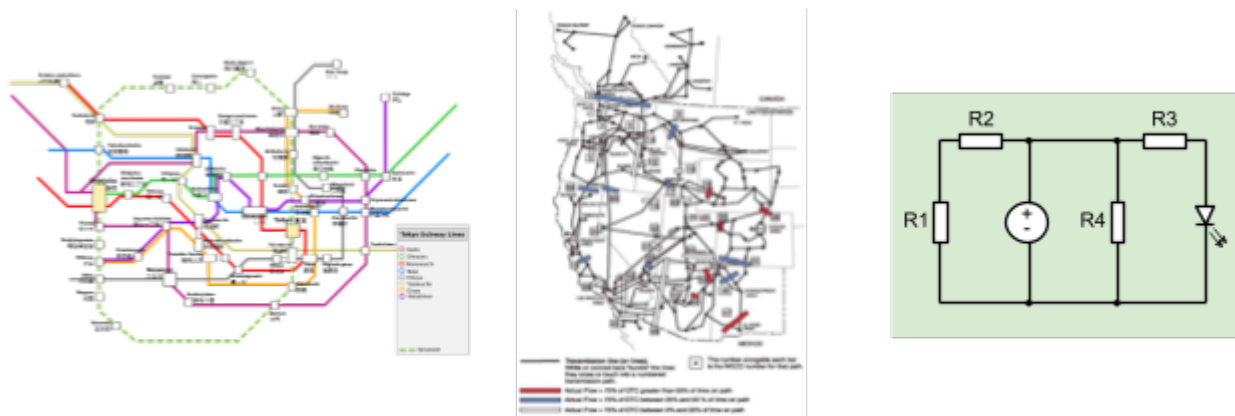# 14.1 Graphs: Introduction

Introduction to graphs

Many items in the world are connected, such as computers on a network connected by wires, cities connected by roads, or people connected by friendship.

Figure 14.1.1: Examples of connected items: Subway map, electrical power transmission, electrical circuit.

Source: Subway map (Comicinker (Own work) / CC-BY-SA-3.0 via Wikimedia Commons), Internet map (Department of Energy / Public domain via Wikimedia Commons), Electrical circuit (zyBooks)

A **graph** is a data structure for representing connections among items, and consists of vertices connected by edges.

- A **vertex** (or node) represents an item in a graph.
- An **edge** represents a connection between two vertices in a graph.

PARTICIPATION ACTIVITY	14.1.1: A graph represents connections among items, like among computers, or people.

Animation captions:

1. Items in the world may have connections, like a computer network.
2. A graph's vertices represent items.
3. A graph's edges represent connections.

4. A graph can represent many different things, like friendships among people. Raj and Maya are friends, but Raj and Jen are not.

For a given graph, the number of vertices is commonly represented as V, and the number of edges as E.

PARTICIPATION ACTIVITY	14.1.2: Graph basics.

Refer to the above graphs.

1) The computer network graph has how many vertices?

   ○ 5

   ○ 6

2) The computer network graph has how many edges?

   ○ 5

   ○ 6

3) Are Maya and Thuy friends?

   ○ Yes

   ○ No

4) Can a vertex have more than one edge?

   ○ Yes

   ○ No

5) Can an edge connect more than two vertices?

   ○ Yes

   ○ No

6) Given 4 vertices A, B, C, and D and at most one edge between a vertex pair, what is the maximum number of edges?

○ 6

Adjacency and paths

In a graph:

- Two vertices are **adjacent** if connected by an edge.
- A **path** is a sequence of edges leading from a source (starting) vertex to a destination (ending) vertex. The **path length** is the number of edges in the path.
- The **distance** between two vertices is the number of edges on the shortest path between those vertices.

| PARTICIPATION ACTIVITY | 14.1.3: Graphs: adjacency, paths, and distance. |

Animation captions:

1. Two vertices are adjacent if connected by an edge. PC1 and Server1 are adjacent. PC1 and Server3 are not adjacent.
2. A path is a sequence of edges from a source vertex to a destination vertex.
3. A path's length is the path's number of edges. Vertex distance is the length of the shortest path: Distance from PC1 to PC2 is 2.

