JOI春季セミナー 中級コース Day2－1

13章 グラフ(1) : グラフ探索 2024.03.23

by **電気通信大学4年 後藤 照佳**

13章 グラフ(1) : グラフ探索

グラフと知り合いになろう

- 担当者：小林 遼平
- 目標：様々な問題をグラフの探索に読み替えて解いてみよう！

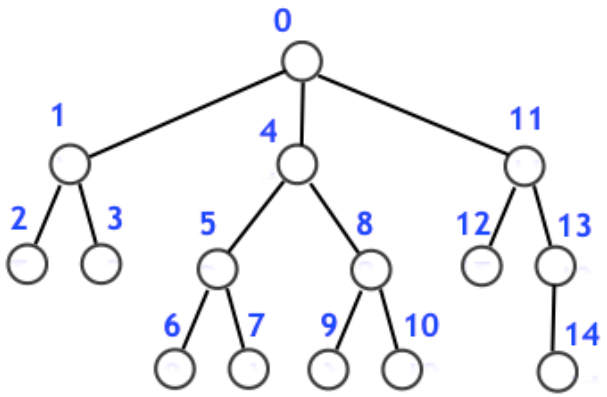
このコマでやること

以下の内容について説明します．最後は学んだことを生かして問題を解きます．(テキストp181-191)

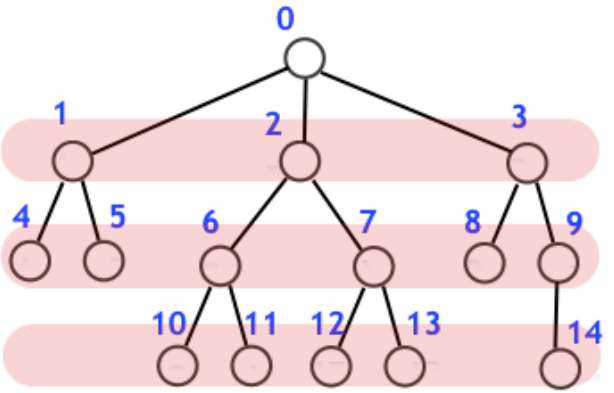
1. **深さ優先探索** と **幅優先探索** の感覚
2. 深さ優先探索の実装
3. 幅優先探索の実装
4. DFSやBFSを用いた応用例

1. 深さ優先探索と幅優先探索の感覚

イメージ画像．わかりやすいね！
さすがけんちょん！



深さ優先探索

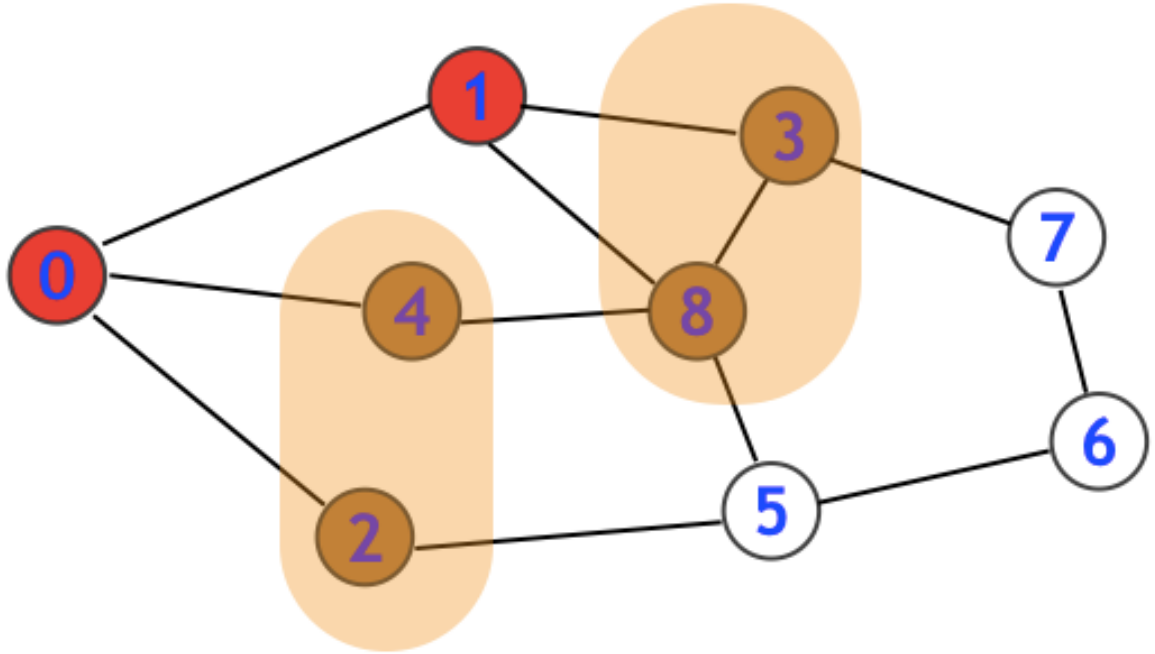


幅優先探索

画像引用元：https://qiita.com/drken/items/4a7869c5e304883f539b

1. 深さ優先探索と幅優先探索の感覚

こんな感じ.
どっちを優先するかで深さか幅かが決まるね



画像引用元：https://qiita.com/drken/items/4a7869c5e304883f539b

2&3. 深さ優先探索と幅優先探索の実装(雰囲気)

大体これで表せる．T(v)はvに隣接してる頂点の集合だと思って欲しい．

1. seen 全体を false に初期化して、todo を空にする
2. seen[s] = true として、todo に s を追加する
3. todo が空になるまで以下を繰り返す:
    i. todo から一つ頂点を取り出して v とする
    ii. T(v) の各要素 w に対して、
        a. seen[w] = true であったならば、何もしない
        b. そうでなかったら、seen[w] = true として、todo に w を追加する

取り出し方によって深さや幅になる．どういうこと？

ここからは少し別資料で動きを説明

2. 深さ優先探索の実装

- スタック
- 再帰

3. 幅優先探索の実装

- キュー

4. DFSやBFSを用いた応用例

1. s-tパス判定(連結判定)
2. 二部グラフ判定

グラフではない問題 を **グラフの探索** のように見立てて解くものもあるよ！

6. 講義内容に関連した問題を解いてみよう！

練習問題一覧です.

1. ATC001 A - 深さ優先探索
2. ATC002 A - 幅優先探索
3. ABC029 C - Brute-force Attack
4. ABC088 D - Grid Repainting
5. ABC070 D - Transit Tree Path
6. ABC138 D - Ki
7. ABC126 D - Even Relation
8. JOI第9回予選 C - パーティー
9. JOI第9回予選 D - カード並べ
10. JOI第8回予選 D - 薄氷渡り
11. JOI第10回予選 E - チーズ (Cheese)

参考にしたもの

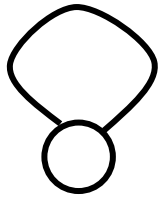
- 昨年の熊田さんのスライド(そのまま使いました)
- 大槻さん(テキスト筆者)/drkenさんのQiita各種

グラフ探索

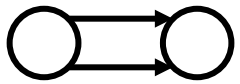
熊田順一

おさらい(グラフ用語)

- 連結...どの 2 頂点も繋がってる
- sからtへのパス...頂点sから辺を伝ってtまで行く経路
- サイクル...辺を伝って元の頂点に戻ってくる経路
- 自己ループ...s->s のように、同じ頂点を結んでいる辺
- 多重辺...ある頂点と別の頂点を結んでいる複数の辺
- 単純...自己ループや多重辺のないグラフ



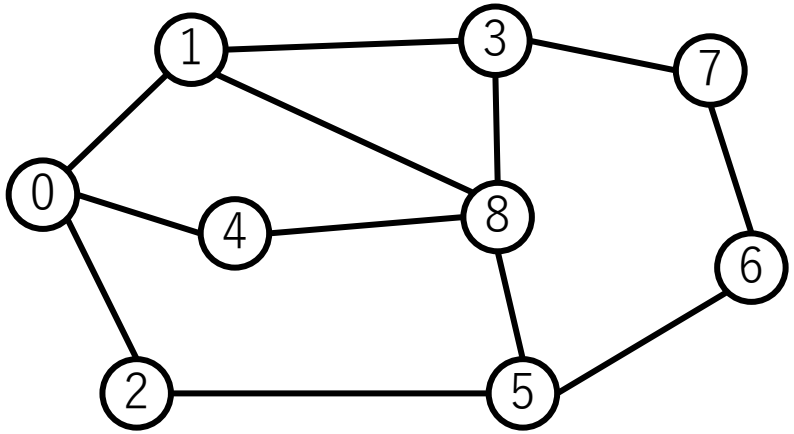
自己ループ

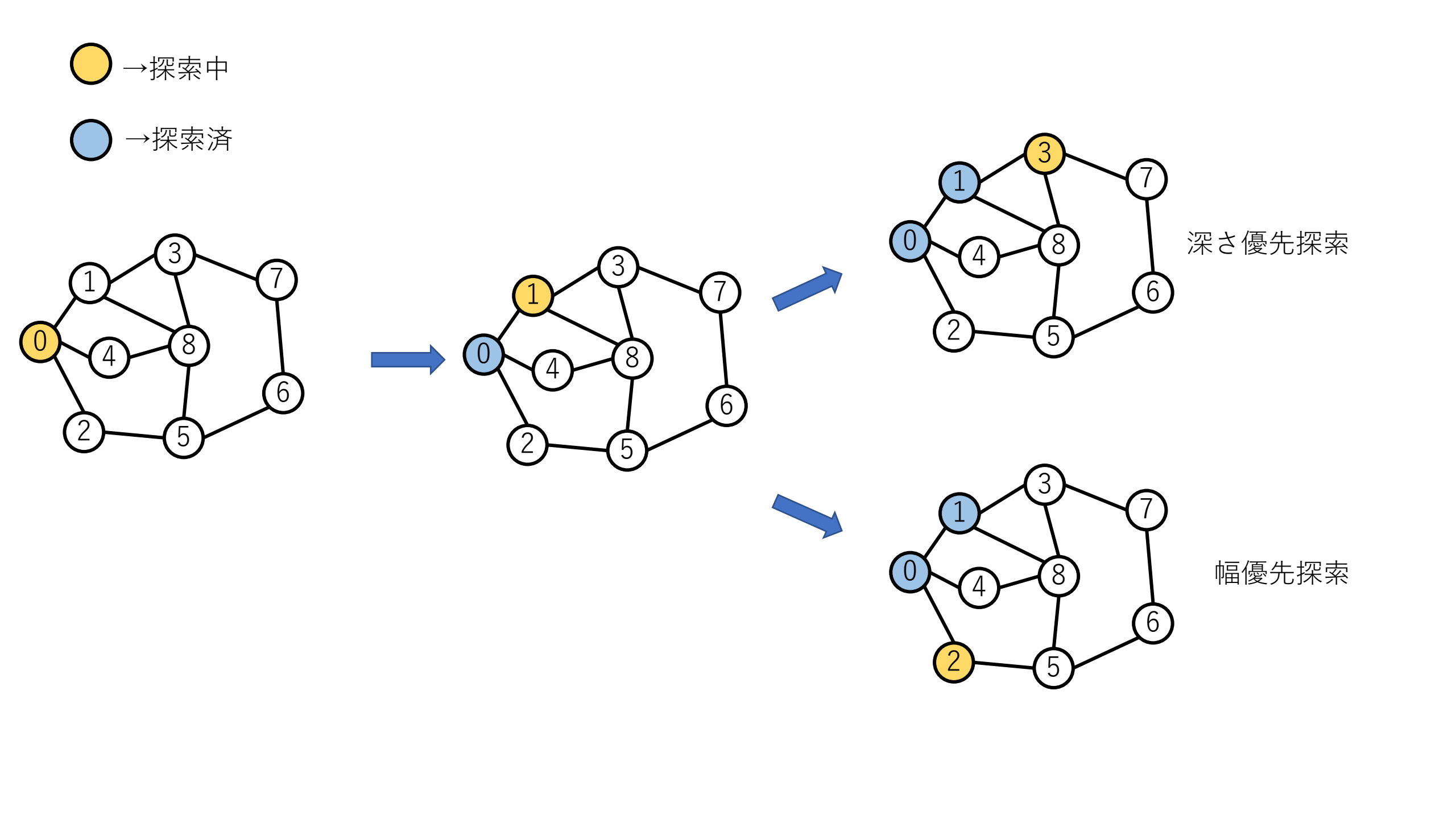


多重辺

グラフ探索

- 全ての頂点を 1 回ずつ探索する
- 次に探索できる頂点は、今まで探索した頂点と辺で繋がっているもの
- 最初は頂点 0 から探索




→探索中
→探索済
0
1
2
3
4
5
6
7
8
深さ優先探索
幅優先探索

深さ優先探索(Depth First Search)

- 前に探索した頂点からどんどん進んでいく探索方法
- 進めなくなったら枝分かれのあるところまで引き返す

- いいところ
  - 引き返す時は進めるところは全部探索されている→トポロジカルソート
  - 枝刈りができる→高速化

- 実装方法
  - スタックを使う
  - 再帰関数

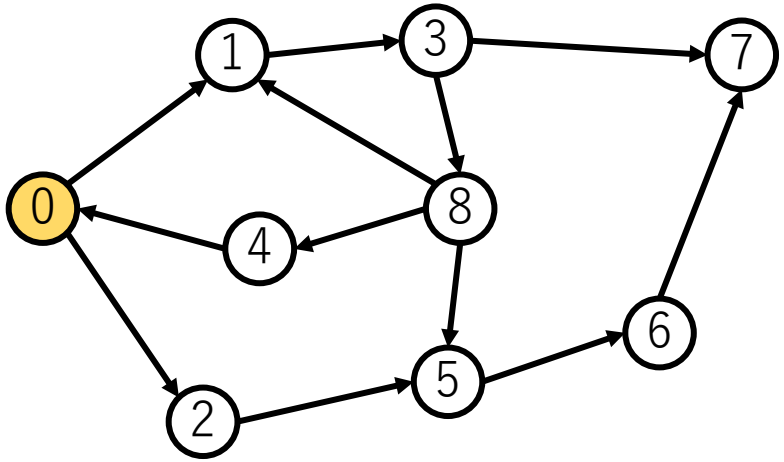
再帰関数を使ったDFS

```
#include <iostream>
#include <vector>
using namespace std;
using Graph = vector<vector<int>>;
vector<bool> seen;
//頂点 s からいけるところを全て探索する(深さ優先探索)
void depthFirstSearch(const Graph& G, int s) {
    //頂点sを探索
    seen[s] = true; //sを訪問済みにする
    for (int v : G[s]) {
        if (seen[v]) {
            continue; //探索済みなら何もしない
        }
        depthFirstSearch(G, v);
    }
    //いけるところを探索しきった後の処理
}
int main() {
    int N;
    cin >> N; //頂点数Nを受け取る
    seen.assign(N, false); //配列を初期化
    Graph G(N);
    //Gの入力
    depthFirstSearch(G, 0);
}
```

