6절 외판원 문제

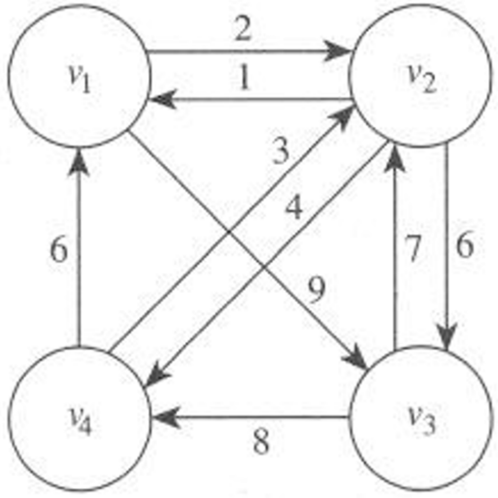
외판원문제 정의

- 일주경로: 한 도시에서 출발하여 다른 모든 도시를 한 번씩만 들린 후 출발한 도시로 돌아오는 경로
- 최적일주경로: 최소거리 일주경로
- 외판원문제: 최소한 하나의 일주경로가 존재하는 가중치포함 방향그래프에서 최적일주경로 찾기

주의사항

- 출발하는 도시가 최적일주경로의 길이와 무관함.
  - 어차피 일주경로를 따지기 때문.
- 따라서 한 지점(마디)에서 출발하는 일주경로만을 대상으로 알고리즘 구현.

예제



- v_1 을 출발점으로 하는 일주경로 3개.

$$
\begin{aligned}
length\,[v_1, v_2, v_3, v_4, v_1] &= 22 \\
length\,[v_1, v_3, v_2, v_4, v_1] &= 26 \\
length\,[v_1, v_3, v_4, v_2, v_1] &= 21
\end{aligned}
$$

- 마지막 경로가 최적.

무차별 대입 방식(brute force) 탐색

- v_1 부터 시작하는 모든 일주경로를 확인하는 방식

부르트포스 탐색 알고리즘

- 참조: [고전 컴퓨터 알고리즘 인 파이썬, 9장 (https://github.com/coding-alzi/ClassicComputerScienceProblemsInPython)](https://github.com/coding-alzi/ClassicComputerScienceProblemsInPython)

- 도시간 거리: 중첩 사전(딕셔너리)으로 구현
    - 키: 도시명
    - 키값(사전 자료형)
        - 키: 해당 도시와 이음선으로 연결된 도시
        - 키값: 그 도시로의 이동 거리

In [1]:

```python
from math import inf
from itertools import permutations

city_distances = {
    "v1":
        {"v2": 2,
         "v3": 9},
    "v2":
        {"v1": 1,
         "v3": 6,
         "v4": 4},
    "v3":
        {"v2": 7,
         "v4": 8},
    "v4":
        {"v1": 6,
         "v2": 3}
}
```

- 도시명 모음

In [30]:
```
cities = list(city_distances.keys())
```

In [31]:
```
print(cities)
```

```
['v1', 'v2', 'v3', 'v4']
```

- v_1 에서 출발하는 모든 일주경로의 순열조합
    - v_1 을 제외한 나머지 $n-1$개의 도시로 만들 수 있는 모든 순열조합
        - 순열조합 수: $(n-1)!$
        - $n = 4$일 경우: 총 $3! = 6$개의 일주경로 존재.

- 주의: 이음선이 없는 경우가 포함된 경로는 이후 최적일주경로 선정 과정에서 제외처리될 것임.

- `city_permutations` 가 가리키는 값은 아래 6개의 항목으로 이루어진 이터러블 자료형

```
In [32]: city_permutations = permutations(cities[1:])
```

```
('v2', 'v3', 'v4'),
('v2', 'v4', 'v3'),
('v3', 'v2', 'v4'),
('v3', 'v4', 'v2'),
('v4', 'v2', 'v3'),
('v4', 'v3', 'v2')
```

- v_1 에서 출발하는 일주경로 완성을 위해 출발도시를 처음과 마지막 항목으로 추가

```
In [26]: tsp_paths = [(cities[0],) + c + (cities[0],) for c in city_permutations]
```

- tsp_paths는 v_1 에서 시작하는 모든 일주경로의 목록을 담은 리스트.

```
[('v1', 'v2', 'v3', 'v4', 'v1'),
 ('v1', 'v2', 'v4', 'v3', 'v1'),
 ('v1', 'v3', 'v2', 'v4', 'v1'),
 ('v1', 'v3', 'v4', 'v2', 'v1'),
 ('v1', 'v4', 'v2', 'v3', 'v1'),
 ('v1', 'v4', 'v3', 'v2', 'v1')]
```

최단 일주경로 길이 확인하기

- v_1 에서 출발하는 모든 일주경로를 대상으로 경로의 길이를 계산하는 단순한 코드임.

