スタートアップゼミ #3

最短経路探索

渡邉 葵

目次

1. モチベーション
2. アルゴリズム
3. 実装に向けて

目標：

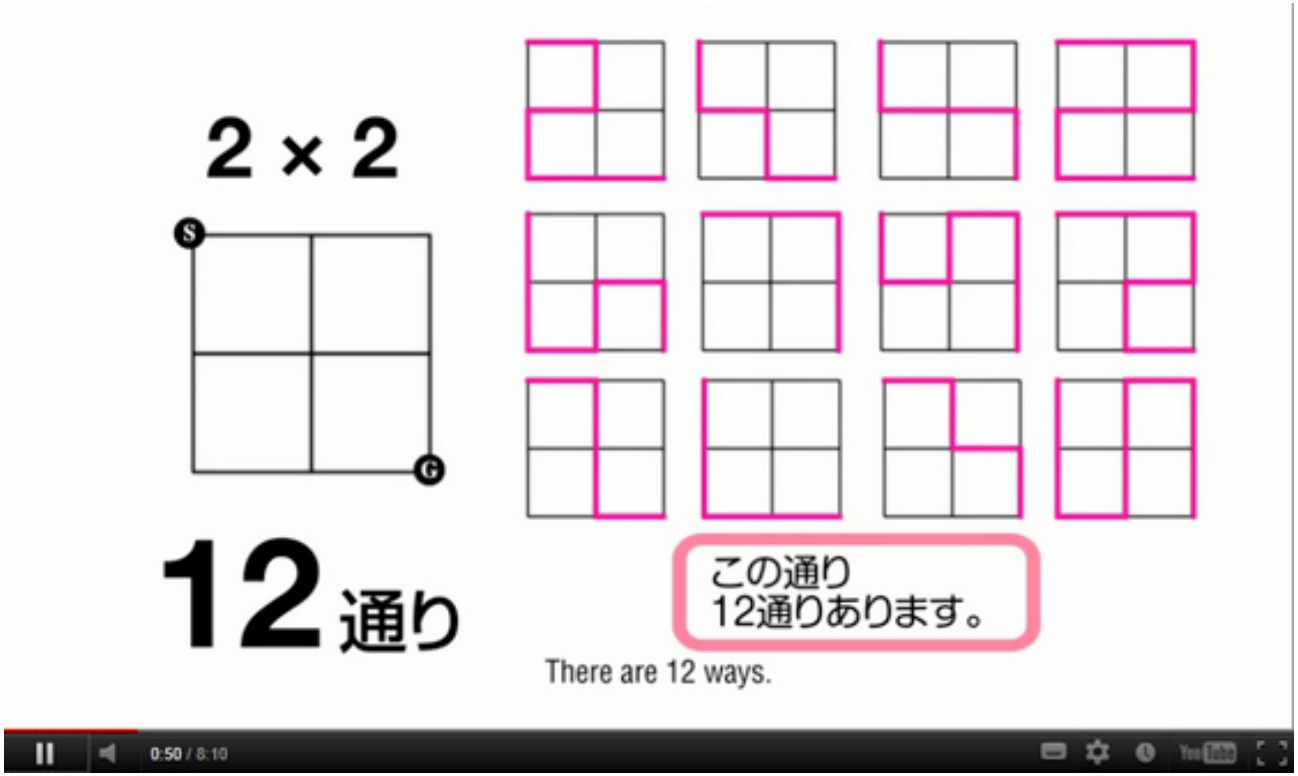
1. 最短経路探索のモチベーションを理解する
2. 代表的なアルゴリズムを理解する
3. 実装できるようになる

1. モチベーション

1. モチベーション

- 組み合わせ爆発

  『フカシギの数え方』 おねえさんといっしょ！ みんなで数えてみよう！
  https://youtu.be/Q4gTV4r0zRs



Q. S→Gへ同じ道を通らずにいく方法は何通り？

1. モチベーション

- 組み合わせ爆発

1×1	2通り	
2×2	12通り	
3×3	184通り	
4×4	8512通り	
5×5	126万2816通り	
6×6	5億7578万564通り	
7×7	7893億6005万3252通り	
8×8	3266兆5984億8698万1642通り	4時間
9×9	4104京4208兆7026億3249万6804通り	6年
10×10	1秭5687垓5803京0464兆7500億1321万4100通り	25万年
11×11	18穣2413秭2915垓1424京8049兆2414億7088万5236通り	290億年

