New Results on Binary Comparison Search Trees

Marek Chrobak, Neal Young
UC Riverside

Mordecai Golin
HKUST

Ian Munro
U Waterloo
Optimal search trees with 2-way comparisons
Marek Chrobak, Mordecai Golin, J. Ian Munro, Neal E. Young
arXiv:1505.00357
Main Result

Constructing Min-Cost Binary Comparison Search Trees
Main Result

Constructing Min-Cost Binary Comparison Search Trees

Wasn’t this completely understood 45 years ago??!!
Main Result

Constructing Min-Cost Binary Comparison Search Trees

Wasn’t this completely understood 45 years ago??!!

Yes and No …
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
Knuth’s Optimal BSTs
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees

• Known: $n$ keys $K_1, K_2, \ldots, K_n$. 
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees

• Known: \( n \) keys \( K_1, K_2, \ldots, K_n \).

• Preprocess keys to create binary tree. Tree query compares query value \( Q \) to keys and returns appropriate response from
  • \( i \) such that \( Q = K_i \)
  • \( i \) such that \( K_i < Q < K_{i+1} \)
  • \( Q < K_1 \) or \( K_n < Q \)
Knuth’s Optimal BSTs

- Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees
- Known: $n$ keys $K_1, K_2, \ldots, K_n$.
- Preprocess keys to create binary tree. Tree query compares query value $Q$ to keys and returns appropriate response from
  - $i$ such that $Q = K_i$
  - $i$ such that $K_i < Q < K_{i+1}$
  - $Q < K_1$ or $K_n < Q$
- Input: probability of successful and unsuccessful searches
  \[
  \beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n
  \]
  \[
  \beta_i = \Pr(Q = K_i) \quad \alpha_i = \Pr(K_i < Q < K_{i+1})
  \]
Knuth’s Optimal BSTs
Knuth’s Optimal BSTs

- $Q=A$?
  - $Q<A$, $\alpha_0$
  - $Q=B$, $\beta_2$:
    - $A<Q<B$, $\alpha_1$
    - $B<Q<C$, $\alpha_2$
  - $Q=C$, $\beta_3$:
    - $C<Q$, $\alpha_3$
- $Q=B$?
  - $Q<A$, $\alpha_1$
  - $Q<C$, $\beta_3$:
    - $A<Q<B$, $\alpha_1$
    - $B<Q<C$, $\alpha_2$
  - $Q=C$, $\beta_3$:
    - $C<Q$, $\alpha_3$
Knuth’s Optimal BSTs

\[ \beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n \]

\[ \beta_i = \Pr(Q = K_i) \quad \alpha_i = \Pr(K_i < Q < K_{i+1}) \]
Knuth’s Optimal BSTs
Knuth’s Optimal BSTs

- Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees

• Input was probability of successful and unsuccessful searches

$$\beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n$$
Knuth’s Optimal BSTs

- Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees
- Input was probability of successful and unsuccessful searches

\[
\beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n
\]

\[
\beta_i = \Pr(Q = K_i) \quad \alpha_i = \Pr(K_i < Q < K_{i+1})
\]
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees

• Input was probability of successful and unsuccessful searches

\[
\beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n
\]

\[
\beta_i = \Pr(Q = K_i) \quad \alpha_i = \Pr(K_i < Q < K_{i+1})
\]

• Cost of tree was average path length
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees

• Input was probability of successful and unsuccessful searches

\[ \beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n \]

\[ \beta_i = \Pr(Q = K_i) \quad \text{and} \quad \alpha_i = \Pr(K_i < Q < K_{i+1}) \]

• Cost of tree was average path length

\[ \sum_{i=1}^{n} \beta_i \text{depth}(\beta_i) + \sum_{i=0}^{n} \alpha_i \text{depth}(\alpha_i) \]
Knuth’s Optimal BSTs

• Knuth [1971] gave algorithm for constructing Optimal Binary Search Trees

• Input was probability of successful and unsuccessful searches

\[ \beta_1, \beta_2, \ldots, \beta_n \quad \text{and} \quad \alpha_0, \alpha_1, \ldots, \alpha_n \]

\[ \beta_i = \Pr(Q = K_i) \quad \alpha_i = \Pr(K_i < Q < K_{i+1}) \]

• Cost of tree was average path length

\[ \sum_{i=1}^{n} \beta_i \text{depth}(\beta_i) + \sum_{i=0}^{n} \alpha_i \text{depth}(\alpha_i) \]

• Dynamic Programming Algorithm

• Constructed O(n^2) DP table

• Knuth reduced O(n^3) running time to O(n^2)

• Technique later generalized as Quadrangle Inequality method by F. Yao
Knuth’s Optimal BSTs
Knuth’s Optimal BSTs
Knuth’s Optimal BSTs

\[(\alpha_0 + \beta_3) + 2(\beta_2 + \alpha_3) + 3(\alpha_1 + \alpha_2)\]
Knuth’s Optimal BSTs

\[(\alpha_0 + \beta_3) + 2(\beta_2 + \alpha_3) + 3(\alpha_1 + \alpha_2)\]  
\[(\beta_1 + \beta_3) + 2(\alpha_0 + \alpha_1 + \alpha_2 + \alpha_3)\]  

Cost = 0.85  
Cost = 1.10

\((\beta_1, \beta_2, \beta_3) = (0.5, 0.1, 0.2)\)  
\(\alpha_i \equiv 0.05\)
Knuth’s Optimal BSTs

\[(\alpha_0 + \beta_3) + 2(\beta_2 + \alpha_3) + 3(\alpha_1 + \alpha_2)\]

\[\beta_1 = .5, .1, .2\]
\[\alpha_i = .05\]

Cost = 0.85

\[(\beta_1 + \beta_3) + 2(\alpha_0 + \alpha_1 + \alpha_2 + \alpha_3)\]

\[(\beta_1, \beta_2, \beta_3) = (.3, .3, .3)\]

Cost = 1.05

\[(\beta_1, \beta_2, \beta_3) = (.3, .3, .3)\]

Cost = 0.80

\[(\beta_1, \beta_2, \beta_3) = (.3, .3, .3)\]
Hu-Tucker Binary Comparison Search Trees
Hu-Tucker Binary Comparison Search Trees

- Knuth constructed optimal binary search trees
Hu-Tucker Binary Comparison Search Trees

- Knuth constructed *optimal binary search trees*
- Trees structure was *binary* but nodes used *ternary* comparisons. Each node needed two binary comparisons to implement the search
Hu-Tucker Binary Comparison Search Trees

• Knuth constructed *optimal binary search trees*

• Trees structure was *binary* but nodes used *ternary* comparisons. Each node needed two binary comparisons to implement the search.

• In a *binary comparison search tree*, each internal node performs only one comparison. Searches all terminate at leaves.

• First such trees constructed by Hu-Tucker, also in 1971. \( O(n \log n) \)
Hu-Tucker Binary Comparison Search Trees
Hu-Tucker Binary Comparison Search Trees
Hu-Tucker Binary Comparison Search Trees


- Assumes all searches are successful; no failures allowed.
Input is only $\beta_1, \beta_2, \ldots, \beta_n$, with no $\alpha_i$'s.
Hu-Tucker Binary Comparison Search Trees

• Hu Tucker (1971) & Garsia-Wachs (1977)

• Assumes all searches are successful; no failures allowed. Input is only $\beta_1, \beta_2, \ldots, \beta_n$, with no $\alpha_i$s.

