最短経路問題

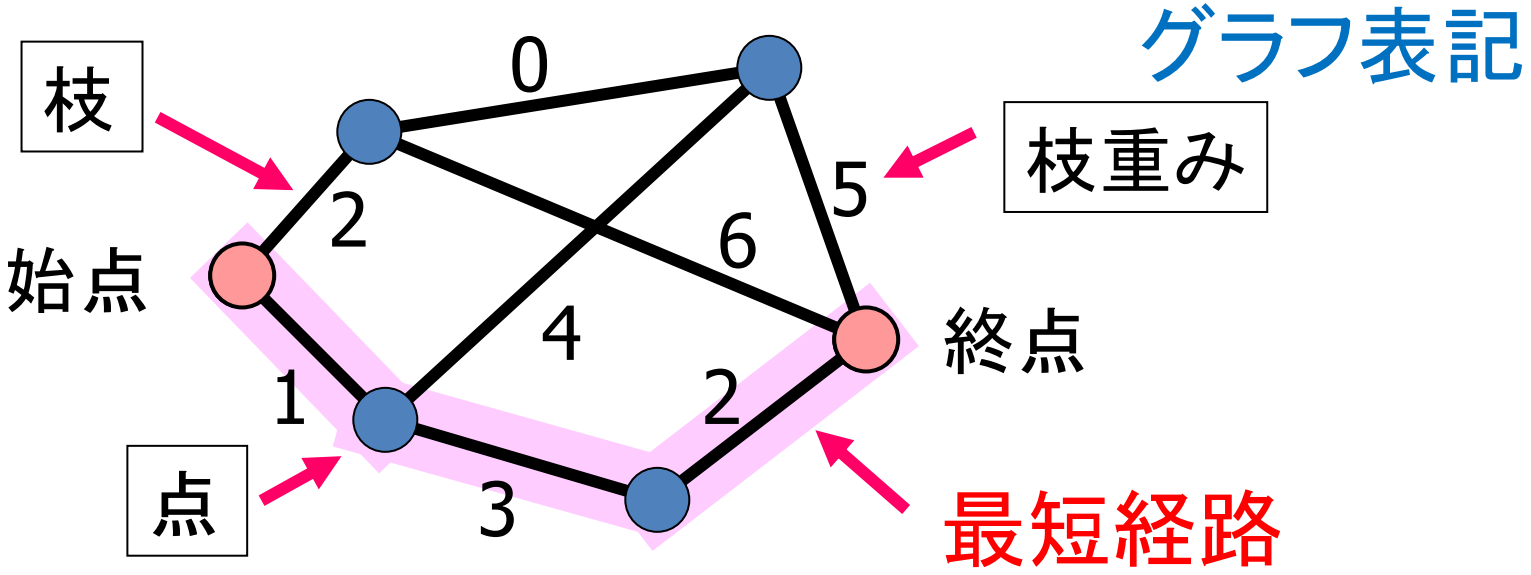
オープンラボ　C言語プログラミング　課題

電気電子物理工学科伊藤研究室

2024年9月24日版

最短経路問題

- 始点から終点へ行く最も短い経路
  ＝枝重み合計が最小の経路を見つける



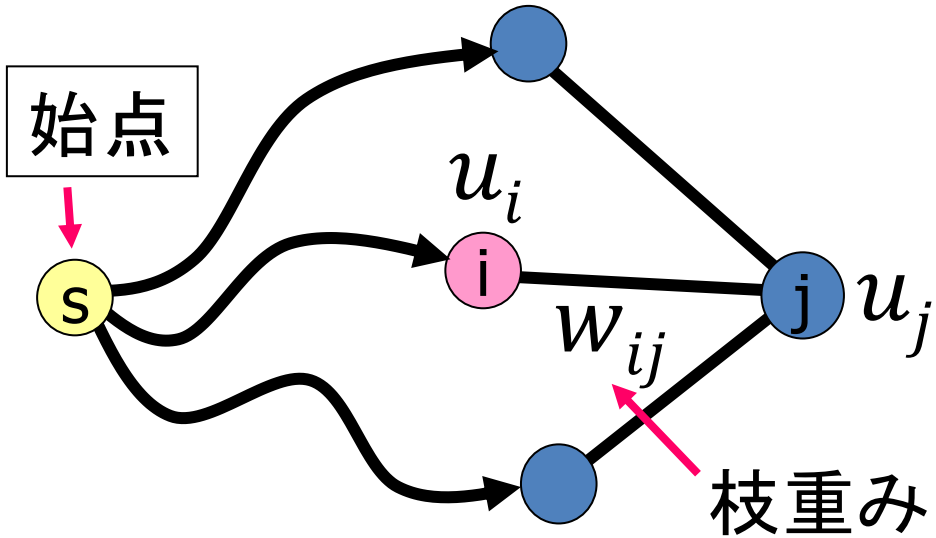
Bellman方程式

- 点jの最短経路長u_jについて成り立つ関係式

$$u_s = 0$$

$$u_j = \min_i \{u_i + w_{ij}\} \quad \forall j \neq s$$

Bellman方程式



- Bellman方程式(連立方程式)を満たすu_jを求めれば、それが最短経路長

Bellman方程式を解く(1)

- 1. 点jの最短経路長u_jに適当な初期値を設定
  - 通常は+∞(実際には十分大きな正の値で代用)
- 2. 全点jと、その全ての入力枝(i, j)について
  $u_i + w_{ij}$がu_jよりも小さければ、その値でu_jを更新
  ($u_j > u_i + w_{ij}$ならばu_jに$u_i + w_{ij}$を代入)
- 3. 全点jについてu_jの更新がなければ終了
  いずれかの点で更新があれば、2.に戻る

⇒ Bellman-Ford法

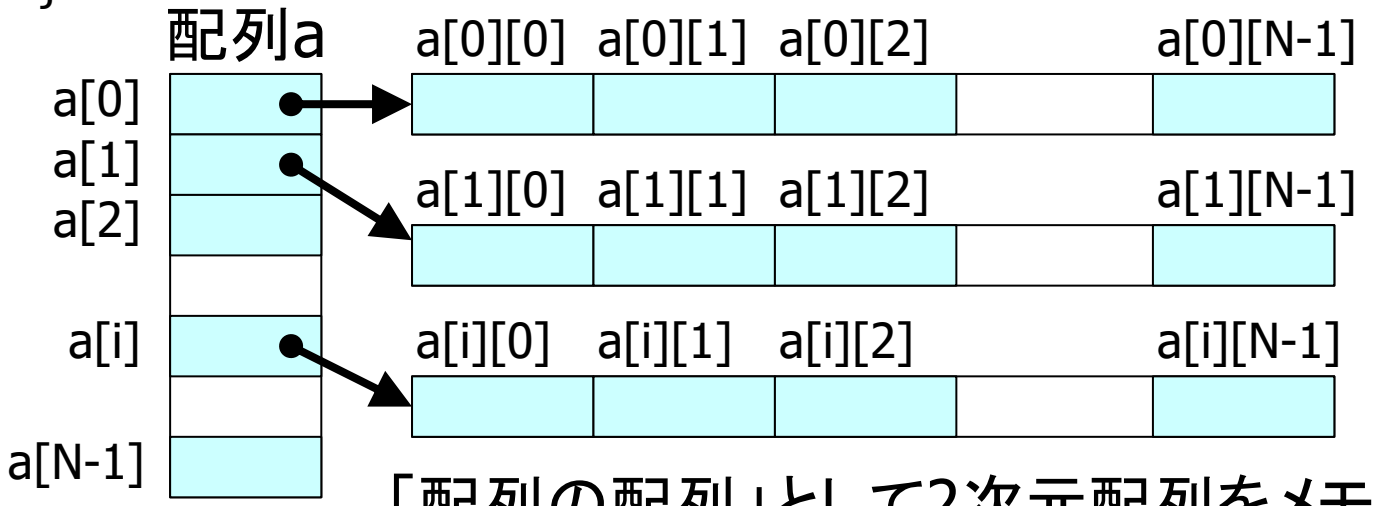
計算複雑度 $O(NE)$ または $O(N^3)$　　(点数N, 枝数E)

