I.　グラフの用語整理

グラフはノードとエッジからなるデータ構造です。グラフの用語を整理します。

i.　ツリー (木)

ツリーは閉路を持たない連結なグラフです。ツリーは以下の性質を持ちます。

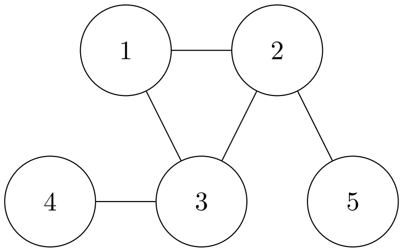
- 連結なグラフである
- 閉路を持たない

ノードの数が n であるグラフ G が木であることは、以下の条件とも同値です。

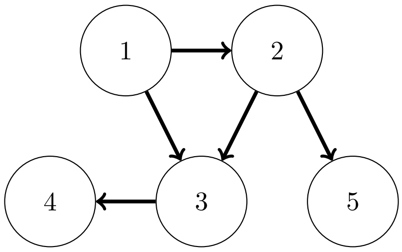
- G には閉路がなく、$n-1$ 本のエッジを持つ
- G は連結であり、$n-1$ 本のエッジを持つ
- G の任意の 2 点を結ぶ経路はただ 1 つ存在する

ii.　無向グラフと有向グラフ

無向グラフはエッジに向きがないグラフです。有向グラフはエッジに向きがあるグラフです。



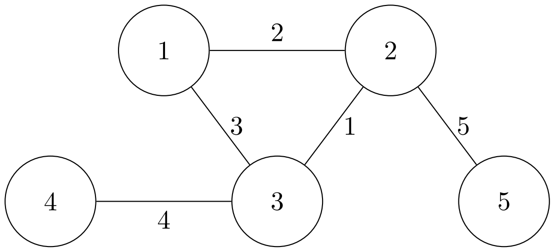
無向グラフ



有向グラフ

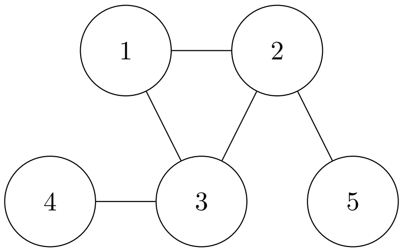
iii.　重み付きグラフ

重み付きグラフはエッジに重みがついたグラフです。重みはエッジのコストや距離を表します。



iv.　隣接行列と隣接リスト

隣接行列と隣接リストはグラフを表現するためのデータ構造です。以下のグラフを例にして、隣接行列と隣接リストを示します。



隣接行列

隣接行列はグラフのエッジを行列で表現したものです。(i, j) 成分が 1 のとき、ノード i とノード j がエッジで結ばれていることを表します。下の図では隣接行列は 1-indexed で表現しています。

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

隣接リスト

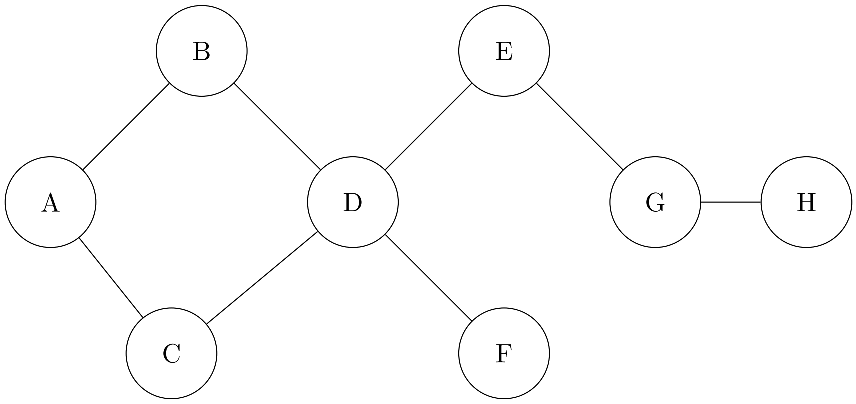
隣接リストは各ノードに隣接するノードをリストで表現したものです。

- 1: 2, 3
- 2: 1, 3, 5
- 3: 1, 2, 4
- 4: 3
- 5: 2

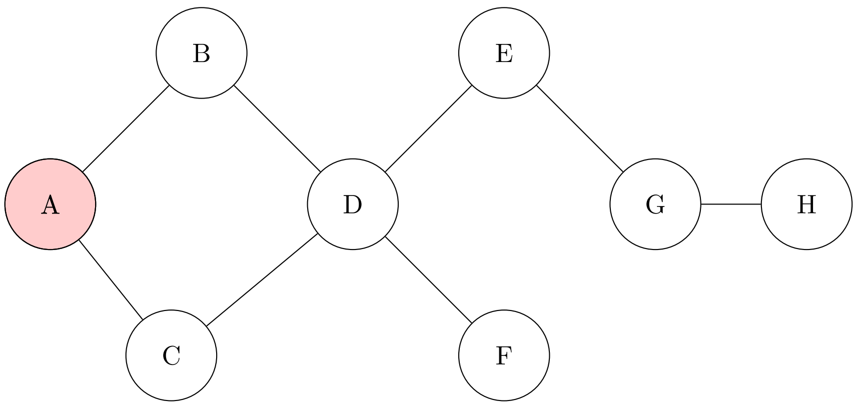
グラフの基本として、深さ優先探索 (DFS) と幅優先探索 (BFS) というグラフの探索アルゴリズムを扱います。

II.　幅優先探索 (BFS)

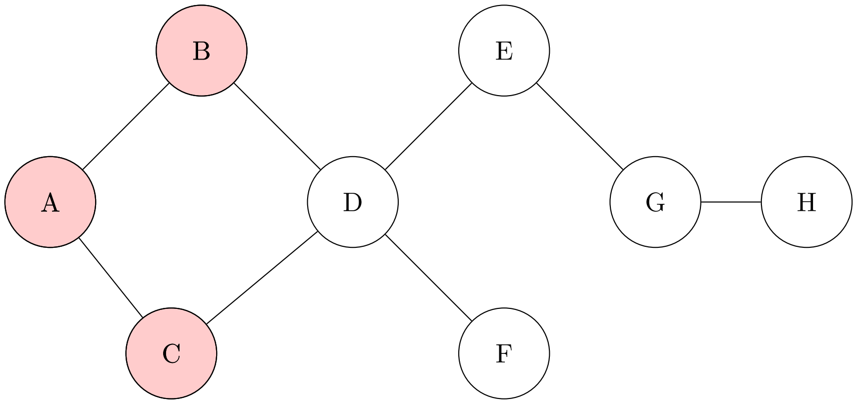
BFS は、後戻りしないように、可能性のあるルートすべてにおいて 1 ステップずつ行くアルゴリズムです。BFS の例を見てみましょう。



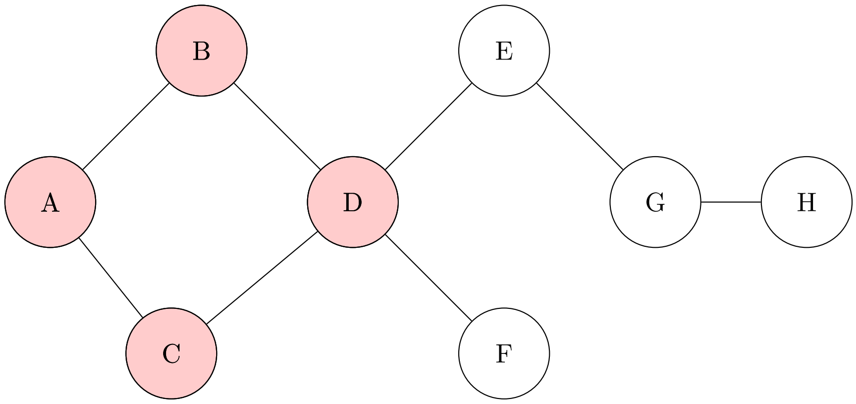
A からスタートします。



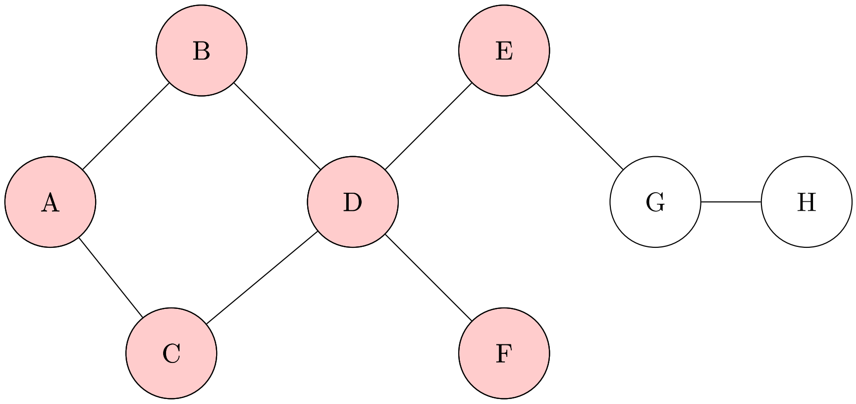
次に A と繋がっているノード B と C を探索します。



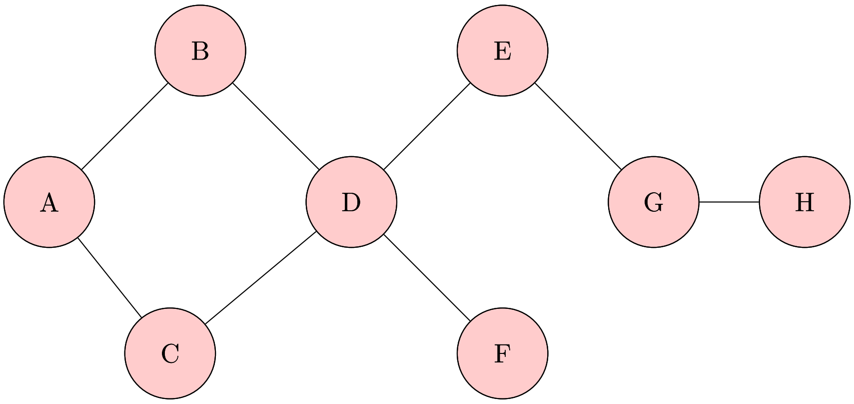
A の探索が終わったので、次に B と C の探索を行います。今回は B から探索します。B と C は同じ深さにあるので、どちらから探索しても問題ありません。B には D が繋がっているので、D を探索します。C から探索を始めようとすると、すでに D はすでに探索済みなので、探索を行いません。



次に D から探索を行います。D には E と F が繋がっているので、E と F を探索します。



最後に G と H を探索します。



これでグラフの探索が終了しました。BFS はスタート地点からの最短距離を求めることができます。

i.　BFS の実装

BFS の実装はキューを用いて行います。実装のポイントは以下の通りです。

- キューを用いて、次に探索するノードを管理する
- 探索済みのノードを管理するために、配列を用いる

隣接リストでも隣接行列でも実装できますが、隣接リストの方が実装が簡単です。また 0-indexed で実装していることに注意してください。

コード 1　深さ優先探索ヒープの実装

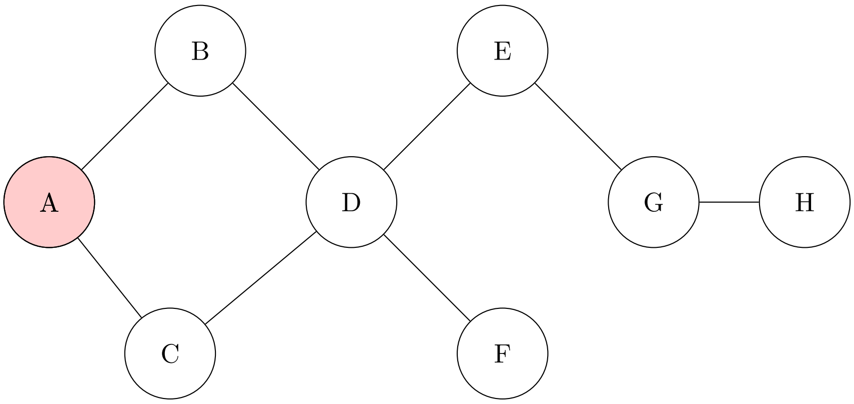
```python
from collections import deque

def bfs(graph: list[list[int]], start: int) -> list[bool]:
    visited = [False] * len(graph)
    todo = deque()

    # スタート地点で初期化
    todo.append(start)

    while todo:
        node = todo.popleft()
        visited[node] = True

        for next_node in graph[node]:
            if not visited[next_node]:
```

```
                todo.append(next_node)

    return visited
```

