Graph Algorithms

Edited by Heshan Du
University of Nottingham Ningbo China

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- Chapter 14. Graphs
- Sections 14.5-14.7
- pp. 609-638

Learning Objectives

- To be able to *understand* the topological sort algorithm, the minimal spanning tree algorithm and Dijkstra’s shortest path algorithm;
- To be able to *analyze* the time complexity of Dijkstra’s shortest path algorithm;
- To be able to *implement* these three graph algorithms;
- To be able to *apply* these graph algorithms to solve problems.

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, then u is before v in the sequence.

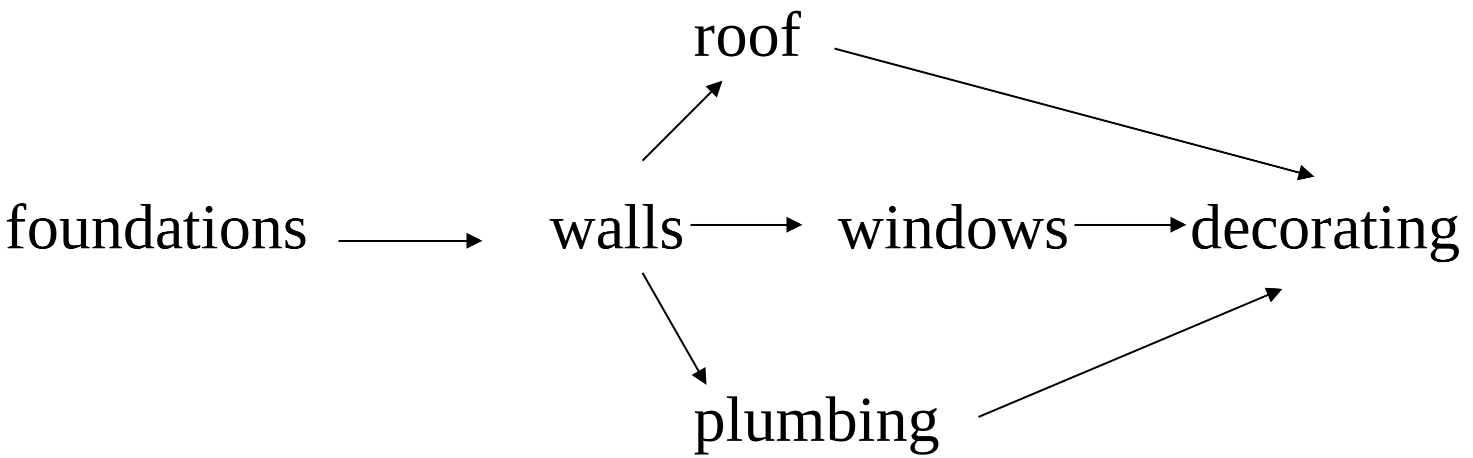
Topological Sort

- *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, then 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:

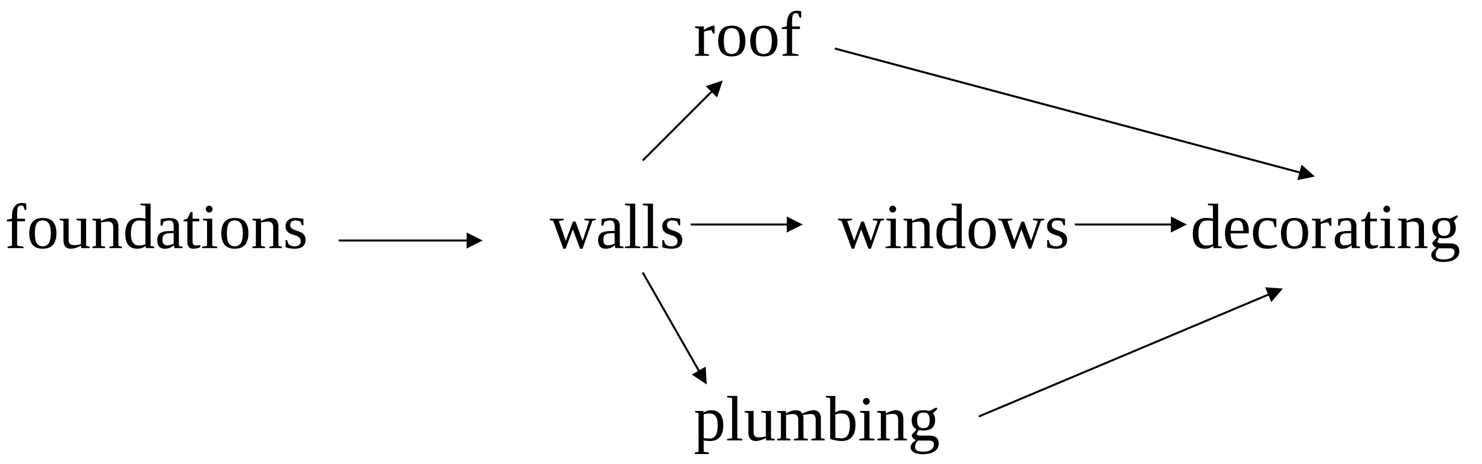
Foundations-Walls-Roof-Windows-Plumbing-Decorating

Applications

- Planning and scheduling

- The algorithm can also be modified to detect cycles.

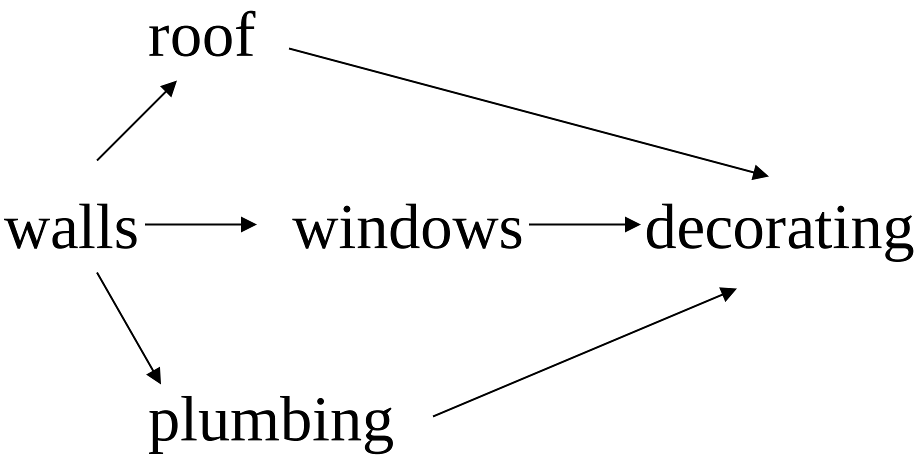
Example:



Array for the linear sequence: size 6

(Initially empty)

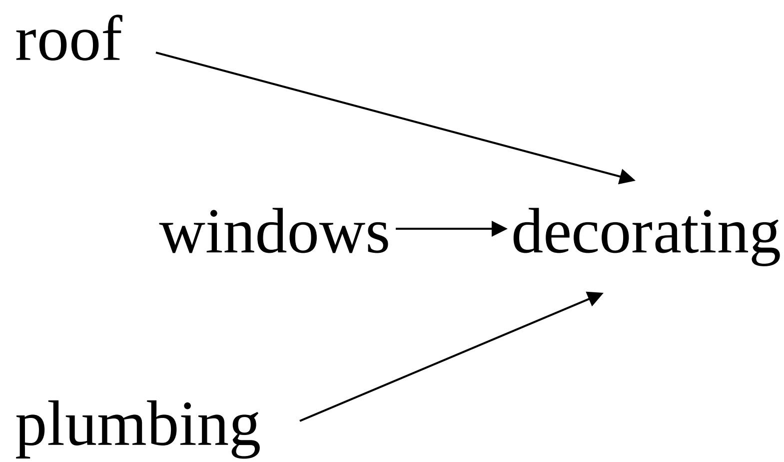
Example:



Array for the linear sequence: size 6

Foundations

Example:



Array for the linear sequence: size 6

Foundations-Walls

Example:

$$
\begin{tikzcd}
 & \text{windows} \arrow[r] & \text{decorating} \\
\text{plumbing} \arrow[urr] & &
\end{tikzcd}
$$

Array for the linear sequence: size 6

Foundations-Walls-Roof

Example:

decorating

plumbing

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

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.

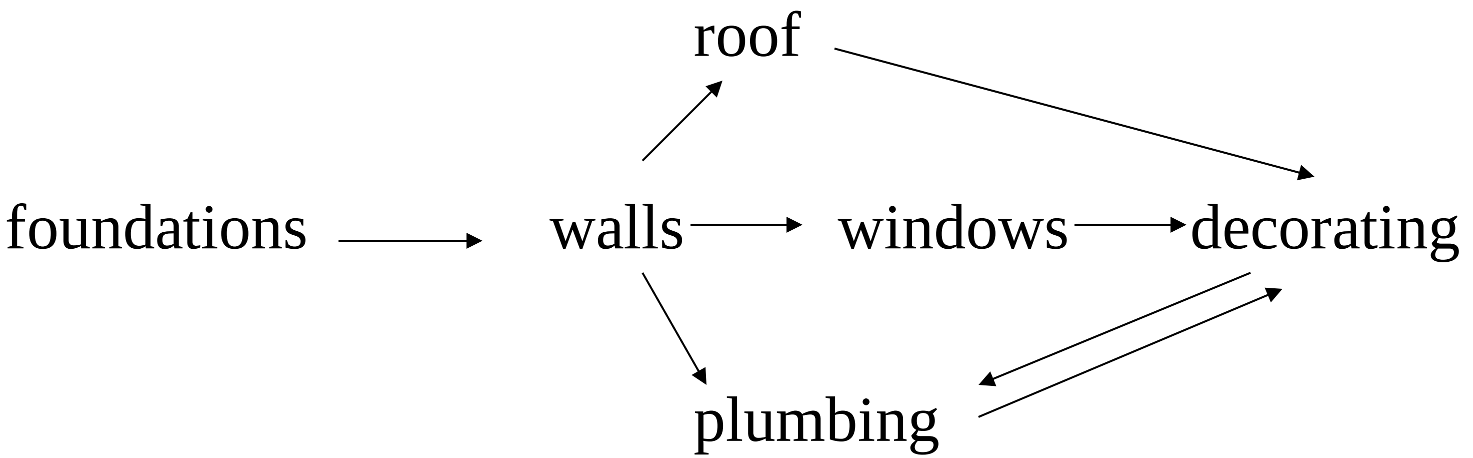
Cycle detection with topological sort

- What happens if we run topological sort on a cyclic graph?

