DET102 Data Structures and Algorithms

Lecture 9: Graph algorithms

Outline

- BFS and DFS in STL
- Topological sorting
- Minimum spanning tree
- Shortest path

BFS using STL

- *In BFS, we start with a node.*
  1. *Create a queue and enqueue source (starting vertex) into it.*
  2. *Mark source as visited.*

  While queue is not empty, do following
    1. *Dequeue a vertex from queue. Let this be f.*
    2. *Print f*
    3. *Enqueue all not yet visited adjacent vertices of f and mark them visited.*

[BFS using STL for competitive coding - GeeksforGeeks]

```
// A Quick implementation of BFS using
// vectors and queue
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

vector<bool> v;
vector<vector<int> > g;

void edge(int a, int b)
{
	g[a].pb(b);

	// for undirected graph add this line
	// g[b].pb(a);
}
```

```
void bfs(int u)
{
    queue<int> q;
    q.push(u);
    v[u] = true;

    while (!q.empty()) {
        int f = q.front();
        q.pop();
        cout << f << " ";
        // Enqueue all adjacent of f and mark them visited
        for (auto i = g[f].begin(); i != g[f].end(); i++) {
            if (!v[*i]) {
                q.push(*i);
                v[*i] = true;
            }
        }
    }
}
```

- *In BFS, we start with a node.*
  1. *Create a queue and enqueue source (starting vertex) into it.*
  2. *Mark source as visited.*

  While queue is not empty, do following
  1. *Dequeue a vertex from queue. Let this be f.*
  2. *Print f*
  3. *Enqueue all not yet visited adjacent vertices of f and mark them visited.*

```
int main()
{
    int n, e;
    cin >> n >> e;

    v.assign(n, false);
    g.assign(n, vector<int>());

    int a, b;
    for (int i = 0; i < e; i++) {
        cin >> a >> b;
        edge(a, b);
    }

    for (int i = 0; i < n; i++) {
        if (!v[i])
            bfs(i);
    }

    return 0;
}
```

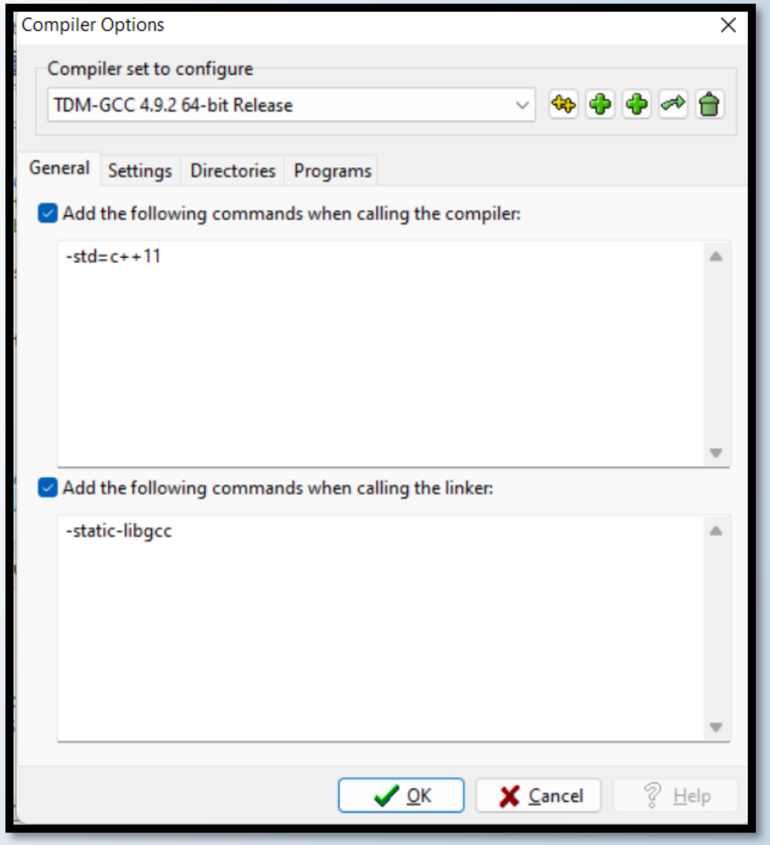
- *In BFS, we start with a node.*
  1. *Create a queue and enqueue source (starting vertex) into it.*
  2. *Mark source as visited.*

  While queue is not empty, do following
  1. *Dequeue a vertex from queue. Let this be f.*
  2. *Print f*
  3. *Enqueue all not yet visited adjacent vertices of f and mark them visited.*

```
Input:
8 10
0 1
0 2
0 3
0 4
1 5
2 5
3 6
4 6
5 7
6 7
Output:
0 1 2 3 4 5 6 7
```

If you cannot compile using Dev c++, Tools-> compiler options->General

Tick the option 'Add the following commands when calling compiler' and add '-std=c++11' in the empty frame.


Compiler Options

Compiler set to configure: TDM-GCC 4.9.2 64-bit Release

General | Settings | Directories | Programs

[✓] Add the following commands when calling the compiler:

```
-std=c++11
```

[✓] Add the following commands when calling the linker:

```
-static-libgcc
```

DFS using STL

- *In DFS, we start with a node.*
  1. *Create a stack and push source (starting vertex) into it.*
  2. *Mark source as visited.*

  While stack is not empty, do following
    1. *Pop a vertex from queue. Let this be f.*
    2. *Print f*
    3. *Push all not yet visited adjacent vertices of f and mark them visited.*

```
void dfs(int u)
{
	stack<int> s;

	s.push(u);
	v[u] = true;

	while (!s.empty()) {

		int f = s.top();
		s.pop();

		cout << f << " ";

		// Push all adjacent of f and mark them visited
		for (auto i = g[f].begin(); i != g[f].end(); i++) {
			if (!v[*i]) {
				s.push(*i);
				v[*i] = true;
			}
		}
	}
}
```

- *In DFS, we start with a node.*
  1. *Create a stack and push source (starting vertex) into it.*
  2. *Mark source as visited.*

  While stack is not empty, do following
    1. *Pop a vertex from queue. Let this be f.*
    2. *Print f*
    3. *Push all not yet visited adjacent vertices of f and mark them visited.*

