# Structured Programming

# COMP1110/6710

Image Courtesy NASA/JPL-Caltech.

Australian National University

# Trees revisited (BFS and DFS)

Australian National University

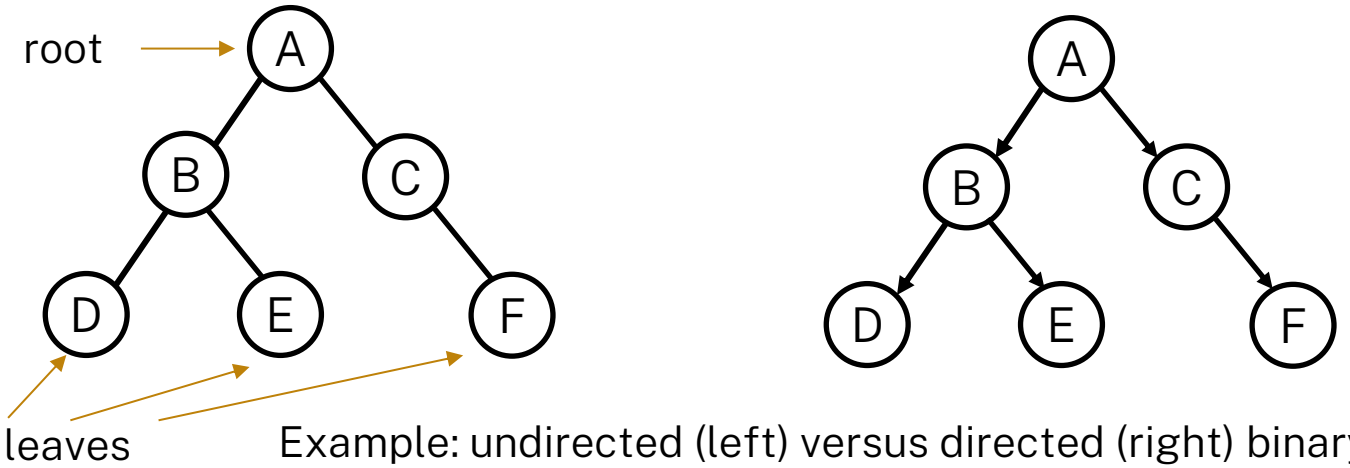
# Trees (recap)

- A tree is a hierarchical data structure that consists of nodes connected by edges
- Each node can have multiple children nodes, but only one parent node
- The node at the top of the hierarchy is called the **root node**, and it is the only node in the tree that has no parent
- A node can have no children. Such nodes are called **leaf nodes** of the tree
- A node that has at least one children is called **inner node**
- Arity (a.k.a. degree) is the max number of children any node in the tree can have
- Trees are non-linear data structures, i.e., there are multiple ways to traverse them, in contrast to linear data structures (such as arrays or lists), in which there is only one natural sequence for traversal (i.e., from first to the last element)
- **Question:** according to the previous definition, are lists also (a special kind of) trees?




# Undirected versus directed trees

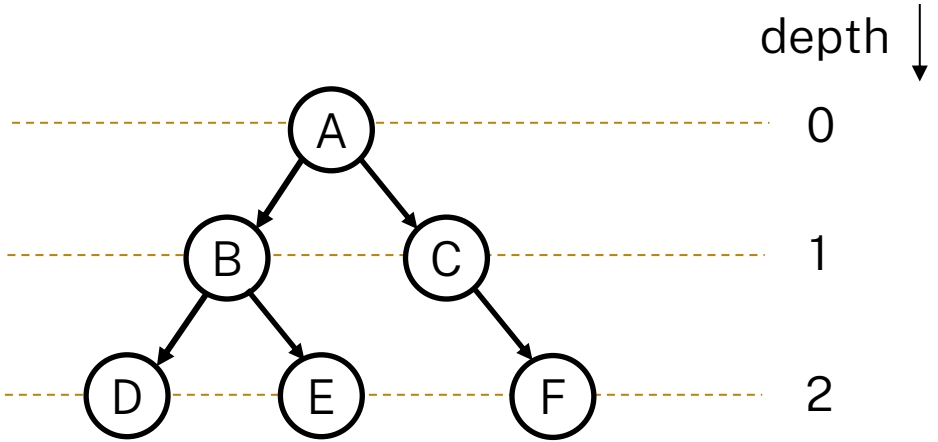
- Trees can (*conceptually*) be either **undirected** or **directed**
- In an **undirected** tree, all edges are bidirectional, i.e., a parent is connected to its children, and children are connected to their parent (note: bidirectional edges are typically implied in diagrams without arrow-heads)
- In a **directed** tree, the edges have a direction. The direction of an edge connecting a parent and its child is such that the edge departs the parent and points to the child




Example: undirected (left) versus directed (right) binary (2-ary) trees

# Depth of a tree node

- In a tree, there is always only one path from the root to any other node in the tree
- The depth of a node is defined as the length (number of edges) of the path from the root to that node



# Tree traversals (DFS and BFS)

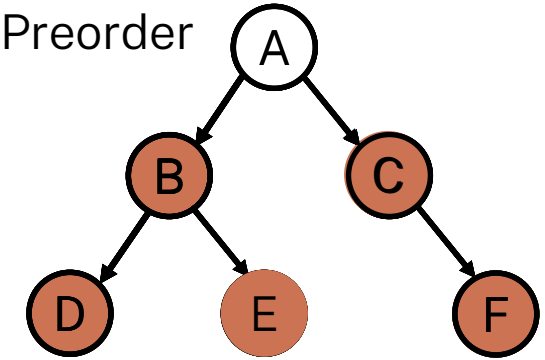
- Refers to the process of visiting the tree nodes **exactly once** in a certain order
- As mentioned earlier, there are multiple ways a tree can be traversed
- Two common forms are:
  - **Depth-first search (DFS).** Explore as deep as possible along a branch till a leaf is reached. Then, backtrack to node in another branch (e.g., sibling of leaf, sibling of parent, sibling of grand-parent, etc.)
  - **Breadth-first search (DFS).** Starting from the root (i.e., depth=0), explore all nodes at given depth before going deeper to the next depth.




# Tree depth-first search (DFS)

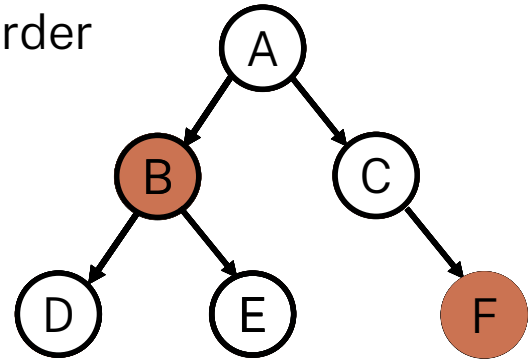
- There are three variants of DFS for trees: **preorder**, **inorder**, and **postorder**
- For binary (2-ary) trees, these are defined as follows:
  - Preorder DFS traversal: current node → left subtree → right subtree
  - Inorder DFS traversal: left subtree → current node → right subtree
  - Postorder DFS traversal: left subtree → right subtree → current node

Preorder



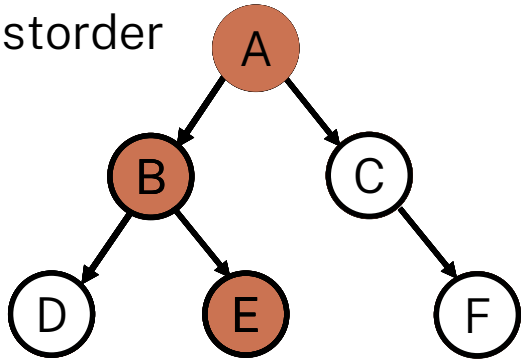

A B D E C F

Inorder



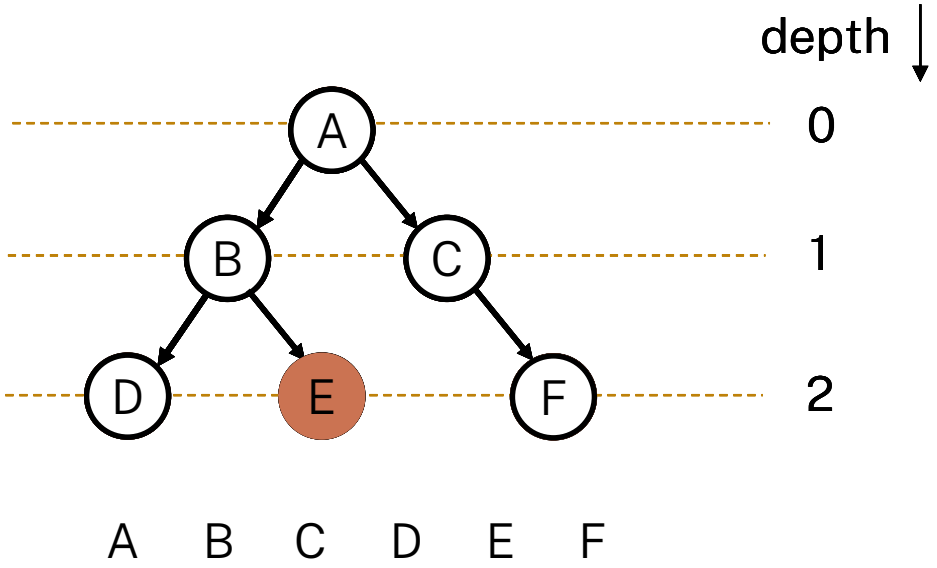

D B E A C F

Postorder




D E B F C A

# Tree breadth-first search (BFS)



# Implementing tree traversals

- Tree depth-first search (DFS)
  - Recursively by implicitly using the call stack (***we have been here before!***)
  - Iteratively **using a stack explicitly**: Last-In First-Out (LIFO) data structure
- Tree breadth-first search (BFS)
  - Iteratively using a queue explicitly: First-In First-Out (FIFO) data structure
  - Recursively passing a list with the nodes at current depth



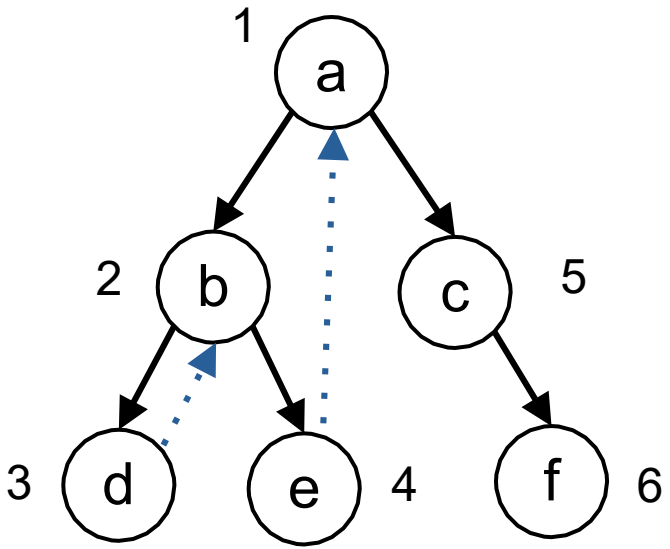

# Iterative Tree DFS (pseudocode)

1. Create an empty **stack** of tree nodes
2. Push the root node into the top of the **stack**
3. If stack is empty, STOP. If stack not empty, pop from **stack** the node at the top of the **stack**. Let us refer to the node at the top of the stack as “current node”
4. Process current node value (e.g., print its value on screen)
5. If right child of current node exists, push right child to the top of the **stack**
6. If left child of current node exists, push left child to the top of the **stack**
7. Go to step 3

Let us code iterative tree DFS …




# Iterative Tree DFS (illustration)




*Pre-order* DFS traversal
**a b d e c f**

**Stack [ ]**: ***push*** onto end, ***pop*** off end

**DFS**: pop node, push its children, repeat

```
0 push a:   [a]
1 pop:      []       a
  push c:   [c]
  push b:   [c b]
2 pop:      [c]      b
  push e:   [c e]
  push d:   [c e d]
3 pop:      [c e]    d
4 pop:      [c]      e
5 pop:      []       c
  push f:   [f]
6 pop:      []       f
```




# Exercise (take away)

Modify the preorder iterative DFS algorithm such that it performs:

