Graph Algorithms

Edited by Heshan Du
University of Nottingham Ningbo China

Reading

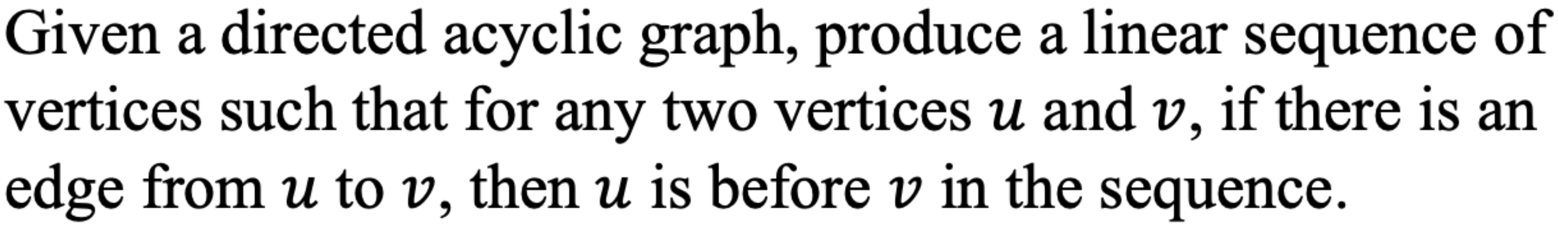
M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- 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.

拓扑排序（Topological Sort） 是应用在**有向无环图（DAG, Directed Acyclic Graph）**上的一种排序方法。
它的目标是：**生成一个线性序列，使得每一条有向边 u v 中，顶点 u 一定排在 v 之前。**
条件：图必须是 **有向** 且 **无环**（即 DAG）。

Topological Sort

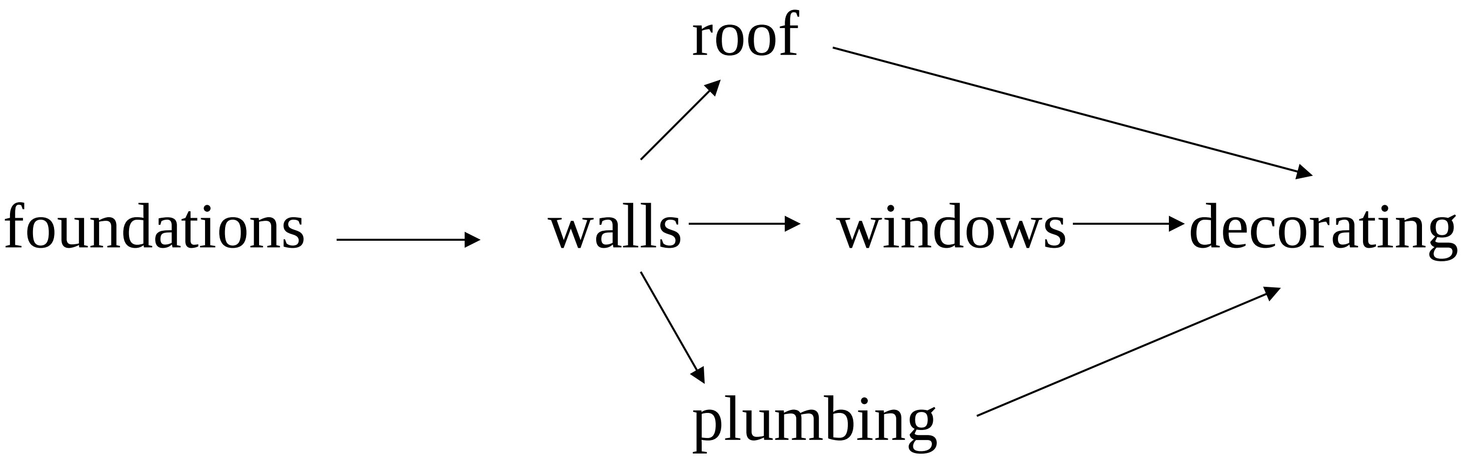
- *Input* to the algorithm: directed acyclic graph 一个有向无环图（DAG）

- *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.

  一组顶点的线性序列，使得：对于任意边 u v，在排序中 u 必须排在 v 之前。

- 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:

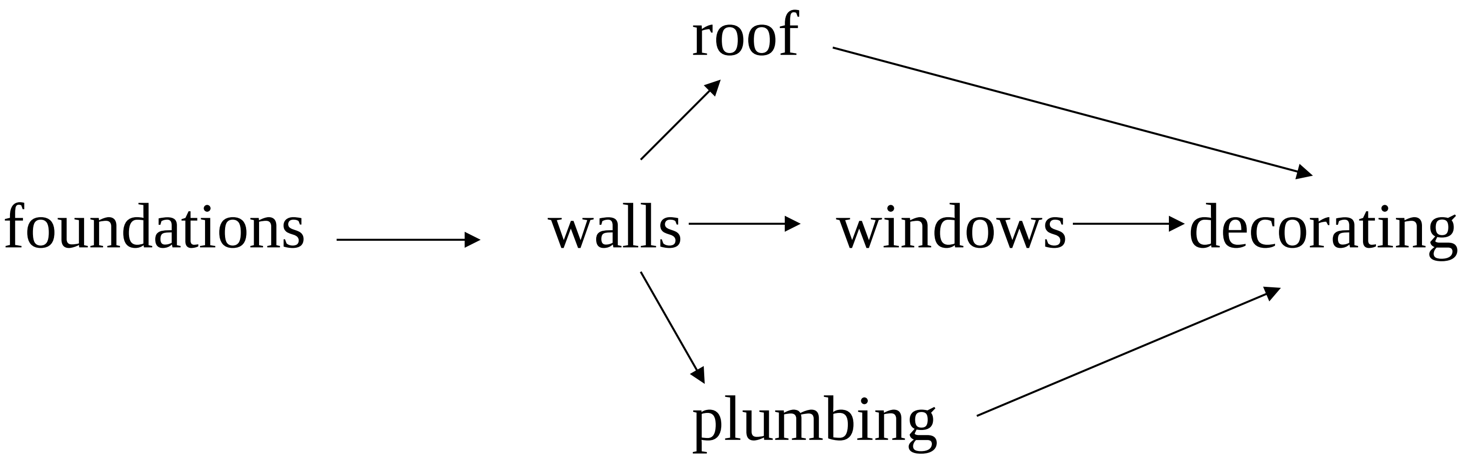
Foundations-Walls-Roof-Windows-Plumbing-Decorating

Applications

- Planning and scheduling 计划与调度

- The algorithm can also be modified to detect cycles. 检测环

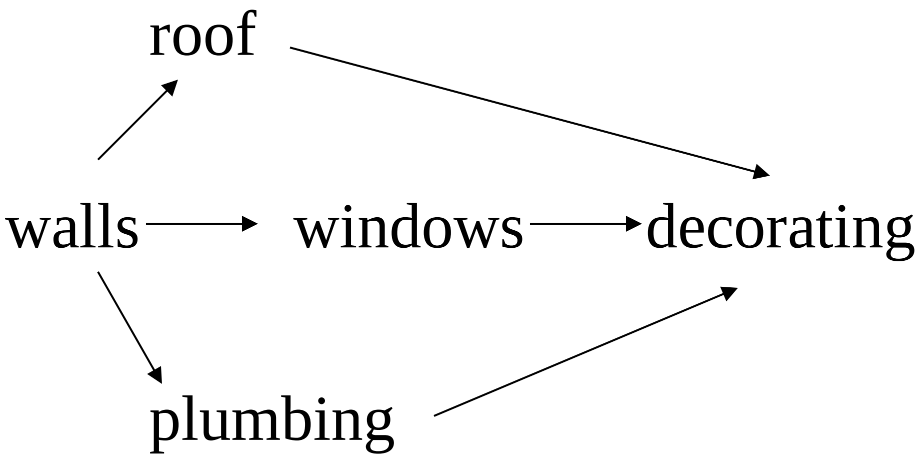
Example:



Array for the linear sequence: size 6

(Initially empty)

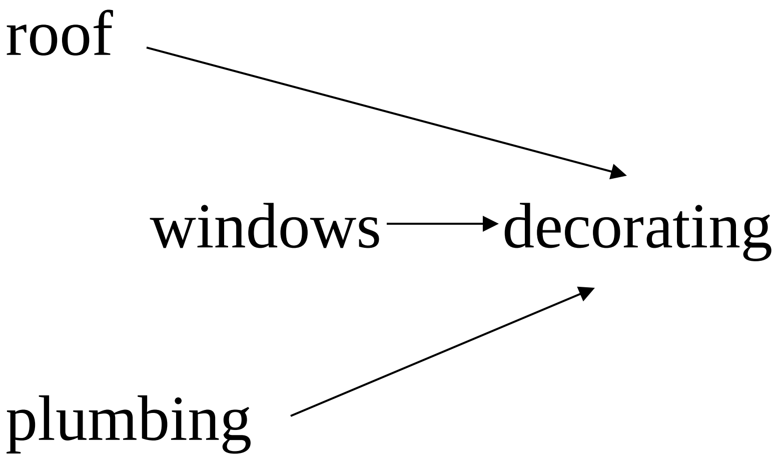
Example:



Array for the linear sequence: size 6

Foundations

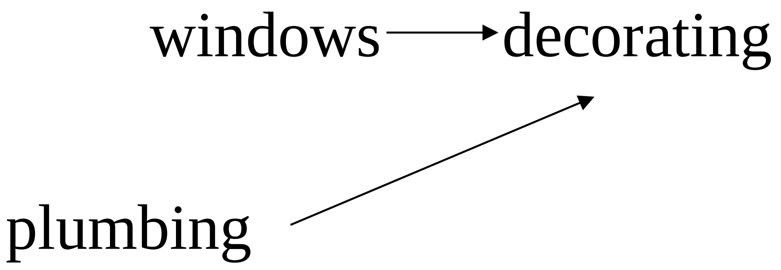
Example:



Array for the linear sequence: size 6

Foundations-Walls

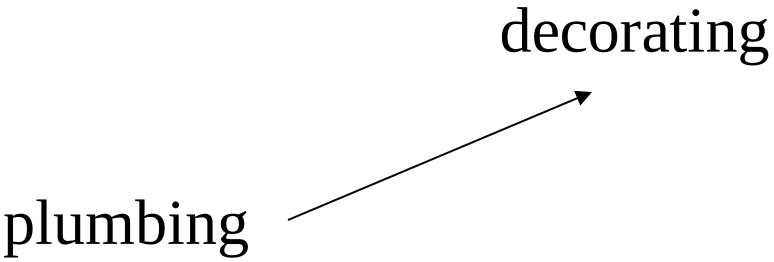
Example:



Array for the linear sequence: size 6

Foundations-Walls-Roof

Example:



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

创建一个空数组，长度为图中节点数（用于保存排序结果）。
当图中还有节点时，重复以下操作： 找出 **没有入边**（即入度为 0、无先决条件）的节点 v。
将这个节点 v**加入到结果数组** 中。
从图中 **删除该节点及其所有出边**（即从其它节点的入边列表中删除）。
注意事项：这个过程会**破坏原图结构**（因为删除了节点），所以推荐先**复制图的副本**再进行操作，以保留原图。

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?

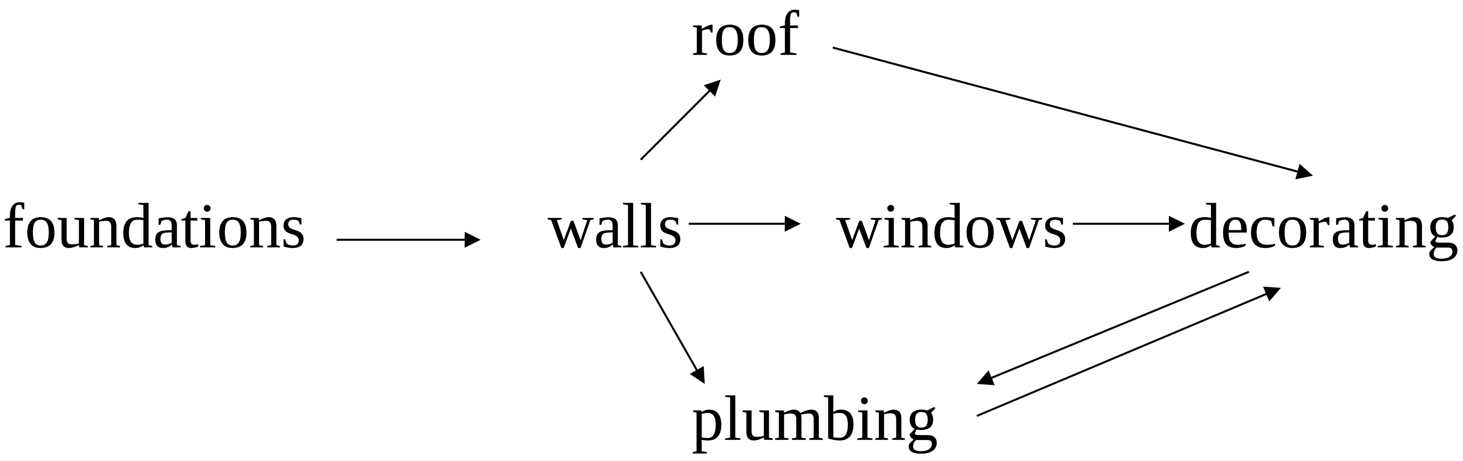
Cycle detection with topological sort

- 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.

