**COMPSCI 330: Design and Analysis of Algorithms** 9/18/2014

# Lecture #8

*Lecturer: Debmalya Panigrahi* *Scribe: Yilun Zhou*

## 1 Overview

This lecture introduces a new algorithm type, greedy algorithm. General design paradigm for greedy algorithm is introduced, pitfalls are discussed, and three examples of greedy algorithm are presented along with running time analysis and proof of correctness.

## 2 Introduction to Greedy Algorithm

Greedy algorithm is a group of algorithms that have one common characteristic, making the best choice locally at each step without considering future plans. Thus, the essence of greedy algorithm is a choice function: given a set of options, choose the current best option. Because of the myopic nature of greedy algorithm, it is (as expected) not correct for many problems. However, there are certain problems that can easily be solved using greedy algorithm, which can be proved to be correct. Examples of such problems include fractional knapsack and minimum spanning tree.

## 3 Knapsack Problem

Consider the situation when a burglar breaks into a house. After seeing all items in the house, he has a clear idea of what the value of each item is and what the volume of each item is. He wants to take everything but he only has a ”knapsack” of a certain size. Thus, he wants to take a subset of items of maximum total value but still fits his knapsack. In addition, all items are indivisible (i.e. he cannot take half of TV and get half of its value). What should he take?

**Definition 1.** *In a knapsack problem, there is a set $I$ containing $n$ items, labeled $1,2,...,n$. Each item is associated with a value $v_1,v_2,...,v_n$ and a weight $w_1,w_2,...,w_n$. In addition, there is a constraint $W$. The problem asks to find the subset $I' \subseteq I$ that maximizes the total value of items in it $\sum_{i\in I'} v_i$ subject to the constraint $\sum_{i\in I'} w_i \leq W$.*

The solution to this knapsack problem will be presented in a later lecture and this problem is a computational hard problem.

### 3.1 Fractional Knapsack Problem

Although the previous knapsack problem is not easy to solve, a variant of it, fractional knapsack problem, can be solved efficiently using greedy algorithm.
Now suppose instead the burglar breaks into a grocery store. All items in the grocery store are divisible. For example, he can take half a bag of flour and get half of its value. Still having the constraint of the total size of his knapsack, what should be his strategy to take items?

**Definition 2.** *In a fractional knapsack problem, there is a set $I$ containing $n$ items, labeled $1, 2, ..., n$. Each item is associated with a value $v_1, v_2, ..., v_n$ and a weight $w_1, w_2, ..., w_n$. In addition, there is a constraint $W$. The problem asks to find the the fraction of each item to take $a_1, a_2, ..., a_n \in \mathbb{R}$ that maximizes the total value of taken items $\sum_{i=1}^{n} a_i v_i$ subject to the constraint $\sum_{i=1}^{n} a_i w_i \leq W$.*

#### 3.1.1 Solution

The solution to fractional knapsack problem is relatively easy to come up with. Because you can take fraction of items, you are guaranteed to be able to fill the knapsack (note that this is not true in the previous knapsack problem: you may have some empty room left that is smaller than all remaining items). Thus, with a constant total weight, finding the maximum value becomes finding the maximum value per unit weight. Thus, you should first sort the items by value per unit weight and then take items in decreasing order (most valuable item first). Take the whole item when possible and take as much as you can for a given item when there is no space for the whole item.

#### 3.1.2 Relation to Original Knapsack Problem

As mentioned before, in the original knapsack problem you cannot take fractions so you may end up wasting some spaces. So in original knapsack problem, $a_1, a_2, ..., a_n = \{0, 1\}$. This difference adds (a huge amount of) complication and makes the difference of a problem that can be solved efficiently (in polynomial time) and a problem that cannot, as we will see later this semester.

## 4 Minimum Spanning Tree

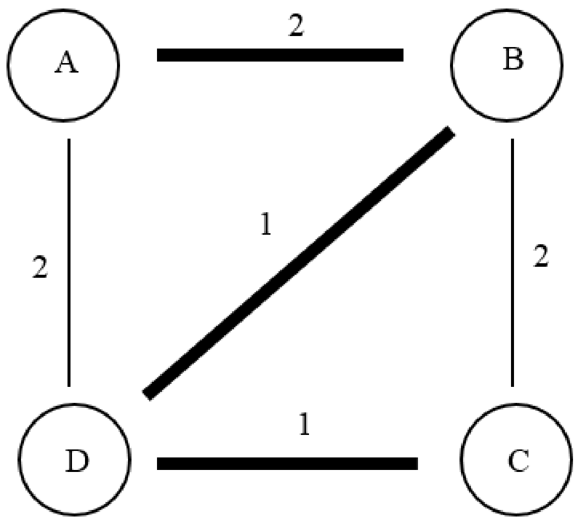
Another important application of greedy algorithm is minimum spanning tree (MST).

