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。

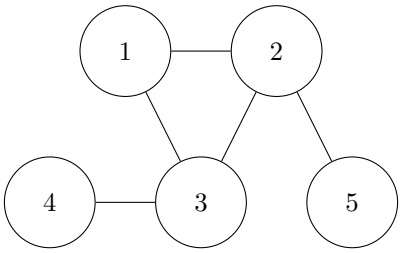
- 連結なグラフである
- 閉路を持たない

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。

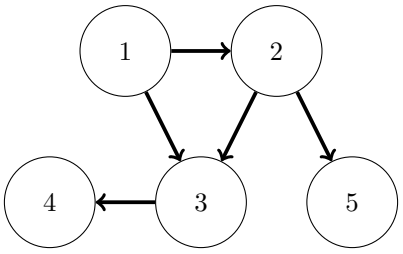
- G には閉路がなく、$n-1$ 本のエッジを持つ
- G は連結であり、$n-1$ 本のエッジを持つ
- G の任意の 2 点を結ぶ経路はただ 1 つ存在する

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。



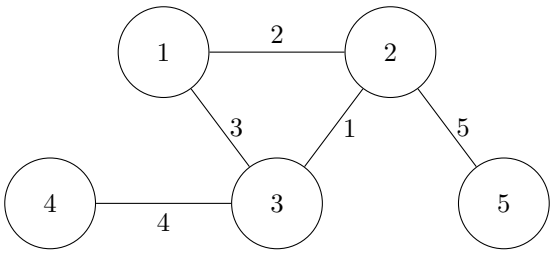
無向グラフ



有向グラフ

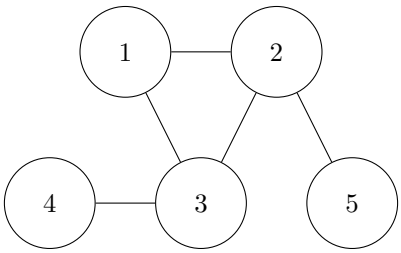
iii. 重み付きグラフ

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

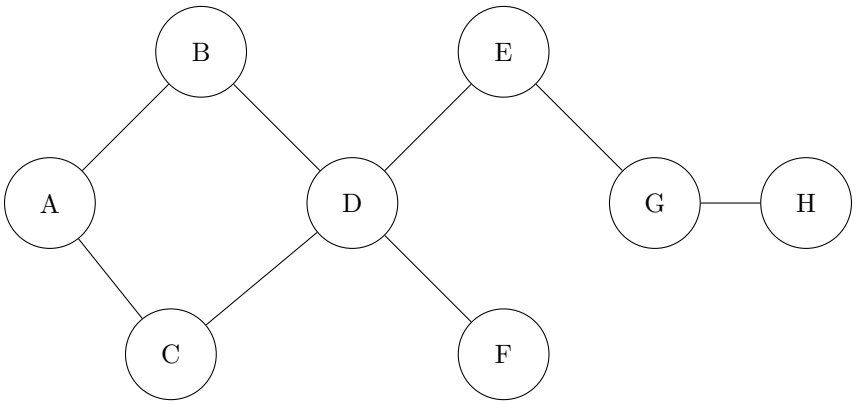
隣接リストは各ノードに隣接するノードをリストで表現したものです。

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

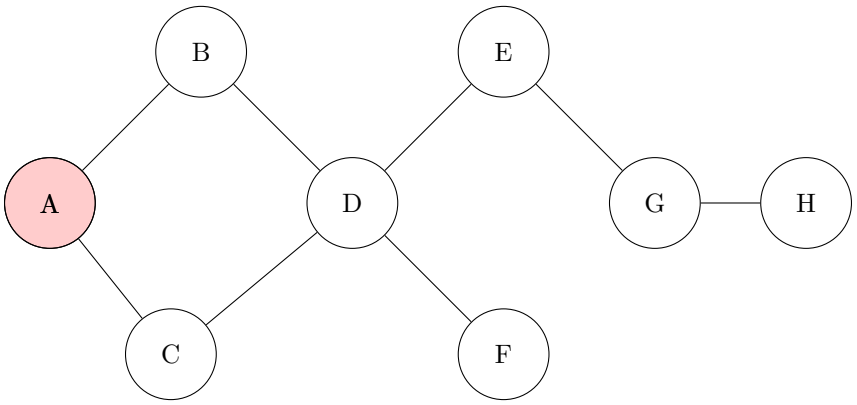
グラフの基本として、深さ優先探索 (DFS) と幅優先探索 (BFS) というグラフの探索アルゴリズムを扱います。

