Topological sort is the process of finding a total ordering consistent with a partial ordering.

In set theory, a relation on a set $S$ is a subset $R$ of the Cartesian product $S \times S$. Commonly infix notation is used for relations: instead of $(x, y) \in R$ one writes $x < y$ or something of the sort. The set $R$ consists of the pairs $(x, y)$ for which the relation holds. Outside of set theory, the set $R$ and the relation $<$ are distinguished: $<$ is called the relation and $R$ is called the graph of this relation.

Often the language of graph theory is used to describe relations. In graph theory, the set $S$ is identified with nodes of a graph and the set $R$ with the edges of the graph: $(x, y) \in R$ means there is a directed edge from the node $x$ to the node $y$. The graph is sometimes said to be a directed graph to indicate that all the edges are directed.

A strict partial ordering has the properties of being antisymmetric and transitive. Antisymmetry is the property

\[(x, y) \in R \text{ implies } (y, x) \notin R\] (1)

Transitivity is the property

\[(x, y) \in R \text{ and } (y, z) \in R \text{ implies } (x, z) \in R\] (2)

Note that (1) implies $(x, x) \notin R$, so that is why these properties axiomatize a strict partial ordering (like $<$ rather than $\leq$).

The language of graph theory that corresponds to strict partial ordering is acyclicity. A directed graph with node set $S$ and edge set $R$ is acyclic if there does not exist a sequence of nodes $x_0, \ldots, x_k$ such that

\[(x_i, x_{i+1}) \in R \text{ for } i = 0, \ldots, k - 1 \text{ and } (x_k, x_0) \in R\]

An acyclic graph automatically has the antisymmetry property (1). It does not necessarily have the transitivity property (2), but it can be extended
by transitivity to make a partial ordering. Let $Q$ be the smallest subset of $S \times S$ containing $R$ and satisfying (2). There exists such a set because the intersection of sets satisfying (2) also satisfies (2). We have $(x_0, x_k) \in Q$ if and only if there exists a sequence of nodes $x_0, \ldots, x_k$ such that

$$(x_i, x_{i+1}) \in R \quad i = 0, \ldots, k - 1$$

because if there exists such a sequence we must have $(x_0, x_k) \in Q$ by transitivity and, conversely, if there does not exist such a sequence then the set $Q \setminus \{(x_0, x_k)\}$ would be a transitive superset of $R$.

Thus every directed acyclic graph (DAG) corresponds to a unique partial ordering determined by extension by transitivity. A topological sort is usually considered to start with a DAG and find a total ordering consistent with the unique strict partial ordering determined by the DAG.

If the graph has $n$ nodes and $e$ edges, it is clearly impossible to do a topological sort using less than $O(n + e)$ time and $O(n + e)$ space. We seek an algorithm that does no worse than that.

One algorithm having this property uses depth first search (DFS) as a subroutine. We say a node of the graph is initial if it has no incoming edges and is terminal if it has no outgoing edges. A DFS visits each initial node (in any order). In the process of visiting a node $x$, DFS also does a DFS of all nodes $y$ such that $(x, y)$ is an edge of the graph. Thus DFS is a recursive algorithm, ideally suited to languages like R and C that allow functions to call themselves. Thus the following code does DFS

```plaintext
foreach initial node $x$ do
  dfs($x$)
end foreach
```

where the DFS subroutine is defined by

```plaintext
function dfs($x$)
{
  if ($x$ has already been visited)
    return
  if ($x$ is in the process of being visited)
    error(the graph is not acyclic)
  foreach $y$ such that $(x, y) \in R$ do
    dfs($y$)
  end foreach
}
```
where “x has already been visited” means the dfs function has previously been called with the same argument x and finished (so does not need to be done again) and where “x is in the process of being visited” means the call to the dfs function has previously been called with the same argument x and has not yet finished. We can keep track of this by recording a status for each node, which is initially 0, set to 1 at the top of the dfs function, and set to 2 at the bottom. Then the dfs function would be defined by

```c
function dfs(x)
{
    if (status[x] = 2)
        return
    if (status[x] = 1)
        error(the graph is not acyclic)
    status[x] := 1
    foreach y such that (x, y) ∈ R do
        dfs(y)
    end foreach
    status[x] := 2
    output x
}
```

where = denotes logical equals (what is == in R and C), where := denotes assignment (what is <- in R and = in C), and where “output x” means put node x in the growing vector of topologically sorted nodes.

This algorithm works because when we output x we have already output each node that can be reached from x by following edges of the graph (Aho, Hopcroft and Ullman, 1983, Sections 6.6 and 6.7).

In order to not have to deal with R structures in C, we insist that the domain of the relation (the set of nodes of the graph) be 1, ..., n for some positive integer n. To allow the from and to arguments of the R function to have arbitrary atomic types (type "character", for example) we use the following code to set up the arguments for our C function

```c
cfrom <- match(from, domain)
cto <- match(to, domain)
cdomain <- seq(along = domain)
```

where from, to, and domain are all elements of the same atomic type and
all(! is.na(domain))
all(from %in% domain)
all(to %in% domain)

are all TRUE. This use of the match function does not violate our requirement to stay within \(O(n + e)\) space and time because match uses a hash table and hence is \(O(n)\) space and time.

We can find initial nodes of the graph in \(O(n)\) space and time, going once over the set of nodes to count incoming edges, going a second time over the set of nodes to find the initial ones, which have zero incoming edges.

The only tricky bit is that in order for the loop

\[
\text{foreach } y \text{ such that } (x, y) \in R \text{ do }
\text{dfs}(y)
\text{end foreach}
\]

to take only \(O(k)\) time, where \(k\) is the number of \(y\) such that \((x, y) \in R\), we must have previously set up a list of such nodes. We can make lists of all outgoing nodes for all nodes in \(O(e)\) space and time by using linked lists. Something like the following. Before defining any functions define the structure

\[
\text{struct nodelist} \{
\text{int item};
\text{struct nodelist } \ast \text{next};
\};
\]

Then inside the tsort function

\[
\text{struct nodelist } \ast \text{outgoing}[n];
\text{struct nodelist buffer}[e];
\]

reserve space for heads of lists of outgoing edges for each node and for items in the lists, where, as above, \(n\) is the number of nodes and \(e\) is the number of edges. Then

\[
\text{for (int } i = 0; i < n; i++)
\text{outgoing}[i] = \text{NULL;}
\text{for (int } i = 0; i < e; i++) \{
\text{register int myfrom } = \text{from}[i] - 1;
\text{register int myto } = \text{to}[i] - 1;
\text{buffer}[i].item } = \text{myto;}
\text{buffer}[i].next } = \text{outgoing}[\text{myfrom}];
\text{outgoing[myfrom]} } = \&\text{buffer}[i];
\}
sets up the lists. So in the dfs function, one can run over the lists using the usual C idiom for iterating over lists

for (struct nodelist *p = outgoing[i]; p != NULL; 
    p = p->next) {
    do what has to be done to node p->item
}

References