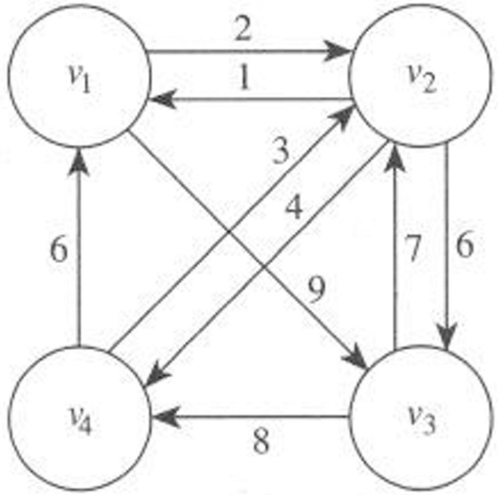
6절 외판원 문제

외판원문제 정의

- 일주여행길: 한 도시에서 출발하여 다른 모든 도시를 한 번씩만 들린 후 출발한 도시로 돌아오는 여행길
- 최적일주여행길: 최소거리 일주여행길
- 외판원문제: 최소한 하나의 일주여행길이 존재하는 가중치포함 방향그래프에서 최적일주여행길 찾기
- 주의: 출발하는 도시가 최적일주여행길의 길이와 무관함.

예제



- v_1 을 출발점으로 하는 일주여행길 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 여행길이 최적.

무차별 대입 방식(brute force) 탐색

- v_1 부터 시작하는 모든 일주여행길을 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: 해당 도시와 연결된 도시와 그 도시와의 거리로 이루어진 사전

In [1]:
```
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 [2]:
```
cities = city_distances.keys()
```

In [3]:
```
print(cities)
```

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

- 모든 경로의 순열조합
    - n개의 도시로 만들 수 있는 모든 순열조합
    - 주의: 이음선이 없는 경우도 포함. 이후 최적일주여행길 선정 과정에서 제외처리될 것임.

In [4]:

```
city_permutations = permutations(cities)
```

- `city_permutation`이 가리키는 값은 아래 항목으로 이루어진 이터러블 자료형

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

- 일주여행길 완성을 위해 출발도시를 마지막 항목으로 추가

In [5]:
```
tsp_paths = [c + (c[0],) for c in city_permutations]
```

- 최단거리 일주여행길 확인

In [6]:
```python
best_path = None                                # 최적일주여행길 저장
min_distance = inf                              # 최적일주여행길 길이 저장

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:
        min_distance = distance
        best_path = path

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

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

부르트포스 탐색 시간복잡도

- 입력크기: 도시(마디) 수 n
- 단위연산: n개의 도시를 일주하는 여행길의 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 최적일주여행길의 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.

다항식 시간복잡도 알고리즘은?

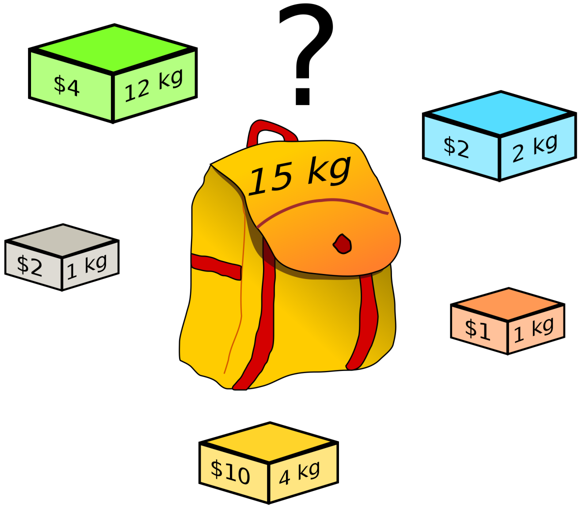
- 다항식 복잡도를 갖는 외판원문제 해결 알고리즘 알려지지 않음.
- 그런 알고리즘이 존재하지 않는다는 증명도 없음.
- 이런 문제를 **NP-hard** 라고 부름.
    - 9장에서 상세히 다룸.

(4장 5절) 0-1 배낭 채우기 문제

0-1 배낭 채우기 문제 정의

- 한정된 용량의 배낭에 주어진 물건을 골라 넣어서 배낭에 담을 수 있는 물건의 최대 이익을 찾는 조합 최적화 문제

예제



<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)

- 최단경로 동적계획법 알고리즘과 유사. 하지만 훨씬 단순.

- 아래 조건을 만족하는 2차원 행렬 P 생성

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

- p_i : i번째 물건의 가치
- w_i : i번째 물건의 무게

- 초기값:

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

- $i > 0$ 이라고 가정

- $w_i > w$ 인 경우:

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

- $w_i \leq w$ 이면서 w_i를 사용하지 않는 경우:

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

- $w_i \leq w$ 이면서 w_i를 사용하는 경우:

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

- 정리하면:

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

- 최적화 원칙도 성립함.

동적계획법 알고리즘 구현

- 물건들의 클래스 지정

- NamedTuple 클래스를 활용하면 쉽게 자료형 클래스를 지정할 수 있음.

In [7]:
```
from typing import NamedTuple

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

- 각 물건 조합의 최상의 결과를 알려주는 표 작성 알고리즘

In [8]:
```python
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

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 [11]:
```
items1 = [Item("item1", 1, 1),
          Item("item2", 1, 2),
          Item("item3", 2, 2),
          Item("item4", 4, 10),
          Item("item5", 12, 4)]
```

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

In [14]:
```python
items2 = [Item("item1", 5, 50),
          Item("item2", 10, 60),
          Item("item3", 20, 140)]
```

In [15]:
```python
for item in solution(items2, 30):
    print(item)
```

```
Item(name='item3', weight=20, value=140)
Item(name='item2', weight=10, value=60)
```

활용 3

In [16]:
```
items3 = [Item("item1", 1, 5),
          Item("item2", 2, 10),
          Item("item3", 1, 15)]
```

In [17]:
```
knapsack(items3, 3)
```

Out[17]:
```
[[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]]
```

In [18]:
```
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]:
```
class Item1:
    def __init__(self, name, weight, value):
        self.name = name
        self.weight = weight
        self.value = value
```

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

In [21]:
```
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]:
```
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]:
```
items4 = [Item1("item1", 1, 1),
         Item1("item2", 1, 2),
         Item1("item3", 2, 2),
         Item1("item4", 4, 10),
         Item1("item5", 12, 4)]
```

In [24]:
```
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

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