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. And as little as possible)*. The branching factor is close to 1. So there’s no reason to sort everything at once. Since penalty is logarithmic; O(n!). Depending of problem space. We might have 40+ nodes (*random positions*). With lazy sorting algorithm we only sort node-by-node basis. Not everything at once. And if we get a cut-off or some other early loop termination. We save a lot of time. Resulting a massive speedup.

This paper explain a general case for lazy sorting.

II. Good use case

Lazy sorting algorithm obviously requires good sorting. So the best nodes must be put first so the branching factor stays close to 1. Use lazy sorting when early stoppage is probable. Lazy sorting don’t add any complexity or extra SLOC [3] to programs. It’s basically just 3 lines of really fast code.

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.

III. Bad use case

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

Even if you call LazySort() everytime. Overhead (slowdown) calling the function should be minimal.

Overhead of calling LazySort() everytime is almost non-existent. Measured with Python code.

LazySort()	
N	10 x 10,000
Time(s)	26.378

SelectionSort()	
N	10 x 10,000
Time(s)	28.241

IV. Mathematics

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.

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

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

N = 100 results 10,000 if-operations and 100 swaps.

N = 2,000 results 4,000,000 if-operations and 2,000 swaps.

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

So lazy sorting saves 10,000 operations in the best case. When N = 100.

V. Bad pseudo-code

In this pseudo-code. We generate all moves. And then sort them all. Resulting wasted time. Due to tons of operations 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 node-by-node basis. Hoping for early loop termination.

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

VII. LazySort() pseudo-code

This algorithm assumes all moves are scored. Which might not be optimum strategy depending of problem space. This algorithm swaps nodes for simplicity. Finding index and doing just 1 swap might be faster.

```
function LazySort(moves, nth) {
    for (let i = nth + 1; i < moves.size(); ++i)
        if (moves[i].score > moves[nth].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 a boolean variable.

If you generate moves in chunks. You can sort small chunks with selection sort [2]. And bigger chunks with lazy sorting. Profiling and testing tell the truth.

IX. Benchmarks

Benchmarks std::sort() / SelectionSort() vs LazySort(). Run at the same time for fair comparison. Here branching factor isn't even close to 1. With better branching factor. Speedup would be much better.

Summa summarum: LazySort() vs SelectionSort() → 2% speedup. LazySort() vs std::sort() → 11% speedup.

LazySort()	
Nodes	544,437,237
Time(s)	150.008
Nodes / second	3,629,388

SelectionSort()	
Nodes	535,690,617
Time(s)	150.010
Nodes / second	3,571,032

std::sort()	
Nodes	490,725,631
Time(s)	150.005
Nodes / second	3,271,395

X. References

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

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

[3] https://en.wikipedia.org/wiki/Source_lines_of_code