ANALYSIS OF ALGORITHMS – GREEDY METHODS

By Elias Debelo

JULY 5, 2022
HARAMAYA UNIVERSITY

CHAPTER THREE

THE GREEDY ALGORITHMS

3.1. THE GENERAL METHODS

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. A ***greedy algorithm*** always makes the choice that looks best at the moment.

That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make; we make the choice that looks best in the current problem, without considering results from subproblems. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.

A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within its optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

Given problem of n inputs and require us to obtain a subset that satisfies some constrains. Any subset that satisfies these constrains is called a ***feasible solution***. We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called ***an optimal solution***.

The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is *in an optimal solution*.

This is done by considering the inputs in an order determined by some *selection procedure*. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added.

The selection procedure itself is based on some *optimization measure*. The measure may be the objective function. In fact, several different optimization measures may be plausible for a given problem. Most of these, however, will result in algorithms that generate suboptimal solutions. This version of the greedy technique is called the subset paradigm.

The function **Select** selects an input from a[ ] and removes it. The selected input's value is assigned to x. **Feasible** is a Boolean-valued function that determines whether x can be included into the solution vector. The function **Union** combines x with the solution and updates the objective function. The function **Greedy** describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions **Select, Feasible,** and **Union** are properly implemented.

```
Algorithm Greedy (a,n)
//a[1 : n] contains the n inputs.
{
      Solution:=ø;// Initialize the solution.
      for i:=1 to n do
      {
            x:=Select(a);
            if Fesible(solution,x) then
                  solution := Union(solution,x);
            }

}
```

* Job Sequencing with deadlines

Given a set of n jobs, associated with job k is an integer deadline $d_k \geq 0$ and a profit $p_k > 0$. Constraint is for any job k the profit p_k earned iff (if and only if) the job is completed by its deadlines. To complete a job, one has to process the job on a machine for one unit of time. And only one machine is available for processing jobs.

- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadlines.
- The value of feasible solution J is the sum of profits of the jobs in J, or $\sum_{k \in J} P_k$
- An optimal solution is a feasible solution that maximize total profit.

- This optimization measure needs us to sort jobs in nonincreasing order. So, the objective is to find a sequence jobs, which is completed within their deadlines and gives maximum profit.

E.g., let us consider a set of given jobs as shown in the following table. We have to find sequence of jobs, which will be completed within their deadlines and will give maximum profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	3	1
Profit	60	100	20	40	20

Sorting jobs according to their profit in a decreasing order;

Job	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	3	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J2, as it can be completed within its deadline and contributes maximum profit.

Or

Job Considered	Slot Assigned	Solution Set (feasible solution)	Total Profit
_	_	ø	0
J2	[0, 1]	{J2}	100
J1	[0, 1] [1, 2]	{J2, J1}	160
J4	[0, 1] [1, 2] [2, 3]	{J2, J1}	200
~~J3~~	[0, 1] [1, 2] [2, 3]	{J2, J1, J4}	200
~~J5~~	[0, 1] [1, 2] [2, 3]	{J2, J1, J4}	200

Thus, the solution is the sequence of jobs (J2, J1, J4), which are being executed within their deadline and gives maximum profit. The total profit of this sequence is 100+60+40 = 200.

Ignoring the sorting step, this job sequencing run in $O(n^2)$ time in worst cases.

* Knapsack Problem (fractional knapsack)

In the knapsack problem, we have n objects and a knapsack or a bag. Object k has a weight w_k and the knapsack has a capacity m. If a fraction x_k, $0 \le x_k \le 1$, of object k is placed into the knapsack, then a profit of $p_k x_k$ is earned. (if $\mathbf{x_k}$ **of k** placed into knapsack, it will give us profit $p_k x_k$.)

