CS 163 & CS 265: Graph Algorithms

Week 2: Spanning trees and DAGs

Lecture 2c: Topological ordering and critical paths

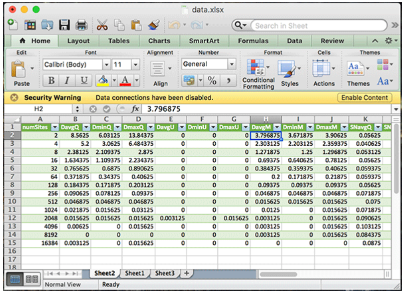
David Eppstein
University of California, Irvine

Winter Quarter, 2025

Spreadsheets

Spreadsheets

Examples: Excel, Visicalc, Google sheets



Security Warning Data connections have been disabled.

H2 3.796875

	A	B	C	D	E	F	G	H	I	J	K
1	numSites	DavgQ	DminQ	DmaxQ	DavgU	DminU	DmaxU	DavgM	DminM	DmaxM	SNavgQ
2	2	8.5625	6.03125	13.84375	0	0	0	3.796875	3.671875	3.90625	0.05625
3	4	5.2	3.0625	6.484375	0	0	0	2.303125	2.203125	2.359375	0.040625
4	8	2.38125	2.109375	2.875	0	0	0	1.271875	1.25	1.296875	0.053125
5	16	1.634375	1.109375	2.234375	0	0	0	0.69375	0.640625	0.78125	0.05625
6	32	0.765625	0.6875	0.890625	0	0	0	0.384375	0.359375	0.40625	0.059375
7	64	0.371875	0.34375	0.40625	0	0	0	0.2	0.171875	0.21875	0.059375
8	128	0.184375	0.171875	0.203125	0	0	0	0.09375	0.09375	0.09375	0.05625
9	256	0.090625	0.078125	0.09375	0	0	0	0.046875	0.046875	0.046875	0.071875
10	512	0.046875	0.046875	0.046875	0	0	0	0.015625	0.015625	0.015625	0.075
11	1024	0.021875	0.015625	0.03125	0	0	0	0.0125	0	0.015625	0.071875
12	2048	0.015625	0.015625	0.015625	0.003125	0	0.015625	0.003125	0	0.015625	0.090625
13	4096	0.00625	0	0.015625	0	0	0	0.003125	0	0.015625	0.103125
14	8192	0	0	0	0	0	0	0.003125	0	0.015625	0.084375
15	16384	0.003125	0	0.015625	0	0	0	0	0	0	0.0875
16											
17											
18											

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

Graph formulation of the natural order problem

Construct a directed graph from the spreadsheet

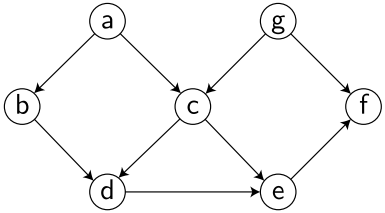
- A vertex for every cell
- Edge $x \to y$ when value from x is used in formula for y

  edge direction = direction of information flow

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

Example:

- Not valid: a, b, c, d, e, f, g
  Edges g→c and g→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 $x \to y$, x is earlier than y in the list

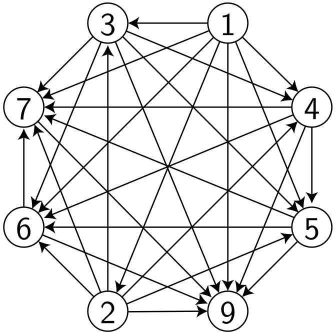
Topological sorting: the problem of computing a topological ordering of a given graph

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 $x \to y$ when $x < y$, then topologically sorting the graph is the same as sorting the numbers

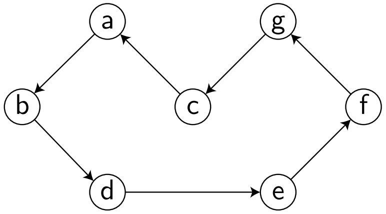
3, 1, 4, 5, 9, 2, 6, 7



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$\rightarrow$a goes the wrong way

Directed acyclic graphs

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

Algorithms

Greedy topological ordering algorithm

Data structures:

- Decorate each vertex v with $\texttt{count}[v]$: number of unprocessed vertices u with an edge $u \to v$
- Maintain a collection `ready`: unprocessed vertices with $\texttt{count}[v] = 0$

Initialize:

$\texttt{count}[v] = \#$ incoming edges

$\texttt{ready}$ = vertices without incoming edges

While `ready` is not empty:

- Let v be any vertex in `ready`
- Remove v from `ready`
- Output v
- For each edge $v \to w$:
  - Subtract one from $\texttt{count}[w]$
  - If $\texttt{count}[w] = 0$, add w to `ready`

If loop exits without processing all vertices, graph is not acyclic!

