ch.04 그리디 알고리즘

⇒ 최적성 원칙

4.0 그리디 알고리즘 근시안적인 선택으로 부분적인 최적해 ⇒ 문제의 최적해

최적화 문제를 해결하는 알고리즘 → 가능한 해들 중에서 가장 좋은 해를 찾는 문제

⇒ 탐욕적인 알고리즘!

1. 일단 한 번 선택하면 이를 절대로 번복 x
2. 매우 단순 or 제한적인 문제들만이 그리디 알고리즘으로 해결됨.

4.2 최소 신장 트리(minimum spanning Tree)

- 주어진 그래프의 신장 트리 → 사이클이 없도록 모든 점을 연결시킨 트리 중 간선들의 가중치 합 ↓
  - 그래프의 점의 수 : n, 간선의 수 : n-1 (간선이 n개가 되면 사이클이 생성됨)

선을 골라서 점을 이어 나감

① • Kruskal 알고리즘

→ 그래프가 2개 이상 연결성분 ⇒ 서로 다른 연결성분의 점들 사이에는 경로 X

  - 그래프에서의 연결성분 : 경로가 있는 극대부분을 나타냄
  - 사이클을 만들지 않는 작은 가중치 간선을 차례차례 저장함

```
KruskalMST(G)

//입력 : 가중치 그래프 G=(V,E), |V|=n(점의 수), |E|=m(간선의 수)
//출력 : 최소 신장 트리 T

가중치 오름차순으로 간선 정렬 -> 정렬된 간선 리스트 = L

T=공집합 //T를 초기화

while(T의 간선 수 < n-1){
  L에서 가장 작은 가중치를 가진 간선 e -> e를 L에서제거
  if( 간선 e가 T에 추가되어 사이클을 만들지 않으면)
```

```
    e를 T에 추가
  else //e가 T에 추가되어 사이클이 만들어지는 경우
    e를 버림
  }
return T //T는 최소 신장 트리
}
```

- time complexity
  - 최악
    - i) 간선들을 가중치로 정렬 → O(mlogm) (m은 간선의 수)
    - ii) T 초기화 → O(1)

      while : m보다 무조건 적게 처리
    - while loop → m번(그래프의 모든 간선이 while loop 내에서 처리되는 경우)
      - 내부 : L로부터 가져온 간선 e가 사이클을 만드는지 검사 → O(log*m) (거의 상수시간)
        - disjoint set 연산을 이용하여 사이클 만드는지 검사
    - iii) → O(mlog*m)
  - ∴ ⇒ O(mlogm) + O(mlog*m) = O(mlogm)

② • Prim 알고리즘

점을 골라서 해당 영역 내의 선을 이어나감

임의의 점 선택 → (n-1)개 간선 하나씩 추가

속하지 않은 정을 추가하기에 사이클이 생기지 X

D의 길이

```
PrimMST(G)

//입력 : 가중치 그래프 G=(V,E), |V|=n(점의 수), |E|=m(간선의 수)
//출력 : 최소 신장 트리 T

그래프 G에서 임의의 점 p를 시작점을 선택 후 D[p]=0으로 놓는다
//D[v]는 T에 있는 점 u와 v를 연결하는 간선의 최소 가중치를 저장한다.

for( 점 p가 아닌 각 점 v에 대하여){ //배열 D의 초기화
  if( 간선 (p,v)가 그래프에 있으면 )
    D[v] = 간선 (p,v)의 가중치
  else
    D[v]= ∞
  }
T={p} //초기에 트리 T는 점 p만을 가진다.
while(T에 있는 점의 수 < n){
  T에 속하지 않은 각 점 v에 대하여, D[v]가 최소인 점 Vmin과 연결된 간선 (u, Vmin)을 T에 추가
  (단, u는 T에 속한 점이고, 이때 점 Vmin도 T에 추가됨)

  for(T에 속하지 않은 각 점 w에 대해서){
    if(간선 (Vmin, W)의 가중치 < D[w])
      D[w] = 간선(Vmin, W)의 가중치 //D[w]를 갱신
    }
  }
```

```
return T //T는 최소 신장 트리

//내역 밖에 있는 점들만 계속해서 추가하기에 사이클을 만들지 않음
```

- time complexity
  - while loop : (n-1)번 반복
    - 1회 반복 시에 D[v]가 최소인 점 찾기 → O(n) (배열의 크기 ⇒ n)

      (1차원 배열 D에서 현재 T에 속하지 않은 점들에 대해서 최솟값을 찾는 것)

  ⇒ (n-1) * O(n) = O(n^2)

