CS787: Advanced Algorithms	
Scribe: Shuchi Chawla	**Lecturer:** Shuchi Chawla
Topic: Introduction and Greedy Algorithms	**Date:** Sept 5, 2007

1.1 Introduction and Course Overview

In this course we will study techniques for designing and analyzing algorithms. Undergraduate algorithms courses typically cover techniques for designing exact, efficient (polynomial time) algorithms. The focus of this course is different. We will consider problems for which polynomial time exact algorithms are not known, problems under stringent resource constraints, as well as problems for which the notion of optimality is not well defined. In each case, our emphasis will be on designing efficient algorithms with provable guarantees on their performance. Some topics that we will cover are as follows:

- **Approximation algorithms for NP-hard problems**. NP-hard problems are those for which there are no polynomial time exact algorithms unless P = NP. Our focus will be on finding near-optimal solutions in polynomial time.
- **Online algorithms**. In these problems, the input to the problem is not known apriori, but arrives over time, in an "online" fashion. The goal is to design an algorithm that performs nearly as well as one that has the full information before-hand.
- **Learning algorithms**. These are special kinds of online algorithms that "learn" or determine a function based on "examples" of the function value at various inputs. The output of the algorithm is a concise representation of the function.
- **Streaming algorithms**. These algorithms solve problems on huge datasets under severe storage constraints—the extra space used for running the algorithm should be no more than a constant, or logarithmic in the length of the input. Such constraints arise, for example, in high-speed networking environments.

We begin with a quick revision of basic algorithmic techniques including greedy algorithms, divide & conquer, dynamic programming, network flow and basic randomized algorithms. Students are expected to have seen this material before in a basic algorithms course.

Note that some times we will not explicitly analyze the running times of the algorithms we discuss. However, this is an important part of algorithm analysis, and readers are highly encouraged to work out the asymptotic running times themselves.

1.2 Greedy Algorithms

As the name suggests, greedy algorithms solve problems by making a series of myopic decisions, each of which by itself solves some subproblem optimally, but that altogether may or may not be

optimal for the problem as a whole. As a result these algorithms are usually very easy to design but may be tricky to analyze, and don't always lead to the optimal solution. Nevertheless there are a few broad arguments that can be utilized to argue their correctness. We will demonstrate two such techniques through a few examples.

1.2.1 Interval Scheduling

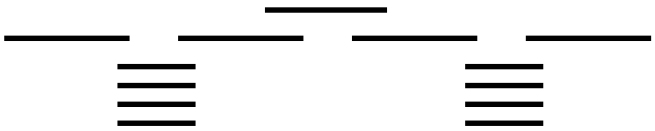
Given: n jobs, each with a start and finish time (s_i, f_i).

Goal: Schedule the maximum number of (non-overlapping) jobs on a single machine.

To apply the greedy approach to this problem, we will schedule jobs successively, while ensuring that no picked job overlaps with those previously scheduled. The key design element is to decide the order in which we consider jobs. There are several ways of doing so. Suppose for example, that we pick jobs in increasing order of size. It is easy to see that this does not necessarily lead to the optimal solution (see the figure below for a counter-example). Likewise, scheduling jobs in order of their arrivals (start times), or in increasing order of the number of conflicts that they have, also does not work.

(a) Bad example for the shortest job first algorithm

(b) Bad example for the earliest start first algorithm



(c) Bad example for the fewest conflicts first algorithm

We will now show that picking jobs in increasing order of finish times gives the optimal solution. At a high level, our proof will employ induction to show that at any point of time the greedy solution is no worse than any *partial* optimal solution up to that point of time. In short, we will show that *greedy always stays ahead.*

Theorem 1.2.1 *The "earliest finish time first" algorithm described above generates an optimal schedule for the interval scheduling problem.*

Proof: Consider any solution S with at least k jobs. We claim by induction on k that the greedy algorithm schedules at least k jobs and that the first k jobs in the greedy schedule finish no later

than the first k jobs in the chosen solution. This immediately implies the result because it shows that the greedy algorithm schedules at least as many jobs as the optimal solution.

