WIKIPEDIA
The Free Encyclopedia
WIKIPEDIA

Optimal binary search tree

In computer science, an **optimal binary search tree (Optimal BST)**, sometimes called a **weight-balanced binary tree**,[1] is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities). Optimal BSTs are generally divided into two types: static and dynamic.

In the **static optimality** problem, the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities. Various algorithms exist to construct or approximate the statically optimal tree given the information on the access probabilities of the elements.

In the **dynamic optimality** problem, the tree can be modified at any time, typically by permitting tree rotations. The tree is considered to have a cursor starting at the root which it can move or use to perform modifications. In this case, there exists some minimal-cost sequence of these operations which causes the cursor to visit every node in the target access sequence in order. The splay tree is conjectured to have a constant competitive ratio compared to the dynamically optimal tree in all cases, though this has not yet been proven.

Static optimality

Definition

In the static optimality problem as defined by Knuth,[2] we are given a set of n ordered elements and a set of $2n+1$ probabilities. We will denote the elements a_1 through a_n and the probabilities A_1 through A_n and B_0 through B_n. A_i is the probability of a search being done for element a_i (or *successful search*).[3] For $1 \leq i < n$, B_i is the probability of a search being done for an element between a_i and a_{i+1} (or *unsuccessful search*),[3] B_0 is the probability of a search being done for an element strictly less than a_1, and B_n is the probability of a search being done for an element strictly greater than a_n. These $2n+1$ probabilities cover all possible searches, and therefore add up to one.

The static optimality problem is the optimization problem of finding the binary search tree that minimizes the expected search time, given the $2n+1$ probabilities. As the number of possible trees on a set of n elements is $\binom{2n}{n}\frac{1}{n+1}$,[2] which is exponential in n, brute-force search is not usually a feasible solution.

Knuth's dynamic programming algorithm

In 1971, Knuth published a relatively straightforward dynamic programming algorithm capable of constructing the statically optimal tree in only $O(n^2)$ time.[2] In this work, Knuth extended and improved the dynamic programming algorithm by Edgar Gilbert and Edward F. Moore introduced in 1958.[4] Gilbert's and Moore's algorithm required $O(n^3)$ time and $O(n^2)$ space and was designed for a particular case of optimal binary search trees construction (known as *optimal alphabetic tree problem*[5]) that considers only the probability of unsuccessful searches, that is, $\sum_{i=1}^{n} A_i = 0$. Knuth's work relied upon the following insight: the static optimality problem exhibits optimal substructure; that is, if a certain tree is statically optimal for a given probability distribution, then its left and right subtrees must also be statically optimal for their appropriate subsets of the distribution (known as monotonicity property of the roots).

To see this, consider what Knuth calls the "weighted path length" of a tree. The weighted path length of a tree of n elements is the sum of the lengths of all $2n+1$ possible search paths, weighted by their respective probabilities. The tree with the minimal weighted path length is, by definition, statically optimal.

But weighted path lengths have an interesting property. Let E be the weighted path length of a binary tree, E_L be the weighted path length of its left subtree, and E_R be the weighted path length of its right subtree. Also let W be the sum of all the probabilities in the tree. Observe that when either subtree is attached to the root, the depth of each of its elements (and thus each of its search paths) is increased by one. Also observe that the root itself has a depth of one. This means that the difference in weighted path length between a tree and its two subtrees is exactly the sum of every single probability in the tree, leading to the following recurrence:

$$E = E_L + E_R + W$$

This recurrence leads to a natural dynamic programming solution. Let E_{ij} be the weighted path length of the statically optimal search tree for all values between a_i and a_j, let W_{ij} be the total weight of that tree, and let R_{ij} be the index of its root. The algorithm can be built using the following formulas:

$$\begin{aligned} E_{i,i-1} = W_{i,i-1} &= B_{i-1} \text{ for } 1 \leq i \leq n+1 \\ W_{i,j} &= W_{i,j-1} + A_j + B_j \\ E_{i,j} &= \min_{i \leq r \leq j} (E_{i,r-1} + E_{r+1,j} + W_{i,j}) \text{ for } 1 \leq i \leq j \leq n \end{aligned}$$

The naive implementation of this algorithm actually takes $O(n^3)$ time, but Knuth's paper includes some additional observations which can be used to produce a modified algorithm taking only $O(n^2)$ time.

