競技プログラミングで使えるグラフ理論の基本

@okuraofvegetabl

目次

第 I 章

グラフの基本知識

初めまして、okuraofvegetable です。題名の通り、今回は「競技プログラミング」でおそらく使えるであろうグラフ理論についての入門記事となっています。ところで、「競技プログラミング」というのは、プログラミングの中でも、開発などをするのではなく、制限時間以内に与えられた課題に対するプログラムを書くもので、オンラインで"TopCoder"や"Codeforces"や"AtCoder"(問題が日本語で取り組みやすいと思います) というサイトなどでコンテストが開かれているので、気軽に登録、参加することができます。
文章中のソースコードではすべて C++ を用いています。また、ソース中では整数型でほとんど int 型を用いていますが、実際に問題を解く際は制約に注意して適宜 long long int 型などを用いてください。問題を解く際などにソースコードをコピペして使ってもらっても構いませんが、何があっても自己責任でお願いいたします。

1 グラフとは?

この記事を読んでくださっている方の中では知っていらっしゃる方も多いと思いますが、基本的な事から説明していきたいと思います。

ここでいうグラフとは、折れ線グラフや放物線のグラフとは違い、頂点 (node) と辺 (edge) からなる集合の組のことを指します。一般に、頂点の集合を V, 辺の集合を E とするグラフを $G = (V, E)$ と表します。また、点 u, 点 v を結ぶ辺を $e(u, v)$ と表します。頂点は点や小さな円、辺は線で表されることが多いです。ここで重要なことは、点や線がどのように描かれているか (大きく遠回りしているなど) は関係なく、どの頂点とどの頂点が結ばれているか、いないかだけだということです。こんなものを考えて、なににになるんだと思う方もいらっしゃるかもしれませんが、日常的に目にしたりしているものの中にも、グラフと捉えられるものはいくつかあります。例えば、電車の線路を辺、駅を頂点だと捉えることができます。

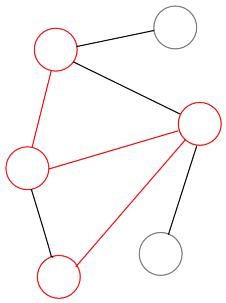


図 I.1　部分グラフ

2　無向グラフ、有向グラフ

グラフには、無向グラフと有向グラフとがあります。有向グラフは矢印のように描かれることが多いです。無向グラフでは、$e(u,v)$ が存在する時、u から v、v から u のどちらの方向にも移動することができます。
しかし、有向グラフでは、$e(u,v)$ が存在する時、u から v のみにしか移動できません。すなわち、無向グラフでは辺 (u,v) と辺 (v,u) は同一のものですが、有向グラフにおいては区別されるという事です。
無向グラフの時、$e(u,v)$ が存在すれば辺 e は頂点 u,v に接続すると言います。また、有向グラフにおいては u を辺 e の始点、v を辺 e の終点と呼びます。
無向グラフでは頂点 v に接続している辺の本数を v の次数といい, 有向グラフでは v を始点とする辺の本数を出次数、終点とする辺の本数を入次数と言います。

3　用語, 記号

$|V|$…頂点数

$|E|$…辺数

部分グラフ

G(V,E) の部分グラフ $P(V',E')$ とは、$V \supseteq V', E \supseteq E'$ であるようなグラフの事言います。

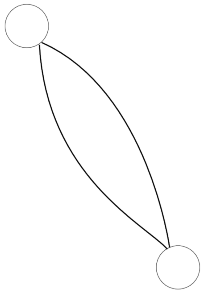


図 I.2　多重辺

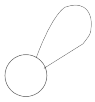


図 I.3　ループ

多重辺、ループ

多重辺とは、2 頂点 u,v 間に、複数の辺 (有向グラフであれば、始点、終点が同一な辺) があるとき、それらを多重辺と呼び、頂点 u を出て u に帰ってくるような辺をループと言います。

単純グラフ

多重辺、ループのないグラフのことです。

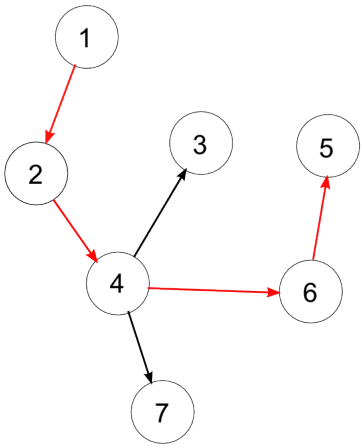


図 I.4　1 から 5 への path

路 (path)