Cycle detection with topological sort

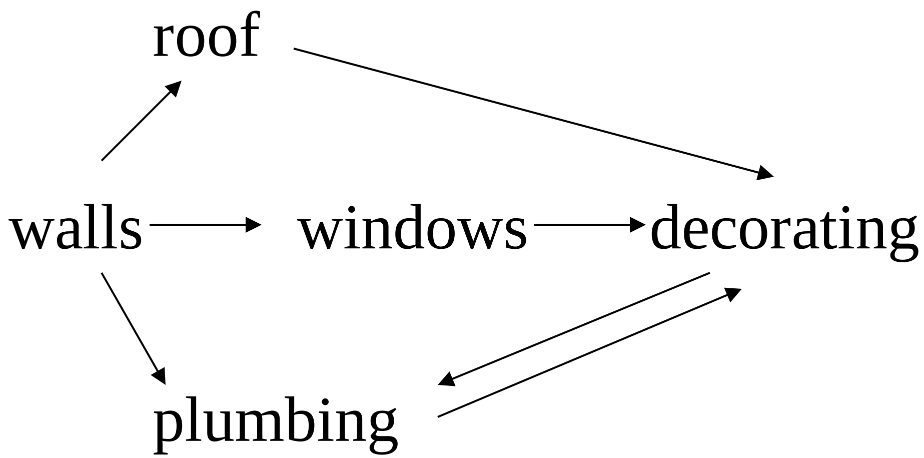
- What happens if we run topological sort on a cyclic graph?
- There will be either no vertex with 0 prerequisites to begin with, or at some point in the iteration.
- If we run a topological sort on a graph and there are vertices left undeleted, the graph contains a cycle.

Example: building a house with a vicious circle

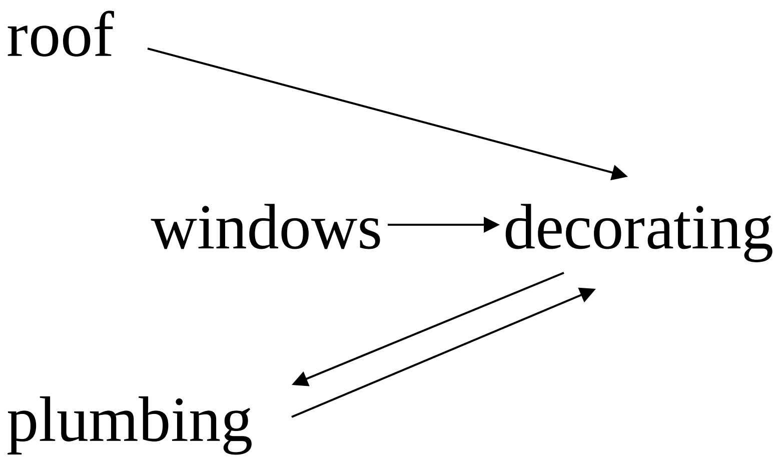


Plumbing depends on decorating and decorating on plumbing

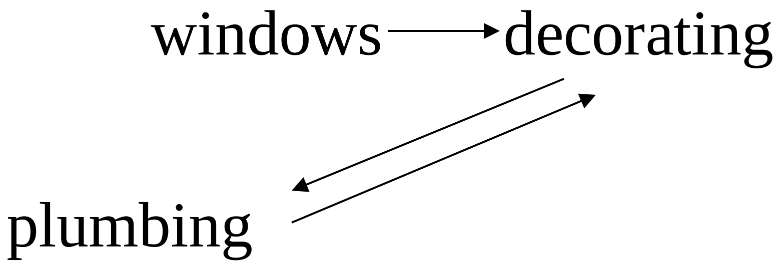
Example: building a house with a vicious circle



Example: building a house with a vicious circle



Example: building a house with a vicious circle



Example: building a house with a vicious circle

plumbing $\rightleftarrows$ decorating

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.

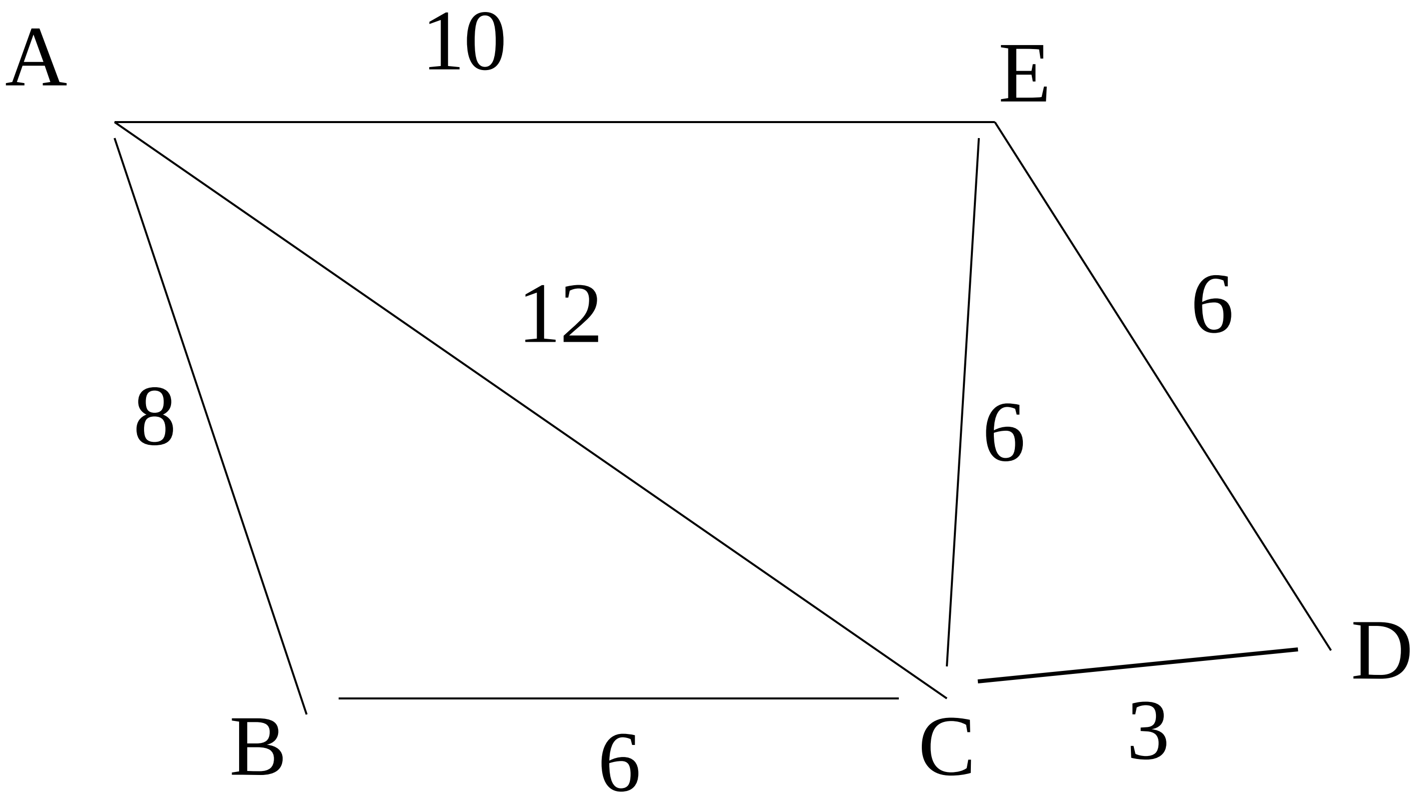
Spanning Tree

- *Input*: connected, undirected graph
- *Output*: a tree which connects all vertices in the graph using only the edges present in the graph

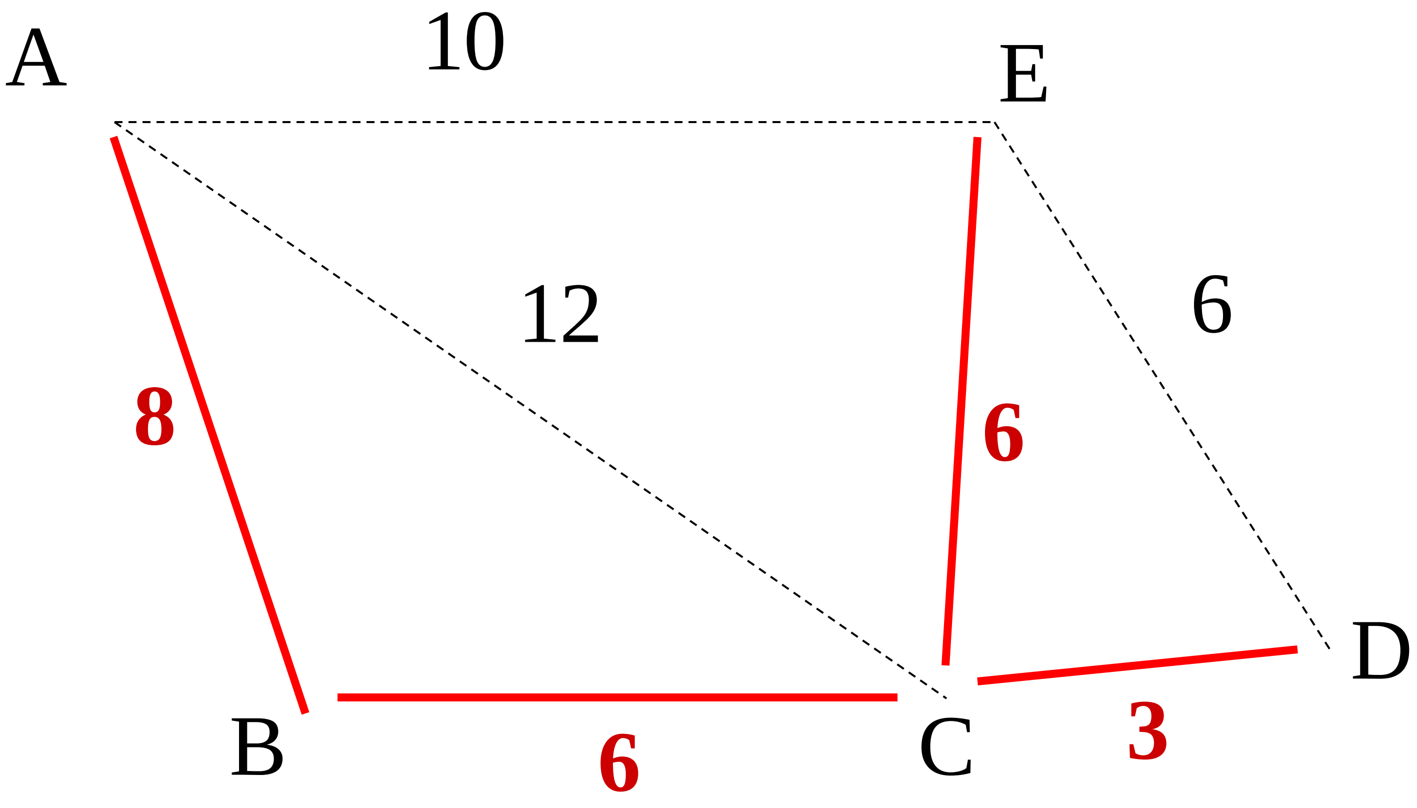
Minimal Spanning Tree

- *Input*: connected, undirected, weighted graph
- *Output*: a spanning tree
  - (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