(1) Iterative inorder DFS traversals

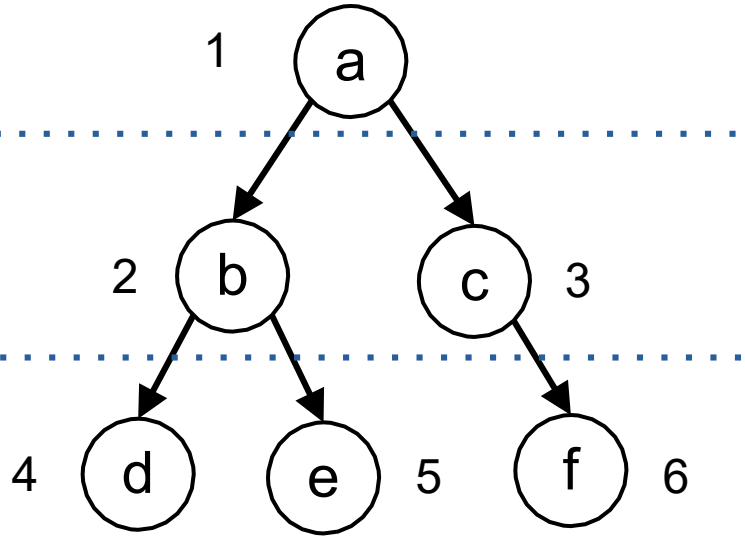
(2) Iterative postorder DFS traversals

# Iterative Tree BFS (pseudocode)

Essentially the same dynamics as DFS but using a **queue** instead of a stack

1. Create an empty **queue** of tree nodes
2. Enqueue the root node into the tail of the **queue**
3. If **queue** is empty, STOP. If **queue** not empty, dequeue the node at the head of the queue. Let us refer to the node at the top of the queue as “current node”
4. Process current node value (e.g., print its value on screen)
5. If left child of current node exists, enqueue left child to the top of the **queue**
6. If right child of current node exists, enqueue right child to the tail of the **queue**
7. Go to step 3

Let us code iterative tree BFS …




BFS traversal

**a b c d e f**

**Queue { }**: *enqueue* onto back, *dequeue* off front

**BFS**: dequeue node, enqueue it's children, repeat

```
0 enq a:    {a}
1 deq:      {}      a
  enq b:    {b}
  enq c:    {b c}
2 deq:      {c}     b
  enq d:    {c d}
  enq e:    {c d e}
3 deq:      {d e}   c
  enq f:    {d e f}
4 deq:      {e f}   d
5 deq:      {f}     e
6 deq:      {}      f
```

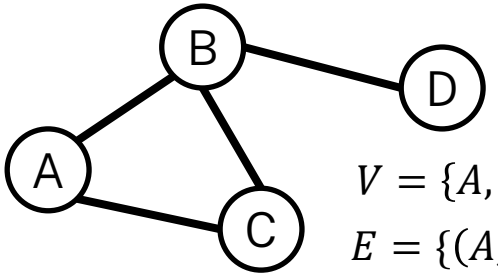



# Graphs (BFS and DFS)

Australian National University

# Graphs

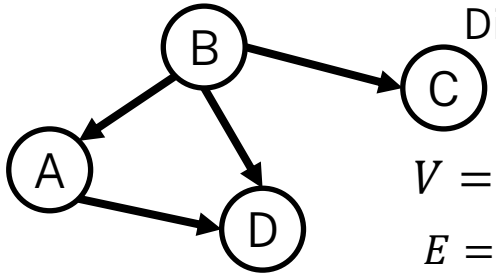
- A graph is a powerful abstraction in computing and mathematics which consists of a finite set of vertices $V$ (a.k.a. nodes) and a finite set of edges $E$ (a.k.a. links) interconnecting those vertices
- Just as trees, graphs can be either undirected (left) or directed (right)




Undirected graph

$$V = \{A, B, C, D\}$$

$$E = \{(A,B),(A,C),(B,A),(B,C),(B,D),(C,A),(C,B),(D,B)\}$$




Directed graph

$$V = \{A, B, C, D\}$$

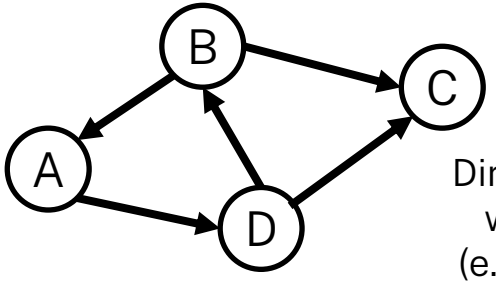
$$E = \{(A,D),(B,A),(B,C),(B,D)\}$$

- Vast array of applications: transportation networks, computer networks, epidemiology and disease spread, supply chain management, geographic information systems (GIS), bioinformatics, parallel computing, and a large etc.

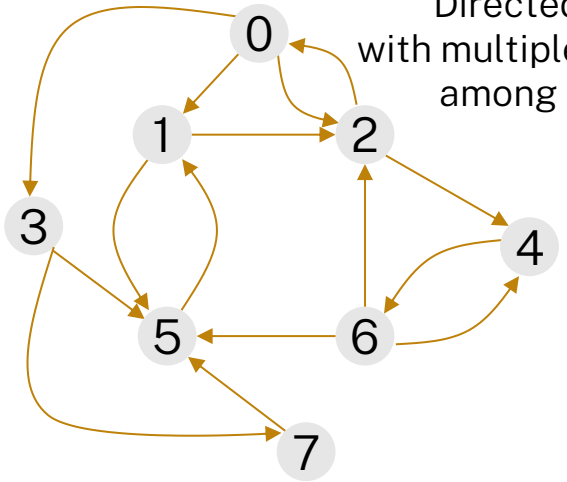



# Graphs and trees

- In contrast to trees, **graphs do not have a root vertex (node).** Thus, there is no natural node from which to start the traversals
- Besides, **nodes may have multiple parents or no parents at all**
- While in trees there is a single path from the root to any other node, **in a graph there might be multiple paths among a pair of nodes, or no paths at all** (graphs with at least a pair of nodes for which there is not a path are called disconnected)
- Graphs can have **cycles** while trees cannot. A cycle is a non-empty trail in which only the first and last vertices are equal. A trail is a path from two vertices with no repeated edge.

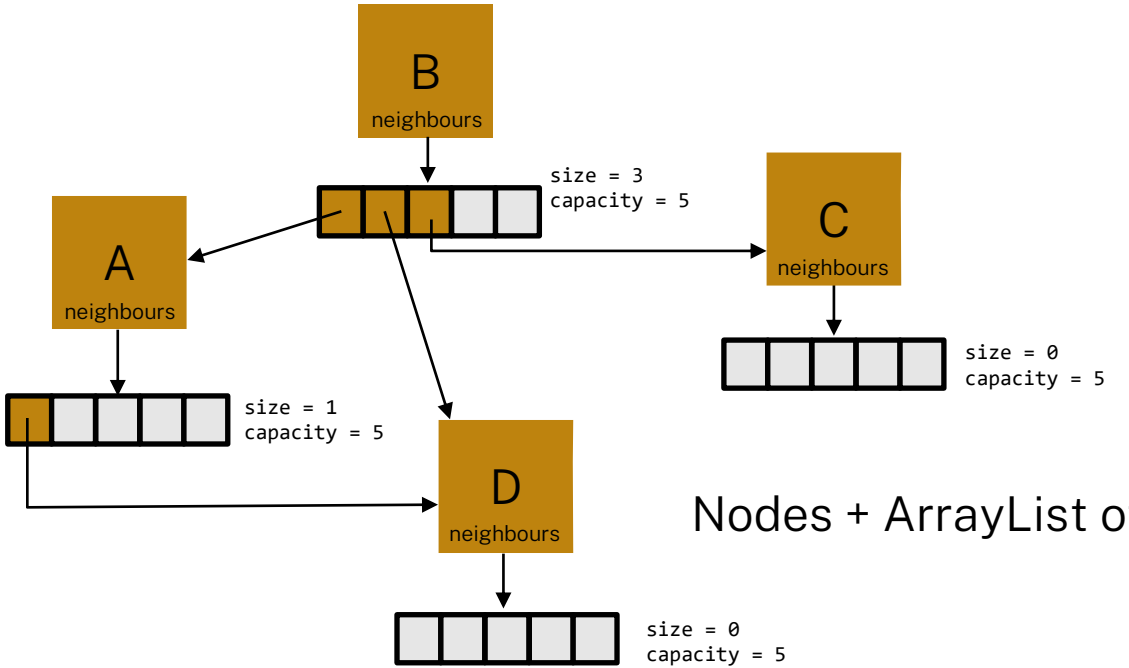


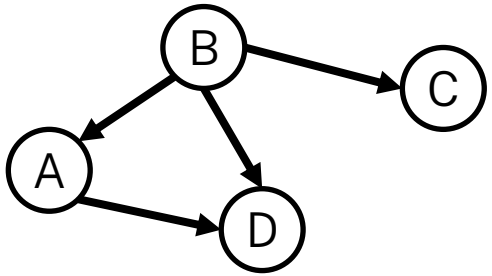

Directed graph with cycles (e.g., A, D, B, A)




Directed graph with multiple paths, e.g., among 0 and 5

# Graph implementations (some examples)




Nodes + ArrayList of references to neighbours

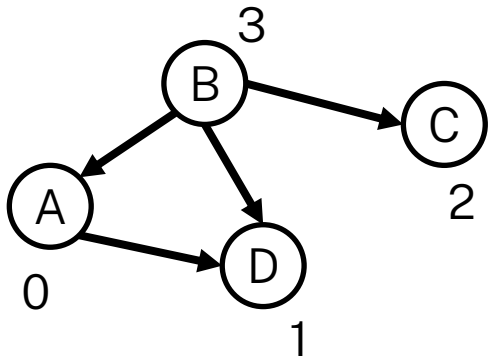
# Graph implementations (some examples)

vertex identifiers
of neighbours of vertex 3

```
[ [1], [], [], [0,1,2] ]
```

```
{ 0: “A”, 1: “D”, 2: “C”, 3: “B”}
```

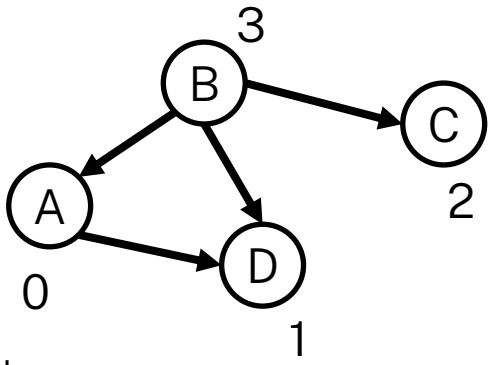
value associated to vertex 3




Adjacency lists with vertex identifiers + Map from vertex identifiers to values

# Graph implementations (some examples)

| Vertex ID | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | false | **true** | false | false |
| 1 | false | false | false | false |
| 2 | false | false | false | false |
| 3 | **true** | **true** | **true** | false |




adjacency matrix
A[i,j] is true iff there is an edge from vertex i to j

```
{ 0: "A", 1: "D", 2: "C", 3: "B"}
```

value associated to vertex 3

Adjacency matrix with vertex identifiers + Map from vertex identifiers to values



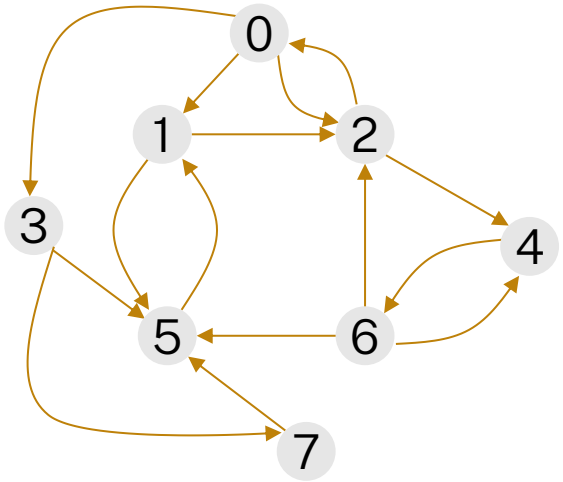

# Graph traversals (DFS and BFS)

- Just as with trees, there are multiple ways to traverse a graph (recap: to visit all nodes **exactly once** in a certain order)
- As with trees, two common traversals are DFS and BFS
- However, **two key differences**:
  1. One needs to select an **initial vertex** from which to start the traversal (typically depends on the problem to be solved)
  2. There is the need to **keep track of vertices that have been already been "touched"** (i.e., seen during the process) to avoid (1) visiting a node more than once; (2) cycling indefinitely (infinite loops)




# Iterative graph BFS (pseudocode)

1. Create an empty **queue** of vertices and an empty set of touched vertices
2. Enqueue the initial vertex at the tail of the **queue**, and add it to the set of touched vertices
3. If queue is empty, STOP. If **queue** not empty, dequeue the vertex at the head of the queue. Let us refer to the vertex at the head of the queue as “current vertex”
4. Process current vertex value (e.g., print its value on screen)
5. Enqueue all non-touched neighbours of current vertex at the tail of the **queue**, and add them to the set of touched nodes
6. Go to step 3

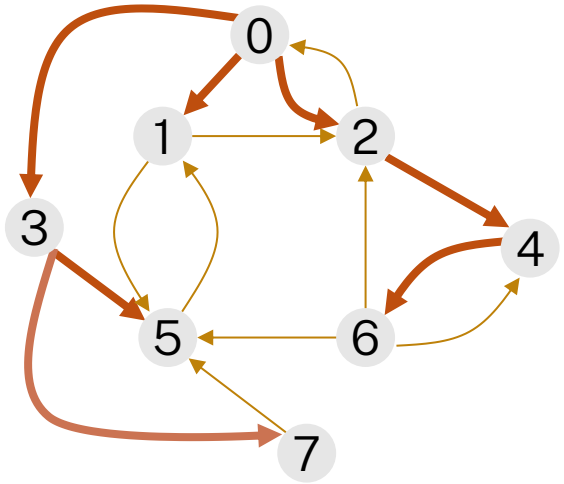



Very nice visualizations of the algorithm for different graphs can be seen [here]

Source: Prof. Galles at USFCA

# Shortest paths problem

- Given a vertex, find shortest paths from that vertex to every other vertex in the graph
- **The length (a.k.a. cost) of a path among two vertices is the number of edges in the path**
- Picture below shows shortest paths from vertex 0 to every other vertex (this forms a tree called the shortest-path tree)




BFS is very well suited to find shortest path lengths as it visits the vertices in depth order. This ensures that shortest path to any node is found first than any other longer path

Let us code iterative BFS ...