• Internal nodes are $<$ comparisons. Searches all terminate at leaves.
Hu-Tucker Binary Comparison Search Trees

- Assumes all searches are successful; no failures allowed. Input is only $\beta_1$, $\beta_2$, …, $\beta_n$, with no $\alpha_i$s.
- Internal nodes are $<$ comparisons. Searches all terminate at leaves.
- Problem is to find tree with *minimum weighted (average) external path length*.
Hu-Tucker Binary Comparison Search Trees

- **Hu Tucker (1971) & Garsia-Wachs (1977)**
- Assumes all searches are successful; no failures allowed. Input is only $\beta_1, \beta_2, \ldots, \beta_n$, with no $\alpha_i$s.
- Internal nodes are $<$ comparisons. Searches all terminate at leaves.
- Problem is to find tree with minimum weighted (average) external path length.
- $O(n \log n)$ algorithm.
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
Adding Equality Comparisons
Adding Equality Comparisons

The Knuth trees use three-way comparisons at each node. These are implemented in modern machines using two two-way comparisons (one $<$ and one $=$). Hu-Tucker trees use only one two-way comparison (a $<$) at each node.
Adding Equality Comparisons

The Knuth trees use three-way comparisons at each node. These are implemented in modern machines using two two-way comparisons (one < and one =). Hu-Tucker trees use only one two-way comparison (a <) at each node.

... machines that cannot make three-way comparisons at once. ... will have to make two comparisons. ... it may well be best to have a binary tree whose internal nodes specify either an equality test or a less-than test but not both.
Adding Equality Comparisons

The Knuth trees use three-way comparisons at each node. These are implemented in modern machines using two two-way comparisons (one < and one =). Hu-Tucker trees use only one two-way comparison (a <) at each node.

. . . machines that cannot make three-way comparisons at once. . . will have to make two comparisons. . . it may well be best to have a binary tree whose internal nodes specify either an equality test or a less-than test but not both.

Adding Equality Comparisons: AKKL[2001]
Adding Equality Comparisons: AKKL[2001]

• AKKL trees are min cost trees with more power.
  instead of being restricted to be $<$, comparisons can be $=$ OR $<$
Adding Equality Comparisons: AKKL[2001]

AKKL trees are min cost trees with more power. Instead of being restricted to be <, comparisons can be = OR <

AKKL trees include HT Trees
Adding Equality Comparisons: AKKL[2001]

• AKKL trees are min cost trees with more power. Instead of being restricted to be $<$, comparisons can be $=$ OR $<$
• AKKL trees include HT Trees
• AKKL trees can be cheaper than HT Trees if some $\beta_i$ much larger than others
Adding Equality Comparisons: AKKL[2001]

- AKKL trees are min cost trees with more power. Instead of being restricted to be $<$, comparisons can be $=$ OR $<$.
- AKKL trees include HT Trees.
- AKKL trees can be cheaper than HT Trees if some $\beta_i$ much larger than others.
- AKKL trees more difficult to construct.
Adding Equality Comparisons: AKKL[2001]
Adding Equality Comparisons: AKKL[2001]

- Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing \(=\) comparisons. AKKL find min-cost tree when the \(n-1\) internal node comparisons are allowed to be in \(\{=,\lt\}\).
Adding Equality Comparisons: AKKL[2001]

• Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing \(=\) comparisons. AKKL find min-cost tree when the \(n-1\) internal node comparisons are allowed to be in \(\{=,\leq\}\).

• Useful when some \(\beta_i\) are very large (relatively)
Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing \( = \) comparisons. AKKL find min-cost tree when the \( n-1 \) internal node comparisons are allowed to be in \( \{=,<\} \).

- Useful when some \( \beta_i \) are very large (relatively)

- AKKL algorithm runs in \( O(n^4) \) time.
Adding Equality Comparisons: AKKL[2001]

• Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing \(=\) comparisons. AKKL find min-cost tree when the \(n-1\) internal node comparisons are allowed to be in \(\{=,\leq\}\).

• Useful when some \(\beta_i\) are very large (relatively)

• AKKL algorithm runs in \(O(n^4)\) time.
  • AKKL note this improves running time of \(O(n^5)\) claimed by Spuler [1994] in his thesis
Adding Equality Comparisons: AKKL[2001]

- Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing $=$ comparisons. AKKL find min-cost tree when the $n-1$ internal node comparisons are allowed to be in $\{=,\leq\}$.

- Useful when some $\beta_i$ are very large (relatively)

- AKKL algorithm runs in $O(n^4)$ time.
  - AKKL note this improves running time of $O(n^5)$ claimed by Spuler [1994] in his thesis
  - Spuler only states $O(n^5)$ algorithm but doesn’t prove that it produces optimal tree, so AKKL is really first polynomial time algorithm
Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing $\leq$ comparisons. AKKL find min-cost tree when the $n-1$ internal node comparisons are allowed to be in $\{=,\leq\}$.

- Useful when some $\beta_i$ are very large (relatively)

- AKKL algorithm runs in $O(n^4)$ time.
  - AKKL note this improves running time of $O(n^5)$ claimed by Spuler [1994] in his thesis
  - Spuler only states $O(n^5)$ algorithm but doesn’t prove that it produces optimal tree, so AKKL is really first polynomial time algorithm

- Reason problem is difficult is that equality nodes can create holes in ranges. This could dramatically (exponentially?) increase search space, destroying DP approach
Adding Equality Comparisons: AKKL[2001]

- Anderson, Kannan, Karloff, Ladner [2002] extended Hu-Tucker by allowing $\equiv$ comparisons. AKKL find min-cost tree when the $n-1$ internal node comparisons are allowed to be in $\{\equiv, <\}$.

- Useful when some $\beta_i$ are very large (relatively)

- AKKL algorithm runs in $O(n^4)$ time.
  - AKKL note this improves running time of $O(n^5)$ claimed by Spuler [1994] in his thesis
  - Spuler only states $O(n^5)$ algorithm but doesn’t prove that it produces optimal tree, so AKKL is really first polynomial time algorithm

- Reason problem is difficult is that equality nodes can create holes in ranges. This could dramatically (exponentially?) increase search space, destroying DP approach
  - AKKL show that if equality comparison exists, then it is always largest probability in range. Allows recovering DP approach with ranges of description size $O(n^3)$ (compared to Knuth’s $O(n^2)$)
Adding Equality Comparisons: AKKL[2001]

• **Comment 1**: Other problem in AKKL is how to deal with repeated weights. This was hardest part.

• **Comment 2**: Both Hu-Tucker and AKKL only work when failures don’t occur. I.e., only $\beta_i$ are allowed and not $\alpha_i$. 
So Far + Obvious Open Problem
So Far + Obvious Open Problem

• Optimal Binary Search Trees
  • Input: \( \beta_i = \Pr(Q = K_i); \ \alpha_i = \Pr(K_{i-1} < Q < K_i) \)
  • \( O(n^2) \) Knuth

• Optimal Binary Comparison Search Trees
  • Input: \( \beta_i = \Pr(Q = K_i); \ \) failures not allowed
  • \( C = \{<\}: \ O(n \log n) \) Hu-Tucker & Garsia-Wachs
  • \( C = \{=,<\}: \ O(n^4) \) AKKL
So Far + Obvious Open Problem

• Optimal Binary Search Trees
  • Input: \( \beta_i = \Pr(Q = K_i); \alpha_i = \Pr(K_{i-1} < Q < K_i) \)
  • \( O(n^2) \) Knuth

• Optimal Binary Comparison Search Trees
  • Input: \( \beta_i = \Pr(Q = K_i); \) failures not allowed
  • \( C = \{<\}: \quad O(n \log n) \) Hu-Tucker & Garsia-Wachs
  • \( C = \{=,<\}: \quad O(n^4) \) AKKL

• Obvious Questions
  • Can we build OBCSTs that allow failures?
    • If yes, for which sets of comparisons?
  • Answer is yes, (for all sets of comparisons) but first need to define problem models
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
BCSTs with Failure Probabilities
BCSTs with Failure Probabilities

- Allows Failures ($\beta_i$ and $\alpha_i$).
- Call this complete input. HT has restricted input.
• Allows Failures ($\beta_i$ and $\alpha_i$).
  • Call this complete input. HT has restricted input.

