Last time: cheapest spanning trees

Two ways: both greedy algorithms

- Kruskal: add the cheapest edge that doesn't make a loop
- Prim: start at v, add cheapest edge to a new vertex

The tricky part:

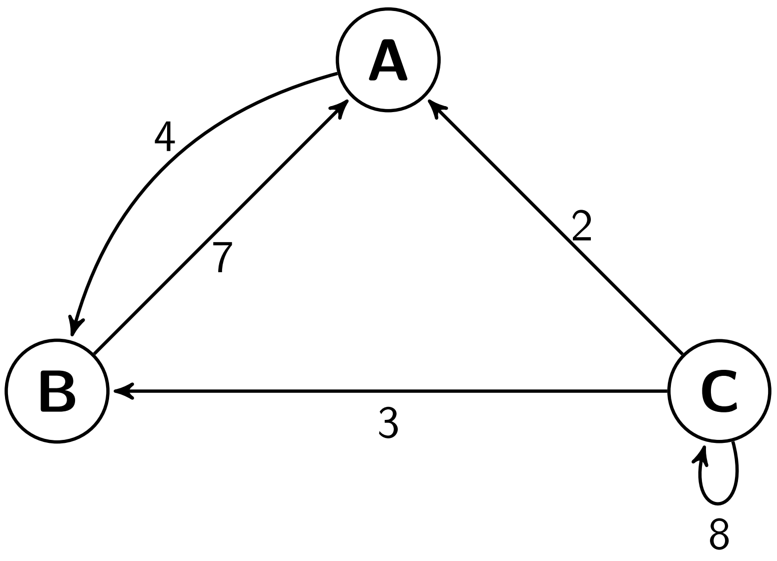
The algorithms are easy to run, but it's not immediately clear that they actually find the cheapest spanning tree. The proof that they do will not appear on the exam.

Today: shortest and longest paths; Travelling Salesperson

Directed graphs

Definition

A *directed graph* is one where each edge has a chosen starting and ending point, usually indicated with arrows.



Walks in directed graphs:

You can only travel the edge in the direction the arrow shows.

Dijkstra's algorithm for shortest path

Input:

A weighted (possibly directed) graph G and starting vertex $v \in G$

Output:

For every vertex $w \in G$ a list of all shortest paths from v to w

Initialize:

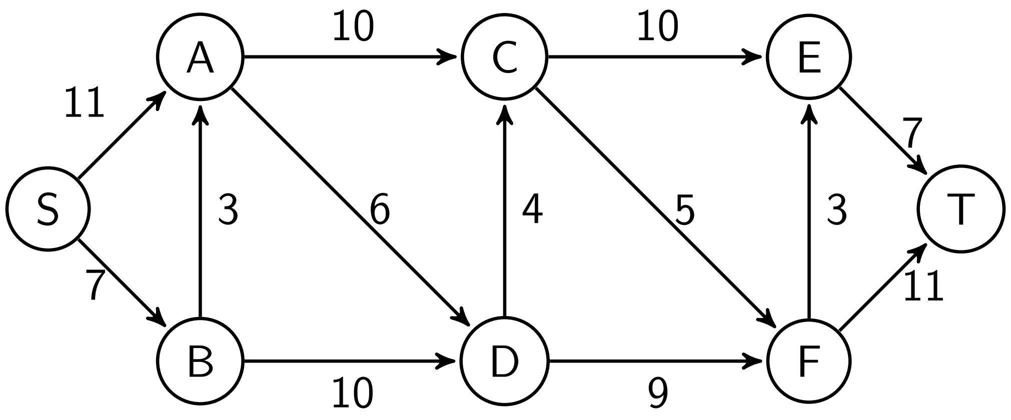
From starting vertex v list every edge out of v as a poential shortest path to corresponding vertex w

Iterate:

- Choose w with cheapest potential shortest path and make these paths permanent
- Update list of potential paths by adding edges out of w to the shortest paths to w and checking if they're cheaper than known paths

Example graph from the 2008 Exam

Find all shortest paths from S to T.



Add on:

- Which edges if made a little longer would make the distance from S to T longer?
- Which edges if made a little shorter would make the distance from S to T shorter?

The main idea of why Dijkstra's works:

Suppose we're at the step were Dijkstra's decides the cheapest path to w.