枝を記録する2次元配列

```
#define INTINF 999999
int N;
int **a;
int i,j;

a = (int **)malloc(N*sizeof(int *));
for(i=0 ; i<N ; i++){
      a[i] = (int *)malloc(N*sizeof(int));
      for(j=0 ; j<N ; j++) a[i][j] = INTINF;
}
```

十分大きい正数で正無限大を近似
Integer infinity
グラフの点の数
枝を表す2次元配列を指す
要素数Nの配列を割り当て
要素数Nの配列を割り当て
各要素を無限大で初期化

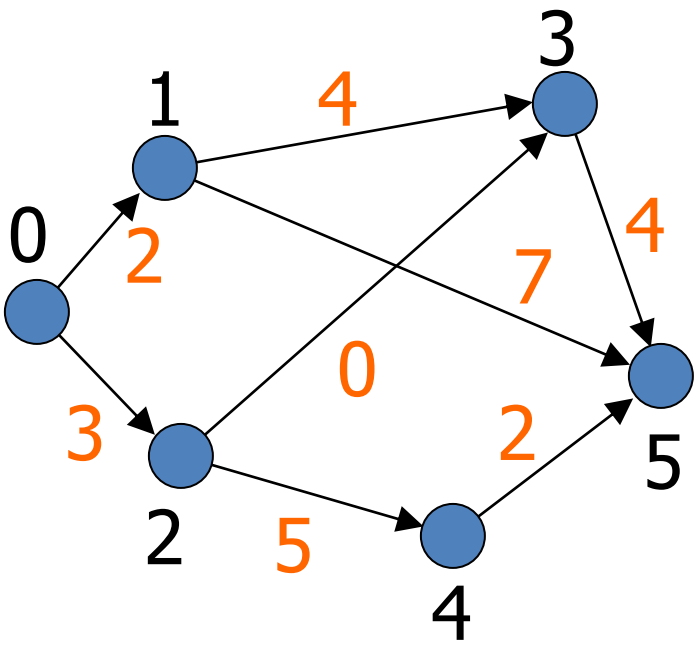


「配列の配列」として2次元配列をメモリ上に構築

グラフを記録するデータファイルの書式

- グラフは、テキストファイルから読み込む

グラフ → グラフを記述するテキストファイル




```
N 6
E 0 1 2
E 0 2 3
E 1 3 4
E 1 5 7
E 2 3 0
E 2 4 5
E 3 5 4
E 4 5 2
L 5 7
```

■テキストファイルの書式

- N *Num*
  グラフの点数が*Num*個
- E *From To Weight*
  始点*From*、終点*To*の
  枝の重みが*Weight*
- L *Node Length*
  点0から点*Node*への
  最短経路長は*Length*

ファイルからデータ読み取り

- テキストデータでファイルに記録されたグラフの情報を読み取る
- C言語プログラムからファイルを開き、内容を読み出し、解析、データ構造に記録、ファイルを閉じる

```
FILE *fp;
if( _tfopen_s( &fp, strFilename, _T("r") ) ) return;

char buf[256];
while(1){
    fgets( buf, 255, fp );
    if( feof(fp) ) break;
    // 読み込んだ行の解析とデータ記録をここで行う
}
fclose(fp);
```

ファイルを開く

ファイル名は変数strFilename
(CString型)に代入されているとする

読み出し(read)モード

ファイルが存在しないなどopenに
失敗したら処理を中止して戻る

無限ループ

1行読み込み

最後の行まで読み終わっていればループ終了

データを読み終わったらファイルを閉じる

読み込んだ行の解析

```
char DLM[] = " ";
char buf[256];
char *ptr, *ptr1, *ptr2;
char *pTokenContext;

while(1){
    fgets( buf, 255, fp );
    if( feof(fp) ) break;
    if( strlen(buf)>0 && buf[strlen(buf)-1] == '¥n' )  buf[strlen(buf)-1] = '¥0';
    switch( buf[0] ){
    case 'N':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        N = atoi( ptr1 );
        // ここで枝を記録する2次元配列を構築
        break;
    }
}
```

区切り文字(文字列を空白文字で区切る)

データ例

N 10000

行末の改行文字を消す(ヌル文字に書き換え)

最初の1文字が 'N' のとき(例えば"N 10000")

"N "を読み取る

ptr1は"10000"を指す

点数を表す数字文字列を数値(整数)に変換

最初の1文字が 'E' や 'L' のときの処理をここに書き加える

注意: ファイルでは、枝の情報(E)よりも先に点の数の情報(N)が記述されている必要あり

枝の読み込みと記録

```
int nFrom, nTo;
while(1){
    // ここでファイルから1行読み込み(詳細は省略)
    switch( buf[0] ){
    case 'E':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        ptr2 = strtok_s( NULL, DLM, &pTokenContext );
        ptr = strtok_s( NULL, DLM, &pTokenContext );
        nFrom = atoi( ptr1 );
        nTo = atoi( ptr2 );
        if(nFrom < 0 || nFrom >= N || nTo < 0 || nTo >= N ) continue;
        a[nFrom][nTo] = atoi( ptr );
        break;
    }
```

枝の始点

枝の終点

枝の重み

枝の情報を記録

始点、終点が範囲外*なら
枝の情報を無視(スキップ)

(*データファイルが正しければ生じない)

データ例

```
E 0 1 2
E 0 2 1
E 1 3 0
```

最短経路長(解)の読み込みと記録

```
int nNodeL,nLengthShortest;
while(1){
    // ここでファイルから1行読み込み(詳細は省略)
    switch( buf[0] ){
    case 'L':
        ptr = strtok_s( buf, DLM, &pTokenContext );
        ptr1 = strtok_s( NULL, DLM, &pTokenContext );
        ptr2 = strtok_s( NULL, DLM, &pTokenContext );
        nNodeL = atoi( ptr1 );
        nLengthShortest = atoi( ptr2 );
        break;
    }
```

データ例

L 9999 509

← 点

← 最短経路長

このグラフにおいて
点nNodeLの最短経路長がnLengthShortest

- 自分がプログラムで求めた結果のu[nNodeL]がnLengthShortest(正解)と一致するか確認

Bellman-Ford法のプログラム

```
#define INTINF 999999
int i,j;
int bUpdate;
int *u = (int *)malloc(N*sizeof(int));

for(i=1 ; i<N ; i++) u[i] = INTINF;
u[0] = 0;

while(1){
    bUpdate = 0;
    for(j=0 ; j<N ; j++){
        for(i=0 ; i<N ; i++){
            if( u[j] > u[i]+a[i][j] ){
                u[j] = u[i]+a[i][j];
                bUpdate = 1;
            }
        }
    }
    if( bUpdate == 0 ) break;
}
```

正無限大を表す十分大きな値
(すでに定義(define)済みならば改めて定義しなくてよい)

