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。

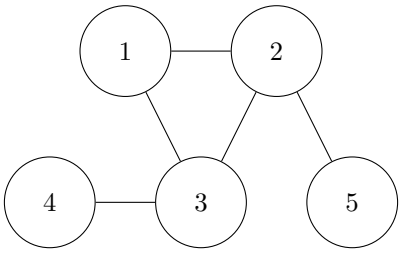
- 連結なグラフである
- 閉路を持たない

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。

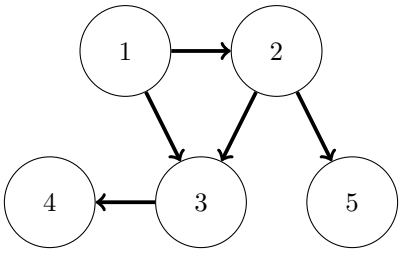
- G には閉路がなく、$n-1$ 本のエッジを持つ
- G は連結であり、$n-1$ 本のエッジを持つ
- G の任意の 2 点を結ぶ経路はただ 1 つ存在する

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。



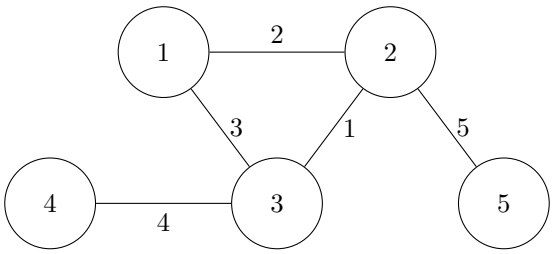
無向グラフ



有向グラフ

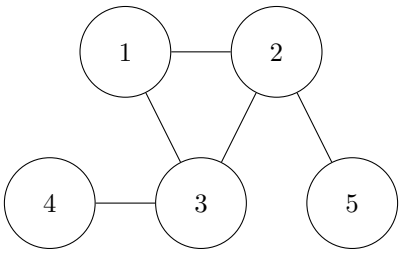
iii. 重み付きグラフ

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

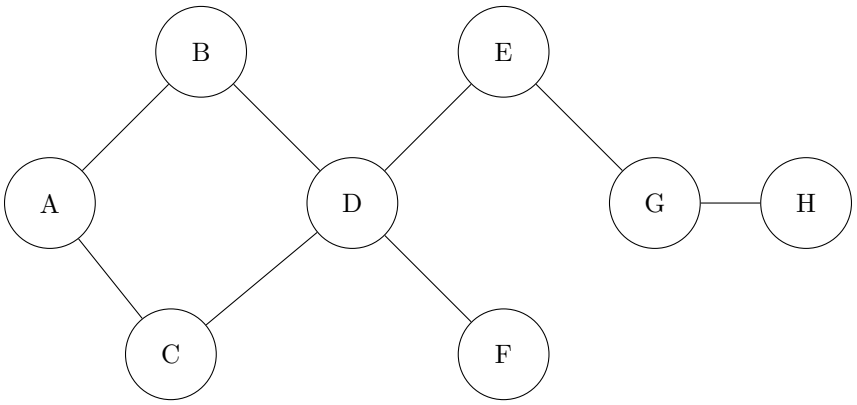
隣接リストは各ノードに隣接するノードをリストで表現したものです。

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

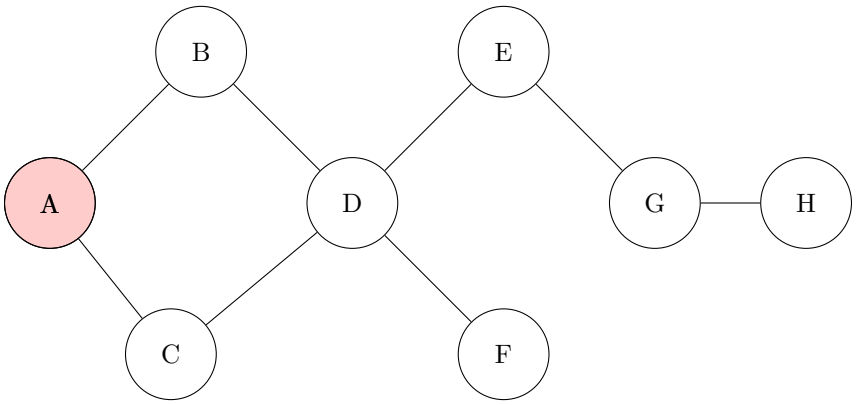
グラフの基本として、深さ優先探索 (DFS) と幅優先探索 (BFS) というグラフの探索アルゴリズムを扱います。