路、道 (path) とは、もとのグラフの部分グラフ $P(V,E), V = \{v_0, v_1 \cdots, v_k\}$,$E = \{v_0v_1, v_1v_2 \cdots, v_{k-1}v_k\}(v_pv_q$ は $e(v_p, v_q)$ を表す) のことで、このとき、$v_0, v_1 \cdots v_k$ はすべて異なります。パスは、$P = v_0v_1 \cdots v_k$ のように単に頂点だけを並べて表すこともあります。

閉路 (cycle)

P$=v_0 \cdots v_{k-1}$ が path で、k $\geqq 3$ のときグラフ $C := P + v_{k-1}v_0$ を閉路と呼びます。path と同様に巡回的な頂点の列で表すこもあります。

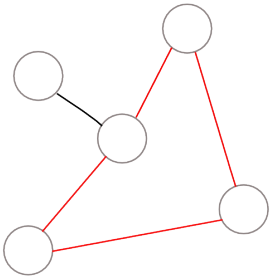


図 I.5　閉路

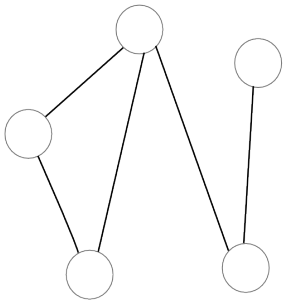


図 I.6　連結なグラフの一例

連結・強連結

任意のどの 2 点 u,v 間にも path が存在するとき、無向グラフでは連結、有向グラフでは強連結といい、連結 (強連結) な頂点集合に分解した際の各集合を連結成分 (強連結成分) と言います。今後出てくるグラフは特に断りのない限り、単純連結グラフとします。(有向、無向については適宜判断お願いします。)

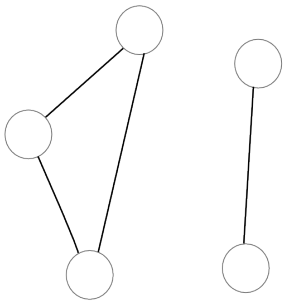


図 I.7　連結でないグラフの一例

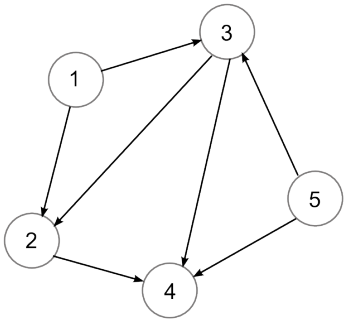


図 I.8　DAG の一例

DAG

閉路のない有向グラフのことを DAG(Directed Acyclic Graph) と言います。

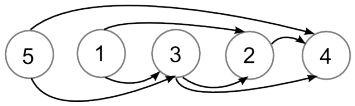


図 I.9　前図の DAG のトポロジカル順序の一例

トポロジカル順序

DAG の各頂点を横に一直線上に並べることを考えます。この時、すべての辺が左から右の頂点へ向くような並び方が必ず存在し、これをトポロジカル順序といい、この順序に並び替える、もしくはこの順序を求めることをトポロジカルソートといいます。トポロジカル順序は一意に定まらないこともあります。

4　グラフの表現

プログラムを書くにあたって、グラフを何らかの形で表現する必要があります。なお、競技プログラミングにおいてはほとんどの場合、必要になる node 数は 10^6 程度です

1 隣接行列

$|V| \times |V|$ の二次元配列でグラフを表現します。graph$[i]\,[j]$ には $e(i,j)$ の有無、コストなどを入れることで管理します。構造体の配列にすることでたくさんの情報を保持でき、O(1) で辺の有無などを判定できますが、無駄な部分が多く、メモリの消費が激しく、$|V|$ が大きくなると配列を取ることができなくなってしまいます。

2 隣接リスト

各頂点ごとに、隣接している頂点を C++ であれば vector や配列などで管理します。こちらも同じく、各々に構造体をもたせることで様々な情報を持っておくことができます。隣接行列と違い、こちらなら、メモリが $O(|V|+|E|)$ しか必要なく、ある程度 $|V|$ が大きくなっても大丈夫です。

PKU などの Online Judge では vector を使うと TLE になる可能性が高いので配列を使うことをおすすめしますが、ここではすべて vector を用いた実装になっています。グラフの扱いに慣れるために問題を一つ解いてみましょう。