• Tree for $n$ keys has $2n+1$ leaves
BCSTs with Failure Probabilities

• Allows Failures ($\beta_i$ and $\alpha_i$).
  • Call this complete input. HT has restricted input.

• Tree for $n$ keys has $2n+1$ leaves
• Distinguishing between $Q=K_i$ and $K_i < Q < K_{i+1}$ always requires querying ($Q=K_i$)
Using Different Types of Comparisons

Q = D

Q ≤ C

Q = B

Q = A

Q = C

Q = D

Q < C

Q < A

Q < B

A < Q < B

B < Q < C

C < Q < D

D

D < Q
Using Different Types of Comparisons

- Left Tree uses \{<,=\}. Right Tree uses \{<,\leq,=\}
- Minimum cost BCST is minimum taken over all trees using given set of comparisons \(C\), e.g., \(C=\{<,=\}\) or \(C=\{<,\leq,=\}\)
Using Different Types of Comparisons

• Left Tree uses \(<,=\). Right Tree uses \(<, \leq, =\)
  • Minimum cost BCST is minimum taken over all trees using given set of comparisons \(C\), e.g., \(C=\{<,=\}\) or \(C=\{<, \leq, =\}\)

• \(C\) is input to the problem.
  • Algorithm is different for different \(Cs\).
How Much Information is Needed for Failure?
How Much Information is Needed for Failure?

• Tree on left shows **Explicit Failure**
  • every failure leaf reports unique failure interval, $K_i < Q < K_{i+1}$.
• Tree on left shows **Explicit Failure**
  • every failure leaf reports unique failure interval, $K_i < Q < K_{i+1}$.

• Tree on right shows **Non-Explicit Failure**:
  • Failure leaves only report failure. Don’t need to specify exact interval. Leaf can be concatenation of successive failure intervals.
New Algorithms: OBCSTs with Failures

<table>
<thead>
<tr>
<th>Permitted Comparisons</th>
<th>Failure Type</th>
<th>Time</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = {=}$</td>
<td>Explicit</td>
<td>$—$</td>
<td>Can not occur</td>
</tr>
<tr>
<td></td>
<td>Non-Explicit</td>
<td>$O(n \log n)$</td>
<td>Trivial. Similar to Linked List</td>
</tr>
<tr>
<td>$C = {&lt;,\leq}$</td>
<td>Explicit</td>
<td>$O(n \log n)$</td>
<td>$O(n)$ Reduction to Hu-Tucker</td>
</tr>
<tr>
<td></td>
<td>Non-Explicit</td>
<td>$—$</td>
<td>Can not occur</td>
</tr>
<tr>
<td>$C = {=,&lt;}, C = {=,\leq}$</td>
<td>Explicit</td>
<td>$O(n^4)$</td>
<td>Follows from Main Lemma</td>
</tr>
<tr>
<td></td>
<td>Non-Explicit</td>
<td>$O(n^4)$</td>
<td>”</td>
</tr>
<tr>
<td>$C = {=,&lt;,\leq}$</td>
<td>Explicit</td>
<td>$O(n^4)$</td>
<td>”</td>
</tr>
<tr>
<td></td>
<td>Non-Explicit</td>
<td>$O(n^4)$</td>
<td>”</td>
</tr>
</tbody>
</table>

- DP Algorithms for last 4 cases are very similar
- Differ slightly in
  - Design of Recurrence Relations
    - $\{=,<\}$ and $\{=,<,\leq\}$ yield slightly different recurrences
  - Initial conditions
    - Explicit and Non-Explicit Failures force different I.C.s
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
Main Lemma:

Lemma
Let T be a Optimal BCST.
If \((Q=K_k)\) is a Descendant of \((Q=K_i)\)
Then \(\beta_k \leq \beta_i\)
Main Lemma:

**Lemma**
Let T be a Optimal BCST. If \((Q=K_k)\) is a Descendant of \((Q=K_i)\) Then \(\beta_k \leq \beta_i\)

*Note: This is true regardless of which inequality comparisons are used and which model BCST is used*
Main Lemma:

**Lemma**
Let $T$ be a Optimal BCST. If $(Q=K_k)$ is a Descendant of $(Q=K_i)$ Then $\beta_k \leq \beta_i$

*Note: This is true regardless of which inequality comparisons are used and which model BCST is used*

**Corollary:** If $T$ is an OBCST and $(Q=K_k)$ an internal node in $T$, then $\beta_k \leq \beta_j$ for all $(Q=K_j)$ on the path from the root to $(Q=K_k)$, i.e., equality weights decrease walking down the tree.
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
Structural Properties of BCSTs
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.
Also assume $C=\{<,=\}$
Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.

Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$.
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different
Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$
• Root of BSCT is search range $[K_0, K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different. Also assume $C=\{<,=\}$.

Every tree node $N$ corresponds to search range of subtree rooted at $N$:

- Root of BSCT is search range $[K_0, K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
- Comparisons cuts ranges:
  - $A (Q<K_i)$ splits $[K_i, K_j)$ into $[K_i, K_k)$ and $[K_k, K_i)$
  - $A (Q=K_i)$ removing $K_i$ from range,

### Structural Properties of BCSTs
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different
Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$

- Root of BSCT is search range $[K_0,K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
- Comparisons cuts ranges
  - A ($Q<K_i$) splits $[K_i,K_j]$ into $[K_i,K_k]$ and $[K_k,K_i]$  
  - A ($Q=K_i$) removing $K_i$ from range,
- Range of subtree rooted at $N$ is some $[K_i,K_j]$ with some keys removed

![Diagram of BCST](image.png)
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.
Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$:

- Root of BSCT is search range $[K_0,K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
- Comparisons cut ranges
  - A ($Q<K_i$) splits $[K_i,K_j)$ into $[K_i,K_k)$ and $[K_k,K_i)$
  - A ($Q=K_i$) removing $K_i$ from range,
- Range of subtree rooted at $N$ is some $[K_i,K_j)$ with some keys removed
- Keys removed (holes) are $K_k$ s.t. ($Q=K_k$) is on the path from $N$ to the root of $T$. 
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different. Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$:

- Root of BSCT is search range $[K_0, K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
- Comparisons cuts ranges
  - $A (Q<K_i)$ splits $[K_i, K_j)$ into $[K_i, K_k)$ and $[K_k, K_i)$
  - $A (Q=K_i)$ removing $K_i$ from range,
- Range of subtree rooted at $N$ is some $[K_i, K_j)$ with some keys removed
- Keys removed (holes) are $K_k$ s.t. $(Q=K_k)$ is on the path from $N$ to the root of $T$. 
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.

Also assume $C = \{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$:

- Root of BSCT is search range $[K_0, K_{n+1})$ (where $K_0 = -\infty$ and $K_{n+1} = \infty$)

- Comparisons cuts ranges:
  - A ($Q < K_i$) splits $[K_i, K_j)$ into $[K_i, K_k)$ and $[K_k, K_i)$
  - A ($Q = K_i$) removing $K_i$ from range,

- Range of subtree rooted at $N$ is some $[K_i, K_j)$ with some keys removed

- Keys removed (holes) are $K_k$ s.t. $(Q = K_k)$ is on the path from $N$ to the root of $T$. 

\[ \begin{align*}
\beta_1 & \quad \beta_2 \\
\gamma & \quad \gamma \\
\delta & \quad \delta
\end{align*} \]
Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.

Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$

- Root of BSCT is search range $[K_0,K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)

- Comparisons cuts ranges
  - A ($Q<K_i$) splits $[K_i,K_j)$ into $[K_i,K_k)$ and $[K_k,K_i)$
  - A ($Q=K_i$) removing $K_i$ from range,

- Range of subtree rooted at $N$ is some $[K_i,K_j)$ with some keys removed

- Keys removed (holes) are $K_k$ s.t. ($Q=K_k$) is on the path from $N$ to the root of $T$. 

### Structural Properties of BCSTs

#### Tree Representation

![Tree Diagram]

- $Q=\beta$  
- $Q=A$ 
- $Q=C$ 
- $Q=D$

- $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4$

- $\beta_1, \beta_2$

- $[-\infty, -\infty)$  
- $[-\infty, C)$  
- $[C, \infty)$  
- $[\infty, \infty)$

- $Q<A$  
- $Q=B$  
- $Q=C$  
- $Q<D$  
- $Q=B$  
- $Q=C$  
- $Q=D$  
- $Q=A$
**Structural Properties of BCSTs**

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.

Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$:

- Root of BSCT is search range $[K_0,K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
- Comparisons cut ranges
  - $A (Q<K_i)$ splits $[K_i,K_j)$ into $[K_i,K_k)$ and $[K_k,K_i)$
  - $A (Q=K_i)$ removing $K_i$ from range,
- Range of subtree rooted at $N$ is some $[K_i,K_j)$ with some keys removed.
- Keys removed (holes) are $K_k$ s.t. $(Q=K_k)$ is on the path from $N$ to the root of $T$. 

```
[\infty,\infty)
[\infty,\beta_2)
[\infty,\xi_3)
[\infty,\beta_4)
[\infty,\beta_5)
[\infty,\alpha_1)
[\infty,\alpha_2)
```
**Structural Properties of BCSTs**

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different.

Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$:

- Root of BSCT is search range $[K_0, K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)

- Comparisons cut ranges:
  - A ($Q<K_i$) splits $[K_i, K_j]$ into $[K_i, K_k]$ and $[K_k, K_j]$
  - A ($Q=K_i$) removing $K_i$ from range,

- Range of subtree rooted at $N$ is some $[K_i, K_j]$ with some keys removed

- Keys removed (holes) are $K_k$ s.t. ($Q=K_k$) is on the path from $N$ to the root of $T$. 
Structural Properties of BCSTs

Henceforth assume distinct key weights, i.e., all of the $\beta_1, \beta_2, \ldots, \beta_n$ are different
Also assume $C=\{<,=\}$

Every tree node $N$ corresponds to search range of subtree rooted at $N$

- Root of BSCT is search range $[K_0,K_{n+1})$ (where $K_0=-\infty$ and $K_{n+1}=\infty$)
- Comparisons cuts ranges
  - $A$ ($Q<K_i$) splits $[K_i,K_j)$ into $[K_i,K_k)$ and $[K_k,K_i)$
  - $A$ ($Q=K_i$) removing $K_i$ from range,
- Range of subtree rooted at $N$ is some $[K_i,K_j)$ with some keys removed
- Keys removed (holes) are $K_k$ s.t. ($Q=K_k$) is on the path from $N$ to the root of $T.$
Structural Properties of OBCSTs
Structural Properties of OBCSTs

- Range associated with Node N is $[K_i,K_j]$ with some (h) keys $K_k$ removed.

- $K_k$ removed are s.t. $(Q=K_k)$ are equality nodes on path from N to root (that fall within $[K_i,K_j]$)
Structural Properties of OBCSTs

• Range associated with Node N is 
  \([K_i,K_j]\) with some (h) keys \(K_k\) removed.

• \(K_k\) removed are s.t. \((Q=K_k)\) are equality nodes 
on path from N to root (that fall within \([K_i,K_j]\))

• From previous Lemma, if T is an OBCST, \(\beta_i\) of nodes 
  path to N are larger than \(\beta_i\) of all equality nodes in T’.

• \(\forall k, (Q=K_k)\) appears somewhere in T. 
  Immediately implies that the h missing keys must be 
  the largest weighted keys in \([K_i,K_j]\)
Structural Properties of OBCSTs

• Range associated with Node N is $[K_i, K_j]$ with some (h) keys $K_k$ removed.

• $K_k$ removed are s.t. $(Q = K_k)$ are equality nodes on path from N to root (that fall within $[K_i, K_j]$)

• From previous Lemma, if T is an OBCST, $\beta_i$ of nodes path to N are larger than $\beta_i$ of all equality nodes in $T'$.

• $\forall k, (Q = K_k)$ appears somewhere in $T$.
  Immediately implies that the h missing keys must be the largest weighted keys in $[K_i, K_j]$

• Define punctured range $[i, j: h]$ to be range $[K_i, K_j]$ with the h highest weighted keys in $[K_i, K_j]$ removed
Structural Properties of OBCSTs

• Range associated with Node N is \([K_i, K_j]\) with some (h) keys \(K_k\) removed.

• \(K_k\) removed are s.t. \((Q=K_k)\) are equality nodes on path from N to root (that fall within \([K_i, K_j]\))

• From previous Lemma, if T is an OBCST, \(\beta_i\) of nodes path to N are larger than \(\beta_i\) of all equality nodes in \(T'\).

• \(\forall k, (Q=K_k)\) appears somewhere in T. Immediately implies that the h missing keys must be the largest weighted keys in \([K_i, K_j]\)

• Define punctured range \([i, j; h]\) to be range \([K_i, K_j]\) with the h highest weighted keys in \([K_i, K_j]\) removed.

• \(\Rightarrow\) every range associated with an internal node of an OBCST is a punctured range
Structural Properties of OBCSTs
Structural Properties of OBCSTs

• \([i,j; h]\) is range \([K_i, K_j]\) with the \(h\) highest weighted keys in \([K_i, K_j]\) removed

• Range associated with an internal node of an OBCST is some \([i,j; h]\)
Structural Properties of OBCSTs

• \([i,j: h]\) is range \([K_i, K_j]\) with the \(h\) highest weighted keys in \([K_i, K_j]\) removed

• Range associated with an internal node of an OBCST is some \([i,j: h]\)

• Define \(\text{OPT}(i,j: h)\) to be the cost of an optimal BCST for range \([i,j: h]\)

• Goal is to find \(\text{OPT}(0,n+1: 0)\) and associated tree

• Will use Dynamic programming to fill in table. Table has size \(O(n^3)\)
  We will (recursively) evaluate \(\text{OPT}(i,j: h)\) in \(O(j-i)\) time, yielding a \(O(n^4)\) algorithm.
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
    • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
Dynamic programming for OBCSTs
Dynamic programming for OBCSTs

• Let T be an OBCST for \([i,j: h]\)
• T has two possible structures
Dynamic programming for OBCSTs

• Let T be an OBCST for \([i,j: h])
• T Has two possible structures

1. Root is a \((Q=K_k)\)
Dynamic programming for OBCSTs

• Let T be an OBCST for \([i,j: h]\)
• T Has two possible structures

1. Root is a (\(Q=K_k\))

2. Root is a (\(Q<K_k\))
Dynamic programing for OBCSTs

1. Root of OPT(i,j: h) is a (Q=K_k)
Dynamic programming for OBCSTs

1. Root of OPT(i,j: h) is a \( Q=K_k \)

- \( K_k \) must be largest key weight in \( [i,j: h) \)
  which is \((h+1)\)th largest key weight in \( [i,j) \)

