Greedy Algorithms

- Greedy algorithm obtains an optimal solution by making a sequence of decisions.
- Decisions are made one by one in some order
- Each decision is made using a greedy-choice property or greedy criteria
- A decision once made is (usually) not changed later

Characteristic & Features

To construct the solution in an optimal way, algo maintains 2 sets
1) One contains chosen item and
2) Other contains rejected items

Greedy algo make good local choices in the hope that they result in
1) optimal solution
2) feasible solution

Applications of Greedy

1) Make a change pb
2) Knapsack Pb
3) Minimum Spanning Tree
4) Single source shortest Path
5) Activity selection Pb
6) Job Sequencing Pb
7) Huffman Code Generation

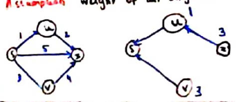
Where Greedy Approach Fails

- In many pbs, greedy algo fails to find an optimal solution
- Moreover, it may produce a worst solution
- Pb like Traveling Salesman & Knapsack cannot be solved using greedy approach

Single source Shortest Path Pb

Given a graph & a start vertex S
- determine distance of vertex from S
- identify shortest path to each vertex
  - express concisely as "shortest path tree"
  - each vertex has a pointer to a predecessor on shortest path

Assumption: weight of all edges is non negative



Greedy

- it optimizes by making the choice that is the best at the moment
- Chooses the locally optimal solution it will lead to globally optimal solution
- does not always find optimal solution but is very fast
- requires almost no memory
- makes decision considering the 1st stage
- for eg. Dijkstra's Algorithm

Control Abstraction

Control Abstraction for greedy approach

Algorithm GREEDY_APPROACH (L, n)

```
// Description: Solve the given pb using greedy
// I/P: L list of possible choices, n: size of pb
// O/P: Set solution containing solution of given pb
solution ← ∅
for i ← 1 to n do
    choice ← Select (L)
    if (feasible (Choice ∪ Solution)) the
        Solution ← Choice ∪ solution
    end
end
return Solution
```

Single Source Shortest Path Algorithm

Algorithm DIJKSTRA_SHORTEST_PATH (G, S, t)

```
// s : source vertex
// t : target vertex
// π(u) stores the parent/previous node of u
// V : set of Vertices in Graph G
dist[s] ← 0
π[s] ← NIL
for each vertex v ∈ V do
    if v ≠ s then
    dist[v] ← ∞
    π[v] ← underdefined
    end
    ENQUEUE(v, Q)
end
while Q is not empty do
  u ← vertex in Q having minimum dist[u]
  if u == t    then
    break
  end
  DEQUEUE (u, Q)
  for each adj.node v of u do
      val ← dist[u] + weight (u,v)
      if val < dist[v] then
         dist[v] ← val
         π [v] ← u
      end
    end
end
```

Note: Dijkstra's Algo cannot handle negative weight

Divide & Conquer

- divides a pb into simpler versions of itself
- applies solution for smaller sub-pb to the larger pb
- combines answer to sub pbs (recursive)
- always find the optimal solution, but slower than greedy
- require some memory to remember recursive calls
- for eg: Merge Sort

Knapsack Pb

- Given a set of items, having some weight and values / profit associated with it
- The Knapsack Pb is to find set of items such that the total weight ≤ given limit (size of knapsack)
- total value/Profit earned is as largest possible
- Knapsack Pb has 2 Variants
  - Binary or 0/1 Knapsack
    - items cannot be broken down into parts
  - Fractional Knapsack
    - items can be broken down into parts

Applications of Knapsack

1) Finding the least wasteful way to cut raw materials
2) Portfolio optimization
3) Cutting stock pb

