Task-scheduler: Discussion of Sorting

Timothy Burchfield

April 27, 2018

Motivation

Comparison sorting algorithms order a list of based on how they compare.

A *topological* sort aims to do a similar, but different operation. Given a digraph, it returns a sequence of nodes such that for all edges (u, v) in the digraph, v appears after u in the sequence. *(If a cycle is found, an error is returned.)*

Comparison sorting has many diverse applications, but topological sorting is usually used for arranging tasks that could be dependent on one another. For example, topological sorting is used to decide the order a Makefile executes in.

However, there is not an obvious way to do a combination of these two "types" of sorts.

Our project implements this kind of combined sort. Stated more precisely, we sort items so that a is not dependent (directly or indirectly) on any items to the right of a, and all items b to the left of a either are less than or equal to a, or there exists an item c less than or equal to a that depends on b, but not a.

(We also return an error if a cycle is found)

Note, this is not as simple as performing one type of sort then the other, since each kind of sort could "ruin" the ordering of the other.

Just for convenience, we will call this kind of sorting problem TwoSort .

We ran into TwoSort while working on our project, so it has direct application to the goal of our project:

Given tasks that could depend on other tasks and have due dates, order the tasks such that for any T_1, it doesn't depend directly or indirectly on anything to the right of it, and all tasks to the left are either due as early or earlier, or are depended on by a task that is due earlier that doesn't depend on T_1.

In a way, this can be thought of as the "optimal" order to complete said tasks. The first task on the list is the one that has the next due date, or the earliest due task that the next due task is dependent on, and so on and so forth.

Algorithm

This algorithm works similar to DFS-based topological sorts. Critical to its correctness is the order in which Nodes are processed. Nodes are processed in sorted order, but also down dependency trees.

This is the pseudocode for the algorithm we use:

Algorithm 1: DSort

```
input  : A digraph G = (V, E) such that vertices can be compared
output: A sorted list of vertices that satisfy TwoSort as described
        above.
begin
    /* We first sort based on comparison of vertices:      */
    S ← SORT(V);
    Sorted ← empty list;
    Finished ← empty set;
    for s ∈ S do
        /* Check if node already processed                 */
        if s ∈ Finished then
            continue;
        end
        Seen ← empty set;
        PSort ← empty list;
        c ← s;
        /* Recursively do the following:                   */
        procedure
            if c ∈ Finished then
                continue;
            end
            Add c to Seen and Finished;
            for (c, u) ∈ E in sorted order do
                if u ∈ Seen then
                    return Error: Circular ;
                end
                if u ∈ Seen then
                    continue;
                end
                Recurse on this block with c as u. Prepend c to PSort
            end
            Append PSort to the end of Sorted.
        end
    end
end
return Sorted
```

Correctness

We hope to show our "DSort" solves TwoSort .

Part of this definition is DSort's result being a correct topological sort. This is equivalent to (B directly or indirectly depends on A) $\implies$ (B is to the right of A).

During computation, each unprocessed s (indicated by not being a member of Finished) will be processed by adding it to Finished and Seen, then processing all unprocessed nodes that it depends on, then adding it to the start of the temporary list PSort. This is done in the recursive block.

(This block also checks for cycles, and errors appropriately in the case of finding one.)

Thus a node will only be fully processed once all of it's children are fully processed. In other words, a node is necessarily processed after its dependencies.

Thus no nodes will be processed after the currently unprocessed s that are depended on by nodes before the current s. Additionally, a node will be added to the temporary list to the left of its dependencies (by the structure of the recursive call). This temporary list is appended to the final list, so a node will be to the left of its dependencies in the returned list.

Every node must be processed, since in the first for loop, every unprocessed node is processed.

Since all nodes are processed, and all nodes are to the left of their dependencies, DSort makes a correct topological sort. In other words, after the sort, (B is dependent on A) $\implies$ (A is to the left of B).

Now let's show (A is to the left of B) $\implies$ (($A \leq B$) $\vee$ ($\exists C$ that is directly or indirectly dependent on A, and that $C \geq B$, and C is not dependent in any way on B)).

