Search Algorithms: Object-Oriented Implementation (Part B)

Improved implementation!

Contents

review

Implementation Reference Notes

- Search Tool Version
  - The Search Tool is implemented in various versions (iteratively expanded & improved).

Previous

  - v0 (강의 . Search Algo. B) : A version where **all the code to implement one algorithm is contained in a single file**

Today's

  - v1 (강의 . Search Algo. B) : A version implemented by **modularizing the logic common to different pieces of code into a separate file**, then importing and using that module/file (code refactoring & reuse)

  - v2 (강의 . Search Algo. C) : A version implemented using **classes** (v2.1, v2.2)

  - v3~v5 : Versions implemented using classes, with some algorithms added, leading to **iterative improvements in the code structure**
    - v3 : 강의 . Search Algo. D
    - v4 : 강의 . Search Algo. E
    - v5 : 강의 . Search Algo. F

review

Implementing Hill-Climbing Algorithms

- Our eventual goal is to implement an optimization tool that can run various search algorithms
- We begin to implement two hill climbing algorithms each for two different types of problems: (we implement a total of **four** hill climbing algorithms)
  - Steepest-ascent hill climbing for numerical optimization (SAHC-N)
  - First-choice hill climbing for numerical optimization (FCHC-N)
  - Steepest-ascent hill climbing for TSP (SAHC-T)
  - First-choice hill climbing for TSP (FCHC-T)

Ref:
- numerical optimization : The task of finding the variables that **maximize or minimize** the objective function/value given a mathematical expression/objective function. // **continuous**
- TSP : The problem of calculating the shortest distance to visit all given cities and return to the starting city. // **discrete**

복습 : Hill Climbing Algo. (왼쪽) *vs* first-choice hill climbing (오른쪽)

[Steepest ascent version]

```
1: function HILL-CLIMBING( problem ) returns a state that is a local maximum
2:
3: current ← MAKE-NODE(problem.INITIAL-STATE)
4: loop do
5:     neighbor ← a highest-valued successor of current
6:     if neighbor.VALUE ≤ current.VALUE then return current.STATE
7:     else current ← neighbor
```

- First-choice (simple) hill climbing:
  - Generates successors randomly until one is found that is better than the current state
  - 후보 state를 무작위로 선택하고, current state 보다 좋다면 선택(계산이 간단)

- If the goal is to maximize the objective function, and there is no better solution (neighbor):
- **XXX.VALUE**: Refers to the **calculated value of the objective function** for the solution **XXX**.
- **Example**: If objective function$(x) = x\text{^}2$, current$(x = 4)$, then current.VALUE $= (4)\text{^}2$.

Implementation with modularized common logic

SEARCH TOOL V1

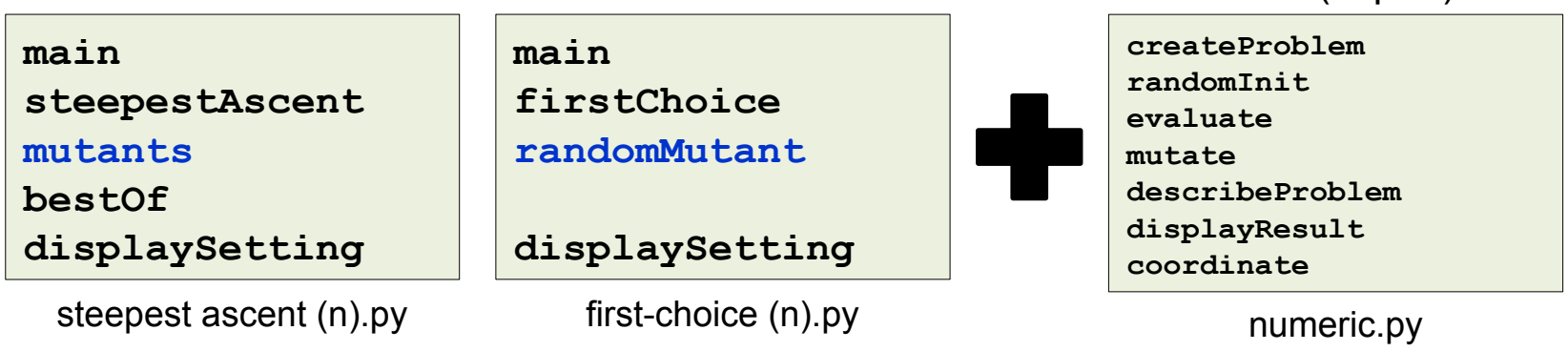
Modularity

- Numeric Optimization Problem
  - **Separate the common module** from the *first-choice(n).py* and *steepest ascent(n).py* files to **create a *numeric.py* file** and configure the original two files to **`import`** from it.

