CS787: Advanced Algorithms	**Lecture 1:** Introduction and Greedy Algorithms

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 *a priori*, 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 beforehand.
- **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.
- **Algorithms in game theory**. For some algorithmic problems, the input is controlled by a number of different self-interested participants (e.g. voters in an election) whose goals may not align with that of the algorithm designer; the participants may misreport their inputs to maximize their chances of success. The goal is to design an algorithm that encourages the participants to be truthful and at the same time achieves (near-)optimality.

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. In addition, we will introduce and use a variety of advanced techniques for these problems, including linear programming, semi-definite programming, and randomization.

Note that some times we will not explicitly analyze the running times of the algorithms we discuss beyond ensuring that it can be bounded from above by a polynomial in the input size. 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. The key to designing a good greedy algorithm is to find an appropriate way of breaking up the problem into several smaller pieces that can then be put together. 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 (Technique: optimality by staying ahead)

Given: n jobs and one machine, job i has a start time s_i and a finish time $f_i \geq s_i$.

Goal: Find the largest subset of jobs that can be scheduled on the machine in a non-overlapping manner (that is, for any two scheduled jobs i and j, either $f_i \leq s_j$ or $f_j \leq s_i$).

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 potential ways of doing so, each attempting to minimize the number of potential overlaps each successive job may cause:

- Shortest job first
- Earliest arrival first
- Fewest conflicts first
- Earliest finish time first

The first three approaches don't necessarily lead to the optimal solution (see the figure below for counter-examples). The last one always leads to an optimal solution, and this is what we show next.

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.*

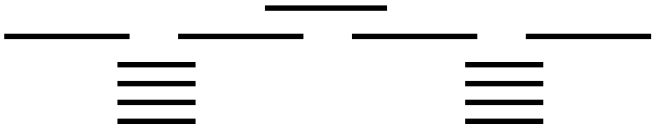
Theorem 1.2.1 *The "earliest finish time first" algorithm described above generates an optimal schedule for the interval scheduling problem.*

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. Let S_k be the kth job in S and G_k the kth job in greedy. The base case, $k = 0$ is trivial. For the inductive step, suppose that the inductive hypothesis holds for $k-1$. Then,

(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

if s_{k-1} exists, G_{k-1} also exists and has an earlier finish time than S_{k-1}. Now consider the kth job in S, S_k. Clearly, $s_{S_k} \geq f_{S_{k-1}} \geq f_{G_{k-1}}$. That is, S_k starts after G_{k-1} has finished. Also, at the time that G_{k-1} is scheduled in greedy, S_k has not been considered yet. So the greedy algorithm can augment its schedule by adding job S_k. It therefore finds a candidate to augment its solution and in particular, picks one that finishes no later than S_k. This completes the proof of the claim. ■

1.2.2 Dijkstra's algorithm (Technique: optimality by staying ahead)

We now present another example of the "staying ahead" approach. This is a classic algorithm by Edsgar Dijkstra for the shortest paths problem.

Given: A graph with lengths ℓ_e on edges; special nodes s and t.

Goal: Find the shortest path between s and t.

Dijkstra's algorithm greedily explores paths starting from s by moving each time to the next closest node. In this manner, it in fact constructs shortest paths from s to every other node in the graph. The formal description of the algorithm is as follows:

- Initialize $Z = \{s\}$. Set $\text{Path}(s) = \emptyset$ and $d(s) = 0$.
- For every node v not in Z, consider the distance $d'(v) = \min_{u \in Z}\{d(u) + \ell_{(u,v)}\}$. Let v be the node that minimizes this distance, and $e = (u, v)$ be the corresponding edge.
- Add v to Z. Set $\text{Path}(v) = \text{Path}(u) \cup \{e\}$, and $d(v) = d(u) + \ell_e$.

Once again we show that greedy stays ahead, that is, it constructs optimal partial solutions. In particular, at any point of time during the algorithm, the paths constructed for nodes in Z are

shortest paths and the distances $d(u)$ are the lengths of these paths.

Theorem 1.2.2 *Dijkstra's algorithm finds the shortest path from s to every other node in the graph.*

Proof: We prove the theorem by induction on the size of Z. The base case of $|Z| = 1$ is trivial: in this case z only contains s and $\text{Path}(s) = \emptyset$. Suppose that the inductive hypothesis holds until step $k-1$ and consider the kth node, v, added to Z. Let $e = (u, v)$ be the corresponding edge added. Suppose, for contradiction, that $\text{Path}(v)$ is not the shortest path from s to v, and let $\text{Path}^*(v)$ be a shortest path. Let x be the last node before v on $\text{Path}^*(v)$ that is in the set Z, and x' be the following node on $\text{Path}^*(v)$ (note that x' may be v itself). Then, by definition, $d(x) + \ell_{(x,x')}$ is no more than the length of $\text{Path}^*(v)$. However, by construction, $d(u) + \ell_{(u,v)} \leq d(x) + \ell_{(x,x')}$, and $d(v) = d(u) + \ell_{(u,v)}$. This implies that $d(v)$ is no more than the length of the shortest path $\text{Path}^*(v)$ and we are done. ■

1.2.3 Minimum Spanning Tree (Technique: exchange argument)

Our third problem is a network design problem.

Given: A graph G with n vertices or machines, and m edges or potential connections. 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.

A number of greedy approaches work for this problem. One can either start with the empty graph and successively add edges while avoiding forming cycles, or start with the complete graph and successively remove edges while maintaining connectivity. The crucial aspect is the order in which edges are considered for addition or deletion. We desrcibe three greedy approaches below, all 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.
3. **Reverse delete**: Start with the entire graph, and consider edges for deletion in order of decreasing lengths. Remove an edge as long as the deletion does not disconnect 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 1.2.3 *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. We first note that e cannot be in $T^* \setminus T$. In particular, if $e \notin T$, adding e to T must form a cycle in T. But all the edges of this cycle must also be present in T^*, by the definition of e, which contradicts the fact that T^* is a tree.

Therefore $e \in T \setminus T^*$. 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 $\ell_{e'} \geq \ell_e$, 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]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.