**학습활동보고서 # 6**

|  |  |  |  |
| --- | --- | --- | --- |
| 대학/학부/학과 | 엘텍공과대학 | 전공 | 컴퓨터공학과 |
| 학번 | 1871001 | 이름 | ZHU ZHAOLING |

|  |  |
| --- | --- |
| 학습활동 주제 및 목표 | 배낭 문제 |

(필요한 항목 및 내용을 자유로운 양식으로 작성하세요)

1. 학습활동 –내용을 상세히 작성합니다.

**6.1 배낭 문제와 탐욕 알고리즘**

**▪ 배낭 채우기: 탐욕 알고리즘 (The Greedy Approach)**

• 탐욕적인 전략: 가장 값어치가 높은 아이템을 먼저 채우는 것

• 1kg당 가격을 기준으로 내림차순으로 정렬

• 배낭의 무게(=30kg)를 초과하지 않을 때까지 비싼 순으로 채우기

**▪ 분할 가능한 배낭 채우기 문제: 탐욕 알고리즘**

• Fractional Knapsack Problem:아이템의 분할이 가능할 경우

• 탐욕 알고리즘으로 최적의 해를 얻을 수 있다.

**Algorithm6.1: Greedy Algorithm for the Fractional Knapsack Problem**

1. **def** knapsack1(W, w, p):
2. n = len(w) - 1
3. K = [0] \* (n + 1)
4. weight = 0
5. **for** i **in** range(1, n + 1):
6. weight += w[i]
7. K[i] = w[i]
8. **if** (weight > W):
9. K[i] -= (weight - W)
10. **break**;
11. **return** K
13. w = [0, 2, 5, 8, 7, 40, 13, 24]
14. p = [0, 15, 12, 8, 8, 7, 5, 2]
15. W = 30
16. K = knapsack1(W, w, p)
17. **print**(K, sum(K))
18. price = 0
19. **for** i **in** range(1, len(K)):
20. price += p[i] \* K[i]
21. **print**("Total price is", price)

**▪ 0-1 배낭 문제: 0-1 Knapsack Problem**

• 분할이 불가능한 0-1 배낭 문제는 최적화 문제이며,

⁃ 탐욕 알고리즘은 최적해를 보장하지 않는다.

• 동적 계획법 or 백트래킹 or 분기한정법

**6.2 0-1 배낭 문제와 동적 계획법**

**▪ 0-1 배낭 문제: 동적 계획법(Dynamic Programming)**

•

⁃

• 재귀 관계식 구하기

**Algorithm 6.2: Dynamic Programming for the 0-1 Knapsack Problem**

1. **def** knapsack2(i, W, w, p):
2. **if** (i <= 0 **or** W <= 0):
3. **return** 0
4. **if** (w[i] > W):
5. **return** knapsack2(i - 1, W, w, p)
6. **else**: # w[i] <= W
7. left = knapsack2(i - 1, W, w, p)
8. right = knapsack2(i - 1, W - w[i], w, p)
9. **return** max(left, p[i] + right)
11. W = 30
12. w = [0, 5, 10, 20]
13. p = [0, 50, 60, 140]
14. profit = knapsack2(len(w)-1, W, w, p)
15. **print**(profit)

**6.3 0-1 배낭 문제와 백트래킹**

**▪ 0-1 배낭 문제: 백트래킹(Backtracking)**

• 최적화 문제 (optimization problem): 전체 탐색이 필요함

• 상태공간트리의 구성: 부분집합의 합 문제와 동일함

⁃ 최적해를 찾는 것이 목표: 공집합을 최적해 집합으로 초기화

⁃ 어떤 아이템을 최적해 집합에 포함시켜서 전체 이익을 계산

⁃ 이 때의 전체 이익이 최적해보다 이익이 많으면 이 집합이 최적해 집합

**▪ 0-1 배낭 문제에 대한 백트래킹 알고리즘**

void checknode ()

{

if (value(v) is better than best)

best = value(v);

if (promising(v))

}

**Algorithm 6.3: Backtracking Algorithm for the 0-1 Knapsack Problem**

1. **def** knapsack3 (i, profit, weight):
2. **global** maxprofit, numbest, bestset
3. **if** (weight <= W **and** profit > maxprofit):
4. maxprofit = profit
5. numbest = i
6. bestset = include[:]
7. **if** (promising(i, profit, weight)):
8. include[i + 1] = True
9. knapsack3(i + 1, profit + p[i+1], weight + w[i+1])
10. include[i + 1] = False
11. knapsack3(i + 1, profit, weight)
13. **def** promising (i, profit, weight):
14. **if** (weight > W):
15. **return** False
16. **else**:
17. j = i + 1
18. bound = profit
19. totweight = weight
20. **while** (j <= n **and** totweight + w[j] <= W):
21. totweight += w[j]
22. bound += p[j]
23. j += 1
24. k = j
25. **if** (k <= n):
26. bound += (W - totweight) \* p[k] / w[k]
27. **return** bound > maxprofit
29. n = 4
30. W = 16
31. w = [0, 2, 5, 10, 5]
32. p = [0, 40, 30, 50, 10]
33. maxprofit = 0
34. numbest = 0
35. bestset = []
36. include = [False] \* (n + 1)
37. knapsack3(0, 0, 0)
38. **print**(bestset[1:len(bestset)])

