2021夏学期ゼミ#2

最短経路探索とPythonの導入

M1　鈴木 大樹

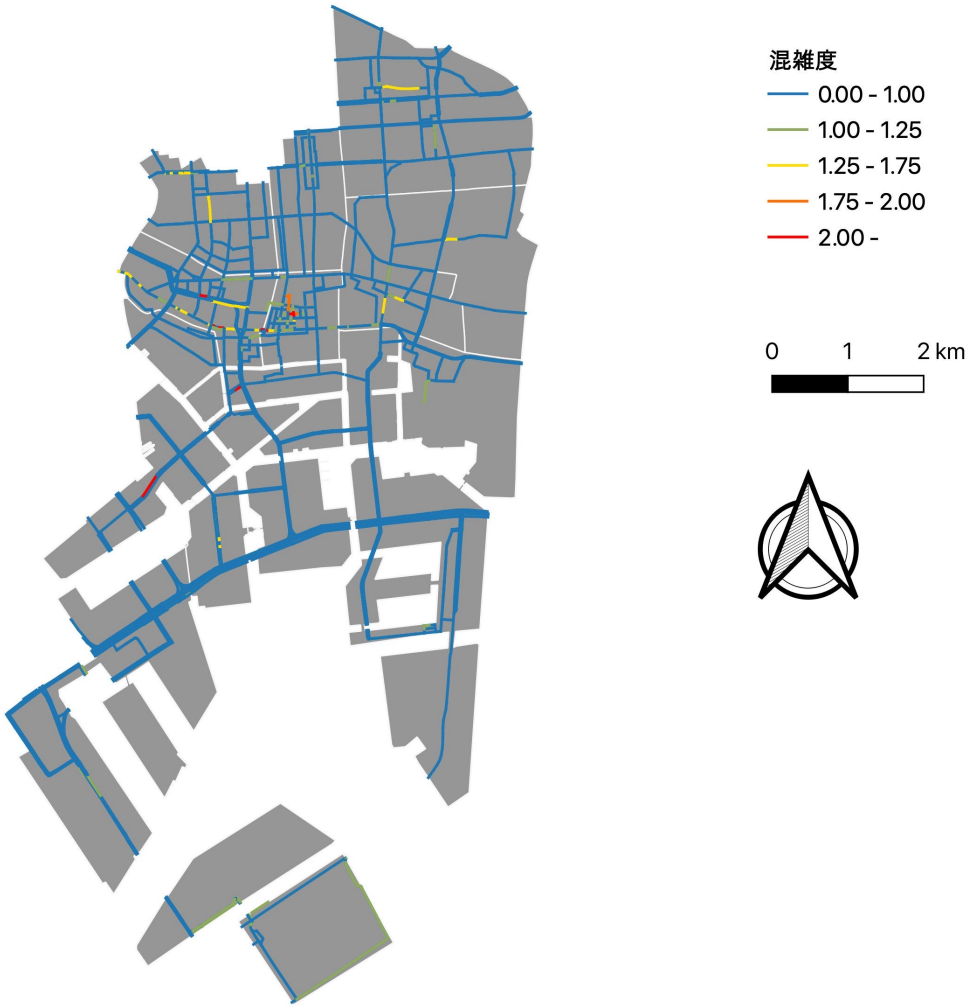
目次

1. 最短経路探索アルゴリズムの必要性

1. 最短経路探索アルゴリズムの必要性

乗り換え案内

交通量配分



1. 最短経路探索アルゴリズムの必要性

最も単純には，経路を全て列挙して，その中から最短の経路を探せば良いが…

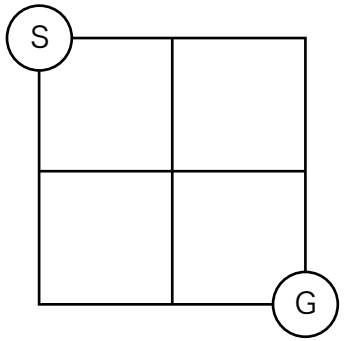
組み合わせ爆発の問題

『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

1. 最短経路探索アルゴリズムの必要性

1×1	2通り
2×2	12通り
3×3	184通り
4×4	8,512通り
5×5	1,262,816通り
6×6	575,780,564通り
7×7	789,360,053,252通り
8×8	3,266,598,486,981,642通り
9×9	41,044,208,702,632,496,804通り
10×10	1,568,758,030,464,750,013,214,100通り
11×11	182,413,291,514,248,049,241,470,885,236通り

⋮



現実的な計算時間で経路を列挙することは不可能
→アルゴリズムに工夫が必要

2. 最短経路探索アルゴリズム

2-1. Dijkstra法

■Dijkstra法のアルゴリズム

① 発生ノードに永久ラベル$u_0^* = 0$を与える．$i = 0$とする．

② ノードiを始点に持つ，全てのリンクl_{ij}に対して，終点jが永久ラベルを持っていなければ，
   i. ノードjが一時ラベルを持っていないならば，一時ラベル$u_j = u_i^* + d_{ij}$を与える．
   ii. ノードjが一時ラベルu_jを持っており，その一時ラベルu_jが$u_i^* + d_{ij}$よりも大きいとき，この一時ラベルを$u_i^* + d_{ij}$の値に改訂する．