- The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as:

  Maximize $\sum_{1 \le k \le n} P_k X_k$
  Which constrained to $\sum_{1 \le k \le n} w_k X_k \le m$ and $0 \le X_k \le 1$, $1 \le k \le n$

Now, any solution that satisfies both constrain is called feasible solution. Likewise, optimal solution is feasible solution that maximize the sum of profit.

```
Algorithm GreedyKS(m, n)
{
      for i:=1 to n do
            x[i] = 0.0;
      k:=1;
      U:=m;
      while k ≤ n do
            if w[k] > U then
                  break;
            x[k] = 1.0;
            U:=U-w[k];
      if k ≤ n then
            x[k]:=U/w[k];
}
```

E.g., where n = 4 and m = 10.

Object (k)	1	2	3	4
Profit	20	16	24	11
Weight	7	3	4	6

There are at least three different measures one can attempt to optimize when determining which object to include into the knapsack. Those are optimization measure which in turn help us to decide/identify our objective function; The total profit, capacity used, and ration of accumulated profit to capacity used.

The selection procedures (order of inputs), or greedy strategies are; taking largest profit first, taking smaller weights first, or the ration of profit to weight – which is called **balanced**. In this ratio inputs are order in nonincreasing order of their ration, or p1/w1 ≥ p2/w2 ≥...≥ pn/wn.

Now let us order given inputs into the above three selection procedures;
* Ordering according to nonincreasing profit:

Object (k)	3	1	2	4
Profit	24	20	16	11
Weight	4	7	3	6

Calculate profit and weight by inserting each input into knapsack, x1, x2, x3, x4 are fractions of object k = 1, 2, 3, and 4.

Following the **order**:

[x3, x1, x2, x4] = [1, 3/7, 1, 0]
$\sum_{1\le k\le 4} P_k X_k$ = 24 + 20*(3/7) + 16 + 0 = **48.57**
$\sum_{1\le k\le n} w_k X_k$ = 4 + 3 + 3 + 0 = **10**

* Order according to nondecreasing order of weights:

Object (k)	2	3	4	1
Profit	16	24	11	20
Weight	3	4	6	7

Applying the same step:

[x2, x3, x4, x1] = [1, 1, 1/2, 0]
$\sum_{1\le k\le 4} P_k X_k$ = 16 + 24 + 11*(1/2) + 0 = **45.5**
$\sum_{1\le k\le n} w_k X_k$ = 3 + 4 + 6*(1/2) + 0 = **10**

* Ordering according to nonincreasing order of ratio of profit to weight:

Object (k)	3	2	1	4
Profit	24	16	20	11
Weight	4	3	7	6

Applying the same step:

[x3, x2, x1, x4] = [1, 1, 3/7, 0]
$\sum_{1\le k\le 4} P_k X_k$ = 24 + 16 + 20*(3/7) + 0 = **48.57**
$\sum_{1\le k\le n} w_k X_k$ = 4 + 3 + 7*(3/7) + 0 = **10**

As you can see from the above; we get the same value for ratio and profit base orders; It could be the same, of course. But we didn't try all possible combination to maximize our gains from the fractions of every object. ***You can try it: try to incorporate fraction of last object in the list into knapsack. Unlike what I did***. For example, try x1 = 1/3 of k1.

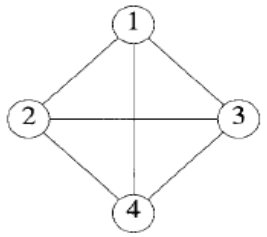
One thing in general, for knapsack problem you can get less or equal valued suboptimal (feasible) solutions: but the ration approach will guarantee you the equal or maximum profit. You can rely on it.

Discarding time for sorting, each of the three strategies require only O(n) time.

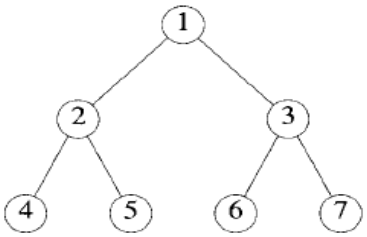
3.2. GRAPHS

* A graph G consists of two sets V and E, G = (V, E). The set V is a finite, nonempty set of *vertices*. The set E is a set of pairs of vertices; these pairs are called *edges*. The notations V(G) and E(G) represents the sets of vertices and edges respectively, of graph G.

