CS-204: Design and Analysis of Algorithms

220001062, 220001033

April 19, 2024

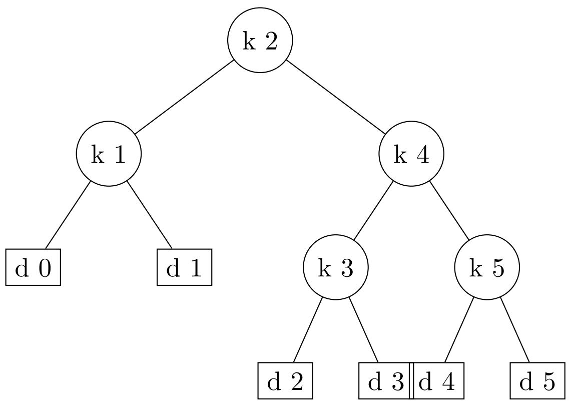
1 Optimal Binary Search Trees

To optimize search time for translating English words to Latvian, we design a binary search tree where frequently used words are closer to the root. This is achieved by constructing an optimal binary search tree, with keys as English words and probabilities of occurrence provided. Dummy keys represent values not in the tree. Placing frequently used words nearer to the root reduces search time.

Given a sequence $K = \{k_1, k_2, \ldots, k_n\}$ of n distinct keys such that $k_1 < k_2 < \ldots < k_n$, we build a binary search tree containing them. For each key k_i, we are given the probability p_i that any search is for key k_i. Since some searches may be for values not in K, we also have $n+1$ dummy keys $d_0, d_1, d_2, \ldots, d_n$ representing those values. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have the probability q_i that a search corresponds to d_i.

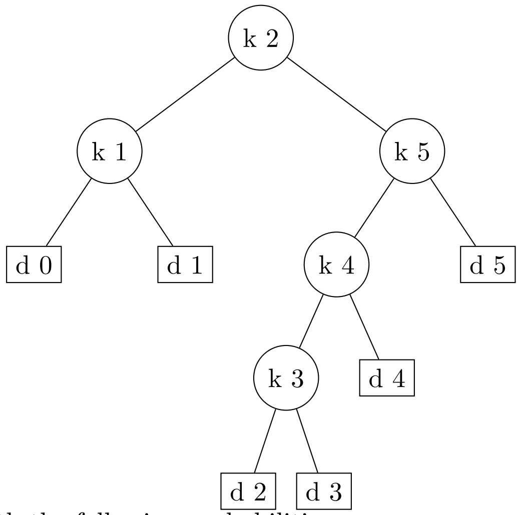
(a)

Node	Depth	Probability	Contribution
$k1$	1	0.15	0.30
$k2$	0	0.10	0.10
$k3$	2	0.05	0.15
$k4$	1	0.10	0.20
$k5$	2	0.20	0.60
$d0$	2	0.05	0.15
$d1$	2	0.10	0.30
$d2$	3	0.05	0.20
$d3$	3	0.05	0.20
$d4$	3	0.05	0.20
$d5$	3	0.10	0.40
Total			2.80



(b)

Node	Depth	Probability	Contribution
$k1$	1	0.15	0.30
$k2$	0	0.10	0.10
$k3$	3	0.05	0.20
$k4$	2	0.10	0.30
$k5$	1	0.20	0.40
$d0$	2	0.05	0.15
$d1$	2	0.10	0.30
$d2$	4	0.05	0.25
$d3$	4	0.05	0.25
$d4$	3	0.05	0.20
$d5$	2	0.10	0.30
Total			2.75



Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4
p_i	0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05
0.10					

(a) A binary search tree with expected search cost 2.80.

(b) A binary search tree with expected search cost 2.75. This tree is optimal.

The expected cost of a search in a given binary search tree T is determined by the probabilities of searches for each key and each dummy key. Assuming the actual cost of a search equals the depth of the node found by the search in T, plus 1, the expected cost of a search in T is given by:

$$E[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

Step 1: Constructing Optimal Binary Search Trees

...

Step 2: A Recursive Solution

...