- kruskal vs prim
  - kruskal : 간선이 하나씩 T에 추가 → tree가 여러 개 → 1개의 신장 트리가 만들어짐
  - prim : T가 점 1개인 Tree에서 시작 → 간선 하나씩 추가 → 1개의 신장 트리

4.3 최단 경로 찾기

- 주어진 가중치 그래프에서 어느 한 출발점에서 또 다른 도착점까지의 최단 경로를 찾는 문제

+) 음수가중치 : 그리디 알고리즘 ⇒ 반복 X , But 음수 가중치가 있는 그래프에서는 반복해야 하는 경우 발생 가능

- prim algorithm과 거의 흡사한 과정으로 진행
  - 다른 점
    1. start : prim → 임의의 점 / Dijkstra → 주어진 출발점에서 시작
    2. 간선 추가 방법
       a. prim : 현재 상태의 트리에서 가장 가까운 점 추가
       b. Dijkstra : 출발점으로부터 최단 거리가 확정되지 않은 점들에서 가장 가까운 점 추가

```
ShortestPath(G,s)

//입력 : 가중치 그래프 G=(V,E), |V|=n(점의 수), |E|=m(간선의 수)
//출력 : 출발점 s로부터 (n-1)개의 점까지 각각 최단 거리를 저장한 배열 D
```

```
배열 D를 ∞로 초기화 시킴(단,  D[s] = 0으로 초기화)

                  (n-1)번 반복
while(s로부터 최단거리가 확정되지 않은 점이 있으면){
  현재까지 s로부터 최단 거리가 확정되지 않은 각 점 v에 대해서
  최소의 D[v]의 값을 가진 점 Vmin을 선택  O(n)
  출발점 s로부터 점 Vmin까지의 최단 거리 D[Vmin]을 확정  O(n)
  s로부터 현재보다 짧은 거리로 점 Vmin을 통해 우회 가능한 각 점 w에 대해서 D[w]을 갱신
}

return D
```

- time complexity
  - while loop → (n-1)번 반복
    1. 최소의 D[v]를 가진 점 Vmin을 찾기 → O(n)
       - Vmin을 찾는 데 heap을 사용하면 → O(mlogn) (m은 입력 그래프의 간선 수)
       - Vmin을 찾는 데 피보나치 heap을 사용하면 → O(m+nlogn)
    2. 출발점 s로부터 점 Vmin까지 최단 거리 D[Vmin] 확정 → O(n)

  ⇒ (n-1) * {O(n) + O(n)} = O(n^2)

4.5 집합 커버 문제

n개의 원소를 가진 집합 U의 부분 집합들을 원소로 하는 집합 F가 주어짐

→ F의 원소들인 집합 중에서 어떤 집합들을 선택하여 합집합하면 U와 같아지는지

ex) 신도시 계획에 있어 학교를 어떻게 배치하여야 하는지 등

(2^n-1) 개를 검사해야 하기 때문

- n이 계속하여 커지면 최적해를 찾는 것은 실질적으로 불가능 → approximation solution

근사알고리즘

```
SetCover()
                  → cover되는 마을의 집합
//입력 : U, F={Si}, i=1 ... n
//출력 : 집합 커버 C

C=∅
while(U!=∅)do{
```

→ 종료조건 : U에 모든 원소가 남아있지 않을 때

```
  U의 원소들을 가장 많이 포함하고 있는 집합 Si를 선택
  U = U-Si
  Si를 F에서 제거하고, Si를 C에 추가
}
return C
```

↳ 더 이상 고려되지 X

- time complexity
  - while loop → 최대 n번 수행(loop 한 번 수행될 때마다 집합 U의 원소 1개씩만 커버)
    - ① loop 검사 조건 → O(1) $U \neq \emptyset$
    - ② Si와 U의 비교 → O(n), Si의 최대 수 → n ⇒ O(n^2) n×O(n)
    - ③ U에서 Si의 원소를 제거 → O(n) n번 확인(최대)
    - ④ Si를 F에서 제거 후 Si를 C에 추가 → O(1)

    loop 1번

  ⇒ {O(1) + O(n^2) + O(n) + O(1)} * O(n) = O(n^2) * O(n) = O(n^3)

  ① ② ③ ④ n번 수행 loop

- approximate algorithm : 근사해가 최적해에 얼마나 근사한지를 나타내는 근사비율을 제시
  - SetCover의 approximatio ratio : Klnn
    - 의미 : 최악 경우의 해일지라도 그 집합 수가 Klnn개를 넘지 않는다는 뜻
  - proof
    -

?.. 잘 모르겠어서 포기함.

