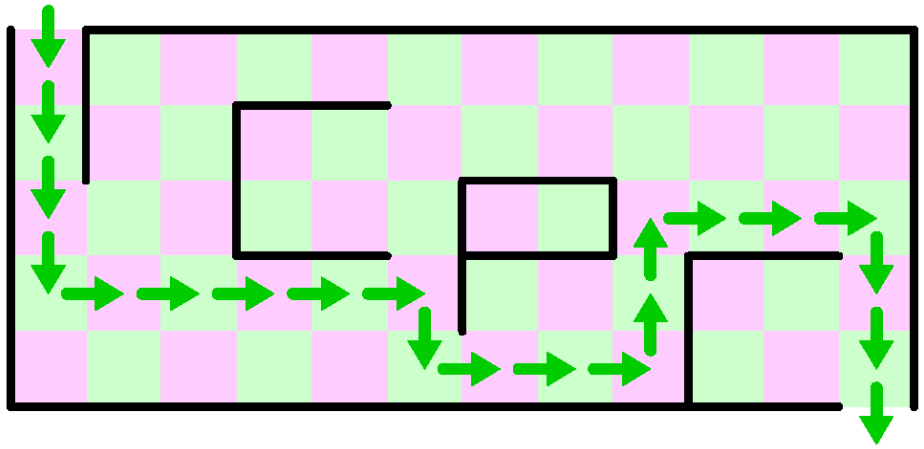
プログラム例@github

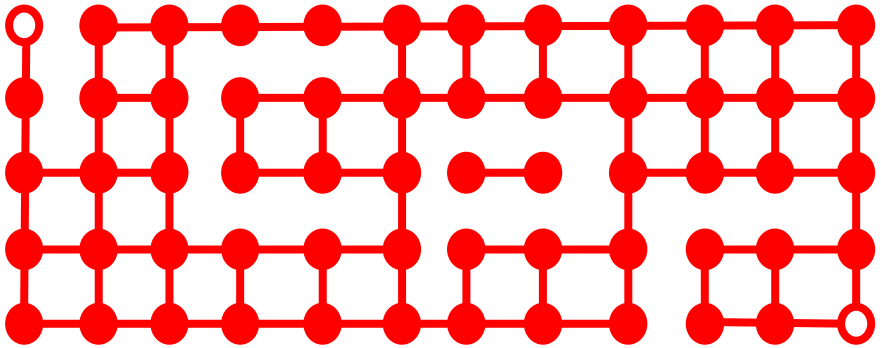
最短経路の探索

幅優先探索、最短経路探索

お題（迷路と命ず）

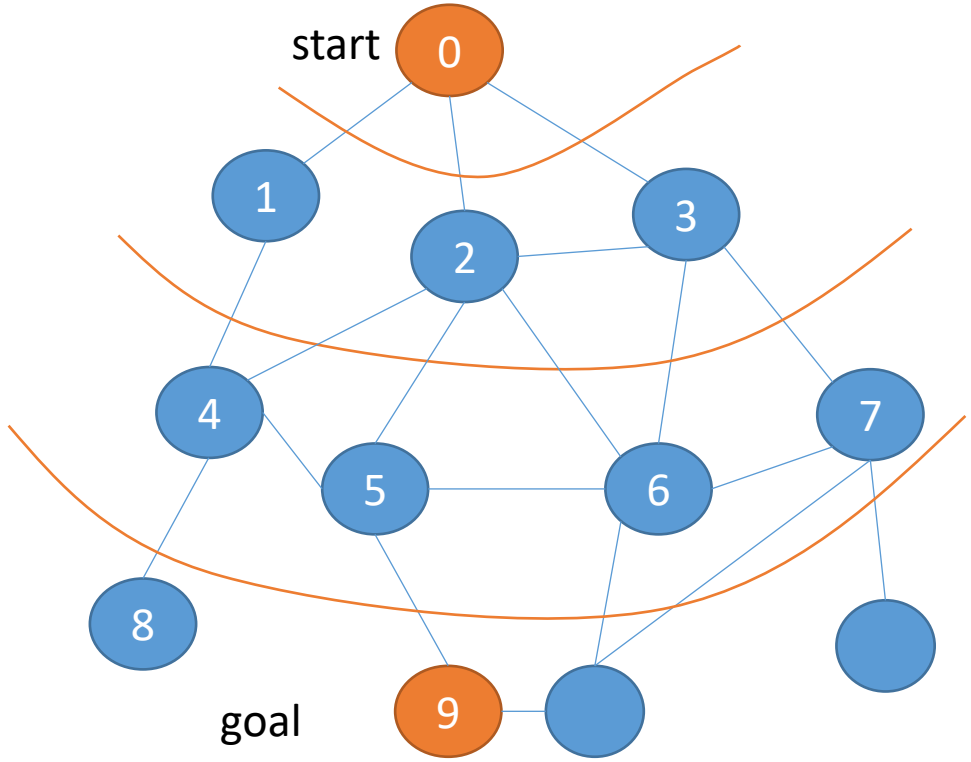
- 問題：ACM ICPC というプログラミングコンテストの2010年国内予選問題（参考 1, 2, AOJ）





幅優先探索という考え方

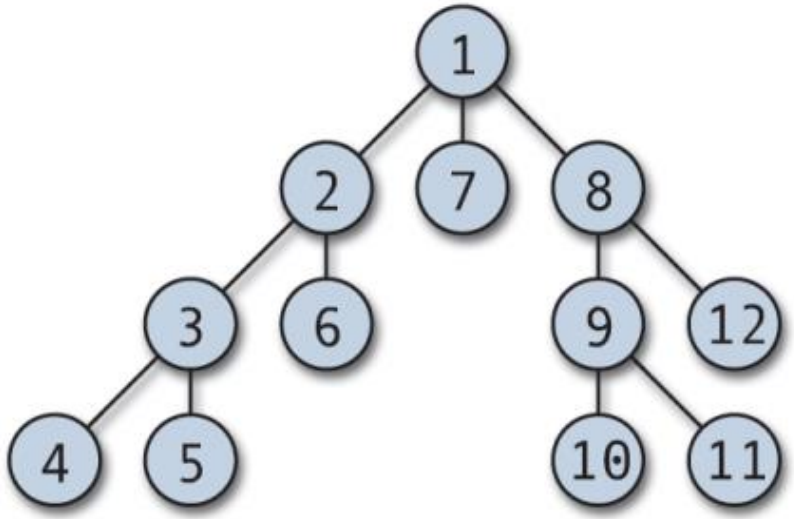
- 開始点から近い順に、隣接点を探索していく方法
  - 一度辿ったところは、再探索しない

