5.4.3 Testing strong connectivity

A directed graph is *strongly connected* if and only if there is a path between every ordered pair of vertices. In some applications, having a strongly connected graph is important. For example, if a graph represents a road network in which some roads are one-way, then the network should be strongly connected, otherwise it would be possible to go to a point and not be able to come back. Figure 12 shows two examples of strongly connected graphs.

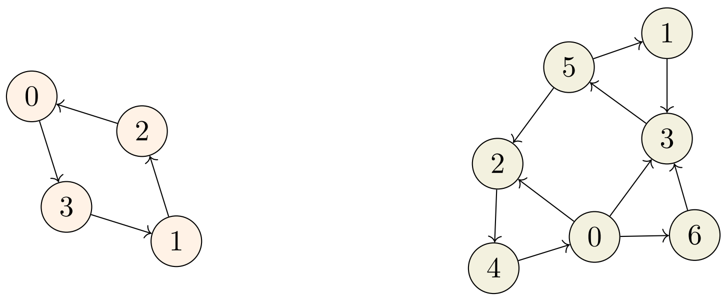


Figure 12: Two examples of strongly connected graphs. In each graph, for every pair of vertices x and y, there is a path from x to y and there is a path from y to x.

A simple algorithm to determine whether a directed graph is strongly connected is to run a traversal algorithm n times, each time starting at a different vertex. If after a traversal (BFS or DFS), all the vertices in the graph are processed, then all vertices are reachable from the starting vertex of the traversal. If this happens after all the n traversals, then all vertices are reachable from each other, and therefore, the graph is strongly connected. The run-time complexity of this algorithm is $O(n(n+m))$.

A more efficient algorithm can be devised. For this purpose, we introduce two new concepts: the *transpose* of a graph, and *hub*s.

Consider a directed graph G. The graph G^T is called the *transpose* or *reverse* of G if it is G with all edges reversed. More formally, given $G = (V, E)$ and $G^T = (V, E^T)$, if $(u, v) \in E$, then $(v, u) \in E^T$, and vice versa. The definition implies that if there is a path from x to y in G, then there is a path from y to x in G^T, and vice versa. It also follows that the transpose of G^T is G. Figure 13 shows an example of transpose.

If G is represented by an adjacency matrix, then G^T can be obtained by transposing G; that is, by reflecting the elements of the matrix along the main diagonal. This operation takes $\Theta(n^2)$ time. If G is represented by an adjacency list, transposing the graph involves looking at all the vertices and reversing edges going out of each vertex. This operation takes $\Theta(n+m)$ time.

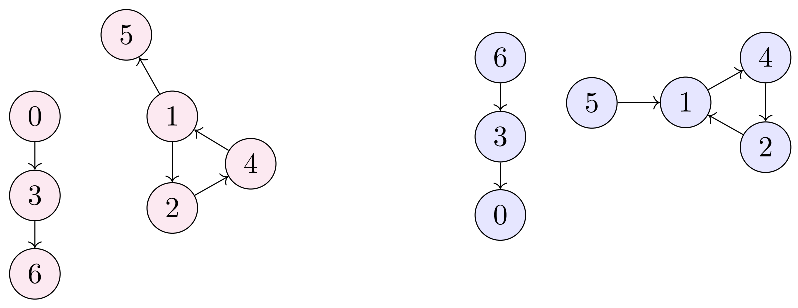


Figure 13: Two graphs that are the transpose of each other. Observe that for every path in one graph, there is a reverse path in the other graph. For example, one graph contains the path $(2, 4, 1, 5)$ and the other contains the path $(5, 1, 4, 2)$.

