**학습활동보고서 # 4**

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

|  |  |
| --- | --- |
| 학습활동 주제 및 목표 | 탐욕 알고리즘 |

(필요한 항목 및 내용을 자유로운 양식으로 작성하세요)

1. 학습활동 –내용을 상세히 작성합니다.

**▪ 탐욕 알고리즘: The Greedy Approach**

• 최종 해답을 찾기 위해서 각 단계마다 하나의 답을 고름

• 각 단계에서 답을 고를 때 가장 좋아 보이는 답을 선택

• 최적화 문제에서,

⁃ 선택할 당시에는 최적의 답을 고르지만 (locally optimal),

⁃ 최종 해답이 반드시 최적임을 보장하지 않음(globally optimal not guaranteed)

**4.1 최소비용 신장트리**

**▪ 최소비용 신장트리 문제의 이해**

• 단순무식하게 풀기(Brute-Force)

⁃ 모든 신장트리를 찾아서 가중치의 합이 가장 작은 것을 선택

• 신장트리를 찾는 법

⁃ 간선의 개수가 n−1인 연결된 트리(acyclic)가 될 때까지 간선을 하나씩 제거

**▪ 최소비용 신장트리: 욕심쟁이 방법(Greedy Approach)**

• 1단계(초기화): 해답의 집합을 공집합으로 둔다.

⁃ 간선 집합 의 부분 집합 공집합으로 둔다.

• 2단계(선택): 최적의 원소 하나를 해답의 집합에 포함시킨다.

• 3단계(검사): 해답의 집합이 최종이면 종료, 아니면 2단계를 반복한다.

**Algorithm 4.1: Prim's Algorithm**

1. **def** prim (W):
2. n = len(W) - 1
3. F = []
4. nearest = [-1] \* (n + 1)
5. distance = [-1] \* (n + 1)
6. **for** i **in** range(2, n + 1):
7. nearest[i] = 1
8. distance[i] = W[1][i]
10. **for** \_ **in** range(n - 1):
11. minValue = INF
12. **for** i **in** range(2, n + 1):
13. **if** (0 <= distance[i] **and** distance[i] < minValue):
14. minValue = distance[i]
15. vnear = i
16. edge = (nearest[vnear], vnear, W[nearest[vnear]][vnear])
17. F.append(edge) # add edge to F
18. distance[vnear] = -1
19. **for** i **in** range(2, n + 1):
20. **if** (distance[i] > W[i][vnear]):
21. distance[i] = W[i][vnear]
22. nearest[i] = vnear
23. **return** F
25. **def** cost (F):
26. total = 0
27. **for** e **in** F:
28. total += e[2]
29. **return** total
30. **def** print\_nd (F, nearest, distance):
31. **print**('F = ', end ='')
32. **print**(F)
33. **print**(' nearest: ', end ='')
34. **print**(nearest)
35. **print**(' distance: ', end ='')
36. **print**(distance)
38. INF = 999
39. W = [
40. [-1, -1, -1, -1, -1],
41. [-1, 0, 1, 3, INF, INF],
42. [-1, 1, 0, 3, 6, INF],
43. [-1, 3, 3, 0, 4, 2],
44. [-1, INF, 6, 4, 0, 5],
45. [-1, INF, INF, 2, 5, 0],
46. ]
47. F = prim(W)
48. **for** i **in** range(len(F)):
49. **print**(F[i])
50. **print**("Minimum Cost is ", cost(F))

▪ 최소비용 신장트리: 크루스칼 알고리즘(Kruskal's Algorithm)

**▪ 사이클 탐지를 어떻게 하지?**

• 서로소 집합 (Disjoint Set)

