Plan

Previous lecture:

- Depth-first and breadth-first search
- Using DFS to detect cycles in directed graphs
- Complexity of breadth-first search
- Complexity of depth-first search

This lecture:

- Topological sort (also cycle detection)
- Dijkstra’s algorithm (if I have time)

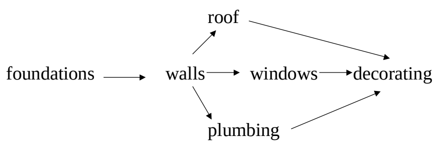
Topological Sort

- Given a directed acyclic graph, produce a linear sequence of vertices such that for any two vertices u and v, if there is an edge from u to v than u is before v in the sequence.

Topological Sort

- Input to the algorithm: directed acyclic graph
- Output: a linear sequence of vertices such that for any two vertices u and v, if there is an edge from u to v than u is before v in the sequence.
- Useful to think of this as: edges correspond to dependencies (pre-requisites), and a vertex could not precede its pre-requisites in the sequence.

Example: building a house



Possible sequence:

Foundations-Walls-Roof-Windows-Plumbing-Decorating

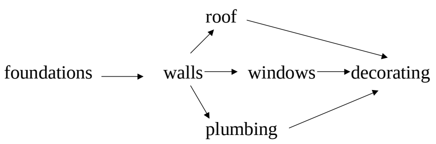
Applications

- Planning and scheduling.
- The algorithm can also be modified to detect cycles.

Topological Sort algorithm

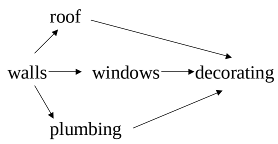
- Create an array of length equal to the number of vertices.
- While the number of vertices is greater than 0, repeat:
  - Find a vertex with no incoming edges (“no pre-requisites”).
  - Put this vertex in the array.
  - Delete the vertex from the graph.
- Note that this destructively updates a graph; often this is a bad idea, so make a copy of the graph first and do topological sort on the copy.

Example:



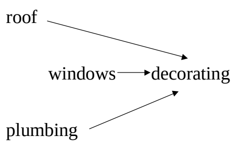
Array for the linear sequence: size 6
(Initially empty)

Example:



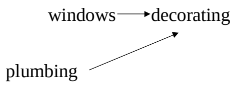
Array for the linear sequence: size 6
Foundations

Example:



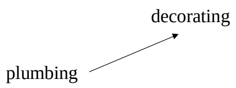
Array for the linear sequence: size 6
Foundations-Walls

Example:



Array for the linear sequence: size 6
Foundations-Walls-Roof

Example:



Array for the linear sequence: size 6
Foundations-Walls-Roof-Windows

Example:

decorating

Array for the linear sequence: size 6
Foundations-Walls-Roof-Windows-Plumbing

Example:

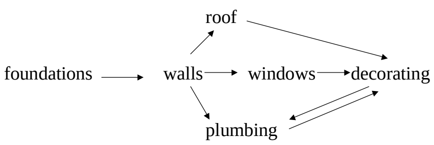
Array for the linear sequence: size 6

Foundations-Walls-Roof-Windows-Plumbing-Decorating

Cycle detection with topological sort

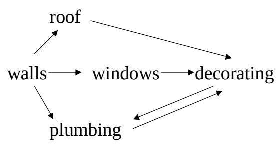
- What happens if we run topological sort on a cyclic graph?
- There will be either no vertex with 0 prerequisites to begin with, or at some point in the iteration.
- If we run a topological sort on a graph and there are vertices left undeleted, the graph contains a cycle.

Example: building a house with a vicious circle

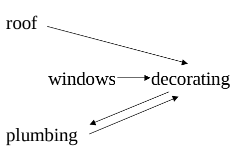


Plumbing depends on decorating and decorating on plumbing

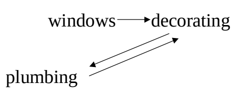
Example: building a house with a vicious circle



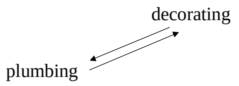
Example: building a house with a vicious circle



Example: building a house with a vicious circle



Example: building a house with a vicious circle



Stuck!