不会找到入度为0的顶点，因为所有顶点都在某个环上，互相依赖，没有起点。
或者一开始能找到入度为0的节点，但在过程中会“卡住”找不到新的入度为0的节点。

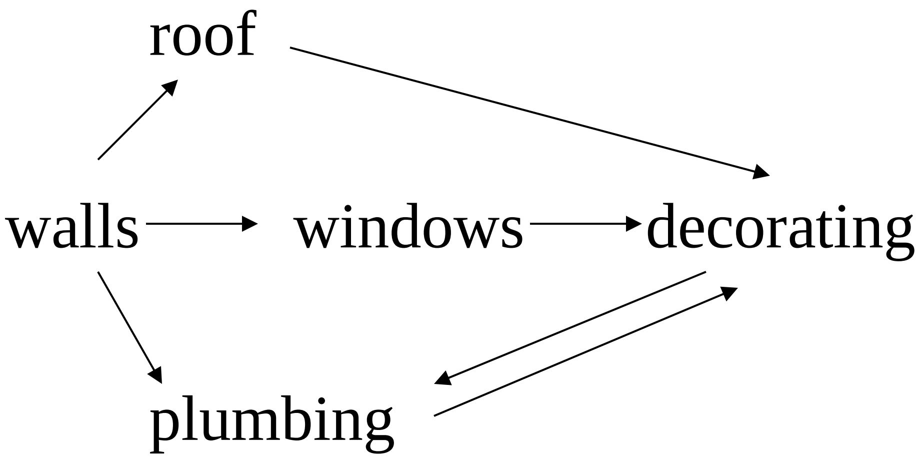
如果你运行拓扑排序，**最终还剩下节点未被删除（或未排序）**，那么图中一定存在环。

Example: building a house with a vicious circle

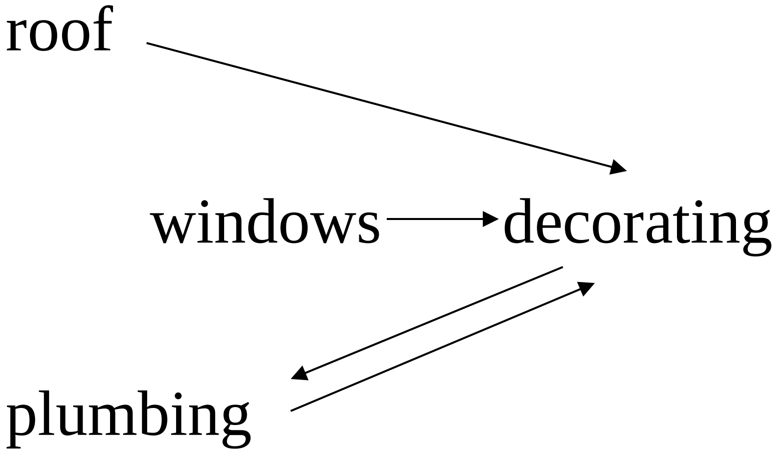


Plumbing depends on decorating and decorating on plumbing

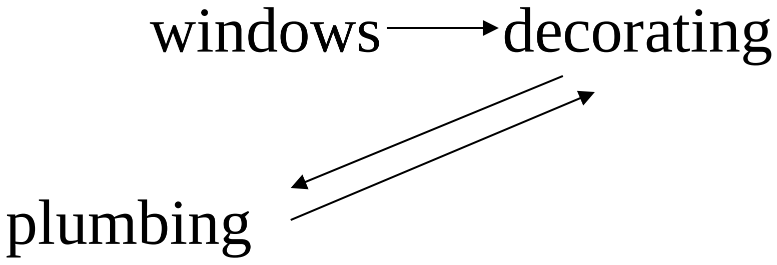
Example: building a house with a vicious circle



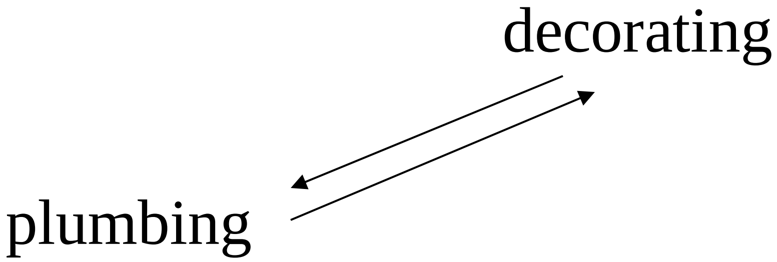
Example: building a house with a vicious circle



Example: building a house with a vicious circle



Example: building a house with a vicious circle



decorating

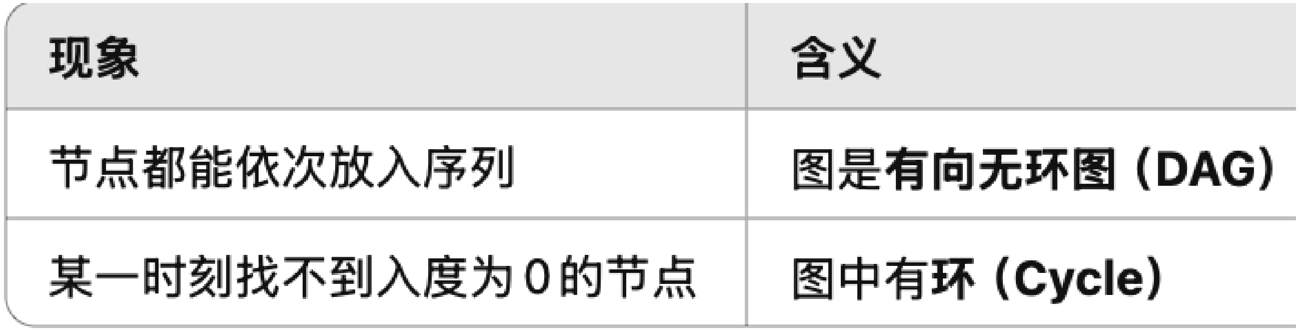
plumbing

没有任何节点入度为0
拓扑排序无法继续
说明图中存在一个 **环（Cycle）**

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.



现象	含义
节点都能依次放入序列	图是**有向无环图（DAG）**
某一时刻找不到入度为0的节点	图中**有环（Cycle）**

Spanning Tree **生成树**

一个连通的无向图（connected, undirected graph）

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

一棵树（无环图）：
连接所有顶点
只使用原图中的边
边数 = 顶点数 1（n1）

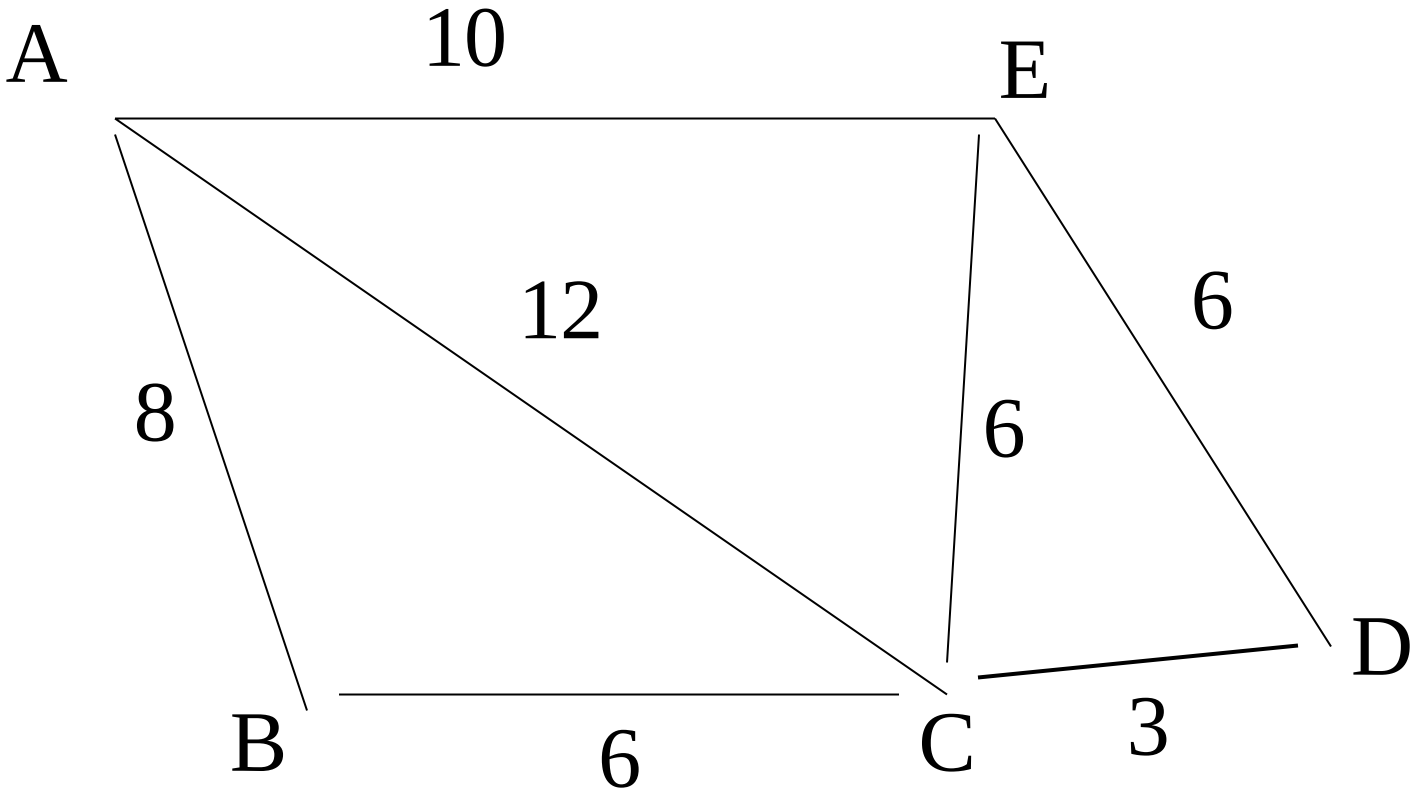
生成树就是从原图中挑出一些边，**既能保持图的连通性，又不形成环**。

Minimal Spanning Tree 最小生成树

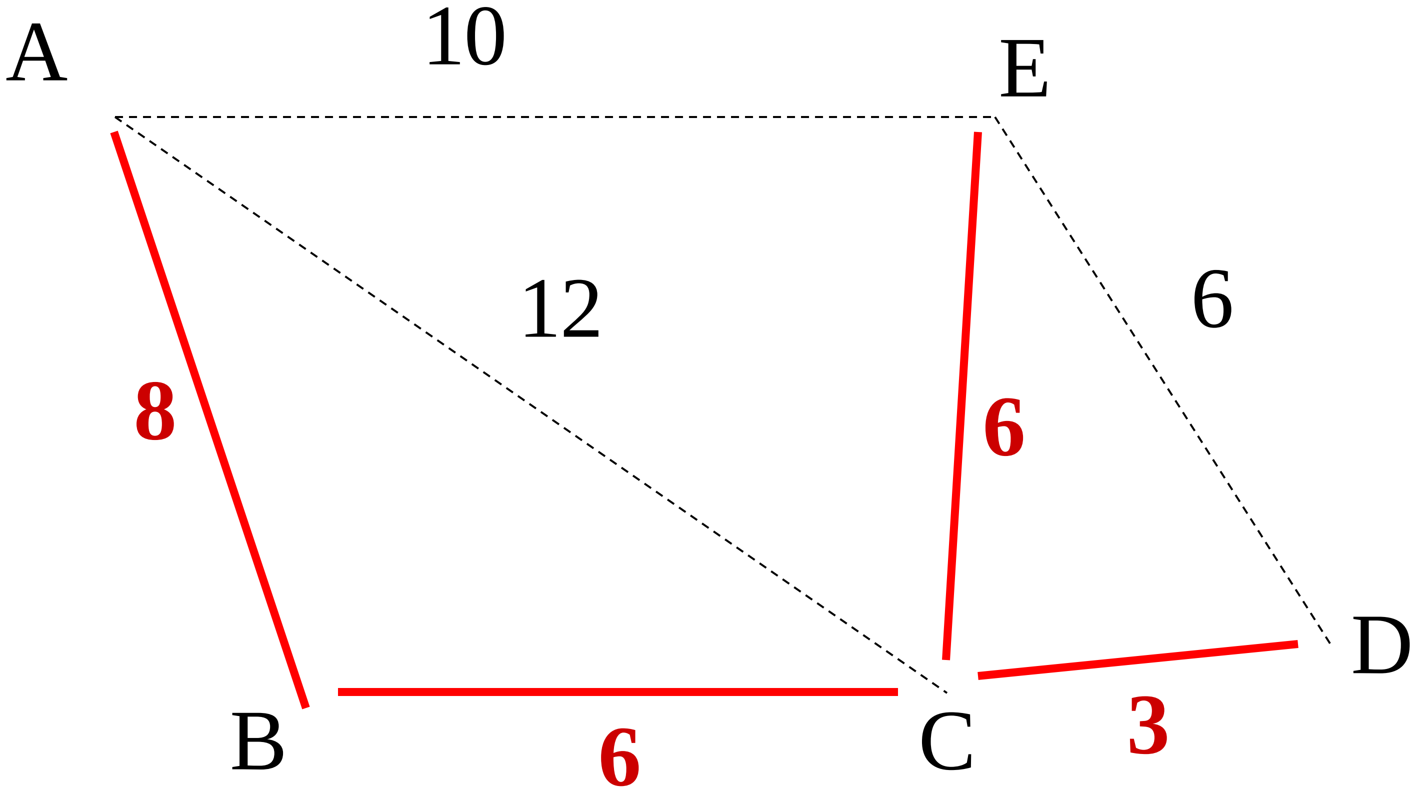
- *Input*: connected, undirected, weighted graph 一个连通的、无向的、有权图
- *Output*: a spanning tree
  - – (connects all vertices in the graph using only the edges present in the graph)
  - – and is *minimal* in the sense that the sum of weights of the edges is the smallest possible

