A Proven-Correct Topological Sort

Bart Massey

2016-05-11

Given a consistent partial order of elements of a set, you can always find a total order of the elements of that set that obeys the partial order. Such a total order is known as the *topological sort* of that set. It is usually described in terms of *precedences* between pairs of items.

Topological Sortedness

Let there be a set of items to be sorted

$$[\mathit{ITEM}]$$

and a pairwise precedence relation $\prec$ between items with no item directly or indirectly preceding itself.

$$\mathbf{relation}\,(\_ \prec \_)$$

$$\begin{array}{|l}
\_ \prec \_ : \mathit{ITEM} \leftrightarrow \mathit{ITEM} \\
\hline
\forall x : \mathit{ITEM} \bullet (x \mapsto x) \notin (\_ \prec \_)^{+}
\end{array}$$

We want to produce a total order between the items: the easiest way to think about this is as a sequence that is a permutation of the items.

$$\begin{array}{|l}
[X] \\
\mathit{permutations} : \mathbb{P}\,X \rightarrow \mathbb{P}\,\mathrm{seq}(X) \\
\hline
\forall s : \mathbb{P}\,X \bullet \forall p : \mathit{permutations}(s) \bullet \\
\quad \# p = \# s \land \mathrm{ran}(p) = s
\end{array}$$

We will say a sequence is topologically sorted if each item in the sequence occurs at most once and the precedence relation is never "backward" from the sequence order. Note that for a given set and precedence relation there may be many topologically sorted sequences.

$$\begin{array}{|l}
\mathit{TSORT} : \mathbb{P}\,\mathrm{seq}(\mathit{ITEM}) \\
\hline
\forall t : \mathit{TSORT} \bullet \\
\quad t \in \mathit{permutations}(\mathrm{ran}(t)) \land \\
\quad \neg\, \exists i_1, i_2 : \mathrm{dom}(t) \mid i_1 < i_2 \bullet \\
\quad\quad t(i_2) \prec t(i_1)
\end{array}$$

We are looking for a sequence that is topologically sorted and covers the entire set of items.

$$
\begin{array}{|l}
\mathit{tsorts} : \mathbb{P}\,\mathit{TSORT} \\
\hline
\forall\, t : \mathit{tsorts} \bullet \\
\qquad \operatorname{ran}(t) = \mathit{ITEM}
\end{array}
$$

The Source-To-Sink Method

This is a nice definition, but it leads to the question of how one might find such a sequence. The algorithm we will give here is known as *source-to-sink*, and is one of the oldest topological sort algorithms. Our implementation will not be at all efficient, but will be provably correct.

Algorithm

The state of the algorithm will be in two parts: a prefix of the final topological sort, and a set of items remaining to be sorted. All items must occur exactly once in exactly one of these sets.

$$
\begin{array}{|l}
\hline
\mathit{STS} \\
\mathit{sorted} : \mathit{TSORT} \\
\mathit{remaining} : \mathbb{P}\,\mathit{ITEM} \\
\hline
\operatorname{ran}(\mathit{sorted}) \cup \mathit{remaining} = \mathit{ITEM} \\
\operatorname{ran}(\mathit{sorted}) \cap \mathit{remaining} = \varnothing \\
\hline
\end{array}
$$

The algorithm starts with an empty sequence and the full range of candidate items available.

$$
\begin{array}{|l}
\hline
\mathit{InitSTS} \\
\mathit{STS} \\
\hline
\mathit{sorted} = \langle\rangle \\
\mathit{remaining} = \mathit{ITEM} \\
\hline
\end{array}
$$

At each step, the algorithm appends to the sequence some item x such that all of x's predecessors are already in the sequence but x is not.