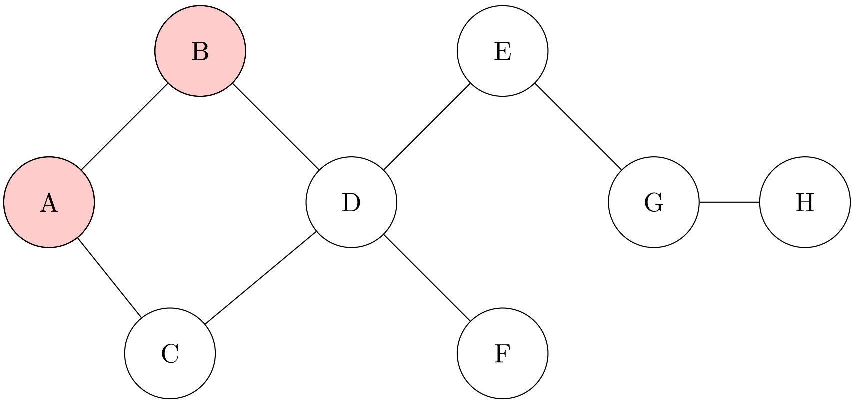
III.　深さ優先探索 (DFS)

DFS は、スタート地点から次のノードに進み、進んだノードに繋がっているノードを行けなくなるまで探索するアルゴリズムです。先ほどのグラフを例にして、DFS の探索を行います。

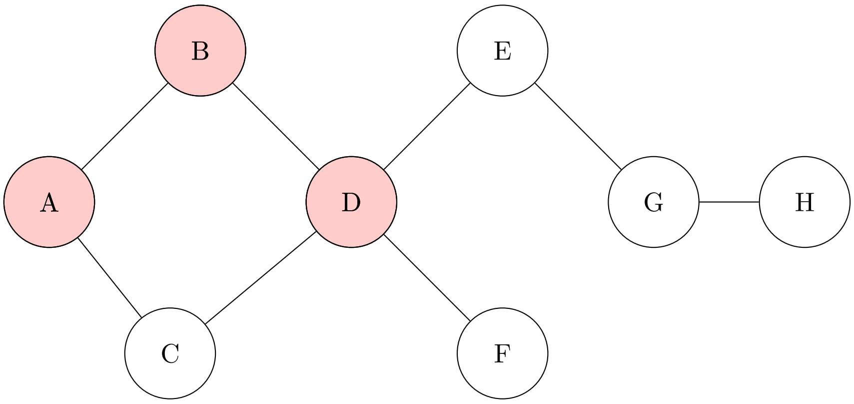
最初は A から探索を行います。



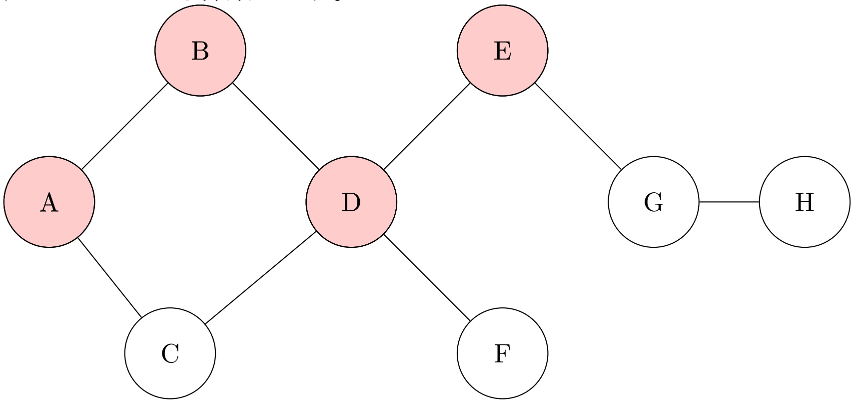
次に A と繋がっているノード B を探索します。



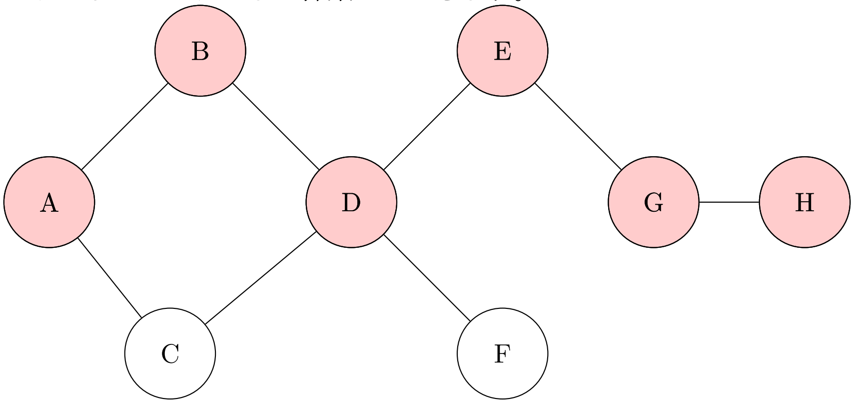
BFS では C を次に探索しますが、DFS では B に繋がっている D を探索します。



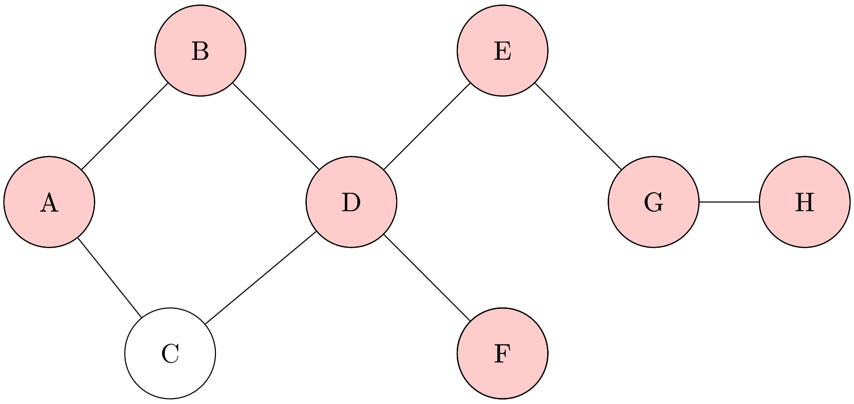
次に D に繋がっている E を探索します。



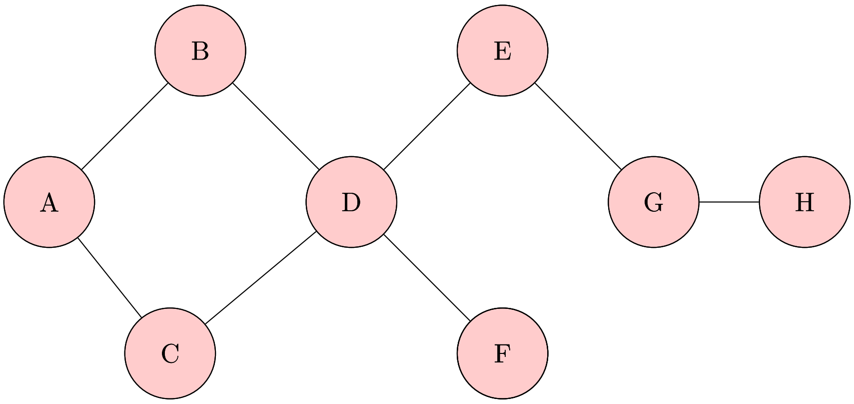
これ以上ノードがないノード H まで探索していきます。



移動する余地の残っている F を探索します。



最後にまだ探索できる A に繋がっている C の探索を行います。



これでグラフの探索が終了しました。DFS は猪突猛進な探索方法で、BFS とは異なり、最短経路を求めることができません。

i.　DFS の実装 (スタック)

DFS の実装はスタックを用いて行います。実装のポイントは以下の通りです。

- スタックを用いて、次に探索するノードを管理する
- 探索済みのノードを管理するために、配列を用いる

隣接リストでも隣接行列でも実装できますが、隣接リストの方が実装が簡単です。また 0-indexed で実装していることに注意してください。DFS との違いは、キューをスタックに変えるだけです。

コード 2　深さ優先探索ヒープの実装

```python
from collections import deque

def dfs(graph: list[list[int]], start: int) -> list[bool]:
    visited = [False] * len(graph)
    todo = deque()

    # スタート地点で初期化
    todo.append(start)

    while todo:
        node = todo.pop()
        visited[node] = True

        for next_node in graph[node]:
```

```
            if not visited[next_node]:
                todo.append(next_node)

    return visited
```

ii. DFS の実装 (再帰)

DFS は再帰を用いて実装することもできます。再帰を用いると、スタックを用いた実装よりも簡潔に実装することができます。

コード 3 深さ優先探索再帰の実装

```
def dfs(graph: list[list[int]], start: int, visited: list[bool]) -> list[
    bool]:
    visited[start] = True
    for next_node in graph[start]:
        if visited[next_node]:
            continue
        else:
            dfs(graph, next_node, visited)

    return visited
```

iii. 問題

問題 1 AtCoder Typical Contest 001 深さ優先探索
DFS を 2 次元グリッドグラフに応用した問題です。DFS や BFS を用いてグラフの到達可能性を調べる問題です。練習なので DFS をスタックと再帰を使った両方で解いてみましょう。再帰で実装する際に Python では再帰の実行回数に制限があるので、sys.setrecursionlimit(10**7) を使って再帰の制限を調整してください。

問題 2 連結成分の個数
グラフの連結成分の個数も DFS を用いることで求められます。すべてのノードを列挙して、DFS を使って到達可能なノードを調べることで連結成分の個数を求めることができます。

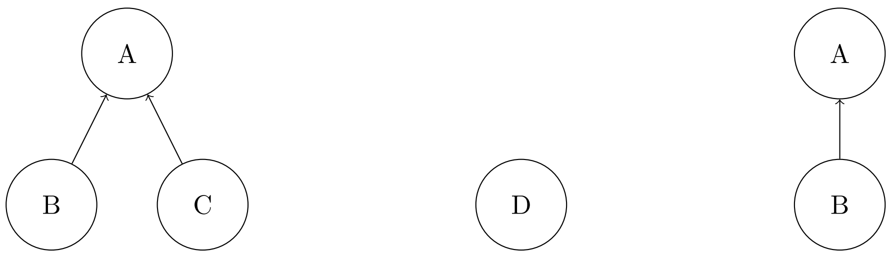
参考

- `https://qiita.com/drken/items/a803d4fc4a727e02f7ba`

IV.　素集合データ構造 (Union-Find 木)

Union-Find 木はノードの集合の連結性を管理するデータ構造です。下の例では、A と D が同じノードにあるかを高速に判定したり、逆に A と D を連結したりする操作を行うことができます。Union-Find 木は以下の操作を行います。

- Union 2 つの集合を結合する
- Find: 2 つのノードが同じ集合に属しているかを判定する



コード 4　Union-Find 木の実装

```
class UnionFind:
  def __init__(self, n: int) -> None:
      self.parent = [i for i in range(n)]
      self.rank = [0] * n

  def _root(self, node: int) -> int:
      if self.parent[node] == node:
          return node
      else:
          # 経路圧縮
          self.parent[node] = self._root(self.parent[node])
          return self.parent[node]

  def unite(self, x: int, y: int) -> None:
      root_x = self._root(x)
      root_y = self._root(y)

      if root_x != root_y:
          if self.rank[root_x] < self.rank[root_y]:
              self.parent[root_x] = root_y
```

```
            else:
                self.parent[root_y] = root_x
                if self.parent[root_x] == self.parent[root_y]:
                    self.rank[root_x] += 1
    def is_same(self, x: int, y: int) -> bool:
        return self.parent[x] == self.parent[y]
```

