Graphs

Helen Cameron

Comp 2140 Fall 2006

(Chapter 14)

Introduction

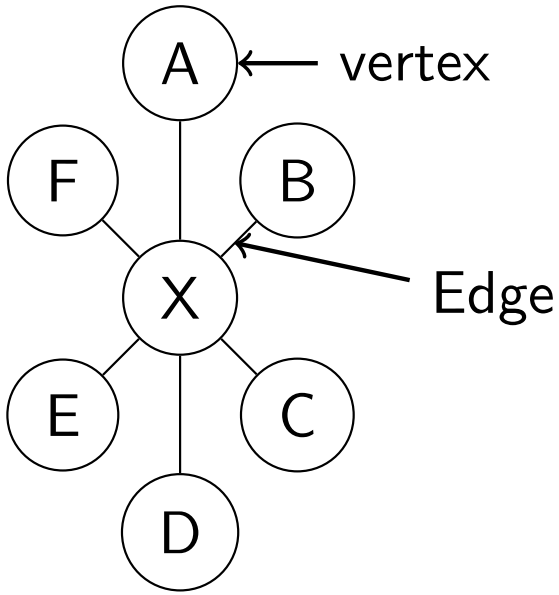
- Linked lists are a special case of trees (the worst kind of trees).
- Trees are a special case of a more general data structure: a graph.
- Graphs are used when complex relationships among data must be represented:
  - Networks.
  - Schedules.

Graph

Definition

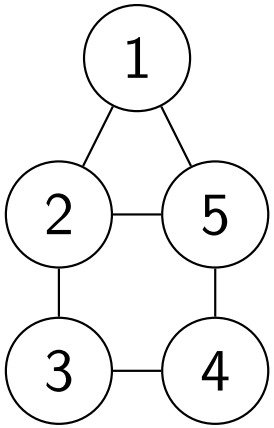
A graph consists of two sets V and E, where

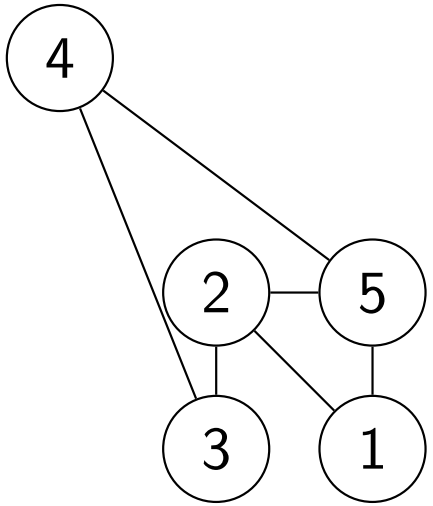
- V is a set of vertices $\{v_0, v_1, \ldots, v_{k-1}\}$, and
- E is a set of edges (pairs of vertices) $\{e_0, e_1, \ldots, e_{n-1}\}$, where each edge is a pair of vertices: $e_h = \langle v_i, v_j \rangle$ means that vertex v_i and vertex v_j are directly connected.



Drawing a Graph

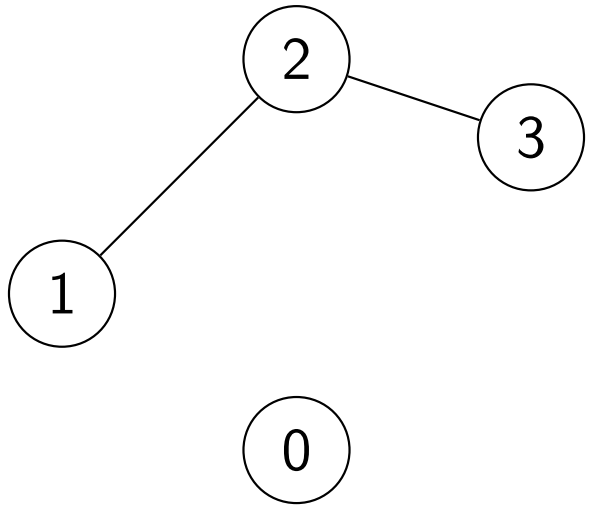
Note that how you draw a graph is not important. For example, these two graphs are the same graph:





Edges and Adjacency

Two vertices are adjacent if they are connected by an edge.

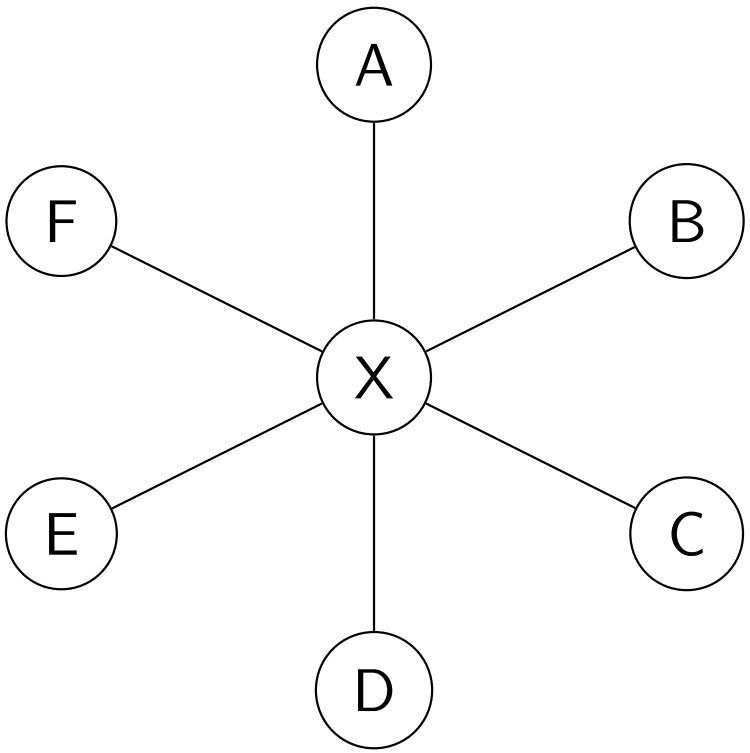


- Vertex 1 is adjacent to vertex 2.
- Vertex 3 is not adjacent to vertex 1.
- Vertex 0 is not adjacent ot any other vertex.

Undirected Graphs

Definition

An undirected graph is a graph in which, for each edge $\langle v_i, v_j \rangle$, there is another edge $\langle v_j, v_i \rangle$. That is, a direct connection between two vertices is always a two-way street — we can move from v_i to v_j and we can move from v_j to v_i.

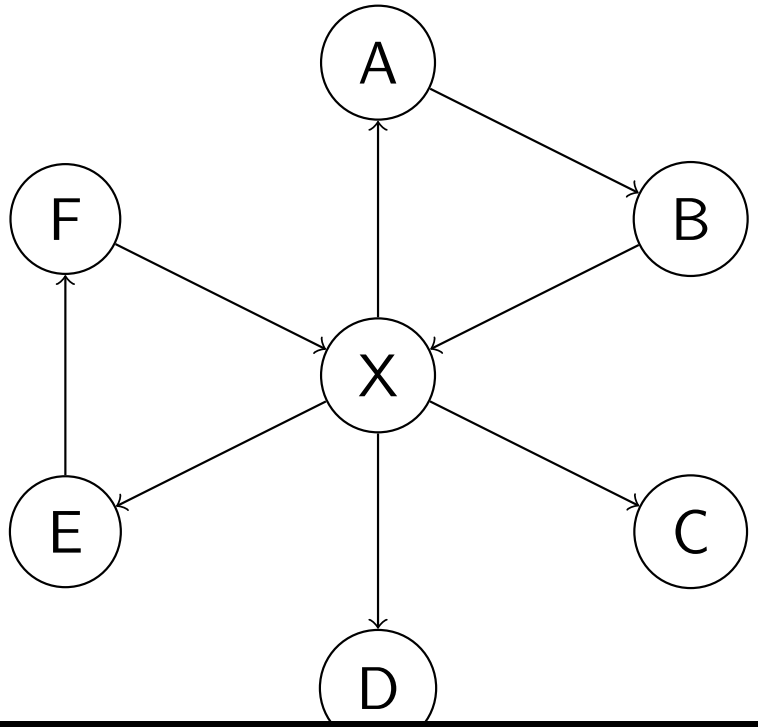


$$
\begin{aligned}
V &= \{A, B, C, D, E, F, X\} \\
E &= \{\langle A, X\rangle, \langle X, A\rangle, \langle B, X\rangle, \\
&\quad \langle X, B\rangle, \langle C, X\rangle, \langle X, C\rangle, \\
&\quad \langle D, X\rangle, \langle X, D\rangle, \langle E, X\rangle, \\
&\quad \langle X, E\rangle, \langle F, X\rangle, \langle X, F\rangle\}
\end{aligned}
$$

Directed Graphs

Definition

An directed graph (or digraph) is a graph in which, for each edge $\langle v_i, v_j \rangle$, there may or may not be another edge $\langle v_j, v_i \rangle$. That is, a direct connection between two vertices may be only a one-way street — we can move from v_i to v_j, but we may not be able to move from v_j to v_i.

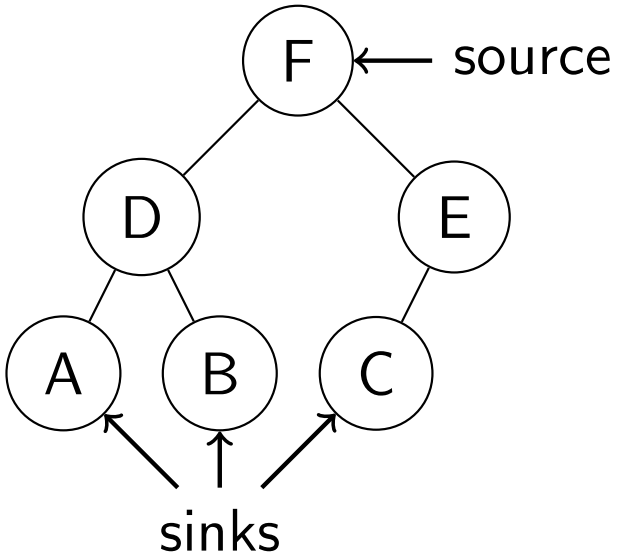


$$
\begin{aligned}
V &= \{A, B, C, D, E, F, X\} \\
E &= \{\langle A, B \rangle, \langle B, X \rangle, \langle E, F \rangle, \\
&\quad \langle F, X \rangle, \langle X, A \rangle, \langle X, C \rangle, \\
&\quad \langle X, D \rangle, \langle X, E \rangle\}
\end{aligned}
$$

Graph Terminology

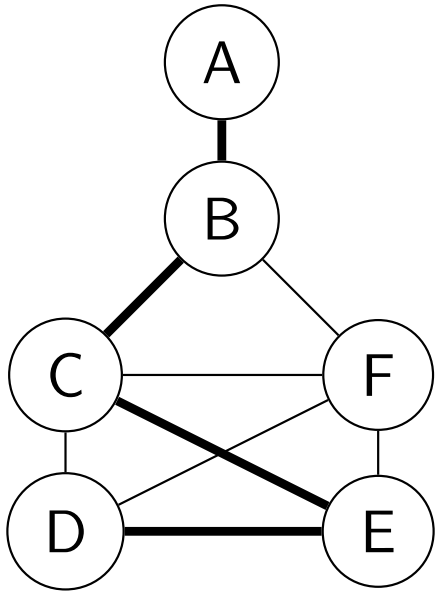
- Each vertex has an in-degree (the number of edges leading into it) and an out-degree (the number of edges leading out of it).
- A sink is a vertex with no edges leading out of it. (In a tree, these are the leaves.)
- A source is a vertex with no edges leading into it. (In a tree, this is the root.)

A Tree is a Directed Graph



A Path in a Graph

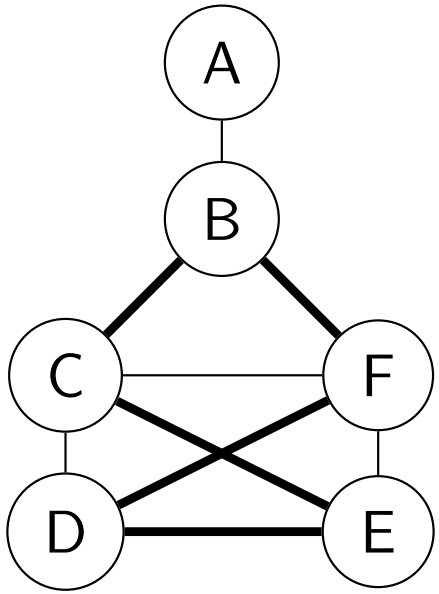
A path is a sequence of vertices $v_0, v_1, v_2, \ldots, v_m$ such that $\langle v_i, v_{i+1} \rangle$ is an edge for every consecutive pair of vertices v_i, v_{i+1} in the sequence.



For example, A, B, C, E, D is a path in the above graph.

A Cycle in a Graph

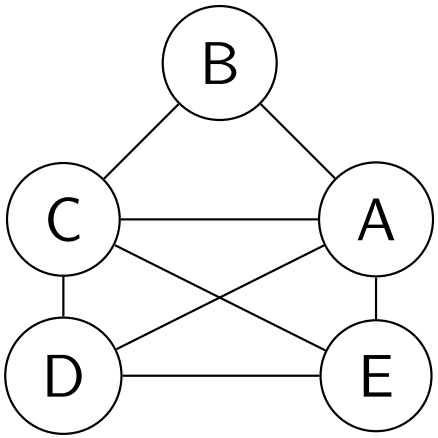
A cycle is a path that ends at the same vertex where it began.



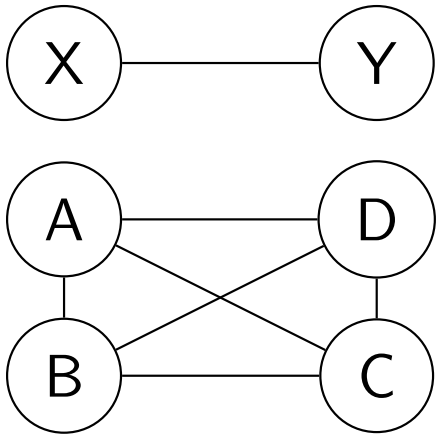
For example, B, C, E, D, F, B is a cycle in the above graph.

A Connected Graph

A graph is connected if a path exists from any vertex to any other vertex.



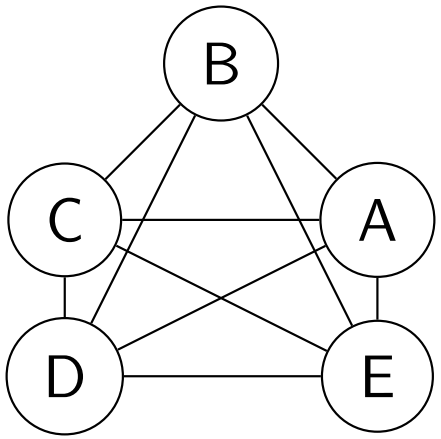
Connected graph



Unconnected graph

A Complete Graph

A graph is complete (or fully connected) if there is an edge between every pair of vertices.



Information in Graphs

- The vertices in a graph may represent something (e.g., cities on a map, computers in a network).
- Vertices in a graph can contain information (e.g., information about the cities or computers represented).
- The edges may represent spatial connections between vertices or some other type of connection between the vertices.
- The edges may also contain information (e.g., distance between two cities, the cost of travel between two cities, capacity of a computer connection, weight of a connection in a neural network). If the information is a number, then the graph is called a weighted graph.

Questions about Graphs

- How can we find a path from one vertex to another?
- What is the cheapest way of going from one vertex to another?
- How can we visit every vertex in the graph effectively?

We need a effective methods of storing a graph to answer these questions.

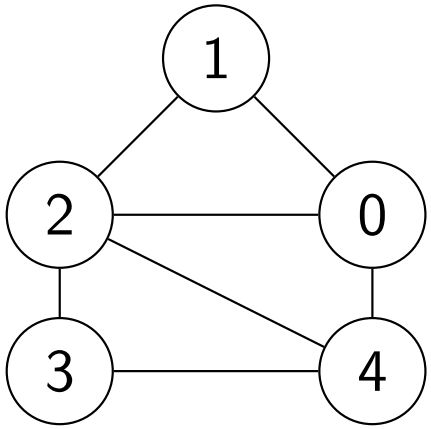
Graphs as Nodes and Links

- We could use nodes and links, just like we did for trees and linked lists.
- Each node v would contain a linked list of pointers to nodes to which v is connected.
- However, there are two other representations of graphs that are more useful in certain algorithms.

Adjacency Matrix (or Incidence Matrix)

- The entries in an adjacency matrix A are bits (boolean).
- The matrix A has one row and one column for each vertex in the graph (and the vertices are assumed to be $0, 1, 2, \ldots, k$).
- The entry $A(i, j)$ is 1 if there is an edge from vertex i to vertex j, and 0 if there isn't.

Adjacency Matrix of an Undirected Graph

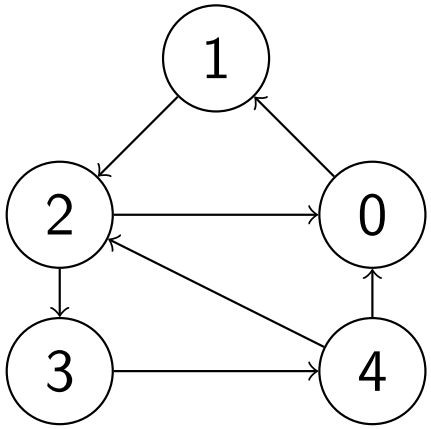


Adjacency Matrix:

	0	1	2	3	4
0	0	1	1	0	1
1	1	0	1	0	0
2	1	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0

Note that the adjacency matrix of an undirected graph is symmetric around the main diagonal.

Adjacency Matrix of a Digraph



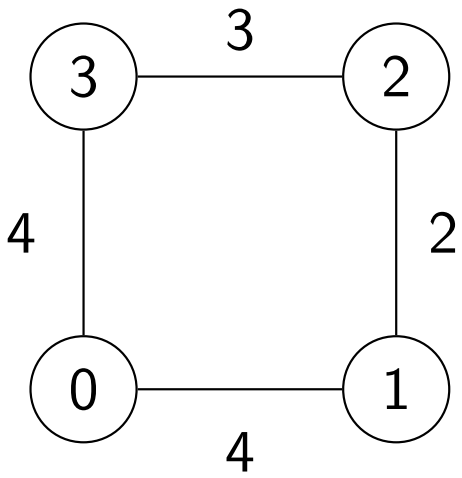
Adjacency Matrix:

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	1	0	0	1	0
3	0	0	0	0	1
4	1	0	1	0	0

Note that the adjacency matrix of a digraph is not usually symmetric.

Adjacency Matrix of a Weighted Graph

If the edges have weights, we can store the weight of edge $\langle i, j \rangle$ in $A(i, j)$:



Adjacency Matrix:

	0	1	2	3
0	0	4	0	4
1	4	0	2	0
2	0	2	0	3
3	4	0	3	0

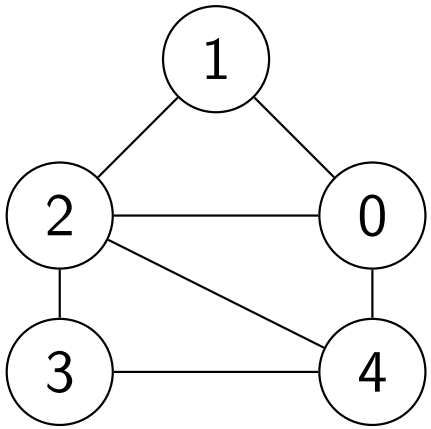
Adjacency Matrix

- Primary advantage: Checking to see if there is an edge from vertex i to vertex j = checking entry $A(i, j)$, which takes constant time. (Fast! Easy!)
- For some graph algorithms, being able to easily do a matrix multiplication is an advantage, too.
- The major disadvantage: Usually, the matrix is sparse, so a lot of space is wasted storing zeroes.
- Also, if we want all the edges out of a vertex i, we have to look at all k entries in row i, even there are no edges out of i.

Adjacency List

- Instead of wasting space on all possible edges (and recording a zero when the edge isn't there), only record the edges that exist.
- Idea: Have an array with one entry per vertex.
- Idea: Entry i contains a list of the edges out of vertex i.

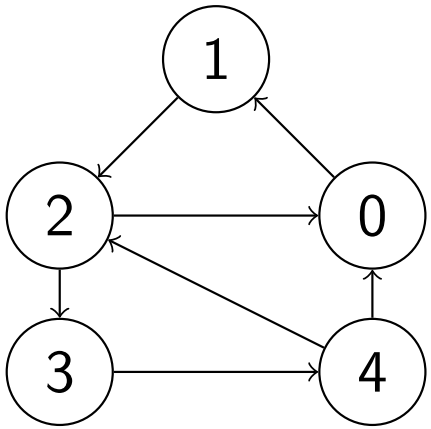
Adjacency List of an Undirected Graph



Adjacency List:

0:	1, 2, 4
1:	0, 2
2:	0, 1, 3, 4
3:	2, 4
4:	0, 2, 3

Adjacency List of a Digraph



Adjacency List:

0:	1
1:	2
2:	0, 3
3:	4
4:	0, 2

Adjacency List

- Checking if there is an edge from vertex i to vertex j is now somewhat more difficult.
- We now have to search the list of edges out of vertex i.
- We must use a linear search of the list.
- Since vertex i can be connected to all the other vertices, that search can cost $O(k)$, where k is the number of vertices.
- But we aren't wasting space on edges that aren't there.
- And we don't have to search through edges that aren't there to find all the edges out of a vertex.

Adjacency Matrix or Adjacency List?

How do you decide which representation to use? You have to look at your application. For example:

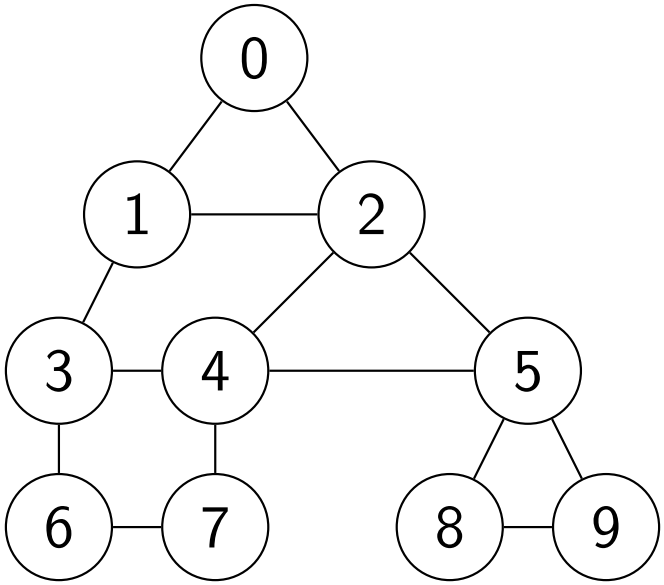
- If you need to be able to quickly tell if there is an edge between vertices i and j, then use an adjacency matrix.
- If you need to perform a matrix multiplication, then use an adjacency matrix.
- If you need fast access to all edges out of a vertex, use the adjacency list.
- If space is an issue (a huge number of vertices), then an adjacency list is a good idea.