⁃

• Union-Find 알고리즘

⁃ 서로소 집합 자료구조를 이용해서

⁃ 두 개의 원소가 같은 집합에 속하는 지를 판단할 수 있는 알고리즘

**Algorithm 4.2: Disjoint Set: Union-Find**

1. **class** DisjointSet:
2. **def** \_\_init\_\_ (self, n):
3. self.U = []
4. **for** i **in** range(n):
5. self.U.append(i)
6. **def** equal (self, p, q):
7. **if** (p == q):
8. **return** True
9. **else**:
10. **return** False
11. **def** find (self, i):
12. j = i
13. **while** (self.U[j] != j):
14. j = self.U[j]
15. **return** j
16. **def** union (self, p, q):
17. **if** (p < q):
18. self.U[q] = p
19. **else**:
20. self.U[p] = q

**Algorithm 4.2: Kruskal's Algorithm**

1. **def** kruskal (n, E):
2. F = []
3. dset = DisjointSet(n)
4. **while** (len(F) < n - 1):
5. edge = E.pop(0)
6. i, j = edge[0], edge[1]
7. p = dset.find(i)
8. q = dset.find(j)
9. **if** (**not** dset.equal(p, q)):
10. dset.union(p, q)
11. F.append(edge)
12. **return** F
14. INF = 999
15. n = 5
16. E = [
17. [0, 1, 1],
18. [2, 4, 2],
19. [0, 2, 3],
20. [1, 2, 3],
21. [2, 3, 4],
22. [3, 4, 5],
23. [1, 3, 6],
24. ]
26. F = kruskal(n, E)
27. **for** i **in** range(len(F)):
28. **print**(F[i])

**4.2 최단 경로와 다익스트라 알고리즘**

**▪ 다익스트라 알고리즘:**

• 최소비용 신장트리 문제의 프림 알고리즘과 유사

while (답을 구하지 못했음):

**Algorithm 4.3: Dijkstra's Algorithm**

1. **def** dijkstra (W):
2. n = len(W) - 1
3. F = []
4. touch = [-1] \* (n + 1)
5. length = [-1] \* (n + 1)
6. **for** i **in** range(2, n + 1):
7. touch[i] = 1
8. length[i] = W[1][i]
10. **for** \_ **in** range(n - 1):
11. minValue = INF
12. **for** i **in** range(2, n + 1):
13. **if** (0 <= length[i] **and** length[i] < minValue):
14. minValue = length[i]
15. vnear = i
16. edge = (touch[vnear], vnear, W[touch[vnear]][vnear])
17. F.append(edge)
18. **for** i **in** range(2, n + 1):
19. **if** (length[i] > length[vnear] + W[vnear][i]):
20. length[i] = length[vnear] + W[vnear][i]
21. touch[i] = vnear
22. length[vnear] = -1
23. **return** F
25. **def** length (F):
26. total = 0
27. **for** e **in** F:
28. total += e[2]
29. **return** total
31. **def** print\_tl (F, touch, length):
32. **print**('F = ', end ='')
33. **print**(F)
34. **print**(' touch: ', end ='')
35. **print**(touch)
36. **print**(' length: ', end ='')
37. **print**(length)
39. INF = 999
40. W = [
41. [-1, -1, -1, -1, -1, -1],
42. [-1, 0, 7, 4, 6, 1],
43. [-1, INF, 0, INF, INF, INF],
44. [-1, INF, 2, 0, 5, INF],
45. [-1, INF, 3, INF, 0, INF],
46. [-1, INF, INF, INF, 1, 0],
47. ]
49. F = dijkstra(W)
50. **for** i **in** range(len(F)):
51. **print**(F[i])
53. **print**("Shortest Path Length is", length(F))

**4.3 마감시간 있는 스케줄 짜기**

**▪ 마감시간 있는 스케줄 짜기: 탐욕법(The Greedy Approach)**

• 보상에 따라서 비오름차순으로 작업을 정렬한다.

• 각 작업을 순서대로 하나씩 가능한 스케줄에 포함시킨다.

작업을 보상이 큰 것부터 차례로 정렬한다.

while (답을 구하지 못했음):

다음 작업을 선택.

if (더 이상 남은 작업이 없다 )

답을 구했음.

