Analysis of Algorithms
CSC 402
2023-24

Subject Incharge

Dr. Bidisha Roy
Associate Professor
Room No. 401
email: Bidisharoy@sfit.ac.in

Module 3

Greedy Method Approach

Greedy Method

Greed is good.

(Some of the time)

General Method

- Makes the choice that looks the best at that moment
  - Example
    - Taking a shorter route
    - Investing in shares
    - Playing a bridge hand
  - The hope: a locally optimal choice will lead to a globally optimal solution.
- Sometimes they work, sometimes don’t.
- Primarily used to solve optimization problems.

Elements of Greedy Algorithms

- Greedy choice property
  - A globally optimal solution is derived from a locally optimal (greedy) choice.
  - When choices are considered, the choice that looks best in the current problem is chosen, without considering results from sub problems.

Elements of Greedy Algorithms

- Optimal substructure
  - A problem has ***optimal substructure*** if an optimal solution to the problem is composed of optimal solutions to subproblems.

Problems to be considered

- Knapsack Problem
- Single source shortest path
- Minimum Spanning Trees
  - Kruskal's Algorithm
  - Prim's Algorithm
- Job sequencing with deadlines
- Optimal storage tapes

Knapsack Problem

- A thief robbing a store finds *n* items; the i^{th} item is worth c_i cost units and weighs w_i weight units. The thief wants to take as valuable load as possible, but he can carry at most *W* weight units in his knapsack.
- **Which items should he take ???** is the 0-1 knapsack problem (each item can be taken or left)
- To be able to solve using greedy approach, we convert into ***fractional knapsack problem***

Knapsack Problem… Greedy Solution

- Uses the maximum cost benefit per unit selection criteria
  - Sort items in decreasing c_i / w_i.
  - Add items to knapsack (starting at the first) until there are no more items, or the next item to be added exceeds W.
  - If knapsack is not yet full, fill knapsack with a fraction of next unselected item.

Knapsack Problem… Algorithm

- **Knapsack(C, W, M, X, n)**
  - for i ← 1 to n
    - do X[i] ← 0 // Initial Solution
  - RC ← M // Remaining capacity of knapsack
  - for i ← 1 to n
    - do if W[i] > RC then
      - break
    - X[i] ← 1
    - RC ← RC – W[i]
  - if i ≤ n then
    - X[i] ← RC/W[i]

Knapsack Problem… Problem

- M = 25, n = 3, C = (25, 24, 17), W = (16, 14, 9)
- M = 20, n = 3, C = (25, 24, 15), W = (18, 15, 10)
- M=15, n=7, C=(5,10,15,7,8,9,4), W=(1,3,5,4,1,3,2)
- Analysis
  - Depends on time taken to arrange elements in descending order of profit, O(nlogn) if a good sort algorithm is used
  - Other parts of algorithm take O(n) time
  - Average Complexity O(nlogn).

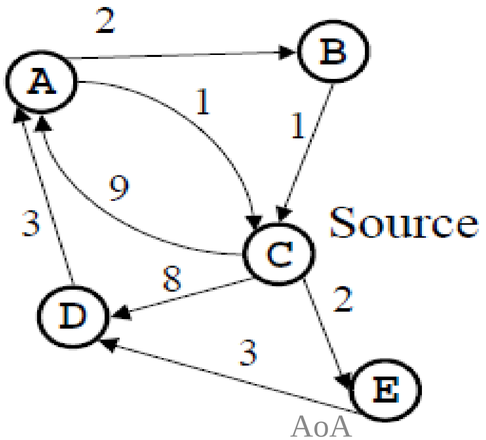
Knapsack Problem… Applications

- Internet download Managers
- Resource Allocation
- Portfolio Optimization
- Cutting stock problems

Single Source Shortest Path

- Given a graph $G = (V,E)$, to find the shortest path from a given **source** vertex ***s* є *V*** to each other vertex ***v* є *V***.