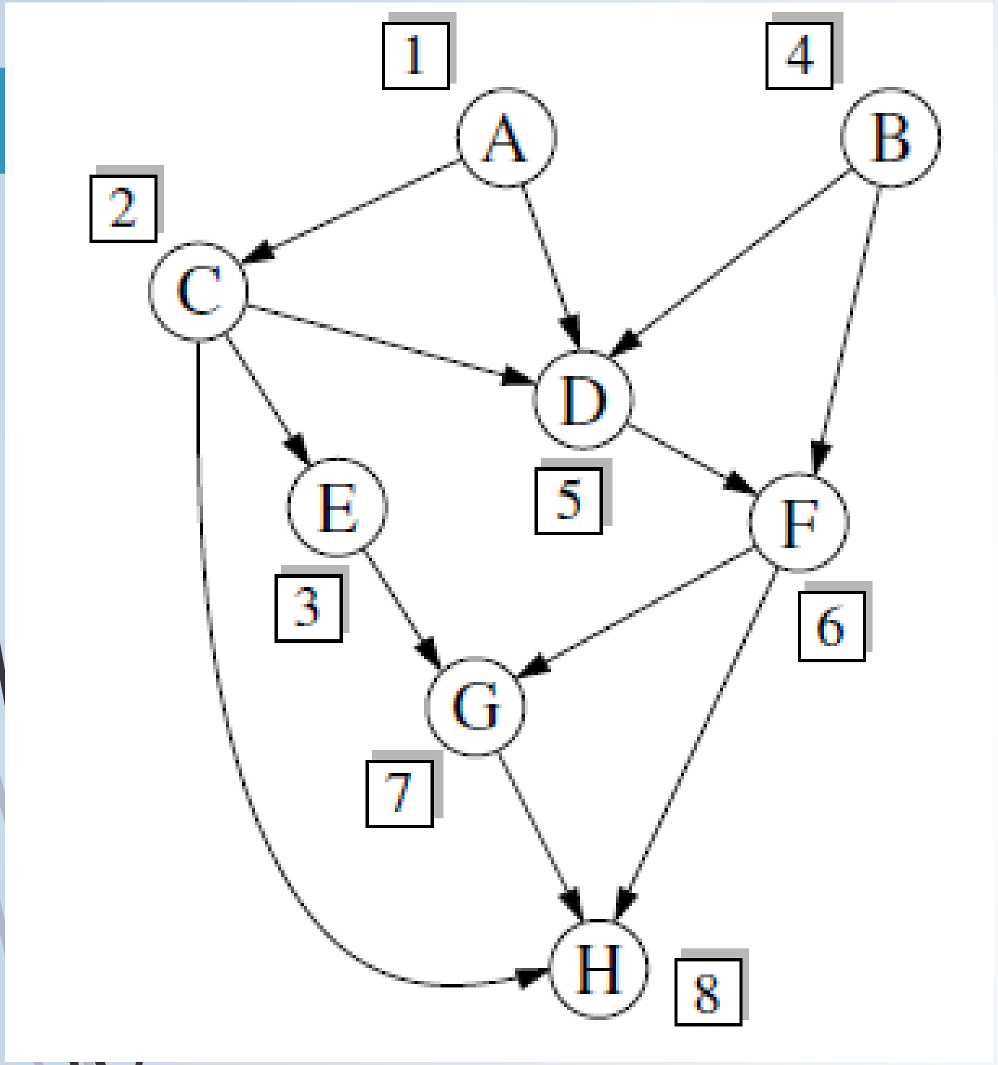
Directed Acyclic Graphs

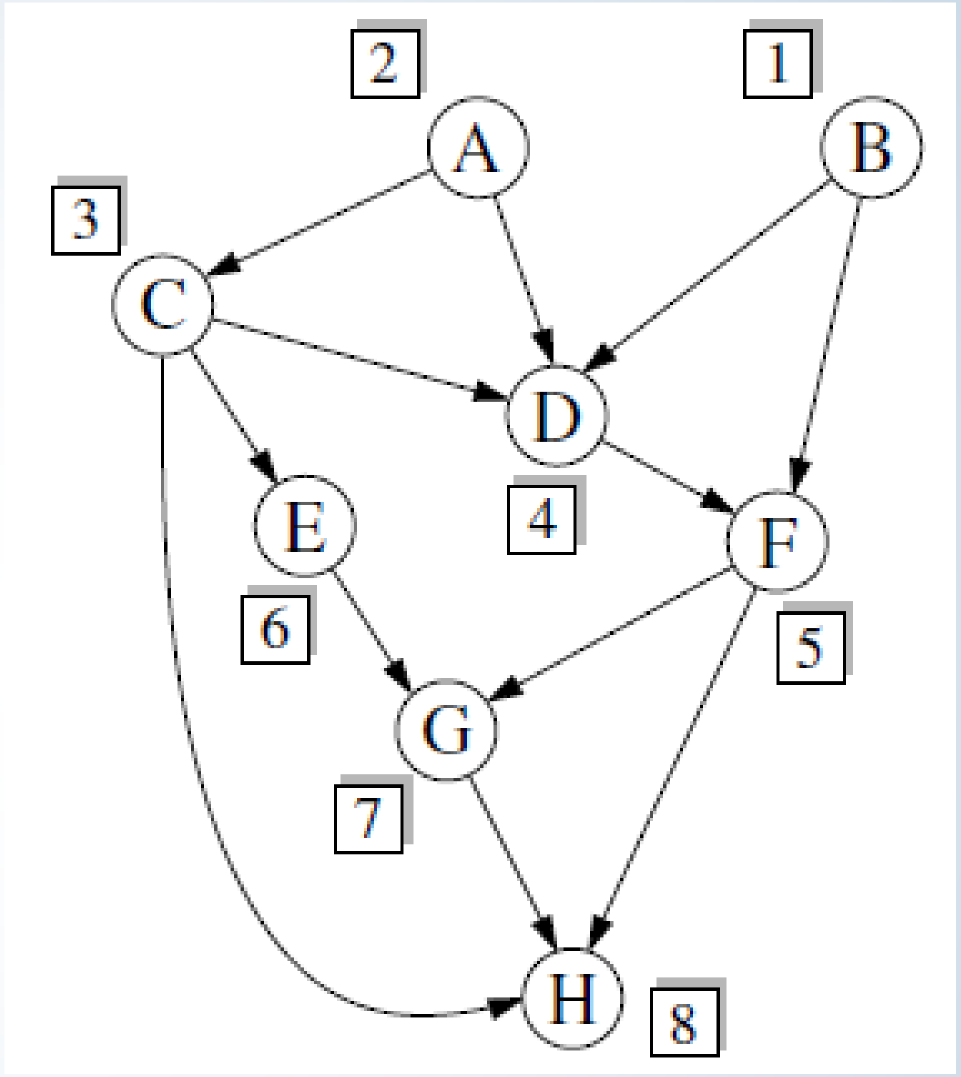
DAG

- Directed graphs without directed cycles are encountered in many applications. Such a directed graph is often referred to as a ***directed acyclic graph***, or ***DAG***, for short.
- Applications of DAG:
  - Prerequisites between courses of a degree program.
  - Scheduling constraints between the tasks of a project.

Topological Ordering

- Let G be a directed graph with *n* vertices. A ***topological ordering*** of G is an ordering $v_1, \ldots, v_n$ of the vertices of G such that for every edge (v_i, v_j) of G, it is the case that $i < j$.
- A topological ordering is an ordering such that any directed path in G traverses vertices in increasing order.
- Note that a directed graph may have more than one topological ordering.





A graph has a topological ordering if and only if it is acyclic

Topological Sorting

$$\begin{tikzcd}
A \arrow[rr, bend left] \arrow[rrr, bend right] & B \arrow[r] & C \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & D & E \arrow[r] & F
\end{tikzcd}$$

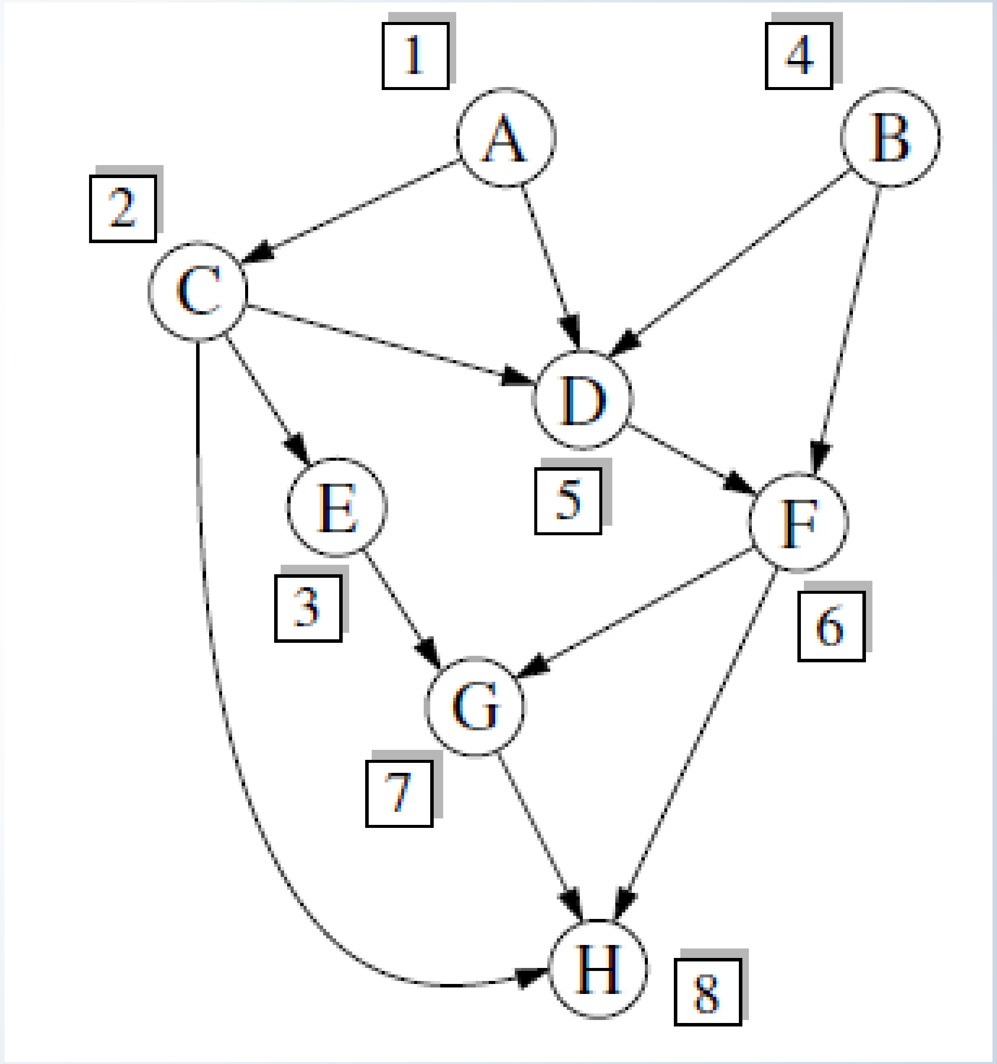
In a DAG, there must exist a vertex with indegree=0

Algorithm 1:
Find the vertex v with indegree=0
Output v
Remove v and all its incident edges.