# Iterative graph DFS (pseudocode)

Essentially the same dynamics as BFS but using a **stack** instead of a queue

1. Create an empty **stack** of vertices and an empty set of touched vertices
2. Push the initial vertex into the top of the **stack**, and add it to the set of touched vertices
3. If stack is empty, STOP. If stack not empty, pop from **stack** the vertex at the top of the **stack**. Let us refer to the vertex at the top of the stack as “current vertex”
4. Process current vertex value (e.g., print its value on screen)
5. Push all non-touched neighbours of current vertex to the top of the **stack**, and add them to the set of touched vertices
6. Go to step 3

Let us code iterative DFS …

# Exercise (take-away)

Implement DFS graph traversal using recursion




# Weighted Graphs (Dijkstra algorithm)

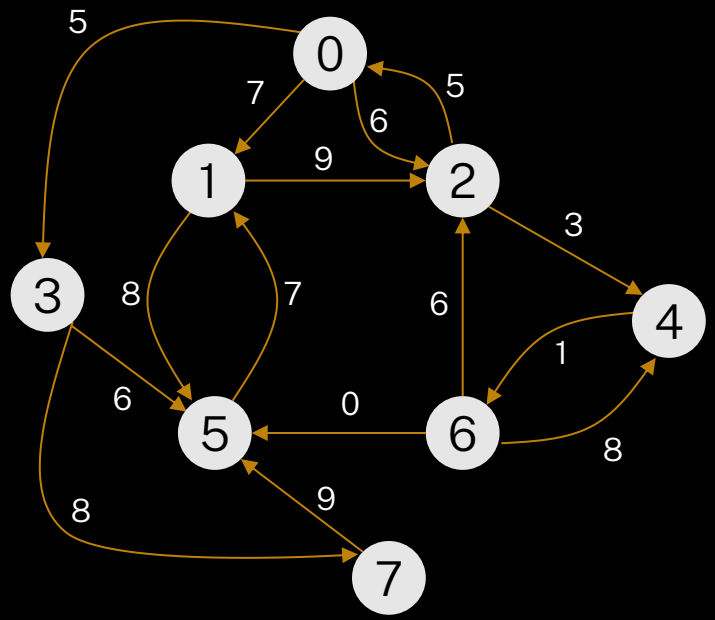
[Distinction-Level Content]



Australian National University

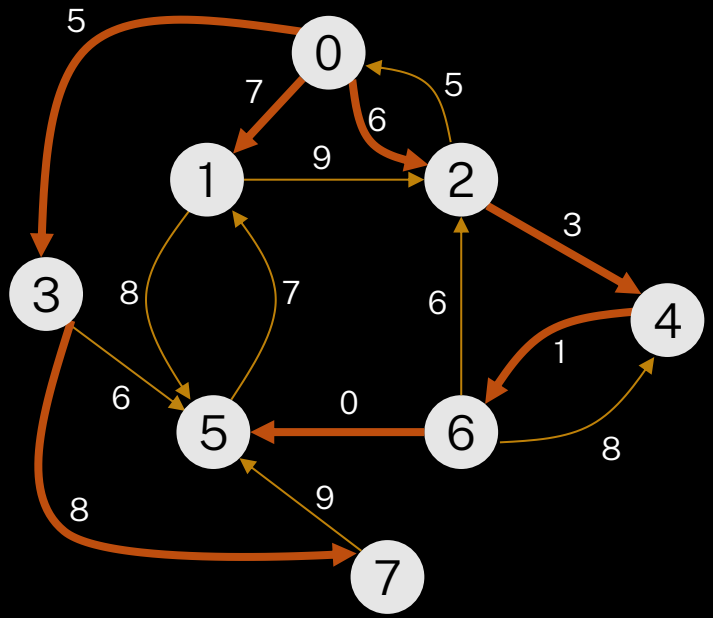
# Weighted graphs

- A weighted graph is a special kind of graph in which every edge is associated a weight
- The weight associated to an edge is typically a non-negative number (e.g., `int` or `float`) that models the cost of travelling across an edge (in the direction of the edge)
- For example, in the graph below, the weight associated to edge connecting vertex 0 and 2 is 6



# Shortest paths problem (weighted graphs)

- Given a vertex, find shortest paths from that vertex to every other vertex in the graph
- The **length** (a.k.a. cost) of a path among two vertices is the sum of weights of the edges in the path
- Picture and table below show shortest paths from vertex 0 to every other vertex (this forms a tree called the shortest-path tree)




| Vertex | Shortest path from vertex 0 | Length of shortest path |
|---|---|---|
| 0 | empty path | 0 |
| 1 | $0 \to 1$ | 7 |
| 2 | $0 \to 2$ | 6 |
| 3 | $0 \to 3$ | 5 |
| 4 | $0 \to 2 \to 4$ | 9 |
| 5 | $0 \to 2 \to 4 \to 6 \to 5$ | 10 |
| 6 | $0 \to 2 \to 4 \to 6$ | 10 |
| 7 | $0 \to 3 \to 7$ | 13 |

# Dijkstra algorithm for shortest paths problem (pseudocode)

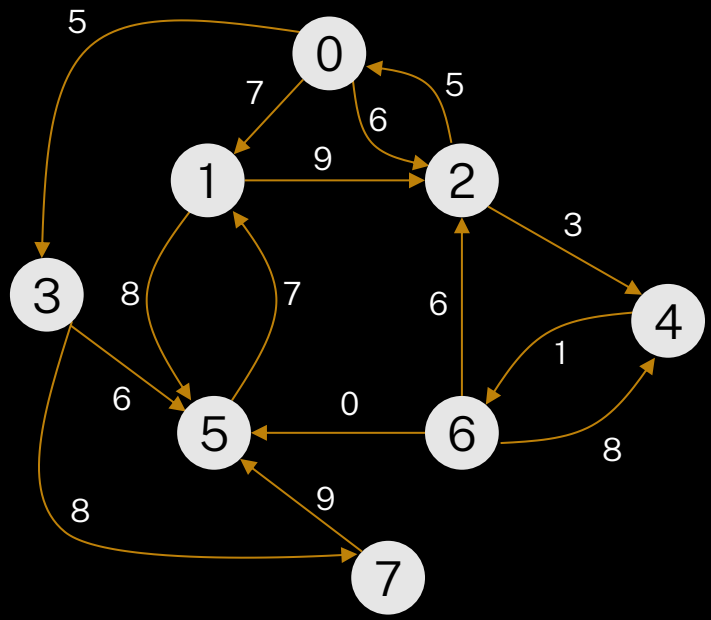
1. Create a set of vertices for which we do not know yet a shortest path from the start vertex. Put all vertices in the graph into this set. Let us refer to this set as the “unknown vertices set”
2. Assign to every vertex in the graph an initial path length. In particular, the initial vertex is assigned an initial path length of 0, and every other vertex different from the initial vertex, an initial path length of ∞ (infinity)
3. Select from the “unknown vertices set” the vertex with the smallest path length. Let us refer to this vertex as to the “current vertex”. If the “unknown vertices set” is empty, or it only contains vertices with infinite path lengths, STOP
4. For all neighbours of the current vertex which are in the “unknown vertices set”, calculate the length of the path from the start vertex to the neighbour through current vertex. If the length of this path is smaller than the one currently known for that neighbour, update the length of the path for the neighbour. Otherwise, keep its current length
5. Remove the current vertex from the “unknown vertices set”
6. Go to step 3




# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

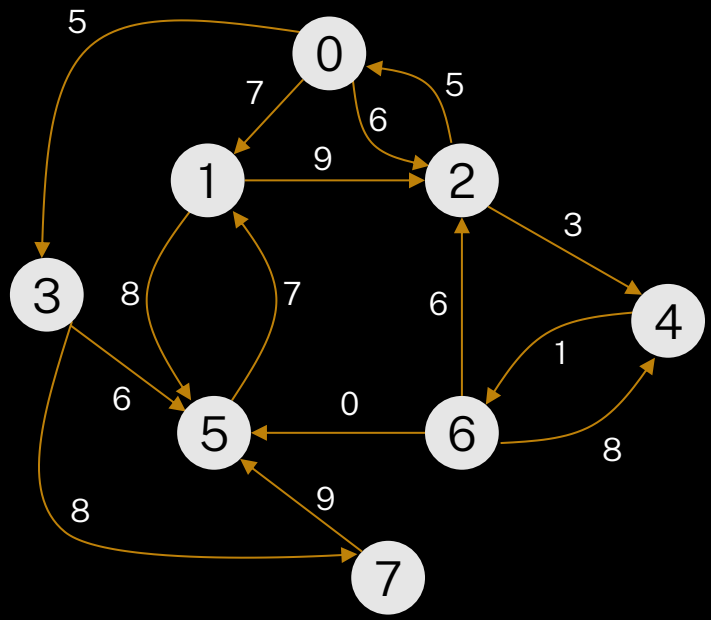
| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |



# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

find smallest path length from vertices in unknown vertices set

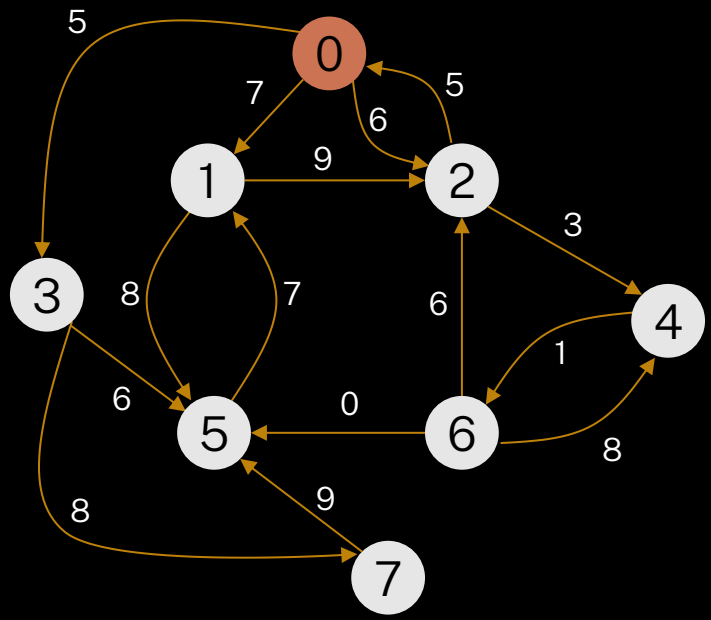



| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

Vertex 0 found



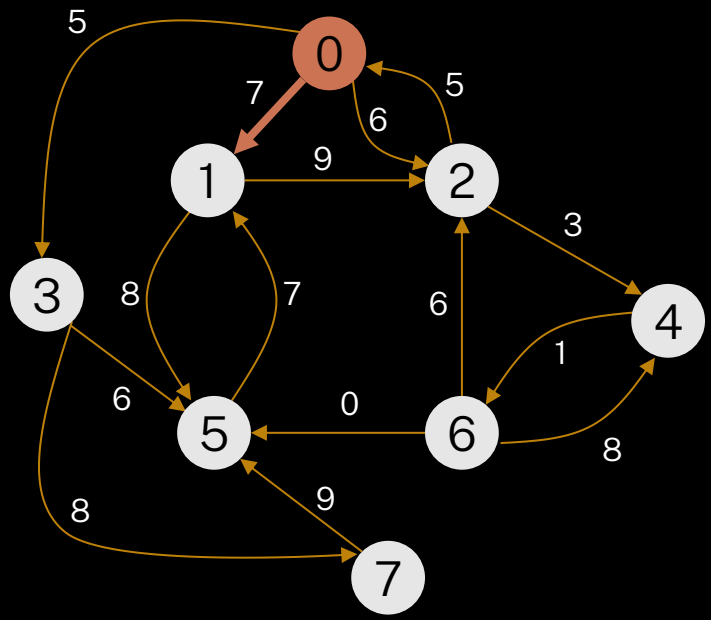

| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | ∞ | -1 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

updating path lengths of neighbours of vertex 0

0+7 < ∞ ?