- Right subtree missing \( h+1 \) largest weights in \( [i,j) \) so right subtree is \( \text{OPT}(i,j: h+1) \)
Dynamic programming for OBCSTs

1. Root of $\text{OPT}(i,j: h)$ is a $(Q=K_k)$
   - $K_k$ must be largest key weight in $[i,j: h)$ which is $(h+1)^{st}$ largest key weight in $[i,j)$
   - Right subtree missing $h+1$ largest weights in $[i,j)$ so right subtree is $\text{OPT}(i,j: h+1)$

Cost of full tree is sum of
- cost of left subtree $\ 0$
- cost of right subtree $\ \text{OPT}(i,j: h+1)$
- Total weight of left + right subtree $W_{i,j:h}$ where $W_{i,j:h} = \text{sum of all } \beta_i, \alpha_i$ in $(i,j: h)$
Dynamic programming for OBCSTs

1. Root of $\text{OPT}(i,j: h)$ is a $(Q=K_k)$
   - $K_k$ must be largest key weight in $[i,j: h)$ which is $(h+1)^{st}$ largest key weight in $[i,j)$
   - Right subtree missing $h+1$ largest weights in $[i,j)$ so right subtree is $\text{OPT}(i,j: h+1)$

Cost of full tree is sum of
   - cost of left subtree $\ 0$
   - cost of right subtree $\ \text{OPT}(i,j: h+1)$
   - Total weight of left + right subtree $W_{i,j:h}$
     where $W_{i,j:h} = \text{sum of all } \beta_i, \alpha_i \text{ in } (i,j: h]$ 

$$EQ(i, j : h) = W_{i,j:h} + \text{OPT}(i, j : h + 1)$$
Dynamic programming for OBCSTs

2. Root of $\text{OPT}(i,j: h)$ is a $(Q<K_k)$
Dynamic programing for OBCSTs

2. Root of $\text{OPT}(i,j; h)$ is a $(Q<K_k)$

- Range is split into $<k$ and $\geq k$
- $h$ holes (largest keys) in $[i,j)$ are split, with $h_1(k)$ on left and $h_2(k) = h - h_1(k)$ on right
Dynamic programming for OBCSTs

2. Root of $\text{OPT}(i,j: h)$ is a $(Q<K_k)$

- Range is split into $<k$ and $\geq k$
- $h$ holes (largest keys) in $[i,j)$ are split, with $h_1(k)$ on left and $h_2(k) = h - h_1(k)$ on right
- $h_1(k)$ keys must be heaviest in $[i,k)$
  $h_2(k)$ keys must be heaviest in $[k,j)$
- So left and right subtrees are OBCSTs for $[i,k: h_1(k))$ and $[k,j: h_2(k))$
2. Root of $OPT(i,j: h)$ is a $(Q<k_k)$

- Range is split into $<k$ and $\geq k$
- $h$ holes (largest keys) in $[i,j)$ are split, with $h_1(k)$ on left and $h_2(k) = h - h_1(k)$ on right
- $h_1(k)$ keys must be heaviest in $[i,k)$
  $h_2(k)$ keys must be heaviest in $[k,j)$
- So left and right subtrees are OBCSTs for $[i,k: h_1(k))$ and $[k,j: h_2(k))$
- Cost of tree is $W_{i,j:h} + OPT(i,k: h_1(k)) + OPT(k,j: h_2(k))$
Dynamic programing for OBCSTs

2. Root of OPT\( (i,j: h) \) is a (Q<K\(_k\))

- Range is split into <\( k \) and \( \geq k \)
- \( h \) holes (largest keys) in \([i,j)\) are split, with \( h_1(k) \) on left and \( h_2(k) = h-h_1(k) \) on right
- \( h_1(k) \) keys must be heaviest in \([i,k)\)
- \( h_2(k) \) keys must be heaviest in \([k,j)\)
- So left and right subtrees are OBCSTs for \([i,k: h_1(k))\) and \([k,j: h_2(k))\)
- Cost of tree is \( W_{i,j:h} + OPT(i,k: h_1(k)) + OPT(k,j: h_2(k)) \)

Don’t know what \( k \) is, so minimize over all possible \( k \)

\[
SPLIT(i, j : h) = \min_{i < k < j} \left\{ W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \right\}
\]
Dynamic programing for OBCSTs
Dynamic programing for OBCSTs

OPT(i,j: h) has two possible structures
Dynamic programming for OBCSTs

OPT(i,j: h) has two possible structures

1. Root is a (Q=K_k)

2. Root is a (Q<K_k)
Dynamic programing for OBCSTs

OPT(i,j: h) has two possible structures

1. Root is a (Q=K_k)

   \[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

2. Root is a (Q<K_k)

   \[ SPLIT(i, j : h) = \min_{i<k<j} \{ W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \} \]
Dynamic programming for OBCSTs

OPT(i,j: h) has two possible structures

1. Root is a (Q=K_k)
   \[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

2. Root is a (Q<K_k)
   \[ SPLIT(i, j : h) = \min_{i<k<j} \{ W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \} \]

This immediately implies
\[ OPT(i, j : h) \geq \min (EQ(i, j : h), SPLIT(i, j : h)) \]
Dynamic programing for OBCSTs

OPT(i,j: h) has two possible structures

1. Root is a (Q=K_k)
   \[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

2. Root is a (Q<K_k)
   \[ SPLIT(i, j : h) = \min_{i<k<j} \{W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k))\} \]

This immediately implies
\[ OPT(i, j : h) \geq \min (EQ(i, j : h), SPLIT(i, j : h)) \]

But every case seen can construct a BCST with that cost, so
\[ OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h)) \]
Dynamic programing for OBCSTs

$$OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h))$$

$$EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1)$$

$$SPLIT(i, j : h) = \min_{i < k < j} \{W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k))\}$$
Dynamic programing for OBCSTs

\[ OPT(i, j : h) = \min \left( EQ(i, j : h), \ SPLIT(i, j : h) \right) \]

\[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i<k<j} \left\{ W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \right\} \]

Set initial conditions for ranges \( OPT(i,i+1,*) \)
Dynamic programing for OBCSTs

\[ OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h)) \]

\[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i<k<j} \{ W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \} \]

Set initial conditions for ranges \( OPT(i,i+1,*)) \)

\[ OPT(i,i+1,1)=0 \]

\[ OPT(i,i+1,0)= \beta_i + \alpha_i \]

\[ OPT(i,i+1,1)=0 \quad \alpha_i \]

\[ OPT(i,i+1,0)= \beta_i + \alpha_i \]

\[ Q=K_i \]

\[ K_i<Q<K_{i+1} \]

\[ \beta_i \]

\[ K_i=Q \]

\[ K_{i}<Q<K_{i+1} \]

\[ \alpha_i \]
Dynamic programing for OBCSTs

\[ OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h)) \]

\[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i < k < j} \{ W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \} \]

Set initial conditions for ranges OPT(i,i+1,*)

\[ OPT(i,i+1,1) = 0 \quad \text{for} \quad K_i < Q < K_{i+1} \]

\[ OPT(i,i+1,0) = \beta_i + \alpha_i \]

Comments
Dynamic programming for OBCSTs

\[ OPT(i, j : h) = \min \left( EQ(i, j : h), \ SPLIT(i, j : h) \right) \]

\[ EQ(i, j : h) = W_{i:j:h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i < k < j} \{ W_{i:j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \} \]