We define a vertex h to be a *hub* if: (a) every vertex is reachable from h; and (b) h is reachable from every vertex. If a graph has a hub h, then it is strongly connected because, for every pair of vertices x and y, x is connected to h and h is connected to y; thus x is connected to y. For the same reason, if a graph has a hub, then every vertex is a hub. Therefore, to test strong connectivity, it suffices to test whether an arbitrary vertex in the graph is a hub. This leads to the following algorithm which takes a graph G and an arbitrary vertex h, and determines whether the graph is strongly connected:

1. Run a graph traversal (BFS or DFS) on G starting at h;
2. If any vertex remains undiscovered, return False;
3. Construct a graph G^T which is the transpose of G;
4. Run a graph traversal (BFS or DFS) on G^T starting at h;
5. If any vertex remains undiscovered, return False, otherwise return True.

Comments:

- If after any of the two traversals, one or more vertices remain undiscovered, then we have found a pair of vertices that do not satisfy the condition of strong connectivity.
- The run time complexity of the algorithm is the same as graph traversal; that is, $O(n + m)$.

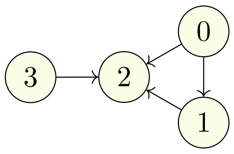
5.4.4 Topological sorting

A *topological ordering* of a directed graph is an ordering of its vertices, such that, if there is an edge (u, v) in the graph, then vertex u is placed in some position before vertex v. Any Directed Acyclic Graph (DAG) has at least one topological ordering.

Example 5.4. Consider the following graph.

```
D 4
0 2
0 1
1 2
3 2
```

The following is the drawing of the graph.



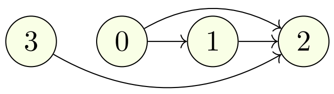
Any of the following is a topological ordering:

$3, 0, 1, 2$

$0, 3, 1, 2$

$0, 1, 3, 2$

The visual interpretation of a topological ordering is an arrangement of the vertices of the graph on a horizontal line such that all the edges point in the same direction (left or right). For example, the first topological ordering listed above can be visualized as follows.



The following includes some examples of topological sorting on DAGs.

- In software engineering, build automation systems (e.g. Make and Apache Maven) apply topological sort to the DAG of dependencies between software components to find a valid order of building software.

- Package management systems (e.g. dpkg, RPM, and Homebrew) apply topological sort to the DAG of package dependencies to find a valid order of installation.
- A prerequisite graph of courses at a university is a DAG. A topological ordering allows doing courses in some valid order.
- Copying relational databases must be done according to a topological ordering of tables (where dependencies are based on foreign keys) so that the referential integrity of the data is maintained during the operation.

Example 5.5. Consider the following graph.

```
D 4
```

The graph does not have any edges. Any ordering of the vertices is a valid topological ordering. There are 4! such orderings.

Example 5.6. Consider the following graph.

```
D 4
0 1
1 2
2 3
3 0
```

The graph has a cycle (i.e. it is not a DAG), and therefore, does not have any topological ordering.

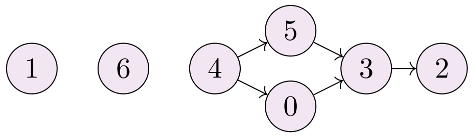
Given a DAG, the following algorithm generates a topological ordering:

1. Initialise the state and parent arrays as usual;
2. Initialise an empty stack;
3. Iterate over all vertices of the graph:
   - If the vertex is undiscovered, start a modified version of DFS-Loop on the vertex. In the modified version, as soon as a vertex is processed, it is pushed onto the stack;
4. Return the content of the stack from top to bottom as a topological ordering.