- TSP Problem
  - **Separate the common module** from the *first-choice(n).py* and *steepest ascent(n).py* files to **create a *tsp.py* file** and configure the original two files to **`import`** from it.

Introducing 'numeric' Module (for Numerical Optimization)

- By moving duplicated codes of SAHC-N and FCHC-N to a separate module named 'numeric', we can easily reuse them in both programs by simply importing the module (공통으로 사용하는 로직을 분리하여 따로 저장)
- Only a few functions remain in the main programs of SAHC-N and FCHC-N after the code migration (기존 코드가 간결해짐)



- **displaySetting** in SAHC-N and that in FCHC-N are of the same purposes, but with slightly different print messages

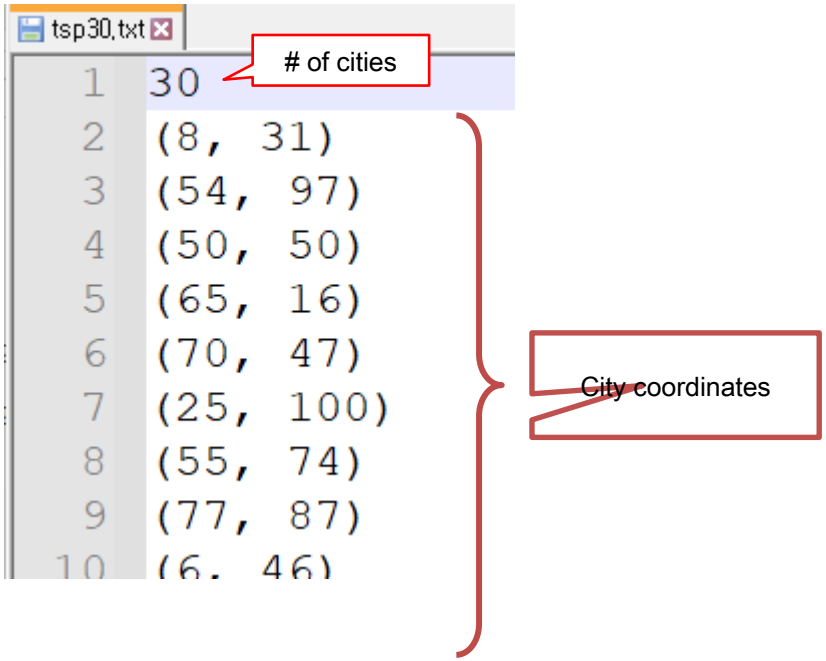
Therefore... not separated into a common module

review

TSP Problem

- The problem of planning the **route (= order of visits)** to visit all cities with the **shortest total distance**.

- Assume the problem is input via a **text file**, as shown below.
  - The **number of cities** is on the first line.
  - The subsequent lines contain the **coordinates of each city.**



TSP 문제 !

Steepest-Ascent Hill Climbing for TSP

- Since the algorithm is the same as that for numerical optimization, the main program of SAHC-N may be reused for TSP without much change
  - **`main`**, **`steepestAscent`**, and **`bestOf`** can be reused without any change at all
  - **`mutants`** should be implemented differently because the representation of candidate solution for TSP is different from that for numerical optimization
  - **`displaySetting`** should also be changed because there is no notion of mutation step size in solving TSPs

- We notice that we need a module like 'numeric' to be imported, but the codes in it should be changed appropriately for TSPs

Steepest-Ascent Hill Climbing for TSP

- A new module named **`'tsp'`** is created for being used as a replacement of the **`'numeric'`** module
  - The functions **`createProblem`**, **`randomInit`**, **`evaluate`**, **`describeProblem`**, and **`displayResult`** in 'numeric' are also needed in 'tsp' but with different implementations
  - And there may be some new functions needed for solving TSPs