Set initial conditions for ranges \( OPT(i,i+1,*) \)

\[ OPT(i,i+1,1) = 0 \quad \alpha_i \quad OPT(i,i+1,0) = \beta_i + \alpha_i \]

Comments
- Must restrict \( h \leq j-i \) (can’t have more holes than keys in interval)
Dynamic programing for OBCSTs

\[ OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h)) \]

\[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i < k < j} \{W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k))\} \]

Set initial conditions for ranges \( OPT(i,i+1,*) \)

\( OPT(i,i+1,1) = 0 \)

\( OPT(i,i+1,0) = \beta_i + \alpha_i \)

Comments

- Must restrict \( h \leq j-i \) (can’t have more holes than keys in interval)
- Need to fill in table in proper order, e.g.,
  (a) \( d= 0 \) to \( n \),
  (b) \( i=0 \) to \( n-d \), \( j=i+d+1 \),
  (c) \( h = (j-i) \) downto \( 0 \)
Dynamic programing for OBCSTs

\[ OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h)) \]

\[ EQ(i, j : h) = W_{i,j:h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i < k < j} \{W_{i,j:h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k))\} \]

Set initial conditions for ranges \( OPT(i,i+1,*) \)

\[ OPT(i,i+1,1)=0 \quad K_i < Q < K_{i+1} \quad \alpha_i \quad OPT(i,i+1,0)= \beta_i + \alpha_i \]

Comments

- Must restrict \( h \leq j-i \) (can’t have more holes than keys in interval)
- Need to fill in table in proper order, e.g.,
  (a) \( d= 0 \) to \( n \), \hfill (b) \( i=0 \) to \( n-d \), \( j=i+d+1 \), \hfill (c) \( h = (j-i) \) downto \( 0 \)
- Need \( O(1) \) method for computing \( h_i(k) \)
  - \( \Rightarrow O(j-i) \) to calculate \( OPT(i,j:h) \)
  - \( \Rightarrow O(n^4) \) to fill in complete table
Dynamic programing for OBCSTs

\[ OPT(i, j : h) = \min (EQ(i, j : h), SPLIT(i, j : h)) \]

\[ EQ(i, j : h) = W_{i,j : h} + OPT(i, j : h + 1) \]

\[ SPLIT(i, j : h) = \min_{i<k<j} \left\{ W_{i,j : h} + OPT(i, k : h_1(k)) + OPT(k, j : h_2(k)) \right\} \]

Set initial conditions for ranges \( OPT(i,i+1,*) \)

\[ OPT(i,i+1,1) = 0 \]

\[ OPT(i,i+1,0) = \beta_i + \alpha_i \]

Comments

- Must restrict \( h \leq j-i \) (can’t have more holes than keys in interval)
- Need to fill in table in proper order, e.g.,
  (a) \( d = 0 \) to \( n \),  
  (b) \( i = 0 \) to \( n-d \),  \( j = i+d+1 \),  (c) \( h = (j-i) \) downto 0
- Need \( O(1) \) method for computing \( h_i(k) \)
  \( \Rightarrow O(j-i) \) to calculate \( OPT(i,j : h) \)
  \( \Rightarrow O(n^4) \) to fill in complete table
- \( OPT(0,n+1:0) \) is optimal cost. Use standard DP backtracking to construct corresponding optimal tree
Perturbing for Key Weight Uniqueness (I)
Perturbing for Key Weight Uniqueness (I)

• Strongly used assumption $\beta_i$ are all distinct to find `weightiest’ keys
• Assumption can be removed using perturbation argument
Perturbing for Key Weight Uniqueness (I)

• Strongly used assumption $\beta_i$ are all distinct to find ‘weightiest’ keys
  • Assumption can be removed using perturbation argument

• All values constructed/compared in algorithm are subtree costs
  • in form $\sum a_i \alpha_i + \sum b_i \beta_i$ where $0 \leq a_i, b_i \leq 2n$ are integral node depths
Perturbing for Key Weight Uniqueness (I)

• Strongly used assumption $\beta_i$ are all distinct to find `weightiest’ keys
  • Assumption can be removed using perturbation argument

• All values constructed/compared in algorithm are subtree costs
  • in form $\sum a_i\alpha_i + \sum b_i\beta_i$ where $0 \leq a_i, b_i \leq 2n$ are integral node depths

• Perturb input by setting $\alpha’_i=\alpha_i$, $\beta’_i = \beta_i + i\epsilon$ where $\epsilon$ is very small
  • $\Rightarrow \beta’_i$ are all distinct
Perturbing for Key Weight Uniqueness (I)

- Strongly used assumption \( \beta_i \) are all distinct to find `weightiest’ keys
  - Assumption can be removed using perturbation argument

- All values constructed/compared in algorithm are subtree costs
  - in form \( \sum a_i\alpha_i + \sum b_i\beta_i \) where \( 0 \leq a_i, b_i \leq 2n \) are integral node depths

- Perturb input by setting \( \alpha_i' = \alpha_i, \beta_i' = \beta_i + i\epsilon \) where \( \epsilon \) is very small
  - \( \Rightarrow \beta_i' \) are all distinct

- Since \( \beta_i' \) are all distinct, algorithm gives correct result for \( \alpha_i', \beta_i' \)
  - Easy to prove that optimum tree for \( \alpha_i', \beta_i' \) is optimum for \( \alpha_i, \beta_i \)
  - \( \Rightarrow \) resulting tree is optimum for original \( \alpha_i', \beta_i' \)
Perturbing for Key Weight Uniqueness (I)

- Strongly used assumption $\beta_i$ are all distinct to find `weightiest’ keys
  - Assumption can be removed using perturbation argument

- All values constructed/compared in algorithm are subtree costs
  - in form $\sum a_i \alpha_i + \sum b_i \beta_i$ where $0 \leq a_i, b_i \leq 2n$ are integral node depths

- Perturb input by setting $\alpha'_i = \alpha_i$, $\beta'_i = \beta_i + i\epsilon$ where $\epsilon$ is very small
  - $\Rightarrow$ $\beta'_i$ are all distinct

- Since $\beta'_i$ are all distinct, algorithm gives correct result for $\alpha'_i, \beta'_i$
  - Easy to prove that optimum tree for $\alpha'_i, \beta'_i$ is optimum for $\alpha_i, \beta_i$
  - $\Rightarrow$ resulting tree is optimum for original $\alpha'_i, \beta'_i$

- In fact don’t actually need to know value of $\epsilon$
Perturbing for Key Weight Uniqueness (II)
Perturbing for Key Weight Uniqueness (II)

- Perturb input: $\alpha'_i = \alpha_i$, $\beta'_i = \beta_i + i\epsilon$ where $\epsilon$ is very small
- Need to find optimum tree for $\alpha'_i, \beta'_i$ (which is also optimum for $\alpha'_i, \beta'_i$)
Perturbing for Key Weight Uniqueness (II)

- Perturb input: $\alpha'_i = \alpha_i$, $\beta'_i = \beta_i + i\epsilon$ where $\epsilon$ is very small
- Need to find optimum tree for $\alpha'_i, \beta'_i$ (which is also optimum for $\alpha'_i, \beta'_i$)

- Recall that algorithm only performs additions/comparisons
  - All values are subtree costs $\sum a_i \alpha_i + \sum b_i \beta_i$ where $0 \leq a_i, b_i \leq 2n$ are integral
Perturbing for Key Weight Uniqueness (II)