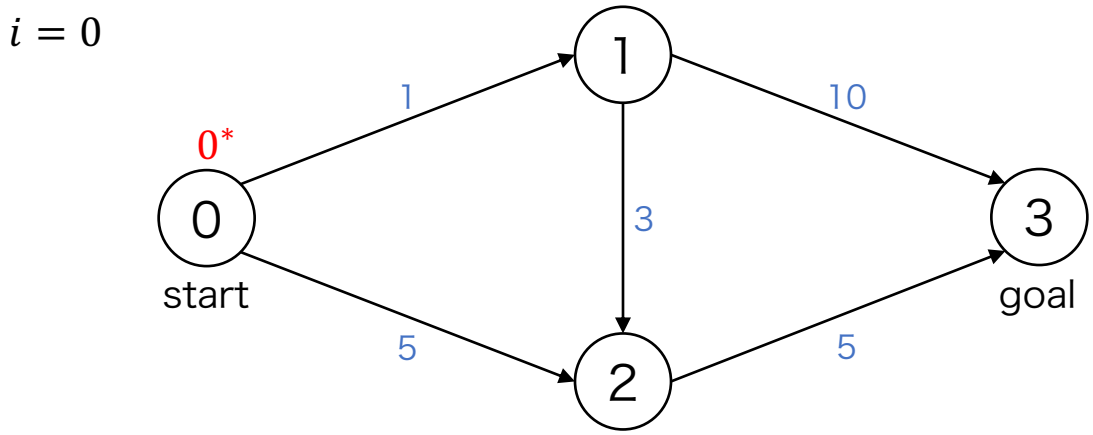
③ 一時ラベル全体の中で最小の値を持つもの1つ選び，iをそのノード番号とする．また，一時ラベルu_iを永久ラベルu_i^*に変える．

④ すべての集中ノードに永久ラベルがつくまで，②・③を繰り返す．

2-1. Dijkstra法

■Dijkstra法のアルゴリズム

① 発生ノードに永久ラベル$u_0^* = 0$を与える．$i = 0$とする．

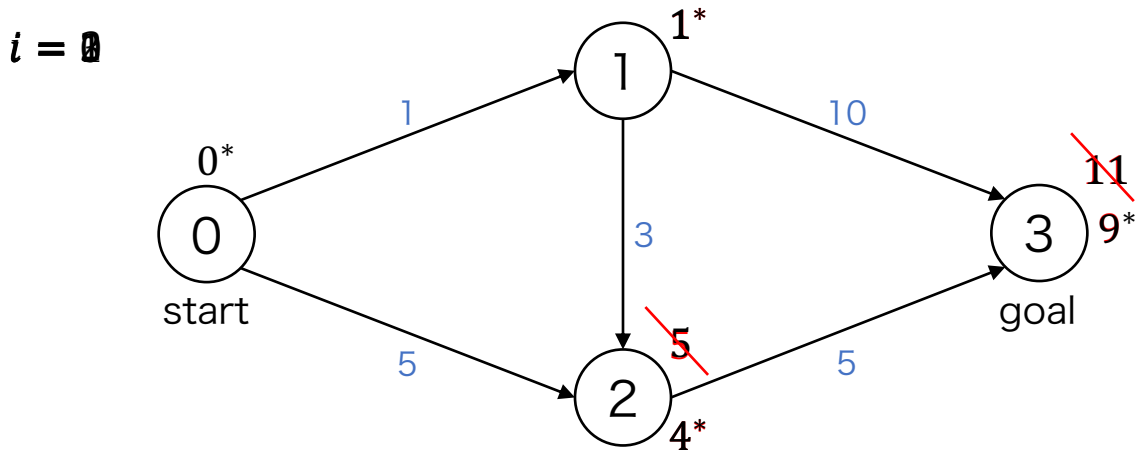


2-1. Dijkstra法

■Dijkstra法のアルゴリズム

② ノードiを始点に持つ，全てのリンクl_{ij}に対して，終点jが永久ラベルを持っていなければ，
   i. ノードjが一時ラベルを持っていないならば，一時ラベル$u_j = u_i^* + d_{ij}$を与える．
   ii. ノードjが一時ラベルu_jを持っており，その一時ラベルu_jが$u_i^* + d_{ij}$よりも大きいとき，この一時ラベルを$u_i^* + d_{ij}$の値に改訂する．