一棵**生成树**，满足以下两个条件：
连接图中所有顶点
权重之和最小

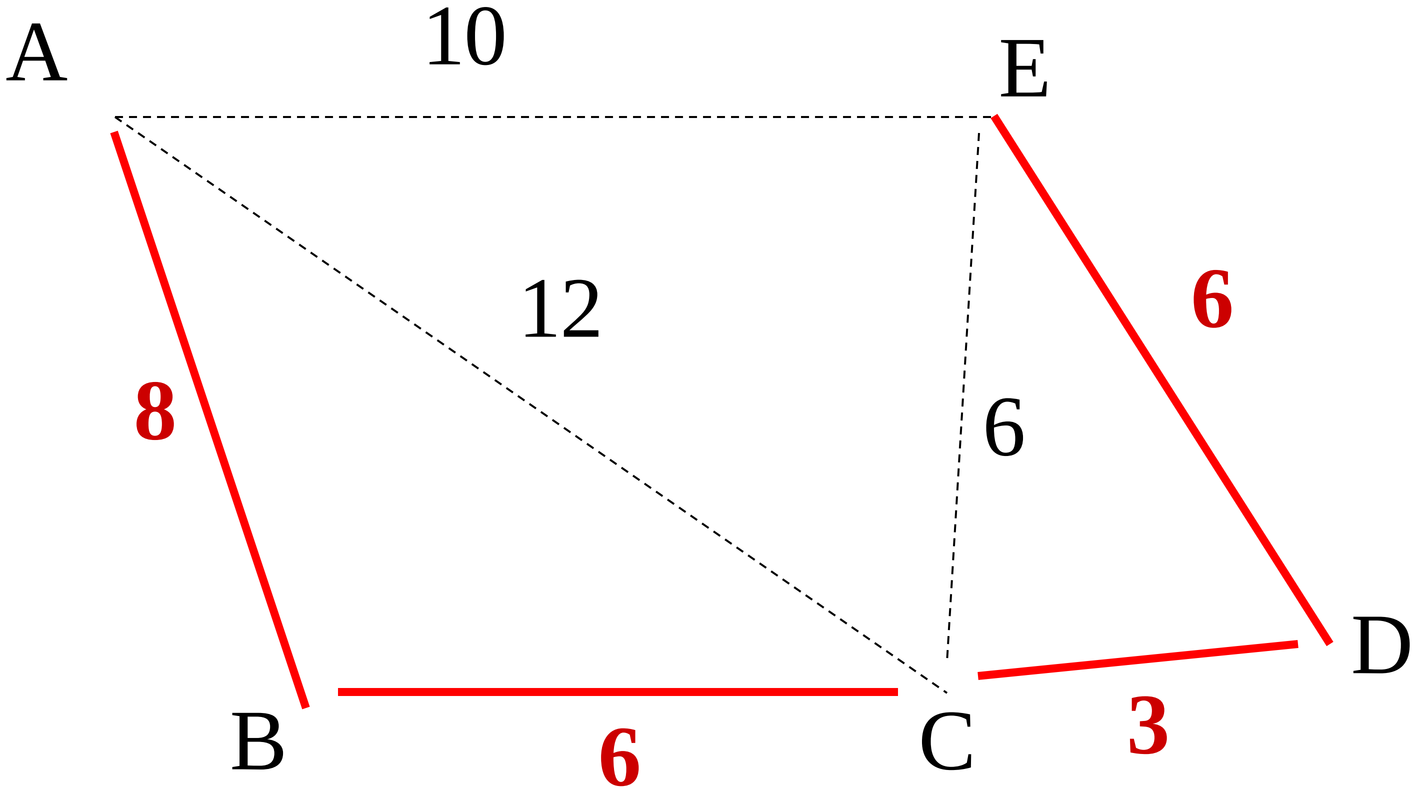
Example: graph



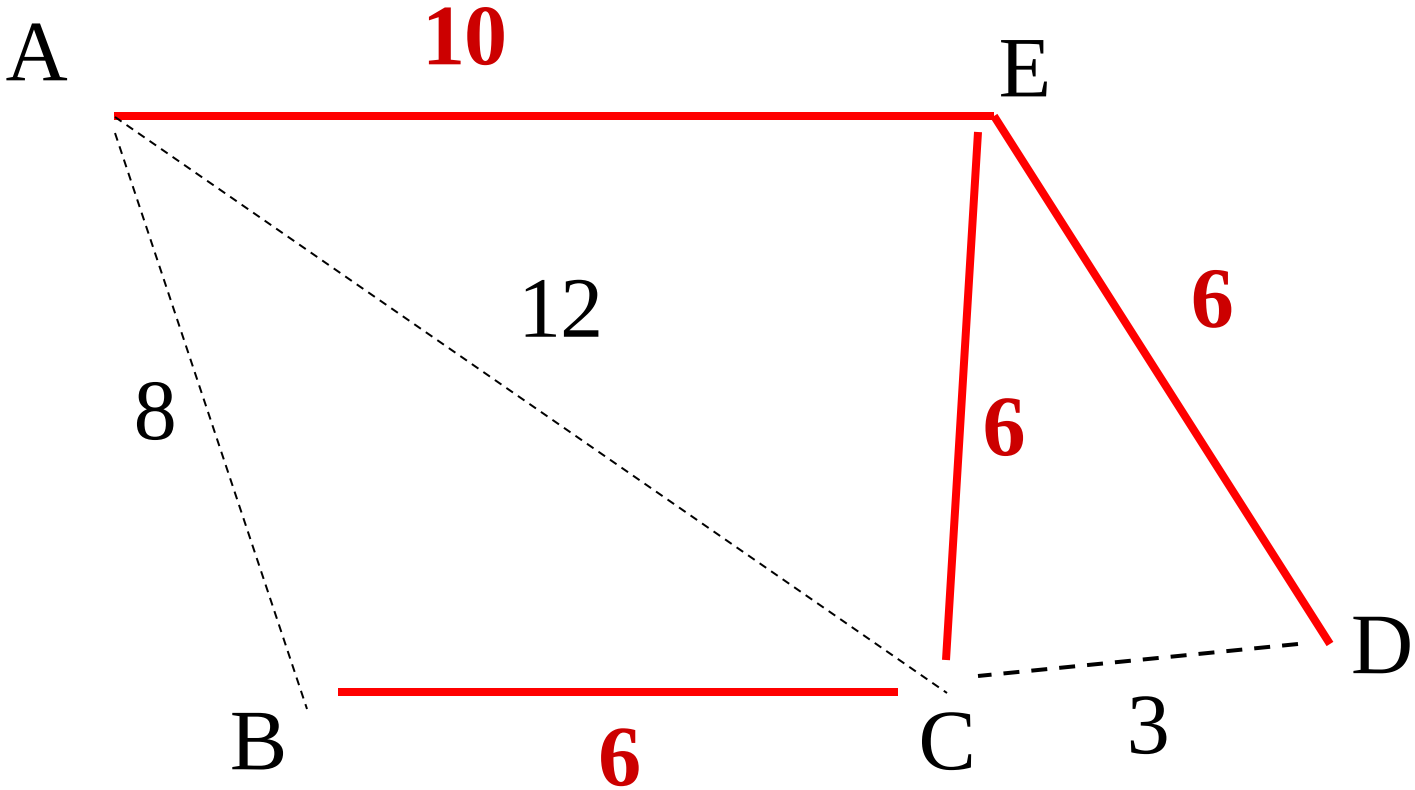
Example: MST (cost 23)



Example: another MST (cost 23)



Example: not MST (cost 28)



我们需要的是最小生成子图（Minimum Spanning Subgraph）：即一个连接所有节点的边的子集。
假设所有权值都是非负的（non-negative）：如果生成的子图中**包含环（cycle）**：可以**删除这个环上的一条边**而不破坏连通性；这样会**减少总权重** 所以原来的图就不是最小的；所以**MST 不可能含有环**。
如果一个图是连通的、且无环，那么它就是一棵树。

Why MST is a tree

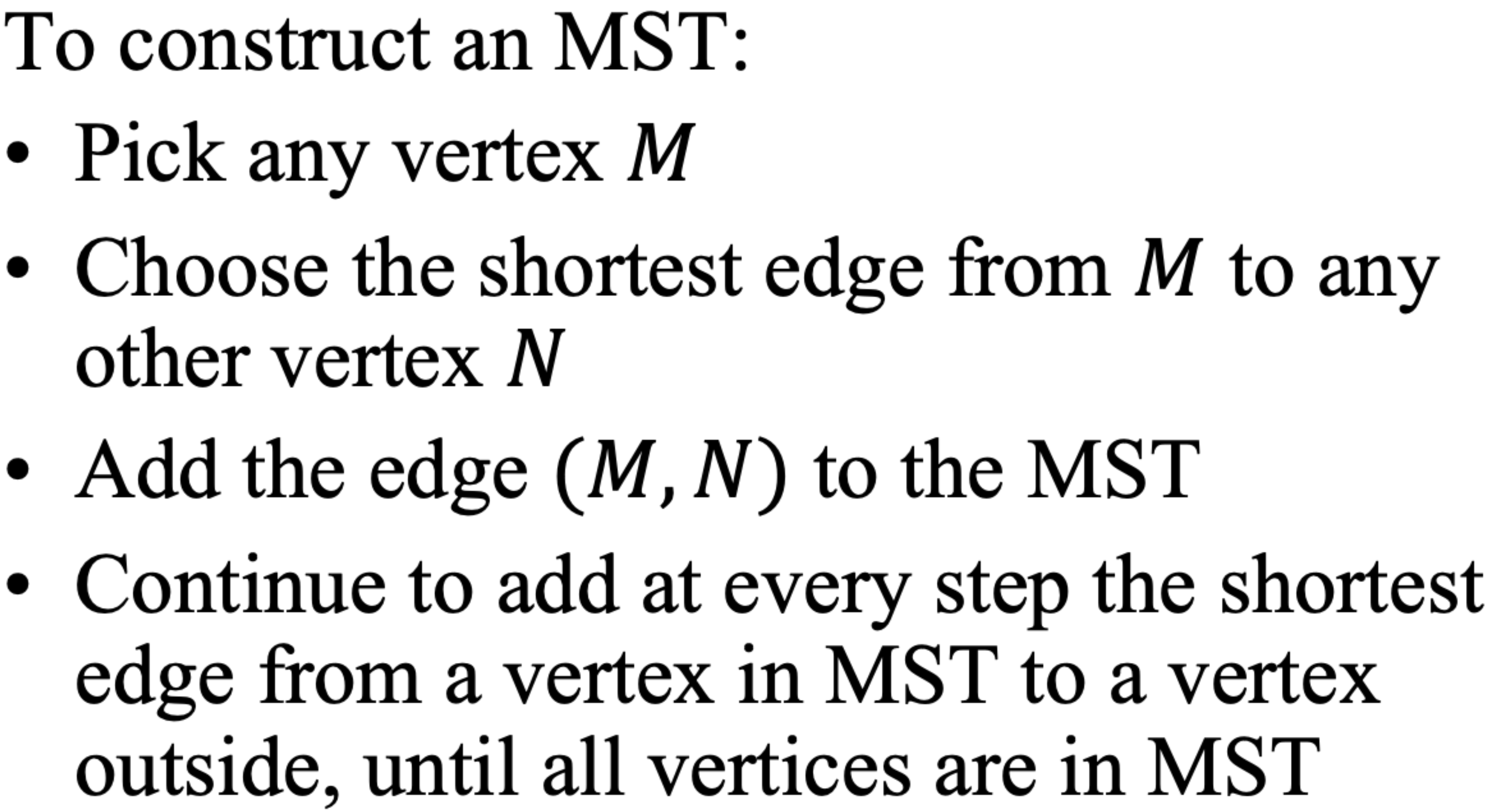
目标：找到一个**连通子图**，使用图中的边，使得：包含所有顶点；总权重最小；无环（即为树结构）。

- 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.