- **`createProblem()`**:
  - Gets a file name from the user and reads problem information from the file
    - Number of cities – saved as an integer
    - City locations – saved as a list of 2-tuples
  - Creates a matrix of distances between every pair of cities (**`calcDistanceTable`** – a new function)
  - Returns the triple: number of cities, locations, distance table

Steepest-Ascent Hill Climbing for TSP

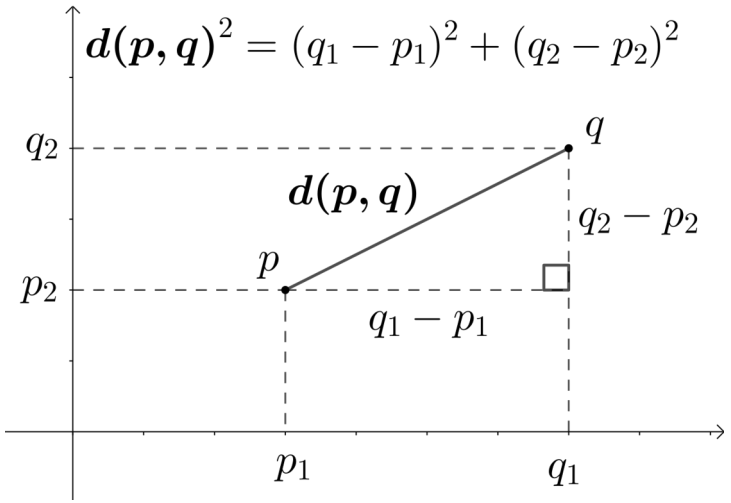
- **`createProblem()`:**

```
def createProblem():
    ## Read in a TSP (# of cities, locatioins) from a file.
    ## Then, create a problem instance and return it.
    fileName = input("Enter the file name of a TSP: ")
    infile = open(fileName, 'r')
    # First line is number of cities
    numCities = int(infile.readline())
    locations = []
    line = infile.readline()  # The rest of the lines are locations
    while line != '':
        locations.append(eval(line)) # Make a tuple and append
        line = infile.readline()
    infile.close()
    table = calcDistanceTable(numCities, locations)
    return numCities, locations, table
```

Read number of cities on first line as Integer

Steepest-Ascent Hill Climbing for TSP

- **`calcDistanceTable(numCities, locations)`**:
  - Calculates an $n \times n$ matrix of pairwise distances based on **`locations`** ($n =$ **`numCities`**) // use Euclidean distance calculation



$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}.$$

Steepest-Ascent Hill Climbing for TSP

- **`randomInit(p)`**:
  - Returns a randomly shuffled list of IDs of the cities in **`p`** (= random initial solution)

```
def randomInit(p):   # Return a random initial tour
    n = p[0]
    init = list(range(n))
    random.shuffle(init)
    return init
```

The shuffle function ensure the random list is a valid solution.

- **`Evaluate(current, p)`**:
  - Calculates the tour cost (= 이동 거리 총합) of **`current`** by looking at the **distance matrix** given in **`p`**
  - Distance matrix for evaluation is the **`table`** output of **`calcDistanceTable`** function.

Steepest-Ascent Hill Climbing for TSP

- **`inversion(current, i, j)`**:
  - Makes a copy of **`current`**, inverts its subsection from ***i*** to ***j***, and returns the mutant // 인덱스 i부터 j까지의 모든 item을 뒤집기
  - This function takes the role of **`mutate`** for numerical optimization // numerical optimization에서 mutate 함수의 역할을 수행

```
def inversion(current, i, j):  # Perform inversion
    curCopy = current[:]
    while i < j:
        curCopy[i], curCopy[j] = curCopy[j], curCopy[i]
        i += 1
        j -= 1
    return curCopy
```

In Python, the exchange of two variables can be easily implemented.