再帰関数による深さ優先探索

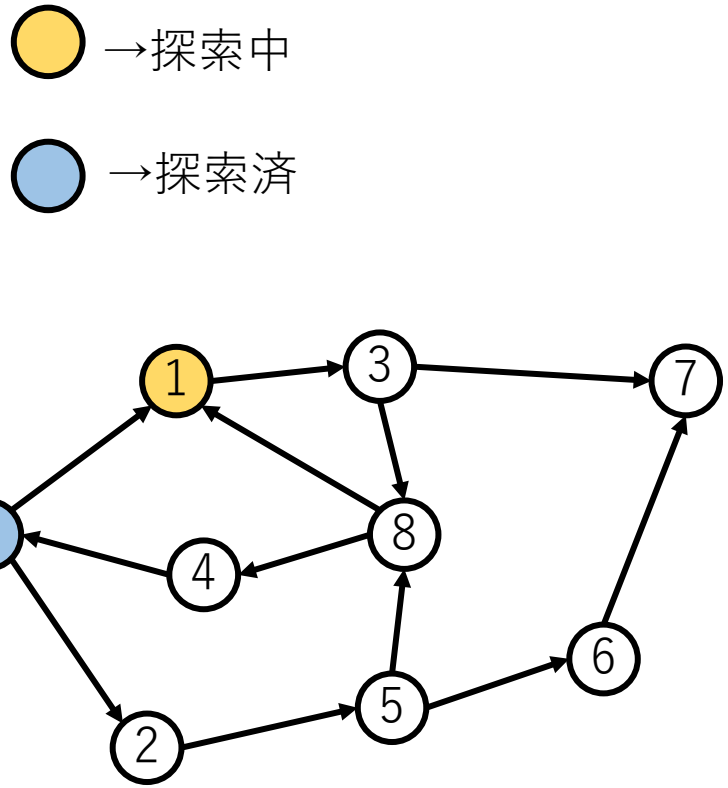
→探索中

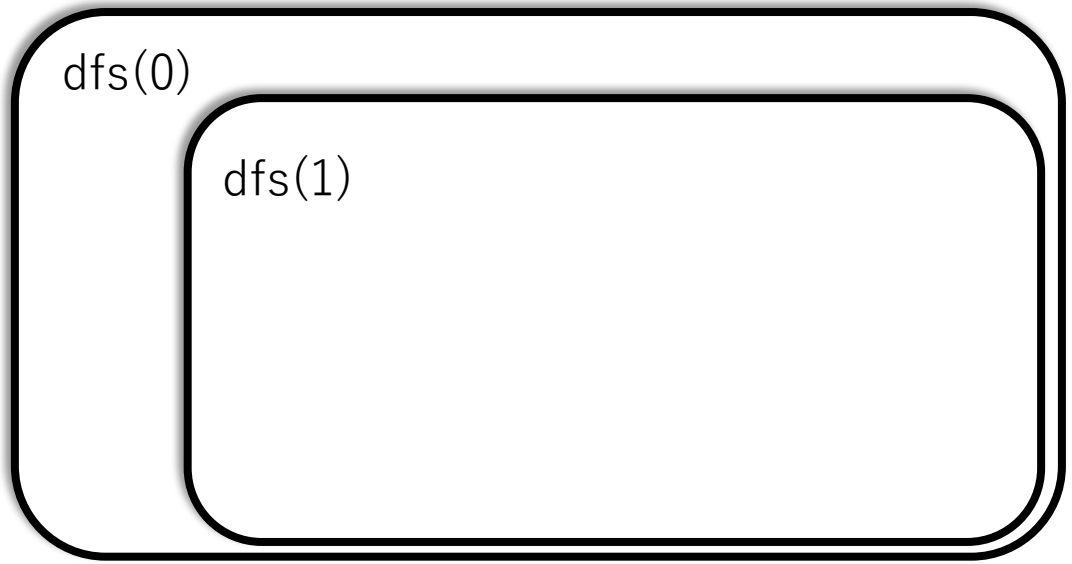
→探索済



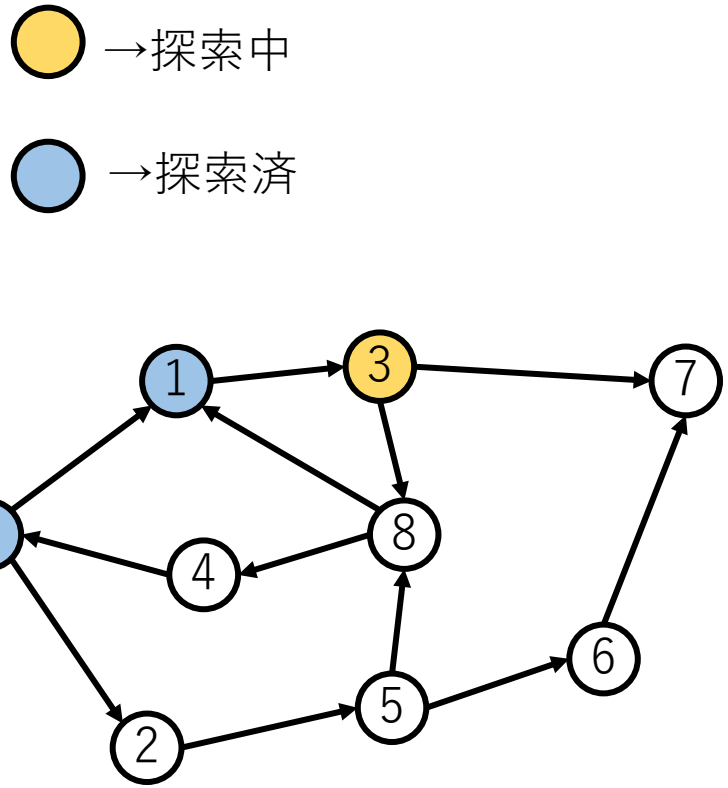
dfs(0)