In addition to its dynamic programming algorithm, Knuth proposed two heuristics (or rules) to produce *nearly (approximation of) optimal binary search trees*. Studying nearly optimal binary search trees was necessary since Knuth's algorithm time and space complexity can be prohibitive when n is substantially large.[6]

Knuth's rules can be seen as the following:

- **Rule I (Root-max):** Place the most frequently occurring name at the root of the tree, then proceed similarly on the subtrees.
- **Rule II (Bisection):** Choose the root so as to equalize the total weight of the left and right subtree as much as possible, then proceed similarly on the subtrees.

Knuth's heuristics implements nearly optimal binary search trees in $O(n \log n)$ time and $O(n)$ space.

The analysis on how far from the optimum Knuth's heuristics can be was further proposed by Kurt Mehlhorn.[6]

Mehlhorn's approximation algorithm

While the $O(n^2)$ time taken by Knuth's algorithm is substantially better than the exponential time required for a brute-force search, it is still too slow to be practical when the number of elements in the tree is very large.

In 1975, Kurt Mehlhorn published a paper proving important properties regarding Knuth's rules. Mehlhorn's major results state that only one of Knuth's heuristics **(Rule II)** always produces nearly optimal binary search trees. On the other hand, the root-max rule could often lead to very "bad" search trees based on the following simple argument.[6]

Let

$$n = 2^k - 1,\ \ A_i = 2^{-k} + \varepsilon_i \ \text{ with } \ \sum_{i=1}^{n} \varepsilon_i = 2^{-k}$$

and

$$\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n > 0 \ \text{ for } \ 1 \leqq i \leqq n \ \text{ and } \ B_j = 0 \,\text{for}\ \ 0 \leqq j \leqq n.$$

Considering the weighted path length P of the tree constructed based on the previous definition, we have the following:

$$\begin{aligned} P &= \sum_{i=1}^{n} A_i(a_i + 1) + \sum_{j=1}^{n} B_j b_j \\ &= \sum_{i=1}^{n} A_i i \\ &\geqq 2^{-k} \sum_{i=1}^{n} i = 2^{-k} \frac{n(n+1)}{2} \geqq \frac{n}{2}. \end{aligned}$$

Thus, the resulting tree by the root-max rule will be a tree that grows only on the right side (except for the deepest level of the tree), and the left side will always have terminal nodes. This tree has a path length bounded by $\Omega(\frac{n}{2})$ and, when compared with a balanced search tree (with path bounded by $O(2 \log n)$), will perform substantially worse for the same frequency distribution.[6]

In addition, Mehlhorn improved Knuth's work and introduced a much simpler algorithm that uses Rule II and closely approximates the performance of the statically optimal tree in only $O(n)$ time.[6] The algorithm follows the same idea of the bisection rule by choosing the tree's root to balance the total weight (by probability) of the left and right subtrees most closely. And the strategy is then applied recursively on each subtree.

That this strategy produces a good approximation can be seen intuitively by noting that the weights of the subtrees along any path form something very close to a geometrically decreasing sequence. In fact, this strategy generates a tree whose weighted path length is at most

$$2 + (1 - \log(\sqrt{5} - 1))^{-1} H = 2 + \frac{H}{1 - \log(\sqrt{5} - 1)}$$

where H is the entropy of the probability distribution. Since no optimal binary search tree can ever do better than a weighted path length of

$$(1/\log 3)H = \frac{H}{\log 3}$$

this approximation is very close.[6]

Hu–Tucker and Garsia–Wachs algorithms

In the special case that all of the A_i values are zero, the optimal tree can be found in time $O(n \log n)$. This was first proved by T. C. Hu and Alan Tucker in a paper that they published in 1971. A later simplification by Garsia and Wachs, the Garsia–Wachs algorithm, performs the same comparisons in the same order. The algorithm works by using a greedy algorithm to build a tree that has the optimal height for each leaf, but is out of order, and then constructing another binary search tree with the same heights.[7]