Traversing in a Graph

- The most common operation is to visit all the vertices in a systematic way.
- Two types of traversals:
  - Depth-first traversal (or depth-first search)
  - Breadth-first traversal (or breadth-first search)
- A traversal starts at a vertex v and visits all the vertices u such that a path exists from v to u.

Depth-first Traversal

A depth-first traversal searches all the way down a path before backing up to explore alternatives — it is a recursive, stack-based traversal.

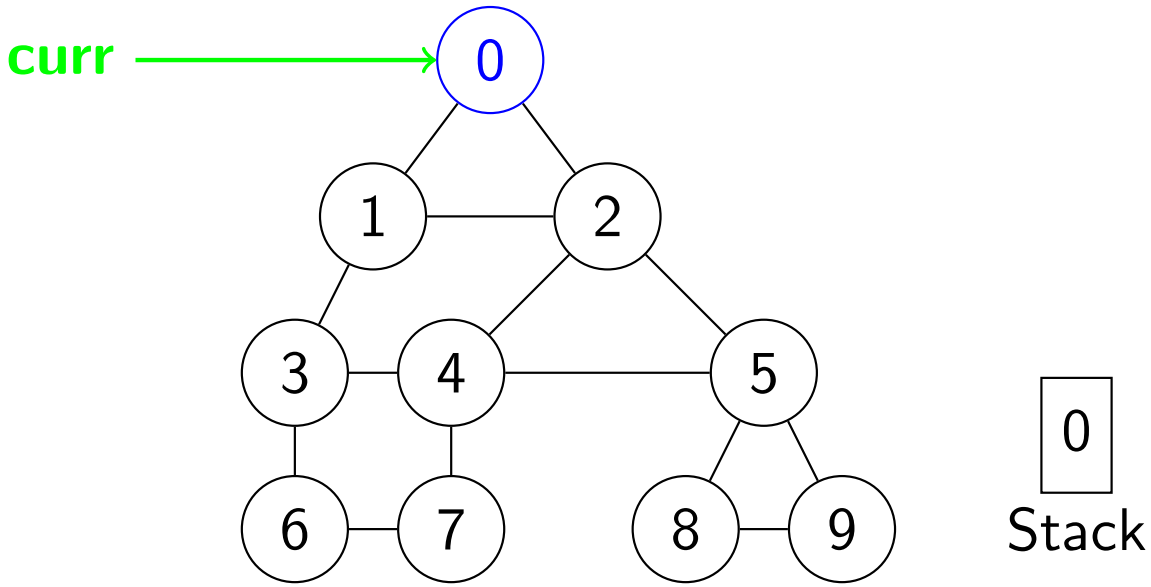


We will traverse the above graph starting at 0, with all vertices currently unvisited.

Depth-first Traversal: Example

To depth-first traverse at vertex 0:

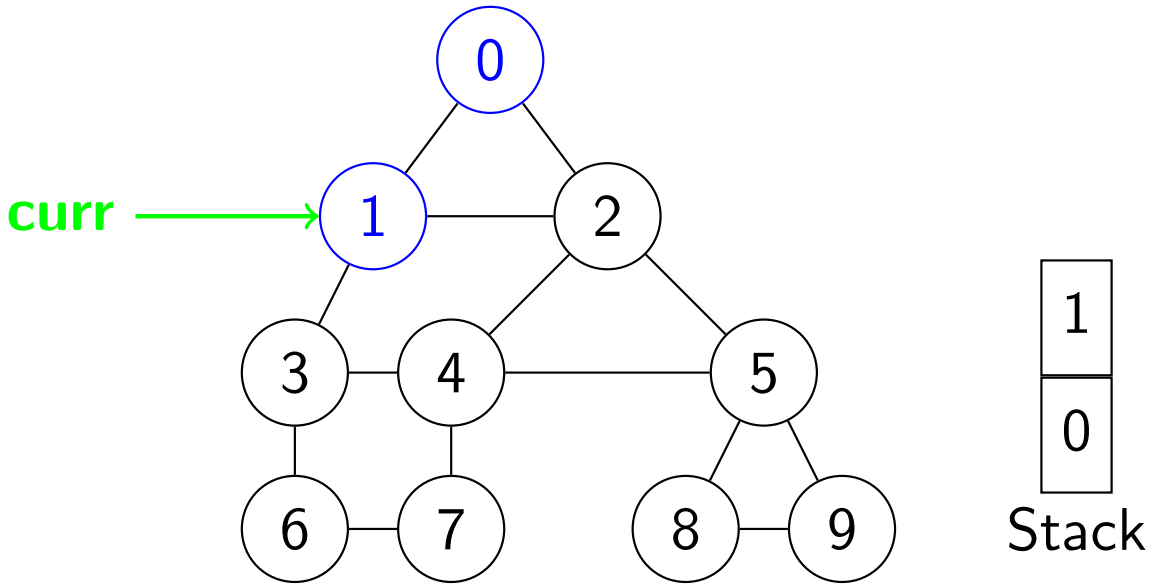
- Mark the current vertex (0) as visited, and then
- Recursively depth-first traverse each of the adjacent unvisited vertices.



Assume that we examine the adjacent vertices in sorted order (so we would first look at vertex 1, then at vertex 2). Vertex 1 is unvisited, so we next recursively depth-first traverse vertex 1.

Depth-first Traversal: Example

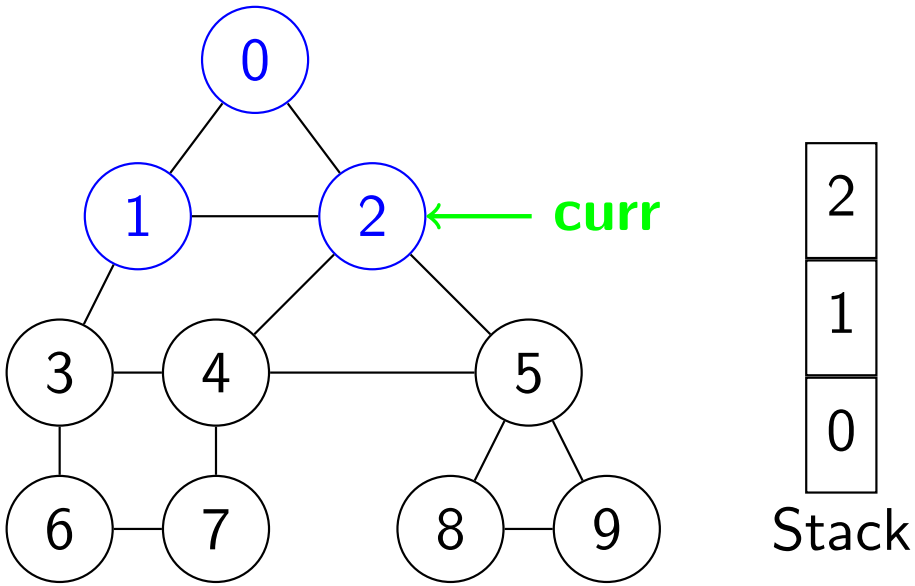
Now mark the current vertex (1) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 2 is unvisited, so we next recursively depth-first traverse vertex 2.

Depth-first Traversal: Example

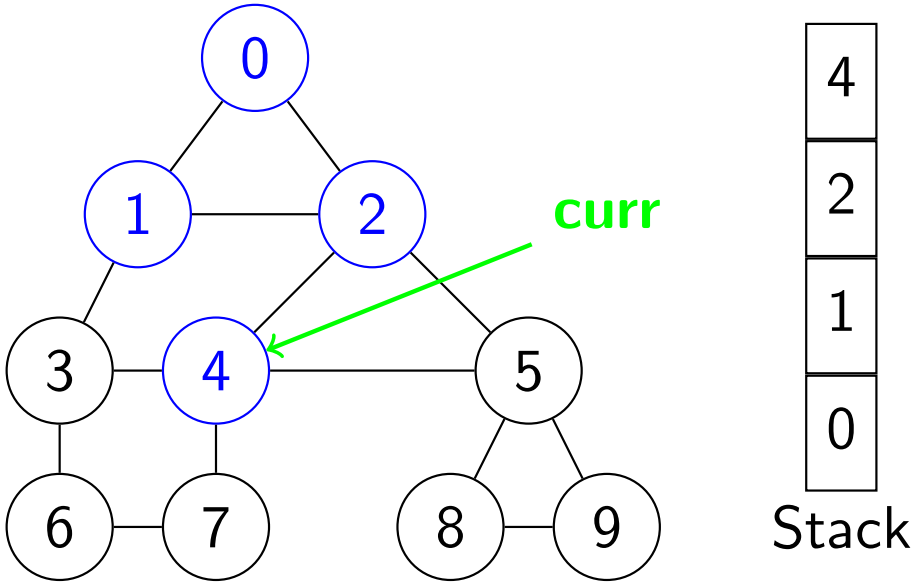
Now mark the current vertex (2) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 4 is unvisited, so we next recursively depth-first traverse vertex 4.

Depth-first Traversal: Example

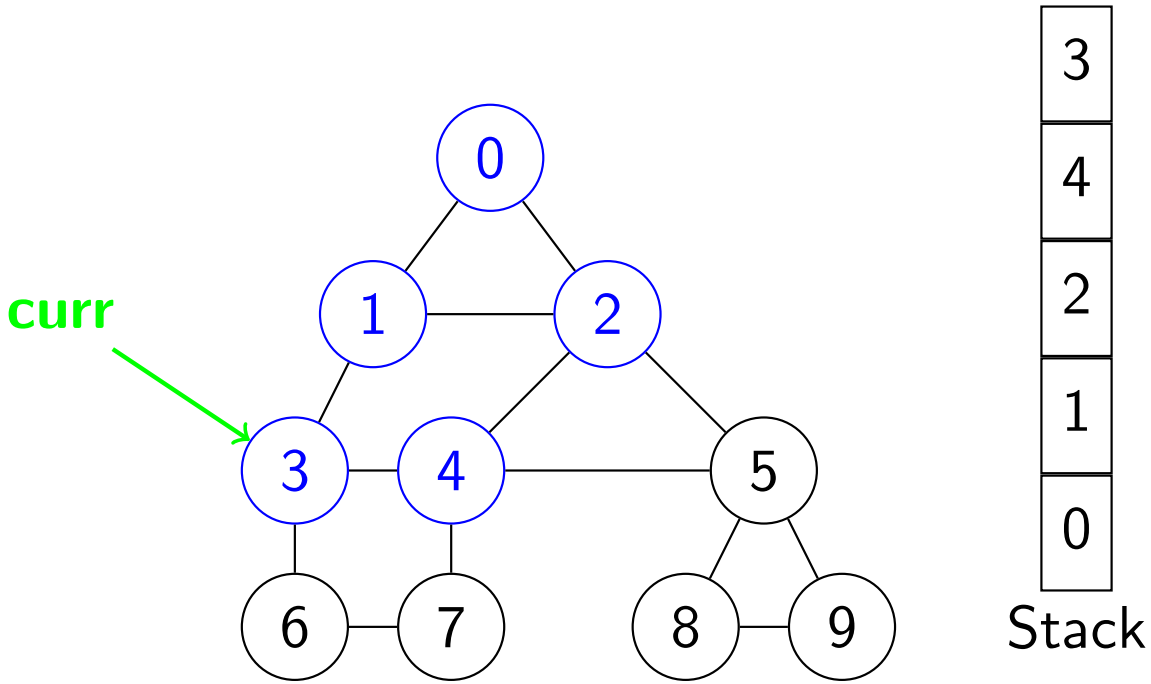
Now mark the current vertex (4) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 3 is unvisited, so we next recursively depth-first traverse vertex 3.

Depth-first Traversal: Example

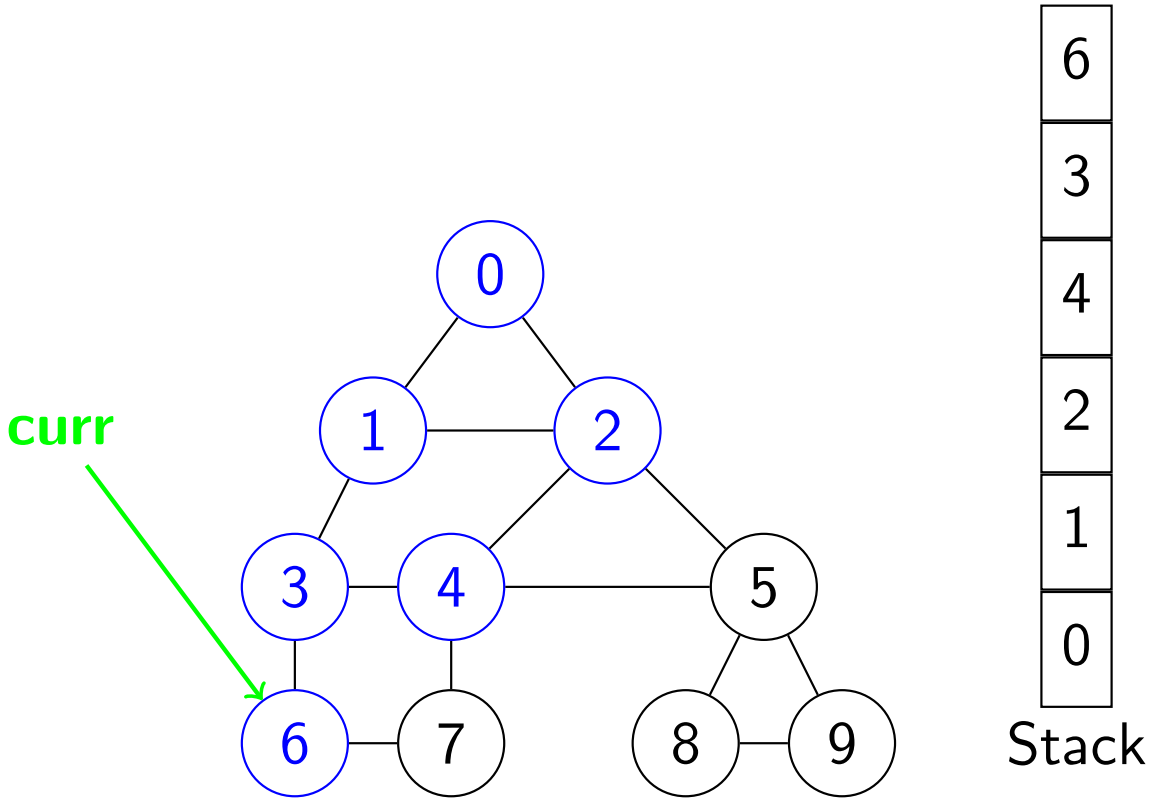
Now mark the current vertex (3) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 6 is unvisited, so we next recursively depth-first traverse vertex 6.

Depth-first Traversal: Example

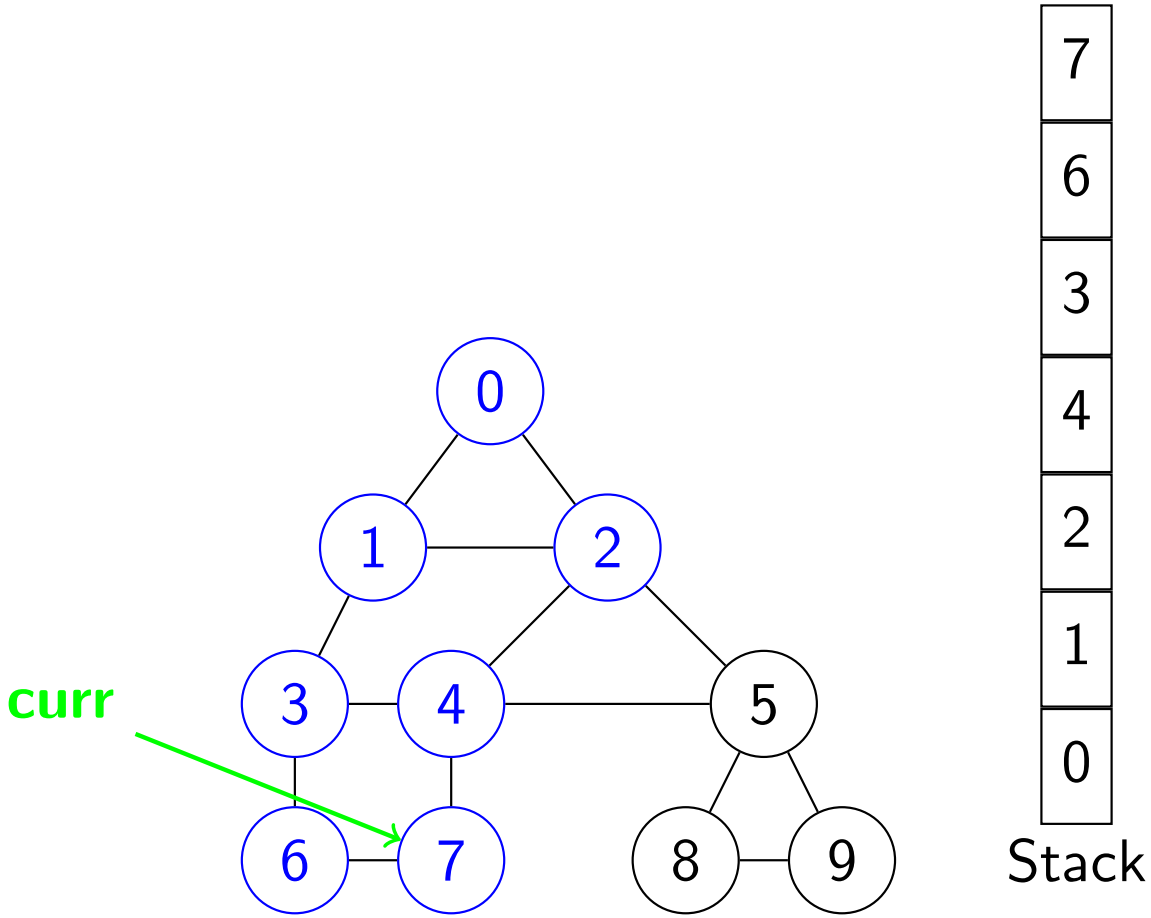
Now mark the current vertex (6) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 7 is unvisited, so we next recursively depth-first traverse vertex 7.

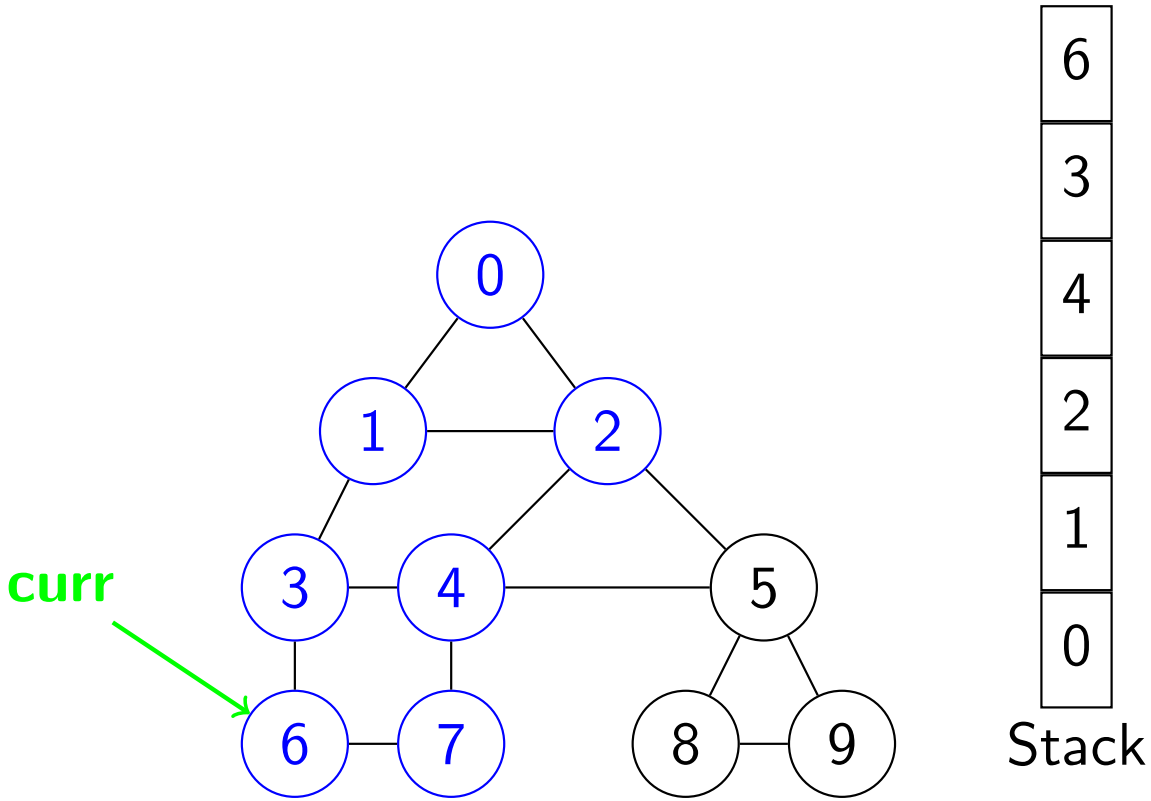
Depth-first Traversal: Example

Mark 7 as visited, and recursively depth-first traverse the adjacent unvisited vertices.



