15-854: Approximations Algorithms **Lecturer:** Anupam Gupta
Topic: Dynamic Programming **Date:** October 12, 2005
Scribe: Mihir Kedia

Note: Starred sections were not specifically covered in lecture - they serve to provide additional background and information.

10.1 Introduction

Only a brief explanation of dynamic programming is provided - see an undergraduate algorithms textbook for a more complete treatment. Dynamic programming is a concept used to efficiently solve optimization problems by caching subproblem solutions. More specifically, we can use this technique to solve problems that display the *optimal substructure* property - if a solution to a given problem includes the solution to a subproblem, then it includes the optimal solution to that problem.

There are a lot of NP-complete problems that are not vulnerable to dynamic programming. Our approach to writing approximation algorithms will be to construct relaxations of these problems problems that we can solve using dynamic programming.

10.2 Knapsack

Problem 10.2.1 (Knapsack) *As input, Knapsack takes a set of n items, each with profit p_i and size s_i, and a knapsack with size bound B (for simplicity we assume that all elements have $s_i < B$). Find a subset of items $I \subset [n]$ that maximizes $\sum_{i \in I} p_i$ subject to the constraint $\sum_{i \in I} s_i \leq B$.*

Knapsack is NP-hard through a reduction from the partition problem (see 10.5).

Using dynamic programming, we can get an exact solution for knapsack in time $O(poly(n, P_{max}) \times \log(nB))$. Unfortunately, this is not polynomial in the size of its representation - P_{max} is actually $\log P_{max}$ in the problem representation. Thus, we go back to our toolbox of approximation algorithms.

10.2.1 A 2-approximation algorithm

As a warmup, let's try a basic greedy algorithm:

Greedy Algorithm?

1. Sort items in non-increasing order of $\frac{P_i}{S_i}$.
2. Greedily pick items in above order.

Intuitively, we want the items with the most "bang for the buck". Unfortunately, this algorithm is arbitrarily bad. Consider the following input as a counterexample:

- An item with size 1 and profit 2
- An item with size B and profit B.

Our greedy algorithm will only pick the small item, making this a pretty bad approximation algorithm. Therefore, we make the following small adjustment to our greedy algorithm:

Greedy Algorithm Redux

1. Sort items in non-increasing order of $\frac{P_i}{S_i}$.
2. Greedily add items until we hit an item a_i that is too big. $\left(\sum_{k=1}^{i} s_i > B\right)$
3. Pick the better of $\{a_1, a_2, ..., a_{i-1}\}$ and $\{a_i\}$.

At first glance, this seems to be a clumsy hack in order to deal with the above counterexample. However, it turns out that this algorithm has far better performance.

Theorem 10.2.2 *Greedy Algorithm Redux is a 2-approximation for the knapsack problem.*

Proof: We employed a greedy algorithm. Therefore we can say that if our solution is suboptimal, we must have some leftover space $B - S$ at the end. Imagine for a second that our algorithm was able to take a *fraction* of an item. Then, by adding $\frac{B-S}{s_{k+1}} p_{k+1}$ to our knapsack value, we would either match or exceed OPT (remember that OPT is unable to take fractional items). Therefore, either $\sum_{k=1}^{i-1} p_i \geq \frac{1}{2} OPT$ or $p_{k+1} \geq \frac{B-S}{s_{k+1}} p_{k+1} \geq \frac{1}{2} OPT$. ■