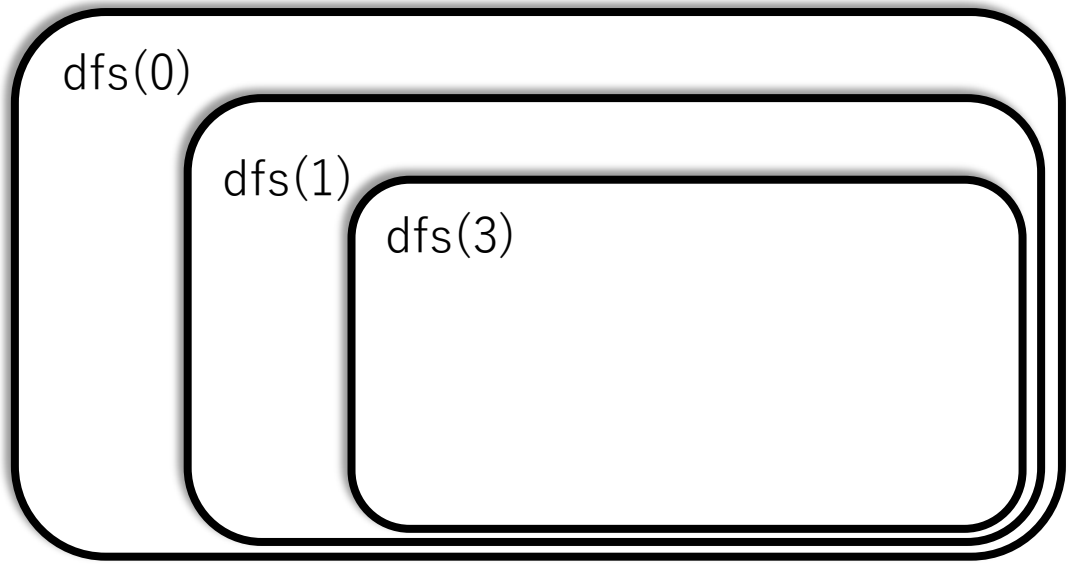
再帰関数による深さ優先探索



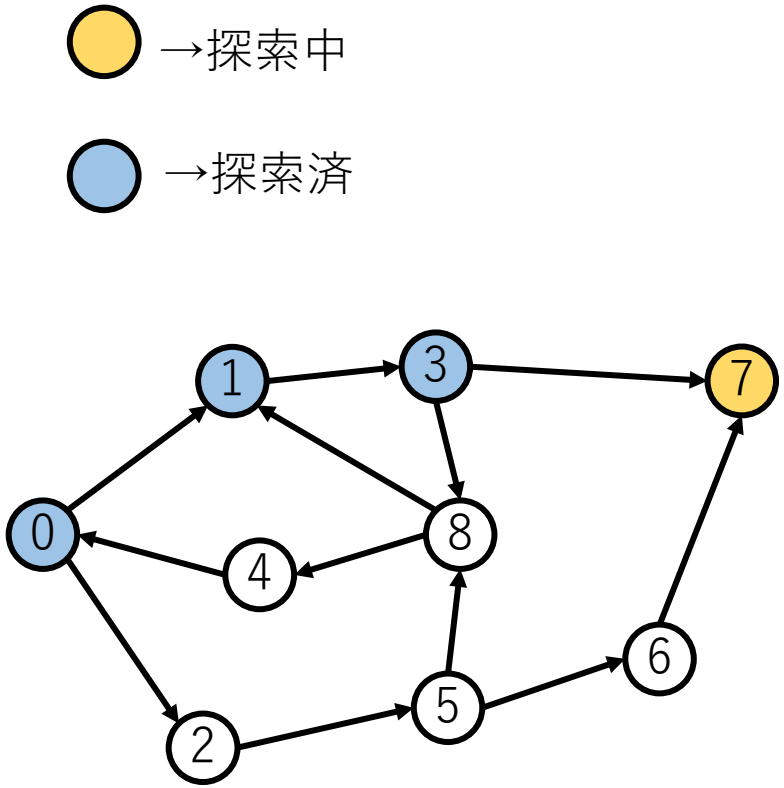


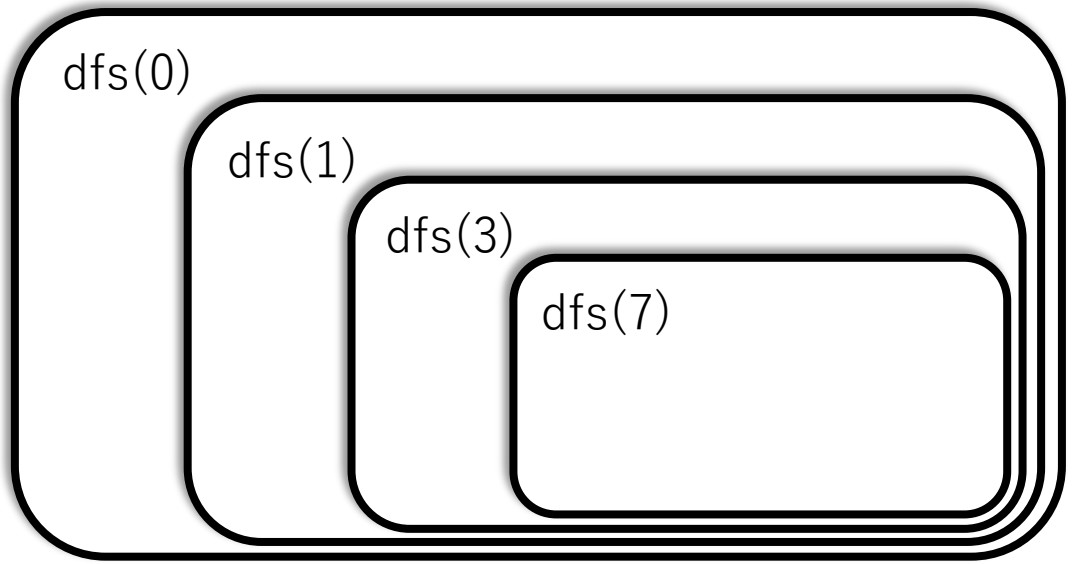
再帰関数による深さ優先探索



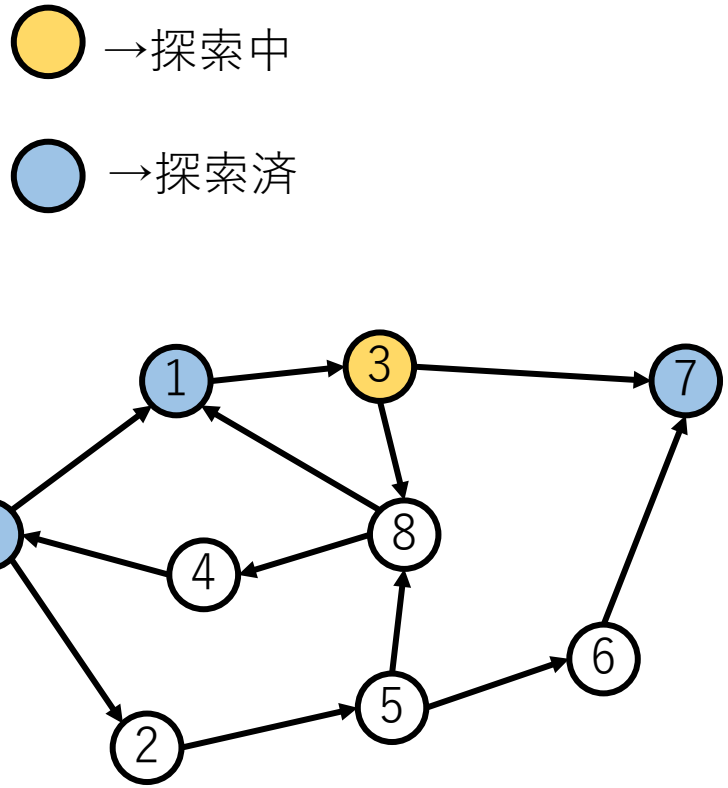


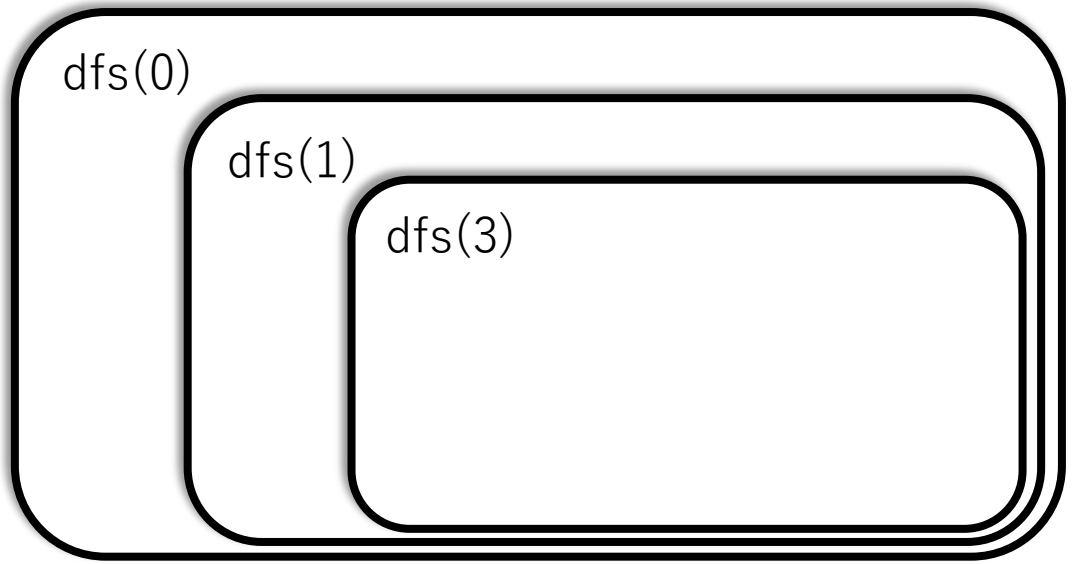
再帰関数による深さ優先探索



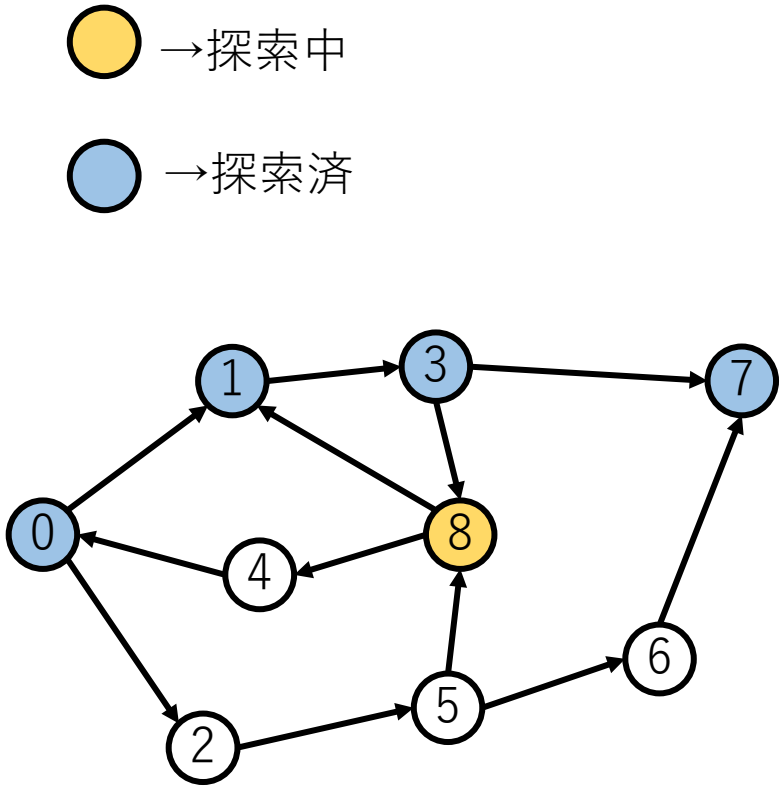


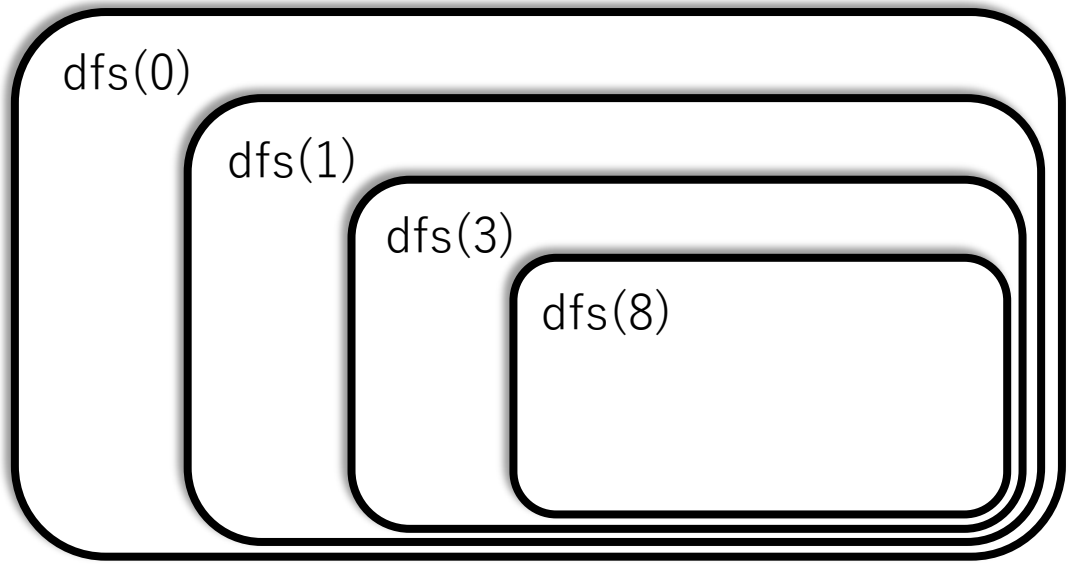
再帰関数による深さ優先探索



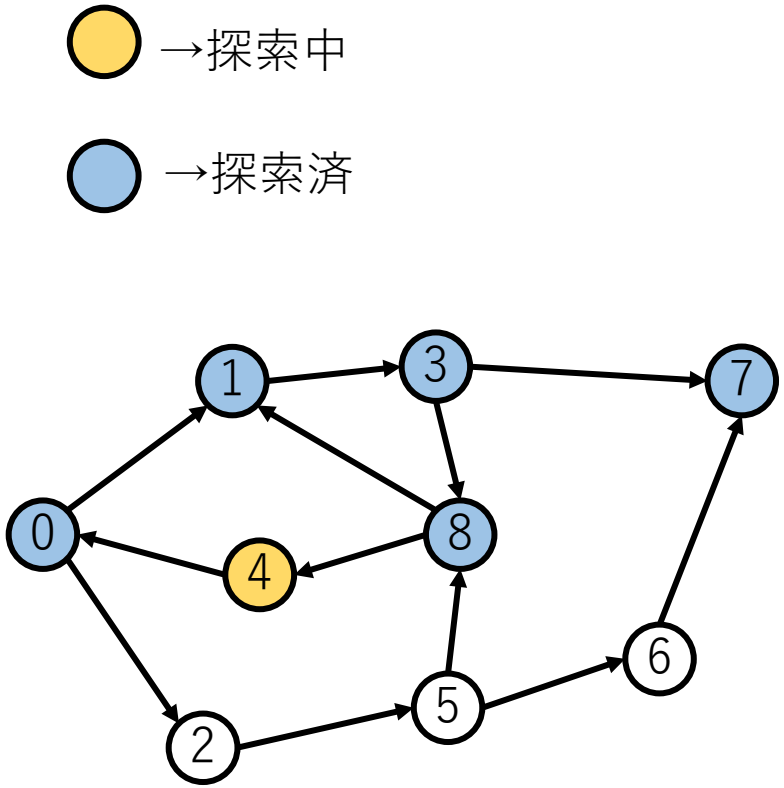


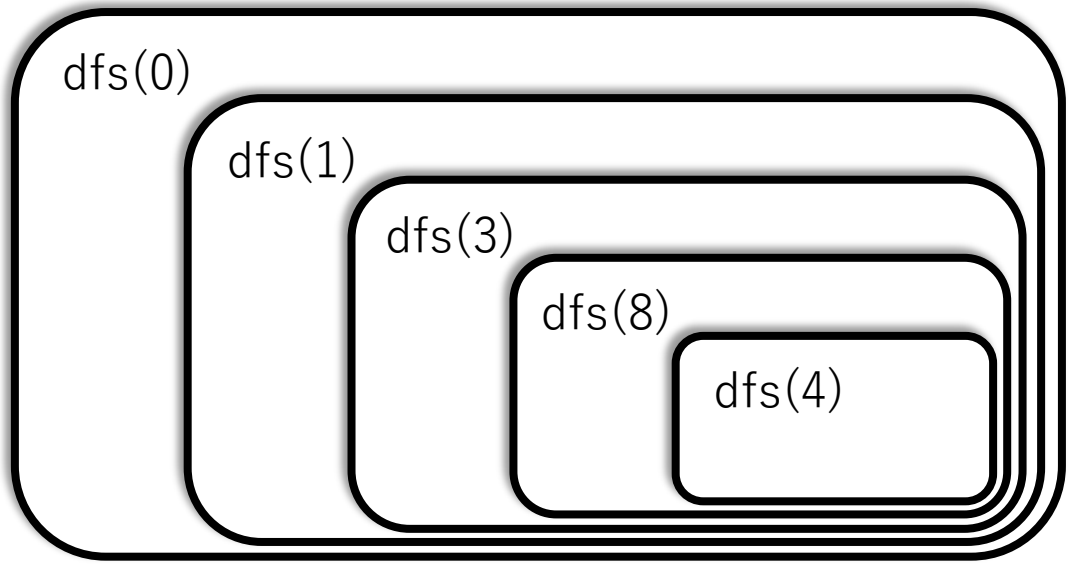
再帰関数による深さ優先探索



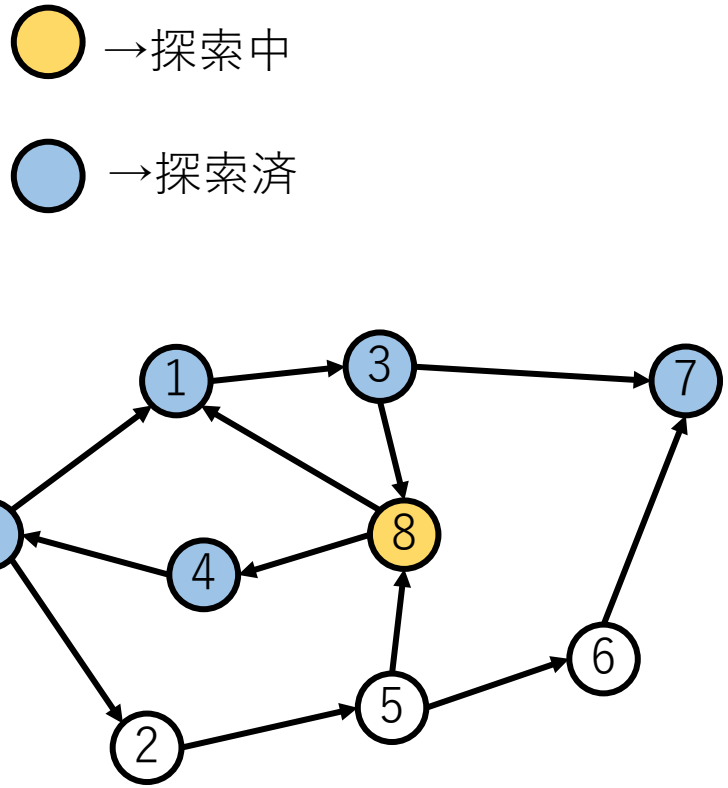


