과제 1009

2315028 김성현

▼ 복잡한 환경 탐색

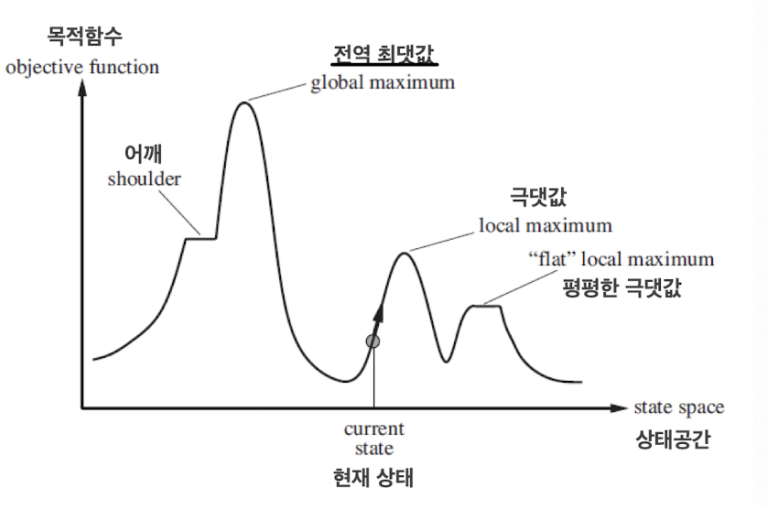
최종 상태만 발견하면 나머지 재구성은 어렵지 않음. 최종상태만 중요하고 경로가 중요하지 않은 문제들

- 국소탐색(local search) : 현재 상태를 개선하는 방향으로 탐색. 대기열이 없음(탐색하지 않는것은 그냥 버림). 앞만 보고 계속 탐색. (=언덕오르기, 모의정련, 담금질, 유전 알고리즘)

  ▼ 국소탐색의 특징

  1. 메모리 소모가 적음
  2. 무한한상태공간에서도 적절한 해답을 찾을 수 있음
  3. 목적함수를 기준으로 가장 좋은 상태를 갖는 최적화 문제에 유용함
  4. 명시적 목표상태가 존재하지 않을 수 있음

     - 목적함수 : 문제에서 최적화하고자하는 목표대상이 되는 함수



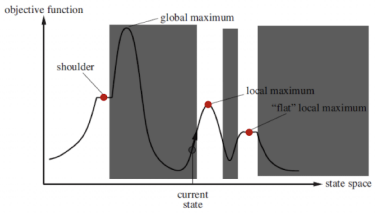
▼ 언덕오르기 : 목적함수를 최대화하는 전역 최댓값을 찾는 과정

현재상태에서 가장 큰 값을 가진 다음 상태로 이동하는 방법. 이웃한 상태만을 살펴봄(그 밖의 다른 값은 살피지 않음). 정상에 도달하면 종료됨. 대기열을 필요로하지 않음

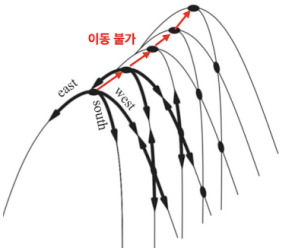
```
Function HILL_CLIMBING(problem)
  current_solution ← problem.initial_solution
  while True
    next_solution ← current_solution의 이웃 상태 중 가장 높은 값을 가진 상태
    if next_solution의 값 > current_solution의 값
      current_solution ← next_solution
  return current_solution
```

▼ 한계 : 상태공간지형의 형태에 크게 영향을 받음.

1. 국소최댓값 - 국소 최댓값에 빠져 전역 최댓값으로 착각할 수 있음
2. 능선 - 능선에 거쳐서 올라가면 문제 풀기가 어려움
3. 고원 - 고원을 만나면 현재 상태를 개선할 수 없다고 착각함



국소최솟값



능선

▼ 한계 극복

1. 횡이동 - 횟수에 제한 두지 않고 계속 나아갈 수 있도록 함
2. 확률적 언덕 오르기 - 무작위로 높지 않아도 올라갈 수 있도록 함(경사가 가파를 수록 갈 확률 높아짐)
3. 무작위 재시작 언덕 오르기 - 시작상태를 무작위로 생성해 여러번 시도하는 방식