Example 5.7. Consider the following graph.

```
D 7
0 3
4 0
5 3
3 2
4 5
```

The drawing of the graph is:



The following table shows the trace of the topological sorting algorithm on this graph. The last two columns show the values of *state* and *stack* after each iteration. We use a Python list for the stack; items are pushed and popped on the right. Note that, the trace does not show the call stack.

iteration	call	*state*	*stack*
	(initialisation)	`[U, U, U, U, U, U, U]`	`[]`
0	DFS-Loop(0)	`[P, U, P, P, U, U, U]`	`[2, 3, 0]`
1	DFS-Loop(1)	`[P, P, P, P, U, U, U]`	`[2, 3, 0, 1]`
2		`[P, P, P, P, U, U, U]`	`[2, 3, 0, 1]`
3		`[P, P, P, P, U, U, U]`	`[2, 3, 0, 1]`
4	DFS-Loop(4)	`[P, P, P, P, P, P, U]`	`[2, 3, 0, 1, 5, 4]`
5		`[P, P, P, P, P, P, U]`	`[2, 3, 0, 1, 5, 4]`
6	DFS-Loop(6)	`[P, P, P, P, P, P, P]`	`[2, 3, 0, 1, 5, 4, 6]`

Some lines in the table involve recursive calls that are not shown. At the beginning of iteration 0, vertex 0 is undiscovered; therefore, the modified version of DFS-Loop is called on 0. At this point, vertex 0 is discovered. In the 'for' loop inside DFS-Loop, the undiscovered vertex 3 is encountered, which leads to calling DFS-Loop (recursively) on vertex 3, which then leads to calling DFS-Loop on vertex 2. Vertex 2 does not have any outgoing edges; therefore, the vertex is marked as processed, 2 is pushed onto the stack, and then the recursive call is finished. Next, the recursive call on vertex 3 resumes but there are no more edges; so vertex 3 is also marked as processed and pushed onto the stack. Then, the DFS-Loop call on 0 resumes, which leads to marking 0 as processed and pushing it onto the stack. At the end of iteration 0, the stack contains `[2, 3, 0]` (where the top of the stack is on the right). Some iterations in the table (e.g. iteration 2) do not involve any calls to DFS because the vertex is already processed.

Why does it work? The topological sorting algorithm gradually pushes vertices onto a stack, and at the end, returns the content of the stack from top to bottom. The definition of topological ordering requires that if the graph has an edge (u, v), then u must appear in some position before v in the ordering. Therefore, we must show that, if there is an edge (u, v) in the graph, the algorithm pushes v onto the stack before u.

Inside DFS, when the edge (u, v) is considered, u is discovered but v could be either undiscovered, discovered, or processed:

- If v is undiscovered, then it becomes discovered and a new DFS call starts at vertex v. This means that the call DFS-Loop(v) finishes before DFS-Loop(u); that is, v is processed before u, and therefore, v is pushed onto the stack before u.
- If v is discovered, that means that v is somewhere on the call stack; that is, we have found a path from v to itself. This is not possible because DAGs cannot have cycles.
- If v is processed, then it has already been pushed onto the stack, while u has not yet been pushed.

5.4.5 Cycle detection

Sometimes we want to know if a graph has a cycle. This is, for example, useful to determine whether a directed graph is a DAG, and thus, has a topological ordering.

During DFS traversal, when examining the outgoing edges of a vertex u, if the edge (u, v) goes to a vertex v that is already discovered (that is, it is on the call stack), then the graph has a cycle.

For undirected graphs, the process is similar, but there is one exception: when examining the outgoing edges of a vertex u, we ignore the edge (u, v) where v is the parent of u (v is guaranteed to be discovered but that doesn't count as a cycle because a cycle has to be a path, and a path is not allowed to use an edge more than once).

For undirected graphs with multiple components, or for directed graphs, multiple DFS calls may need to be made in order to find a cycle. Therefore, similar to finding the components of a graph or topological ordering, a for loop must check the state of each vertex in the graph, and run a DFS from that vertex if it is undiscovered.

6 Weighted graph algorithms

6.1 Minimum spanning trees

A *spanning tree* of an undirected graph is a subgraph that is a tree and includes all the vertices of the graph. A *minimum spanning tree* (MST) of a weighted undirected graph is a spanning tree that has the lowest total weight among all other spanning trees. Figure 14 shows a weighted undirected graph and one of its MSTs.