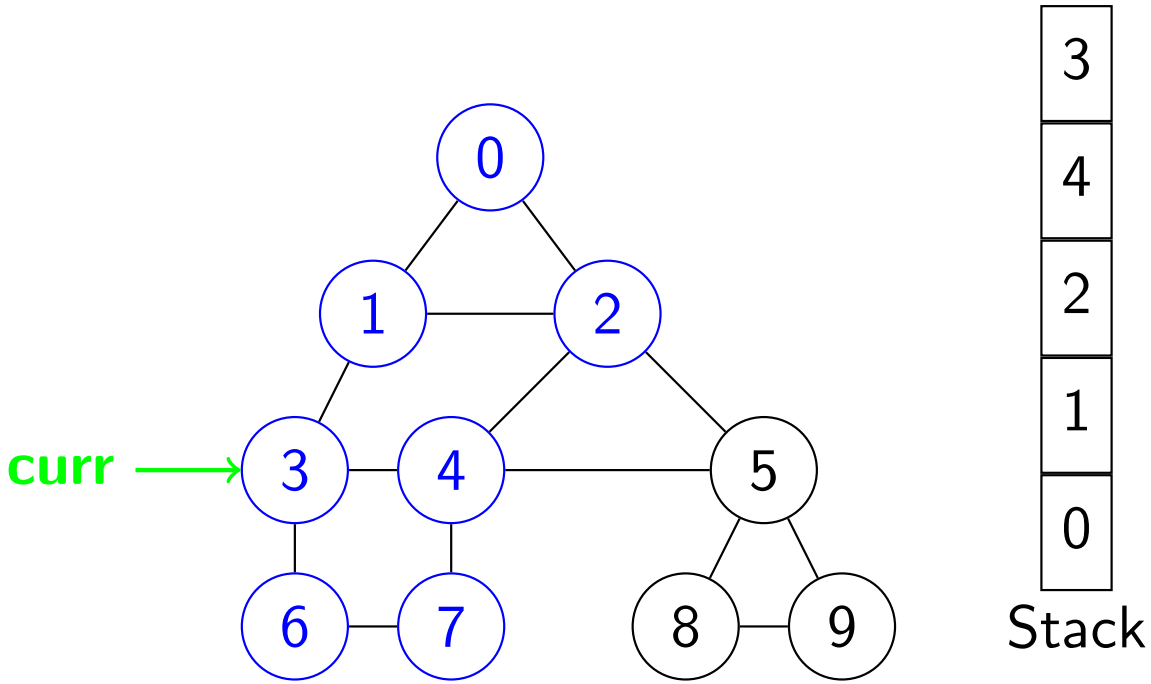
No adjacent vertices are unvisited, so pop the stack to return to a previous vertex and look for unvisited adjacent vertices there.

Depth-first Traversal: Example



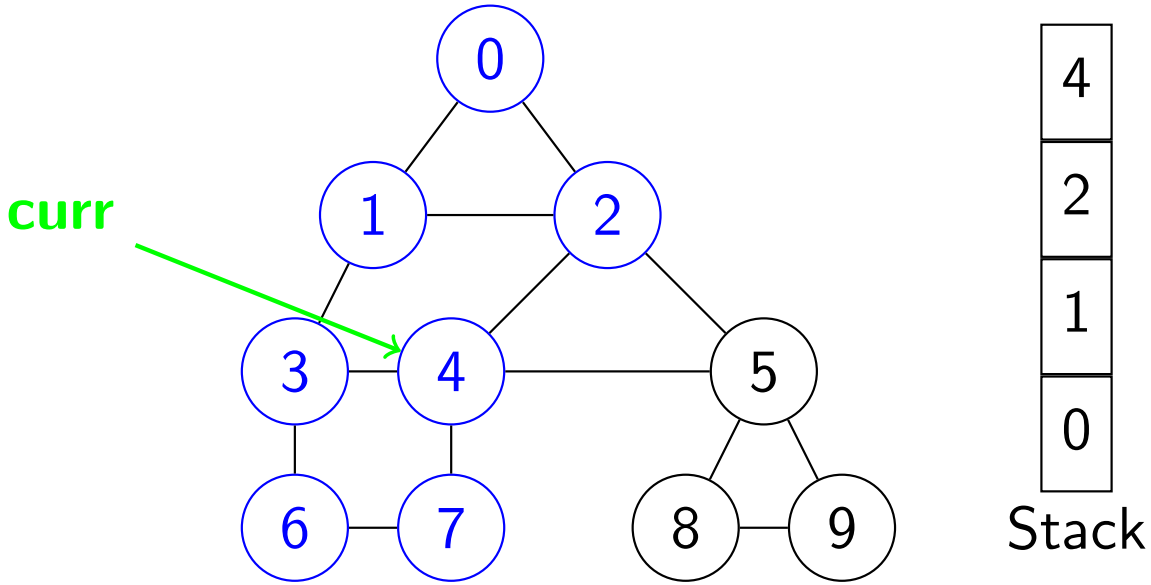
No vertices adjacent to 6 are unvisited, so pop the stack again.

Depth-first Traversal: Example



No vertices adjacent to 3 are unvisited, so pop the stack again.

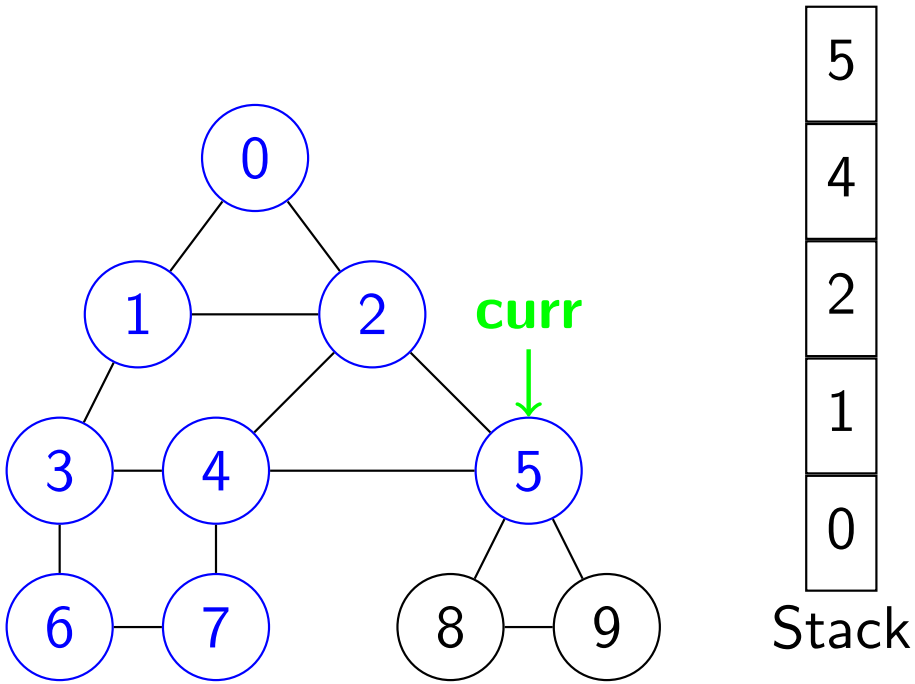
Depth-first Traversal: Example



Vertex 5 is adjacent to 4 and is unvisited, so depth-first traverse 5.

Depth-first Traversal: Example

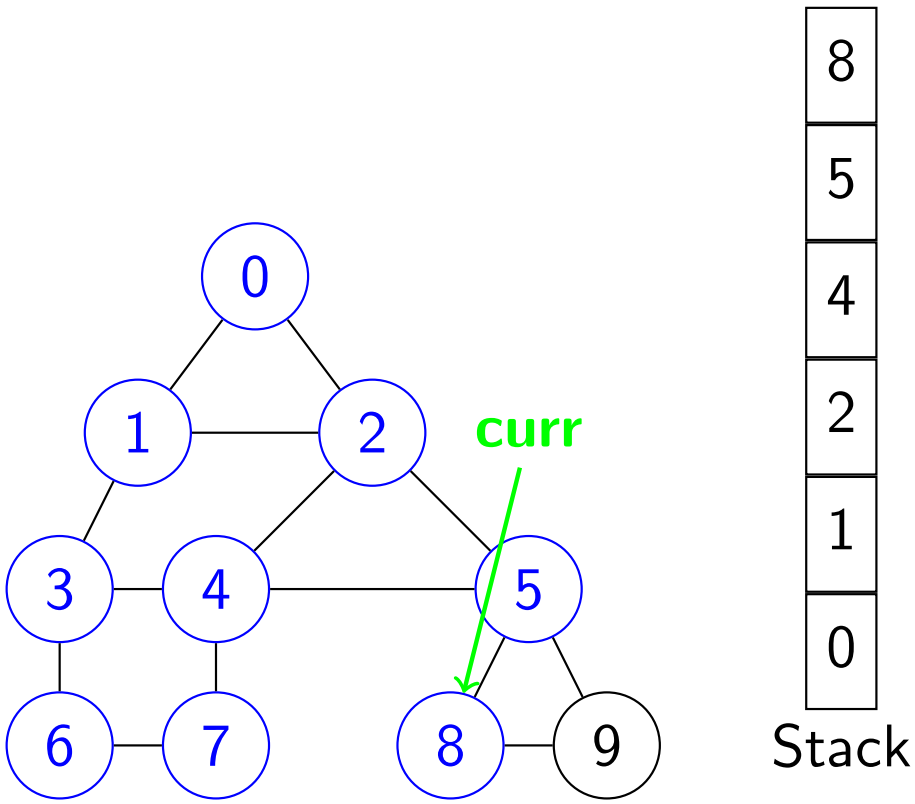
Mark 5 as visited, and recursively depth-first search adjacent unvisited vertices.



Vertex 8 is adjacent to 5 and is unvisited, so depth-first traverse 8.

Depth-first Traversal: Example

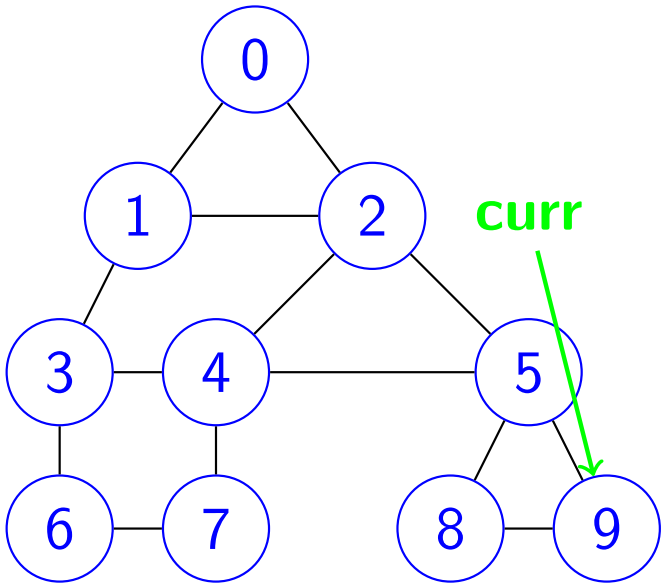
Mark 8 as visited, and recursively depth-first search adjacent unvisited vertices.

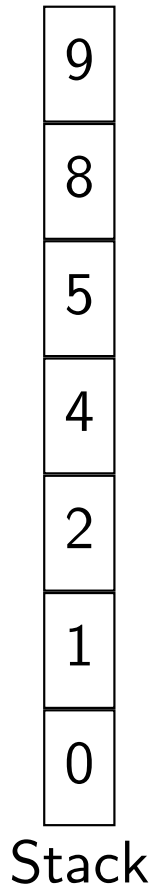


Vertex 9 is adjacent to 8 and is unvisited, so depth-first traverse 9.

Depth-first Traversal: Example

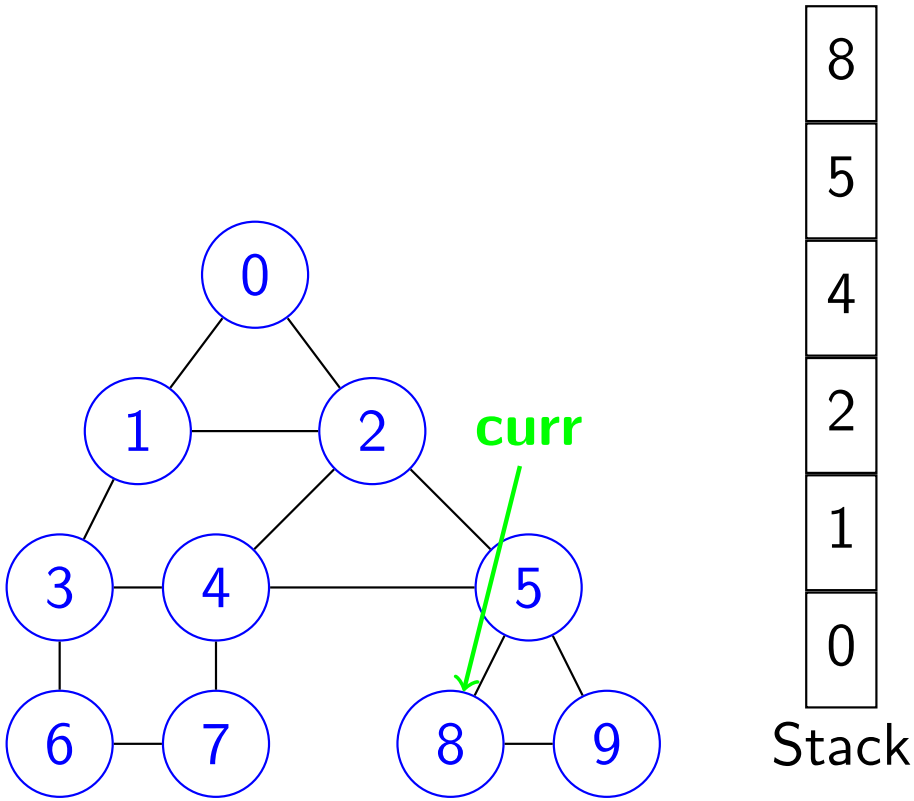
Mark 9 as visited, and recursively depth-first search adjacent unvisited vertices.





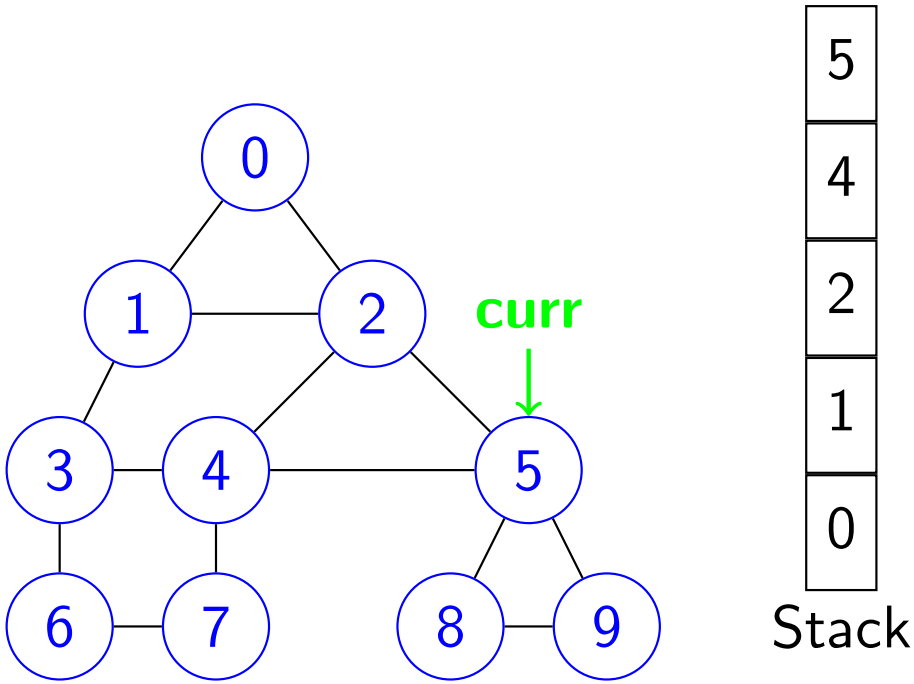
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



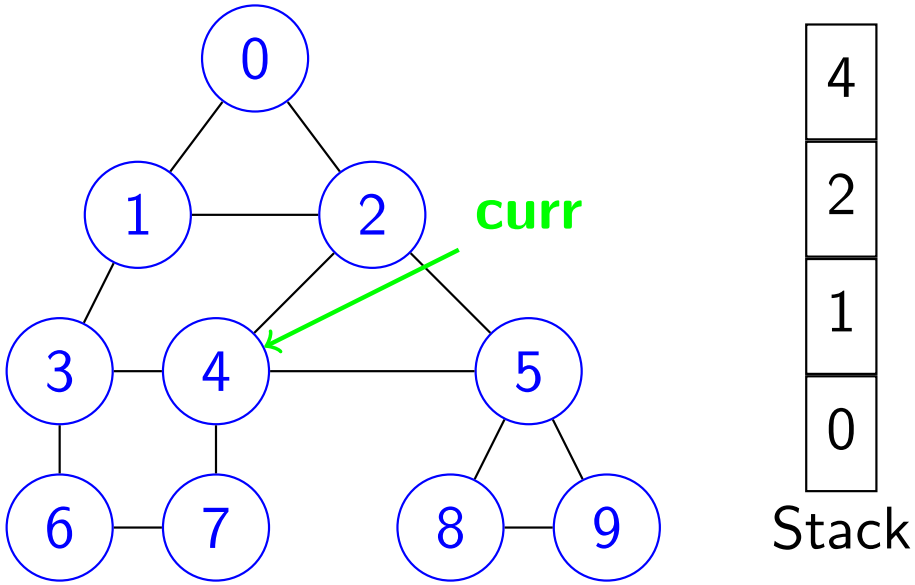
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



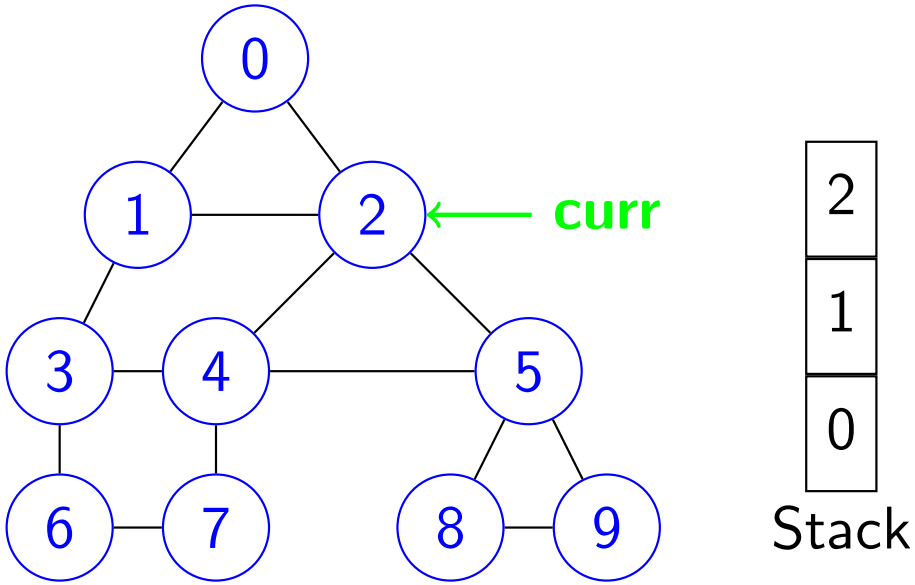
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



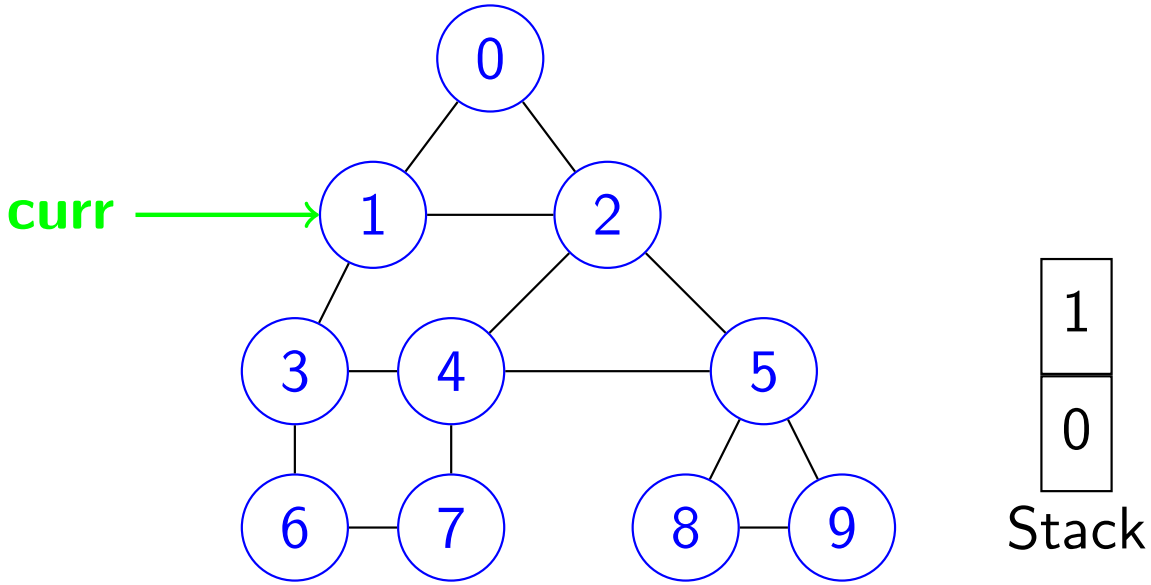
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



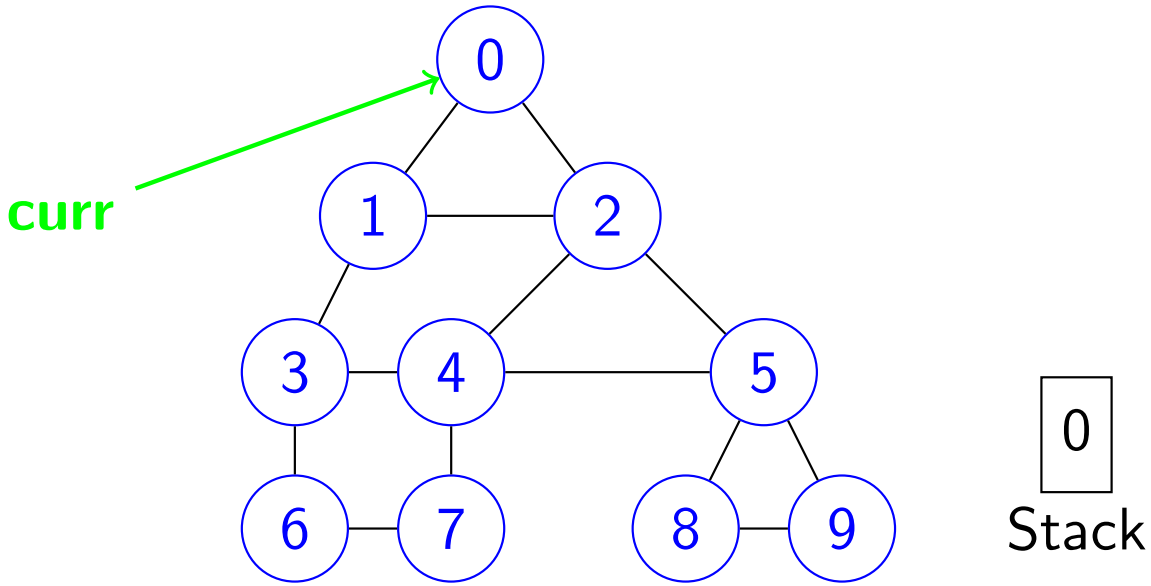
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



No adjacent vertex is unvisited, so pop the stack.