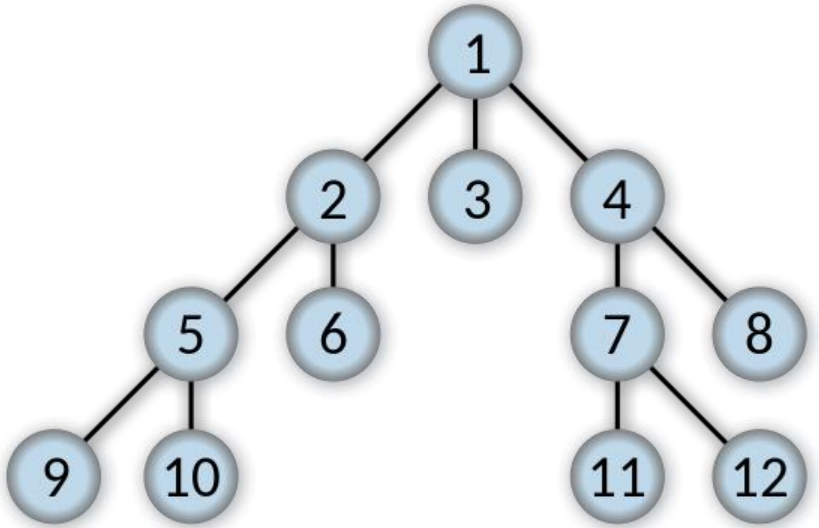



```
def solve() -> int:
    q = deque([ 0 ])
    while q:
        here = q.popleft()
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            q.append(隣接ノード)
    return 0
```

深さ優先探索と幅優先探索

- 深さ優先探索
  - 最近探索した点を優先
  - Stack or 関数再帰呼出し利用
- 幅優先探索
  - 開始点から近い順
  - Queue を利用

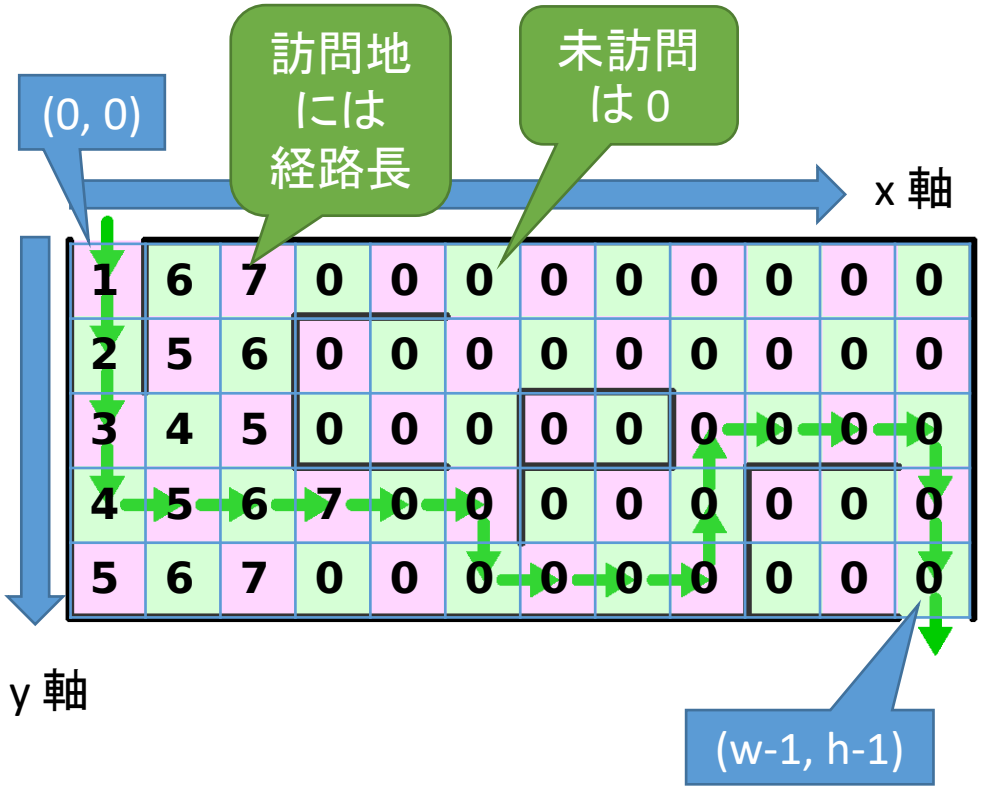




それぞれの図は Wikipedia から引用

幅優先探索してみる

- なにがいるかな？
  - ●各方向に進めるか？
    - ➢壁判定がいりそう
  - ●開始点から順番に探索
    - ➢ queue が便利
  - ●各地点は訪問済み？
    ステップ数の記録は？
    - ➢2次元配列でOK



幅優先探索してみる

プログラム例@github

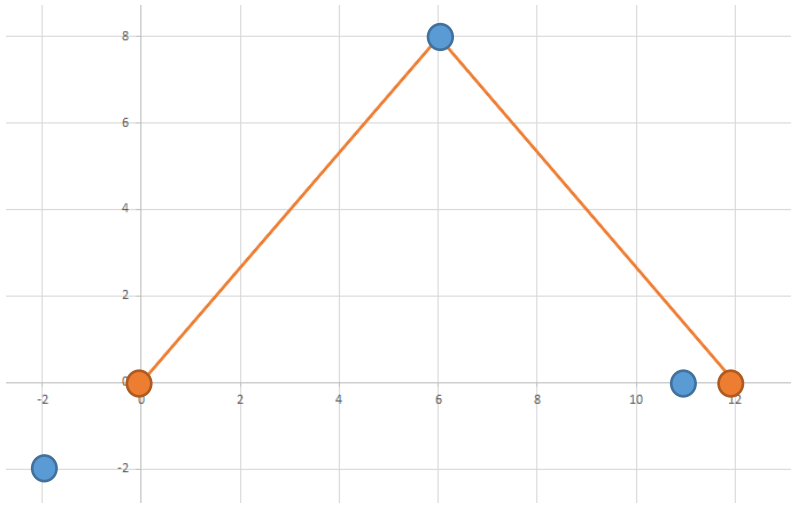
- なにがいるかな？
  - 各方向に進めるか？
    - 壁判定がいりそう
  - 開始点から順番に探索
    - queue が便利
  - 各地点は訪問済み？
    ステップ数の記録は？
    - 2次元配列でOK

