Lazy sorting algorithm

Toni Helminen

I. The idea

This algorithm is based on simple observation that in alpha-beta-algorithms [1] and such. *(“Lazy” means: We only sort when absolutely necessary)*. The branching factor is close to 1. So there’s no reason to sort everything. Since penalty is logarithmic; O(n!). Depending of game type. We have 50+ moves per random positions. With lazy sorting sorting algorithm we only sort 1 move. Not all. And if we get a cut-off or some other termination of the current node. We save a lot of time. Resulting in massive speedup.

This algorithm works with “staged” move generator too. Where we generate a chunk of moves. Which we still avoid sorting all at once. Hence hoping for speedup.

This paper explains general case for lazy sorting. Obviously alpha-beta [1] searchers are the most common applications for this.

II. Use case

Lazy sorting algorithm obviously requires good sorting. So the best moves must be put first so the branching factor stays close to 1. Use lazy sorting when early stoppage is probable.

III. Don’t use lazy sorting when

There’s no point using lazy sorting if you sort the whole list every time. If there’s hope for early termination. Lazy sorting results great speedups.

IV. Mathematics behind lazy sorting

Assume a list of N members.

Then formula of selection sort [2] in steps is: (N – 1) * (N / 2).

In lazy sorting with branching factor 1 it becomes: N – 1.

So in best case scenario we get a linear speedup: ((N-1)*(N/2)) / (N - 1) → N / 2.

In worst case scenario. We get a slowdown because the overhead of calling lazy sorting function all the time.

Here we see the worst case creates very branchy and slow code.

N = 100 results 10000 if-operations. 100 swaps.

N = 2000 results 4000000 if-operations. 2000 swaps.

Lazy sorting results 99 if-operations and 1 swap. When N = 100.

So lazy sorting saves ~10000 operations in the best case. When N = 100.

V. Slow pseudo-code

In this pseudo-code. We generate all moves. And then sort all of them. Resulting in loss of time. Due to lots of useless work done.

```
let moves = MoveGenerator();
SortAll(moves);
for (let i = 0; i < moves.size(); ++i) {
    …
}
```

VI. Good pseudo-code

In this pseudo-code. We generate all moves. And then sort only one move by wanted logic. Hoping for early cut-off. Note: *i* indicates the move to be sorted.

```
let moves = MoveGenerator();
for (let i = 0; i < moves.size(); ++i) {
    LazySort(moves, i);
    …
}
```

VII. LazySort() pseudo-code

This algorithm assumes all moves are valued.

```
LazySort(moves, nth) {
    let score = moves[nth].score;
    for (let i = 0; i < moves.size(); ++i) {
        if (moves[i].score > score) {
            score = moves[i].score;
            Swap(moves[nth], moves[i]);
        }
    }
}
```

VIII. Optimizations

Lazy sorting algorithm can be optimized in many ways.

If there’s no need to sort everything you can stop sorting with boolean variable.

If you generate moves in chunks. Which is messy code. But you can sort small chunks with selection sort [2]. And bigger chunks with lazy sorting. Profiling and testing tells the truth.

IX. Benchmarks

Here are some benchmarks sorting everything versus lazy sorting. My branching factor isn’t even close to the state-of-the-art algorithms. With better branching factor . The speedup will become greater.

Note 1: SortAll() implemented using C++’s fast std::sort(…) function. However all moves were sorted. Which isn’t optimum in this problem context.

Note 2: The speedup isn’t as great as it could be. Because in this context most of the time is spent in evaluation and only ~1% in actual sorting.

Summa summarum: Lazy sorting results a massive +10.6% improvement in search speed.

SortAll():

Nodes	245293839
Time(s)	75.005
Nodes Per Second	3270235

LazySort():

Nodes	271390722
Time(s)	75.008
Nodes Per Second	3618301

X. References

[1] https://www.chessprogramming.org/Alpha-Beta

[2] https://en.wikipedia.org/wiki/Selection_sort