CS 420: Advanced Algorithm Design and Analysis
Spring 2015 – Lecture 7

Department of Computer Science
University of British Columbia

January 27, 2015

Announcements

Assignments...

- Asst3...due Thursday

Upcoming Exams / Q/A Sessions ...

- review session: Tuesday, Feb. 03, 5:30-7:00; DMPT 301
- exam: Wednesday, Feb. 04, 5:30-7:00; DMPT 301
  - covers material up to (and including) Lecture 8 (January 29)

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...

Assignment 1 discussion

Compact universal families $\mathcal{H}$ exist and are efficient to construct

- Kleinberg&Tardos (section 13.6) describe one construction based on modular arithmetic

Applications of universal hashing (cont.)

- finding the closest pair of points in a point set
  - intuition from 1-d: reduction to sorting
  - divide and conquer in 2-d [Kleinberg&Tardos (section 5.4)]
  - identification of *neighbourly* point pairs, using *Voronoi diagrams*

Today...

Applications of universal hashing (cont.)

- finding the closest pair of points in a point set
  - a randomized approach [Kleinberg&Tardos (section 13.7)]

Today...

Applications of universal hashing (cont.)

- finding the closest pair of points in a point set
  - a randomized approach [Kleinberg&Tardos (section 13.7)]

Dictionaries with non-uniform access patterns

- fixed (known) access frequencies
  - list-structured dictionaries
  - tree-structured dictionaries...*optimal binary search trees*

Finding the closest pair of points in a point set

Problem definition

Given a collection of n points in real d-dimensional space, identify the pair of points $\{p_i, p_j\}$whose *separation* ($||p_i - p_j||$) is smallest.

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

Historical note...

- The first problem that popularized the idea of using *randomization* in the design of algorithms. [Michael Rabin, early 1970's]

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

Historical note...

- The first problem that popularized the idea of using *randomization* in the design of algorithms. [Michael Rabin, early 1970's]
- *Las Vegas* algorithms (unlike *Monte Carlo* algorithms) use randomization to reduce the complexity of deterministic algorithms, *without compromising correctness*.
- The approach described here (and in Kleinberg) is a modification published by Golin et al., 1995.

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

Algorithm randomized closest-pair in $[0,1]^2$

re-order input points randomly: $p_1, p_2, \ldots, p_n$
$\sigma_{\min} \leftarrow \sigma(p_1, p_2)$; $i \leftarrow 3$
while $i < n+1$ **do**
    while $\mathrm{N}_{\sigma_{\min}}(p_i) \cap \{p_1, \ldots, p_{i-1}\} = \emptyset$ **do**
        $i \leftarrow i+1$
    end while
    $p_j \leftarrow$ closest point in $\{p_1, \ldots, p_{i-1}\}$ to p_i
    $\sigma_{\min} \leftarrow \sigma(p_i, p_j)$
    $i \leftarrow i+1$
end while

where $\mathrm{N}_{\sigma_{\min}}(p)$ denotes the $\sigma_{\min}$-neighbourhood of p

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

Algorithm randomized closest-pair in $[0,1]^2$

re-order input points randomly: $p_1, p_2, \ldots, p_n$
$\sigma_{\min} \leftarrow \sigma(p_1, p_2)$; $i \leftarrow 3$
while $i < n+1$ **do**
    while $\mathrm{N}_{\sigma_{\min}}(p_i) \cap \{p_1, \ldots, p_{i-1}\} = \emptyset$ **do**
        $i \leftarrow i+1$
    end while
    $p_j \leftarrow$ closest point in $\{p_1, \ldots, p_{i-1}\}$ to p_i
    $\sigma_{\min} \leftarrow \sigma(p_i, p_j)$
    $i \leftarrow i+1$
end while

where $\mathrm{N}_{\sigma_{\min}}(p)$ denotes the $\sigma_{\min}$-neighbourhood of p

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

The algorithm proceeds in a sequence of *stages* during which the *stage invariant* $\mathrm{N}_{\sigma_{\min}}(p_i) \cap \{p_1, \ldots, p_{i-1}\} = \emptyset$ holds.

Between stages we update $\sigma_{\min}$ by computing
$p_j \leftarrow$ closest point in $\{p_1, \ldots, p_{i-1}\}$ to p_i

Finding the closest pair of points in a point set

A randomized incremental approach in $\Re^2$

The algorithm proceeds in a sequence of *stages* during which the *stage invariant* $\mathrm{N}_{\sigma_{\min}}(p_i) \cap \{p_1, \ldots, p_{i-1}\} = \emptyset$ holds.