$$
\begin{array}{|l}
\textit{StepSTS} \\
\hline
\Delta \textit{STS} \\
x : \textit{ITEM} \\
\hline
x \in \textit{remaining} \\
\neg \exists y : \textit{remaining}' \bullet \\
\qquad y \prec x \\
\textit{remaining}' = \textit{remaining} \setminus \{x\} \\
\textit{sorted}' = \textit{sorted} \frown \langle x \rangle \\
\hline
\end{array}
$$

When it has constructed a total permutation, the algorithm stops and outputs it.

$$
\begin{array}{|l}
\textit{EndSTS} \\
\hline
\textit{STS} \\
\textit{result}! : \textit{TSORT} \\
\hline
\textit{remaining} = \varnothing \\
\textit{result}! = \textit{sorted} \\
\textit{result}! \in \textit{tsorts} \\
\hline
\end{array}
$$

The complete STS algorithm just initializes its state, then iterates until it has exhausted the items.

$$\textit{AlgSTS} == \textit{InitSTS}\,' \mathbin{⨟} \textit{StepSTS} \mathbin{⨟} \textit{EndSTS}$$

Correctness

We should show that our topological sort algorithm is partially correct: that the result is in fact a complete topological sort of the items. We should also show that the algorithm always terminates. These properties together will prove the total correctness of the STS algorithm.

Partial Correctness

To show partial correctness, we concentrate on the state invariant that is enforced by the state schema: *sorted* is a topological sort and its range plus *remaining* forms a partition of the items. Note that this invariant is trivially satisfied in the initial state. If the invariant is satisfied in the final state, the result must be a topological sort. It remains to verify that each step of the algorithm preserves the invariant.

We want to show that

$$\forall\, StepSTS \bullet \theta\, STS \in STS \Rightarrow \theta\, STS' \in STS$$

This is going to be a bit tedious to do formally, since there's a lot of expansion going on. After performing it, we arrive at something like

$$\begin{array}{l}
\#\, sorted = \#\, \mathrm{ran}(sorted) \ \wedge \neg\ \exists\, i_1, i_2 : \mathrm{dom}(sorted) \mid i_1 < i_2 \bullet \\
\quad sorted(i_2) \prec sorted(i_1) \ \wedge \\
\mathrm{ran}(sorted) \cup remaining = ITEM \ \wedge \\
\mathrm{ran}(sorted) \cap remaining = \varnothing \ \wedge remaining' = remaining \setminus \{x\} \ \wedge sorted' = sorted \frown \langle x \rangle \ \Rightarrow \\
\#\, sorted' = \#\, \mathrm{ran}(sorted') \ \wedge \neg\ \exists\, i_1, i_2 : \mathrm{dom}(sorted') \mid i_1 < i_2 \bullet \\
\quad sorted'(i_2) \prec sorted'(i_1) \ \wedge \mathrm{ran}(sorted') \cup remaining' = ITEM \ \wedge \\
\mathrm{ran}(sorted') \cap remaining' = \varnothing
\end{array}$$

Taking the right-hand-sides of the implication one at a time, one can quickly crank through the proofs:

- The *sorted'* relation is still a permutation:

  $$\#\, sorted' = \#\, \mathrm{ran}(sorted')$$

  Since we know x was not in $\mathrm{ran}(sorted)$, it must be the case that

  $$\#\, \mathrm{ran}(sorted') = \#\, sorted' = \mathrm{ran}(sorted) + 1$$

- The *sorted'* relation still obeys the total order:

  $$\neg\ \exists\, i_1, i_2 : \mathrm{dom}(sorted') \mid i_1 < i_2 \bullet \qquad sorted'(i_2) \prec sorted'(i_1)$$

  Since we know this property held for *sorted*, the only element that could violate it now is x ($i_2 = \#\, sorted'$). But by construction, we know that x can precede no element in $\mathrm{ran}(sorted)$.