Why does it work?

- Topological sort: a vertex cannot be removed before all its prerequisites have been removed. So it cannot be inserted in the array before its prerequisite.
- Cycle detection: in a cycle, a vertex is its own prerequisite. So it can never be removed.

Greedy graph algorithms

So far:

- Introduction to graphs
- Adjacency lists and adjacency matrices
- Breadth-first search and depth-first search
- Topological sort

Today: greedy algorithms in general and greedy graph algorithms in particular

Shortest path

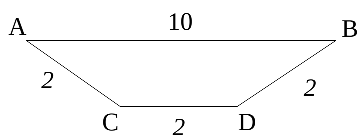
- Find the shortest route between two vertices u and v.
- It turns out that we can just as well compute shortest routes to ALL vertices reachable from u (including v). This is called *single-source shortest path problem* for weighted graphs, and u is the source.

Dijkstra’s Algorithm

- An algorithm for solving the single-source shortest path problem.
- The first version of the Dijkstra's algorithm (traditionally given in textbooks) returns not the actual path, but a number - the shortest distance between u and v.
- (Assume that weights are distances, and the length of the path is the sum of the lengths of edges.)

Example

- Dijkstra’s algorithm should return 6 for the shortest path between A and B:



Dijkstra's algorithm

To find the shortest paths (distances) from s:

- keep a priority queue PQ of vertices to be processed
- keep an array with current known shortest distances from s to every vertex (initially set to be infinity for all but s and 0 for s)
- order the queue so that the vertex with the shortest distance is at the front.

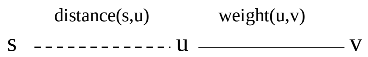
Dijkstra's algorithm

Loop until there are vertices in the queue PQ:

- dequeue a vertex u
- recompute shortest distances for all vertices in the queue as follows: if there is an edge from u to a vertex v in PQ and the current shortest distance to v is greater than distance(s,u) + weight(u,v) then replace distance(s,v) with distance (s,u) + weight(u,v).

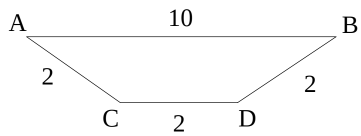
Computing the shortest distance

If the shortest distance from s to u is distance(s,u) and the weight of the edge between u and v is weight(u,v), then the current shortest distance from s to v is distance(s,u) + weight(u,v).



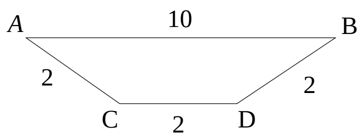
Example

- Distances: (A,0), (B,INF), (C,INF), (D,INF)
- PQ = {A,B,C,D}



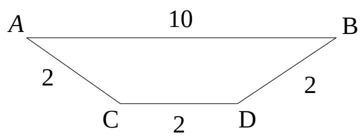
Example (dequeue A)

- Distances: (A,0), (B,INF), (C,INF), (D,INF)
- PQ = {B,C,D}



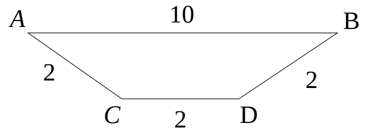
Example (recompute distances)

- Distances: (A,0), (B,10), (C,2), (D,INF)
- PQ= {C,B,D}



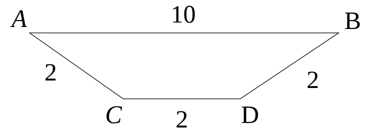
Example (dequeue C)

- Distances: (A,0), (B,10), (C,2), (D,INF)
- PQ = {B,D}



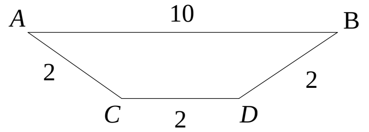
Example (recompute distances)