Then:

A cheaper path to w would have to go through a vertex u we haven't found the cheapest path to yet.

But!

Even *getting* to u costs more than our cheapest path to w.

An observation:

Dijkstra's algorithm depends on the edge weights being non-negative!

Culture: performance of Dijkstra's algorithm

In a limited sense, Dijkstra's algorithm is optimal:

If *all we know* is that we have a weighted graph, then you can't do better than Dijkstra's algorithm.

In practise, often not very good:

When finding path from Sheffield to Edinburgh, Dijkstra's algorithm explores every street in London.

Real world maps have extra information:

It's easy to calculate the distance between two points as the crow flies, and we know the driving distance has to be at least that large.

The A* algorithm avoids searching London:

Supposes we have an easy to calculate "heuristic distance" $h(v, w)$, that is a lower bound for the actual distance $d(v, w)$.

Finding the longest path

Scheduling a large problem with many parts

For example, building a house.

- Some can be done at same time: (finishing interior rooms)
- Some need to be done in order: (foundation before walls)
- How early could whole project be finished?

Solution: longest path

- Encode tasks as edges in directed weighted graph
- Edge e follows edge f if task f requires task e
- Length of longest path is the shortest time to complete project

Building the directed graph from a list of tasks and dependencies can require a few tricks and won't be tested.

Longest path might not exist:

A necessary assumption:

If the graph has a directed cycle, we could get an infinitely long path by repeating graph over and over again.

Definition

A directed graph G is *acyclic* if it has no directed cycles.

Graphs in scheduling applications are acyclic:

Otherwise we'd have a cycle of tasks that all depend on each other and we could never start the project!

Ordering vertices / "topological sort"

If G is acyclic it's easy to order the vertices of G so that if there's an edge from v to w, then w comes after v.

The longest path algorithm:

Initializing:

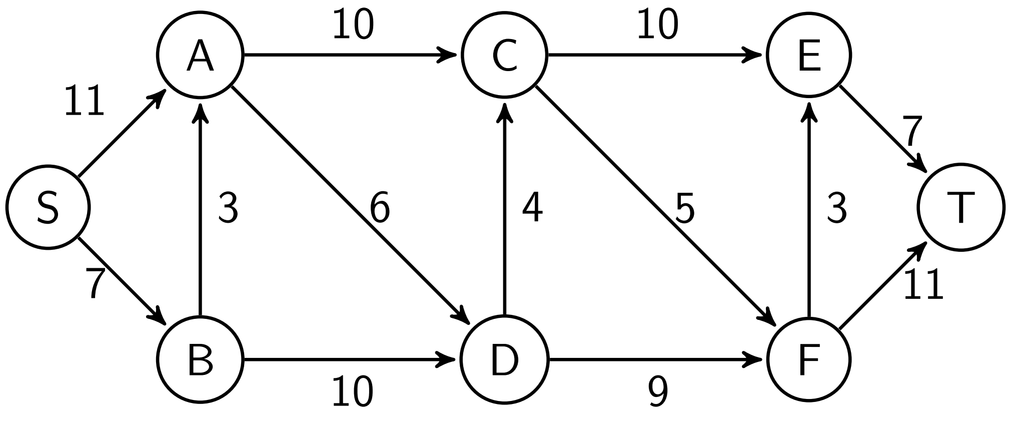
- Topologically sort the vertices of G
- List every edge of starting vertex as potential longest path

Iterate:

- Make the potential longest path to the first vertex w on the list permament
- Update the list of potential longest paths adding edges out of w to longest paths to w and seeing if they create new longest paths

Example graph from the 2008 Exam

Find all longest paths from S to T.


$$\begin{tikzcd}
 & A \arrow[r, "10"] \arrow[dr, "6"] & C \arrow[r, "10"] \arrow[dr, "5"] & E \arrow[dr, "7"] & \\
S \arrow[ur, "11"] \arrow[dr, "7"'] & & & & T \\
 & B \arrow[uu, "3"] \arrow[r, "10"'] & D \arrow[uu, "4"] \arrow[r, "9"'] & F \arrow[uu, "3"] \arrow[ur, "11"'] &
\end{tikzcd}$$

Add on:

- Which edges if made a little longer would make the longest path from S to T longer?
- Which edges if made a little shorter would make the longest path from S to T shorter?