Example

$\texttt{counts} = \{a:0, b:1, c:2, d:2, e:2, f:2, g:0\}$
$\texttt{ready}= \{a, g\} \Rightarrow$ choose a

$\texttt{counts} = \{a:0, b:0, c:1, d:2, e:2, f:2, g:0\}$
$\texttt{ready}= \{g, b\} \Rightarrow$ choose g

$\texttt{counts} = \{a:0, b:0, c:0, d:2, e:2, f:1, g:0\}$
$\texttt{ready}= \{b, c\} \Rightarrow$ choose b

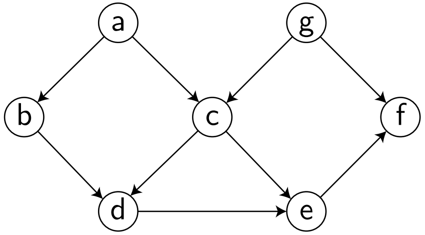
$\texttt{counts} = \{a:0, b:0, c:0, d:1, e:2, f:1, g:0\}$
$\texttt{ready}= \{c\} \Rightarrow$ choose c

$\texttt{counts} = \{a:0, b:0, c:0, d:0, e:1, f:1, g:0\}$
$\texttt{ready}= \{d\} \Rightarrow$ choose d

$\texttt{counts} = \{a:0, b:0, c:0, d:0, e:0, f:1, g:0\}$
$\texttt{ready}= \{e\} \Rightarrow$ choose e

$\texttt{counts} = \{a:0, b:0, c:0, d:0, e:0, f:0, g:0\}$
$\texttt{ready}= \{f\} \Rightarrow$ choose f

$\texttt{counts} = \{a:0, b:0, c:0, d:0, e:0, f:0, g:0\}$
$\texttt{ready} =$ empty $\Rightarrow$ exit



Output order: a, g, b, c, d, e, f

All vertices were processed, so this order is valid

What happens when the algorithm gets stuck?

Suppose we run out of ready-to-process vertices before outputting everything

Then all remaining vertices have $\texttt{count}[v] > 0$
$\Rightarrow$ each remaining vertex has an incoming neighbor that is also a remaining vertex

Algorithm to find a cycle:

- Start at any remaining vertex
- Walk backwards to another remaining vertex, continuing until you see the same vertex twice
- The sequence between the two repetitions of the same vertex is a cycle (in reverse order)

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 $x \to y$ 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

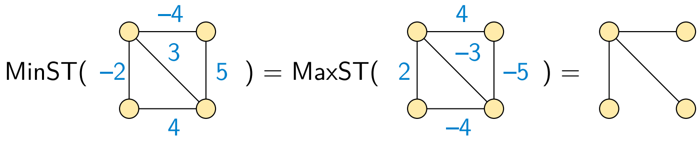
- If a bug leads to an incorrect answer, discover it quickly
- Because checkers are simpler than the algorithms they check, you can be more confident that their implementation is correct

Longest vs shortest paths

When minimization and maximization are the same

In the minimum spanning tree problem,

- Minimum tree for weights $w(e)$
  $=$ maximum tree for weights $-w(e)$
- Algorithms don't care if numbers are positive or negative
- Can use same algorithms (with minor changes) for both minimum and maximum

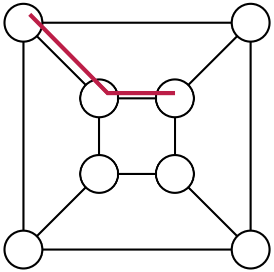


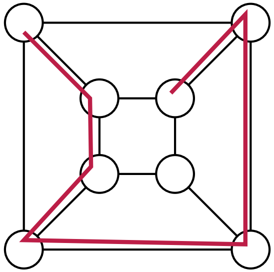
Same is true for shortest/ longest paths in directed acyclic graphs
(today's topic; our application will use longest paths)