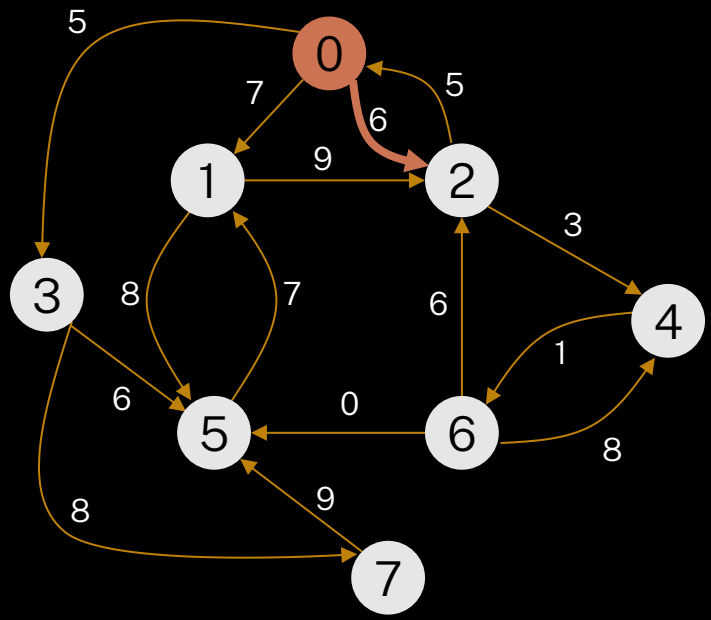

| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | ∞ | -1 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

updating path lengths of neighbours of vertex 0

0+6 < ∞ ?



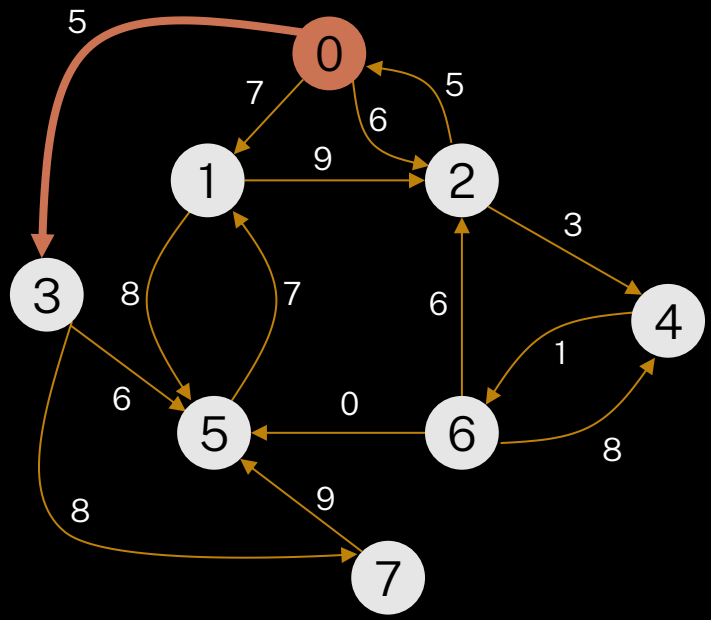

| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | ∞ | -1 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

updating path lengths of neighbours of vertex 0

0+5 < ∞ ?

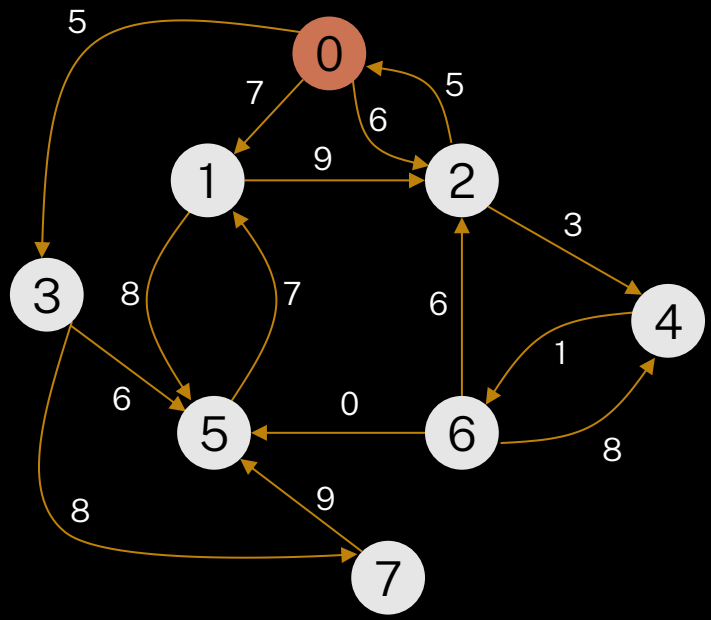



| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

remove vertex 0 from the unknown vertices set

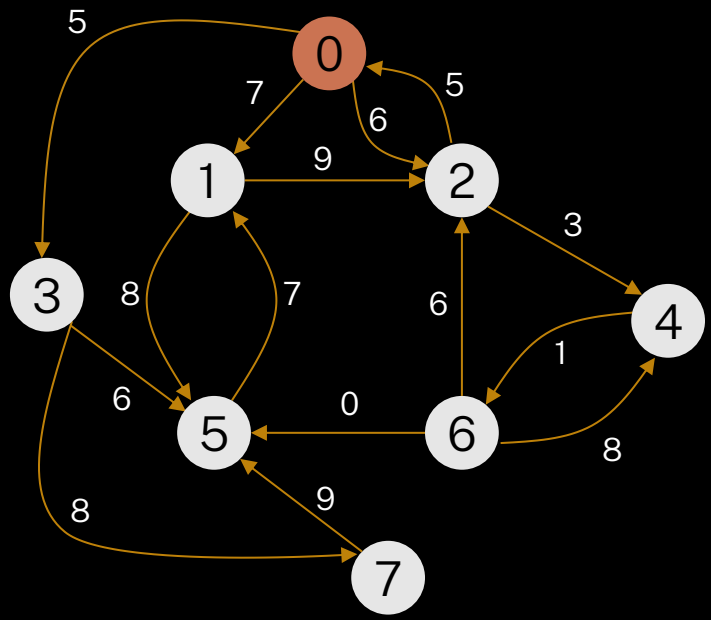



| Unknown vertices set |
|---|
| 0, 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

find smallest path length from vertices in unknown vertices set

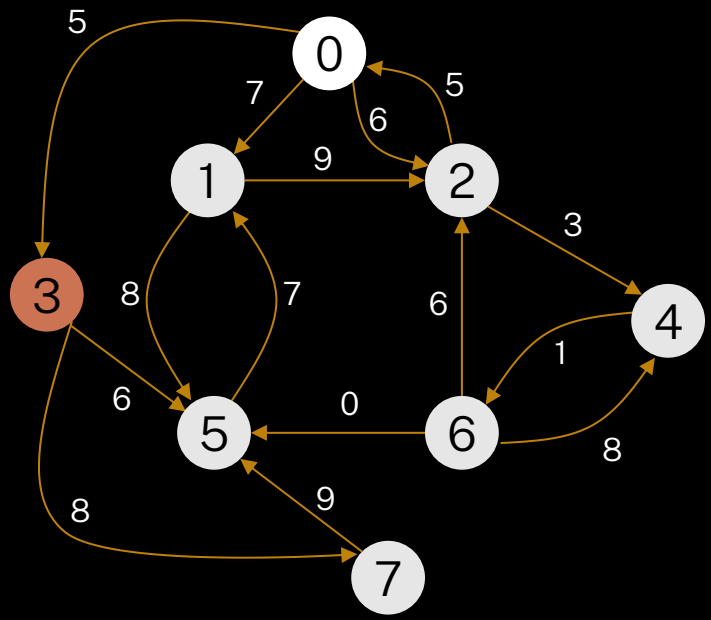



| Unknown vertices set |
|---|
| 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

vertex 3 found



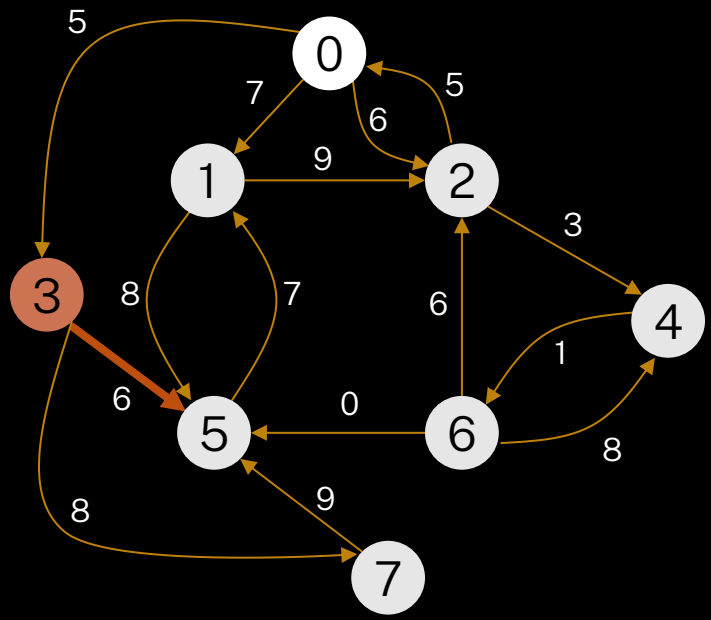

| Unknown vertices set |
|---|
| 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | ∞ | -1 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

updating path lengths of neighbours of vertex 3

5+6 < ∞ ?



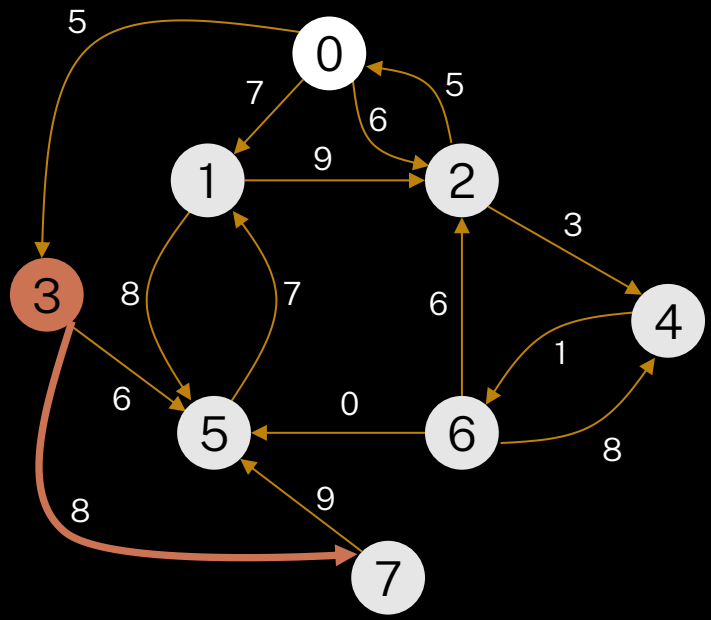

| Unknown vertices set |
|---|
| 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | ∞ | -1 |

updating path lengths of neighbours of vertex 3

5+8 < ∞ ?