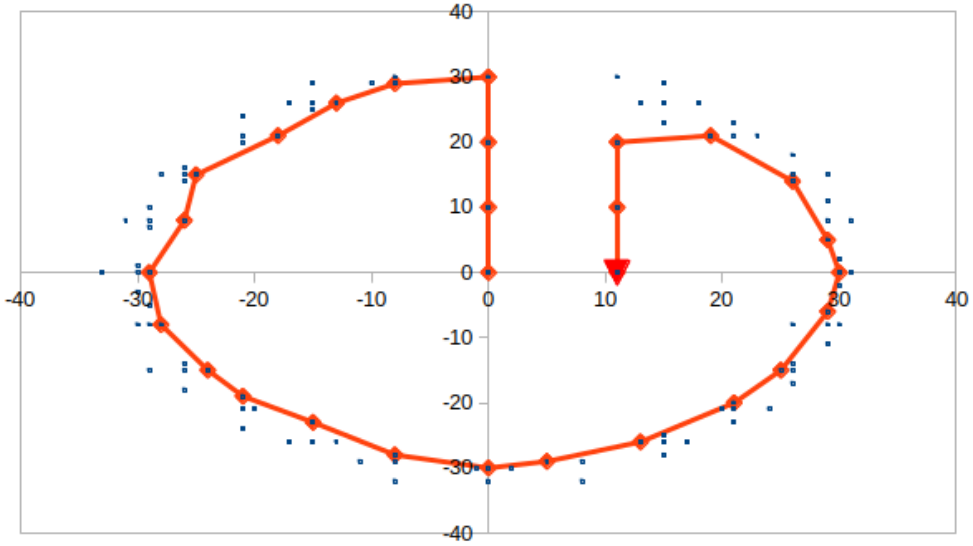
```
def solve() -> int:
    goal = [w - 1, h - 1]
    q = deque([[0, 0]])
    steps = [[0 for _ in range(w)] for _ in range(h)]
    steps[0][0] = 1
    while q:
        x, y = q.popleft()
        current_step = steps[y][x]
        for dx, dy in directions:
            if can_go(x, y, dx, dy):
                nx, ny = [x + dx, y + dy]
                if steps[ny][nx] > 0:
                    continue
                if [nx, ny] == goal:
                    return current_step + 1
                steps[ny][nx] = current_step + 1
                q.append([nx, ny])
    return 0
```

