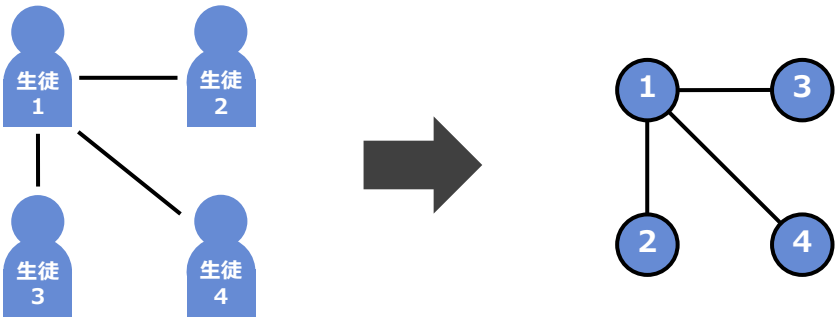
第 9 章

グラフアルゴリズム

9.1 問題 B61：Influencer（難易度：★2相当）

まず、この問題で出てくる友達関係は、以下のように重みなし無向グラフで表現することができます。



このとき、**「生徒 i の友達の人数」は「グラフの頂点 i の次数」と一致する**ため、次数が最も大きい頂点の番号を出力すれば正解です。

隣接リスト表現を使ってグラフを管理したときの実装例を以下に示します。`G[i].size()` は頂点 i の次数に対応することに注意してください。

解答例（C++）

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

int N, M, A[100009], B[100009];
vector<int> G[100009];

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]); // 「頂点 A[i] に隣接する頂点」として B[i] を追加
        G[B[i]].push_back(A[i]); // 「頂点 B[i] に隣接する頂点」として A[i] を追加
    }

    // 次数（= 友達の数）が最大となる生徒の番号を求める
    int MaxFriends = 0; // 友達の数の最大値
    int MaxID = 0;      // 番号
```

```
    for (int i = 1; i <= N; i++) {
        if (MaxFriends < (int)G[i].size()) {
                MaxFriends = (int)G[i].size();
                MaxID = i;
        }
    }

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

※Python のコードはサポートページをご覧ください

補足：グラフを使わない解法

9 章のテーマは「グラフアルゴリズム」なので、解答例ではグラフを使って実装しました。しかし、以下のようにグラフを使わない解法もあります。

- 生徒 i の現在の友達の数を表す配列 `cnt[i]` を用意する
- $i = 1,2,\ldots,M$ について、`cnt[A[i]]` と `cnt[B[i]]` に 1 を足す

これを実装すると、以下のようになります。計算量は $O(N+M)$ です。

```
// 入力
int cnt[100009];
cin >> N >> M;
for (int i = 1; i <= N; i++) cnt[i] = 0;
for (int i = 1; i <= M; i++) {
    cin >> A[i] >> B[i];
    cnt[A[i]] += 1; cnt[B[i]] += 1;
}

// 友達の数が最大となる生徒の番号を求める
int MaxFriends = 0; // 友達の数の最大値
int MaxID = 0;      // 番号
for (int i = 1; i <= N; i++) {
    if (MaxFriends < cnt[i]) {
        MaxFriends = cnt[i];
        MaxID = i;
    }
}
cout << MaxID << endl;
```

9.2 問題 B62：Print a Path (難易度：★4相当)

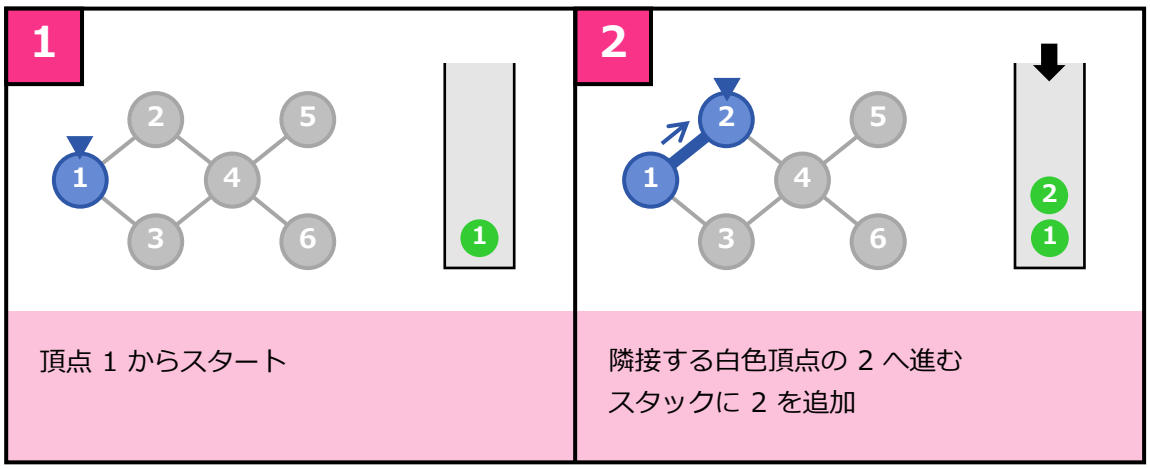
本の 9.2 節では、「進めるだけ進み、行き詰まったら一歩戻る」という深さ優先探索を使って、頂点 1 から各頂点まで到達可能かどうかを判定する方法を紹介しました。しかし、頂点 1 から N までの具体的な経路を得るにはどうすれば良いのでしょうか。

