CS 420: Advanced Algorithm Design and Analysis
Spring 2015 – Lecture 8

Department of Computer Science
University of British Columbia

January 29, 2015

Announcements

Assignments...

- Asst3...due today
- Asst2...back

Upcoming Exams / Q/A Sessions ...

- review session: Tuesday, Feb. 03, 5:30-7:00; DMPT 301
  - Note...this *replaces* the group office hour normally held on Wednesday 3:30-5:00
- Midterm I: Wednesday, Feb. 04, 5:30-7:00; DMPT 301
  - covers material up to (and including) Lecture 8 (today)

Announcements

Readings...

- material on hashing [Kleinberg, 13.6; Cormen+, chap 11; Erickson, chapt 12]
- material on closest-pair problem [Kleinberg]
- material on optimal binary search trees [Erickson 3.5, 5.6; Cormen+, chapt 13]
- material on adaptive (self-adjusting) search structures; splay trees [Erickson, chapt. 16]

Looking ahead...

Our goal, in the next few lectures is to understand how we might circumvent this lower bound, by *stepping outside the abstract comparison-based model*. We will consider:

- exploiting assumptions about the structure/size of the key space $\mathcal{U}$
- exploiting assumptions about the distribution of keys in S
- exploiting assumptions about the pattern of successive queries
- (if time permits) other issues: randomization, error tolerance...

Last class...

Applications of universal hashing (cont.)

- finding the closest pair of points in a point set
  - a randomized approach [Kleinberg&Tardos (section 13.7)]
    - Golin et al variation of Rabin's algorithm
    - randomized incremental approach...keep updating the closest pair
    - use neighbourhood search structure...implemented as a hash table

Last class...

Dictionaries with non-uniform access patterns

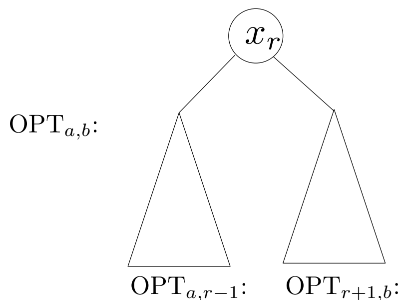
- fixed (known) access frequencies
  - list-structured dictionaries...sorted by access frequency (decreasing) is optimal
  - tree-structured dictionaries...*optimal binary search trees*
    - definition of optimality...natural heuristics don't work
    - optimal substructure property
    - cost recurrence
    - solution by dynamic programming

Today...

Dictionaries with non-uniform access patterns

- fixed (known) access frequencies
  - finish discussion of optimal BSTs
- unknown/changing access probabilities...adaptive search structures
  - list structures...natural adaptive heuristics
    - *competitive analysis* of move-to-front
    - application in data compression
  - tree structures...splay trees

What does an *optimal* binary search tree look like?



Optimal trees satisfy the *optimal substructure* property

How can we describe its cost?

Let $C_{a,b}$ denote the *cost* of $\mathrm{OPT}_{a,b}$. How does $C_{a,b}$ relate to $C_{a,r-1}$ and $C_{r+1,b}$?

$$C_{a,b} = \begin{cases} 0 & \text{if } a > b \\ \min_{a \le r \le b}\{C_{a,r-1} + C_{r+1,b} + W_{a,b}\} & \text{if } a \le b \end{cases} \quad (1)$$

where

$$W_{a,b} = \sum_{a \le i \le b} p_i$$

denotes the cost associated with the root node (x_r).

How can we compute its cost?

Recursively (using (1))? bad idea

- each application of (1) generates many subproblems of size comparable to the original
- there are only $\Theta(n^2)$ distinct subproblems *in total*!

This should suggest a different approach: *dynamic programming*

A dynamic programming solution

Algorithm pseudocode for optimal BST cost computation

```
1: for l = 1 to n do
2:     for a = 1 to n − l + 1 do
3:         evaluate and tabulate C_{a,a+l−1} using equation (1)
4:     end for
5: end for
```

Analysis:

- each new entry $C_{a,a+l-1}$ can be computed in $O(n)$ time
- total cost ($O(n^2)$ entries) is $O(n^3)$
- Note: all $W_{a,b}$ values can be computed in $O(n)$ time!

Additional remarks

- the approach extends naturally to the situation where we know q_j values: the probability of accessing a key *not in the dictionary*, between x_{j-1} and x_j.
- total cost can be reduced to $O(n^2)$, by exploiting the fact that the root of $\mathrm{OPT}_{a,b}$ must lie between the root of $\mathrm{OPT}_{a,b-1}$ and the root of $\mathrm{OPT}_{a+1,b}$.