Between stages we update $\sigma_{\min}$ by computing
$p_j \leftarrow$ closest point in $\{p_1, \ldots, p_{i-1}\}$ to p_i

The cost depends on

- the cost of testing the stage invariant
- the number and cost of stage transitions

Testing the stage invariant

Testing the stage invariant

- divide space $[0, 1]^2$ into cells of side length $\sigma_{\min}/2$

Testing the stage invariant

- divide space $[0,1]^2$ into cells of side length $\sigma_{\min}/2$
- point p belongs to $\mathrm{cell}(p) = (\lfloor \frac{p.x}{\sigma_{\min}/2} \rfloor, \lfloor \frac{p.y}{\sigma_{\min}/2} \rfloor)$

Testing the stage invariant

- divide space $[0,1]^2$ into cells of side length $\sigma_{\min}/2$
- point p belongs to $\mathrm{cell}(p) = (\lfloor \frac{p.x}{\sigma_{\min}/2} \rfloor, \lfloor \frac{p.y}{\sigma_{\min}/2} \rfloor)$
- by construction *no cell contains more than one point* among $\{p_1, \ldots, p_{i-1}\}$

Testing the stage invariant

- divide space $[0, 1]^2$ into cells of side length $\sigma_{\min}/2$
- point p belongs to $\mathrm{cell}(p) = (\lfloor \frac{p.x}{\sigma_{\min}/2} \rfloor, \lfloor \frac{p.y}{\sigma_{\min}/2} \rfloor)$
- by construction *no cell contains more than one point* among $\{p_1, \ldots, p_{i-1}\}$
- stage invariant fails if point p_i has a point among $\{p_1, \ldots, p_{i-1}\}$ in the *neighbourhood* of $\mathrm{cell}(p_i)$

Cost of stage transitions

Cost of stage transitions

- when $\sigma_{\min}$ is updated need to rebuild the neighbourhood search structure

Cost of stage transitions

- when $\sigma_{\min}$ is updated need to rebuild the neighbourhood search structure
  recall implicit initialization

Cost of stage transitions

- when $\sigma_{\min}$ is updated need to rebuild the neighbourhood search structure
  recall implicit initialization
- need to re-insert i points if stage ends on i-th input

Cost of stage transitions

- when $\sigma_{\min}$ is updated need to rebuild the neighbourhood search structure
  recall implicit initialization
- need to re-insert *i* points if stage ends on *i*-th input
- how many stages do we expect?

Cost of stage transitions

- when $\sigma_{\min}$ is updated need to rebuild the neighbourhood search structure
  recall implicit initialization
- need to re-insert i points if stage ends on i-th input
- how many stages do we expect?
  $\Theta(\lg n)$

How do we represent the neighbourhood search structure?

How do we represent the neighbourhood search structure?

It is just a BIG dictionary...

How do we represent the neighbourhood search structure?

It is just a BIG dictionary... use a hash table

How do we represent the neighbourhood search structure?

It is just a BIG dictionary... use a hash table

- $O(1)$-time expected cost for insertion and neighbourhood queries (find)

How do we represent the neighbourhood search structure?

It is just a BIG dictionary... use a hash table

- $O(1)$-time expected cost for insertion and neighbourhood queries (find)
- space is $O(n)$

Total expected cost

Total expected cost

- find operations: $O(n)$ (only look in $O(1)$ cells per point)

Total expected cost

- find operations: $O(n)$ (only look in $O(1)$ cells per point)
- distance calculations: $O(n)$ (only compute distance with $O(1)$ neighbours)

Total expected cost

- find operations: $O(n)$ (only look in $O(1)$ cells per point)
- distance calculations: $O(n)$ (only compute distance with $O(1)$ neighbours)
- rebuild operations: $O(s)$, where s is the number of stages

Total expected cost

- find operations: $O(n)$ (only look in $O(1)$ cells per point)
- distance calculations: $O(n)$ (only compute distance with $O(1)$ neighbours)
- rebuild operations: $O(s)$, where s is the number of stages
- insert operations: $n + \sum_{1 \leq i \leq n}(iX_i)$, where $X_i = 1$ if the i-th insert leads to a closest pair update (and $X_i = 0$ otherwise).

Total expected cost

- find operations: $O(n)$ (only look in $O(1)$ cells per point)
- distance calculations: $O(n)$ (only compute distance with $O(1)$ neighbours)
- rebuild operations: $O(s)$, where s is the number of stages
- insert operations: $n + \sum_{1 \leq i \leq n}(iX_i)$, where $X_i = 1$ if the i-th insert leads to a closest pair update (and $X_i = 0$ otherwise).

So the total expected cost is $O(n)$.

See Kleinberg&Tardos (Section 13.7) for full details...

Exploiting non-uniform access patterns

Exploiting non-uniform access patterns

Known access probabilities for individual keys

- suppose each key $x_i \in S$ has a fixed (and known) associated access probability p_i

Exploiting non-uniform access patterns

Known access probabilities for individual keys

- suppose each key $x_i \in S$ has a fixed (and known) associated access probability p_i
- how can we exploit this??

Exploiting non-uniform access patterns

Known access probabilities for individual keys

- suppose each key $x_i \in S$ has a fixed (and known) associated access probability p_i
- how can we exploit this??

Suppose our dictionary is a *list* L

- how should we organize L to minimize the *expected access cost*: $\sum_{x_i \in S} p_i \cdot \mathrm{posn}_L(x_i)$?