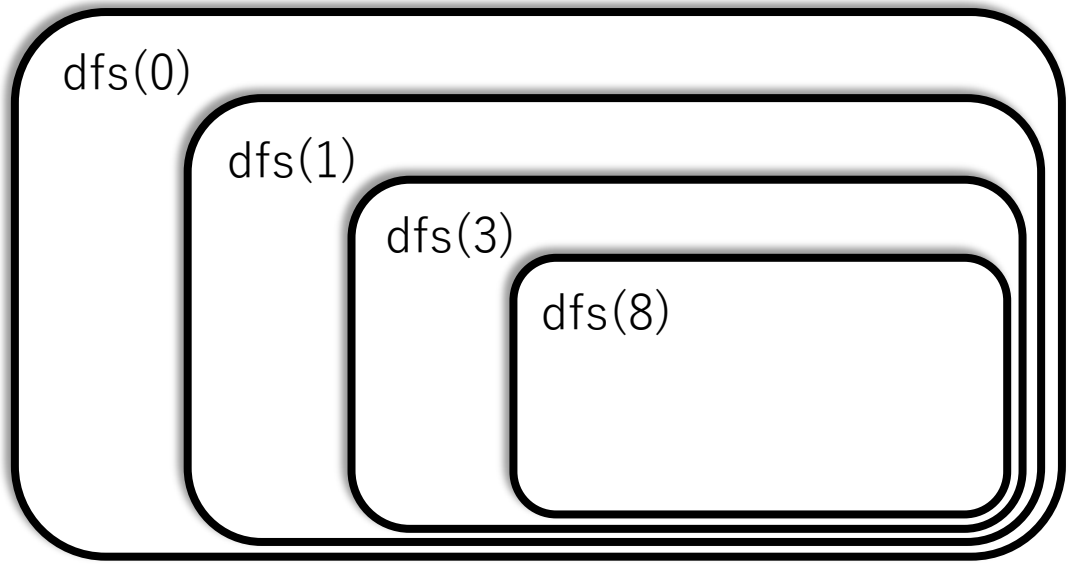
再帰関数による深さ優先探索



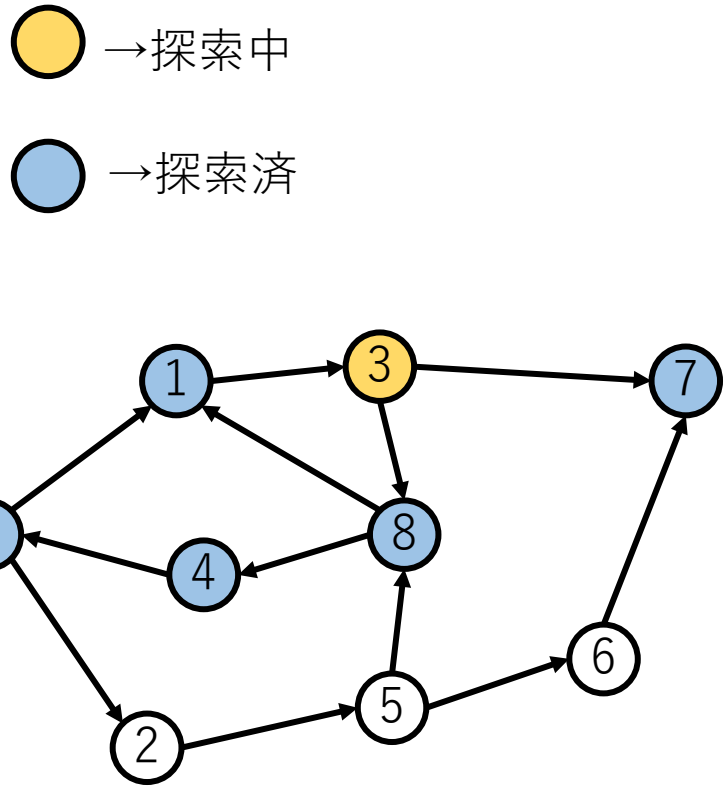


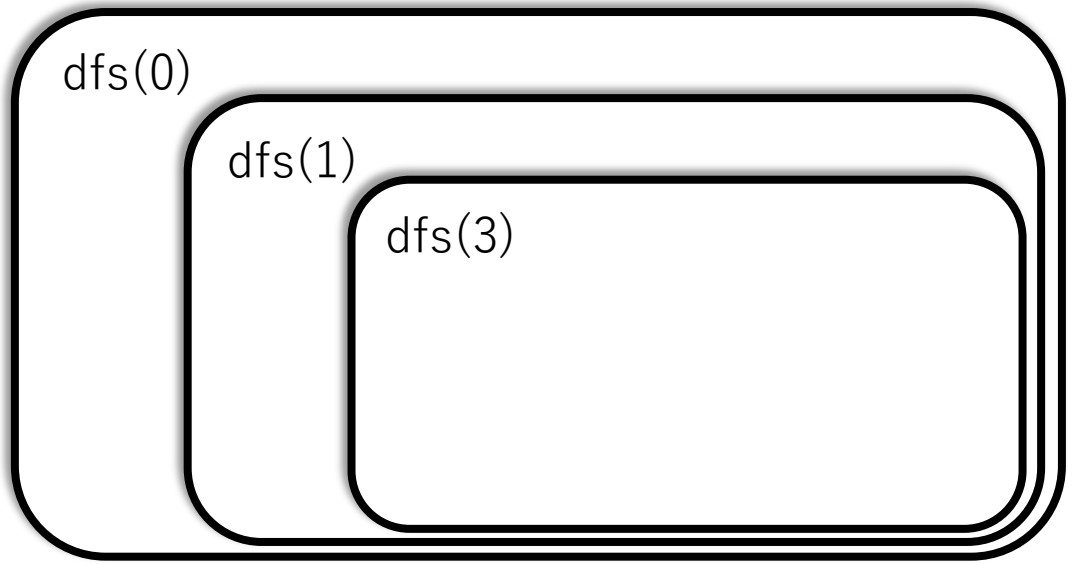
再帰関数による深さ優先探索



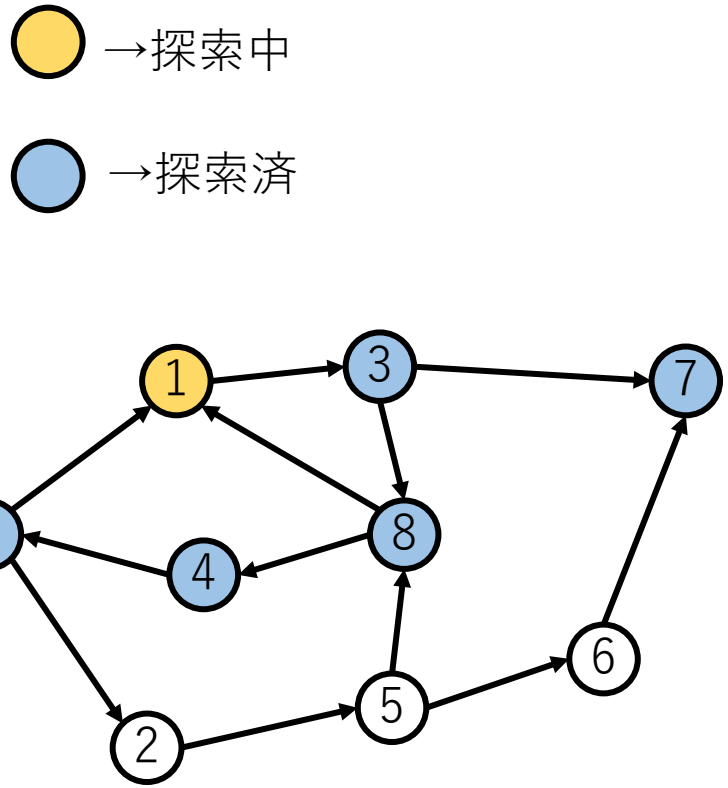


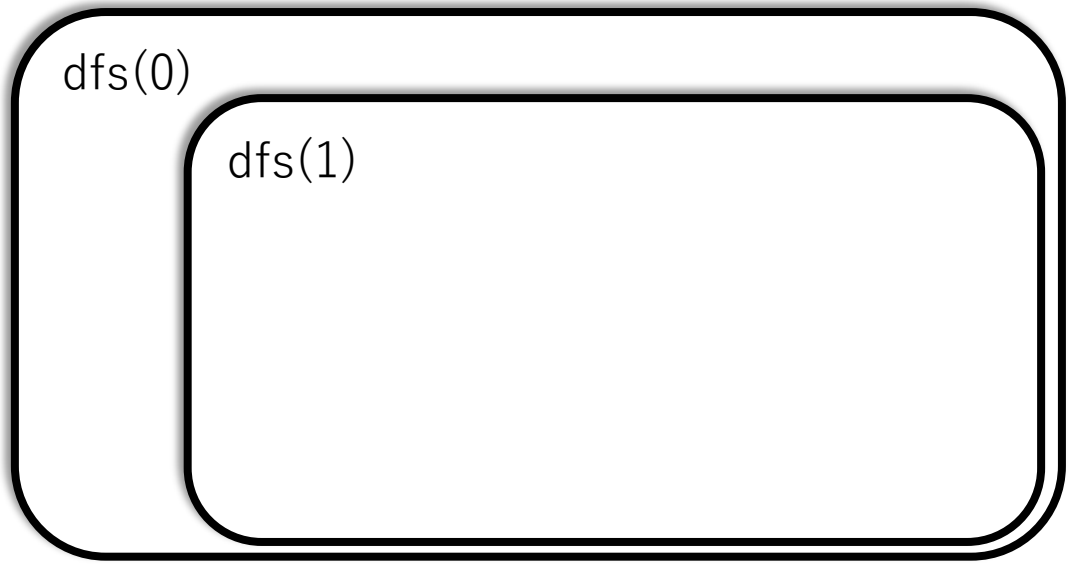
再帰関数による深さ優先探索





再帰関数による深さ優先探索

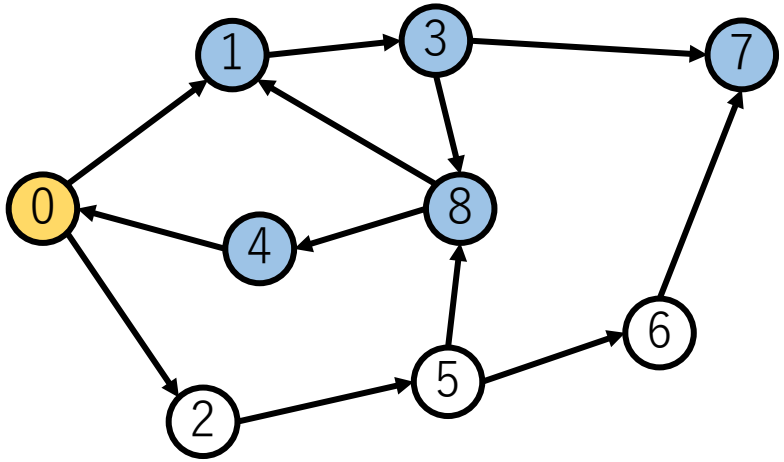




再帰関数による深さ優先探索

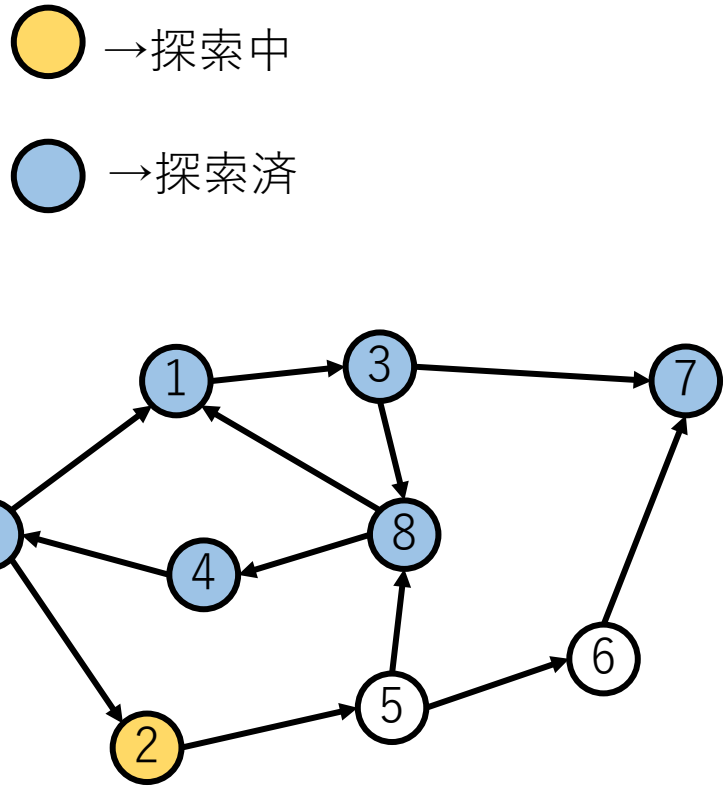
→探索中

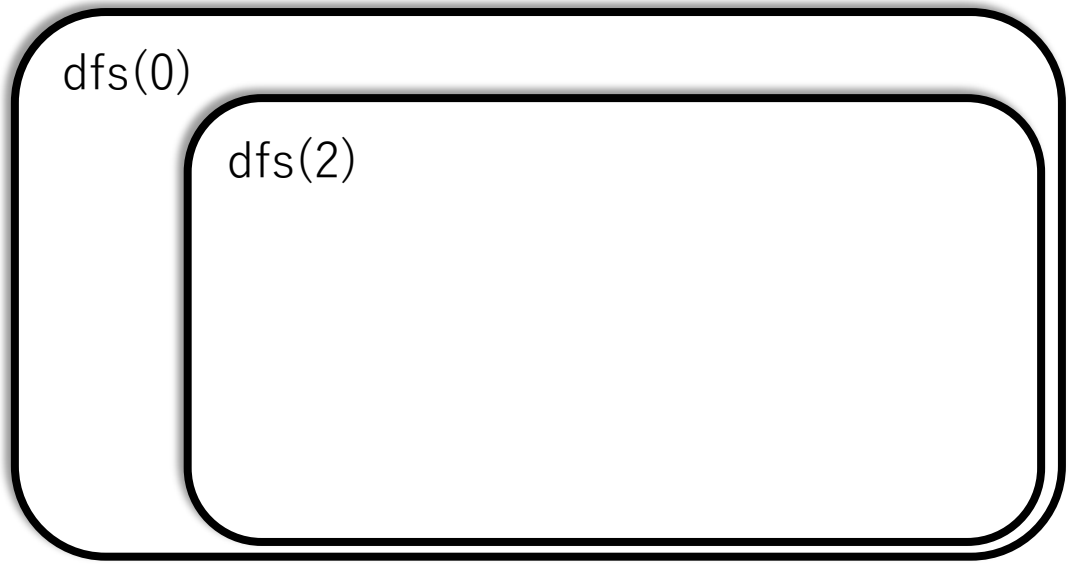
→探索済



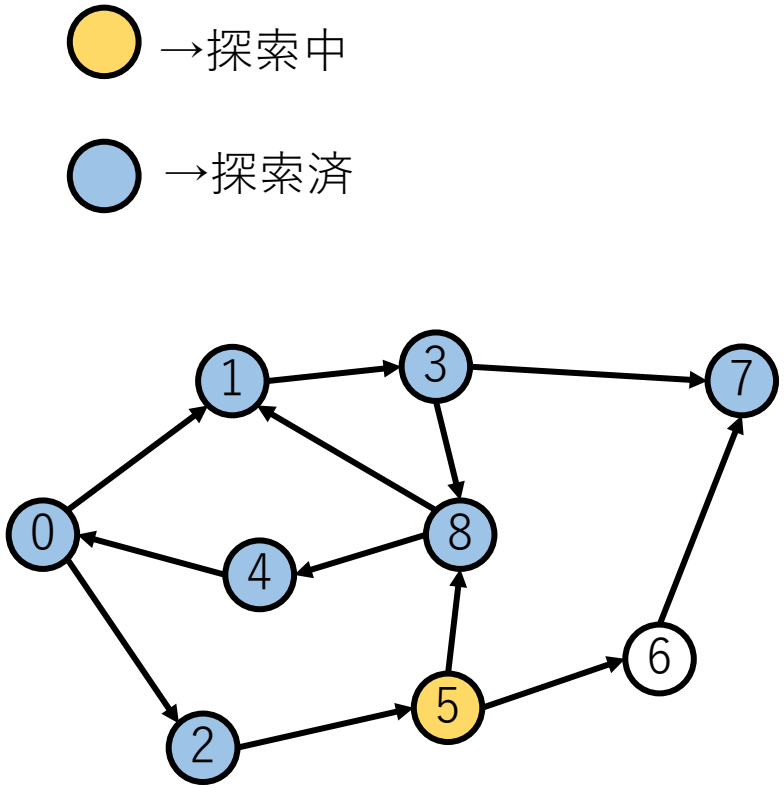
dfs(0)