- 경사 하강법 : 비용함수(목적함수)를 최소화하는 전역 최솟값을 찾는 과정

▼ 모의정련(simulated annealing) : 정련(가열했다가 식혀서 단련)과정에서 착안한 과정

$$p = e^{\frac{E_2 - E_1}{kT}}$$

e = 자연상수
$E1$ = 기존 상태(해)
$E2$ = 새로운 상태(해)
k = 온도 감소율
T = 온도

- 열역학에서 온도 T에 따라 입자가 에너지 상태 E에 분포하는 확률을 나타내는 볼츠만 분포(Boltzman Distribution) 공식에 기반해, 특정 에너지 상태에서 새로운 에너지 상태로의 전이가 일어날 때 얼마나 상태가 나빠지는지에 대한 확률을 보여줌

상태의 변화에 따라 얼마나 손해, 손실을 얻는가를 확률적으로 나타냄

```
Function SIMULATED_ANNEALING(problem, schedule)
  current_solution ← problem.initial_solution
  for t = 1 to ∞
    T ← schedule(t)
    if T = 0
      return current_solution
    next_state ← current_solution의 이웃 상태 중 무작위 선택
    ΔE ← next_solution의 값 - current_solution의 값
    if ΔE > 0
      current_solution ← next_solution
    else
      current_solution ← next_solution # 단, e^(ΔE/T)의 확률로!
```

시간이 늘어날때마다 온도를 감소시키게 함

schedule(t) : 시간이 얼마나인지에 따라 온도를 조정

언덕오르기와 다른점 : 모의정련은 무작위로 선택하고 델다 e에서 높은지 낮은지 판단.

언덕오르기는 무조건 높은 곳으로 감

- E2 - E1 : 기존의 상태에서 새로운 상태로의 이동에 따른 차이

  → 양수, 이동시 상황이 좋아지는 경우 → 무조건 올라감

  → 음수, 이동시 상황이 나빠지는 경우 → 갈지, 말지 결정

- T : 확률을 변하게 함

  → 높은경우일수록, 받아들일 확률이 높아지게 됨(1에 가까워지게 됨)

  → 낮아질 수록, 받아들일 확률이 0에 가까워짐

  처음에 높게 설정했다가 점점 낮게 설정

  처음에 많이 받아들이다가 점점 적게 받아들임

- K : 온도감소율, 온도가 감쇠하는 속도를 결정

- P : 상태이동시 임의로 나쁜 해를 받아들일지 말지 결정

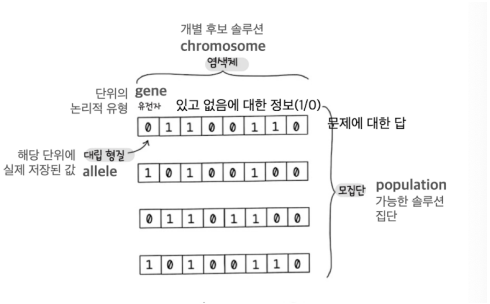
⇒ 확률적 방법을 활용해 극댓값에 빠졌을때 빠져나오기 위한 방법

▼ 유전알고리즘(genetic algorithm) : 진화론에 기반해 최적해를 찾아내는 최적화 방법 일종

▼ 진화론 : 다윈 자연선택설에 기반, 자손은 부모 유전자의 조합을 가지며 돌연변이 과정을 통해 유전자에게 작은 변화를 지니게 됨

모집단의 개체 중 환경에 잘 적응하는 개체만이 살아남는 적자 생존 법칙에 따라 종이 진화

진화론을 기반으로 한 알고리즘은 여러개 존재함.



유전 알고리즘은 좋은 솔루션을 찾기 위해 유전자로 표현된 유전체 집단의 큰 탐색공간을 탐색.