* In an *undirected graph* the pair of vertices representing any edge is unordered. Thus, the pair (u, v) and (v, u) represent the edge. In a *directed graph* each edge is represented by a direct pair (u, v); u is tail and v is head of the edge. Therefore, (u, v) and (v, u) represent two different edges.



(a) G_1



(b) G_2

(c) G_3

The set representations of these graphs are

$$V(G_1) = \{1,2,3,4\} \qquad E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$
$$V(G_2) = \{1,2,3,4,5,6,7\} \quad E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$$
$$V(G_3) = \{1,2,3\} \qquad E(G_3) = \{\langle 1,2\rangle,\langle 2,1\rangle,\langle 2,3\rangle\}$$

If (u, v) is an edge in E(G), then we say vertices u and v are adjacent and edge (u, v) is *incident* on vertices u and v.

Work out: *Find out which vertices are adjacent and list incident edges of each vertex. E.g., vertices 1 and 2 are adjacent to each other in*

graph G1, and edges (1, 2) and (1, 3) are incident on vertex 1 in G1 and G2.

* A *graph* of G is a graph G' such that V(G') subset of V(G) and E(G') subset of E(G). A *Cycle* is a simple path in which the first and last vertices are the same.

3.3. MINIMUM SPANNING TREES

* Let G= (V, E) be an undirected connected graph. A graph t = (V, E') of G is a spanning tree of G, iff T is a tree. Spanning trees arises from the property that a spanning tree is a minimal subgraph G' of G such that V(G') = V(G) and G' is connected.

* A minimal subgraph is one with fewest numbers of edges. Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees.

In practical situations, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree. The cost of the spanning tree is the sum of all the edges in that tree.

E.g., Look at the following graph a, and its minimum spanning tree b.

The two known algorithms used to find minimum spanning tree of a graph are Prim's and Kruskal's algorithms.

Finding minimum spanning tree can be accomplished by greedy approach, because it builds the tree edge by edge. The inclusion of next edge is decided based on optimization measure. Simply the optimization measure could be "*choose an edge that results in a minimum increase in the sum of the costs of the edges so far included*".

KRUSKAL'S ALGORITHM

- Actualize the optimization criterion by ordering edges according to nondecreasing order of their costs.
- Let *cost[u, v]* be weight of edge (u, v).
- t be a set of edges in the minimum-cost spanning tree. And i be the number of edges in t.
- The set t can be represented as a sequential list using a two-dimensional array t[1:n-1, 1:2]. So, the edge (u, v) added to t as t[i,1]=u and t[i,2]=v.
- Let graph G, has n vertices.

```
Algorithm Kruskal (E, cost, n, t)
{
   Construct a heap out of the edge costs using Heapify;
   for i := n do
         parent [i] := -1;
         //now, each vertex is in a different set.
   i := 0 ; mincost := 0.0;
   while ( i < n – 1 ) and ( heap not empty )) do
   {
         Delete a minimum cost edge (u, v) from the min-heap and re-
         min-heapify;
         j := Find (u) ;
         k := Find (v) ;
         if (j ≠ k) then
         {
               i := i+1 ;
               t[ i, 1] := u ; t [i, 2 ] := v ;
               mincost := mincost + cost [u, v] ;
               Union (j, k);
         }
   }

   if (i ≠ n – 1) then
         Print("There is no spanning tree");
   else
         return mincost;
}
```