| PARTICIPATION ACTIVITY | 14.1.4: Graph properties. |

Refer to the following graph.
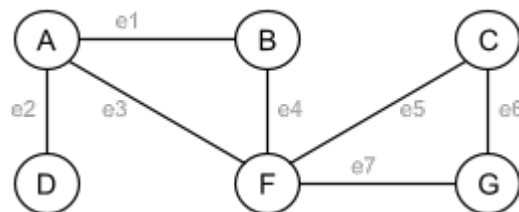
1) A and B are adjacent.

○ True

○ False

2) A and C are adjacent.

○ True

○ False

3) Which of the following is a path from
B to G?

    ○   e3, e7

    ○   e1, e3, e7

    ○   No path from B to G.

4) What is the distance from D to C?

    ○   3

    ○   4

    ○   5

14.2 Applications of graphs

Geographic maps and navigation

Graphs are often used to represent a geographic map, which can contain information about places and travel routes. Ex: Vertices in a graph can represent airports, with edges representing available flights. Edge weights in such graphs often represent the length of a travel route, either in total distance or expected time taken to navigate the route. Ex: A map service with access to real-time traffic information can assign travel times to road segments.

PARTICIPATION ACTIVITY	14.2.1: Driving directions use graphs and shortest path algorithms to determine the best route from start to destination.

Animation content:

undefined

Animation captions:

1. A road map can be represented by a graph. Each intersection of roads is a vertex. Destinations like the beach or a house are also vertices.
2. Roads between vertices are edges. A map service with realtime traffic information can assign travel times as edge weights.
3. A shortest path finding algorithm can be used to find the shortest path starting at the house and ending at the beach.

PARTICIPATION ACTIVITY	14.2.2: Using graphs for road navigation.

1) The longer a street is, the more vertices will be needed to represent that street.

   ○ True

   ○ False

2) When using the physical distance between vertices as edge weights, a shortest path algorithm finds the fastest route of travel.

   ○ True

   ○ False

3) Navigation software would have no need to place a vertex on a road in a location where the road does not intersect any other roads.

   ○ True

   ○ False

4) If navigation software uses GPS to automatically determine the start location for a route, the vertex closest to the GPS coordinates can be used as the starting vertex.

   ○ True

   ○ False

PARTICIPATION ACTIVITY	14.2.3: Using graphs for flight navigation.

Suppose a graph is used to represent airline flights. Vertices represent airports and edge weights represent flight durations.

1) The weight of an edge connecting two airport vertices may change based on _____.

   ○ flight delays

○ weather conditions

○ flight cost

2) Edges in the graph could potentially
be added or removed during a single
day's worth of flights.

○ True

○ False

Product recommendations

A graph can be used to represent relationships between products. Vertices in the graph
corresponding to a customer's purchased products have adjacent vertices representing products
that can be recommended to the customer.

PARTICIPATION ACTIVITY	14.2.4: A graph of product relationships can be used to produce recommendations based on purchase history.

Animation content:

```
undefined
```

Animation captions:

1. An online shopping service can represent relationships between products being sold using a graph.
2. Relationships may be based on the products alone. A game console requires a TV and games, so the game console vertex connects to TV and game products.
3. Vertices representing common kitchen products, such as a blender, dishwashing soap, kitchen towels, and oven mitts, are also connected.
4. Connections might also represent common purchases that occur by chance. Maybe several customers who purchase a Blu-ray player also purchase a blender.
5. A customer's purchases can be linked to the product vertices in the graph.
6. Adjacent vertices can then be used to provide a list of recommendations to the customer.

PARTICIPATION ACTIVITY	14.2.5: Product recommendations.

1) If a customer buys only a Blu-ray
player, which product is not likely to

be recommended?

○ Television 1 or 2

○ Blender

○ Game console

2) Which single purchase would produce the largest number of recommendations for the customer?

○ Tablet computer

○ Blender

○ Game console

3) If "secondary recommendations" included all products adjacent to recommended products, which products would be secondary recommendations after buying oven mitts?