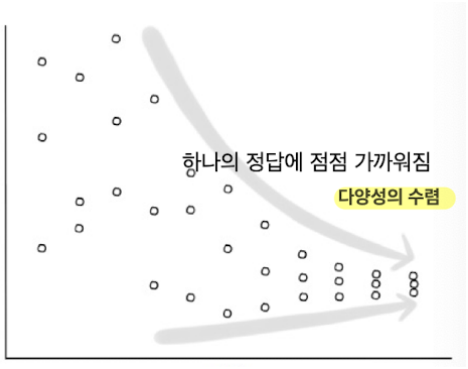
항상 최적의 솔루션을 보장하진 못하지만 전역적으로는 최고의 솔루션을 찾으려함

- 인코딩 방식

  이진 인코딩 : 0과 1로 존재여부를 표기

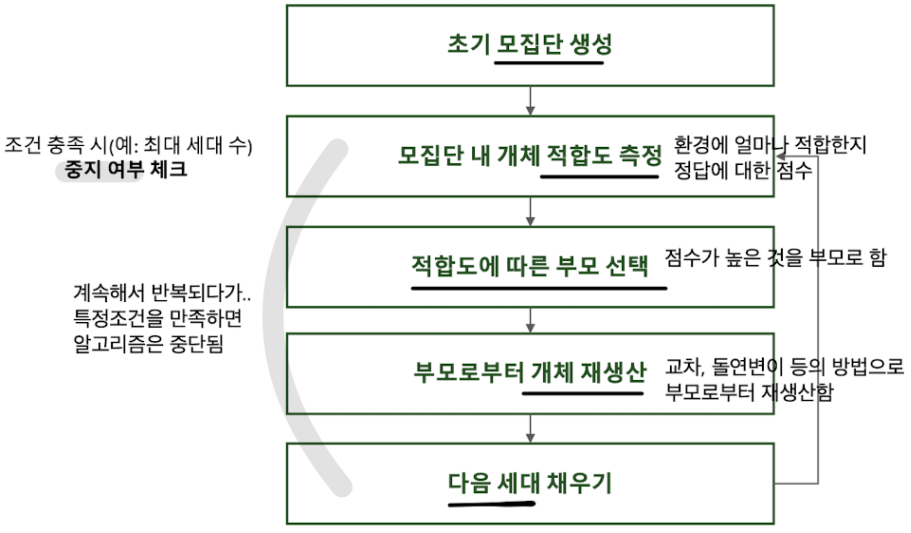
  실숫값 인코딩 : 수치를 넣어 표기

  → 실숫값 인코딩을 사용하면 더 복잡해짐



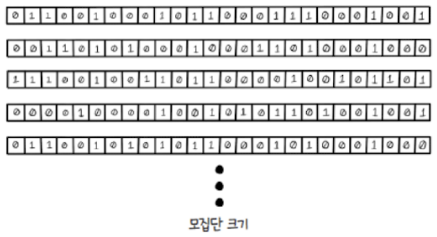
국소 최적해에 빠지지않게 하기위해 처음에는 다양성을 지니도록 만들며 세대를 거치며 점차 수렴시켜 하나의 솔루션, 정답에 가까워지게 함

▼ 과정



1. 초기 모집단 생성

2. 모집단 내 개체 적합도 측정

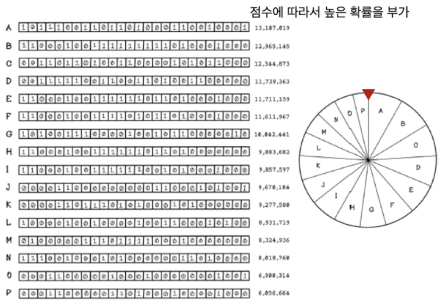


무작위로 초기 모집단을 생성

단, 문제의 제약조건으로 고려해 설계해야함

ex_ 제약조건 위반시 나쁜 적합도 점수 부여

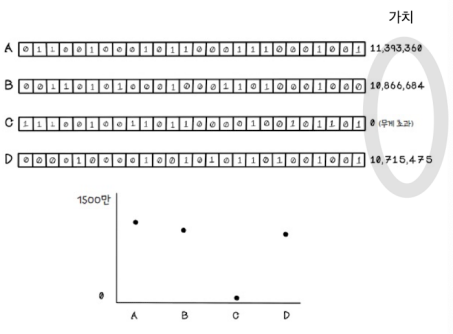
3. 적합도에 따른 부모 선택



적합도에따른 새로운 개체의 부모가 될 확률을 결정

룰렛 휠 선택 : 모든 염색체를 선택할 확률을 적합도에 따라 무작위로 선택함

5. 다음 세대 채우기



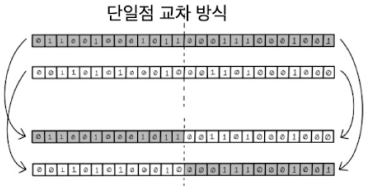
적합도 측정 - 솔루션의 성능, 새로운 개체 및 부모 선택과정에 영향을 미침. 좋은 솔루션을 찾기 위한 방법