**Definition 3.** *Given an undirected weighted graph $G(V, E)$ with $l_{ij}$ denoting the weight of the edge between $i$ and $j$, a spanning tree is a subset $G'(V, E' \subseteq E)$ of $G$ such that $\forall v_i, v_j \in V$, there is a path from $v_i$ to $v_j$ and $G'$ is acyclic.*

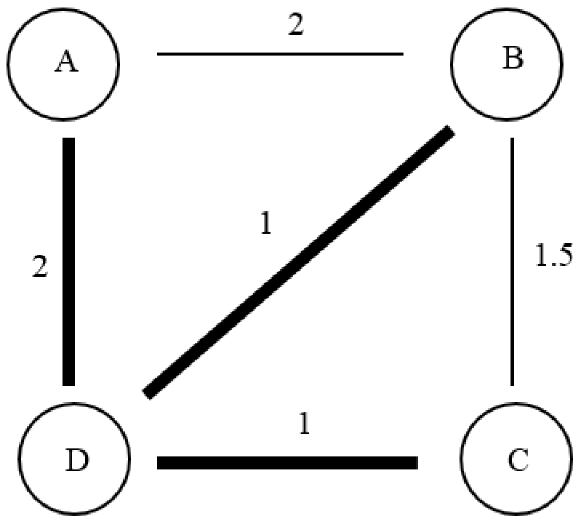
**Definition 4.** *A minimum spanning tree is a spanning tree with smallest total edge weight.*

### 4.1 Intuition for Finding MST

There are two properties of MST. First, an MST makes all vertices reachable from all other vertices. Second, an MST should have smaller (or at least the same) total edge length than other spanning trees. Thus, to find an MST, we need to consider two things at the same time: 1. to use as short edge as possible; 2. to connect as much of the graph as possible.
With these two objectives in mind, it is intuitive to start with the shortest edge. For example, for the following graph, we should start with one of the two edges with length 1. Then we have one edge of length 1 and three edges of length 2. So we should still try incorporating the edge of length 1 into our MST. Last, we must pick an edge of length 2. Picking edge $BC$ will not be useful as $B$ and $C$ are already connected by two edges of length 1. In addition, picking either $AB$ or $AD$ will complete the spanning tree and it is very easy to verify that these two spanning trees are indeed MST. One solution is shown below, with bold edges belonging to the MST.



For the following graph, we can also add the two edges of length 1. However, we cannot proceed to the next shortest edge *BC* with length 1.5. This edge is useless because *B* and *C* are already connected. Therefore, we need to skip it and add either *AB* or *AD*. Therefore, the general description of the algorithm is *to add the current shortest path that connect two connected components* to the MST at each iteration until the whole graph is connected.



## 4.2 Kruskal's Algorithm

It turns out that the algorithm described above is correct and is called Kruskal's algorithm. The detailed procedure is shown below.

```
1   sort edges in E to form {e_1, e_2, e_m}
2   assume e_i connects u_i and v_i
3   initialize E’ to empty set
4   initialize components set {{v}: v in V}
5
6   for i=1 to m, repeat until all vertices are in the same component
7           if u_i and v_i are not in the same component
8                   add e_i to E’
9                   merge components of u_i and v_i
```

In line 4 the components set is initialized to $\{\{v_1\},\{v_2\},...,\{v_n\}\}$. Then for example merging the *components* of $v_1$ and $v_2$ will result in $\{\{v_1,v_2\},\{v_3\},...,\{v_n\}\}$. Then merging the components of $v_3$ and $v_5$ will result in $\{\{v_1,v_2\},\{\{v_3,v_5\},\{v_4\},...,\{v_n\}\}$. Last, merging the components of $v_2$ and $v_5$ will result in $\{\{v_1,v_2,v_3,v_5\},\{v_4\},...,\{v_n\}\}$. Therefore, a merge operation actually influences all vertices that are in the same components with the two vertices to be merged.

