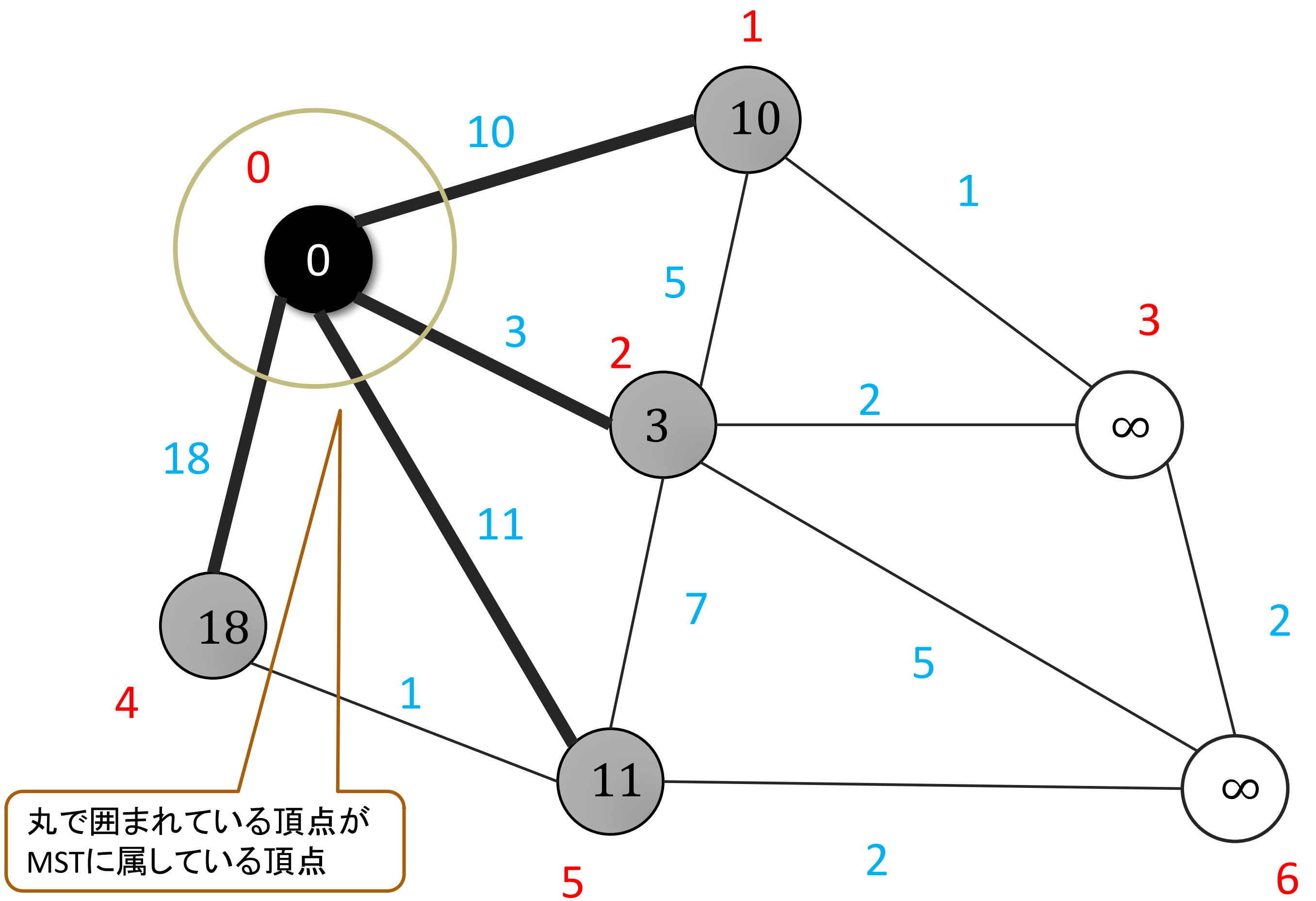
   それをMSTの辺とし、uをTに追加する



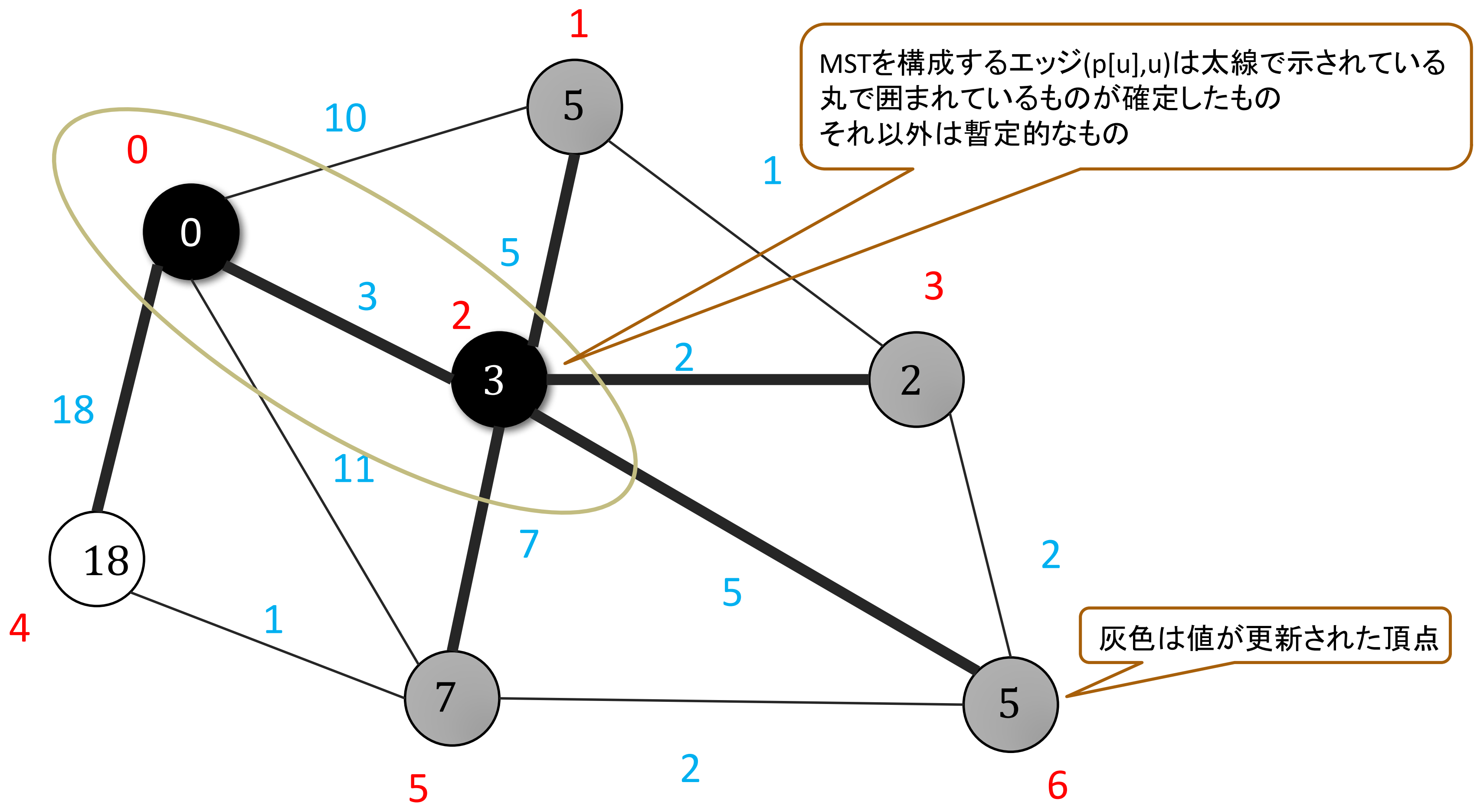
プリムのアルゴリズムの適用例



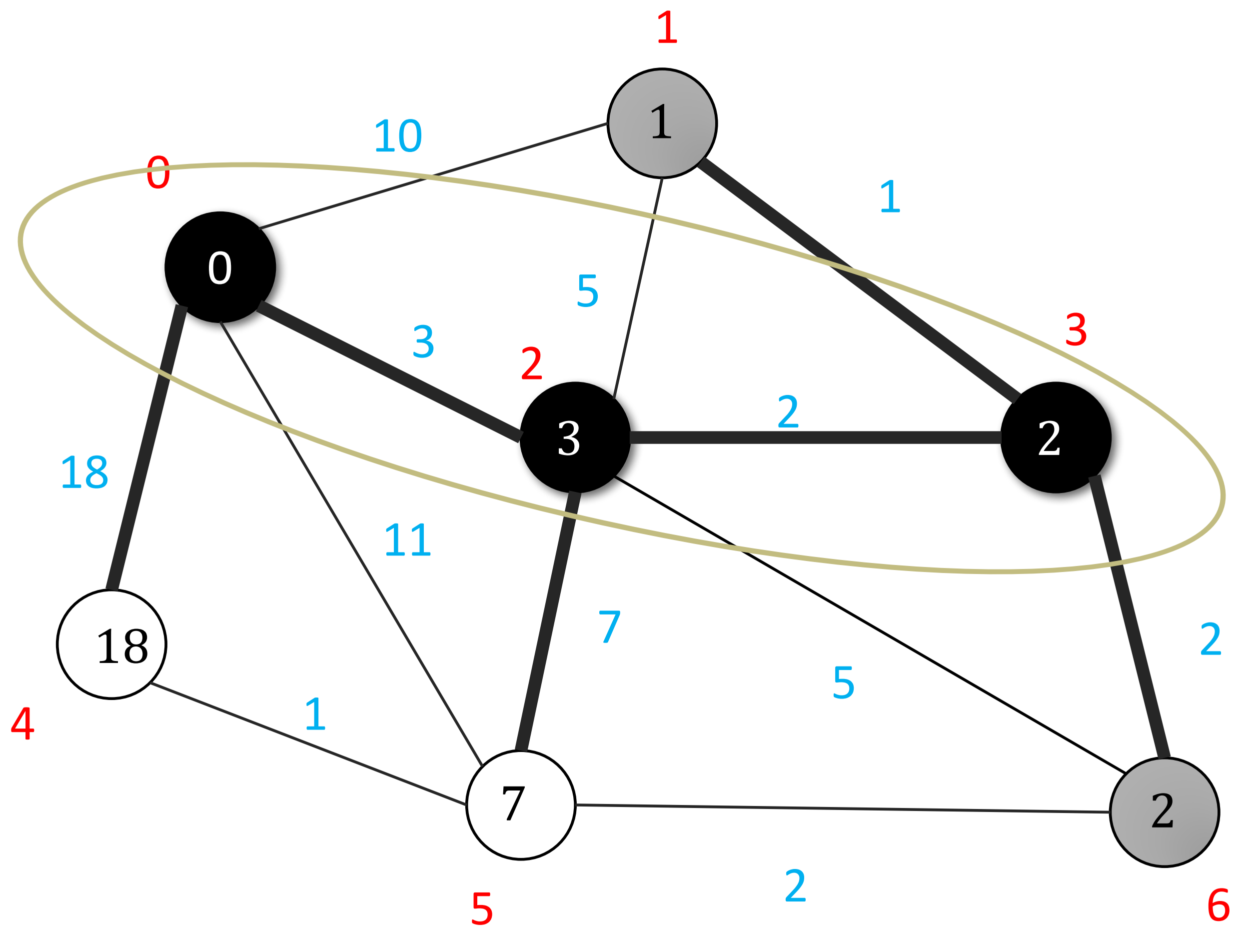
プリムのアルゴリズムの適用例



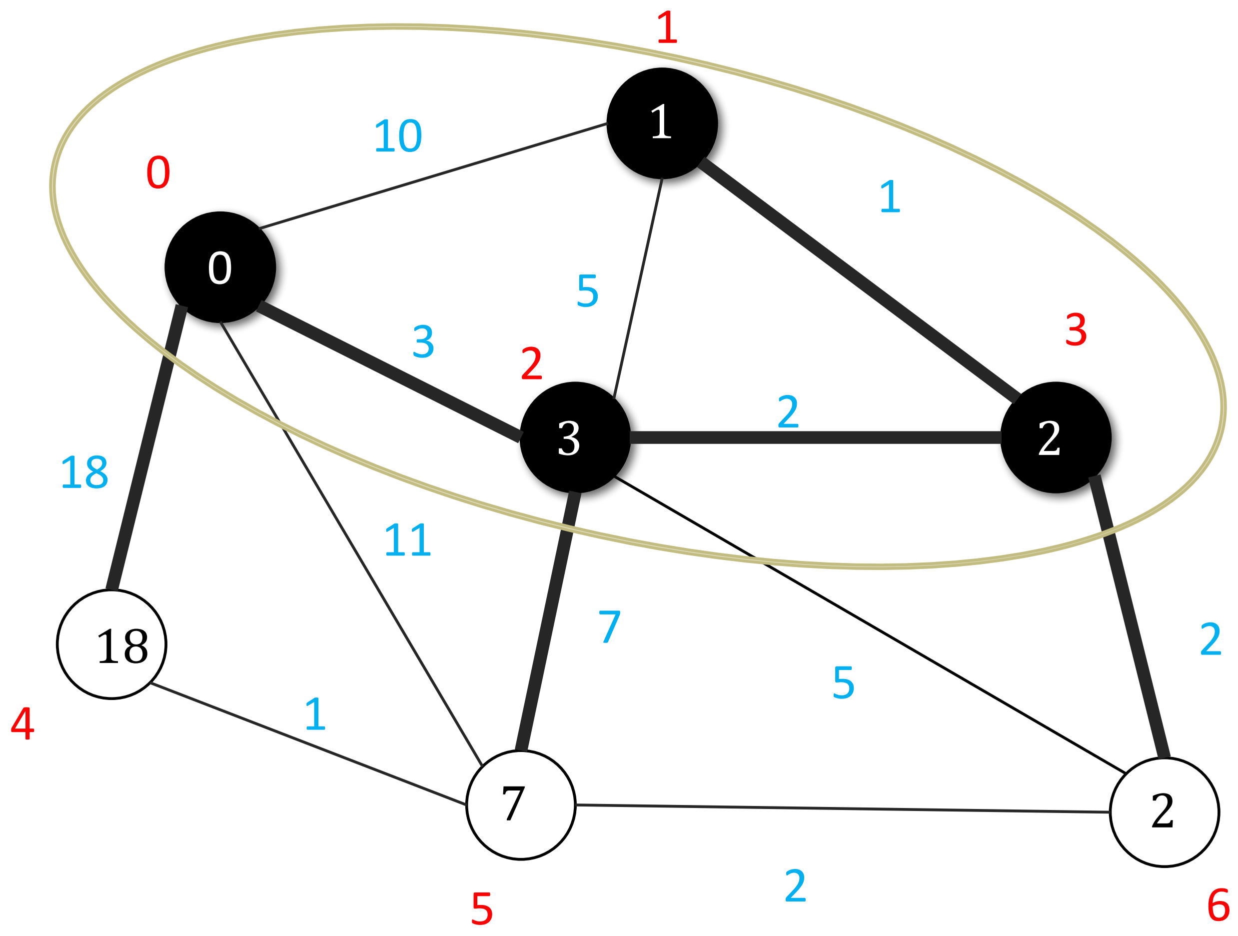
プリムのアルゴリズムの適用例



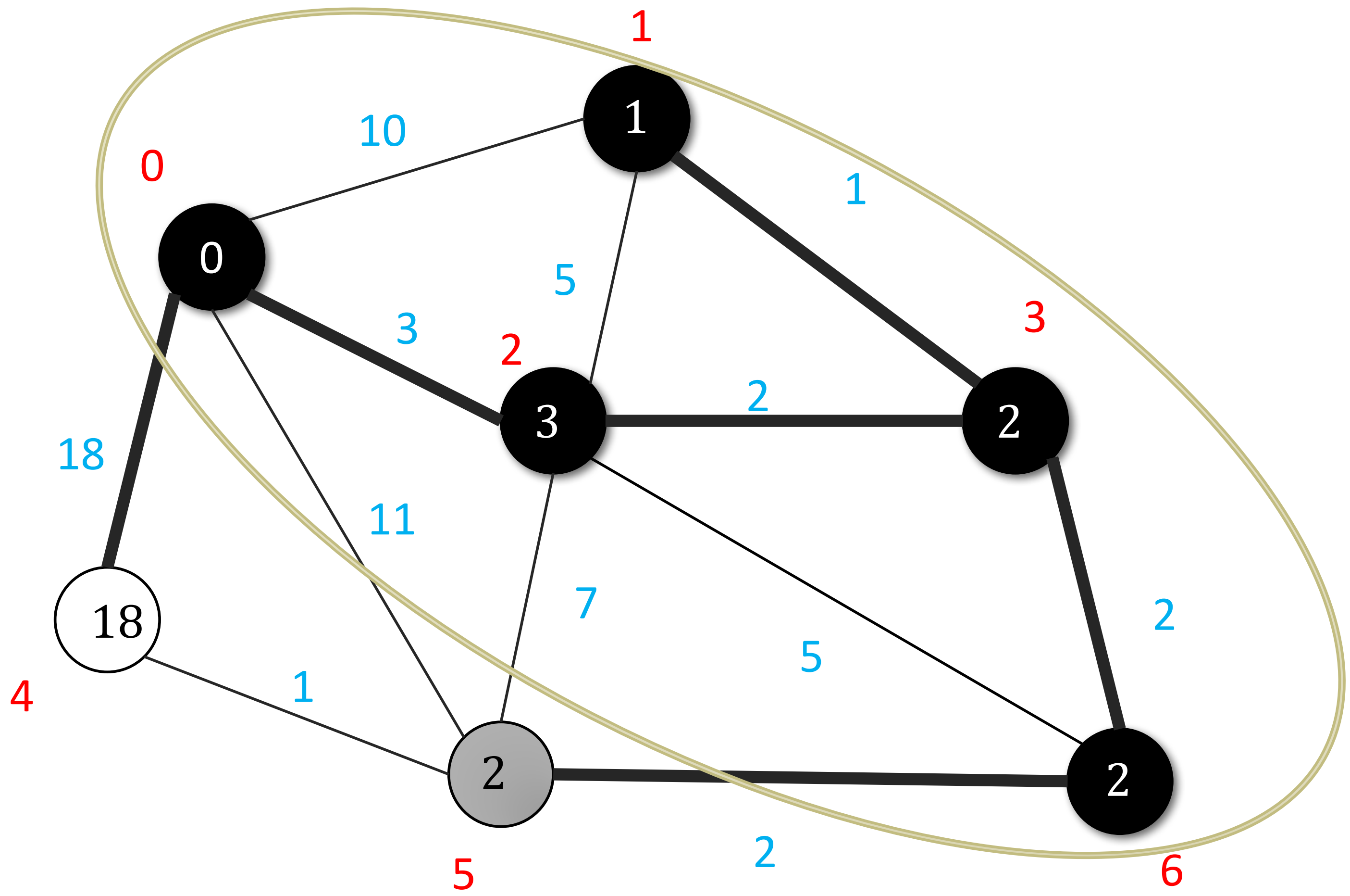
プリムのアルゴリズムの適用例



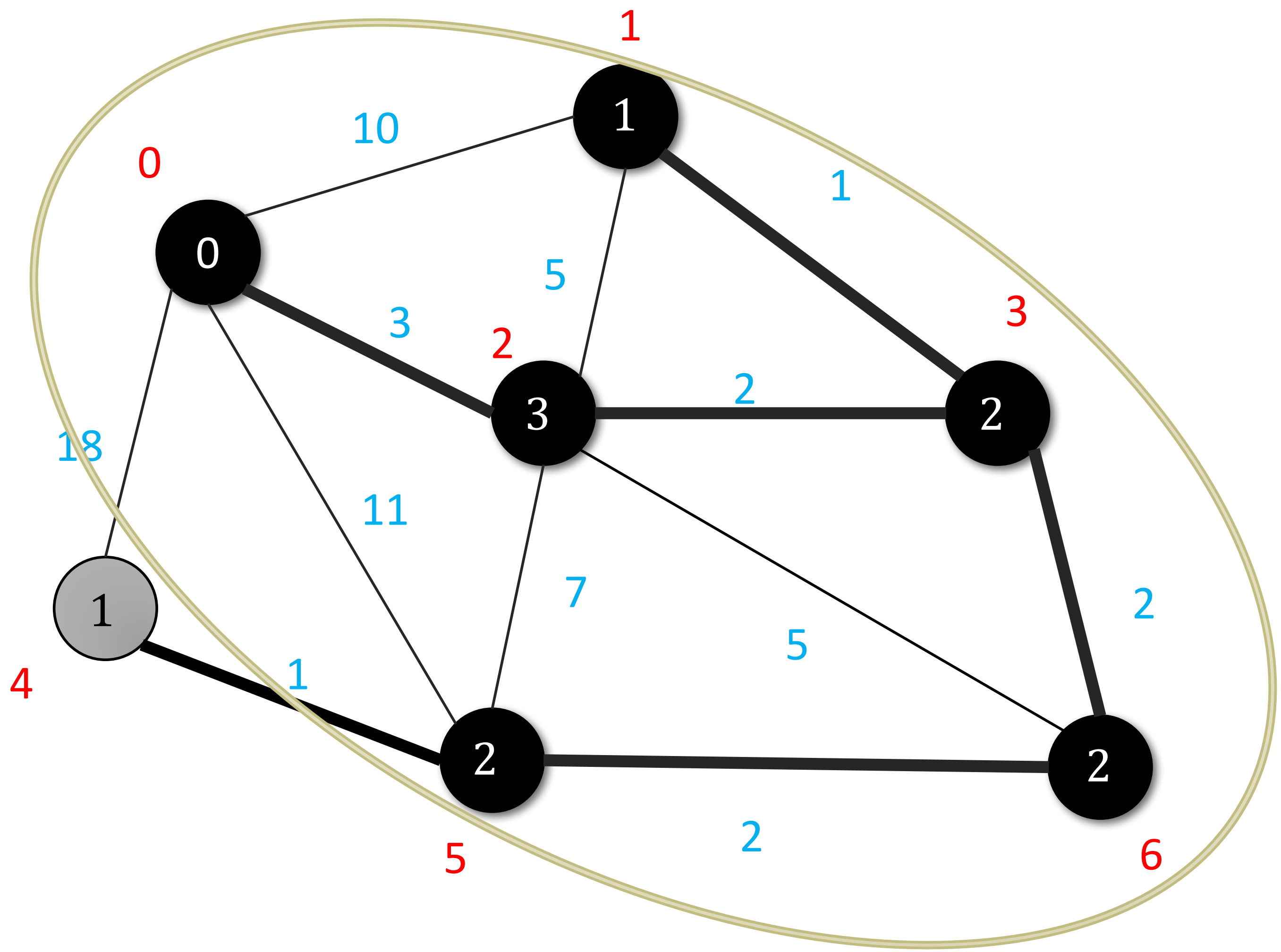
プリムのアルゴリズムの適用例



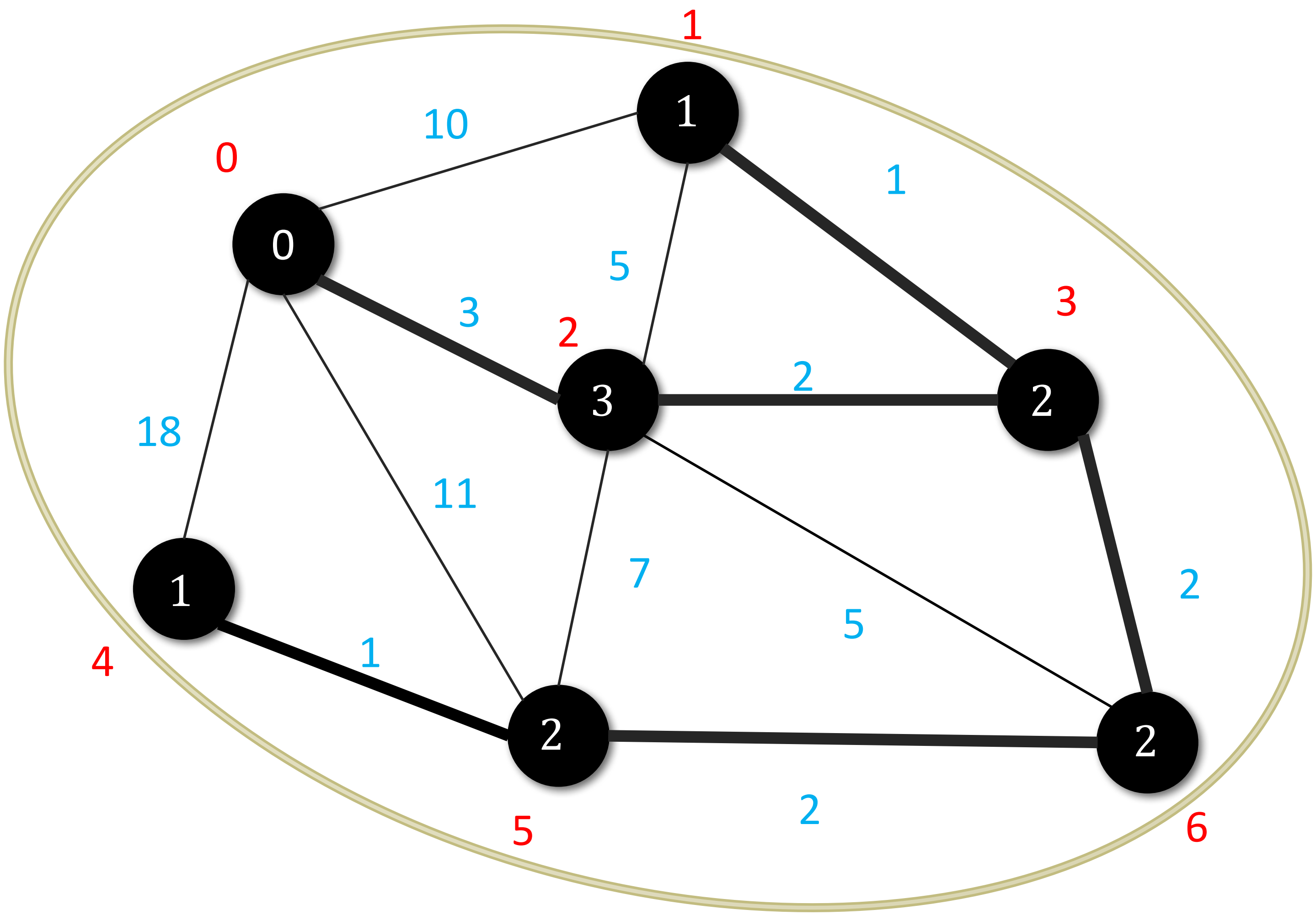
プリムのアルゴリズムの適用例



プリムのアルゴリズムの適用例



プリムのアルゴリズムの適用例



プリムのアルゴリズム

プリムのアルゴリズムは、

辺の重みが最小である頂点uを探すために、グラフの頂点の数だけ調べる必要がある

この探索を頂点の数だけ行うので、$O(|V|^2)$のアルゴリズムとなる

二分ヒープ（優先度付きキュー）を使って頂点を決定するようにすれば、高速化が可能

問題1

重み付きグラフに対する最小全域木の重みを計算

入力:

最初の行にGの頂点数nが与えられる

続くn行でGを表すn×nの隣接行列Aが与えられる

Aの要素a_{ij}は、頂点iと頂点jを結ぶ辺の重みを表す