적합도 함수를 문제에 맞게 최대, 최소화함

4. 부모로부터 개체 재생산 ⇒ 자손 번식

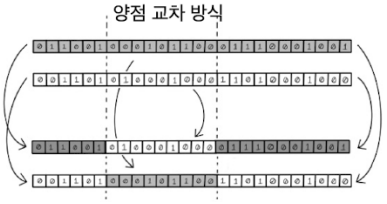
- 교차방식 : 부와 모의 염색체 일부를 혼합해 자손개체를 생성

  1)단일 점 교차



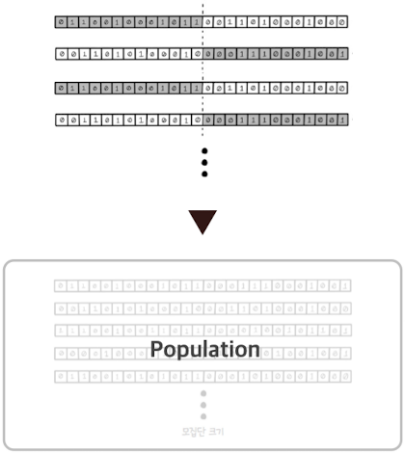
부, 모로부터 한 부분씩 상속

  2)양 점 교차

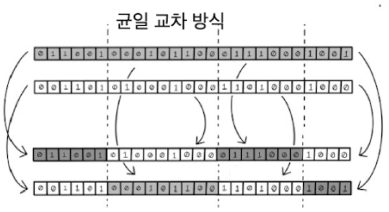


여러 부분을 상속

  3)균일 교차



고정된 모집단 크기만큼 다음세대를 위한 개체를 선택함



많은 부분을 교차해 상속

- 돌연변이 : 자손개체를 무작위로 변형해 모집단에 변화를 줌

  1)문자열 돌연변이 : 유전자를 다른값으로 변경

  2)반전 돌연변이 : 유전자를 반대값으로 변경

▼ 알고리즘

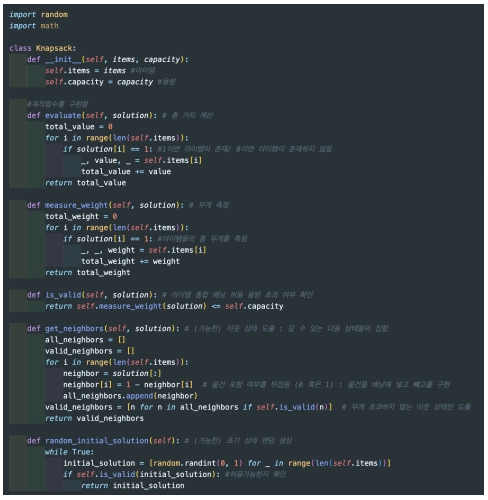
```
                                        모집단 크기          확률들          세대를 거듭하는 과정
Function GENETIC_ALGORITHM(problem, population_size, mutation_rate, max_generations)
  population ← problem.random_initial_population(population_size) 초기 모집단
  best_solution ← null
  best_fitness ← 0

  For generation = 1 to max_generations do      개체별 적합도 측정
    fitness_scores ← [problem.fitness(individual) for individual in population]

    max_fitness_in_gen ← max(fitness_scores) 세대 안에서 가장 큰 값을 찾음
    if max_fitness_in_gen > best_fitness then
      best_fitness ← max_fitness_in_gen                                  가장 큰값을 지닌
      best_solution ← population[fitness_scores.index(max_fitness_in_gen)] 인덱스를 찾음

    new_population ← []
    For i = 1 to population_size // 2 do 모집단의 크기 유지(절반만큼을 계속해서 순회함)
      parent1 ← problem.select(population, fitness_scores) 개체 적합도를 바탕으로
      parent2 ← problem.select(population, fitness_scores) 부모를 선택함

      child1, child2 ← problem.crossover(parent1, parent2)

      if (random() < mutation_rate) then child1 ← problem.mutate(child1) 돌연변이 적용
      if (random() < mutation_rate) then child2 ← problem.mutate(child2)
           확률을 바탕으로 변이가 일어날 수 있게 함
      new_population.append(child1)
      new_population.append(child2)

    population ← new_population 다음세대로 교체

  return best_solution, best_fitness   가장 좋은 정답을 찾음
```