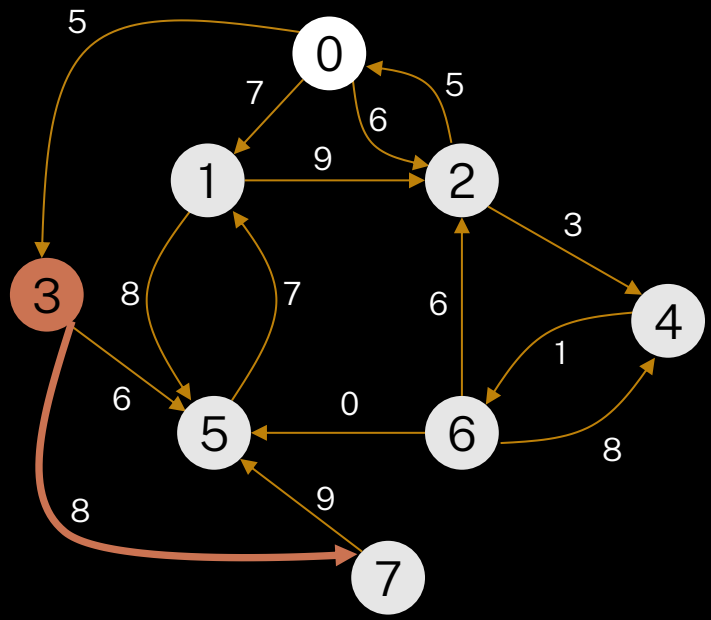

| Unknown vertices set |
|---|
| 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 3

5+8 < ∞ ?

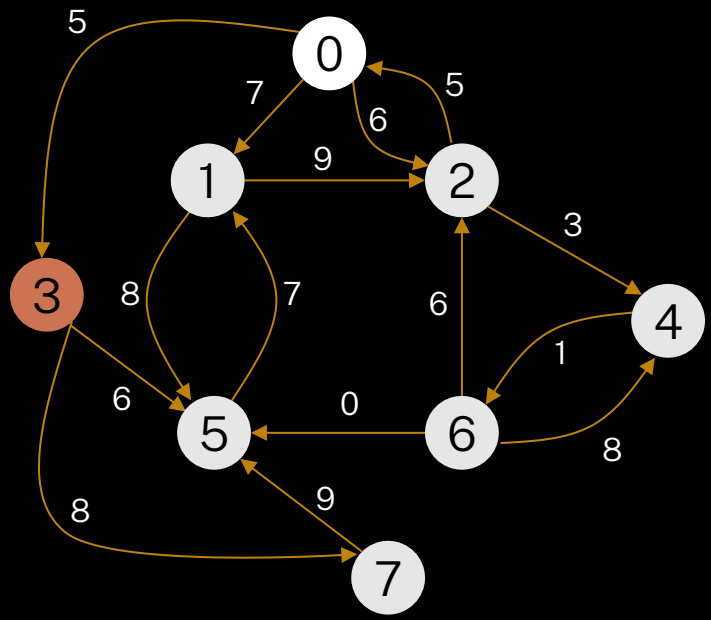



| Unknown vertices set |
|---|
| 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

remove vertex 3 from the unknown vertices set

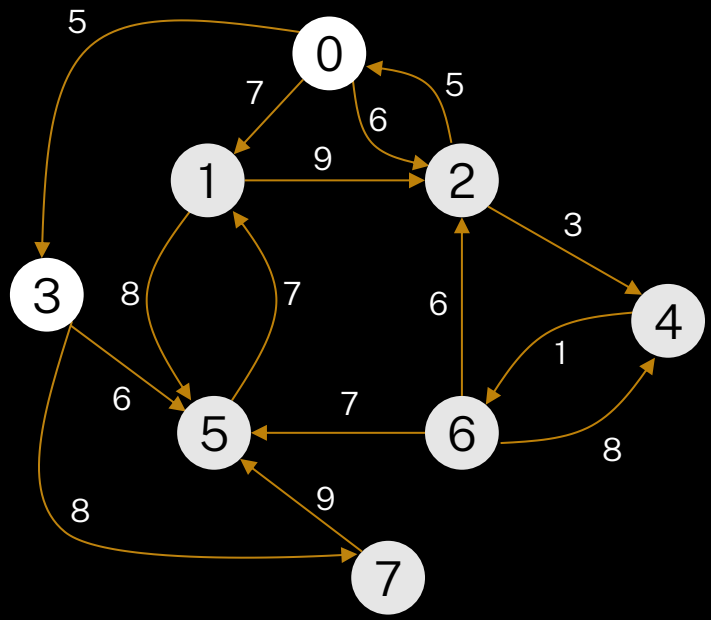



| Unknown vertices set |
|---|
| 1, 2, 3, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

find smallest path length from vertices in unknown vertices set

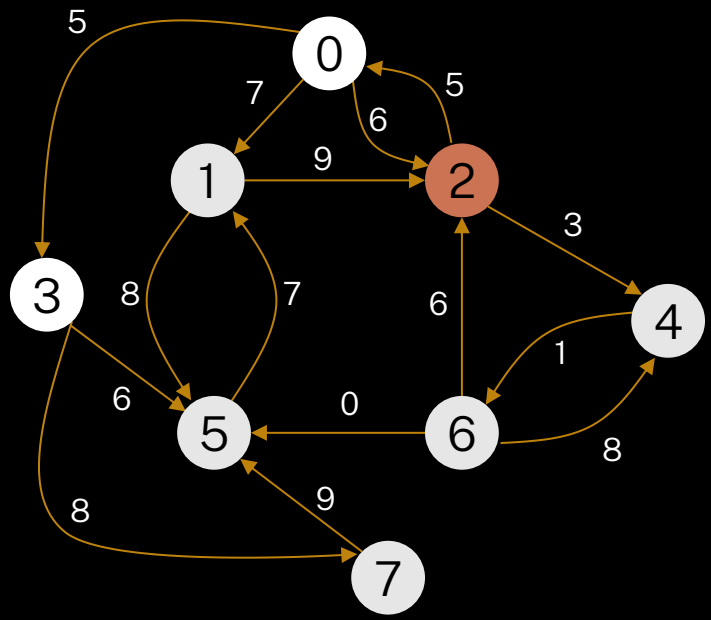



| Unknown vertices set |
|---|
| 1, 2, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

vertex 2 found



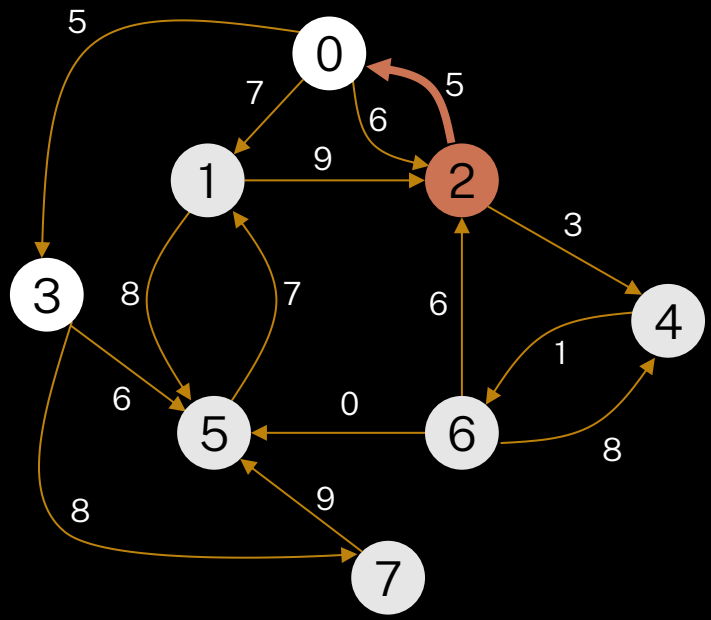

| Unknown vertices set |
|---|
| 1, 2, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 2

vertex 0 already known (do nothing)



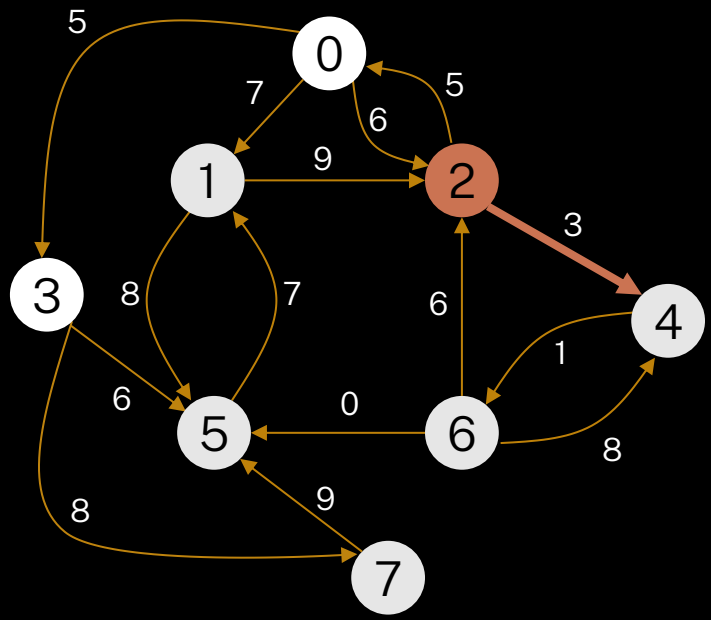

| Unknown vertices set |
|---|
| 1, 2, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | -1 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 2

6+3 < ∞ ?

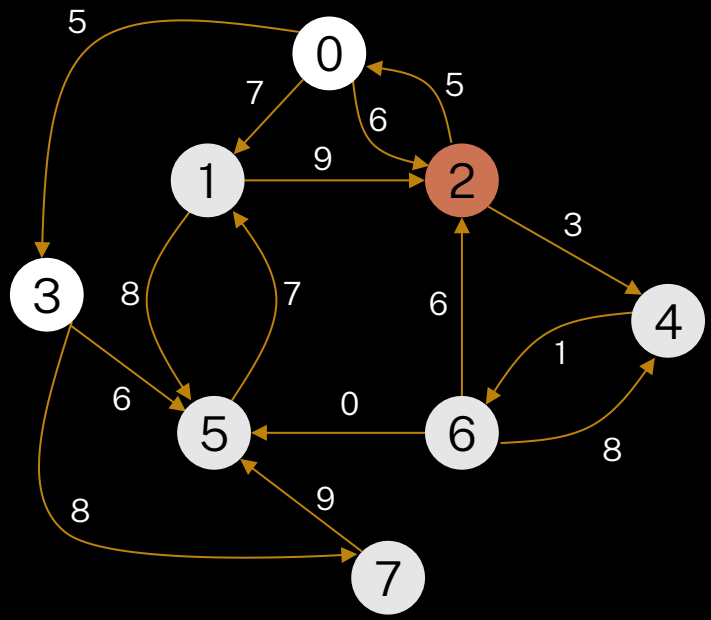



| Unknown vertices set |
|---|
| 1, 2, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

remove vertex 2 from the unknown vertices set

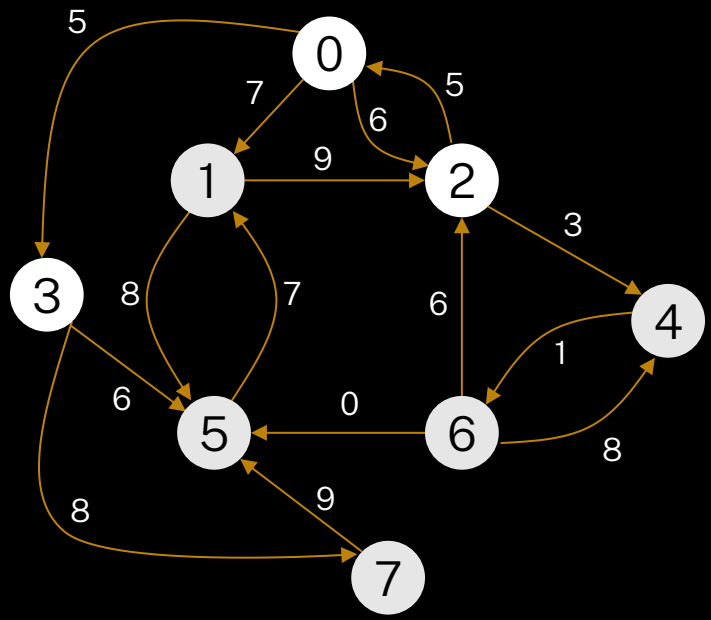



| Unknown vertices set |
|---|
| 1, 2, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

