COMPSCI 311: Introduction to Algorithms

Lecture 25: Randomized and Approximation Algorithms

Marius Minea

University of Massachusetts Amherst

Review: Randomized Algorithms

- Efficient in expectation
- Optimal with high probability
- Break (undesired) symmetry
- Show some solution exists, or derive bound on number

Types of randomized algorithms:

- Fail with some small probability (Monte Carlo)
- Always succeed, but running time is random (Las Vegas)

Last time:

- Min-Cut
- Contention Resolution
- Max 3-SAT

This time:

- Median selection

Median Selection

Let's precisely define the median.

Given a set $S = \{a_1, a_2, \ldots, a_n\}$, the median is the k^{th} smallest element, where $k = \lceil n/2 \rceil$:
• $k = n/2$, if n even
• $k = (n+1)/2$, if n odd

More generally: find k^{th} smallest number. Complexity ?

$k = 1$: min $\quad O(n)$

$k = n$: max $\quad O(n)$

$k = n/2$: ???
$O(n \log n)$ by sorting

Can do deterministic $O(n)$ (Blum, Floyd, Pratt, Rivest, Tarjan '72)

Today: randomized (similar idea). Assume no duplicates.

Divide and Conquer: Choose a Splitter

- Choose a splitter (pivot) $a_i \in S$
- Find elements smaller and larger than splitter:

$S^- = \{a_j : a_j < a_i\}$, $S^+ = \{a_j : a_j > a_i\}$.

Cases:

- $|S^-| = k-1$: a_i is the sought element, done
- $|S^-| \geq k$: recurse on (S^-, k)
- $|S^+| < k-1$: recurse on $(S^+, k - (|S^|+1))$

How to choose splitter?

Would it help to land "close" to the sought element?
Not if we land inside the larger of the two sets.

Work done (splitting) is proportional to size of whole set
⇒ Want recursive calls to work on (much) smaller sets

- Best splitter is the median!

  $T(n) \leq T(n/2) + cn \quad \Rightarrow O(n)$ runtime
- Worst case: splitter is extremal element

  $T(n) \leq T(n-1) + cn$
- General: splitter separates ϵn elements, $(1-\epsilon)n$ remain

  $T(n) \leq T((1-\epsilon)n) + cn$

  $T(n) \leq cn[1 + (1-\epsilon) + (1-\epsilon)^2 + \ldots] \leq \frac{cn}{\epsilon}$

Randomized Splitters

Simple Idea: Choose splitter uniformly at random.

Can't hope to split into *exactly* half, but will see how long it takes to separate at least $1/4$

- Call a splitter *central* if it separates $1/4$ of elements
- $\Pr[\text{centralsplitter}] = 1/2$ (must be in [1/4, 3/4] length)
- expected number of attempts N: $E[N] = 1/p = 2$

A central splitter reduces the set to 3/4 the size.

⇒ count *phases* of the algorithm, where $(\frac{3}{4})^{j+1} n < |S| \leq (\frac{3}{4})^j n$ (initially in phase 0)

Running Time Analysis

Let X be a random variable counting the elementary steps done by the algorithm, and X_j the steps in phase j

$X = X_0 + X_1 + X_2 + \ldots$

One iteration on size n: cn steps.

Expected: two iterations per phase, size $\leq (\frac{3}{4})^j n$

$E[X] = \sum_j E[X_j] \leq \sum_j 2cn(\frac{3}{4})^j = 2cn \sum_j (\frac{3}{4})^j \leq 8cn$

Application: Randomized Quicksort

Modified Quicksort:

- Choose random pivot, retry if not central (expected tries: 2)
- Divide array into S^-, S^+
- Recursively sort both
- Concatenate

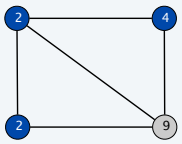
Easy to analyze: *expected* work on each level is $\Theta(n)$

Can prove the same if continuing with any pivot

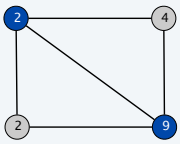
Weighted vertex cover

Definition. Given a graph $G = (V, E)$, a vertex cover is a set $S \subseteq V$ such that each edge in E has at least one end in S.

Weighted vertex cover. Given a graph G with vertex weights, find a vertex cover of minimum weight.



weight = 2 + 2 + 4



weight = 11

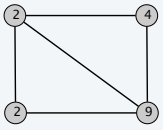
slide credit: Kevin Wayne / Pearson

Pricing method

Pricing method. Each edge must be covered by some vertex.
Edge $e = (i, j)$ pays price $p_e \geq 0$ to use both vertex i and j.

Fairness. Edges incident to vertex i should pay $\leq w_i$ in total.

for each vertex i : $\sum_{e=(i,j)} p_e \leq w_i$



Fairness lemma. For any vertex cover S and any fair prices p_e: $\sum_e p_e \leq w(S)$.

Pf. $\sum_{e \in E} p_e \leq \sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in S} w_i = w(S)$. ▪

each edge e covered by at least one node in S

sum fairness inequalities for each node in S

slide credit: Kevin Wayne / Pearson

Pricing method

Set prices and find vertex cover simultaneously.

WEIGHTED-VERTEX-COVER (G, w)

$S \leftarrow \emptyset$.

FOREACH $e \in E$

  $p_e \leftarrow 0$.

$\sum_{e=(i,j)} p_e = w_i$

WHILE (there exists an edge (i, j) such that neither i nor j is tight)

  Select such an edge $e = (i, j)$.