Knapsack Pb are categorized as
- fractional Knapsack
- Knapsack

Algorithm GREEDY_FRACTIONAL_KNAPSACK (X, V, W, M)

```
// Description: Solve the Knapsack pb using greedy
// Input : X: Array of n items
           V: An array of profit associated with each item
           W: An array of weight associated with each item
           M: Capacity of Km
// output: SW: Weight of selected item
           SP: Profit of selected item
// Items are presorted in decreasing order of pi/wi ratio
S ← ∅     // Set of selected items, initially empty
SW ← 0   // weight of selected items
SP ← 0   // profit of selected items
i ← 1
while i ≤ n do
 if (SW + W[i]) < M then
    S ← S ∪ X[i]
    SW ← SW + W[i]
    SP ← SP + V[i]
else
  frac ← (M - SW) / W[i]
   S ← S ∪ X[i] * frac   // Add fraction of item X[i]
   SP ← SP + V[i] frac   // Add fraction of profit
   SW ← SW + W[i] * frac // Add fraction of weight
end
  i   i+1
end
```

Dynamic Programming

- breaks a pb down into sub pbs
- the sub-pbs are overlapping and recurring; dynamic pgrm will calculate them once and save their values
- sacrifices space to save time by remembering old sub pb values
- always finds optimal solution, but may be pointless on small data set
- requires a lot of memory for memorisation / tabulation.
- makes decisions at every stage
- for eg: Memorized Fibonacci series

Job Sequencing with Deadline

- n jobs to processed on a machine
- each job i has a deadline $d_i \geq 0$ & Profit $P_i > 0$
- profit is earned if only if job is completed by its deadline
- job is completed if it is processed on machine for a unit time
- only one machine is available for processing job
- only one job is processed at a time on machine
- a feasible solution is a subset of jobs J such that each job is completed by deadline
- optimal solution is a feasible solution with maximum profit value

Algorithm JOB_SCHEDULING (J, D, P)

```
// Description: Schedule the jobs using greedy approach which maximize the profit
// Input: J: Array of N jobs
          D: Array of deadline for each job
          P: Array of profit associated with each job
Sort all jobs in J in decreasing order of profit
S ← ∅ // S is set of scheduled jobs initial empty
~~S ← 0 // Sum is profit earned~~
for i ← 1 to N do
    if Job J[i] is feasible    then
    Schedule the job in latest possible free slot meeting its deadline
    S ← S ∪ J[i]
    SP ← SP + P[i]
    end
 end
```

Minimum Spanning Tree

- Graph
  Graph G = (V, E) is defined by set of vertices V & set of edges E joining those vertices
- Weighted Graph
  Graph G = (V, E, W) is called weighted graph if some weight or cost is associated with each of its edges
  W → set of weight associated with each edge

Tree
Tree T = (V', E') → subset of G = (V, E) where V' is subset of V & E' → subset of E
Tree doesn't contain a cycle, while graph or subgraph can have a cycle

Spanning Tree
Spanning Tree T = (V', E') is a tree of connected, undirected, weighted graph G = (V, E, W) which contains all vertices of G and some or all edge of G
So V' = V and E' ⊆ E

Minimum Spanning Tree
- Graph G can have many S.T with a ≠ cost
- MST → ST with min cost

Optimal Storage on Tapes

Algorithm for optimal storage (n, m)

```
{
K = 0; // Next tape to be stored
for i = 1 to n do
 {
  Write (i, K). // "Assign program"; i; "to tape"; K
  K = (k+1) mode m;
  }
}
```

Storage on multiple stage

- Pb of minimizing MRT on retrieval of prgm from multiple tapes
- Instead of single tape, prgm are to be stored on multiple tapes
- Greedy solves the pb, it sort the program according to increasing length of program and stores the prgm one by one in each tape.

Prim's Algo	Kruskal's Algo
- Greedy algo that finds MST for a weighted undirected graph	- a MST algo which finds an edge of the least possible way that connects any 2 trees in forest
- generates MST starting from root vertex	- generates MST starting from least weighted edge
- selects the root vertex	- select shortest edge
- select shortest edge connected to root vertex	- selects the next shortest edge
- sorting of edges not required	- sorting of edges compulsory
- better choice for dense graph	- better choice for sparse graph

Applications of Spanning Tree

- Network Design
- Implement efficient routing algorithm
- To solve Travelling Salesman problem
- Cluster Analysis