초기 모집단 생성 → 모집단 내 개체 적합도 측정(개체 별 적합도 계산 / 현 세대 최고 적합개체 갱신) → 개체 적합도 기반 부모 선택 → 개체 재생산 (자식생성/ 돌연변이 적용) → 다음 세대 채우기

▼ 복잡한환경 탐색

▼ 배낭문제

: 가방(Knapsack)에 물건을 여러 개 담을 때 가방에 담은 물건들의 조합의 가치를 극대화 할 수 있는 방법을 찾는 문제
단, 배낭의 허용 용량을 초과하여 아이템을 담을 수 없다는 제약이 있음



```python
import random
import math

class Knapsack:
    def __init__(self, items, capacity):
        self.items = items #아이템
        self.capacity = capacity #용량

    #목적함수를 구현함
    def evaluate(self, solution): # 총 가치 계산
        total_value = 0
        for i in range(len(self.items)):
            if solution[i] == 1: #(1)이면 아이템이 존재/ 0이면 아이템이 존재하지 않음
                _, value, _ = self.items[i]
                total_value += value
        return total_value

    def measure_weight(self, solution): # 무게 측정
        total_weight = 0
        for i in range(len(self.items)):
            if solution[i] == 1: #아이템들의 총 무게를 측정
                _, _, weight = self.items[i]
                total_weight += weight
        return total_weight

    def is_valid(self, solution): # 아이템 총합 배낭 허용 용량 초과 여부 확인
        return self.measure_weight(solution) <= self.capacity

    def get_neighbors(self, solution): # (가능한) 이웃 상태 도출 : 갈 수 있는 다음 상태들의 집합
        all_neighbors = []
        valid_neighbors = []
        for i in range(len(self.items)):
            neighbor = solution[:]
            neighbor[i] = 1 - neighbor[i]  # 물건 포함 여부를 뒤집음 (0 혹은 1) : 물건을 배낭에 넣고 빼고를 구현
            all_neighbors.append(neighbor)
        valid_neighbors = [n for n in all_neighbors if self.is_valid(n)]  # 무게 초과하지 않는 이웃 상태만 도출
        return valid_neighbors

    def random_initial_solution(self): # (가능한) 초기 상태 랜덤 생성
        while True:
            initial_solution = [random.randint(0, 1) for _ in range(len(self.items))]
            if self.is_valid(initial_solution): #유효가능한지 확인
                return initial_solution
```

`def evaluate( self , solution )` : *총 가치 계산*

`def measure_weight( self , solution )` : *무게 측정*

`def is_valid( self , solution )` : *아이템 총합 배낭 허용 용량 초과 여부 확인*

`def get_neighbors( self , solution )` : *(가능한) 이웃 상태 도출 : 갈 수 있는 다음 상태들의 집합*

`def random_initial_solution( self )` : *(가능한) 초기 상태 랜덤 생성*

▼ 언덕오르기 알고리즘

: 하나의 현재 상태를 추적하며, 반복마다 가장 큰 값을 가진 이웃상태로 이동하는 탐욕적인 국소탐색기법

```
Function HILL_CLIMBING(problem)
  current_solution ← problem.initial_solution
  while True
    next_solution ← current_solution의 이웃 상태 중 가장 높은 값을 가진 상태
    if next_solution의 값 > current_solution의 값
      current_solution ← next_solution
  return current_solution
```