V.　最短経路問題

BFS を用いた最短経路は上で紹介しましたが、今回はより効率的な最短経路問題の解法を紹介します。DFS では重さが同じのグラフでしか最短経路を求めることができませんが、ダイクストラ法を用いることで重さに異なる重み付きグラフでも最短経路を求めることができます。また、負の重みがあっても最短経路を求めることができるベルマン・フォード法も紹介します。

i.　ダイクストラ法

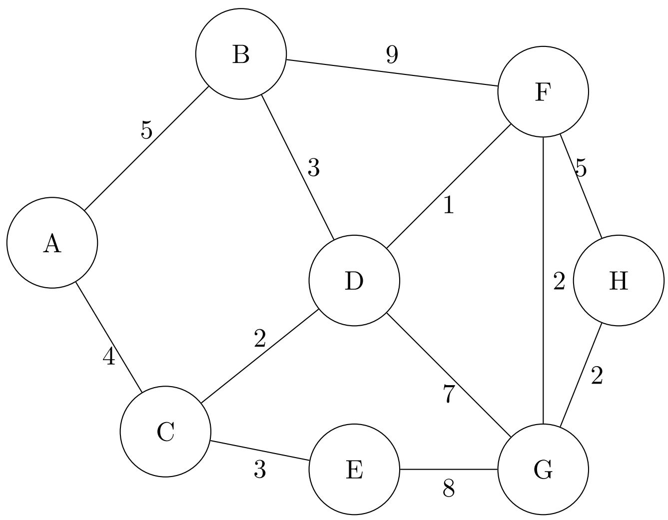
ダイクストラ法を理解する上で重要な重み付きグラフの性質を紹介します。

経路緩和性

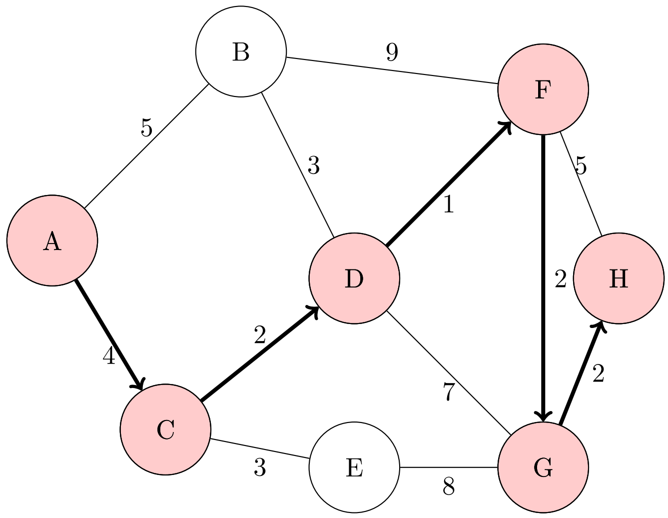
最短経路の部分経路も最短経路である

簡単な証明
P を最短経路とし、その部分経路を Q とする。もしも Q よりも短い経路 R が存在するとすると、R を使った経路の方が P よりも短い経路になるため、P は最短経路ではない。P が最短経路であるという過程に矛盾が生じるため、Q も最短経路である。

具体例を挙げて説明します。以下のグラフを考えます。



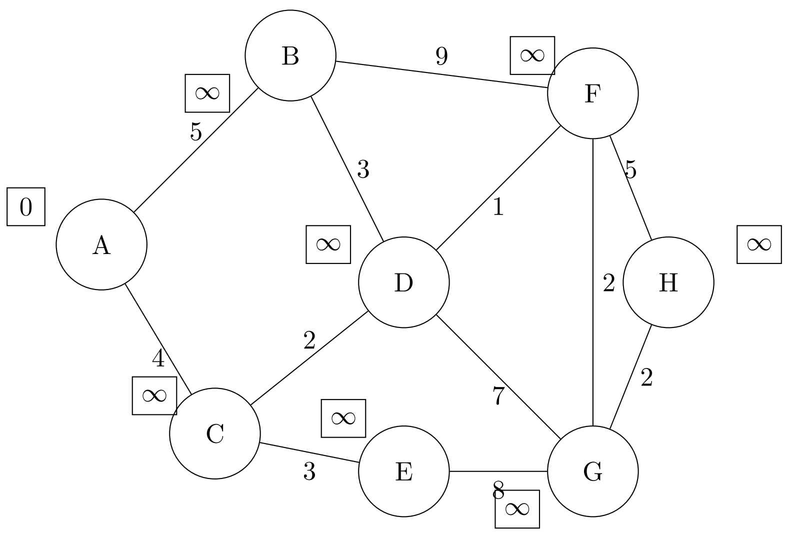
A から H までの最短経路は以下の通りです。もしもゴールが G、F、D、C いずれの場合でも、最短経路は A → C → D → F → G → H の経路になります。



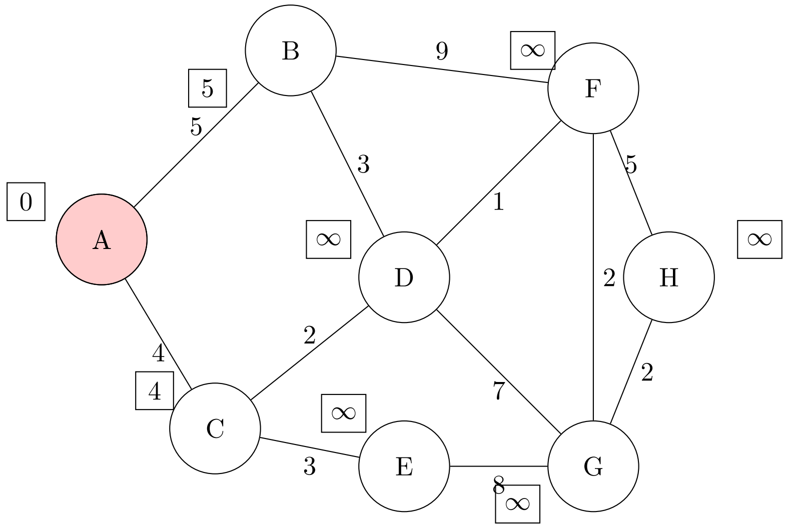
以上の性質より、あるノードまでの最短経路を考えるときはそのノードの前のノードまでの最短経路を考えればよいことがわかります。この性質を利用したのアルゴリズムがダイクストラ法です。ダイクストラ法は以下の手順で最短経路を求めることができます。

1. まだ距離が確定していないノードのうち、最も距離が短いノード x を選択する
2. ノード x に繋がっているノードの距離を更新する
3. 更新が終わるとノード x の距離を確定する
4. すべての頂点が確定するまで 1 から 3 を繰り返す

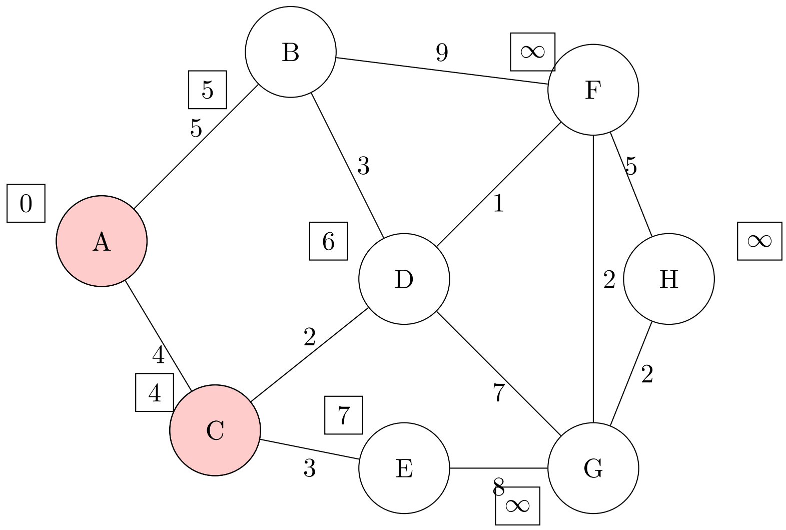
距離は ∞、スタート地点は 0 で初期化します。上のグラフを例にしてダイクストラ法の例を見てみます。最初の距離が確定していないのーどで最も距離が短いノードは A です。A に繋がっているノードの距離を更新します。



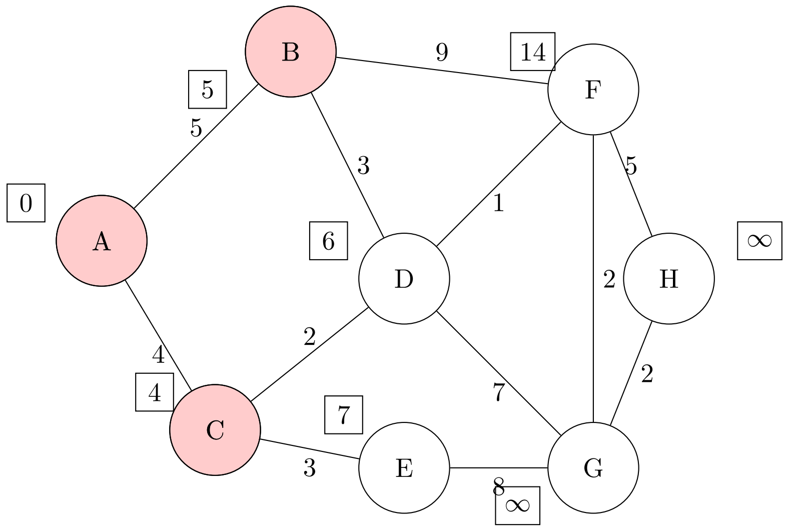
最初の距離が確定していないのーどで最も距離が短いノードは A です。A に繋がっているノードの距離を更新します。更新が終わったので、A の距離を確定します。



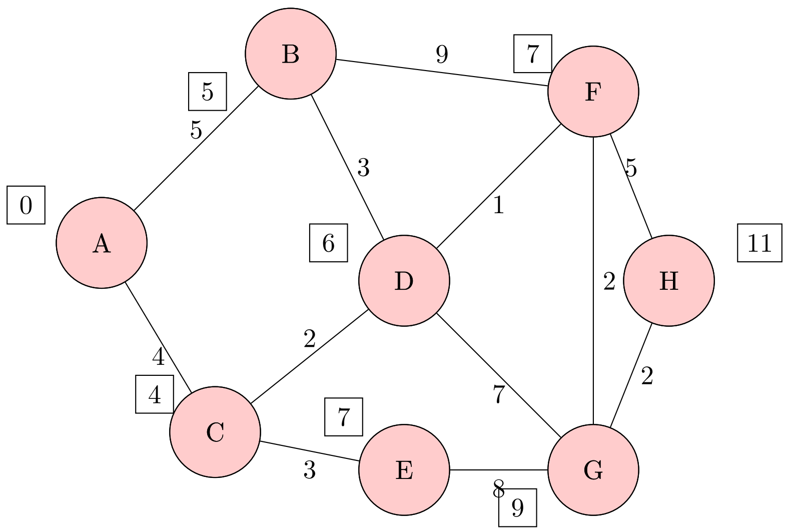
次に距離が確定していないノードで最も距離が短いノードを選択します。今回は C です。C に繋がっているノードの距離を更新します。C に繋がっているノードを更新したら、C の距離を確定します。



次に距離が確定していないノードで最も距離が短いノードを選択します。今回は B です。B に繋がっているノードの距離を更新します。D に関しては、すでにわかっている経路の方が短いので更新しません。B の距離を確定します。



以下同様に更新すると、最終的に以下のような結果になります。



1.　ダイクストラ法の実装

ダイクストラ法はとてもシンプルなアルゴリズムです。ダイクストラ法の実装のポイントは以下の通りです。

- 最短距離を格納する配列を用意する
- まだ確定していないノードのうち、最も距離が短いノードを選択する。最も短いノードを $O(\log n)$ で取得するためにヒープを使う
- 選択したノードに繋がっているノードの距離を前回の距離と比較して更新する
- すべてのノードが確定するまで繰り返す

Python の標準ライブラリのヒープは tuple などを渡す index が早い要素から比較してくれるので、ダイクストラ法の実装に適しています。(移動距離、ノード) の tuple でヒープに追加することで、最短距離が短いノードを取得することができます。もちろん自分で実装したヒープを使っても問題ありません。

与えられる重み付きグラフは (終点、重み) の形式で隣接リストで与えられるとします。以下にダイクストラ法の実装を示します。

コード 5　ダイクストラ法の実装

```python
from heapq import heappop, heappush

def dijkstra(graph: list[list[int, int]], start: int) -> list[int]:
    """
    args:
      graph: graph[i]は[(j, cost)]でiからjへのコストがcostであることを示す
      start: 始点のノード番号
    """
    done = [False] * len(graph)
    dist = [1 << 60] * len(graph)
    todo = []

    # 初期化
    dist[start] = 0
    heappush(todo, [dist[start], start])

    while todo:
        prev_distance, node = heappop(todo)

        if done[node]:
            continue

        for next_node, next_distance in graph[node]:
            if prev_distance + next_distance < dist[next_node]:
                dist[next_node] = prev_distance + next_distance
                heappush(todo, [dist[next_node], next_node])

        done[node] = True
```

```

    return dist
```

ii.　ベルマン・フォード法

ベルマン・フォード法はダイクストラ法と異なり、負の重みを持つ辺があっても機能する単一始点全点間最短路を求めるアルゴリズムです。負の閉路の検出も可能です。ダイクストラ法とは違って最短距離の選択を行わずに、毎回すべての辺を更新します。