問題2つめ（最短経路問題）

- 盤面上にいくつか点がある
  - ●2点の距離が10以下の場合は移動可能
  - ●始点から終点まで、最短経路で移動したい

最短経路を
求めましょう

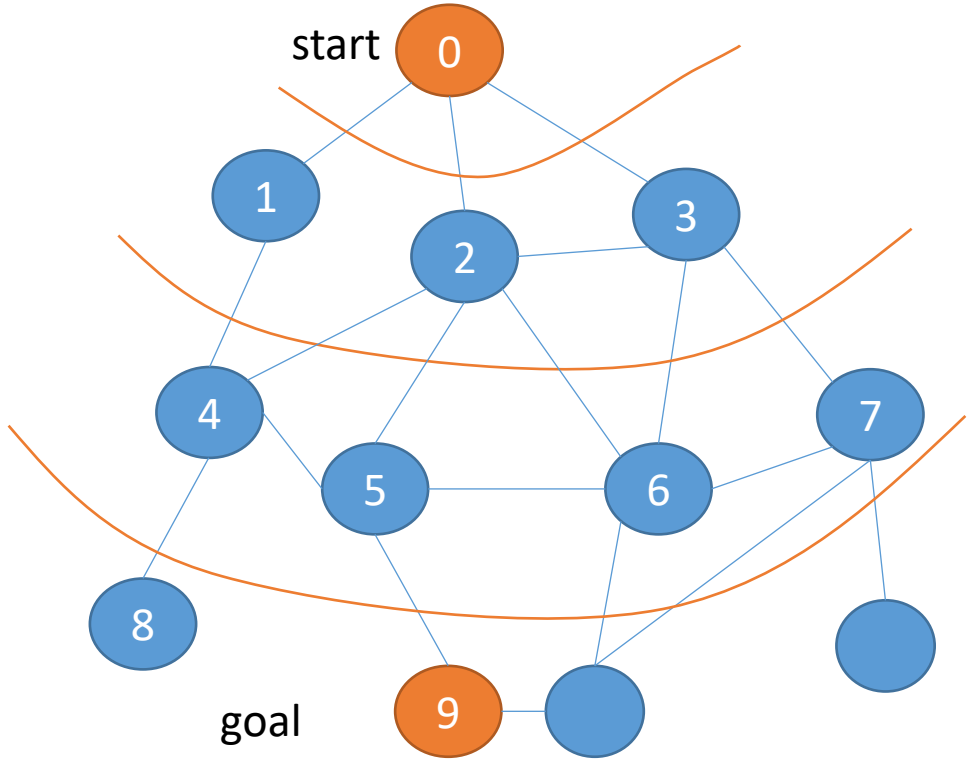




幅優先探索という考え方

再掲

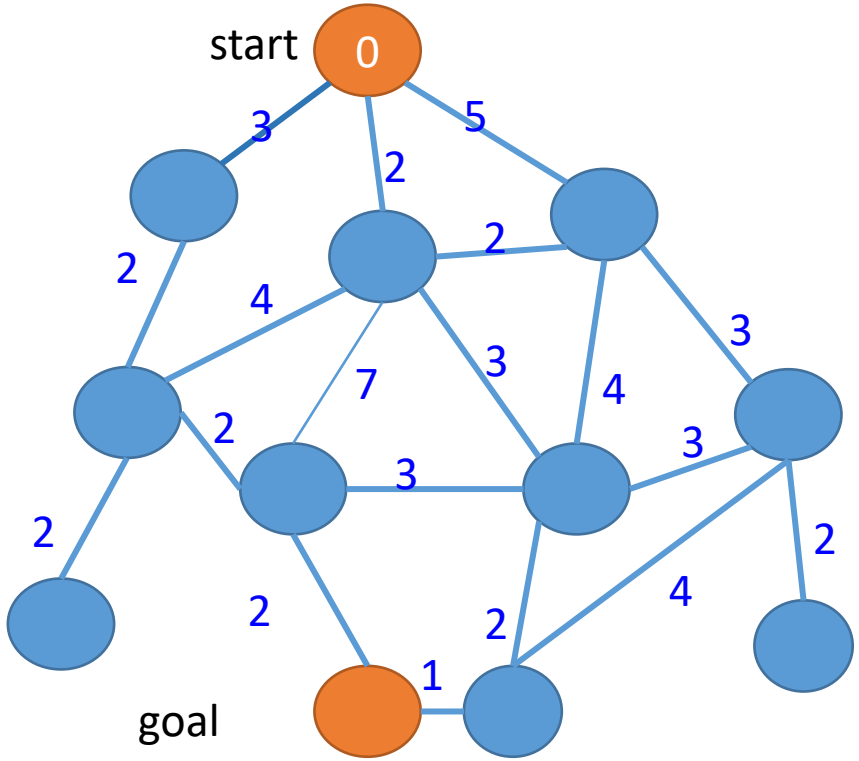
- 開始点から近い順に、隣接点を探索していく方法
  - 一度辿ったところは、再探索しない




```
def solve() -> int:
    q = deque([ 0 ])
    while q:
        here = q.popleft()
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            q.append(隣接ノード)
    return 0
```