- No items have been lost:

  $$\begin{array}{l} \mathrm{ran}(sorted') \cup remaining' = ITEM \\ \mathrm{ran}(sorted') \cap remaining' = \varnothing \end{array}$$

  We know this property held before, and we moved x from *remaining* to *sorted'*. It is straightforward to see that it still holds.

```
from sys import stdin

item = set()
predec = set()

for l in stdin:
    xs = l.split()
    assert len(xs) == 2
    [x1, x2] = xs
    item.add(x1)
    item.add(x2)
    predec.add((x1, x2))

# InitSTS
sorted = []
remaining = set(item)

# StepSTS
while remaining:
    def find_next():
        def check(x):
            for y in remaining:
                if (y, x) in predec:
                    return False
            return True

        for x in remaining:
            if check(x):
                return x
        assert False

    x = find_next()
    for y in remaining:
        assert (y, x) not in predec
    remaining.remove(x)
    sorted.append(x)

# EndSTS
for x in sorted:
    print(x)
```

Figure 1: STS `tsort` in Python

Total Correctness

To see that the algorithm is totally correct, we must show that it completes on any valid input. The two ways that it could not complete are to fail early or to infinite loop.

The only way the algorithm can fail early is if at some *StepSTS* we have that $remaining \neq \varnothing$ but

$$\neg\ \exists\, x : remaining \bullet \qquad \neg\ \exists\, y : remaining' \bullet \qquad\qquad y \prec x$$

However, given that the elements of *remaining* obey a partial order, there is always a minimal element. (Proof left as an exercise.)

The only way the algorithm can nonterminate is if *remaining* never becomes empty. However, each *StepSTS* removes an element from *remaining*. Thus, *remaining* must eventually become empty.

Implementation

Once we have a verified correct algorithm, it remains to construct a program implementing it. The Python program of Figure 1 implements the UNIX `tsort` command using the STS algorithm specified here.

For the most part, no proof of correctness is required here: the code is correct by construction. The exception is the code involved with choosing an x. This code needs a proof that it always selects a valid value. (Proof left as an exercise.) It helps that Python provides reasonable data types for this implementation, including native set, tuple and sequence types.

Inspection and Testing

It is rarely enough just to have proved code correct. Before executing it for the first time, the code of Figure 1 was carefully read, and a simple randomized test generator was constructed. The test generator imposes all the local precedences on natural numbers in the range $1 \mathinner{..} k$ for some k, in random order.

The code passed multiple runs of this test. We regard inspection and testing as an important process, since proofs can have bugs.

Performance

Unfortunately, on an input precedence relation with just 1000 tuples, the `tsort` implementation of the previous sections takes about 10 seconds on a modern box. (Yes, Python is slow: using PyPy to JIT the code results in about a 3 second runtime.)

To see why the performance might be poor, it is worth doing some complexity analysis. Let us say that $n = \# \mathit{ITEM}$ and $m = \#(\_ \prec \_)$. Note that in our application, $n \leq m$, since the set of items is being derived from the tuples of the precedence relation.

The outer `while` loop will run n times. Each time through the loop, `find_next()` will run $O(n)$ times in the worst case, calling `check()` each time. `check()` takes $O(m)$ steps to check the precedence relation in the worst case. The overall worst-case complexity of the algorithm is thus $O(mn^2)$. This is pretty bad.

Kahn's Method: A Faster Topo Sort

A better topological sort algorithm would give us right answers faster. Let's try a different approach. As before, we will extend a topological sort until it contains all the items. The algorithm given here is known as Kahn's Algorithm and dates back to 1962.