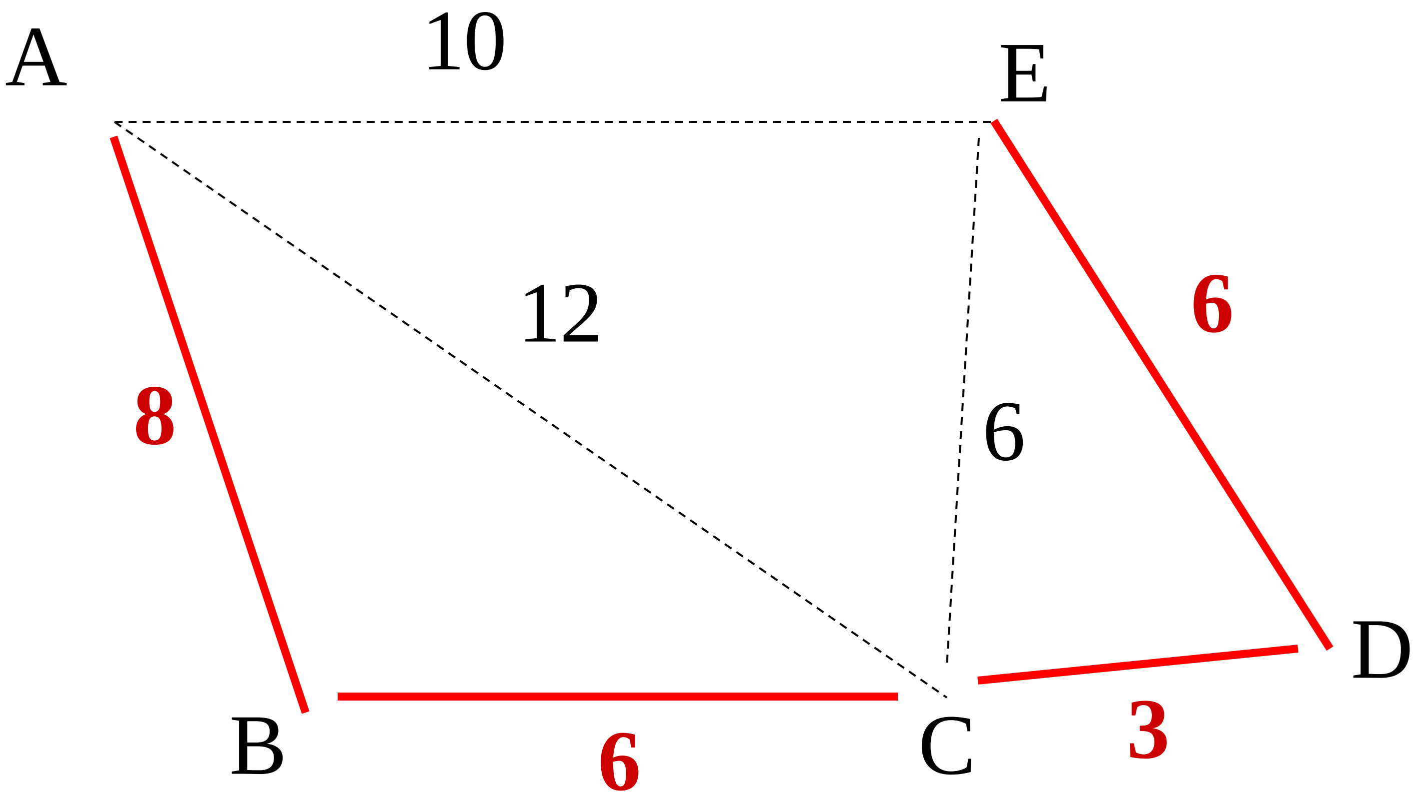
Example: graph



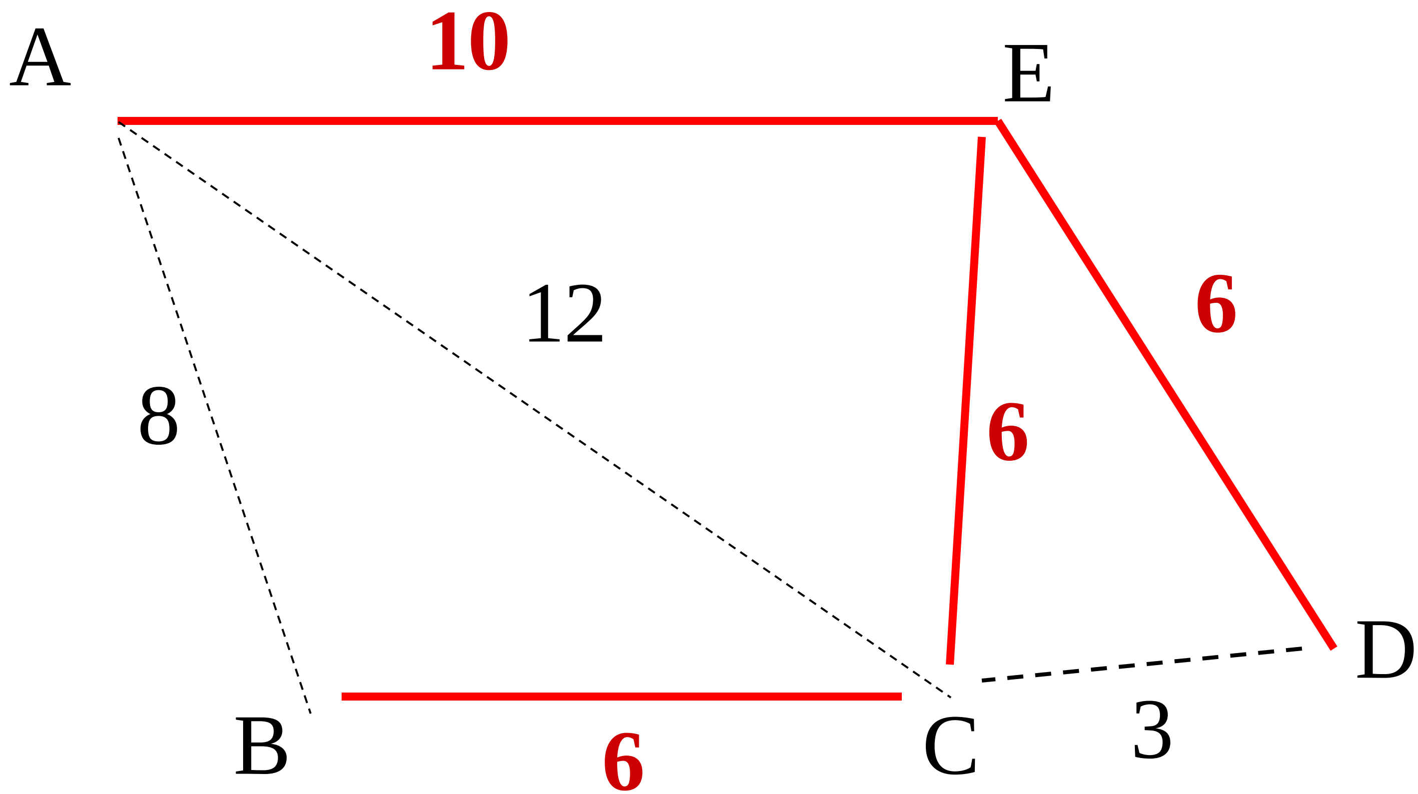
Example: MST (cost 23)



Example: another MST (cost 23)



Example: not MST (cost 28)



Why MST is a tree

- We want a minimum spanning sub-graph
  - a subset of the edges that is connected and that contains every node
- (Assuming all weights are non-negative)

