HOW TO WRITE A FAST OPTIMAL BINARY SEARCH TREE IMPLEMENTATION

Jeremia Bär, Stefan Dietiker

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this report our aim is to investigate methods to optimally implement a dynamic programming algorithm to construct optimal binary search trees. Our experimental results clearly demonstrate that concise changes to the structure of the algorithm, such as the order of the loops, the memory layout, etc., boost its performance significantly. Furthermore, we demonstrate how to apply vectorization to this algorithm. The best implementation outperforms the straight-forward baseline implementation by a factor of 6.

1. INTRODUCTION

Binary search trees are amongst the most fundamental data structures in computer science. An ordered set of keys is held at the nodes of the tree where smaller keys reside in the left subtree, larger keys in the right subtree. This presents an ordered structure of the keys. Given the simple and recursive structure of a binary search tree, it is easy to deduce an algorithm to find a specific key in the tree. The tree can be used as a set or as a key-value store, if references to data is stored in the nodes.

The structure of the binary search tree is not fixed. Different keys can be chosen as root, resulting in different subtrees. There are great variances among these tree structures concerning the lookup, insertion and deletion cost. Depending on the application domain, algorithms enforcing different structures on the binary search tree should be used optimizing the cost for the most important operations on the tree. Such considerations include: Is the tree allowed to change once it is generated (dynamic vs. static)? What is the insertion, deletion and lookup cost? What is the distribution of such operations in my application domain?

For dynamic binary search trees, various algorithms have been developed to balance the tree on inseration and deletion. In the case of static binary search trees, on the other hand, the set of keys is given a priori, along with a probability distribution that indicates the likeliness of an individual key being searched for. Based upon this, one can construct an optimal binary search tree, that is, a search tree which has a minimal expected lookup cost.

In the following we introduce a dynamic programming algorithm, which solves the problem by examining combinations of optimal subtrees. Our objective then is to optimally implement this algorithm on a modern Intel-based hardware platform.

Our interest in this algorithm emerges from its straightforward design which, not only makes its own algorithmic properties apparent and easy to analyze, but also provides us with clear conceptual guidance when optimizing the algorithm. Our hope is thus that the observations made in this report can be translated to similar problems.

2. BACKGROUND

In this section we give a mathematical description of the optimal binary search tree problem. A discussion of the applied algorithm and its cost follows. The presentation is based on the book by Cormen et al. [1]. We conclude with a brief overview of two alternative algorithms.

Problem Statement. Let $K = \{k_1, k_2, \ldots, k_n\}$ be a sequence of distinct ordered keys. For each key k_i, let p_i be the probability that a given search is for key k_i. Such a search is called successful. Further, let $D = \{d_0, d_1, \ldots, d_n\}$ be the set of dummy keys returned for unsuccessful searches as follows: d_i represents searches for values between k_i and k_{i+1}, d_0 represent the values smaller k_0 and d_n the values larger k_n. For each dummy key d_i, let q_i be the probability that a given search returns d_i. A valid solution to the static binary search tree problem is any binary search tree T that has K as nodes and D as leaves. The cost of a search in a such a tree T is defined as the depth of the key found plus one. Since K and D cover all possible searches, $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$ and we can compute the expected cost of a search in T as:

$$\begin{aligned}\sum_{i=1}^{n}(\mathrm{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(\mathrm{depth}_T(d_i)+1)\cdot q_i \\ = 1 + \sum_{i=1}^{n}\mathrm{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n}\mathrm{depth}_T(d_i)\cdot q_i\end{aligned} \tag{1}$$

A static binary search tree is called optimal if its expected search cost is minimal amongst all valid solution trees. We can now formulate the static optimal binary search tree problem as follows: Given K, P and Q, find the optimal binary search tree.

Algorithm. Devising an algorithm to solve the problem requires a crucial insight on the problem structure: Given an optimal tree T for keys k_1 to k_n with root k_r, its left subtree T_L is an optimal solution for the keys k_1 to k_{r-1}. Clearly, it has to be a binary search tree for these keys. Then, if it were not optimal, one could replace T_L by an optimal binary search tree for the keys $k_1, \ldots, k_{r-1}$ obtaining a better solution T' for the original problem. However, this contradicts the optimality of T and hence T_L is optimal as well. This argument holds analogous for all subtrees in T.