- `best_path`: 최적일주경로 저장
    - 초기값은 `None`.
- `best_distance`: 최적일주경로의 길이 저장
    - 초기값은 무한대(`inf`).
    - `inf`: 어떤 수보다 큰 값을 나타내는 기호.

In [37]:

```
best_path = None
min_distance = inf
```

- 모든 경로를 대상으로 길이를 확인한 다음 최적일주경로의 길이를 업데이트함.

- 일주경로상에서 두 마디 사이에 이음선이 존재하지 않으면 일주경로의 길이를 무한대(`inf`)로 처리함.
    - 이런 방식으로 실제로 존재하지 않는 일주경로를 최소거리 경쟁에서 제외시킴.

- 두 마디 사이에 이음선 존재여부 확인
    - 사전 자료형의 `get` 메서드가 `None`을 반환하는 성질 활용

- last와 next 를 차례대로 업데이트하면서 일주경로의 길이 계산
    - last: 경로상에서 외판원의 현재 위치
    - next: 경로상에서 외판원이 방문할 다음 위치

In [41]:
```
for path in tsp_paths:
    distance = 0
    last = path[0]

    for next in path[1:]:
        last2next = city_distances[last].get(next)

        if last2next:                          # last에서 next로의 경로가 존재하는 경우
            distance += last2next

        else:                                  # last에서 next로의 경로가 존재하지 않는 경우 None
 반환됨.
            distance = inf                     # 무한대로 처리하며, 결국 최솟값 경쟁에서 제외됨.
        last = next

    if distance < min_distance:                # 최단경로를 업데이트 해야 하는 경우
        min_distance = distance
        best_path = path
```

In [40]:

```python
print(f"최적일주경로는 {best_path}이며 길이는 {min_distance}이다.")
```

```
최적일주경로는 ('v1', 'v3', 'v4', 'v2', 'v1')이며 길이는 21이다.
```

- 함수로 정리하기

In [68]:
```
def tsp_bruteforce(city_distances:dict):
    # v1에서 시작하는 모든 일주경로 확인
    cities = list(city_distances.keys())
    city_permutations = permutations(cities[1:])
    # 최적경로와 최단길이 기억
    best_path = None
    min_distance = inf

    # 각 일주경로의 길이확인. 동시에 최적경로와 최단길이 업데이트
    for path in tsp_paths:
        distance = 0
        last = path[0]
        for next in path[1:]:
            last2next = city_distances[last].get(next)
            if last2next:                          # last에서 next로의 경로가 존재하는 경우
                distance += last2next
            else:                                  # last에서 next로의 경로가 존재하지 않는 경우 N
one 반환됨.
                distance = inf                     # 무한대로 처리하며, 결국 최솟값 경쟁에서 제외됨.
            last = next
        if distance < min_distance:                # 최단경로를 업데이트 해야 하는 경우
            min_distance = distance
            best_path = path
    # 최적경로와 최단길이 반환
    return best_path, min_distance
```

In [67]:

```python
best_path, min_distance = tsp_bruteforce(city_distances)

print(f"최적일주경로는 {best_path}이며 길이는 {min_distance}이다.")
```

```
최적일주경로는 ('v1', 'v3', 'v4', 'v2', 'v1')이며 길이는 21이다.
```

부르트포스 탐색 시간복잡도

- 입력크기: 도시(마디) 수 n
- 단위연산: v_1 을 제외한 나머지 $n-1$개의 도시를 일주하는 경로의 모든 경로를 고려하는 방법

$$(n-1)(n-2)\cdots 1 = (n-1)! \in \Theta(n!)$$

- 설명: 하나의 도시에서 출발해서
  - 둘째 도시는 $(n-1)$개 도시 중 하나,
  - 셋째 도시는 $(n-2)$개 도시 중 하나,
  - ....
  - $(n-1)$번째 도시는 2개 도시 중 하나,
  - 마지막 n번째 도시는 남은 도시 하나.