ベルマン・フォード法のアルゴリズムを例を使って説明します。以下のグラフを例に考えます。概要はダイクストラ法とあまり変わりません。ただし、グラフの情報が (始点、終点、重み) の list で与えられるとします。例えば、(A, B, 5) は A から B へでている重みが 5 のノードであることを示します。

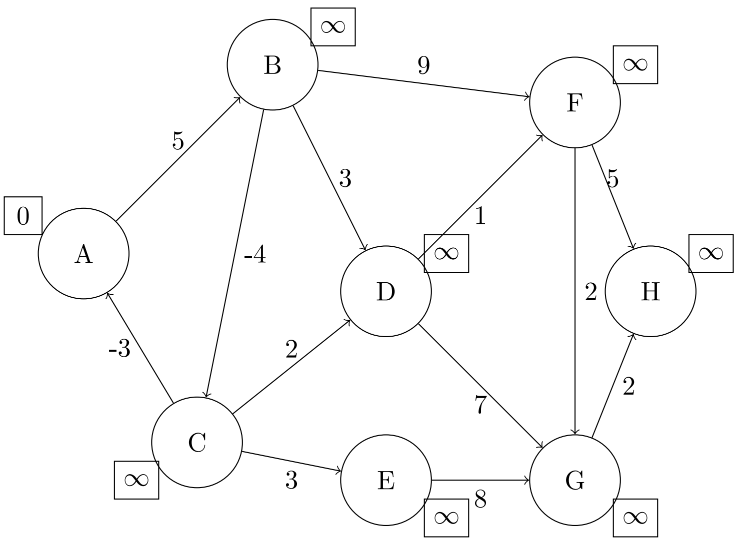
すべて列挙すると、以下の様になります。

$$\begin{aligned}\text{edges} =&(A,B,5),(B,C,-4),(C,A,-3),(C,D,2),(B,D,3),(B,F,9),(F,H,5),\\&(C,E,3),(E,G,8),(D,G,7),(D,F,1),(G,H,2),(F,G,2)\end{aligned}$$

初期化として始点の距離は 0, それ以外は ∞ とします。ベルマン・フォード法は以下のアルゴリズムに従っています。

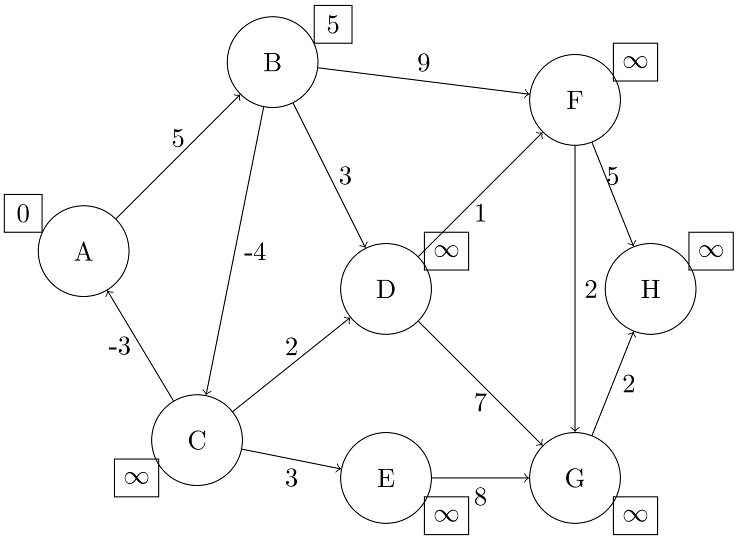
1. edges をすべて列挙し、始点から終点への重みを更新する
2. 1 を V - 1 回繰り返す (V はノードの数)

V-1 回の更新で、最短距離が確定します。もし、V 回目にも更新がある場合は負の閉路が存在することになります。最短経路を求めるのに V - 1 回の更新で十分な理由は、経路緩和性によって開始地点から終点が最も遠い場合でも、V - 1 回の更新で確定するからです。

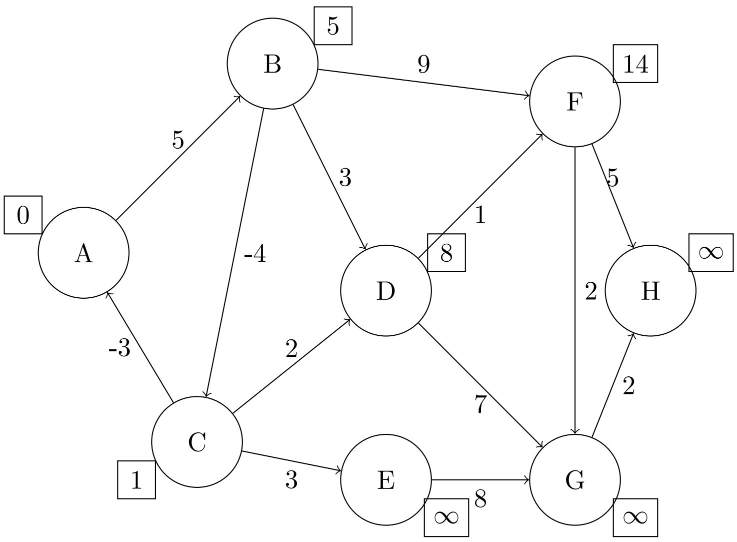


ベルマン・フォード法の流れを確認します。最初は A と繋がっている B が更新されます。他のノードも edges をすべて列挙して更新を図りますが、$\infty+$ 有限の値と考えるため、更新されないと

みなすことができます。



2 回目の更新です。



以下のように更新を繰り返すと、最終的に以下のような結果になります。

コード 6　ベルマン・フォード法の実装

```
def bellman_ford(v: int, edges: list[tuple[int, int, int]]) -> list[int]
    | int:
    # 初期化
    dist = [1 << 60] * v
    dist[0] = 0
```

```python

for _ in range(v - 1):
    for start, end, weight in edges:
        if dist[start] != 1 << 60 and dist[start] + weight < dist[end]:
            dist[end] = dist[start] + weight

# 一度v-1回更新した後に負の閉路を検出
for start, end, weight in edges:
    if dist[start] != 1 << 60 and dist[start] + weight < dist[end]:
        return -1

return dist
```

参考

- https://qiita.com/ko-ya346/items/359a3e03c5e20b04c573

iii.　SPFA(Shortest Path Faster Algorithm)

SPFA はベルマン・フォード法を高速化したアルゴリズムです。基本はベルマン・フォード法と同じですが、毎回すべての辺をチェックすること防ぐ工夫がされています。あるノード x が更新されなければ、そのノードに繋がっている他のノードの距離も更新されません。実装上は更新が必要なノードが出てきたらそれを queue に入れ、queue がからになるまで処理を続けます。

SPFA 実装の実装は以下の様になります。

コード 7　SPFA の実装

```python
from collections import deque

def spfa(v: int, edges: list[list[tuple[int, int]]]):
      inf = 1 << 60
      dist = [inf] * v
      dist[0] = 0

      node_to_check = deque()
      in_queue = [False] * v

      while node_to_check:
          current_node = node_to_check.popleft()
          in_queue[current_node] = False

          for end, weight in edges[current_node]:
              if dist[current_node] + weight < dist[end]:
                  dist[end] = dist[current_node] + weight

              if not in_queue[end]:
                  in_queue[end] = True
                  node_to_check.append(end)

      return dist
```

iv. ワーシャルフロイド法

ワーシャルフロイド法は全点対最短経路問題を解くアルゴリズムです。負の閉路がない限り負のエッジがあっても最短経路を求めることができます。ワーシャルフロイド法はコードを見た方が理解しやすいと思うので、以下に実装を示します。