経路長を記録する配列を用意

各点の経路長を無限大に初期化

点0を始点とし、経路長は0とする

無限ループ
(最短経路長が求まるまで繰り返し)

iとjに関する2重ループ

経路長を更新したことを記録

経路長の更新がなかった、
すなわち最短経路長が求まったので終了

改善1: ループ多重化の順序検討

- Bellman方程式とBellman-Ford法プログラム

Bellman方程式 $u_j = \min_i \{u_i + w_{ij}\}$

⇩ Bellman方程式の定義通りにプログラム化

```
for(j=0 ; j<N ; j++){
    for(i=0 ; i<N ; i++){
        if( u[j] > u[i]+a[i][j] ){
            u[j] = u[i]+a[i][j];
        }
    }
}
```

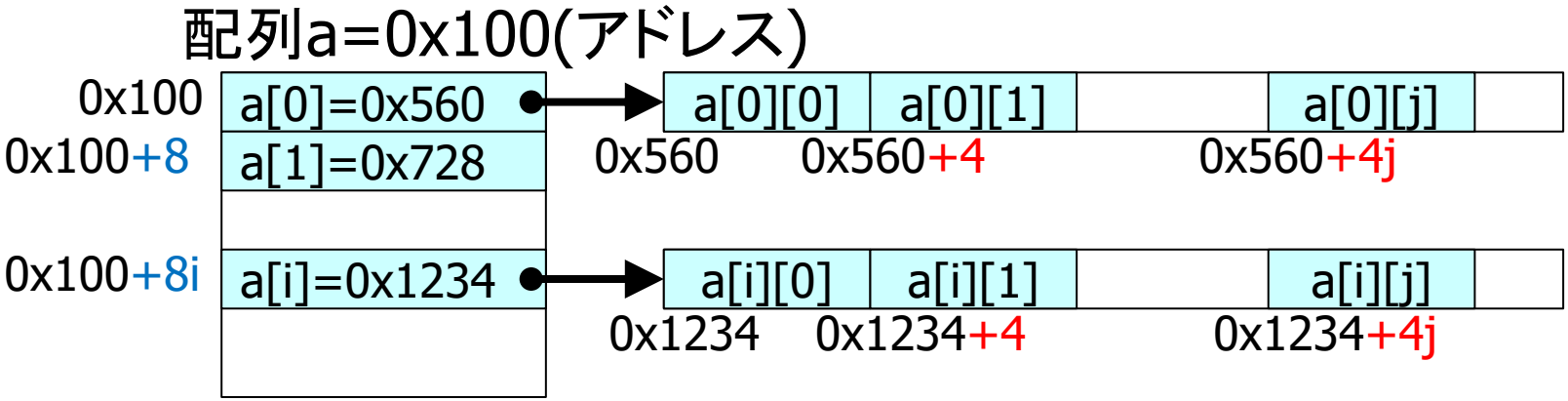
⇦ jに関するループの中にiに関するループがある二重ループになっている

・・・配列aがa[0][j],a[1][j],a[2][j],…の様に参照される

改善1: ループ多重化の順序検討

- 配列要素の参照(読み書き)は手間がかかる

整数型配列 a[i][j]（要素1つ当たり4バイト)



① iの値を8倍*する　　計算機システムが行う処理
② 配列aのアドレスに加算してa[i]のアドレスを求める
③ a[i]の値(配列a[i]の先頭アドレス)を読み出す
④ jの値を4倍する
⑤ 配列a[i]のアドレスに加算してa[i][j]のアドレスを求める
⑥ a[i][j]を読み出し、または書き込み

*64ビットモードの場合。32ビットモードの場合は4倍する。

改善1: ループ多重化の順序検討

- iに関するループとjに関するループを入れ替え

```
for(j=0 ; j<N ; j++){
    for(i=0 ; i<N ; i++){
        if( u[j] > u[i]+a[i][j] ){
            u[j] = u[i]+a[i][j];
        }
    }
}
```

```
for(i=0 ; i<N ; i++){
    for(j=0 ; j<N ; j++){
        if( u[j] > u[i]+a[i][j] ){
            u[j] = u[i]+a[i][j];
        }
    }
}
```

i値とj値のすべての組み合わせを試すことは変化なし

➡ 得られる結果に違いはない

・・・配列aがa[i][0],a[i][1],a[i][2],…の様に参照される

C言語コンパイラの最適化により、前ページ①②③の処理がjに関するループの外で1回だけ実行されるようになる

※使用するコンパイラや最適化レベルによって効果は異なる

改善2: 配列→リスト

- 枝(i,j)の重みを二次元配列で記録　a[i][j]
- 点jを0,1,2,…と順に増やして枝(i,j)の有無を調査

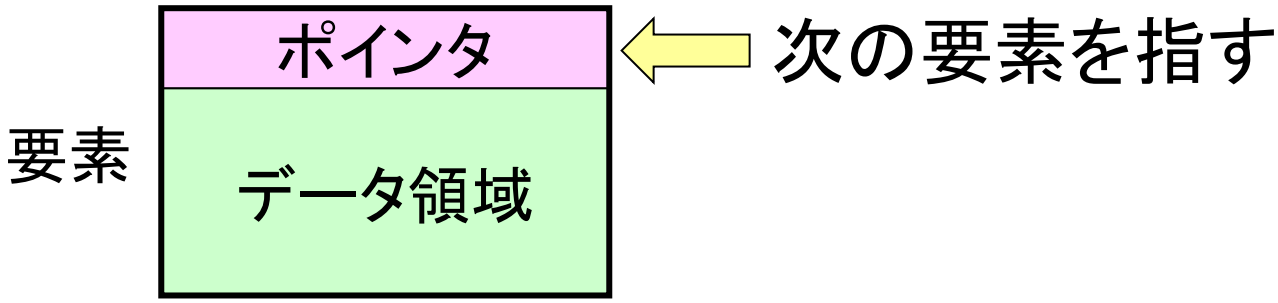
  枝数が少ない場合は無駄が多く、効率が悪い
  (多くの場合に点i, jの間に枝は存在しない)