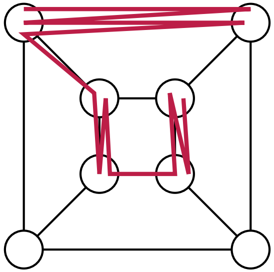
When minimization and maximization differ

For unweighted undirected graphs
(meaning that the length of a path is the number of edges)

- Shortest paths can be found by breadth first search
- Longest path is NP-hard (unlikely to have an efficient algorithm): includes Hamiltonian path through all vertices
- Longest walk doesn't generally exist (can go back and forth infinitely many times on same edge)







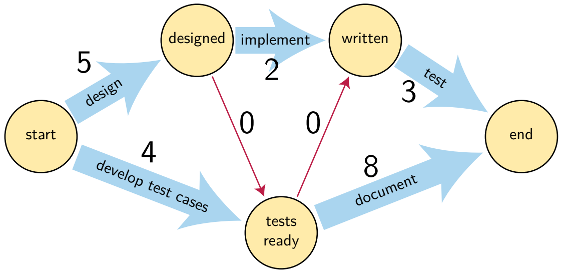
Critical path scheduling

Critical path scheduling (“PERT method”)

- Input describes a project consisting of multiple tasks, each taking different amounts of time
- Some tasks depend on each other: certain tasks must be finished before others can start
- When two tasks do not depend on each other, they can both be active at the same time without slowing each other down (you have enough people working on the project to assign independent teams to each task)
- Goal: Find a schedule for all of the tasks that finishes the whole project in the minimum possible time

Activity-on-edge graph

Directed acyclic graph



Vertices = project milestones

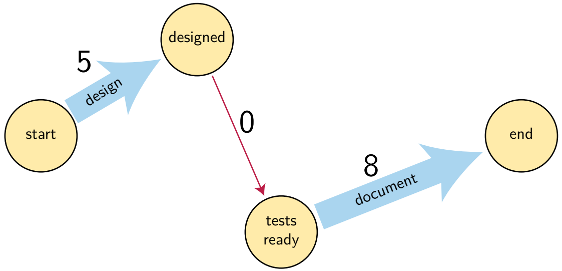
- Single start vertex has no incoming edges
- Single end vertex has no outgoing edges

Edges = tasks or ordering constraints

- Tasks (thick blue arrows) labeled by amount of time
- Constraints (thin red arrows) take zero time

Critical path

Longest path from start to end
(length = sum of edge labels)

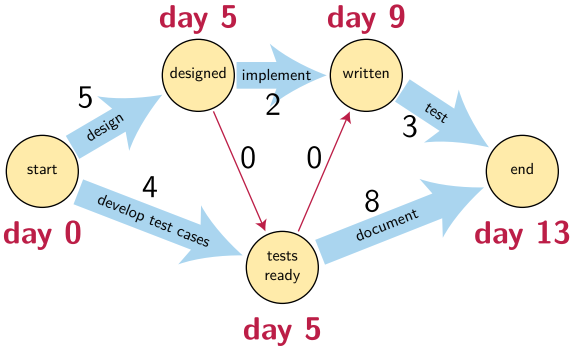


Determines how long whole project must take

What is a valid schedule?

Assignment of times to milestones

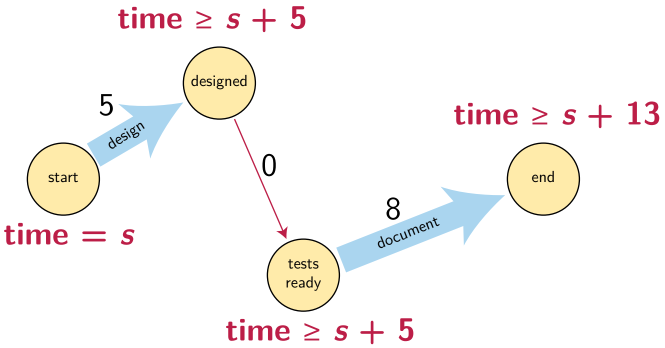
Tasks have enough time and ordering constraints are met:
For every edge, time at destination $-$ time at source $\geq$ edge length