- Distances: (A,0), (B,10), (C,2), (D,4)
- PQ = {D,B}



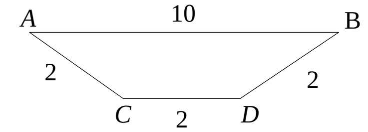
Example (dequeue D)

- Distances: (A,0), (B,10), (C,2), (D,4)
- PQ = {B}



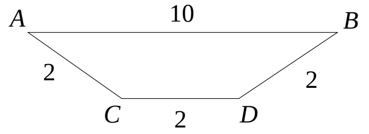
Example (recompute distances)

- Distances: (A,0), (B,6), (C,2), (D,4)
- PQ = {B}



Example (dequeue B)

- Distances: (A,0), (B,6), (C,2), (D,4)
- PQ = {}



Pseudocode for D's Algorithm

- INF is supposed to be greater than any number
- *dist* : array holding shortest distances from source s
- *PQ* : priority queue of unvisited vertices prioritised by shortest recorded distance from source
- *PQ.reorder()* reorders PQ if the values in *dist* change.

Pseudocode for Dijkstra's Algorithm

```
for(each v in V){
  dist[v] = INF;
  dist[s] = 0;
}
PriorityQueue PQ = new PriorityQueue();
// insert all vertices in PQ,
// in reverse order of dist[]
// values
```

Pseudocode for D's Algorithm

```
while (! PQ.isempty()){
  u = PQ.dequeue();
  for(each v in PQ adjacent to u){
    if(dist[v] > (dist[u]+weight(u,v)){
       dist[v] = (dist[u]+weight(u,v));
    }
  }
  PQ.reorder();
}
return dist;
```

Modified algorithm

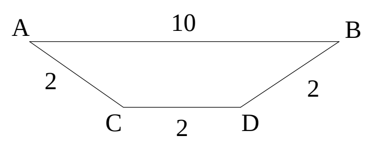
How to make Dijkstra's algorithm to return the path itself, not just the distance:

- In addition to distances, maintain a path (list of vertices) for every vertex
- In the beginning paths are empty
- When assigning dist(s,v)=dist(s,u)+weight(u,v) also assign path(v)=path(u).
- When dequeuing a vertex, add it to its path.

Example

- Distances and paths:

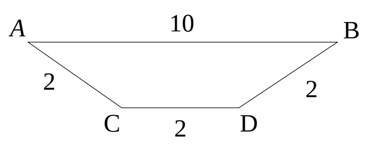
(A,0,{}), (B,INF,{}), (C,INF{}), (D,INF,{})



Dequeue A, recompute paths

- Distances and paths:

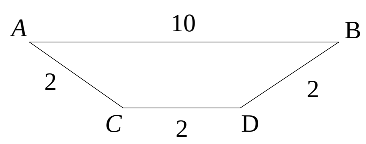
(A,0,{A}), (B,10,{A}), (C,2,{A}), (D,INF,{})



Dequeue C, recompute paths

- Distances and paths:

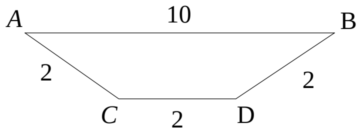
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,INF,{})



Dequeue C, recompute paths

- Distances and paths:

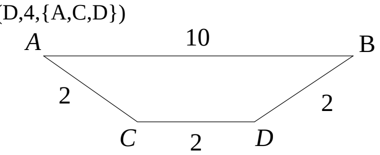
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,4,{A,C})



Dequeue D, recompute paths

- Distances and paths:

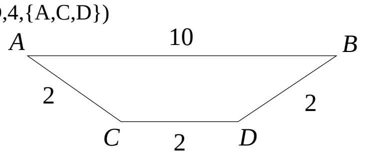
(A,0,{A}), (B,6,{A,C,D}), (C,2,{A,C}), (D,4,{A,C,D})



Dequeue B, recompute paths

- Distances and paths:

(A,0,{A}), (B,6,{A,C,D,B}), (C,2,{A,C}), (D,4,{A,C,D})



Greedy algorithms

- No long-term strategy: maximise profit at the moment (make locally optimal choices).
- If you need to minimise distance, pick the closest vertex at each step.
- If you need to minimise some other property, pick a step with the minimal cost with respect to that property.