II.　幅優先探索 (BFS)

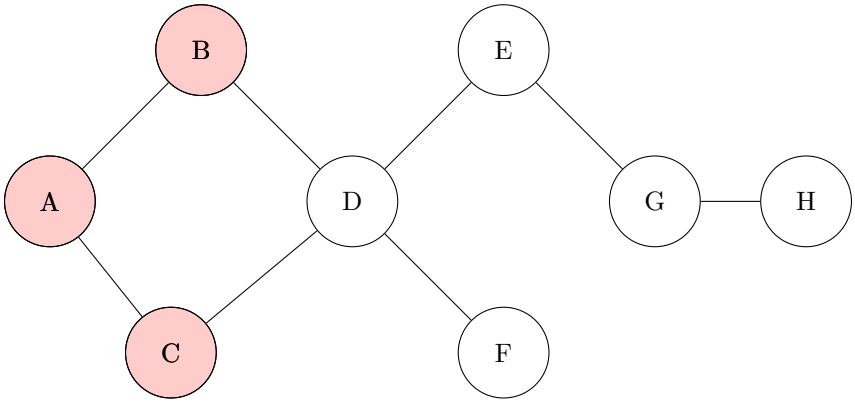
BFS は、後戻りしないように、可能性のあるルートすべてにおいて 1 ステップずつ行くアルゴリズムです。BFS の例を見てみましょう。



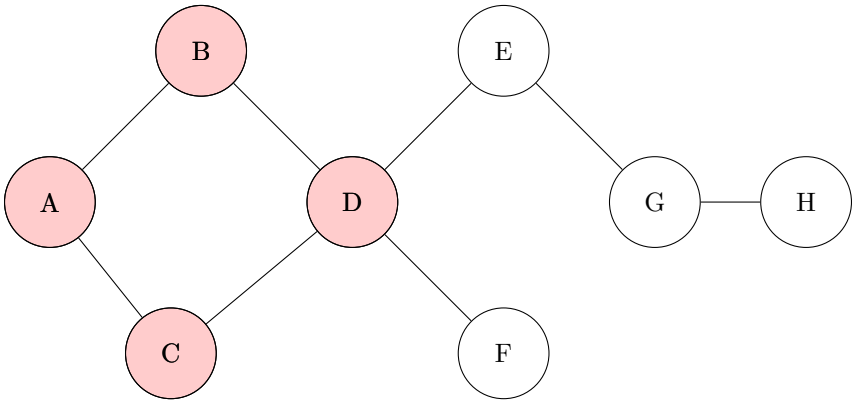
A からスタートします。



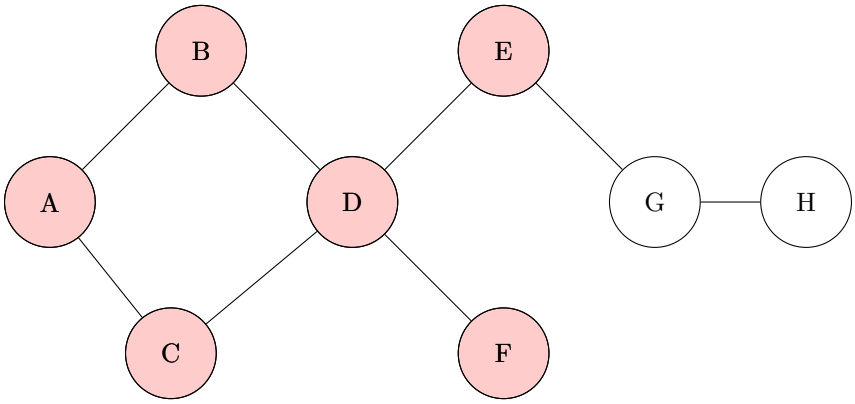
次に A と繋がっているノード B と C を探索します。



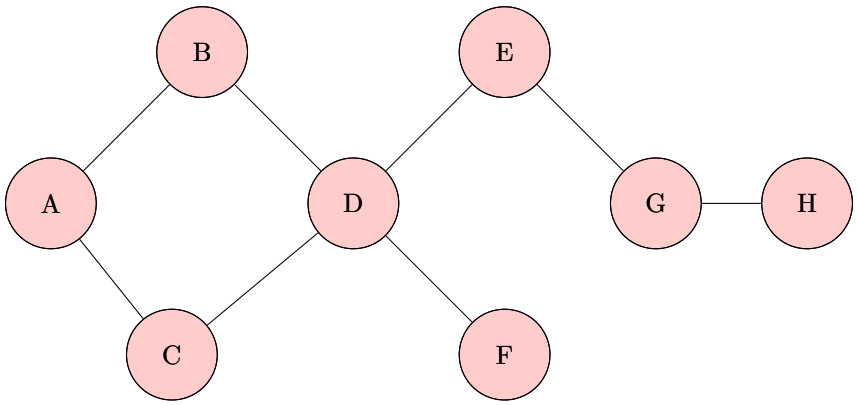
A の探索が終わったので、次に B と C の探索を行います。今回は B から探索します。B と C は同じ深さにあるので、どちらから探索しても問題ありません。B には D が繋がっているので、D を探索します。C から探索を始めようとすると、すでに D はすでに探索済みなので、探索を行いません。