find smallest path length from vertices in unknown vertices set

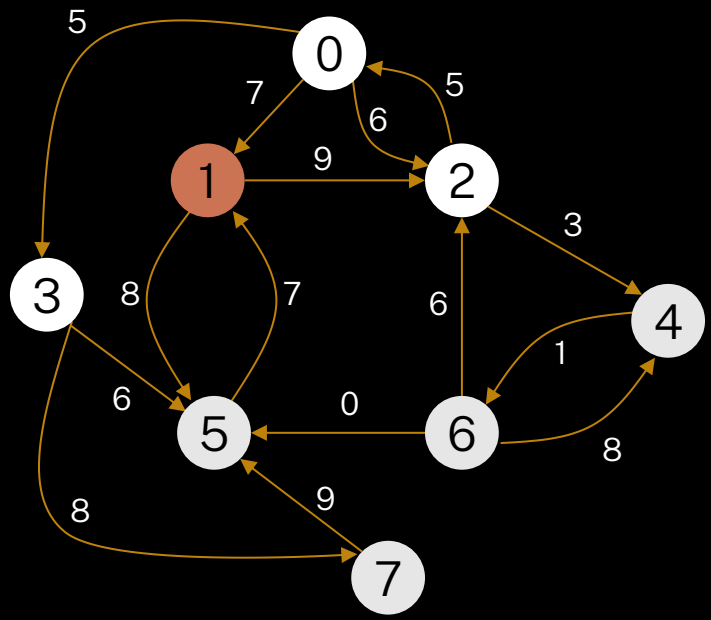



| Unknown vertices set |
|---|
| 1, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

vertex 1 found



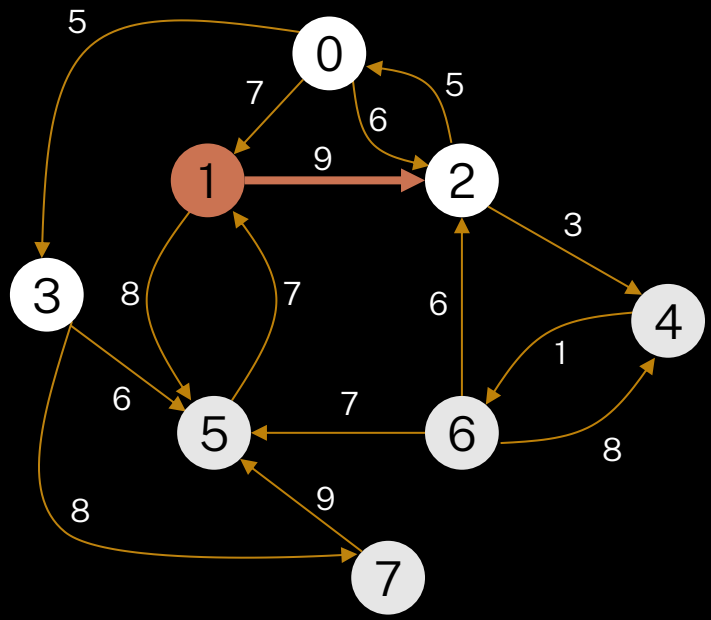

| Unknown vertices set |
|---|
| 1, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 1

vertex 2 already known (do nothing)



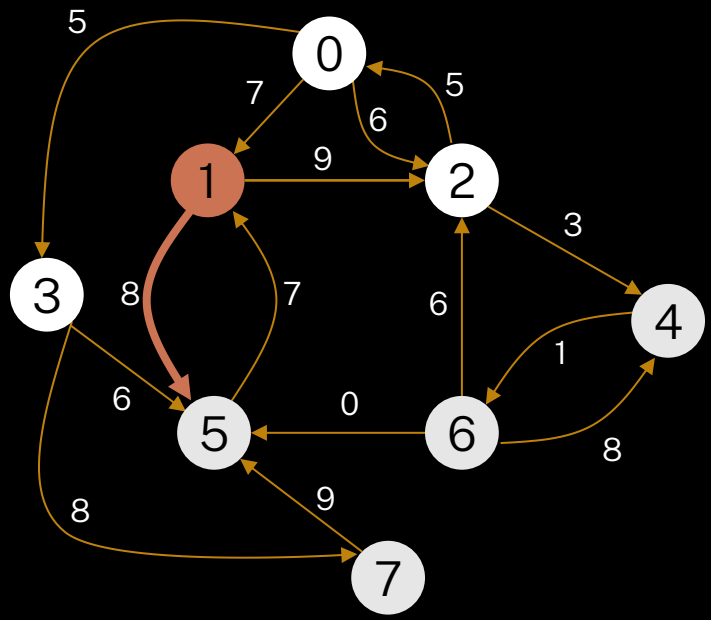

| Unknown vertices set |
|---|
| 1, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 1

7+8 < 11 ?

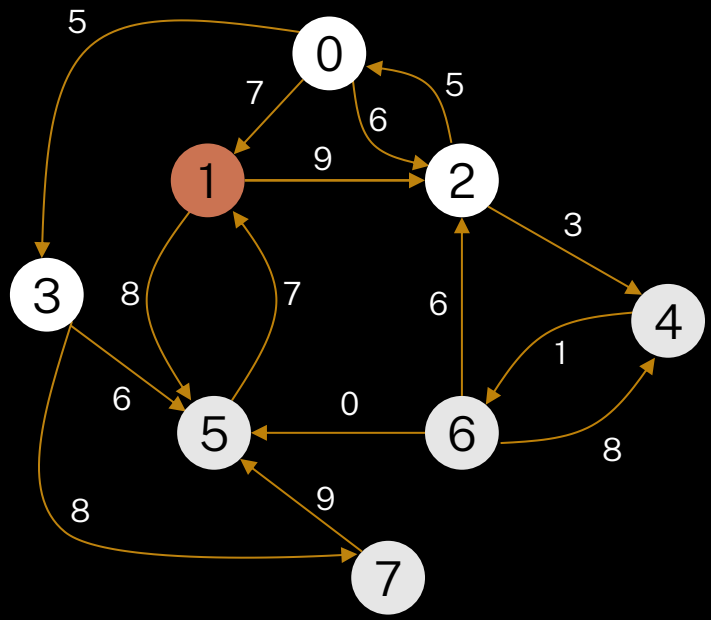



| Unknown vertices set |
|---|
| 1, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

remove vertex 1 from the unknown vertices set

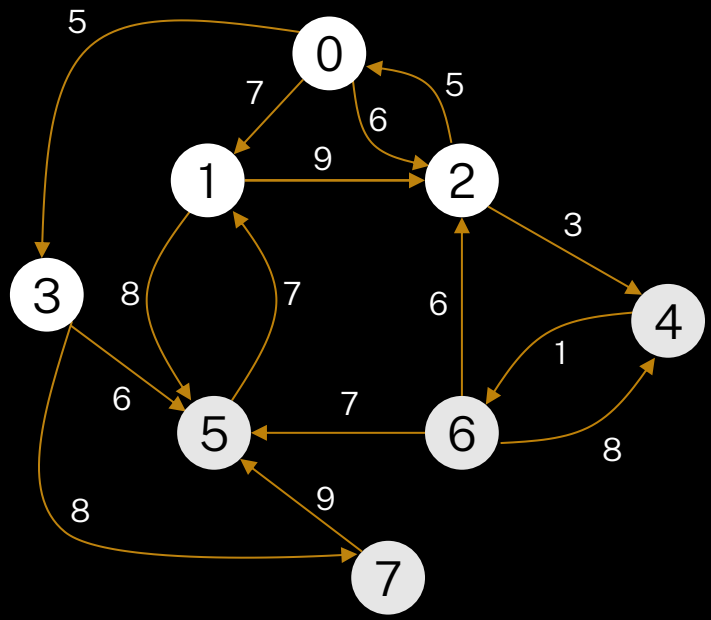



| Unknown vertices set |
|---|
| 1, 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

find smallest path length from vertices in unknown vertices set

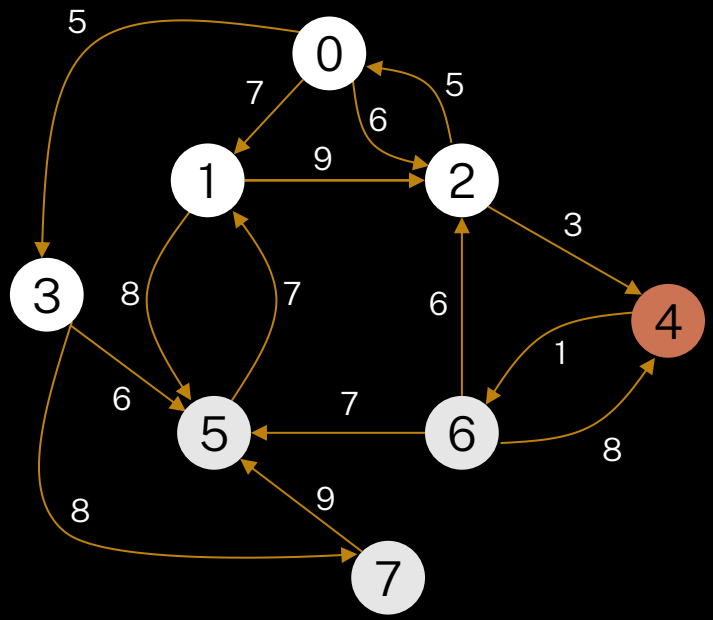



| Unknown vertices set |
|---|
| 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

Vertex 4 found



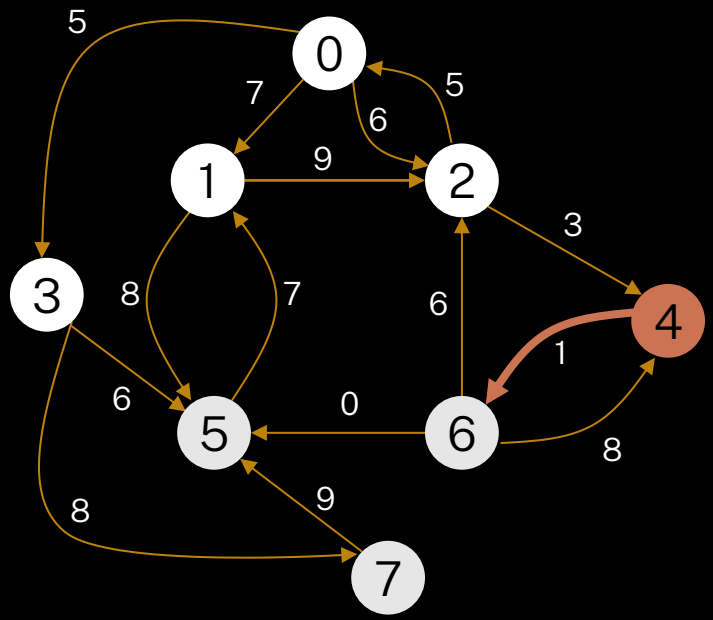

| Unknown vertices set |
|---|
| 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | ∞ | -1 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 4

9+1 < ∞ ?