From this insight we can construct an algorithm. Let $e[i, j]$ denote the expected search cost in the optimal binary search tree containing keys k_i through k_j. If such a tree is used as a left or right subtree to construct a tree containing more keys, the depth of its nodes increases by one. Following Equation 1, the subtree's expected search cost increases by

$$\begin{aligned} w(i,j) &= \sum_{l=1}^{j} p_l + \sum_{l=i-1}^{j} q_l \\ &= w(i, r-1) + p_r + w(r+1, j) \end{aligned} \tag{2}$$

where the second expression reflects the recursive structure of the problem again. This allows to express the search cost of a binary search tree over k_i to k_j with root k_r constructed from its subtrees as

$$e[i,j] = e[i, r-1] + e[r+1, j] + w(i,j) \tag{3}$$

Knowing the optimal expected search costs for all possible subtrees, we can construct the optimal binary search tree by choosing the key as root that minimizes Equation 3, i.e [1]

$$e[i,j] = \min_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i,j)\} \tag{4}$$

This expression can now directly be translated into code, as is shown in Listing 1. The cost of the subtrees is computed using dynamic programming. The table `e[IDX(i,j)]` is used to store the expected search cost for the optimal search tree covering keys in k_i through k_j[2]. We fill the table `e` diagonal by diagonal, as illustrated in Figure 2. This corresponds to computing first all the subtrees containing one key, then containing two keys and so forth, as indicated by the length variable `l`. The innermost `r`-loop iterates over all valid roots for the subtree to find the optimal binary search

[1]Note that for simplicity, we do not include border cases here. For a complete discussion of the algorithm and its mathematics refer to [1].

[2]Note that the code actually uses zero-indexing.

```
1  for (l = 1; l < n+1; l++)
     for (i = 0; i < n-l+1; i++)
3      j = i+l;
       e[IDX(i,j)] = INFINITY;
5      w[IDX(i,j)] = w[IDX(i,j-1)] + p[j-1] + q[j];
       for (r = i; r < j; r++ )
7        t = e[IDX(i,r)] + e[IDX(r+1,j)]
             + w[IDX(i,j)];
9        if (t < e[IDX(i,j)])
           e[IDX(i,j)]    = t;
11         root[IDX(i,j)] = r;
```

Listing 1. Basline Implementation

tree for the current keys. We refer to the expected cost `e[IDX(i,j)]` to be computed as the *target cell*. A second table called `root` is used to keep track which root was chosen for the current subtree to be able to reconstruct the overall optimal tree in the end. In the code section, we have omitted the initialization code. The computation is dominated by the triple loop shown.

Cost Analysis. The presented algorithm has runtime $O(n^3)$. As can be seen in Listing 1, the computation involves additions and comparisons only. We define the cost function of the algorithm as the number of floating point additions and floating point comparisons:

$$C(n) = (\#\text{adds}(n), \#\text{comps}(n))$$

The body of the second loop is executed a total of

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

times. The body of the innermost loop is executed a total of

$$\sum_{i=1}^{n} i \cdot (n - i + 1) = \frac{1}{6}(n^3 + 3n^2 + 2n)$$

times. Hence, the cost function is defined as:

$$C(n) = \left(\frac{1}{3}(n^3 + 6n^2 + 5n), \frac{1}{6}(n^3 + 3n^2 + 2n)\right)$$

Alternative Algorithms. The presented algorithm was originally published by Knuth. He provides further analysis yielding that under certain conditions the root will not change, allowing to neglect some of the `r`-iterations of the algorithm. The final algorithm has runtime $O(n^2)$. Details are found in [2]. If an approximation of the result is sufficient, Mehlhorn [3] showed that balancing probabilities in the left and right subtrees already yields results close to optimal.

3. OPTIMIZATION

In the following, we present the methods we used to gradually improve the performance of the algorithm. We name the different implementations that we obtained according to the methods we employed. Alongside the presented methods, we also applied basic block optimizations such as scalar replacement, strength reduction and loop unrolling. Figure 1 gives an overview of the optimizations applied.

