Algorithms and Data Structures

Optimal Search Trees; Tries

Ulf Leser
Static Key Sets

- Sometimes, the **set of keys is fixed**
  - Names in a city, customers of many companies, elements of a programming language, ...

- "Fixed" means: Searches are much more often than insdels
  - We may spend "more" time in reorganizing the tree after updates
  - These costs will get amortized over the next many searches

- Not unusual – example large-scale search engines
  - Recall: A search engine creates a dictionary; every word has a link to the set of documents containing it
  - The dictionary must be accessed very fast, changes are rare
  - Often, engines build complex structures to optimally support searching over the current set of documents
  - Changes are buffered and inserted in-bulk periodically
Optimal Search Trees

• Assume a set $K$ of keys and a bag $R$ of requests
  – Every search searches one $k \in K$; $k$’s may appear multiple times in $R$
• Naïve approach
  – Build an AVL tree over $K$
  – Every $r \in R$ costs $O(\log(|K|))$, i.e., we need $O(|R| \cdot \log(|K|))$
  – This is optimal, if every $k \in K$ appears with the same frequency in $R$
• What if $R$ is highly skewed?
  – Skewed (here): $k$’s are not equally distributed in $R$
  – Some keys are searched very often, others very rare
  – This is rather the norm than the exception in real life (Zipf, ...)
• Can we exploit this information for faster search?
Example

- $K = \{1,2,3,5,7,8,9,12,14\}$
- We build an AVL tree

- $R_1 = \{2,5,8,7,3,12,1,8,8\}$
  - $2+1+3+4+3+2+3+3+3 = 31$ comparisons
- $R_2 = \{9,9,1,9,2,9,5,3,9,1\}$
  - $4+4+3+4+2+4+1+3+4+3 = 32$ comparisons
Example

- \( K = \{1, 2, 3, 5, 7, 8, 9, 12, 14\} \)
- Let’s optimize the tree for R2

- \( R2 = \{9, 9, 1, 9, 2, 9, 5, 3, 9, 1\} = \{9, 9, 9, 9, 9, 1, 1, 2, 5, 3\} \)
  - 9 and 1 must be high in the tree
  - \( 1+1+1+1+2+2+4+3+5=21 \)
    - Versus 32
- Not good for R1
  - \( R1 = \{2, 5, 8, 7, 3, 12, 1, 8, 8\} \)
  - \( 4+3+5+4+5+2+2+5+5=35 \)
    - Versus 31
Content of this Lecture

- Optimal Search Trees
- Construction of Optimal Search Trees
- Searching Strings: Tries
Setting

- Assume a set $K$ of keys, $K = \{k_1, k_2, \ldots, k_n\}$
- Every $k$ is searched with frequency $a_1, a_2, \ldots, a_n$
- Furthermore, there are searches that fail
- Intervals $]-\infty, a_1[, a_1, a_2[, \ldots, a_{n-1}, a_n[, a_n, +\infty[$ are searched with frequencies $b_0, b_1, \ldots, b_n$
- We summarize these as $R = \{a_1, a_2, \ldots, a_n, b_0, b_1, \ldots, b_n\}$
Setting

- Assume a set $K$ of keys, $K = \{k_1, k_2, \ldots, k_n\}$
- Every $k$ is searched with frequency $a_1, a_2, \ldots, a_n$
- Furthermore, there are searches that fail
- Intervals $]-\infty, a_1[ , ]a_1, a_2[, , \ldots , ]a_{n-1}, a_n[, , ]a_n, +\infty[$ are searched with frequencies $b_0, b_1, \ldots, b_n$
- We summarize these as $R = \{a_1, a_2, \ldots, a_n, b_0, b_1, \ldots, b_n\}$
Optimal Search Trees

- We need to assign a **cost to every search tree** over $K$.
- **Definition**

  *Let $T$ be a search tree over $K$. The cost of $T$ for $R$ is:*

  $$P(T) = \sum_{i=1}^{n} \left( \text{depth}(k_i) + 1 \right) \cdot a_i + \sum_{j=0}^{n} \text{depth}(\langle k_j, k_{j+1} \rangle) \cdot b_j$$

- **Definition**

  *Let $K$ be a set of keys and $R$ a set of requests for $K$ (and not $K$). A search tree $T$ is optimal for $K$, $R$, iff*

  $$P(T) = \min \{ P(T') \mid T' \text{ is search tree for } K \}$$
One More Definition