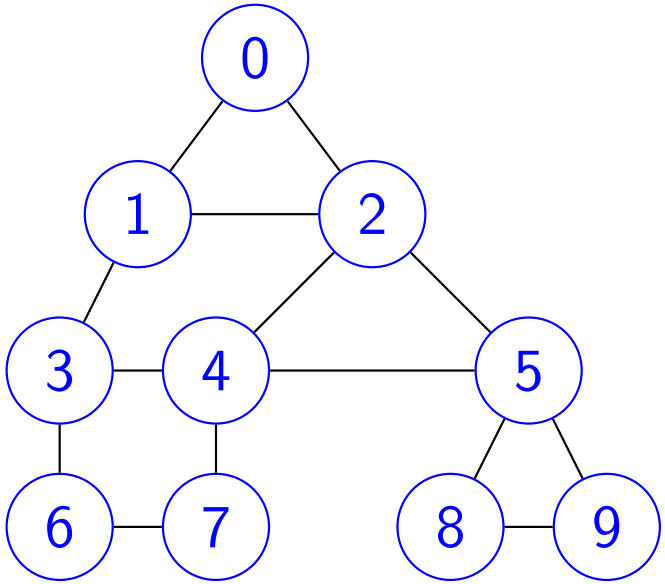
Depth-first Traversal: Example



No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example

Now the stack is empty, so we have traversed the whole graph.



Stack

Depth-first Traversal and Paths

- Notice how the stack holds the path we took from the starting vertex to the current vertex.
- If we wanted to find a path from some vertex u to some other vertex v, we could perform a depth-first traversal starting at u and simply output the stack (from bottom to top) when we find v.
- Notice that the path we find will not necessarily be the shortest path from u to v, but we will find a path if one exists.

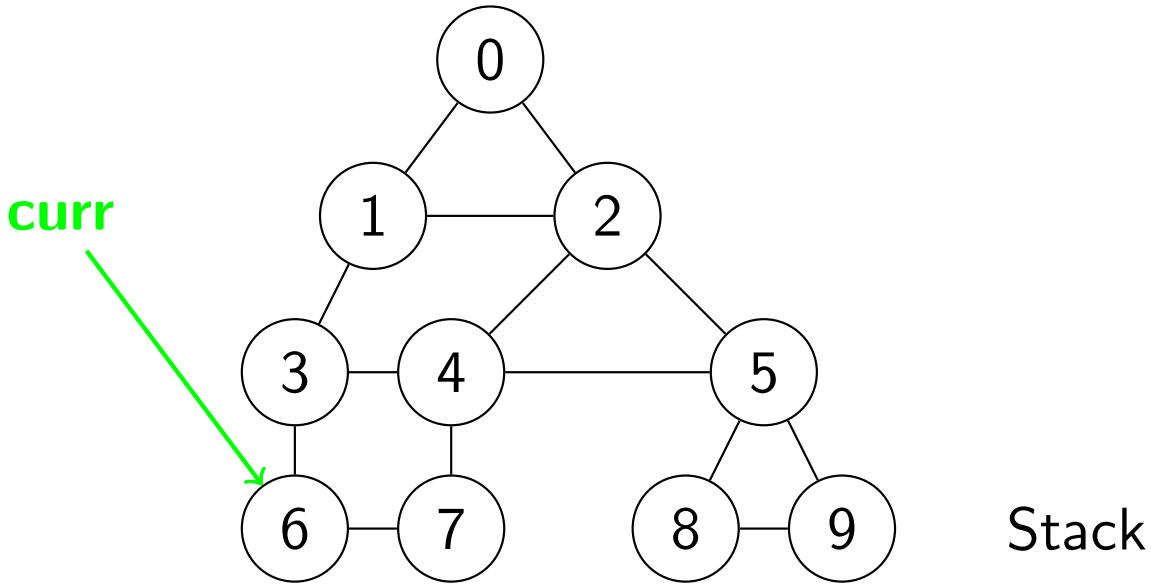
Printing in a Depth-First Traversal

- We "visit" a vertex when we mark it as visited.
- In our example, we did nothing when we visited a vertex.
- If we print out the contents of the vertex when we visit it, then the output would be

$$0, 1, 2, 4, 3, 6, 7, 5, 8, 9$$

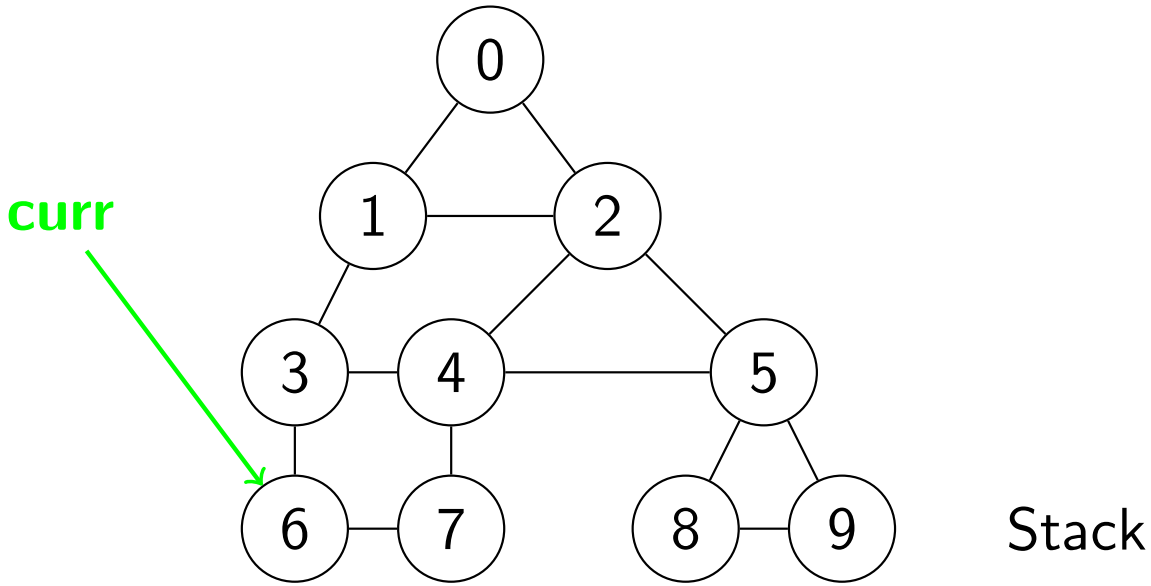
Depth-first Traversal: Example

What would the output be if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



Depth-first Traversal: Example

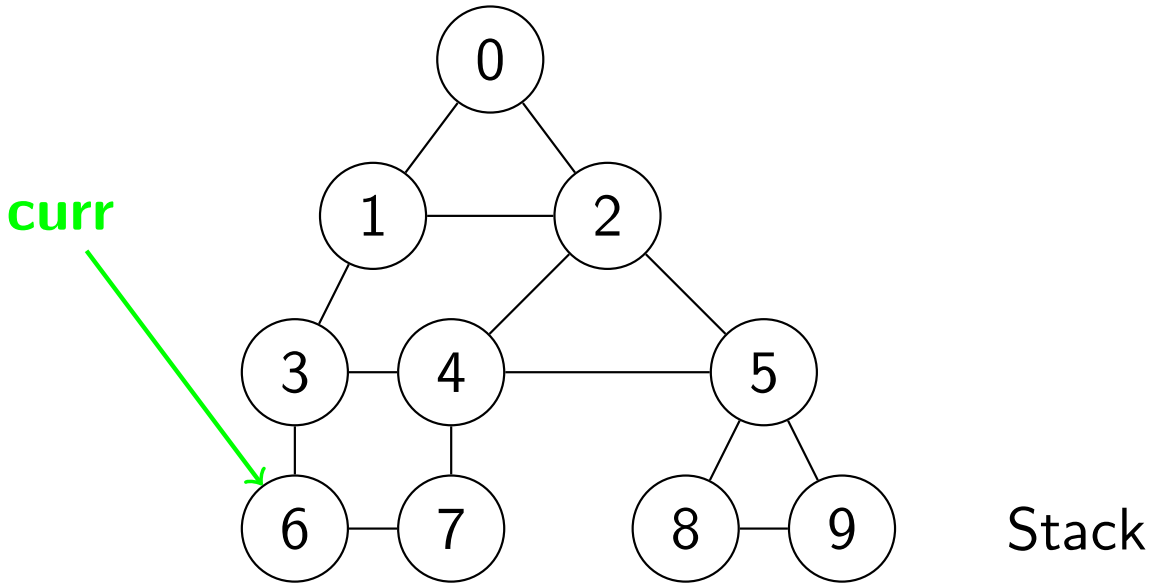
What would the output be if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



Answer: 6, 3, 1, 0, 2, 4, 5, 8, 9, 7

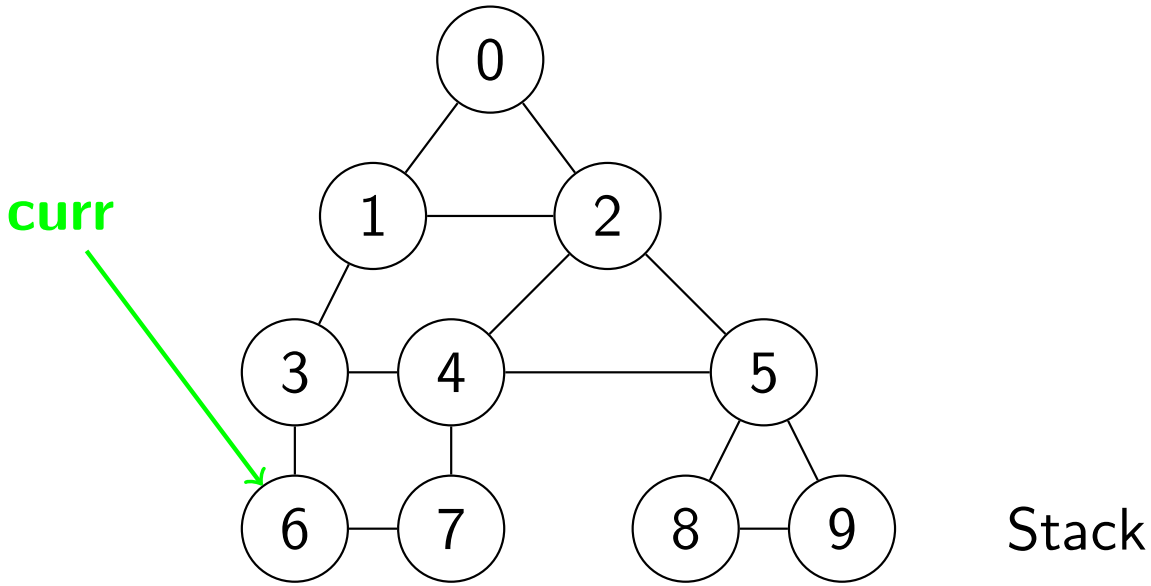
Depth-first Traversal: Example

What path would we find to vertex 7 if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



Depth-first Traversal: Example

What path would we find to vertex 7 if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



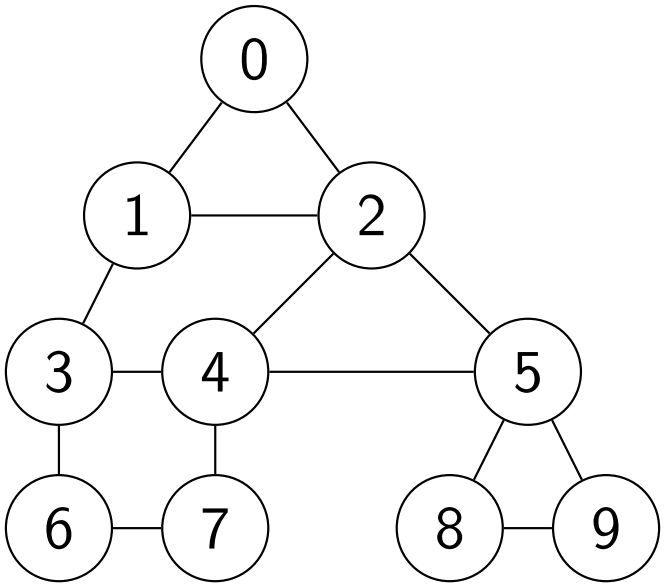
Answer: 6, 3, 1, 0, 2, 4, 7

Depth-first Traversal Pseudocode

```
public void depthFirstTraveral() {

  mark this vertex as visited;
  visit the vertex (e.g., print);
  for each vertex v adjacent to this
    if v is unvisited
      v.depthFirstTraversal();

} // end depthFirstTraversal
```

Breadth-first Traversal

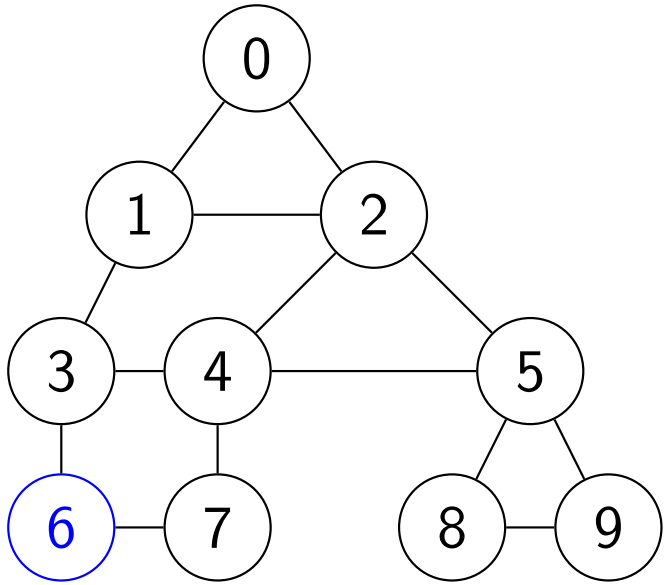
A breadth-first traversal visits all nearby vertices first before moving farther away. It is a queue-based, iterative traversal.



We will breadth-first traverse the above graph starting at vertex 6.

Breadth-first Traversal Example

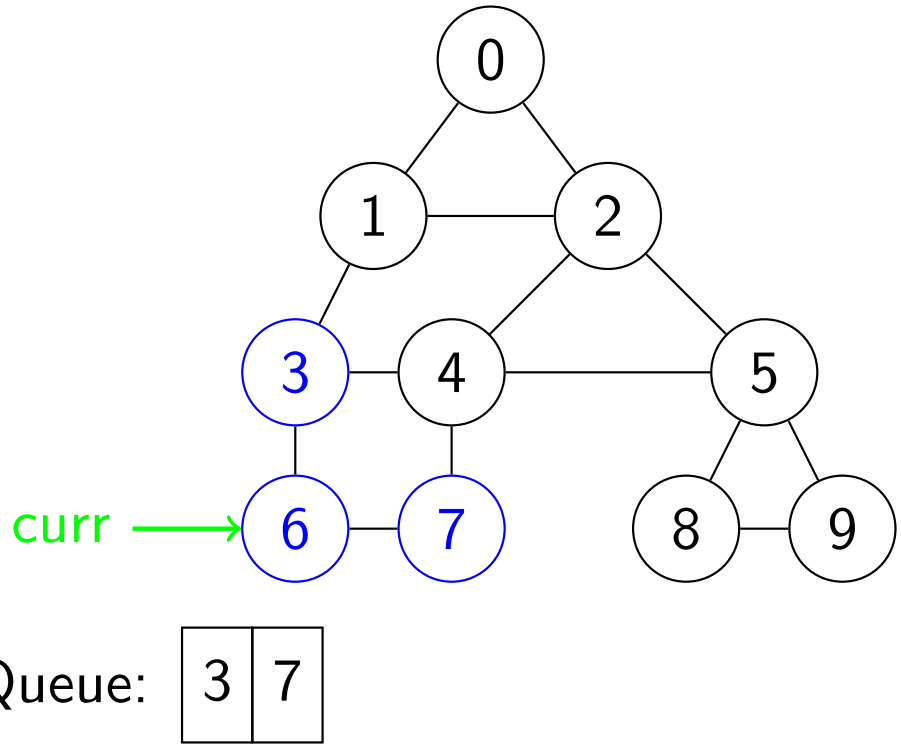
To begin the breadth-first traversal, visit the starting vertex and put it on the queue.



Queue: 6

Breadth-first Traversal Example

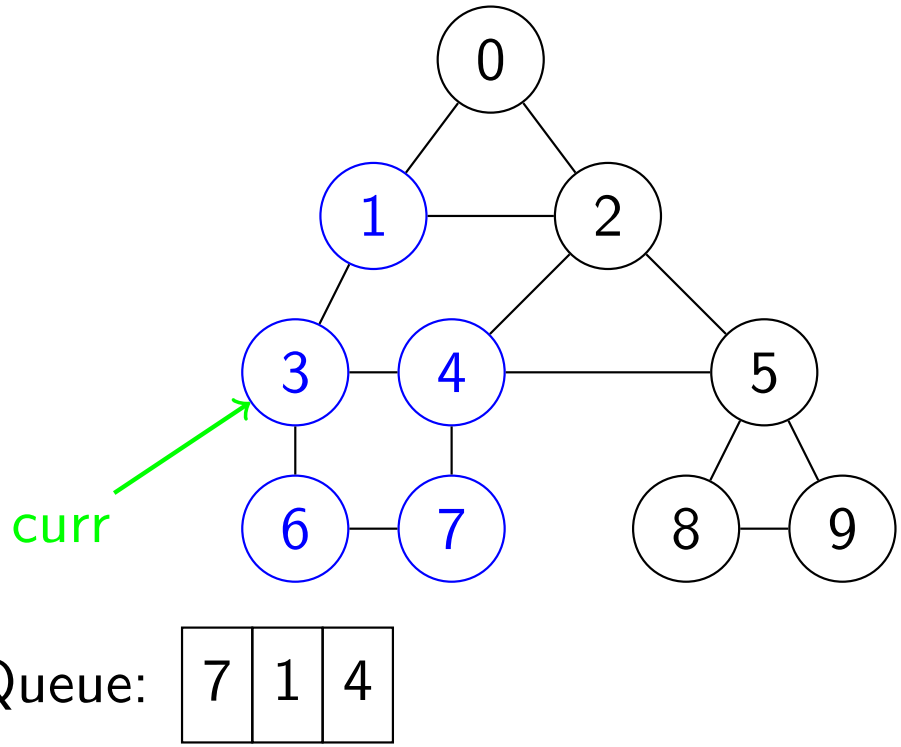
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



3 and 7 were the unvisited vertices we found adjacent to 6.

Breadth-first Traversal Example

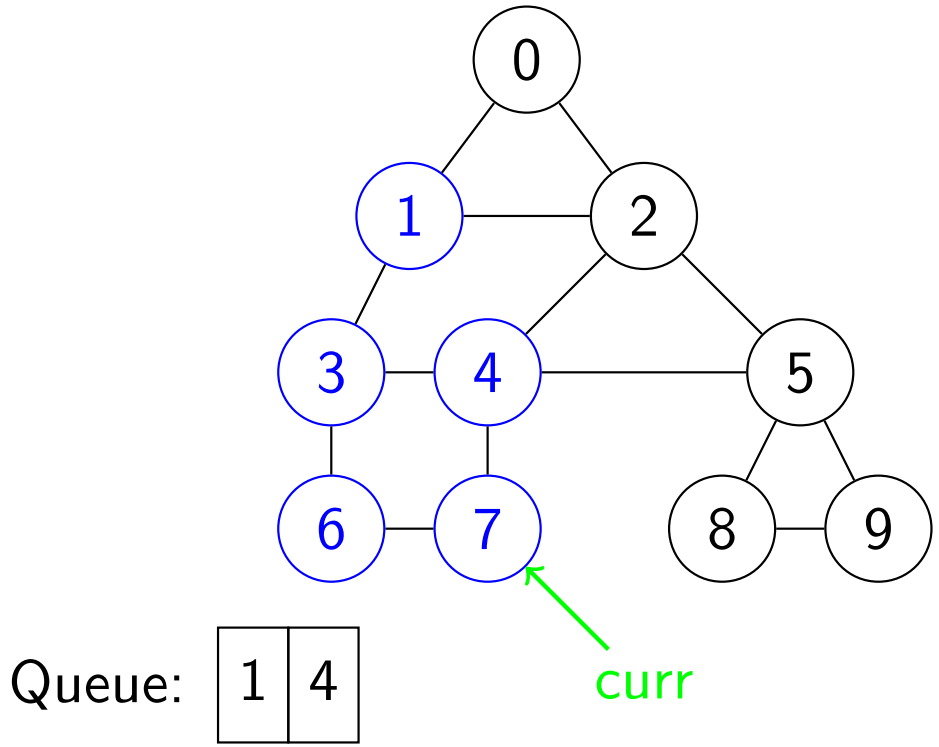
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



1 and 4 were the unvisited vertices we found adjacent to 3.