II.　幅優先探索 (BFS)

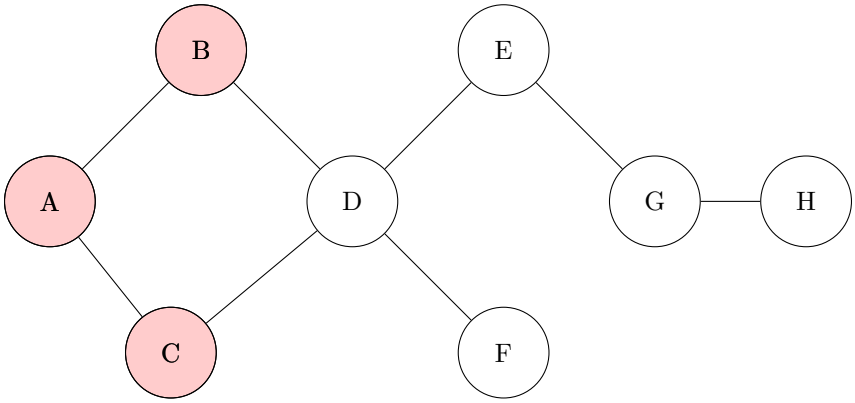
BFS は、後戻りしないように、可能性のあるルートすべてにおいて 1 ステップずつ行くアルゴリズムです。BFS の例を見てみましょう。



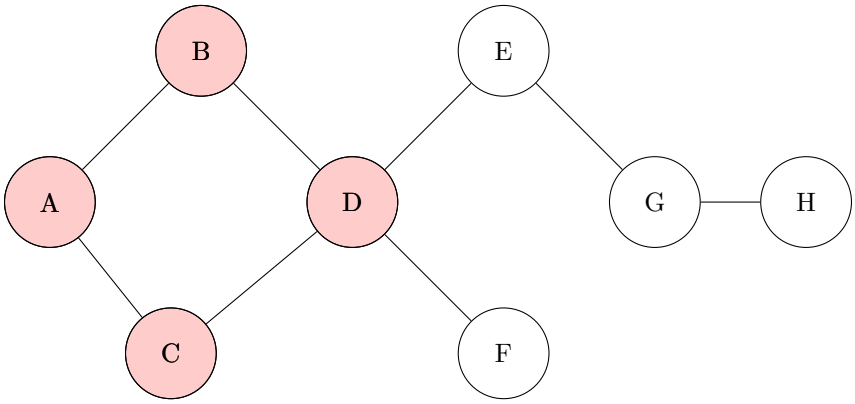
A からスタートします。



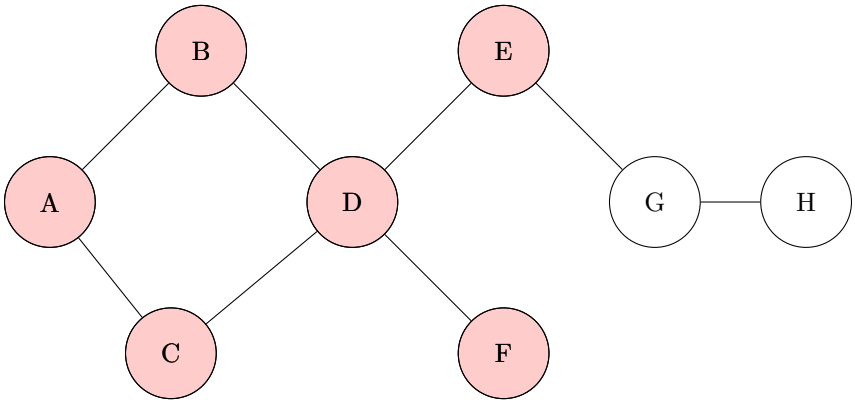
次に A と繋がっているノード B と C を探索します。



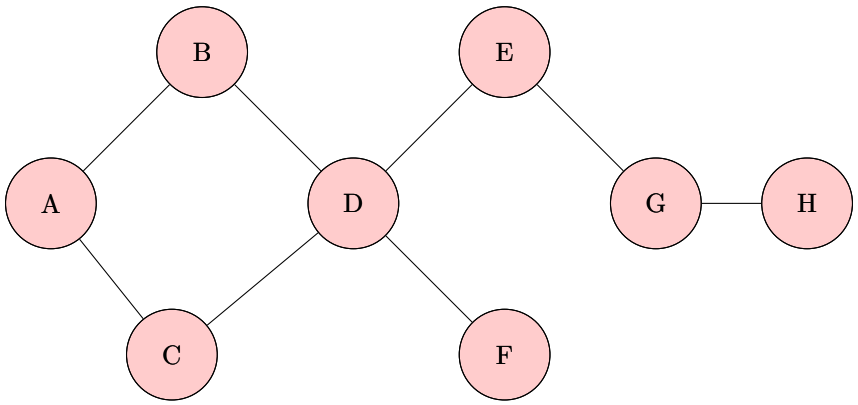
A の探索が終わったので、次に B と C の探索を行います。今回は B から探索します。B と C は同じ深さにあるので、どちらから探索しても問題ありません。B には D が繋がっているので、D を探索します。C から探索を始めようとすると、すでに D はすでに探索済みなので、探索を行いません。



