Week 13 - Wednesday

COMP 4500

Last time

- What did we talk about last time?
- Finished load balancing approximation
- Set cover approximation

Questions?

Assignment 7

Logical warmup

- U2 has 17 minutes to cross a bridge for a concert
- Plan a way to get them across in the darkness
- They have one flashlight
- A maximum of two people can cross the bridge at one time, and one of them must have the flashlight
- The flashlight must be walked back and forth
- Each band member walks at a different speed
  - **Bono:** 1 minute to cross
  - **The Edge:** 2 minutes to cross
  - **Adam:** 5 minutes to cross
  - **Larry:** 10 minutes to cross
- A pair must walk together at the rate of the slower man's pace

Three Sentence Summary of Knapsack Approximation

Knapsack

- We've seen knapsack in dynamic programming (but with a pseudo-polynomial running time)
- We've seen knapsack as an NP-complete problem
- Now, can we approximate it in fully polynomial time?
- Recall:
  - We have n items
  - Each item has a weight w_i and a value v_i
  - We want to maximize total value without going over our weight capacity W

The best approximation yet!

- Our algorithm will take those items and the capacity ***W*** as well as a parameter ε
- We will find a set of items ***S*** within the weight capacity whose value is at worst $\frac{1}{1+\varepsilon}$ of the optimal!
- And the algorithm will be polynomial for any **particular** choice of ε
  - But it will not be polynomial in ε, if that makes sense
- This kind of algorithm is called a **polynomial-time approximation scheme** (PTAS)

Algorithm design

- We had a pseudo-polynomial algorithm for knapsack that ran in time $O(\boldsymbol{nW})$
- The book gives details on how we can flip around weights and values to get a dynamic programming knapsack algorithm that runs in time $O(\boldsymbol{n}^2\boldsymbol{v}^*)$ where $\boldsymbol{v}^*$ is the largest value of any item (if values are integers)
- Let's assume that algorithm is correct and build our approximation algorithm out of it

Algorithm design continued

- If v^* is a small integer, then we can run the algorithm as is
- However, if v^* is large, we can round the values up and use small integers instead:
  - v_1 = 1,983,929
  - v_2 = 2,437,888
  - v_3 = 621,653
- Rounding up to millions we get
  - v_1 = 2,000,000
  - v_2 = 3,000,000
  - v_3 = 1,000,000
- We can treat those values like 2, 3, and 1, respectively

Rounding notation

- We use a rounding factor ***b***
- Each rounded value $\widetilde{v_i} = \lceil v_i/b \rceil b$
- Note that $v_i \leq \widetilde{v_i} \leq v_i + b$
- To get small values, we can scale the rounded values down by ***b***:

$$\widehat{v_i} = \frac{\widetilde{v_i}}{b} = \lceil v_i/b \rceil$$

- Note that the knapsack problem with values $\widetilde{v_i}$ has the same optimum solution as the problem with $\widehat{v_i}$, if you scale the answers by ***b***

Approximate knapsack algorithm

- Knapsack-Approx(ε)
  - Set $b = (\varepsilon/(2n)) \max_i v_i$
  - Solve the Knapsack problem with values $\hat{v}_i$
  - Return the set S of items found

Approximation running time

- We only rounded the values, not the weights, so the answer we get is legal
- The algorithm we use as a subroutine runs in time O($\boldsymbol{n}^2\boldsymbol{v}^*$) where $\boldsymbol{v}^*$ is the biggest value
- Since $\boldsymbol{b} = (\varepsilon/(2\boldsymbol{n}))\max_i \boldsymbol{v}_i$, the biggest value $\boldsymbol{v}_j$ will also have the biggest rounded value:

$$\hat{v}_j = \lceil v_j/b \rceil = \left\lceil \frac{v_j}{v_j\varepsilon/(2n)} \right\rceil = \left\lceil \frac{2n}{\varepsilon} \right\rceil = c \cdot n\varepsilon^{-1}$$

- So our algorithm on rounded values runs in time O($\boldsymbol{n}^3\varepsilon^{-1}$)

Approximation bound

- **Claim:**
  - If ***S*** is the solution found by our approximation algorithm and ***S**** is any other solution such that $\sum_{i \in S^*} w_i \leq W$, then $(1+\varepsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$.
- **Proof:**
  - Let ***S**** be any set such that $\sum_{i \in S^*} w_i \leq W$.
  - Our algorithm finds the **optimal** solution with values $\widetilde{v}_i$ so

$$\sum_{i \in S} \widetilde{v}_i \geq \sum_{i \in S^*} \widetilde{v}_i$$

Approximation bound continued

- The rounded values are close to the real values, so

$$\sum_{i \in S^*} v_i \le \sum_{i \in S^*} \tilde{v}_i \le \sum_{i \in S} \tilde{v}_i \le \sum_{i \in S} (v_i + b) \le nb + \sum_{i \in S} v_i$$

- To make sense of this, we need to bound ***nb***
- Let ***j*** be the item with the largest value
- By our choice of ***b***, $v_j = 2\varepsilon^{-1} nb$, making $v_j = \tilde{v}_j$
- Assuming that each item could fit by itself in the knapsack

$$\sum_{i \in S} \tilde{v}_i \ge \tilde{v}_j = 2\varepsilon^{-1} nb$$

Approximation bound continued

- On the previous slide, we established that $\sum_{i \in S} v_i \geq \sum_{i \in S} \tilde{v}_i - nb$
- Since $\sum_{i \in S} \tilde{v}_i \geq \tilde{v}_j = 2\varepsilon^{-1} nb$,

$$\sum_{i \in S} v_i \geq 2\varepsilon^{-1} nb - nb = (2\varepsilon^{-1} - 1) nb$$

- For ε ≤ 1, $2 - \varepsilon \geq 1$, thus,

$$nb \leq (2 - \varepsilon) nb \leq \varepsilon \sum_{i \in S} v_i$$

- Leading finally to

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + nb \leq (1 + \varepsilon) \sum_{i \in S} v_i$$

■

Polynomial-time approximation schemes (PTAS)

- The consequences are that we can approximate knapsack arbitrarily well
  - It will take time polynomial with respect to $\frac{1}{\varepsilon}$ and get us an approximation within $\frac{1}{1+\varepsilon}$ of the optimal!
  - Of course, as ε gets closer to zero, the running time shoots to exponential
- Lots of variations of knapsack also have a PTAS
- Partitioning numbers into subsets that are as close as possible has a PTAS
- The Euclidean traveling salesman problem (in which all the locations are locations on a plane or in 3D space) has a PTAS
- There are also randomized algorithms that have a high probability of being within a factor of the optimal
- Many NP-hard problems do not have a PTAS
  - ... unless P = NP

And that's that.

- Now you have a sense of the problems we know how to solve
  - Greedy algorithms take the best thing at a given moment
  - Divide and conquer divides problems into subproblems, sometimes improving the speed we could solve with greedy
  - Dynamic programming allows us to manage problems that have many (but only polynomially many) subproblems
- NP-complete and NP-hard problems appear to take too long to solve
  - But some can be approximated!
- Undecidable problems simply cannot be solved with algorithms
- Complex as this course was, it's only a taste of the richness out there

Quiz

Upcoming

Next time…

- Review up to Exam 1 and a little beyond
- Review Chapters 1-3

Reminders

- Work on Assignment 7
  - Due the last day of class