Breadth-first Traversal Example

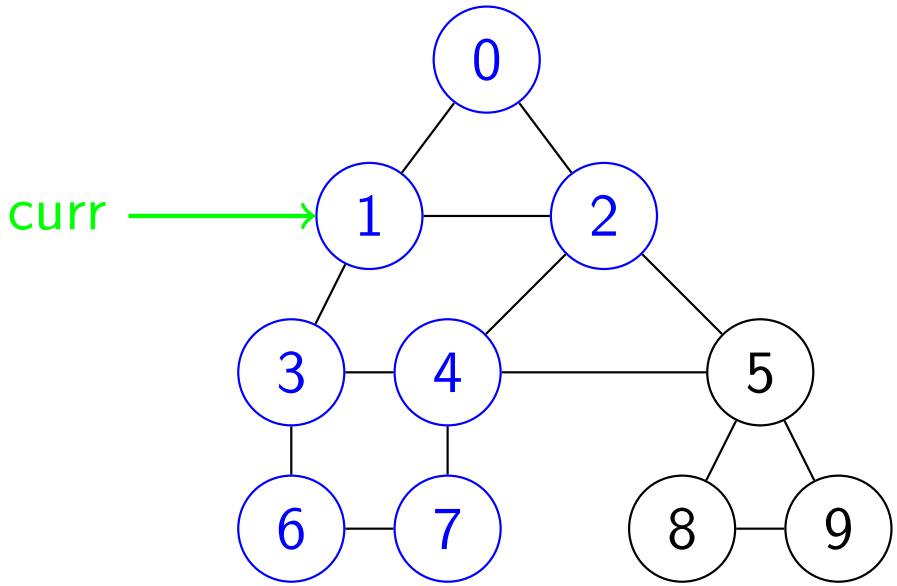
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



There were no unvisited vertices found adjacent to 7.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



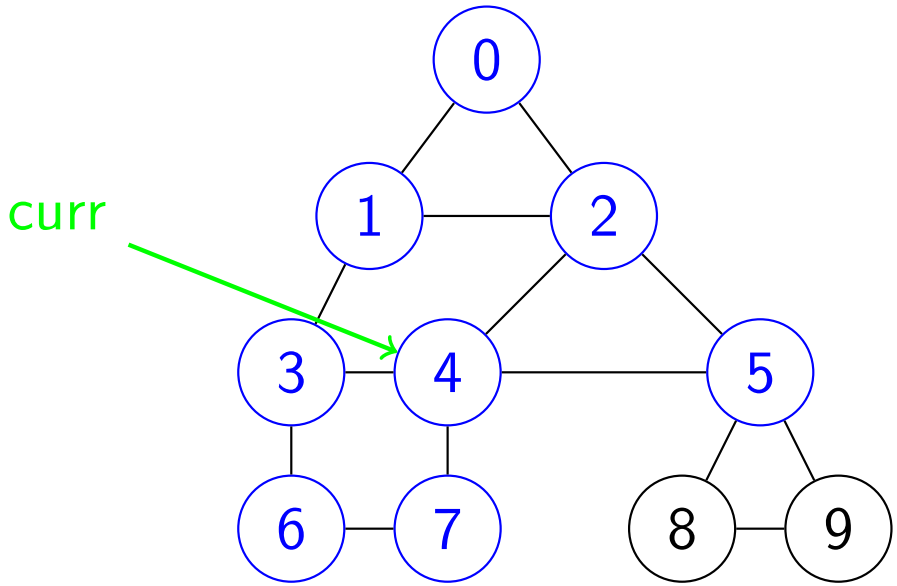
Queue:

4	0	2

0 and 2 were unvisited vertices adjacent to 1.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



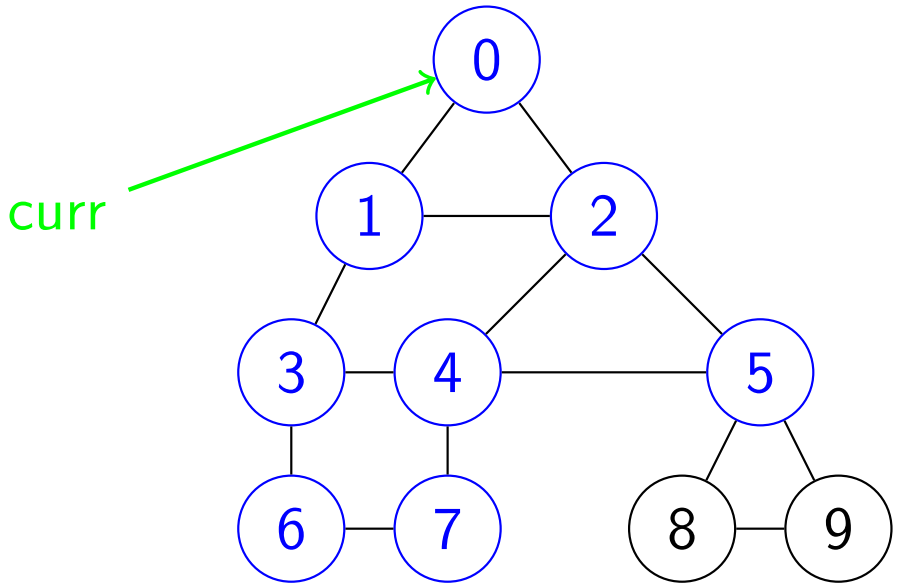
Queue:

0	2	5

5 was an unvisited vertex adjacent to 4.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



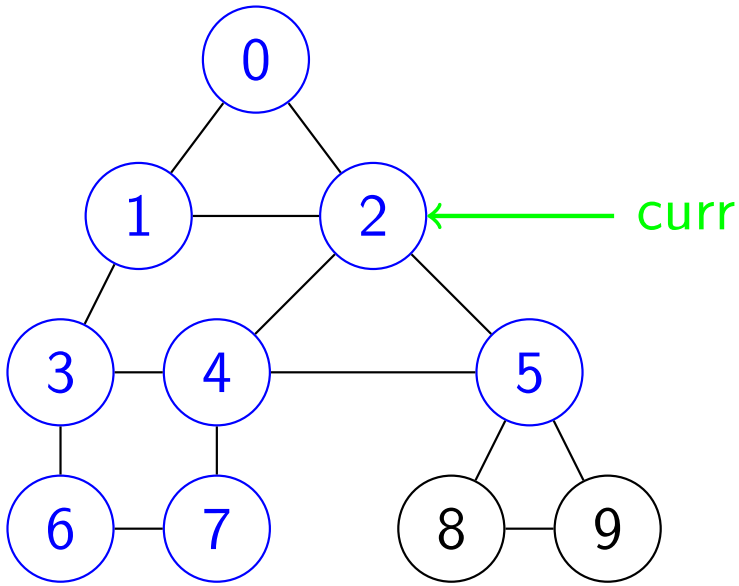
Queue:

2	5

No unvisited vertices were found adjacent to 0.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.

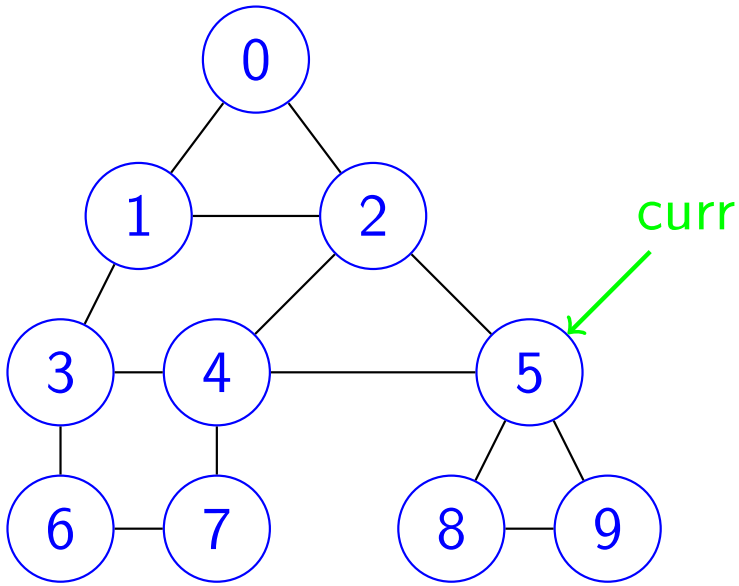


Queue: 5

No unvisited vertices were found adjacent to 2.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



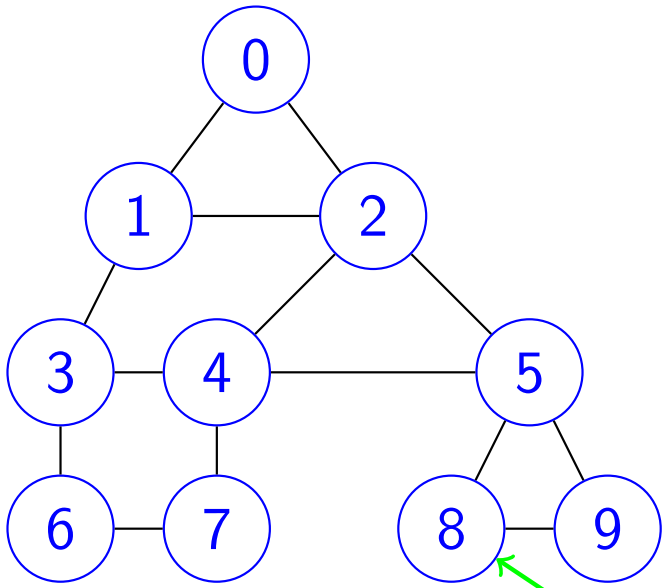
Queue:

8	9

8 and 9 were unvisited vertices adjacent to 5.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.

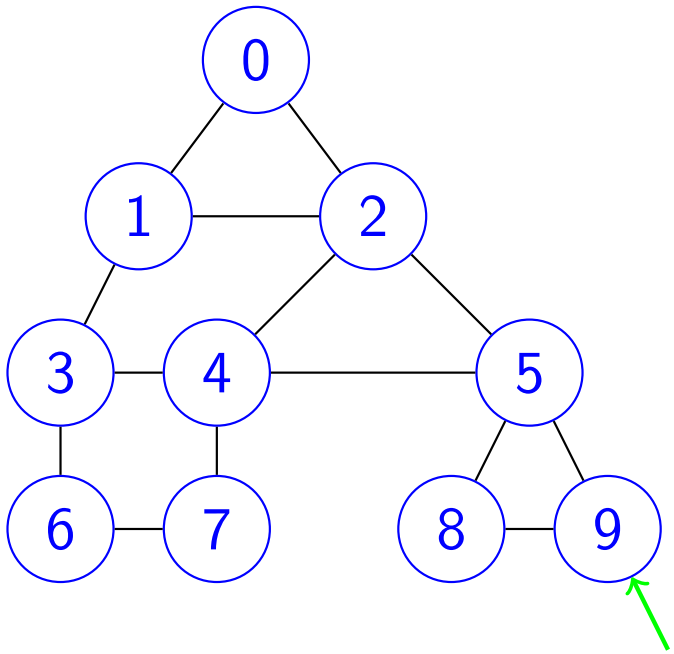


Queue: 9

No unvisited vertices were found adjacent to 8.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



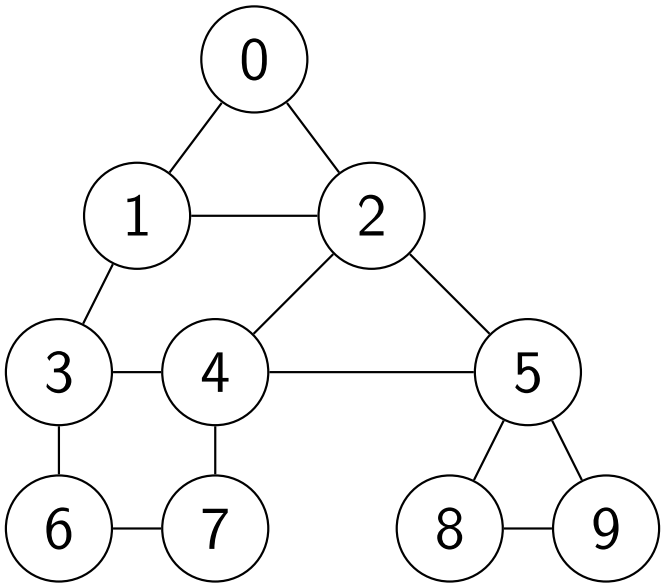
Queue:

No unvisited vertices were found adjacent to 9. Since the queue is now empty, the traversal is finished.

Printing in a Breadth-first Traversal

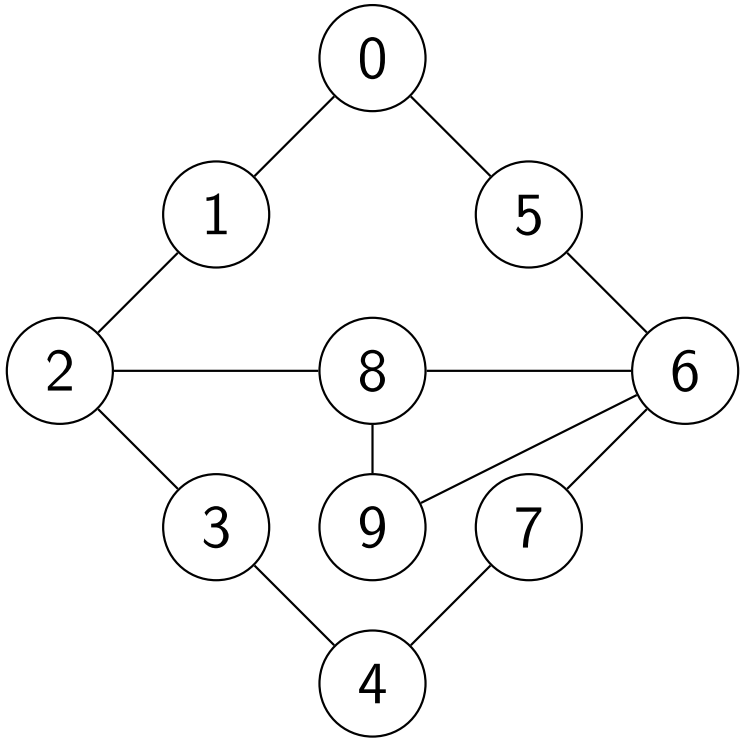
- In our example, we did nothing at each vertex when we visited it.
- If we print out the contents of the vertex when we visit it, then the output would be

$$6, 3, 7, 1, 4, 0, 2, 5, 8, 9$$



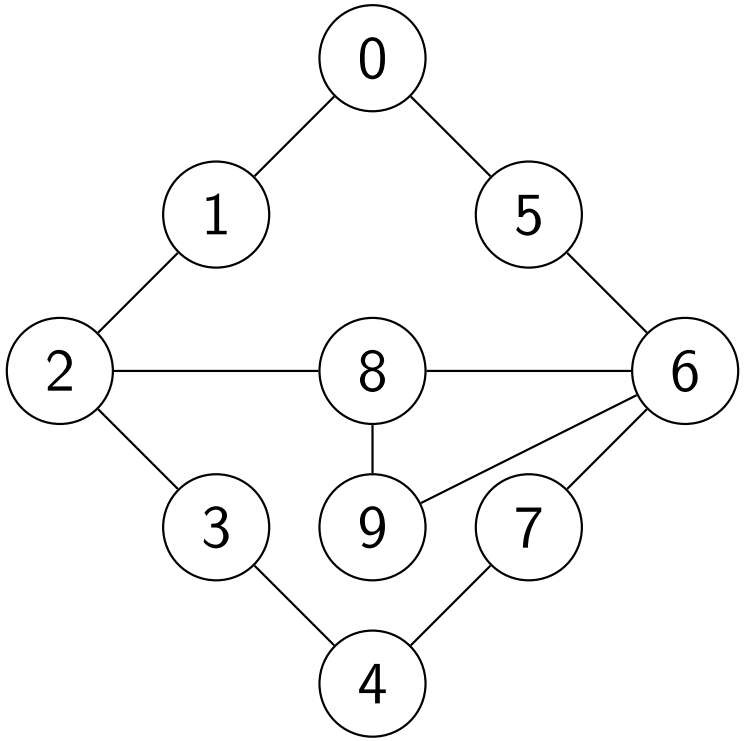
Breadth-first Traversal: Another Example

What would the output be if we performed a breadth-first traversal of the following graph, beginning at vertex 0:



Breadth-first Traversal: Another Example

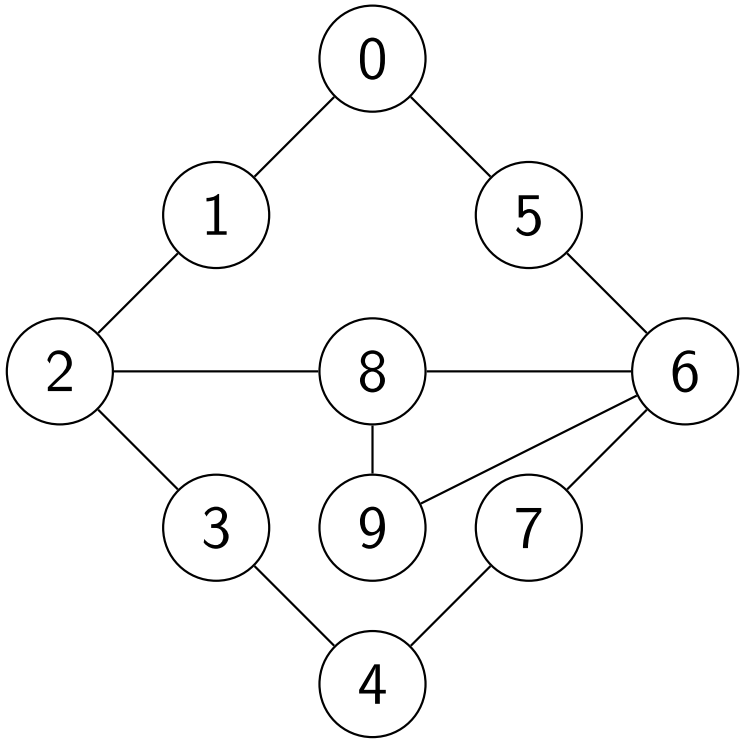
What would the output be if we performed a breadth-first traversal of the following graph, beginning at vertex 0:



Answer: 0, 1, 5, 2, 6, 3, 8, 7, 9, 4

Breadth-first Traversal: Another Example

What would the output be if we performed a depth-first traversal of the following graph, beginning at vertex 0:



Breadth-first answer: 0, 1, 5, 2, 6, 3, 8, 7, 9, 4
Depth-first answer: 0, 1, 2, 3, 4, 7, 6, 5, 8, 9

Breadth-first Traversal Pseudocode

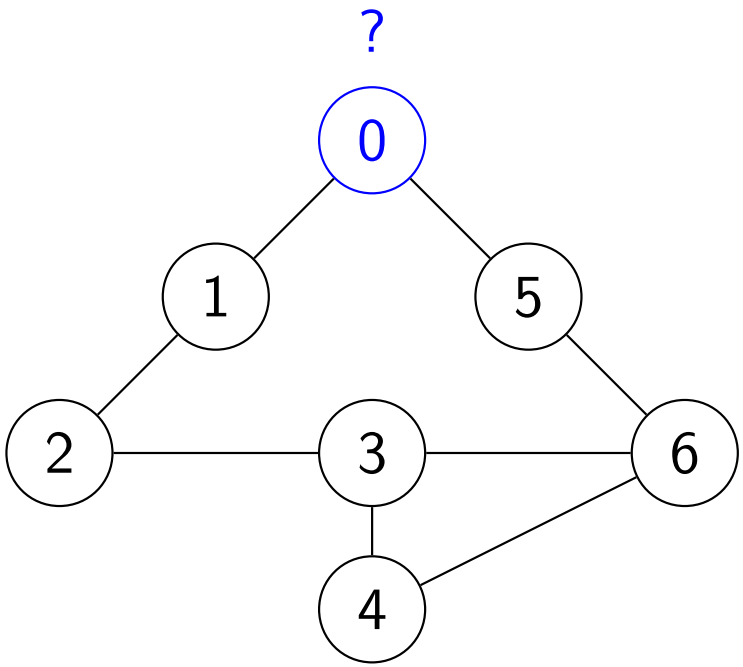
```
public void breadthFirstTraversal(Vertex start) {

  Vertex curr;
  VertexQueue q = new VertexQueue();

  visit start and mark it as visited;
  q.enter( start );
  while (!q.empty()) {
    curr = q.leave();
    for each unvisited vertex v adjacent to curr {
      visit v and mark it as visited;
      q.enter( v );
    } // end for
  } // end while

} // end breadthFirstTraversal
```

Breadth-first Traversal and Paths

- The vertices we passed through to get to a vertex v are no longer on the queue when v is visited and placed on the queue.
- However, we can reconstruct the path to v if we not only mark a vertex as visited, but also record what vertex it is adjacent to when it is visited (i.e., each vertex would have not only a "visited" bit, but also a "previous vertex" pointer).
- When the traversal is finished, we can trace the path from the starting vertex to vertex v backwards, by starting at v and following previous pointers back to the starting vertex.

Breadth-First Traversal Paths

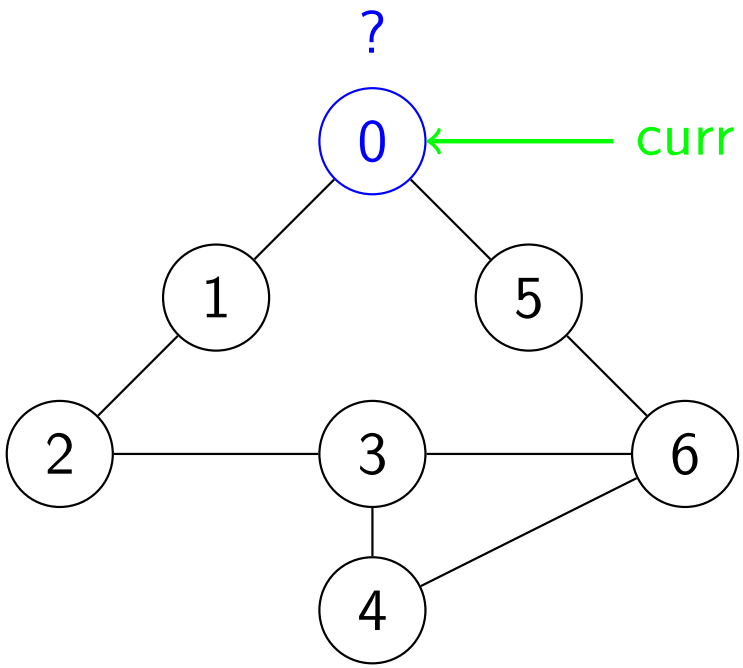
For example, beginning at vertex 0:



Queue: 0

Breadth-First Traversal Paths

Vertex 0 is first out of the queue:

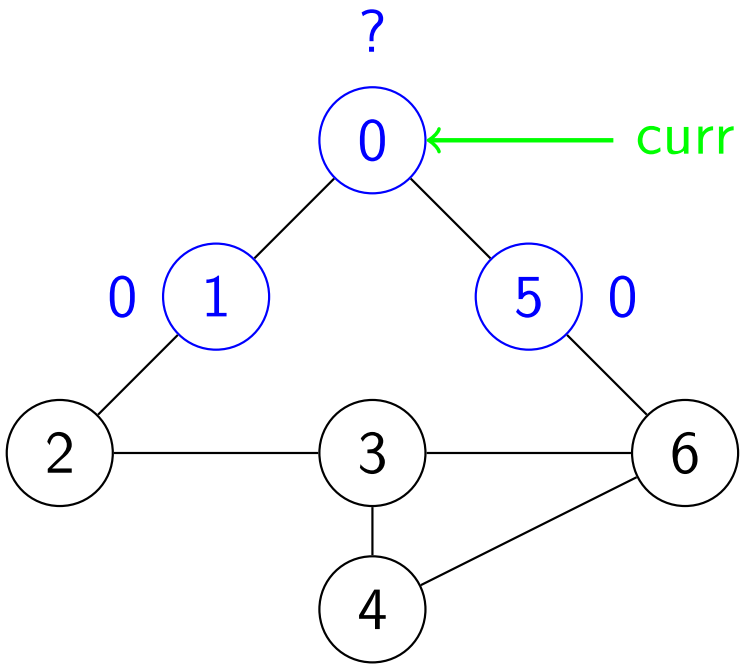


Queue:

We will visit 1 and 5 next, marking "0" as their previous vertex.