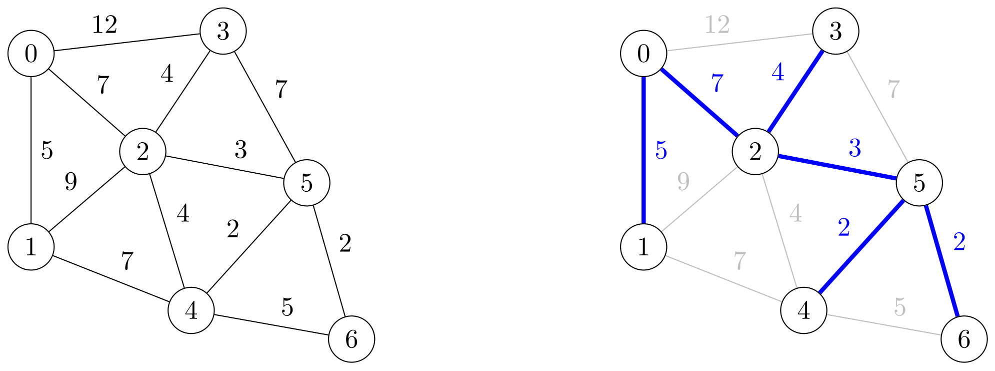


Figure 14: A weighted undirected graph (left) and one of its minimum spanning trees (right). MSTs are not necessarily unique. For this graph, another MST can be obtained by removing the edge $\{0, 2\}$ from the current MST and adding the edge $\{1, 4\}$. All MSTs of a graph have the same total weight.

6.1.1 Prim's algorithm

Given a weighted undirected graph with one component, Prim's algorithm finds a minimum spanning tree. The algorithm is called with an arbitrary vertex as the starting point which forms a one-vertex tree. The tree gradually grows in each iteration by adding the smallest edge between a vertex that is part of the tree and a vertex that is not.

Running Prim's algorithm on vertex 0 of the graph in Figure 14 starts with a tree which initially has only one vertex, 0. The algorithm then looks at all the edges that connect a vertex in the tree to a vertex out of the tree. These edges are $\{\{0, 1\}, \{0, 2\}, \{0, 3\}\}$. Among these, $\{0, 1\}$ with the weight of 5 is the smallest, and therefore, is added to the tree. Next, the algorithm looks at edges $\{\{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 4\}\}$. At this stage, the algorithm can pick either $\{0, 2\}$ or $\{1, 4\}$ which are equally good. The algorithm continues until all vertices of the graph become part of the tree.

```
procedure $\textsc{Prim}(\mathit{Adj}, s)$
  $n \leftarrow$ number of vertices (the length of $\mathit{Adj}$)
  $\mathit{in\text{-}tree} \leftarrow$ an array of length $n$ filled with $\textsc{False}$
  $\mathit{distance} \leftarrow$ an array of length $n$ filled with $\infty$
  $\mathit{parent} \leftarrow$ an array of length $n$ filled with $\textsc{null}$
  $\mathit{distance}[s] \leftarrow 0$
  while $\neg \mathit{all}(\mathit{in\text{-}tree})$
    $u \leftarrow \textsc{Next-Vertex}(\mathit{in\text{-}tree}, \mathit{distance})$
    $\mathit{in\text{-}tree}[u] \leftarrow \textsc{True}$
    for $v, \mathit{weight}$ in $\mathit{Adj}[u]$
      if $\neg\, \mathit{in\text{-}tree}[v] \wedge \mathit{weight} < \mathit{distance}[v]$
        $\mathit{distance}[v] \leftarrow \mathit{weight}$
        $\mathit{parent}[v] \leftarrow u$
  return $\mathit{parent}, \mathit{distance}$
```

In the algorithm:

- the *in-tree* array shows which vertices are currently part of the growing tree;
- the *distance* array contains the current distance of each vertex to the growing tree;
- the *parent* array represents the structure of the tree;
- The function Next-Vertex(*in-tree*, *distance*) returns a closest vertex that is not yet part of the tree; and
- the variable *weight* is the weight of the edge (u, v) that is being considered.