○ Blender, kitchen towels, and muffin pan

○ Dishwashing soap, Blu-ray player, and muffin mix

○ Game console and tablet computer case

Social and professional networks

A graph may use a vertex to represent a person. An edge in such a graph represents a relationship between 2 people. In a graph representing a social network, an edge commonly represents friendship. In a graph representing a professional network, an edge commonly represents business conducted between 2 people.

PARTICIPATION ACTIVITY	14.2.6: Professional networks represented by graphs help people establish business connections.

Animation content:

undefined

Animation captions:

1. In a graph representing a professional network, vertices represent people and edges represent a business connection.
2. Tuyet conducts business with Wilford, adding a new edge in the graph. Similarly, Manuel conducts business with Keira.
3. The graph can help professionals find new connections. Shayla connects with Octavio through a mutual contact, Manuel.

PARTICIPATION ACTIVITY	14.2.7: Professional network.

Refer to the animation's graph representing the professional network.

1) Who has conducted business with Eusebio?

- ○ Giovanna
- ○ Manuel
- ○ Keira

2) If Octavio wishes to connect with Wilford, who is the best person to introduce the 2 to each other?

- ○ Shayla
- ○ Manuel
- ○ Rita

3) What is the length of the shortest path between Salvatore and Reva?

- ○ 5
- ○ 6
- ○ 7

14.3 Graph representations: Adjacency lists

Adjacency lists

Various approaches exist for representing a graph data structure. A common approach is an adjacency list. Recall that two vertices are **adjacent** if connected by an edge. In an **adjacency list**

graph representation, each vertex has a list of adjacent vertices, each list item representing an edge.

PARTICIPATION ACTIVITY	14.3.1: Adjacency list graph representation.

Animation captions:

1. Each vertex has a list of adjacent vertices for edges. The edge connecting A and B appears in A's list and also in B's list.
2. Each edge appears in the lists of the edge's two vertices.

Advantages of adjacency lists

A key advantage of an adjacency list graph representation is a size of O(V + E), because each vertex appears once, and each edge appears twice. V refers to the number of vertices, E the number of edges.

However, a disadvantage is that determining whether two vertices are adjacent is O(V), because one vertex's adjacency list must be traversed looking for the other vertex, and that list could have V items. However, in most applications, a vertex is only adjacent to a small fraction of the other vertices, yielding a sparse graph. A **sparse graph** has far fewer edges than the maximum possible. Many graphs are sparse, like those representing a computer network, flights between cities, or friendships among people (every person isn't friends with every other person). Thus, the adjacency list graph representation is very common.

PARTICIPATION ACTIVITY	14.3.2: Adjacency lists.
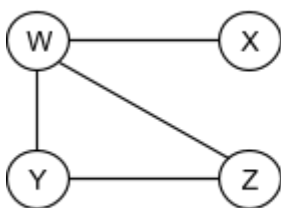
Vertices	Adjacent vertices (edges)		
W	X	**(a)**	Z
X	W		
Y	**(b)**		
Z	**(c)**		

1) (a)

○ Z

○ Y

2) (b)

○ W, X, Z

○ W, X

3) (c) ○ W, Z

○ W

○ W, X

○ W, Y

PARTICIPATION ACTIVITY	14.3.3: Adjacency lists: Bus routes.

The following adjacency list represents bus routes. Ex: A commuter can take the bus from Belmont to Sonoma.

Vertices	Adjacent vertices (edges)			
Belmont	Marin City	Sonoma		
Hillsborough	Livermore	Marin City	Sonoma	
Livermore	Hillsborough	Sonoma		
Marin City	Belmont	Hillsborough	Sonoma	
Sonoma	Belmont	Hillsborough	Livermore	Marin City

1) A direct bus route exists from Belmont to Sonoma or Belmont to _____.

[ ]

Check   **Show answer**

2) How many buses are needed to go from Livermore to Hillsborough?

[ ]

Check   **Show answer**

3) What is the minimum number of buses needed to go from Belmont to Hillsborough?