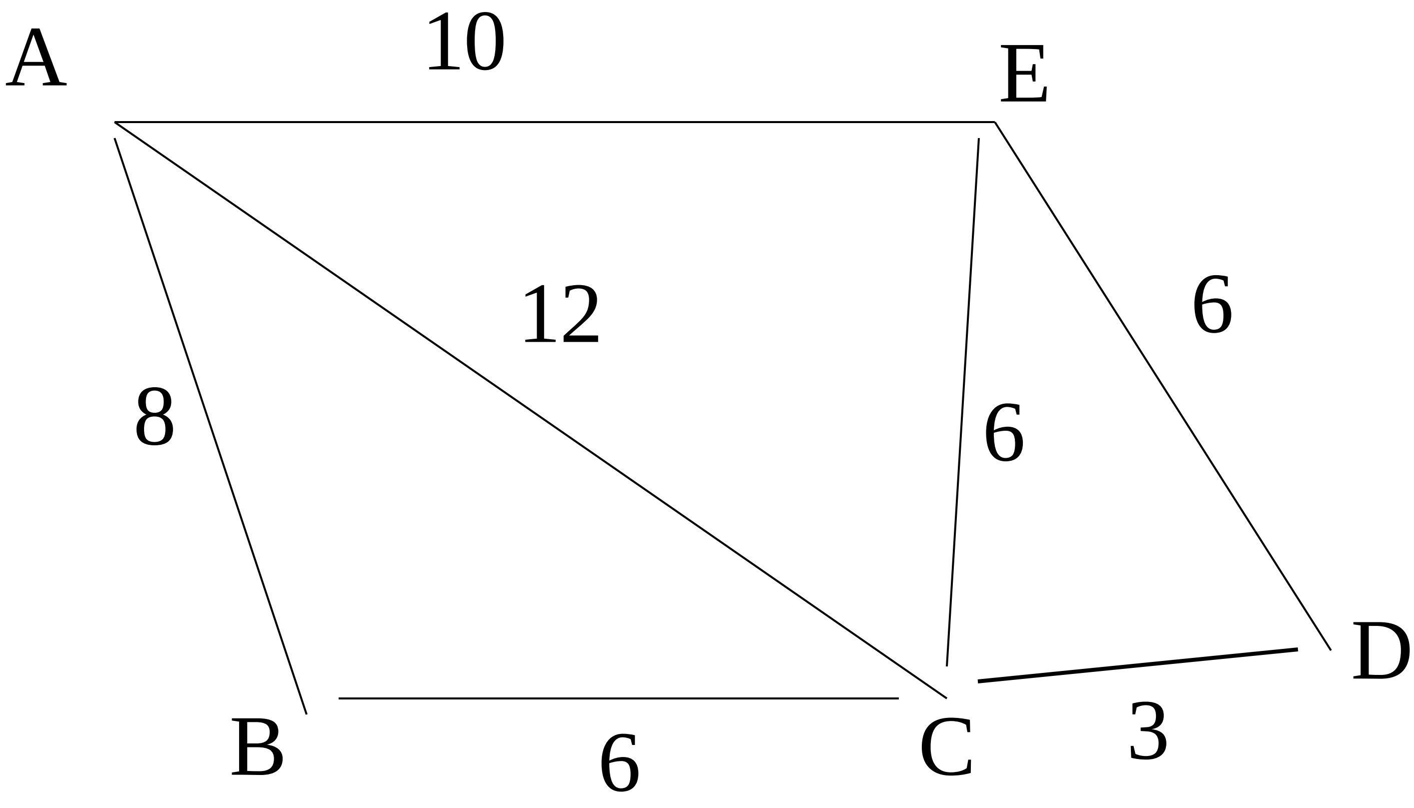
  If the graph has a cycle, then we can remove an edge of the cycle, and the graph will still be connected and will have a smaller weight.
- If a graph is connected and acyclic, then it is a tree.

Prim’s algorithm

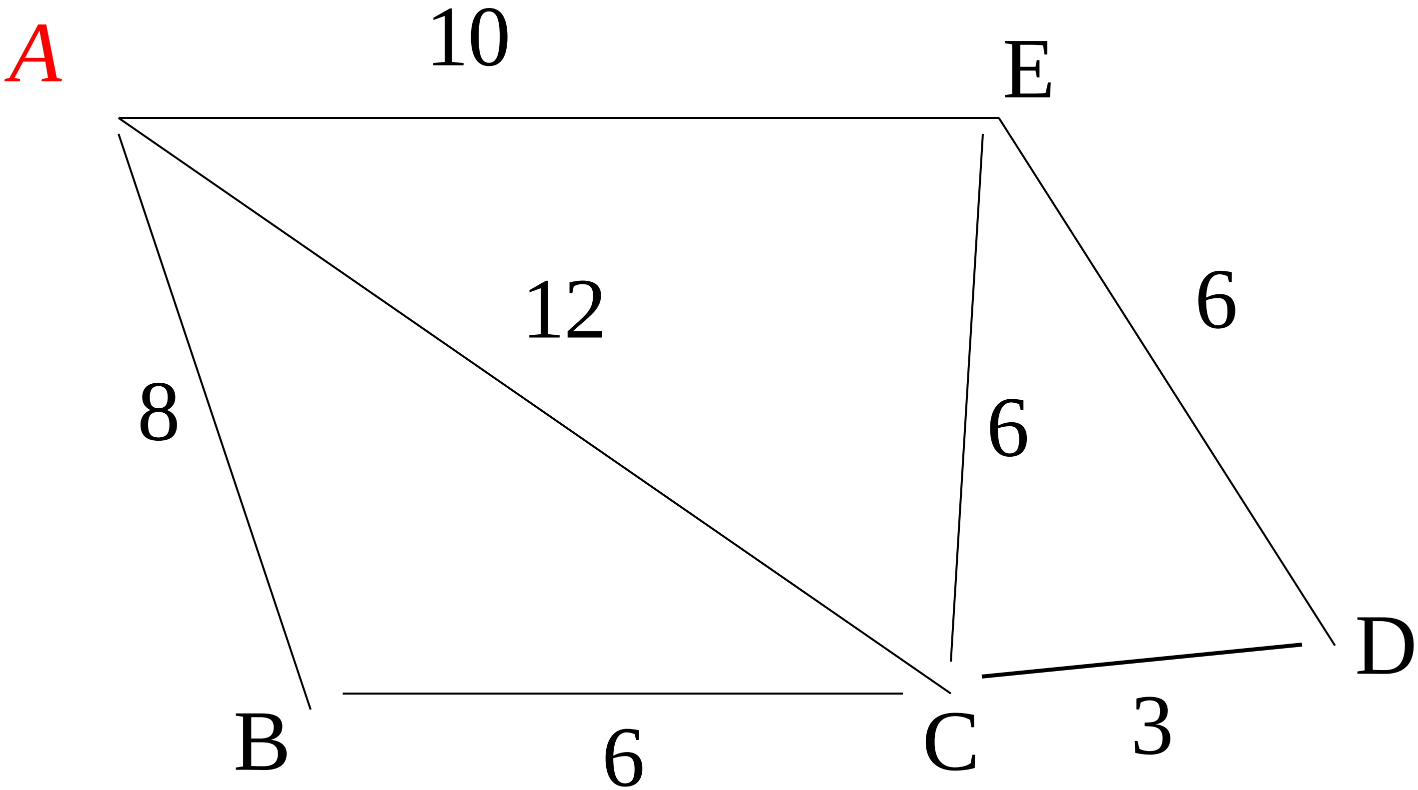
To construct an MST:

- Pick any vertex M
- Choose the shortest edge from M to any other vertex N
- Add the 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