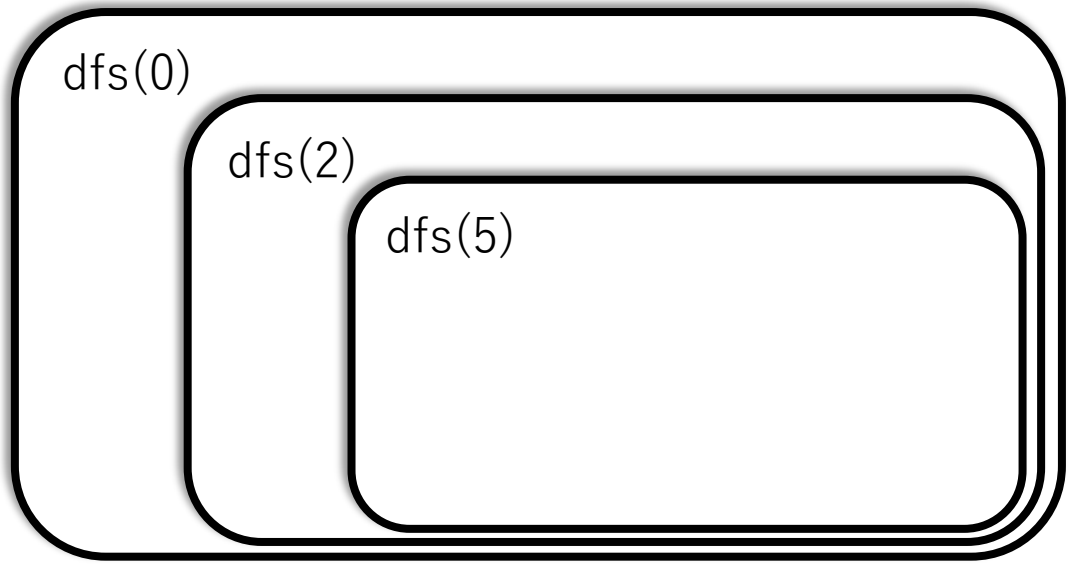
再帰関数による深さ優先探索



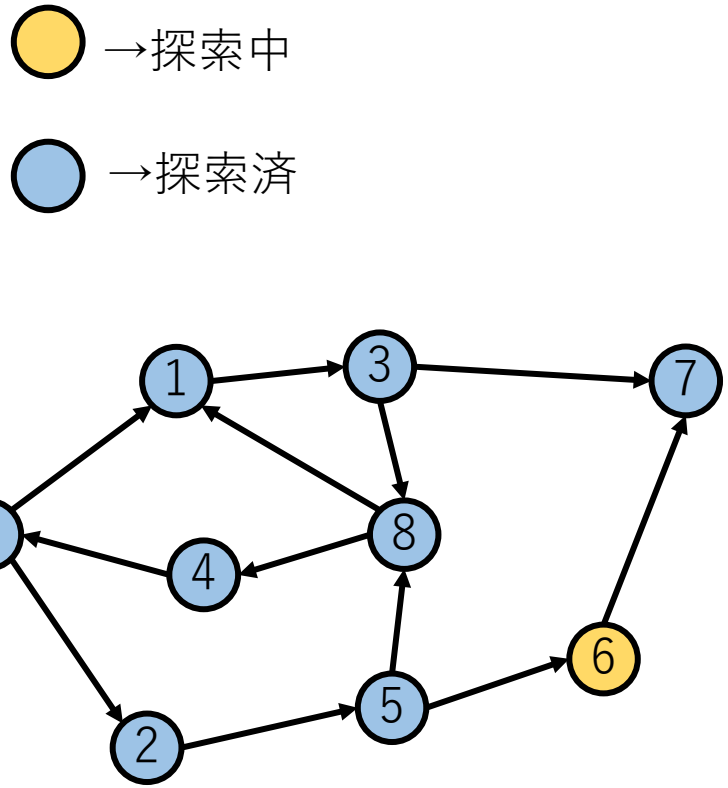


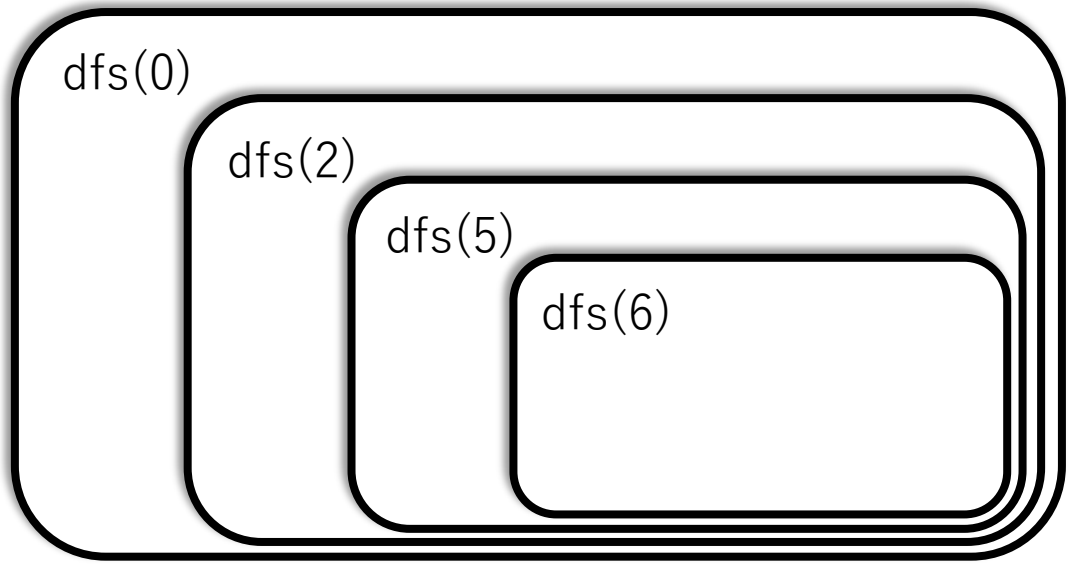
再帰関数による深さ優先探索



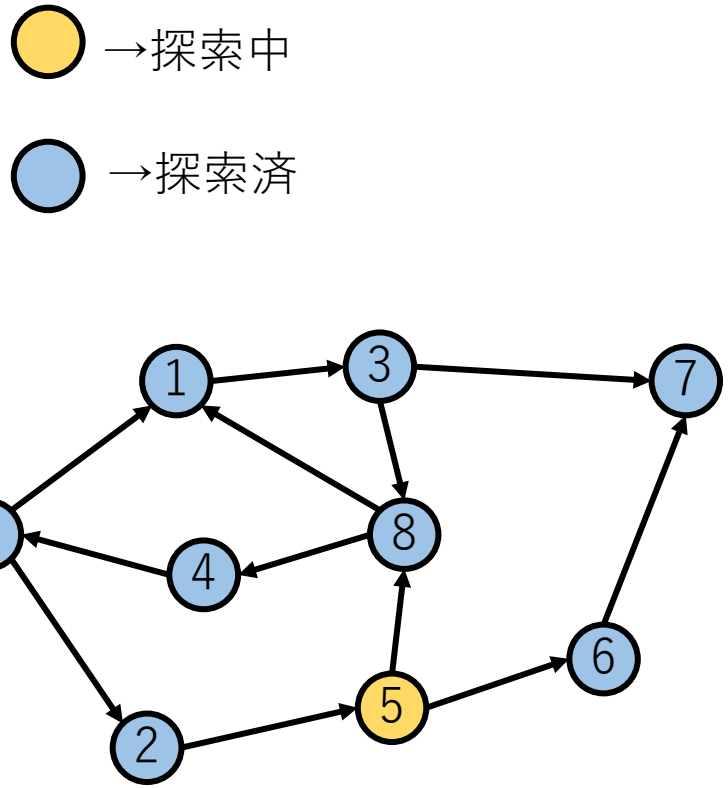


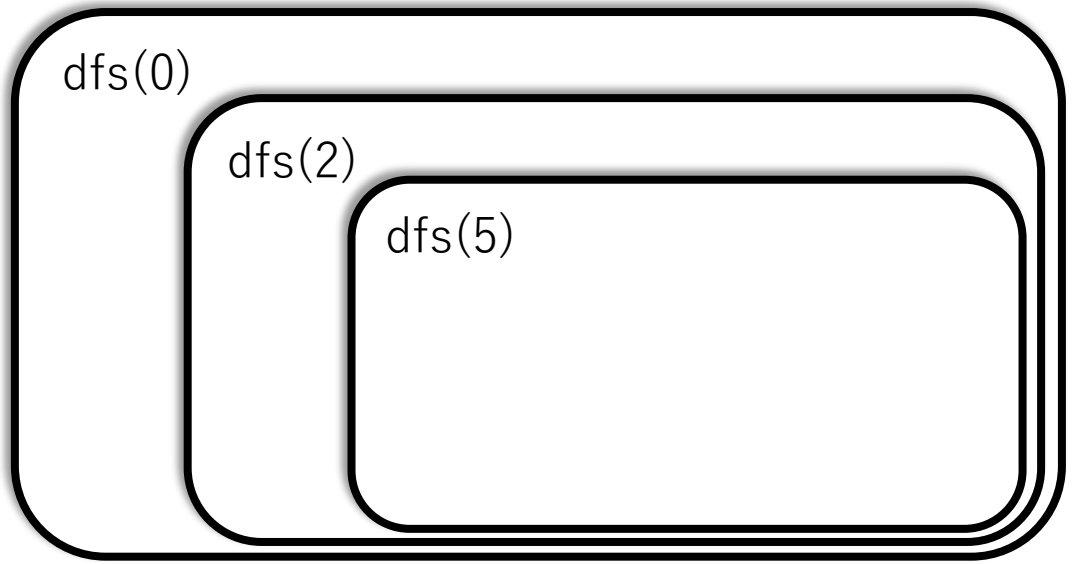
再帰関数による深さ優先探索



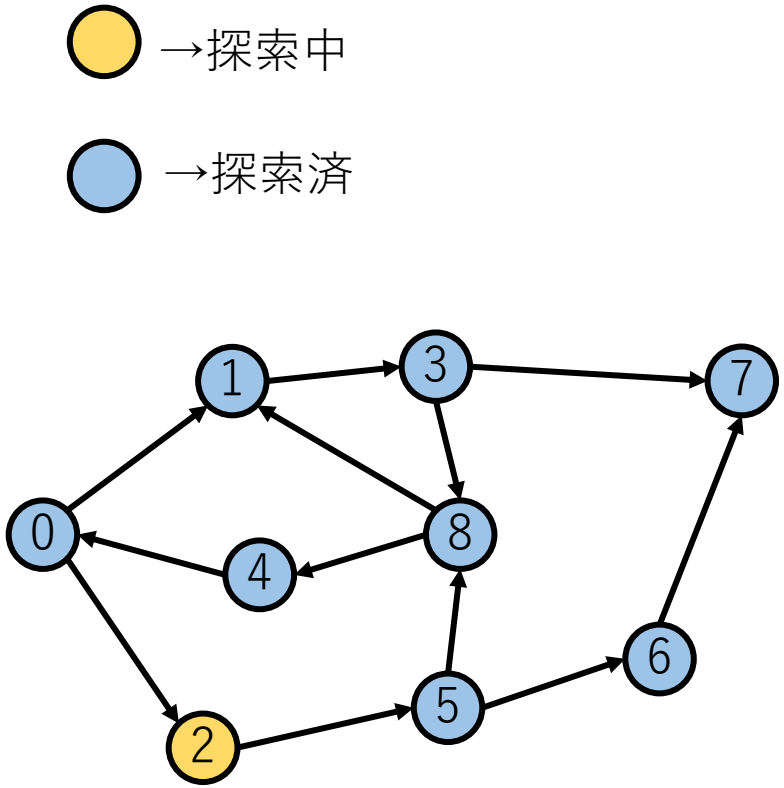


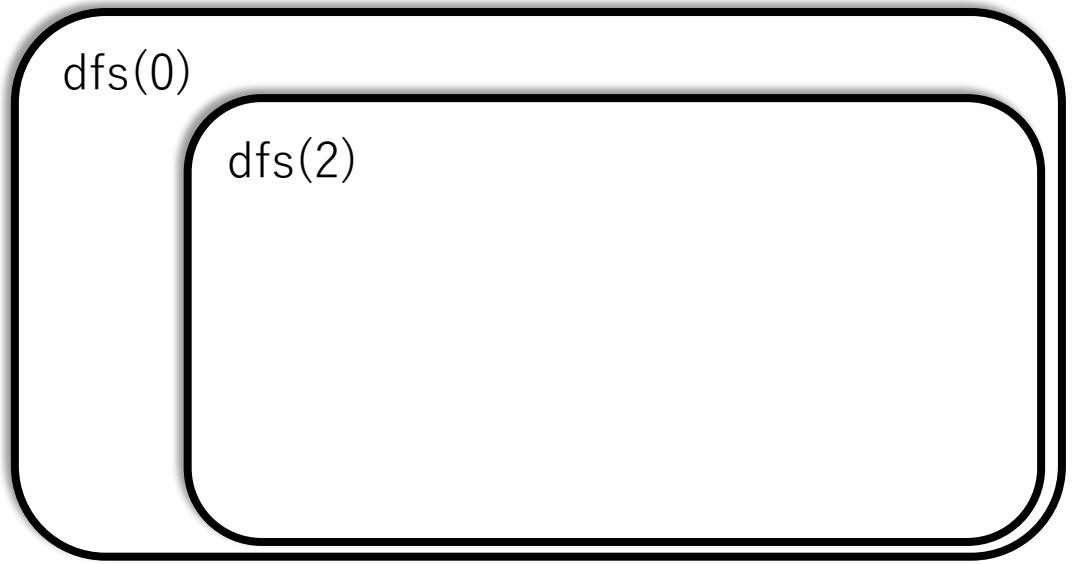
再帰関数による深さ優先探索





再帰関数による深さ優先探索

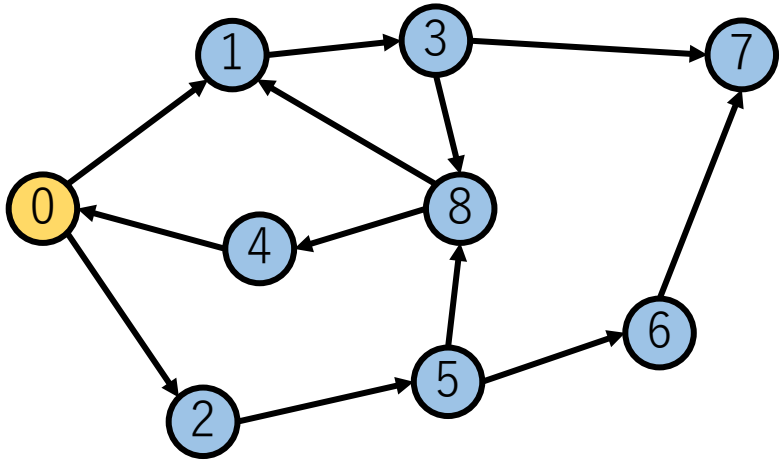




再帰関数による深さ優先探索

→探索中

→探索済

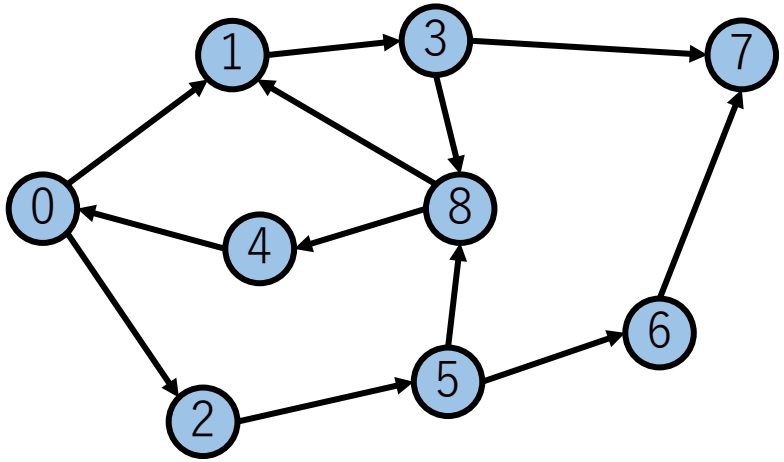


dfs(0)

再帰関数による深さ優先探索

→探索中

→探索済



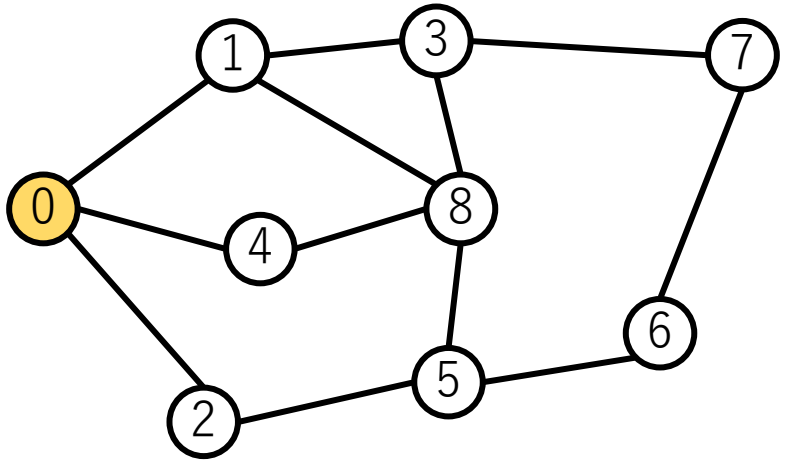
スタックを使ったDFS

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;
using Graph = vector<vector<int>>;