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

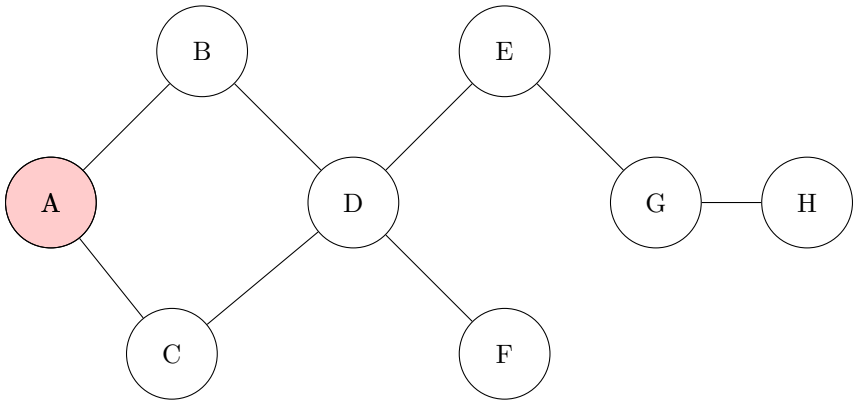
```
from collections import deque

  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
```

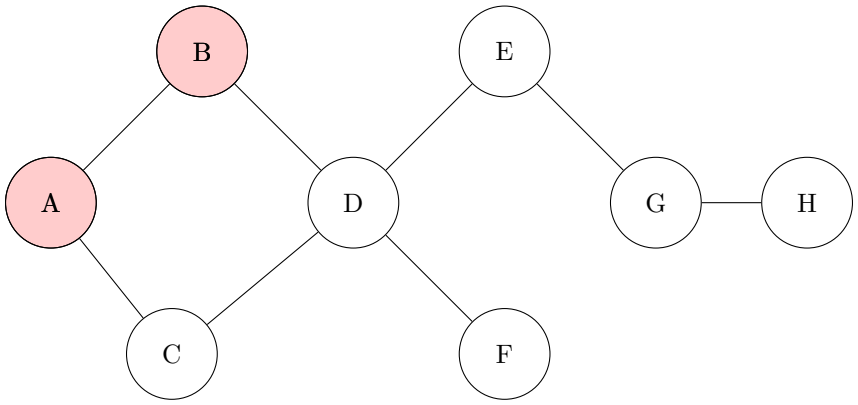
III.　深さ優先探索 (DFS)