③ 一時ラベル全体の中で最小の値を持つもの1つ選び，iをそのノード番号とする．また，一時ラベルu_iを永久ラベルu_i^*に変える．



2-1. Dijkstra法

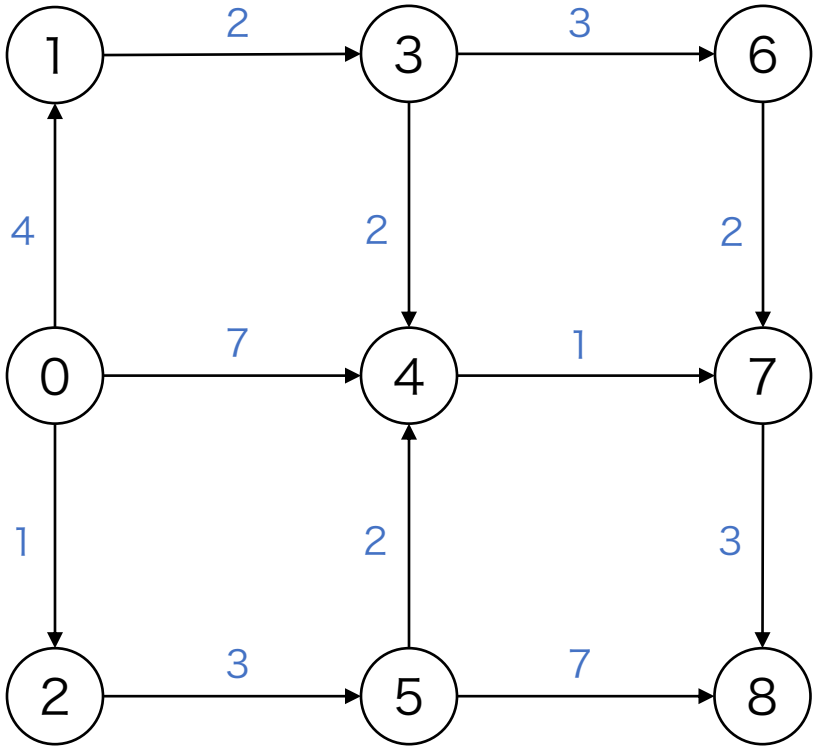
Dijkstra法＝動的計画法的アルゴリズム

- 「手近で明らかなことから順次確定していき，その確定した情報をもとに更に遠くまで確定していく」

2-1. Dijkstra法

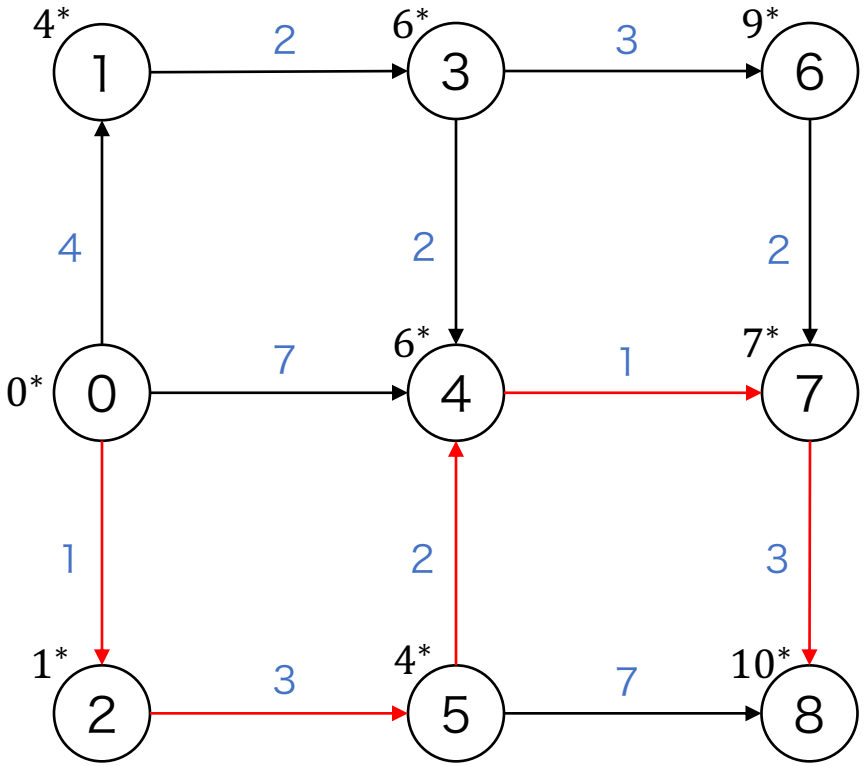
■演習問題

以下に示すネットワークについて，ノード0からノード8までの最小経路コストを求めてください.



2-1. Dijkstra法

■解答



2-2. A*アルゴリズム

■概要とヒューリスティックコスト

A*アルゴリズム = Dijkstra法 + ヒューリスティック関数

- あるノードnを通る最短経路のコスト$f^*(n)$を考える際に，スタートからのコスト$g^*(n)$と，ゴールまでのコスト$h^*(n)$の和を考える．

$$f^*(n) = g^*(n) + h^*(n)$$

探索の過程で近づけていくことができる
→動的計画法的＝Dijkstra法

ゴールに辿り着くまで分からない

人間が設計した推定値$h(n)$を与える！

ヒューリスティック関数

- ヒューリスティック関数には，マンハッタン距離やユークリッド距離がよく用いられる．
  - ✓ マンハッタン距離$h(n) = x(n) + y(n)$

    ノードnからゴールまでのx座標差　y座標差
  - ✓ ユークリッド距離$h(n) = \sqrt{\{x(n)\}^2 + \{y(n)\}^2}$

2-2. A*アルゴリズム

■A*アルゴリズム

① グラフ・スタートノード・ゴールノード・オープンリスト・クローズリストを用意する.

② スタートノードをオープンリストに入れる.

③ オープンリストでトータルコストfが最も低いノードnを取り出す．取り出したノードnがゴールのとき，ノードnをクローズリストに入れて終了.

④ 取り出したノード n に隣接するすべてのノード m について，以下の操作を行う.

i. コスト$f(m), g(m), h(m)$を計算する.
ii. ノードmが以下の条件に当てはまれば，親をnとしてオープンリストに入れる.
- そのノードmが両リストにない.
- mがどちらかのリストにあり，計算した$f(m)$がリスト中の$f(m)$よりも小さい場合

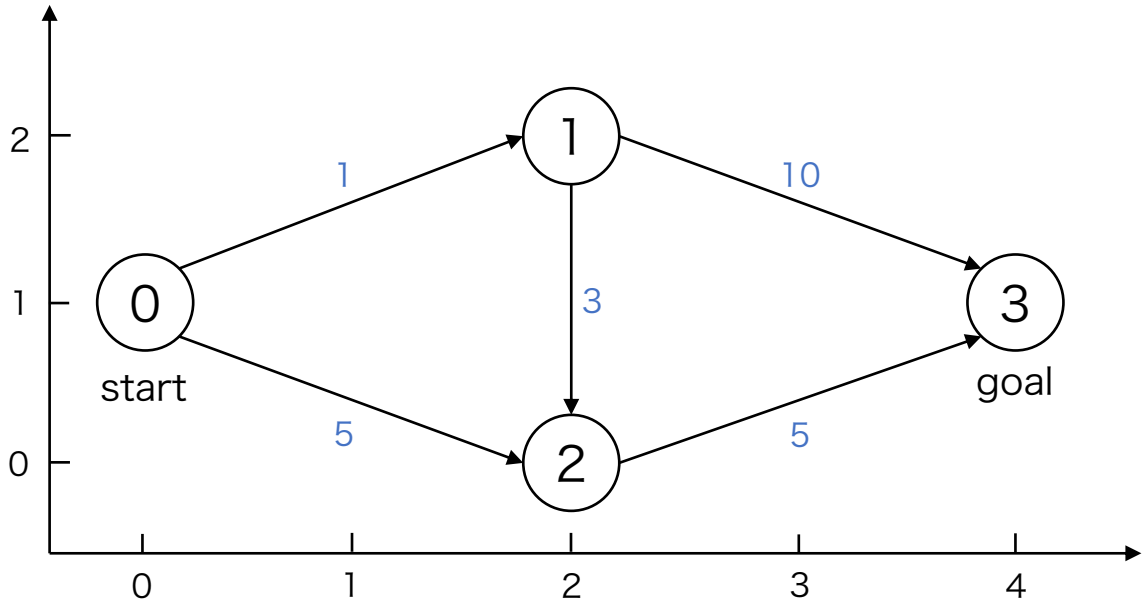
⑤ 取り出したノードnをクローズリストに入れる.