- Algorithms to be considered in greedy approach
  - Djikstra's shortest path Algorithm



Djikstra's Shortest Path Algorithm

- Solves the single-source shortest-paths problem on a ***weighted, directed graph*** where all edge weights are ***nonnegative.***
- Data structure
  - S: a set of vertices whose final shortest-path weights have already been determined
  - Q: a min-priority queue keyed by their distance values
- Idea
  - Repeatedly select the vertex u $\in$V-S (kept in Q) with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

Djikstra's Algorithm … contd

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ V[G]
    do d[v] ← ∞
       π[v] ← NIL
d[s] ← 0
```

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
```

Djikstra's Algorithm … contd

- ```
  void dij(int n, int v, int cost[10][10], int dist[])
  {
  int i,u,count,w,visited [10],min;
  for(i=1;i<=n;i++)
          visited[i]=0;dist[i]=cost[v][i];
   visited[v]=1;count=2;
        while(count<=n)
       {
          min=99;
          for(w=1;w<=n;w++)                    //Extract-min
                  if(dist[w]<min && !visited[w])
                          min=dist[w],u=w;
          visited[u]=1; //A minimum distance vertex is removed from Q
          count++;
          for(w=1;w<=n;w++)
                  if((dist[u]+cost[u][w]<dist[w]) && !visited[w])
                          dist[w]=dist[u]+cost[u][w];
          }
  }
  ```
  ```

Djkstra’s Algorithm … Analysis

- If priority queue maintained as a linear array, each Extract-Min takes |V| time for V vertices, so $O(V^2)$.
- Scanning edges in adjacency list takes O(E) time
- Other operations take linear time of either E or V.
- Average complexity using linear priority queue is $O(V^2)$.

Djikstra's Algorithm … Analysis

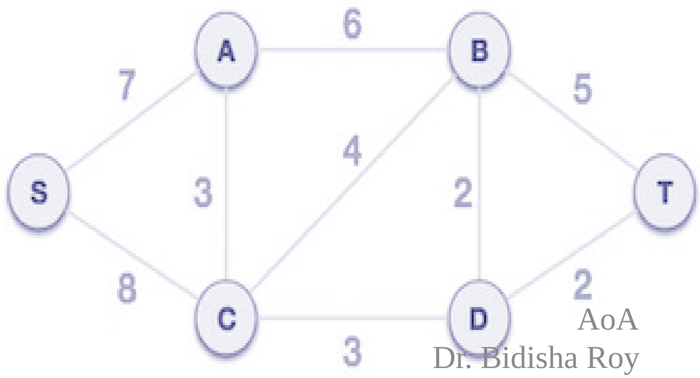
- If priority queue maintained as a binary heap, each Extract-Min takes |logV| time for V vertices, so O(VlogV).
- Build heap operation takes O(V) time.
- Other operations take linear time of either E or V.
- Average complexity using binary heap queue is O(ElogV).

Djikstra's Algorithm … Applications

- Route Planning and Navigation Systems/ Maps
- IP routing to find **OSPF**
- Telephone/Cellular Networks
- https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6305611 (The application of Dijkstra's algorithm in the intelligent fire evacuation system)
- Pathfinding in Video Games and Robotics
- Social Network Analysis

Spanning Trees

- Given a connected, undirected graph, a ***spanning tree*** of that graph is a subgraph which is a tree and connects all the vertices together.
- Let *G = (V, E)* be an undirected connected graph. A subgraph *T = (V, E')* of G is a ***spanning tree*** iff *T* is a tree.



Minimum Spanning Trees

- A ***minimum spanning tree*** (MST) or ***minimum weight spanning tree*** is then a spanning tree with weight less than or equal to the weight of every other spanning tree.
- Let $G = (V, E)$ be an undirected graph.

  $T = (V, E')$ is a ***minimum spanning tree*** of G if T E is an acyclic subset that connects all of the vertices and whose total weight $w(T) = \sum$ $w(u, v)$ [where u, v belong to V] is minimized.

Minimum Spanning Trees

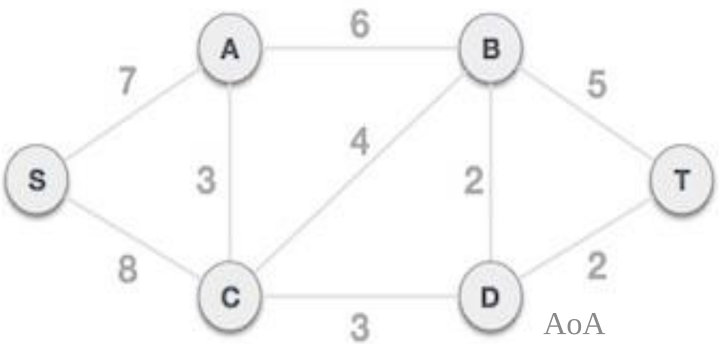
- Greedy Algorithms for MST
  - Kruskal's Algorithm
  - Prim's Algorithm

Kruskal's Algorithm

- Was put forward by Joseph Kruskal.
- In Kruskal's algorithm,
  - The set A is a forest.
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.

Kruskal's Algorithm … contd

- MST_KRUSKAL (G, w)
  - A ← Ø
  - for each vertex *v* ϵ V[G]
    - do Make-Set (v)
  - Sort the edges of *E* by increasing weight w
  - for each edge *(u, v)* ϵ *E,* in order by nondecreasing weight
    - do if Find-Set(u) ≠ Find-Set(v) then
      - A ← A U { (u, v) }
      - Union (u,v)
  - Return *A.*



Kruskal’s Algorithm … contd

- Make-Set(x)-creates a new set whose only member is x.
- Find-Set(x)- returns a representative element from the set that contains x
  - determine whether two vertices u and v belong to the same tree by testing whether FIND_SET(u) equals FIND_SET(v).
- Union(x, y) –unites the sets that contain x and y, say, S_x and S_y, into a new set that is the union of the two sets.
  - Combining of trees is accomplished.

Kruskal’s Algorithm … contd

- Analysis
  - O(V) time required to initialize the V disjoint sets.
  - Sorting generally done using a comparison sort on average requires O(ElogE) time.
  - O(ElogE) time for checking the existence of cycles (not belonging to same tree)
  - Run time is O(ElogE) in average case.

Kruskal's Algorithm … contd