From prev. lecture

Ref) Variable swapping

- The general method for swapping two variables (using a temporary variable)

```
temp = var1;
var1 = var2;
var2 = temp;
```

- Method for swapping without a temporary variable:

  1. (Arithmetic)

```
int x = 10, y = 5;

// Code to swap 'x' and 'y'
x = x + y; // x now becomes 15
y = x - y; // y becomes 10
x = x - y; // x becomes 5
```

  2. XOR

```
int x = 10, y = 5;
// Code to swap 'x' (1010) and 'y' (0101)
x = x ^ y; // x now becomes 15 (1111)
y = x ^ y; // y becomes 10 (1010)
x = x ^ y; // x becomes 5 (0101)
```

Steepest-Ascent Hill Climbing for TSP

- **`describeProblem(p)`**:
  - Prints the number of cities in **`p`**, followed by the city locations, five locations per line

```
def describeProblem(p):
    print()
    n = p[0]
    print("Number of cities:", n)
    print("City locations:")
    locations = p[1]
    for i in range(n):
        print("{0:>12}".format(str(locations[i])), end = '')
        if i % 5 == 4:
            print()
```

Steepest-Ascent Hill Climbing for TSP

- **`displayResult(solution, minimum)`**:
  - Displays **`solution`** (the best tour found) (**`tenPerRow`**), **`minimum`** (its evaluation value), and the total number of evaluations

```
def displayResult(solution, minimum):
    print()
    print("Best order of visits:")
    tenPerRow(solution)       # Print 10 cities per row
    print("Minimum tour cost: {0:,}".format(round(minimum)))
    print()
    print("Total number of evaluations: {0:,}".format(NumEval))
```

- **`tenPerRow(solution)`**:
  - Prints city ids in order of visit, ten ids per row
  - 최종 결과 (= 도시 방문 순서) 를 10 개씩 화면에 출력하는 함수

Steepest-Ascent Hill Climbing for TSP

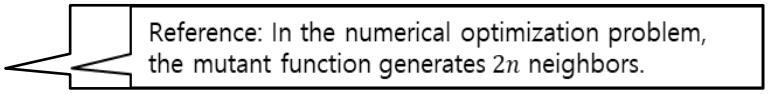
- **`tenPerRow(solution)`**:
  - Prints city ids in order of visit, ten ids per row
  - 최종 결과 (= 도시 방문 순서) 를 한 줄에 10 개씩 화면 출력

```
def tenPerRow(solution):
    for i in range(len(solution)):
        print("{0:>5}".format(solution[i]), end='')
        if i % 10 == 9:
            print()
```

Steepest-Ascent Hill Climbing for TSP

- Below, we describe how mutants in the main program is implemented differently for TSPs than numerical optimization
  - 현재의 솔루션을 기준으로 , 이웃하는 솔루션을 생성하는 방법 설명

- **`mutants(current, p)`**:



  - Returns n neighbors of **`current`** (n = number of cities in **`p`**),
  - Each neighbor is generated by inverting the subsection beginning from **`i`** and ending at **`j`** (**`inversion`**), where the (**`i`**, **`j`**)-pair is chosen randomly
  - The inversion for (**`i`**, **`j`**)-pair is applied only when **`i`** $\neq$ **`j`** and the pair has never been tried before

Steepest-Ascent Hill Climbing for TSP

- **`mutants(current, p)`:**

```
def mutants(current, p): # Apply inversion
    n = p[0]
    neighbors = []
    count = 0
    triedPairs = []
    while count <= n:  # Pick two random loci for inversion
        i, j = sorted([random.randrange(n) for _ in range(2)])
        if i < j and [i, j] not in triedPairs:
            triedPairs.append([i, j])
            curCopy = inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors
```

FC 알고리즘

First-Choice Hill Climbing for TSP

- The main program of FCHC-N can be reused without much change
  - **`main`** and **`firstChoice`** can be reused without change
  - **`displaySetting`** needs to be changed because the mutation step size is now irrelevant
  - **`randomMutant`** should be implemented differently because of the different representation of candidate solution for TSPs

- **`randomMutant(current, p)`**:
  - Returns a mutant of **`current`**, which is made by inverting the sub-section beginning from **`i`** and ending at **`j`** (**`inversion`**), where **`i`** and **`j`** are chosen randomly