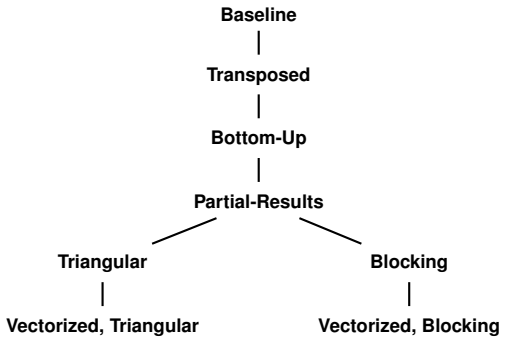

Fig. 1. Overview of optimizations.

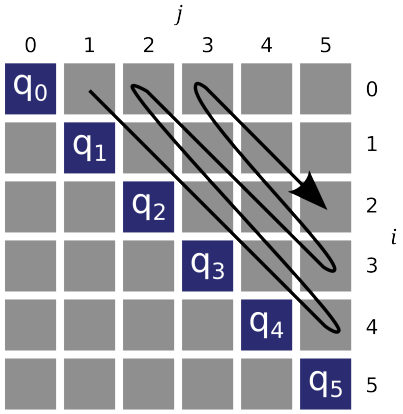

Fig. 2. Access Pattern of reference implementation.

Transposed. Examining the innermost loop, we see that the table is accessed row- and column-wise. Thus, the memory is accessed in strides of 1 and $n+1$, respectively. However, with two simple changes to the algorithm, we can avoid strides of $n+1$ and replace them with accesses of stride 1: We make use of the lower half of the square table in that we store newly calculated values not only at (i,j) but also at (j,i). That is, instead of accessing a column in the upper half, we access a row in the lower-half. This needs only two minor changes to the algorithm as described in Listing 1: First, line 7 turns into

```
t = e[IDX(i,r)] + e[IDX(j,r+1)],
```

and after line 10, we would insert

```
e[IDX(j,i)] = t;.
```

Bottom-Up. The order in which individual values are calculated is along the diagonals, as visualized in Figure 2. We notice that, for two distinct cells on a diagonal, their corresponding rows and columns intersect exactly in one cell. By changing the outermost two loops determining the target cell such that the table is built up row-wise and bottom-up, as visualized in Figure 3, we can improve temporal locality with respect to the accesses to the row that is currently being built-up.

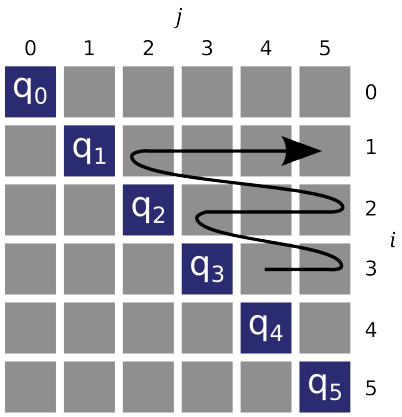

Fig. 3. Access Pattern of Bottom-Up.

Partial Results. Instead of calculating the minimum value over a full row and column, we can store intermediate values for an entire row by swapping the innermost two loops. That is, for row i, we add the value $e[i,r], i \leq r \leq n$ to all entries $e[r,j], r \leq j \leq n$ and store the minimum of that sum and $e[i,r]$ in $e[i,r]$. Figure 4 visualizes the pattern for one consecutive executions of the r-loop (which is now the second innermost loop): the value contained in the cell with the unfilled circle is added to all values contained in the cells with the filled circle and the minimum is stored in the white cells. This allows to have sequential access in the innermost loop without the need to keep the transposed version of the data values in the lower left half of the triangle.

Triangular. While the above approach need not necessarily improve the performance over the *Bottom Up*-approach, allocating the full square memory is no longer required. We can *compress* the memory layout by concatenating the rows of the "upper" triangular table. As for each target row i we compute, the full triangle below it is accessed row after row, this yields unit stride access even at the end of each intermediate row being traversed. The unused sections between the left end of the square and the beginning of the

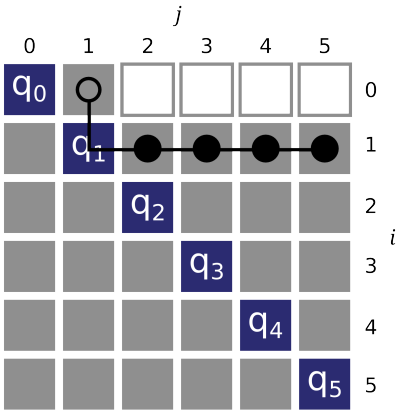

Fig. 4. Storing Partial results.