数秒で解ける

1. モチベーション

- 最短経路探索の活用
  - 乗り換え案内
  - 交通量配分（交通量配分の計算の大半は最短経路探索）→ 次回ゼミ

- 問題の定義

  入力：
  - ネットワーク（各リンクがコストを持ったグラフ）
  - 始点と終点

  出力：
  - コスト最小の経路（パス）
  - コスト

2. アルゴリズム

2. アルゴリズム

- 最短経路探索のアルゴリズム：B4より
  - Dijkstra法
  - Bellman-Ford法
  - A*アルゴリズム

3. 実装に向けて

3. 実装に向けて

- 出力結果を可視化したイメージ

3. 実装に向けて

- 入力

link.csv

LinkID	n1	n2	LinkCost
0	1	2	78.2822349
1	1	11	109.11072
2	1	1521	28.12929
3	2	1	78.2822349
4	2	3	52.7219106
5	2	7	37.8627898
6	3	2	52.7219106
7	3	4	106.359096
8	3	7	24.044997
9	4	3	106.359096
10	4	17	56.4702932
11	4	769	20.5772428
12	4	1176	47.5969258
13	5	6	71.2853778
14	5	769	87.6892776
15	5	1157	23.5683397
16	5	1528	39.7667228
17	6	5	71.2853778
18	6	30	52.0246326
19	6	768	52.4046436

node.csv

nodeID	latitude	longitude
1	35.6644695	139.691301
2	35.6643753	139.692158
3	35.6642893	139.69273
4	35.6641289	139.693889
5	35.6644609	139.694998
6	35.6639184	139.695421
7	35.6641801	139.692501
8	35.6639902	139.692683
9	35.6637878	139.692125
10	35.6635943	139.691617
11	35.6634849	139.691311
12	35.6631126	139.691322
13	35.6632303	139.691661
14	35.6634362	139.692293
15	35.6634729	139.69248
16	35.6636697	139.692996
17	35.6638192	139.693393
18	35.6630313	139.692839
19	35.6632522	139.693366

3. 実装に向けて

- 出力

```
node_number = 1612
edge_number = 4832
```

```
Dijkstra法
20 -> 1148
cost = 1347.1851155699997
path : [20, 26, 27, 28, 34, 33, 48, 49, 65, 66, 72, 56, 50, 108, 121, 124, 156, 155, 154, 166, 958,
171, 172, 178, 944, 945, 946, 189, 188, 187, 1148]
calculaton time = 0.037509918212890625
```

```
Bellman-Ford法
20 -> 1148
cost = 1347.1851155699997
path : [20, 26, 27, 28, 34, 33, 48, 49, 65, 66, 72, 56, 50, 108, 121, 124, 156, 155, 154, 166, 958,
171, 172, 178, 944, 945, 946, 189, 188, 187, 1148]
calculaton time = 6.222381114959717
```

```
A*アルゴリズム
20 -> 1148
cost = 1347.1851155699997
path : [20, 26, 27, 28, 34, 33, 48, 49, 65, 66, 72, 56, 50, 108, 121, 124, 156, 155, 154, 166, 958,
171, 172, 178, 944, 945, 946, 189, 188, 187, 1148]
calculaton time = 0.025625228881835938
```

3. 実装に向けて

- Dijkstra法の疑似コード

```
Dijkstra(始点s, 終点t){
  for(すべての頂点v){ D[v] ← +∞, visited[v] ← no, previous[v] ← nil; }
  集合S ← {s}, D[s] ← 0;  //初期化

  while(Sが空でない){
    w ← Sの中でD[w](sからのパス長)が最も小さい頂点w;
    if(w = t){ D[w]を返し，終了;}  //探索終了 //この行を削除すれば全頂点探索
    Sからwを削除;
    visited[w] ← yes;  //wまでの最短路が確定．wは訪問済みとする．

    for(すべてのwの未訪問の隣接頂点x){
      new_dist ← D[w] + d(w,x);  //d(w,x)は(w,x)の辺長
      if(D[x] > new_dist){
        D[x] ← new_dist, previous[x] ← w;
        xがSに入っていなければxをSに加える;
      }
    }
  }
  +∞を返し，終了;  //探索失敗
}
```

D[x]：始点sからxまでの最短経路長(visited[x]=yesの点xについては正しい最短経路長)
visited[x]：頂点xを既に訪問したか否か(最短経路が確定したか否か)
previous[x]：始点sからxへの最短経路におけるxの直前にいた頂点
S：着目頂点に対する隣接頂点のうち最短路がまだ確定していない点の集合