Breadth-First Traversal Paths

We're currently at vertex 0:

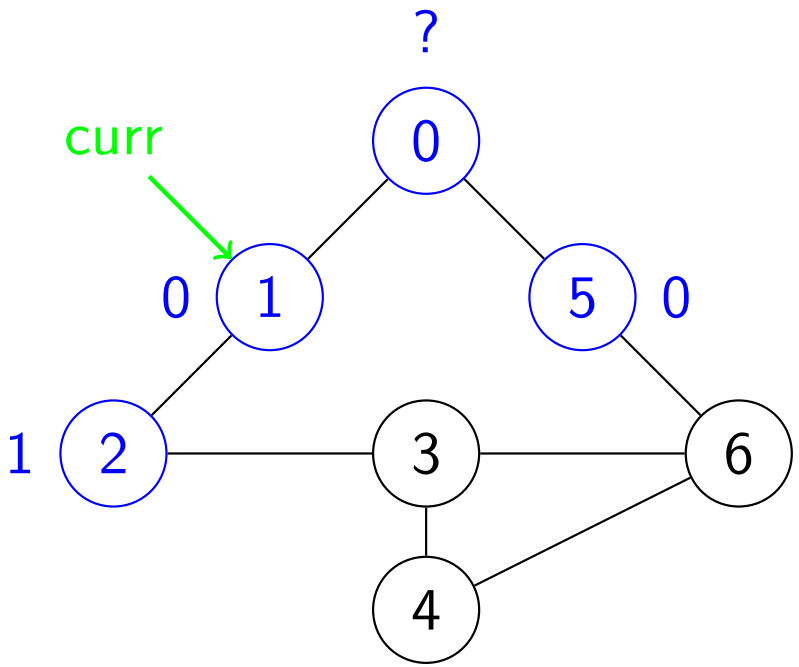


Queue:

1	5

Breadth-First Traversal Paths

At vertex 1, we visit vertex 2:

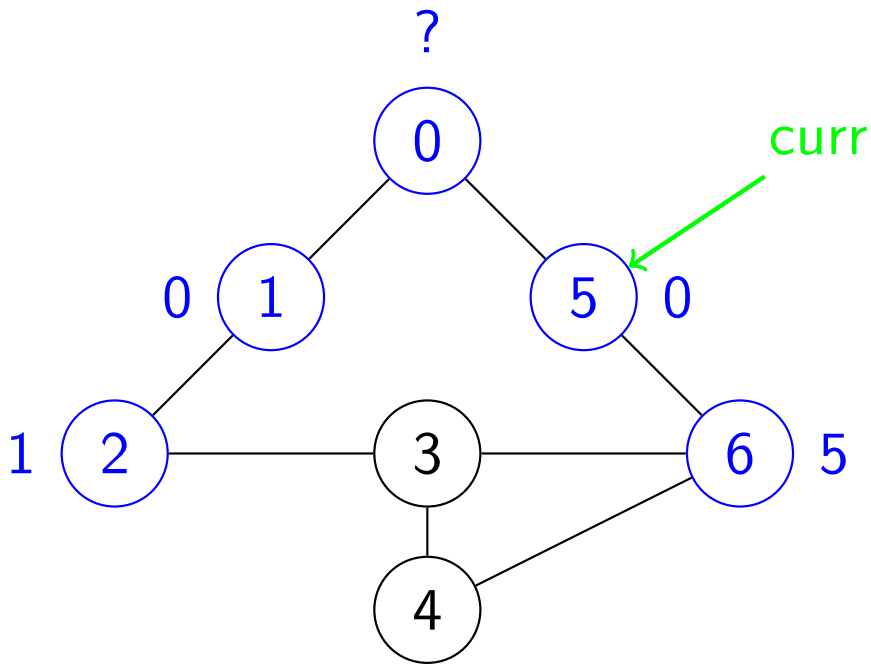


Queue:

5	2

Breadth-First Traversal Paths

At vertex 5, we visit vertex 6:

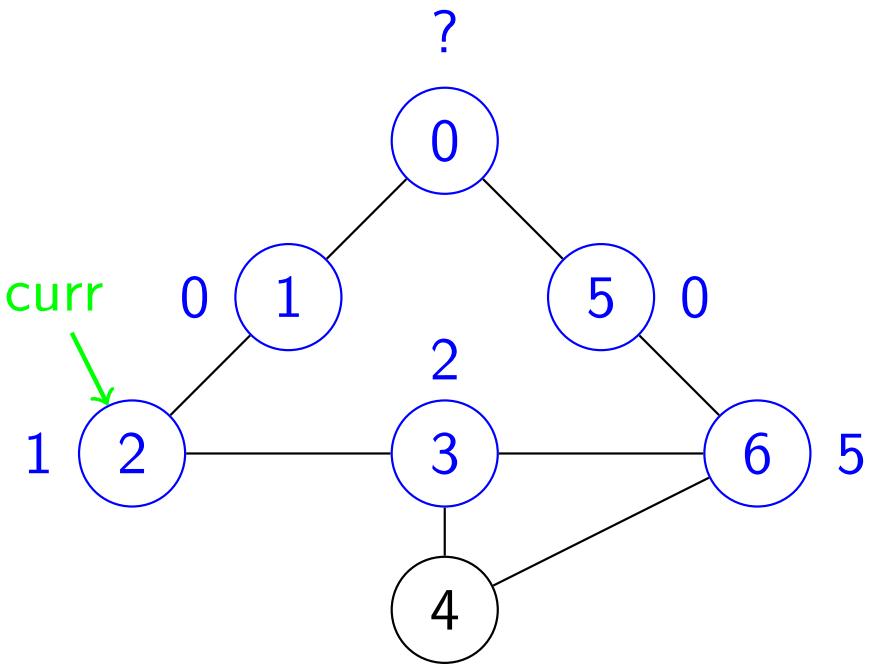


Queue:

2	6

Breadth-First Traversal Paths

At vertex 2, we visit vertex 3:

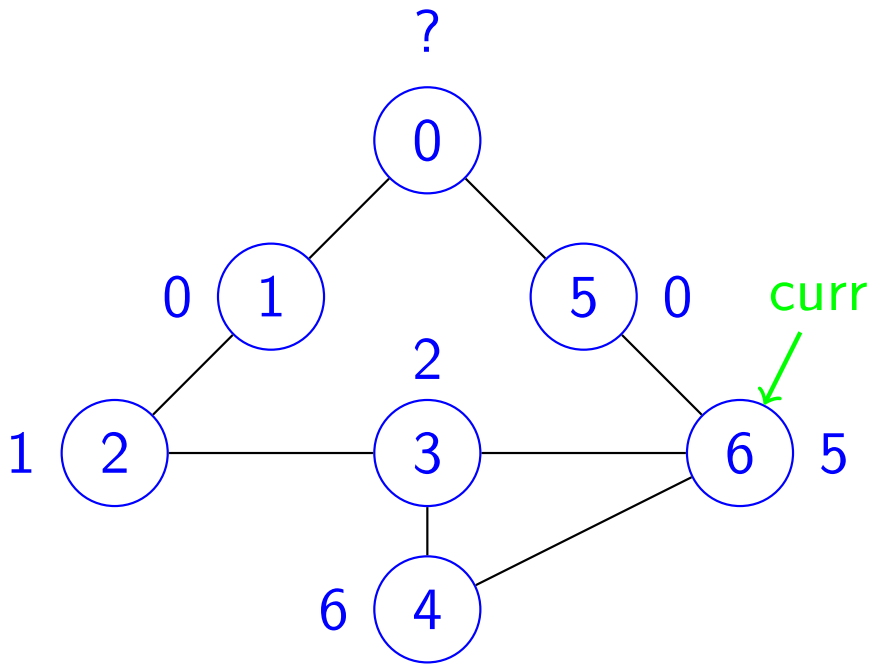


Queue:

6	3

Breadth-First Traversal Paths

At vertex 6, we visit vertex 4:



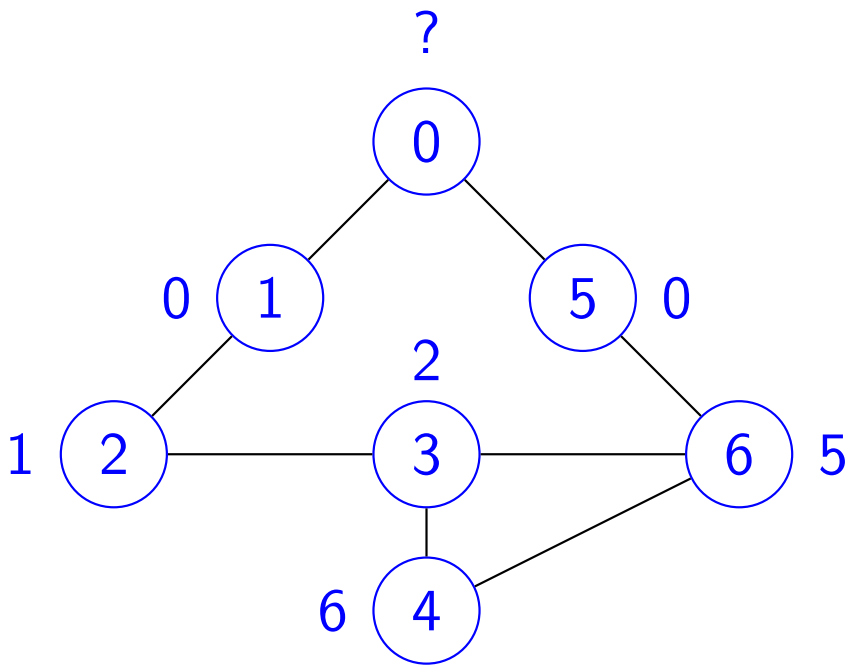
Queue:

3	4

When we remove 3 and 4 from the queue, there is nothing left to visit, so we will skip those steps.

Breadth-First Traversal Paths

Suppose we want to find the path taken to 4 from 0:

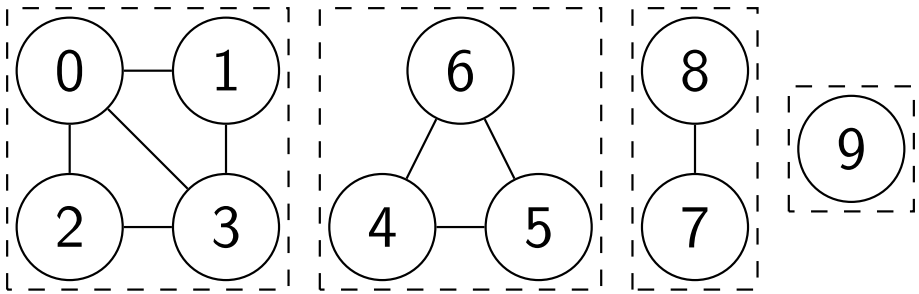


We follow "previous vertex" pointers starting from vertex 4, which gives us the path from 0 to 4 backwards:

$$4 \leftarrow 6 \leftarrow 5 \leftarrow 0$$

Connected Component

An unconnected graph can be divided into connected components. Two vertices i and j are in the same connected component if there is a path from i to j.



The above unconnected graph has 4 connected components (inside dashed rectangles).

Traversals and Unconnected Graphs

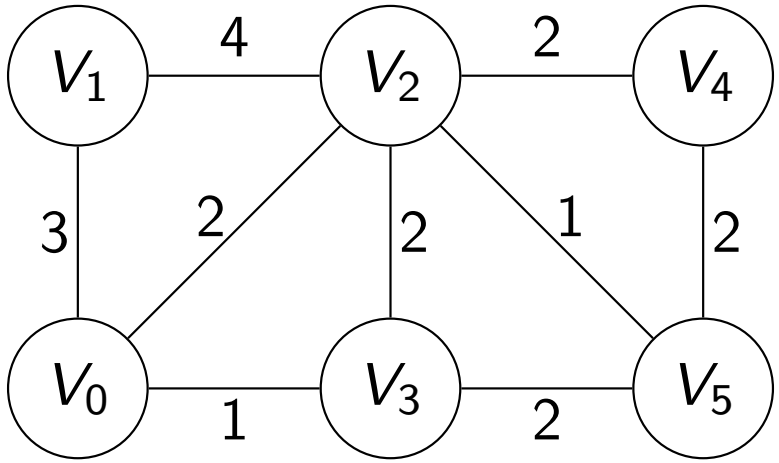
- A traversal starts at a vertex v and visits all the vertices that can be reached by paths from v.
- If the graph is unconnected, then a traversal will visit all the vertices in the same component as v.
- To visit the whole graph, we must repeatedly find a vertex that has not yet been visited and perform a traversal from that vertex, until all vertices have been visited.

Traversals and Paths

- While depth-first search can find a path from the start vertex to another vertex, it does not find the shortest path (the path with the fewest edges).
- Breadth-first search also finds a path, and the path is the shortest path.
- However, things are different if the graph is a weighted graph.

Paths in Weighted Graphs

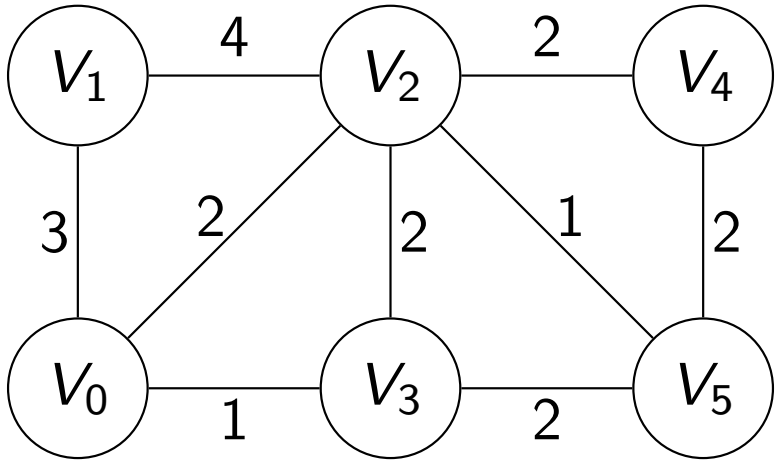
The cost of a path in a weighted graph is the sum of the weights on the edges in the path.



For example, the cost of path V_0, V_1, V_2, V_5 is $3 + 4 + 1 = 8$.

Paths in Weighted Graphs

A shortest path from v_i to v_j in a weighted graph is a path with the lowest cost.



For example, a shortest path from V_0 to V_5 is V_0, V_3, V_5 with a cost of $1 + 2 = 3$. (V_0, V_2, V_5 is also a shortest path in this example.)

Dijkstra's Algorithm

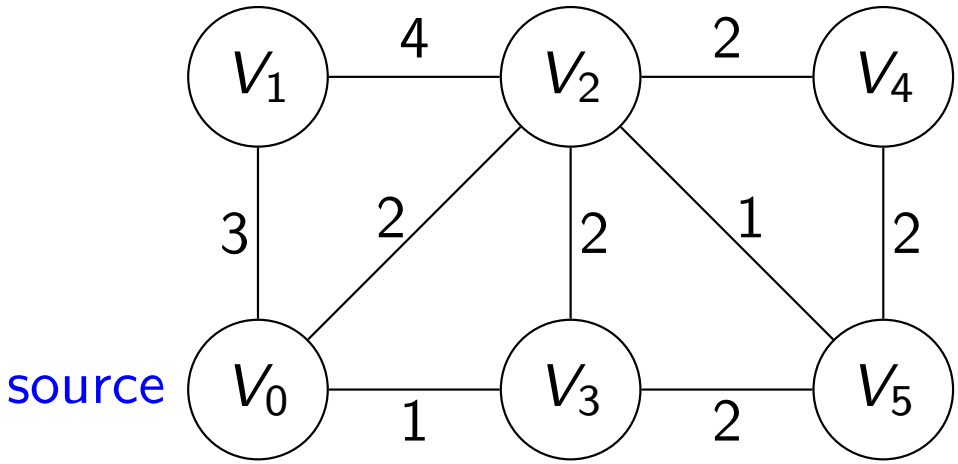
- Finding the shortest path from a vertex to every other vertex in a weighted graph is called the single-source shortest-path problem.
- A breadth-first traversal does not necessarily find the shortest path in a weighted graph.
- Dijkstra's algorithm solves the single-source shortest-path problem.
- It is a greedy algorithm: Dijkstra's algorithm makes the locally optimal choice at each step to construct a globally optimal solution.

Dijkstra's Algorithm

For each vertex, keep track of

- Cost: The cost of the cheapest path we've found to this vertex from the source (the start vertex) so far.
- Previous: The previous vertex (i.e., the second-last vertex) on the cheapest path we've found to this vertex from the source so far.
- Done: A bit telling us whether we've found the shortest path to this vertex yet (or whether there are still some alternative paths to this vertex to consider).

Dijkstra's Algorithm: Initial Setup



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	∞	∞	∞	∞	∞
Previous	-	-	-	-	-	-
Done	F	F	F	F	F	F

Dijkstra's Algorithm

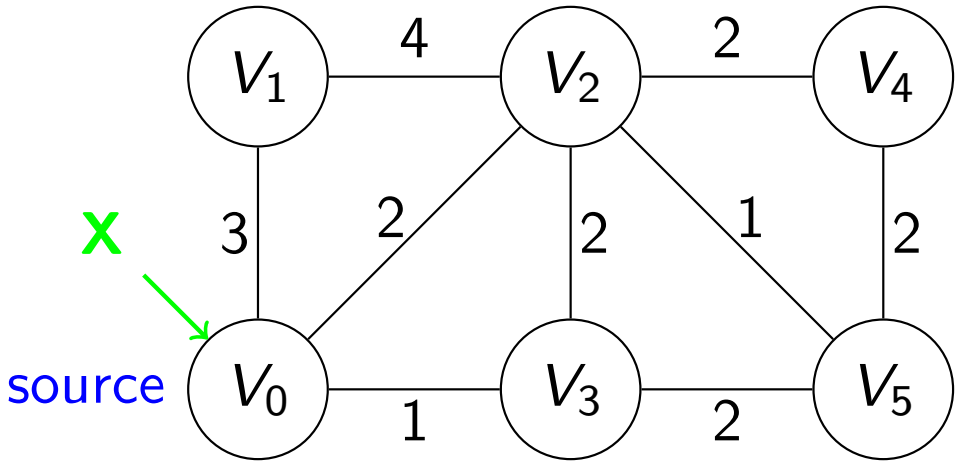
At each step:

- Find the vertex x that has the cheapest cost among vertices that are not done.
- Mark x as done — the cheapest path to x from the source has already been found.
- For each vertex y adjacent to x
  - Calculate the cost of getting to v by taking the cheapest path to x and then the edge from x to v (i.e., add the cost of the cheapest path to x and the weight of the edge from x to v).
  - If the cost of the path through x to v is less than the current cost recorded for v
    - Replace the cost recorded for v with the cost of the path through x.
    - Set v's previous vertex to x.

Stop when all vertices are marked as done.

Dijkstra's Algorithm: First Step

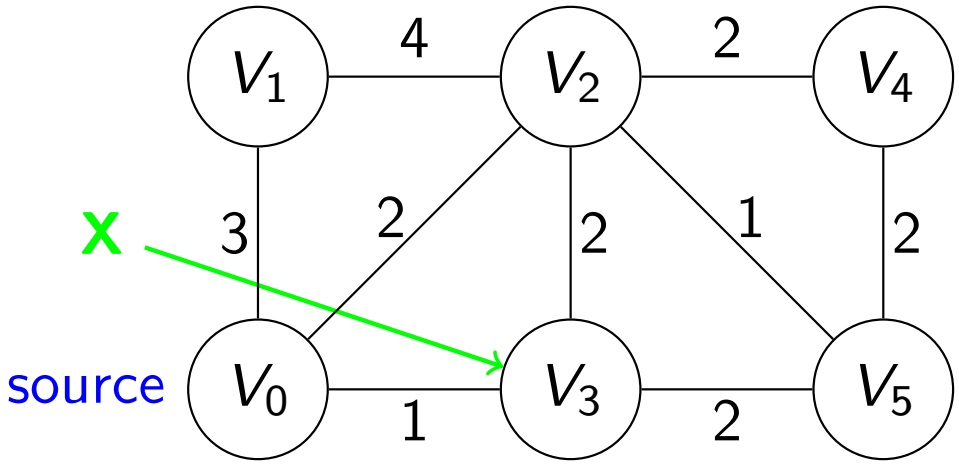
V_0 is the vertex that has the cheapest cost among vertices that are not done.



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	**3**	**2**	**1**	∞	∞
Previous	-	**0**	**0**	**0**	-	-
Done	**T**	F	F	F	F	F

Dijkstra's Algorithm: Second Step

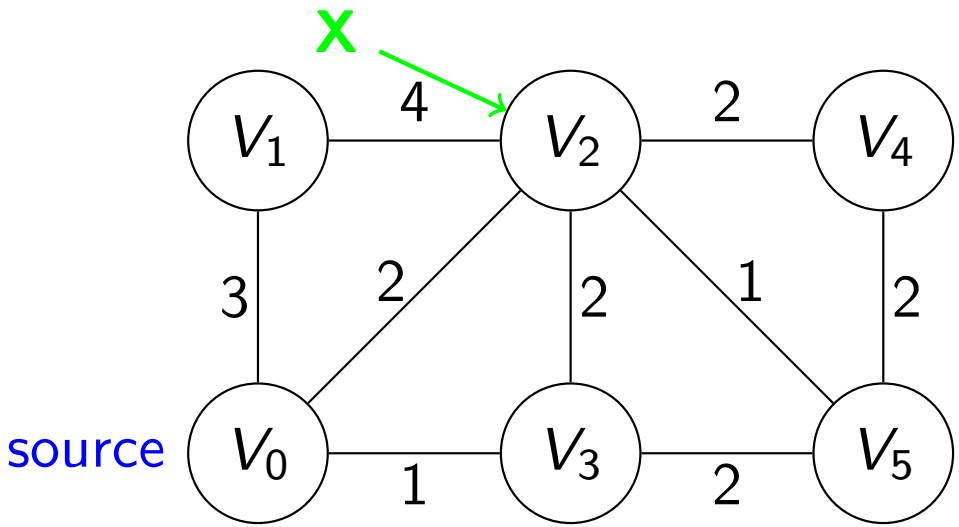
V_3 is the vertex that has the cheapest cost among vertices that are not done.