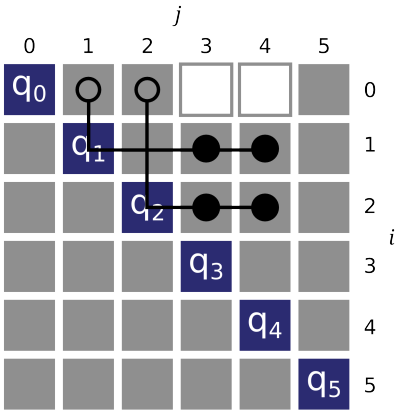

Fig. 5. Vectorization of Partial Results.

triangle is now omitted. This potentially also improves the performance of the hardware prefetcher. We call this memory layout *triangular*–as opposed to the *square* layout.

Using this memory layout significantly complicates the index calculation. The correct index-macro for this memory layout is:

```
#define IDX(i,j)\
((n+1)*(n+2)/2 - (n-(i)+1)*(n-(i)+2)/2 + (j) - (i))
```

Strength reduction can be applied to alleviate the computation. Index variables with constant offset changes over loop iterations make the involved index computation obsolete.

Vectorized Triangular. Historically, our initial approaches at blocking, explained below, were unsuccessful. Thus, we focused on vectorizing the triangular layout, which is straightforward: The current row is processed until the number of remaining cells is a multiple of the vector length. From then on, vector instructions are used.

Blocking. Based upon our *Partial Results* approach, we employed a blocking strategy as follows: We unrolled the two innermost loops. Thus, instead of storing the partial results in the target row for each subsequent row, we store calculate the minimum across multiple subsequent rows before storing the result. Figure 5 visualizes the approach: The value stored in the cell with the unfilled circle is added to the corresponding values stored in the cells containing the filled circles. The minimum across the columns containing the filled circles is stored in the empty cells which therefore contain partial results.

Vectorized Blocking. Once we properly implemented blocking, it is straight-forward to apply vectorization.

4. EXPERIMENTAL RESULTS

This section assesses the performance increase in the algorithm based on the methods described in the previous section. The test machine and software tools used are presented.

Manufacturer	Intel Corp.
CPU Name	Core i7-3520M
L1-Cache	32 KB
L2-Cache	256 KB
L3-Cache	4 MB
CPU-core frequency	2901 MHz
Scalar FP-Add Cycles/issue	1
Vect. FP-Add Cycles/issue (SSE)	2
Vect. FP-Add Cycles/issue (AVX)	4

Table 1. Processor details.

Measurement Setup. Measurements were performed on a Lenovo ThinkPad T430s with 8 GiB RAM and the processor shown in Table 1. The operating system was Kubuntu 14.04 64-bit. The compiler used was the GNU C Compiler 4.8.2 with the flags:

```
-O3 -m64 -march=corei7-avx -mavx
  -fno-tree-vectorize
```

Peak performance is equal to the floating point additions cycles/issue of the processor, see Table 1, since our algorithm does not perform any other floating point operations. That is, 1 fl/c for scalar code, 2 fl/c for SSE and 4 fl/c for AVX code. To asses the performance, libperfmon 4[3] was used to measure the number of cycles. The results were obtained by averaging over 10 consecutive executions for a given problem size. To obtain the measurement data for the roofline-plots, perfplot was used, together with libpcm 2.6.