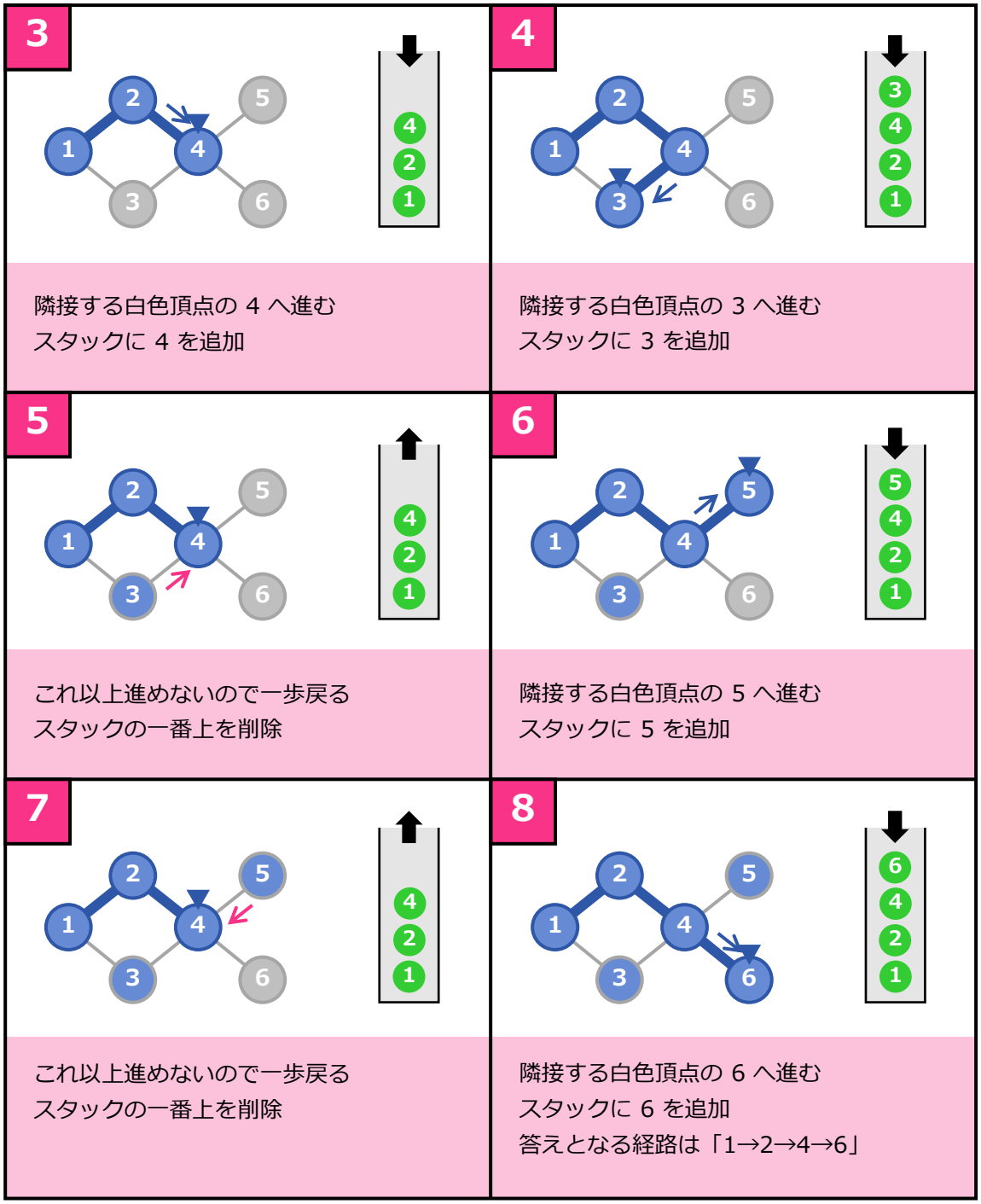
経路を得る方法

まず、以下のようにして、**深さ優先探索における移動経路の跡（本の 354 ページの図の青線部分に対応）**を管理することを考えます。

- スタック Path を用意する。
- 頂点 pos へ進んだときには、Path の一番上に pos を追加する。
- 一歩戻るときには、Path の一番上の要素を削除する。

このとき、頂点 1 から N までの経路は、「頂点 N へ進んだ時点での移動経路の跡」となります。具体例を以下に示します。





ここまでの内容を実装すると、以下の解答例のようになります。頂点 N にたどり着いた時点でのスタックの要素を、下から順番に出力すれば良いです。

解答例（C++）

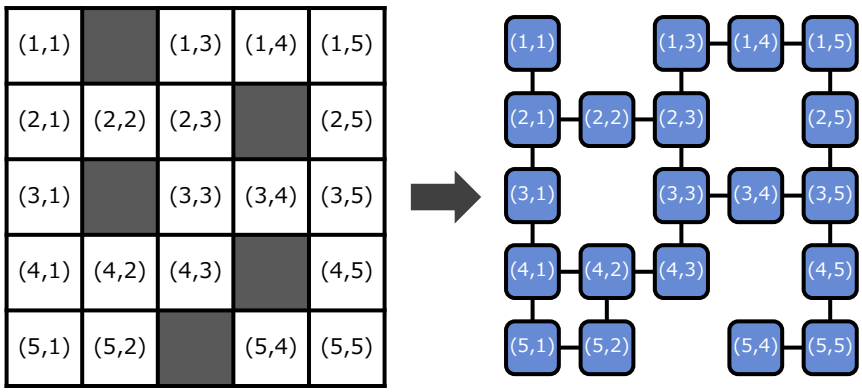
```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;
```

```cpp
int N, M, A[100009], B[100009];
vector<int> G[100009];   // グラフ
bool visited[100009];    // 頂点 i が青か白か
stack<int> Path, Answer; // 移動経路の跡

void dfs(int pos) {
    // ゴール地点にたどり着いた！
    if (pos == N) {
        Answer = Path;
        return;
    }

    // その他の場合
    visited[pos] = true;
    for (int i = 0; i < G[pos].size(); i++) {
        int nex = G[pos][i];
        if (visited[nex] == false) {
                Path.push(nex); // 頂点 nex を経路に追加
                dfs(nex);
                Path.pop(); // 頂点 nex を経路から削除
        }
    }
    return;
}

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 深さ優先探索
    for (int i = 1; i <= N; i++) visited[i] = false;
    Path.push(1); // 頂点 1（スタート地点）を経路に追加
    dfs(1);

    // スタックの要素を「下から順に」に記録
    vector<int> Output;
    while (!Answer.empty()) {
        Output.push_back(Answer.top());
        Answer.pop();
    }
    reverse(Output.begin(),Output.end());

    // 答えの出力
    for (int i = 0; i < Output.size(); i++) {
        if (i >= 1) cout << " ";
        cout << Output[i];
    }
    cout << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

9.3 問題 B63：幅優先探索 (難易度：★4相当)

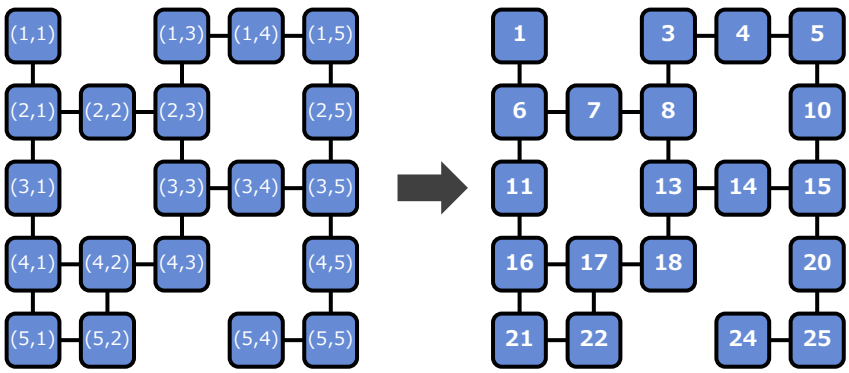
まず、迷路を以下のようなグラフとして表現することを考えます。マスが頂点、隣接するマスの関係が辺に対応しています。



すると、左上マスから右下マスまでの最短手数は、**頂点 $(1,1)$ から頂点 (H,W) までの最短経路長**になります。

そのため、例題 A63（9.3 節）でやったように、幅優先探索で最短経路長を計算すれば正解が得られます。

なお、次ページの実装例では、頂点 (i,j) の番号を 1 つの整数 $(i-1)\times W+j$ で表していることに注意してください※1。



※1　もちろん、二次元配列 `vector<int> G[i][j]` を使って隣接リストを管理すれば、頂点番号がそのままでも幅優先探索を行うことができますが、1 つの整数の方が実装が楽です

解答例（C++）

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

// 入力
int H, W;
int sx, sy, start; // スタートの座標 (sx, sy) と頂点番号 sx*H+sy
int gx, gy, goal;  // ゴールの座標 (gx, gy) と頂点番号 gx*W+gy
char c[59][59];

// グラフ・最短経路
int dist[2509];
vector<int> G[2509];

int main() {
    // 入力
    cin >> H >> W;
    cin >> sx >> sy; start = sx * W + sy;
    cin >> gx >> gy; goal = gx * W + gy;
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) cin >> c[i][j];
    }

    // 横方向の辺 [(i, j) - (i, j+1)] をグラフに追加
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W - 1; j++) {
                int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
                int idx2 = i * W + (j + 1); // 頂点 (i, j+1) の頂点番号
                if (c[i][j] == '.' && c[i][j + 1] == '.') {
                        G[idx1].push_back(idx2);
                        G[idx2].push_back(idx1);
                }
        }
    }

    // 縦方向の辺 [(i, j) - (i+1, j)] をグラフに追加
    for (int i = 1; i <= H - 1; i++) {
        for (int j = 1; j <= W; j++) {
                int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
                int idx2 = (i + 1) * W + j; // 頂点 (i+1, j) の頂点番号
                if (c[i][j] == '.' && c[i + 1][j] == '.') {
                        G[idx1].push_back(idx2);
                        G[idx2].push_back(idx1);
                }
        }
    }

    // 幅優先探索の初期化
    for (int i = 1; i <= H * W; i++) dist[i] = -1;
    queue<int> Q;
    Q.push(start); dist[start] = 0;
```

```
    // 幅優先探索
    while (!Q.empty()) {
        int pos = Q.front();
        Q.pop();
        for (int i = 0; i < G[pos].size(); i++) {
                int to = G[pos][i];
                if (dist[to] == -1) {
                        dist[to] = dist[pos] + 1;
                        Q.push(to);
                }
        }
    }

    // 答えを出力
    cout << dist[goal] << endl;
    return 0;
}
```