选择任意一个顶点M 作为起点。
从当前生成树中的顶点 M，**选取一条权重最小的边**(M, N)，使得：M 在当前生成树中；N 不在生成树中。**将边** (M, N) 加入到生成树中。
重复步骤 2：每次从生成树中的任意顶点出发；选择连接 “ 树内 ” 和 “ 树外 ” 的最短边；直到**所有顶点都加入生成树为止**。

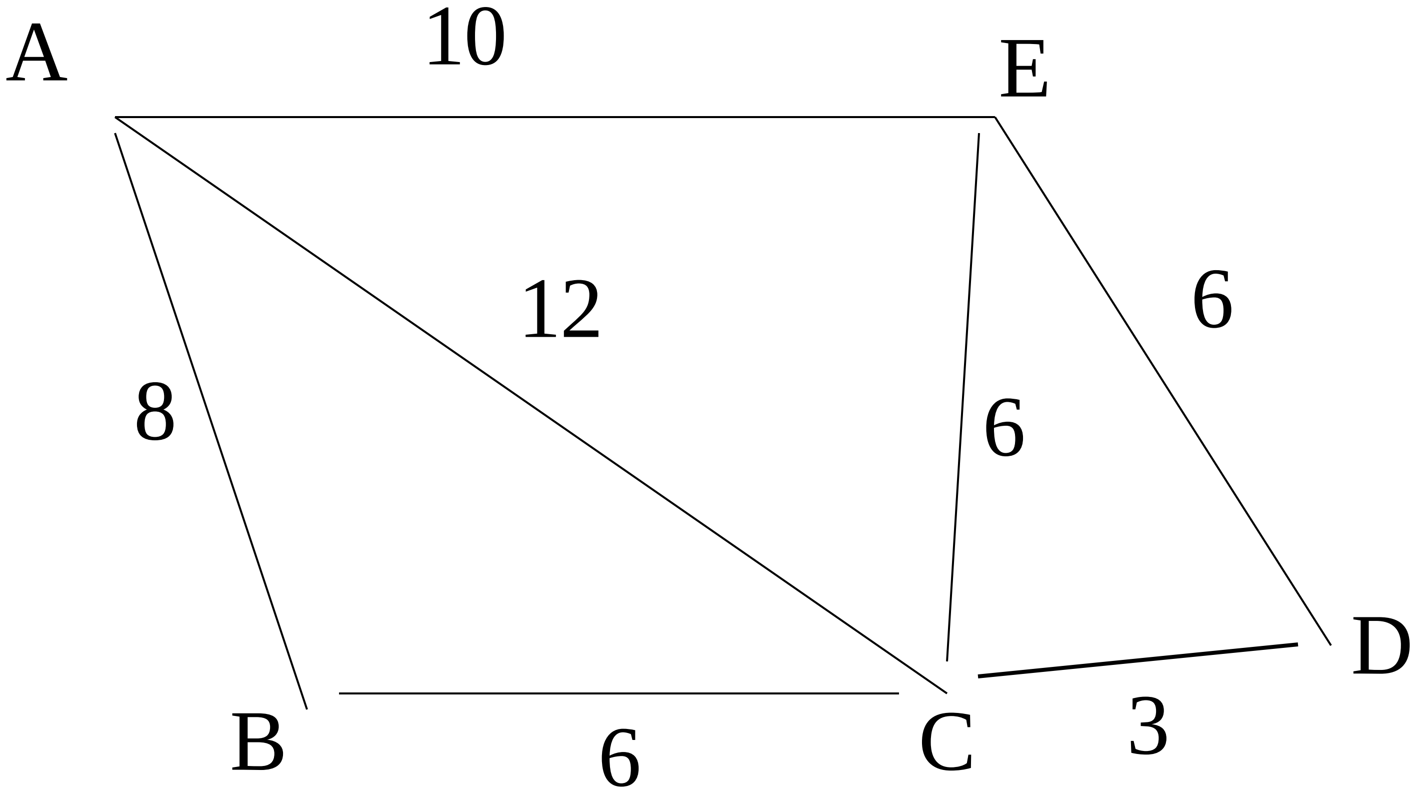
Prim’s algorithm



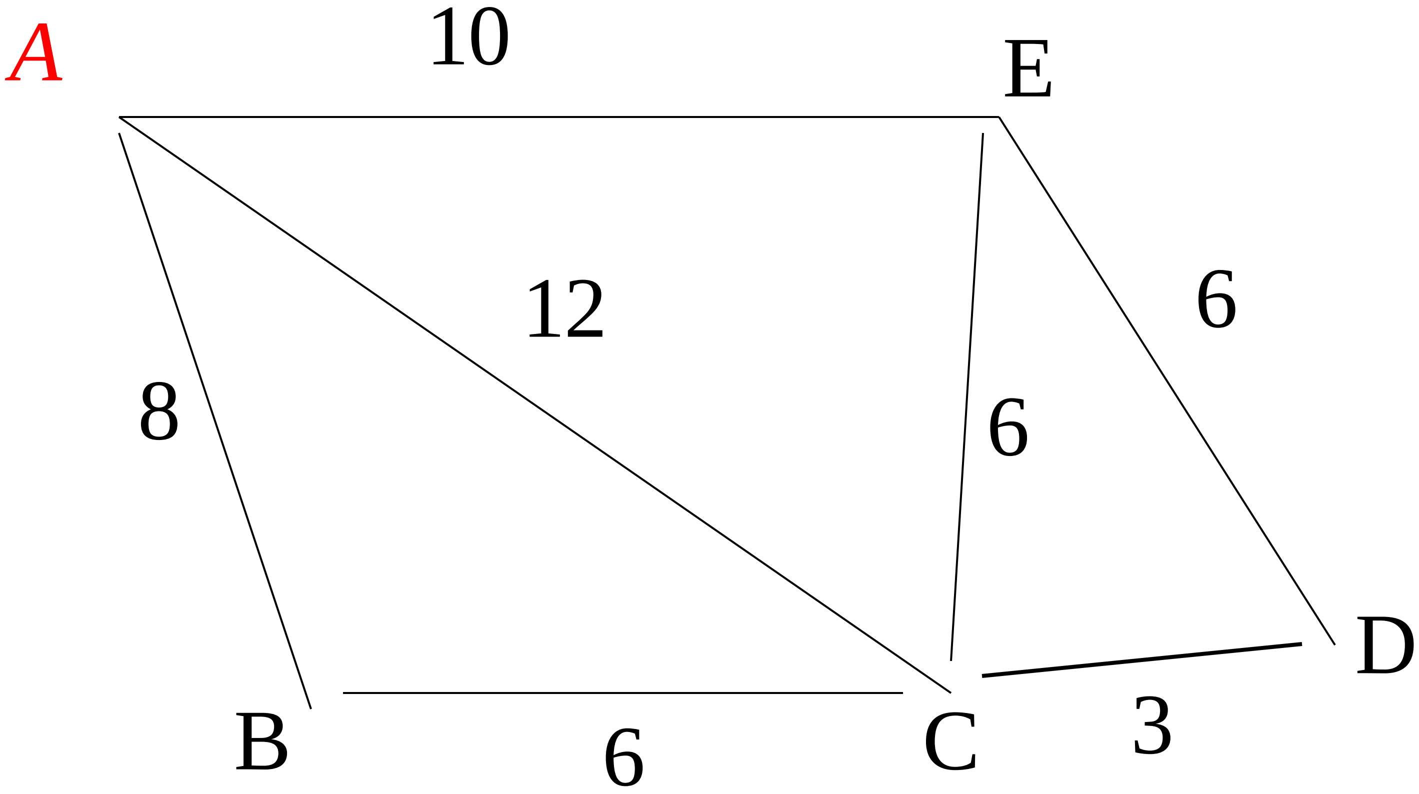
To construct an MST:

- 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

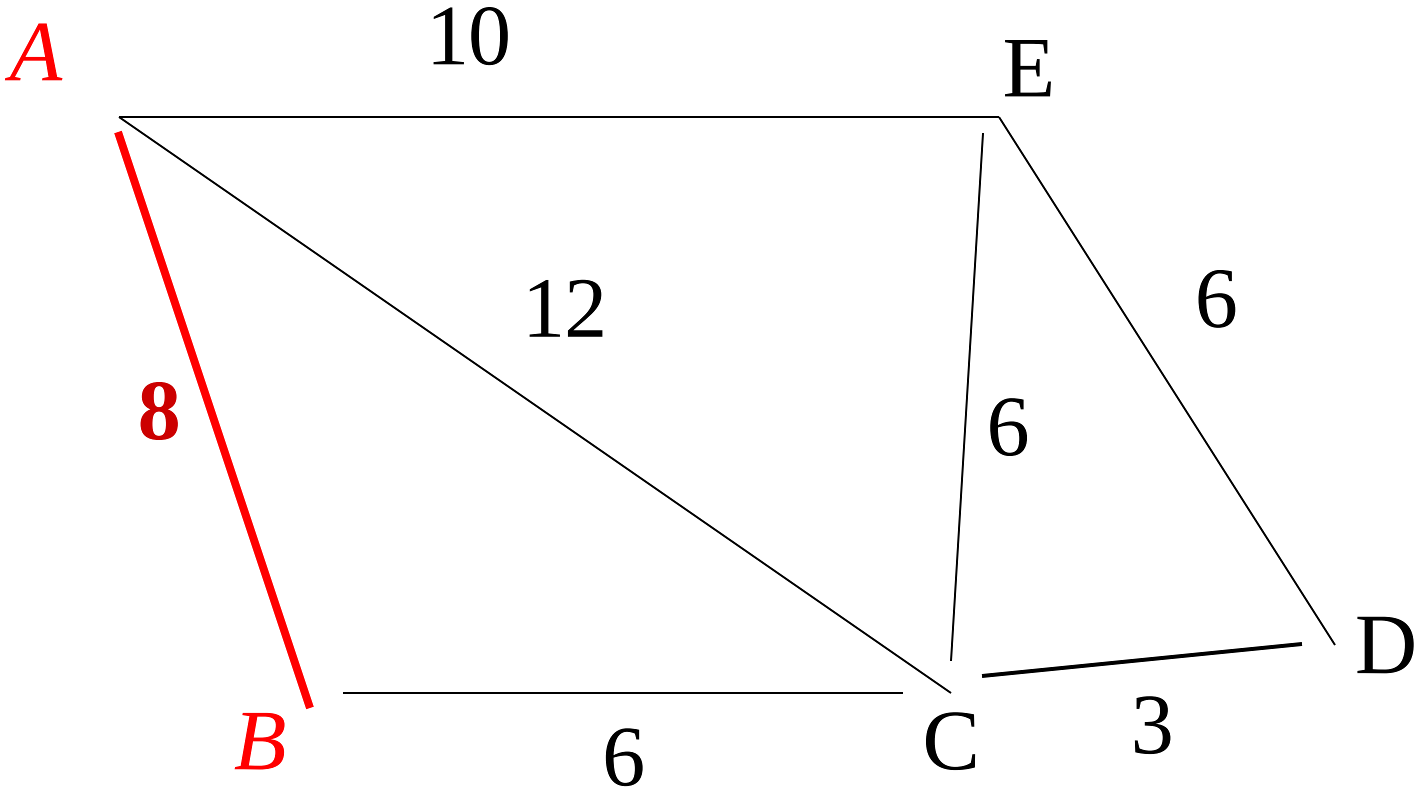
Example



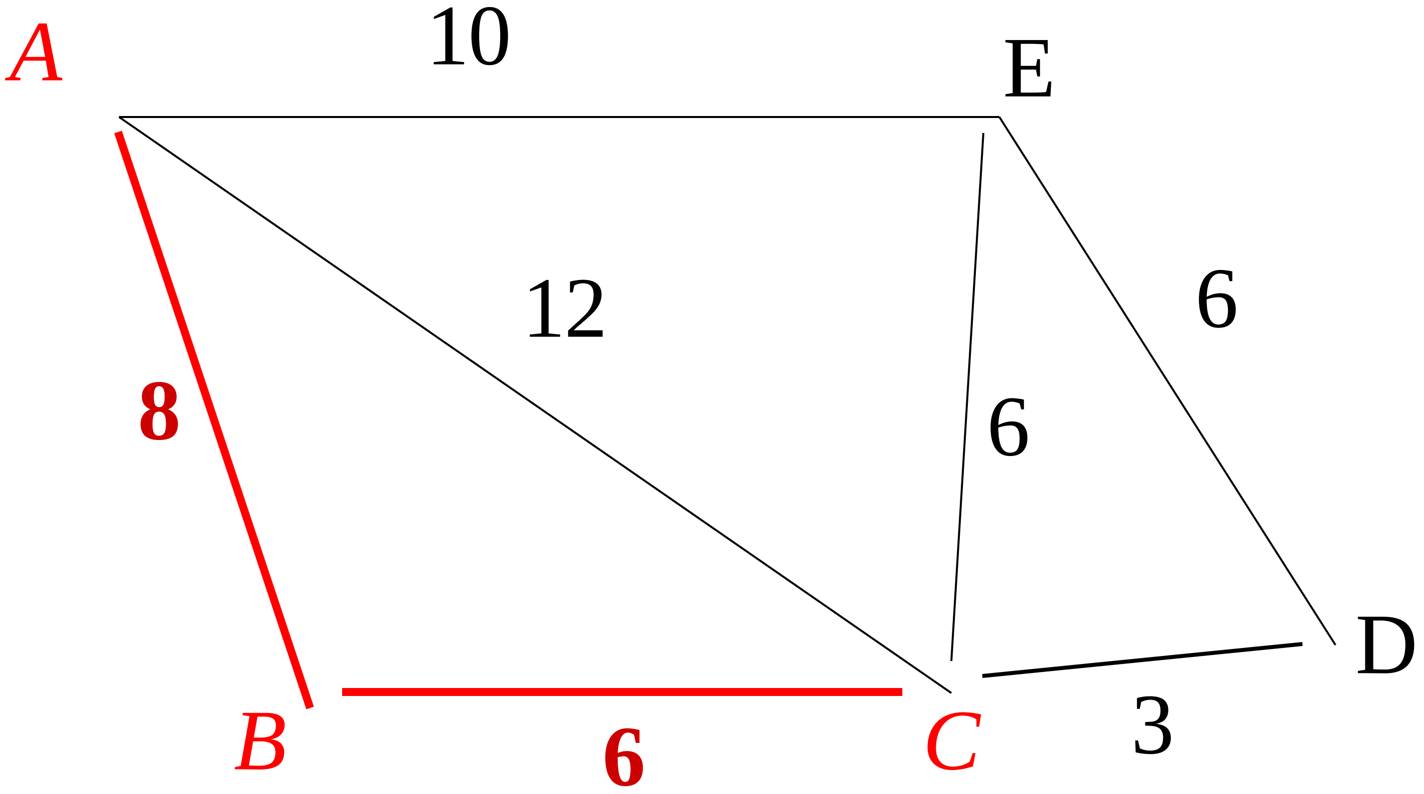
Example



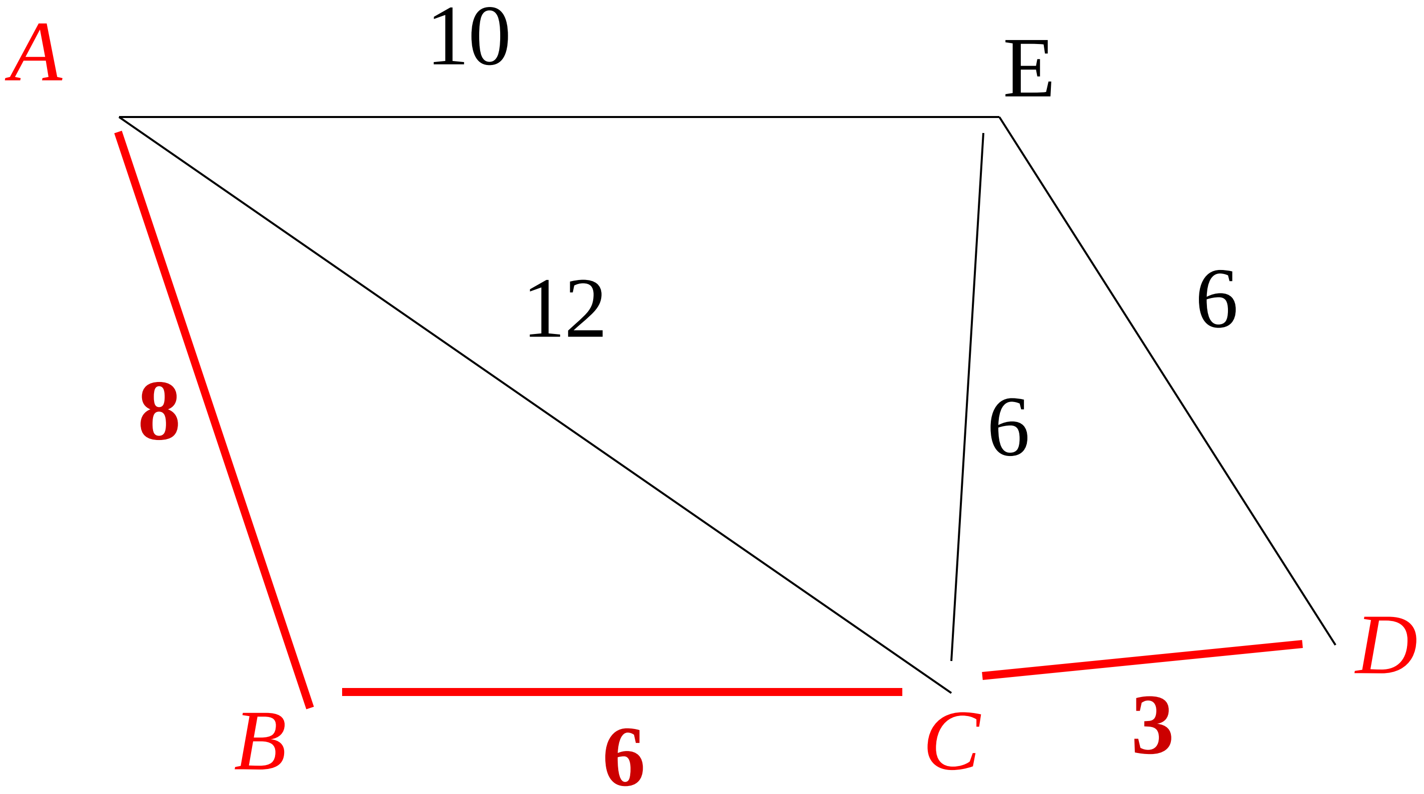
Example



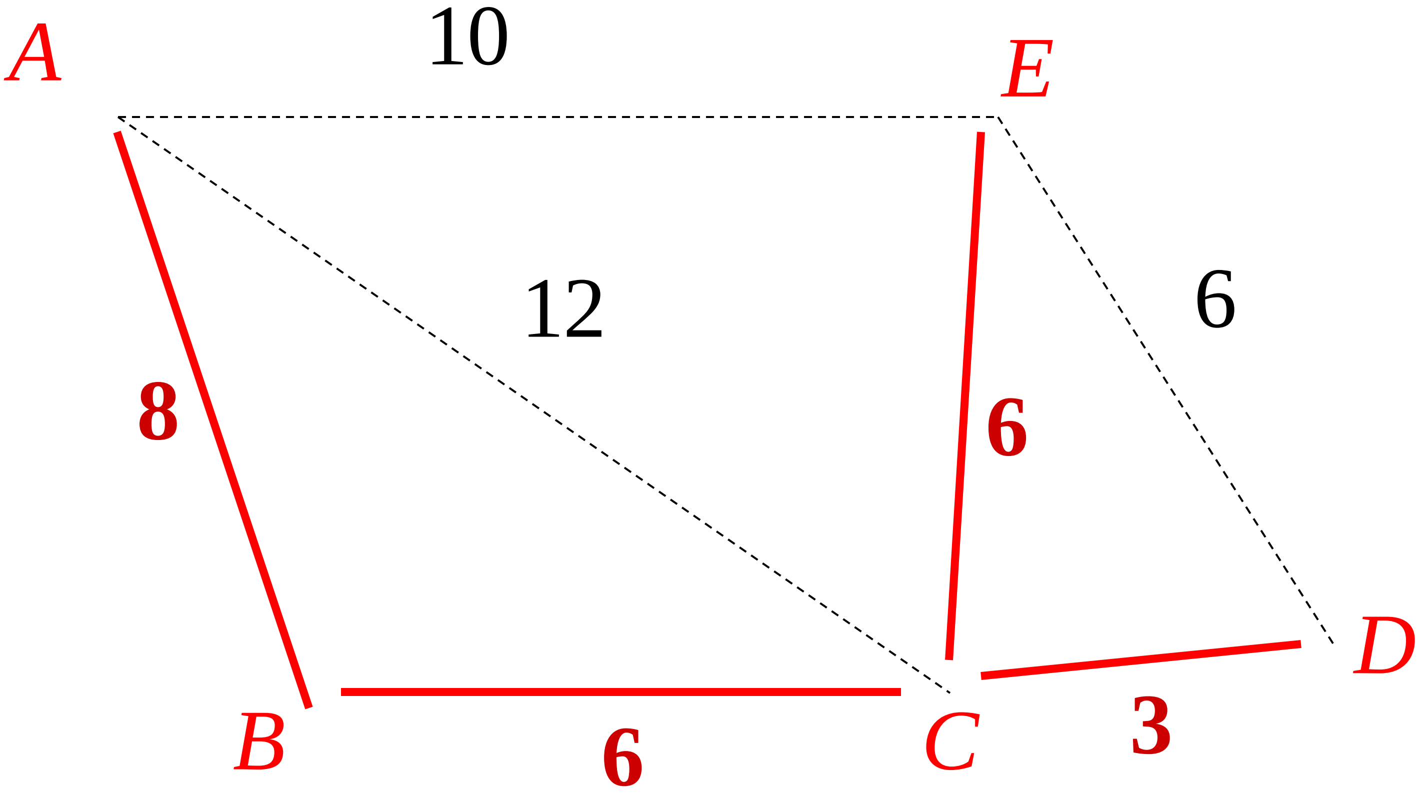
Example



Example



Example



对于一个加权连通图 G，将其顶点集合划分为两个**不相交非空子集**V_1 和 V_2。如果边 e 是所有连接 V_1 和 V_2 的边中**权重最小的一条边**，那么：**存在一个包含该边 e 的最小生成树** T。

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

对于图 G 的任意切分（partition）V_1 和 V_2，若 e 是连接 V_1 和 V_2 的最小权重边，则 **存在一个最小生成树** T，**包含**这条边 e。

假设有一个 MST T，但 **不包含**这条边 e。把边 e**加入到** T 中（这会产生一个**环**，因为原来是树，多加一条边一定成环）。这个环中一定有**另一条边** f 连接 V_1 和 V_2。由于 e 是跨切分的最小边，说明 w(e) \le w(f)。把环中的边 f 移除（也就是从 T \cup \{e\} 中去掉 f）：得到的新图仍然是一个生成树；总权重不超过原来的树。结论：你得到了另一个**权重不更大的生成树**，说明它也是 MST，且包含了边 e。

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

贪心算法在每一步选择中都采取**当前最优（最便宜、最短等）**的策略，而不从全局角度做复杂的规划。

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.

每一步只选当前最短的边，连接一个**未被包含**的顶点；
不回溯、不重考虑是否未来更优；
局部最优选择最后构建出一个全局最优解（MST）。

Dijkstra 算法的贪心策略：

每次从未访问的节点中选取**当前路径最短的点**；
这个点被访问后，其路径就是**最终最短路径**（无法再被更新）；
然后**更新其邻居节点的最短距离**，继续迭代。

Greedy Algorithms

为什么它是全局最优？

因为每个节点**一旦出队（被选中）后就不再更新**，说明从起点到它的路径已经是**最短路径**；
可严格证明：一旦你选出某个点，它的路径不可能再被某条边“ 绕路改进 ”。

- 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.

但并不是所有贪心算法都这样！有些贪心策略看起来聪明，但其实会错过全局最优；
比如在背包问题、调度问题中，一些贪心策略只能得到**近似解**（不是最优解）。

贪心算法不能得到最优解的经典例子 —— **零钱兑换问题**

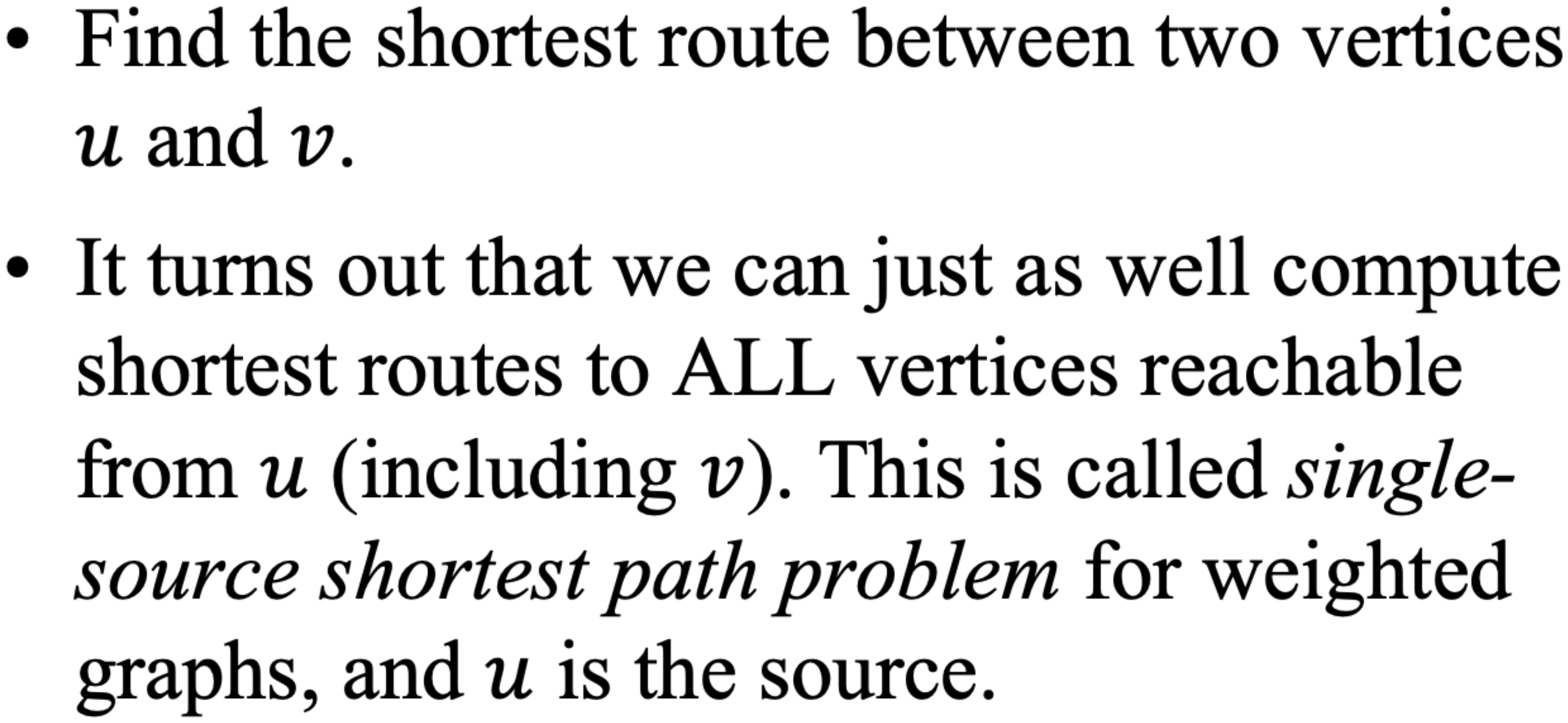
Example: non-optimal greedy algorithm

- 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.