問題

頂点が N 個, 辺が M 個の有向グラフが与えられ、その後 2 つの頂点の組 (u, v) が Q 個与えられるので u から v への辺がある時"YES", ない時"NO"と出力せよ。
(ただし各頂点には $0 \sim N-1$ の番号がつけられており、1 行目に N,M,Q が空白区切りで与えられ、続く M 行に $(a_0, b_0) \cdots (a_{m-1}, b_{m-1})$ が与えられる。各組は a_i から b_i への辺があることを示す。つづく Q 行に $(u_0, v_0) \cdots (u_{q-1}, v_{q-1})$ が与えられる。)

制約: $0 \leq N \leq 1000$, $0 \leq M \leq N(N-1)/2$, $0 \leq Q \leq 10000$
Time Limit 1sec , Memory Limit 64MB

$N \leq 1000$ と小さいので、これは隣接行列、隣接リストの両方を用いて解くことができます。辺があるかどうかの判定なので隣接行列を用いるのが普通ですが、隣接リストに慣れるために隣接リストを用いた解法も載せておきます。最悪 $O(NQ)$ なので間に合います。

・隣接行列

```
#include <cstdio>
#include <cstring>
bool graph[1050][1050];
int main()
{
        int N,M,Q;
        memset(graph,false,sizeof(graph));
        scanf("%d %d %d",&N,&M,&Q);
        for(int i=0;i<M;i++)
        {
```

```
                int a,b;
                scanf("%d %d",&a,&b);
                graph[a][b]=true;
        }
        for(int i=0;i<Q;i++)
        {
                int u,v;
                scanf("%d %d",&u,&v);
                if(graph[u][v])printf("YES\n");
                else printf("NO\n");
        }
        return 0;
}
```

・隣接リスト

```
#include <cstdio>
#include <vector>
std::vector<int> graph[1050];
int main()
{
        int N,M,Q;
        scanf("%d %d %d",&N,&M,&Q);
        for(int i=0;i<M;i++)
        {
                int a,b;
                scanf("%d %d",&a,&b);
                graph[a].push_back(b);
        }
        for(int i=0;i<Q;i++)
        {
                int u,v;
                scanf("%d %d",&u,&v);
                for(int j=0;j<graph[u].size();j++)
                {
                        if(graph[u][j]==v)
                        {
```

```
                                printf("YES\n");
                                goto end;
                        }
                }
                printf("NO\n");
                end:;
        }
        return 0;
}
```

これでグラフを扱うことができるようになりました。
これから、グラフに関するもっと具体的な問題について考えていきましょう。

第 II 章

最短路問題

最短路問題というのは、「辺にコストのあるグラフにおいて、u から v への path 上の辺のコストの合計の最小化」です。最短路問題を解くためのアルゴリズムにはいろいろありますが、必要になる情報、計算量などで使い分ける必要があります。

1 Bellman-Ford(ベルマンフォード) 法

このアルゴリズムでは、1 つの頂点から他のすべての頂点への最短距離が求まります。各頂点 i の始点 s からの最短距離を $d[i]$ とすると

$$d[i] = min\{d[j] + e(j,i) \text{ のコスト } |e(j,i) \in E\}$$

が成り立ちます。なので $d[s] = 0, d[i] = INF$(十分大きい数) としてすべての辺を見て、上の式を適用して最小値を更新していきます。コストの和が負の閉路がない時、始点 s から各頂点までの最短路が同じ頂点を通ることがないので、このループはたかだか $|V| - 1$ 回しか起こりません。逆に、$|V|$ 回目のループでも更新が起こるとき、そのグラフはコストの和が負の閉路を含むことがわかります。これらより、このアルゴリズムの計算量は $O(|V||E|)$ であることがわかります。
以下のコードでは、コストの和が負の閉路がある時、"Negative loop exists."と出力し、それ以外の時に始点 s から各頂点までの最短距離 (辿りつけない場合は INF) を出力します。辺の情報を辺が出ていくノード、行き先のノード、コストの構造体で管理しています。

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;
#define INF 2000000000
#define MAX_V 10000
```