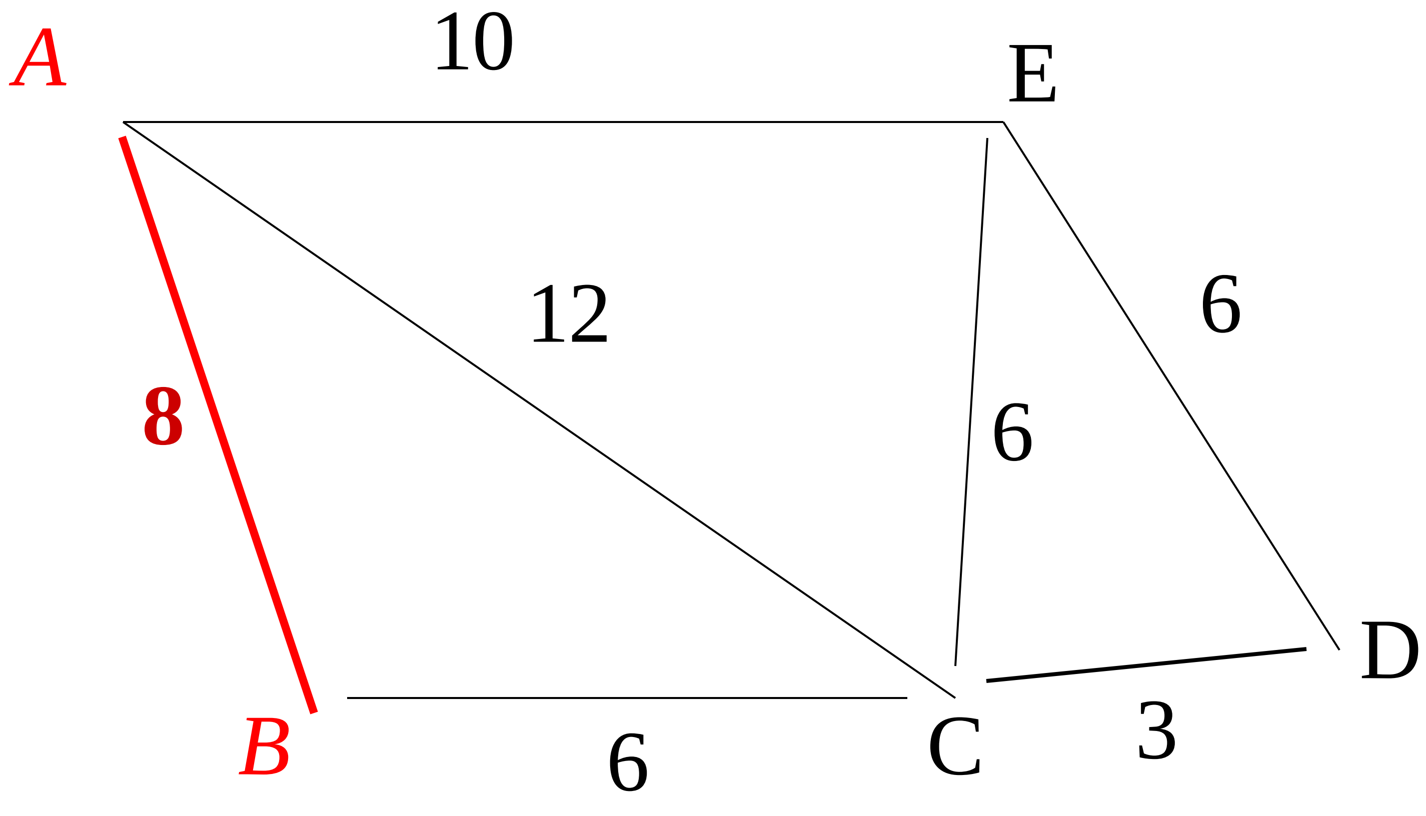
Example



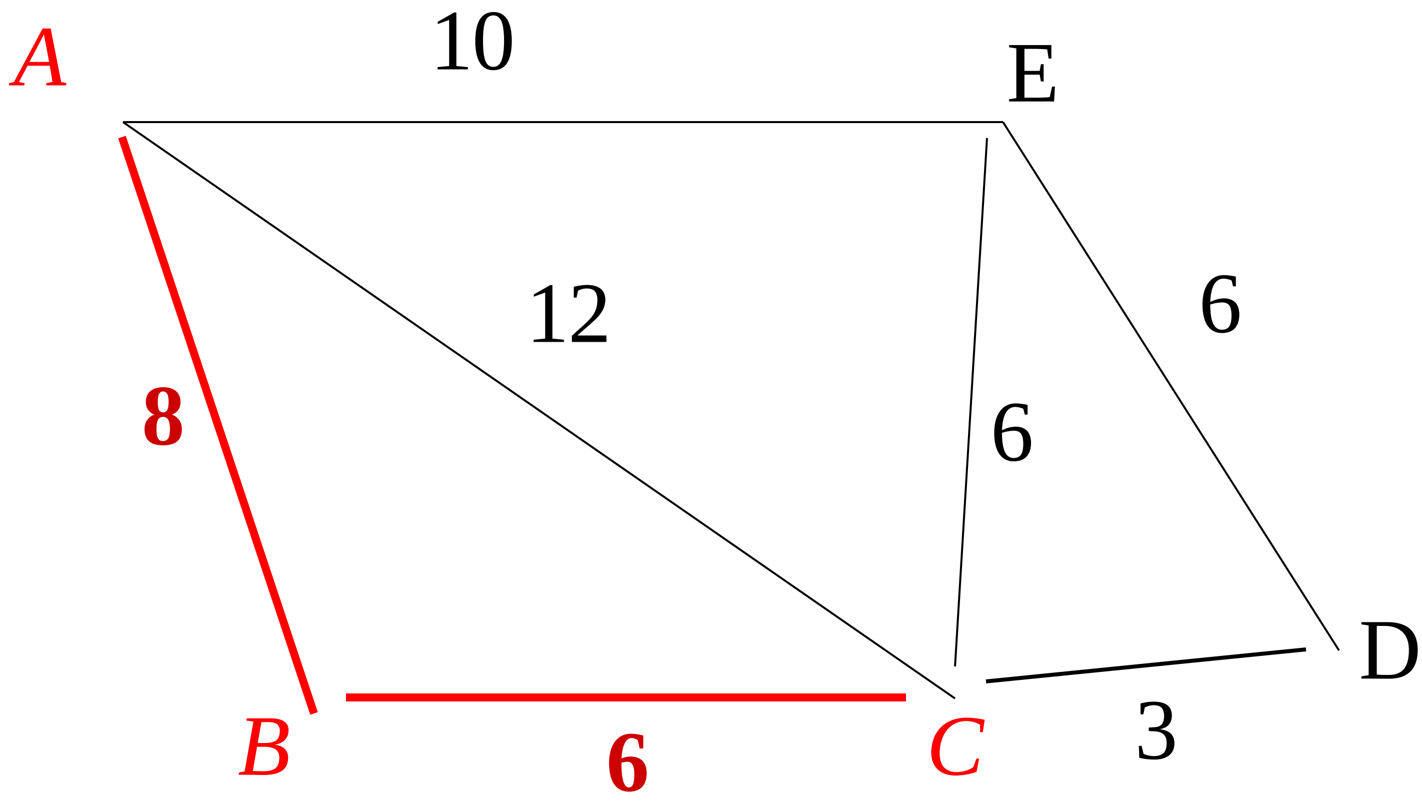
Example



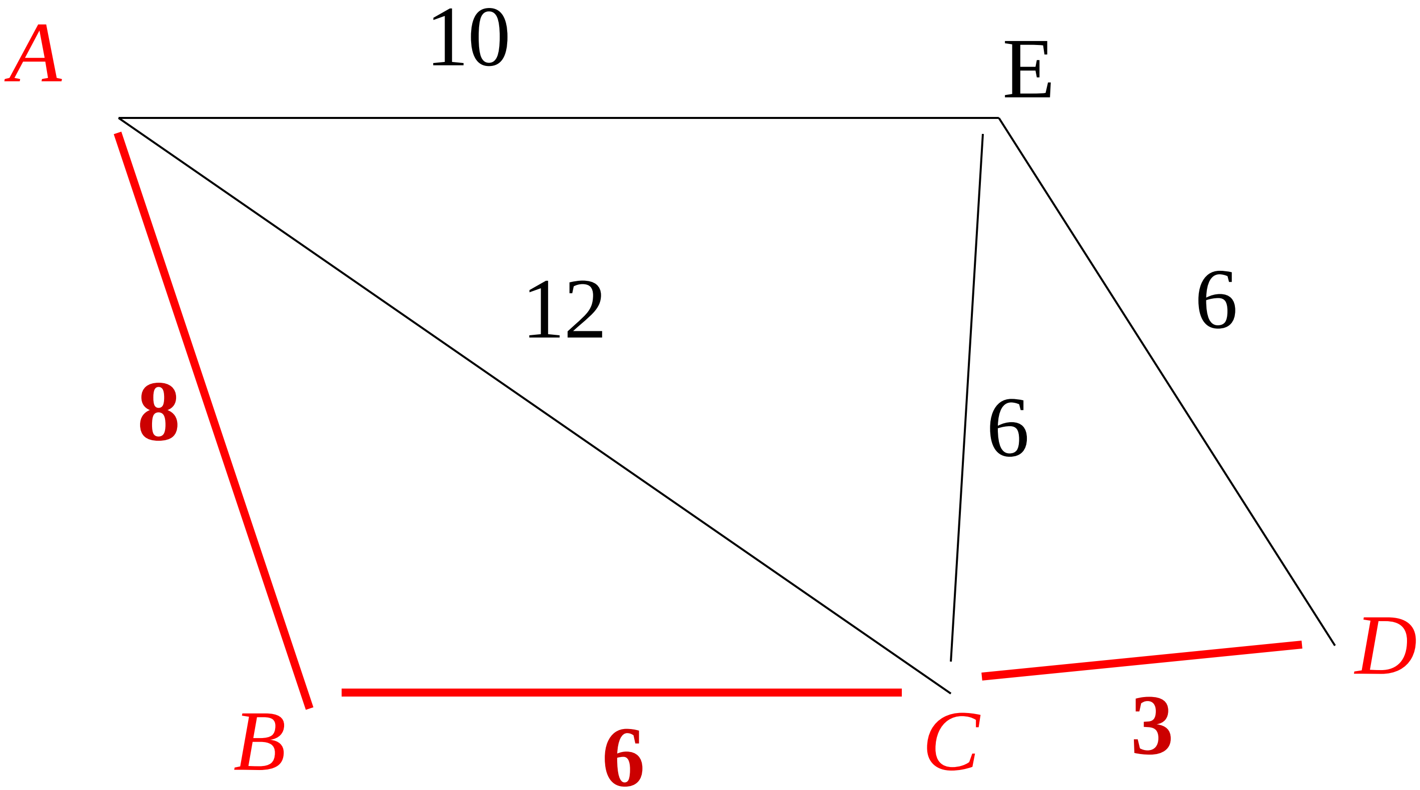
Example



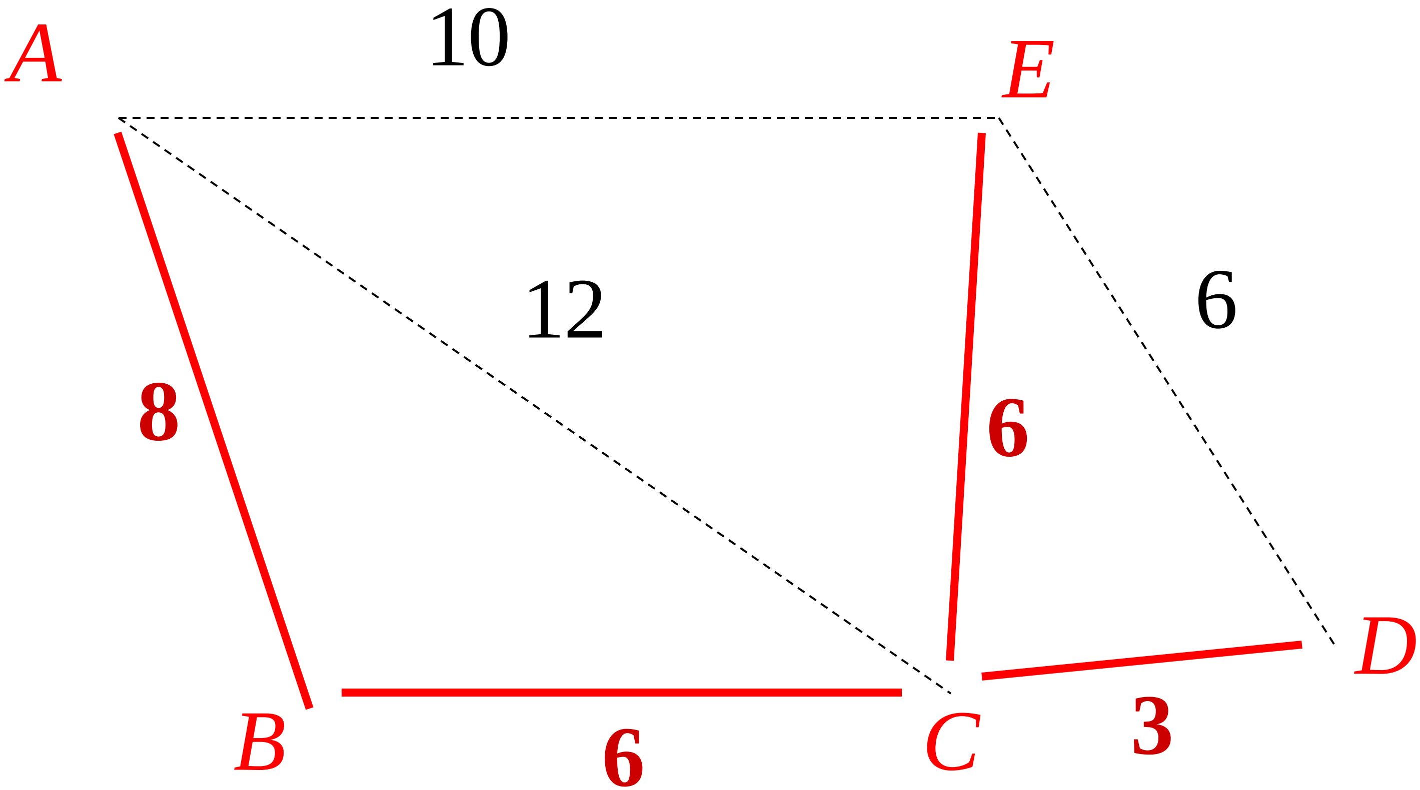
Example



Example



Example



Correctness of Prim's algorithm

Proposition 1: Let G be a weighted connected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. *There is a minimum spanning tree T that has e as one of its edges.*

Reading Section 14.7 Minimum Spanning Trees

Justification of Proposition 1

Let T be a minimum spanning tree of G. If T does *not* contain edge e, the addition of e to T must create a cycle. Therefore, there is some edge $f \neq e$ of this cycle that has one endpoint in V_1 and the other in V_2. Moreover, by the choice of e, $w(e) \leq w(f)$. If we remove f from $T \cup \{e\}$, we obtain a spanning tree whose total weight is no more than before. Since T was a minimum spanning tree, this new tree must also be a minmum spanning tree.