- Perturb input: $\alpha'_i = \alpha_i$, $\beta'_i = \beta_i + i \epsilon$ where $\epsilon$ is very small
- Need to find optimum tree for $\alpha'_i, \beta'_i$ (which is also optimum for $\alpha'_i, \beta'_i$)

- Recall that algorithm only performs additions/comparisons
  - All values are subtree costs $\sum a_i \alpha_i + \sum b_i \beta_i$ where $0 \leq a_i, b_i \leq 2n$ are integral
- Don’t actually need to know or store value of $\epsilon$
Perturbing for Key Weight Uniqueness (II)

- Perturb input: \( \alpha'_i = \alpha_i, \beta'_i = \beta_i + \epsilon \) where \( \epsilon \) is very small
- Need to find optimum tree for \( \alpha'_i, \beta'_i \) (which is also optimum for \( \alpha'_i, \beta'_i \))

- Recall that algorithm only performs additions/comparisons
  - All values are subtree costs \( \sum a_i \alpha_i + \sum b_i \beta_i \) where \( 0 \leq a_i, b_i \leq 2n \) are integral
  - Don’t actually need to know or store value of \( \epsilon \)
  - Every value in algorithm is in form \( x = x_1 + x_2 \epsilon \), where \( x_2 = O(n^3) \) is an integer
  - Forget \( \epsilon \). Store pair \((x_1, x_2)\)
Perturbing for Key Weight Uniqueness (II)

- Perturb input: \( \alpha'_i = \alpha_i \), \( \beta'_i = \beta_i + i\epsilon \) where \( \epsilon \) is very small
- Need to find optimum tree for \(\alpha'_i, \beta'_i\) (which is also optimum for \(\alpha'_i, \beta'_i\))

- Recall that algorithm only performs additions/comparisons
  - All values are subtree costs \( \sum a_i\alpha_i + \sum b_i\beta_i \) where \( 0 \leq a_i, b_i \leq 2n \) are integral

- Don’t actually need to know or store value of \( \epsilon \)

- Every value in algorithm is in form \( x = x_1 + x_2\epsilon \), where \( x_2 = O(n^3) \) is an integer
  - Forget \( \epsilon \). Store pair \((x_1, x_2)\)

- (A) Addition is pairwise-addition
  - \((x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)\)

- (C) Comparison is lexicographic-comparison
  - \((x_1, x_2) < (y_1, y_2) \) iff \( x_1 < y_1 \) or \( x_1 = y_1 \) and \( x_2 = y_2 \)
Perturbing for Key Weight Uniqueness (II)

• Perturb input: $\alpha'_i = \alpha_i$, $\beta'_i = \beta_i + i\epsilon$ where $\epsilon$ is very small
• Need to find optimum tree for $\alpha'_i, \beta'_i$ (which is also optimum for $\alpha'_i, \beta'_i$)

• Recall that algorithm only performs additions/comparisons
• All values are subtree costs $\sum a_i\alpha_i + \sum b_i\beta_i$ where $0 \leq a_i, b_i \leq 2n$ are integral

• Don’t actually need to know or store value of $\epsilon$

• Every value in algorithm is in form $x = x_1 + x_2\epsilon$, where $x_2 = O(n^3)$ is an integer
• Forget $\epsilon$. Store pair $(x_1, x_2)$

• (A) Addition is pairwise-addition
  • $(x_1, x_2) + (y_1, y_2) = (x_1 + y_1, x_2 + y_2)$

• (C) Comparison is lexicographic-comparison
  • $(x_1, x_2) < (y_1, y_2)$ iff $x_1 < y_1$ or $x_1 = y_1$ and $x_2 = y_2$

• Both (A) and (C) can be implemented in $O(1)$ time without knowing $\epsilon$
• Perturbed algorithm has same asymptotic running time as regular one
Odds and Ends
Odds and Ends

- Designed $O(n^4)$ algorithm for constructing OBCSTs when $C=\{<,=\}$ and need to report Exact Failures
Odds and Ends

- Designed $O(n^4)$ algorithm for constructing OBCSTs when $C=\{<,=\}$ and need to report Exact Failures

- Strongly used assumption $\beta_i$ are all distinct
  - Assumption can be removed using perturbation argument
Odds and Ends

• Designed $O(n^4)$ algorithm for constructing OBCSTs when $C=\{<,=\}$ and need to report Exact Failures

• Strongly used assumption $\beta_i$ are all distinct
  • Assumption can be removed using perturbation argument

• To solve problem $C=\{<,=\}$ with Non-Exact failures
  • only need to modify initial conditions
Odds and Ends

• Designed $O(n^4)$ algorithm for constructing OBCSTs when $C=\{<,=\}$ and need to report Exact Failures

• Strongly used assumption $\beta_i$ are all distinct
  • Assumption can be removed using perturbation argument

• To solve problem $C=\{<,=\}$ with Non-Exact failures
  • only need to modify initial conditions

• Symmetry argument gives algorithms for $C=\{\leq, =\}$
Odds and Ends

• Designed $O(n^4)$ algorithm for constructing OBCSTs when $C=\{<,=\}$ and need to report Exact Failures

• Strongly used assumption $\beta_i$ are all distinct
  • Assumption can be removed using perturbation argument

• To solve problem $C=\{<,=\}$ with Non-Exact failures
  • only need to modify initial conditions

• Symmetry argument gives algorithms for $C=\{\leq, =\}$

• Algorithms for $C=\{<, \leq, =\}$ requires only slight modifications of SPLIT(i,j: h)
Odds and Ends

- Designed $O(n^4)$ algorithm for constructing OBCSTs when $C=\{<,=\}$ and need to report Exact Failures

- Strongly used assumption $\beta_i$ are all distinct
  - Assumption can be removed using perturbation argument

- To solve problem $C=\{<,=\}$ with Non-Exact failures
  - only need to modify initial conditions

- Symmetry argument gives algorithms for $C=\{\leq, =\}$

- Algorithms for $C=\{<, \leq, =\}$ requires only slight modifications of SPLIT(i,j: h)

- If $C=\{<, \leq\}$, ranges have no holes and problem can be solved in $O(n \log n)$ similar to Hu-Tucker
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• New Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
    • Proof of The Main Lemma (Sketch)
• Extensions and Open Problems
Proof of Main Lemma

\[ Q = x \]

\[ Q < K_j \]

\[ T_1 \]

\[ Q < z \]

\[ Q = y \]

\[ T_3 \]

\[ Q = y \]

\[ T_3 \]
Proof of Main Lemma

Let T be an OBCST. Assume

\[
\begin{align*}
T_3 & \quad Q = x \\
T_1 & \quad Q < K_j \\
T_3 & \quad Q < K_j \\
T_3 & \quad Q = y \\
T_3 & \quad Q = y \\
\end{align*}
\]
Proof of Main Lemma

Let $T$ be an OBCST. Assume

- $y < x$ (the ordering is symmetric)
Proof of Main Lemma

Let T be an OBCST. Assume

- \( y < x \) (\( x > y \) is symmetric)
- \( (Q=x) \) is above \( (Q=y) \)
Proof of Main Lemma

Let T be an OBCST. Assume

- \( y < x \) (\( x > y \) is symmetric)
- \( (Q=x) \) is above \( (Q=y) \)
  - \( \Rightarrow \beta_x < \beta_y \) will show contradiction
Proof of Main Lemma

Let $T$ be an OBCST. Assume

- $y < x$ (symmetric)
- $(Q = x)$ is above $(Q = y)$
  - $\Rightarrow \beta_x < \beta_y$ will show contradiction
  - $\Rightarrow \beta_x \geq \beta_y$ and Thm correct
Proof of Main Lemma