コード 8 ワーシャルフロイド法の実装

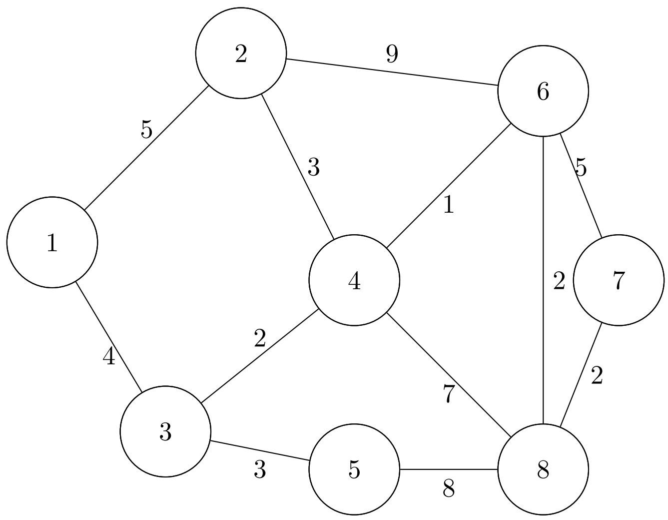
```python
def warshall_floyd(n: int, dist: list[list[int]]):
  for k in range(n):
    for i in range(n):
      for j in range(n):
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

ワーシャルフロイド法は 3 重ループのそれぞれがグラフの何に対応しているかを理解することが重要です。対応は以下のようになっています。

- k: 経由するノード
- i: 始点
- j: 終点

始点と終点を決めて、その間に経由するノードを変えていくことで、最短経路を求めることができます。dist は隣接行列で与えられるとします。

理解を深めるために、例を見てみましょう。



上のグラフの隣接行列 dist は以下のようになっています。無向グラフなので直接繋がっているノード同士以外は無限大で表されています。

	1	2	3	4	5	6	7	8
1	0	5	4	∞	∞	∞	∞	∞
2	5	0	∞	3	∞	9	∞	∞
3	4	∞	0	2	3	∞	∞	∞
4	∞	3	2	0	∞	1	7	∞
5	∞	∞	3	∞	0	∞	8	∞
6	∞	9	∞	1	∞	0	2	5
7	∞	∞	∞	7	8	2	0	2
8	∞	∞	∞	∞	∞	5	2	0

$\mathrm{k} = 1, \mathrm{i} = 1, \mathrm{j} = 1 \cdots 8$ の場合を考えましょう。1 からスタートして $1 \cdots 8$ に行く経路のうち 1 を経由する経路を考えます。

- $\mathrm{j} = 1$: $\mathrm{dist}[1][1] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][1])$
- $\mathrm{j} = 2$: $\mathrm{dist}[1][2] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][2], \mathrm{dist}[1][2]) = \min(5, \infty)$
- $\mathrm{j} = 3$: $\mathrm{dist}[1][3] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][3], \mathrm{dist}[1][3]) = \min(4, \infty)$
- $\mathrm{j} = 4$: $\mathrm{dist}[1][4] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][4], \mathrm{dist}[1][4]) = \min(\infty, \infty)$
- $\mathrm{j} = 5$: $\mathrm{dist}[1][5] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][5], \mathrm{dist}[1][5]) = \min(\infty, \infty)$
- $\mathrm{j} = 6$: $\mathrm{dist}[1][6] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][6], \mathrm{dist}[1][6]) = \min(\infty, \infty)$
- $\mathrm{j} = 7$: $\mathrm{dist}[1][7] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][7], \mathrm{dist}[1][7]) = \min(\infty, \infty)$
- $\mathrm{j} = 8$: $\mathrm{dist}[1][8] = \min(\mathrm{dist}[1][1] + \mathrm{dist}[1][8], \mathrm{dist}[1][8]) = \min(\infty, \infty)$

以上を反映した dist は以下のようになります。

	1	2	3	4	5	6	7	8
1	0	5	4	∞	∞	∞	∞	∞
2	5	0	∞	3	∞	9	∞	∞
3	4	∞	0	2	3	∞	∞	∞
4	∞	3	2	0	∞	1	7	∞
5	∞	∞	3	∞	0	∞	8	∞
6	∞	9	∞	1	∞	0	2	5
7	∞	∞	∞	7	8	2	0	2
8	∞	∞	∞	∞	∞	5	2	0

1 に繋がっていたノードへの最短距離が更新されました。以下同様に繰り返しを行うと最終的には以下のようになります。

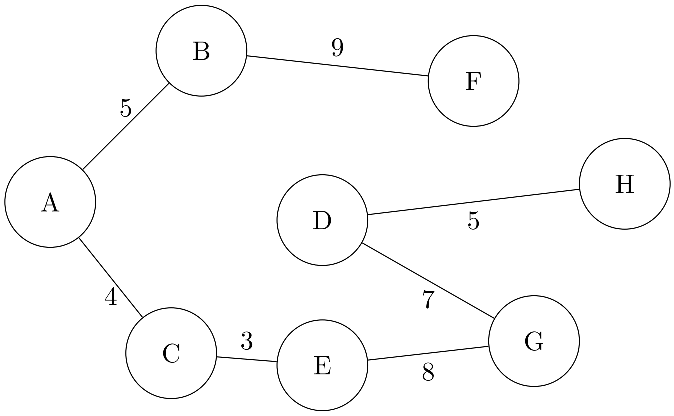
	1	2	3	4	5	6	7	8
1	0	5	4	6	7	7	9	11
2	5	0	5	3	8	4	6	8
3	4	5	0	2	3	3	5	7
4	6	3	2	0	5	1	3	5
5	7	8	3	5	0	6	8	10
6	7	4	3	1	6	0	2	4
7	9	6	5	3	8	2	0	2
8	11	8	7	5	10	4	2	0

v.　問題

問題 1 AtCoder Beginner Contest 051 D - Candidates of No Shortest Paths 経路を復元する問題です。ダイクストラ法でも扱いましたが、ワーシャルフロイド法でも解くことができます。

VI.　最小全域木

全域木とは、すべてのノードが繋がっている木のことをいいます。また**最小全域木**とは、全域木の中で重さが最小になるもののことをいいます。

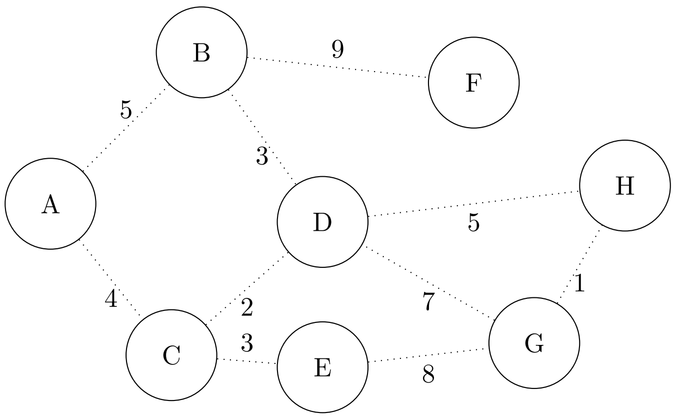


最小全域木を求めるには、辺ベースのアプローチとノードベースのアプローチの 2 種類があります。それぞれ**クラスカル法**、**プリム法**と呼ばれています。

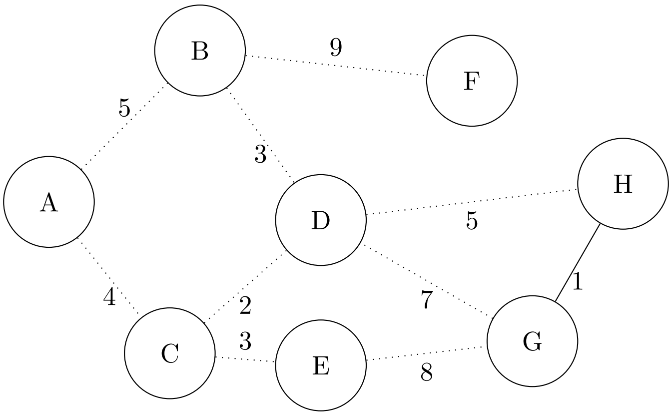
i.　クラスカル法

クラスカル法は、存在する辺を重さが小さい順に並べて入れていき、閉路ができないことが確認できた場合は追加し、すべての辺をチェックし終えたら終了する貪欲法です。以下のアルゴリズムに従っています。

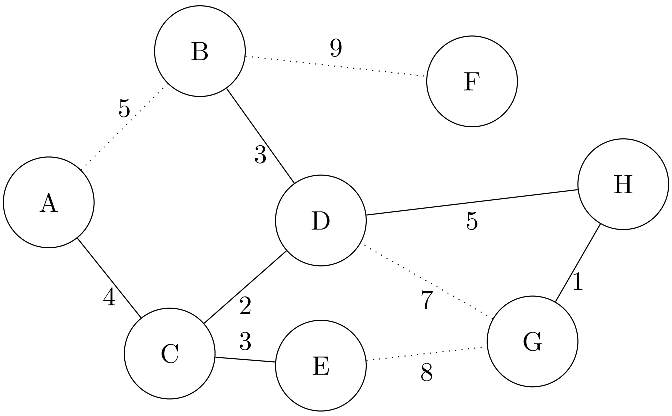
1. すべての辺を重さの小さい順にソート
2. 重さの最も小さい辺を選ぶ
3. 今までに選んだ辺から構成される木に 2 で選んだ辺 BD を追加した時に、閉路が生まれないことを確認する。閉路が新しくできないならこの辺を追加する
4. すべての辺をチェックし終えるまで 2 から 3 を繰り返す



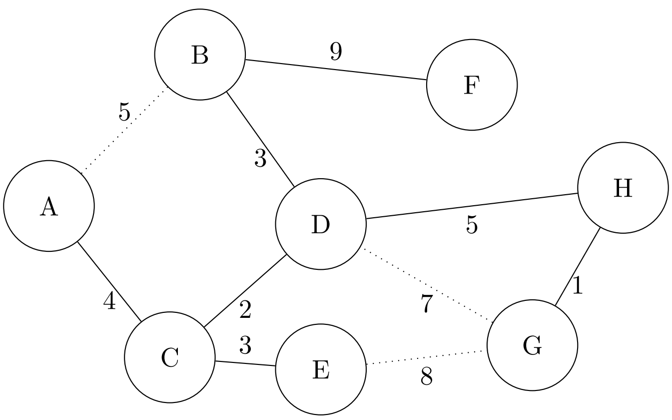
クラスカル法の例を見てみましょう。まずは、すべての辺の中で最も重さが小さい辺を選んでそれを木に追加したときに、閉路ができないのでその辺を追加します。



同様に、CD、CE、BD、AC、DH を順に追加しても閉路はできないので、追加します。



しかし、AB、DG、EG を繋げると閉路ができてしまうので追加しません。最後に BF を追加するとすべての辺をチェックし終えたので以下のような最小全域木ができあがります。



クラスカル法の実装上のポイントは以下の 2 つです。

- すべての辺を重さの小さい順にソートする
- 辺を追加したときに閉路かどうかを判定する

最初のポイントは、辺を重さの小さい順にソートすることで、簡単に実装できます。2 つ目のポイントは、BFS や DFS を使っても実装できますが、効率が良くありません。そこで、Union-Find 木を使って実装することが一般的です。

ある辺をグラフに木に挿入しようとしたとき、辺を作る 2 点が同じ集合に属している場合に辺を繋げると閉路ができてしまいます。つまり、Union-Find を使ってふたつのノードが同じ集合に属しているかを判断します。

コード 9　クラスカル法の実装

```python
class UnionFind:
  def __init__(self, n: int) -> None:
      self.parent = [i for i in range(n)]
      self.rank = [0] * n

  def _root(self, node: int) -> int:
      if self.parent[node] == node:
          return node
      else:
          # 経路圧縮
          self.parent[node] = self._root(self.parent[node])
          return self.parent[node]

  def unite(self, x: int, y: int) -> None:
      root_x = self._root(x)
      root_y = self._root(y)

      if root_x != root_y:
          if self.rank[root_x] < self.rank[root_y]:
              self.parent[root_x] = root_y
          else:
              self.parent[root_y] = root_x
              if self.rank[root_x] == self.rank[root_y]:
                  self.rank[root_x] += 1

```

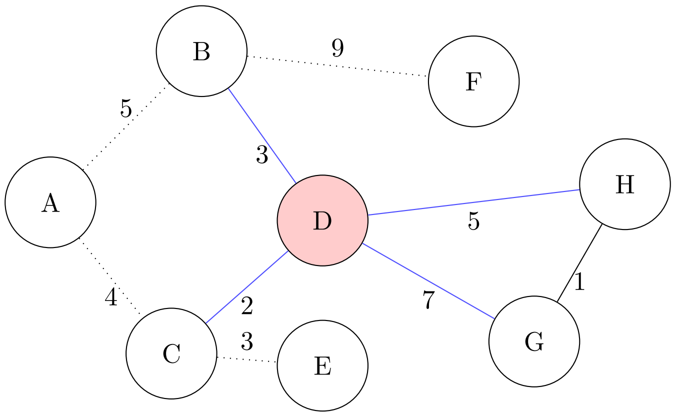
```python
    def is_same(self, x: int, y: int) -> bool:
        return self.parent[x] == self.parent[y]

def kruskal(v: intedges: list[tuple[int, int, int]]) -> list[list[int,
    int]]:
    """
    args:
        v: node size
        edges: (start, end, weight)
    """
    sorted_edge_costs = []
    for edge in edges:
        sorted_edge_costs.append([edges[2], edges[0], edges[1]])

    sorted_edge_costs.sort()

    uf_tree = UnionFind(v)

    minimum_spanning_tree = []

    for weight, start, end in sorted_edge_costs:
        if not uf_tree.is_same(start, end):
            uf_tree.unite(start, end)
            minimum_spanning_tree.append([start, end])

    return minimum_spanning_tree
```

ii.　プリム法

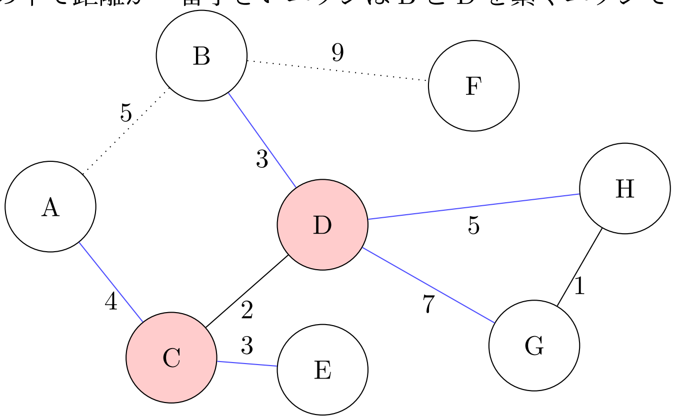
プリム法とは、すでに到達した頂点の集合からまだ到達していない頂点の集合への辺のうち、距離が最短のものを追加し、すべてのノードがつながったら終了するアルゴリズムです。プリム法は以下のアルゴリズムに従っています。

1. 任意のノードを選び、訪問済みにする
2. そのノードに繋がっているすべての辺を最小全域木の候補の辺として追加する
3. 最小全域木の候補の辺の中から、接続先のノードが未訪問である最短の距離の辺を選ぶ
4. 選んだ辺を最小全域木に入れ、その接続先のノードを訪問済にする。
5. 4 で新しく訪問したノードから、さらにその先に繋がっている辺のうち、接続先のノードが未訪問のすべての辺を最小全域木の候補に追加する
6. すべてのノードが訪問済になるまで 2 から 4 を繰り返す

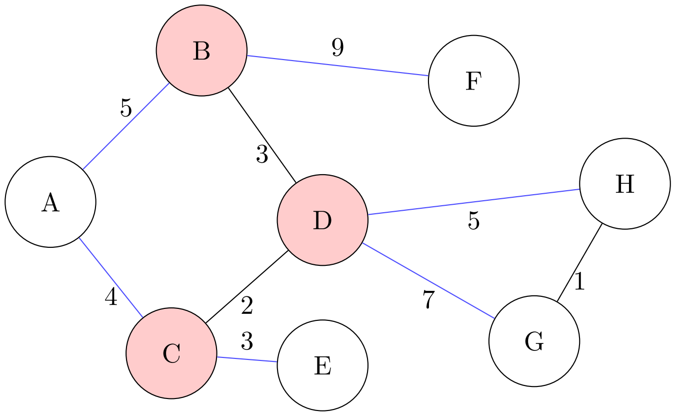
プリム法の具体例を見てみましょう。まずは、任意にノードを選び、訪問済にします。ここでは D を選びます。D から繋がっているエッジを青色で示します。その中で最も距離が短いエッジは C と D を繋ぐエッジです。



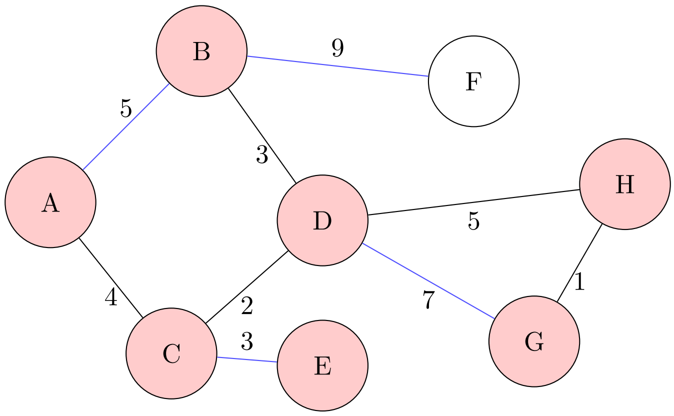
C は未訪問なので、C を訪問済にして、C から繋がっているエッジを最小全域木の候補に追加します。青色のエッジの中で距離が一番小さいエッジは B と D を繋ぐエッジです。



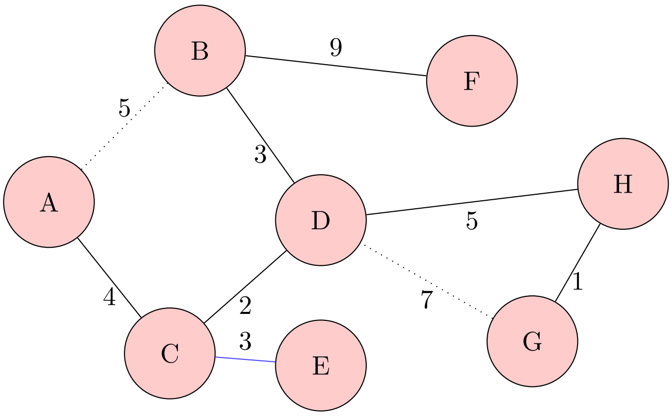
B は未訪問なので、B を訪問済にして、B から繋がっているエッジを最小全域木の候補に追加します。青色のエッジの中で最も短いエッジは C と E を繋ぐエッジです。



同様にして、A と C を結ぶエッジ、D と H を結ぶエッジ、G と H を結ぶエッジを追加します。



残りは、A と B を結ぶエッジ、D と G を結ぶエッジがありますが、A と B を結ぶエッジと D と G を結ぶエッジを追加すると閉路ができてしまうので追加しません。最後に D と G を結ぶエッジを追加すると、すべてのノードがつながったので終了です。以下のような最小全域木ができあがります。



プリム法の実装例を以下に示します。

コード 10　プリム法の実装