• Definition

Let $T$ be a search tree over $K$. The weight of $T$ for $R$ is:

$$W(T) = \sum_{i=1}^{n} a_i + \sum_{j=0}^{n} b_j$$

• Thus, we weight of $T$ is simply $\sum R_i$
• But we will need this definition for subtrees
Content of this Lecture

- Optimal Search Trees
- Construction of Optimal Search Trees
- Searching Strings: Tries
Finding the Optimal Search Tree

- Bad news: There are **exponentially many search trees**
  - Proof omitted
- Implication: We cannot enumerate all search trees, compute their cost, and then chose the cheapest
- Good news: Optimal search trees have certain properties
Towards Divide&Conquer

• We can compute $P(T)$ recursively
• Let $k_r$ be root of $T$ and $T_l=\text{leftChild}(k_r)$, $T_r=\text{rightChild}(k_r)$
• Clearly: $P(T) = P(T_l) + P(T_r) + a_r + W(T_l) + W(T_r)
  = P(T_l) + P(T_r) + W(T)$
  – Since $W(T)$ is the same for every possible search tree, the cost of a tree only depends on the cost of its subtrees
• It follows: $T$ is optimal iff $T_l$ and $T_r$ are optimal
• Great! If we can solve the problem for smaller trees (=ranges of keys), we can inductively construct solutions for larger trees
  – We can use a divide&conquer strategy
Divide & Conquer

- Consider a range $R(i,j)$ of keys and intervals
  - $R(i,j) = \{ \left] k_i, k_{i+1} \right[, \left] k_{i+1}, k_{i+2} \right[, \ldots, k_j, \left] k_j, k_{j+1} \right[ \}$
- Assume that $R(i,j)$ is represented as a subtree of $T=T(1,n)$
  - Need not be the case in general; the “left” part of $R$ could lie in a different subtree than the “right” part
- One of the $k_i \in T(i,j)$ must be the root of this subtree
- Thus, $k_l$ divides $R(i,j)$ in two halves $R(i,l-1)$, $R(l,j)$
- Assume we know the optimal trees for all subranges $R(i,i+1)$, $R(i,i+2)$, $\ldots$, $R(i,j-1)$, $R(i+1,j)$, $\ldots$, $R(j-1,j)$
- Then, we can find $l$ and thus the optimal tree $T(i,j)$

$$P(T) = \min_{l=i+1 \ldots j} \left( P(T(i,l-1)) + P(T(l,j)) \right)$$
Illustration
Other subtrees are possible
Illustration
Illustration
Illustration
Bottom-Up

• We must **systematically enumerate** smaller $T(i,j)$ and puzzle them together to larger ones
• To compute $P(i,j)$, we need to know the smaller $T$-values, the smaller $W$-values, and we need to find $l$
• We perform **induction over the width of the interval**
• **Start**: $j=i+1$
  - The root is $k_{i+1}$
    - The only key in the interval; $l=i+1$
  - $\forall 0 \leq i < n$: $W(i,i+1) = b_i + a_{i+1} + b_{i+1}$
    $P(i,i+1) = W(i,i+1)$
Induction

- General case: $j-i=b>1$
  - By induction assumption, we know $W$ and $P$ values for all intervals of breadth $b-1$
  - For all pairs $i,j$ with $j-i=b$ and $0 \leq i<j<n$
  - First find the index $l$ for the optimal root of the subtree
  - Then compute
    \[
    W(i,j) = W(i,l-1) + a_l + W(l,j)
    \]
    \[
    P(i,j) = P(i,l-1) + W(i,j) + P(l,j)
    \]

- Done
Implementation

• There are only \((n+1)\times(n+1)\) different pairs \(i,j\)
• We need one two-dimensional matrix of size \((n+1)\times(n+1)\) for \(W\) and one for \(P\)
• Since \(j>i\), we actually only need half of each matrix
• Both matrixes are iteratively filled from the main diagonal to the upper-right corner

\[
\begin{array}{c}
\text{Matrix} 1 \quad \leftrightarrow \quad \text{Matrix 2} \quad \leftrightarrow \quad \ldots \quad \leftrightarrow \quad \text{Matrix 3}
\end{array}
\]
Constructing the tree