By the contrapositive, this is equivalent to showing that if $B < A$ and no C exists that satisfies the above, B is to the left of A.

If B is less than A, B began to the left of A (since we began by sorting V). Since no C exists as described above, A is unprocessed when B is done being processed. This means B is appended to the final sorted list before A. This means B is to the left of A.

(Note, were we not to include the last phrase, stipulating C must not be dependent on B, this would be a weaker claim, and would allow the ordering to place dependencies of the currently processed item in the wrong order in comparison to each other. This is why we move over the vertices adjacent to c in the recursive block in order. These are sorted. We manage this efficiently in our implementation by representing the graph in *sorted* adjacency list form.)

Complexity

The initial sorting of vertices can be done in $O(Vlog(V))$ time by any off-the-shelf comparison sort.. The subsequent for-loop runs over every Node. Since the

recursive procedure is only called on unprocessed nodes, it can only be entered $O(V)$ times total (not per iteration of the outermost for-loop – *total*). Since the for-loop within the recursive procedure is only encountered at most once per node, and iterates over each adjacent node, similar to Kahn's algorithm, the contents of this inner for-loop does not run $O(E)$ times every time it is encountered, but $O(E)$ times *total*.

In summary, the sort takes $O(Vlog(V))$ time. The contents of the outermost for-loop runs $O(V)$ times total. The contents of the recursive procedure run $O(V)$ times total. The contents of the innermost for-loop runs $O(E)$ times total. Thus, DSort runs in $O(Vlog(V) + E)$ time.

Optimality

The restraints of TwoSort imply our algorithm, running in $O(Vlog(V) + E)$ time, is optimal.

Consider if there existed an algorithm $\mathcal{A}$ solving TwoSort that ran asymptotically faster with respect to V. We can reduce comparison sorting to TwoSort with the following algorithm, $\mathcal{B}_{\mathcal{A}}$:

Given a sequence of items to be comparison sorted $X = x_0, x_1 \dots x_n$:

1. Construct a graph $G = (X, \varnothing)$.
2. Run $\mathcal{A}$ on G, and return what $\mathcal{A}$ returns.

Since step 1 can clearly be done in $O(n)$ time, and by our assumption step 2 can be done in better than $O(nlog(n))$ time, $\mathcal{B}_{\mathcal{A}}$ runs in better than $O(nlog(n))$ time. Moreover, B_A solves comparison sorting, in better than $O(nlog(n))$ time (TwoSort only allows element comparison).

But this is a contradiction! Comparison sorting cannot be solved in better than $O(nlog(n))$ time.[1]

Thus our algorithm is optimal with respect to V.

Consider if there existed an algorithm $\mathcal{A}$ that ran asymptotically faster with respect to E. Since DSort runs in $O(E)$ time, this means there is a number of edges such that for all graphs with more than this amount of edges, $\mathcal{A}$ cannot examine all of the edges. Let's call this number k. Make a graph G with $k + 1$ vertices, $V = v_0, v_1 \dots v_k$. The edges are just (v_i, v_{i+1}) for all i from 0 to $k - 1$. This means $\mathcal{A}$ cannot examine all edges.

Note, the right answer for TwoSorting G is $v_k, v_{k-1}, \dots, v_1, v_0$, and this is only true because every edge we have specified is there, if any edge were gone, this would not be the correct answer.

[1] https://www.cs.cmu.edu/ avrim/451f11/lectures/lect0913.pdf

Since $\mathcal{A}$ cannot look at all of the edges of G, there is an edge $\mathcal{A}$ does not look at.

Create G' where G' is identical to G except that it lacks this edge. The right answer for G' is never the right answer for G. However, since $\mathcal{A}$ never looks for that edge in G, and G' is identical except for the nonexistence of that edge, $\mathcal{A}$ must answer identically for G and G', meaning $\mathcal{A}$ answers incorrectly for G or G'.

Thus there cannot exist an algorithm solving TwoSort in time better than $O(E)$ w/r/t E.

Thus our algorithm is optimal with respect to E.

This means our algorithm is optimal.