//スタックを使ってsから深さ優先探索をする
void depthFirstSearch(const Graph& G, int s) {
    int N = (int)G.size(); //グラフの頂点数
    stack<int> todo; //これから探索する頂点のスタック
    vector<bool> seen(N, false); //スタックに入ったことがあるか判定する

    seen[s] = true; //s はtodoに入る
    todo.push(s); //s を todo に入れる

    while (!todo.empty()) {
        int u = todo.top(); todo.pop(); //uを探索
        for (int v : G[u]) { //範囲for文
            if (seen[v]) {
                continue; //すでにスタックに入ったことがあるので、何もしない
            }
            todo.push(v);
            seen[v] = true;
        }
    }
}
```

スタックによる深さ優先探索

→探索中

→探索済



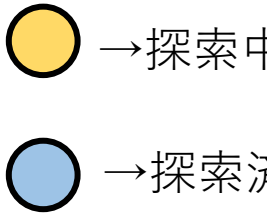
[1, 2, 4]

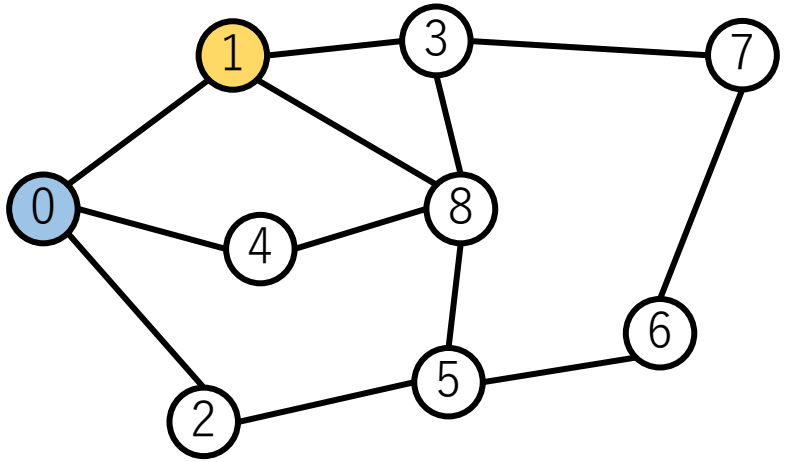
←top　　back→

in : 1, 2, 4

out : 0

スタックによる深さ優先探索





[3, 8, 2, 4]

←top　　back→

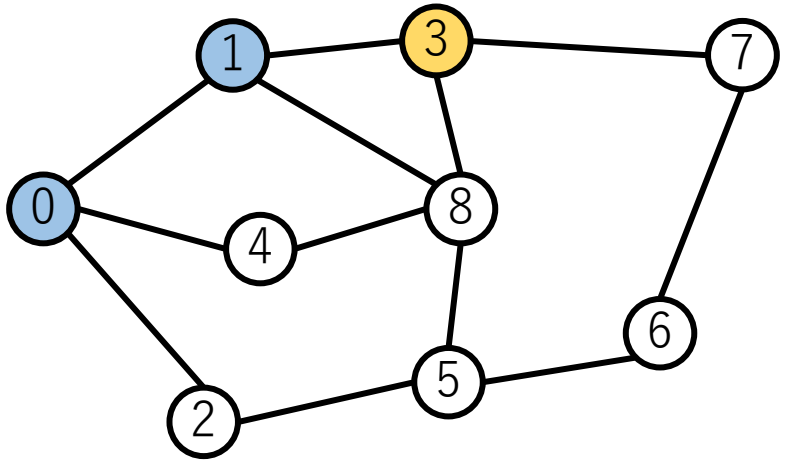
in : 3, 8

out : 1

スタックによる深さ優先探索

→探索中

→探索済



[7, 8, 2, 4]

←top　　back→

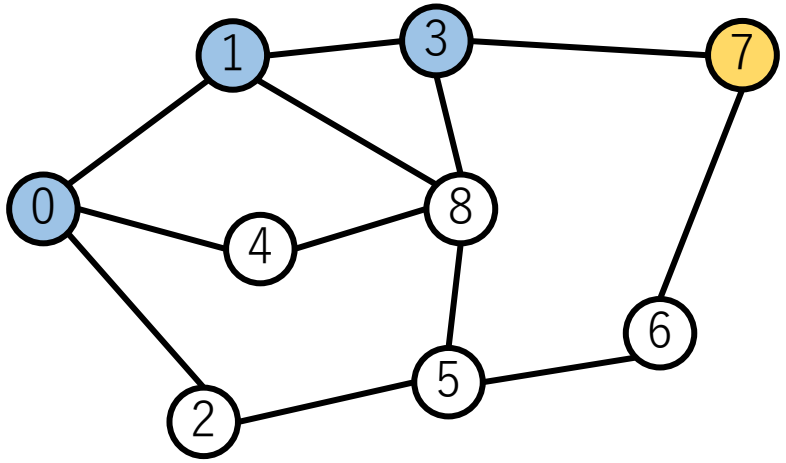
in : 7

out : 3

スタックによる深さ優先探索

→探索中

→探索済



[6, 8, 2, 4]

←top　　back→

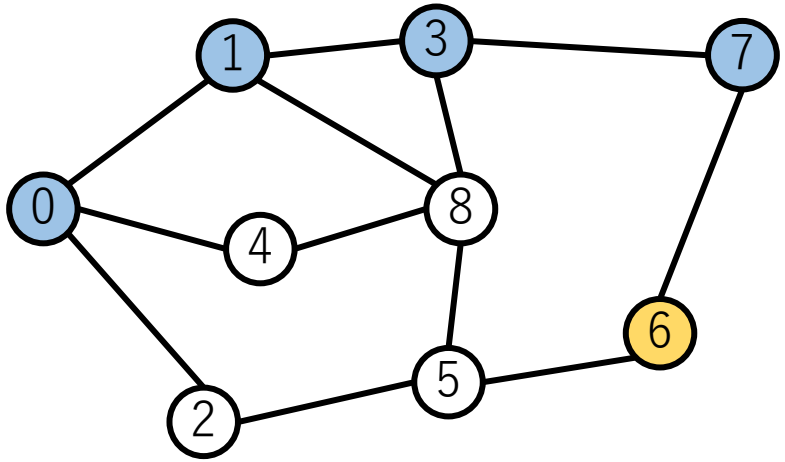
in : 6

out : 7

スタックによる深さ優先探索

→探索中

→探索済



[5, 8, 2, 4]

←top back→

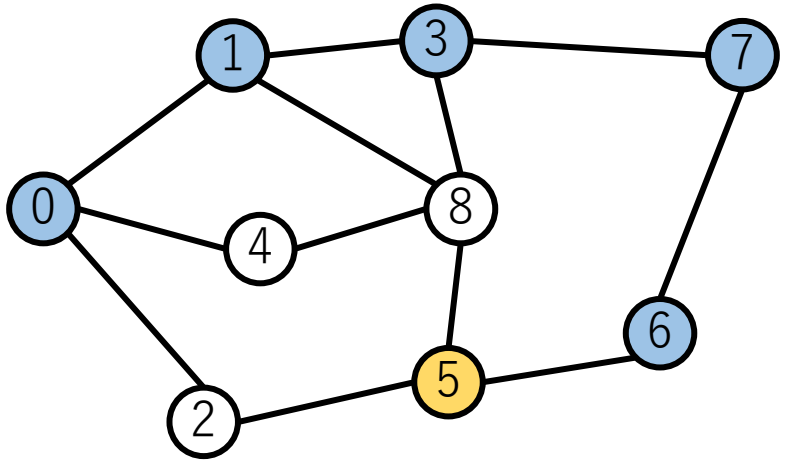
in : 5

out : 6

スタックによる深さ優先探索

→探索中

→探索済



[8, 2, 4]

←top　　back→

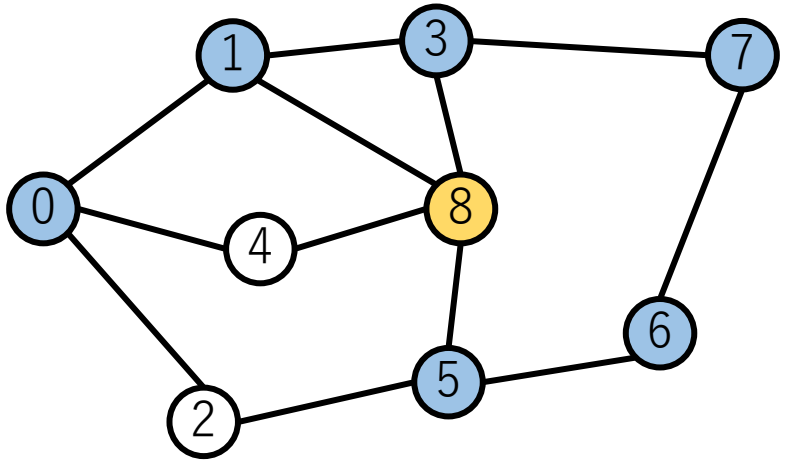
in :

out : 5

スタックによる深さ優先探索

→探索中

→探索済



[2, 4]

←top　　back→

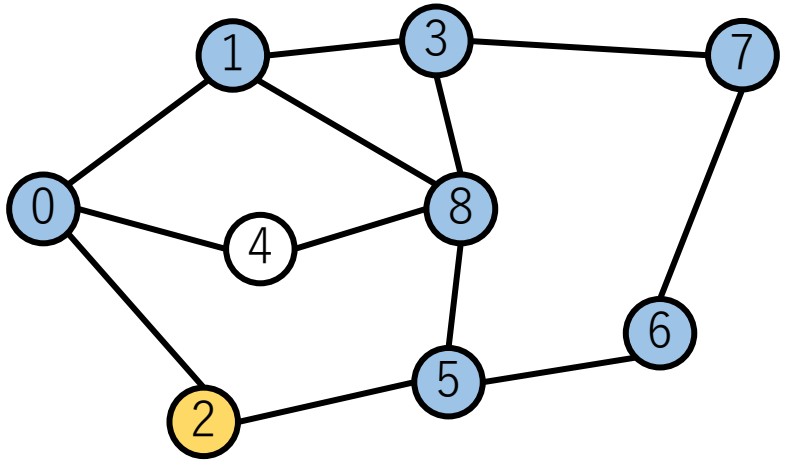
in :

out : 8

スタックによる深さ優先探索

→探索中

→探索済



[4]

←top　　back→

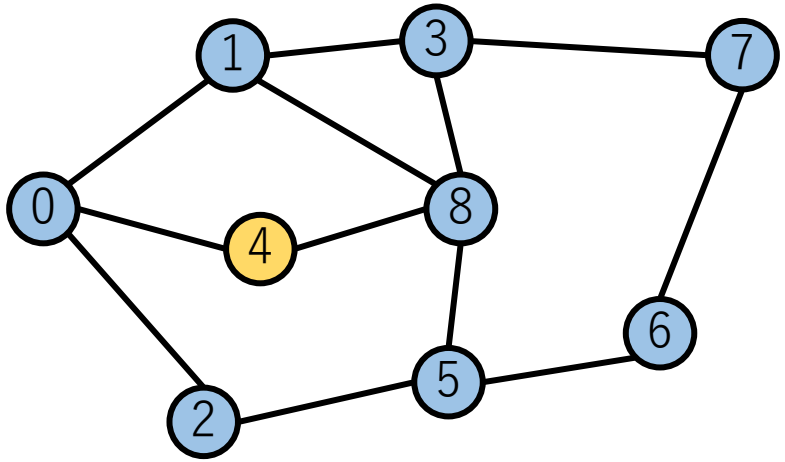
in :

out : 2

スタックによる深さ優先探索

→探索中

→探索済



[]

←top　　back→

in :

out : 4

幅優先探索(Breadth First Search

- 始点から近い順に調べる探索手法
- 始点から水を流していくイメージ

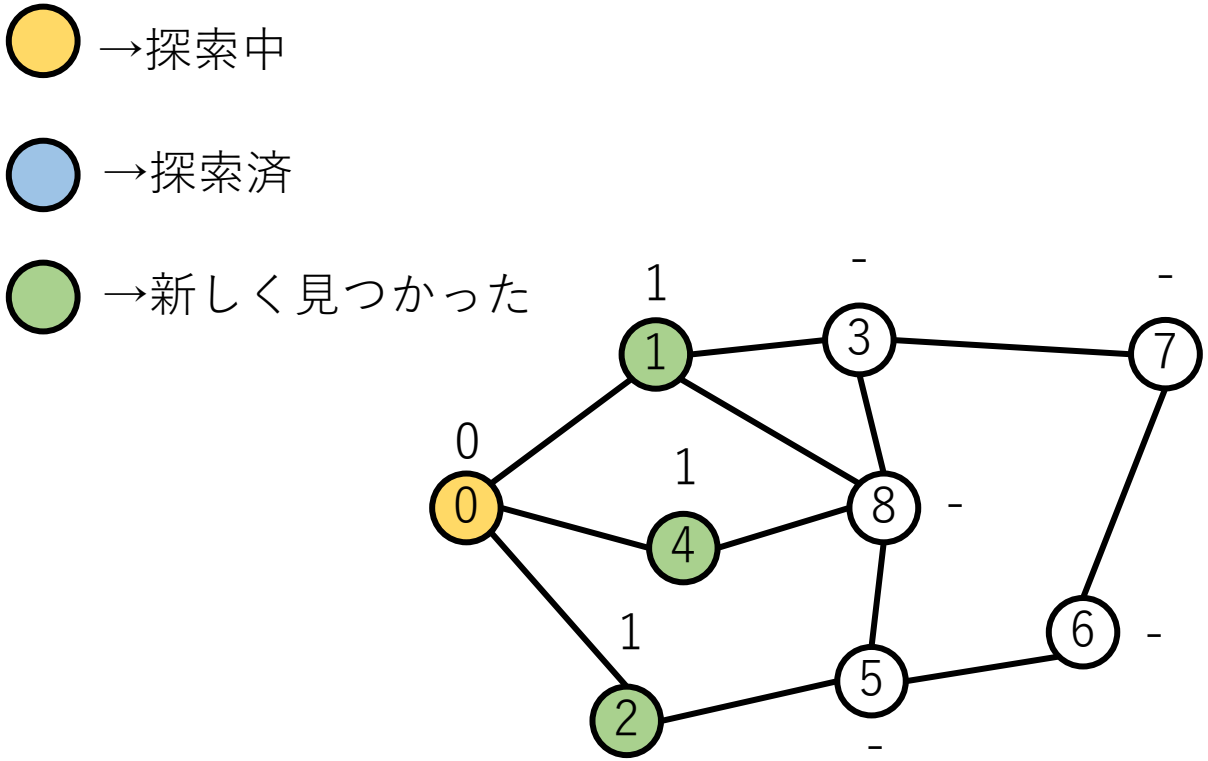
- いいところ
  - 探索を進めていくことで、始点からの距離がわかる
    - ダイクストラ法(この次とかで扱う)とか 0-1BFS とか

- 実装方法
  - キューを使う

幅優先探索の実装方法

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
using Graph = vector<vector<int>>;
//探索するついでに各頂点の始点からの最短距離を求める
vector<int> breadthFirstSearh(const Graph& G, int s) {
    int N = G.size(); //頂点数
    vector<int> dist(N, -1); //最短距離を格納(-1 は未訪問状態)
    queue<int> que;
    //初期化
    que.push(s);
    dist[s] = 0;
    while (!que.empty()) {
        int u = que.front();
        que.pop();
        //頂点 u の探索
        for (int v : G[u]) {
            if (dist[v] != -1) {
                continue; //訪問済みなら何もしない
            }
            que.push(v);
            dist[v] = dist[u] + 1;
        }
    }
    return dist;
}
```

キューによる幅優先探索



[1, 2, 4]

