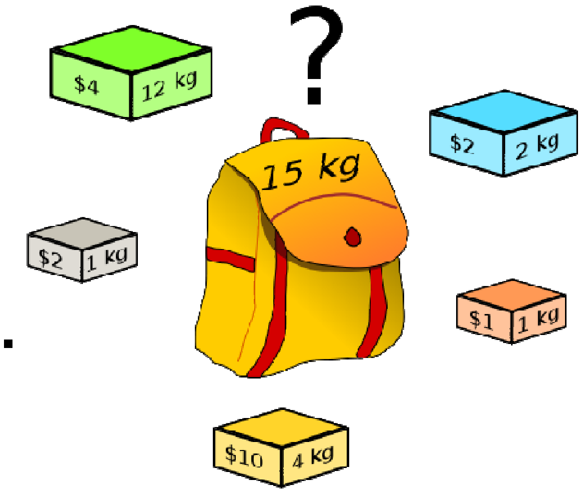
Approximation algorithms 3
Scheduling, Knapsack

CS240 Spring 2022

Rui Fan

The knapsack problem

- We have a set of items, each having a weight and a value.
- We have a knapsack that can carry up to W amount of weight.
- We want to put items in the knapsack to maximize the total value, but not exceed the weight limit.
- Ex Items 3 and 4 are the highest value items with weight ≤ 11.
- Assume all items have weight ≤ W, i.e. any single item fits in knapsack.



W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

A dynamic program for knapsack

- Let OPT(i,v) = minimum weight of a subset of items 1,...,i that has value ≥ v.
- If optimal solution uses item i.
  - Then we pay w_i weight for item i, and need to achieve value ≥ $v-v_i$ using items 1,...,i-1 using min weight.
  - So $OPT(i,v)=w_i+OPT(i-1,v-v_i)$.
- If optimal solution doesn't use item i.
  - Then we need to achieve value ≥ v using items 1,...,i-1.
  - So $OPT(i,v)=OPT(i-1,v)$.
- Choose the case that gives smaller weight.
- $OPT(i,v) = \begin{cases} 0 & \text{if } v=0 \\ \infty & \text{if } i=0,\ v>0 \\ \min(OPT(i-1,v),\ w_i+OPT(i-1,v-v_i)) & \text{otherwise} \end{cases}$

Running time of dynamic program

- Say there are n items, and the largest value of any item is v^*.
- The max value we can pack into the knapsack is nv^*, where v^* is the largest v value.
- Solve all subproblems of the form OPT(i,v), where $i \leq n$ and $v \leq nv^*$.
  - This is a total of $O(n^2v^*)$ subproblems.
- The solution to Knapsack is the max value V that can be packed with weight $\leq$ W.
- Having solved all the subproblems, we can find V by finding the subproblem with the largest value that has optimum weight $\leq$ W.
  - $V = \max_{v \leq nv^*} OPT(n,v) \leq W$.
- So solving Knapsack takes total time $O(n^2v^*)$.

Running time of dynamic program

- The DP gives an optimal solution to Knapsack and takes $O(n^2v^*)$ time. Have we found a polytime algorithm for an NP-complete problem?
- No. The problem size is $O(n \log(v^*))$, because it takes $\log(v^*)$ bits to express each item's value. But $O(n^2v^*)$ is not polynomial in $n \log(v^*)$.
- To make this DP fast, we have to make the largest value small.

PTAS

- Let $\varepsilon>0$ be any number. We'll give a $(1+\varepsilon)$-approximation for knapsack.
- By setting ε sufficiently small, we can get as good an approximation as we want!
  - This type of algorithm is called a polynomial time approximation scheme, or PTAS.
- Contrast this with earlier algs we studied, which had worse approx ratios, e.g. 2 or log n.
- But the running time will be $O(n^3/\varepsilon)$. Hence we can't set $\varepsilon=0$ get the optimal solution.
- We're trading accuracy for time. The more accurate (smaller ε), the more time the algorithm takes.

Main idea: rounding

- Since we only need an approximate solution, we can change the values of the items a little (round the values) and not affect the solution much.
- We scale and round the original values to make them small.
- The previous DP took $O(n^2 v^*)$ time. So if the rounded values are small, this DP is fast.