```
import heapq

def prim(v: int, edges: list[list[int, int, int]]):
      edges_from = [[] for _ in range(v)]

      for start, end, weight in edges:
          edges_from[start].append([weight, start, end])
          edges_from[end].append([weight, end, start])  # 無向グラフのため逆方向のエッジも追加

      edge_heap = []
      minimum_spanning_tree = []
      included = [False] * v

      # ノード 0 を選択して開始
      included[0] = True
      for edge in edges_from[0]:
          heapq.heappush(edge_heap, edge)

      while edge_heap:
          weight, start, end = heapq.heappop(edge_heap)
          if not included[end]:
              included[end] = True
              minimum_spanning_tree.append([start, end])
```

```python

                for edge in edges_from[end]:
                    if not included[edge[2]]:
                        heapq.heappush(edge_heap, edge)

    return minimum_spanning_tree
```

iii.　問題

問題 1 AtCoder 典型アルゴリズム問題集 F - 最小全域木問題

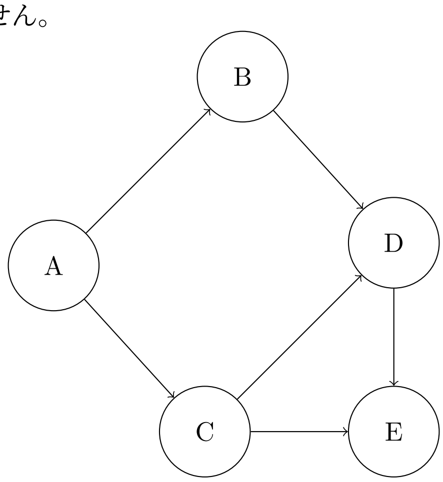
参考

- `https://dai1741.github.io/maximum-algo-2012/docs/minimum-spanning-tree/`

VII. トポロジカルソート

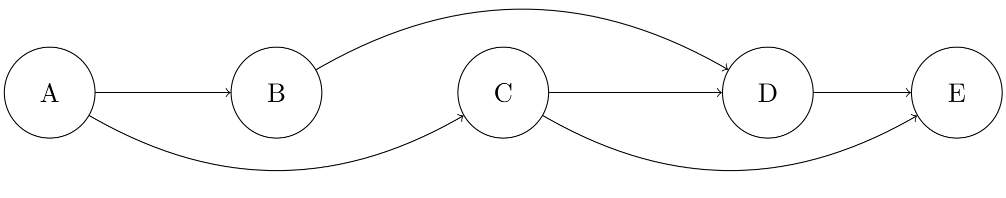
DAG という性質を持ったグラフをソートするアルゴリズムであるトポロジカルソートを扱います。**DAG**(Directed Acyclic Graph) とは、その名の通り有向グラフで閉路を持たないグラフのことを指します。

以下に DAG の例を示します。有向グラフで閉路がないため、どこのノードから辿っても元の位置に戻ってくることはできません。



DAG の例

トポロジカルソートはすべての辺が同じ方向を向くようにノードをソートするアルゴリズムです。上の DAG をトポロジカルソートをすると、以下のようになります。有向辺がすべて右向きになっていることがわかります。注意点として、トポロジカルソートの結果は一意ではありません。



有向グラフの性質をより理解するために**次数、入次数、出次数**という用語を導入します。次数とは、ノードにつながっている辺の数を指します。入次数とは、ノードに入ってくる辺の数を指し、出次数とは、ノードから出ていく辺の数を指します。次数 $=$ 入次数 $+$ 出次数 です。DAG の性質として、必ず入次数 0 のノードが存在します。

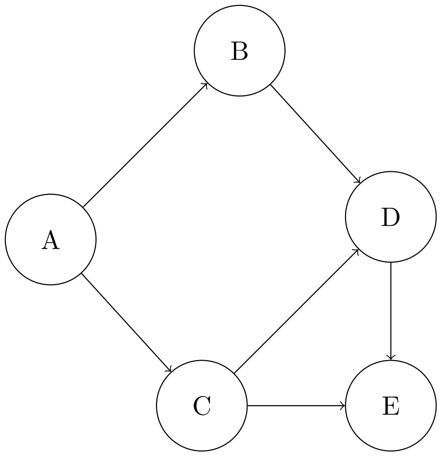
トポロジカルソートの代表的なアルゴリズムとして、Kahn のアルゴリズムと Tarjan のアルゴリズムを紹介します。

i. Kahn のアルゴリズム

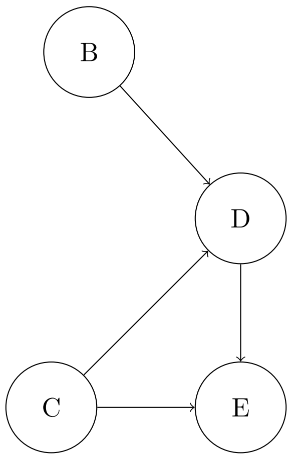
Kahn のアルゴリズムは、トポロジカルソートを行うアルゴリズムの一つです。アルゴリズムの流れは以下の通りです。

1. 入次数が 0 のノードをグラフから取り除き、ソート済配列に追加する
2. 1 を入次数 0 のノードがなくなるまで繰り返す

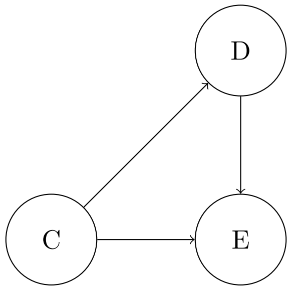
Kahn のアルゴリズムの例を以下に示します。



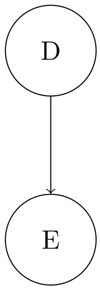
入次数 0 のノード A を取り除き、A に繋がっているノードの次数を更新します。



入次数 0 のノード B を取り除きます。



入次数 0 のノード C を取り除きます。最後にノード D と E を順に取り出せばトポロジカルソートは完成です。



Kahn のアルゴリズムの実装例を以下に示します。

コード 11　Kahn のアルゴリズムの実装

```python
from collections import deque

def topo_sort(v: int, edges: list[list[int, int]]) -> list[int] | None:
    e = len(edges)
    indeg = [0] * v
    outedge = [[] for _ in range(v)]

    for start, end in edges:
        indeg[end] += 1
        outedge[start].append[end]

    sorted_g = [i for i in range(v) if indeg[i] == 0]
    deq = deque(sorted_g)

    while deq:
        node = deq.popleft()
        for connected_node in outedge[node]:
```

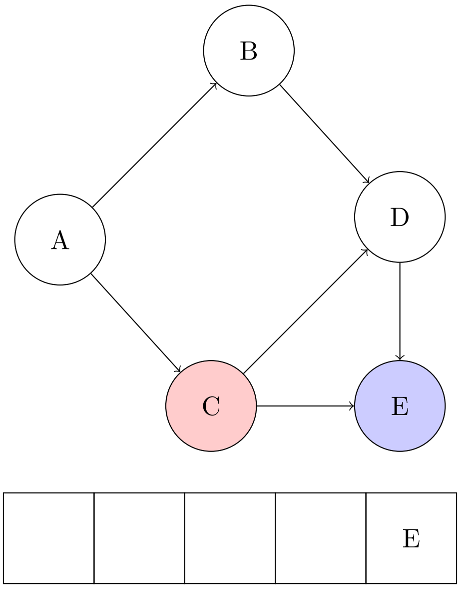
```
                e -= 1
                indeg[connected_node] -= 1
                if indeg[connected_node] == 0:
                    sorted_g.append(connected_node)
                    deq.append(connected_node)

    if e != 0:
        return None

    return sorted_g
```

ii. Tarjan のアルゴリズム

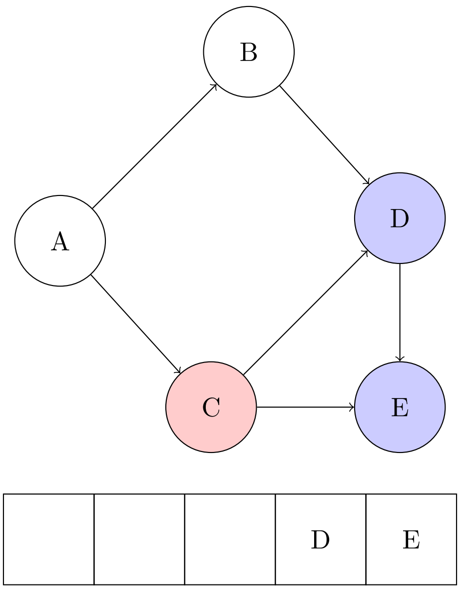
Tarjan のアルゴリズムは、深さ優先探索を用いてトポロジカルソートを行うアルゴリズムです。アルゴリズムの流れは以下の通りです。

1. 未訪問のノードを訪問済にする
2. そのノードに繋がっているノードを再帰的に訪問する
3. そのノードのすべての子ノードを訪問し終えたら、そのノードをソート済配列に先頭から追加する

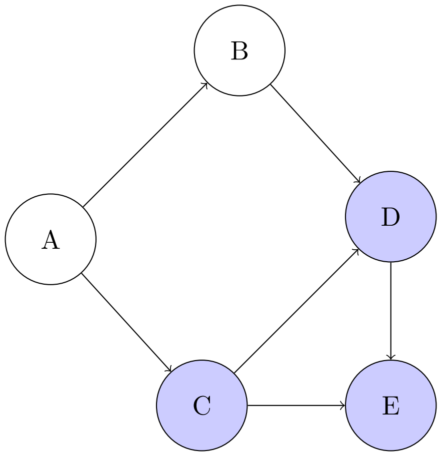
Tarjan のアルゴリズムの例を以下に示します。最初は任意のノードを選びます。今回は C を選びます。C から DFS を開始すると、E を探索します。E からはもう子ノードがないので、E をソート済配列に追加します。



C から D にも移動でき、D からはこれ以上移動することができないので、D をソート済配列に追加します。

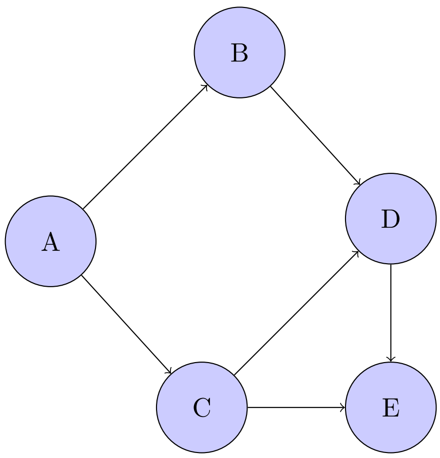


C からはもう移動できないので、C をソート済配列に追加します。



		C	D	E

最後に A から B への移動を考えると、トポロジカルソートの完成です。



A	B	C	D	E

Tarjan のアルゴリズムの実装例を以下に示します。

コード 12　Tarjan のアルゴリズムの実装