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

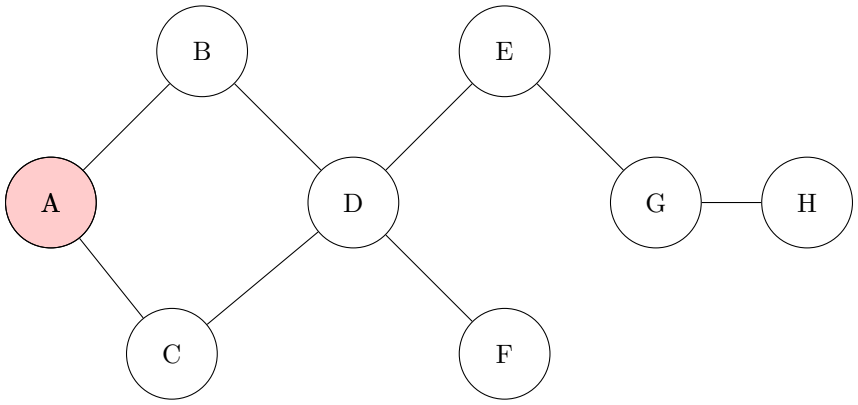
```
from collections import deque

  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]:
```

```
16                     todo.append(next_node)
17
18         return visited
```

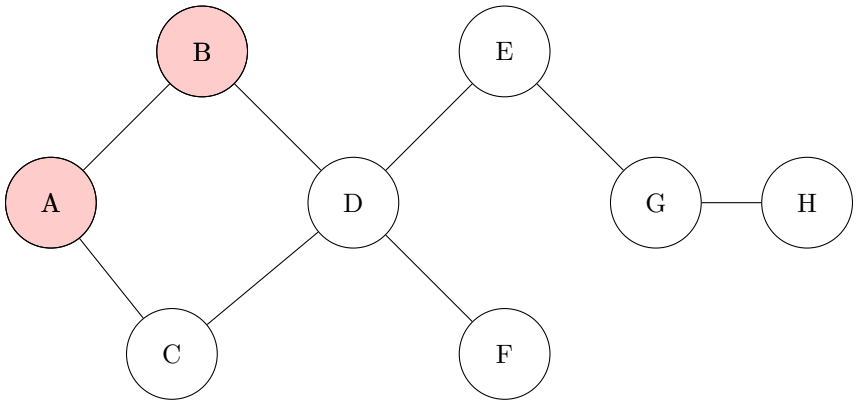
III.　深さ優先探索 (DFS)

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。

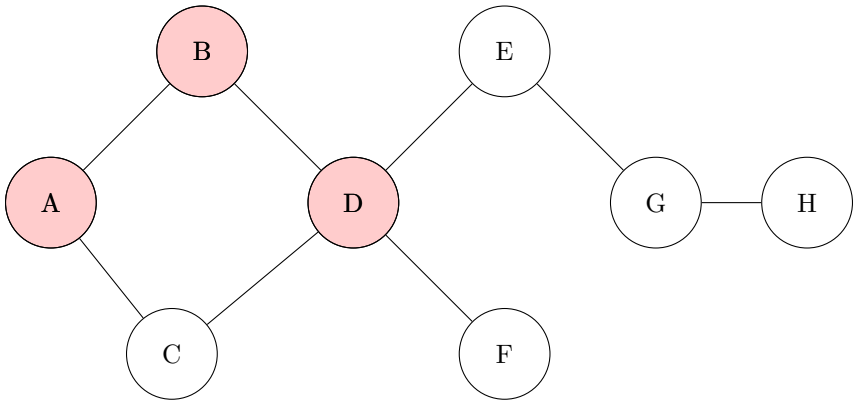
最初は A から探索を行います。



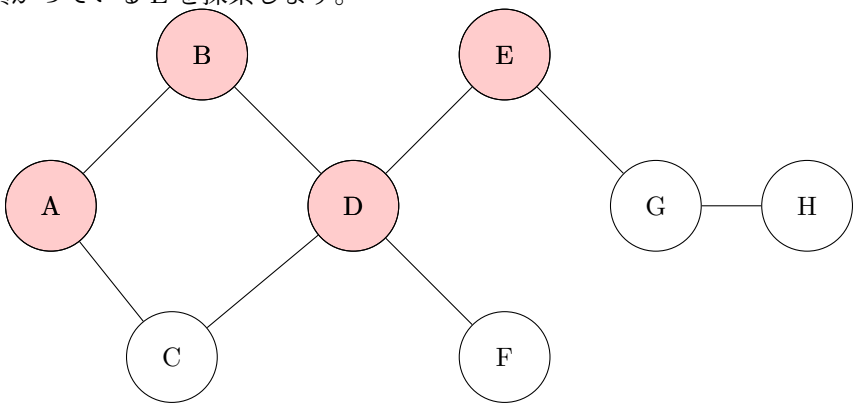
次に A と繋がっているノード B を探索します。



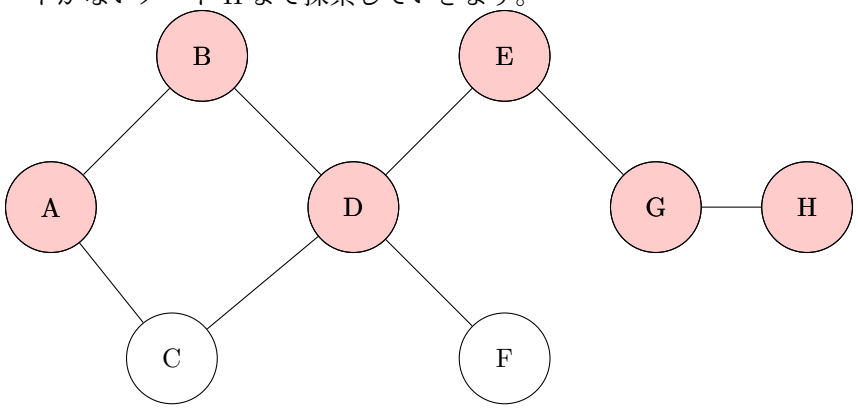
BFS では C を次に探索しますが、DFS では B に繋がっている D を探索します。