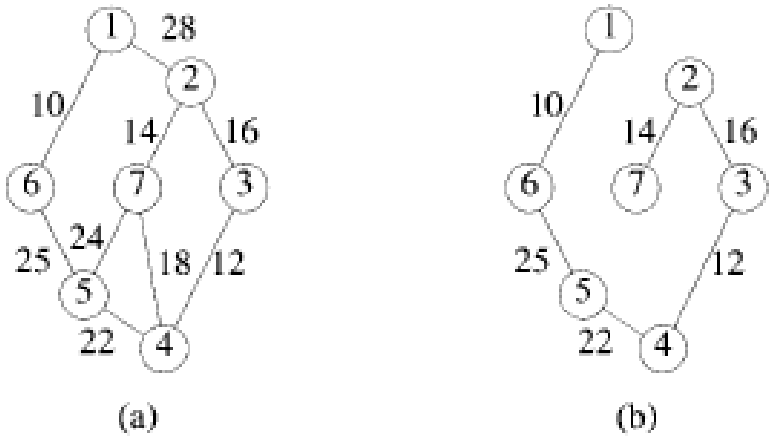
* j = find(u) returns a root node representing a tree that contain node u. Now, the parent of j, parent[j] is going to be -1. Because the parent of root is -1 in disjoint set.
* Union(j, k) – this makes k a parent of j. That means a tree represented by root j, is going to be connected to tree k though j.

Kruskal in simple terms:

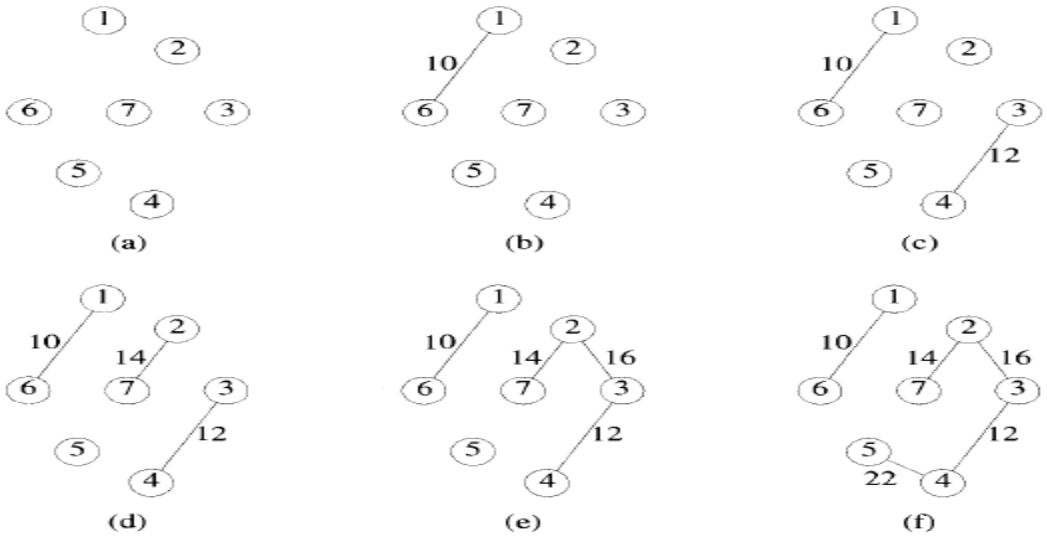
1. Plot all vertices of the graph.

2. Arrange the edges in nondecreasing order of their weight.
3. Choose every edge in such a way that it does not form a cycle on the graph.
4. Repeat till all the vertices are connected.

E.g., given the following graph at figure a, and the minimum spanning tree of the graph on figure b (left):



Steps in Kruskal algorithm to find the MST are given as follow;



Complexity Analysis

- Construction of min-heap (build-min-heap) takes |E|log|E| time.
- Making the forest graph or disconnected vertices takes |v| or n time.
- The while loop could go n times, that is equal to |V|, which in turn can be taken as |E| (at most). And both find takes 2log|E|, deleting min element of min-heap and min-heapify takes log|E| and union takes the same, log|E|. Then,

- |E|log|E| + |E|(log|E| + 2log|E| + log|E|) = 5(|E|log|E|)= O(|E|log|E| time.

3.4. Single-Source Shortest Path

Let **G = (V, E)** be a weighted, directed graph with weight function w:E->R, and assume that G contains nonnegative weight cycle reachable from the source vertex *s ∈ V*, so that the shortest paths are well defined.