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	3	2	1	∞	**3**
Previous	-	0	0	0	-	**3**
Done	T	F	F	**T**	F	F

Dijkstra's Algorithm: Third Step

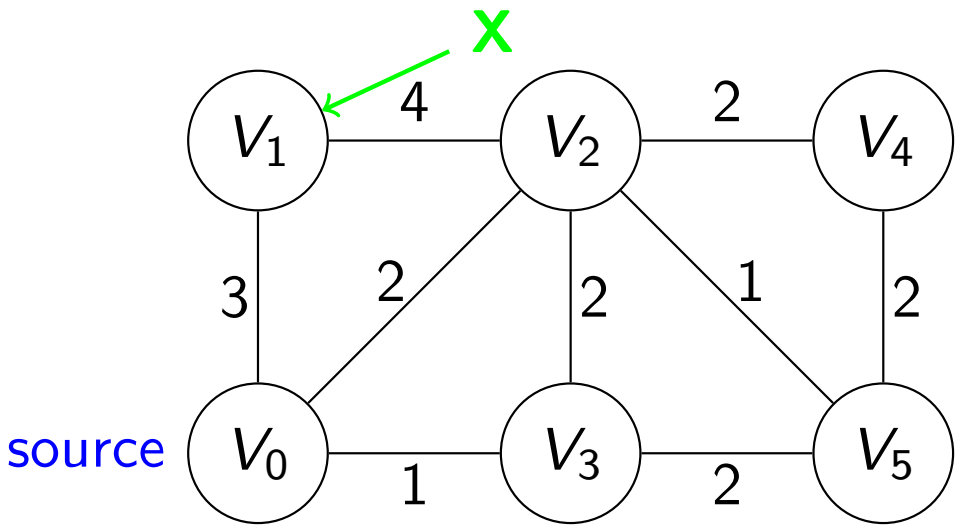
V_2 is the vertex that has the cheapest cost among vertices that are not done.



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	3	2	1	**4**	3
Previous	-	0	0	0	**2**	3
Done	T	F	**T**	T	F	F

Dijkstra's Algorithm: Fourth Step

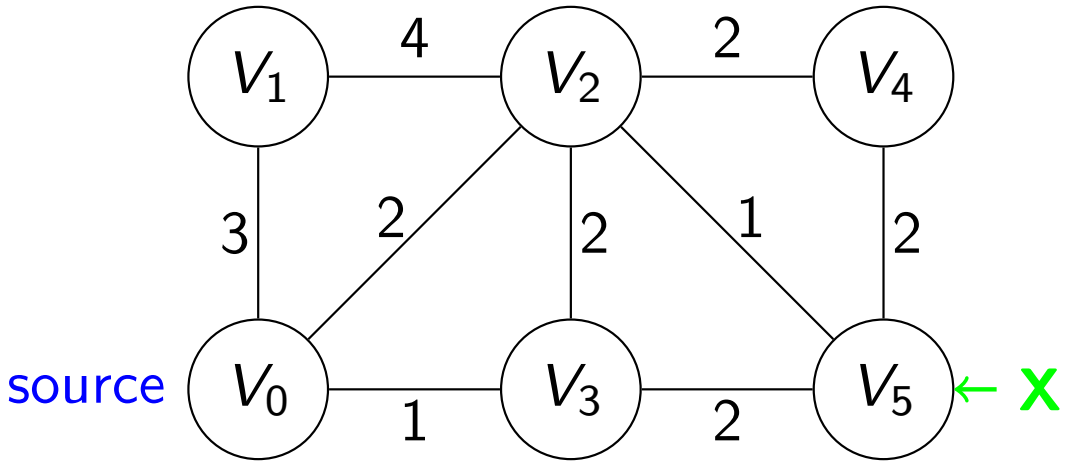
V_1 is a vertex that has the cheapest cost among vertices that are not done.



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	3	2	1	4	3
Previous	-	0	0	0	2	3
Done	T	**T**	T	T	F	F

Dijkstra's Algorithm: Fifth Step

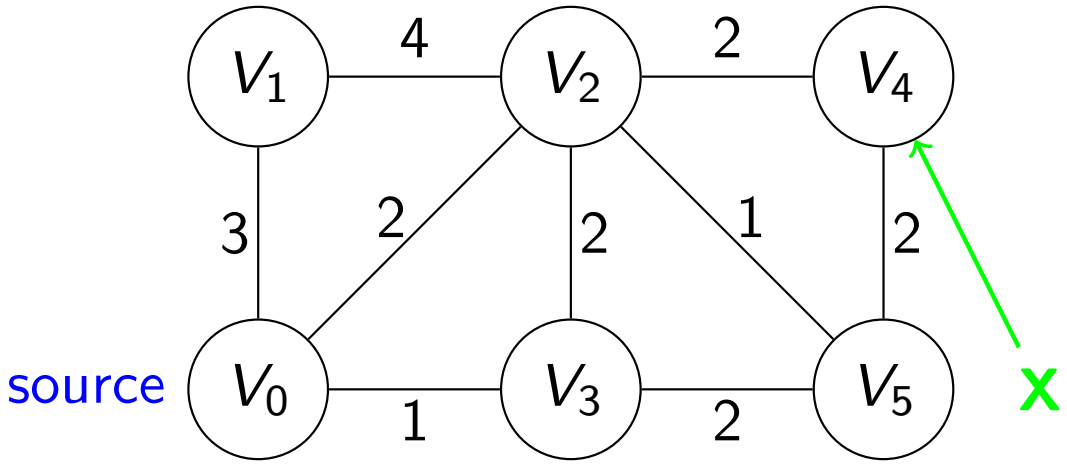
V_5 is the vertex that has the cheapest cost among vertices that are not done.



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	3	2	1	4	3
Previous	-	0	0	0	2	3
Done	T	T	T	T	F	**T**

Dijkstra's Algorithm: Sixth Step

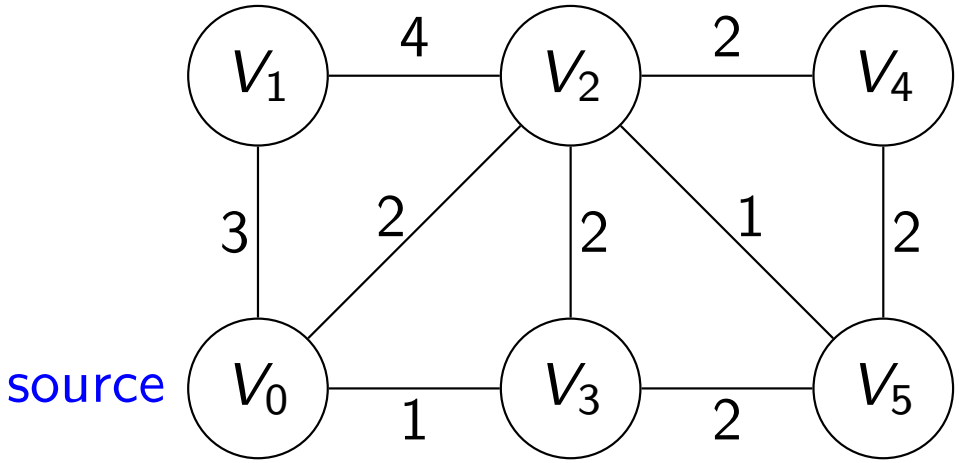
V_4 is the vertex that has the cheapest cost among vertices that are not done.



	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	3	2	1	4	3
Previous	-	0	0	0	2	3
Done	T	T	T	T	**T**	T

Extracting Paths

Once the algorithm completes, the path to any vertex v can be traced backward to the source by following the previous vertex pointers.

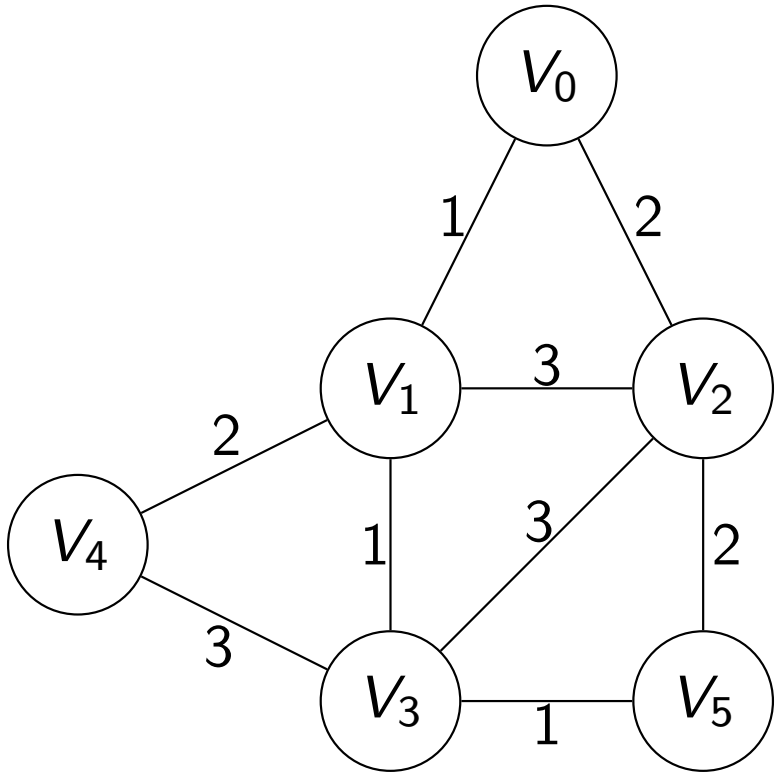


	V_0	V_1	V_2	V_3	V_4	V_5
Cost	0	3	2	1	4	3
Previous	-	0	0	0	2	3
Done	T	T	T	T	T	T

For example, the shortest path from V_0 to V_5 is (in reverse) $V_5 \leftarrow V_3 \leftarrow V_0$.

Another Example

Find the shortest paths to all vertices from V_0:

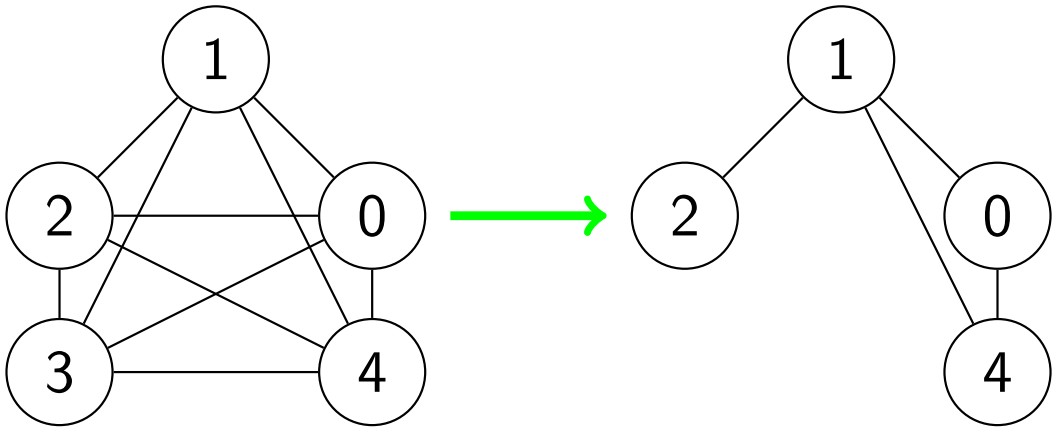


Efficiency of Dijkstra's Algorithm

- If we use an adjacency matrix, Dijkstra's algorithm takes $O(k^2)$ time, where k is the number of vertices.
- If we use an adjacency list, Dijkstra's algorithm can achieve $O(n \log k)$ time, where n is the number of edges and k is the number of vertices. (This implementation is better if $n < k^2$.)

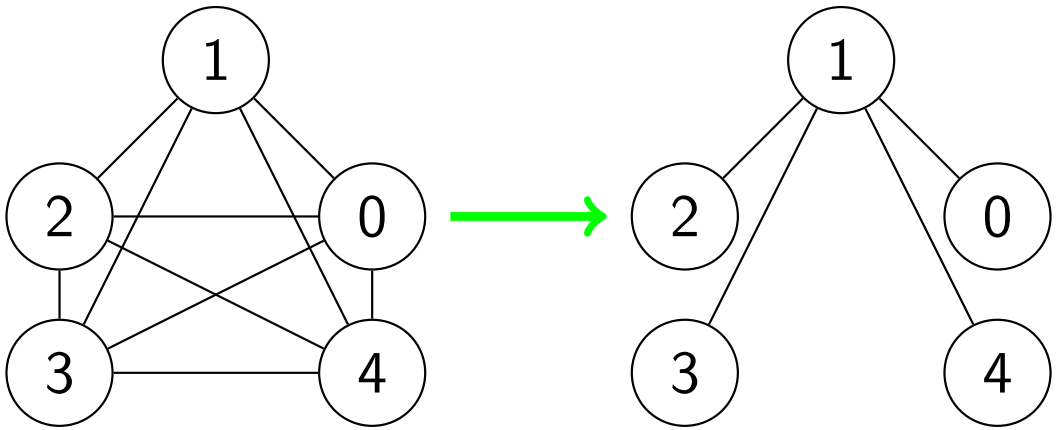
Subgraphs

A subgraph of a graph is a subset of the vertices and edges.



Spanning Tree

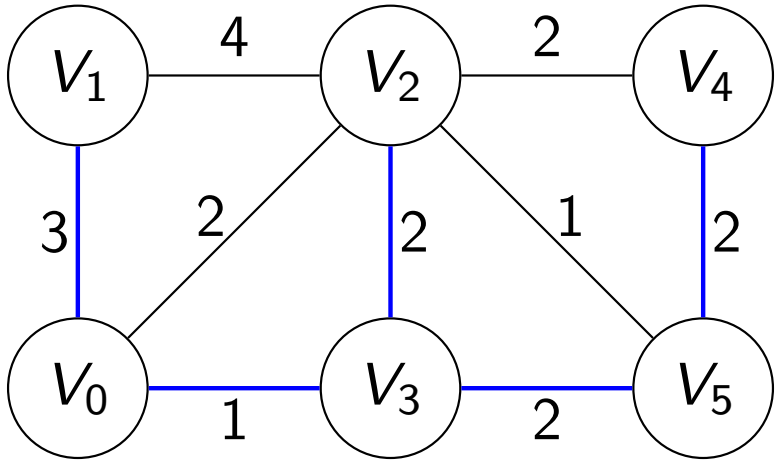
A spanning tree is a subgraph that contains all the vertices of the original graph and is a tree (a connected graph that contains no cycles).



- Often, a graph has many different spanning trees.

Spanning Trees and Weighted Graphs

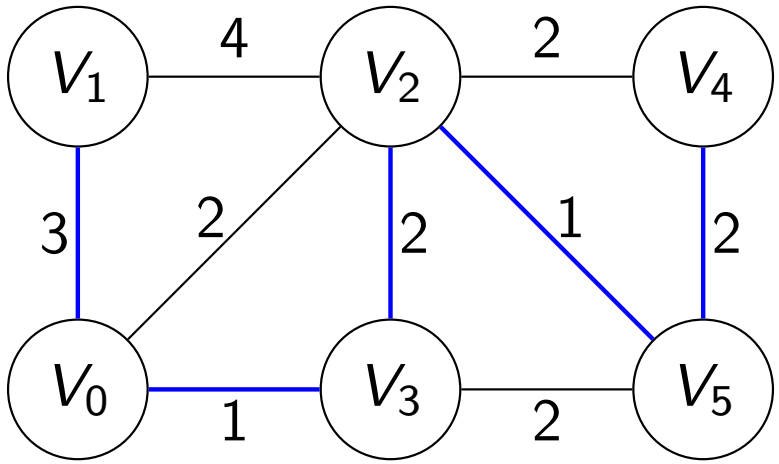
The cost of a spanning tree in a weighted graph is the sum of the costs of the edges in the spanning tree.



The spanning tree given by the blue edges has cost $3 + 1 + 2 + 2 + 2 = 10$.

Minimum Spanning Tree

A minimum spanning tree of a weighted graph is a spanning tree with the minimum cost.



This minimum spanning tree has cost $3 + 1 + 2 + 1 + 2 = 9$.

Minimum Spanning Tree

Minimum spanning trees are useful in constructing networks:

- A minimum spanning tree gives the way to connect a set of points with the smallest total amount of wire.

Many uses for minimum spanning trees are collected at `http://www.ics.uci.edu/~eppstein/gina/mst.html`.

Prim's Algorithm

- Prim's algorithm finds a minimum spanning tree (MST) for a given weighted graph.
- Prim's algorithm builds the MST one vertex at a time.
- Prim's algorithm keeps track of:
  - U, the set of vertices included in the MST so far, and
  - $V - U$, the set of vertices not in the MST yet and, for each such vertex v in $V - U$, the cheapest edge connecting v to any vertex in U.

One Step in Prim's Algorithm

We begin with $U = \{V_0\}$. Then, at each step, we

- Look at all the vertices in $V - U$.
- Choose the vertex V_j with the cheapest edge $\langle V_i, V_j \rangle$, where V_i is in U and V_j is in $V - U$.
- Then add V_j to U.

After k steps (where k is the number of vertices), the chosen vertices and their cheapest edges form a MST for the graph.

Prim's Algorithm Uses a Priority Queue

To find a vertex in $V - U$ with the cheapest edge connecting it to some vertex in U:

- Keep the vertices in $V - U$ in a priority queue.
- The priority of a vertex v in the priority queue is the cost of the cheapest edge connecting v and some vertex in U. (If there is no such edge, then v's priority is infinity.)
- Record the edge that gives a vertex its priority, because when a vertex is removed from the priority queue, its recorded edge becomes a spanning tree edge.

Goal: a cheapest edge $\longrightarrow$ the lowest cost is the highest priority (use a min heap).

Prim's Algorithm Pseudocode

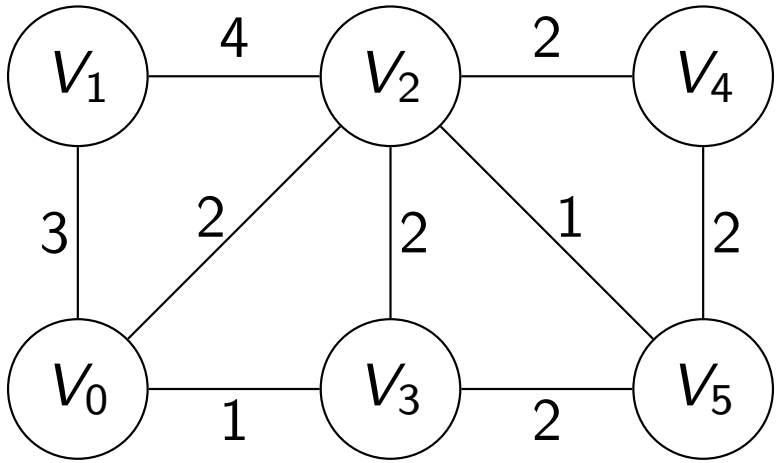
```
// First, initialize U and the priority queue.

U = { V0 };

for each other vertex Vi ≠ V0 {
  if there's an edge e = ⟨ V0, Vi ⟩ {
     priority of Vi = the cost of edge e;
     the edge recorded for Vi = e;
  } // end if
  else
    priority of Vi = ∞;
} // end for

Now heapify all the vertices not in U into
  a heap called vertexPQ;
```

Prim's Algorithm Pseudocode

```
// Now construct the MST one vertex and edge at a time.

while vertexPQ is not empty {
  vertex v = the highest priority vertex in vertexPQ;
  U = U ∪ { v };
  for every vertex x adjacent to v {
    if x is not in U   // x is in vertexPQ
      if the cost of edge ⟨ v, x ⟩ < priority of x {
        priority of x = cost of edge ⟨ v, x ⟩;
        sift up x in vertexPQ;
      } // end if
  } // end for
} // end while
```

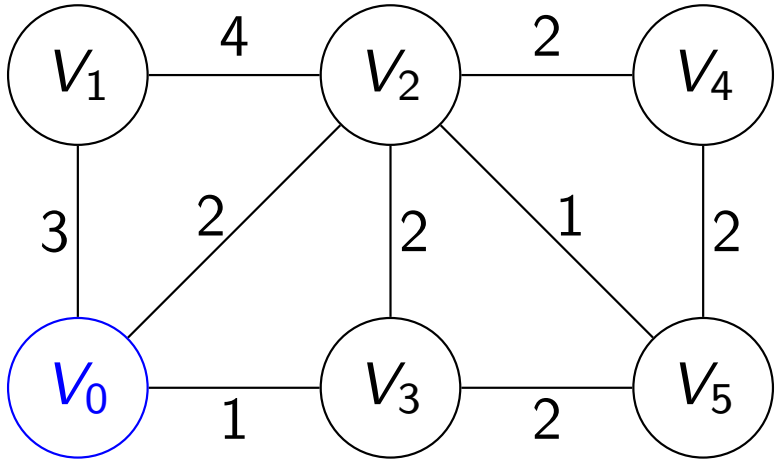
Prim's Algorithm: Example

Find a minimum spanning tree of the following weighted graph using Prim's algorithm.



Prim's Algorithm: Example

Start with $U = V_0$:

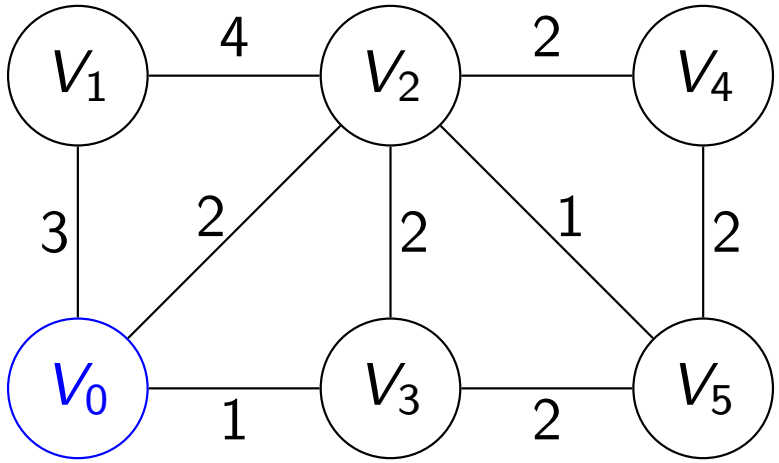


Vertex:	V_3	V_2	V_1	V_4	V_5
Priority:	1	2	3	∞	∞
Recorded edge:	$\langle V_0, V_3 \rangle$	$\langle V_0, V_2 \rangle$	$\langle V_0, V_1 \rangle$	—	—

(I have represented the priority queue as an ordered list, rather than as a heap, and the vertices in U are blue.)