贪心算法解法（错误的）：目标：7先选最大硬币：5 剩 2剩 2，最大能用的硬币是 1 选两个 1
总共用了：**5 + 1 + 1 3 枚硬币**

正确的最优解：选 3 + 4 恰好 7总共用了：**2 枚硬币**

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 算法是用于解决 **加权图中起点到其他所有点最短路径** 的算法，**前提是边权重不能为负数**。

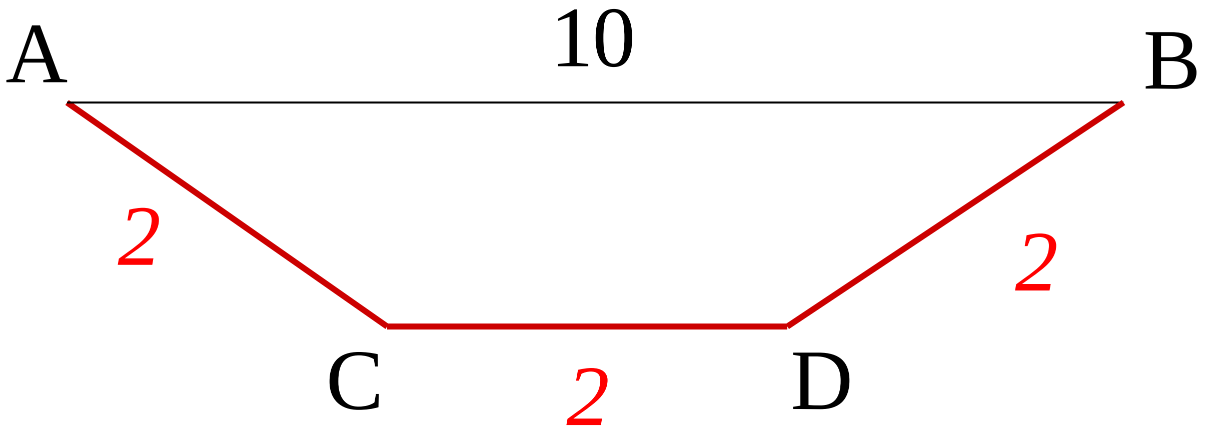
从起点开始，初始化起点到自己的距离为 0，其他为 。每次从 **未访问的节点中选出当前距离起点最近的节点**（即“ 已知最短路径最小 ”的点）。以该节点为中心，**尝试更新它的邻接点的距离**（relax）。重复步骤 2 和 3，直到所有点都被访问。这就是典型的“ 局部最优 全局最优 ”的 **贪心策略**。

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:



优先队列 PQ：存放所有待处理的节点，按照当前已知的最短距离排序（最短距离最小的节点在前）。实现方式：可以用 **最小堆**（Min Heap）。

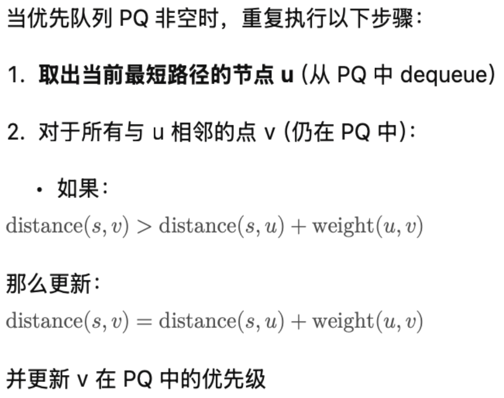
距离数组 dist[]：用来记录从起点 s 到图中每个节点的当前最短距离。初始化：所有点为 ，dist[s] = 0。

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



当优先队列 PQ 非空时，重复执行以下步骤：

1. **取出当前最短路径的节点 u** (从 PQ 中 dequeue)
2. 对于所有与 u 相邻的点 v (仍在 PQ 中)：
   - 如果：

$\text{distance}(s,v) > \text{distance}(s,u) + \text{weight}(u,v)$

那么更新：

$\text{distance}(s,v) = \text{distance}(s,u) + \text{weight}(u,v)$

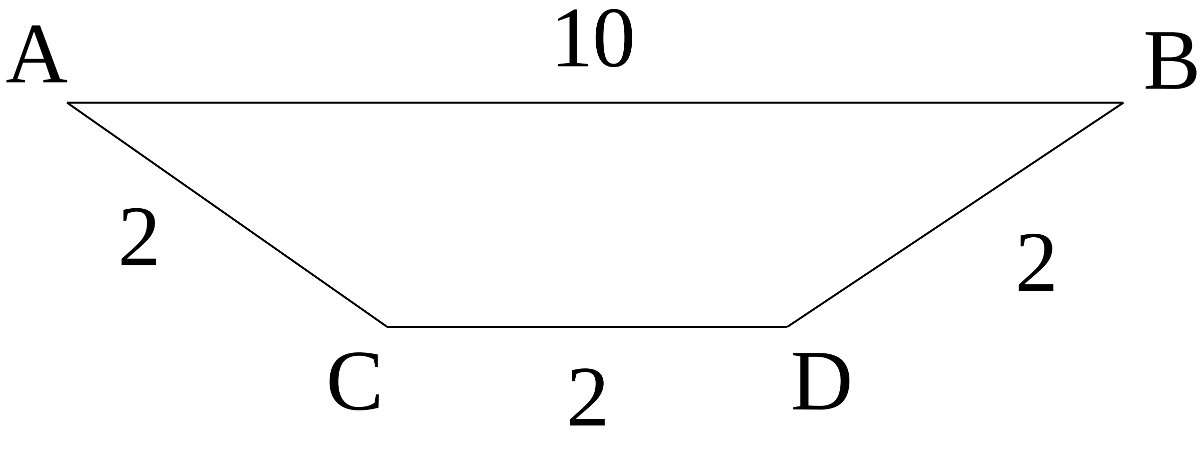
并更新 v 在 PQ 中的优先级

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

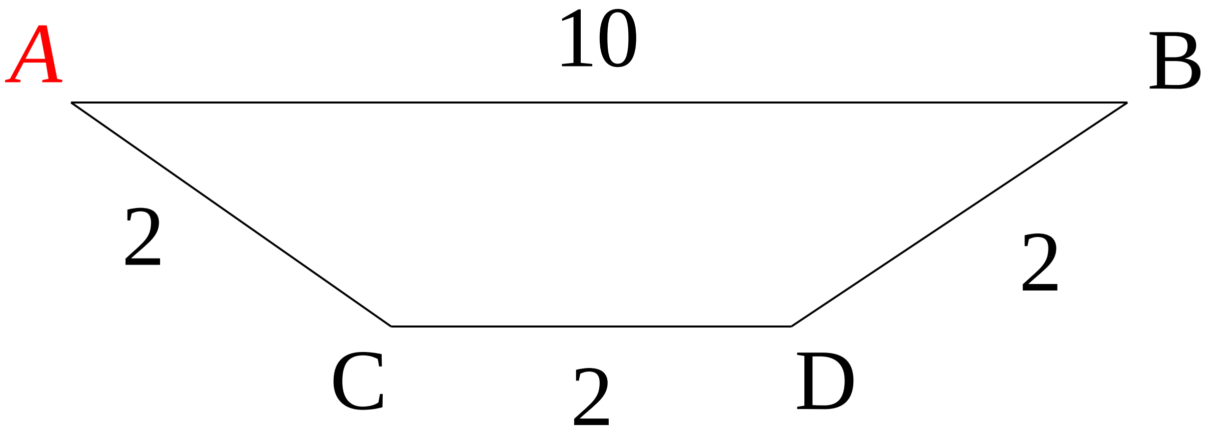
Example

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



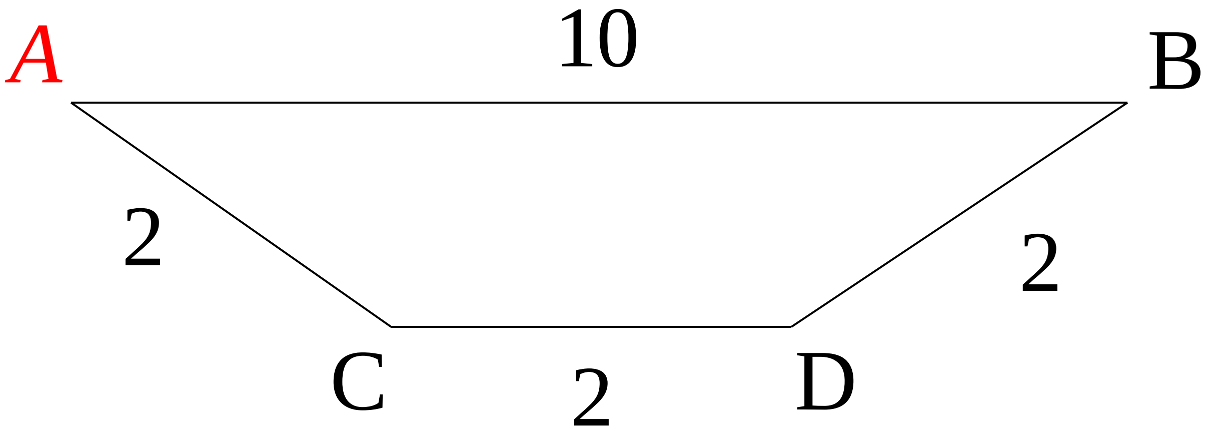
Example (dequeue A)

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



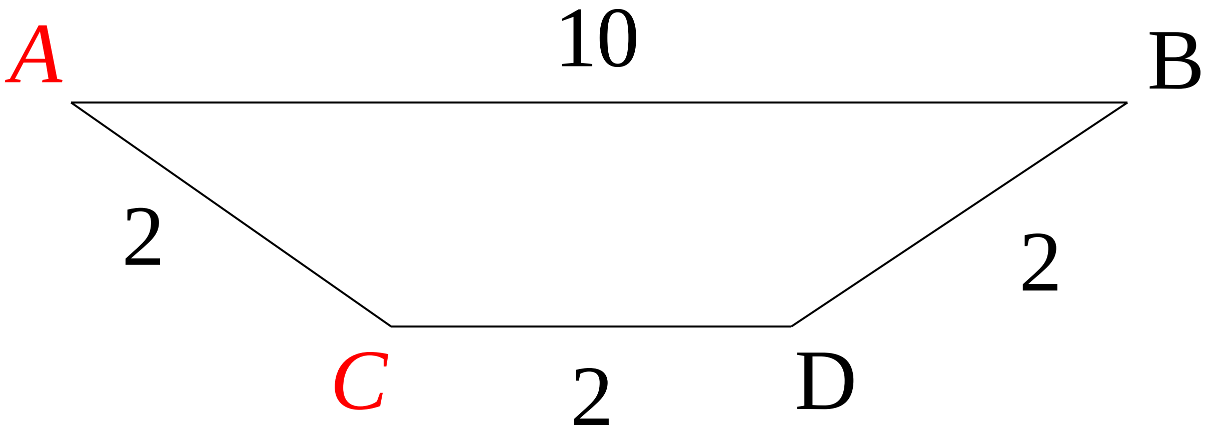
Example (recompute distances)

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



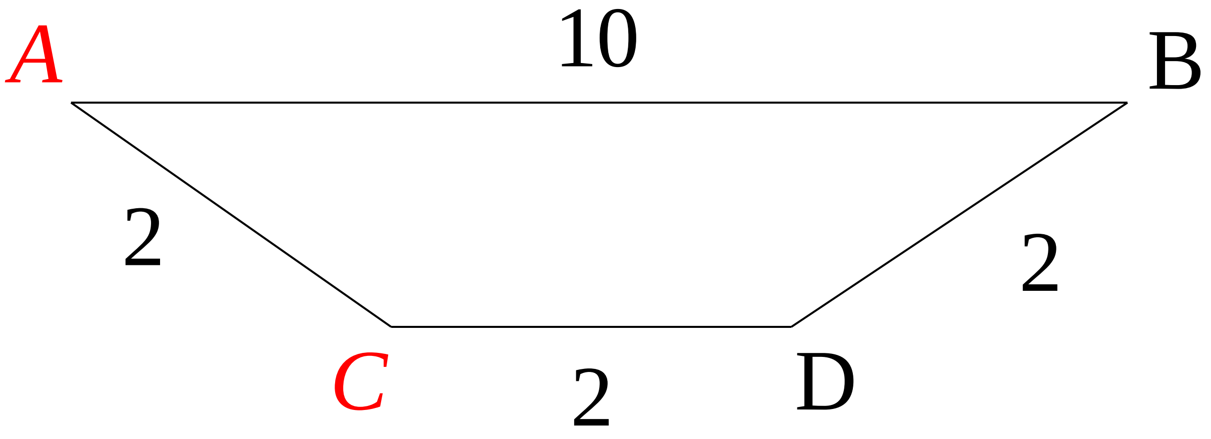
Example (dequeue C)

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



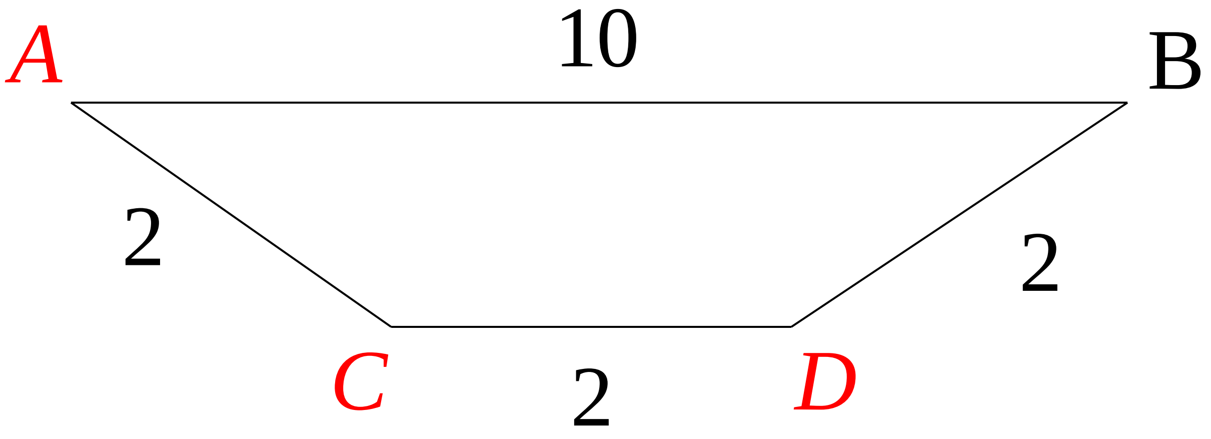
Example (recompute distances)