Algorithm

We will group the items as a function (dictionary, array) mapping to the set of successors. This is known as an *adjacency list* representation of the partial order graph.

$$suc : \mathit{ITEM} \rightarrow \mathbb{P}\, \mathit{ITEM}$$

$$\forall\, x : \mathit{ITEM} \bullet$$
$$\quad suc(x) = \{ y : \mathit{ITEM} \mid x \prec y \}$$

We can compute, up front, the number of items preceding each item in the relation. Let us store this as a function (dictionary, array). We will place available items on the end of the sorted list, and keep a *todo* index of which items still need their successors processed.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Kahn}} \\ \hline
\textit{sorted} : \textit{TSORT} \\
\textit{todo} : \mathbb{N}_1 \\
\textit{count} : \textit{ITEM} \rightarrow \mathbb{N} \\ \hline
\operatorname{ran}(\operatorname{ran}(\textit{sorted}) \lhd \textit{count}) = \{0\} \\ \hline
\end{array}
$$

To start, we properly initialize all the counts, set the sorted sequence to contain (in any order) the set of items with zero count, and mark the whole sequence as *todo*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitKahn}} \\ \hline
\textit{Kahn} \\ \hline
\forall x : \textit{ITEM} \bullet \\
\qquad \textit{count}(x) = \#((\_ \prec \_) \rhd \{x\}) \\
\textit{sorted} \in \textit{permutations}(\{x : \textit{ITEM} \mid \textit{count}(x) = 0\}) \\
\textit{todo} = 1 \\ \hline
\end{array}
$$

Now we take a series of steps. In each step, we examine the next *todo* item in the sequence and decrement the counts of its successors, in the process adding now-ready successors to the sorted list for processing. Finally, we increment *todo*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StepKahn}} \\ \hline
\Delta \textit{Kahn} \\ \hline
\textit{todo} \leq \# \textit{sorted} \\
\textit{count}' = \textit{count} \oplus \{y : \textit{suc}(\textit{sorted}(\textit{todo})) \bullet y \mapsto \textit{count}(y) - 1\} \\
\exists p : \textit{permutations}(\operatorname{dom}(\textit{count}' \rhd \{0\}) \cap \operatorname{dom}(\textit{count} \rhd \{1\})) \bullet \\
\qquad \textit{sorted}' = \textit{sorted} \frown p \\
\textit{todo}' = \textit{todo} + 1 \\ \hline
\end{array}
$$

When we run out of things to do, we will have all the counted items in sorted order.

$$\begin{array}{|l}
\hline
\mathit{EndKahn} \\
\mathit{Kahn} \\
\mathit{result!} : \mathit{tsorts} \\
\hline
\mathit{todo} > \#\,\mathit{sorted} \\
\operatorname{ran}(\mathit{count}) = \{0\} \\
\mathit{result!} = \mathit{sorted} \\
\hline
\end{array}$$

The overall flow is the same as with $AlgSTS$:

$$\mathit{AlgKahn} == \mathit{InitKahn}' \fatsemi \mathit{StepKahn} \fatsemi \mathit{EndKahn}$$

Correctness

This correctness proof sketch mirrors the one for the previous algorithm. It is arguably easier than the previous one, because the invariant is simpler to maintain.

Partial Correctness

To see that this algorithm is partially correct, we follow essentially the same procedure as before. Calculating all the constraints for $StepKahn$, we find that we need to ensure that

$$\begin{array}{l}
\mathit{sorted} \in \mathit{TSORT} \ \land \\
\operatorname{ran}(\operatorname{ran}(\mathit{sorted}) \lhd \mathit{count}) = \{0\} \ \land \\
\mathit{todo} \leq \#\,\mathit{sorted} \ \land \\
\mathit{count}' = \mathit{count} \oplus \{y : \mathit{suc}(\mathit{sorted}(\mathit{todo})) \bullet y \mapsto \mathit{count}(y) - 1\} \ \land \\
(\exists\, p : \mathit{permutations}(\operatorname{dom}(\mathit{count}' \rhd \{0\}) \cap \operatorname{dom}(\mathit{count} \rhd \{1\})) \bullet \\
\qquad \mathit{sorted}' = \mathit{sorted} \frown p) \ \land \\
\mathit{todo}' = \mathit{todo} + 1 \ \Rightarrow \\
\operatorname{ran}(\operatorname{ran}(\mathit{sorted}') \lhd \mathit{count}') = \{0\} \ \land \\
\mathit{sorted}' \in \mathit{TSORT}
\end{array}$$

- To establish that the count of each $sorted'$ item is zero:

  $$\operatorname{ran}(\operatorname{ran}(\mathit{sorted}') \lhd \mathit{count}') = \{0\}$$

  it is sufficient to note that the old items of $sorted'$ already had the property, and the newly-added items must, since they are constructed in such a way that $count'$ is necessarily 0.