Prim's Algorithm: Example

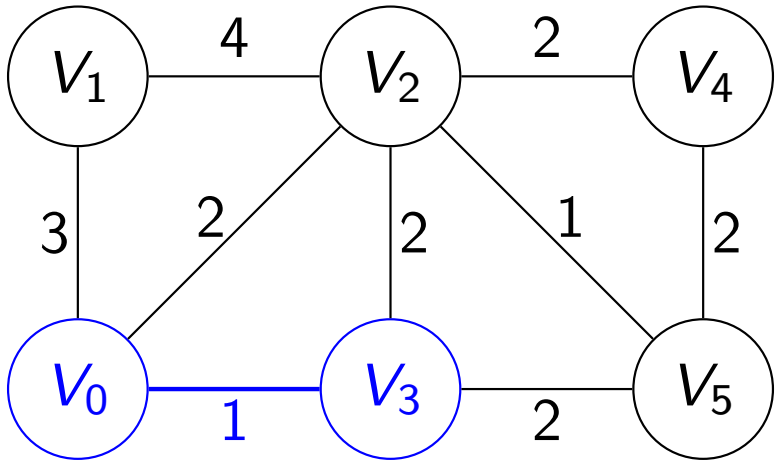
Now, begin to construct the MST. V_3 will be the first chosen vertex.



Vertex:	V_3	V_2	V_1	V_4	V_5
Priority:	1	2	3	∞	∞
Recorded edge:	$\langle V_0, V_3 \rangle$	$\langle V_0, V_2 \rangle$	$\langle V_0, V_1 \rangle$	—	—

Prim's Algorithm: Example

V_3 and its recorded edge are added to the MST:

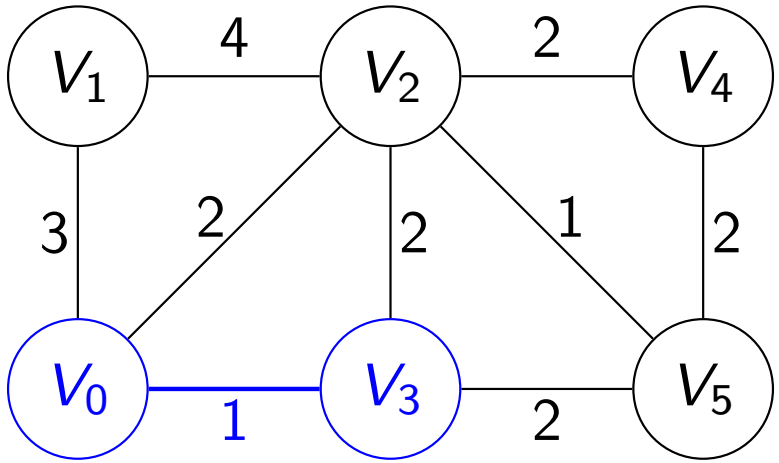


Vertex:	V_2	V_1	V_4	V_5
Priority:	2	3	∞	∞
Recorded edge:	$\langle V_0, V_2 \rangle$	$\langle V_0, V_1 \rangle$	—	—

Now check if we should change the priorities of vertices in the priority queue that are adjacent to V_3.

Prim's Algorithm: Example

V_5's priority and its recorded edge are changed:

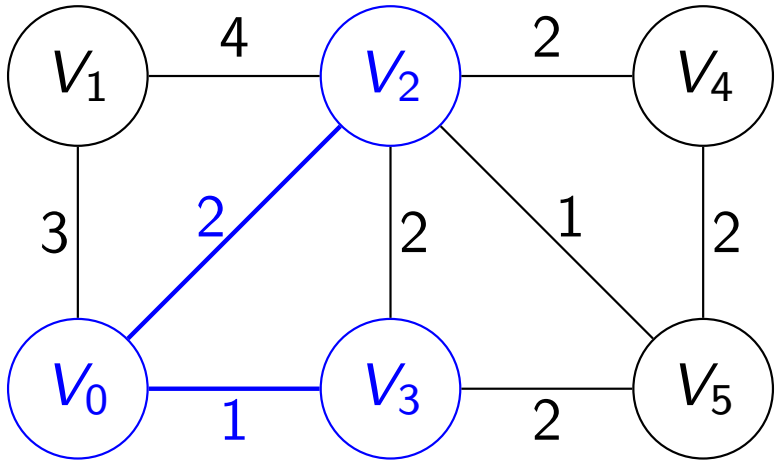


Vertex:	V_2	V_5	V_1	V_4
Priority:	2	2	3	∞
Recorded edge:	$\langle V_0, V_2 \rangle$	$\langle V_3, V_5 \rangle$	$\langle V_0, V_1 \rangle$	—

Now take the highest priority vertex out the priority queue again.

Prim's Algorithm: Example

V_2 and its recorded edge are added to the MST:

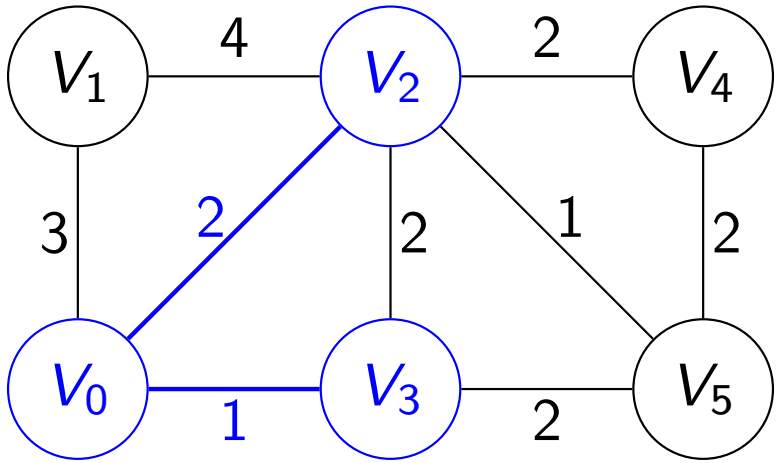


Vertex:	V_5	V_1	V_4
Priority:	2	3	∞
Recorded edge:	$\langle V_3, V_5 \rangle$	$\langle V_0, V_1 \rangle$	—

The priorities of V_4 and V_5 (and their recorded edges) must now be changed.

Prim's Algorithm: Example

V_4 and V_5 are changed in the priority queue:

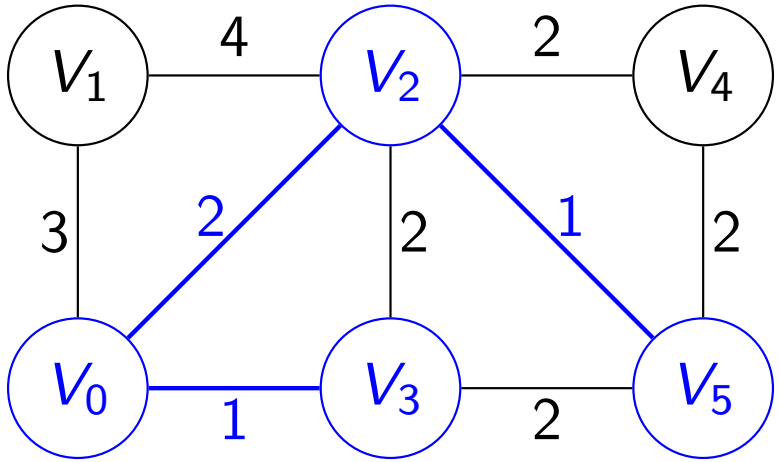


Vertex:	V_5	V_4	V_1
Priority:	1	2	3
Recorded edge:	$\langle V_2, V_5 \rangle$	$\langle V_2, V_4 \rangle$	$\langle V_0, V_1 \rangle$

Loop again, removing the highest priority vertex from the priority queue.

Prim's Algorithm: Example

V_5 and edge $\langle V_2, V_5 \rangle$ are added to the MST:

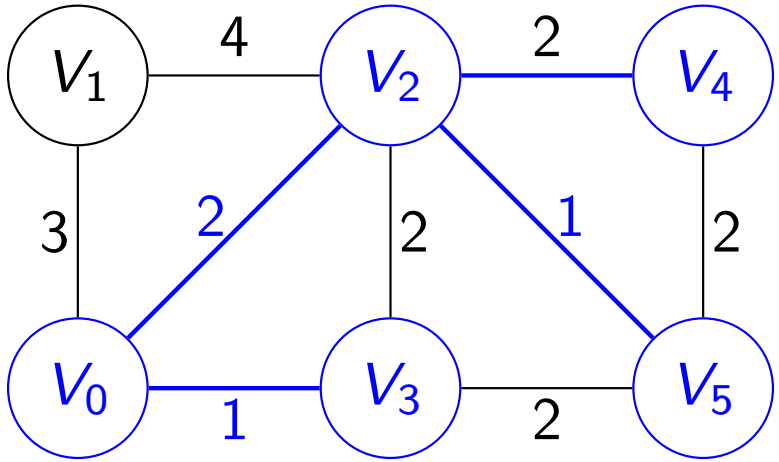


Vertex:	V_4	V_1
Priority:	2	3
Recorded edge:	$\langle V_2, V_4 \rangle$	$\langle V_0, V_1 \rangle$

This time, no priorities in the priority queue need to be changed. Loop again, removing the highest priority vertex from the priority queue.

Prim's Algorithm: Example

V_4 and edge $\langle V_2, V_4 \rangle$ are added to the MST:

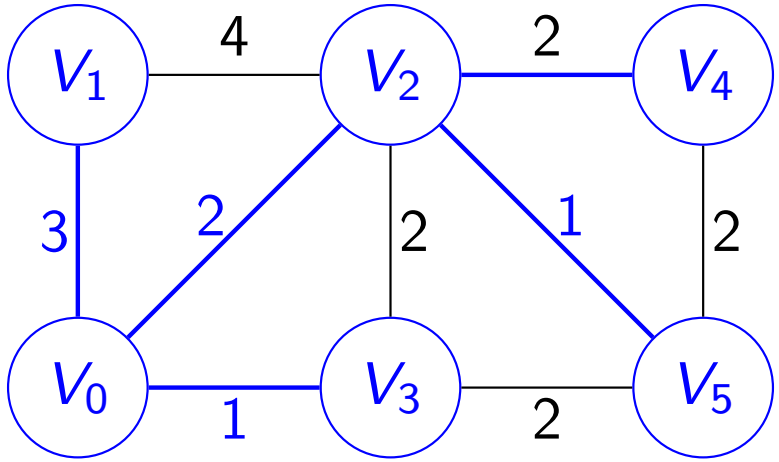


Vertex:	V_1
Priority:	3
Recorded edge:	$\langle V_0, V_1 \rangle$

Again, no priorities in the priority queue need to be changed. Loop one last time, removing the highest priority vertex from the priority queue.

Prim's Algorithm: Example

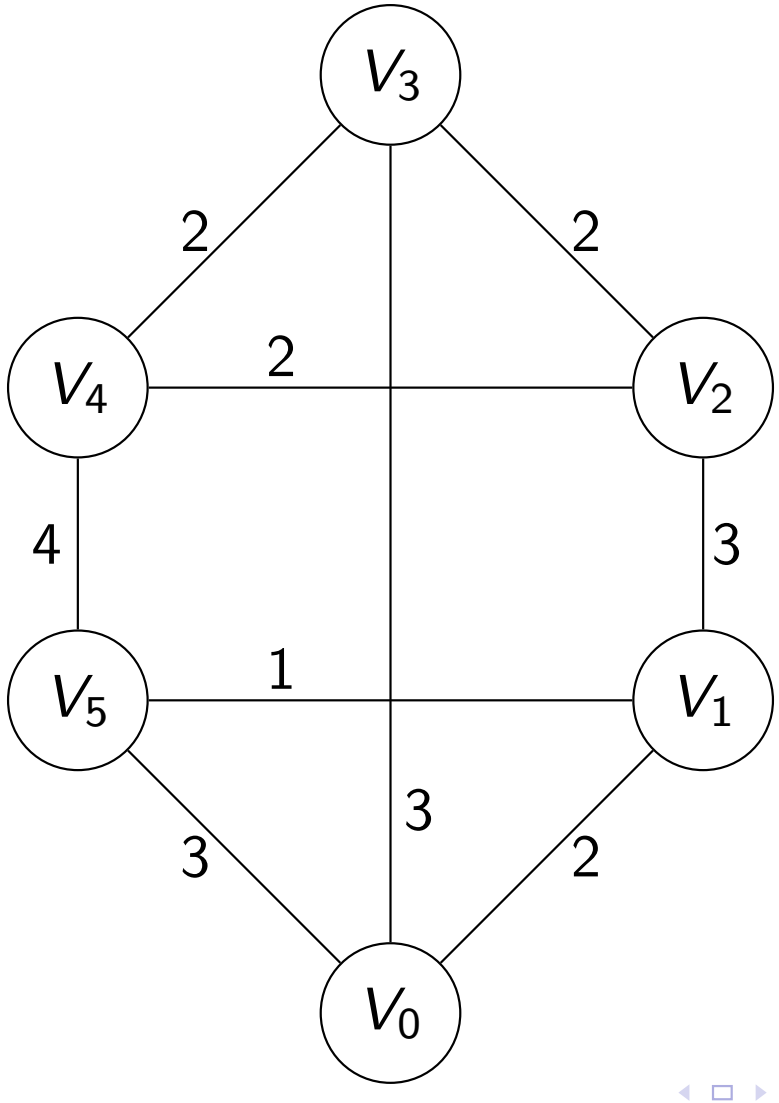
V_1 and edge $\langle V_0, V_3 \rangle$ are added to the MST:



The priority queue is empty, so we have a MST!

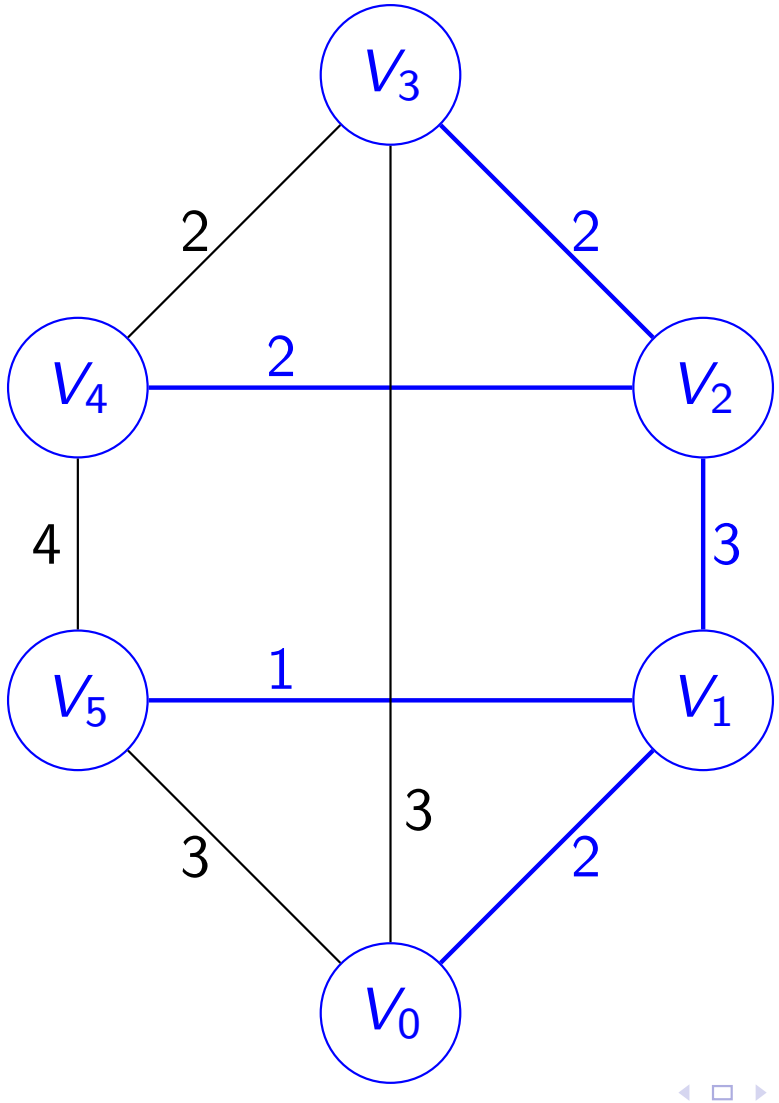
Prim's Algorithm: Another Example

Find a MST for the following weighted graph, starting with $U = \{V_0\}$.



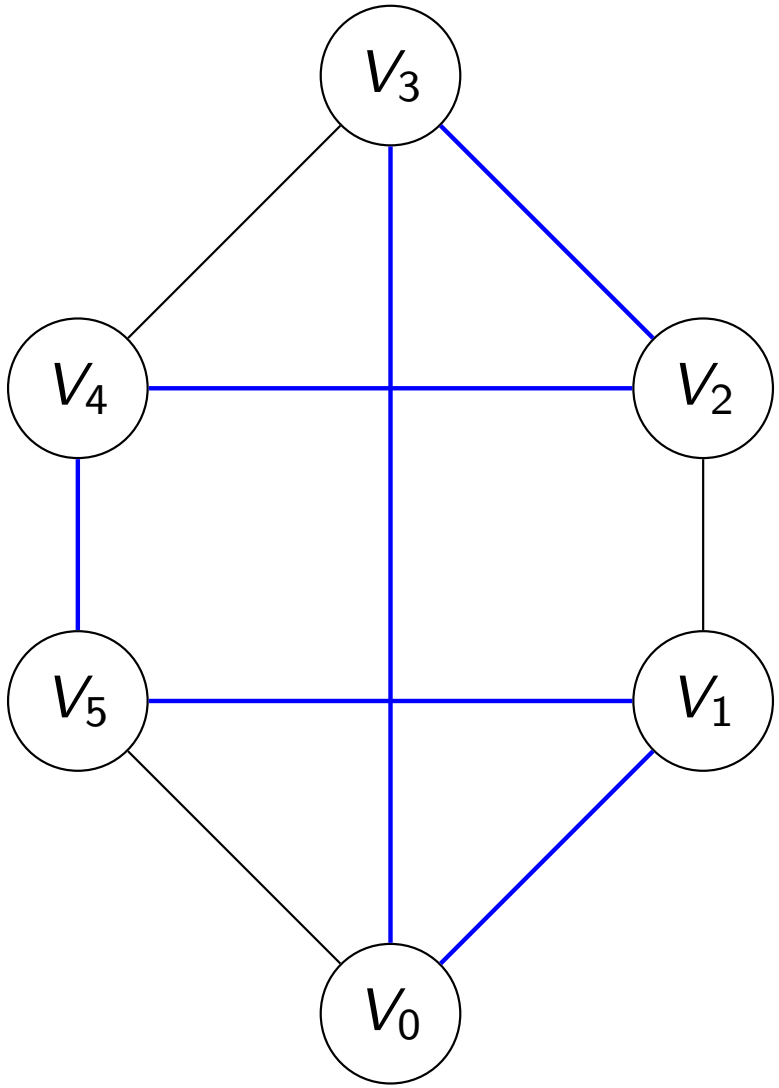
Prim's Algorithm: Another Example

Find a MST for the following weighted graph, starting with $U = \{V_0\}$.



Hamiltonian Circuit

A Hamiltonian circuit is a cycle that goes through every vertex in the graph exactly one.



The Traveling Salesperson Problem

The traveling salesperson problem is the problem of finding the minimum cost Hamiltonian circuit in a weighted graph.

- Imagine that the vertices are cities and the edges are highways between cities.
- The weight on an edge could be the cost of traveling between the two cities or the distance between the two cities.
- In the traveling salesperson problem, you are trying to find the cheapest route (or the route with the least distance traveled) for a salesperson who must visit all the cities, returning to home base at the end.

Traveling Salesperson Problem: An Intractable Problem

- Currently, nobody knows a polynomial time algorithm that can solve the Traveling Salesperson Problem — it is an intractable problem.
- The best known algorithm for solving this problem takes $O(2^n)$ time (in a graph with n vertices).
- Simplest idea: Try all possible orderings of the n vertices in the graph as potential routes.
  - There are $n!$ different orderings.
  - Therefore, this idea takes $O(n!)$ time.
  - $10! = 3{,}628{,}800$.
- Both $O(n^2)$ and $O(n!)$ are unacceptably slow running times.

Growth Rates versus Input Size

Recall this table from Garey and Johnson's book Computers and Intractability shows the time taken for various algorithm running times ($f(n)$) on
various sizes of inputs (n). We assume 1 microsecond per operation.

	n					
$f(n)$	10	20	30	40	50	60
n	.00001s	.00002s	.00003s	.00004s	.00005s	.00006s
n^2	.0001s	.0004s	.0009s	.0016s	.0025s	.0036s
n^3	.001s	.008s	.027s	.064s	.125s	.216s
n^5	.1s	3.2s	24.3s	1.7m	5.2m	13m
2^n	.001s	1.0s	17.9m	12.7d	35.7yrs	366cen
3^n	.059s	58min	6.5yrs	3,855 cen	2×10^8 cen	1.3×10^{13} cen
$n!$	3.6s	78.5cen				

Intractability, Optimality and Heuristics

- Many problems are as hard as the traveling salesperson problem.
- Just because they are intractable doesn't mean we can ignore them!
- To cope: use some kind of heuristic to get something close to a good solution to an intractable problem, rather than insisting on the optimal solution.
- Example traveling salesperson heuristic: Always travel to the next closest city.
- Using the example heuristic gives an $O(n^2)$ algorithm, but it doesn't necessarily produce the minimum Hamiltonian circuit.

The End of Comp 2140!

- 2nd and 3rd year courses directly building from this material:
  - 2150: Object orientation
  - 3170: Design and analysis of algorithms
  - 3190: Artificial intelligence (heuristic algorithms)
  - Other 3rd year courses such as databases (tree structured indices)
- 4th year courses on algorithms and data structures, computability, AI, graph theory
- Nearly every computer science course after this one assumes you have a good understanding of data structures!