Analysis. In a graph with n vertices and m edges, the while-loop iterates n times until all vertices are part of the tree. The call Next-Vertex(*in-tree*, *distance*) takes time proportional to n. Hence, the time inside the while-loop, but outside the for-loop, is $O(n^2)$. The time inside the for-loop over all iterations of the while-loop is $O(m)$ as every edge is considered at most once (as in BFS). Thus, the total time complexity is in $O(\max(m, n^2)) = O(n^2)$ because $m \leq n^2$.

If the distance array is replaced with a more suitable data structure, such as a heap-based priority queue, the time to find the next vertex reduces to $\log n$. Then, the time complexity of the entire algorithm will be in $O(m + n \log n)$. Such a priority queue should support an 'update' operation (because of line 12); otherwise, further modifications must be done.

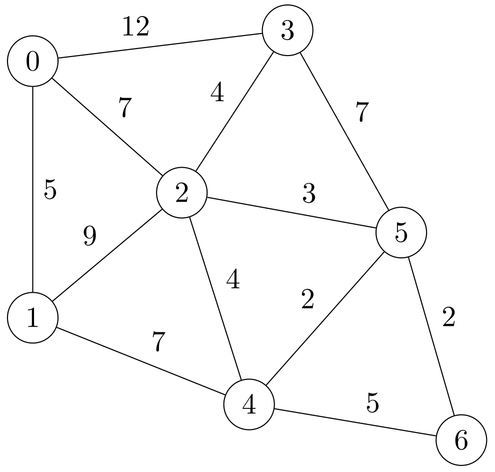
Example trace of Prim's algorithm

```
graph_string = """\
U 7 W
0 1 5
0 2 7
0 3 12
1 2 9
2 3 4
1 4 7
2 4 4
2 5 3
3 5 7
4 5 2
4 6 5
5 6 2
"""

adj_list = adjacency_list(graph_string)

0 [[(1, 5), (2, 7), (3, 12)],
1  [(0, 5), (2, 9), (4, 7)],
2  [(0, 7), (1, 9), (3, 4), (4, 4), (5, 3)],
3  [(0, 12), (2, 4), (5, 7)],
4  [(1, 7), (2, 4), (5, 2), (6, 5)],
5  [(2, 3), (3, 7), (4, 2), (6, 2)],
6  [(4, 5), (5, 2)]]

prim(adj_list, 0)
```



```
u         Distance                    Parent                    In-tree
    of vertex i to tree
    0  1  2  3  4  5  6      0  1  2  3  4  5  6      0  1  2  3  4  5  6
   [0, *, *, *, *, *, *]    [-, -, -, -, -, -, -]    [F, F, F, F, F, F, F]
0  [0, 5, 7,12, *, *, *]    [-, 0, 0, 0, -, -, -]    [T, F, F, F, F, F, F]
1  [0, 5, 7,12, 7, *, *]    [-, 0, 0, 0, 1, -, -]    [T, T, F, F, F, F, F]    '*' indicates infinity.
2  [0, 5, 7, 4, 4, 3, *]    [-, 0, 0, 2, 2, 2, -]    [T, T, T, F, F, F, F]
5  [0, 5, 7, 4, 2, 3, 2]    [-, 0, 0, 2, 5, 2, 5]    [T, T, T, F, F, T, F]    '-' indicates None.
4  [0, 5, 7, 4, 2, 3, 2]    [-, 0, 0, 2, 5, 2, 5]    [T, T, T, F, T, T, F]
6  [0, 5, 7, 4, 2, 3, 2]    [-, 0, 0, 2, 5, 2, 5]    [T, T, T, F, T, T, T]
3  [0, 5, 7, 4, 2, 3, 2]    [-, 0, 0, 2, 5, 2, 5]    [T, T, T, T, T, T, T]
```

The first line shows the values of the variables before entering the while-loop (after initialisation).
The remaining lines show the values after completion of each iteration of the while-loop.