```
int V;//number of vertices
struct edge{int from,to,cost;};
vector<edge> edges;
int d[MAX_V];
int main()
{
        fill(d,d+V,INF);
        int s;
        scanf("%d",&s);
        d[s]=0;
        for(int i=0;i<V;i++)
        {
                for(int j=0;j<edges.size();j++)
                {
                        edge e=edges[j];
                        if(d[e.to]>d[e.from]+e.cost)
                        {
                                d[e.to]=d[e.from]+e.cost;
                                if(i==V-1)
                                {
                                        printf("Negative loop exists.\n");
                                        return 0;
                                }
                        }
                }
        }
        for(int i=0;i<V;i++)printf("%d\n",d[i]);
        return 0;
}
```

2 Dijkstra(ダイクストラ) 法

ベルマンフォード法と同様に 1 つの始点から他の各頂点までの最短距離を求めるためのアルゴリズムです。このアルゴリズムは辺のコストが、非負の時に限ります。非負の時に限ります。大事なことなので 2 度言いました。Dijkstra 法を使える問題が多いので、慣れてくるとたまにコストが負の可能性がある問題で爆死したりするので気をつけましょう。

まず先ほどと同様に $d[s]=0, d[i]=INF$ とします。そして、

1. まだ使われていない頂点の中で $d[i]$ が最小の頂点を探す
2. その頂点に隣接する頂点をさきほどの式を適用して更新する。
   ここでつかった頂点はもう使わない。

を繰り返します。
このアルゴリズムで最短距離が正しくもとまっていることを証明してみましょう。頂点集合 V が既に使われた (最短距離が確定した) 頂点集合 S とそれ以外の頂点集合 P に分かれた状態を考えます。この状態から、先ほどの更新を行なって、その後もこの状態が保たれていることを示し、帰納的に証明します。まだ最短距離が確定していない頂点集合 P の中から

$$d[i] = min\{d[j] \mid j \in P\}$$

を満たす頂点 i が選ばれ, 最短距離が確定した頂点集合に入れられるので、この $d[i]$ が始点 s からの本当の最短距離になっていることを示せばよいことがわかります。それを示すためには始点 s から i への任意の path のコストが $d[i]$ 以上になることを示せば良いです。path を s から巡った時、最初に訪れる P に含まれる頂点を k とするとすべての辺のコストは非負なので

$$d[k] \geq d[i]$$

であり、path のコストは $d[i]$ より短くなり得ないです。ゆえに、$d[i]$ は本当の最短距離です。これで、ダイクストラ法が正しいことを証明できました。
次にこのアルゴリズムの計算量について考えます。まず更新は $|V|$ 回行われます。次に使う頂点を探すのに、すべての頂点を見て最小値を探すと、$O(|V|^2)$ となります。ここで、優先度付きキュー (priority queue) を用いて次の頂点を $O(log|V|)$ で求めることで、計算量は $O(|E|log|V|)$ になります。これからわかるように、非常に密な (頂点に対して辺が多い) グラフに対しては前者、その他の場合では後者のほうが早く動きます。以下に実装例を示しておきます。入力で始点 s を受け取って関数 dijkstra の引数として与えて、最後に各頂点の s からの最短距離 (辿りつけない場合は INF) を出力しています。
graph には隣接リストの形でグラフの情報が入っているものとします。
なお、辺の情報は、行き先ノードの index, コストの構造体で管理しています。

$O(|V|^2)$

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <vector>
#define INF 2000000000
```

```cpp
#define MAX_V 10000
using namespace std;
int V;//number of vertices
int d[MAX_V];
bool used[MAX_V];
struct edge{int to,cost;};
vector<edge> graph[MAX_V];
void dijkstra(int s)
{
        fill(d,d+V,INF);
        memset(used,false,sizeof(used));
        d[s]=0;
        for(int i=0;i<V;i++)
        {
                int id;
                for(int j=0;j<V;j++)if(!used[j]&&d[id]>d[j])id=j;
                used[id]=true;
                for(int j=0;j<graph[id].size();j++)
                {
                        edge e=graph[id][j];
                        d[e.to]=min(d[e.to],d[id]+e.cost);
                }
        }
        return;
}
int main()
{
        int s;
        scanf("%d",&s);//0-indexed
        dijkstra(s);
        for(int i=0;i<V;i++)printf("%d\n",d[i]);
        return 0;
}
```