더 좋은 알고리즘

- 외판원 문제에 대한 쉬운 해결책은 없음.
- 도시가 많은 경우 대부분의 알려진 알고리즘은 최적일주경로의 근사치를 계산함.
- 동적계획법 또는 유전 알고리즘을 이용하면 시간복잡도가 **조금** 더 좋은 알고리즘 구현 가능
    - 하지만 모두 지수함수 이상의 복잡도를 가짐.

동적계획법으로 구현한 외판원문제 알고리즘 복잡도

- 일정 시간복잡도: $\Theta(n^2\ 2^n)$

- 일정 공간복잡도: $\Theta(n\ 2^n)$

- 부르트포스 알고리즘보다 훨씬 빠르기는 하지만 여전히 실용성은 없음.
  - 실제로 구현하기도 쉽지 않음.
  - 다양한 트릭이 있지만 알고리즘 공부에 별 도움되지 않음.

- 유전 알고리즘 기법을 활용하여 적절한 근사치를 빠르게 계산하는 알고리즘에 대한 연구가 많이 진행되어 왔음.
  - 필요할 경우 가정 적절한 유전 알고리즘 활용해야 함.

다항식 시간복잡도 알고리즘은?

- 다항식 복잡도를 갖는 외판원문제 해결 알고리즘 알려지지 않음.
- 그런 알고리즘은 존재할 수 없다는 증명도 알려지지 않음.
- 이와같이 해답구하기가 매우 어려운 문제들에 대해 9장에서 좀 더 상세히 다룸.

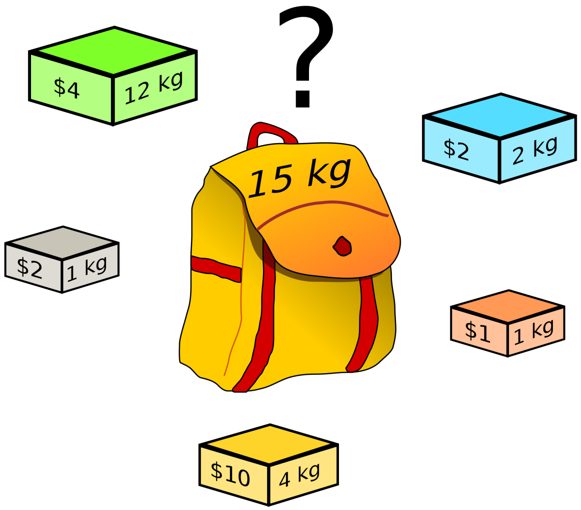
(4장 5절) 0-1 배낭 채우기 문제

0-1 배낭 채우기 문제 정의

- n개의 주어진 물건들 중에서, 한정된 용량(W)의 배낭에 물건을 골라 넣었을때 얻을 수 있는 최대 값어치를 찾는 조합 최적화 문제

예제

- $n = 5$
- $W = 15$



<그림 출처:[배낭 문제: 위키피디아 (https://en.wikipedia.org/wiki/Knapsack_problem)](https://en.wikipedia.org/wiki/Knapsack_problem)>

무차별 대입 방식(brute force approach)

- 배낭에 넣을 수 있는 모든 물건의 조합 살피기
- n개의 물건이 있을 때 총 2^n 개의 조합 존재
- 따라서 $\Theta(2^n)$의 시간복잡도를 가짐. 따라서 실용성 없음.

동적계획법 알고리즘

- 참조: [고전 컴퓨터 알고리즘 인 파이썬, 9장 (https://github.com/coding-alzi/ClassicComputerScienceProblemsInPython)](https://github.com/coding-alzi/ClassicComputerScienceProblemsInPython)

- 이항계수 동적계획법 알고리즘과 유사.

- 아래 조건을 만족하는 `(n+1,W+1)` 모양의 2차원 행렬 P 생성

$$P[i][w] = \text{총 무게가 } w\text{를 넘기지 않는 조건하에서}$$
$$\text{처음 } i \text{ 개의 물건만을 이용해서 얻을 수 있는 최대 이익}$$