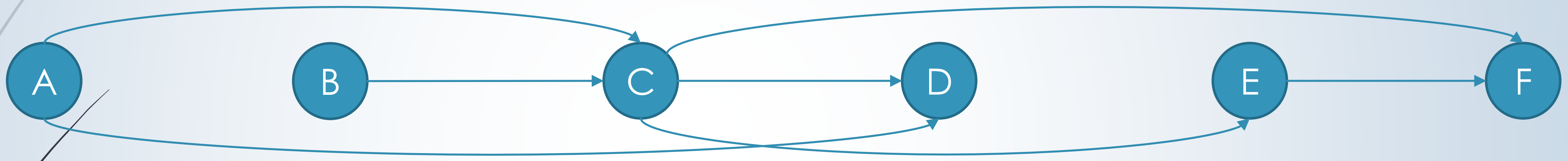
Topological sorting

- computes a topological ordering of a directed graph.

- Step1: Sort all the vertices according to degree.
- Step2: Always choose the vertex with indegree=0, and remove all its incident edges.



Topological Sorting

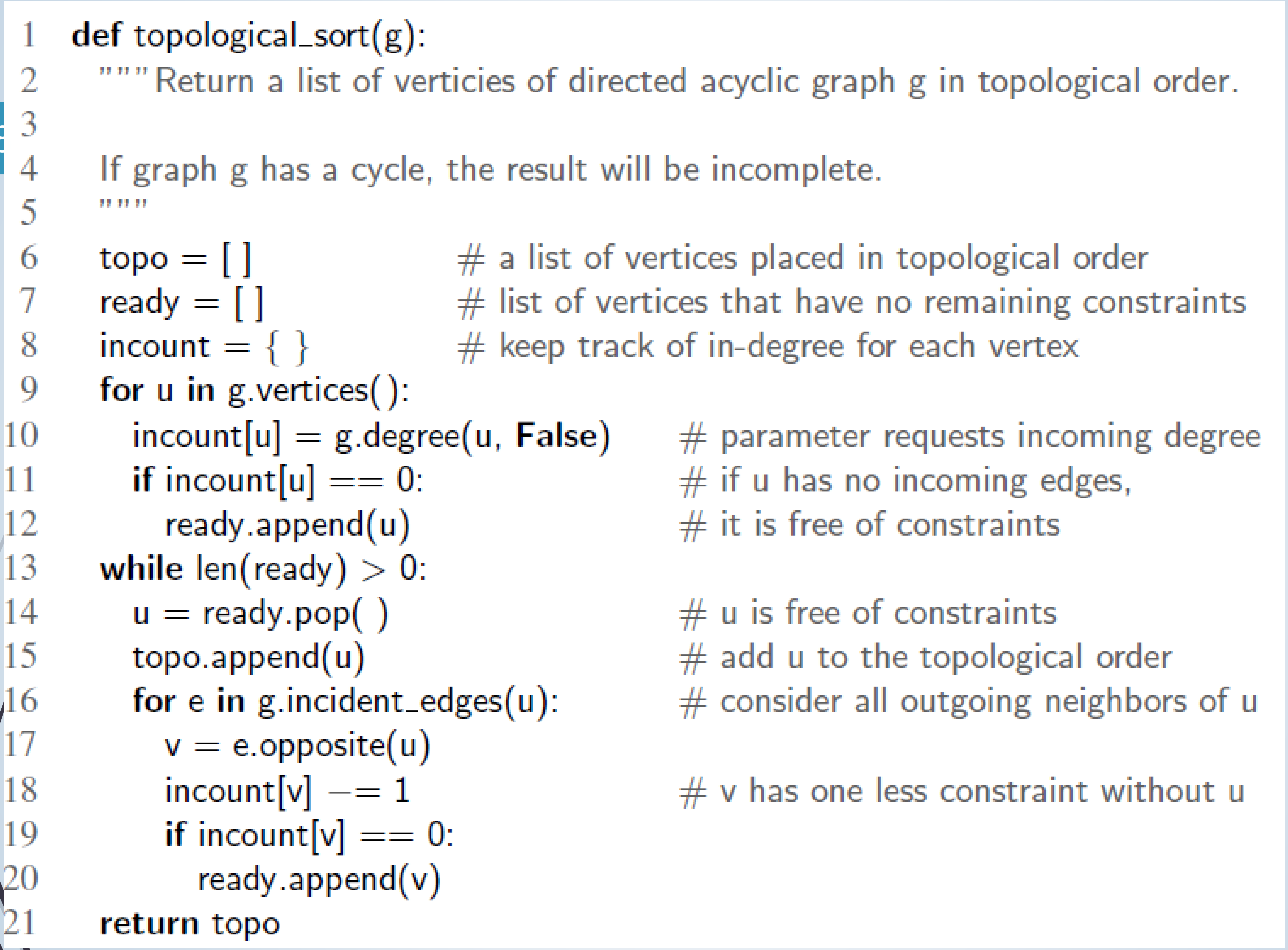


Algorithm 2:
Run DFS from any vertex.
    Whenever there is a backtrack, push the vertex to the stack.
Pop all the vertices from the stack

Input:
```
6 7
0 2
0 3
1 2
2 3
2 4
2 5
4 5
```

Topological Sorting

```
DFS (v) {                          //Starts with vertex v:
    visited[v] = true;
    for each vertex w adjacent to v {
      if (! visited[w])
          DFS (w);                         //Recursion
    }
    Push v to the stack
}
Pop all the items from the stack
```



```
def topological_sort(g):
    """Return a list of verticies of directed acyclic graph g in topological order.

    If graph g has a cycle, the result will be incomplete.
    """
    topo = [ ]                  # a list of vertices placed in topological order
    ready = [ ]                 # list of vertices that have no remaining constraints
    incount = { }               # keep track of in-degree for each vertex
    for u in g.vertices( ):
        incount[u] = g.degree(u, False)   # parameter requests incoming degree
        if incount[u] == 0:               # if u has no incoming edges,
            ready.append(u)               # it is free of constraints
    while len(ready) > 0:
        u = ready.pop( )                  # u is free of constraints
        topo.append(u)                    # add u to the topological order
        for e in g.incident_edges(u):     # consider all outgoing neighbors of u
            v = e.opposite(u)
            incount[v] -= 1               # v has one less constraint without u
            if incount[v] == 0:
                ready.append(v)
    return topo
```

Minimum Spanning Tree

Motivation Example 1

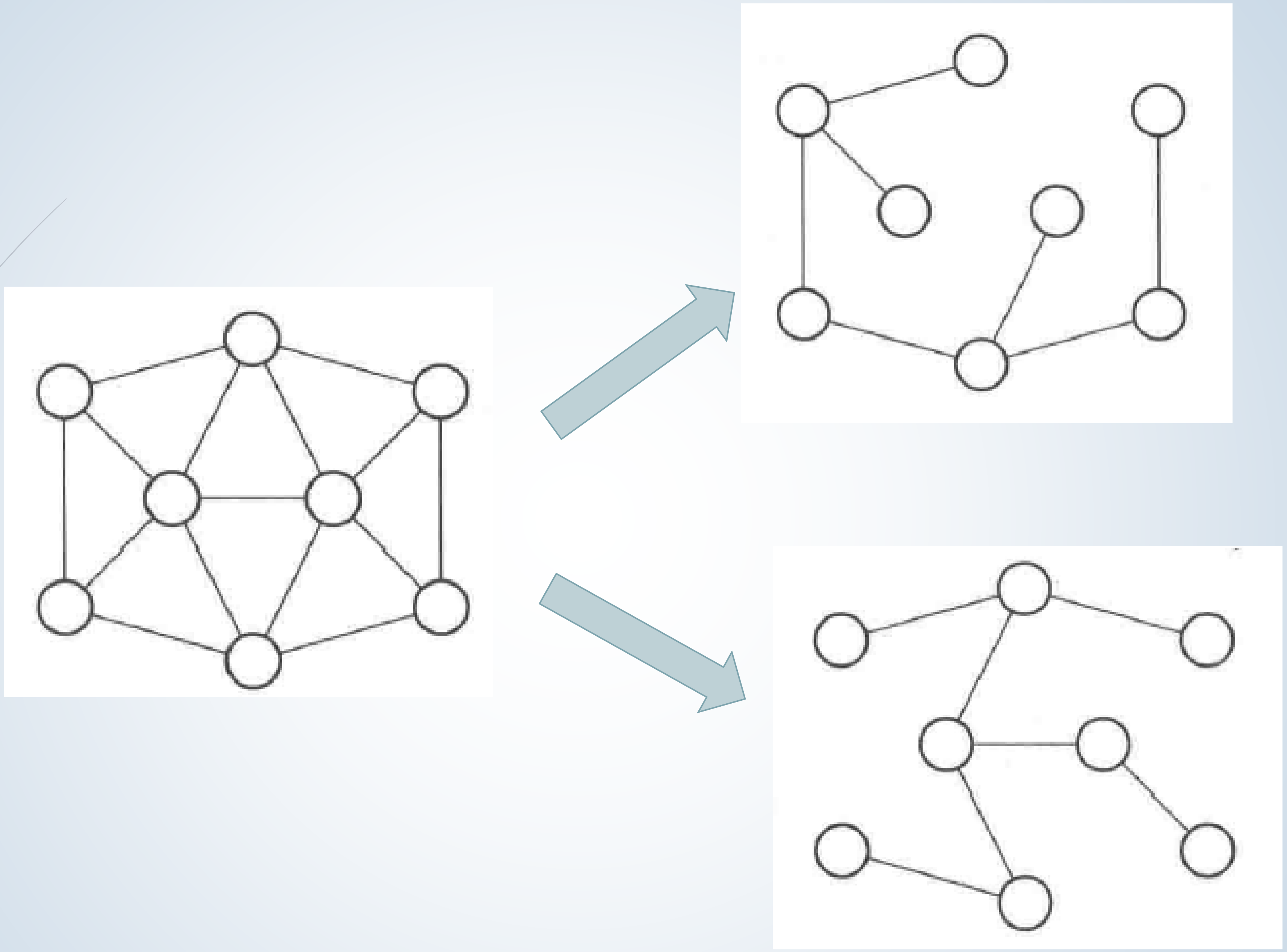
- One example would be a telecommunications company trying to lay cable in a new neighborhood.
- If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths.
- Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality.