4.6 작업 스케줄링

→ 여러 종류가 있음.

↳ 작업의 길이도 주어진 것

기계에서 수행되는 n개의 작업 t1, t2, … tn 이 있고, 각 작업은 시작시간과 종료시간이 있음.

작업의 수행 시간이 중복되지 않도록 모든 작업을 가장 적은 수의 기계에 배정

↳ 기계가 놀지 않게 배정

→ (이 버전은 빠른 시작시간 작업을 우선 배정하는 Greedy 알고리즘)

```
JobScheduling()

//입력 : n개의 작업 t1, t2 ... tn
//출력 : 각 기계에 배정된 작업 순서

시작 시간의 오름차순으로 정렬한 작업 리스트 -> L
while(L!=∅){
  L에서 가장 이른 시작시간을 가진 작업 t1을 가져옴.
  if(ti를 수행할 기계가 있으면)
```

```
    ti을 수행할 수 있는 기계에 배정
  else
    새로운 기계에 ti를 배정
  ti를 L에서 제거 //더 이상 작업 배정에 고려되지 않도록
  }

return 각 기계에 배정된 작업 순서
```

- time complexity
  1 n개의 작업을 정렬 → O(nlogn)
  2 while loop → n번 m = 사용된 기계의 수
    - 작업을 L에서 가져다가 수행 가능한 기계를 찾아서 배정 → O(m)

  ⇒ O(nlogn) + O(mn)
  1 2

4.7 허프만 압축

- 파일 압축
  - 주어진 파일의 크기를 줄이는 방법
  - 파일의 각 문자 : 일반적으로 고정된 크기의 코드로 표현
    - ex. 파일의 각 문자가 8bit ASCII code로 저장되면 bit 수 = 8*(파일의 문자 수)
  - 파일의 크기를 줄이고 필요시 원래의 파일로 변환할 수 있으면 메모리 공간 효율적 사용 + 파일 전송 시간 단축
    ⇒ file compression

- huffman 압축 ∈ file compression
  - 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당,

    드물게 나타나는 문자는 긴 이진 코드 할당
  - prefix property(접두부 특성) : 각 문자에 할당된 이진 코드는 어떤 다른 문자에 할당된 이진 코드의 접두부가 되지 않음.
    - 코드와 코드 사이를 구분할 특별한 코드가 필요 x → 이거 없어도 압축 해제 가능

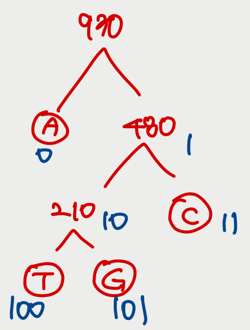
- huffman 코드

- 허프만 압축
  - 입력 파일에 대해 각 문자의 출현 빈도수에 기반을 둔 이진트리 제작
  - 각 문자에 이진 코드를 할당

  ⇒ 이진 코드 ⇒ 허프만 코드

```
 HuffmanCoding()

//입력 : 입력 파일의 n개의 문자에 대한 각각의 빈도수
//출력 : 허프만 트리

각 문자에 대해 노드를 만들고, 그 문자의 빈도수를 노드에 저장
n개의 노드들의 빈도수에 대해 우선순위 큐 Q를 만든다.

while(Q에 있는 노드 수>=2){
   빈도수가 가장 작은 2개의 노드(A와 B)를 Q에서 제거
   새 노드 N을 만들고, A와 B를 N의 자식 노드로 만든다.
   N의 빈도수 <- A의 빈도수 + B의 빈도수
   노드 N을 Q에 삽입
   }

return Q //허프만 트리의 루트를 리턴
```

A	T	G	C
450	90	120	270



압축된 file bit수 : (450×1)+(90×3)+(120×3)+(270×2)=1620
"0" "100" "101" "11"

ASCII code file 크기 : (450+90+120+270) ×8 = 7440

- time complexity
  - 1 n개의 노드를 만들고 각 빈도수를 노드에 저장 → O(n)
  - 2 n개의 노드로 우선순위 큐 Q, heap 사용하면 → O(n)
  - 3 while loop → (n-1)번
    - 최소 빈도수를 가진 노드 2개를 제거(Deleteheap()) + 새 노드를 삽입 → O(logn)

    → (n-1) * O(logn) = O(nlogn)
  - 4 tree의 root를 return → O(1)

  ⇒ O(n) + O(n) + O(logn) + O(1) = O(nlogn)
  1 2 3 4