주어진 조건

- i 번째 물건의 무게와 값어치 ($0 \leq i \leq n$)
    - 무게: w_i
    - 값어치: p_i

$P[i][j]$의 재귀식

- 초기값: $i = 0$인 경우
    - 물건을 전혀 사용하지 못하기 때문에 물건을 전혀 배낭에 담지 못함.
    - 따라서 모든 $0 \le w \le W$에 대해 다음 성립:

$$P[0][w] = 0$$

- 귀납단계: $i > 0$ 이라고 가정.
    - 3 가지 경우 존재

- 경우 1
    - $w_i > w$
    - 즉, i번째 물건을 가방에 전혀 넣을 수 없음.
    - 따라서 아래 재귀식 성립

$$P[i][w] = P[i-1][w]$$

- 경우 2
  - $w_i \le w$ 이지만 i번째 물건이 최적 조합에 사용되지 않는 경우

$$P[i][w] = P[i-1][w]$$

- 경우 3
  - $w_i \le w$ 이고 i번째 물건이 최적 조합에 사용되는 경우

$$P[i][w] = p_i + P[i-1][w-w_i]$$

- 정리하면:

$$P[i][w] = \begin{cases} \max(P[i-1][w], p_i + P[i-1][w-w_i]) & \text{if } w_i \leq w, \\ \\ P[i-1][w] & \text{if } w_i > w \end{cases}$$

- 최적화 원칙도 성립함.

동적계획법 알고리즘 구현

- 물건들의 클래스 지정

- NamedTuple 클래스를 활용하면 쉽게 자료형 클래스를 지정할 수 있음.

In [43]:
```
from typing import NamedTuple

class Item(NamedTuple):
    name: str
    weight: int
    value: float
```

- 예제

In [44]:
```
items = [Item("item1", 1, 1),
         Item("item2", 1, 2),
         Item("item3", 2, 2),
         Item("item4", 4, 10),
         Item("item5", 12, 4)]
```

- 각 물건 조합의 최상의 결과를 알려주는 표 작성 알고리즘

In [70]:
```
# 아이템(물건) 개수와 용량 한도
n = len(items)
W = 15
```

- 행렬 P를 영행렬로 초기화 하기

In [71]:
```
# (n+1,W+1) 모양

P = [[0.0 for _ in range(W+1)] for _ in range(n+1)]
```

- P 행렬의 항목을 1번행부터 행단위로 업데이트함.
    - 0번행과 0번열은 그대로 0으로 둠.

In [58]:

```
for i, item in enumerate(items):          # 행 인덱스(물건 번호)는 0부터 시작함에 주의

    wi = item.weight                      # (i+1) 번째 아이템 무게
    pi = item.value                       # (i+1) 번째 아이템 가치

    for w in range(1, W + 1):             # 열 인덱스(용량 한도) 역시 0부터 시작

        previous_items_value = P[i][w]    # i번 행값을 이미 계산하였음. 예를 들어, P[0][w]
 = 0.

        if w >= wi:                       # 현재 아이템의 가방에 들어갈 수 있는 경우

            previous_items_value_without_wi = P[i][w - wi]

            P[i+1][w] = max(previous_items_value,
                            previous_items_value_without_wi + pi)

        else:                             # 현재 아이템이 너무 무거운 경우
            P[i+1][w] = previous_items_value
```

- 위 과정을 하나의 함수로 지정

In [63]:

```
def knapsack(items, W):
    """
    items: 아이템(물건)들의 리스트
    W: 최대 저장용량
    """

    # 아이템(물건) 개수
    n = len(items)

    # P[i][w]를 담는 2차원 행렬을 영행렬로 초기화
    # (n+1) x (W+1) 모양
    P = [[0.0 for _ in range(W+1)] for _ in range(n+1)]

    for i, item in enumerate(items):
        wi = item.weight                          # (i+1) 번째 아이템 무게
        pi = item.value                           # (i+1) 번째 아이템 가치

        for w in range(1, W + 1):
            previous_items_value = P[i][w]    # i번 행값을 이미 계산하였음. i는 0부터 시작함
의 주의할 것
            if w >= wi:                           # 현재 아이템의 무게가 가방에 들어갈 수 있는 경
우
                previous_items_value_without_wi = P[i][w - wi]
                P[i+1][w] = max(previous_items_value,
                                previous_items_value_without_wi + pi)
            else:
                P[i+1][w] = previous_items_value

    return P
```

- 최적의 조합을 알려주는 알려주는 함수
    - 생성된 2차 행렬 P로부터 최적의 조합 찾아낼 수 있음.