Motivation Example 2

- Suppose we wish to connect all the computers in a new office building using the least amount of cable.
- We can model this problem using an undirected, weighted graph G whose vertices represent the computers, and whose edges represent all the possible pairs (*u*,*v*) of computers, where the weight *w*(*u*,*v*) of edge (*u*,*v*) is equal to the amount of cable needed to connect computer *u* to computer *v*.
- Rather than computing a shortest-path tree from some particular vertex *v*, we are interested instead in finding a tree *T* that contains all the vertices of G and has the minimum total weight over all such trees.

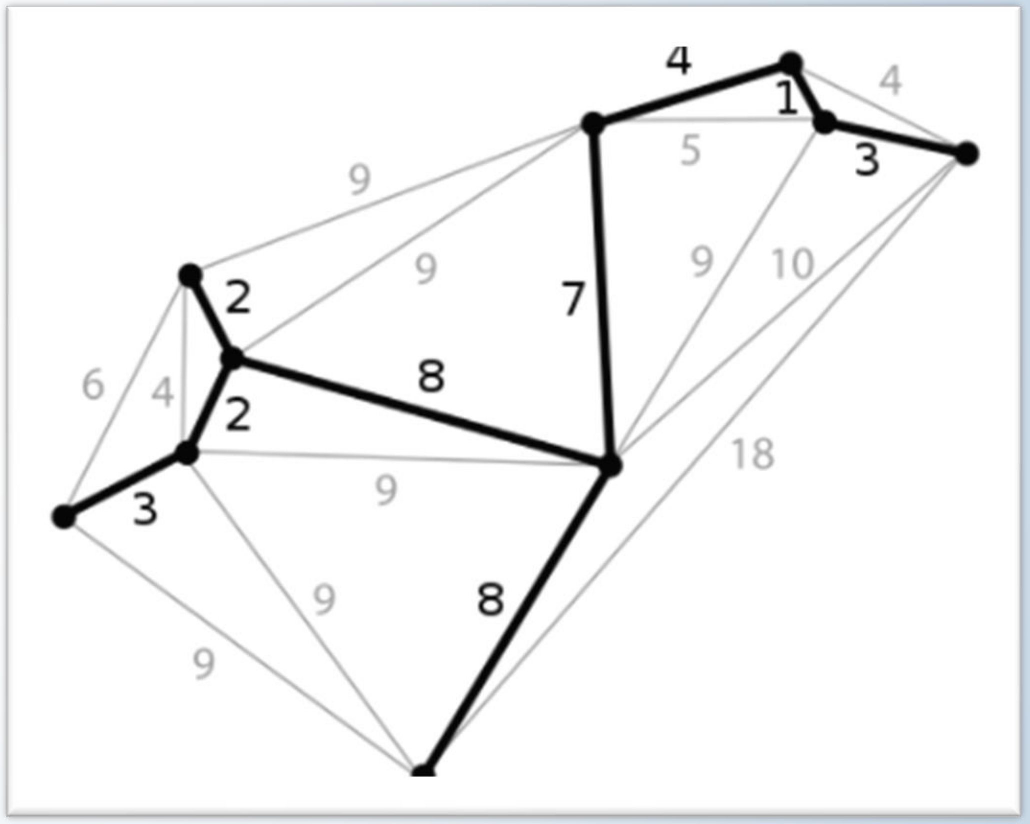
Spanning Tree

- In the mathematical field of graph theory, a **spanning tree** *T* of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G, with a minimum possible number of edges**,**
- A graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree

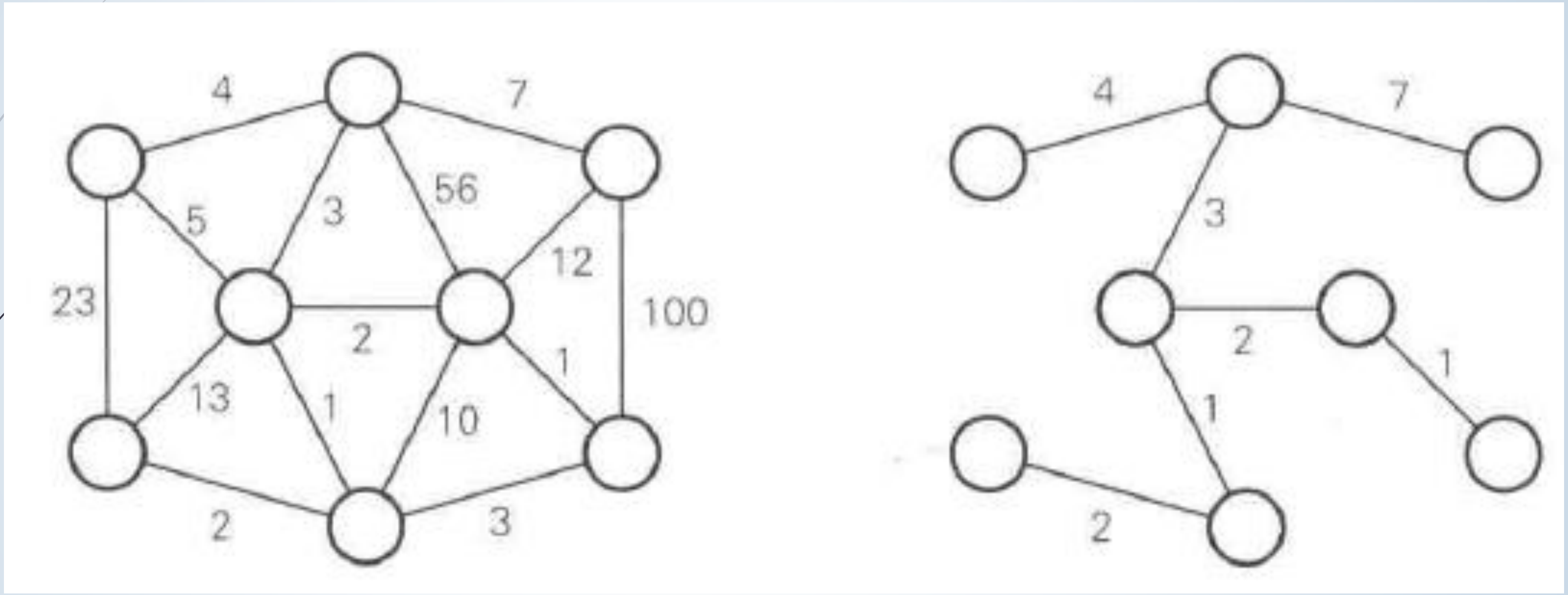


Minimum Spanning Tree

- A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.



https://en.wikipedia.org/wiki/Minimum_spanning_tree


4
7
5
3
56
23
12
100
2
13
1
10
1
2
3
4
7
3
2
1
1
2

Properties of MST

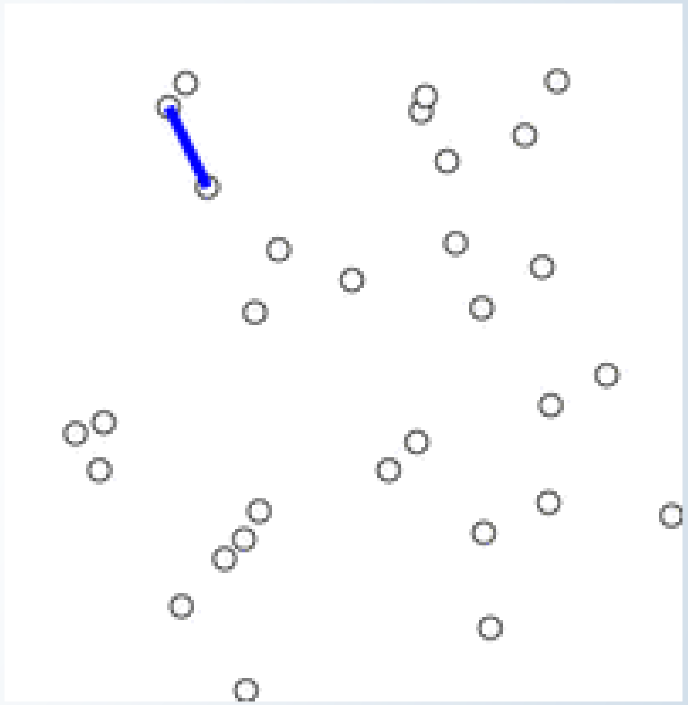
- If the graph has n vertices, the MST has n-1 edges.
- There may be several minimum spanning trees of the same weight; in particular, if all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
- If each edge has a distinct weight then there will be only one, unique minimum spanning tree. (proof ?)