Adaptive search with list-structured dictionaries

The rules...

- we maintain the set $S = \{x_1, \ldots, x_n\}$ as a linear list L, and perform sequential search in L for each query.
- we are free to reorganize the list to try and minimize the cumulative search cost
- we charge
  - cost i if we access the i-th element on the list
  - no additional cost to relocate accessed element anywhere earlier in the list
  - all other restructuring is done by *exchanges* of adjacent elements, at a unit cost per exchange
- can consider insertion and deletion as well...

Adaptive search with list-structured dictionaries

Natural restructuring ideas...

- Frequency-Count: Maintain a frequency count of individual key accesses. Maintain the list in order of decreasing access frequency
- Transpose: Following the access of a key, exchange it with its predecessor on the list
- Move-to-Front: Following the access of a key, move it all the way to the front of the list (keeping the order of other keys unchanged)
- Decreasing-Total-Frequency: Build an optimal static list, based on knowledge of total access frequency (probability)
- Clairvoyant: Restructure to minimize total cost, knowing in advance the sequence of queries (optimal dynamic list).

Adaptive search with list-structured dictionaries

Competitive analysis...

- We will compare the cost of Move-to-Front (MF) with the Decreasing-Total-Frequency (DF) and the Clairvoyant-Restructuring (CR) strategies *on the same family of access sequences*
- Let p_i, $1 \leq i \leq n$, be the access probability for key x_i; we assume $p_i \geq p_j$, for all $i \leq j$.
- We already know that the expected cost of DF, E_{DF}, is $\sum_{1 \leq i \leq n} p_i \cdot i$
- How does the expected cost of MF, E_{MF}, compare?

MF is 2-competitive..

Theorem

$E_{MF} \leq 2 \cdot E_{DF} - 1$

Proof.

It suffices to demonstrate the Lemma below, since
$E_{MF} = 1 + 2 \cdot \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j} \leq 1 + 2 \cdot \sum_{1 \leq i < j \leq n} p_j$
$\leq 1 + 2 \cdot \sum_{1 \leq j \leq n} (j-1) p_j = 2 \cdot E_{DF} - 1$ □

Lemma

$E_{MF} = 1 + 2 \cdot \sum_{1 \leq i < j \leq n} \frac{p_i p_j}{p_i + p_j}$

Key idea: what is the probability (in "steady state") that key x_j appears before key x_i?

$\frac{p_j}{p_i + p_j}$

MF is 2-competitive..

In fact, the Move-to-Front heuristic is 2-competitive in an even stronger sense [c.f. D.D. Sleator and R.E. Tarjan, "Amortized efficiency of list update and paging rules", *Communications of ACM*, 28(2), 1985, pp. 202-208.]...

Theorem

For any algorithm A and any sequence s of m access, insert, and delete operations starting with the empty list

$$C_{MF}(s) \leq 2C_A(s) - m$$

Proof.

an *amortization argument*...
...using the number of *inversions* between competing lists as a *potential function*

Data Compression

Linear list based scheme [Bentley et al., C.ACM '86]

- first encode S as a sequence of integers
- then encode the integers using a variable length prefix code

Intuition:

- integer j can be encoded in close to $\lg j$ bits
  - in fact $1 + \lfloor \lg j \rfloor + 2\lfloor \lg(1 + \lfloor \lg j \rfloor) \rfloor$ bits suffice
- move-to-front on list Σ can be used to encode frequently occurring symbols with small integers

Data Compression

Linear list based scheme [Bentley et al., C.ACM '86]

Properties:

- the (adaptive) scheme is never much worse than the optimal static scheme (Huffman) and can be much better if the local frequency of symbols can deviate significantly from the global frequency
- if strings are generated by a fixed (but unknown) probability distribution, the code length is optimal (up to lower order terms)
- idea combines with *Burrows-Wheeler transform* to enable modern compression schemes (such as bzip2)

What about tree-structured dictionaries?

The rules...

- we maintain the set $S = \{x_1, \ldots, x_n\}$ as a binary search tree T, and perform search in T for each query.
- we are free to reorganize the tree to try and minimize the cumulative search cost
- we charge
  - cost i if we access an element at depth $i - 1$ in the tree
  - all restructuring is done by *local modifications* within the tree, at a unit cost per operation
- what is the analog of the (list) transpose operation?
  $+$ *tree rotation*!

Next time...

Exploiting non-uniform access patterns

- unknown/changing access probabilities
  ... adaptive (self-organizing) search structures
  - tree-structured dictionaries