W = 11

Item	Value	Weight
1	134,221	1
2	656,342	2
3	1,810,013	5
4	22,217,800	6
5	28,343,199	7

W = 11

Item	Value	Weight
1	2	1
2	7	2
3	19	5
4	223	6
5	284	7

Rounding

- Let $\varepsilon>0$ be the precision we want.
- Set $\theta = \varepsilon v^*/2n$ to be a scaling factor.
  - v^* is the largest value of any item.
- Scale all values down by θ then round up.
  - $v' = \lceil v/\theta \rceil$.
- Make a problem where each value v_i is replaced by v'_i.
  - Call this the scaled rounded problem.
- Let v^ be max value in the scaled rounded problem. Then $\hat{v} = \lceil v^*/\theta \rceil = \lceil v^*/(\varepsilon v^*/2n) \rceil = \lceil 2n/\varepsilon \rceil$.
- Running time of DP on scaled rounded problem is $O(n^2 \hat{v}) = O(n^3/\varepsilon)$.

Solving the original problem

- Make another new problem in which each value v_i is replaced by $u_i = \lceil v_i/\theta \rceil * \theta$.
  - Call this the rounded problem.
  - We have $u_i \geq v_i$, and $u_i \leq v_i + \theta$.
- Note u values are equal to v' values multiplied by θ.
  - Thus, the optimal solution for the rounded problem and the scaled rounded problem are the same.
- We now have 3 problems, the original problem, the scaled rounded problem, and the rounded problem.
- Let S be the optimal solution to the scaled rounded problem, which we can find in time $O(n^3/\varepsilon)$. S is also optimal for the rounded problem.
- We'll show S is a $1+\varepsilon$ approximation for the original problem.

Correctness

- Thm Let S* be the optimal solution to the original problem. Then $(1+\varepsilon)\sum_{i\in S} v_i \geq \sum_{i\in S^*} v_i$. Hence S is a (1+ε)-approximate solution.

- Proof

$$\sum_{i\in S^*} v_i \leq \sum_{i\in S^*} u_i$$ $u_i \geq v_i$

$$\leq \sum_{i\in S} u_i$$ S is opt soln for rounded problem

$$\leq \sum_{i\in S} (v_i + \theta)$$ $u_i \leq v_i+\theta$

$$\leq \sum_{i\in S} v_i + n\theta$$ $|S| \leq n$

Correctness

- Suppose item j has the largest value, so $v^*=v_j$. Then $n\theta = \frac{\varepsilon}{2} v_j \le \frac{\varepsilon}{2} u_j \le \frac{\varepsilon}{2} \sum_{i \in S} u_i$
  - Last inequality because item j itself is feasible solution, so opt solution S is no smaller.
- So $\sum_{i \in S} v_i \ge \sum_{i \in S} u_i - n\theta \ge \left(\frac{2}{\varepsilon} - 1\right) n\theta$, where first inequality comes inequalities on last page.
- Assuming $\varepsilon \le 1$, then $n\theta \le \varepsilon \sum_{i \in S} v_i$
- Finally, we have

$$\sum_{i \in S^*} v_i \le \sum_{i \in S} v_i + n\theta \le \sum_{i \in S} v_i + \varepsilon \sum_{i \in S} v_i = (1+\varepsilon) \sum_{i \in S} v_i$$

Summary

- We gave a DP for Knapsack.
- We scale and round to reduce number of different item values.
- Running the DP on the scaled rounded problem and using the solution for the original problem leads to an arbitrarily good approximation for Knapsack, a PTAS.
- There are PTAS's for a number of other problems.
  - Multiprocessor scheduling.
  - Bin packing.
  - Euclidean TSP.
- However, there are also many problems for which PTAS's do not exist, unless P=NP.

Lower Bounds

Upper and lower bounds

- What is the minimum resources (time, space, etc.) needed to solve a problem?
- Consider sorting n numbers.
  - Insertion sort takes $O(n^2)$ time.
  - This puts an upper bound of $O(n^2)$ on the time to sort n numbers.
  - Merge sort takes $O(n \log n)$ time.
  - This puts an upper bound of $O(n \log n)$ on the time to sort n numbers.
- We want to make the upper bound as low as possible, i.e. solve the problem faster.
- Suppose an algorithm A solves problem X in $f(n)$ time when input size is n.
  - Then $f(n)$ is an upper bound on the complexity of X.