```
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,adj[9][9],parent[9];
 while(ne < n)
   {
        for(i=0,min=99;i<n;i++)
        {
        for(j=0;j <n;j++)
        {
        if(adj[i][j] < min)
      {
            min=adj[i][j];
            a=u=i;
            b=v=j;
        }
    }
    }
    u=find(u); v=find(v);
    if(uni(u,v))
    {
          printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
          mincost +=min;
    }
    adj[a][b]=adj[b][a]=999;
  }
```

```
int find(int i)
{
        while(parent[i])
        i=parent[i];
        return i;
}
int uni(int i,int j)
{
        if(i!=j)
        {
        parent[j]=i;
        return 1;
        }
        return 0;
}
```

Prim’s Algorithm

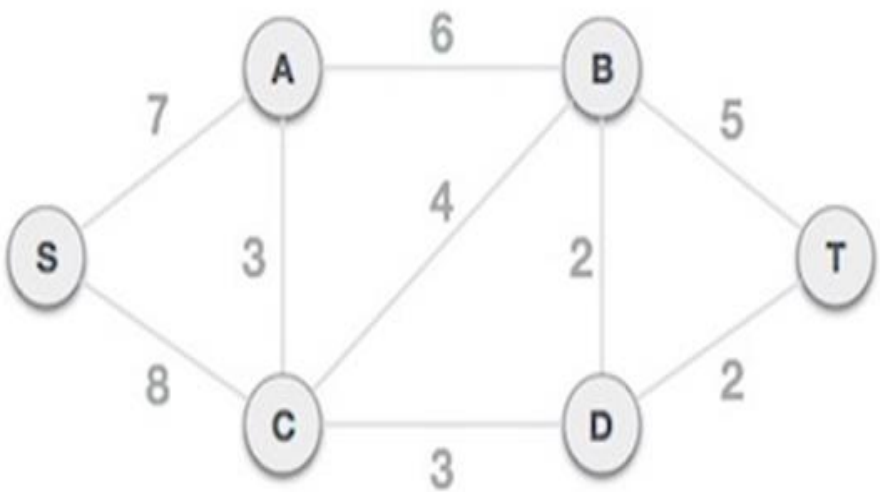
- MST grows “naturally” starting from a arbitrary root.
- Has the property that the edges in the set A always form a single tree.
- Logic
  - The tree starts from an arbitrary root vertex *r*.
  - Grow the tree until it spans all the vertices in set V.
- Data Structure
  - Q: a minimum priority queue keyed by edge values.

Prim's Algorithm … contd

- MST-PRIM(G, w, r)
  - **for** each $u \in V[G]$
    - **do** key[u] $\leftarrow \infty$
    - π[u] $\leftarrow$ NIL
  - key[r] $\leftarrow$ 0
  - $Q \leftarrow V[G]$
  - while Q $\neq \varnothing$
    - **do** u $\leftarrow$ Extract-Min (Q)
      - **for** each $v \in Adj[u]$
        - » **do if** $v \in Q$ and $w(u, v) < key[v]$

          then $\pi[v] \leftarrow u$

          key[v] $\leftarrow w(u, v)$



Prim's Algorithm … contd

- Analysis
  - Performance depends on how we implement priority queue *Q*.
  - If implemented as a heap, initialization takes O(V) time
  - Body of while loop executed *V* times. Extract-Min takes O(logV) time.
    - Total time is O(VlogV).
  - For loop executes in O(E) times
    - Test for membership within for loop executes constant time.
    - Assignment for key takes O(logV) time.
  - Average complexity is O(ElogV).

Applications of MST

- Network Design (Designing LANs, Laying Communication Lines, etc.)
- Approximation algorithms for NP-hard problems
- Cluster Analysis
- https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/
- https://www.javatpoint.com/applications-of-minimum-spanning-tree