Self-Study

Let G be a weighted connected graph, if the weights in G are distinct, then the minimum spanning tree is unique. *Why*?

Reading Section 14.7 Minimum Spanning Trees

Greedy algorithm

Prim's algorithm for constructing a Minimal Spanning Tree is a ***greedy algorithm***:

- it just adds the shortest edge
- without worrying about the overall structure, without looking ahead
- It makes a locally optimal choice at each step.

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.
- e.g., coins of values 1, 3, 4, 5; change is 7.

Shortest path

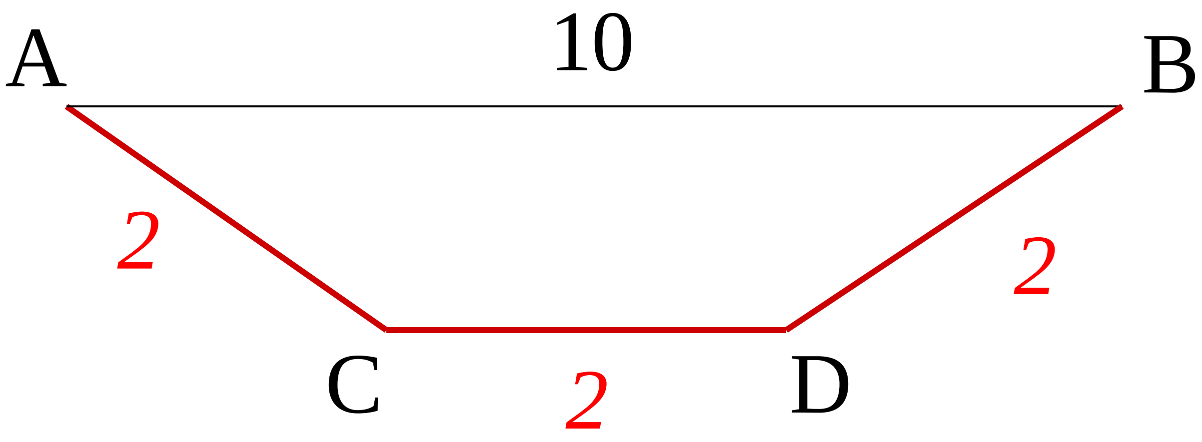
- Find the shortest route between two vertices u and v.
- It turns out that we can just as well compute shortest routes to ALL vertices reachable from u (including v). This is called *single-source shortest path problem* for weighted graphs, and u is the source.

Dijkstra’s Algorithm

- An algorithm for solving the single-source shortest path problem. Greedy algorithm.
- 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 the start vertex *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.

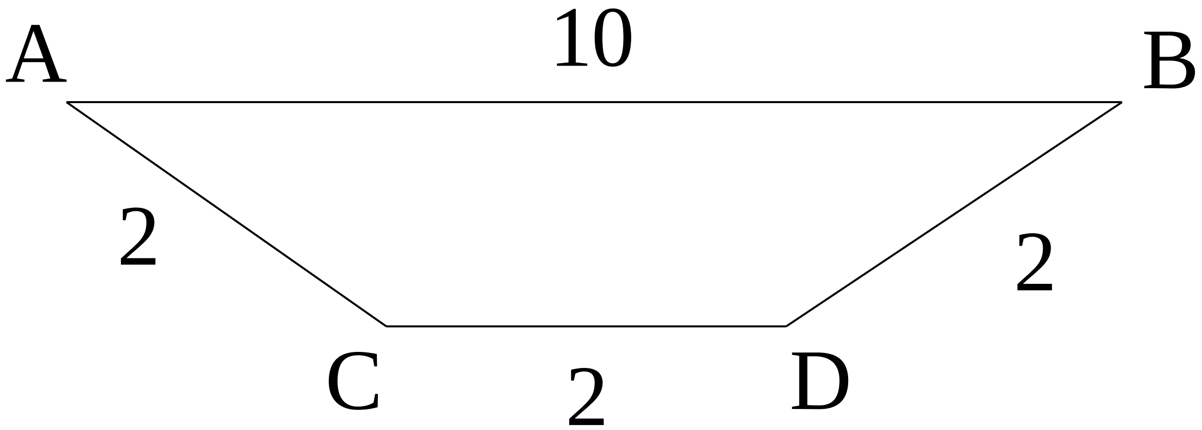
Dijkstra's algorithm

Loop while 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 $distance\ (s, u)\ +\ weight(u, v)$.

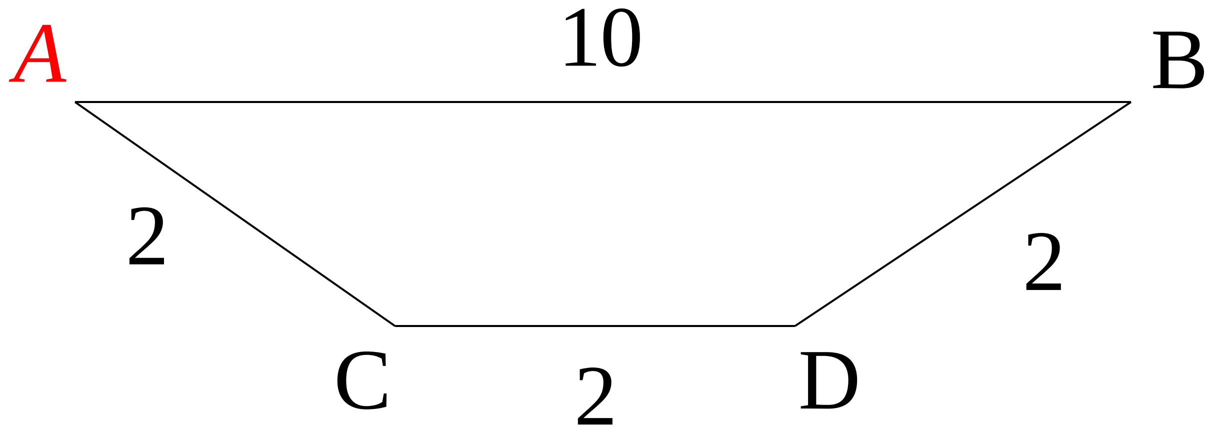
Example

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



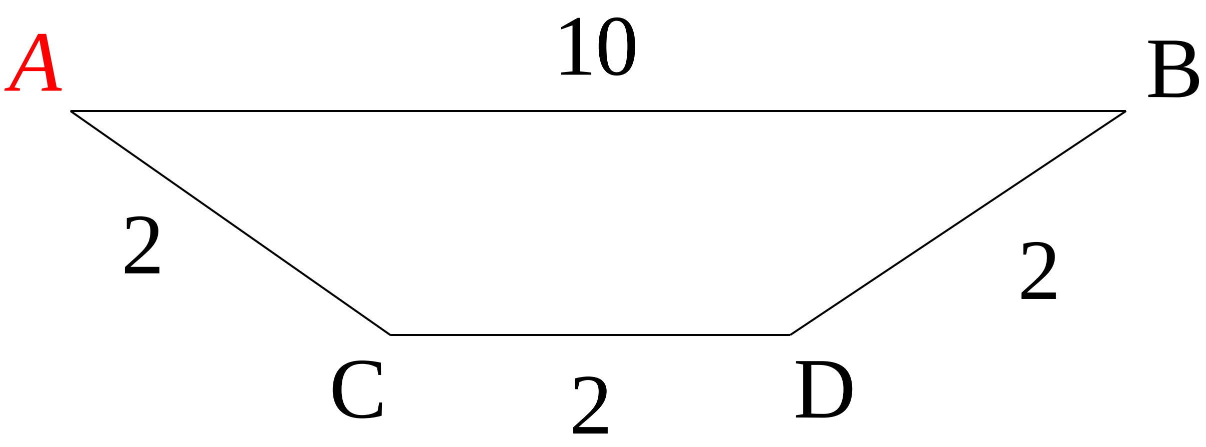
Example (dequeue A)

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



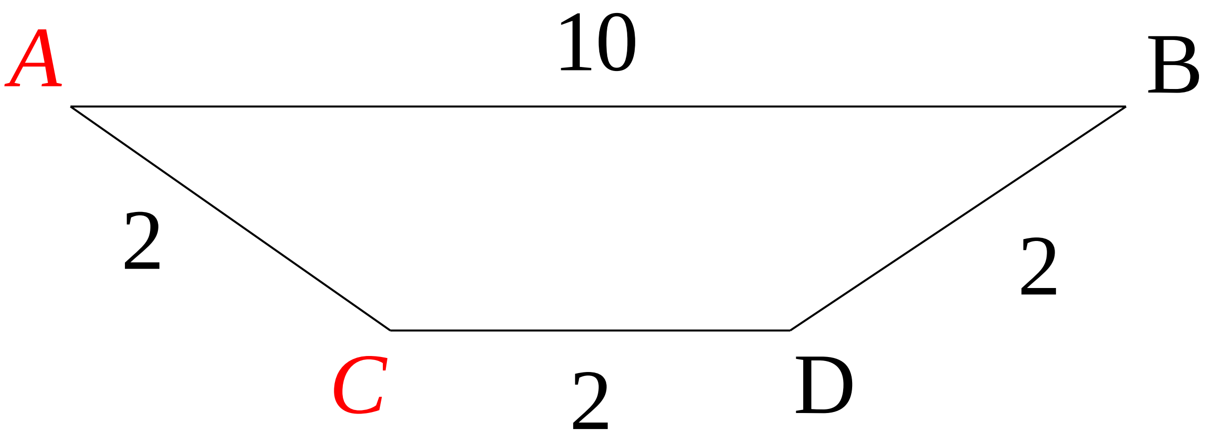
Example (recompute distances)