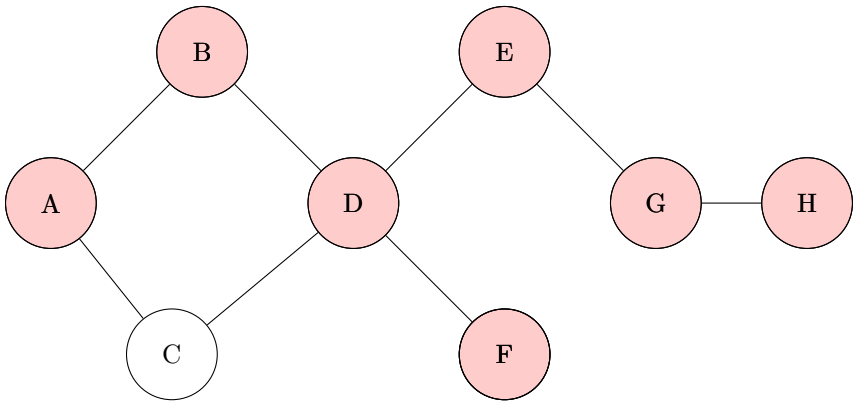
次に D に繋がっている E を探索します。



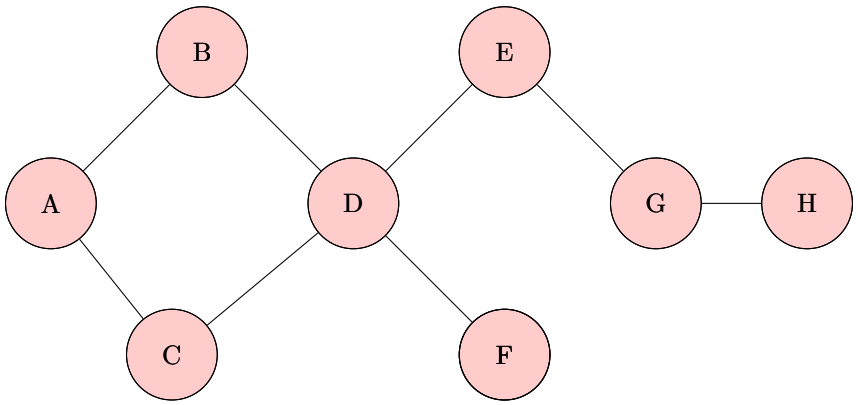
これ以上ノードがないノード H まで探索していきます。



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]:
```

```
15                if not visited[next_node]:
16                    todo.append(next_node)
17
18        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。

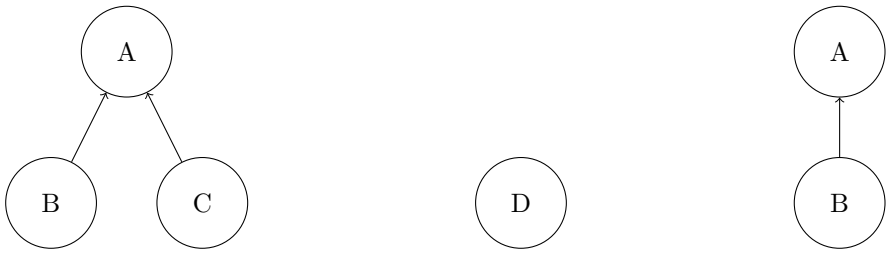
コード 3　深さ優先探索再帰の実装

```
1  def dfs(graph: list[list[int]], start: int, visited: list[bool]) -> list[
       bool]:
2      visited[start] = True
3      for next_node in graph[start]:
4          if visited[next_node]:
5              continue
6          else:
7              dfs(graph, next_node, visited)
8
9      return visited
```