Job Sequencing using deadlines

- We are given a list of n jobs. Every job i is associated with an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i, profit is earned if and only if the job is completed within its deadline.
- Only one machine to process the jobs for one unit of time.
- To find the optimal solution and feasibility of jobs we are required to find a subset J such that each job of this subset can be completed by its deadline.
- The value of a feasible solution J is the sum of profits of all the jobs in J, or $\sum_{i \in J} p_i$.

Job Sequencing … Algorithm

1. Sort p_i into nonincreasing order i.e. $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_i$.
2. Add the next job i to the solution set J if i can be completed by its deadline.
   - Assign i to time slot [r-1, r], where r is the largest integer such that $1 \leq r \leq d_i$ and [r-1, r] is free i.e. Schedule the job in its latest possible free slot if its available
3. Stop if all jobs are examined. Otherwise, go to step 2.
4. Complexity: $O(n^2)$

Job Sequencing using deadlines

- n=4, $(p_1,p_2,p_3,p_4)=(100,10,15,27)$, $(d_1,d_2,d_3,d_4)=(2,1,2,1)$
- n=5, $\{p_1,p_2,..,p_5\}= \{20,15,10,5,1\}$, $\{d_1, d_2, .., d_5\} = \{2,2,1,3,3\}$

i	1	2	3	4	5	6	7
p_i	5	4	8	7	6	9	3
d_i	3	2	1	3	2	1	2

Optimal Storage on Tapes

- There are 'n' programs that are to be stored on a computer tape of length 'l'.
- Associated with each program i is the length l_i, $1 \le i \le n$.
  - All the programs can only be written on the tape if the sum of all the lengths of the program is at most l.
- Assumption: that whenever a program is to be retrieved from the tape , the tape is positioned at the front.

Optimal Storage on Tapes

- If the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j

$$t_j = \sum_{k=1}^{j} l_{i_k}$$

- If all programs are retrieved equally often, then the mean retrieval time (MRT) $= \frac{1}{n}\sum_{j=1}^{n} t_j$

- Self Study

Optimal Storage on Tapes

- n=3 and (L1, L2, L3) = (5, 10, 3)

Ordering	MRT
L1,L2,L3	5+(5+10)+(5+10+3)/3=38/3
L1,L3,L2	5+(5+3)+(5+10+3)/3=31/3
L2,L1,L3	10+(5+10)+(5+10+3)/3=43/3
L2,L3,L1	10+(3+10)+(5+10+3)/3=41/3
L3,L1,L2	3+(5+3)+(5+10+3)/3=29/3
L3,L2,L1	3+(3+10)+(5+10+3)/3=34/3

Other Problems

- Huffman Coding

Next Topic

- Dynamic Programming