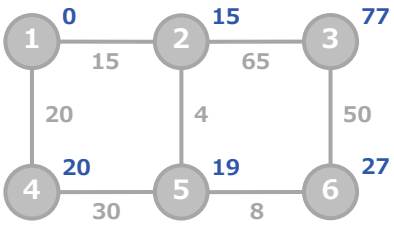
※Python のコードはサポートページをご覧ください

9.4 問題 B64：Shortest Path (難易度：★4相当)

4.2 節では、動的計画法で具体的な解を求める方法として、「逆から順番に考える方法」を紹介しました。

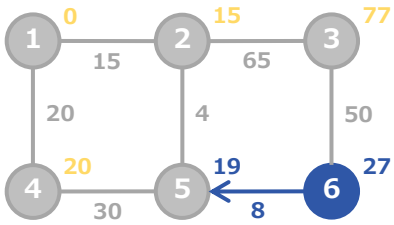
今回も同じように、**ゴール地点の頂点 N から順番に考えれば**、頂点 1 から頂点 N までの具体的な最短経路を求めることができます。例を以下に示します（本の 362 ページの入力例と同じです）。

1



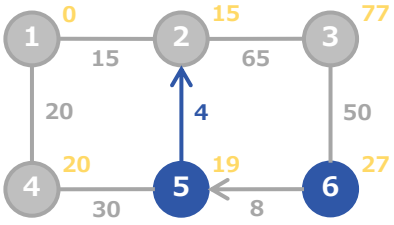
まずはダイクストラ法で、頂点 1 から各頂点までの最短経路長を求める

2



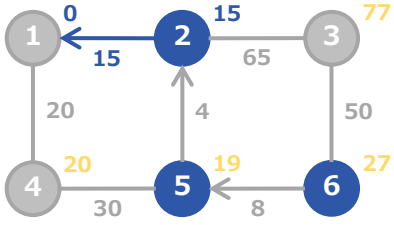
頂点 6 の一個前の頂点は？
dist[5]+8=27 なので、頂点 5

3



頂点 5 の一個前の頂点は？
dist[2]+4=19 なので、頂点 2

4



頂点 2 の一個前の頂点は？
dist[1]+15=15 なので、頂点 1
これで 1→2→5→6 という経路が得られた

解答例（C++）

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

// 入力・グラフ
int N, M, A[100009], B[100009], C[100009];
vector<pair<int, int>> G[100009];

// ダイクストラ法
int cur[100009]; bool kakutei[100009];
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> Q;

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i] >> C[i];
        G[A[i]].push_back(make_pair(B[i], C[i]));
        G[B[i]].push_back(make_pair(A[i], C[i]));
    }

    // 配列の初期化
    for (int i = 1; i <= N; i++) kakutei[i] = false;
    for (int i = 1; i <= N; i++) cur[i] = 2000000000;

    // スタート地点をキューに追加
    cur[1] = 0;
    Q.push(make_pair(cur[1], 1));

    // ダイクストラ法
    while (!Q.empty()) {
        // 次に確定させるべき頂点を求める
        int pos = Q.top().second; Q.pop();

        // Q の最小要素が「既に確定した頂点」の場合
        if (kakutei[pos] == true) continue;

        // cur[x] の値を更新する
        kakutei[pos] = true;
        for (int i = 0; i < G[pos].size(); i++) {
                int nex = G[pos][i].first;
                int cost = G[pos][i].second;
                if (cur[nex] > cur[pos] + cost) {
                        cur[nex] = cur[pos] + cost;
                        Q.push(make_pair(cur[nex], nex));
                }
        }
    }
```

```
    // 答えの復元（Place は現在の位置：ゴールから出発）
    vector<int> Answer;
    int Place = N;
    while (true) {
        Answer.push_back(Place);
        if (Place == 1) break;

        // Place の前の頂点としては、一体どこが良いのか？
        for (int i = 0; i < G[Place].size(); i++) {
                int nex = G[Place][i].first;
                int cost = G[Place][i].second;
                if (cur[nex] + cost == cur[Place]) {
                        Place = nex;
                        break;
                }
        }
    }
    reverse(Answer.begin(), Answer.end());

    // 出力
    for (int i = 0; i < Answer.size(); i++) {
        if (i >= 1) cout << " ";
        cout << Answer[i];
    }
    cout << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

9.5 問題 B65：Road to Promotion Hard (難易度：★4相当)

まずは簡単のため、問題 A65（本の 9.5 節）と同じように、社員 1 が上司であり、番号の小さい社員の方が偉い場合を考えましょう。

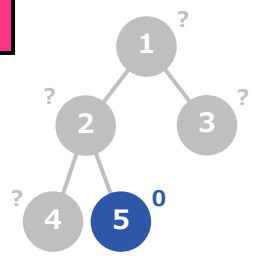
社員 x の直属の部下の階級がそれぞれ $r_1, r_2, \ldots, r_k$ であるとき、社員 x の階級は次式で表される：