ただし、辺がなければ-1で示される

出力:

Gの最小全域木の辺の重みの総和を1行に出力する

制約:

$1 \leq n \leq 100$

$0 \leq a_{ij} \leq 2000$（$a_{ij} \neq -1$のとき）

$a_{ij} = a_{ji}$

グラフGは連結である

入力例

```
5
-1 2 3 1 -1
2 -1 -1 4 -1
3 -1 -1 1 1
1 4 1 -1 3
-1 -1 1 3 -1
```

出力例

```
5
```

C++の場合

```
#include <iostream>
using namespace std;
static const int MAX = 100;
static const int INFTY = (1<<21)
static const int WHITE = 0;
static const int GRAY = 1;
static const int BLACK = 2;
int n, M[MAX][MAX];

int prim() {
   // ここを実装する
}

int main() {
   cin >> n
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         int e; ein >> e;
         M[i][j] = (e == -1) ? INFTY : e;
      }
   }
   cout << prim() << endl;
   return 0;
}
```

解説

以下のような変数を用意する（n=|V|とする）

color[n]： color[v]にvの訪問状態WHITE, GRAY, BLACKを記録する

M[n][n]： M[u][v]にuからvへの辺の重みを記録した隣接行列

d[n]： d[v]にTに属する頂点とV-Tに属する頂点をつなぐ辺の中で、
重みが最小の辺の重みを記録する

p[n]： p[v]にMSTにおける頂点vの親を記録する

解説

```
prim()
   全ての頂点 u について color[u] を WHITE とし、d[u] を INFTY へ初期化
   d[0] = 0
   p[0] = -1

   while true
      mincost = INFTY
      for i が 0 から n-1 まで
         if color[i] != BLACK && d[i] < mincost
            mincost = d[i]
            u = i

      if mincost == INFTY
          break

      color[u] = BLACK

      for v が 0 から n-1 まで
          if color[v] != BLACK かつ u と v の間に辺がある
          if M[u][v] < d[v]
              d[v] = M[u][v]
              p[v] = u
              color[v] = GRAY
```