今回は枝に長さがある

- 一番近いところを探すのがちょっと面倒？
- でも、近いところからやらないと、何度もやり直し

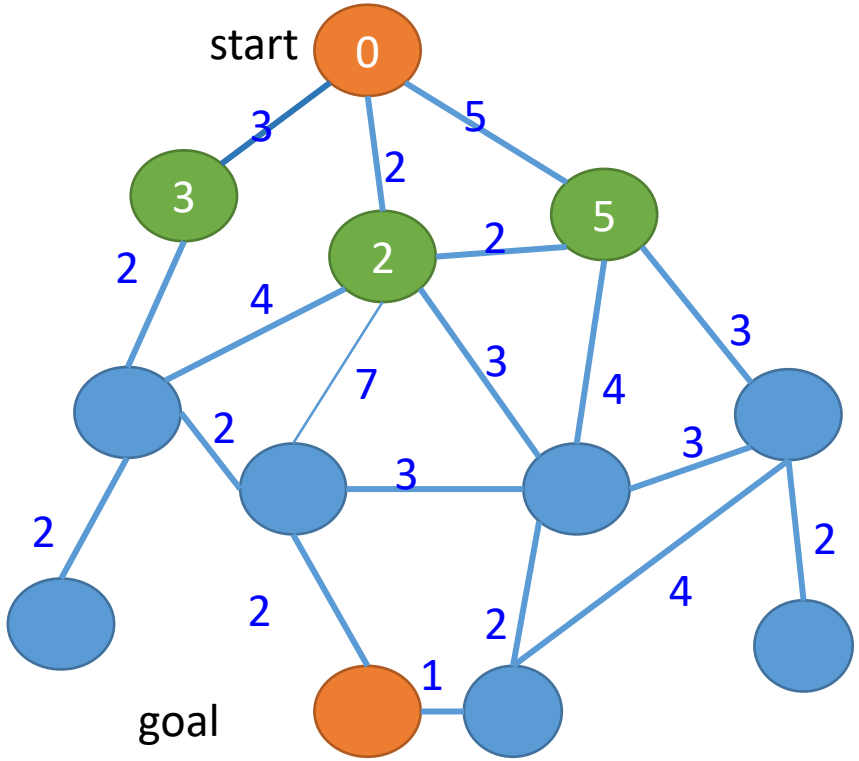



```
def solve() -> int:
    q = deque([ 0 ])
    while q:
        here = q.popleft()
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            q.append(隣接ノード)
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

プログラム例@github

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば

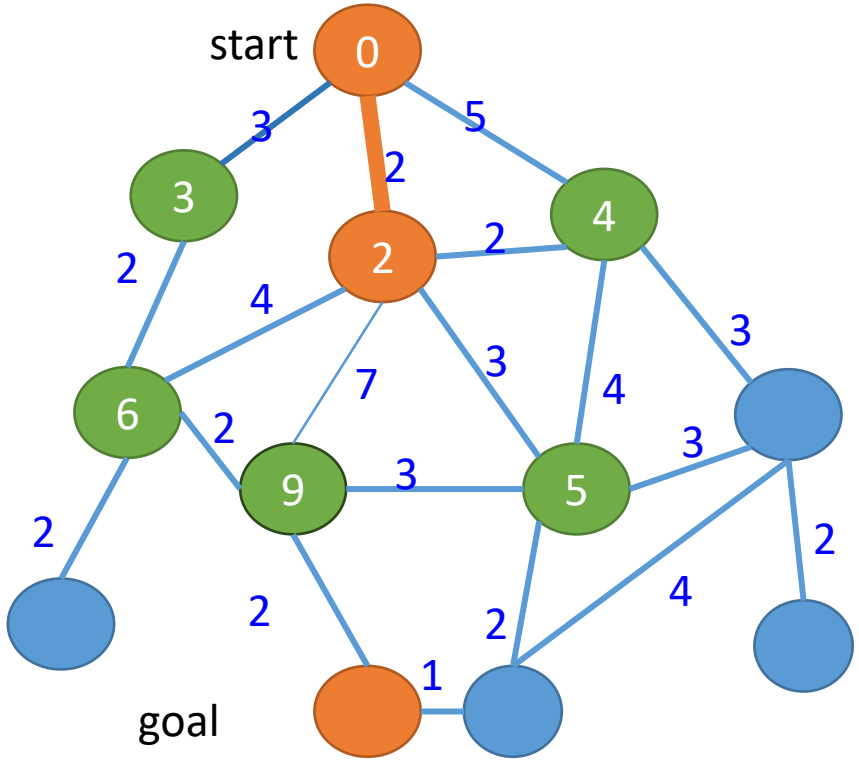


HeapQueue に
「コスト」と「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば

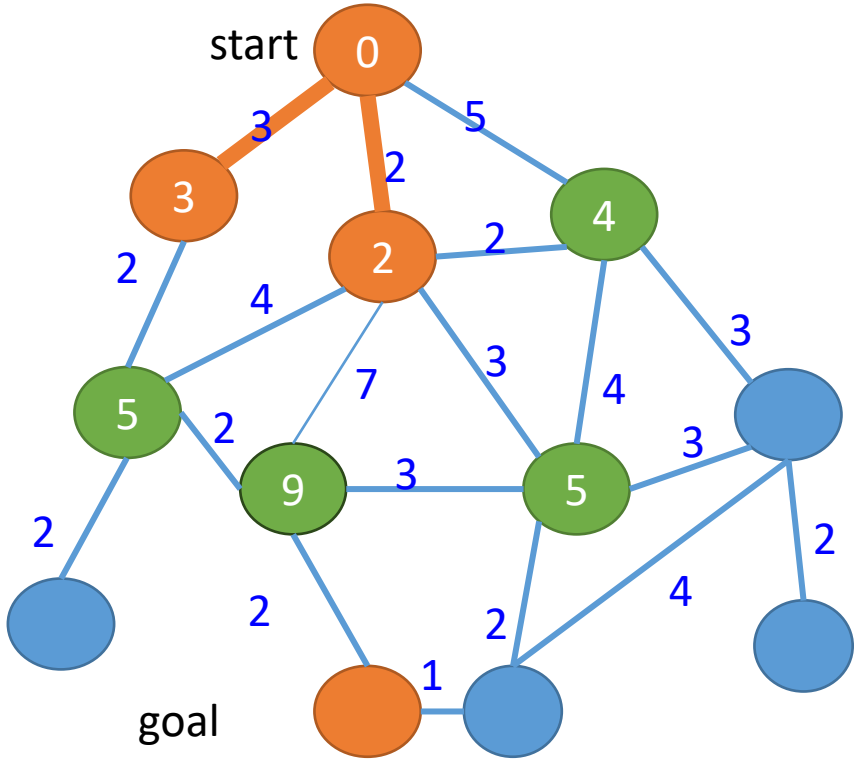


HeapQueue に
「コスト」と「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば

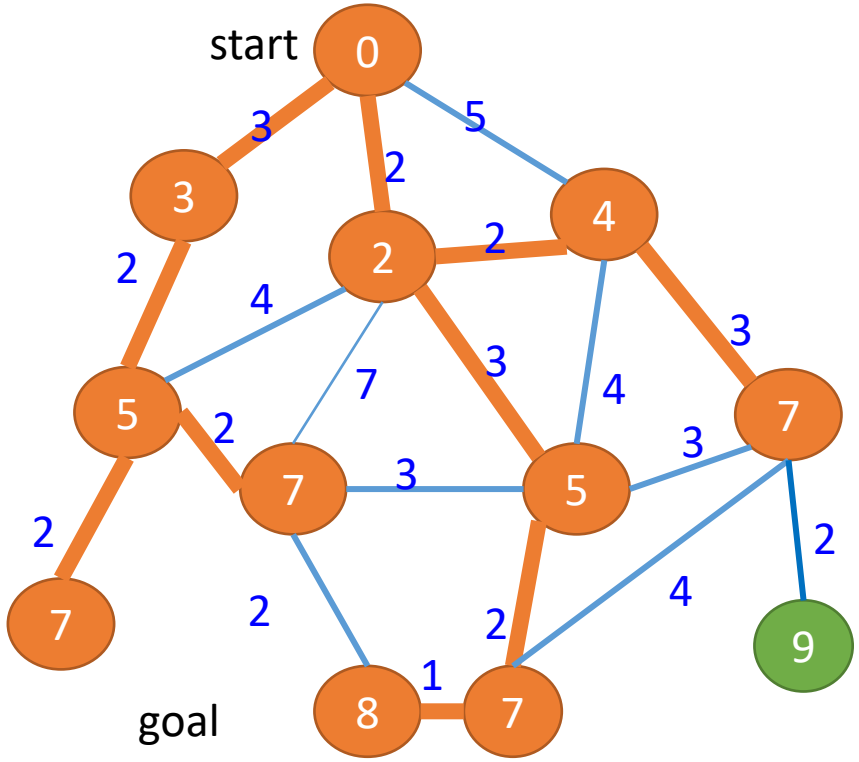


HeapQueue に