의사코드

```python
initial_state = Knapsack(items, capacity)
solution, total_value = hill_climbing(initial_state)

# 결과 출력
selected_items = [items[i][0] for i in range(len(items)) if solution[i] == 1]
print(f"결과값: {total_value}, 아이템 목록: {solution}(={selected_items})")
```

```python
import random

def hill_climbing(knapsack):
    # 실습
    current_solution = knapsack.random_initial_solution()
    current_value= knapsack.evaluate(current_solution)
    count =1
    while True:
        neighbors = knapsack.get_neighbors(current_solution)
        evaluated_neighbors = [knapsack.evaluate(n) for n in neighbors]
        print(f"{count}번째 이웃상태: {evaluated_neighbors}")

        next_soultion = None
        for n in neighbors:
            n_value = knapsack.evaluate(n)
            if n_value > current_value:
                next_soultion = n
                current_value = n_value

        if next_soultion is None:
            break

        current_soultion = next_soultion
        count +=1
    return current_solution, current_value
```

```
1번째 이웃상태: [50, 380, 100, 90, 30, 140, 95, 180]
2번째 이웃상태: [50, 380, 100, 90, 30, 140, 95, 180]
결과값: 380, 아이템 목록: [1, 0, 0, 0, 1, 0, 0, 0](=['책', '간식'])
```

가장 큰 값을 지닌 이웃상태 '책', '간식'
380가치를 지닌 이웃을 선택

▼ 모의 정련 알고리즘

: 금속, 유리의 정련과정을 모사한 확률적 요소를 도입한 국소 탐색기법

```
Function SIMULATED_ANNEALING(problem)
  current_solution ← problem.initial_solution
  t ← 0
  while True
    T ← schedule(t)
    if T = 0
      return current_solution
    next_state ← current_solution의 이웃 상태 중 무작위 선택
    ΔE ← next_solution의 값 − current_solution의 값
    if ΔE > 0
      current_solution ← next_solution
    else
      current_solution ← next_solution #단, e^(ΔE/T)의 확률로!
    t += 1
  return current_solution
```

- 단, schedule의 경우 냉각 스케쥴링 함수(t가 올라감에 따라 온도를 어떻게 감소시킬지 결정) 필요
- (본 수업에서는 cooling_rate을 도입하여, 시간이 지남에 따라 온도가 선형적으로 감소하도록 만들 계획)

```python
def cooling_schedule(t, initial_temp, cooling_rate):
    temp = initial_temp - cooling_rate * t # 반복마다 선형적으로 온도가 감소
    if temp < 0:
        temp = 0
    return temp

def simulated_annealing(knapsack, initial_temp=1000, cooling_rate=0.95):
    # 실습
    current_solution = knapsack.random_initial_solution()
    current_value= knapsack.evaluate(current_solution)
    count =1
    t=0


    while True:
        T = cooling_schedule(t, initial_temp, cooling_rate)
        if T == 0:
            break
        neighbors = knapsack.get_neighbors(current_solution)
        evaluated_neighbors = [knapsack.evaluate(n) for n in neighbors]
        print(f"{count}번째 이웃상태: {evaluated_neighbors}")

        next_solution = random.choice(neighbors)
        next_value = knapsack.evaluate(next_solution)
        delta_e = next_value - current_value
        if delta_e >0 :
            current_solution = next_solution
            current_value = next_value
        else:
            if random.random() < math.exp(delta_e/T):
                current_solution = next_solution
                current_value = next_value

        if next_solution is None:
            break

        current_soultion = next_solution
        count +=1
        if count == 1000:
            break
        t+=1
    return current_solution, current_value
    return current_solution, current_value
```

```python
initial_state = Knapsack(items, capacity)
solution, total_value = simulated_annealing(initial_state)

# 결과 출력
selected_items = [items[i][0] for i in range(len(items)) if solution[i] == 1]
print(f"결과값: {total_value}, 아이템 목록: {solution}(={selected_items})")
```

```
1번째 이웃상태: [80, 90, 120, 160, 50, 125, 210]
2번째 이웃상태: [110, 380, 60, 90, 130, 20, 95, 180]
3번째 이웃상태: [80, 360, 320, 395]
4번째 이웃상태: [95, 375, 335, 380]
5번째 이웃상태: [80, 360, 320, 395]
6번째 이웃상태: [350, 20, 300, 330, 370, 380, 335]
7번째 이웃상태: [330, 0, 320, 310, 350, 360, 315, 400]
8번째 이웃상태: [350, 20, 300, 330, 370, 380, 335]
9번째 이웃상태: [30, 310, 320]
10번째 이웃상태: [60, 330, 10, 20, 80, 90, 45, 130]
11번째 이웃상태: [110, 60, 70, 30, 140, 95, 180]
12번째 이웃상태: [160, 170, 130, 80]
13번째 이웃상태: [110, 60, 70, 30, 140, 95, 180]
14번째 이웃상태: [60, 330, 10, 20, 80, 90, 45, 130]
15번째 이웃상태: [30, 40, 50, 110, 120, 75, 160]
16번째 이웃상태: [60, 330, 10, 20, 80, 90, 45, 130]
17번째 이웃상태: [30, 310, 320]
18번째 이웃상태: [350, 20, 300, 330, 370, 380, 335]
19번째 이웃상태: [70, 350, 320]
20번째 이웃상태: [100, 370, 50, 80, 20, 130, 85, 170]
21번째 이웃상태: [200, 150, 180, 120, 230, 185, 70]
22번째 이웃상태: [150, 100, 130, 170, 180, 135, 20]
23번째 이웃상태: [165, 115, 145, 185, 195, 120, 35]
24번째 이웃상태: [150, 100, 130, 170, 180, 135, 20]
25번째 이웃상태: [160, 110, 120, 180, 190, 145, 30]
...
997번째 이웃상태: [190, 180, 170, 210, 100, 175, 60]
998번째 이웃상태: [205, 195, 185, 225, 115, 160, 75]
999번째 이웃상태: [215, 205, 175, 235, 125, 170, 85]
결과값: 215, 아이템 목록: [1, 0, 0, 1, 0, 1, 1, 1](=['책', '필통', '이어폰', '공책', '전공책'])
```