$$(\text{社員 } x \text{ の階級}) = \max(r_1, r_2, \ldots, r_k) + 1$$

番号の小さい方が上司なので、社員 $N, \ldots, 2, 1$ の順に、上式の通りに階級を計算していけば、計算量 $O(N)$ で答えがわかる。

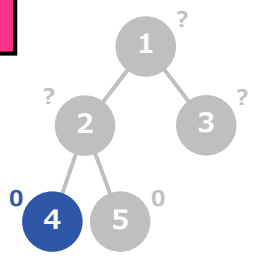
$N = 5$ のときの具体例を以下に示します。

1



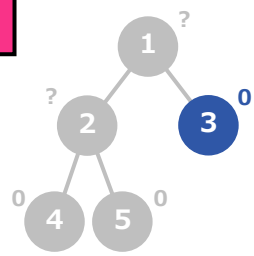
社員 5 は部下がいないので階級 0

2



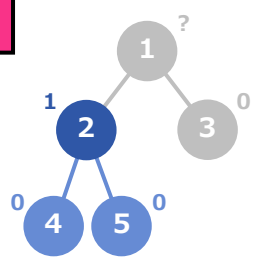
社員 4 は部下がいないので階級 0

3



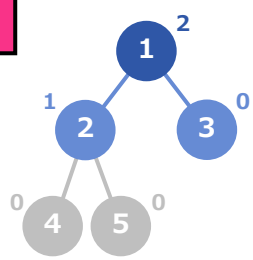
社員 3 は部下がいないので階級 0

4



社員 2 の階級は max(0,0)+1=1

5



社員 1 の階級は max(1,0)+1=2

6

すべての社員の階級がわかった！

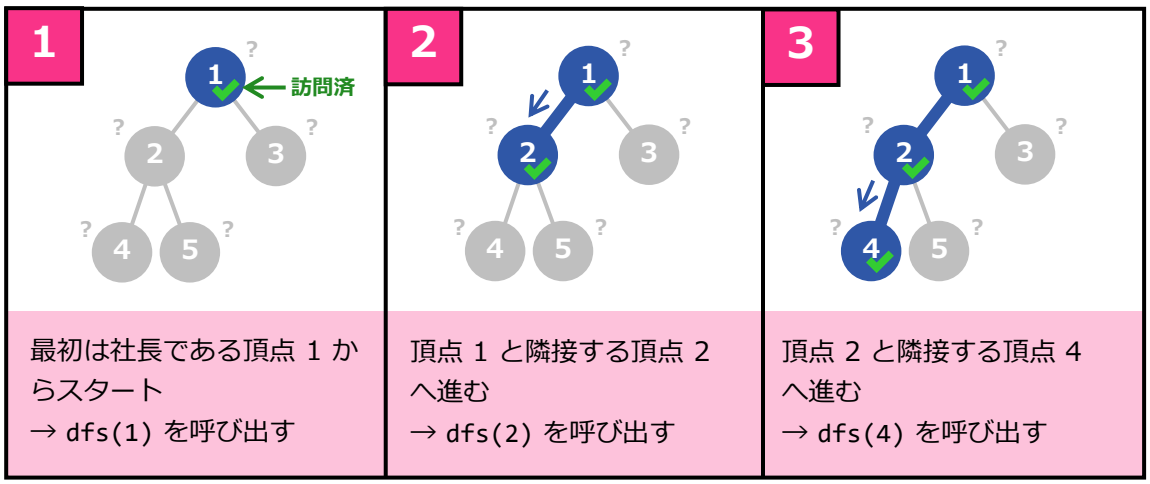
一般の場合を解く

ここまでは、番号の小さい方が偉いという簡単なケースの解き方を説明しました。ところが実際はそうとは限らないため、社員 $N, \ldots, 2, 1$ の順番に求める方法が通用しません。

そこで、深さ優先探索を使って、**社長と遠い方から順番に階級を求めていく**と上手くいきます（注：社長との距離が近い方が上司であるという性質を使っています）。実装例を以下に示します。社長の番号を X とするとき、最初は `dfs(X)` が呼び出されます※2。

```
// 深さ優先探索を行う関数（pos は現在位置）
// 返り値は社員 pos の階級
int dfs(int pos) {
    // 最初、社員 pos の階級を 0 に設定する
    visited[pos] = true;
    Answer[pos] = 0;

    // 探索をする
    for (int i = 0; i < G[pos].size(); i++) {
        int nex = G[pos][i];
        if (visited[nex] == false) {
                int ret = dfs(nex);
                Answer[pos] = max(Answer[pos], ret + 1); // 階級を更新する
        }
    }

    // 値を返す
    return Answer[pos];
}
```

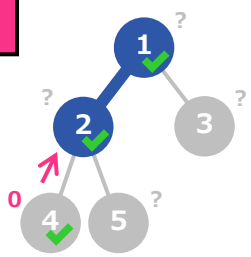
ここで、再帰関数の動きは少し複雑ですので、以下に例を示します（最初の例と同じ）。イメージとしては、**「隣接する頂点をすべて訪問して一歩戻るときに、階級を計算する」**といった感じです。



1	2	3
最初は社長である頂点 1 からスタート → `dfs(1)` を呼び出す	頂点 1 と隣接する頂点 2 へ進む → `dfs(2)` を呼び出す	頂点 2 と隣接する頂点 4 へ進む → `dfs(4)` を呼び出す

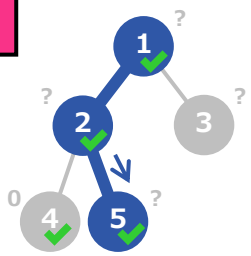
※2　`G` は上下関係を表すグラフの隣接リスト、`Answer[i]` は社員 `i` の階級を表します