$O(|E|log|V|)$

```
#include <cstdio>
#include <cstring>
#include <functional>
#include <queue>
#include <utility>
#include <vector>
using namespace std;
typedef pair<int,int> P;
#define mp make_pair
#define MAX_V 10000
#define INF 2000000000
int V;
int d[MAX_V];
struct edge{int to,cost;};
vector<edge> graph[MAX_V];
void dijkstra(int s)
{
        fill(d,d+V,INF);
        d[s]=0;
        priority_queue<P,vector<P>,greater<P> > q;
        q.push(mp(0,s));//(distance,index)
        while(!q.empty())
        {
                P a=q.top();
                q.pop();
                if(d[a.second]<a.first)continue;
                for(int i=0;i<graph[a.second].size();i++)
                {
                        edge e=graph[a.second][i];
                        if(d[e.to]>d[a.second]+e.cost)
                        {
                                d[e.to]=d[a.second]+e.cost;
                                q.push(mp(d[e.to],e.to));
                        }
                }
        }
        return;
```

```
}
int main()
{
        int s;
        scanf("%d",&s);
        dijkstra(s);
        for(int i=0;i<V;i++)printf("%d\n",d[i]);
        return 0;
}
```

3 Warshall-Floyd(ワーシャルフロイド) 法

このアルゴリズムでは DP(動的計画法) を用いて全点対の最短距離を求めます。まず

$$dp\,[k+1]\,[i]\,[j] := \text{頂点 } 0\sim k, i, j \text{ のみを通る } i \text{ から } j \text{ への } path \text{ のコストの最小値}$$

とします。ただし、$k+1 = 0$ のときは i、j のみを使う path のコスト、すなわち $e(i,j)$ のコスト (存在しなければ INF) とします。$0\sim k$ の頂点と i,j を通るコスト最小の path は頂点 k を通るか、通らないかの 2 通りが考えられます。後者の場合は明らかに $dp\,[k+1]\,[i]\,[j] = dp\,[k]\,[i]\,[j]$ が成り立ちます。前者の場合、path を $i\to k, k\to j$ に分解すると $dp\,[k+1]\,[i]\,[j] = dp\,[k]\,[i]\,[k] + dp\,[k]\,[k]\,[j]$ となることが容易にわかります。ゆえに

$$dp\,[k+1]\,[i]\,[j] = min(dp\,[k]\,[i]\,[j]\,, dp\,[k]\,[i]\,[k] + dp\,[k]\,[k]\,[j])$$

であることがわかります。この式より、$dp\,[k+1]\,[i]\,[j]\,(i,j$ は任意) を求めたい時、$dp\,[k]\,[i]\,[j]$ が求まっていればよいことがわかります。また、最終的に必要なのは $dp\,[V]\,[i]\,[j]$ のみだから、同じ配列を使いまわせばよいことがわかります。このアルゴリズムの計算量はソースコードを見れば明らかなように $O(|V|^3)$ です。ですから、Time Limit が 1 sec の問題なら、$|V|$ は 500~600 程度が限界です。また、式を見て明らかなように、このアルゴリズムではグラフを隣接行列で表すのが適しています。(任意の 2 点の間の辺の情報が $O(1)$ でわかるため)

```
#include <algorithm>
using namespace std;
#define MAX_V 500
int V;//number of vertices
int dp[MAX_V][MAX_V];
//if e(u,v) exists,d[u][v]=cost of e(u,v)
```

```
//otherwise,d[u][v]=INF(but if i=j,dp[i][i]=0)
void warshallfloyd()
{
        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]);
        return;
}
int main()
{
        warshallfloyd();
        return 0;
}
```

これらのアルゴリズムでは、配列を用意して更新が起こった時にその頂点 (to) の前 (from) をメモしておくことで、最短路上の合計コストだけでなく、最短路を復元することが可能です。

4 線形計画問題

最短路問題ではグラフにおける 2 頂点間の最短距離を求めているわけですが、この 2 点間の距離の最小値はほかの捉え方ができないでしょうか。

任意の 2 点 i、j で $i \to j$ への (有向) 辺がある時、$d[i] + e(i,j)$ のコスト$\geq d[j]$ (d は始点 s からの各頂点の最短距離) が成り立ちます。これらの制約をすべて満たす d における $d[i]$ の最大値が s と頂点 i 最短距離です。このような問題は LP(線形計画問題) と呼ばれる問題で、様々なアルゴリズムがありますが非常に複雑なものばかりです。

ここまでで、グラフにおける最短路 (距離) 問題が線形計画問題に帰着できることがわかりました。ここで、発想を転換してみましょう。最短路問題が線形計画問題に帰着できるという事は当然線形計画問題の一部が最短路問題に帰着できる事になります。

すなわち、**制約の式の形が似た線形計画問題は最短路問題として解ける**!という事です。

その例がこちらです PKU 3169 Lay out

まず、この問題を定式化してみましょう。

各牛の位置を $d[i]$ とすると、各 i において $d[i] \leq d[i+1]$、各 (AL,BL,DL) において、$d[AL] + DL \geq d[BL]$, 各 (AD,BD,DD) において $d[AD] + DD \leq d[BD]$ を満たすような d の $d[N] - d[1]$ の最大値がこの問題の答えとなります。これを最短路問題に帰着すると、