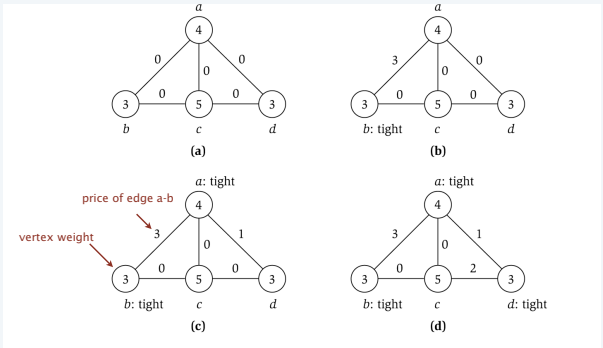
  Increase p_e as much as possible until i or j tight.

$S \leftarrow$ set of all tight nodes.

RETURN S.

slide credit: Kevin Wayne / Pearson

Pricing method example



slide credit: Kevin Wayne / Pearson

Pricing method: analysis

Theorem. Pricing method is a 2-approximation for WEIGHTED-VERTEX-COVER.

Pf.

- Algorithm terminates since at least one new node becomes tight after each iteration of while loop.
- Let S = set of all tight nodes upon termination of algorithm. S is a vertex cover: if some edge (i, j) is uncovered, then neither i nor j is tight. But then while loop would not terminate.
- Let S^* be optimal vertex cover. We show $w(S) \le 2\, w(S^*)$.

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e \le \sum_{i \in V} \sum_{e=(i,j)} p_e = 2 \sum_{e \in E} p_e \le 2w(S^*). \quad \blacksquare$$

all nodes in S are tight; $S \subseteq V$, prices ≥ 0; each edge counted twice; fairness lemma

30

slide credit: Kevin Wayne / Pearson

Knapsack Problem

- n items, weights w_i, values v_i, total capacity W
- **Goal**: maximize total value without exceeding capacity

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Knapsack is NP-complete: SUBSET-SUM $\le_P$ KNAPSACK
- We already saw a $O(nW)$ dynamic programming algorithm

Knapsack Problem Dynamic Programming v1

Choose from objects 1 through j, index by achieved weight

$$\mathsf{OPT}(j, w) = \begin{cases} \mathsf{OPT}(j-1, w) & w_j > w \\ \max \left\{ \begin{array}{c} \mathsf{OPT}(j-1, w) \\ v_j + \mathsf{OPT}(j-1, w - w_j) \end{array} \right\} & w_j \le w \end{cases}$$

Good for small (maximum) weight.

Knapsack Problem Dynamic Programming v2

Definition. $\mathsf{OPT}(i, v)$ = min weight of a knapsack for which we can obtain a solution of value $\ge v$ using a subset of items $1, \ldots, i$

$$\mathsf{OPT}(i, v) = \min \{\mathsf{OPT}(i-1, v),\ w_i + \mathsf{OPT}(i-1, v - v_i)\}$$
$$\mathsf{OPT}(i, v) = 0 \quad \text{if } v \le 0$$
$$\mathsf{OPT}(0, v) = \infty \quad \text{if } v > 0$$

- Running time is $O(nV)$ where V is an upper bound on total value, e.g. $V = nv_{\max}$
- Not polynomial in input size
- Polynomial if values are small integers

Knapsack Problem Approximation Algorithm

Idea: round all values (up) to coarser units and run dynamic programming algorithm

item	value	weight
1	934,221	1
2	5,956,342	2
3	17,810,013	5
4	21,217,800	6
5	27,734,384	7

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

The result will never exceed the weight capacity, but may lose some value due to rounding error.

Rounding Details

Round to nearest multiple of some integer b:

- v_i = original value. 5,956,342
- $\bar{v}_i = \left\lceil \frac{v_i}{b} \right\rceil b$ = rounded value. 6,000,000
- $\left\lceil \frac{v_i}{b} \right\rceil$ = value actually used in DP. 6

Analysis

- Let S = rounded DP solution, S^* = optimal solution
- We'll show that S gets nearly as much value as S^*:

$$\begin{aligned}
\sum_{i \in S^*} v_i &\leq \sum_{i \in S^*} \bar{v}_i && \text{round up} \\
&\leq \sum_{i \in S} \bar{v}_i && S \text{ optimal for rounded values} \\
&\leq \sum_{i \in S} (v_i + b) && \text{rounding amount} \leq b \\
&\leq \sum_{i \in S} v_i + nb \\
&\leq (1+\epsilon) \sum_{i \in S} v_i
\end{aligned}$$

The last inequality is true if $nb \leq \epsilon \sum_{i \in S} v_i$. We need $b \leq \epsilon v_{\max}/n$.

Running Time

- Recall $b \leq \epsilon v_{\max}/n$.
  We need $b \geq 1$, thus we can approximate if $\epsilon \geq n/v_{\max}$.
  Require $\epsilon \geq \frac{c}{c-1} n/v_{\max}$ for some $c > 1$
- $b = \lfloor \epsilon v_{\max}/n \rfloor > \frac{\epsilon v_{\max}}{n} - 1 \geq \frac{\epsilon v_{\max}}{n}(1 - \frac{c-1}{c}) = \frac{\epsilon v_{\max}}{cn}$, or $v_{\max}/b < cn/\epsilon$.
- $\bar{v}_{\max} = \lceil v_{\max}/b \rceil < v_{\max}/b + 1 < cn/\epsilon + 1$
- Running time for rounded DP is $O(n^2 \bar{v}_{\max}) = O(n^3/\epsilon)$

Theorem: The rounding algorithm computes a solution whose value is within a $(1+\epsilon)$ factor ($\epsilon > 0$) of the optimum in $O(n^3/\epsilon)$ time.