4



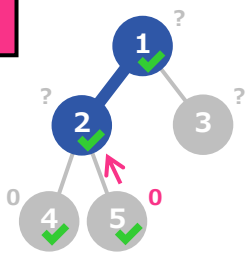
行き詰まったので一歩戻る
社員 4 の階級は 0

5



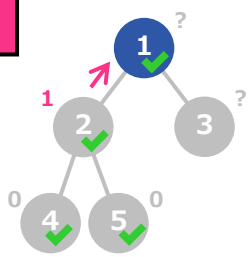
頂点 2 と隣接する頂点 5 へ進む
→ `dfs(5)` を呼び出す

6



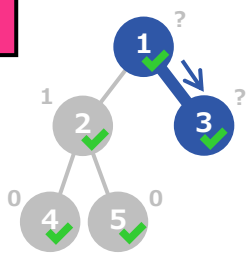
行き詰まったので一歩戻る
社員 5 の階級は 0

7



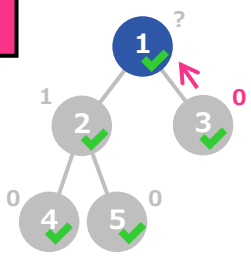
行き詰まったので一歩戻る
社員 2 の階級は
max(0,0)+1=1

8



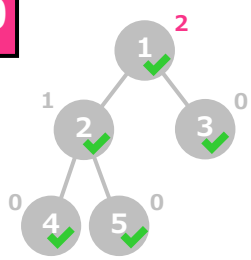
頂点 1 と隣接する頂点 3 へ進む
→ `dfs(3)` を呼び出す

9



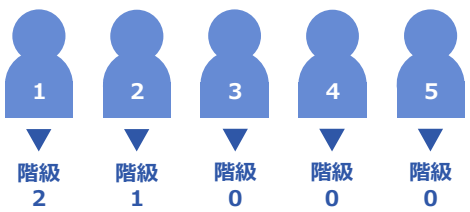
行き詰まったので一歩戻る
社員 3 の階級は 0

10



行き詰まったので計算終了
社員 1 の階級は
max(1,0)+1=2

11



これですべての社員の階級がわかった！

最後に、入力部分・出力部分・グラフの隣接リストを作る部分などを加えると、実装は以下の解答例のように穴ります。

解答例（C++）

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

```cpp
// 入力される変数・答え
int N, T, A[100009], B[100009];
int Answer[100009];

// グラフ・深さ優先探索
vector<int> G[100009];
bool visited[100009];

// 深さ優先探索を行う関数（pos は現在位置）
// 返り値は社員 pos の階級
int dfs(int pos) {
    // 最初、社員 pos の階級を 0 に設定する
    visited[pos] = true;
    Answer[pos] = 0;

    // 探索をする
    for (int i = 0; i < G[pos].size(); i++) {
        int nex = G[pos][i];
        if (visited[nex] == false) {
                int ret = dfs(nex);
                Answer[pos] = max(Answer[pos], ret + 1); // 階級を更新する
        }
    }

    // 値を返す
    return Answer[pos];
}

int main() {
    // 入力
    cin >> N >> T;
    for (int i = 1; i <= N - 1; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]); // A[i]→B[i] の方向に辺を追加
        G[B[i]].push_back(A[i]); // B[i]→A[i] の方向に辺を追加
    }

    // 深さ優先探索
    dfs(T);

    // 出力
    for (int i = 1; i <= N; i++) {
        if (i >= 2) cout << " ";
        cout << Answer[i];
    }
    cout << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

9.6 問題 B66：Typhoon (難易度：★5相当)

※この問題は例題と比べて相当難しいです。

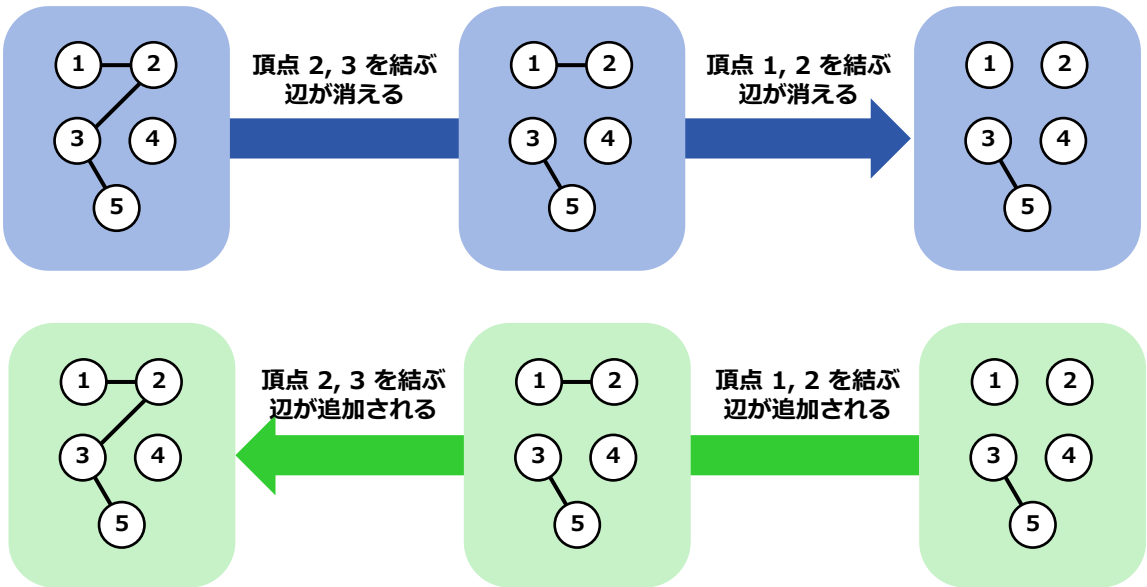
まず、駅を頂点とし、鉄道路線を辺としたグラフを考えると、クエリ 1・2 はそれぞれ以下のような処理に対応します。

クエリ1： x 本目の路線が運休になる	x 番目の辺が消える
クエリ2： 駅 s から駅 t へ移動できるかを答える	頂点 s と t は同じ連結成分かを答える

これらの処理は Union-Find で行えるクエリ（本の 375 ページ）と非常によく似ていますが、残念ながら辺を消す操作だけはできません。一体どうすれば良いのでしょうか。

一般の場合を解く

解決策の一つとして、**クエリを逆順に処理する**という方法があります。もし逆順になれば、「辺が消える操作」は「辺を追加する操作」に変わるため、Union-Find で処理できる形になります。



このアイデアを実装すると、次ページの解答例のようになります。なお、最後のクエリの時点でも運休になっていない路線が存在する場合があるため、このような辺を事前に追加しておく必要があることに注意してください。

解答例（C++）

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

class UnionFind {
public:
    int par[100009];
    int siz[100009];

    // N 頂点の Union-Find を作成
    void init(int N) {
        for (int i = 1; i <= N; i++) par[i] = -1; // 最初は親が無い
        for (int i = 1; i <= N; i++) siz[i] = 1; // 最初はグループの頂点数が 1
    }

    // 頂点 x の根を返す関数
    int root(int x) {
        while (true) {
                if (par[x] == -1) break; // 1 個先（親）がなければ、ここが根
                x = par[x]; // 1 個先（親）に進む
        }
        return x;
    }

    // 要素 u と v を統合する関数
    void unite(int u, int v) {
        int RootU = root(u);
        int RootV = root(v);
        if (RootU == RootV) return; // u と v が同グループのときは処理を行わない
        if (siz[RootU] < siz[RootV]) {
                par[RootU] = RootV;
                siz[RootV] = siz[RootU] + siz[RootV];
        }
        else {
                par[RootV] = RootU;
                siz[RootU] = siz[RootU] + siz[RootV];
        }
    }

    // 要素 u と v が同一のグループかどうかを返す関数
    bool same(int u, int v) {
        if (root(u) == root(v)) return true;
        return false;
    }
};

// 入力で与えられる変数・答え
int N, M, A[100009], B[100009];
int Q, QueryType[100009], x[100009], u[100009], v[100009];
string Answer[100009];

// その他の変数
UnionFind UF;
bool cancelled[100009];
```

```cpp
int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) cin >> A[i] >> B[i];
    cin >> Q;
    for (int i = 1; i <= Q; i++) {
        cin >> QueryType[i];
        if (QueryType[i] == 1) cin >> x[i];
        if (QueryType[i] == 2) cin >> u[i] >> v[i];
    }

    // 最初に運休になっている路線を求める
    for (int i = 1; i <= M; i++) cancelled[i] = false;
    for (int i = 1; i <= Q; i++) {
        if (QueryType[i] == 1) cancelled[x[i]] = true;
    }

    // Union-Find の初期化（その日の最後の状態にする）
    UF.init(N);
    for (int i = 1; i <= M; i++) {
        if (cancelled[i] == false && UF.same(A[i], B[i]) == false) {
                UF.unite(A[i], B[i]);
        }
    }

    // クエリを逆から処理
    for (int i = Q; i >= 1; i--) {
        if (QueryType[i] == 1) {
                // 駅 A[x[i]] と駅 B[x[i]] を結ぶ路線が開通
                if (UF.same(A[x[i]], B[x[i]]) == false) UF.unite(A[x[i]], B[x[i]]);
        }
        if (QueryType[i] == 2) {
                if (UF.same(u[i], v[i]) == true) Answer[i] = "Yes";
                else Answer[i] = "No";
        }
    }

    // 出力
    for (int i = 1; i <= Q; i++) {
        if (QueryType[i] == 2) cout << Answer[i] << endl;
    }
    return 0;
}
```

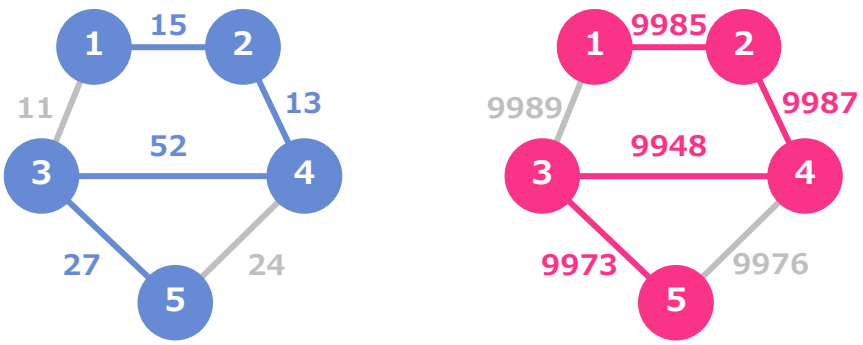
※Python のコードはサポートページをご覧ください

9.7 問題 B67：Max MST (難易度：★5相当)

グラフの最大全域木は、**大きい辺から順番に追加していく**という非常に単純な貪欲法で求めることができます。

なぜなら、「グラフ G の最大全域木」と「各辺の長さを $10000 - x$ に変えたグラフ G' の最小全域木」が一致するからです※3。

そして、最小全域木は本の 383 ページで説明したように、小さい辺から順番に追加していくという貪欲法（クラスカル法）で求められるからです。



解答例（C++）

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

// Union-Find クラスの実装は本の 9.6 節（379〜380 ページ）参照
int N, M;
int A[100009], B[100009], C[100009];
UnionFind UF;

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) cin >> A[i] >> B[i] >> C[i];
```

※3　このことは、「もしグラフ G で選んだ辺の重みの総和が w であり、グラフ G' でもまったく同じ辺を選ぶとき、グラフ G' での辺の重みの総和が $10000(N-1)-w$ になり、w の値が大きいほど小さくなること」から説明できます（N を頂点数とする）。

たとえば上図の例のように頂点が 5 個である場合、グラフ G' の重みの総和が $40000 - w$ となり、この値は w が最大のとき最小になります。