実装例(C++)

```
int prim() {
  int u, minv;
  int d[MAX], p[MAX], color[MAX];

  for (int i = 0; i < n; i++) {
    d[i] = INFTY;
    p[i] = -1;
    color[i] = WHITE;
  }
  d[0] = 0;
  while (1) {
    minv = INFTY;
    u = -1;
    for (int i = 0; i < n; i++) {
      if (minv > d[i] && color[i] != BLACK) {
        u = i;
        minv = d[i];
      }
    }
    if (u == -1 ) break;
    color[u] = BLACK;
    for (int v = 0; v < n; v++) {
      if (color[v] != BLACK && M[u][v] != INFTY) {
        if (d[v] > M[u][v]) {
          d[v] = M[u][v];
          p[v] = u;
          color[v] = GRAY;
        }
      }
    }
  }
  int sum = 0;
  for (int i = 0; i < n; i++) {
    if (p[i] != -1)  sum += M[i][p[i]];
  }
  return sum;
}
```

目次

1. 最小全域木(重み付き無向グラフ)

2. 単一始点最短経路(重み付き有向グラフ)

最短経路問題

重み付きグラフG=(V,E)において、ある与えられた頂点の組 s, d を接続する経路の中で、辺の重みの総和が最小であるパスを求める問題

主に以下の2つに分類される

- 単一始点最短経路 (SSSP)

  グラフGにおいて、与えられた頂点sから、他の全ての頂点d_iへの最短経路を求める問題

- 全点対間最短経路 (APSP)

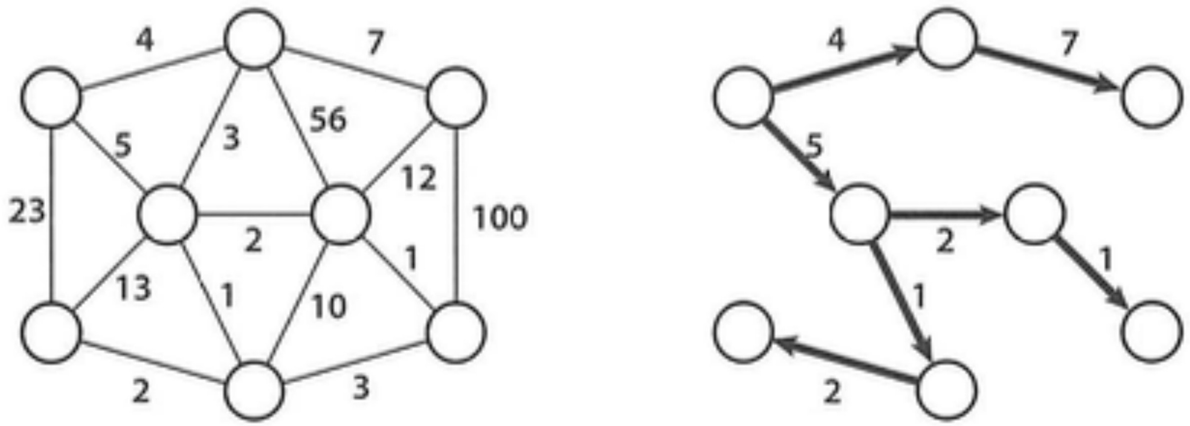
  グラフGにおいて全ての ‘頂点のペア’ 間の最短経路を求める問題

最短経路木

辺のコストが非負である重み付きグラフG(V,E)について、

頂点sからGの全ての頂点に対してパスがあるとき、sを根とし、

sからGの全ての頂点への最短経路を包含するGの全域木Tが存在する

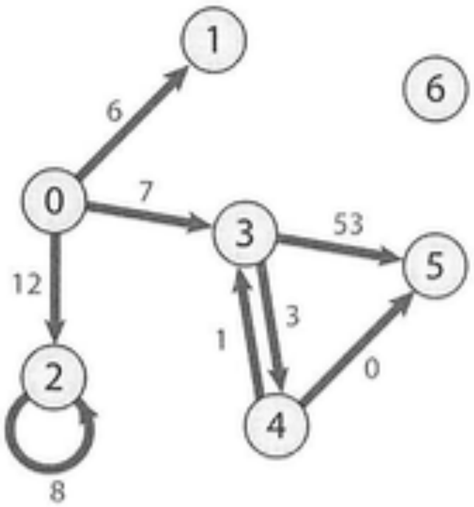


このような木を最短経路木という

重み付き有向グラフの隣接行列

重み付き有向グラフの隣接行列では、

頂点iから頂点jへ向かって重さwの辺がある場合、M[i][j]の値をwとする



j →, *i* ↓

	0	1	2	3	4	5	6
0	∞	6	12	7	∞	∞	∞
1	∞	∞	∞	∞	∞	∞	∞
2	∞	∞	8	∞	∞	∞	∞
3	∞	∞	∞	∞	3	53	∞
4	∞	∞	∞	1	∞	0	∞
5	∞	∞	∞	∞	∞	∞	∞
6	∞	∞	∞	∞	∞	∞	∞

辺がない場合は、問題に応じて∞ (大きい値) などに設定する

ダイクストラのアルゴリズム

グラフG=(V,E)における単一始点最短経路を求めるためのアルゴリズム

- グラフG=(V, E)の頂点全体の集合をV, 始点をs,
  最短経路木に含まれる頂点の集合をSとする

- 各計算ステップで、最短経路木の辺と頂点を選びSへ追加していく

ダイクストラのアルゴリズム

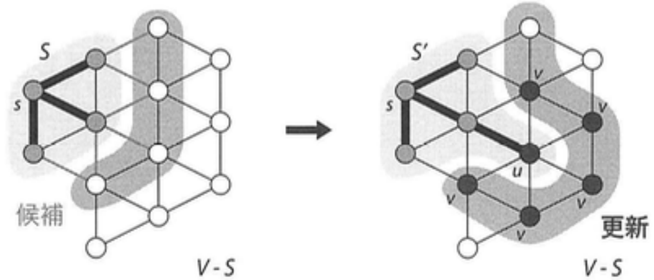
各頂点iについて、S内の頂点のみを経由したsからiへの最短経路のコストをd[i]、最短経路木におけるiの親をp[i]とする

1. 初期状態で、Sを空にする

   sに対してd[s]=0、s以外のVに属する全ての頂点iに対してd[i]=∞、と初期化する

2. 以下の処理をS=Vとなるまで繰り返す

   V-Sの中から、d[u]が最小である頂点uを選択する（右図）

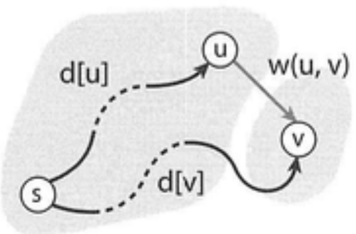


   uをsに追加すると同時に、uに隣接しかつV-Sに属する全ての頂点vに対する値を以下のように更新する（下図）

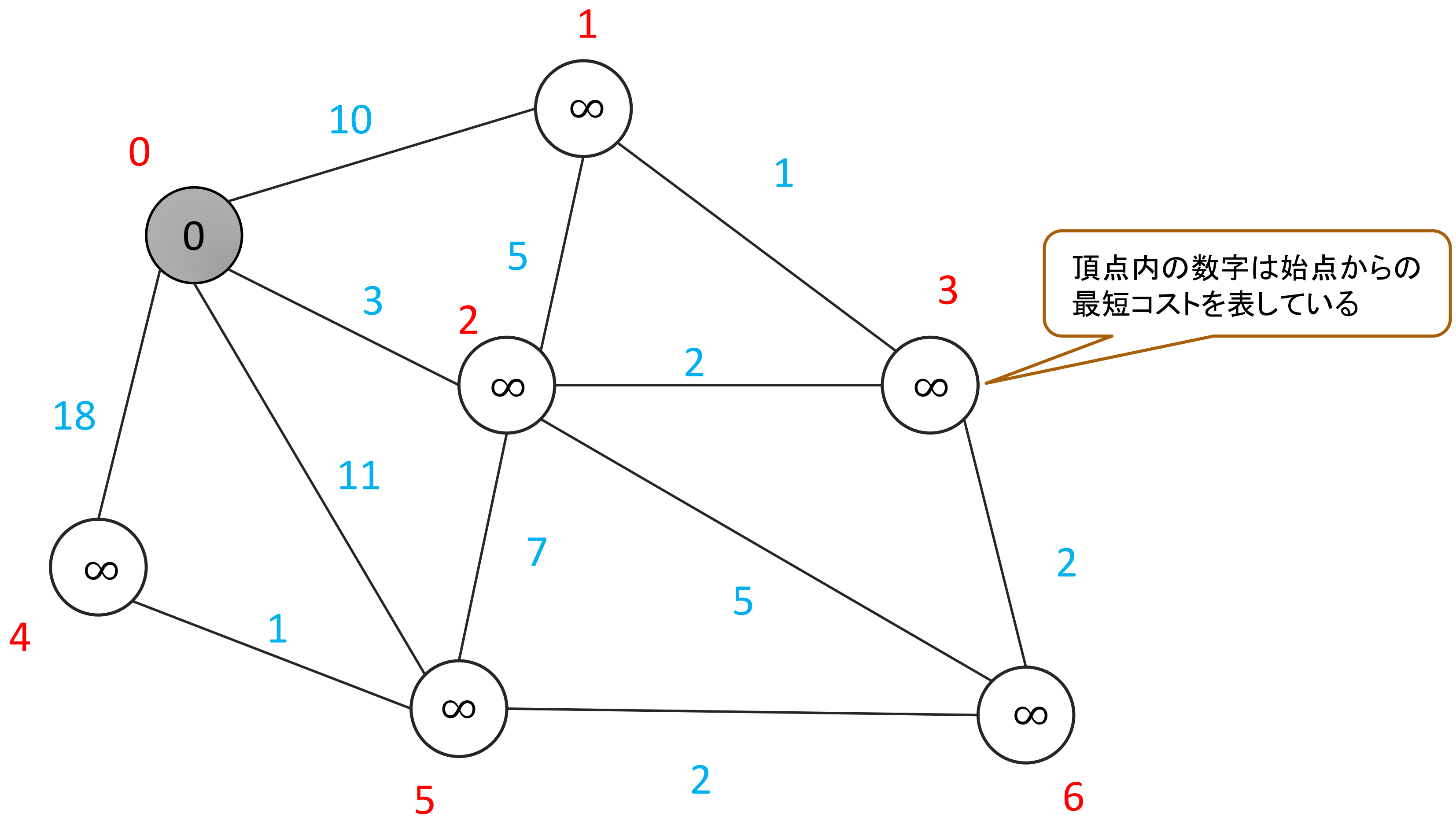
   if d[u] + w(u,v) < d[v]

      d[v] = d[u] + w(u,v)