There is a crucial difference between the performance plots and the roofline plots: For the latter, all floating point operations were counted, including comparisons, whereas

[3] http://perfmon2.sourceforge.net/

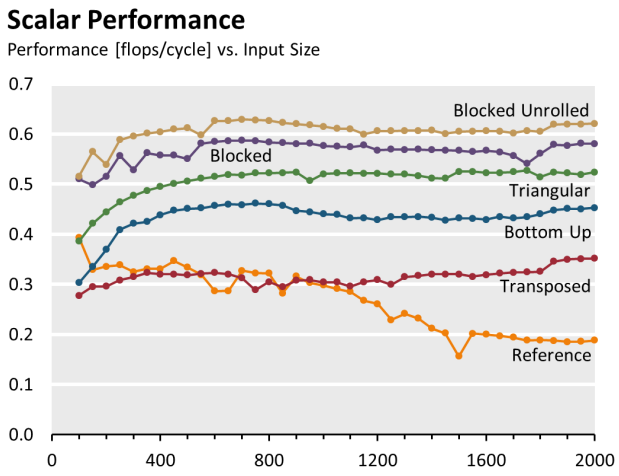

Fig. 6. Performance of the scalar implementations.

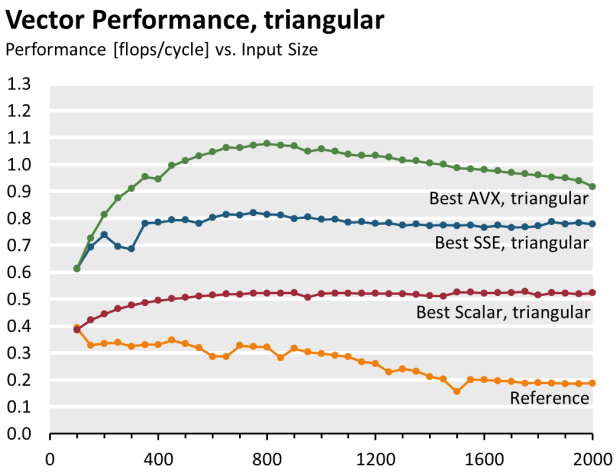

Fig. 7. Vectorized Performance based on *Triangle*.

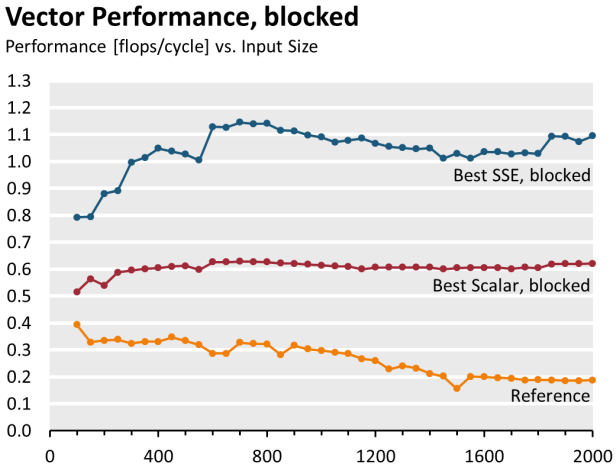

Fig. 8. Vectorization based on *Blocking*.

for the former, the number of additions from the cost analysis was used as a basis.

We used randomized input data for our computation. To have reproducible results, the seed was fixed.

Scalar Performance. Figure 6 shows the performance of the different scalar implementations. Peak performance lies at 1 fl/c. The reference implementation has a performance of 0.2 fl/c for large input sizes. Using the *transposed* approach improving spatial locality, we can already see stable performance for large input sizes that do no longer fit the last level cache. We can again see the expected performance boost when changing the access pattern of the table to *bottom-up*, thanks to further improved spatial locality. With a performance of about 4.5 fl/c, this corresponds to a performance gain of more than 2x to the baseline. Changing to the triangular memory layout improves performance to up to 0.5 fl/c. We assume this is due to the increased performance of the CPU prefetcher. However, we were not able to verify this. The *blocking* approach, based on the squared memory layout, gives another performance boost due to increased reuse in the CPU registers and fewer memory write-backs. The final scalar performance of our implementation lies at 62% peak performance which is equivalent to a performance gain of roughly 3x.

Vectorized Performance, triangular. Figure 7 shows the performance of our SSE and AVX vector code based on the *Triangle* scalar code. With 0.8 fl/c performance for the SSE code, the speedup through vectorization lies at 1.6x below the theoretical maximum of 2x. From SSE to AVX, the speedup lies at 1.2x compared to the expected 2x. Overall, the speedup from the baseline lies at 4.6x.