- 二次元配列の代わりに
  枝が存在する終点jについてのみ、
  終点jと枝重みを記録する方式が必要

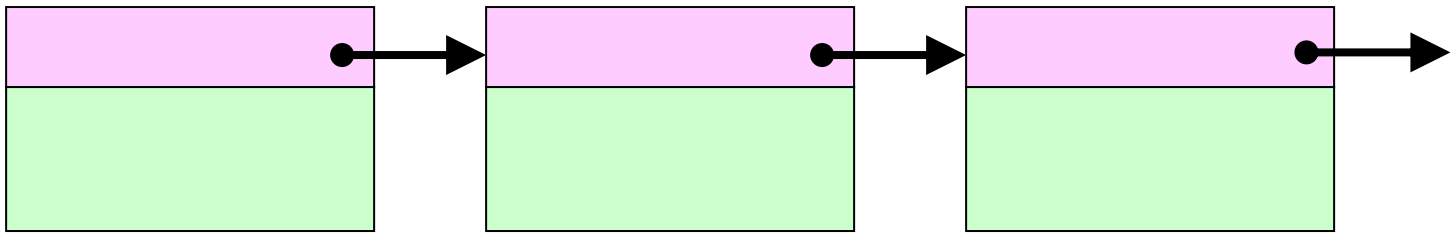
  ⇒ リスト構造(Linked list)を利用

リスト構造とは

- 要素が順につながったデータ構造
- データ領域の他に、次の要素を指すポインタを備える



- ポインタを用いて要素をつなげる=リスト

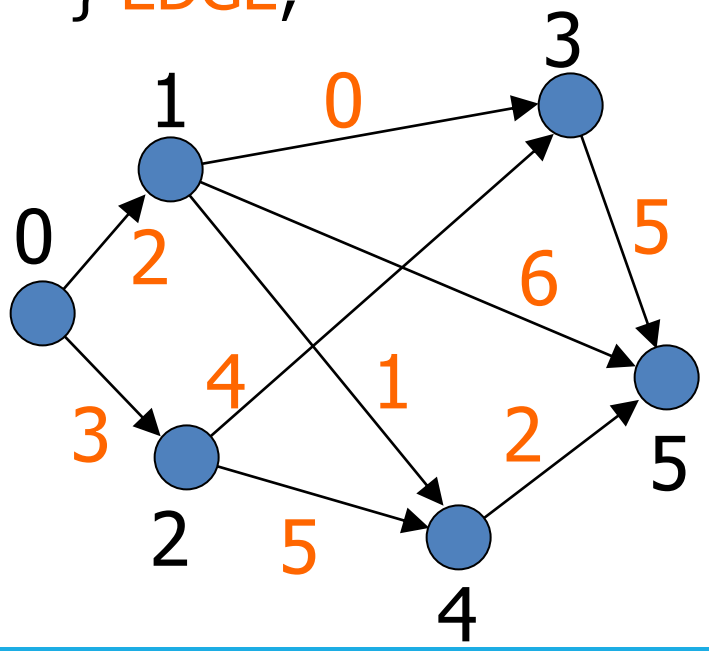


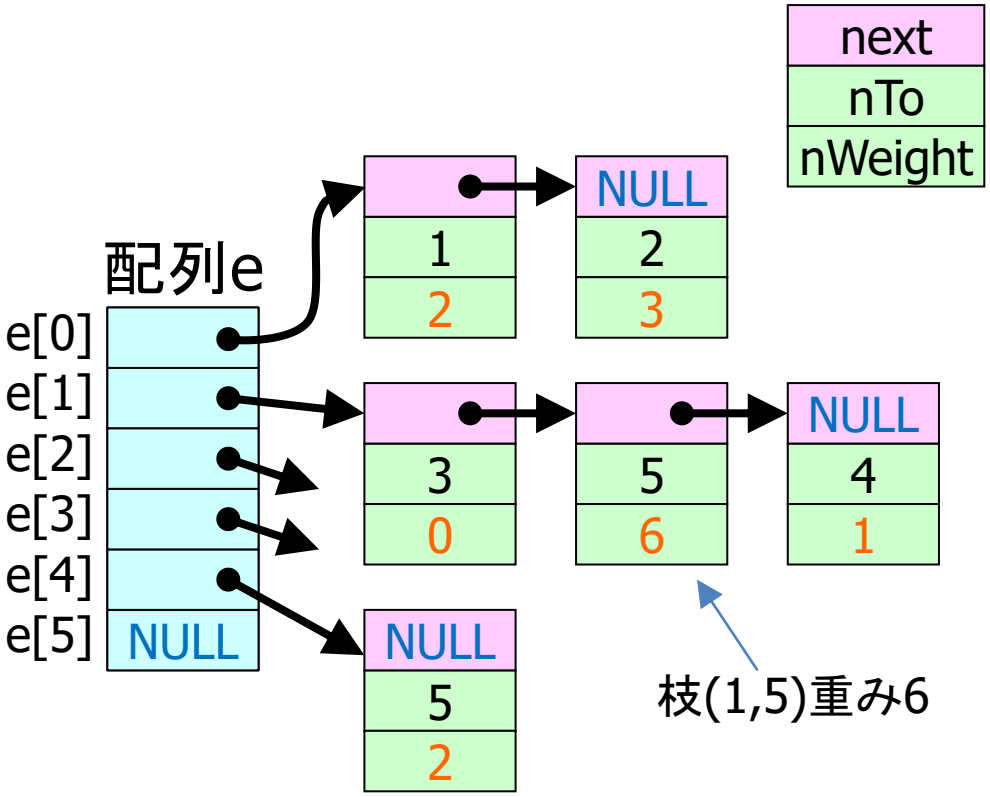
グラフの枝をリスト構造で表す

- リストの要素(枝)

```
typedef struct EdgeRec {
struct EdgeRec *next;   // 次の要素を指すポインタ
int nTo;          // 終点番号
int nWeight;  // 枝重み
} EDGE;
```

構造体





グラフの枝をリスト構造で表す

始点の配列を作成

```
int N;
EDGE **e;
int i;

e = (EDGE **)malloc(N*sizeof(EDGE *));
for( i=0 ; i<N ; i++ ) e[i] = NULL;
```

グラフの点の数

始点ごとの枝リストへの先頭を記録する1次元配列

要素数Nの配列を割り当て

各リストを空に初期化

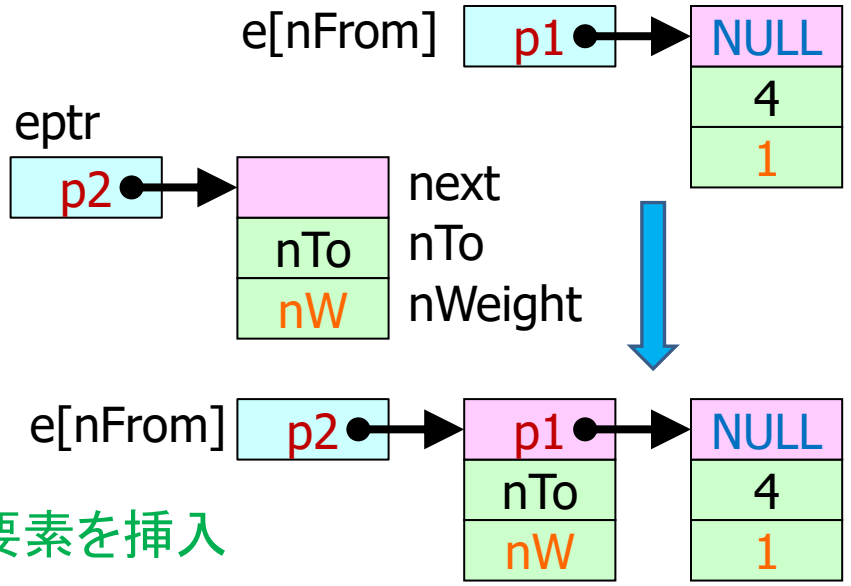
枝(nFrom,nTo)を追加

```
int nTo,nFrom,nW;
EDGE *eptr;

eptr = (EDGE *)malloc(sizeof(EDGE));
eptr->nTo = nTo;
eptr->nWeight = nW;
eptr->next = e[nFrom];
e[nFrom] = eptr;
```

要素を1つ確保

リストの先頭に要素を挿入



グラフの枝をリスト構造で表す

- 枝を順に調べる

```
int i,j,nW;
EDGE *eptr;

for( eptr=e[i] ; eptr!=NULL ; eptr=eptr->next ){
      j = eptr->nTo;
      nW = eptr->nWeight;
      // ここで枝(i,j)の重みがnW
      // 点jの最短経路長の比較・更新などを処理
}
```

始点iの枝を順に調べる

終点j、枝重みnW

Bellman-Ford法+リスト構造

```
#define INTINF 999999
int i,j;
int bUpdate;
int *u = (int *)malloc(N*sizeof(int));
for( i=1 ; i<N ; i++ ) u[i] = INTINF;
u[0] = 0;

while(1){
    bUpdate = 0;
    for( i=0 ; i<N ; i++ ){
        点iを始点とする枝(i,j)のそれぞれについて{
            L = u[i]+枝(i,j)の重み;
            if( u[j] > L ){
                u[j] = L;
                bUpdate = 1;
            }
        }
    }
    if( bUpdate == 0 ) break;
}
```

正無限大を表す十分大きな値
(すでに定義(define)済みならば改めて定義しなくてよい)