3. 実装に向けて

- Bellman-Ford法の疑似コード

```
Bellman-Ford(始点s){
 for(すべての頂点v){ D[v] ← +∞, previous[v] ← nil; }
 D[s] ← 0;  //初期化

 for(i=1からn-1まで){  //n:頂点数．n-1回繰り返す．
  for(すべての辺e=(u,v)){
   if(D[v] > D[u] + d(u,v)){  //d(u,v)は(u,v)の辺長
    D[v] ← D[u] + d(u,v);
    previous[v] ← u;
   }
  }
 }
}
```

パスの本数がk本以下という制限を持つ最短路の計算を，kを1からn-1まで1ずつ増やしていくことにより最短路を計算．

D[x]：始点sからxまでの最短経路長(visited[x]=yesの点xについては正しい最短経路長)
previous[x]：始点sからxへの最短経路におけるxの直前にいた頂点

3. 実装に向けて

- A*アルゴリズムの疑似コード

```
A_star(始点s, 終点t){
  for(すべての頂点v){ D[v] ← +∞, visited[v] ← no, previous[v] ← nil; }
  集合S ← {s}, D[s] ← 0;   //初期化

  while(Sが空でない){
    w ← Sの中でD[w]+h_t(w)(wを通る最短路長の予測値)が最も小さい頂点w;
    if(w = t){ D[w]を返し，終了;}  //探索終了 //この行を削除すれば全頂点探索
    Sからwを削除;
    visited[w] ← yes;  //wまでの最短路が確定．wは訪問済みとする．

    for(すべてのwの未訪問の隣接頂点x){
      new_dist ← D[w] + d(w,x);  //d(w,x)は(w,x)の辺長
      if(D[x] > new_dist){
        D[x] ← new_dist, previous[x] ← w;
        xがSに入っていなければxをSに加える;
      }
    }
  }
  +∞を返し，終了;  //探索失敗
}
```

h_t(w)：wから終点tまでの経路長の予測値．
D[x]：始点sからxまでの最短経路長(visited[x]=yesの点xについては正しい最短経路長)
visited[x]：頂点xを既に訪問したか否か(最短経路が確定したか否か)
previous[x]：始点sからxへの最短経路におけるxの直前にいた頂点
S：着目頂点に対する隣接頂点のうち最短路がまだ確定していない点の集合

3. 実装に向けて

- 小課題

1. 疑似コードを理解する
2. 疑似コードをもとに実際のコードを書く
3. 他の実装法を試してみる(データの持ち方を変える，言語を変えるとか)

(2)についての補足
正直，先にひと通り基礎的なことを手を動かしながら体系的にやった方が良いように思います(参考：https://sites.google.com/view/ut-python/)．疑似コードをもとに自分でコードを書くのは慣れていないと難しいと思います．詰まってしまったら，配布している実際のコードを見て理解していくのが良いでしょう．print()とかを使って，一行一行，挙動を確認していくのが大切です．デバッグするときもよくやる方法です．補足として，最後に今回の実装で使えそうなTipsをつけときました．理解した疑似コードを実際のコードにするとき，参考にしてみてください．

(3)についての補足
データの持ち方として，疑似コードのものはリンクデータを保持するやり方ですが，隣接行列を作ってやるやり方もあります．Driveに参考として入れた昨年のデータは隣接行列を用いた方法になっています．
また，Pythonはfor文のループを重ねたりすると計算時間が増えやすいそうです．むしろモジュールとかを使った方が早いらしい．Explicitに書くならC系で書く方がやはり早い．