- The shortest-path tree rooted at **S** is a directed subgraph G'=(V',E'), where V'⊆V and E'⊆E, such that
  1. V' is the set of vertices reachable from s in G.
  2. G' forms a rooted tree with root s, and
  3. For all v *∈ V',* the simple path from s to v in G' is a shortest path from s to v in G. But it is not so necessary for a shortest path be unique.

Relaxation

* For each vertex v *∈ V,* we maintain an attribute d[v], which is an upper bound on the weight of shortest path from source s to v. We call d[v] a *shortest path estimate*.

Algorithm Initialize(G, s)
{
    for each vertex v *∈ V[G] do*
        d[v]:= ∞
}
Then, d[v]= ∞ for all v *∈ V-{s}, m*eaning, except set containing s, all other vertices have d[v] of ∞.

Algorithm Relax(u, v, w)
{
    if d[v] > d[u] + w(u, v) then
        d[v]:=d[u] + w(u, v)
}
Meaning, replace shortest path estimate d[v] with less value, where w(u, v) represent weight of edge (u, v).

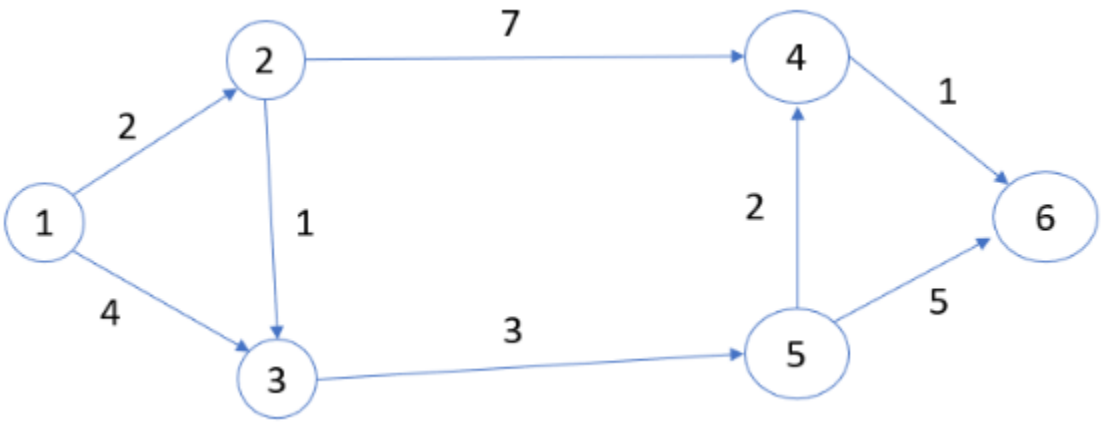
Now, we need an algorithm that put all things (initializing, sorting, extracting each vertex on min path, repeat the relation for every edges) together.

DIJKSTRA ALGORITHM (DA)

- DA solves the *single-source shortest path* problem on a weighted, directed graph G=(V, E) for the case in which all the edge weights are nonnegative, w(u, v)≥0 for edge (u, v) ϵ E.
- DA maintain a set S of vertices whose final shortest-path weights from source s have already been determined.
- The algorithm repeatedly selects the vertex u ϵ V-S with minimum shortest path estimate, adds u to S, and relaxes all edges leaving u.
- We use min-priority queue Q of vertices, *keyed* by their d values.

```
Algorithm Dijkstra (G, w, s)
{
        Initialize (G, s)
        S=∅
        Q=V[G]
        While Q ≠ ∅ do
          u=EXRACT-MIN(Q)
          S=S+{u}
          for each vertex v ϵ Adj[u] do
               Relax (u, v, w)
}
```

E.g.,



Complexity (simplified)

- |V| number of vertices considered each at a time.

- |E| number of edges relaxed for each v ϵ V, to the worst |V|. This may happen when the graph is *complete-each vertex connected to every other in the graph*.
- By definition, |V|=n.

Then, T(n) = |V|*|V| = n*n= O(n^2).