

플로이드-워셜 알고리즘

주요 내용

- 방향 그래프
- 최단 경로 문제
- 플로이드-워셜 알고리즘
- 최적의 원칙

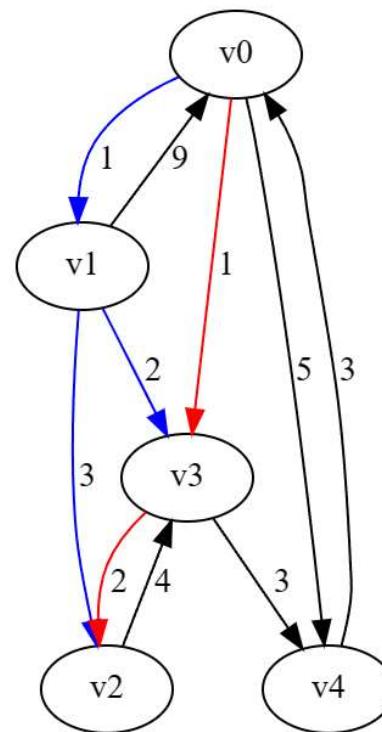
그래프 용어

| 용어 | 개념 |
|--------------------------|-------------------------------------|
| 노드 node | 그래프의 구성 요소. 꼭짓점 vertex 또는 정점으로도 불림. |
| 간선 edge | 두 개의 노드를 연결하는 선. 이음선 또는 변으로도 불림. |
| 방향 그래프 directed graph | 간선의 방향이 정해진 그래프 |
| 무방향 그래프 undirected graph | 간선의 방향이 없는 그래프 |
| 가중치 weight | 간선에 추가된 숫자 |
| 가중 그래프 weighted graph | 가중치가 있는 간선을 사용하는 그래프 |

용어**개념**

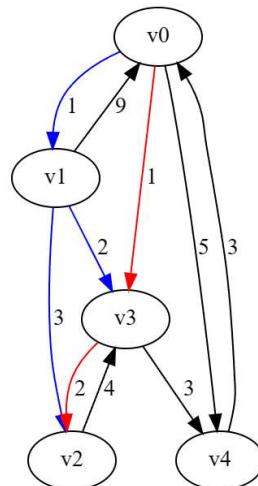
| | |
|--------------------------|---|
| 경로 path | 간선으로 연결된 노드들의 나열. 즉, 하나의 노드에서 다른 노드로 가는 간선들의 연결. |
| 단순 경로 simple path | 같은 노드를 두 번 지나지 않는 경로 |
| 순환 cycle | 하나의 노드에서 출발하여 다시 그 노드로 돌아오는 경로 |
| 순환 그래프 cyclic graph | 순환이 포함된 그래프 |
| 비순환 그래프 acyclic graph | 순환이 포함되지 않은 그래프 |
| 경로의 길이 | 가중 그래프의 경우엔 경로 상에 있는 가중치의 합, 비가중 그래프의 경우엔 경로 상에 있는 이음선의 수 |

예제: 가중 방향 그래프



최단 경로 문제

- 위 가중 방향 그래프에서 v_0 에서 v_2 로 가는 단순 경로는 다음 세 종류:
 - $v_0 \rightarrow v_1 \rightarrow v_2$: 경로 길이는 $1 + 3 = 4$.
 - $v_0 \rightarrow v_3 \rightarrow v_2$: 경로 길이는 $1 + 2 = 3$.
 - $v_0 \rightarrow v_1 \rightarrow v_3 \rightarrow v_2$: 경로 길이는 $1 + 2 + 2 = 5$.
- 이중에서 $v_0 \rightarrow v_3 \rightarrow v_2$ 가 v_0 에서 v_2 로 가는 최단 경로임



완전 탐색

- v_0, v_1, \dots, v_{n-1}
노드 사용
- 모든 노드들 사이에 간선이 존재한다고 가정
- v_0 에서 v_{n-1} 으로 가는 경로 중에서 나머지 모든 노드를 한 번씩 꼭 거쳐서 가는 경로들의 수
 - v_0 에서 출발하여 첫 경유지로 사용될 수 있는 노드의 수는 $(n - 2)$ 개
 - 그 중에 하나를 선택하면, 그 다음 경유지로 사용될 수 있는 노드의 가지 수는 $(n - 3)$ 개
 - ...