Step 3: Computing Expected Search Cost

...

article algorithm algorithmic
=0

```
OPTIMAL-BST($p$, $q$, $n$)
Let $e[1 \ldots n+1, 0 \ldots n]$, $w[1 \ldots n+1, 0 \ldots n]$, and root$[1 \ldots n, 1 \ldots n]$ be new tables.
for $i \leftarrow 1$ to $n+1$ do
    {base cases}$e[i, i-1] \leftarrow q_{i-1}$ {equation } $w[i, i-1] \leftarrow q_{i-1}$
end for
for $l \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n-l+1$ do
        $j \leftarrow i+l-1$
        $e[i, j] \leftarrow 1$
        $w[i][j] \leftarrow w[i, j-1] + p_j + q_j$ {equation }
        for $r \leftarrow i$ to $j$ do
            {try all possible roots $r$}$t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$ if $t < e[i, j]$ then {new
            minimum?}
            $e[i, j] \leftarrow t$
            root$[i, j] \leftarrow r$
          end if
        end for
    end for
end for
return $e$ and root =0
```

2. Subset Sum Problem

Given a set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.

Examples:

Input: set[ ] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
Explanation: There is a subset (4, 5) with sum 9.

Input: set[ ] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
Explanation: There is no subset that adds up to 30.

Subset Sum Problem using Dynamic Programming :

To solve the problem in Pseudo-polynomial time we can use the Dynamic programming approach.

So we will create a 2D array of size $(n+1) \times (sum+1)$ of type boolean. The state $dp[i][j]$ will be true if there exists a subset of elements from $set[0 \dots i]$ with sum value j.

The dynamic programming relation is as follows:

```
if $(A[i-1] > j)$
    $dp[i][j] = dp[i-1][j]$
else
    $dp[i][j] = dp[i-1][j] OR dp[i-1][j - set[i-1]]$
```

This means that if the current element has a value greater than the current sum value, we will copy the answer for previous cases. Otherwise, if the current sum value is greater than the *ith* element, we will see if any of the previous states have already experienced the sum j OR any previous states experienced a value $(j - set[i])$ which will solve our purpose.

Complexity:

- Time Complexity: $O(sum \cdot n)$, where n is the size of the array.
- Auxiliary Space: $O(sum \cdot n)$, as the size of the 2-D array is $sum \cdot n$.

Illustration

Consider $set[] = \{3, 4, 5, 2\}$ and $sum = 6$.

The table will look like the following where the column number represents the sum and the row number represents the index of $set[]$:

Table 1: Subset Sum Problem Dynamic Programming Table

Set Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
No element (0)	T	F	F	F	F	F	F
0 (3)	T	F	F	T	F	F	F
1 (4)	T	F	F	T	T	F	F
2 (5)	T	F	F	T	T	T	F
3 (2)	T	F	T	T	T	T	T

---

3. Weighted Interval Scheduling Dynamic Programming

Given a list of jobs where each job has a start and finish time, and a profit associated with it, find a maximum profit subset of non-overlapping jobs.

For example, consider the following jobs with their starting time, finishing time, and associated profit. The maximum profit is 80, and the jobs involved in the maximum profit are: (1, 4, 30) and (5, 9, 50).
Job 1: (0, 6, 60)
Job 2: (5, 9, 50)
Job 3: (1, 4, 30)
Job 4: (5, 7, 30)
Job 5: (3, 5, 10)
Job 6: (7, 8, 10)

The idea is first to sort given jobs in increasing order of their start time. Let jobs[0...n-1] be the sorted array of jobs. We can define an array maxProfit[] such that maxProfit[i] itself is an array that stores the non-conflicting jobs with maximum profit that ends with the i'th job.

Therefore, maxProfit[i] can be recursively written as:

$$\text{maxProfit}[i] = \begin{cases} \text{jobs}[i] + \max(\text{maxProfit}[j]), & \text{if } \exists\ j < i \text{ such that jobs}[j]\text{.finish} \leq \text{jobs}[i]\text{.start} \\ \text{jobs}[i], & \text{otherwise} \end{cases}$$

Complexity:
The time complexity of the above solution is $O(n^2)$ and requires $O(n)$ extra space, where n is the total number of jobs.