Check   **Show answer**

4) Is the following a path from
Livermore to Marin City? Type:
Yes or No
Livermore, Hillsborough,
Sonoma, Marin City

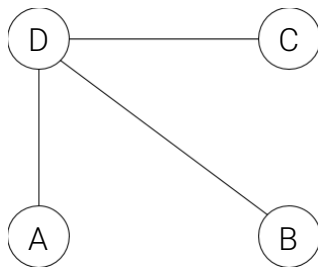**Check**   **Show answer**

**CHALLENGE
ACTIVITY**   14.3.1: Adjacency lists.

422352.2723990.qx3zqy7

Start

Select the missing vertices to complete the adjacency list representation of the given grap

Vertices	Adjacent vertices (edges)
A	(1)
B	(2)
C	(3)
D	A B (4)

(1): [A ⌄]
(2): [A ⌄]
(3): [A ⌄]
(4): [A ⌄]

1	2

Check   Next

14.4 Graph representations: Adjacency matrices

Adjacency matrices

Various approaches exist for representing a graph data structure. One approach is an adjacency matrix. Recall that two vertices are **adjacent** if connected by an edge. In an **adjacency matrix** graph representation, each vertex is assigned to a matrix row and column, and a matrix element is 1 if the corresponding two vertices have an edge or is 0 otherwise.

PARTICIPATION ACTIVITY	14.4.1: Adjacency matrix representation.

Animation captions:

1. Each vertex is assigned to a row and column.
2. An edge connecting A and B is a 1 in A's row and B's column.
3. Similarly, that same edge is a 1 in B's row and A's column. (The matrix will thus be symmetric.)
4. Each edge similarly has two 1's in the matrix. Other matrix elements are 0's (not shown).

Analysis of adjacency matrices

Assuming the common implementation as a two-dimensional array whose elements are accessible in O(1), then an adjacency matrix's key benefit is O(1) determination of whether two vertices are adjacent: The corresponding element is just checked for 0 or 1.

A key drawback is $O(V^2)$ size. Ex: A graph with 1000 vertices would require a 1000 x 1000 matrix, meaning 1,000,000 elements. An adjacency matrix's large size is inefficient for a sparse graph, in which most elements would be 0's.

An adjacency matrix only represents edges among vertices; if each vertex has data, like a person's name and address, then a separate list of vertices is needed.
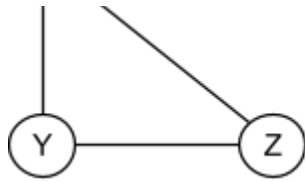
PARTICIPATION ACTIVITY	14.4.2: Adjacency matrix.
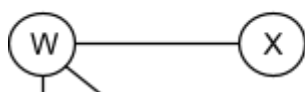
	W	X	Y	Z
W	0	(a)	(b)	1

X	1	0	0	**(c)**
Y	**(d)**	0	0	**(e)**
Z	1	0	**(f)**	0

1) (a)

- ○ 0
- ○ 1

2) (b)

- ○ 0
- ○ 1

3) (c)

- ○ 0
- ○ 1

4) (d)

- ○ 0
- ○ 1

5) (e)

- ○ 0
- ○ 1

6) (f)

- ○ 0
- ○ 1

PARTICIPATION ACTIVITY	14.4.3: Adjacency matrix: Power grid map.

The following adjacency matrix represents the map of a city's electrical power grid. Ex: Sector A's power grid is connected to Sector B's power grid.

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	1	1
C	1	1	0	1	1

D	1	1	1	0	0
E	0	1	1	0	0

1) How many edges does D have?

[   ]

Check   **Show answer**

2) How many edges does the graph contain?

[   ]

Check   **Show answer**

3) Assume Sector D has a power failure. Can power from Sector A be diverted directly to Sector D? Type: Yes or No

[   ]

Check   **Show answer**

4) Assume Sector E has a power failure. Can power from Sector A be diverted directly to Sector E? Type: Yes or No

[   ]

Check   **Show answer**

5) Is the following a path from Sector A to E? Type: Yes or No
AD, DB, BE

[   ]

Check   **Show answer**

CHALLENGE ACTIVITY | 14.4.1: Adjacency matrices.