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。

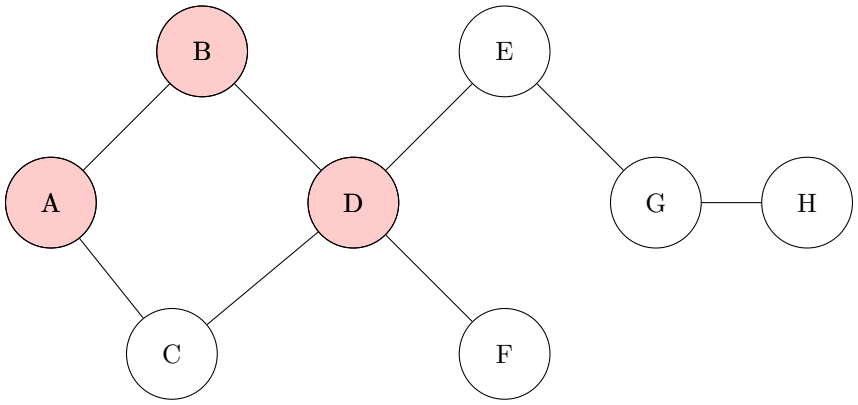
最初は A から探索を行います。



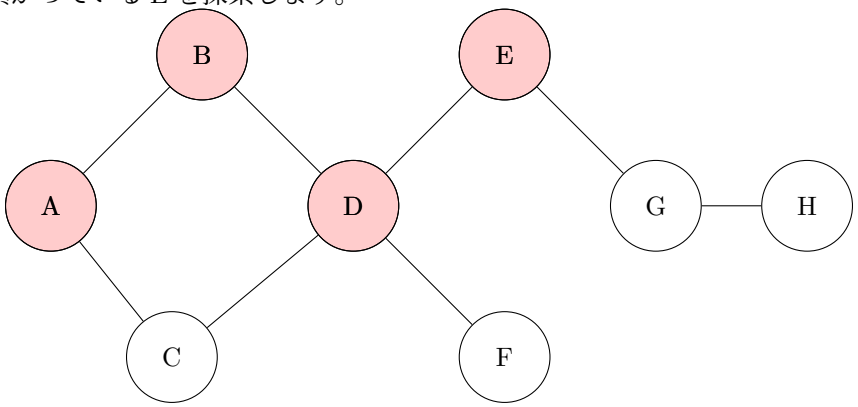
次に A と繋がっているノード B を探索します。



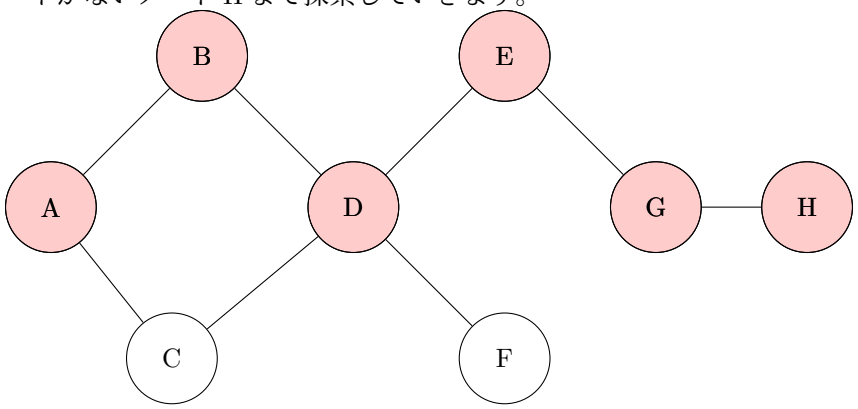
BFS では C を次に探索しますが、DFS では B に繋がっている D を探索します。



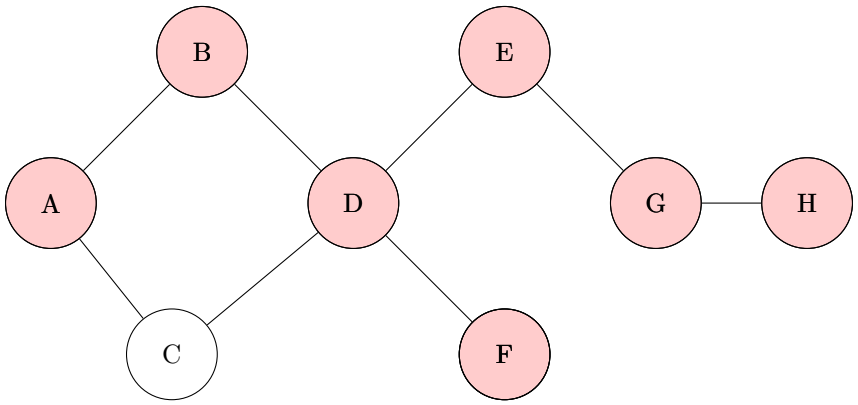
次に D に繋がっている E を探索します。



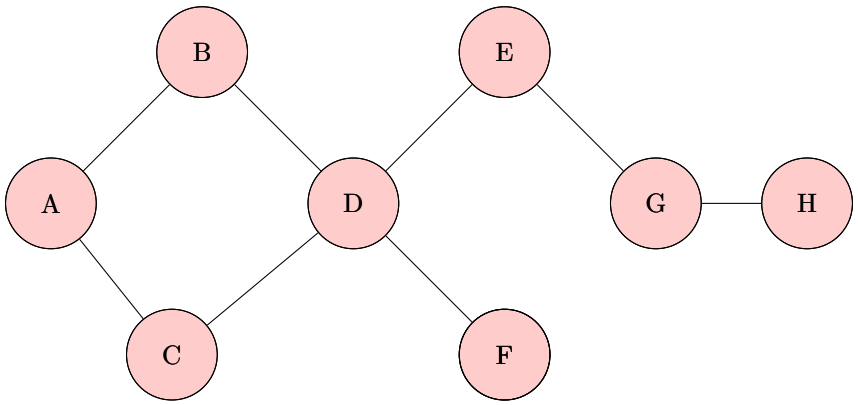
これ以上ノードがないノード H まで探索していきます。



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.　BFS と DFS を用いた問題

BFS と DFS は実装はシンプルですが、応用先は広いです。以下にいくつかの問題を紹介します。

i.　最短経路問題

ii.　オイラーツアー

iii.　LCA

iv.　橋の検出

V.　最短経路問題

DFS 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。

i.　ダイクストラ法

ii.　ベルマンフォード法

iii.　SPFA(Shortest Path Faster Algorithm)

iv.　ワーシャルフロイド法

VI.　最小全域木

VII.　トポロジカルソート

VIII. 最大流問題