999번째 시도끝에 '책', '필통', '이어폰', '공책', '전공책' 215 가치를 지닌 이웃을 선택

▼ 유전 알고리즘

: 진화론의 교차, 돌연변이 등의 아이디어를 사용해 세대를 거듭할수록 다양성을 줄여가 최적해를 찾아가는 기법

- Knapscak 변형

```python
class KnapsackGA:
    def __init__(self, items, capacity):
        self.items = items
        self.capacity = capacity
        self.chromosome_size = len(items)  # 염색체 크기 (아이템의 개수)

    # 개체 적합도 계산
    def fitness(self, chromosome):
        total_value = 0
        total_weight = 0
        for i in range(self.chromosome_size):
            if chromosome[i] == 1:  # 해당 아이템을 선택한 경우
                _, value, weight = self.items[i]
                total_value += value
                total_weight += weight
        if total_weight > self.capacity:  # 배낭 용량 초과 시 적합도 0으로 처벌하기
            return 0
        return total_value

    # 초기 모집단 생성
    def random_initial_population(self, population_size):
        population = []
        for _ in range(population_size):
            chromosome = [random.randint(0, 1) for _ in range(self.chromosome_size)] #염색체를 모집단의 크기만큼 랜덤으로 설정
            population.append(chromosome)
        return population

    # 부모 선택 (룰렛 휠 선택 - 적합도를 기반에 확률적으로 뽑음)
    def select(self, population, fitness_scores):
        total_fitness = sum(fitness_scores) # 전체 적합도 합계

        if total_fitness == 0:
            return random.choice(population)  # 적합도가 모두 0일 경우 임의의 부모 반환

        pick = random.uniform(0, total_fitness) # 0에서 total_fitness 사이의 무작위 값 선택 (룰렛 휠에서의 선택)

        # 적합도를 누적하여 룰렛 휠에서 선택된 개체 찾기
        current = 0
        for i in range(len(population)):
            current += fitness_scores[i]  # 적합도를 누적
            if current > pick:  # 누적된 값이 현재 누적된 적합도보다 작으면 개체 선택
                return population[i]
        return population[0] # 기본으로 첫 번째 개체 반환
```

```python
# 교차
def crossover(self, parent1, parent2):
    i = random.randint(1, self.chromosome_size - 1)
    child1 = parent1[:i] + parent2[i:]  # 부모특성을 섞음
    child2 = parent2[:i] + parent1[i:]
    return child1, child2

# 돌연변이
def mutate(self, chromosome, mutation_rate):
    for i in range(self.chromosome_size): # 유전자를 하나씩 꺼내서 유전자를 뒤집을지 말지 결정
        if random.random() < mutation_rate: #mutation_rate가 높으면 급격히 바뀌기에 낮은값으로 시작
            chromosome[i] = 1 - chromosome[i]  # 유전자를 0에서 1로, 또는 1에서 0으로 뒤집음
```

```python
knapsack_problem = KnapsackGA(items, capacity)

# 유전 알고리즘 설정
population_size = 10  # 모집단 크기
mutation_rate = 0.05  # 돌연변이 확률
max_generations = 150  # 최대 세대 수

# 유전 알고리즘 실행
best_solution, best_fitness = genetic_algorithm(knapsack_problem, population_size, mutation_rate, max_generations)

print(f"\n최적 해: {best_solution}, 총 가치: {best_fitness}")
```

```
0 세대 최대 적합도 : 722
1 세대 최대 적합도 : 797
2 세대 최대 적합도 : 797
3 세대 최대 적합도 : 797
4 세대 최대 적합도 : 797
5 세대 최대 적합도 : 830
6 세대 최대 적합도 : 830
7 세대 최대 적합도 : 830
8 세대 최대 적합도 : 830
9 세대 최대 적합도 : 830
10 세대 최대 적합도 : 830
11 세대 최대 적합도 : 830
12 세대 최대 적합도 : 830
13 세대 최대 적합도 : 830
14 세대 최대 적합도 : 830
15 세대 최대 적합도 : 830
16 세대 최대 적합도 : 830
17 세대 최대 적합도 : 830
18 세대 최대 적합도 : 830
19 세대 최대 적합도 : 830
20 세대 최대 적합도 : 840
21 세대 최대 적합도 : 840
22 세대 최대 적합도 : 840
23 세대 최대 적합도 : 840
24 세대 최대 적합도 : 840
...
148 세대 최대 적합도 : 890
149 세대 최대 적합도 : 890

최적 해: [0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1], 총 가치: 890
```