422352.2723990.qx3zqy7

Start

Select the missing elements to complete the adjacency matrix representation of the given
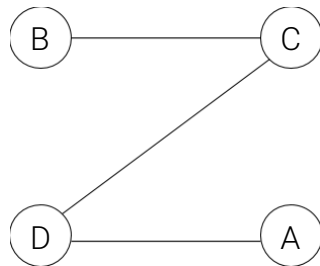

```
     A    B    C    D
A    0    0    0   (a)
B    0    0   (b)   0
C   (c)   1    0    1
D   (d)   0    1    0
```

(a): 0
(b): 0
(c): 0
(d): 0

1	2

Check   **Next**

14.5 Graphs: Breadth-first search

Graph traversal and breadth-first search

An algorithm commonly must visit every vertex in a graph in some order, known as a **graph traversal**. A **breadth-first search** (BFS) is a traversal that visits a starting vertex, then all vertices of distance 1 from that vertex, then of distance 2, and so on, without revisiting a vertex.

PARTICIPATION ACTIVITY | 14.5.1: Breadth-first search.

Animation captions:

1. Breadth-first search starting at A visits vertices based on distance from A. B and D are distance 1.
2. E and F are distance 2 from A. Note: A path of length 3 also exists to F, but distance metric uses shortest path.
3. C is distance 3 from A.
4. Breadth-first search from A visits A, then vertices of distance 1, then 2, then 3. Note: Visiting order of same-distance vertices doesn't matter.
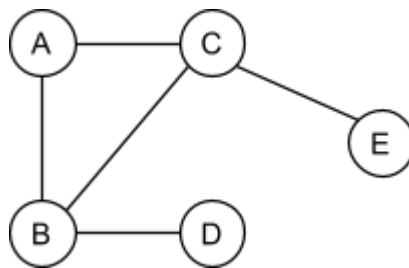
PARTICIPATION ACTIVITY	14.5.2: Breadth-first search traversal.

Perform a breadth-first search of the graph below. Assume the starting vertex is E.

1) Which vertex is visited first?

[   ]

Check   **Show answer**

2) Which vertex is visited second?

[   ]

Check   **Show answer**

3) Which vertex is visited third?

[   ]

Check   **Show answer**

4) What is C's distance?

[   ]

Check   **Show answer**

5) What is D's distance?

[   ]

Check   Show answer

6) The BFS traversal of a graph is
   unique. Type: Yes or No

[   ]

**Check**   Show answer

Example: Social networking connection recommender

Example 14.5.1: Social networking connection recommender using
breadth-first search.

Social networking sites like Facebook and LinkedIn use graphs to represent connections
among people. For a particular user, a site may wish to recommend new connections. One
approach does a breadth-first search starting from the user, recommending new
connections starting at distance 2 (distance 1 people are already connected with the
user).
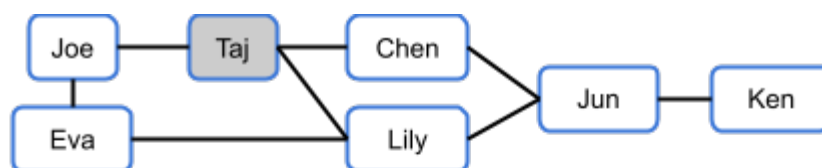
Breadth-first traversal:	Taj	Joe	Chen	Lily	Eva	Jun	Ken
	0	1	1	1	2	2	3
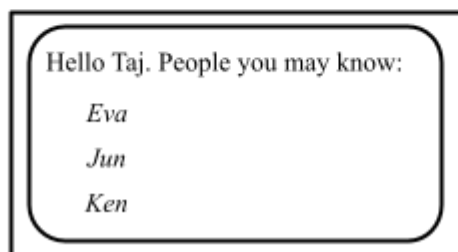
Hello Taj. People you may know:

Eva

Jun

Ken

PARTICIPATION ACTIVITY | 14.5.3: BFS: Connection recommendation.