←top　　back→

in : 1, 2, 4

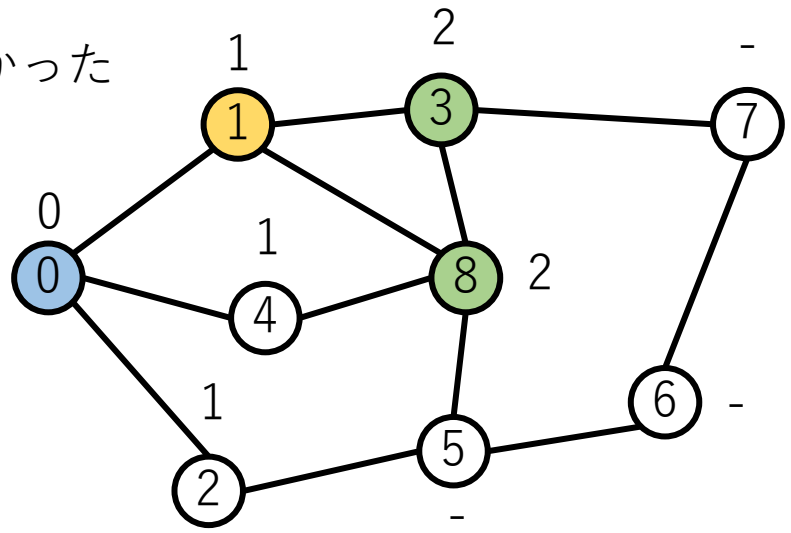
out : 0

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[2, 4, 3, 8]

←top　　back→

in : 3, 8

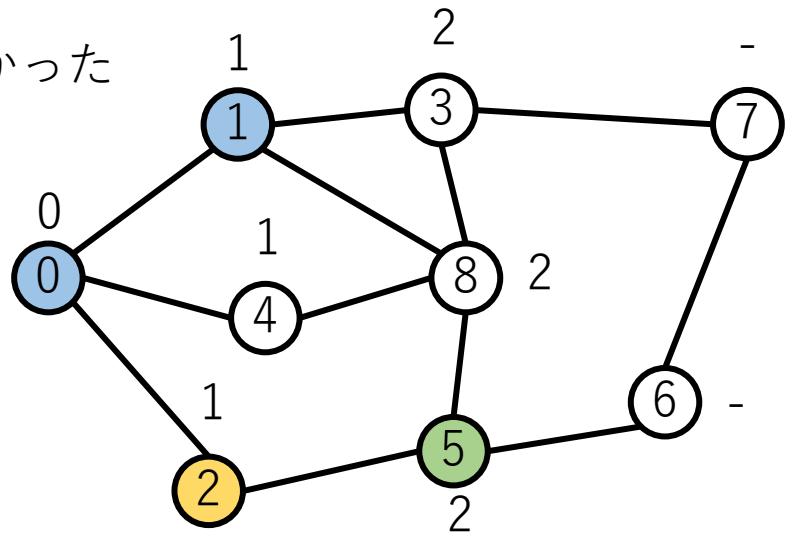
out : 1

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[4, 3, 8, 5]

←top　　back→

in : 5

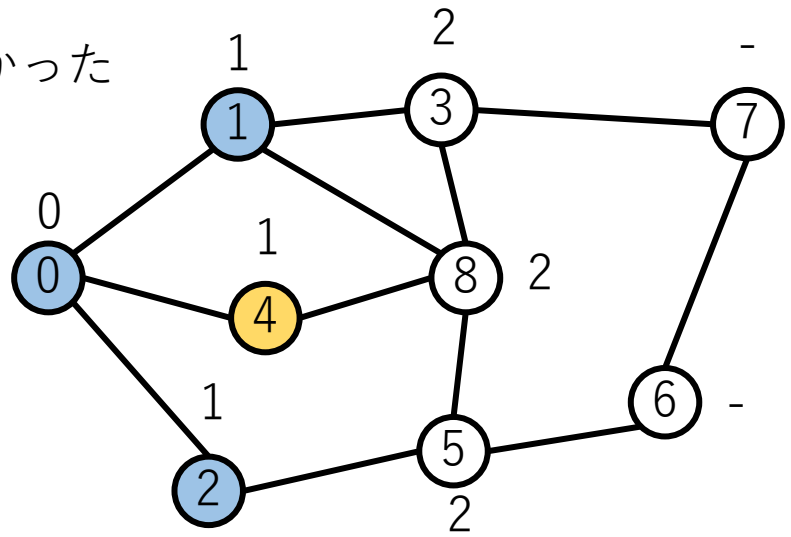
out : 2

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[3, 8, 5]

←top　　back→

in :

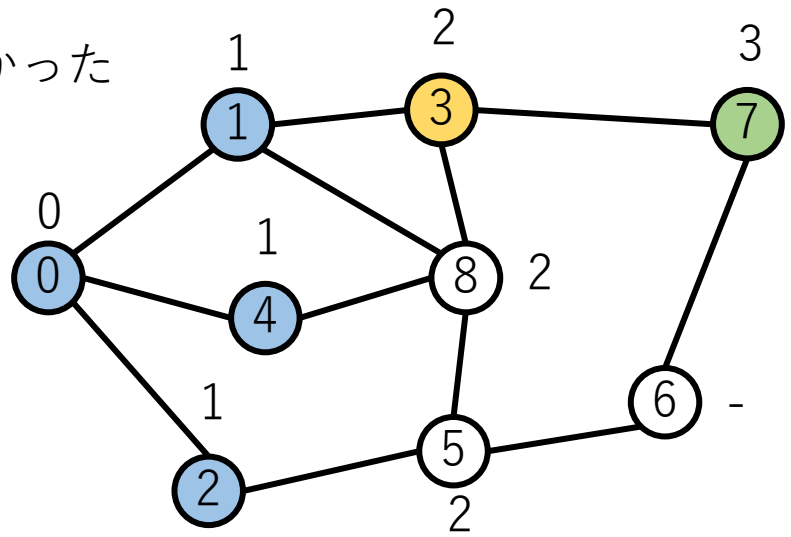
out : 4

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[8, 5, 7]

←top　　back→

in : 7

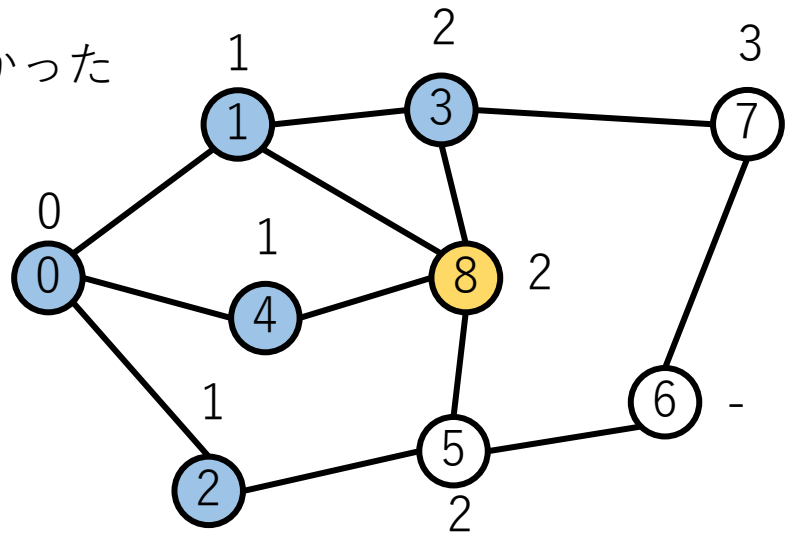
out : 3

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[5, 7]

←top　　back→

in :

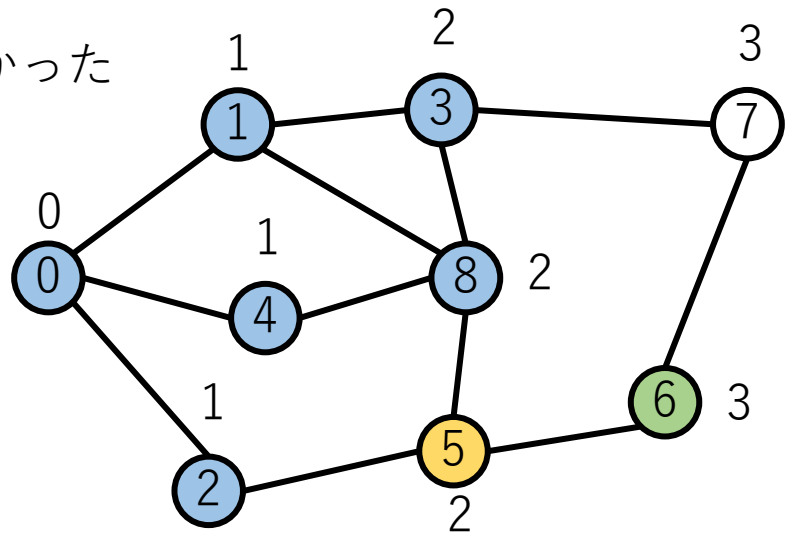
out : 8

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[7, 6]

←top　　back→

in : 6

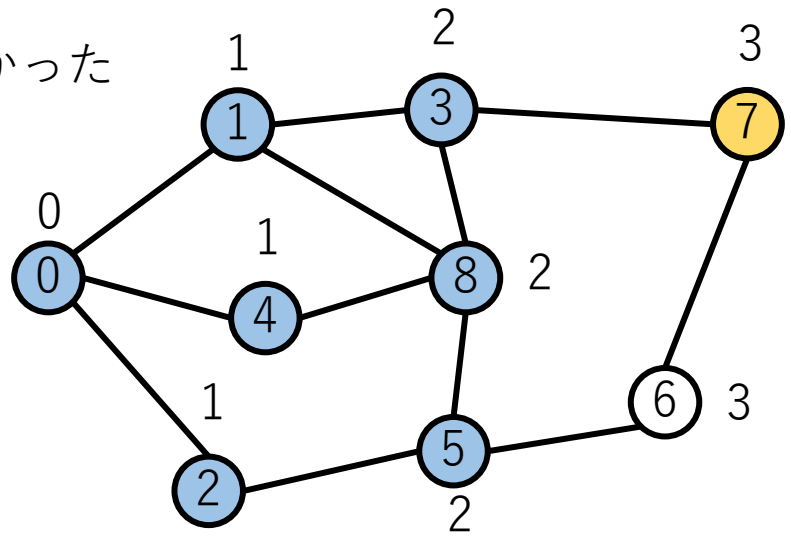
out : 5

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[6]

←top　　back→

in :

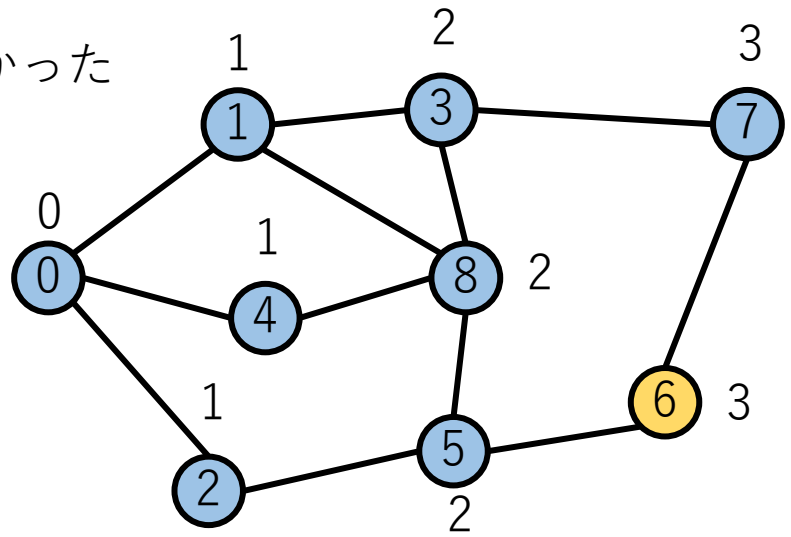
out : 7

キューによる幅優先探索

→探索中

→探索済

→新しく見つかった



[]

←top　　back→

in :

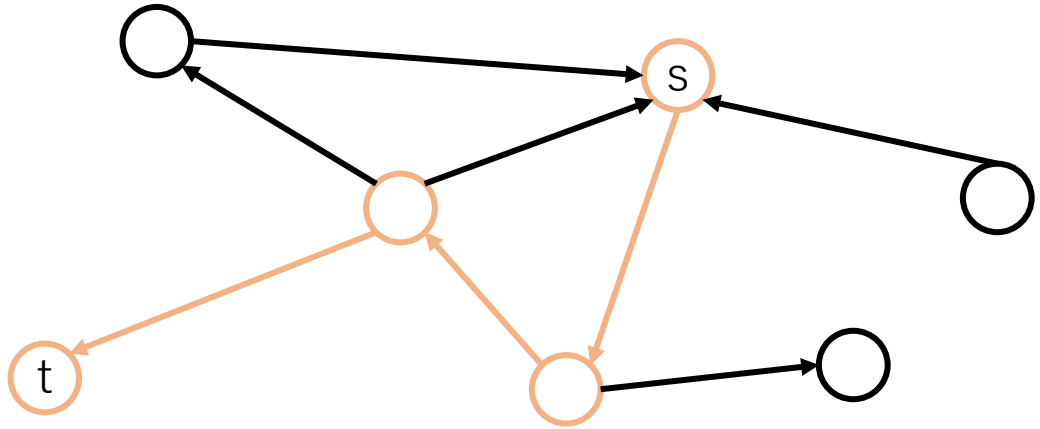
out : 6

計算量

- 全部 O(|V| + |E|)
- V...頂点の集合(vertex)
- E...辺の集合(edge)

グラフ探索の応用例 (1)s-t パス

- グラフ G に頂点 s から t へのパスが存在するか判定する
- s-t パスが存在する⇔ s からグラフ探索すると、tを訪問する

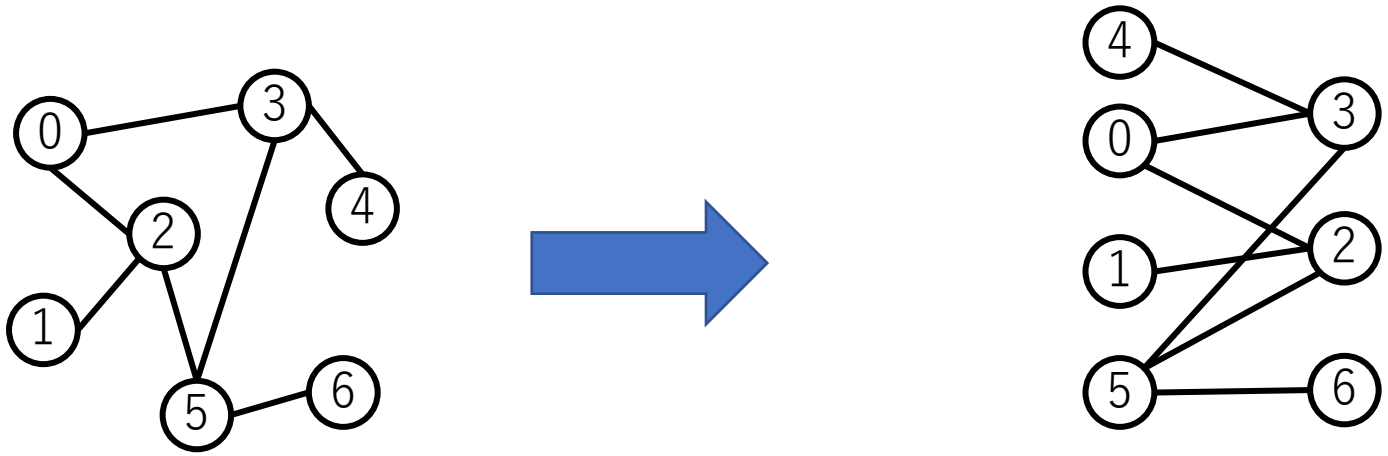


実装例(再帰dfs)