$i+1$ から i へコスト 0 の、AL から BL へコスト DL の、BD から AD へコスト $-DD$

の辺を張ったグラフにおける、頂点 1 と頂点 N の最短距離が答えであるとわかります。なので、コストが負の辺があることに注意してベルマンフォード法を用いれば、この問題を解くことができます。

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;
#define INF 2000000000
struct edge
{
        int from,to,cost;
        edge(int f,int t,int c):from(f),to(t),cost(c){}
};
vector<edge> edges;
int d[1050];
int N,ML,MD;
int main()
{
        scanf("%d %d %d",&N,&ML,&MD);
        for(int i=1;i<N;i++)
        {
                edge in(i,i-1,0);
                edges.push_back(in);
        }
        for(int i=0;i<ML;i++)
        {
                int AL,BL,DL;
                scanf("%d %d %d",&AL,&BL,&DL);
                AL--;BL--;//1...N -> 0...N-1
                edge in(AL,BL,DL);
                edges.push_back(in);
        }
        for(int i=0;i<MD;i++)
        {
                int AD,BD,DD;
```

```
                scanf("%d %d %d",&AD,&BD,&DD);
                AD--;BD--;
                edge in(BD,AD,-DD);
                edges.push_back(in);
        }
        fill(d,d+N,INF);
        d[0]=0;
        for(int i=0;i<N;i++)
        {
                for(int j=0;j<edges.size();j++)
                {
                        edge e=edges[j];
                        if(d[e.to]>d[e.from]+e.cost)
                        {
                                d[e.to]=d[e.from]+e.cost;
                                if(i==N-1)
                                {
                                        //negative loop exists
                                        printf("-1\n");