経路長を記録する配列を用意

各点の経路長を無限大に初期化

点0を始点とし、経路長は0とする

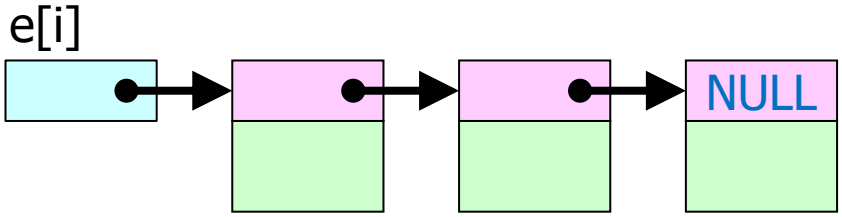
無限ループ
(最短経路長が求まるまで繰り返し)

経路長を更新したことを記録

経路長の更新がなかった、
すなわち最短経路長が求まったので終了

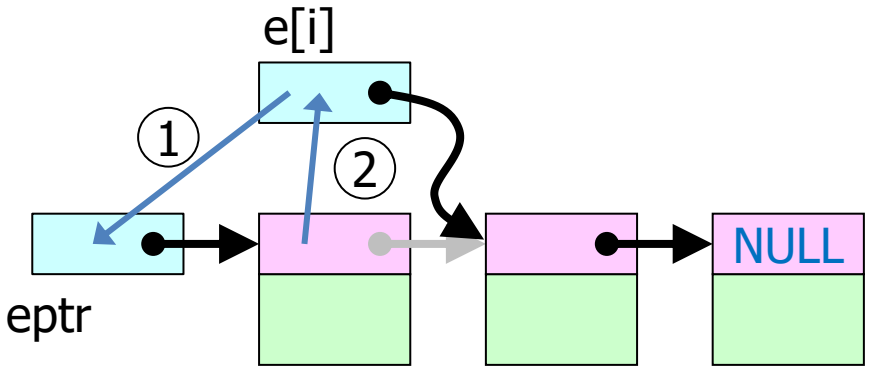
不要になったリストの削除

- 要素のために確保したメモリを解放する



リスト上の要素を先頭から順に削除

```
while( e[i] ){
        eptr = e[i];          ①
        e[i] = e[i]->next;    ②
        free( eptr );
}
```



全てのiについてリスト削除
配列e自体も解放(free)する

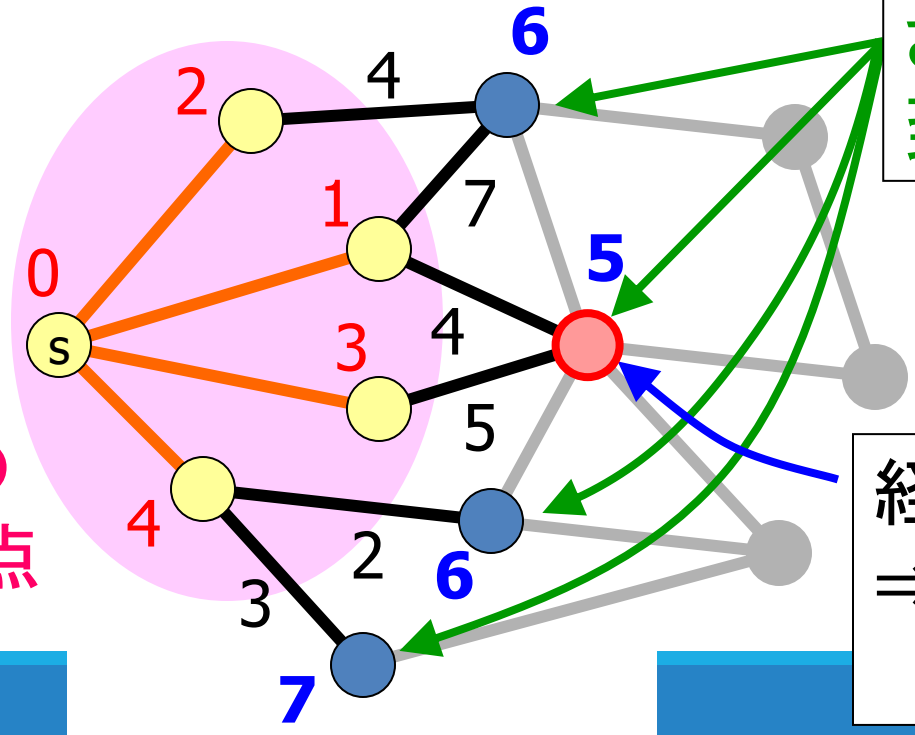
改善3: アルゴリズムを変更

- Bellman-Ford法は、枝重みが正、ゼロ、負の場合に最短経路を求める
  (ただし、枝重み和が負のループはない場合)
- 枝重みが正、ゼロ(つまり負でない)に限定される場合は、もっと効率よく最短経路を求めることができる
- 今回のプログラミング演習では、枝重みは正、ゼロのみ

非負枝重みの最短経路問題(1)

- 枝重みが非負(0または正)に限定されている場合

最短経路長の定まった点から、
さらに枝1本で到達する点のうち、
経路長最短の点は、最短経路と決定できる



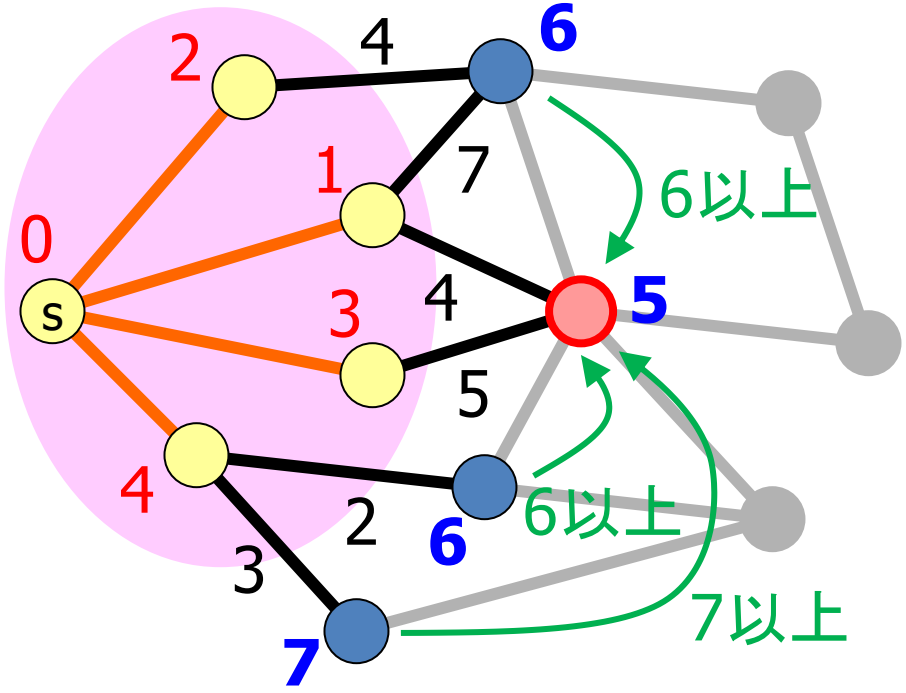
さらに枝1本で
到達する点

最短経路長の
決まっている点

経路長が最小
⇒最短経路を
決定できる点

非負枝重みの最短経路問題(2)

- 枝重みが非負(0または正)なので、他の点を経由したときの経路長は、他の点の経路長以上



現在の最小値5が
その点の最短経路長

全ての点の最短経路長が決まるまで繰り返す

Dijkstra法　計算複雑度 $O(N^2)$　(点数N)