Example Code Snippet

The following code snippet determines an optimal binary search tree when given a set of keys and probability values that the key is the search key:

```
public static float calculateOptimalSearchTree(int numNodes, float[] probabilities, int[][] roots) {
       float[][] costMatrix = new float[numNodes + 2][numNodes + 1];
       for (int i = 1; i <= numNodes; i++) {
           costMatrix[i][i - 1] = 0;
           costMatrix[i][i] = probabilities[i];
           roots[i][i] = i;
           roots[i][i - 1] = 0;
       }
       for (int diagonal = 1; diagonal <= numNodes; diagonal++) {
           for (int i = 1; i <= numNodes - diagonal; i++) {
               int j = i + diagonal;
               costMatrix[i][j] = findMinCost(costMatrix, i, j) + sumProbabilities(probabilities, i,
j);
               // Note: roots[i][j] assignment is missing, this needs to be fixed if you want
               // to reconstruct the tree.
           }
       }
       return costMatrix[1][numNodes];
}
```

Dynamic optimality

Unsolved problem in computer science:

Do splay trees perform as well as any other binary search tree algorithm?

(more unsolved problems in computer science)

Definition

There are several different definitions of dynamic optimality, all of which are effectively equivalent to within a constant factor in terms of running-time.[8] The problem was first introduced implicitly by Sleator and Tarjan in their paper on splay trees,[9] but Demaine et al. give a very good formal statement of it.[8]

In the dynamic optimality problem, we are given a sequence of accesses x_1, ..., x_m on the keys 1, ..., n. For each access, we are given a pointer to the root of our BST and may use the pointer to perform any of the following operations:

1. Move the pointer to the left child of the current node.
2. Move the pointer to the right child of the current node.
3. Move the pointer to the parent of the current node.
4. Perform a single rotation on the current node and its parent.

(It is the presence of the fourth operation, which rearranges the tree during the accesses, which makes this the *dynamic* optimality problem.)

For each access, our BST algorithm may perform any sequence of the above operations as long as the pointer eventually ends up on the node containing the target value x_i. The time it takes a given dynamic BST algorithm to perform a sequence of accesses is equivalent to the total number of such operations performed during that sequence. Given any sequence of accesses on any set of elements, there is some minimum total number of operations required to perform those accesses. We would like to come close to this minimum.

While it is impossible to implement this "God's algorithm" without foreknowledge of exactly what the access sequence will be, we can define OPT(X) as the number of operations it would perform for an access sequence X, and we can say that an algorithm is dynamically optimal if, for any X, it performs X in time O(OPT(X)) (that is, it has a constant competitive ratio).[8]

There are several data structures conjectured to have this property, but none proven. It is an open problem whether there exists a dynamically optimal data structure in this model.

Splay trees

The splay tree is a form of binary search tree invented in 1985 by Daniel Sleator and Robert Tarjan on which the standard search tree operations run in $O(\log(n))$ amortized time.[10] It is conjectured to be dynamically optimal in the required sense. That is, a splay tree is believed to perform any sufficiently long access sequence X in time O(OPT(X)).[9]

Tango trees

The tango tree is a data structure proposed in 2004 by Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătrașcu which has been proven to perform any sufficiently-long access sequence X in time $O(\log\log n\,\mathrm{OPT}(X))$. While this is not dynamically optimal, the competitive ratio of $\log\log n$ is still very small for reasonable values of n.[8]

Other results

In 2013, John Iacono published a paper which uses the geometry of binary search trees to provide an algorithm which is dynamically optimal if any binary search tree algorithm is dynamically optimal.[11] Nodes are interpreted as points in two dimensions, and the optimal access sequence is the smallest arborally satisfied superset of those points. Unlike splay trees and tango trees, Iacono's data structure is not known to be implementable in constant time per access sequence step, so even if it is dynamically optimal, it could still be slower than other search tree data structures by a non-constant factor.