```
    // 辺を長さの大きい順にソートする
    vector<pair<int, int>> EdgeList;
    for (int i = 1; i <= M; i++) EdgeList.push_back(make_pair(C[i], i));
    sort(EdgeList.begin(), EdgeList.end());
    reverse(EdgeList.begin(), EdgeList.end()); // 問題 A67 と異なる唯一の部分

    // 最大全域木を求める
    int Answer = 0; UF.init(N);
    for (int i = 0; i < EdgeList.size(); i++) {
        int idx = EdgeList[i].second;
        if (UF.same(A[idx], B[idx]) == false) {
                UF.unite(A[idx], B[idx]);
                Answer += C[idx];
        }
    }
    cout << Answer << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

9.8 問題 B68：ALGO Express

(難易度：★7相当)

※この問題は例題と比べて相当難しいです。

この問題は、いきなり N が大きい場合や P_i が負の場合を考えると難しくなってしまうので、まずは以下のケースを考えましょう。

ALGO 鉄道には $N = 3$ 個の駅があり、各駅に対して以下のコストがかかる（特急駅に指定しない場合もコストがかかることに注意）。

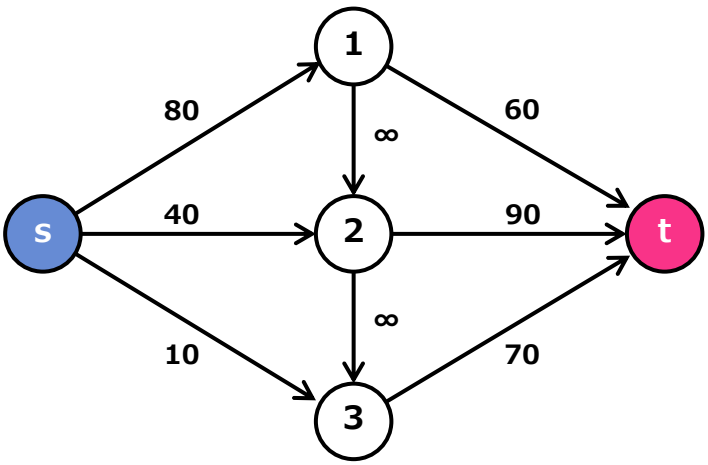
選択	駅 1 のコスト	駅 2 のコスト	駅 3 のコスト
特急駅に指定する場合	60 円	90 円	70 円
特急駅に指定しない場合	80 円	40 円	10 円

また、以下の $M = 2$ 個の条件がある。

- 駅 1 が特急駅ならば駅 2 も特急駅でなければならない
- 駅 2 が特急駅ならば駅 3 も特急駅でなければならない

すべての条件を満たすときの最小コストは何円か。

実は、この問題の答えは、以下のグラフの最大フロー、すなわち最小カットの重みの総和と一致することが知られています。



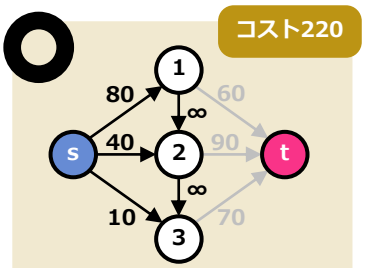
では、なぜそうなるのでしょうか。最小カット問題において、

- 駅 i を特急駅に指定しない場合、s と頂点 i を結ぶ辺を切る
- 駅 i を特急駅に指定する場合、頂点 i と t を結ぶ辺を切る

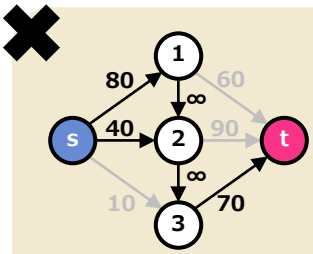
という辺の切り方をすることを考えます（切った辺の重みの総和がコストの合計となります）。

このとき、**$M = 2$ 個すべての条件を満たす場合は既に s から t へ到達不可能**になっており、追加で辺を消す必要がありません。しかし 1 つでも満たさない条件がある場合は、重み ∞ の辺を消す必要があり、最小カットが ∞ になってしまいます。

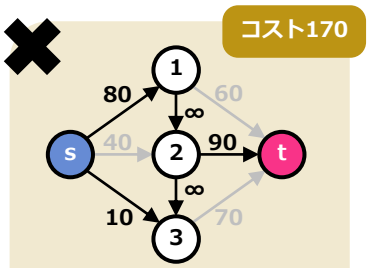
そのため、最小カットは「すべての条件を満たす場合の最小コスト」に対応します。以下に例を示します。