Upper and lower bounds

- What about the least amount of time to solve X?
- Suppose we know that any algorithm that solves X takes at least g(n) time, when X has size n.
  - Then g(n) is a lower bound on the complexity of X.
- If the lower bound g(n) is large, it means problem X is hard to solve.
  - Ex NP-Hard problems are hard because they (probably) have super-polynomial lower bounds.
- To show a lower bound, we need to give a proof.
  - Usually we show if an algorithm takes too little time, it must sometimes produce the wrong answer.
- The lower bound for a problem depends on the computational model.
  - If a model has very powerful primitive operations, then algorithms can run faster, and the lower bound is smaller.
- If the complexity of an algorithm for problem X matches the lower bound for problem X, the algorithm is optimal, and the lower bound is tight.

Sorting

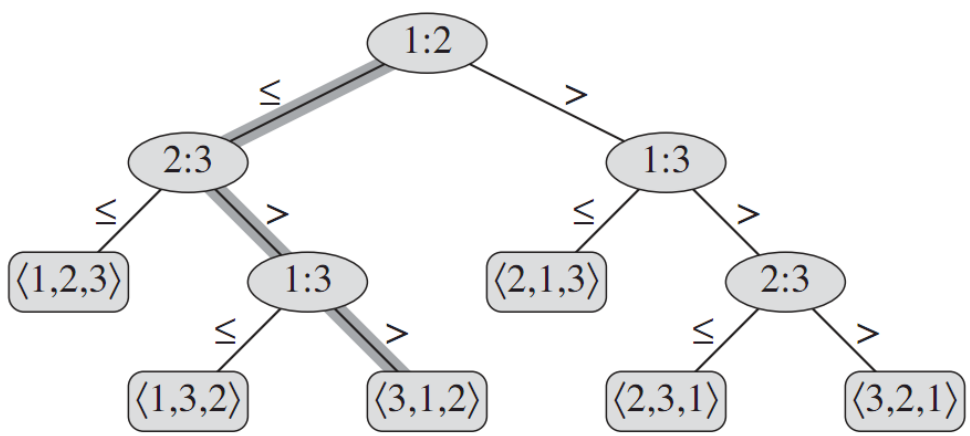
- How many comparisons are needed to sort n numbers?
- Upper bound: O(n log n) using merge sort.
- Lower bound: $\Omega(n \log n)$.
- To prove the lower bound, we first need a model for how a comparison-based sorting algorithm works.
  - This is called the decision tree model.
- The lower bound is not valid in other models.
  - If an algorithm can do things besides comparing two numbers, e.g. look at the digits of a number, it can sort faster than $\Omega(n \log n)$ time.
  - Lower bounds can be very sensitive to the computational model.

Decision trees

- In this model, in each step, algorithm can only compare a pair of numbers x, y.
- Based on result of the comparison, it decides next pair of numbers to compare.
  - So an execution of the algorithm is a sequence of comparisons, each comparison determined by result of previous comparison.
- When the algorithm terminates, it outputs a permutation representing the sorted order of the input.
- The complexity of the algorithm is the most number of comparisons it does before terminating.

Decision trees

- Model behavior of the algorithm by a binary tree.
  - Each internal node is a pair of number x,y to compare.
  - If x≤y, go to left child. If x>y, go to right child.
  - Each leaf represents an output, and is labeled with a permutation representing the sorted order of the inputs.
- An execution is simply a path from root to a leaf.
  - At any node, the algorithm has obtained some info from the comparisons it's done.
  - It uses this info to decide the next comparison to do.
  - Eventually, it obtains enough info to generate an output.
- Complexity of algorithm is the length of the longest root-leaf path.



Lower bound for sorting

- Given n numbers as input, they can be in n! different orders.
- Given an input order, algorithm must output that order.
  - So decision tree of algorithm must have a leaf labeled with that order.
  - So the decision tree has ≥ n! leafs.
- Say height of decision tree is h.
  - The complexity of the algorithm is h.
  - Since decision tree is binary, it has $\leq 2^h$ leaves.
- So $2^h \geq$ (# leaves of dec tree) ≥ n!, and so $h \geq \log_2(n!)$.
  - $\log_2(n!) = \log_2 n + \log_2(n-1) + \cdots + \log_2 1 \geq \log_2 n + \log_2(n-1) + \cdots + \log_2(n/2) \geq \frac{n}{2}(\log_2 n - 1) = \Omega(n \log n)$.
  - Can also use Stirling's approximation.
- So we proved the algorithm does $\Omega(n \log n)$ comparisons.