Greedy Algorithms

- Dijkstra's algorithm: pick the vertex to which there is the shortest path currently known at the moment.
- For Dijkstra's algorithm, this also turns out to be globally optimal: can show that a shorter path to the vertex can never be discovered.
- There are also greedy strategies which are not globally optimal.

Example: non-optimal greedy algorithm

- Problem: given a number of coins, count the change in as few coins as possible.
- Greedy strategy: start with the largest coin which is available; for the remaining change, again pick the largest coin; and so on.

Example

Coins: 2x50p, 4x20p, 10x1p

Count 80p:

80p = 50 +

Example

Coins: 2x50p, 4x20p, 10x1p

Count 80p:

80p = 50 + 20 +

Example

Coins: 2x50p, 4x20p, 10x1p

Count 80p:

80p = 50 + 20 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

Could have counted 20 + 20 + 20 +20

Non-optimal algorithm (does not give the optimal solution.)

Optimality of Dijkstra's algorithm

So, why is Dijkstra’s algorithm optimal (gives the shortest path)?

Let us first see where it *could* go wrong.

What the algorithm does

- For every vertex in the priority queue, we keep updating the current distance downwards, until we remove the vertex from the queue.
- After that the shortest distance for the vertex is set.
- What if a shorter path can be discovered later?

Optimality proof

- Base case: the shortest distance to the start node is set correctly (0)
- Inductive step: assume that the shortest distances are set correctly for the first n vertices removed from the queue. Show that it will also be set correctly for the n+1st vertex.

Optimality proof

- Assume that the n+1st vertex is u. It is at the front of the priority queue and it's current known shortest distance is dist(s,u). We need to show that there is no path in the graph from s to u with the length smaller than dist(s,u).

Optimality proof

- Proof by contradiction: assume there is such a (shorter) path:

$$s \text{ - - - - - - } v1 \longrightarrow v2 \text{ - - - - - - } u$$

Optimality proof

- Here the vertices from s to v1 have correct shortest distances (inductive hypothesis) and v2 is still in the priority queue.

$$s \text{ - - - - - - } v1 \longrightarrow v2 \text{ - - - - - - } u$$

Optimality proof

- So dist(s,v1) is indeed the shortest path from s to v1. Current distance to v2 is dist(s,v2)=dist(s,v1)+weight(v1,v2)

$$s \text{ - - - - - - } v1 \longrightarrow v2 \text{ - - - - - - } u$$

Optimality proof

- If v2 is still in the priority queue, then dist(s,v1)+weight(v1,v2) >= dist(s,u)

$$s \text{ - - - - - - } v1 \longrightarrow v2 \text{ - - - - - - } u$$

Optimality proof

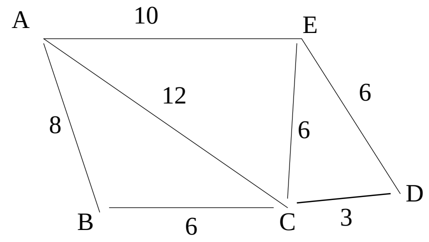
- But then the path going through v1 and v2 cannot be shorter than dist(s,u). QED

$$s \text{ - - - - - - } v1 \longrightarrow v2 \text{ - - - - - - } u$$

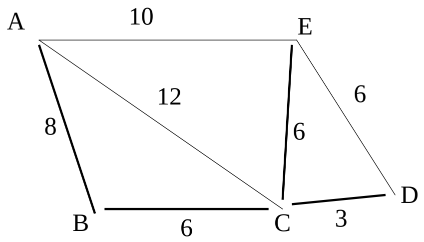
Minimal spanning tree

- Input: connected, undirected, weighted graph
- Output: a tree which connects all vertices in the graph using only the edges present in the graph and is minimal in the sense that the sum of weights of the edges is the smallest possible