Refer to the connection recommendation example above.

1) A distance greater than 0 indicates people are not connected with the user.

○ True
○ False

2) People with a distance of 2 are recommended before people with a distance of 3.

○ True
○ False

3) If Chen is the user, the system also recommends Eva, Jun, and Ken.

○ True
○ False

Example: Find closest item in a peer-to-peer network

Example 14.5.2: Application of BFS: Find closest item in a peer-to-peer network.

In a **peer-to-peer network**, computers are connected via a network and may seek and download file copies (such as songs or movies) via intermediary computers or routers. For example, one computer may seek the movie "The Wizard of Oz", which may exist on 10 computers in a network consisting of 100,000 computers. Finding the closest computer (having the fewest intermediaries) yields a faster download. A BFS traversal of the network graph can find the closest computer with that movie. The BFS traversal can be set to immediately return if the item sought is found during a vertex visit. Below, visiting vertex G finds the movie; BFS terminates, and a download process can begin, involving a path length of 2 (so only 1 intermediary). Vertex H also has the movie, but is further from B so wasn't visited yet during BFS. (Note: Distances of vertices visited during the BFS from B are shown below for convenience).
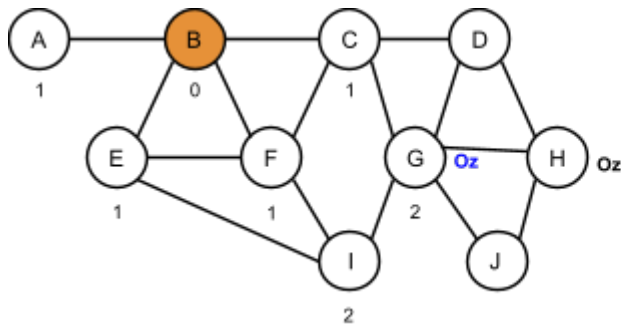
PARTICIPATION ACTIVITY	14.5.4: BFS application: Peer-to-peer search.

Consider the above peer-to-peer example.

1) In some BFS traversals starting from vertex B, vertex H may be visited before vertex G.

   ○ True

   ○ False

2) If vertex J sought the movie Oz, the download might occur from either G or H.

   ○ True

   ○ False

3) If vertex E sought the movie Oz, the download might occur from either G or H.

   ○ True

   ○ False

Breadth-first search algorithm

An algorithm for breadth-first search enqueues the starting vertex in a queue. While the queue is not empty, the algorithm dequeues a vertex from the queue, visits the dequeued vertex, enqueues that vertex's adjacent vertices (if not already discovered), and repeats.

PARTICIPATION ACTIVITY	14.5.5: BFS algorithm.

Animation content:

undefined

Animation captions:

1. BFS enqueues the start vertex (in this case A) in frontierQueue, and adds A to discoveredSet.
2. Vertex A is dequeued from frontierQueue and visited.
3. Undiscovered vertices adjacent to A are enqueued in frontierQueue and added to discoveredSet.
4. Vertex A's visit is complete. The process continues on to next vertices in the frontierQueue, D and B.
5. E is visited. Vertices B and F are adjacent to E, but are already in discoveredSet. So B and F are not again added to frontierQueue or discoveredSet.
6. The process continues until frontierQueue is empty. discoveredSet shows the visit order. Note that each vertex's distance from start is shown on the graph.

When the BFS algorithm first encounters a vertex, that vertex is said to have been ***discovered***. In the BFS algorithm, the vertices in the queue are called the ***frontier***, being vertices thus far discovered but not yet visited. Because each vertex is visited at most once, an already-discovered vertex is not enqueued again.

A "visit" may mean to print the vertex, append the vertex to a list, compare vertex data to a value and return the vertex if found, etc.
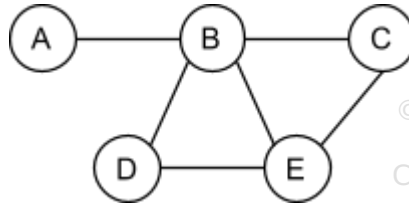
PARTICIPATION ACTIVITY | 14.5.6: BFS algorithm.