The interleave lower bound is an asymptotic lower bound on dynamic optimality.

See also

- Trees
- Splay tree
- Tango tree
- Geometry of binary search trees
- Interleave lower bound

Notes

1. Tremblay, Jean-Paul; Cheston, Grant A. (2001). *Data Structures and Software Development in an object-oriented domain*. Eiffel Edition/Prentice Hall. ISBN 978-0-13-787946-5.
2. Knuth, Donald E. (1971), "Optimum binary search trees", *Acta Informatica*, **1** (1): 14–25, doi:10.1007/BF00264289 (https://doi.org/10.1007%2FBF00264289), S2CID 62777263 (https://api.semanticscholar.org/CorpusID:62777263)
3. Nagaraj, S. V. (1997-11-30). "Optimal binary search trees" (https://www.sciencedirect.com/science/article/pii/S0304397596003209). *Theoretical Computer Science*. **188** (1): 1–44. doi:10.1016/S0304-3975(96)00320-9 (https://doi.org/10.1016%2FS0304-3975%2896%2900320-9). ISSN 0304-3975 (https://search.worldcat.org/issn/0304-3975). S2CID 33484183 (https://api.semanticscholar.org/CorpusID:33484183).
4. Gilbert, E. N.; Moore, E. F. (July 1959). "Variable-Length Binary Encodings" (https://ieeexplore.ieee.org/document/6768523). *Bell System Technical Journal*. **38** (4): 933–967. doi:10.1002/j.1538-7305.1959.tb01583.x (https://doi.org/10.1002%2Fj.1538-7305.1959.tb01583.x).
5. Hu, T. C.; Tucker, A. C. (December 1971). "Optimal Computer Search Trees and Variable-Length Alphabetical Codes" (https://dx.doi.org/10.1137/0121057). *SIAM Journal on Applied Mathematics*. **21** (4): 514–532. doi:10.1137/0121057 (https://doi.org/10.1137%2F0121057). ISSN 0036-1399 (https://search.worldcat.org/issn/0036-1399).
6. Mehlhorn, Kurt (1975), "Nearly optimal binary search trees" (http://edoc.mpg.de/344677), *Acta Informatica*, **5** (4): 287–295, doi:10.1007/BF00264563 (https://doi.org/10.1007%2FBF00264563), S2CID 17188103 (https://api.semanticscholar.org/CorpusID:17188103)

7. Knuth, Donald E. (1998), "Algorithm G (Garsia–Wachs algorithm for optimum binary trees)", *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2nd ed.), Addison–Wesley, pp. 451–453. See also History and bibliography, pp. 453–454.
8. Demaine, Erik D.; Harmon, Dion; Iacono, John; Patrascu, Mihai (2004), "Dynamic optimality—almost" (http://erikdemaine.org/papers/Tango_SICOMP/paper.pdf) (PDF), *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pp. 484–490, CiteSeerX 10.1.1.99.4964 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.4964), doi:10.1109/FOCS.2004.23 (https://doi.org/10.1109%2FFOCS.2004.23), ISBN 978-0-7695-2228-9
9. Sleator, Daniel; Tarjan, Robert (1985), "Self-adjusting binary search trees", *Journal of the ACM*, **32** (3): 652–686, doi:10.1145/3828.3835 (https://doi.org/10.1145%2F3828.3835), S2CID 1165848 (https://api.semanticscholar.org/CorpusID:1165848)
10. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald; Stein, Clifford (2009). *Introduction to algorithms* (http://ressources.unisciel.fr/algoprog/s00aaroot/aa00module1/res/%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf) (PDF) (Third ed.). The MIT Press. p. 503. ISBN 978-0-262-03384-8. Retrieved 31 October 2017.
11. Iacono, John (2013), "In pursuit of the dynamic optimality conjecture", arXiv:1306.0207 (https://arxiv.org/abs/1306.0207) [cs.DS (https://arxiv.org/archive/cs.DS)]