Dijkstra法の実装1

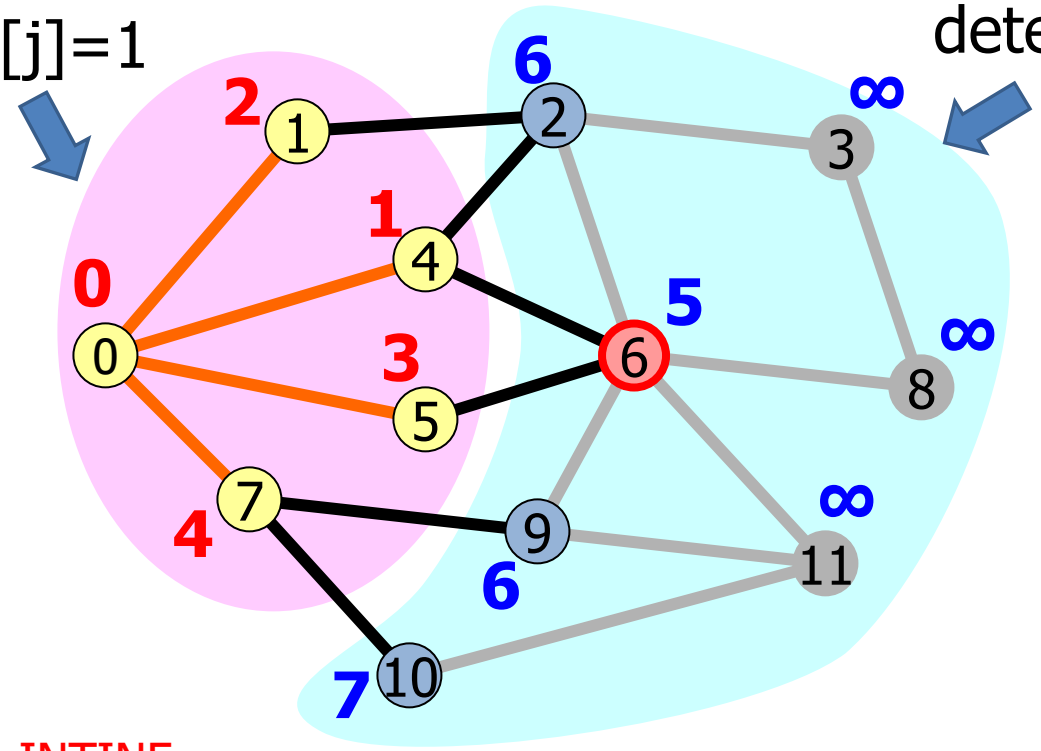
```
#define INTINF 999999   ← 正無限大を表す十分大きな値
int *u = (int *)malloc(N*sizeof(int));
int *determined = (int *)malloc(N*sizeof(int));
int L;
for(i=0 ; i<N ; i++) determined[i] = 0;   ← 最短経路長決定済みか否か
for(i=1 ; i<N ; i++) u[i] = INTINF;   ← 最短経路長uを無限大に初期化
s = 0;
u[s] = 0;   ← 始点を点0とし、その最短経路長は0とする
determined[s] = 1;   ← 始点sの最短経路長は決定済み

for(i=1 ; i<N ; i++){   ← 残りのN-1個の点について最短経路長を求める
    点sを始点とする枝(s,j)のそれぞれについて{   ①
        L = u[s]+枝(s,j)の重み;
        if( u[j] > L ) u[j] = L;   ← Lは枝(s,j)を通り点jへ到達する経路の経路長
    }
    最短経路長が未確定(determined[j]=0)な点のうち、u[s]が最小の点sを求める;   ②
    determined[s] = 1; //点sの最短経路長を確定
}
```

Dijkstra法の実装1の考え方

最短経路長決定済み
determined[j]=1

最短経路長未決定
determined[j]=0



② の実装

```
int nMin = INTINF;
for( j=1 ; j<N ; j++ ){
      if( determined[j]==0 && nMin>u[j] ){
            nMin = u[j];
            s = j;
      }
}
```

← 最短経路長が未決定の点の中で、経路長が最短の点を探索する

改善4: Dijkstra法の実装1の改善

2つのループ

```
①  for( eptr=e[s] ; eptr!=NULL ; eptr=eptr->next ){
        j = eptr->nTo;
        L = u[s]+eptr->nWeight;
        if( u[j] > L ) u[j] = L;
    }

    int nMin = INTINF;
②  for( j=1 ; j<N ; j++ ){
        if( determined[j]==0 && nMin>u[j] ){
            nMin = u[j];
            s = j;
        }
    }
```

点sを始点とする全ての枝について繰り返し

(0を除く)全ての点について繰り返し

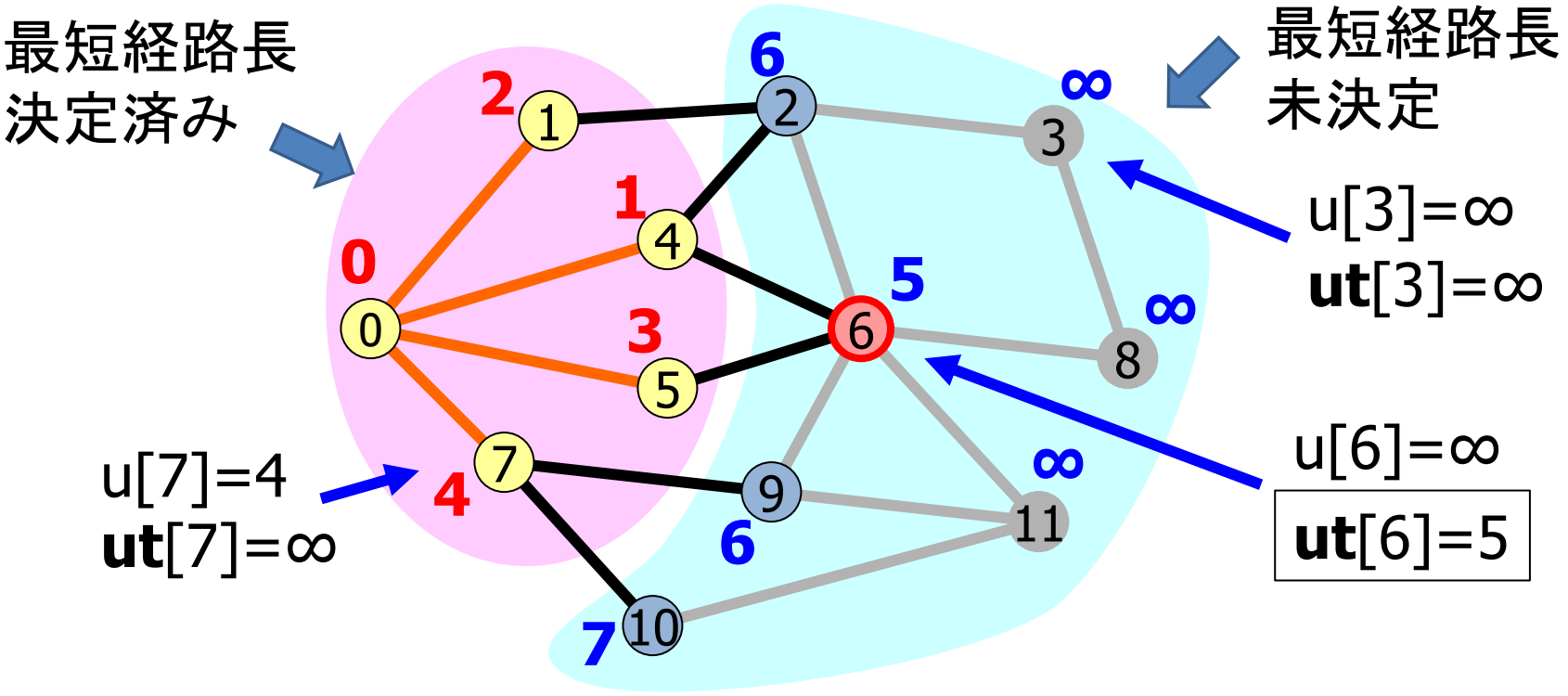
“determined[j]==0”と“nMin>u[j]”の2つの条件判定を実行