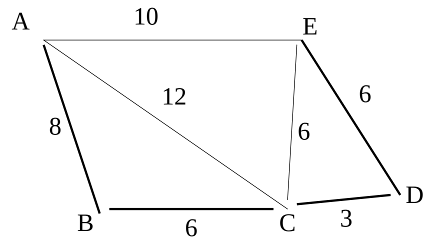
Example: graph



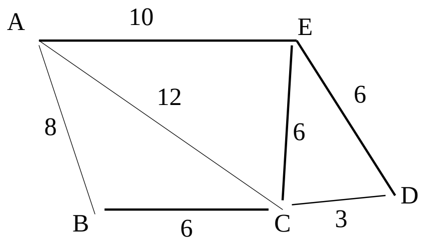
Example: MST (cost 23)



Example: another MST (cost 23)



Example: not MST (cost 28)



Why MST is a tree

- We just need to keep the resulting graph connected.
- For every vertex need only one in-coming edge (if there are two, one can be removed and the graph is still connected).
- A graph where every vertex has only one in-coming edge is a tree.

Prim's algorithm

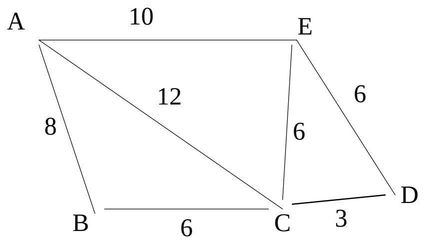
To construct an MST:

- Pick any vertex M
- Choose the shortest edge from M to any other vertex N
- Add edge (M,N) to the MST
- Continue to add at every step the shortest edge from a vertex in MST to a vertex outside, until all vertices are in MST

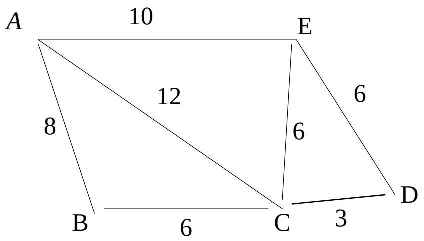
Greedy algorithm

- Note that Prim's algorithm is also greedy: just adds a shortest edge without worrying about the overall structure
- It is also optimal: see Shaffer Theorem 7.1 (p.217)

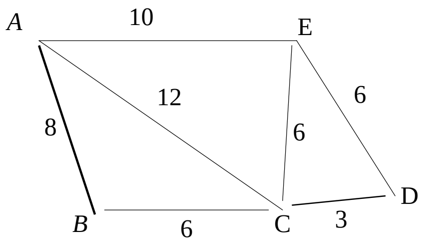
Example



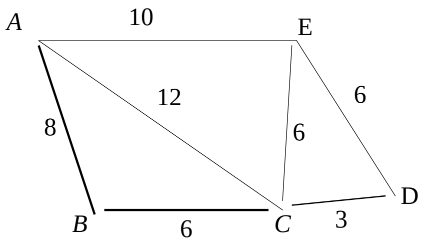
Example



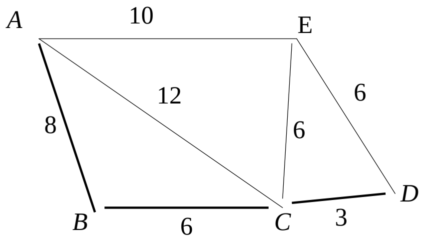
Example



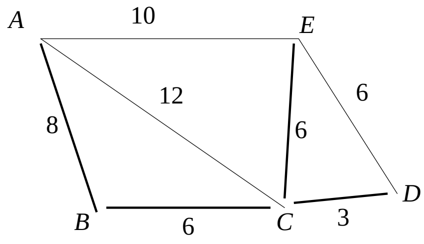
Example



Example



Example



Further reading

- More on graph algorithms: Shaffer, Chapter 7, or any other ADS textbook (Floyd’s and Kruskal’s algorithms).