- 유전 알고리즘

```
Function GENETIC_ALGORITHM(problem, population_size, mutation_rate, max_generations)
  population ← problem.random_initial_population(population_size)
  best_solution ← null
  best_fitness ← 0

  For generation = 1 to max_generations do
    fitness_scores ← [problem.fitness(individual) for individual in population]

    max_fitness_in_gen ← max(fitness_scores)
    if max_fitness_in_gen > best_fitness then
      best_fitness ← max_fitness_in_gen
      best_solution ← population[fitness_scores.index(max_fitness_in_gen)]

    new_population ← []
    For i = 1 to population_size // 2 do
      parent1 ← problem.select(population, fitness_scores)
      parent2 ← problem.select(population, fitness_scores)

      child1, child2 ← problem.crossover(parent1, parent2)

      if (random() < mutation_rate) then child1 ← problem.mutate(child1)
      if (random() < mutation_rate) then child2 ← problem.mutate(child2)

      new_population.append(child1)
      new_population.append(child2)

    population ← new_population

  return best_solution, best_fitness
```

```python
def genetic_algorithm(knapsack_problem, population_size, mutation_rate, max_generations):

    population = knapsack_problem.random_initial_population(population_size)
    best_solution = None
    best_fitness = 0

    for generation in range(max_generations):
        fitness_scores = [knapsack_problem.fitness(chromosome) for chromosome in population]
        max_fitness_in_gen = max(fitness_scores)

        if max_fitness_in_gen > best_fitness:
            best_fitness = max_fitness_in_gen
            best_solution = population[fitness_scores.index(max_fitness_in_gen)]

        print(f"{generation} 세대 최대 적합도 : {best_fitness}")

        new_population = []

        for _ in range(population_size // 2):
            parent1 = knapsack_problem.select(population, fitness_scores)
            parent2 = knapsack_problem.select(population, fitness_scores)

            child1, child2 = knapsack_problem.crossover(parent1, parent2)
            knapsack_problem.mutate(child1, mutation_rate)
            knapsack_problem.mutate(child2, mutation_rate)

            new_population.append(child1)
            new_population.append(child2)

        population = new_population
    return best_solution, best_fitness
```

149번 시도끝에

가치 890인 세대를 선택함