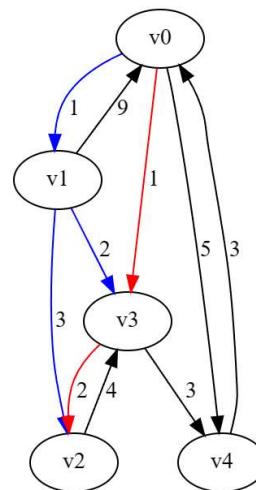
$$(n - 2) \cdot (n - 3) \cdots 2 \cdot 1 = (n - 2)!$$

인접 행렬

- 가중 그래프의 인접 행렬: 노드와 노드를 잇는 간선과 가중치의 정보를 표현하는 $n \times n$ 행렬

$$W[i][j] = \begin{cases} \text{간선 가중치} & v_i \text{에서 } v_j \text{로의 간선이 존재하는 경우} \\ \infty & v_i \text{에서 } v_j \text{로의 간선이 존재하지 않는 경우} \\ 0 & i = j \text{인 경우} \end{cases}$$

예제



```
In [1]: from math import inf

W = [[0, 1, inf, 1, 5],
      [9, 0, 3, 2, inf],
      [inf, inf, 0, 4, inf],
      [inf, inf, 2, 0, 3],
      [3, inf, inf, inf, 0]]
```

동적계획법 전략

- v_i 에서 v_j 로 가는 최단 경로를 계산
- 경유 노드를 확대해 나가면서 최단 경로를 업데이트하는 전략 사용
- k 를 0부터 n 까지 변하게 하면서 아래 조건에 맞는 행렬 $D^{(k)}$ 를 순차적으로 생성

$D^{(k)}[i][j] =$ 집합 $\{v_0, v_1, \dots, v_{k-1}\}$ 에 속하는 노드만을 경유해서
 v_i 에서 v_j 로 가는 최단 경로의 길이

- $D^{(0)}$: 인접 행렬 W 와 동일
- $D^{(n)}$: 두 노드 사이의 최단 경로의 길이로 구성됨

$D^{(k)}[i][j]$ 의 재귀적 성질

- 0보다 큰 k 에 대해 다음 성립:

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

- 경우 1

- $\{v_0, v_1, \dots, v_{k-1}\}$ 에 속한 노드들만을 통해서 v_i 에서 v_j 로 가는 최단 경로가 v_{k-1} 를 경유하는 경우.
- v_{k-1} 가 무시되기에 결국 $\{v_0, v_1, \dots, v_{k-2}\}$ 만 경유하는 최단 거리와 동일함.

$$D^{(k)}[i][j] = D^{(k-1)}[i][j]$$

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

- 경우 2

- $\{v_0, v_1, \dots, v_{k-1}\}$ 에 속한 노드들만을 통해서 v_i 에서 v_j 로 가는 최단 경로가 v_{k-1} 를 경유하지 않는 경우.
- 단순 경로만 고려해야 하기에 다음 v_{k-1} 을 한 번만 경유하며 따라서 다음 식이 성립함.

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

동적 계획법 전략 적용 예제: $D^{(k)}[1][4]$ 계산

- $D^{(0)}[1][4] = W[1][4] = \infty$
- $D^{(1)}[1][4] = \min(D^{(0)}[1][4], d^{(1)}) = \min(\infty, 14) = 14$
 - $d^{(1)}$ 은 v_0 를 경유하는 최단 경로 길이:
$$d^{(1)} = \text{len}([v_1, v_0, v_4]) = 9 + 5 = 14$$
- $D^{(2)}[1][4] = D^{(1)}[1][4] = 14$
 - v_0 와 v_1 을 경유할 수 있는 최단 경로 길이는 v_0 를 경유할 수 있는 최단 경로 길이와 동일

- $D^{(3)}[1][4] = D^{(2)}[1][4] = 14$
 - v_2 를 경유할 수 없음. 왜냐하면 v_2 를 경유하면 반드시 v_3 를 경유해야 하는데 그것이 허용되지 않기 때문임.
- $D^{(4)}[1][4] = \min(D^{(3)}[1][4], d^{(4)}) = \min(14, 5) = 5.$
 - $d^{(4)}$ 는 v_3 를 거치는 최단 경로 길이:
$$\begin{aligned} d^{(4)} &= \min(\text{length}[v_1, v_3, v_4], \text{length}[v_1, v_0, v_3, v_4], \text{length}[v_1, v_2, v_3, v_4]) \\ &= \min(5, 13, 10) \\ &= 5 \end{aligned}$$
- $D^{(5)}[1][4] = D^{(4)}[1][4] = 5$
 - v_4 는 최종 도착점이기 때문임.

플로이드-워셜 알고리즘

플로이드-워셜 Floyd-Warshall 알고리즘: 앞서 설명한 동적계획법 전략으로 아래 화살표 과정을 구현하는 알고리즘

$$W = D^{(0)} \longrightarrow D^{(1)} \longrightarrow D^{(2)} \longrightarrow \dots \longrightarrow D^{(n-1)} \longrightarrow D^{(n)} = D$$

```
In [2]: from copy import deepcopy
```

```
def floyd_marshall(W):
    n = len(W)
    # 사전을 이용하여 D^(0), ..., D^(n) 저장
    # 키는 0, 1, ..., n 사용
    D = dict()

    # D^(0) 지정
    # 주의: deepcopy를 사용하지 않으면 W에 혼란을 발생시킴
    D[0] = deepcopy(W)

    # D^(k)로부터 D^(k+1)를 생성
    for k in range(0, n):
        D[k+1] = deepcopy(D[k])
        # 행렬의 인덱스는 0부터 (n-1)까지 이동
        for i in range(0, n):
            for j in range(0, n):
                if D[k][i][k]+D[k][k][j] < D[k][i][j]:
                    D[k+1][i][j] = D[k][i][k]+D[k][k][j]

    # 최종 완성된 D[n] 반환
    return D[n]
```