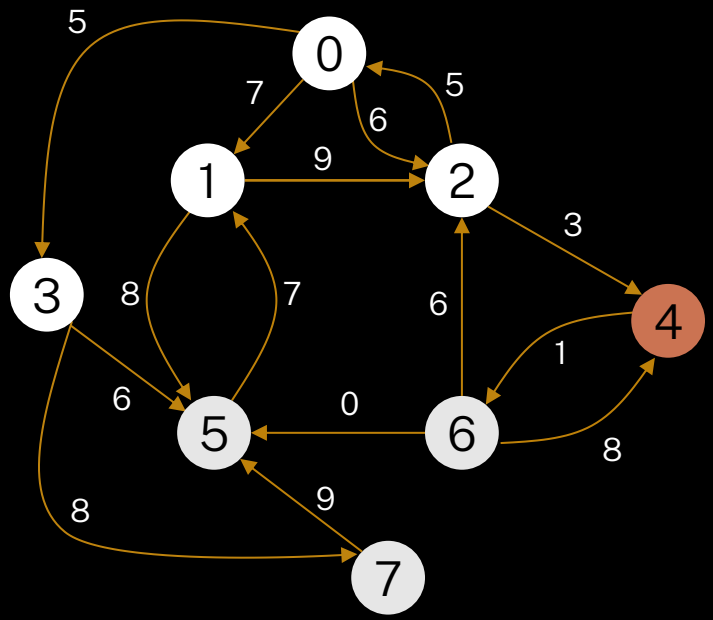



| Unknown vertices set |
|---|
| 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

remove vertex 4 from the unknown vertices set

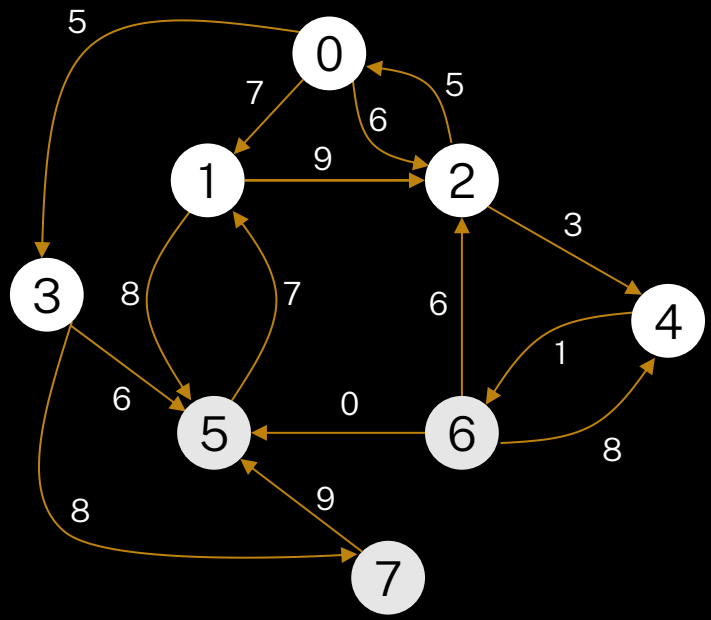



| Unknown vertices set |
|---|
| 4, 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

find smallest path length from vertices in unknown vertices set

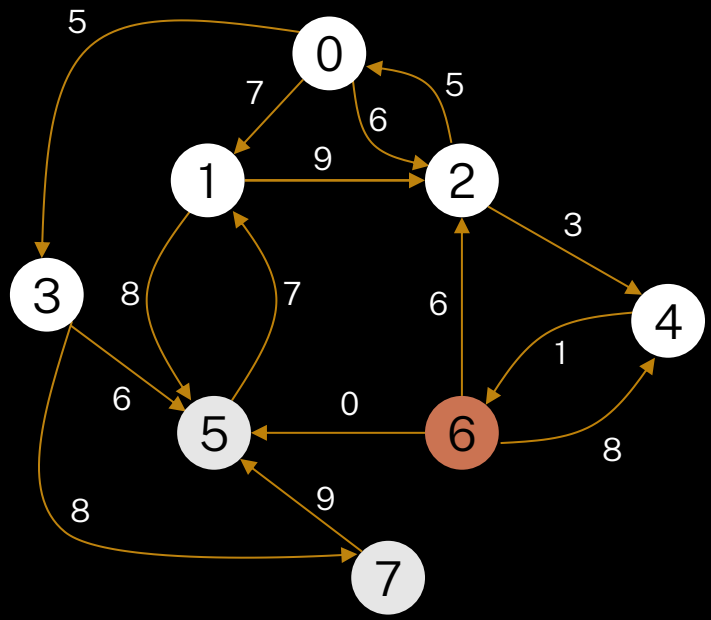



| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

Vertex 6 found



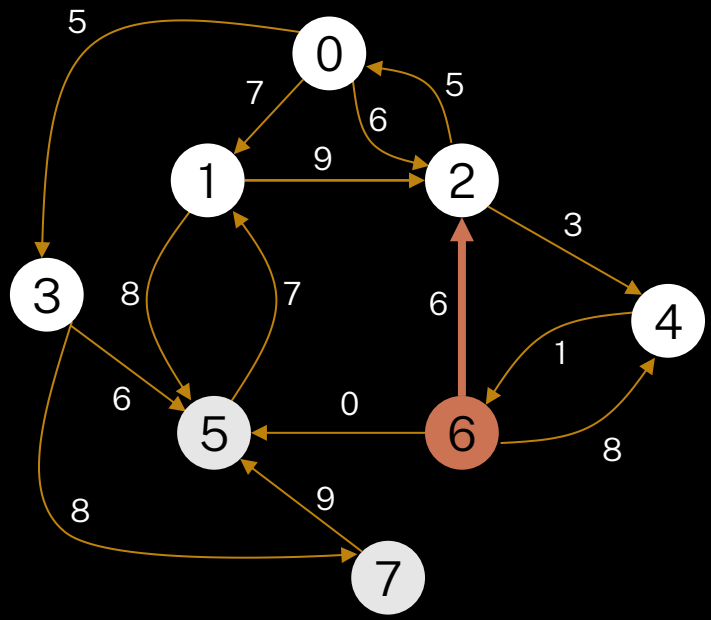

| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 6

vertex 2 already known (do nothing)



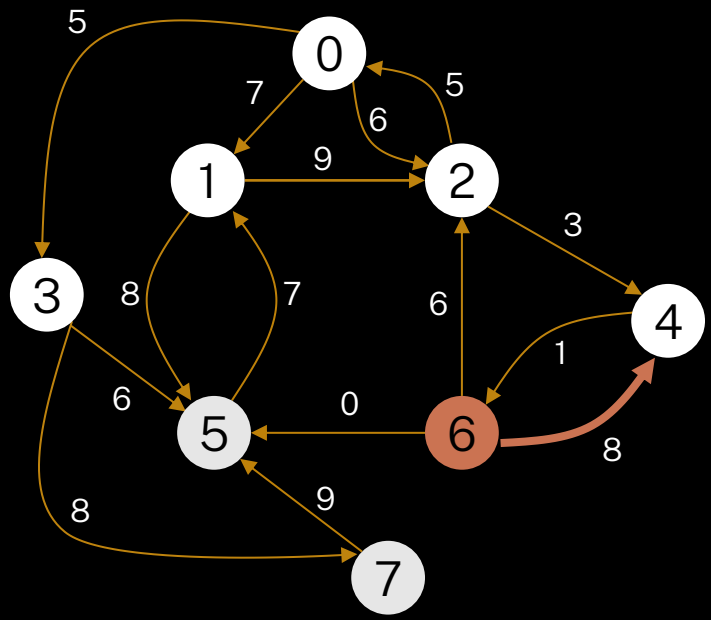

| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 6

vertex 4 already known (do nothing)



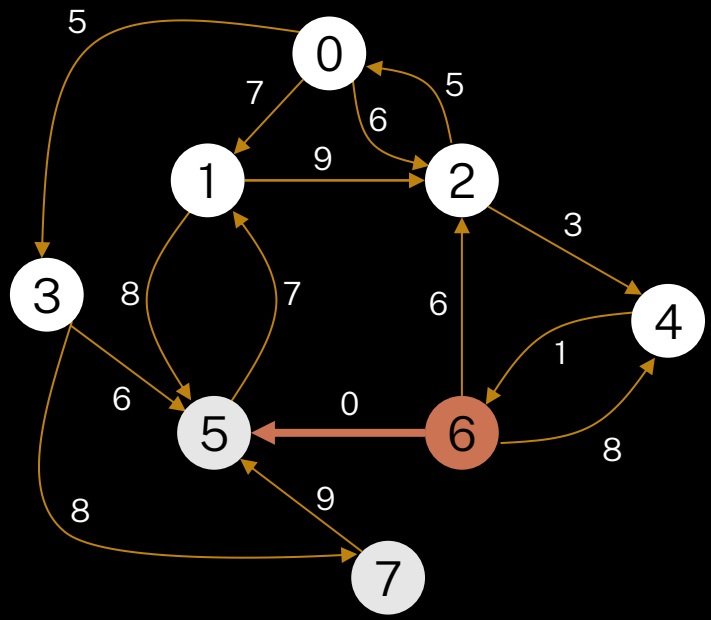

| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 11 | 3 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 6

10+0 < 11 ?

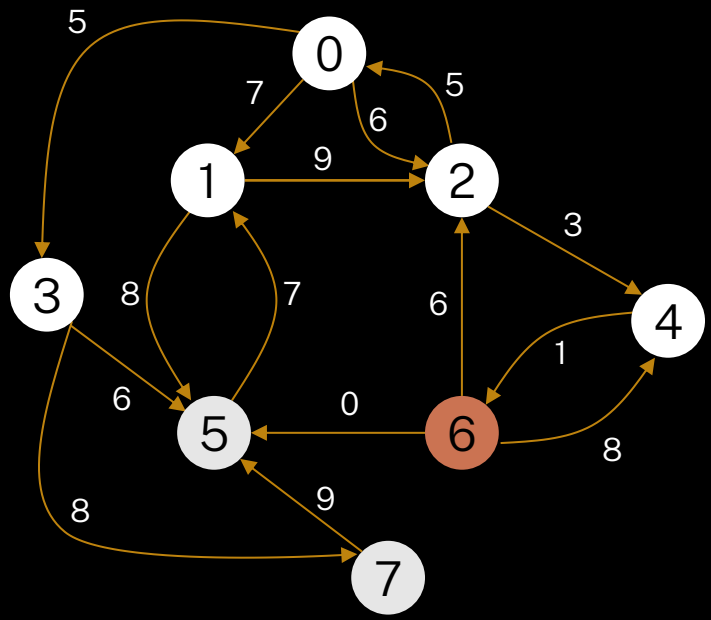



| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

remove vertex 6 from the unknown vertices set

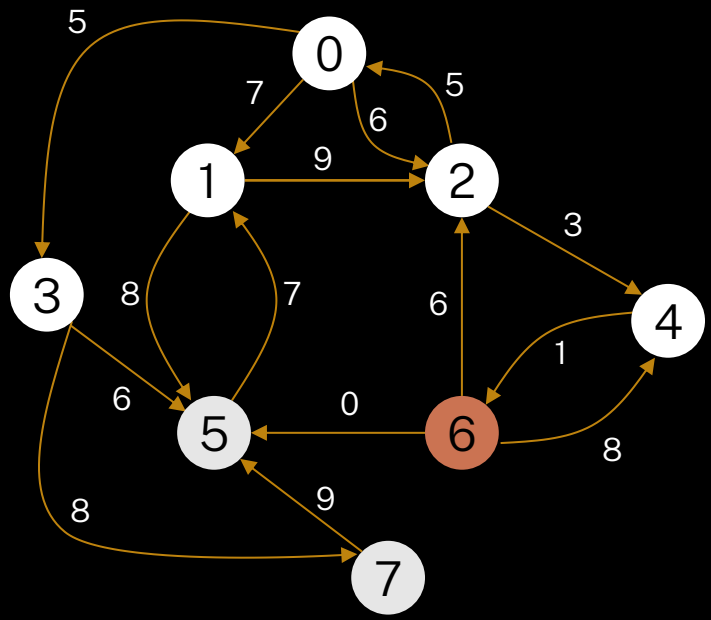



| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

remove vertex 6 from the unknown vertices set

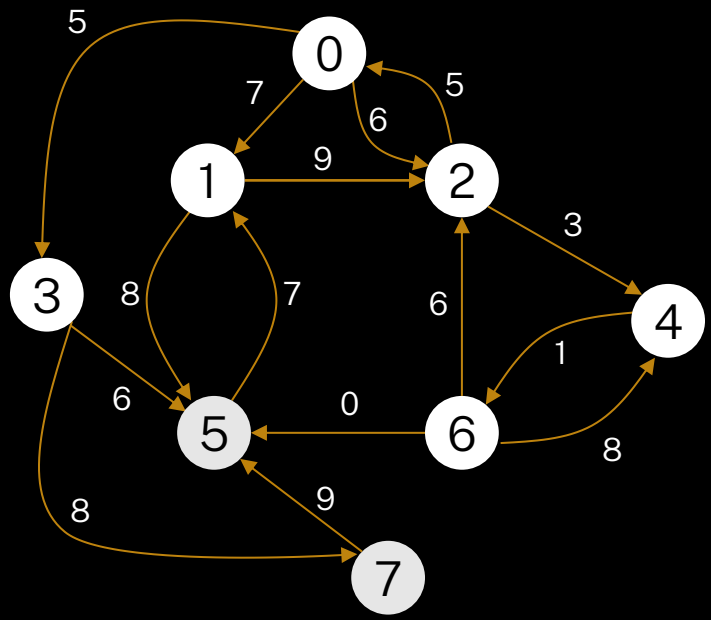



| Unknown vertices set |
|---|
| 5, 6, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