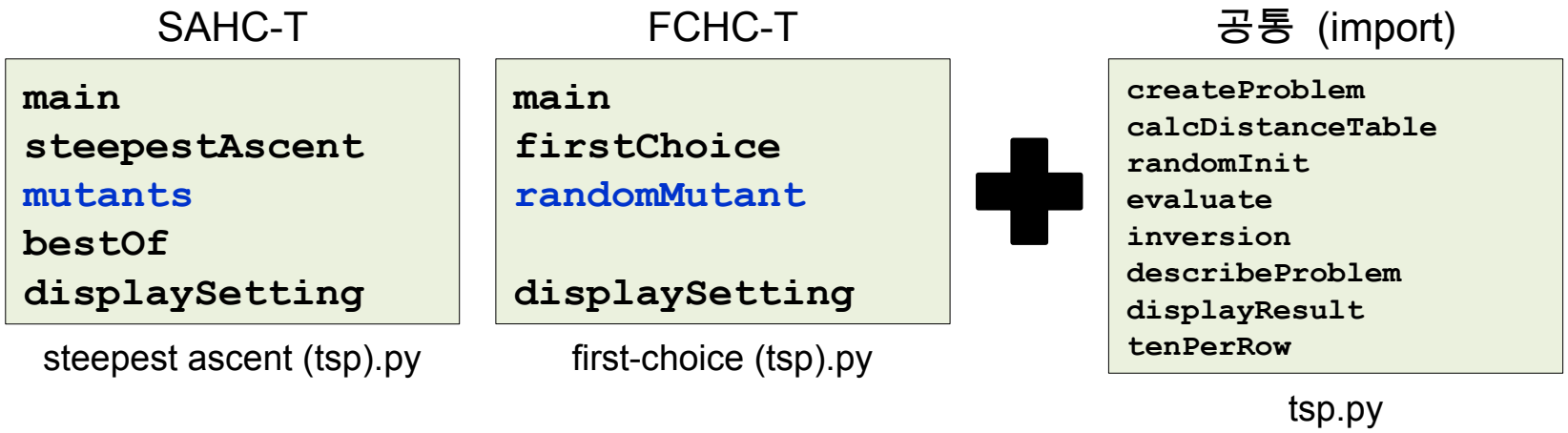
First-Choice Hill Climbing for TSP

- **`randomMutant(current, p)`**:
  - Returns a mutant of **`current`**, which is made by inverting the sub-section beginning from **`i`** and ending at **`j`** (**`inversion`**), where **`i`** and **`j`** are chosen randomly

```
def randomMutant(current, p): # Apply inversion
    while True:
        i, j = sorted([random.randrange(p[0])
                       for _ in range(2)])
        if i < j:
            curCopy = inversion(current, i, j)
            break
    return curCopy
```

Search Tool v1 for TSP

- Code configuration after modularization



TSP Problem

- Execution Results (Example using the **First-Choice** Algorithm)

```
Enter the file name of a TSP: problem/tsp30.txt

Number of cities: 30
City locations:
     (8, 31)    (54, 97)    (50, 50)    (65, 16)    (70, 4?)
   (25, 100)    (55, 74)    (77, 87)     (6, 46)    (70, 78)
    (13, 38)   (100, 32)    (26, 35)    (55, 16)    (26, 77)
    (17, 67)    (40, 36)    (38, 27)     (33, 2)     (48, 9)
    (62, 20)    (17, 92)     (30, 2)    (80, 75)    (32, 3?)
    (43, 79)    (57, 49)    (18, 24)    (96, 76)    (81, ?)

Search algorithm: First-Choice Hill Climbing
Max evaluations with no improvement: 100 iterations

Best order of visits:
    4   26    2   24   16   27    0    8   10   12
   17   18   22   19    3   13   20   29   11   28
   23    7    9    6   25    1   14   15   21    5
Minimum tour cost: 491

Total numb    f evaluations: 677
```

Input the name and path of the **.txt file defining the problem** (User Input).

City 4 (starts from index 0)

The last city is City 29

The **Optimal Order of Visits**

The **Total Distance** traveled when following the optimal route.