一般に、①の繰り返し回数よりも②の繰り返し回数が多い

⇒ ②のループ内の処理を簡素化すると高速化

Dijkstra法の実装2の考え方

- 2つの最短経路長配列を利用 { u[n]: 最短経路長 / **ut**[n]: 暫定最短経路長



最短経路長決定済みの点については **ut**[n]=∞ とする

➡ **ut**[j] が最小の点を探索すれば、経路長未決定の点の中から暫定経路長最短の点を探索することになる

Dijkstra法の実装2

```
int *u = (int *)malloc(N*sizeof(int));
for(i=0 ; i<N ; i++) ut[i] = INTINF;
for(i=1 ; i<N ; i++) u[i] = INTINF;
s = 0;
u[s] = 0;

for(i=1 ; i<N ; i++){
    点sを始点とする枝(s,j)のそれぞれについて{ ①
        L = u[s]+枝(s,j)の重み;
        if( u[j]==INTINF && ut[j] > L ) ut[j] = L;
    }
    int nMin = INTINF;
    for( j=1 ; j<N ; j++ ){
        if( nMin > ut[j] ){ ②
            nMin = ut[j];
            s = j;
        }
    }
    u[s] = ut[s];
    ut[s] = INTINF
}
```

暫定最短経路長ut、および最短経路長uを無限大に初期化

始点を点0とし、その最短経路長は0とする

残りのN-1個の点について最短経路長を求める

Lは枝(s,j)を通り点jへ到達する経路の経路長

点jは最短経路長が未決定

ループ②の中の処理が実装1より減少

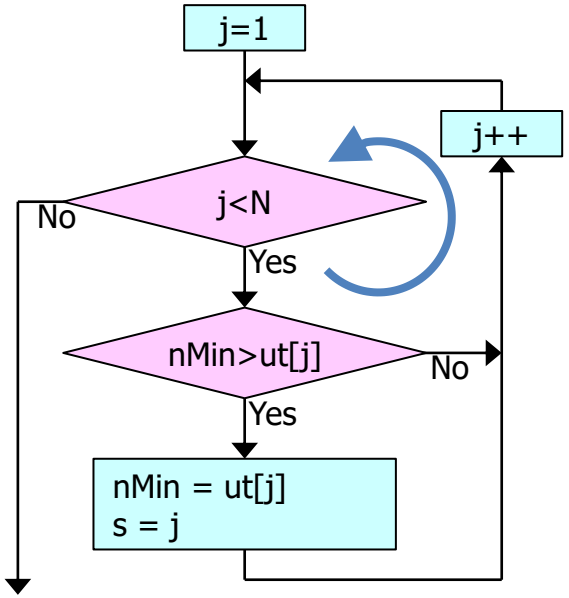
点sの最短経路長を確定

点sの暫定最短経路長を無限大に

改善5: 監視員法(sentinel法)

- ループ②に着目

```
int nMin = INTINF;
for( j=1 ; j<N ; j++ ){
    if( nMin > ut[j] ){
        nMin = ut[j];   ← 最小値をnMinに記録
        s = j;
    }
}
```

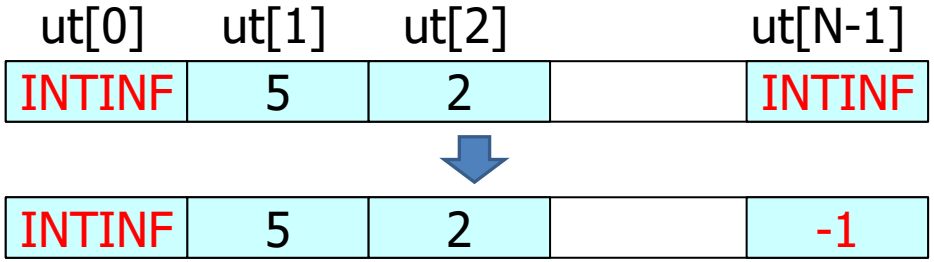


ほとんどの場合にj<Nが成り立ち、
次にnMin>ut[j]を調べるので
必ず2回比較を実行している

比較の実行回数を減らす工夫として、
nMin>ut[j]が成り立つときにのみj<Nを調べる

監視員(sentinel)とは

- ループ最後のj==N-1のときに必ずnMin>ut[j]が成り立つようにする



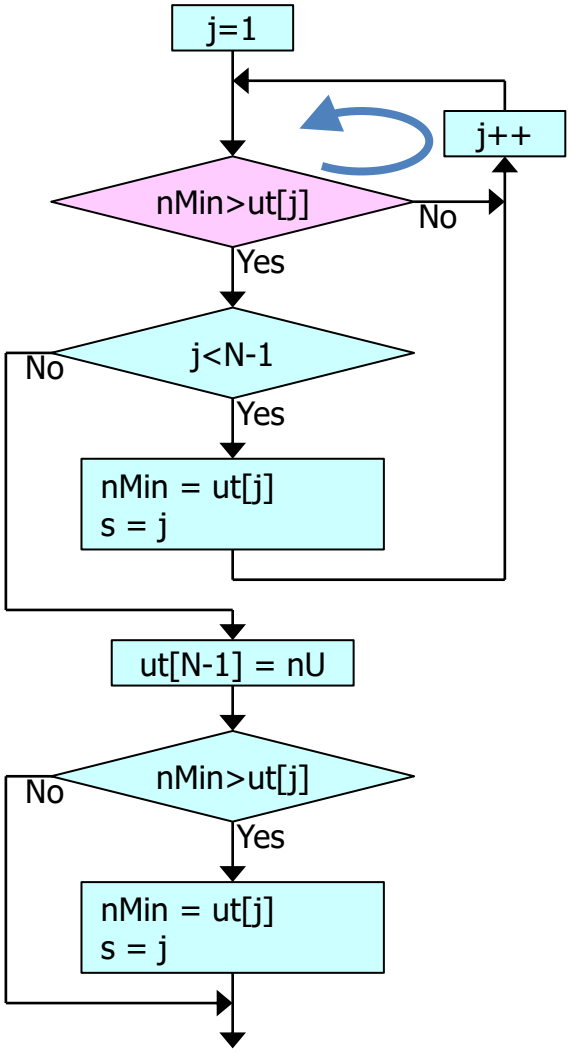
十分小さな値
j==N-1のとき
nMin>ut[j]が
必ず成立
・・・ループの最後を監視

- nMin>ut[j]が成り立つときに、j<N-1かj==N-1かによって必要な処理を行う

監視員法(sentinel法)の実装

```
int nMin = INTINF;
int nU = ut[N-1];          ← ut[N-1]の値を保存
ut[N-1] = -1;              ← 十分小さな値を設定
for( j=1 ; ; j++ ){        ← jを1ずつ増やしながら繰り返し
    if( nMin > ut[j] ){
        if( j < N-1 ){     ← j<N-1の場合
            nMin = ut[j];
            s = j;
        }else{             ← j==N-1の場合
            ut[N-1] = nU;  ← ut[N-1]の値を戻す
            if( nMin > ut[j] ){ ← 改めて比較実行
                nMin = ut[j];
                s = j;
            }
            break;
        }
    }
}
```