「コスト」と「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ダイクストラ法 (Dijkstra's algorithm)

- start から近いところから、確定させていこう
  - queue から、一番近いものが取り出せれば



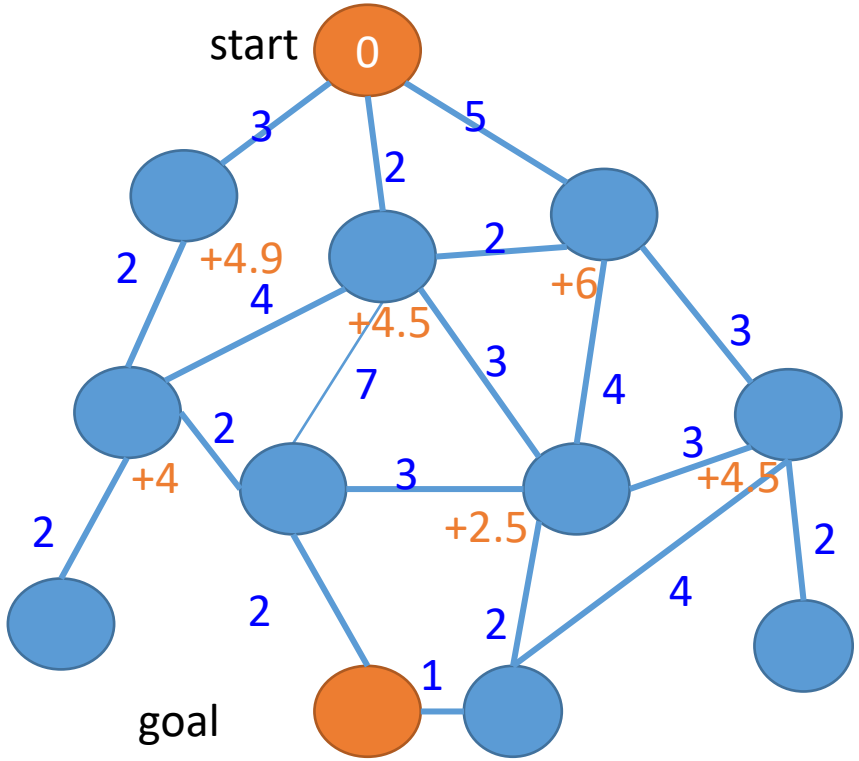
HeapQueue に「コスト」と「ノード」の組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

A* (A-star)アルゴリズム

プログラム例@github

- goal への「見積もり値」を加味して小さい順に
  - 例えば、「goal への直線距離」を利用

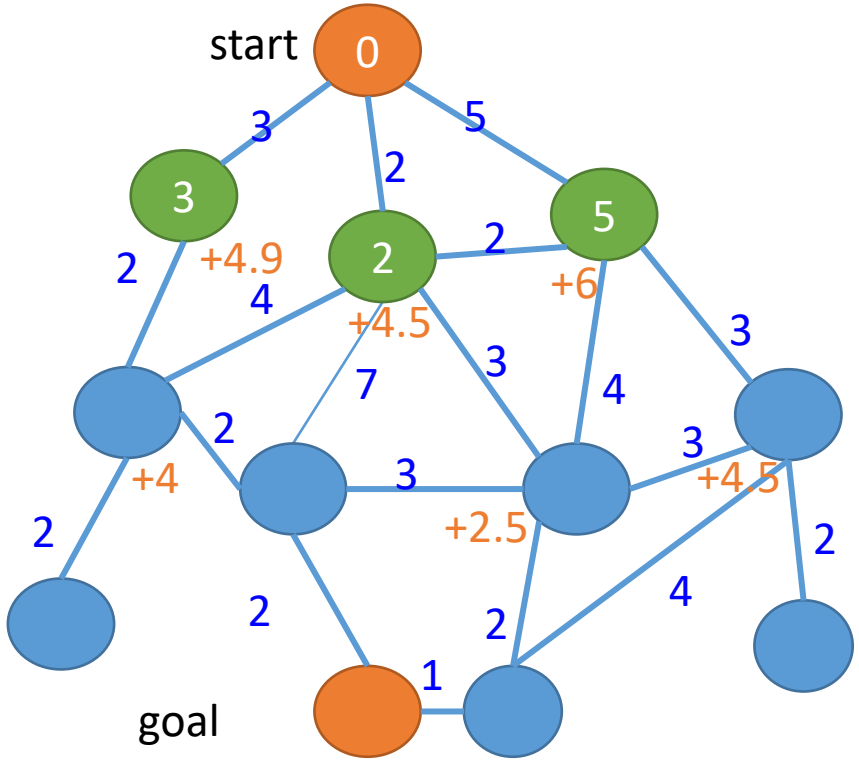


HeapQueue に
「コスト」 と 「ノード」の
組を登録すればOK

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

A* (A-star)アルゴリズム

- goal への「見積もり値」を加味して小さい順に
  - 例えば、「goal への直線距離」を利用

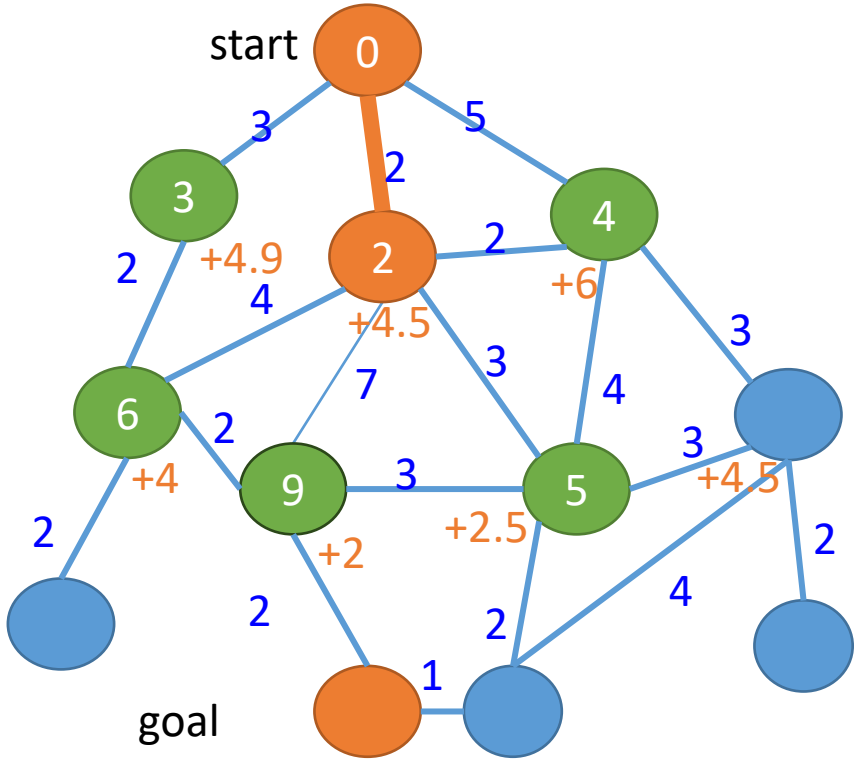


「各ノードまでの経路」＋「goal への見積もり値」で順位付け

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

A* (A-star)アルゴリズム

- goal への「見積もり値」を加味して小さい順に