find smallest path length from vertices in unknown vertices set

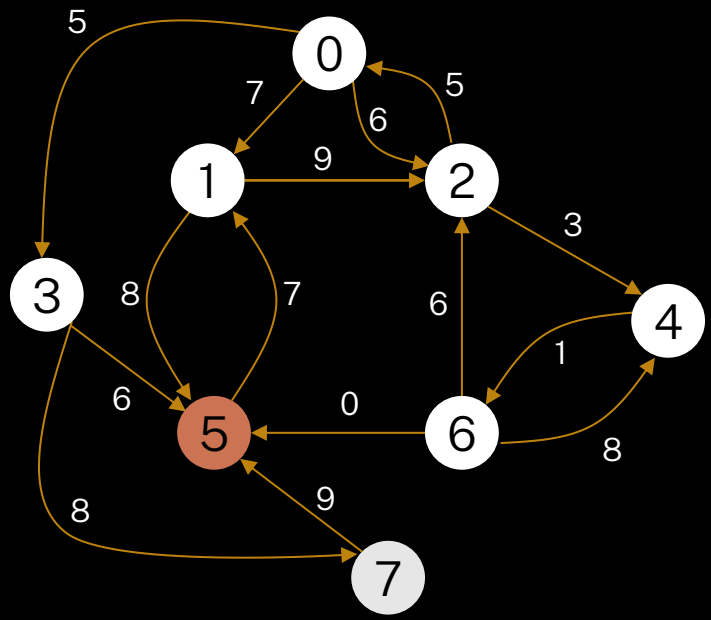



| Unknown vertices set |
|---|
| 5, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

vertex 5 found



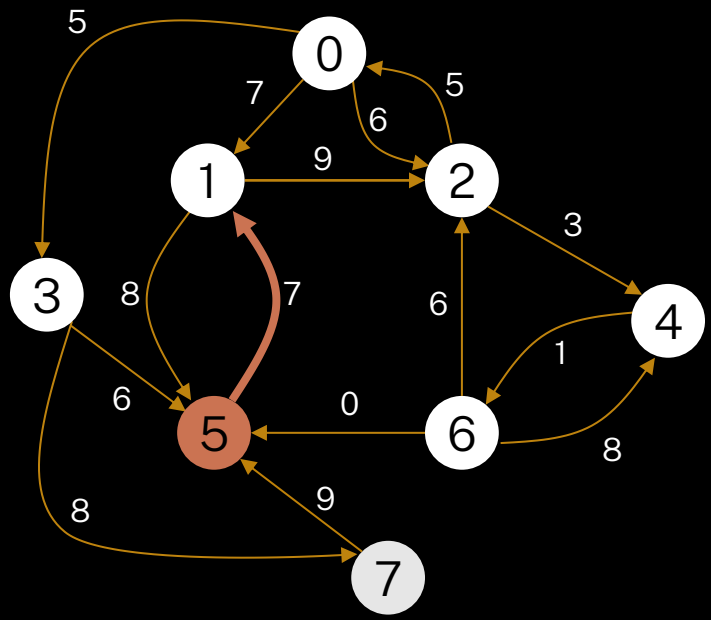

| Unknown vertices set |
|---|
| 5, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 5

vertex 1 already known (do nothing)

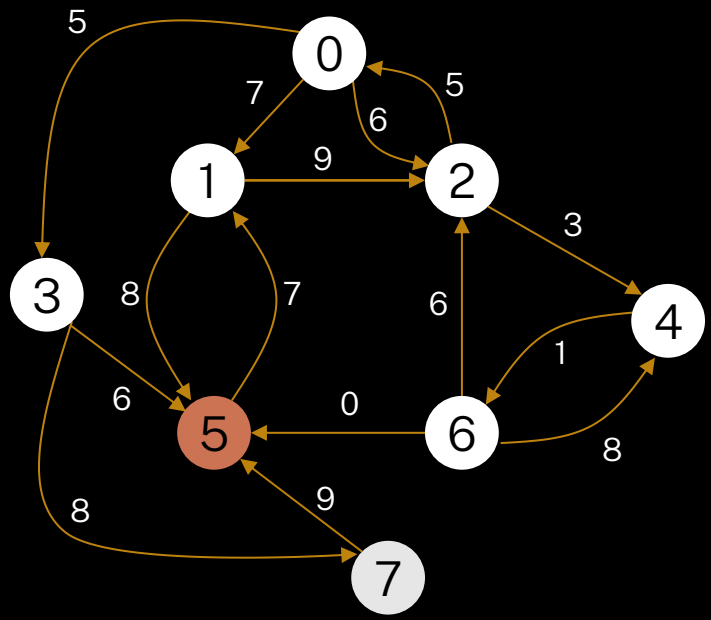



| Unknown vertices set |
|---|
| 5, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

remove vertex 5 from the unknown vertices set

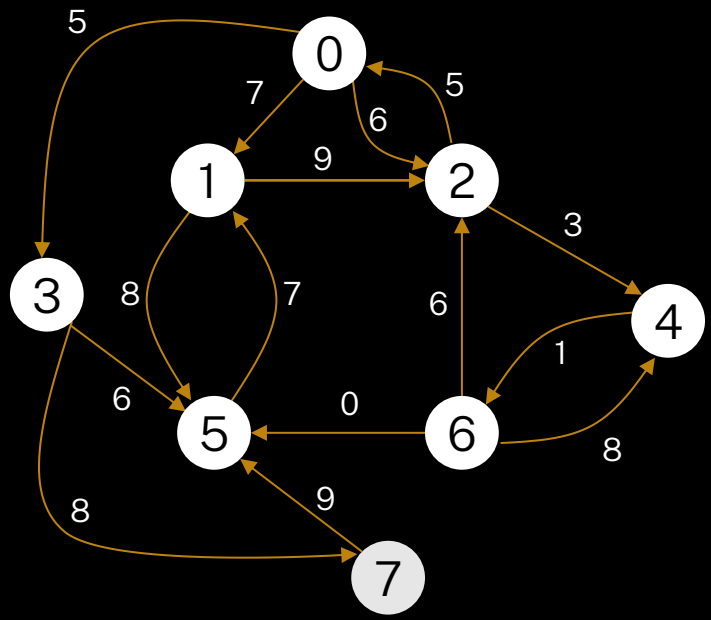



| Unknown vertices set |
|---|
| 5, 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

find smallest path length from vertices in unknown vertices set

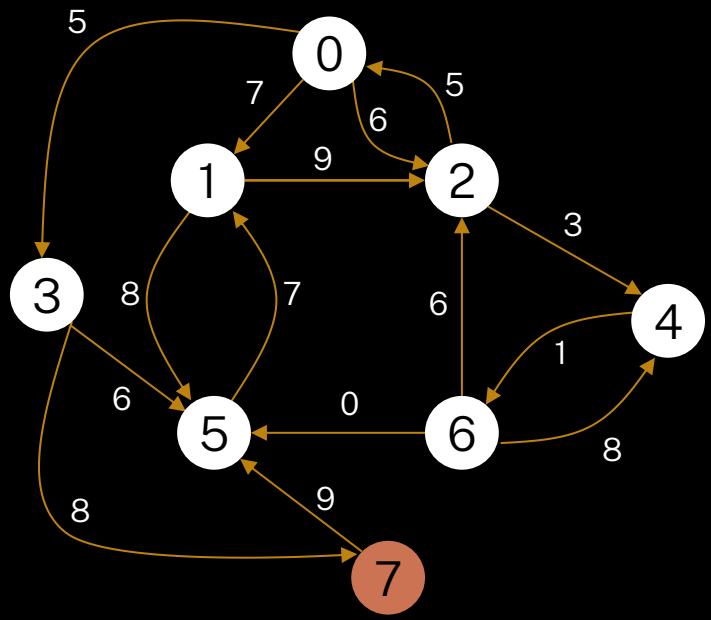



| Unknown vertices set |
|---|
| 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

vertex 7 found



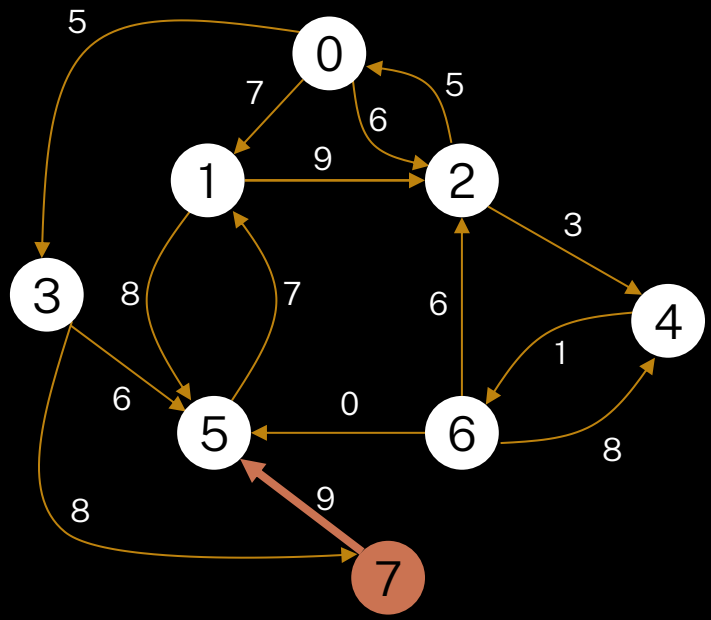

| Unknown vertices set |
|---|
| 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

updating path lengths of neighbours of vertex 7

vertex 5 already known (do nothing)

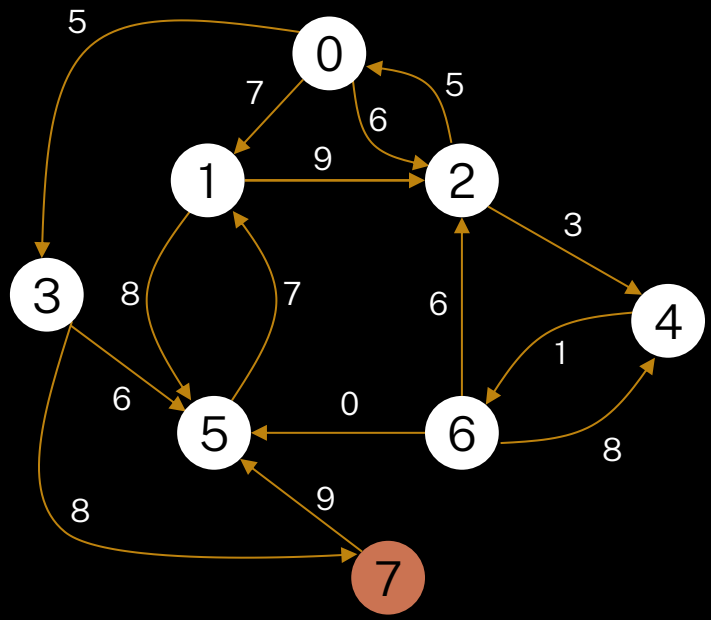



| Unknown vertices set |
|---|
| 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

remove vertex 7 from the unknown vertices set

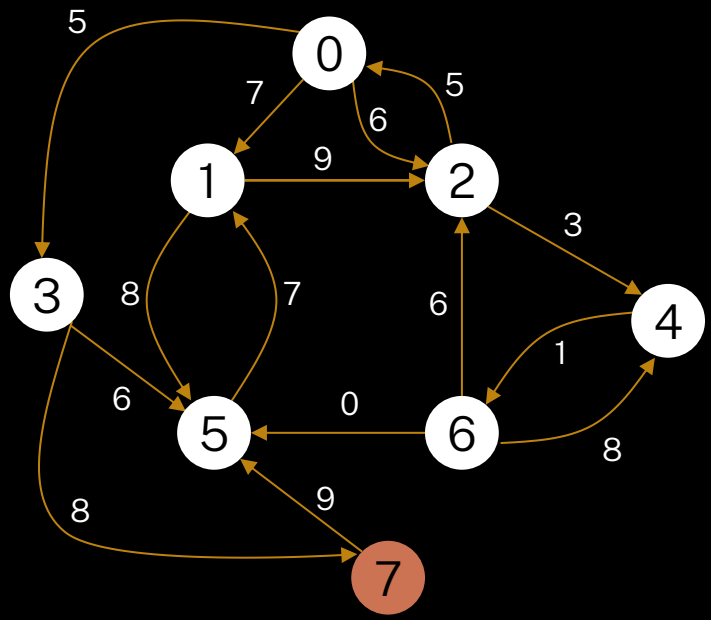



| Unknown vertices set |
|---|
| 7 |

# Dijkstra algorithm (step by step)

| Vertex | Path length from vertex 0 | Previous vertex in path |
|---|---|---|
| 0 | 0 | -1 |
| 1 | 7 | 0 |
| 2 | 6 | 0 |
| 3 | 5 | 0 |
| 4 | 9 | 2 |
| 5 | 10 | 6 |
| 6 | 10 | 4 |
| 7 | 13 | 3 |

Unknown vertices set is empty → STOP




| Unknown vertices set |
|---|
| |