CS 163 & CS 265: Graph Algorithms
Week 2: Spanning trees and DAGs
Lecture 7: Spreadsheets and topological ordering

David Eppstein
University of California, Irvine

Fall Quarter, 2022
Spreadsheets
Spreadsheets

Examples: Excel, Visicalc, Google sheets

Array of cells, each containing some information:

A static string or a number

A formula for computing the cell value from other cells
What happens when you change one cell?

Each formula cell also stores its value, used when the spreadsheet is displayed or printed.

The formula for this value may use values from other cells.

When any of those other cells changes, we need to recompute the formula value.

Changes its value, may cause other formulas to be recomputed.

But if many formula values need recomputing, what order should we use?

Top-down left-to-right can be very bad for spreadsheets where information propagates bottom-up.
Natural order recalculation

Choose a special ordering of spreadsheet cells such that:

- If any cell \( x \) has a formula that uses the value of another cell \( y \), then \( x \) is later than \( y \) in the ordering.
- Therefore, \( y \)'s value will have already been recalculated by the time it is needed to recalculate \( x \).
- If we can order cells in this way, we can propagate changes to all cells that depend on the changed cell (directly or indirectly) while recalculating each cell at most once.

Can be applied either to whole spreadsheet, or to only the subset of cells that need recalculation.
Algorithms for this problem have been known and published since 1963.

Despite that, Pardo and Landau filed for a patent in 1970, and (after a long process) it was granted in 1983.

Became an important precedent for the question of whether algorithms can be patented.

It was ruled unenforceable in 1995 and has since expired.
Graph formulation of the natural order problem

Construct a directed graph from the spreadsheet

- A vertex for every cell
- Edge $x \rightarrow y$ when value from $x$ is used in formula for $y$
  
  edge direction = direction of information flow

Natural order of cells = list of all vertices so that, for every edge $x \rightarrow y$, $x$ is earlier than $y$ in the list

Example:

- Not valid: a, b, c, d, e, f, g
  Edges $g \rightarrow c$ and $g \rightarrow f$ go wrong way, later to earlier
- Valid: a, b, g, c, d, e, f
Definitions
Topological order and topological sorting

Given a directed graph $G$

Topological ordering of $G$: List of all vertices in $G$ so that, for every edge $x \rightarrow y$, $x$ is earlier than $y$ in the list

Topological sorting: the problem of computing a topological ordering of a given graph

Key questions: Which graphs have topological orders? How can we find them quickly?
Why it’s called sorting

if you turn a list of numbers into a graph with a vertex for each number and an edge $x \rightarrow y$ when $x < y$, then topologically sorting the graph is the same as sorting the numbers.

3, 1, 4, 5, 9, 2, 6, 7
Some graphs cannot be ordered

Tour = walk that starts and ends at the same vertex

Cycle = tour with no other repetitions

A graph that contains a cycle has no topological ordering

The vertex from the cycle that is first in the ordering will have an incoming edge from a later vertex

Example: for ordering a, b, c, d, e, f, g, edge c→a goes the wrong way
Theorem: For every directed graph $G$, either
- $G$ has a topological order, or
- $G$ contains a cycle
(but not both!)

How to prove it: find an algorithm that always finds a cycle or a topological order

In spreadsheet application, raise an error message when a new formula would create a cycle

The directed graphs that do not have cycles (and that do have topological orders) are called directed acyclic graphs
Condensation

When a graph has cycles, collapsing its strongly connected components to supervertices produces a directed acyclic graph.

Can be used to extend DAG algorithms to other graphs.
Algorithms
Certifying algorithms

Easier to check correctness of a topological order than to find one:

▶ Label vertices by their position in the order
▶ Check that all vertices are listed exactly once
▶ Check that each edge $x \rightarrow y$ has $\text{label}(x) < \text{label}(y)$

It's also easy to test that a cycle really is a cycle in the graph

If an algorithm produced only a Boolean answer (there is a cycle) rather than outputting one, it would be harder to check

Some software systems use this as a design principle: whenever possible, produce an easy-to-test answer, and then test it

▶ If a bug leads to an incorrect answer, discover it quickly
▶ Because checkers are simpler than the algorithms they check, you can be more confident that their implementation is correct
DFS-based topological ordering algorithm

List vertices in reverse of postorder traversal of DFS forest

```python
def toporder(G):
    def visit(v):
        add v to reached
        for each edge from v to w:
            if w not in reached:
                visit(w)
        add v to end of postorder
    reached = empty set
    postorder = empty list
    for each vertex v in G:
        if v not in reached:
            visit(v)
    return reverse(postorder)
```

Postorder of a tree:
list children recursively, then root

```
|    a    |
|   /   |
|  b   c |
| /  /  |
| d e f h|
```

Postorder:
d, e, b, f, g, h, c, a

Reverse:
a, c, h, g, f, b, e, d
visit(a)
  visit(b)
    visit(d)
      visit(e)
        visit(f)
          add f to postorder
          add e to postorder
          add d to postorder
          add b to postorder
          visit(c)
            add c to postorder
            add a to postorder
            visit(g)
              add g to postorder
Postorder: f, e, d, b, c, a, g
Reverse: g, a, c, b, d, e, f
Why does this work?

Back to classification of edges in DFS, from strong connectivity:

DFS tree edges, forward edges, and cross edges are all ordered correctly (earlier to later) by reverse postorder.

Back edges are not ordered correctly.

Back edge + path in tree from ancestor to descendant = cycle.
Modifications to DFS to detect and find cycles

- Maintain an “active set” of the ancestors of the current vertex (the vertices still on the call stack); add a vertex to the set when its visit starts, and remove before returning
- Store pointers from each vertex to its parent in the DFS forest
- When we find an edge to a vertex in the active set, it is a back edge and forms a cycle
- Use parent pointers to find the cycle and raise an exception
DFS-based topological ordering + cycle-finding

```python
def visit(v):
    add v to active
    add v to reached
    for each edge v to w:
        if w not in reached:
            parent[w] = v
            visit(w)
        else if w in active:
            cycle = [v]
            while v != w:
                v = parent[v]
                cycle.append(v)
            raise cycle
    add v to end of postorder
    remove v from active

active = empty set
reached = empty set
postorder = empty list
parent = empty dictionary
for each vertex v in G:
    if v not in reached:
        visit(v)
    return reverse(postorder)
```
Morals of the story

Every directed graph has either a topological ordering, or a cycle
(usually, many orderings or many cycles)
We can find an ordering or a cycle in linear time using DFS
There may be many orderings or cycles; we only find one

If it has a topological ordering, it’s called a directed acyclic graph

Topological orderings are good for recomputing spreadsheet values
We will see other applications

Certifying algorithms: Useful design style