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

```python
class UnionFind:
  def __init__(self, size: int) -> None:
      self.size: int= size
      self.parent: list[int] = [i for i in range(self.size)]
      self.depth: list[int] = [1] * self.size

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

  def is_same(self, x: int, y: int) -> bool:
      return self._root(x) == self._root(y)

  def unite(self, x: int, y: int) -> None:
      x, y = map(self.root, (x, y))
      # すでに同じグループに属するなら何もしない
      if x == y:
```

```
21            return
22        # union by size y側のサイズが小さくなるようにする
23        if self.depth[x] < self.depth[y]:
24            x, y = y, x
25        # yをxの子とする
26        self.parent[y] = x
27        self.depth[x] += self.depth[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 1 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。

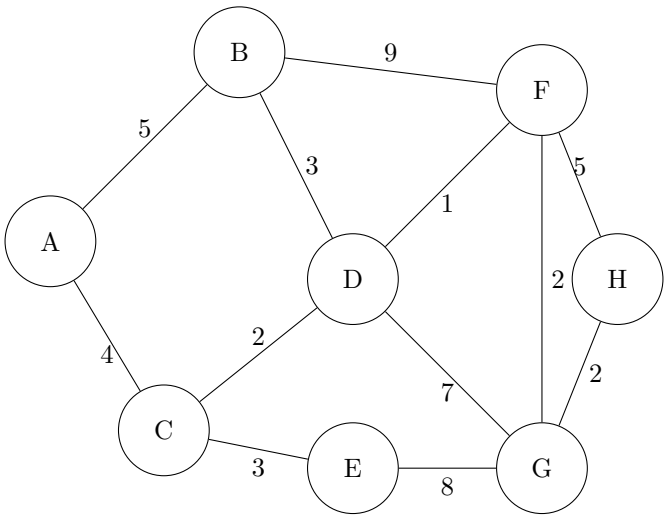
経路緩和性

最短経路の部分経路も最短経路である

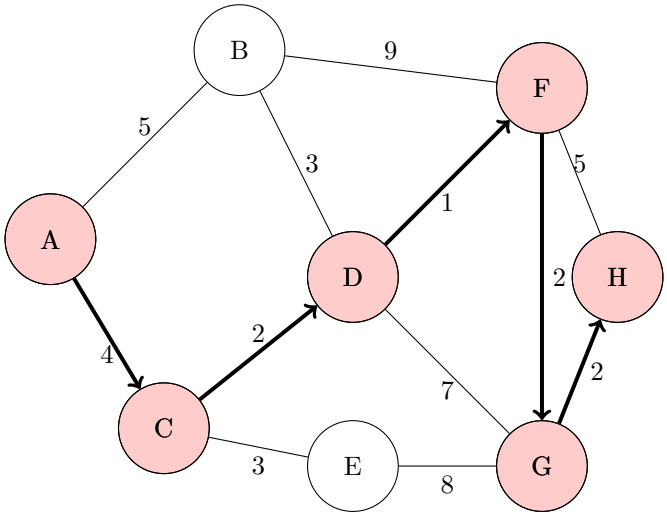
簡単な証明

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。



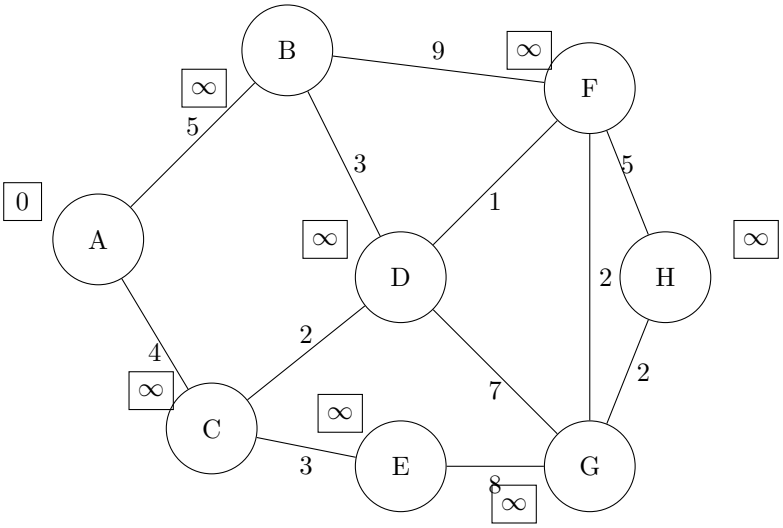