MST Algorithms

- Borůvka's algorithm:
  - By Otakar Borůvka in 1926
  - Complexity: O(mlogn)
- Prim's algorithm
  - invented by Vojtěch Jarník in 1930
  - rediscovered by Prim in 1957 and Dijkstra in 1959
  - Complexity: O(m log n) or O(m + n log n)
- Kruskal's algorithm
  - Complexity: O(mlogn)
- Reverse-delete algorithm:
  - Not commonly used
  - Complexity: $O(m \log n (\log \log n)^3)$.

Prim’s Algorithm

1.Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2.Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3.Repeat step 2 (until all vertices are in the tree).



https://en.wikipedia.org/wiki/Prim%27s_algorith

https://algorithms.discrete.ma.tum.de/graph-algorithms/mst-prim/index_en.html

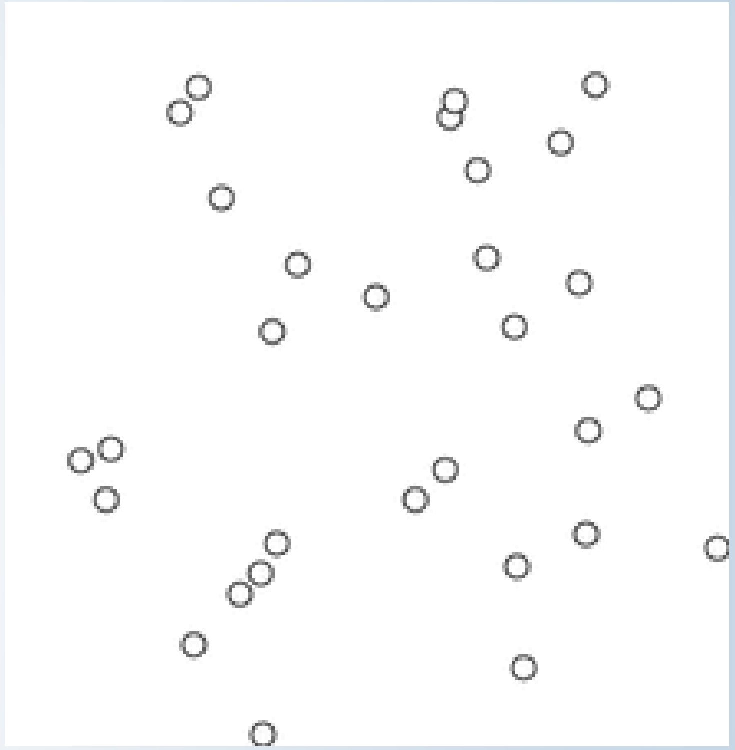
Kruskal’s Algorithm

- **Kruskal's algorithm** finds a minimum spanning forest of an undirected edge-weighted graph.
- If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized.
- For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

Kruskal’s Algorithm

1. create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
2. create a set *S* containing all the edges in the graph
3. while *S* is nonempty and *F* is not yet spanning
   A. remove an edge with minimum weight from *S*
   B. if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree



https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

```
algorithm Kruskal($G$) is
    F:= $\emptyset$
    for each v $\in$ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) $\neq$ FIND-SET(v) then
            F:= F $\cup$ {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

A disjoint set is used to implement Kruskal algorithm.

More examples of Kruskal algorithms.

Shortest path

Shortest Path

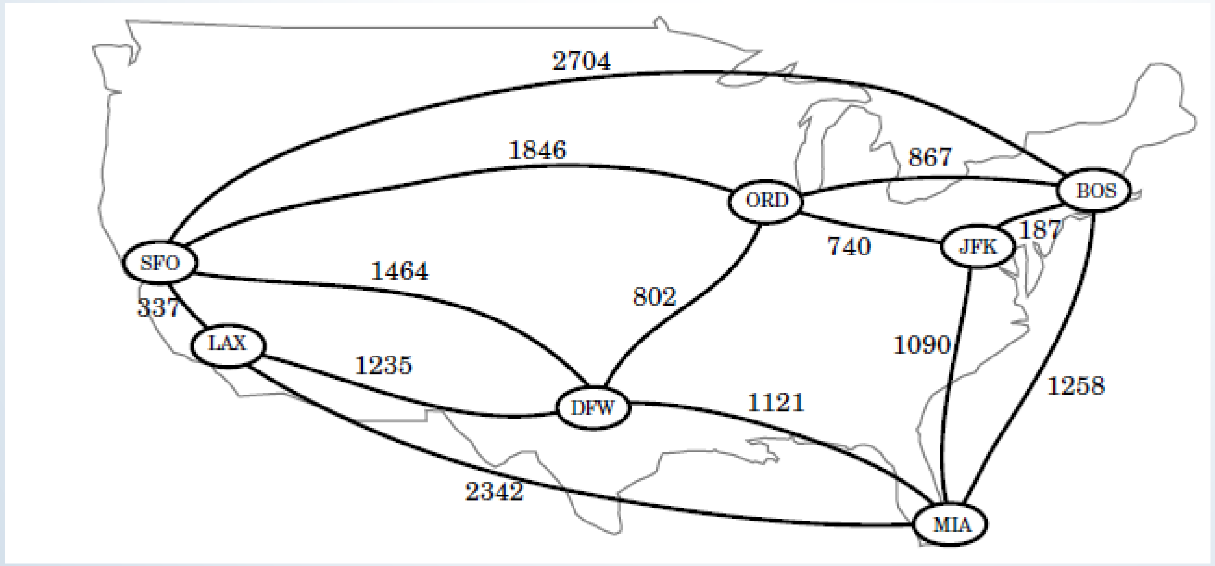
$\left\{ \begin{array}{l} \text{Single sourced} \\ \\ \text{All pairs} \end{array} \right.$

- Weighted graph: each edge e=(u,v) has a weight w(e)=w(u,v)
- The distance from a vertex u to a vertex v in G, denoted d(u,v), is the length of a minimum-length path (also called *shortest path*) from u to v, if such a path exists.

- The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges' weights is minimum.
  - If all edge weights are the same, this problem could be solved easily using **(BFS).**
  - If the edge weights are different, there are many different algorithms.

Single-sourced Shortest path

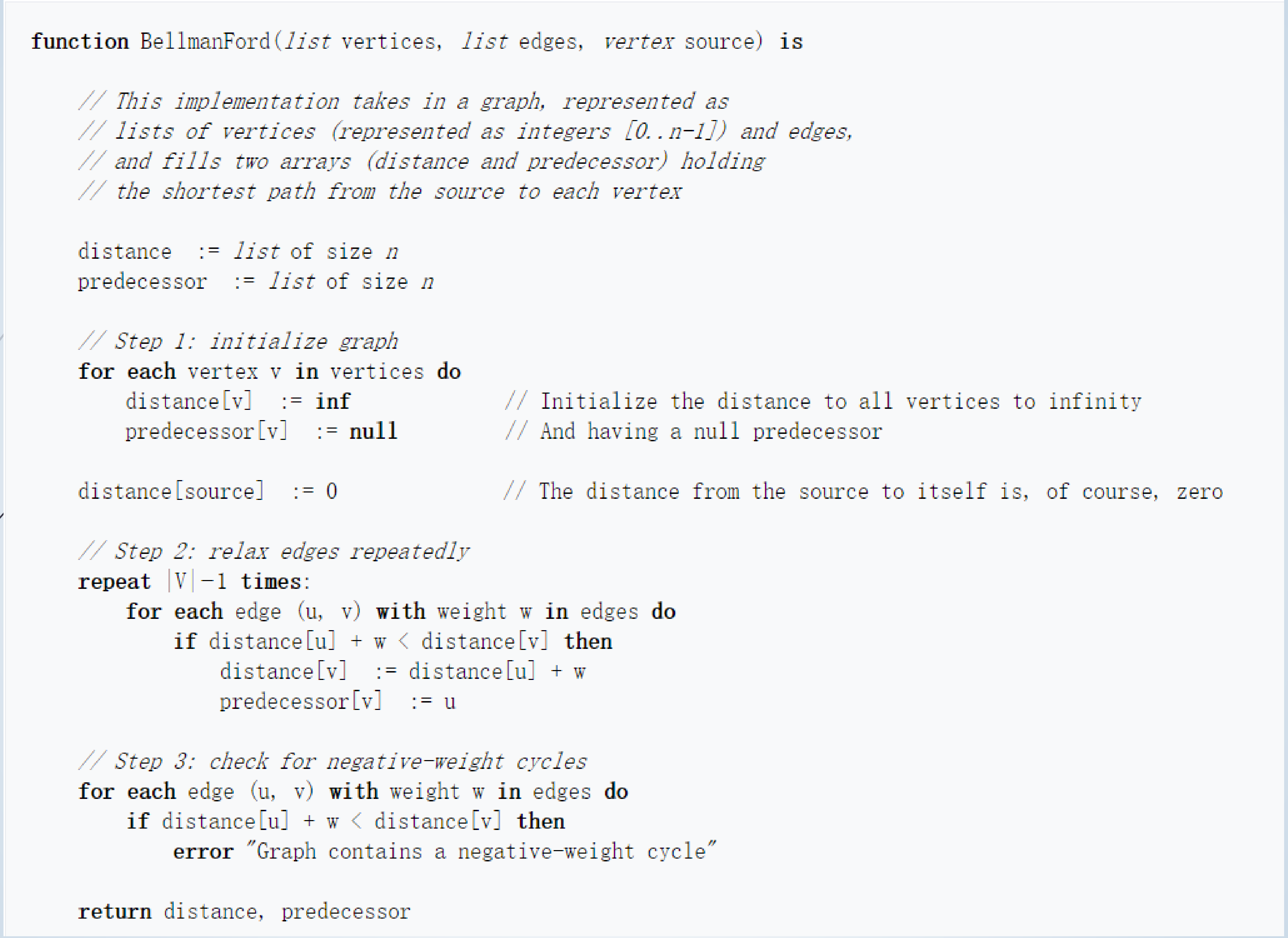
- Single-sourced:
  - find a shortest path from some vertex s to each other vertex in G, viewing the weights on the edges as distances.



Bellman Ford’s Algorithm

- Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph.
- It depends on the following concept:
  - Shortest path contains at most n-1 edges, because the shortest path couldn't have a cycle.

Why the shortest path couldn’t have a cycle?



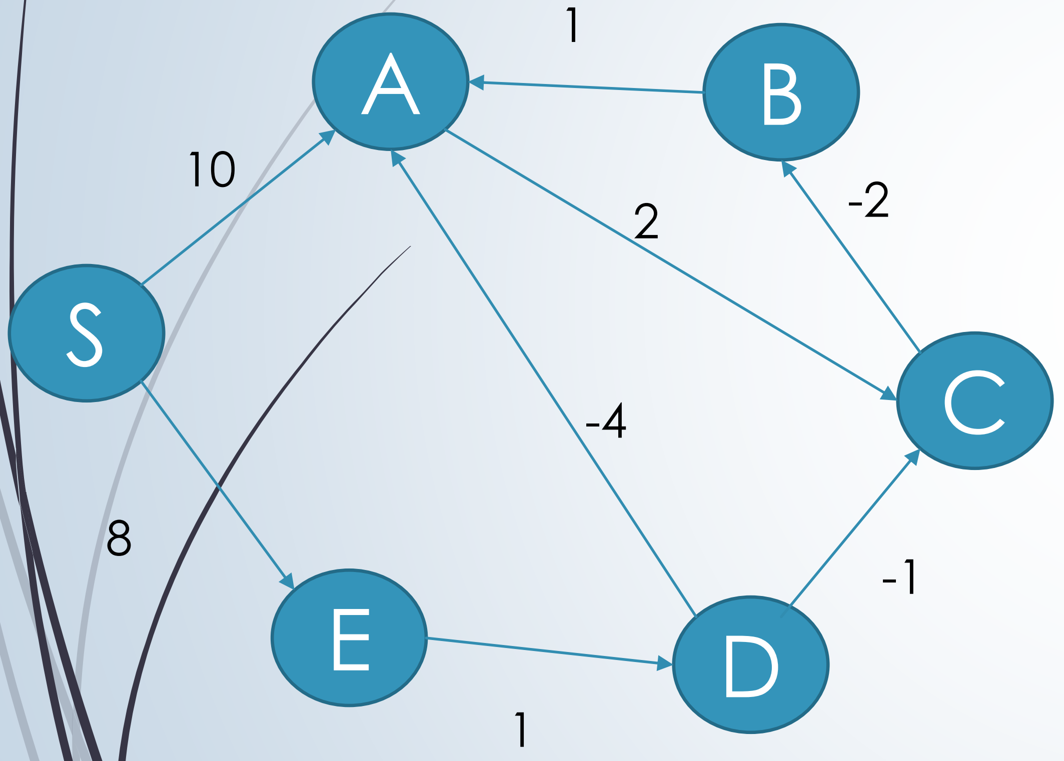
```
function BellmanFord(list vertices, list edges, vertex source) is