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

vector<bool> seen;
void dfs(const Graph& G, int s) {
    seen[s] = true;
    for (int v : G[s]) {
        if (seen[v]) {
            continue; //訪問済みなところは訪れない
        }
        dfs(G, v);
    }
}
int main() {
    int N, M, s, t;
    cin >> N >> M >> s >> t;
    Graph gr(N + 1);
    seen.assign(N + 1, false);
    for (int i = 0; i < M; i++) {
        int a, b;
        cin >> a >> b;
        gr[a].push_back(b);
    }
    dfs(gr, s);
    cout << (seen[t] ? "Yes" : "No") << endl;
}
```

応用例 (2) 2部グラフ判定

- 2部グラフ...頂点を2グループに分けて、同じグループ間に辺が存在しないようにできるグラフ
- 奇数長のサイクルの検出に使われたりする
  - グラフ G が 2部グラフ⇔Gが奇数長のサイクルを**含まない**



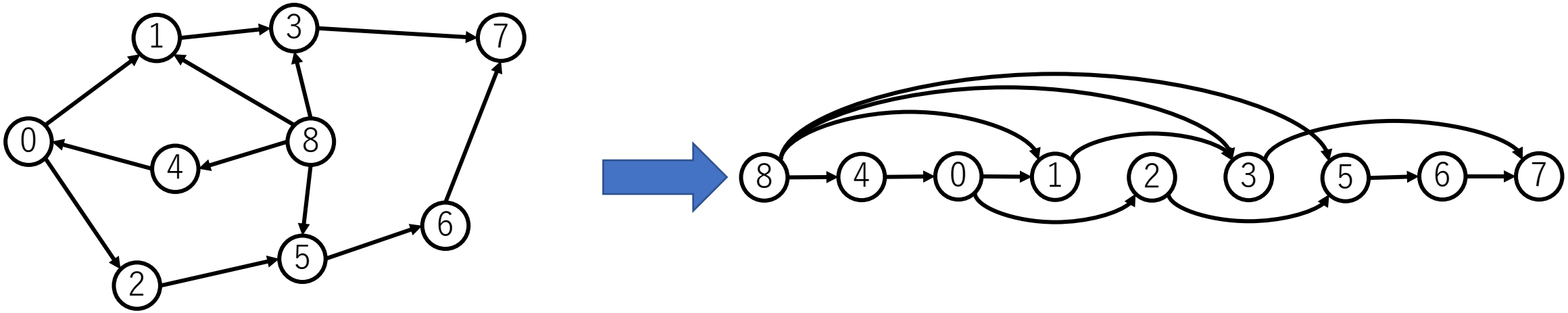
連結なグラフに対しての実装例 (再帰dfs)

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

vector<int> color; //グループ番号を記録。-1は未訪問
//2部グラフだったらtrue
bool isBipartiteGraph(const Graph& G, int s, int cl = 0) { //最初はグループ0
    color[s] = cl;
    for (int v : G[s]) {
        if (color[v] != -1) {
            if (color[v] == cl) { return false; //同じグループだと矛盾する }
            continue; //探索済なので、矛盾しなければ何もしない
        }
        //番号を変更し、再帰的に計算
        if (!isBipartiteGraph(G, v, 1 - cl)) { return false; }
    }
    return true; //矛盾がなければ2部グラフ
}
int main() {
    int N, M;
    cin >> N >> M;
    Graph gr(N + 1);
    color.assign(N + 1, -1);
    for (int i = 0; i < M; i++) {
        int a, b;
        cin >> a >> b;
        gr[a].push_back(b);
        gr[b].push_back(a);
    }
    cout << (isBipartiteGraph(gr, 1) ? "Yes" : "No") << endl;
}
```

- 適当な頂点からグラフ探索
- 頂点を0, 1にグループ分け
  - 0 に繋がっているものは1
  - 1 に繋がっているものは0
- 矛盾するところが現れたら2部グラフではない
- 矛盾しなければ2部グラフ

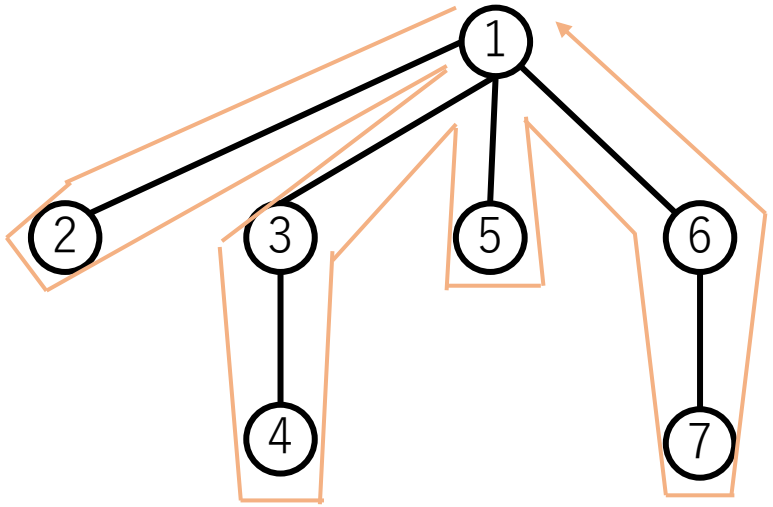
応用例 (3) トポロジカルソート

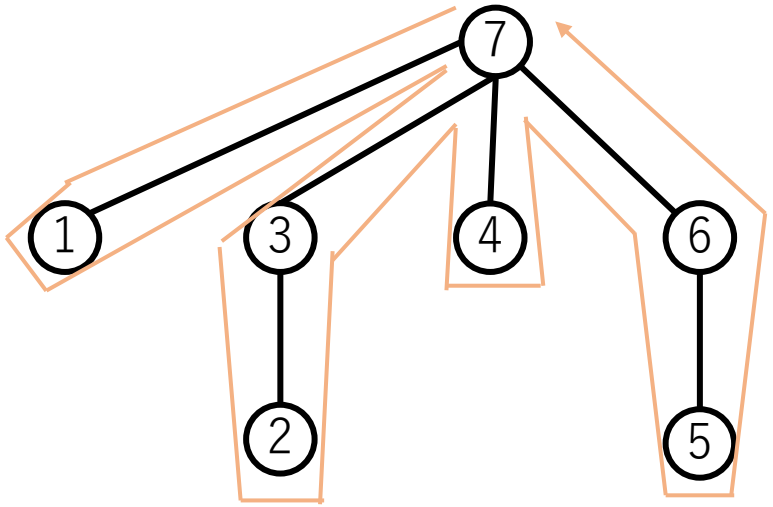
- トポロジカルソート...後ろの頂点から前の頂点への有向辺がないように頂点を並び替えること。
  - DAG (有向サイクルのないグラフ)や木を扱う問題に効果的なことが多い。
- 帰りがけ順に並び替えたものを逆にすればよい。



行きがけ順、帰りがけ順

- 行きがけ順...頂点の探索を**始めた**順番
- 帰りがけ順...頂点の探索を終えて**引き返す**順番
  - DAG であれば、帰りがけ順で頂点に番号がつく時、そこからいける全ての頂点に番号がついている



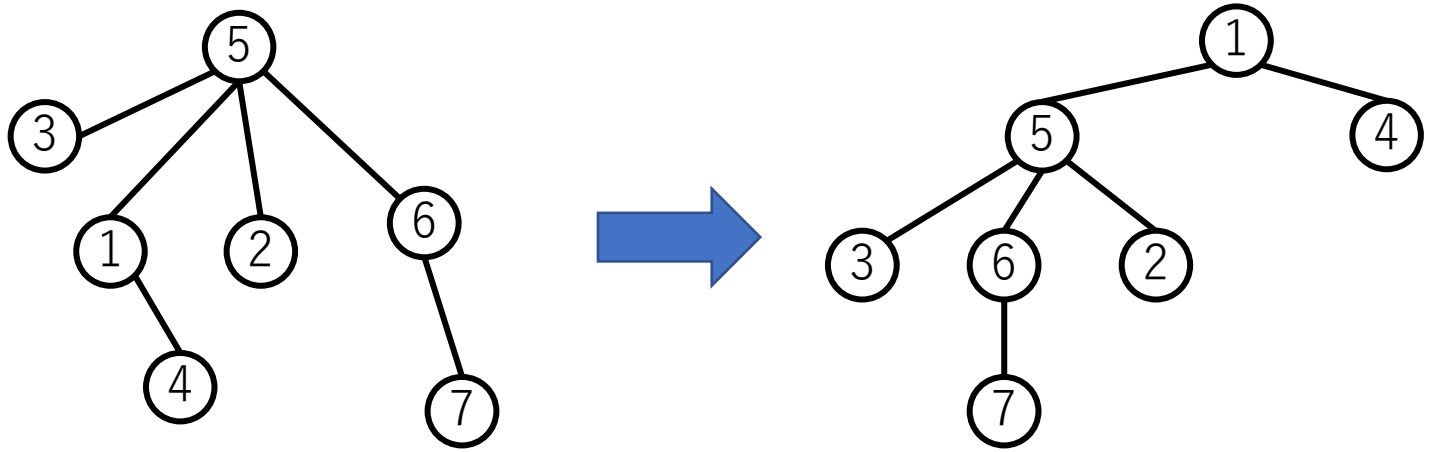


実装例(再帰dfs)

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

void tprSortSub(const Graph& G, int s, vector<int>& tpr, vector<bool>& vis) {
    vis[s] = true;
    for (int v : G[s]) {
        if (vis[v]) { continue; //訪問済みなら何もしない}
        tprSortSub(G, v, tpr, vis);
    }
    tpr.push_back(s);
}

vector<int> tprSort(const Graph& G) {
    int N = G.size();
    vector<bool> vis(N, false);
    vector<int> tpr;
    for (int i = 0; i < N; i++) {
        if (!vis[i]) {
            tprSortSub(G, i, tpr, vis);
        }
    }
    return tpr;
}
```

応用例 (3) 木上のdp

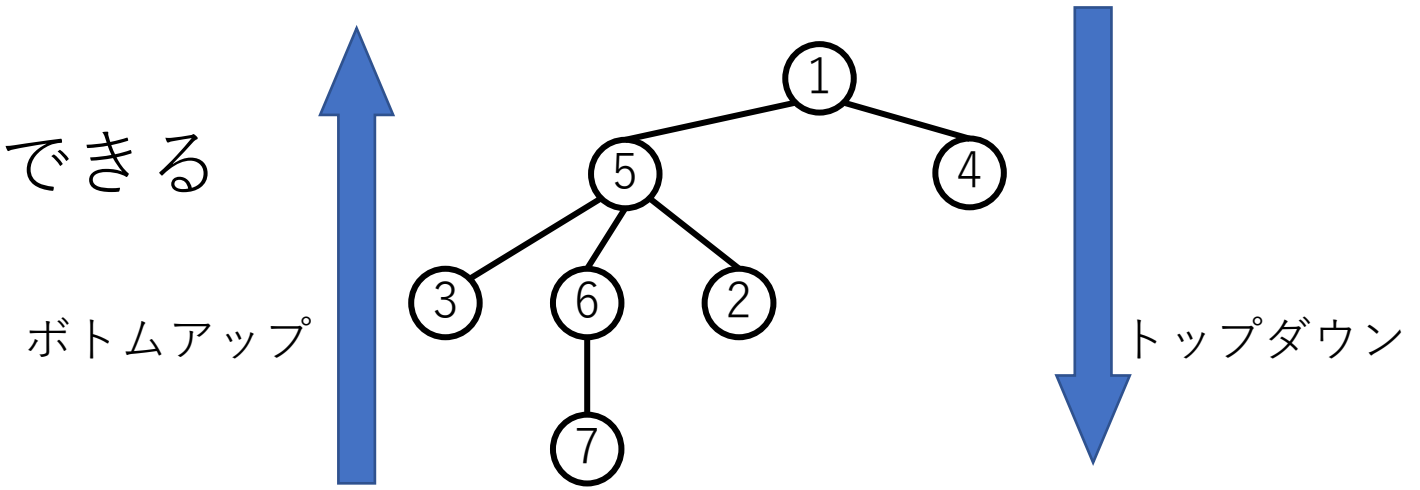
- 木に関する問題を解く時に、根付き木でなくても
  適当な頂点を根とすることで解きやすくなることがある。
- 今回は頂点 1 を根とした時の
  - 各頂点の深さ
  - 各頂点を根とした部分木のサイズ
- を求める。



応用例 (3) 木上のdp

- 頂点 v の深さ ＝ v の親 p の深さ ＋ 1
  - 頂点 1 の深さ ＝ 0
- 頂点 1 からトップダウンに計算できる→行きがけ順

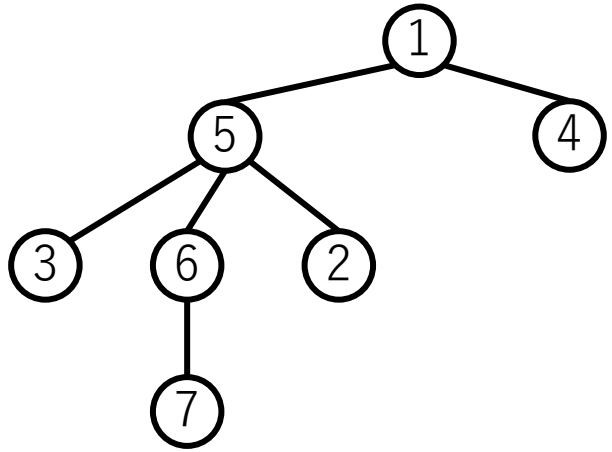
- 頂点 v を根とする部分木の大きさ＝ siz[v] とすると、
- $\mathrm{siz}[v] = 1 + \sum_{c:v\text{の子}} \mathrm{siz}[c]$
  - 1 は v 自体を考慮するため
- 葉からボトムアップに計算できる
  帰りがけ順



深さを求める実装例(再帰dfs)

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

vector<int> depth; //各頂点の深さ
void calcTreeDepth(const Graph& G, int u, int d = 0, int par = -1) {
    depth[u] = d;
    for (int v : G[u]) {
        if (v == par) { continue; //親方向には行かない }
        calcTreeDepth(G, v, d + 1, u);
    }
}
```

- 木にサイクルはないので、親方向に行かなければ訪問済みの頂点を訪れることはない



部分木の大きさを求める実装例 (再帰dfs)

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

vector<int> subtree_size;
void calcSubTreeSize(const Graph& G, int u, int p = -1) {
    subtree_size[u] = 1;
    for (int v : G[u]) {
        if (v == p) { continue; //親方向には行かない }
        calcSubTreeSize(G, v, u);
        subtree_size[u] += subtree_size[v];
    }
}
```

まとめ

- グラフ探索は主に2種類
  - 深さ優先探索(真っ直ぐすすむ)、幅優先探索(広がっていく)
- 深さ優先探索はスタックや再帰関数で実装できる
- 幅優先探索はキューで実装できる

- 全部の頂点を調べるだけならどっちでもOK
- 行きがけ順・帰りがけ順で調べたいなら深さ優先探索
- 最短距離が知りたかったら幅優先探索