  - 例えば、「goal への直線距離」を利用

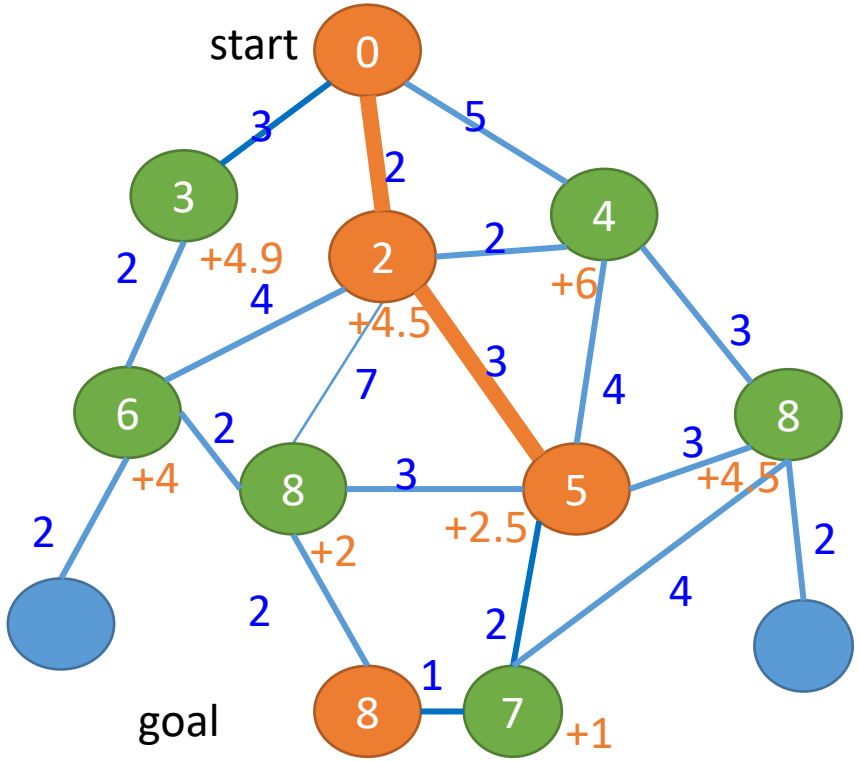


「各ノードまでの経路」＋「goal への見積もり値」で順位付け

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

A* (A-star)アルゴリズム

- goal への「見積もり値」を加味して小さい順に
  - 例えば、「goal への直線距離」を利用

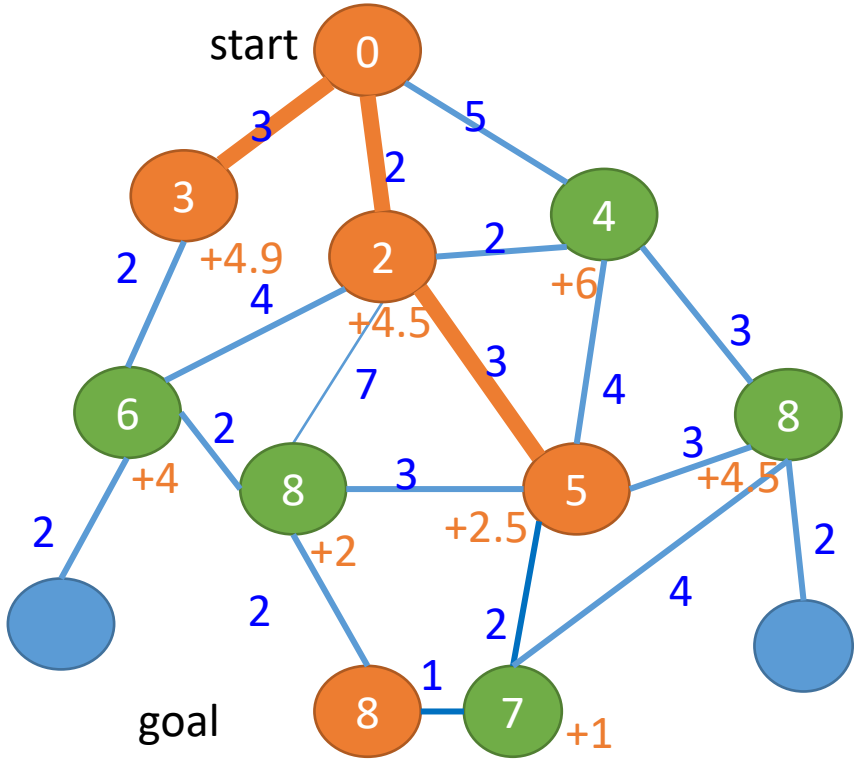


「各ノードまでの経路」＋「goal への見積もり値」で順位付け

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

A* (A-star)アルゴリズム

- goal への「見積もり値」を加味して小さい順に
  - 例えば、「goal への直線距離」を利用

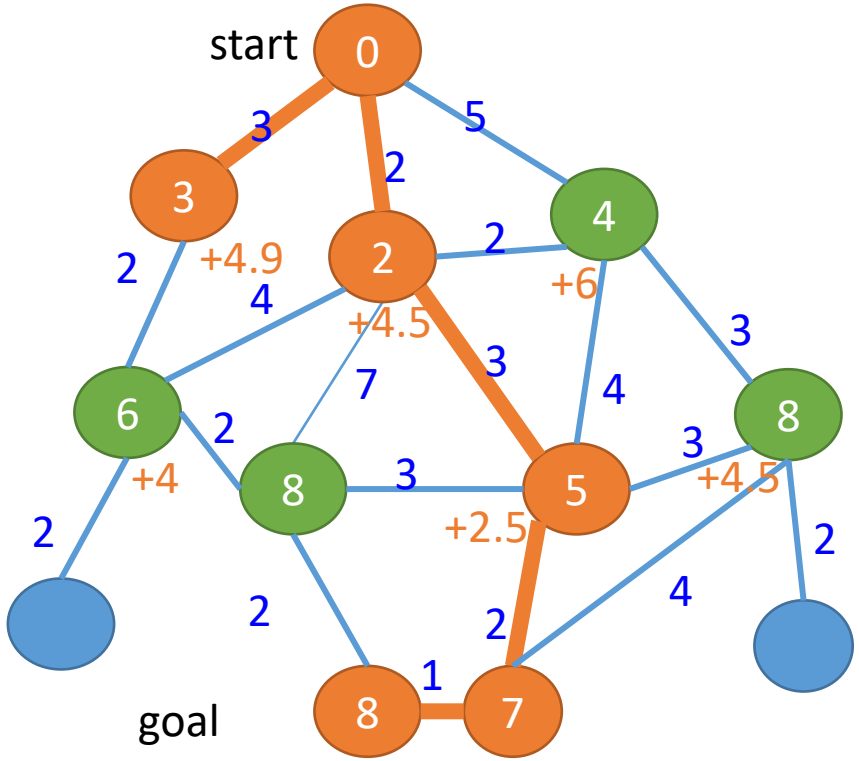


「各ノードまでの経路」＋
「goal への見積もり値」
で順位付け

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

A* (A-star)アルゴリズム

- goal への「見積もり値」を加味して小さい順に
  - 例えば、「goal への直線距離」を利用



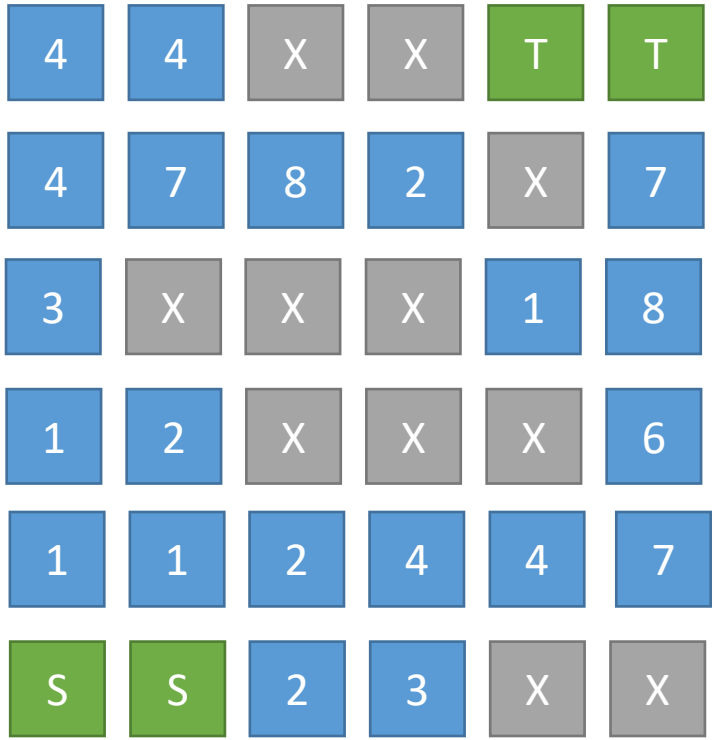
「各ノードまでの経路」＋「goal への見積もり値」で順位付け