**6.4 분기 한정법과 0-1 배낭 문제**

**▪ DFS와 BFS**

• 깊이우선탐색: DFS (Depth-First-Search)

• 너비우선탐색: BFS (Breadth-First-Search)

**▪ 분기 한정 (The Branch-and-Bound)**

• 백트래킹과 동일하게 상태공간트리로 문제를 해결

• DFS를 기반으로 하는 백트래킹과 달리, BFS를 기반으로 함 • 최적화 문제에서 유리한 점이 있음: 항상 한계값(bound)를 고려

• 분기한정 가지치기 최적우선탐색

⁃ Best-First-Search with Branch-and-Bound Pruning

**▪ 0-1 배낭 문제: 너비우선탐색으로 풀기**

def breadth\_first\_search (tree T):

queue\_of\_node Q;

node u, v;

initialize(Q);

v = root of T;

visit v;

enqueue(Q, v);

while (! empty(Q)):

dequeue(Q, v);

for (each child u of v):

visit u;

enqueue(Q, u);

▪ 0-1 배낭 문제: 분기 한정법으로 풀기

• 일반적인 방법으로는 BFS가 DFS보다 좋은 점이 없음

• 유망 함수 외에 추가적인 용도로 한계값(bound)을 사용해야 함

• 주어진 어떤 노드의 모든 자식 노드를 방문한 후

⁃ 유망하면서 확장하지 않은 노드들을 모두 살펴보고

⁃ 그 중에서 한계값이 가장 좋은 노들을 우선적으로 확장한다.

• 지금까지 찾은 최적의 해답보다 그 한계값이 더 좋다면 그 노드는 유망함

from queue import PriorityQueue

PQ = PriorityQueue()

data = [4, 1, 3, 2]

for i in range(len(data)):

PQ.put(data[i])

while (not PQ.empty()):

print(PQ.get())

PQ = PriorityQueue()

data = ["Apple", "Banana", "Pear"]

for i in range(len(data)):

PQ.put((len(data[i]), data[i]))

while (not PQ.empty()):

print(PQ.get())

PQ = PriorityQueue()

data = ["Apple", "Banana", "Pear"]

for i in range(len(data)):

PQ.put((-len(data[i]), data[i]))

while (not PQ.empty()):

print(PQ.get()[1])

**Algorithm6.4: Branch-and-Bound for the 0-1 Knapsack Problem**

1. **from** queue **import** PriorityQueue
2. **class** SSTNode:
3. **def** \_\_init\_\_ (self, level, profit, weight):
4. self.level = level
5. self.profit = profit
6. self.weight = weight
7. self.bound = 0
8. **def** **print**(self):
9. **print**(self.level, self.profit, self.weight, self.bound);
11. **def** knapsack4 (p, w, W):
12. PQ = PriorityQueue()
13. v = SSTNode(0, 0, 0)
14. maxprofit = 0
15. v.bound = bound(v, p, w)
16. PQ.put((-v.bound, v))
17. **while** (**not** PQ.empty()):
18. v = PQ.get()[1]
19. **if** (v.bound > maxprofit):
20. level = v.level + 1
21. weight = v.weight + w[level]
22. profit = v.profit + p[level]
23. u = SSTNode(level, profit, weight)
24. **if** (u.weight <= W **and** u.profit > maxprofit):
25. maxprofit = u.profit
26. u.bound = bound(u, p, w)
27. **if** (u.bound > maxprofit):
28. PQ.put((-u.bound, u))
29. u = SSTNode(level, v.profit, v.weight)
30. u.bound = bound(u, p, w)
31. **if** (u.bound > maxprofit):
32. PQ.put((-u.bound, u))
33. **return** maxprofit
35. **def** bound(u, p, w):
36. n = len(p) - 1
37. **if** (u.weight >= W):
38. **return** 0
39. **else**:
40. result = u.profit
41. j = u.level + 1
42. totweight = u.weight
43. **while** (j <= n **and** totweight + w[j] <= W):
44. totweight += w[j]
45. result += p[j]
46. j += 1
47. k = j
48. **if** (k <= n):
49. result += (W - totweight) \* p[k] / w[k]
50. **return** result
52. profit = [0, 40, 30, 50, 10]
53. weight = [0, 2, 5, 10, 5]
54. W = 16
56. maxprofit = knapsack4(profit, weight, W)
57. **print**('maxprofit =', maxprofit)
58. 느낀 점 - 이번 학습활동으로 배운 점 혹은 시행착오를 분석한 후, 다음 학습활동 반영합니다.

동적 프로그래밍 알고리즘으로 0-1 배낭 문제를 푸는 시간 복잡도는 무차별 대입 방법에 비해 크게 감소하다.이해의 핵심은 상태 전이 방정식의 연역 과정에 있다.