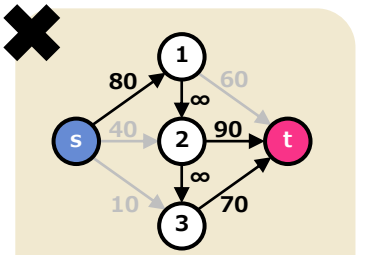
駅 1, 2, 3 をすべて選ぶ
→到達不可能



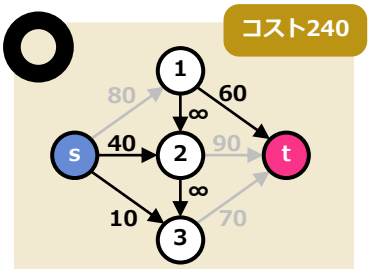
駅 1, 2 を選び
駅 3 を選ばない
→到達可能・・・



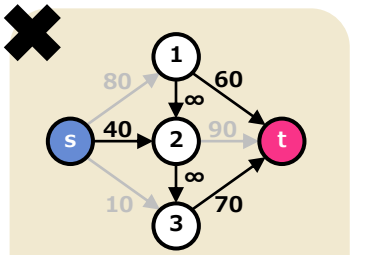
駅 1, 3 を選び
駅 2 を選ばない
→到達可能・・・



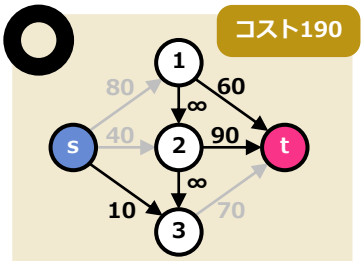
駅 1 を選び
駅 2, 3 を選ばない
→到達可能・・・



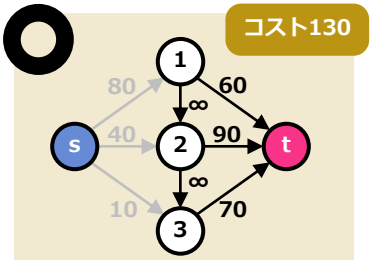
駅 2, 3 を選び
駅 1 を選ばない
→到達不可能



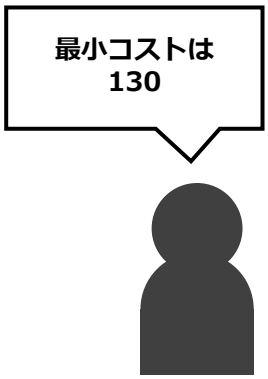
駅 2 を選び
駅 1, 3 を選ばない
→到達可能・・・



駅 3 を選び
駅 1, 2 を選ばない
→到達不可能



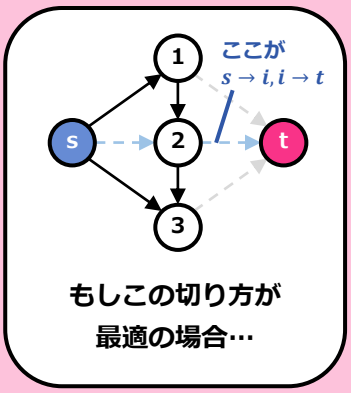
駅 1, 2, 3 を
すべて選ばない
→到達不可能

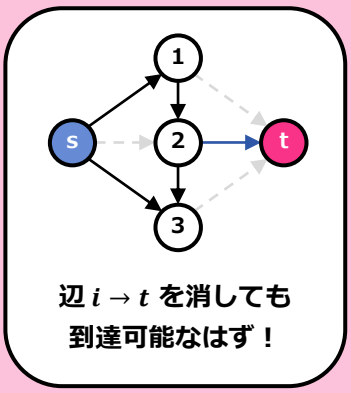


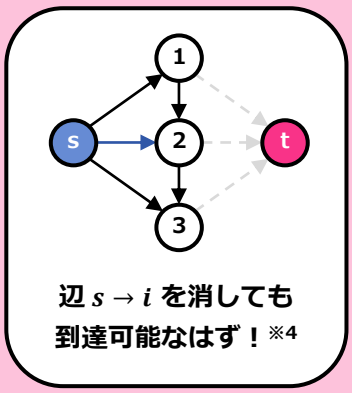
もしかしたら、連結のままである場合に、重さ ∞ の辺の代わりに他の辺を消す方が良いのではないか（つまり、ある頂点 i に対して辺 $s \to i$ と辺 $i \to t$ を両方消す）と思うかもしれません。

しかし、このような消し方は絶対に最適にはなりません。これは次のように証明することができます（非常に難しいので読み飛ばしてもかまいません）。

仮にこのような消し方が最適である場合、**辺 $s \to i$ だけを追加しても、辺 $i \to t$ だけを追加しても、始点 s から終点 t まで到達可能になるはず**である（たとえば辺 $s \to i$ を追加しても到達不可能である場合、辺 $s \to i$ を消さない方がが明らかに最適に近い）。







ところが、辺 $s \to i$ だけを追加しても、辺 $i \to t$ だけを追加しても、到達不可能なものが到達可能になるようなケースは絶対に存在しない。

なぜなら、辺 $s \to i$ だけを追加したときの s から t までの経路は**「$s \to i \to$ (適当なパスA) $\to t$」**という形で必ず表されるからである。

また、辺 $i \to t$ だけを追加したときの s から t までの経路は**「$s \to$ (適当なパスB) $\to i \to t$」**という形で必ず表されるからである。

そして、辺 $s \to i$ も辺 $i \to t$ も追加しない元の切り方の場合でも、「$s \to$ (適当なパスA) $\to i \to$ (適当なパスB) $\to t$」という経路を通ることで、s から t まで到達可能になってしまうからである。

これは「元の切り方で始点 s から終点 t へ到達不可能である」という仮定に矛盾する（背理法）。

実際の問題を解く

それではいよいよ本題に入ります。駅 i 特急駅に指定するときの利益 P_i が正・負どちらもあり得る場合は、どうやって解けば良いのでしょうか。たとえば以下のケースを考えましょう。

ALGO 鉄道には $N = 3$ 個の駅があり、各駅を特急駅に指定した場合、以下のような利益が得られる。

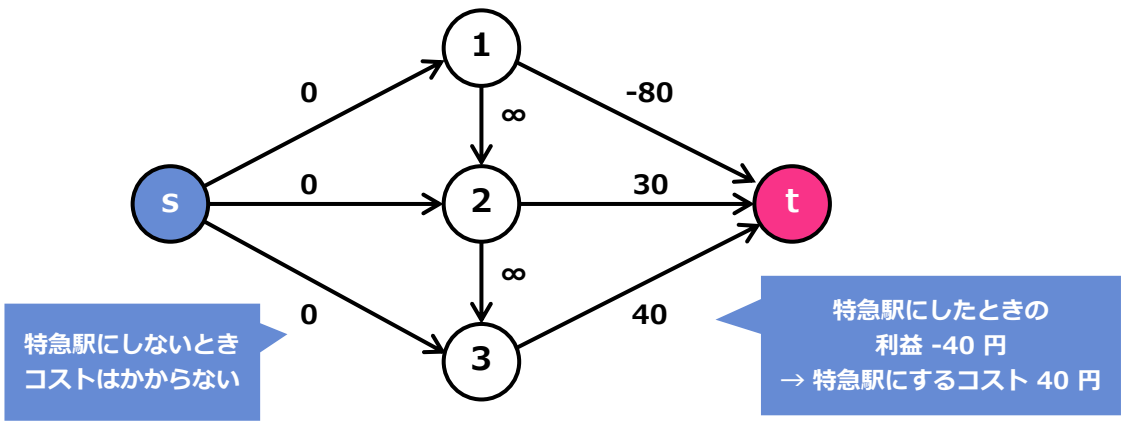
選択	駅 1 のコスト	駅 2 のコスト	駅 3 のコスト
特急駅に指定するときの利益	80円	-30円	-40円

また、以下の $M = 2$ 個の条件がある。

- 駅 1 が特急駅ならば駅 2 も特急駅でなければならない
- 駅 2 が特急駅ならば駅 3 も特急駅でなければならない

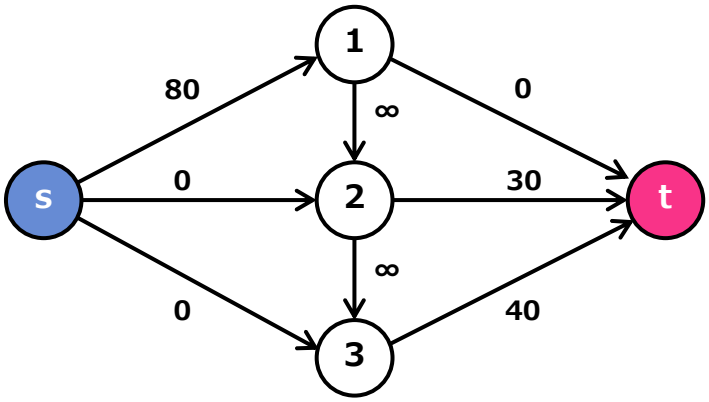
すべての条件を満たすときの利益の最大値は何円か。

一見すると、このケースでの答えは、以下のようなグラフの最小カットを計算することで求められると思うかもしれません。



しかし Ford-Fulkerson 法は、重みが負の辺があるときに正しく動作しません。そこで負の辺を消すために、**「特急駅にしたときのコストが $-x$ 円であること」を「特急駅にしないときのコストが x 円であること」に言い換える**ことを考えましょう。

たとえば先程の例の場合、グラフは以下のように修正されます。「特急駅にしたときのコストが -80 円」が「特急駅にしないときのコストが 80 円」に言い換えられています。



このとき、求める最大の利益は **80-(最小カット)** になります。ここで最小カットが足し算ではなく引き算になっている理由は、最小カットで求めているのはコストの最小値であって、利益の最大値ではないからです。

また 80 という値は、「元々特急駅にしたときコスト -80 円・しないとき 0 円だったのが、特急駅にしたときコスト 0 円・しないとき 80 円」になり、一律 80 円上がったことに対する補正から出ています。

一般のケースで解く

ここまでの内容は分かりましたでしょうか。最後に一般のケースの解き方を説明します。まずは以下のようなグラフを作ります。

辺の始点／終点	コスト
スタート s → 頂点 i	**$P_i < 0$ のとき 0、$P_i > 0$ のとき P_i**
頂点 i → ゴール t	**$P_i > 0$ のとき 0、$P_i < 0$ のとき $-P_i$**
頂点 A_j → 頂点 B_j	**∞**

そこで、$P_i > 0$ であるような P_i の合計を `Offset` とするとき、求める最大の利益は `Offset`-(グラフの最小カット) となります。