                                        return 0;
                                }
                        }
                }
        }
        if(d[N-1]==INF)
        {
                //can't arrive at N-1
                printf("-2\n");
                return 0;
        }
        printf("%d\n",d[N-1]);
        return 0;
}
```

このように、最短路問題に帰着することで解ける線形計画問題のことはこの問題が元になって、競技プログラミング界隈で"牛ゲー"と呼ばれています。

第 III 章

橋, 関節点

1　橋, 関節点とは

　橋はグラフから取り除いた時に連結成分が増える辺で、関節点はそのような頂点です。(頂点を取り除く時、その頂点を始点とする辺も一緒に取り除きます)

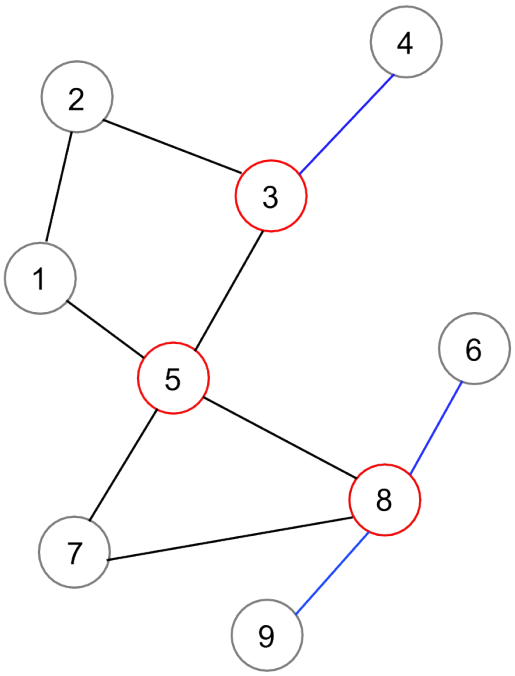


図 III.1　橋、関節点

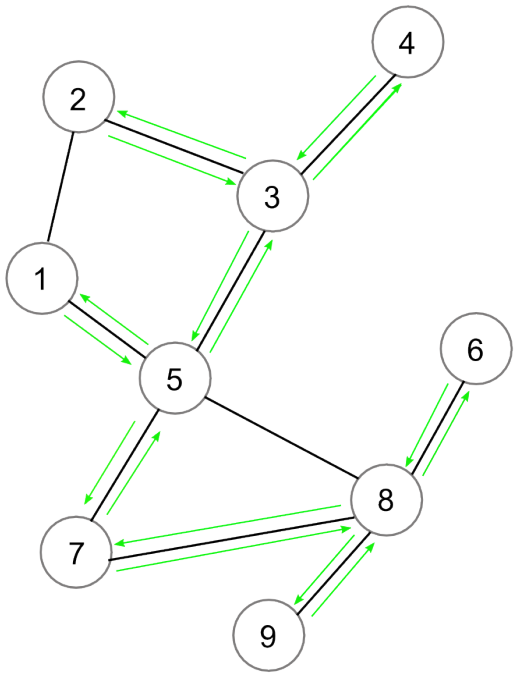


図 III.2　DFS

2　二重連結性

任意の 1 つの頂点を取り除いても連結なグラフ、すなわち関節点を持たないグラフを二重連結なグラフと言います。また、2 重連結な最も大きい部分グラフを二重連結成分と言います。

3　橋・関節点の求め方

深さ優先探索

深さ優先探索 (DFS/Depth-First Search) はスタックを用いた探索方法で、グラフのノードを下の図の様に巡ります。

関数の再帰はスタックを用いて実現されているため、DFS は再帰関数で簡単に実装できます。自前のスタックを用いてもできますが少し実装が重くなります。再帰が深くなる時はスタックオーバーフローする可能性があるのでそのようにしなければいけません。

DFS をした時に通った辺と頂点は、DFS を始めた頂点を根とする木になっています。こ

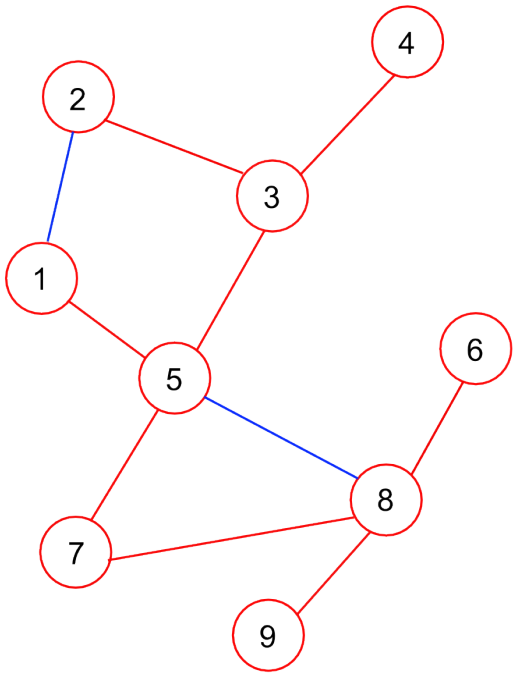


図 III.3　DFS 木 (赤) と後退辺 (青)

れを DFS 木 (DFS Tree) といいます。

グラフが連結な時、DFS をするとすべての頂点を通ります。ですがすべての辺を通る保証はありません。通らなかった辺のことを後退辺 (backward edge) と言います。また、後退辺が結ぶ 2 つの頂点は必ず先祖、子孫の関係にあることは少し考えるとすぐ解ると思います。

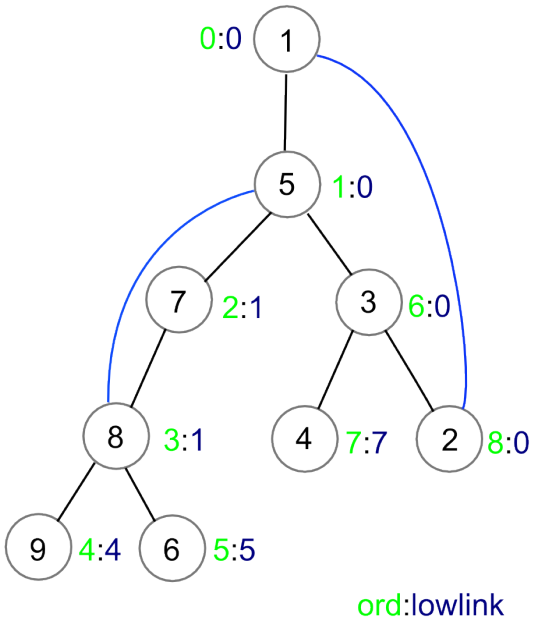
lowlink

橋・関節点を効率良く求める際に重要なのが lowlink です。

連結単純無向グラフで、DFS を行い、頂点を訪れた順に順序付け (ord) しておきます。各頂点 v の lowlink を次のように定めます。

> 頂点 v から DFS 木の葉方向の辺を 0 回以上、後退辺を 0 回以上 1 回以下通って行ける頂点の ord の最小値

これは一体何を表しているのか、言葉だけではわからないと思うのでしたの図を見てください。これでなんとなく lowlink が表しているものがわかったのではないでしょうか。下



ord,lowlink

のソースコードでは ord,lowlink を DFS を行なって求めています。