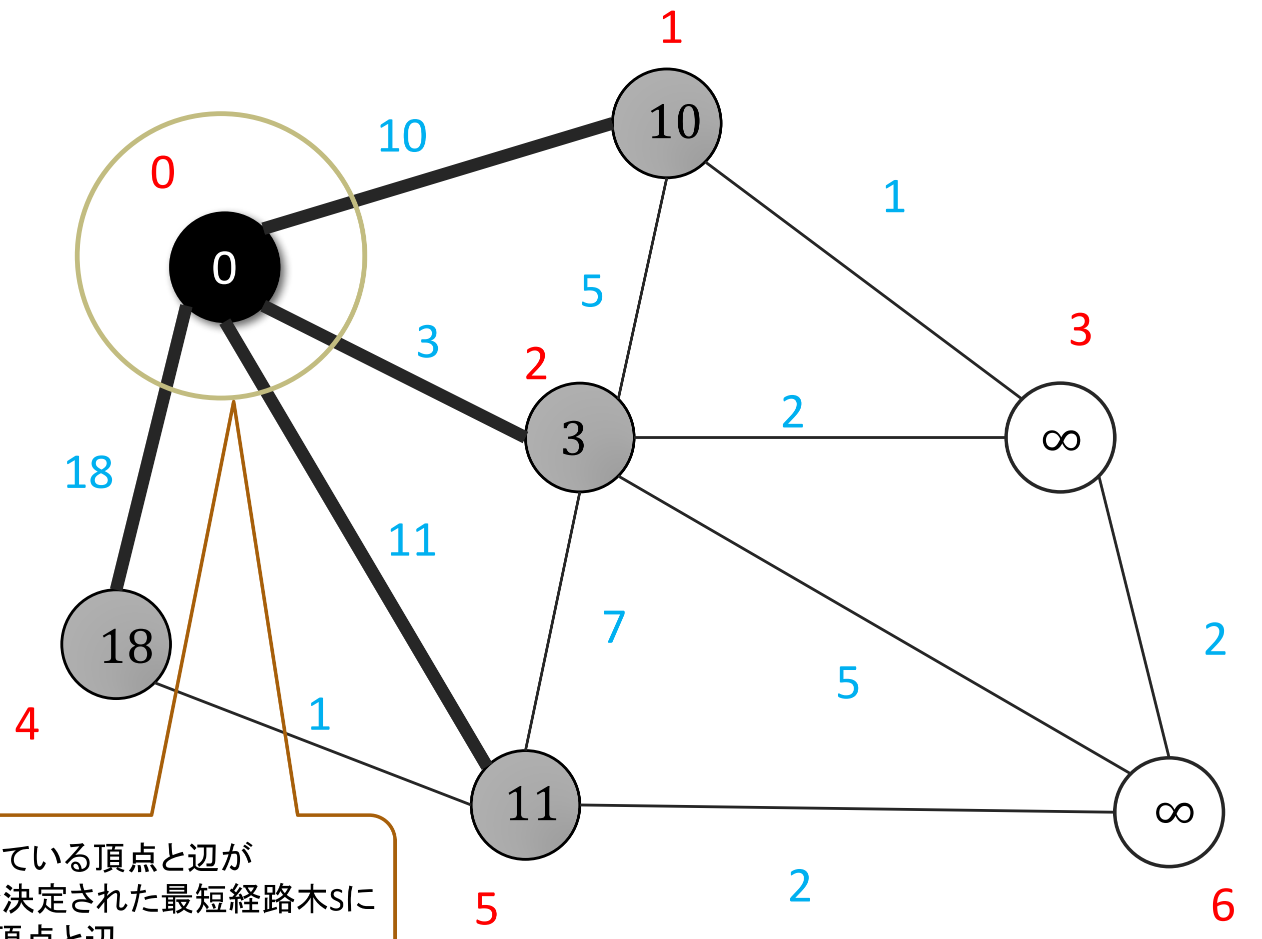
      p[v] = u



ダイクストラのアルゴリズムの適用例

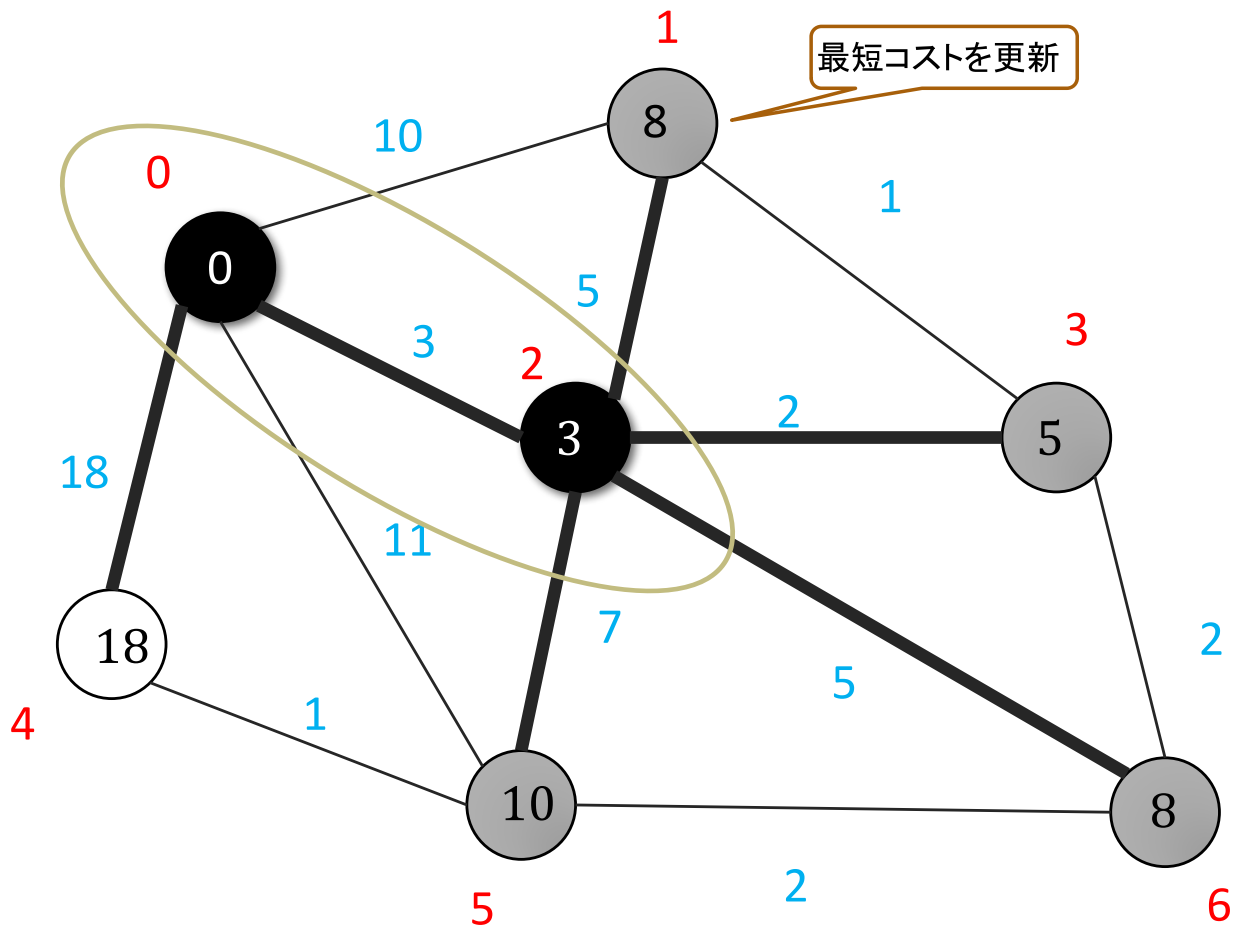


ダイクストラのアルゴリズムの適用例

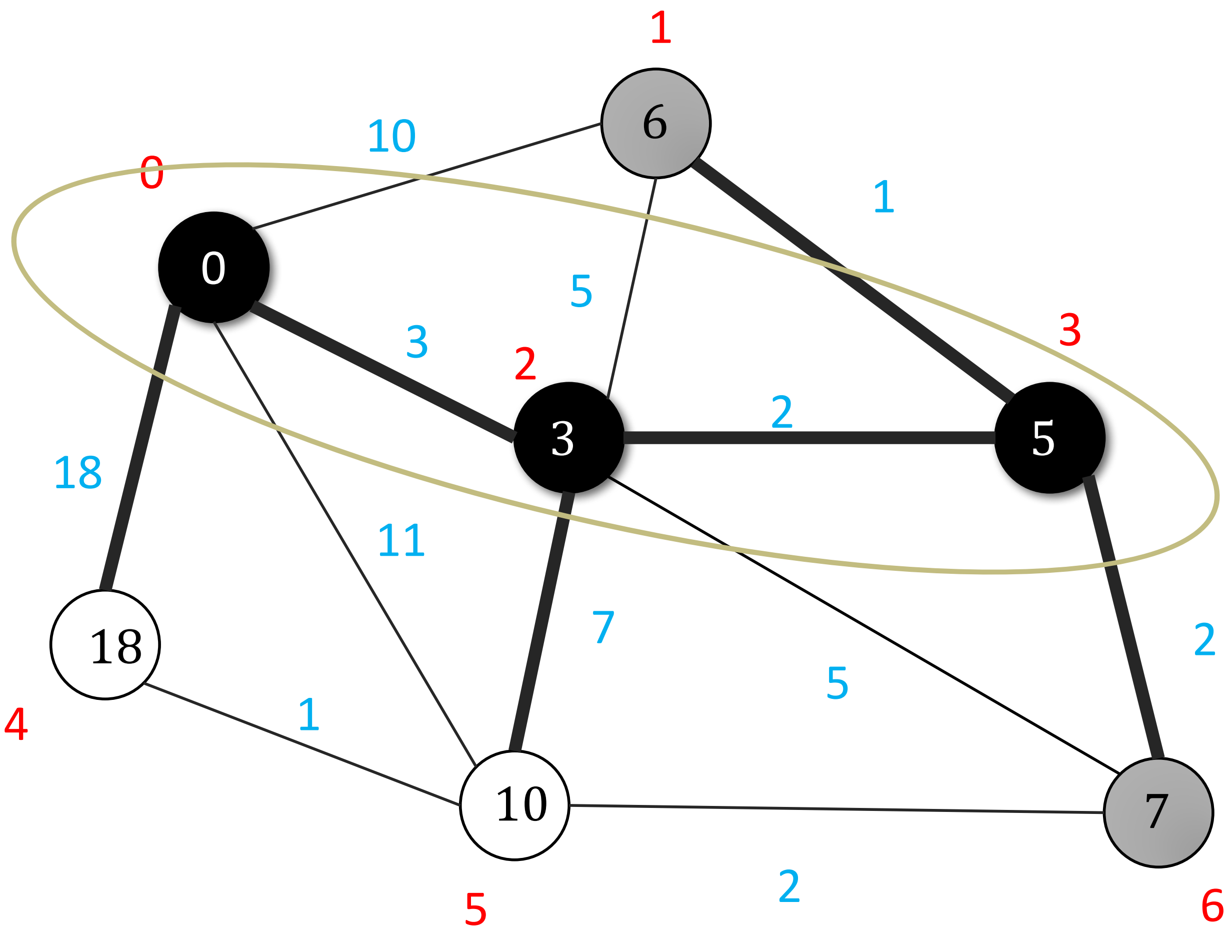


丸で囲まれている頂点と辺が
その時点で決定された最短経路木Sに
属している頂点と辺

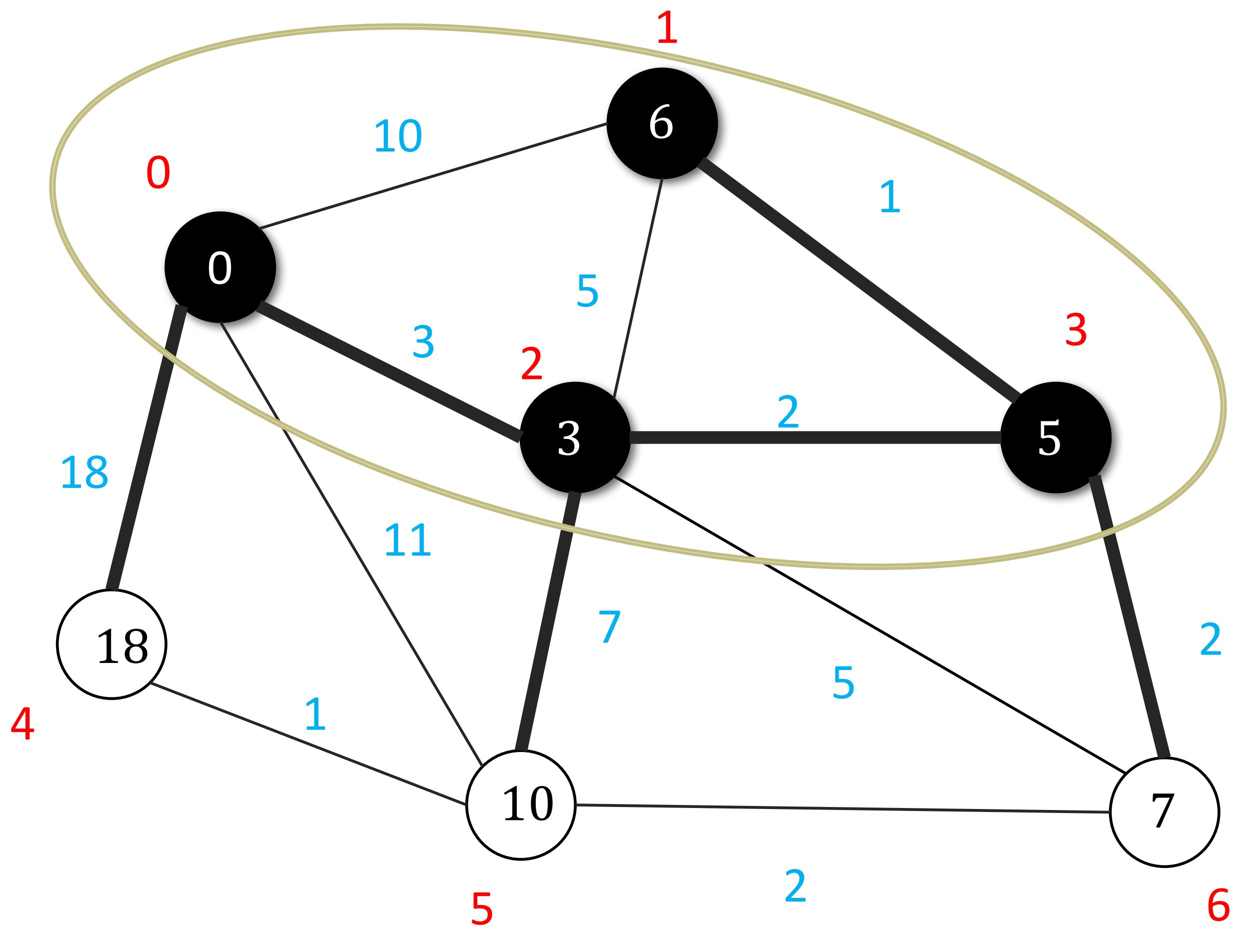
ダイクストラのアルゴリズムの適用例



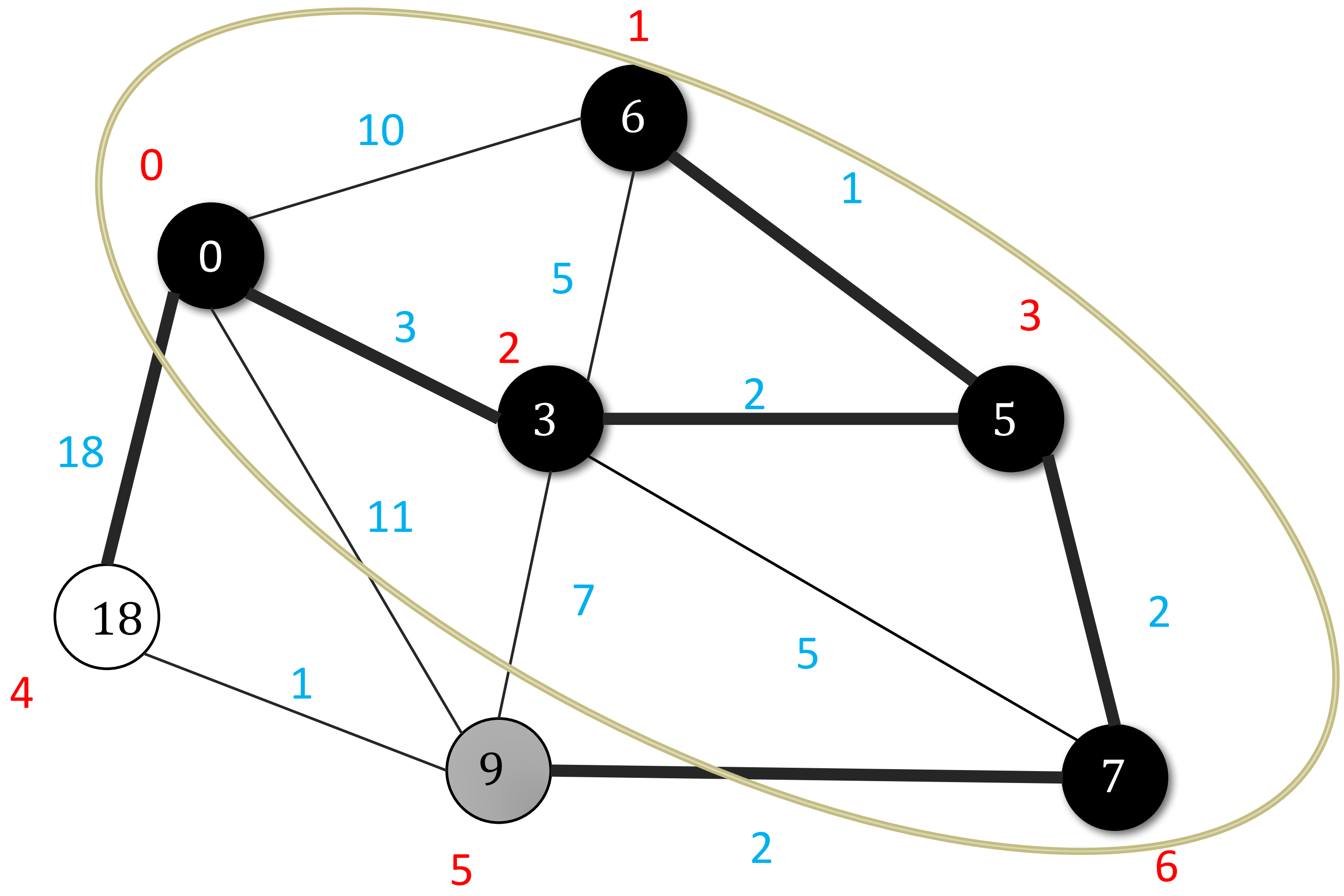
ダイクストラのアルゴリズムの適用例