    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance  := list of size n
    predecessor  := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v]  := inf              // Initialize the distance to all vertices to infinity
        predecessor[v]  := null          // And having a null predecessor

    distance[source]  := 0               // The distance from the source to itself is, of course, zero

    // Step 2: relax edges repeatedly
    repeat |V|−1 times:
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v]  := distance[u] + w
                predecessor[v]  := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"

    return distance, predecessor
```

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

Bellman Ford's Algorithm

- The outer loop traverses from 0 : n−1.
- Loop over all edges,
  - check if the next node distance > current node distance + edge weight,
  - in this case update the next node distance to "current node distance + edge weight".
- A very important application of Bellman Ford is to check if there is a negative cycle in the graph,
- Time Complexity of Bellman Ford algorithm is O(V·E), in case $E=V^2$ the complexity will be $O(V^3)$.

Example

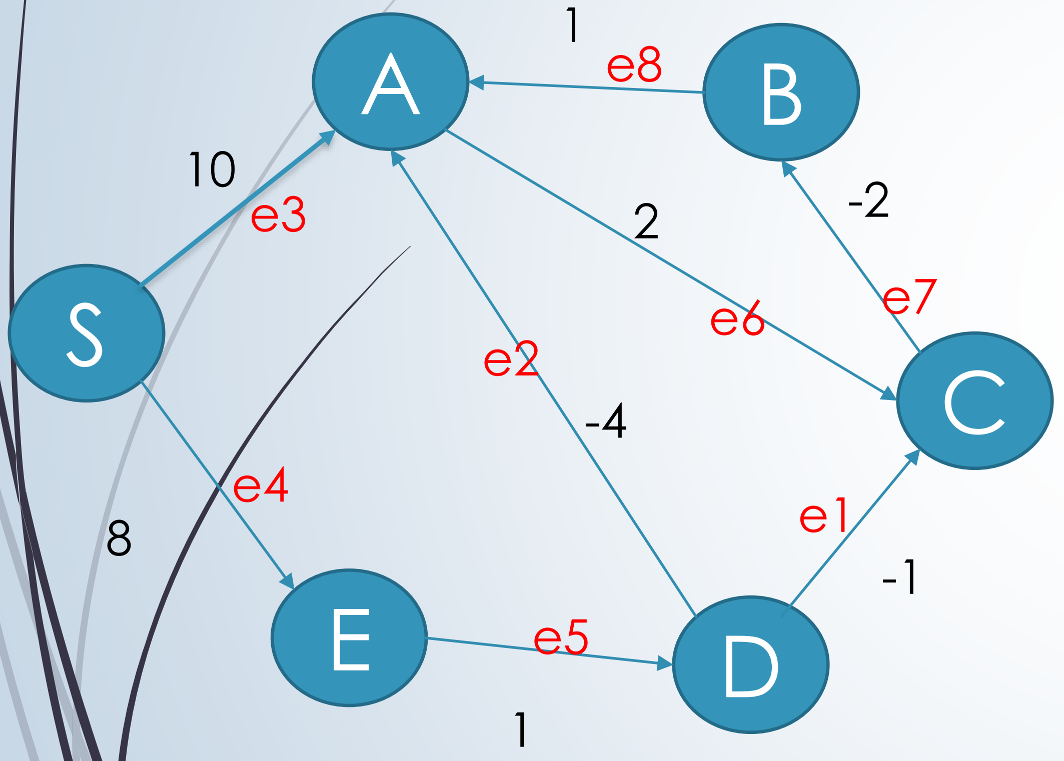


There are 6 vertices, so we need 5 iterations.

Initialize:

S	0
A	∞
B	∞
C	∞
D	∞
E	∞

Example

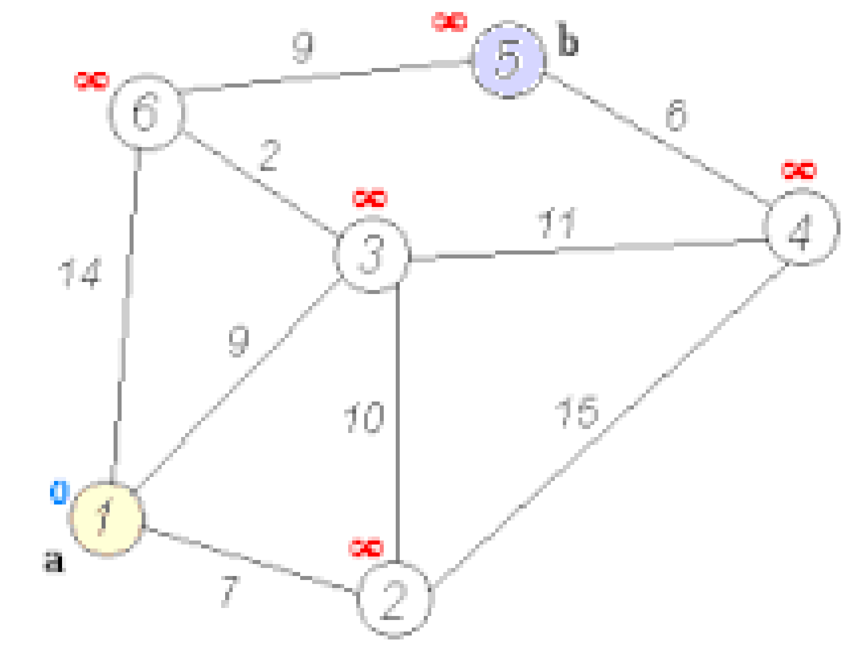


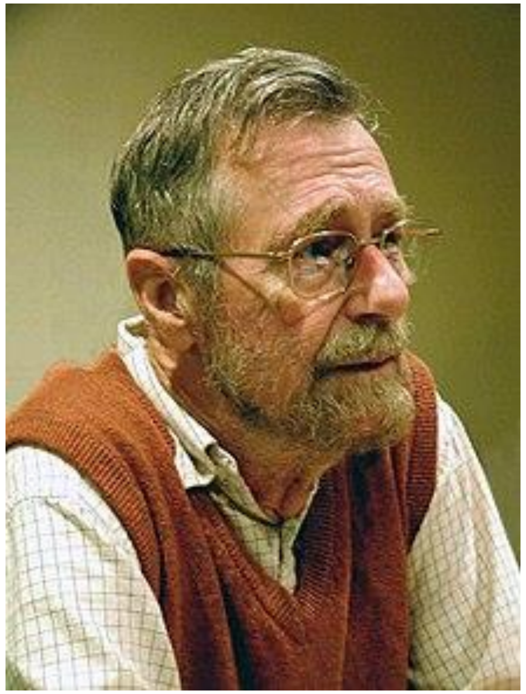