TSP Problem

- Execution Results (Example using the **Steepest-Ascent** Algorithm)

```
Enter the file name of a TSP: problem/tsp30.txt

Number of cities: 30
City locations:
     (8, 31)    (54, 97)    (50, 50)    (65, 16)    (70, 47)
   (25, 100)    (55, 74)    (77, 87)     (6, 46)    (70, 78)
    (13, 38)   (100, 32)    (26, 35)    (55, 16)    (26, 77)
    (17, 67)    (40, 36)    (38, 27)     (33, 2)     (48, 9)
    (62, 20)    (17, 92)     (30, 2)    (80, 75)    (32, 36)
    (43, 79)    (57, 49)    (18, 24)    (96, 76)    (81, 39)

Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:
   11   29   20   13    3    4   28   23    9    1
    7    6    2   16   26   25   21    5    8   15
   14   10    0   12   27   24   17   19   18   22
Minimum tour cost: 620

Total num     of evaluations: 807
```

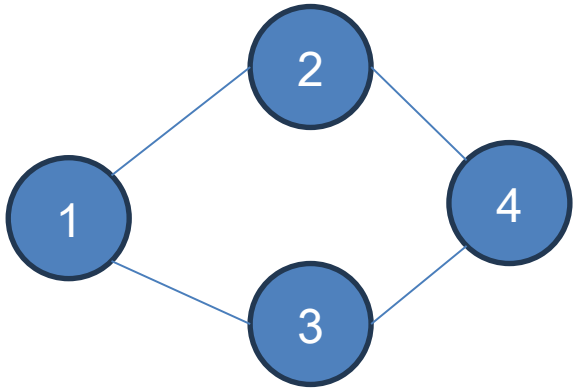
Input the name and path of the **.txt file defining the problem** (User Input).

The **Optimal Order of Visits**

The **Total Distance** traveled when following the optimal route.

Ref) Adjacency Matrix / 인접 행렬

- Adjacency Matrix
  - One of the effective ways to represent an **Undirected Graph**.
  - Directly adjacent vertices (i, j) are represented by **1**, and two vertices that are not adjacent are represented by **0**.
  - The entry for (i, i) (a vertex's adjacency to itself) is represented by **0**.



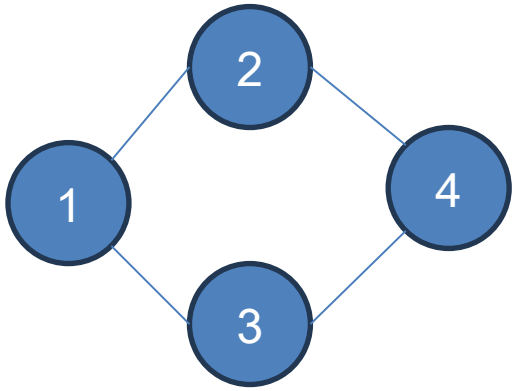
$$\begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{matrix}$$

Ref) Adjacency Matrix / 인접 행렬

- Formal Definition
  - In graph theory and computer science, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.
  - In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected (i.e. all of its edges are bidirectional), the adjacency matrix is symmetric.
    - simple graph : a graph that does not have more than one edge between any two vertices and no edge starts and ends at the same vertex

Ref) Adjacency Matrix / 인접 행렬

- Interesting properties of Adjacency Matrix : Matrix powers
  - Let **A** be the Adjacency Matrix representing the graph below



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

  - In this case, the result of **A^2** is shown on the left, and the result of **A^3** is shown on the right. **What is the meaning of each result?**?

2	0	0	2
0	2	2	0
0	2	2	0
2	0	0	2

0	4	4	0
4	0	0	4
4	0	0	4
0	4	4	0

Ref) Adjacency Matrix / 인접 행렬

- Interesting properties of Adjacency Matrix : Matrix powers

Unless specified, all graphs are assumed to be simple and connected, that is, there is at most one edge between each pair of vertices, there are no loops, and there is at least one path between every two vertices. The adjacency matrix A or $A(G)$ of a graph G having vertex set $V = V(G) = \{1, ..., n\}$ is an $n \times n$ symmetric matrix a_{ij} such that $a_{ij} = 1$ if vertices i and j are adjacent and 0 otherwise.

Powers of the Adjacency Matrix

The following well-known result will be used frequently throughout:

Theorem 0.1 *The $(i, j)^{th}$ entry a_{ij}^k of A^k, where $A = A(G)$, the adjacency matrix of G, counts the number of walks of length k having start and end vertices i and j respectively.*

A^k 계산 결과에서 (i,j)번째 값이 n 이라면, i에서 출발해서 j까지 도착하는 서로 다른 경로의 수가 n개라는 의미

끝 .