```
def solve():
    heappush(q, [コスト0, 0 ])
    while q:
        cost, here = heappop(q)
        if(ゴール？) return 答え
        未訪問の隣接ノードに対し
            heappush(q, [次cost, 隣接ノード])
    return 0
```

ACM ICPC 2007 国内予選

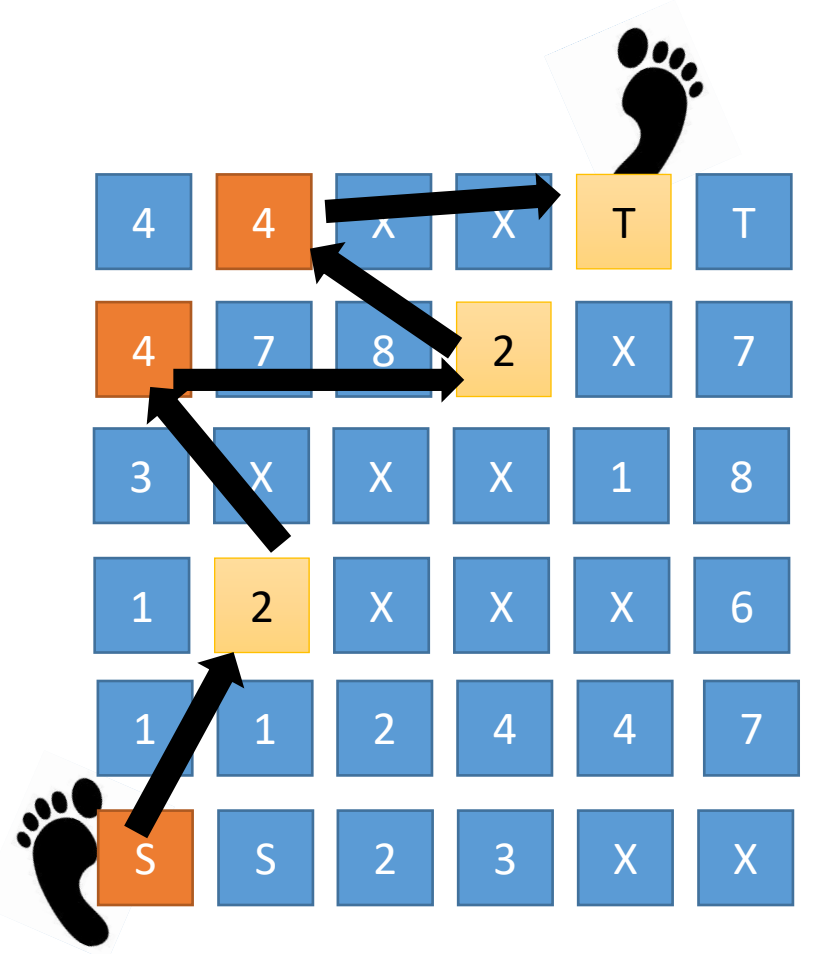
さまざまな最短経路問題

- スタートエリアからゴールエリアに移動
  - 各位置には、左足か右足を置ける
  - 各位置に書かれた分の時間を消費
    - 足を置けない場所もある
  - 次に置くのは逆の足
    - 置ける範囲が決まっている



解き方を考えてみよう

- スタートエリアからゴールエリアに移動
  - 各位置に、左足か右足を置く
  - 各位置にかかれた分の時間を消費
    - 足を置けない場所もある
  - 次に置くのは逆の足
    - 置ける範囲が決まっている
- 考え方？
  - 最短経路っぽい？
  - でも、左足右足って？
  - 同じ場所に両足置くことも？



考え方

プログラム例(C++)
プログラム例(python)

- [足の位置、左／右]をノードとするグラフをイメージすれば？
  - ノードをイメージできれば、最短経路問題