```
#include <vector>
#include <algorithm>
using namespace std;
#define MAX_V 10000
vector<int> graph[MAX_V];
bool used_v[MAX_V],used_e[MAX_V][MAX_V];
int ord[MAX_V],lowlink[MAX_V];
int k=0;
void dfs(int v)
{
        used_v[v]=true;
        ord[v]=lowlink[v]=k++;
        for(int i=0;i<graph[v].size();i++)
        {
```

```
                if(!used_v[graph[v][i]])
                {
                        used_e[v][graph[v][i]]=true;
                        dfs(graph[v][i]);
                        lowlink[v]=min(lowlink[v],lowlink[graph[v][i]]);
                }
                else if(!used_e[graph[v][i]][v])
                //then,e(v,graph[v][i]) is backward edge
                {
                        lowlink[v]=min(lowlink[v],ord[graph[v][i]]);
                }
        }
        return;
}
```

この lowlink をつかうことで橋・関節点を効率良く求めることができます。

ord,lowlonk と橋・関節点の関係

まず橋について考えてみましょう。

ord は、DFS で訪れた順につけているので、DFS 木の親の ord は必ず子の ord よりも小さくなっています。後退辺を無視したグラフを考えると、$e(u,v)$(u が v の親とする) を取り除いた時、グラフは DFS を始めた頂点を根とする木と v を根とする木の、2 つにわかれます。この 2 つの木が、後退辺によってつながれていれば $e(u,v)$ は橋ではなく、そうでなければ $e(u,v)$ は橋という事になります。さきほど書いたように、後退辺は必ず先祖と子孫の関係にある頂点同士を結んでおり、lowlink は v を含む木の頂点からの後退辺で行ける DFS 木の最も根に近い頂点の ord を表しているので、

$$\text{辺 } e(u,v) \text{ が橋} \Leftrightarrow ord\,[u] < lowlink\,[v]$$

であることがわかります。なので、ord,lowlink が求まっていればある辺が橋かどうか $O(1)$ で判断できることがわかりました。これで、橋は ord,lowlink を求める DFS で $O(|V|)$, すべての辺が橋であるかのチェックで $O(|E|)$ の合計 $O(|V|+|E|)$ で求められることがわかりました。

次に関節点について考えてみましょう。

結論から言うと、橋の時と同様に、ord、lowlink から関節点かどうかの判別が行えます。まず、DFS 木の根 (DFS を始めた頂点) の判断は簡単で、子が 1 つなら関節点ではなく、そうでなければ関節点です。

根以外の頂点ではどうでしょうか。ある頂点 v 関節点でない時、v の子孫のどれかが後退辺で v の先祖と繋がっています。逆も正しいことは簡単に証明できます。なので、v のすべての子孫 u について $lowlink\,[u] > ord\,[v]$ かどうかをチェックすることで頂点 v が関節点でないことがわかります。しかし、ここで lowlink の定め方を思い出すと、v のすべての子孫ではなく、v の子のみを調べれば十分です。なので

頂点 v が関節点⇔$ord\,[u] \leq lowlink\,[v]$ となる v の子 u が存在する

であることがわかります。

これも橋と同じく計算量は、$O(|V| + |E|)$ となります。

第 IV 章

あとがき

最後まで読んでくださってありがとうございます。部誌を書き始めたのが遅かったため、非常に内容が薄くなってしまって申し訳ありません。最初は高度な内容の記事を書こうと思ったのですが、下調べをする時間がなかったので、初めての方向けの入門記事にしました。とはいえ、Online Judge などでたくさん問題を解いていて慣れている人でも橋、関節点の問題は最短路問題に比べて少ないため、忘れていた、という方もいらしたのではないでしょうか。この記事を読んで競技プログラミングに興味を持ってもらえたりグラフ理論についてもっと知りたいと思っていただけたのなら幸いです。もし計算量等、大ウソを書いている場所があれば@okuraofvegetabl まで連絡していただけると嬉しいです。

参考文献

[1] R. ディーステル『グラフ理論』(丸善出版,2012)
[2] 秋葉拓哉, 岩田陽一, 北川宣稔『プログラミングコンテストチャレンジブック 第 2 版』(マイナビ,2010)