Vectorized Performance, blocked. Figure 8 shows the performance of our SSE vector code based on the *Blocked*

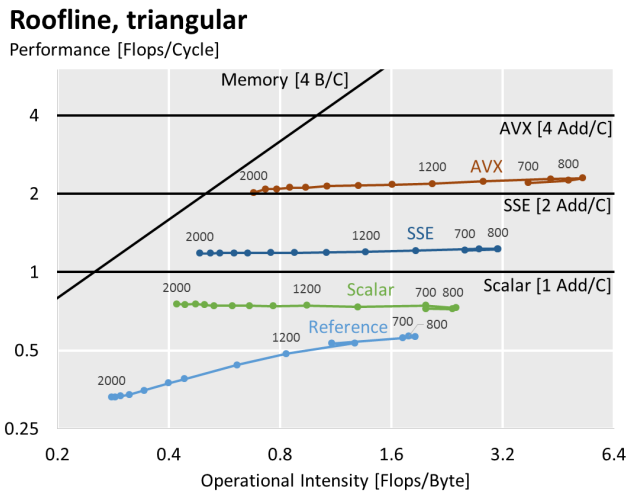

Fig. 9. Roofline Plot based on *Triangle*.

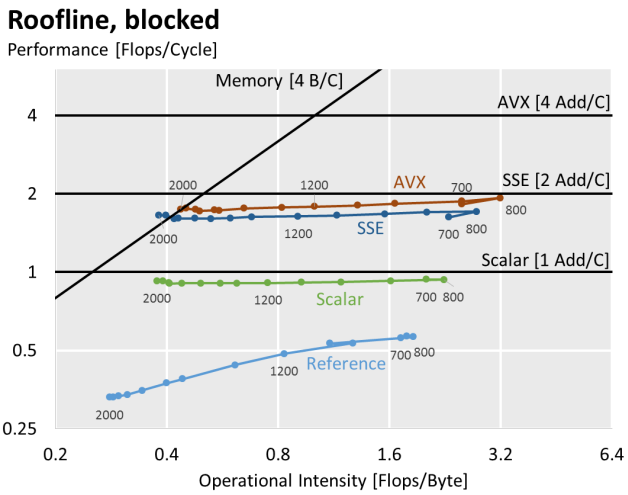

Fig. 10. Roofline Plot based on *Blocked*.

approach. The AVX variant performance worse and is thus omitted. Having a performance of about 1.1 fl/c, the SSE code yields a 1.8x speedup compared to the best scalar code and a 5.5x speedup to the baseline implementation. Note that the blocking SSE code performs better than the triangular AVX code.

Roofline Analysis. Figure 9 and Figure 10 show the roofline plots of our respective solutions. Note that for the roofline plots, libpcm was used to measure all floating point operations including comparisions. Thus, performance numbers are different than in the performance plots. We can see in Figure 10, scalar performance is almost at the peak of 1 fl/c. Also, the SSE code comes close to optimal. In contrast to the performance plots using our theoretical flop count, here the AVX variant performs slightly better than the SSE one. The blocked scalar and SSE versions are the fastest for their category. For the *Triangle* case in Figure 9, we have the best overall performance with the AVX variant yielding around 2 fl/c. This corresponds to an overall speedup achieved of roughly 6x.

5. CONCLUSIONS

We could demonstrate that with few relatively simple changes to the original algorithm, one can significantly improve the performance of an implementation. Since calculating tables by combining existing entries is inherent to dynamic programming, it is likely that the observations we made can be translated to other dynamic programming problems as well.

While in the case of the scalar implementations, very little is to gain by further optimizing the code, as we are close to the computational bound, the roofline-plots clearly indicate that in the case of vectorization, there is room for refinement.

A. REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, chapter 15.5, pp. 397 – 403, MIT Press, 3rd edition, 2009.

[2] D.E. Knuth, "Optimum binary search trees," *Acta Informatica*, vol. 1, no. 1, pp. 14–25, 1971.

[3] Kurt Mehlhorn, "Nearly optimal binary search trees," *Acta Informatica*, vol. 5, no. 4, pp. 287–295, 1975.