### 4.2.1 Running Time

The running time of Kruskal's algorithm highly depends on the implementation of the merging two components and check for component belonging. Using a datastructure called Union-Find that will be introduced in later lectures can make these two operations both run in $O(\log n)$, where $n$ is the number of vertices in the Union-Find. Thus, the total running time is $O(m \log n)$ (because the original graph is connected, $m < n$. So the initial edge sorting time $O(m \log m)$ will be dominated by the Union-Find datastructure query time).

### 4.2.2 Proof of Correctness

There are two parts of the proof. First, we need to prove that Kruskal's algorithm does not produce any cycle. Second we need to prove that it indeed finds an MST (though not necessarily the only MST, as we have seen from previous examples that a graph can have multiple MSTs).

**Theorem 1.** *Kruskal's algorithm does not contain any cycles.*

*Proof.* Assume that there is a cycle of $v_1 - v_2 - v_3 - ... - v_n - v_1$. Then there must be an edge that is added last. Denote the edge $(v_i, v_j)$. However, immediately before the addition of this edge, $v_i$ and $v_j$ are connected by the remaining part of the circle. Thus, $(v_i, v_j)$ will not be added by Kruskal's algorithm. Thus the MST will not contain cycle. □

**Theorem 2.** *Kruskal's algorithm will find an MST.*

*Proof.* For this property we use induction.

**Definition 5.** *$E'$ is "good" at an intermediate stage if and only if there exists some minimum spanning tree $T$ such that $E' \subseteq T$.*

Clearly $E'$ is "good" at the beginning. Then we show that if $E'$ is true after $n$ iterations. Then at $n+1$ iteration, it will add edge $e$. If $e$ is in $T$, then $E'$ remains good after this iteration. If $e$ is not in $T$, Then $T+e$ will contain a cycle (because the two ends of $e$ are connected by $T$). Thus, there must be an edge $f$ in the cycle that is in $T$ but not in the MST that Kruskal's algorithm will produce. However, because Kruskal's algorithm does not choose $f$ and instead, it chooses $e$, then it means $e$ is as short as $f$. Moreover, the fact that $T$ is assumed to be an MST shows that $f$ is as short as $e$. Thus, these two edges must be of the same length. Therefore, $T-f+e$ is still an MST and $E'$ is still good after this iteration.
By induction, Kruskal's algorithm will produce an MST. □

### 4.3 Prim's Algorithm

Another algorithm for finding minimum spanning tree is Prim's algorithm. This algorithm is somewhat less intuitive, but closely resembles Dijkstra's algorithm. In fact, there is only one difference between these two algorithms.
The pseudocode for Prim's Algorithm is shown below:

```
1   c[s] = 0 // c[] is array of cost
2   for all v!=s, v in V:
3           c[v]=+oo
4   make minheap H=V with value c[v]
5   while H is not empty:
6           u = deletemin(H)
7                   for all (u,v) in E:
8
9                           if c[v]>l(u,v):
10                                  c[v] = l(u,v)
11                                  prev[v] = u
12                                  decreasekey(v)
```

The only difference form Dijkstra's algorithm is at line 9, the update rule, which will be explained below. Because the update rule in both algorithms is constant time operation, Prim's algorithm has exactly the same running time as Dijkstra's algorithm, $O(|E|\log|V|)$.

#### 4.3.1 Proof of Correctness

The proof of correctness lies in the update step. Prim's algorithm maintains a cluster of vertices that are already connected by an MST. Then it expands outside. There must be an edge from any point within the cluster to any point outside of it, or the MST is not connected. Therefore, at each step it finds the vertex that is closest to *any point* in the cluster and then connect these two points. That edge must be in some MST because there is no way we can do better.

## 5 Summary

In this lecture we talked about greedy algorithm. Greedy algorithm is a powerful tool to solve algorithmic problems. For greedy algorithm to work, the problem must have the property that locally optimal solution is also part of globally optimal solution. This can be sometimes hard to prove. However, if the problem does have this nice property, then greedy algorithm can be carried out very easily: iteratively find the locally optimal solution to assemble the globally optimal solution. Three representative algorithms are fractional knapsack, Kruskal's algorithm, and Prim's algorithm. Procedure and proof of correctness for these three algorithms are demonstrated and in the future lecture the data structure to efficiently implement Kruskal's algorithm will be introduced.