Let T be an OBCST. Assume

- $y < x$ (x>y is symmetric)
- $(Q=x)$ is above $(Q=y)$
  - => $\beta_x < \beta_y$ will show contradiction
  - => $\beta_x \geq \beta_y$ and Thm correct

- All comparisons between $(Q=x)$ and $(Q=y)$ are inequalities
  - otherwise $\exists (Q=w)$ on path with either $\beta_x < \beta_w$ or $\beta_w < \beta_y$ and can show contradiction with (x,w) or (w,y)
Proof of Main Lemma

Let T be an OBCST. Assume

- \(y < x\) (\(x > y\) is symmetric)
- \((Q=x)\) is above \((Q=y)\)
  - \(\Rightarrow \beta_x < \beta_y\) will show contradiction
  - \(\Rightarrow \beta_x \geq \beta_y\) and Thm correct
- All comparisons between \((Q=x)\) and \((Q=y)\) are inequalities
  - otherwise \(\exists (Q=w)\) on path with either \(\beta_x < \beta_w\) or \(\beta_w < \beta_y\) and can show contradiction with \((x,w)\) or \((w,y)\)
- \(x,y \in \text{Range}((Q=x))\) by definition
  If \(x,y \in \text{Range}((Q=y))\)
  then could swap \((Q=X)\) and \((Q=y)\) to get cheaper tree.
Proof of Main Lemma

x would be here
Proof of Main Lemma

Let $T$ be an OBCST. Assume

- $y < x$ (x > y is symmetric)
- $(Q = x)$ is above $(Q = y)$
- $\Rightarrow \beta_x < \beta_y$ will show contradiction

- All comparisons between $(Q = x)$ and $(Q = y)$ are inequalities
Proof of Main Lemma

Let T be an OBCST. Assume

- \( y < x \) (\( x > y \) is symmetric)
- \((Q=x)\) is above \((Q=y)\)
- \( \Rightarrow \beta_x < \beta_y \) will show contradiction

- All comparisons between \((Q=x)\) and \((Q=y)\) are inequalities

- Since \( x \notin \text{Range}((Q=y)) \)
  \( \Rightarrow \) Path \((Q=x)\) to \((Q=y)\) contains \((Q<z)\)
  s.t. \( z \)'s children's ranges are \([i,z,h'), [z,j,h'')\) where \( y \in [i,z) \) and \( x \in [z,j) \).
  \( z \) is called *splitter*.

- \( P' \) is (red) path from \((Q=x)\) to \((Q=y)\)
Proof of Main Lemma
Proof of Main Lemma

• $P$ is path in $T$ from $(Q=x)$ to $(Q=y)$. $y < x$. $z$ is $x$-$y$ splitter on $P$
• $P'$ is path from $(Q=x)$ to $(Q=z)$
Proof of Main Lemma

- P is path in T from (Q=x) to (Q=y). y < x. z is x-y splitter on P
- P’ is path from (Q=x) to (Q=z)
- Proof will be case analysis of structure of P’
- For every P’, will show can build cheaper OBCST T’ contradicting optimality of T
Proof of Main Lemma

- P is path in T from (Q=x) to (Q=y). \( y < x \). \( z \) is x-y splitter on P
- \( P' \) is path from (Q=x) to (Q=z)
- Proof will be case analysis of structure of \( P' \)
- For every \( P' \), will show can build cheaper OBCST T’ contradicting optimality of T

**Case 1: \( P' \) is one edge**
Proof of Main Lemma

• P is path in T from (Q=x) to (Q=y). y < x. z is x-y splitter on P
• P’ is path from (Q=x) to (Q=z)
• Proof will be case analysis of structure of P’
• For every P’, will show can build cheaper OBCST T’ contradicting optimality of T

Case 1: P’ is one edge
Proof of Main Lemma

- P is path in T from (Q=x) to (Q=y). \( y < x \). z is x-y splitter on P
- P’ is path from (Q=x) to (Q=z)
- Proof will be case analysis of structure of P’
- For every P’, will show can build cheaper OBCST T’ contradicting optimality of T

**Case 1: P’ is one edge**

\( y \in A \implies \text{Weight}(A) \geq \beta_y > \beta_x \)
Proof of Main Lemma

- P is path in T from (Q=x) to (Q=y). y < x. z is x-y splitter on P
- P’ is path from (Q=x) to (Q=z)
- Proof will be case analysis of structure of P’
- For every P’, will show can build cheaper OBCST T’ contradicting optimality of T

**Case 1: P’ is one edge**

\[ y \in A \implies \text{Weight}(A) \geq \beta_y > \beta_x \]

\[ \implies \text{replacing left subtree by right subtree in T yields new BCST T’ with lower cost than T, contradicting T being OBCST} \]
Proof of Main Lemma

• P is path in T from (Q=x) to (Q=y).  y<x.  z is x-y splitter on P
• P’ is path from (Q=x) to (Q=z)

Case 2: P’ is two edges ≠
Proof of Main Lemma

• P is path in T from (Q=x) to (Q=y). y<x. z is x-y splitter on P
• P’ is path from (Q=x) to (Q=z)

Case 2: P’ is two edges ≠

y∈A => Weight(A) ≥ β_y > β_x
Proof of Main Lemma

- P is path in T from (Q=x) to (Q=y). y<x. z is x-y splitter on P
- P’ is path from (Q=x) to (Q=z)

**Case 2: P’ is two edges ≠**

y∈A => Weight(A) ≥ β_y > β_x

=> again replacing left tree by right tree in T yields new BCST T’ with lower cost than T, contradicting T being OBCST
Proof of Main Lemma
Proof of Main Lemma

• P is path in T from (Q=x) to (Q=y). y<x. z is x-y splitter on P
• P’ is path from (Q=x) to (Q=z)
• Proof will be case analysis of structure of P’
Proof of Main Lemma

- \( P \) is path in \( T \) from \((Q=x)\) to \((Q=y)\). \( y < x \). \( z \) is \( x \)-\( y \) splitter on \( P \)
- \( P' \) is path from \((Q=x)\) to \((Q=z)\)
- Proof will be case analysis of structure of \( P' \)

- Already saw first two cases of \( P' \)
  - Showed for each that assumptions allow replacing subtree rooted at \((Q=x)\) with cheaper subtree for some range. Replacement leads to cheaper BCST, contradicting optimality of \( T \)
Proof of Main Lemma

- P is path in T from (Q=x) to (Q=y). y<x. z is x-y splitter on P
- P’ is path from (Q=x) to (Q=z)
- Proof will be case analysis of structure of P’

- Already saw first two cases of P’
  - Showed for each that assumptions allow replacing subtree rooted at (Q=x) with cheaper subtree for some range. Replacement leads to cheaper BCST, contradicting optimality of T

- The full proof splits P’ into 7 cases.
  - For each, can show replacement with cheaper subtree, contradicting optimality of T.
Outline

• History
  • Binary Search Trees
  • Hu-Tucker Trees
  • AKKL Trees
• Optimal Binary Comparison Search Trees with Failures
  • Problem Models
  • List of New Results
• Our Results
  • The Main Lemma
  • Structural Properties of OBCSTs
  • Dynamic Programming for OBCSTs
  • Proof of The Main Lemma
• Extensions and Open Problems
Extensions & Open Problems

• If the $\beta_i, \alpha_i$ are probabilities (sum to 1) can show an $O(n)$ algorithm that constructs BCST within 
  additive error 3 of optimal for Exact Failure Case
  • Modification of similar algorithm for Hu-Tucker case.

• $O(n^4)$ is quite high for worst case.
  • Can we do better?