6.2 Shortest path trees

In order to find a path that has the shortest distance (sum of weights) from a source vertex to a destination vertex, we solve a more general version of the problem, which is finding the shortest path from the source vertex to every other vertex in the graph. It turns out that the more general version can be solved without loss of efficiency.

Given a single-component weighted undirected graph G, and a vertex s, a shortest-path tree rooted at vertex s is a spanning tree, such that, the path distance from the root s to any other vertex v in the tree is the shortest path distance from s to v in the graph G. Figure 15 shows an example of a shortest path tree. Note that, unlike minimum spanning trees, the shortest path tree is only valid for one source vertex which must be at the root of the tree. For example, in Figure 15, the tree does not provide the shortest path between vertices 4 and 5.

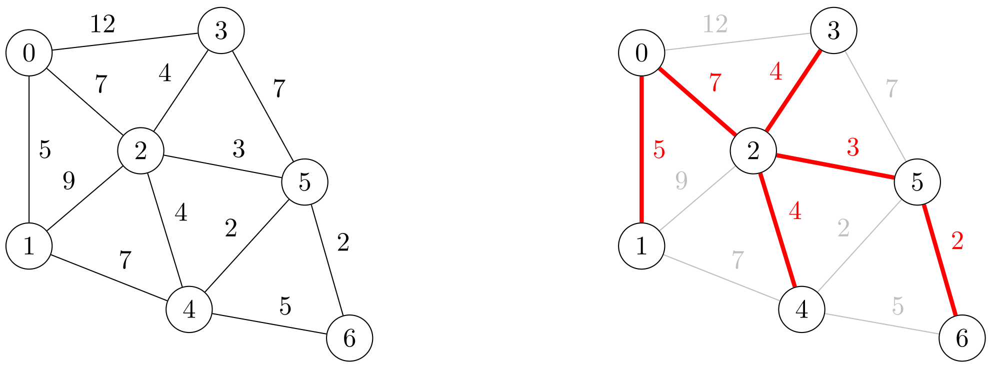


Figure 15: Left: A weighted undirected graph; Right: a shortest path tree rooted at vertex 0. Every path in the tree from vertex 0 to any other vertex, is the shortest distance (smallest sum of weights) among all other possibles paths in the graph from 0 to that vertex.

6.2.1 Dijkstra's algorithm

Given a graph with non-negative edge weights and a starting vertex, the algorithm finds the shortest path from the starting vertex to any other vertex reachable from it.

The algorithm gradually grows a *shortest path tree* rooted at the starting vertex. In each iteration, a new edge is added to the tree by selecting an edge that connects a vertex in the tree to a vertex outside that is closest to the starting vertex.

```
1  procedure DIJKSTRA($Adj, s$)
2    $n \leftarrow$ number of vertices (the length of $Adj$)
3    $in\text{-}tree \leftarrow$ an array of length $n$ filled with FALSE
4    $distance \leftarrow$ an array of length $n$ filled with $\infty$
5    $parent \leftarrow$ an array of length $n$ filled with NULL
6    $distance[s] \leftarrow 0$
7    while $\neg all(in\text{-}tree)$
8      $u \leftarrow$ NEXT-VERTEX($in\text{-}tree, distance$)
9      $in\text{-}tree[u] \leftarrow$ TRUE
10     for $v, weight$ in $Adj[u]$
11       if $\neg\, in\text{-}tree[v] \wedge distance[u] + weight < distance[v]$
12         $distance[v] \leftarrow distance[u] + weight$
13         $parent[v] \leftarrow u$
14   return $parent, distance$
```

Note the following.