In [9]:

```
def solution(items, W):
    P = knapsack(items, W)
    n = len(items)
    w = W

    # 선택 아이템 저장
    selected = []

    # 선택된 아이템을 역순으로 확인
    for i in range(n, 0, -1):
        if P[i - 1][w] != P[i][w]:          # (i-1) 번째 아이템이 사용된 경우. 인덱스가 0부터 출발함에 주의
            selected.append(items[i - 1])
            w -= items[i - 1].weight         # (i-1) 번째 아이템의 무게 제거
    return selected
```

- 획득된 최대 값어치를 알려주는 함수

In [10]:
```python
def max_value(items, W):
    selected = solution(items, W)
    sum = 0

    for item in selected:
        sum += item.value

    return sum
```

활용 1

In [12]:
```
for item in solution(items1, 15):
    print(item)
```

```
Item(name='item4', weight=4, value=10)
Item(name='item3', weight=2, value=2)
Item(name='item2', weight=1, value=2)
Item(name='item1', weight=1, value=1)
```

In [13]:
```
max_value(items1, 15)
```

Out[13]:
```
15
```

활용 2

- 행렬 P를 살펴보기 위한 좀 작은 용량의 배낭채우기 문제

In [61]:
```
items2 = [Item("item1", 1, 5),
          Item("item2", 2, 10),
          Item("item3", 1, 15)]
```

- 최대용향 3까지 허용할 때 최대 값어치로 이루어진 `(4, 4)` 모양의 행렬 P

In [64]:
```python
knapsack(items2, 3)
```

Out[64]:
```
[[0.0, 0.0, 0.0, 0.0],
 [0.0, 5.0, 5.0, 5.0],
 [0.0, 5.0, 10.0, 15.0],
 [0.0, 15.0, 20.0, 25.0]]
```

- 행렬 P로부 최적의 조합 알아내기
  - 오직 아래 등식이 성립할 때 i 번째 아이템이 선택됨.

$$P[i][w] \neq P[i-1][w], \qquad P[i][w] = p_i + P[i-1][w-w_i]$$

- 따라서 $P[4][4]$에서 시작하여 역순으로 사용되는 아이템 확인 가능

In [18]:
```python
for item in solution(items3, 3):
    print(item)
```

```
Item(name='item3', weight=1, value=15)
Item(name='item2', weight=2, value=10)
```

NamedTuple 클래스를 사용하지 않는 경우

- 기본 클래스 정의를 활용하면 해야할 일이 좀 더 많아짐.

In [19]:
```python
class Item1:
    def __init__(self, name, weight, value):
        self.name = name
        self.weight = weight
        self.value = value
```

In [20]:
```python
items4 = [Item1("item1", 1, 1),
          Item1("item2", 1, 2),
          Item1("item3", 2, 2),
          Item1("item4", 4, 10),
          Item1("item5", 12, 4)]
```

In [21]:
```python
for item in solution(items4, 15):
    print(item)
```

```
<__main__.Item1 object at 0x7f9297108ed0>
<__main__.Item1 object at 0x7f9297108e90>
<__main__.Item1 object at 0x7f9297108e50>
<__main__.Item1 object at 0x7f9297108e10>
```

- `__str__()` 메서드 구현 필요

In [22]:
```python
class Item1:
    def __init__(self, name, weight, value):
        self.name = name
        self.weight = weight
        self.value = value

    def __str__(self):
        return 'Item(' + self.name + ', ' + str(self.weight) + ', ' + str(self.val
ue) + ')'
```

In [23]:
```python
items4 = [Item1("item1", 1, 1),
          Item1("item2", 1, 2),
          Item1("item3", 2, 2),
          Item1("item4", 4, 10),
          Item1("item5", 12, 4)]
```

In [24]:
```python
for item in solution(items4, 15):
    print(item)
```

```
Item(item4, 4, 10)
Item(item3, 2, 2)
Item(item2, 1, 2)
Item(item1, 1, 1)
```

시간복잡도

- 입력크기: 물건(item) 수 n과 가장 최대 용량 W
- 단위연산: 채워야 하는 행렬 P의 크기

$$nW \in \Theta(nW)$$

절대 선형이 아님!

- 예를 들어, $W = n!$이면, $\Theta(n \cdot n!)$의 복잡도가 나옴.
- 즉, W값에 복잡도가 절대적으로 의존함.

개선된 알고리즘

- 행렬 P 전체를 계산할 필요 없음.

- $P[n][W]$ 을 계산하기 위해 필요한 값들만 계산하도록 하면 됨.
    - 교재 참조

- 이렇게 구현하면 아래의 복잡도를 갖는 알고리즘 구현 가능

$$O(\min(2^n, nW))$$