⑥ ③〜⑤を繰り返す.

2-2. A*アルゴリズム

■A*アルゴリズム

① グラフ・スタートノード・ゴールノード・オープンリスト・クローズリストを用意する.



計算中のノード

ノード	$g(n)$	$h(n)$	$f(n)$	親

取り出したノード

ノード	$g(n)$	$h(n)$	$f(n)$	親

オープンリスト

ノード	$g(n)$	$h(n)$	$f(n)$	親

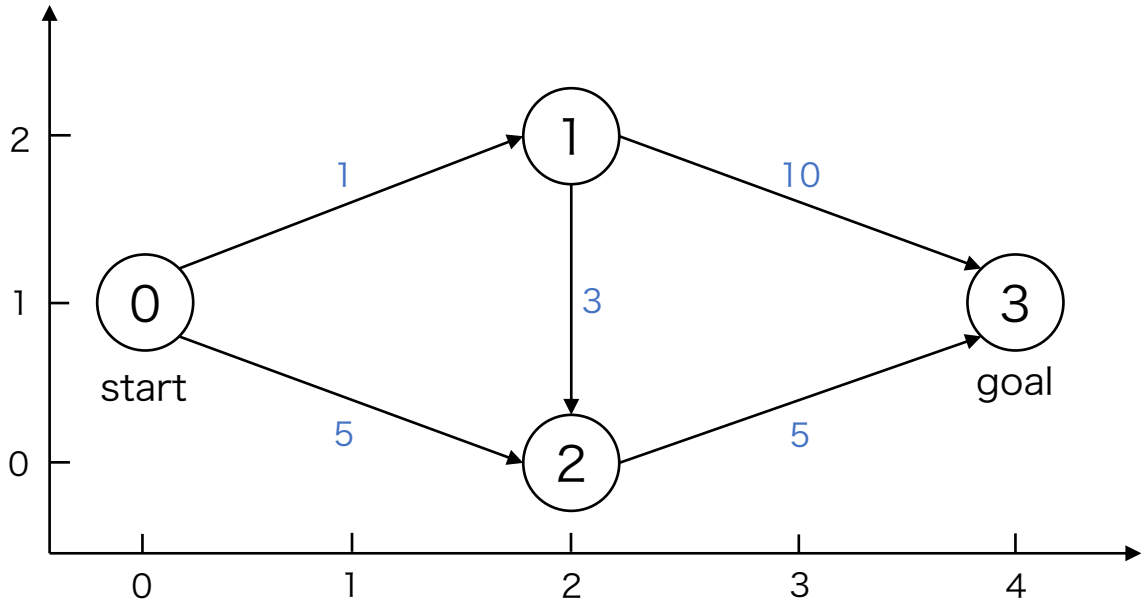
クローズリスト

ノード	$g(n)$	$h(n)$	$f(n)$	親

2-2. A*アルゴリズム

■A*アルゴリズム

② スタートノードをオープンリストに入れる.