- The algorithm is very similar to Prim's algorithm; only lines 11 and 12 are different because the meaning of distance is different in the two algorithms.
- The *distance* array has the current distance of each vertex from the starting (source) vertex. For vertices that are part of the tree, this is the sum of weights along the shortest path from the starting vertex.
- The *in-tree* array shows the vertices to which the shortest path (from the starting vertex) is known so far.
- Shortest paths can be extracted from the *parent* array.
- The algorithm has the same run-time complexity and Prim's algorithm. It can also be improved in the same way by using an efficient priority queue.

Comments:

- The correctness of Dijkstra's algorithm does not hold for graphs in which some edges have negative weights. For such graphs, the algorithm may produce a non-optimal output (a path that is not the shortest distance).
- An MST and a shortest path tree of a graph are not necessarily the same. In other words, a path from the (arbitrary) root of an MST to another vertex is not necessarily a path with the shortest distance in the graph. Similarly, while a shortest path tree is a spanning tree, it is not necessarily an MST.

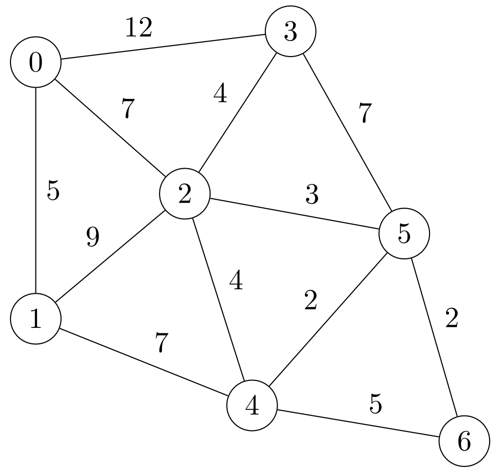
Example trace of Dijkstra’s algorithm

```
graph_string = """\
U 7 W
0 1 5
0 2 7
0 3 12
1 2 9
2 3 4
1 4 7
2 4 4
2 5 3
3 5 7
4 5 2
4 6 5
5 6 2
"""

adj_list = adjacency_list(graph_string)

0 [[(1, 5), (2, 7), (3, 12)],
1  [(0, 5), (2, 9), (4, 7)],
2  [(0, 7), (1, 9), (3, 4), (4, 4), (5, 3)],
3  [(0, 12), (2, 4), (5, 7)],
4  [(1, 7), (2, 4), (5, 2), (6, 5)],
5  [(2, 3), (3, 7), (4, 2), (6, 2)],
6  [(4, 5), (5, 2)]]

dijkstra(adj_list, 0)
```



```
u         Distance                    Parent                   In-tree
    of vertex i to root
    0  1  2  3  4  5  6       0  1  2  3  4  5  6       0  1  2  3  4  5  6
   [0, *, *, *, *, *, *]    [-, -, -, -, -, -, -]    [F, F, F, F, F, F, F]
0  [0, 5, 7,12, *, *, *]    [-, 0, 0, 0, -, -, -]    [T, F, F, F, F, F, F]
1  [0, 5, 7,12,12, *, *]    [-, 0, 0, 0, 1, -, -]    [T, T, F, F, F, F, F]       '*' indicates infinity
2  [0, 5, 7,11,11,10, *]    [-, 0, 0, 2, 2, 2, -]    [T, T, T, F, F, F, F]
5  [0, 5, 7,11,11,10,12]    [-, 0, 0, 2, 2, 2, 5]    [T, T, T, F, F, T, F]       '-' indicates None.
3  [0, 5, 7,11,11,10,12]    [-, 0, 0, 2, 2, 2, 5]    [T, T, T, T, F, T, F]
4  [0, 5, 7,11,11,10,12]    [-, 0, 0, 2, 2, 2, 5]    [T, T, T, T, T, T, F]
6  [0, 5, 7,11,11,10,12]    [-, 0, 0, 2, 2, 2, 5]    [T, T, T, T, T, T, T]
```

The first line shows the values of the variables before entering the while-loop (after initialisation).
The remaining lines show the values after completion of each iteration of the while-loop.