```
def tarjan_topo_sort(v: int, edges: list[list[int, int]]) -> list[int]:
    def dfs(node: int):
        visited[node] = True
        for connected_node in outedge[node]:
```

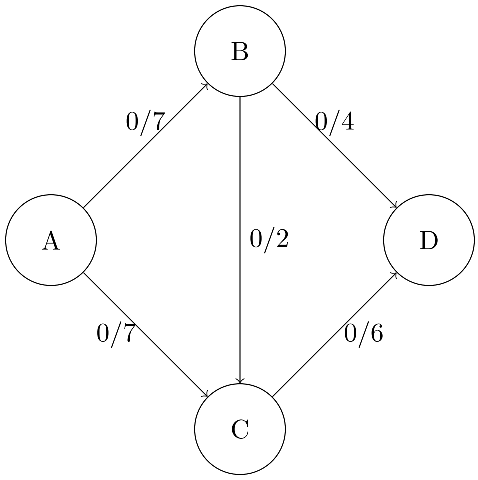
```
            if not visited[connected_node]:
                dfs(connected_node)
        sorted_g.append(node)

    outedge = [[] for _ in range(v)]
    visited = [False] * v
    sorted_g = []

    for start, end in edges:
        outedge[start].append(end)

    for i in range(v):
        if not visited[i]:
            dfs(i)

    return sorted_g[::-1]
```

iii.　問題

問題 1 AtCoder Educational DP Contest G - Longest Path

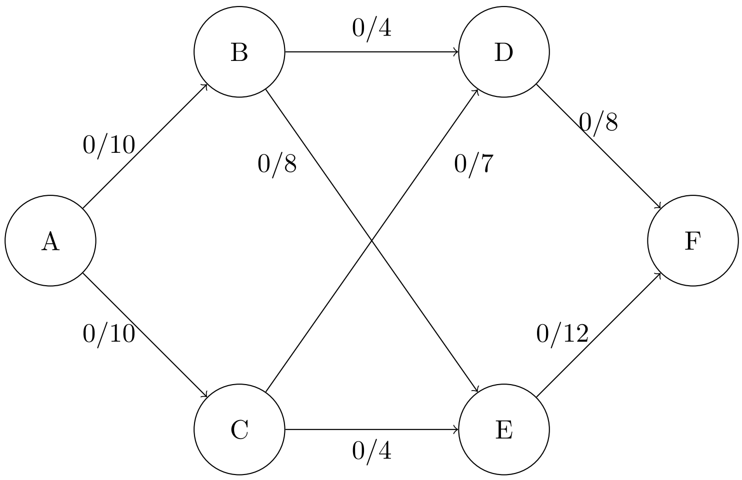
VIII.　最大流問題

グラフに流れ (フロー) があるグラフを考えます。特に、フローは始点 (source) のみから生まれ、終点 (sink) でのみ消滅するという性質を持つ場合を考えます。グラフでフローを扱う分野を**ネットワークフロー**といいます。エッジに容量を導入して、各エッジを通るフローの量が容量を超えないという条件のもとで終点に最も多くのフローを流す問題を**最大流問題**といいます。



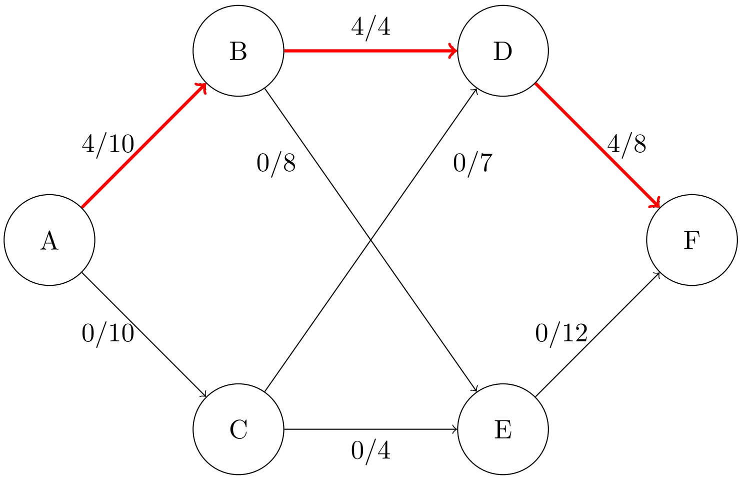
i.　最大流問題と貪欲法の限界

DFS を使って見つかった道から順番にフローを流す貪欲法から考えてみます。下のネットワークグラフを例にして考えます。A を source、F を sink とします。

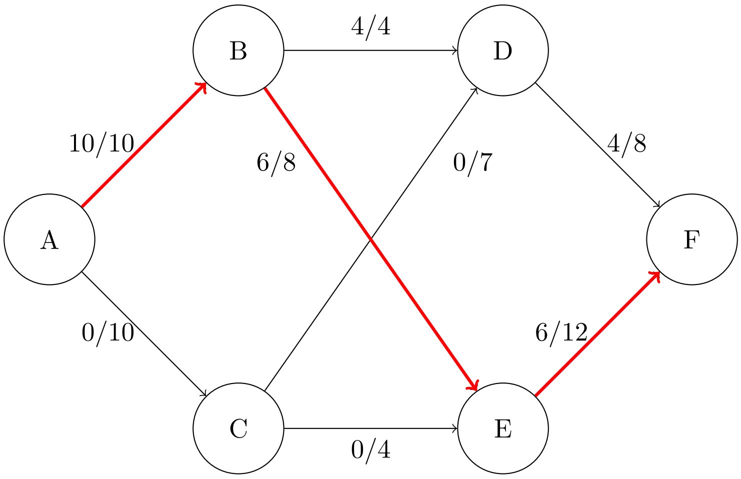


DFS を使って順番に経路を見つけていきます。最初は ABDF の経路にフローを流してみます。この経路の中で容量が最も小さい経路に合わせて流さないと溢れてしまうので、今回は BD の経路

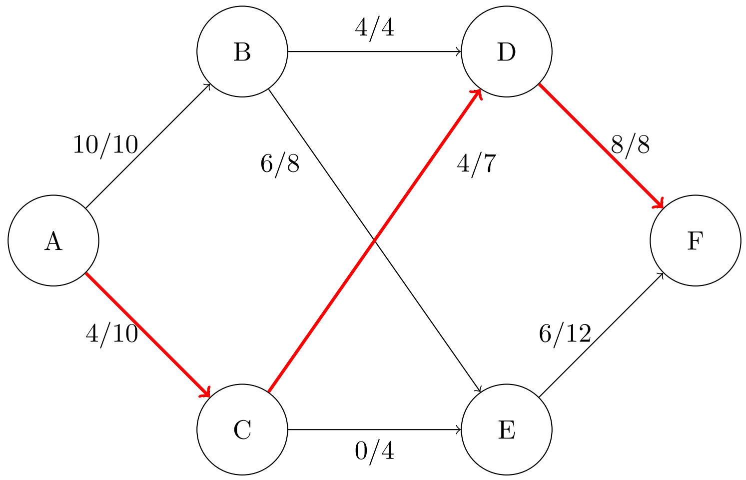
の容量の 4 を流します。



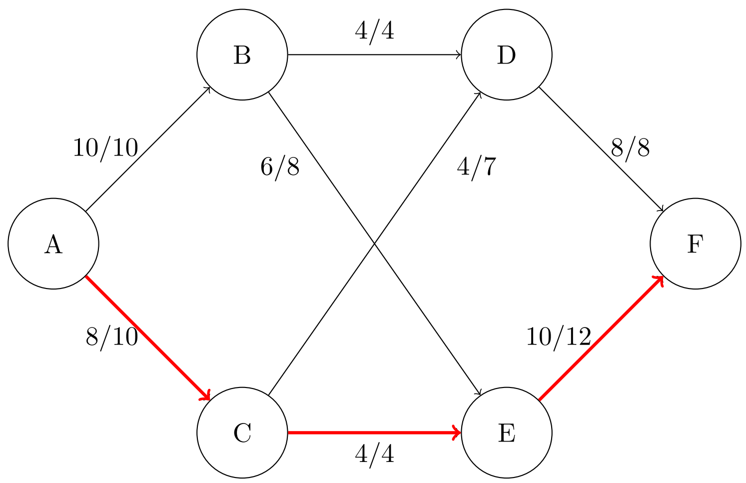
次に ABEF の経路にフローを流します。経路を見ていると AB の経路は残り 6 のフローを流せて ABEF の経路で最も小さい経路なので、6 のフローを流します。



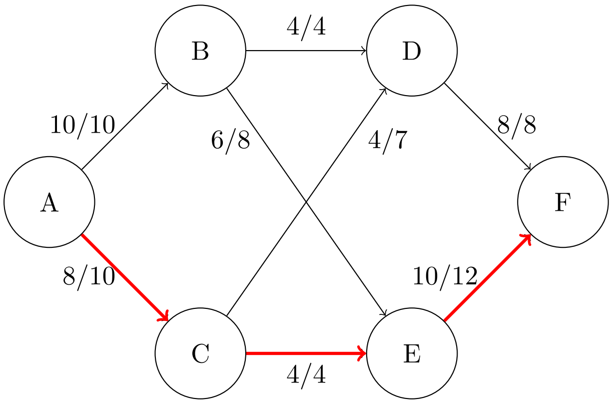
次に ACDF の経路を考える。DF の残り容量 4 が経路の中で最も小さな容量なので、4 のフローを流します。



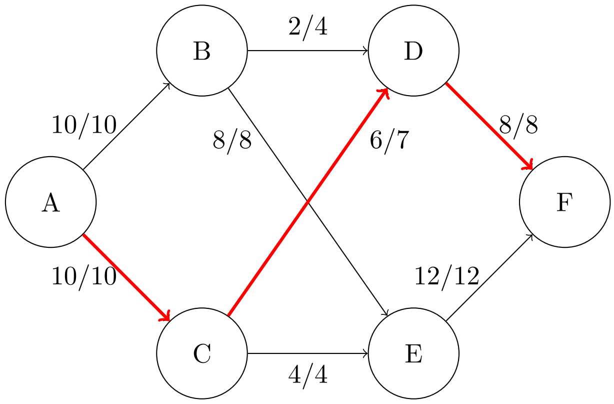
次は ACEF の経路を考える。経路 CE の残り容量 4 が経路の中で最小の容量になっているので、フロー 4 を流します。



F につながっているエッジのフローを見てみると、18 のフローが流れていることがわかります。しかし、このグラフの最大フローは 20 です。貪欲法では最大フローを求めることができません。貪欲法によるフロート最大フローのネットワークを見比べましょう。

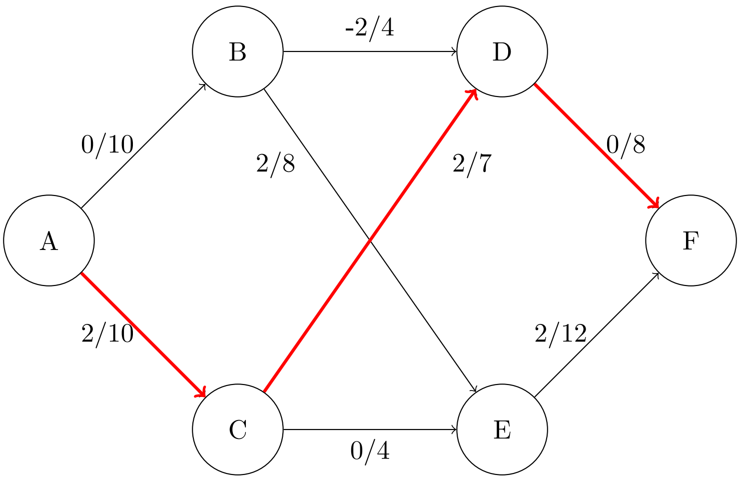


貪欲法によるフロー

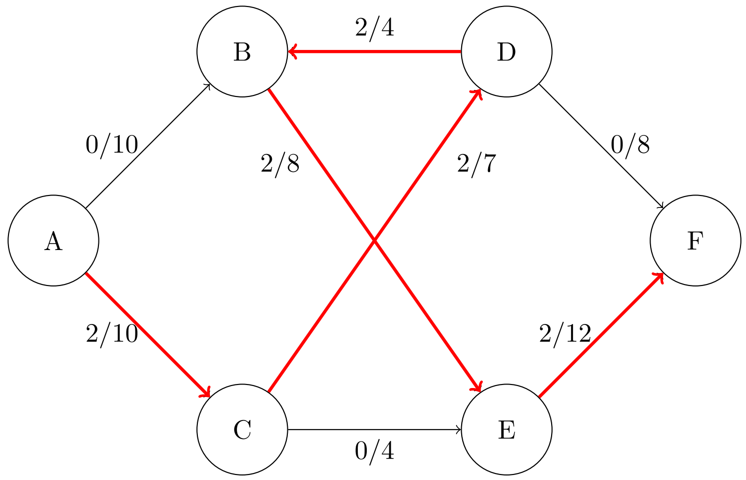


最大フロー

最大フローのネットワークから貪欲法のネットワークを引いてみましょう。

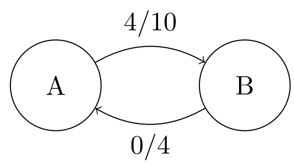


負の経路が現れました。負の経路を反対方向のフロートすると貪欲法では見えなかった経路が現れます。貪欲方に加えて、この A -> D -> B -> E -> F の経路を考慮したアルゴリズムを考える必要があります。

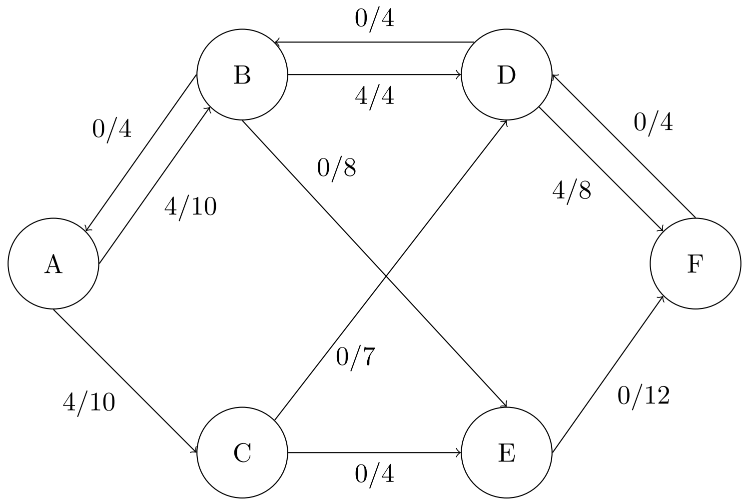


ii.　フォード・ファルカーソン法

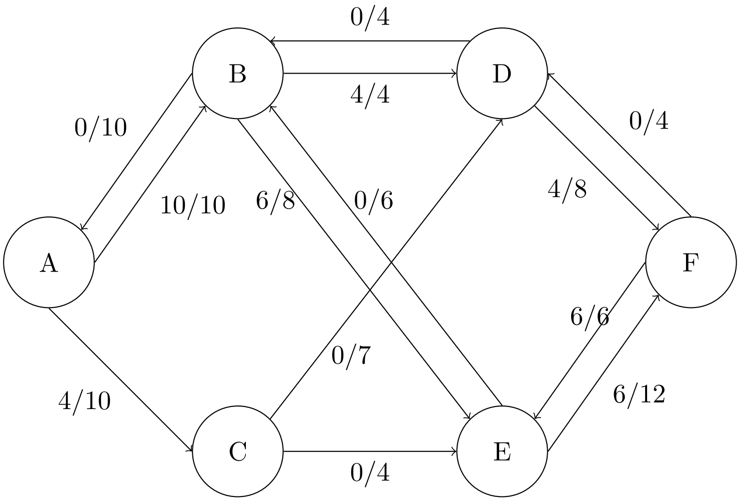
負の閉路まで考えた経路を考慮することで最大フローを求めるアルゴリズムがフォード・ファルカーソン法です。フォード・ファルカーソン法は基本的には貪欲法と同じですが、DFS で経路を辿るたびに流したフローのマイナスの経路を追加します。マイナスの経路を追加するとは、以下のように順方向のフローが流れたときに逆方向に容量を追加することを意味します。



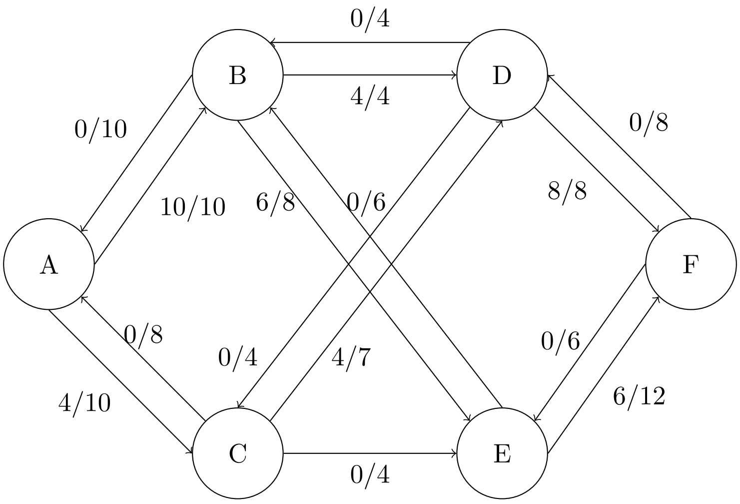
フォード・ファルカーソン法の例を見てみましょう。まずは A -> B -> D -> F の経路を考えます。4 のフローを流せるので逆方向の容量を 4 追加します。



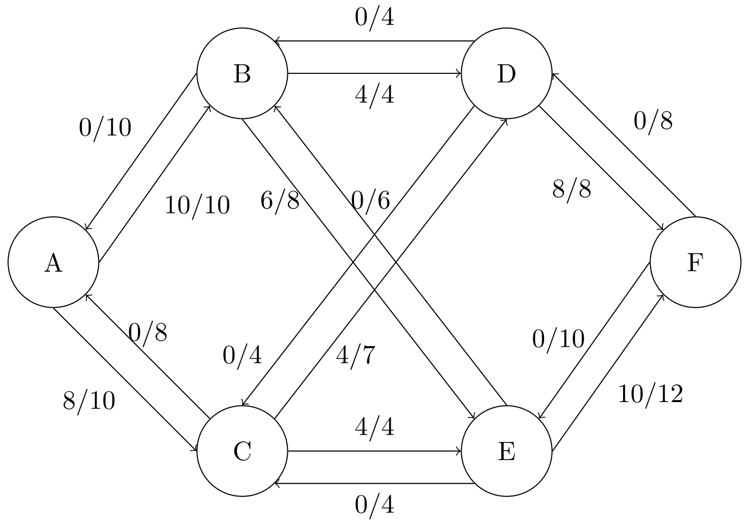
次に ABEF の経路を考えます。



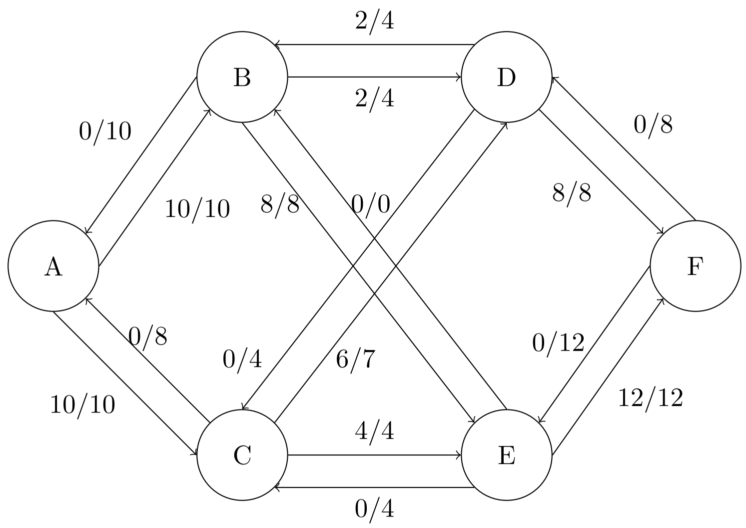
次に ACDF の経路を考えます。



次に ACEF の経路を考えます。



貪欲法ではここで終了しました。負の経路があるのでまだ ACDBEF の経路があります。この経路を考えると、最大フローは 20 になります。



最後にフォード・ファルカーソン法の実装について扱います。今までのグラフ探索アルゴリズムでは隣接リスト形式でグラフを表現してきましたが、今回は隣接行列を使ってグラフを表現します。例のグラフの隣接行列は以下のようになります。DFS で探索するたびに対象な位置の要素を更新していきます。

	A	B	C	D	E	F
A	0	10	10	0	0	0
B	0	0	0	4	8	0
C	0	0	0	0	4	0
D	0	0	0	0	0	12
E	0	0	0	0	0	12
F	0	0	0	0	0	0

コード 13 フォード・ファルカーソン法の実装