최단 경로 확인 알고리즘

- 이전 함수를 약간 수정하여 최단 경로를 출력하는 함수를 구현
- 두 노드 사이의 최단 경로에 사용된 노드 중에서 가장 큰 인덱스를 기억하는 행렬 P 동시 생성

$$P[i][j] = \begin{cases} k & v_i \text{에서 } v_j \text{로의 최단 경로의 경유지로 사용된} \\ & \text{노드의 인덱스 중에서 가장 큰 값이 } k \\ -1 & v_i \text{에서 } v_j \text{로의 간선이 최단 경로이거나} \\ & \text{두 노드 사이의 경로가 존재하지 않는 경우} \end{cases}$$

```
In [3]: from copy import deepcopy

def floyd_marshall2(W):
    n = len(W)
    # 사전을 이용하여 D^(0), ..., D^(n) 저장
    # 키는 0, 1, ..., n 사용
    D = dict()

    # 경로 기억 어레이
    P = [[-1] * n for i in range(n)] # -1로 초기화

    # D^(0) 지정
    # 주의: deepcopy를 사용하지 않으면 W에 혼란을 발생시킴
    D[0] = deepcopy(W)

    # D^(k)로부터 D^(k+1)를 생성
    for k in range(0, n):
        D[k+1] = deepcopy(D[k])
        # 행렬의 인덱스는 0부터 (n-1)까지 이동
        for i in range(0, n):
            for j in range(0, n):
                if D[k][i][k]+D[k][k][j] < D[k][i][j]:
                    P[i][j] = k
                    D[k+1][i][j] = D[k][i][k]+D[k][k][j]

    # 최종 완성된 D[n] 반환
    return D[n], P
```

```
In [4]: D, P = floyd_marshall2(W)
```

```
In [5]: D
```

```
Out[5]: [[0, 1, 3, 1, 4],  
         [8, 0, 3, 2, 5],  
         [10, 11, 0, 4, 7],  
         [6, 7, 2, 0, 3],  
         [3, 4, 6, 4, 0]]
```

```
In [6]: P
```

```
Out[6]: [[-1, -1, 3, -1, 3],  
         [4, -1, -1, -1, 3],  
         [4, 4, -1, -1, 3],  
         [4, 4, -1, -1, -1],  
         [-1, 0, 3, 0, -1]]
```

최단 경로 찍어보기

```
In [7]: def path2(P, q, r, route):
    if P[q][r] != -1:
        v = P[q][r]

        path2(P, q, v, route)
        route.append(v)
        path2(P, v, r, route)

    return route
```

```
In [8]: path2(P, 4, 2, [])
```

```
Out[8]: [0, 3]
```

```
In [9]: def print_path2(P, i, j):
    route = path2(P, i, j, [])
    route.insert(0, i)
    route.append(j)
    print(" -> ".join([str(v) for v in route]))
```

```
In [10]: print_path2(P, 4, 2)
```

4 -> 0 -> 3 -> 2

```
In [11]: print_path2(P, 1, 4)
```

1 -> 3 -> 4

최적의 원칙

- 동적계획법에 의한 설계 절차
 - 문제의 입력에 대해 최적의 해답을 제공하는 재귀 관계식 설정
 - 상향식으로 작은 입력값에 대한 최적의 해답을 계산하여 최종적으로 최적의 해답 구축
- 최적의 원칙: 주어진 문제 사례에 대한 최적의 해가 그 사례를 분할한 모든 부분사례에 대한 최적의 해를 포함하
- 동적계획법에 의해 얻어진 해답이 최적이 되려면 해당 문제에 대해 최적의 원칙이 적용될 수 있어야 함

예제: 최단 경로 문제

- 최적의 원칙 성립
- v_k 가 v_i 에서 v_j 로 가는 단순 최단 경로 상에 위치하며 가장 큰 인덱스를 갖는 노드 일 때 v_i 에서 v_k 로 가는 부분경로와 v_k 에서 v_j 로 가는 부분경로 또한 두 노드 사이의 최단 경로임

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

예제: 최장 경로 문제

- 최적의 원칙 미성립
- 아래 그래프에서 v_0 에서 v_3 로의 단순 최장 경로는 $v_0 \rightarrow v_2 \rightarrow v_1 \rightarrow v_3$ 임
- 그러나 이 경로의 부분 경로인 v_0 에서 v_2 으로의 비순환 최장 경로는 $v_0 \rightarrow v_2$ 가 아니라 $v_0 \rightarrow v_1 \rightarrow v_2$ 임.