計算中のノード

ノード	$g(n)$	$h(n)$	$f(n)$	親

取り出したノード

ノード	$g(n)$	$h(n)$	$f(n)$	親

オープンリスト

ノード	$g(n)$	$h(n)$	$f(n)$	親
0	0	4	4	-

クローズリスト

ノード	$g(n)$	$h(n)$	$f(n)$	親

2-2. A*アルゴリズム

■A*アルゴリズム

③　オープンリストでトータルコストfが最も低いノードnを取り出す．取り出したノードnがゴールのとき，ノードnをクローズリストに入れて終了．

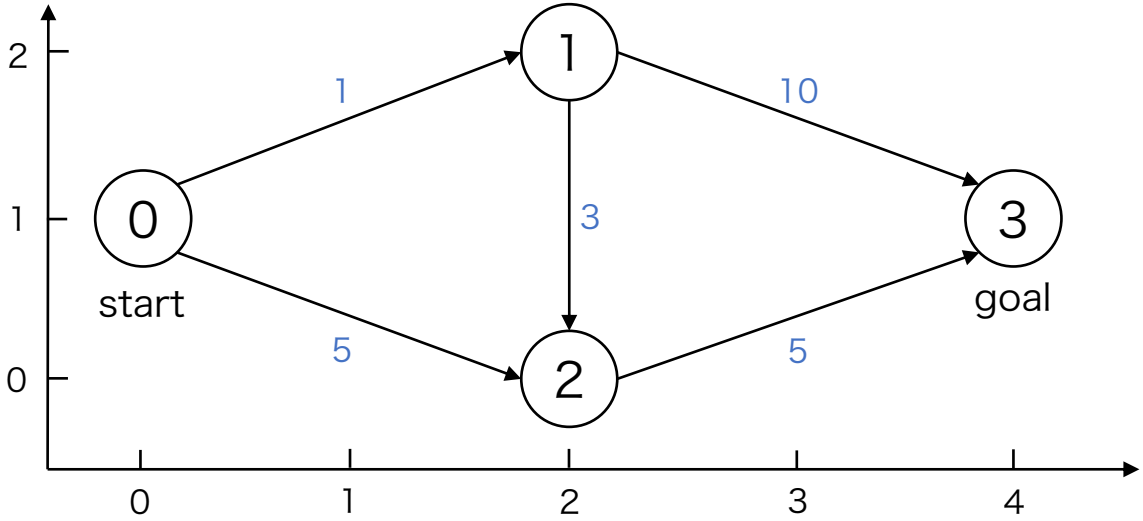
④　取り出したノードnに隣接するすべてのノードmについて，以下の操作を行う．

i.　コスト$f(m), g(m), h(m)$を計算する．

ii.　ノードmが以下の条件に当てはまれば，親をnとしてオープンリストに入れる．

- そのノードmが両リストにない．
- mがどちらかのリストにあり，計算した$f(m)$がリスト中の$f(m)$よりも小さい場合

⑤　取り出したノードnをクローズリストに入れる．



計算中のノード

ノード	$g(n)$	$h(n)$	$f(n)$	親
2	15	0	15	0

取り出したノード

ノード	$g(n)$	$h(n)$	$f(n)$	親
0	0	4	4	-

オープンリスト

ノード	$g(n)$	$h(n)$	$f(n)$	親
1	1	3	4	0
2	15	0	15	0

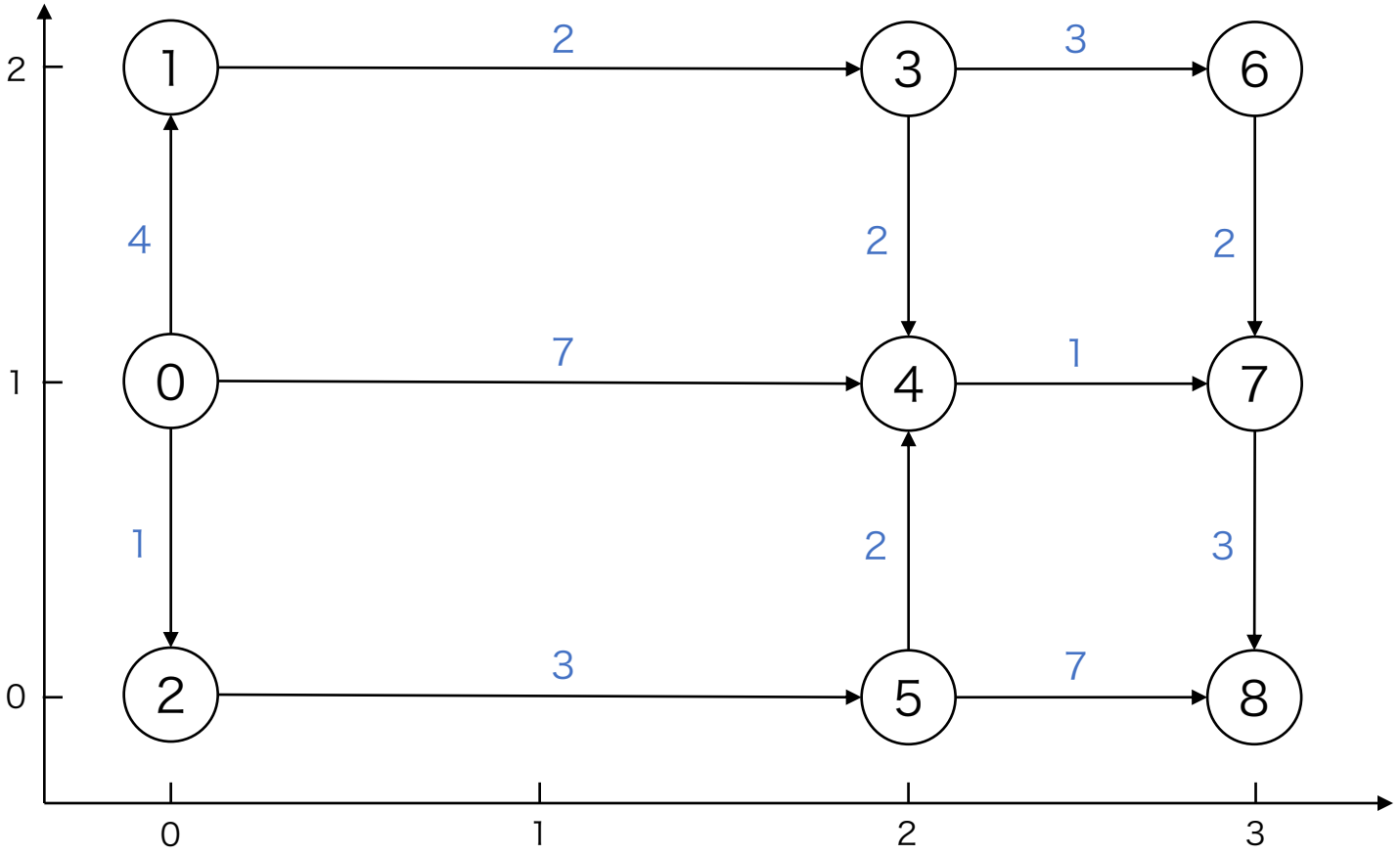
クローズリスト

ノード	$g(n)$	$h(n)$	$f(n)$	親
0	0	4	4	-
1	1	3	4	0
2	4	3	7	1
3	9	0	9	2

2-2. A*アルゴリズム

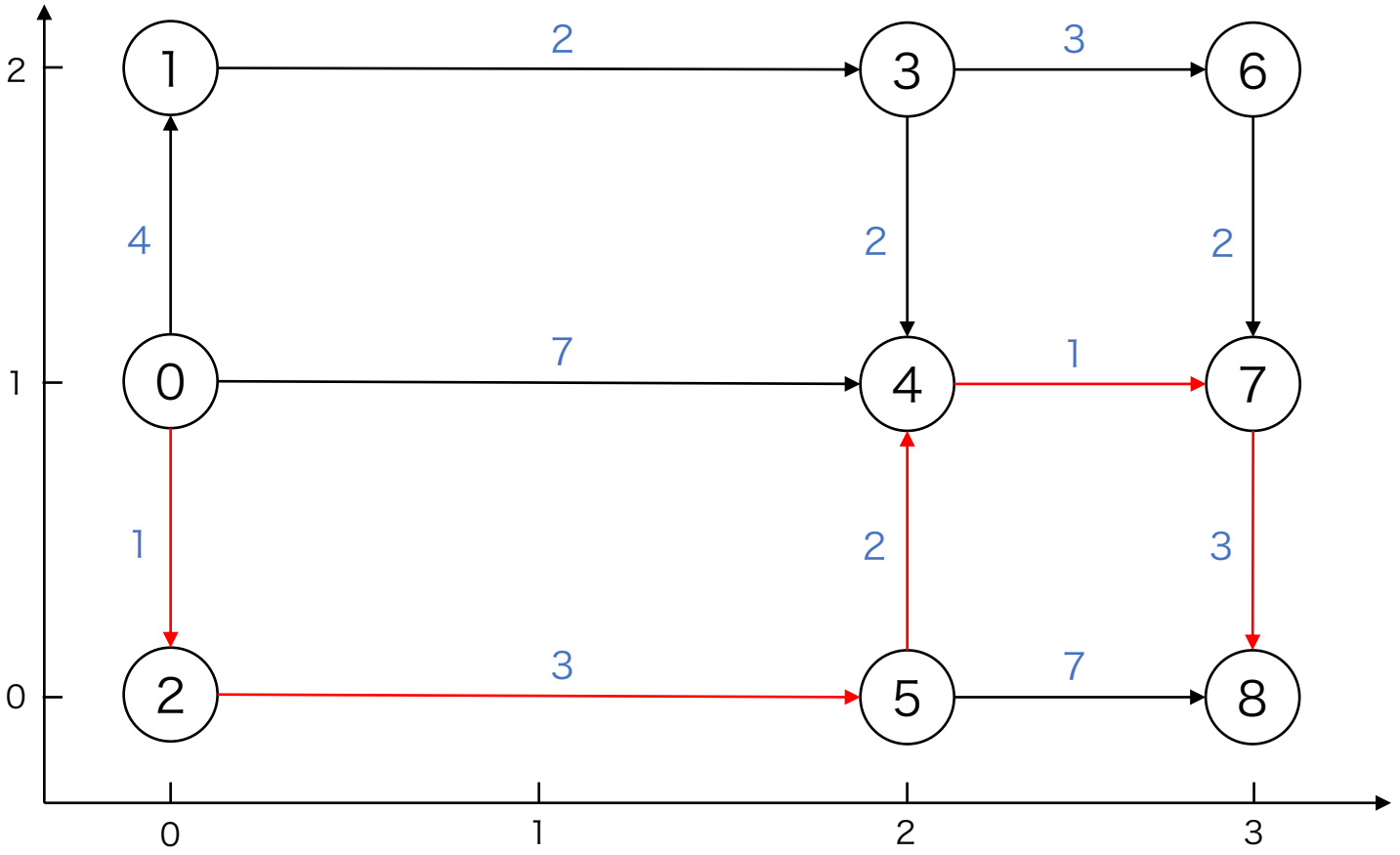
■演習問題

以下に示すネットワークについて，ノード0からノード8までの最小経路コストを求めてください.



2-2. A*アルゴリズム

■解答



2-3. その他のアルゴリズム

■Bellman-Ford法

- 各頂点に与えられたコストを，より小さい値で置き換えていくアルゴリズム.