**Algorithm 4.4: Scheduling with Deadlines**

1. **def** schedule (deadline):
2. n = len(deadline) - 1
3. J = [1]
4. **for** i **in** range(2, n + 1):
5. K = insert(J, i, deadline)
6. **if** (feasible(K, deadline)):
7. J = K[:]
8. **return** J
10. **def** feasible (K, deadline):
11. **for** i **in** range(1, len(K) + 1):
12. **if** (i > deadline[K[i - 1]]):
13. **return** False
14. **return** True
15. deadline = [0, 3, 1, 1, 3, 1, 3, 2]
16. K = [2, 1]
17. **print**(K, feasible(K, deadline))
19. **def** insert(J, i, deadline):
20. K = J[:]
21. **for** j **in** range(len(J), 0, -1):
22. **if** (deadline[i] >= deadline[K[j-1]]):
23. j += 1
24. **break**
25. K.insert(j - 1, i)
26. **return** K
27. J = [1]
28. K = insert(J, 2, deadline)
29. **print**(K)
31. deadline = [0, 3, 1, 1, 3, 1, 3, 2]
32. profit = [0, 40, 35, 30, 25, 20, 15, 10]
33. J = schedule (deadline)
34. **print**("Schedule:", J)
36. maxprofit = 0
37. **for** job **in** J:
38. maxprofit += profit[job]
39. **print**("Max Profit =", maxprofit)

**4.4 허프만 코드와 허프만 알고리즘**

**▪ 허프만 코드 (Huffman's code)**

• 허프만 알고리즘에 의해 생성된 최적 이진코드

**▪ 허프만 알고리즘 (Huffman's algorithm)**

• 허프만 코드에 해당하는 이진트리를 구축하는 탐욕알고리즘

**▪ 허프만 알고리즘: 탐욕 알고리즘(The Greedy Approach)**

주어진 데이터 파일내 문자의 개수

PQ: 빈도수가 낮은 노드를 먼저 리턴하는 우선순위큐(min-heap)

for i in [1..n-1]:

remove(PQ, p)

remove(PQ, q)

r = new node

r->left = p

r->right = q

r->frequency = p->frequency + q->frequency

insert(PQ, r)

remove(PQ, r)

return r

**Algorithm 4.5: Huffman's Algorithm for the Huffman Code**

1. **class** HuffNode:
2. **def** \_\_init\_\_ (self, symbol, freq):
3. self.symbol = symbol
4. self.freq = freq
5. self.left = None
6. self.right = None
8. **def** preorder(self):
9. **print**(self.freq, end=" ")
10. **if** (self.left **is** **not** None):
11. self.left.preorder()
12. **if** (self.right **is** **not** None):
13. self.right.preorder()
15. **def** inorder(self):
16. **if** (self.left **is** **not** None):
17. self.left.inorder()
18. **print**(self.freq, end=" ")
19. **if** (self.right **is** **not** None):
20. self.right.inorder()
22. **def** huffman (n, PQ):
23. **for** \_ **in** range(n - 1):
24. p = PQ.get()[1]
25. q = PQ.get()[1]
26. r = HuffNode(' ', p.freq + q.freq)
27. r.left = p
28. r.right = q
29. PQ.put((r.freq, r))
30. **return** PQ.get()[1]
32. codes = ['b', 'e', 'c', 'a', 'd', 'f']
33. freqs = [5, 10, 12, 16, 17, 25]
35. **from** queue **import** PriorityQueue
36. PQ = PriorityQueue()
37. **for** i **in** range(len(codes)):
38. node = HuffNode(codes[i], freqs[i])
39. PQ.put((node.freq, node))
41. root = huffman(len(codes), PQ)
43. **print**("Preorder:", end=" ")
44. root.preorder()
45. **print**("\nInorder:", end=" ")
46. root.inorder()
47. **print**()
48. 느낀 점 - 이번 학습활동으로 배운 점 혹은 시행착오를 분석한 후, 다음 학습활동 반영합니다.

최단 경로 문제:

• 모든 정점의 쌍에 대한 최단 경로 구하기

⁃ 플로이드 알고리즘: 동적 계획법

• 단일 정점에서 모든 다른 정점으로의 최단 경로 구하기

⁃ 다익스트라 알고리즘: 탐욕법