```python
def ford_fulkerson(cap: list[list[int]], n: int, start: int, end: int) -> int:
  def dfs_ff(start: int, end: int, flow: int) -> int:
      if start == end:
          return flow

      visited[start] = True

      for i in range(n):
          if not visited[i] and cap[start][i] > 0:
              next_cap = cap[start][i]
              new_cap = min(flow, next_cap)
              f = dfs_ff(i, end, new_cap)

              if f > 0:
                  cap[start][i] -= f
                  cap[i][start] += f
                  return f

      return 0

  max_flow = 0

```

```
    while True:
        visited = [False] * n
        f = dfs_ff(start, end, 10 ** 9)
        if f == 0:
            break
        max_flow += f

    return max_flow
```

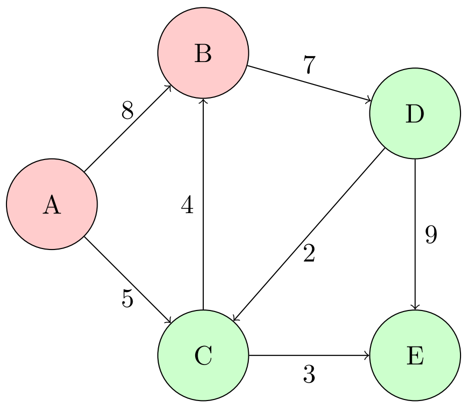
iii. 最大フロー問題の再考

最大フローに関わるグラフの性質を確認します。

1. s-t カット

グラフのカットはあるグラフを異なる 2 つの部分に分けることを指します。このとき、グラフの頂点集合 V を S と T に分けることを考えます。これは (S, V \ S) のように表します。**カットエッジ**とは、S と T を結ぶ辺のことを指します。

例を見てみましょう。以下のグラフを考えます。A と B、C と D と E からなるグループに分けます。カットエッジは A と C を結ぶエッジ、B と D を結ぶエッジ、B と C を結ぶエッジになります。



s-t カットとは、集合 S に開始ノード、集合 T に終了ノードを含むカットのことを指します。上のグラフでは A をソース、E をシンクと考えれば上のカットは s-t カットになります。**最小カット**とは、s-t カットの中で S から T に向かう辺の容量の総和 U(S → T) が最小になるカットのことを指します。

iv.　最大流・最小カットの定理

最大流・最小カットの定理

グラフのおいて、s-t カットの最大フローは、s-t カットの最小カットと等しい。

参考

- `https://www.nic.ad.jp/ja/materials/iw/2013/proceedings/s8/s8-kaneko.pdf`

v.　問題

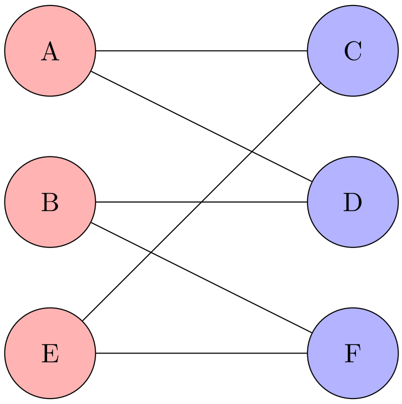
問題 1 AtCoder 典型アルゴリズム問題集 上級 エキスパート編 E - 最大流

IX.　最小費用流問題

i.　プライマル・デュアル法

X.　二部グラフ (bipartite graph)

二部グラフとは、ノードを 2 つのグループに分けて、同じグループに属するノード同士は辺で結ばれていないグラフのことを指します。



i.　二部グラフ判定

ii.　重み付き二部グラフの最大マッチング問題

参考

- `https://qiita.com/drken/items/e805e3f514acceb87602`