- コストが負の値の辺が含まれていても計算可能（Dijkstra法，A*アルゴリズムでは不可能）.
- Dijkstra法・A*アルゴリズムよりも計算量が多い.

参考： https://nw.tsuda.ac.jp/lec/BellmanFord/など

■ZDD（Zero-suppressed binary Decision Diagram）

- 省略や節点の共有により，ネットワークを圧縮した表現方法.
- ZDDを用いたアルゴリズムにより，経路の高速列挙が可能.
- 詳細は6月8日（火）の小川さんの会で.

3. 実装に向けて

3-1. Pythonの導入

■Pythonとは

- Google3大言語の1つ（Java，Python，C++）.
- コードがシンプルで扱いやすく，設計・可読性が高い.
- 標準ライブラリや関数が豊富で大規模なツール.

3-1. Pythonの導入

■リスト

- 複数の要素をまとめたオブジェクト.

[1, ‘Suzuki’, 3.14, True]
int型　str型　float型　bool型

- リストの要素へのアクセス：リスト[インデックス/スライス式]
- リストの要素の変更　　：リスト[インデックス] = 変更後の値
- リストの要素数を得る　　：len(リスト)
- リストの末尾に要素を追加：リスト.append(要素)
- リストから要素を削除　　：リスト.remove(要素)

3-1. Pythonの導入

```python
# リスト

# リストの例
L = [1, 'Suzuki', 3.14, True]

# リストの要素へのアクセス
print(L[1])

# リストの要素の変更
L[2] = 3.1415
print(L[2])

# 空リストもある
S = []

# 要素数を得る
print(len(S))

# リストの末尾に要素を追加
L.append('Eiji')
print(L)

# リスト内包表記
V = [i for i in range(1, 10) if i % 2 == 0]
print(V)
```

```
Suzuki
3.1415
0
[1, 'Suzuki', 3.1415, True, 'Eiji']
[2, 4, 6, 8]
```