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



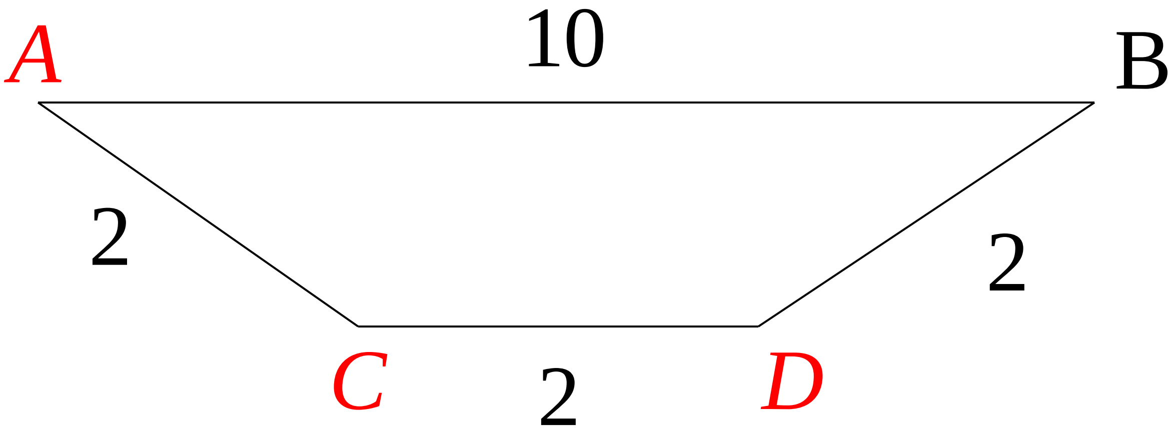
Example (dequeue D)

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



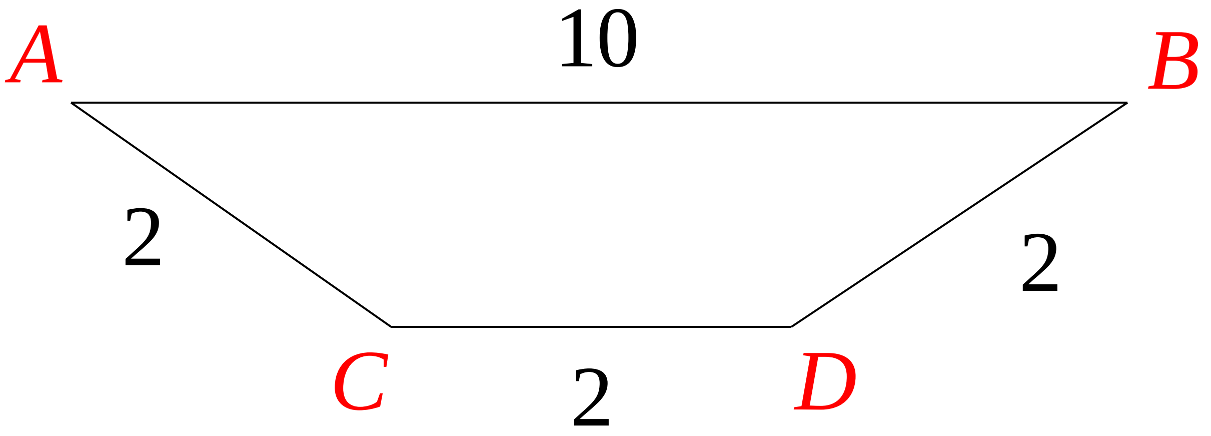
Example (recompute distances)

- 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

表示 “ 无穷大 ”，比图中任何可能的边权重都大。用于初始化起点以外所有点的最短路径值（dist），表示尚未可达。

- INF is supposed to be greater than any number
- *dist* : array holding shortest distances from source s 是一个数组，记录从起点 s 到图中每个顶点的当前最短距离。
- *PQ* : priority queue of unvisited vertices prioritised by shortest recorded distance from source 优先队列，保存 “ 尚未确定最短路径 ” 的所有顶点。以当前 dist 值为优先级，小的先出队。每次取出的是 “ 当前距离最小 ” 的顶点，进行扩展（relaxation）。
- *PQ.reorder()* reorders PQ if the values in *dist* change. 如果某个顶点的最短距离（dist[v]）在过程中更新了，优先队列需要重新排列，确保新的最短距离的顶点能在下一轮被优先处理。