A から H までの最短経路は以下の通りです。もしもゴールが G、F、D、C いずれの場合でも、最短経路は A → C → D → F → G → H の経路になります。



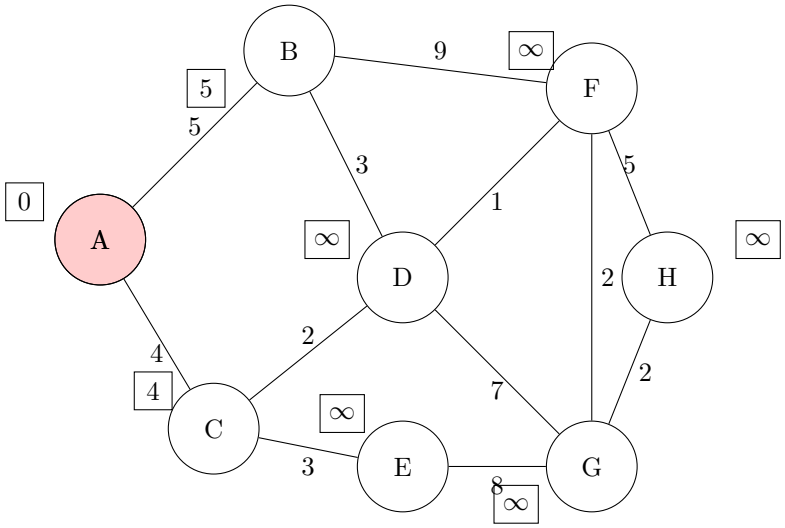
以上の性質より、あるノードまでの最短経路を考えるときはそのノードの前のノードまでの最短経路を考えればよいことがわかります。この性質を利用したのアルゴリズムがダイクストラ法です。ダイクストラ法は以下の手順で最短経路を求めることができます。

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

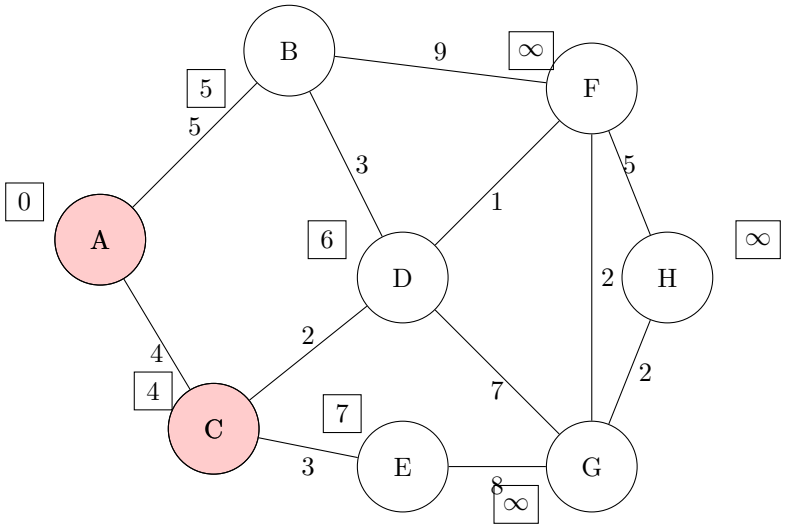
距離は ∞、スタート地点は 0 で初期化します。上のグラフを例にしてダイクストラ法の例を見てみます。最初の距離が確定していないのーどで最も距離が短いノードは A です。A に繋がっているノードの距離を更新します。



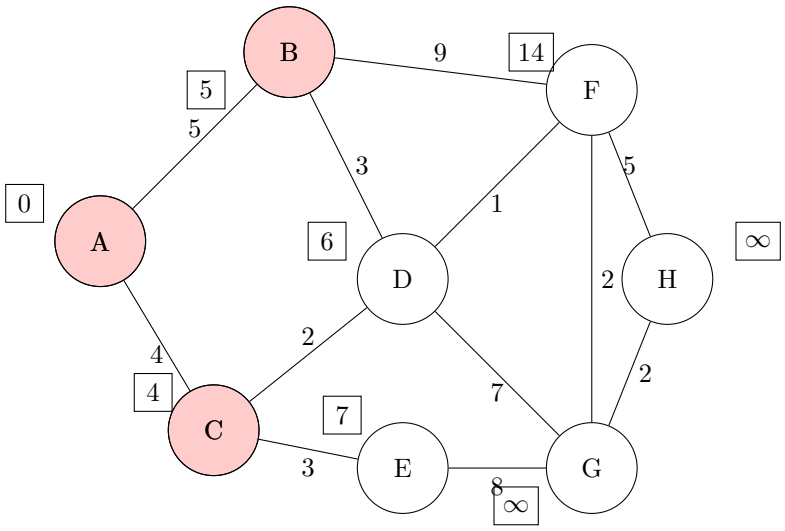
最初の距離が確定していないのーどで最も距離が短いノードは A です。A に繋がっているノードの距離を更新します。更新が終わったので、A の距離を確定します。



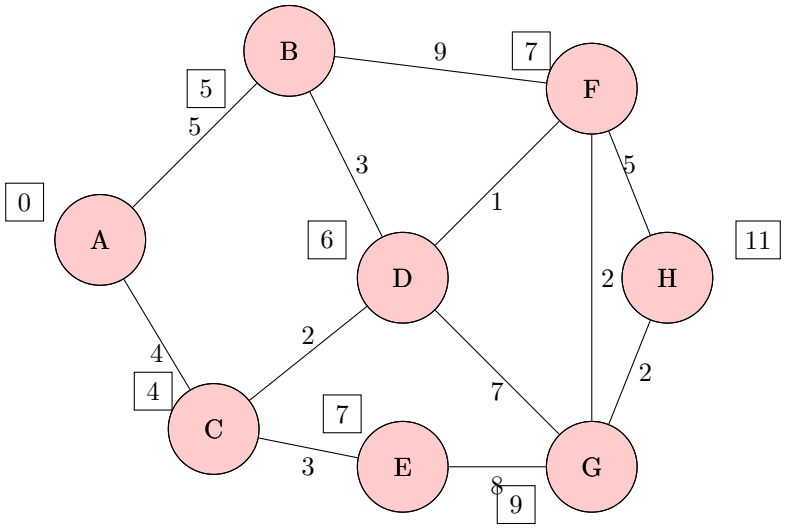
次に距離が確定していないノードで最も距離が短いノードを選択します。今回は C です。C に繋がっているノードの距離を更新します。C に繋がっているノードを更新したら、C の距離を確定します。



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]], start: int) -> list[int]:
    done = [False] * len(graph)
    dist = [1 << 60] * len(graph)
    todo = []

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

    while todo:
        distance, node = heappop(todo)
        if done[node]:
            continue

        # 更新
        for connected_node, weight in graph[node]:
            if distance + weight < dist[connected_node]:
                dist[connected_node] = distance + weight
                heappush(todo, (dist[connected_node], connected_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。

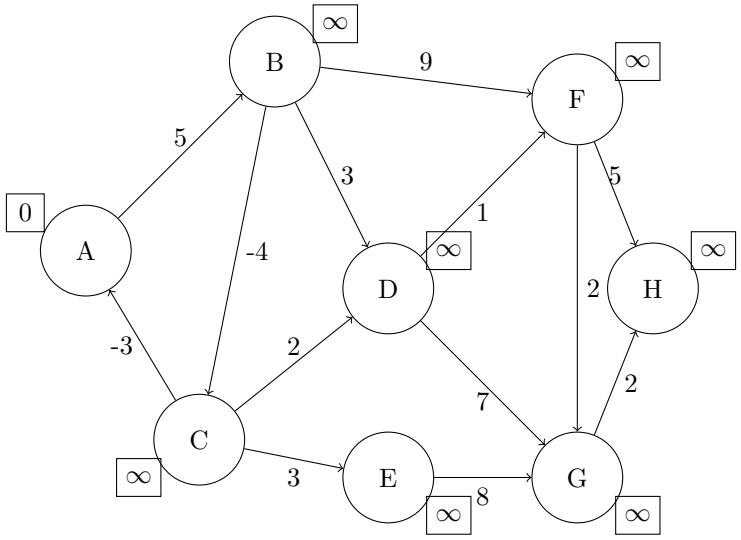
すべて列挙すると、以下の様になります。

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

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。

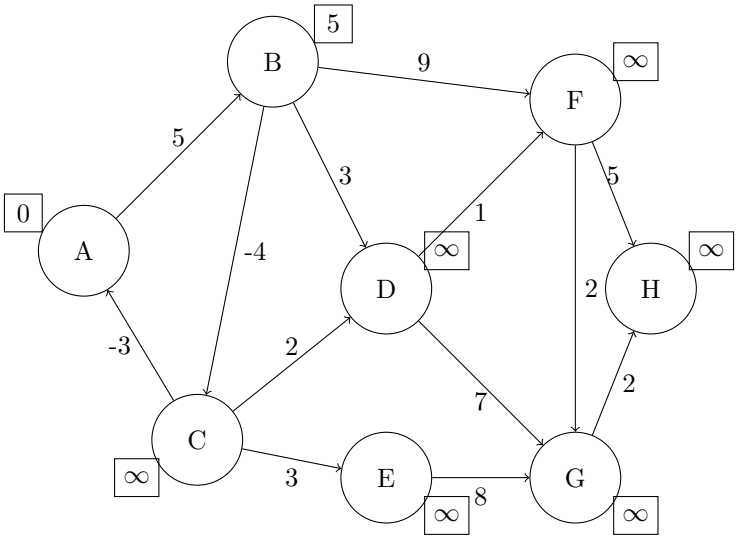
1. edges をすべて列挙し、始点から終点への重みを更新する
2. 1 を V - 1 回繰り返す (V はノードの数)

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。

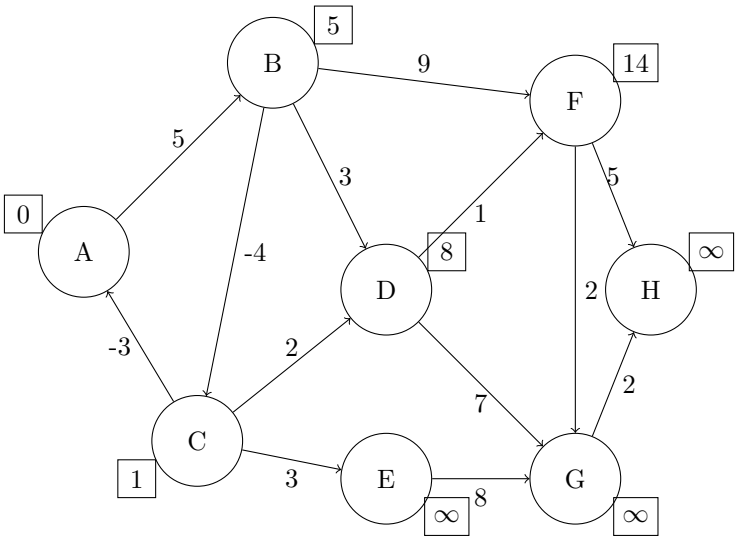


ベルマン・フォード法の流れを確認します。最初は A と繋がっている B が更新されます。他のノードも edges をすべて列挙して更新を図りますが、∞+ 有限の値と考えるため、更新されないと

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
```

```
5
6   for _ in range(v - 1):
7       for start, end, weight in edges:
8           if dist[start] != 1 << 60 and dist[start] + weight < dist[end]:
9               dist[end] = dist[start] + weight
10
11  # 一度v-1回更新した後に負の閉路を検出
12  for start, end, weight in edges:
13      if dist[start] != 1 << 60 and dist[start] + weight < dist[end]:
14          return -1
15
16  return dist
```