Exploiting non-uniform access patterns

Known access probabilities for individual keys

- suppose each key $x_i \in S$ has a fixed (and known) associated access probability p_i
- how can we exploit this??

Suppose our dictionary is a *list* L

- how should we organize L to minimize the *expected access cost*: $\sum_{x_i \in S} p_i \cdot \mathrm{posn}_L(x_i)$?
- simple interchange argument demonstrates that order by decreasing access probability is *optimal*

Known access probabilities for individual keys

What if our dictionary is a tree?

- how should we organize the tree T to minimize the *expected access cost*:

$$\sum_{x_i \in S} p_i \cdot (\text{depth}_T(x_i) + 1) = 1 + \sum_{x_i \in S} p_i \cdot \text{depth}_T(x_i)$$

Known access probabilities for individual keys

What if our dictionary is a tree?

- how should we organize the tree T to minimize the *expected access cost*:

$$\sum_{x_i \in S} p_i \cdot (\text{depth}_T(x_i) + 1) = 1 + \sum_{x_i \in S} p_i \cdot \text{depth}_T(x_i)$$

- this tree is called an *optimal binary search tree*

Some reasonable *heuristics*...

What does our experience suggest?

Some reasonable *heuristics*...

What does our experience suggest?

- balance the tree (minimize the maximum node depth)

Some reasonable *heuristics*...

What does our experience suggest?

- balance the tree (minimize the maximum node depth)
- choose root to balance the probabilities of subtrees

Some reasonable *heuristics*...

What does our experience suggest?

- balance the tree (minimize the maximum node depth)
- choose root to balance the probabilities of subtrees
- put element with highest probability at the root

Some reasonable *heuristics*...

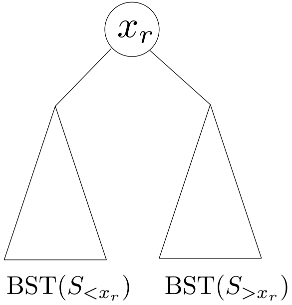
What does our experience suggest?

- balance the tree (minimize the maximum node depth)
- choose root to balance the probabilities of subtrees
- put element with highest probability at the root

None of these guarantees optimal behaviour.

What does a binary search tree look like?

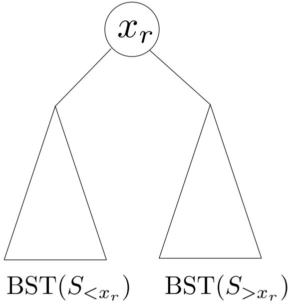
BST(S):



BST($\emptyset$)=NIL:

What does a binary search tree look like?

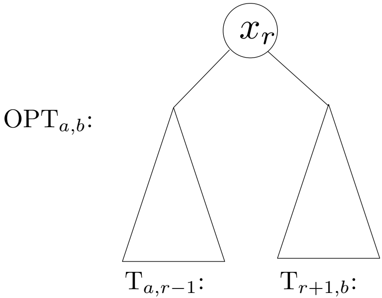
BST(S):



BST($\emptyset$)=NIL:

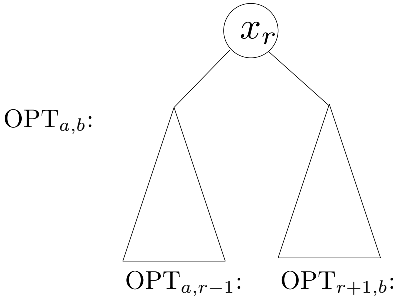
$$\text{depth}_{\text{BST}(S)}(x) = \begin{cases} 1 + \text{depth}_{\text{BST}(S_{<x_r})}(x) & \text{if } x < x_r \\ 0 & \text{if } x = x_r \\ 1 + \text{depth}_{\text{BST}(S_{>x_r})}(x) & \text{if } x > x_r \end{cases}$$

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



$\mathrm{OPT}_{a,b}$ denotes the optimal tree for the key sequence $x_a, x_{a+1}, \ldots, x_b$

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 $C_{r+1,b}$?

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 $C_{r+1,b}$?

$$C_{a,b} = \begin{cases} 0 & \text{if } a > b \\ \min_{a \leq r \leq b}\{C_{a,r-1} + C_{r+1,b} + W_{a,b}\} & \text{if } a \leq b \end{cases} \tag{1}$$

where

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

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

How can we compute its cost?

Recursively (using (1))?

How can we compute its cost?

Recursively (using (1))? bad idea

How can we compute its cost?

Recursively (using (1))? bad idea

- each application of (1) generates many subproblems of size comparable to the original

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*!

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:

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

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

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 $O(n^3)$

A dynamic programming solution

Algorithm pseudocode for optimal BST cost computation

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

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.

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 $\mathrm{OPT}_{a+1,b}$.

Next time...

Exploiting non-uniformity access patterns

- unknown/changing access probabilities; adaptive search structures (splay trees)