ここに比較が無いのがポイント



nMin>ut[j]が成り立つ確率は低いので、
平均的に比較実行回数が低減

発展

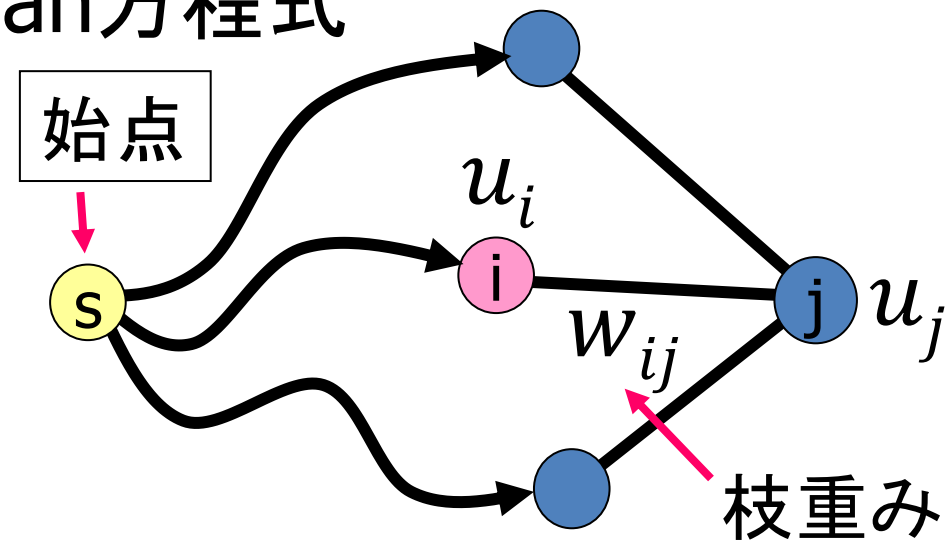
- 最短経路長を与える経路を知るには？
- 2番目の最短経路、3番目の最短経路は？

経路長最短の経路を表示する(発展)

最短経路に関するBellman方程式

$$u_s = 0$$

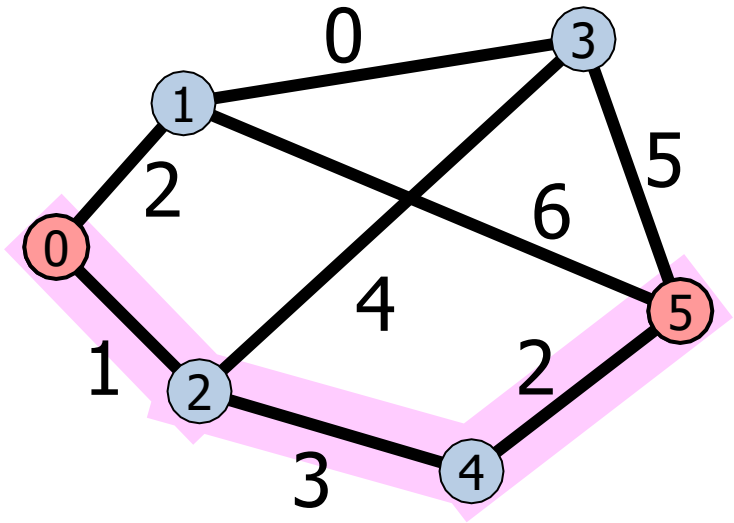
$$u_j = \min_i \{u_i + w_{ij}\} \quad \forall j \neq s$$



最短経路上で直前の点がどれかが分かる

➡ 配列 prev[j] を用意し、prev[j] = i を記録

経路長最短の経路を表示する(発展)



prev[0] = -1
prev[1] = 0
prev[2] = 0
prev[3] = 1
prev[4] = 2
prev[5] = 4

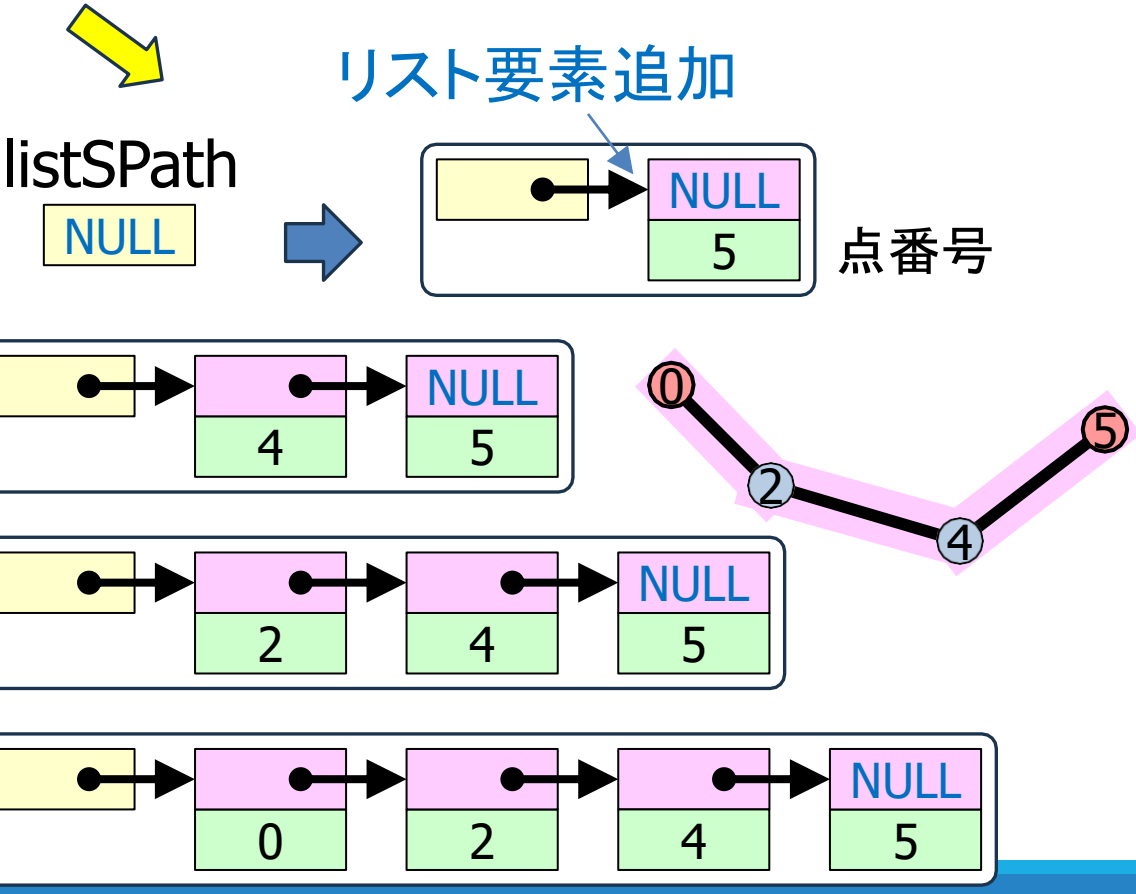
終点から順に直前点をたどる
終点番号 5
➡ 4 = prev[5] ➡ 2 = prev[4] ➡ 0 = prev[2]

経路は 5 ← 4 ← 2 ← 0 ・・・逆順になっている

経路長最短の経路を表示する(発展)

```
点番号 j = 5; // 経路の終点
while( j >=0 ){
    点jを経路として記録;
    j = prev[j];
}
```

prev[0] = -1 ・・・始点まで
たどれば終了



リストへの追加
により逆順化

リストの先頭から
順に表示

経路長最短の経路を表示する(発展)

```
int *prev = (int *)malloc( N*sizeof(int) );
```

最短経路を求める
　配列prevに直前点を記録

配列prevを基に最短経路上の点をリストに逆順に記録

リスト上の点を順に表示

リスト要素と配列prevのメモリ解放

以上