Total length $=$ time at end vertex $-$ time at start vertex

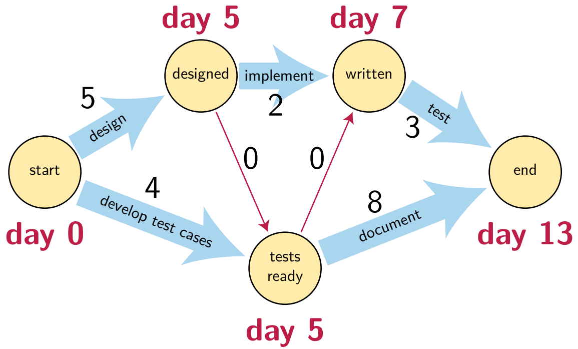
Valid schedule length $\geq$ critical path length

By induction on number of steps from start, each milestone must have a scheduled time $\geq$ length of the path from start



Optimal schedule length $\leq$ critical path length

Schedule each milestone at
time = length of longest path from start
(start milestone = 0, end milestone = critical path length)



Every edge has enough time, because otherwise it would be part of a longer path to its destination milestone

How to find the optimal schedule

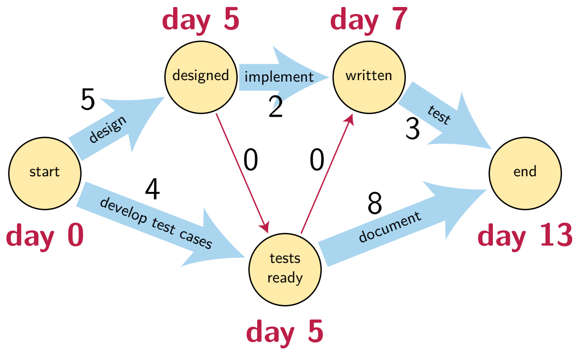
```
$L$ = empty dictionary

for each vertex $v$ in topological order:
    if $v$ is the starting vertex:
        $L[v] = 0$
    else:
        $L[v] = \max_{\text{edge } u \to v} L[u] + \text{length}(u \to v)$
```

This computes longest path distances from start to all other vertices in linear time

Topological ordering implies that all predecessor vertices u already have the correct distance by the time we need them in the calculation for the distance for v

Change max to min in last line to get shortest path distances

Example of finding optimal schedule



Start: is starting vertex, $L[v] = 0$

Designed: edge from $u =$ start, $L[v] = L[u] +$ length $= 5$

Tests ready: two in-edges, $L[u] +$ length $= 0 + 4, 5 + 0$, max $= 5$

Written: two in-edges, $L[u] +$ length $= 5 + 2, 5 + 0$, max $= 7$

End: two in-edges, $L[u] +$ length $= 7 + 3, 5 + 8$, max $= 13$

How to find the critical path

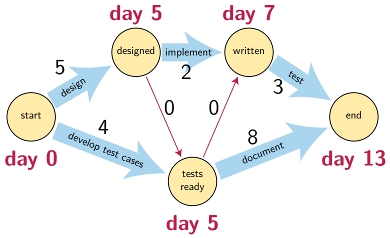
(or one of the critical paths, if more than one equally long path)

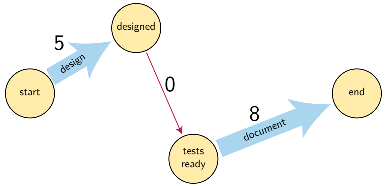
```
pathlist = single-element list containing the end vertex
while the last vertex $v$ in pathlist is not the start vertex:
    find an edge $u \to v$ such that $L[v] = L[u] + \text{length}(u \to v)$
    add $u$ to end of pathlist
return reverse(pathlist)
```

This is just the backtracking method for turning a dynamic programming algorithm that computes the value of an optimal solution into an algorithm for the solution

There are multiple critical paths if we ever have more than one choice for u

Example of finding critical path





pathlist = [end]

edge to "end" with $L[u] + \text{length} = L[v]$ is from "tests ready"

pathlist = [end, tests ready]

edge to "tests ready" with $L[u] + \text{length} = L[v]$ is from "designed"

pathlist = [end, tests ready, designed]

edge to "designed" with $L[u] + \text{length} = L[v]$ is from "start"

pathlist = [end, tests ready, designed, start]

Reverse(list) is path: start $\longrightarrow$ designed $\longrightarrow$ tests ready $\longrightarrow$ end

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

There may be many orderings or cycles; we only find one

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

Applications of topological orderings include recomputing spreadsheet values and finding critical paths

Certifying algorithms: Useful design style