We now prove the claim. The base case, $k = 0$ is trivial. For the inductive step, consider the $(k+1)$th job, say J, in the solution S. Then this job begins after the kth job in S ends, which happens after the kth job in the greedy schedule ends (by the induction hypothesis). Therefore, it is possible to augment the greedy schedule with the job J without introducing any conflicts. The greedy algorithm finds a candidate to augment its solution and in particular, picks one that finishes no later than the time at which J ends. This completes the proof of the claim. ■

1.2.2 Minimum Spanning Tree

Our second problem is a network design problem.

Given: A graph G with n vertices or machines, and m edges or potential cables. Each edge has a specified length—ℓ_e for edge e.

Goal: Form a network connecting all the machines using the minimum amount of cable.

Our goal is to select a subset of the edges of minimum total length such that all the vertices are connected. It is immediate that the resulting set of edges forms a spanning tree—every vertex must be included; Cycles do not improve connectivity and only increase the total length. Therefore, the problem is to find a spanning tree of minimum total length.

The greedy approach to this problem involves picking edges one at a time while avoiding forming cycles. Again, the order in which we consider edges to be added to the tree forms a crucial component of the algorithm. We mention two variants of this approach below, both of which lead to optimal tree constructions:

1. **Kruskal's algorithm**: Consider edges in increasing order of length, and pick each edge that does not form a cycle with previously included edges.
2. **Prim's algorithm**: Start with an arbitrary node and call it the root component; at every step, grow the root component by adding to it the shortest edge that has exactly one end-point in the component.

A third greedy algorithm, called **reverse-delete**, also produces an optimal spanning tree. This algorithm starts with the entire set of edges and deletes edges one at a time in decreasing order of length unless the deletion disconnects the graph.

We will now analyze Kruskal's algorithm and show that it produces an optimal solution. The other two algorithms can be analyzed in a similar manner. (Readers are encouraged to work out the details themselves.) This time we use a different proof technique—an *exchange argument*. We will show that we can transform any optimal solution into the greedy solution via local "exchange" steps, without increasing the cost (length) of the solution. This will then imply that the cost of the greedy solution is no more than that of an optimal solution.

Theorem 1.2.2 *Kruskal's algorithm finds the minimum spanning tree of a given graph.*

Proof: Consider any optimal solution, T^*, to the problem. As described above, we will transform this solution into the greedy solution T produced by Kruskal's algorithm, without increasing its length. Consider the first edge in increasing order of length, say e, that is in one of the trees T and T^* but not in the other. Then $e \in T \setminus T^*$ (convince yourself that the other case, $e \in T^* \setminus T$, is not possible). Now consider adding e to the tree T^*, forming a unique cycle C. Naturally T does not contain C, so consider the most expensive edge $e' \in C$ that is not in T. It is immediate that $\ell_{e'} \leq \ell_e$, by our choice of e, and because e' belongs to one of the trees and not the other. Let T_1^* be the tree T^* minus the edge e' plus the edge e. Then T_1^* has total length no more than T^*, and is closer (in hamming distance[1]) to T than T^* is. Continuing in this manner, we can obtain a sequence of trees that are increasingly closer to T in hamming distance, and no worse than T^* in terms of length; the last tree on this sequence is T itself. ■

1.2.3 Set Cover

As we mentioned earlier, greedy algorithms don't always lead to globally optimal solutions. In the following lecture, we will discuss one such example, namely the set cover problem. Following the techniques introduced above we will show that it nevertheless produces a near-optimal solution. The set cover problem is defined as follows:

Given: A universe U of n elements. A collection of subsets $S_1, \cdots, S_k$ of U.

Goal: Find the smallest collection $\mathcal{C}$ of subsets that covers U, that is, $\cup_{S \in \mathcal{C}} S = U$.

[1]We define the hamming distance between two trees to be the number of edges that are contained in one of the trees and not the other.