3-1. Pythonの導入

■関数

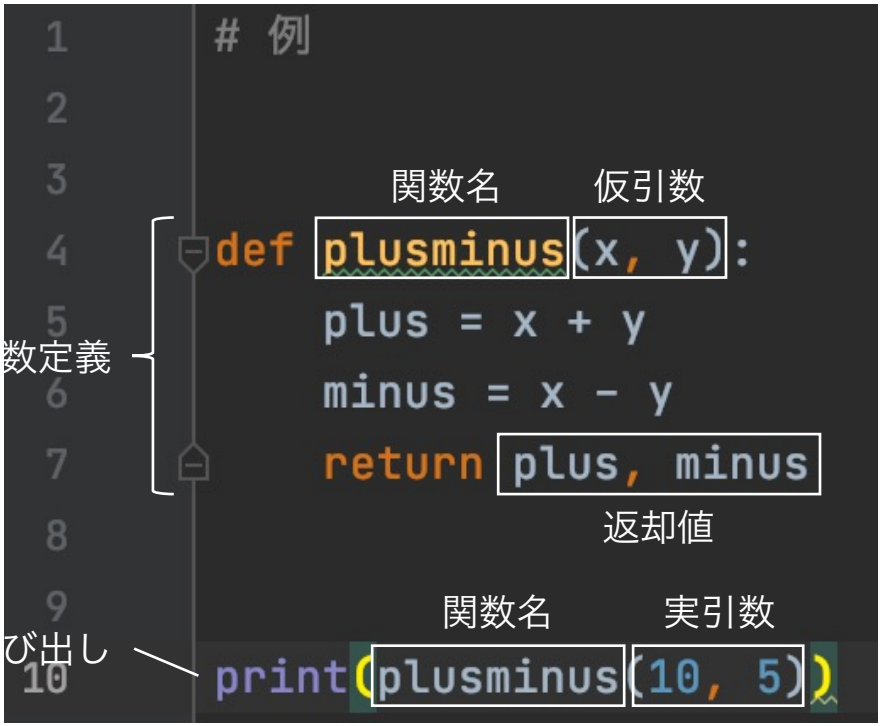
- プログラムの「部品」.
- 引数に補助的な指示を受け取って,
  目的の処理を行い, 処理した結果を
  返却値として返す.

○関数定義

```
def 関数名(仮引数1, 仮引数2, …):
    関数本体
    return 返却値
```

○関数呼び出し

```
関数名(実引数1, 実引数2, …)
```




```
(15, 5)

Process finished with exit code 0
```

3-1. Pythonの導入

■関数：演習

2変数x，yの積及びxのy乗の値を求める関数を定義し，x = 2，y = 3のときの結果を表示してください.

ヒント：aとbの積はa * b，aのb乗はa ** b.

3-1. Pythonの導入

■関数：演習（解答例）

```python
# 関数：解答例


def prod_expo(x, y):
    prod = x * y
    expo = x ** y
    return prod, expo


print(prod_expo(2, 3))
```

```
(6, 8)

Process finished with exit code 0
```

3-1. Pythonの導入

■for文・while文

- 繰り返し処理に用いる.

for 変数 in range(開始, 終了)/リスト等:
    繰り返したい処理

- 変数にrangeやリスト内の値が1つずつ代入されて処理が実行される.

while (判定式):
    繰り返したい処理

- 判定式がTrueである限り処理が行われる.

- for文やwhile文の中に更にfor文やwhile文が入ることもある（多重ループ）.

```
1   # for文
2   sum1 = 0
3   for i in range(1, 11):
4       sum1 += i
5
6   print('for文：', sum1)
7
8   # while文
9   sum2 = 0
10  i = 1
11  while (i <= 10):
12      sum2 += i
13      i += 1
14
15  print('while文：', sum2)
```

開始 終了
変数
判定式

```
for文： 55
while文： 55

Process finished with exit code 0
```

3-1. Pythonの導入

■if文

- 条件分岐に用いる.

```
if 条件:
    処理
elif 条件:
    処理
elif 条件:
    処理
    …
else:
    処理
```

- elif以下は適宜省略可能.

```python
# if文

# 学年を入力
year = 'B4'

# メンバーを表示
if year == 'B4':
    print('近藤・増橋・村橋・望月')
elif year == 'M1':
    print('小島・鈴木・月田・前田・増田')
elif year == 'M2':
    print('新井・小川・小関・沈・須賀・黛・渡邉')
else:
    print('入力が不正です. ')
```

```
近藤・増橋・村橋・望月

Process finished with exit code 0
```

3-1. Pythonの導入

■for文・while文・if文：演習

if文を用いて，以下の通り条件分岐して処理をするプログラムを作成してください.

- 変数methodに0が指定されたとき

→for文を二重に使って右の表を表示する.

ヒント

- print()はデフォルトで末尾に改行を出力する．末尾に出力する文字を変更するには，print(~, end=…)で指定する.

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

- 変数methodに1が指定されたとき

→1から9までの積をfor文で求めて，appendメソッドを用いてリストresultの末尾に結果を逐次追加する．ただし，while文を用いて，リストの要素数が10になったら停止する．作成されたリストを表示する.

- 変数methodに0・1以外が指定されたとき→「指定が不正です．」と表示する.

3-1. Pythonの導入

■for文・while文・if文：演習（解答例）

```
1   method = 0
2
3   if method == 0:
4       for i in range(1, 10):
5           for j in range(1, 10):
6               print(i * j, end=' ')
7           print()
8   elif method == 1:
9       result = []
10      while (len(result) <= 9):
11          prod = 1
12          for i in range(1, 10):
13              prod *= i
14          result.append(prod)
15      print(result)
16  else:
17      print('入力が不正です. ')
```

3-1. Pythonの導入

■NumPy

- 配列処理系の拡張モジュール.
- 多次元配列の数値データを扱う.
- 計算速度が速い.