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



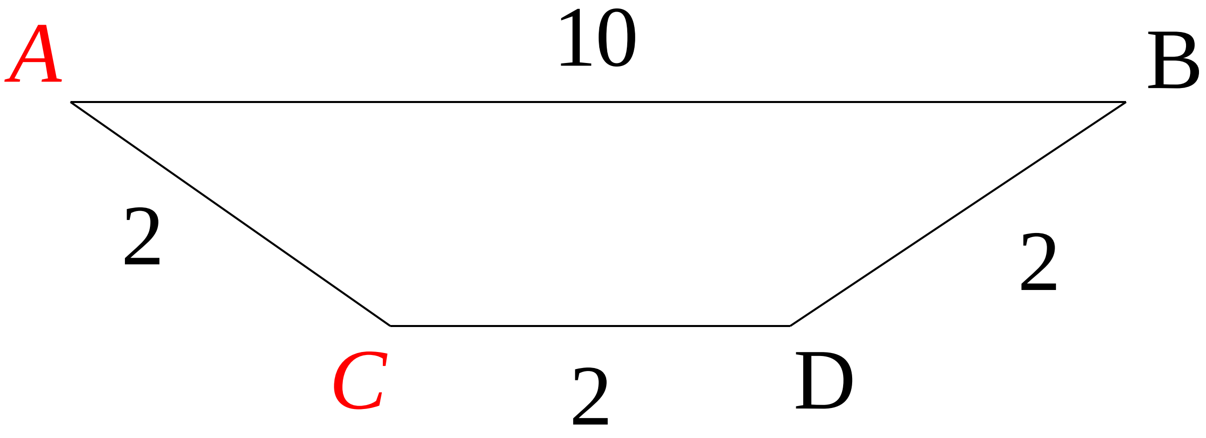
Example (dequeue C)

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



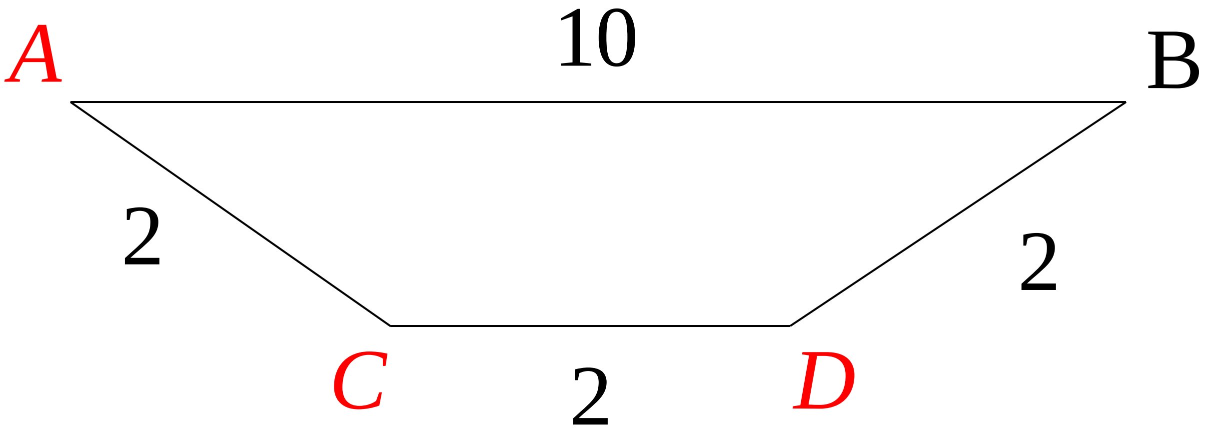
Example (recompute distances)

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



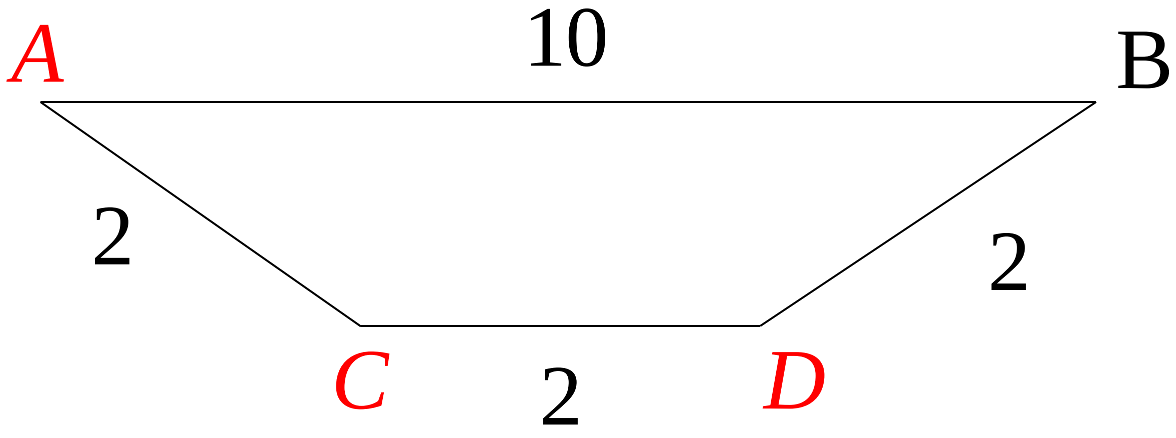
Example (dequeue D)

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



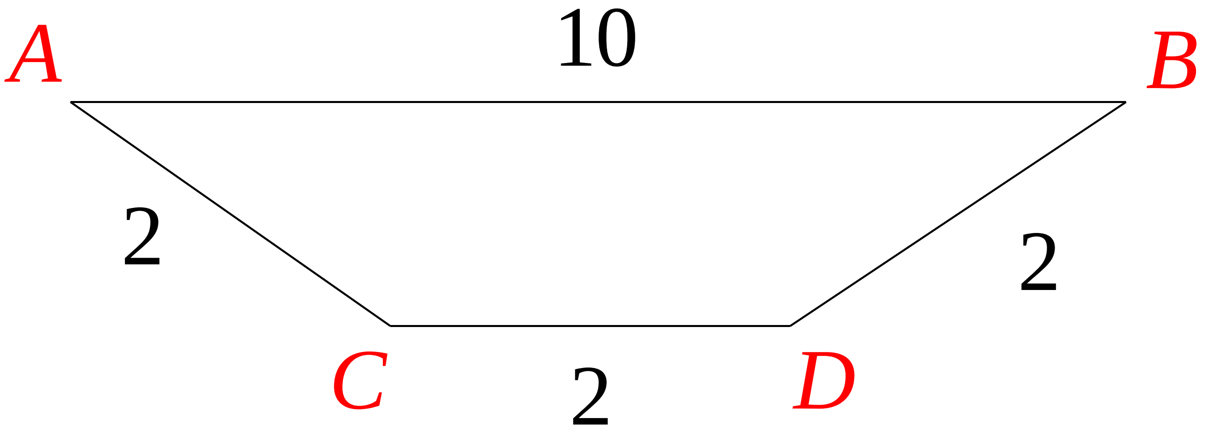
Example (recompute distances)

- 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

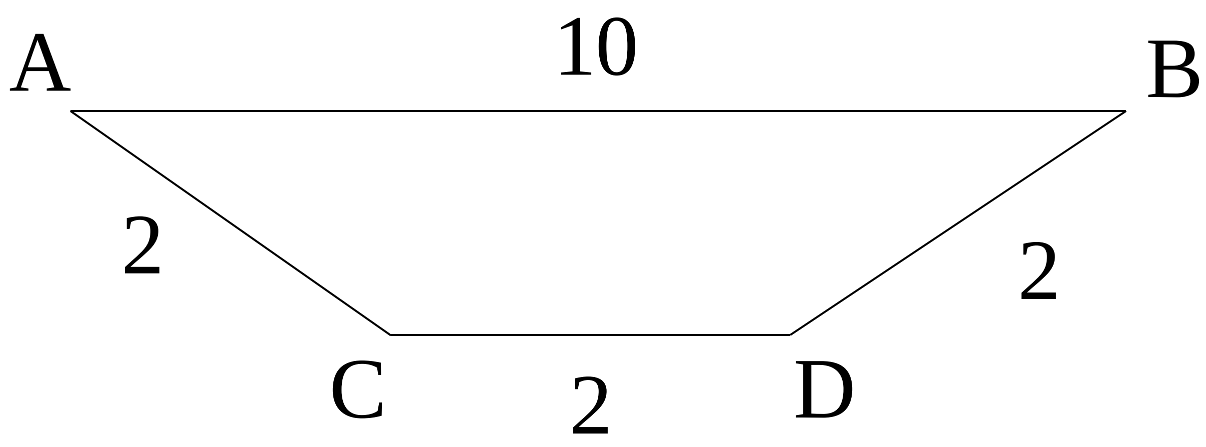
To make Dijkstra's algorithm to return the path itself, not just the distance:

- In addition to distances, maintain a path (list of vertices) for every vertex.
- At 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:

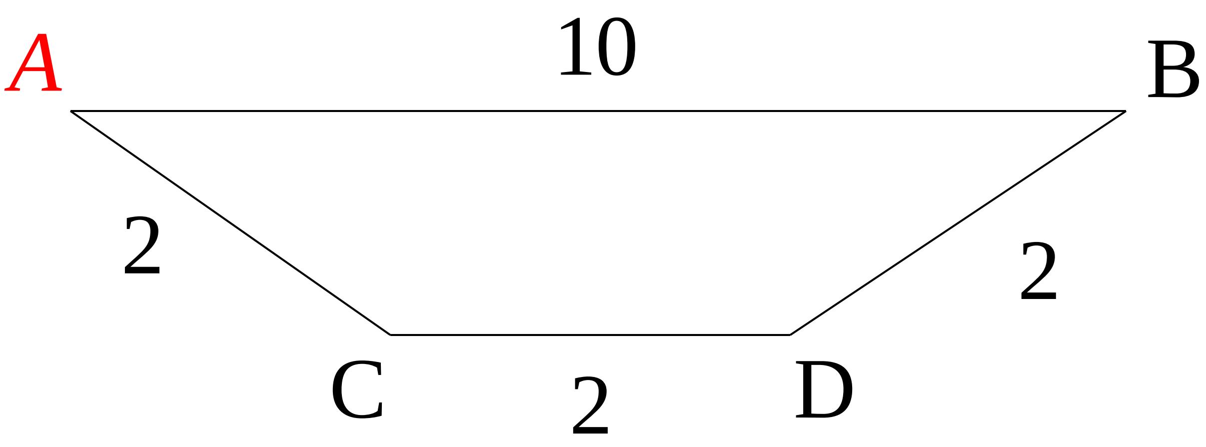
(A,0,{}), (B,INF,{}), (C,INF,{}), (D,INF,{})



Dequeue A, recompute paths

Distances and paths:

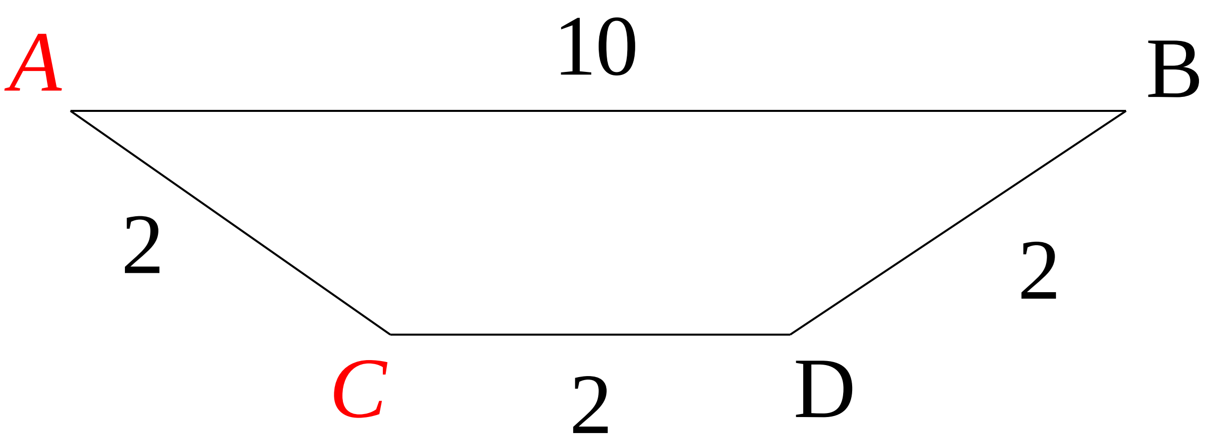
(A,0,{A}), (B,10,{A}), (C,2,{A}), (D,INF,{})



Dequeue C, recompute paths

Distances and paths:

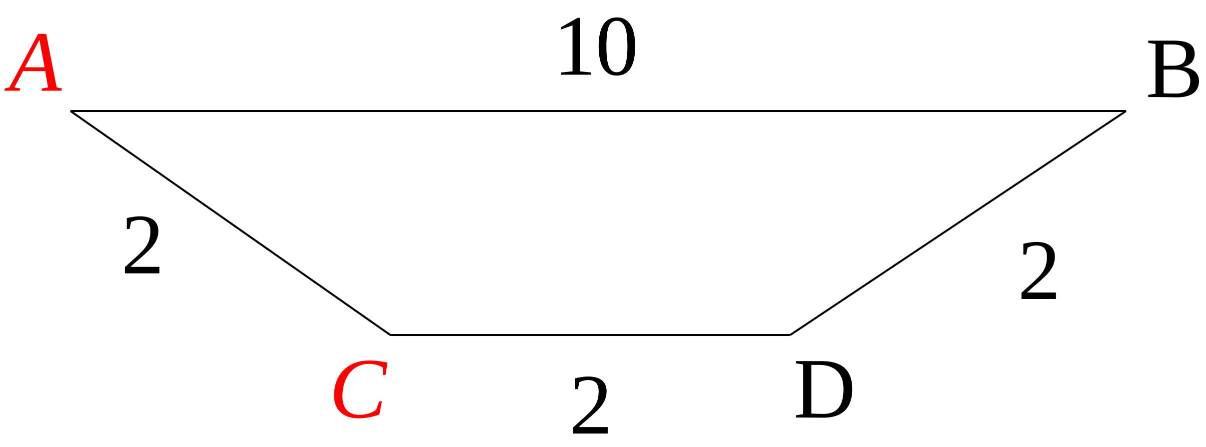
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,INF,{})



Dequeue C, recompute paths

Distances and paths:

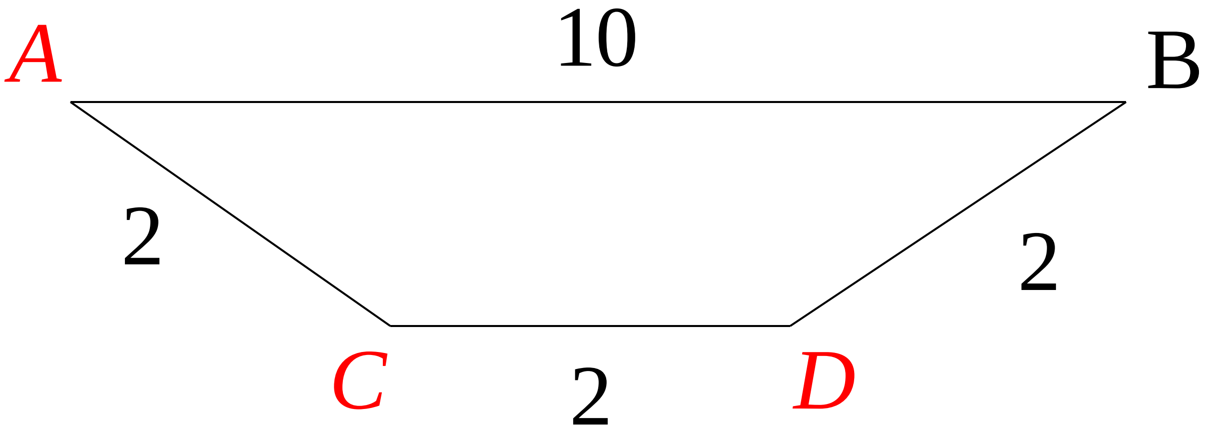
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,4,{A,C})



Dequeue D, recompute paths

Distances and paths:

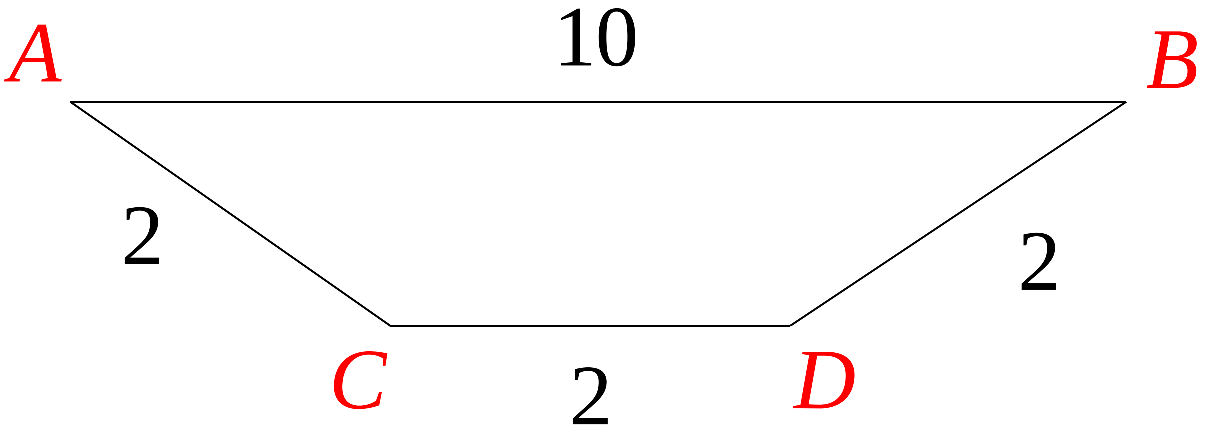
(A,0,{A}), (B,6,{A,C,D}), (C,2,{A,C}), (D,4,{A,C,D})



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})



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 $(n + 1)$ vertex.

Optimality proof

Assume that the $(n+1)$ vertex is u. It is at the front of the priority queue and its 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:

$$\begin{array}{ccccc} & v1 & & v2 & \\ s \text{ - - - - - - - - - - - - -} & & \longrightarrow & \text{- - - - - - - - - -} & u \end{array}$$

Optimality proof

Here the vertices from s to v_1 have correct shortest distances (inductive hypothesis) and v_2 is still in the priority queue.

$$s \dashrightarrow v_1 \longrightarrow v_2 \dashrightarrow u$$

Optimality proof

So $dist(s, v_1)$ is indeed the shortest path from s to v_1. Current distance to v_2 is $dist(s, v_2) = dist(s, v_1) + weight(v_1, v_2)$.

$$\begin{array}{ccccccc} & & v_1 & & v_2 & & \\ s & \text{- - - - - - - - - - - -} & & \longrightarrow & & \text{- - - - - - - - -} & u \end{array}$$

Optimality proof

If v_2 is still in the priority queue, then
$dist(s, v_1) + weight(v_1, v_2) \geq dist(s, u)$.

$$s \dashrightarrow v_1 \longrightarrow v_2 \dashrightarrow u$$

Optimality proof

But then the path going through v_1 and v_2 cannot be shorter than $dist(s, u)$. QED

$$s \text{- - - - - - - - - - - -} \; v_1 \longrightarrow v_2 \; \text{- - - - - - - - -} \; u$$

Complexity

- Assume that the priority queue is implemented as a heap;
- At each step (dequeueing a vertex u and recomputing distances) we do $O(|E_u|*\log(|V|))$ work, where E_u is the set of edges with source u.
- We do this for every vertex, so total complexity is $O((|V|+|E|)*\log(|V|))$.
- Really similar to BFS and DFS, but instead of choosing some successor, we re-order a priority queue at each step, hence the $*\log(|V|)$ factor.