- To establish that $sorted'$ is still a topological sort of its elements:

  $$\mathit{sorted}' \in \mathit{TSORT}$$

  we first note that it cannot be the case that the newly-added items should have come before any item in $sorted$: all items x in $sorted$ have $count(x) = 0$, so they cannot have any unresolved predecessors. Similarly, we note that all the newly-added items x have $count'(x) = 0$, so they cannot have any not-yet-added predecessors.

```
from sys import stdin

item = set()
suc = dict()

for l in stdin:
    xs = l.split()
    assert len(xs) == 2
    [x1, x2] = xs
    item.add(x1)
    item.add(x2)
    if x1 not in suc:
        suc[x1] = set()
    if x2 not in suc:
        suc[x2] = set()
    suc[x1].add(x2)

# InitKahn
count = dict()
for x in item:
    count[x] = 0
for x in item:
    for y in suc[x]:
        count[y] += 1
sorted = []
for x in item:
    if count[x] == 0:
        sorted.append(x)
todo = 0

# StepKahn
while todo < len(sorted):
    x = sorted[todo]
    assert count[x] == 0
    for y in suc[x]:
        count[y] -= 1
        if count[y] == 0:
            sorted.append(y)
    todo += 1

# EndKahn
assert len(sorted) == len(item)
for x in sorted:
    print(x)
```

Figure 2: Kahn `tsort` in Python

Total Correctness

To see that the algorithm terminates, it is sufficient to note that $\# \mathit{sorted}$ never exceeds $\# \mathit{ITEM}$, but at each step *todo* increases by one. *StepKahn* must eventually terminate, since *todo* must increase to exceed $\# \mathit{sorted}$.

Implementation

The Python program of Figure 2 implements the UNIX `tsort` command using the Kahn algorithm specified here.

It is notable that a trivial but not syntactical bug was found in the initial implementation. When setting up the adjacency list, rather than following the literal description in the specification, an inline algorithm was used to initialize the list: this algorithm forgot to create empty successor sets for sink nodes (with no successors). The proven code worked fine on the first try.

Performance

To see why the performance might be better, it is again worth doing some complexity analysis. We will assume for convenience that set insertion, set membership, dictionary lookups and the like can be done in time $O(1)$.

The input loop takes time $O(m)$, since it processes each edge. *InitKahn* takes time $O(m)$, since it is dominated by the inner loop that increments `count`. *EndKahn* takes time $O(n)$ just to print out.

This leaves *StepKahn*. Here, it appears that the dominant work is also the $O(m)$ total cost of executing this operation to completion: each edge in `predec` is processed exactly once. `y`. (However, interestingly, there's also the cost of computing `len(sorted)` at each iteration. We assume for now that Python caches this length, but this should be checked.) This makes the overall worst-case cost $O(m)$. This is a vast improvement over the $O(mn^2)$ cost of the earlier algorithm.

In practice, the runtime of this implementation on one million precedence edges is roughly the same as that of the original implementation on one thousand edges. This is a dramatic speedup, especially since it does not apparently compromise correctness.

Conclusion and Future Work

There is still a lot more to do here. In particular, the extremely sketchy proof sketch of section 3.2.1 should be formalized and validated, arguably by machine.

That said, this work argues that it is in fact feasible to produce correct code for algorithms for real problems. Further, a library of such algorithms would be a real help in the construction of larger correct systems.