- 使用するにはインポートが必要. 以下のように別名をつけてインポートすることが多い.

```
import numpy as np
```

3-1. Pythonの導入

■NumPy（ndarray）

- ndarray：多次元配列オブジェクト

- ndarrayの生成
  - ✓ np.array(リスト等)：リスト等からndarrayを生成
  - ✓ np.full(要素数, 値)：指定された要素数・値のndarrayを生成
  - ✓ np.zeros(要素数)：指定された要素数で，全ての要素が0のndarrayを生成

- ndarrayの要素へのアクセス
  - ✓ 1次元なら，ndarray[インデックス]
  - ✓ 2次元なら，ndarray[インデックス, インデックス]
  - ✓ 要素へのアクセスにはスライス式（開始:終了）も用いることができる

- 最小の値を持つ要素のインデックスを求める
  - ✓ np.argmin(配列等)：指定された配列等の最小要素のインデックスを返す

- 条件に応じた処理
  - ✓ ndarray[np.where(条件)]：条件を満たす要素を取り出す

3-1. Pythonの導入

■NumPy（ndarray）

```
1  # ndarray
2
3  import numpy as np
4
5  # ndarrayの生成
6  a = np.full(10, 5)
7  b = np.zeros(10).astype(int)
8  print(a)
9  print(b)
10
11 list1 = [9, 2, 4, 1, 6, 3, 7]
12 c = np.array(list1)
13 print(c)
14
15 # 2次元配列
16 list2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
17 d = np.array(list2)
18 print(d)
```

```
[5 5 5 5 5 5 5 5 5 5]
[0 0 0 0 0 0 0 0 0 0]
```

```
[9 2 4 1 6 3 7]
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

3-1. Pythonの導入

■NumPy（ndarray）

```
20  # 要素へのアクセス
21  print(c[3])
22  print(d[1])
23  print(d[1, 2])
24  print(d[0:2, 2])
25  print(d[:, 2])
26
27  # 最小の値を持つインデックスを返す
28  print(np.argmin(c))
29
30  # 条件に応じた処理
31  print(c[np.where(c >= 4)])
32  print(c[np.where((c >= 4) & (c <= 7))])
```

```
1
[4 5 6]
6
[3 6]
[3 6 9]
3
[9 4 6 7]
[4 6 7]
```

3-1. Pythonの導入

■NumPy（CSVファイルの読み込み）

np.loadtxt('sample.csv', delimiter=',', skiprows=1)

- CSVファイルをndarray形式で読み込み

```
# CSVファイルの読み込み

import numpy as np

a = np.loadtxt('/Users/taikisuzuki/Desktop/laboratory/summer_semester_seminar/#2/sample.csv',
               delimiter=',', skiprows=1).astype(int)

print(a)
```

sample.csv

	A	B
1	number	digits
2	12	2
3	5469	4
4	7	1
5	982542	6

```
[[    12      2]
 [  5469      4]
 [     7      1]
 [982542      6]]
```

3-2. 出入力

■入力

○○_link.csv

	A	B	C	D
1	LinkID	n1	n2	Cost
2	1	1	2	77
3	2	1	40	98
4	3	1	482	89
5	4	1	484	310
6	5	2	3	95
7	6	2	39	73
8	7	3	4	94
9	8	3	38	75
10	9	4	36	80
11	10	4	576	53
12	11	5	335	42
13	12	5	570	38
14	13	5	571	35
15	14	6	333	55
16	15	6	337	64
17	16	6	570	61
18	17	6	574	25
19	18	7	8	120
20	19	7	19	33

○○_node.csv

	A	B	C
1	nodeID	緯度(日本測地	経度(日本測地
2	1	33.8377058	132.767457
3	2	33.8377475	132.768485
4	3	33.837715	132.769497
5	4	33.8377475	132.770603
6	5	33.8377983	132.771621
7	6	33.8378675	132.772693
8	7	33.8379275	132.773427
9	8	33.8380336	132.774794
10	9	33.8381997	132.775968
11	10	33.8383061	132.776734
12	11	33.8384675	132.777885
13	12	33.8377333	132.777902
14	13	33.8377367	132.777392
15	14	33.8377675	132.776701
16	15	33.8377672	132.77593
17	16	33.8378111	132.775142
18	17	33.8377981	132.774789
19	18	33.8374933	132.774761
20	19	33.8375322	132.773409

3-2. 出入力

■始点・終点・計算法の指定

```
10  """
11  始点・終点，計算法の指定
12  """
13  # 始点と終点の指定
14  s = 80  # 始点ノード番号
15  t = 1148# 終点ノード番号
16
17  # 計算法の指定
18  # Dijkstra法 = 1, Bellman-Ford法 = 2, A* = 3
19  method = 1
```

3-2. 出入力

■出力

```
node_number = 1612
edge_number = 4832
Dijkstra法
80 -> 1148
cost = 1399.8839482099997
path : [80, 81, 83, 87, 90, 93, 138, 137, 136, 150, 151, 849, 152, 153, 202, 154, 166, 958, 171, 172, 178, 944, 945, 946, 189, 188, 187, 1148]
calculaton time = 0.044532060623168945
```

```
node_number = 1612
edge_number = 4832
A*アルゴリズム
80 -> 1148
cost = 1399.8839482099997
path : [80, 81, 83, 87, 90, 93, 138, 137, 136, 150, 151, 849, 152, 153, 202, 154, 166, 958, 171, 172, 178, 944, 945, 946, 189, 188, 187, 1148]
calculaton time = 0.04011988639831543
```

4. 課題

4. 課題

1. 疑似コードを元に，Dijkstra法とA*アルゴリズムのコードを実装する.
   - ドライブの「#2/assignment/path_search_exercise.py」の該当箇所を埋める.
   - インプットデータは「#2/assignment/input」内の「Shibuya_link.csv」「Shibuya_node.csv」.
   - 疑似コードは「#2/assignment/pseudo_code.txt」

2. 複数のODペアに対して最短経路探索を行い，計算時間を比較し，簡単に考察する.

補足

- Pythonを使った経験が浅い人は，課題をやる前に，一通り基礎的なことを手を動かしながら体系的に勉強した方が良いと思います（参考： https://utokyo-ipp.github.io/index.html）．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.

- 余裕がある人は，書き方（for文を使うかモジュールを使うか，等）や言語（PythonではなくC系で書いてみる，等），A*アルゴリズムのヒューリスティック関数などを変えて計算時間を比較してみると良いでしょう．また，GISに最短経路を表示させると面白いと思います.

- 解答例は「#2/assignment/path_search.py」にあるので，必要に応じて参考にしてください.