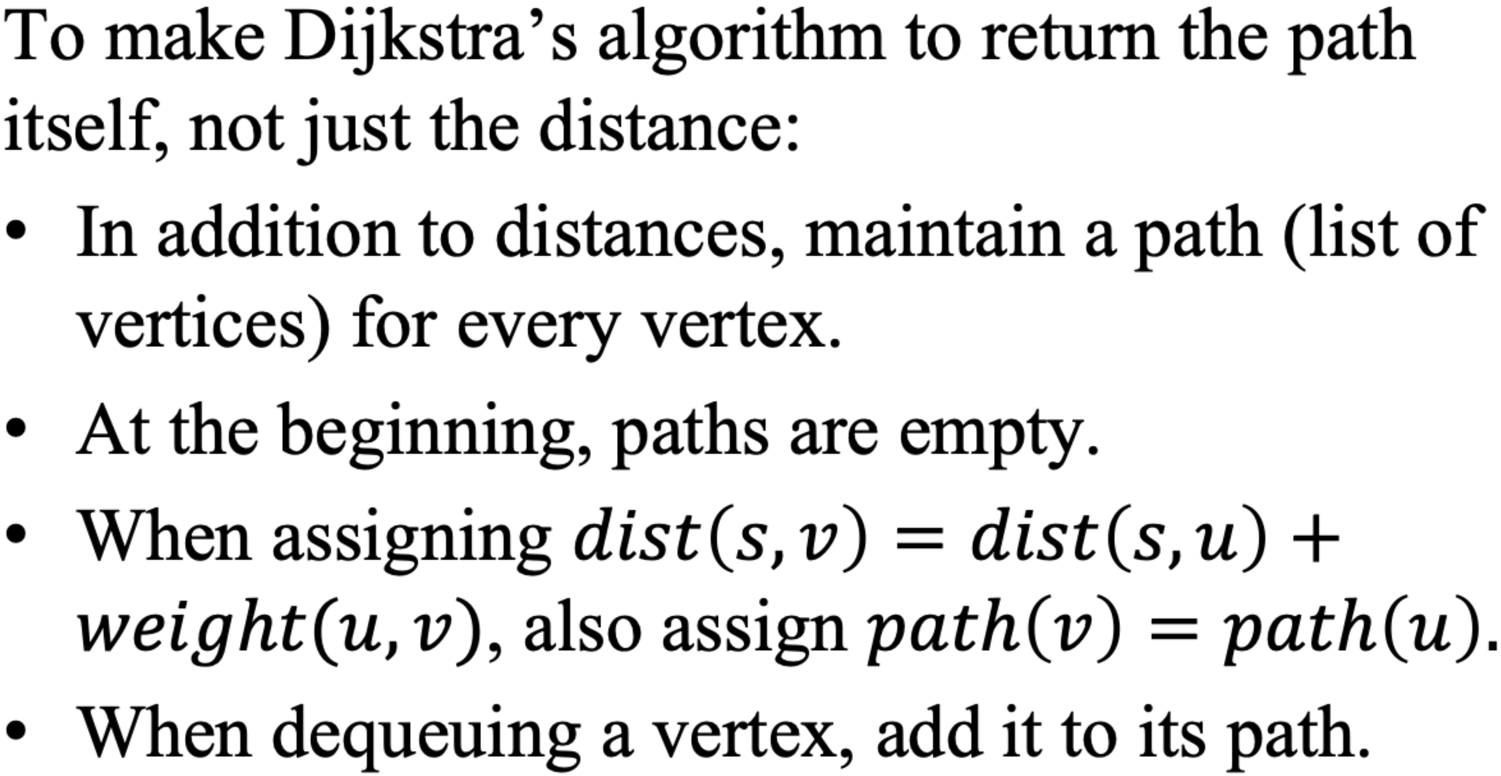
Pseudocode for Dijkstra's Algorithm

```
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



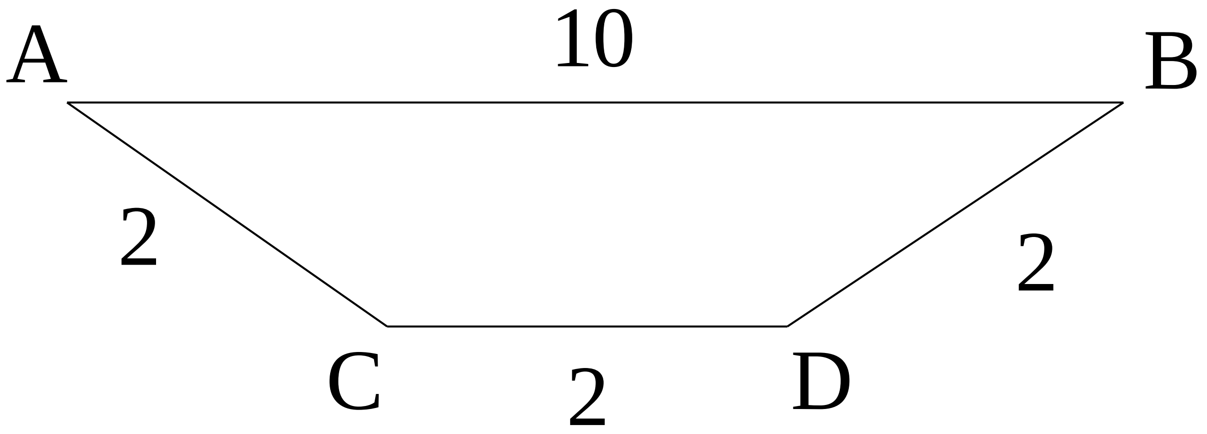
To make Dijkstra's algorithm to return the path itself, not just the distance:

- 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:

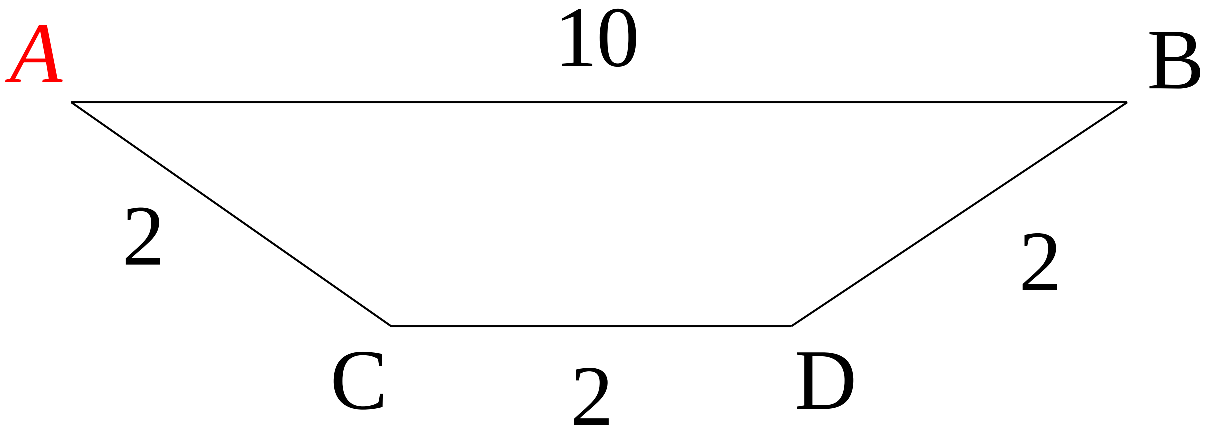
(A,0,{}), (B,INF,{}), (C,INF,{}), (D,INF,{})



Dequeue A, recompute paths

Distances and paths:

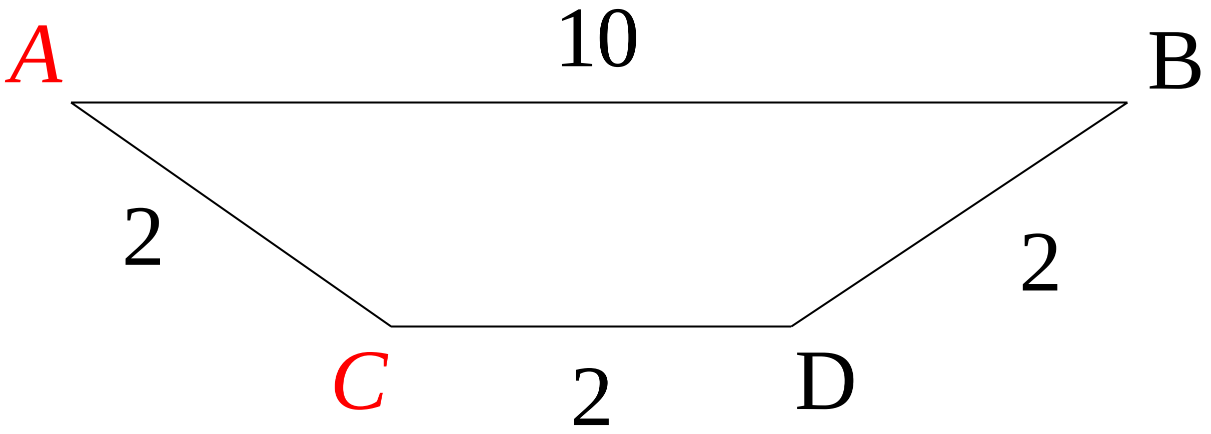
(A,0,{A}), (B,10,{A}), (C,2,{A}), (D,INF,{})



Dequeue C, recompute paths

Distances and paths:

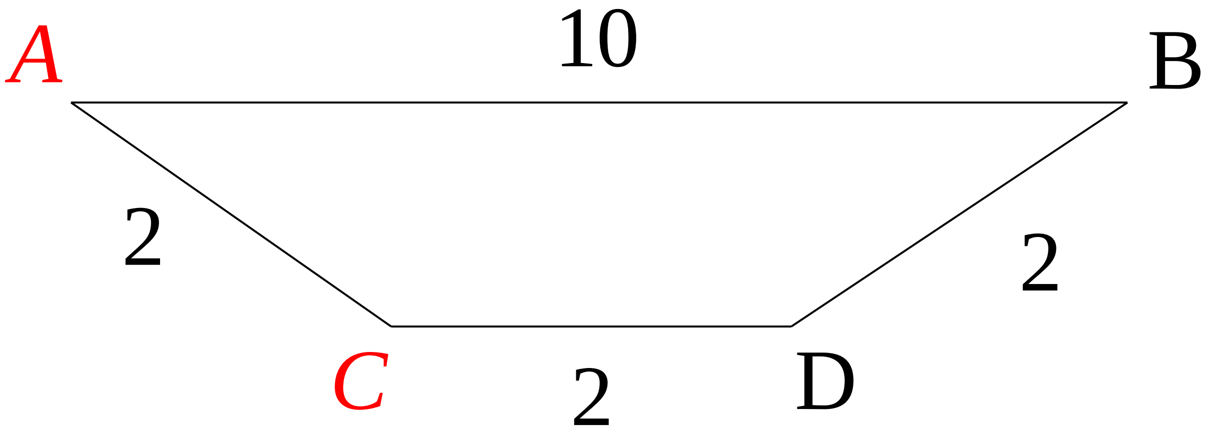
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,INF,{})



Dequeue C, recompute paths

Distances and paths:

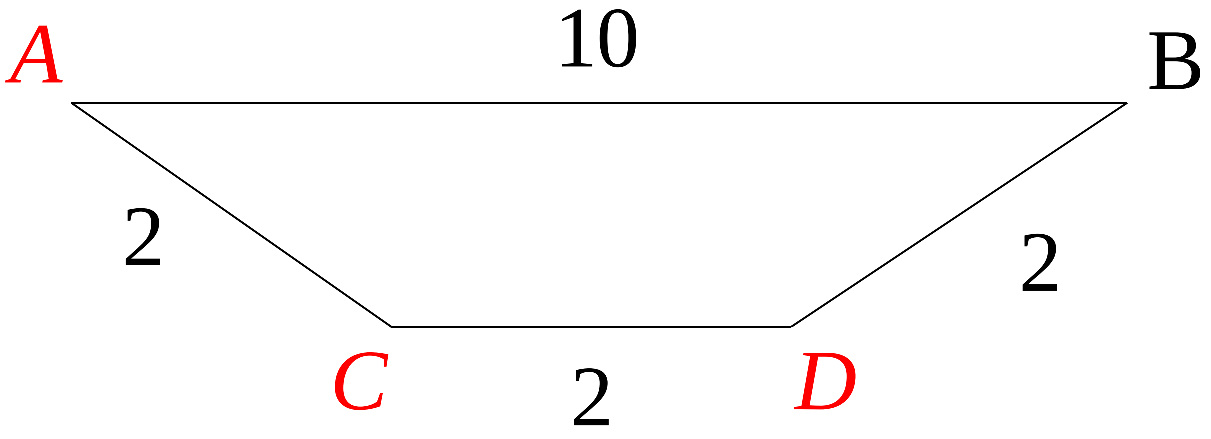
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,4,{A,C})



Dequeue D, recompute paths

Distances and paths:

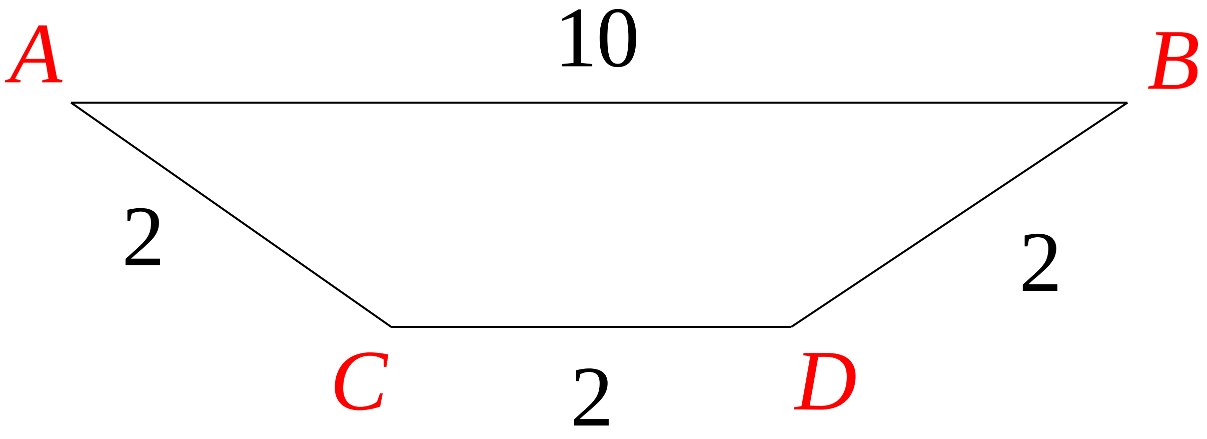
(A,0,{A}), (B,6,{A,C,D}), (C,2,{A,C}), (D,4,{A,C,D})



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?

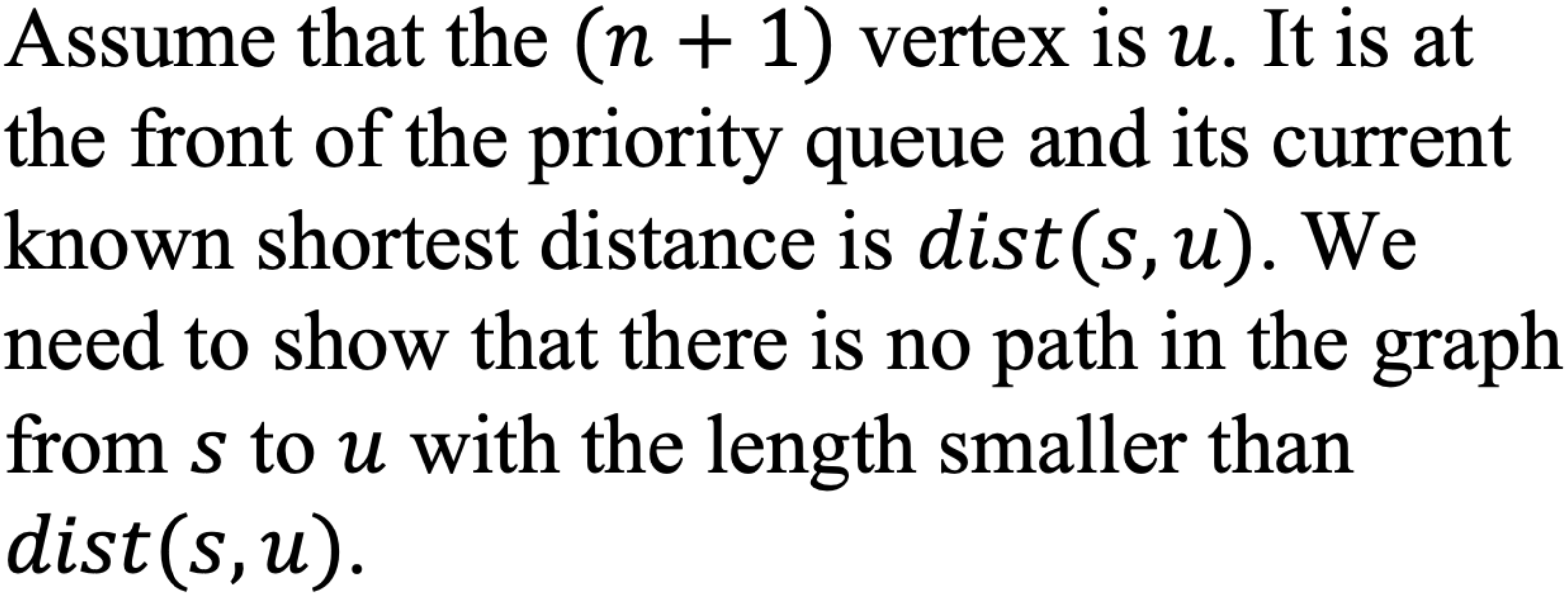
Optimality proof

基础情况（Base Case）：
起点 s 的最短距离是 0（dist(s) = 0），这是由算法初始化设置的，显然正确。

- 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.

归纳步骤（Inductive Step）：
假设已经**正确找到了前 n 个被移除顶点的最短距离**（即，它们出队时，其最短距离已经固定）；
现在我们要证明：**第 n**
+1 个出队顶点 u 的距离也是正确的。

Optimality proof

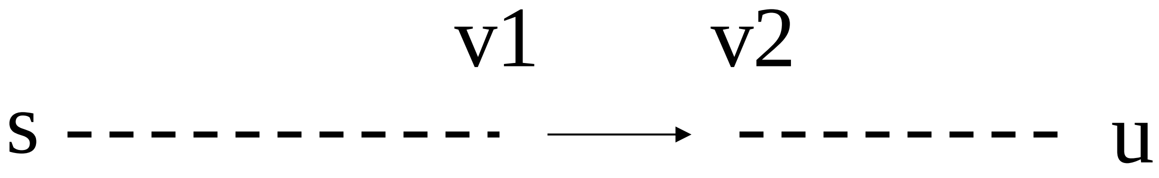


Assume that the $(n+1)$ vertex is u. It is at the front of the priority queue and its current known shortest distance is $dist(s,u)$. We need to show that there is no path in the graph from s to u with the length smaller than $dist(s,u)$.

我们假设当前是第 n+1 个顶点 u，它现在**位于优先队列最前面**，我们已经计算得出当前最短距离为：
\text{dist}(s, u)
我们需要**证明**：不存在任何从源点 s 到 u 的路径，其路径长度小于 \text{dist}(s, u)。

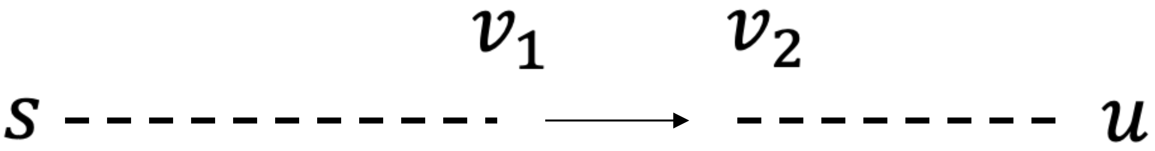
Optimality proof

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



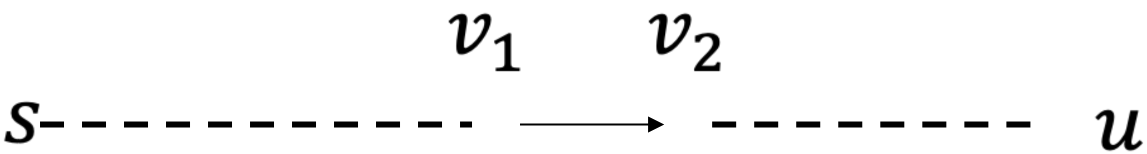
Optimality proof

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



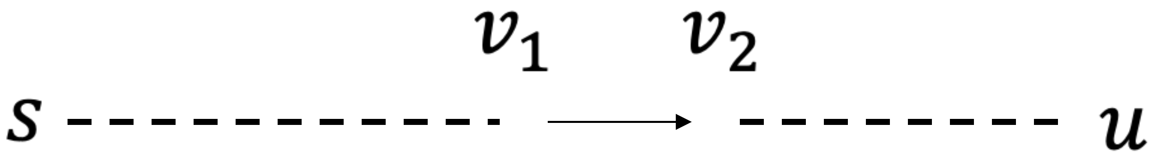
Optimality proof

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



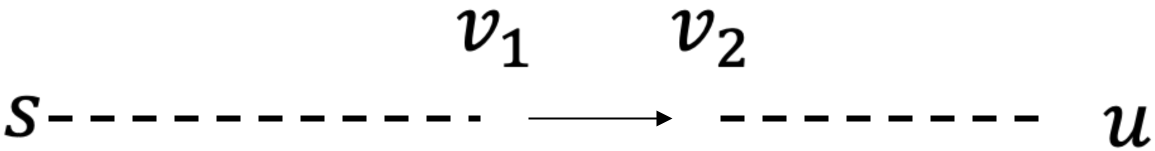
Optimality proof

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

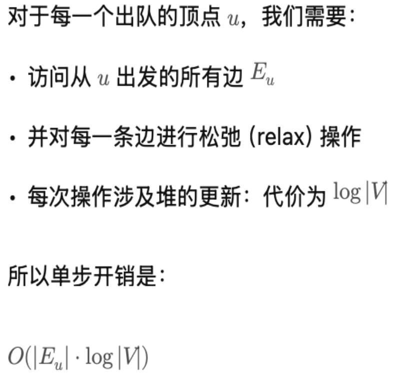


Optimality proof

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



Complexity

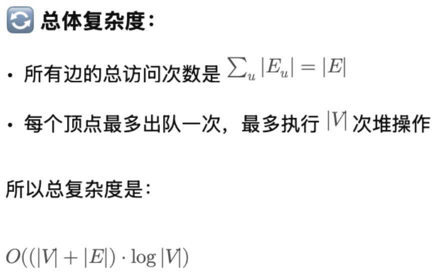


对于每一个出队的顶点 u，我们需要：

- 访问从 u 出发的所有边 E_u
- 并对每一条边进行松弛（relax）操作
- 每次操作涉及堆的更新：代价为 $\log |V|$

所以单步开销是：

$$O(|E_u| \cdot \log |V|)$$



🔄 **总体复杂度：**

- 所有边的总访问次数是 $\sum_u |E_u| = |E|$
- 每个顶点最多出队一次，最多执行 $|V|$ 次堆操作

所以总复杂度是：

$$O((|V| + |E|) \cdot \log |V|)$$

- 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.