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

```
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
```

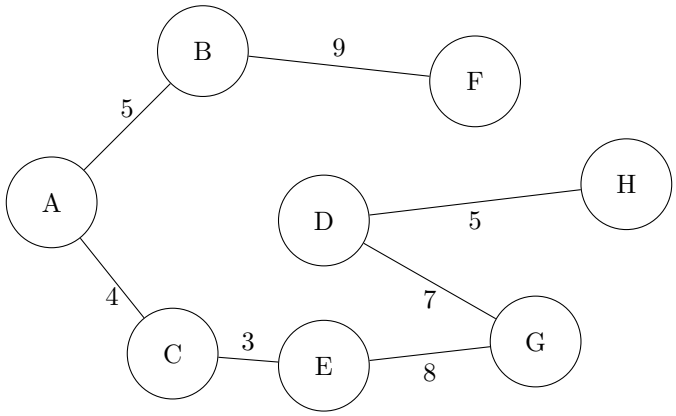
iv.　ワーシャルフロイド法

コード 8　ワーシャルフロイド法の実装

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

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。

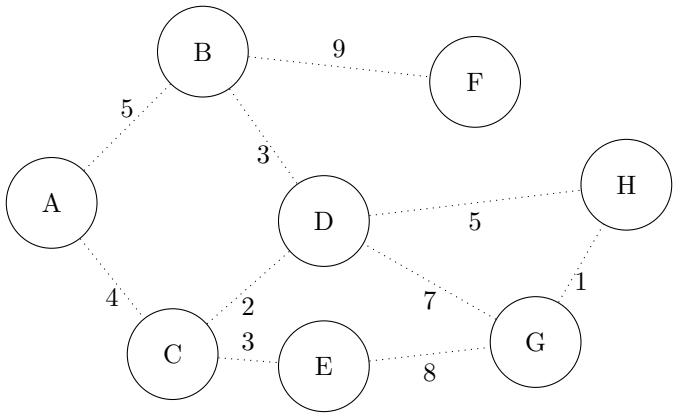


最小全域木を求めるには、辺ベースのアプローチとノードベースのアプローチの 2 種類があります。それぞれ**クラスカル法**、**プリム法**と呼ばれています。

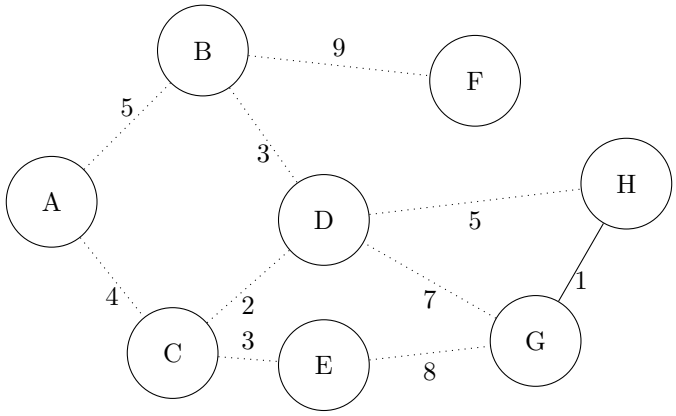
i.　クラスカル法

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アルゴリズムです。以下のアルゴリズムに従っています。

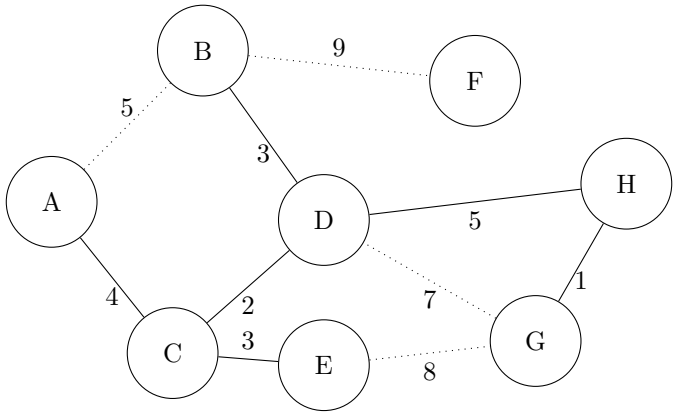
1. すべての辺を重さの小さい順にソート
2. 重さの最も小さい辺を選ぶ
3. 今までに選んだ辺から構成される木に 2 で選んだ辺を追加した時に、閉路が生まれないことを確認する。閉路が新しくできないならこの辺を追加する
4. すべての辺をチェックし終えるまで 2 から 3 を繰り返す



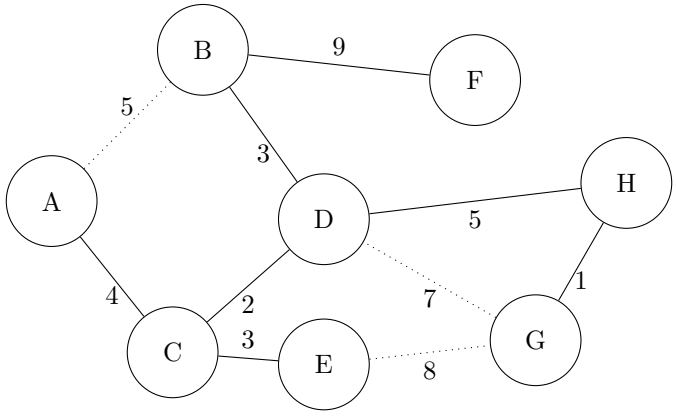
クラスカル法の例を見てみましょう。まずは、すべての辺の中で最も重さが小さい辺を選んでそれを木に追加したときに、閉路ができないのでその辺を追加します。



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。

ii.　プリム法

VII.　トポロジカルソート

VIII. 最大流問題

IX. 参考

ベルマンフォード法

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