• We only showed how to compute the cost of the optimal tree, but not the tree itself
• But this is simple since we never revise decisions
• We can grow the tree whenever we have computed a new optimal root l
• For instance, we can define a r(i,j):=l in every step; these values fully determine the tree
Analysis

- **Space**
  - We need 2 arrays of size $O(n \times n)$
  - **Space complexity is $O(n^2)$**

- **Time**
  - We enumerate all breadths from 2 to n
  - For each breadth, we look at all possible start positions
  - In each range, we need to find the optimal $l$
  - A range has max size n-1
  - **Together: $O(n^3)$**
  - [Can be improved to $O(n^2)$]

1. Initialize $W(i,i+1)$;
2. Initialize $P(i,i+1)$;
3. for $b = 2$ to $n$ do
4.   for $i = 0$ to $(n-b)$ do
5.     $j := i+b$;
6.     find optimal $l$ in $[i,j]$;
7.     $W(i,j) := ...$
8.     $P(i,j) := ...$
9.   end for;
10. end for;
Relevance

- Nice and instructive
- But: $O(n^2)$ is much too expensive for any $n$ where such algorithms make sense
- Fortunately, we can compute "almost" optimal search trees in linear time
  - Not this lecture
Content of this Lecture

- Optimal Search Trees
- Construction of Optimal Search Trees
- Searching Strings: Tries
Keys that are Strings

- Assume $K$ is a set of strings of maximal length $m$
- We can build an AVL tree over $K$
- Searching requires $O(\log(n))$ comparisons
- Each string-comp requires $m$ char-comps in worst-case
  - We have seen that this is very pessimistic, but still
- Together: We need $O(|k| \cdot \log(n))$ character comparisons for searching a key $k$
- But:
  - Similar strings will be close neighbors in the tree
  - These will share prefixes (the longer, the more similar)
  - These are compared again and again
k="verhalten"

Diagramm mit Wörtern: 'verdauen', 'verkaufen', 'verbauen', 'verlaufen'.
Tries

- Tries are edge-labeled trees of order $|\Sigma|$
  - Developed for Information Retrieval
- Edges are labeled with characters from $\Sigma$
- Idea: Common prefixed of keys are represented only once
- Problem: Is “verl” a key?
  - Trick: Add a “$” (not in $\Sigma$) to every string
  - Then every and only leaves represent keys
Analysis

• Construction of a trie over $K$?
  – Let $\text{len}(K) = \Sigma |k| \ | k \in K$
  – We start with an empty tree and iteratively add all $k$s
  – Upon adding $k$, we simply match $k$ in the tree as long as possible
  – As soon as no continuation is found, we build a new branch
  – This requires $O(k)$ operations (char-comps or node creations)
  – It follows: Construction is in $O(\text{len}(K))$

• Searching a key $k$ (which maybe in or not in $K$)
  – We match $k$ from root down the tree
  – When $k$ is exhausted and we are in a leaf: $k \in K$
  – If no continuation is found or we end in an inner node: $k \notin K$
  – It follows: Searching is in $O(|k|)$
  – But ...
Space Complexity

- We have at most $\text{len}(K)$ edges and $\text{len}(K)+1$ nodes
  - Shared prefixes make the actual number smaller
- But we also need **pointer to children**
- We hold our search complexity, choosing the right pointer must be possible in $O(1)$
- This adds $(\text{len}(K)+1-|K|)*|\Sigma|$ pointers
- Impossible for any non-trivial alphabet
  - **Digital tries** are a popular data structure in coding theory
  - There, $|\Sigma|=2$, so the pointers don’t matter much
- Furthermore, most of the pointers will be null
  - Depending on $|\Sigma|$, $|K|$, and lengths of shared prefixes
Alternatives

- Advantage: $O(|k|)$ search
- Disadvantage: Excessive place consumption

- Advantage: $O(\text{len}(K))$ space
- Disadvantage: At least $O(|k|\times \log(|\Sigma|))$ search
Compressed Tries = Patricia Trees

- We can save further space
- A patricia tree is a trie where edges are labeled with strings, not with characters
- All sequences $S=\langle$node, edge$\rangle$ which do not branch are compressed into a single edge labeled with the concatenation of the labels in $S$
- More compact, less pointer
- Slightly more complicated implementation
  - E.g. insert requires splitting of labels