したがって、以下の解答例のように実装すると、正しい答えが得られます。なお、このプログラムでは頂点番号を 1 つの整数で表すため、スタート地点の番号を $N+1$ に設定し、ゴール地点の番号を $N+2$ に設定していることに注意してください。

解答例（C++）

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

// MaximumFlow クラスは本の 9.8 節（393〜394 ページ）参照
int N, P[159];
int M, A[159], B[159];
MaximumFlow Z;

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= N; i++) cin >> P[i];
    for (int i = 1; i <= M; i++) cin >> A[i] >> B[i];

    // グラフを作る（前半パート）
    int Offset = 0;
    Z.init(N);
    for (int i = 1; i <= N; i++) {
        // 効果が正の場合は、特急駅にしない場合（始点→i）のコストを P[i] に設定
        if (P[i] >= 0) {
                Z.add_edge(N + 1, i, P[i]);
                Offset += P[i];
        }
        // 効果が負の場合は、特急駅にする場合（i→終点）のコストを -P[i] に設定
        if (P[i] < 0) {
                Z.add_edge(i, N + 2, -P[i]);
        }
    }

    // グラフを作る（後半パート）
    for (int i = 1; i <= M; i++) {
        Z.add_edge(A[i], B[i], 1000000000);
    }

    // 答えを求める
    int Answer = Offset - Z.max_flow(N + 1, N + 2);
    cout << Answer << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください

9.9 問題 B69：Black Company 2 (難易度：★6相当)

まず、以下のような $N+26$ 頂点のグラフを考えます。

[頂点の情報]

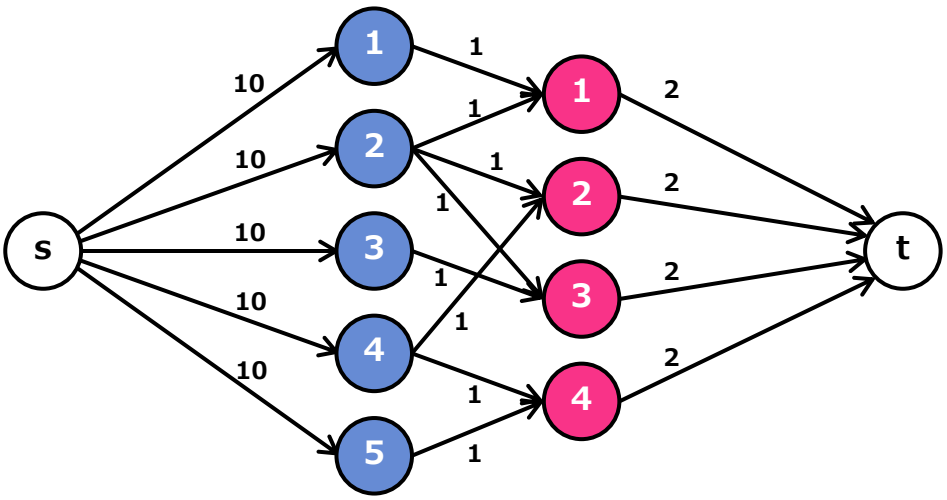
- 青色頂点は N 個あり、従業員に対応する
- 赤色頂点は 24 個あり、時間帯に対応する
- 他にも、スタート地点 s とゴール地点 t が存在する

[辺の情報]

以下の 3 種類の辺を追加する：

辺の始点／終点	容量	備考
スタート s → 青色頂点 i	10	10 は労働時間の上限に対応
青色頂点 i → 赤色頂点 j	1	$C_{i,j}=1$ のとき（働けるとき）のみ追加
赤色頂点 j → ゴール t	M	M は必要労働者数に対応

$N=5, M=2$ のときのグラフの例を以下に示します（ここでは図の大きさの都合上、1 日が 24 時間ではなく 4 時間の場合を例としています）。



このとき、最大フローの総流量が $24\times M$ であるときに限り答えは Yes、そうでなければ答えは No となります。

この理由は、以下の 2 つのポイントから説明できます。

[ポイント1]

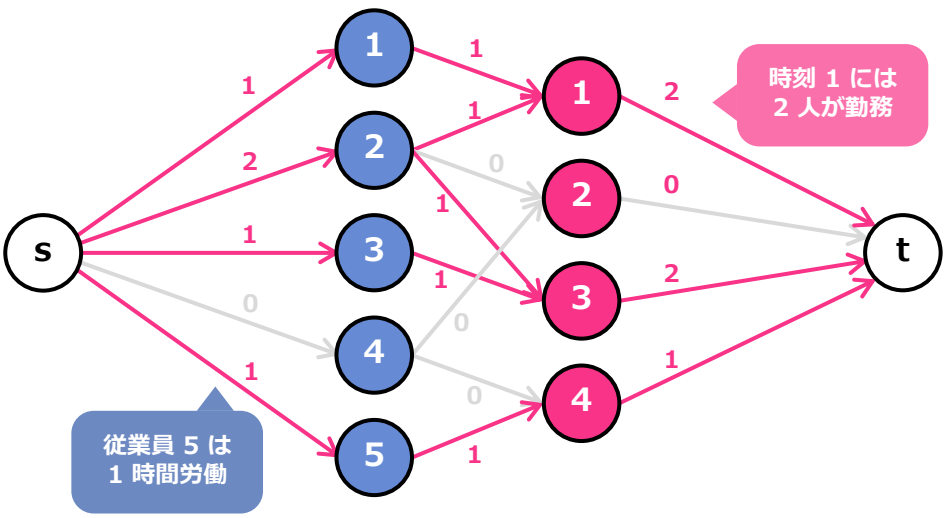
フローにおける各辺の流量は、以下のような値に対応する。

辺の始点／終点	値の意味
スタート s → 青色頂点 i	従業員 i の労働時間
青色頂点 i → 赤色頂点 j	流量が 1 のとき、従業員 i が時刻 j に働く 流量が 0 のとき、従業員 i が時刻 j に働かない
赤色頂点 j → ゴール t	時刻 j に勤務している社員数

[ポイント2]

すべての時刻について M 人が勤務している場合に限り、フローの総流量は $24 \times M$ になる※4。

ポイント 1 の例を下図に示します（赤い数字は流量）。この図のようにフローを流した場合、たとえば従業員 2 は（時刻 1・3 に）2 時間労働することに対応します。



実装について

ここまでのアイデアを実装すると次ページの解答例のようになります。`MaximumFlow` クラスでは頂点番号を整数にする必要があるため、青色頂点の番号を $1 \sim N$、赤色頂点の番号を $N+1 \sim N+24$、スタート地点の番号を $N+25$、ゴール地点の番号を $N+26$ に再設定していることに注意してください。

※4　ここで、$M+1$ 人以上を勤務させるのは明らかに無駄であることに注意してください

解答例（C++）

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

// MaximumFlow クラスは本の 9.8 節（393〜394 ページ）参照
int N, M;
char C[59][24];
MaximumFlow Z;

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= 23; j++) cin >> C[i][j];
    }

    // グラフを作る（前半パート）
    Z.init(N + 26);
    for (int i = 1; i <= N; i++) {
        Z.add_edge(N + 25, i, 10); // 従業員は 10 時間までしか働けない
    }
    for (int i = 0; i <= 23; i++) {
        Z.add_edge(N + i, N + 26, M); // シフトは M 人以上欲しい
    }

    // グラフを作る（後半パート）
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= 23; j++) {
                if (C[i][j] == '1') Z.add_edge(i, N + j, 1);
        }
    }

    // 答えを求める
    int Answer = Z.max_flow(N + 25, N + 26);
    if (Answer == 24 * M) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

※Python のコードはサポートページをご覧ください