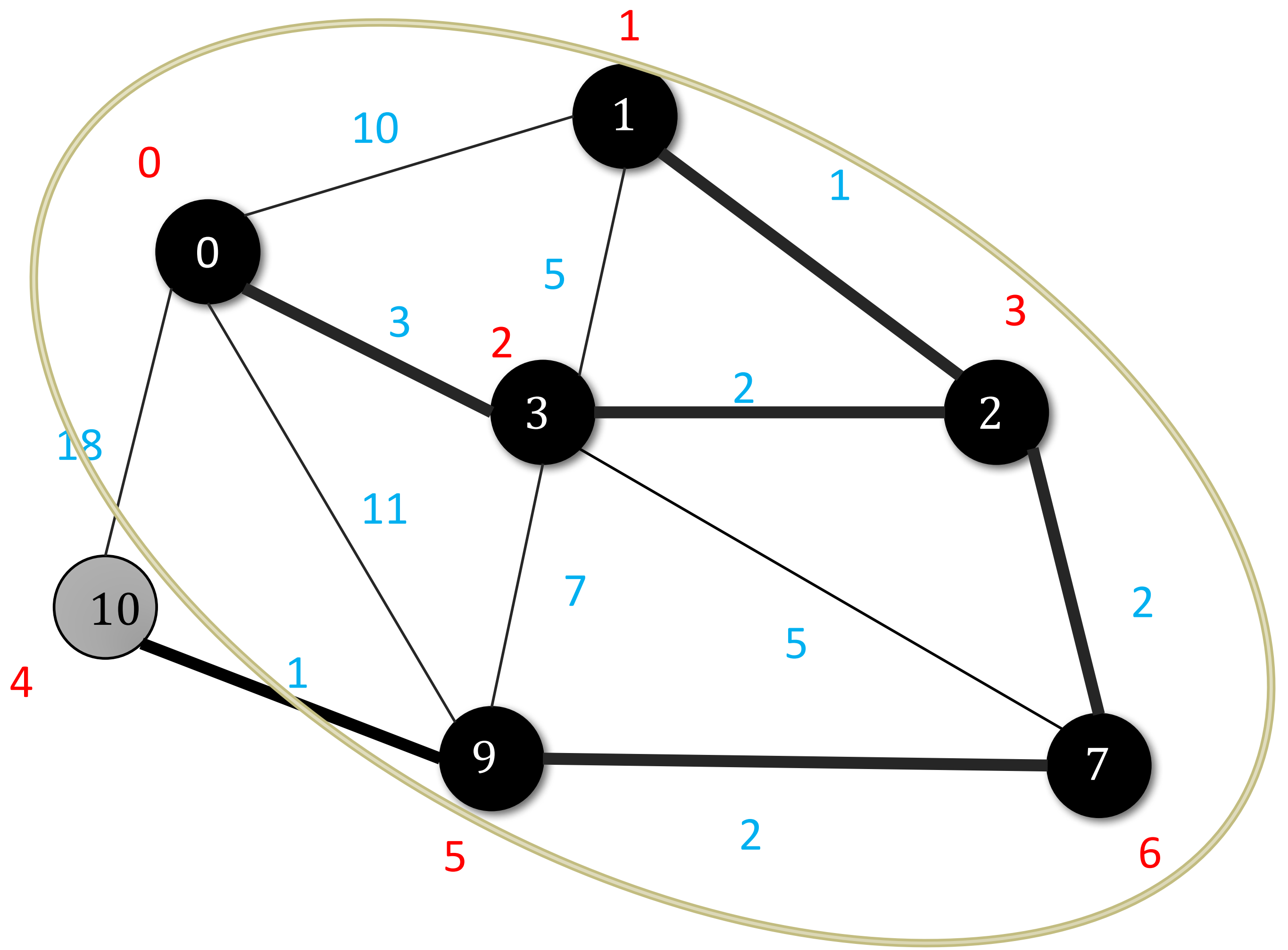
ダイクストラのアルゴリズムの適用例



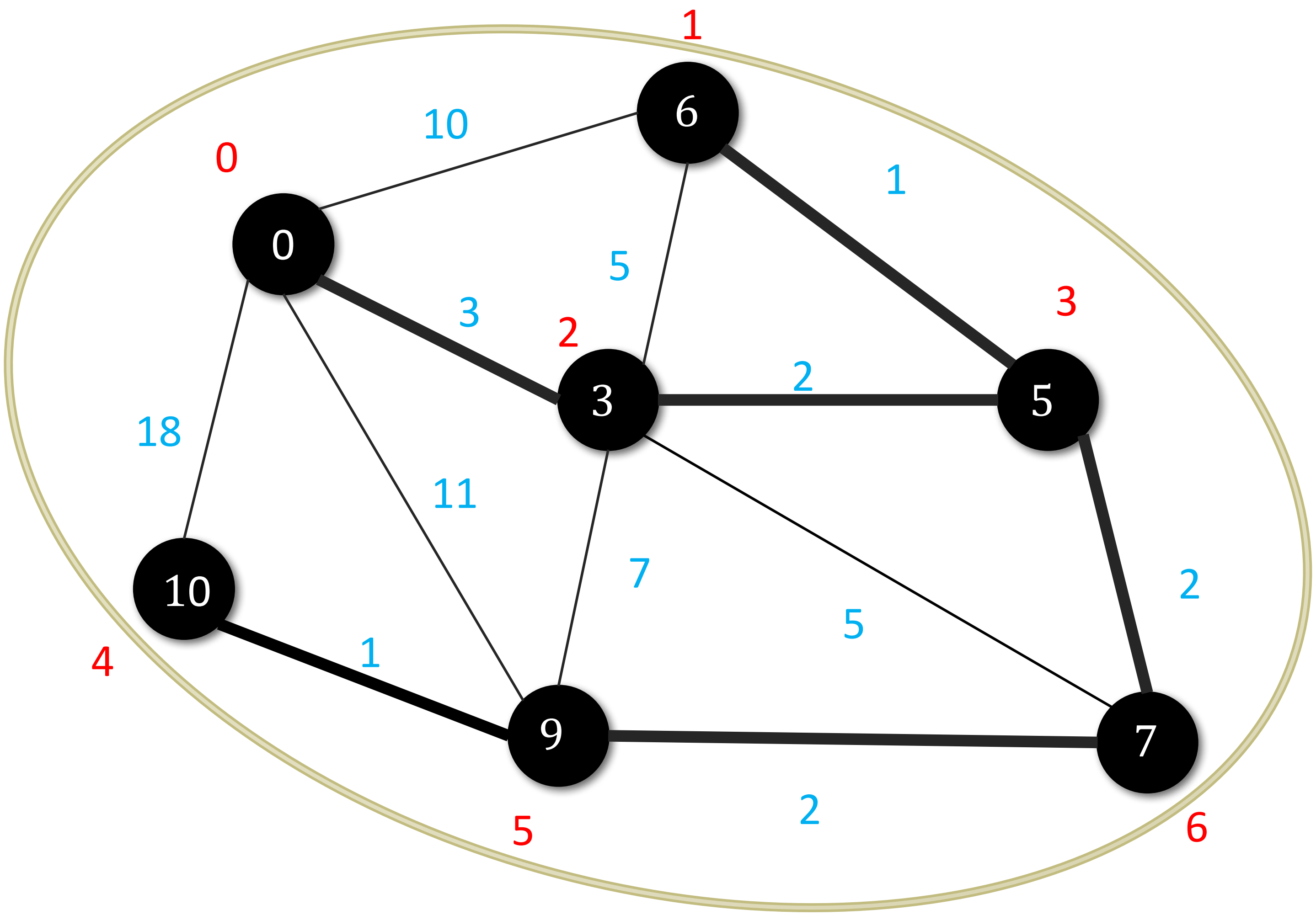
ダイクストラのアルゴリズムの適用例



ダイクストラのアルゴリズムの適用例



ダイクストラのアルゴリズムの適用例



ダイクストラのアルゴリズム

隣接行列を用いたダイクストラのアルゴリズムは、

　頂点uに隣接する頂点を$O(|V|)$で調べられる

　これらの処理を|V|回行うため、$O(|V|^2)$のアルゴリズムとなる

ダイクストラのアルゴリズムは負の重みの辺を含むグラフには適用できない

→ ベルマンフォードのアルゴリズム

　ワーシャルフロイドのアルゴリズムなどが適用可能

問題2

重み付き有向グラフについて、単一始点最短経路のコストを求める

入力:

最初の行に頂点数nが与えられる

続くn行で各頂点uの隣接リストが以下の形式で与えられる

$u k v_1 c_1 v_2 c_2 \cdots v_k c_k$

Gの各頂点には0からn-1の番号がふられている

uは頂点の番号であり、kはuの出次数を示す

$v_i (i = 1,2,\cdots k)$はuに隣接する頂点の番号であり、C_iはuとv_iをつなぐ有効辺の重みを示す

入力例

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

出力:

各頂点の番号vと距離d[v]を1つの空白区切りで順番に出力する

出力例

```
0 0
1 2
2 2
3 1
4 3
```

制約:

$1 \leq n \leq 100$

$0 \leq c_i \leq 100000$

0から各頂点へは必ず経路が存在する

C++の場合

```
#include <iostream>
using namespace std;
static const int MAX = 100;
static const int INFTY =
(1<<21);
static const int WHITE = 0;
static const int GRAY = 1;
static const int BLACK;
int n, M[MAX][MAX]

void dijkstra(){
   // ここを実装する
}
```

```
int main() {
   cin >> n;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         M[i][j] = INFTY;
      }
   }

   int k, c, u, v;
   for (int i = 0;  i < n; i++) {
       cin >> u >> k;
       for (int j = 0; j < k; j++) {
          cin >> v >> c;
          M[u][v] = c;
      }
   }

   dijkstra();

   return 0;
}
```

解説

以下のような変数を用意する (n=|V|とする)

color[n] : color[v]にvの訪問状態WHITE, GRAY, BLACKのいずれかを記録

M[n][n] : M[u][v]にuからvへの辺の重みを記録した隣接行列

d[n] : d[v]に始点sからvまでの最短コストを記録する

p[n] : p[v]に最短経路木における頂点vの親を記録する

解説

disjkstra(s)

全ての頂点uについて color[u]をWHITEとし、d[u]をINFTYへ初期化

d[s] = 0, p[s] = -1

```
while true
    mincost = INFTY
    for i が 0 から n-1 まで
        if color[i] != BLACK && d[i] < mincost
            mincost = d[i]
            u = i
    if mincost == INFTY
        break
    color[u] = BLACK
    for v が 0 から n-1 まで
        if color[v] != BLACK かつ u と v の間に辺がある
            if d[u] + M[u][v] < d[v]
                d[v] = d[u] + M[u][v]
                p[v] = u
                color[v] = GRAY
```

実装例(C++)

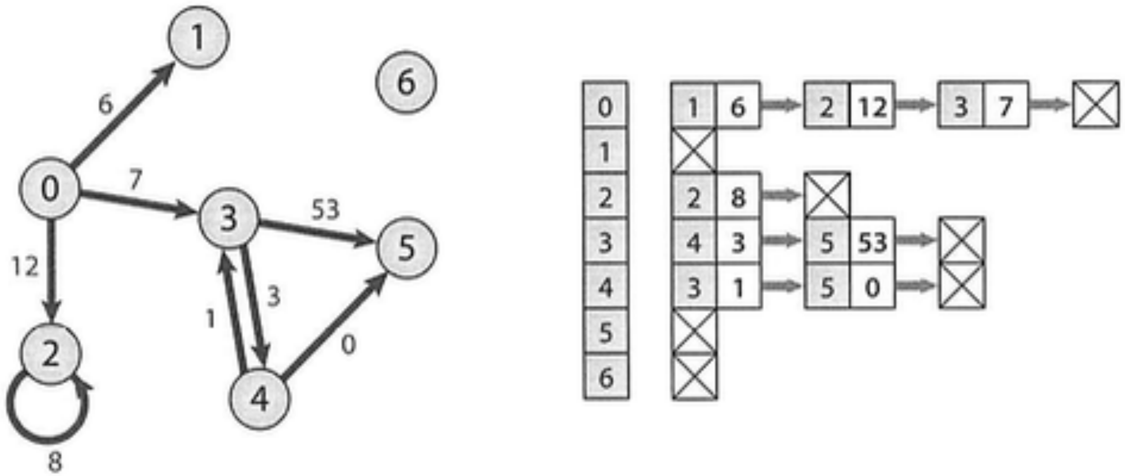
```
void dijkstra() {
   int minv;
   int d[MAX], color[MAX];
   for (int i = 0; i < n; i++) {
      d[i] = INFTY;
      color[i] = WHITE;
   }
   d[0] = 0;
   color[0] = GRAY;
   while (1) {
      minv = INFTY;
      int u = -1;
      for (int i = 0; i < n; i++){
         if (minv > d[i] && color[i] != BLACK) {
            u = i;
            minv = d[i];
         }
      }
      if (u == -1)  break;
      color[u] = BLACK;
      for (int v = 0; v < n; v++) {
         if (color[v] != BLACK && M[u][v] != INFTY) {
            if (d[v] > d[u] + M[u][v]) {
               d[v] = d[u] + M[u][v];
               color[v] = GRAY;
            }
         }
      }
   }
   for (int i = 0; i < n; i++) {
      cout << i << " " << (d[i] == INFTY ? -1 : d[i]) << endl;
   }
}
```

ダイクストラのアルゴリズム(高速版)

隣接行列を使用したダイクストラのアルゴリズムは$O(|V|^2)$の計算量

→ 隣接リストによる表現と、二分ヒープ（優先度付きキュー）を

応用することにより高速化が可能



重み付きグラフの隣接リストでは、番号だけでなく重みもリストの要素に追加する

問題3

重み付き有向グラフについて、単一始点最短経路のコストを求める
（問題2よりも早いアルゴリズムを実装する）

入力：

最初の行に頂点数nが与えられる

続くn行で各頂点uの隣接リストが以下の形式で与えられる

$u k v_1 c_1 v_2 c_2 \cdots v_k c_k$

Gの各頂点には0からn-1の番号がふられている

uは頂点の番号であり、kはuの出次数を示す

$v_i (i = 1,2, \cdots k)$はuに隣接する頂点の番号であり、C_iはuとv_iをつなぐ有効辺の重みを示す

入力例

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

出力：

各頂点の番号vと距離d[v]を1つの空白区切りで順番に出力する

出力例

```
0 0
1 2
2 2
3 1
4 3
```

制約：

$1 \leq n \leq 100$

$0 \leq c_i \leq 100000$

0から各頂点へは必ず経路が存在する

ダイクストラのアルゴリズム(高速版)

各頂点iについて、S内の頂点のみを経由したsからiへの最短経路のコストをd[i]、最短経路木におけるiの親をp[i]とする

1. 初期状態で、Sを空とする

   sに対してd[s] = 0

   s以外のVに属する全ての頂点iに対してd[i] = ∞ と初期化する

   d[i]をキーとして、Vの頂点をmin-ヒープHとして構築する

2. 次の処理をS=Vとなるまで繰り返す

   Hからd[u]が最小である頂点uを取り出す

   uをSに追加すると同時に、uに隣接しかつV-Sに属する全ての頂点vに対する値を以下のように更新

   if d[u] + w(u,v) < d[v]

   d[v] = d[u] + w(u,v)

   p[v] = u

   vを起点にヒープHを更新する

ヒープによるダイクストラのアルゴリズム

```
dijkstra(s)
  全ての頂点uについてcolor[u]をWHITEとし、d[u]をINFTYへ初期化
  d[s] = 0

  Heap heap = Heap(n,d)
  heap.constract()

  while heap.size >= 1
    u = heap.extractMin()
    color[u] = BLACK

    // uに隣接する頂点vが存在する限り
    while v = next(u) != NIL
      if color[v] != BLACK
        if d[u] + M[u][v] < d[v]
          d[v] = d[u] + M[u][v]
          color[v] = GRAY
          heap.update(v)
```

二分ヒープと連携した実装はやや複雑

二分ヒープの代わりに優先度付きキューに
候補となる頂点を挿入していく方が比較的簡単

優先度付きキューによるダイクストラのアルゴリズム

```
dijkstra(s)
  全ての頂点uについてcolor[u]をWHITEとし、d[u]をINFTYへ初期化
  d[s] = 0

  PQ.push(Node(s,0))     //優先度付きキューに始点を挿入
  // 最初にsがuとして選ばれる

  while PQが空でない
    u = PQ.extractMin()

    color[u] = BLACK

    if d[u] < uのコスト     // 最小値を取り出し、それが最短でなければ無視
      continue

    while v = next(u) != NIL       // uに隣接する頂点vが存在する限り
      if color[v] != BLACK
        if d[u] + M[u][v] < d[v]
          d[v] = d[u] + M[u][v]
          color[v] =GRAY
          PQ.push(Node(v,d[v]))
```

ダイクストラのアルゴリズム（高速版）

隣接リストと二分ヒープを用いたダイクストラのアルゴリズムの計算量は、

頂点uを二分ヒープから取り出すために$O(|V| \log |V|)$, d[v]を更新するために$O(|E| \log |V|)$

→ $O((|V| + |E|) \log |V|)$

隣接リストと優先度付きキューを用いた場合は、

|V|の数だけキューから頂点が取り出され、|E|の数だけキューに挿入される

→ $O\left((|V| + |E|) \log |V|\right)$

C++の場合

```
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;
static const int MAX = 10000;
static const int INFTY = (1<<20);
static const int WHITE = 0;
static const int GRAY = 1;
static const int BLACK = 2;

int n;
vector<pair<int,int>> adj[MAX];    //重み付き有向グラフの隣接リスト表現

void dijkstra() {
    // ここを実装する
}

int main() {
    int k, u, v, c;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> u >> k;
        for (int j = 0; j < k; j++) {
            cin >> v >> c;
            adj[u].push_back(make_pair(v, c));
        }
    }
    dijkstra();
    return 0;
}
```

実装例(C++)

```
void dijkstra() {
  priority_queue<pair<int, int>> PQ;
  int color[MAX];
  int d[MAX];
  for (int i = 0; i < n; i++) {
    d[i] = INFTY;
    color[i] = WHITE;
  }
  d[0] = 0;
  PQ.push(make_pair(0,0));
  color[0] = GRAY;
  while (!PQ.empty()){
    pair<int, int> f = PQ.top(); PQ.pop();
    int u = f.second;
    color[u] = BLACK;

    // 最小値を取り出し、それが最短でなければ無視
    if (d[u] < f.first * (-1))  continue;
    for (int j = 0; j < adj[u].size(); j++) {
      int v = adj[u][j].first;
      if (color[v] == BLACK)  continue;
      if (d[v] > d[u] + adj[u][j].second) {
        d[v] = d[u] + adj[u][j].second;
        //priority_queueはデフォルトで大きい値を優先するため-1をかける
        PQ.push(make_pair(d[v] * (-1), v));
        color[v] = GRAY;
      }
    }
  }
  for (int i = 0; i < n; i++) {
    cout << I << " " << (d[i] == INFTY ? -1 : d[i]) << endl;
  }
}
```