	0-th	1st	2nd	3rd	4th	5th
S	0					
A	∞	10	5			
B	∞	10	5			
C	∞	12	~~8~~7			
D	∞	9				
E	∞	8				

Improvements

- The Bellman–Ford algorithm may be improved in practice (although not in the worst case) by the observation that, if an iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes.
- With this early termination condition, the main loop may in some cases use many fewer than $|V| - 1$ iterations, even though the worst case of the algorithm remains unchanged.
- Worst case complexity: O(|V||E|)

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm





(1930-2002) He received the 1972 Turing Award for fundamental contributions to developing programming languages

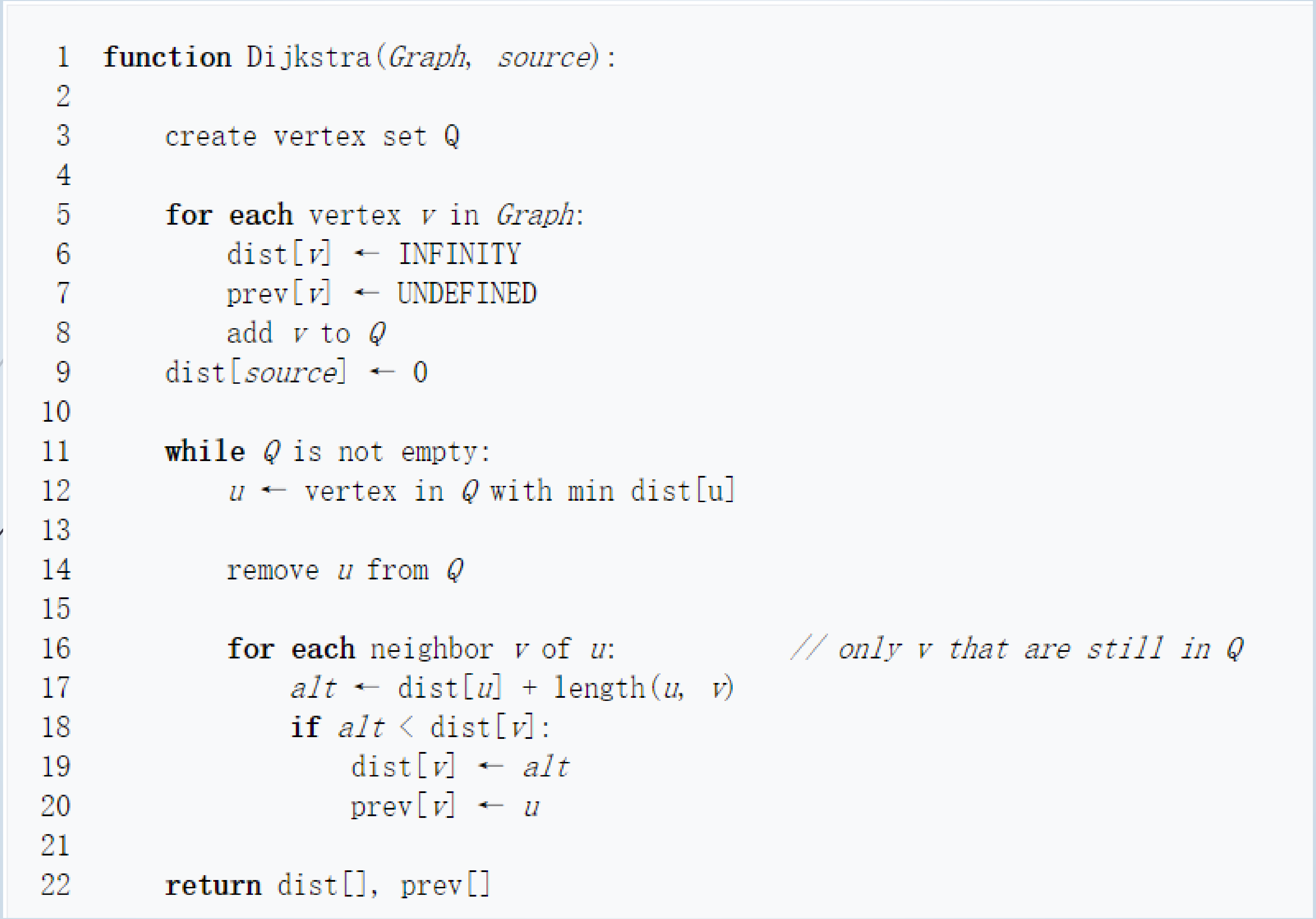
Dijkstra

Dijkstra's algorithm

- Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
- It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.
- Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph, producing a shortest-path tree.

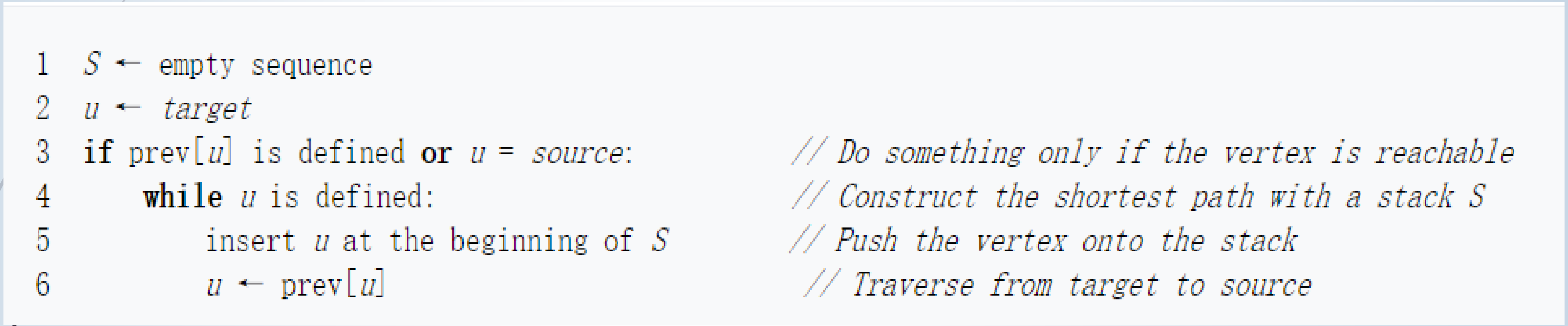
Dijkstra's Algorithm

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.



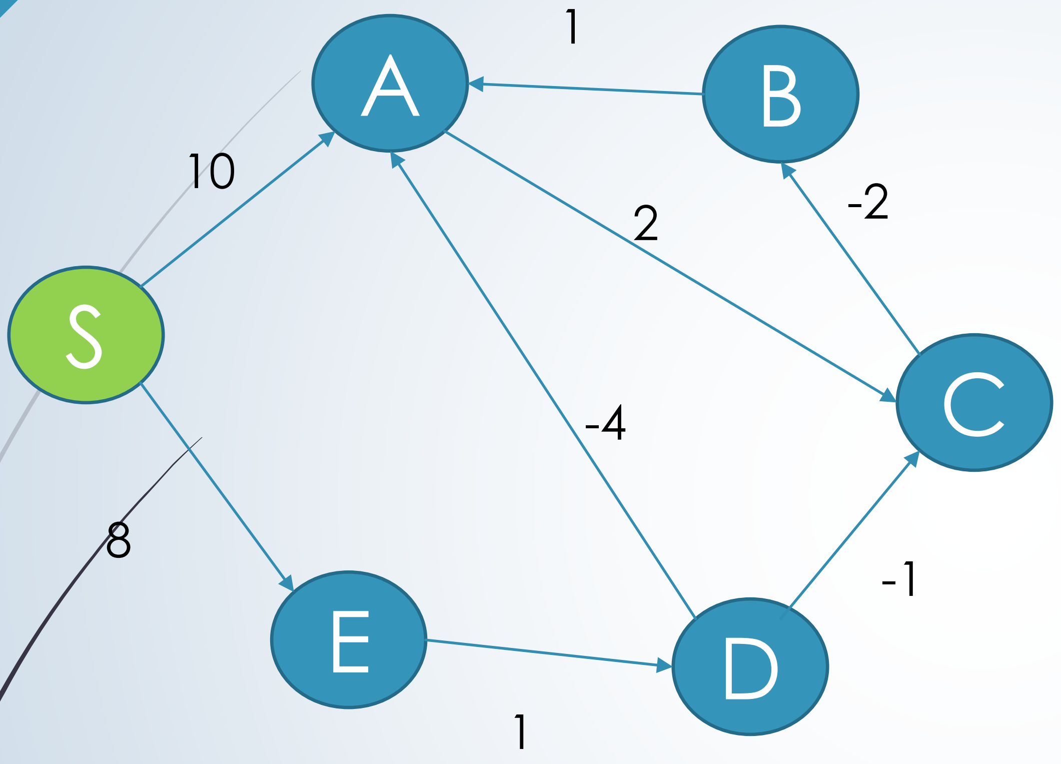
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

If we are only interested in a shortest path between vertices `source` and `target`, we can terminate the search after line 15 if $u = target$. Now we can read the shortest path from `source` to `target` by reverse iteration:



```
S ← empty sequence
u ← target
if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
    while u is defined:                       // Construct the shortest path with a stack S
        insert u at the beginning of S        // Push the vertex onto the stack
        u ← prev[u]                           // Traverse from target to source
```

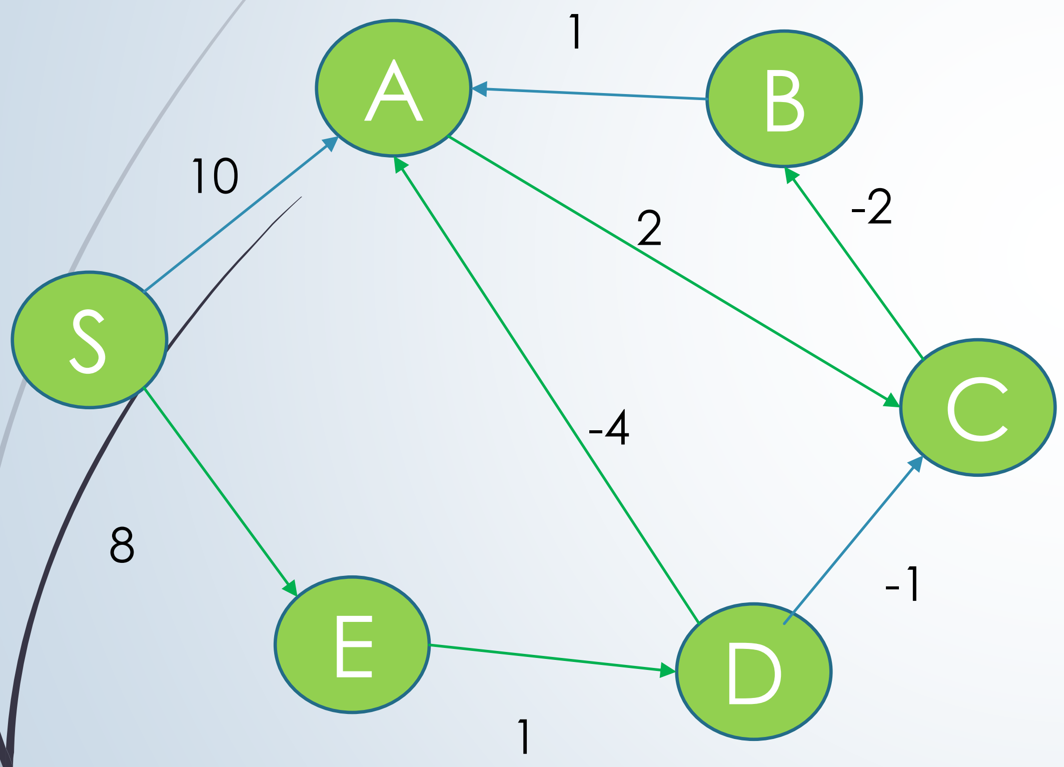
Now sequence `S` is the list of vertices constituting one of the shortest paths from `source` to `target`, or the empty sequence if no path exists.


S
A
B
C
D
E
10
8
1
2
-2
-4
-1
1

Complexity

- The original algorithm uses a min-priority queue and runs in time O((|V|+|E|)log |V|) (where |V| is the number of nodes and|E| is the number of edges)
- It can also be implemented in $O(|V|^2)$ using an array.
- Fredman & Tarjan propose using a Fibonacci heap min-priority queue to optimize the running time complexity to O(|E|+|V|log|V|). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.
- However, if we have to find the shortest path between all pairs of vertices, all of the above methods would be expensive in terms of time.

Shortest path tree

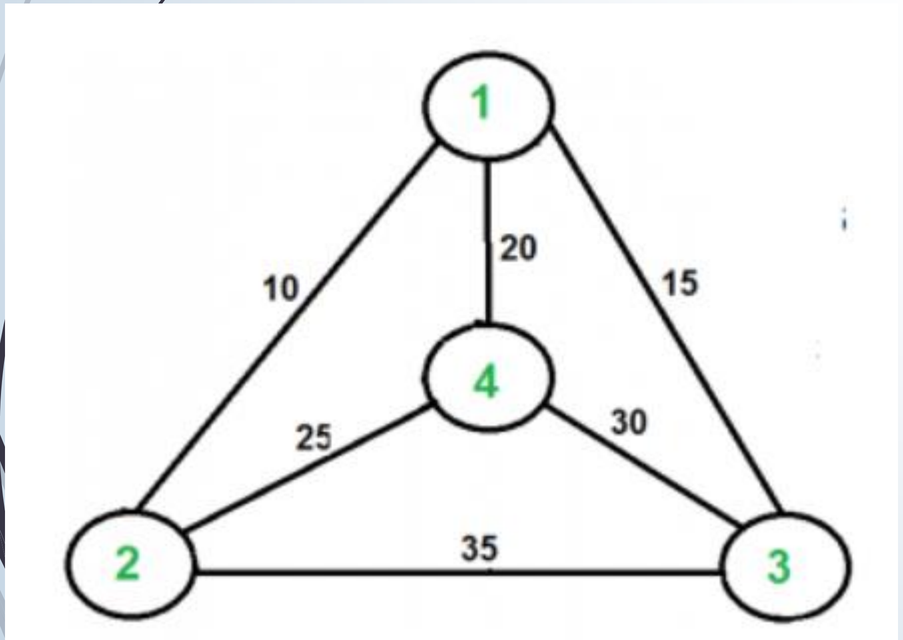


The collection of all shortest paths emanating from source *s* can be compactly represented by what is known as the ***shortest-path tree***. The paths form a rooted tree because if a shortest path from *s* to *v* passes through an intermediate vertex *u*, it must begin with a shortest path from *s* to *u*.

Travelling salesman problem (TSP)

Travelling salesman problem

- The Hamiltonian cycle problem: Given a graph, is there a tour that visits every city exactly once.
- The travelling salesman problem: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?



There are many Hamiltonian cycles. The shortest one is the solution of travelling salesman problem.

1->2->4->3->1

Travelling Salesman Problem

- The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem.
- In computational complexity theory, **NP-hardness** (non-deterministic polynomial-time hardness) is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP".
- For example: travelling salesman, Hamiltonian cycle, longest path, subset sum, partitioning.

Algorithm to TSP

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

Look at the Python code for TSP in the above webpage, how does the author implement