Debug Day
渡邉担当　4/24(金) 13:00～15:00
須賀担当　4/26(日) 午前

補足1：ヒープ構造

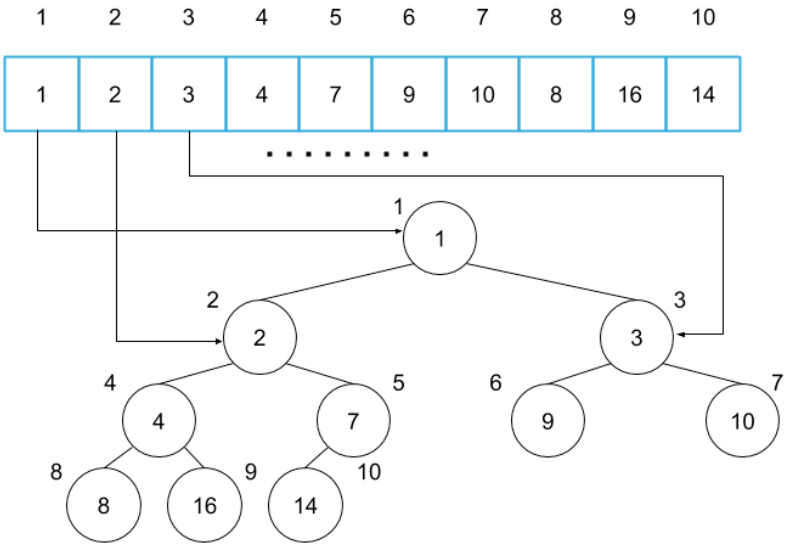
- 最小値の探索

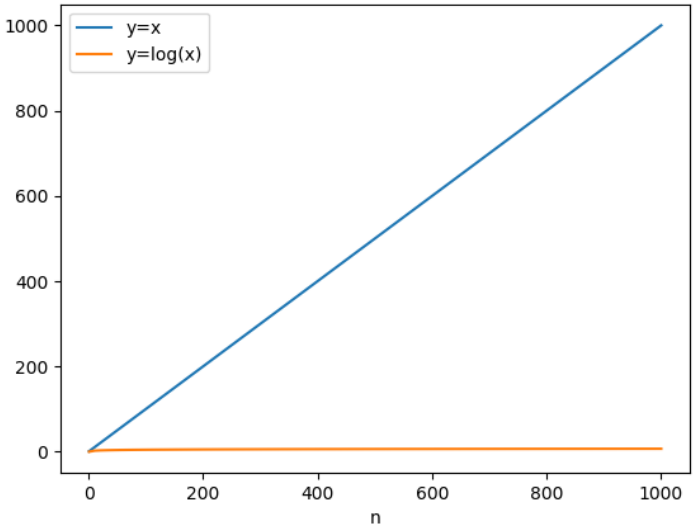
「Sの中でD[w](sからのパス長)が最も小さい頂点wを探す」という操作（疑似コード6行目）

→ 配列の中から最小値を探すアルゴリズム

何も考えずにやると…$O(n)$
ヒープでデータを保持していれば…$O(\log n)$

ヒープ条件：
$a[i] \leq a[2i], a[i] \leq a[2i+1]$





↓極めてわかりやすい（ヒープをわかりやすく解説してみた）
https://medium.com/@yasufumy/data-structure-heap-ecfd0989e5be

補足2：使えそうなTips

■ リストへの操作

append()：末尾に要素を追加

insert()：指定位置に要素を追加

remove()：指定した値と同じ要素を検索し，最初の要素を削除

set()：重複する値の削除．list()を重ねることでリストに戻せる

```
l = list(range(3))
print(l)
# [0, 1, 2]

l.append(100)
print(l)
# [0, 1, 2, 100]
```

```
l = list(range(3))
print(l)
# [0, 1, 2]

l.insert(0, 100)
print(l)
# [100, 0, 1, 2]
```

```
l = ['Alice', 'Bob', 'Charlie', 'Bob', 'Dave']
print(l)
# ['Alice', 'Bob', 'Charlie', 'Bob', 'Dave']

l.remove('Alice')
print(l)
# ['Bob', 'Charlie', 'Bob', 'Dave']
```

```
l = [3, 3, 2, 1, 5, 1, 4, 2, 3]

print(set(l))
# {1, 2, 3, 4, 5}

print(list(set(l)))
# [1, 2, 3, 4, 5]
```

補足：使えそうなTips

■ Numpy配列の用意

np.zeros()：0で初期化

np.full()：任意の値で初期化

```
import numpy as np

print(np.zeros(3))
# [ 0. 0. 0.]

print(np.full(3, 100))
# [100 100 100]
```

■ 最小値の取得

np.argmin()：最小値となる要素のインデックス（位置）を取得

```
import numpy as np

a = np.array([1, 100, 10])
print(a)
# [ 1 100 10]

print(np.argmin(a))
# 0
```

■ 条件を指定して抽出

np.where()：条件に一致するデータのインデックスを抽出

```
import numpy as np

a = np.array([
    [1, 1, 0],
    [1, 0, 1],
    [0, 1, 0],
    [0, 0, 1],
    [1, 1, 1],
])

# 最終列が1のデータのみ取得
index = np.where(a[:, 2] == 1)

print(a[index])
# array([[1, 0, 1],
#        [0, 0, 1],
#        [1, 1, 1]])
```