BFS is run on the following graph. Assume C is the starting vertex.

1) Which vertices are in the frontierQueue before the first iteration of the while loop?

   ○ A

   ○ C

   ○ A, B, C, D, E

2) Which vertices are in discoveredSet after the first iteration of the while loop?

   ○ C, B, E

   ○ B, E

   ○ C

3) In the second iteration, currentV = B. Which vertices are in discoveredSet after the second iteration of the while loop?

   ○ C, B, E, A

   ○ B, E, A, D

   ○ C, B, E, A, D

4) In the second iteration, currentV = B. Which vertices are in frontierQueue after the second iteration of the while loop?

   ○ C, B, E, A, D

   ○ E, A, D

   ○ frontierQueue is empty

5) BFS terminates after the second
   iteration, because all vertices are in
   the discoveredSet.

   ○ True

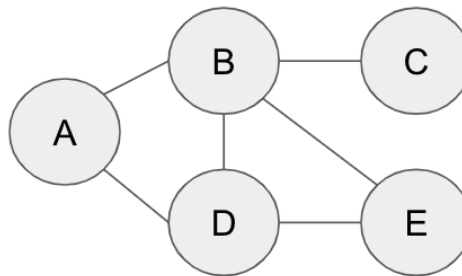   ○ False

CHALLENGE ACTIVITY	14.5.1: Breadth-first search.

422352.2723990.qx3zqy7

Start

Given the graph below and the starting vertex D:

What is C's distance? [Ex: 5]

What is B's distance? [   ]

What is D's distance? [   ]

What is E's distance? [   ]

What is A's distance? [   ]

| **1** | 2 | 3 | 4 |

Check   Next

14.6 Graphs: Depth-first search

Graph traversal and depth-first search

An algorithm commonly must visit every vertex in a graph in some order, known as a **graph traversal**. A **depth-first search** (DFS) is a traversal that visits a starting vertex, then visits every vertex along each path starting from that vertex to the path's end before backtracking.

PARTICIPATION ACTIVITY	14.6.1: Depth-first search.

Animation captions:

1. Depth-first search starting at A descends along a path to the path's end before backtracking.
2. Reach path's end: Backtrack to F, visit F's other adjacent vertex (E). B already visited, backtrack again.
3. Backtracked all the way to A. Visit A's other adjacent vertex (D). No other adjacent vertices: Done.

PARTICIPATION ACTIVITY	14.6.2: Depth-first search traversal.

Perform a depth-first search of the graph below. Assume the starting vertex is E.
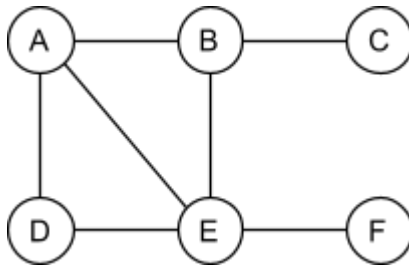
1) Which vertex is visited first?

[   ]

Check   **Show answer**

2) Assume DFS traverses the following vertices: E, A, B. Which vertex is visited next?

[   ]

Check   **Show answer**

3) Assume DFS traverses the
following vertices: E, A, B, C.
Which vertex is visited next?

[   ]

Check   **Show answer**

4) Is the following a valid DFS
traversal? Type: Yes or No
E, D, F, A, B, C

[   ]

Check   **Show answer**

5) Is the following a valid DFS
traversal? Type: Yes or No
E, D, A, B, C, F

[   ]

Check   **Show answer**

6) The DFS traversal of a graph is
unique. Type: Yes or No

[   ]

Check   **Show answer**

Depth-first search algorithm

An algorithm for depth-first search pushes the starting vertex to a stack. While the stack is not
empty, the algorithm pops the vertex from the top of the stack. If the vertex has not already been
visited, the algorithm visits the vertex and pushes the adjacent vertices to the stack.

| PARTICIPATION ACTIVITY	14.6.3: Depth-first search traversal with starting vertex A.

Animation content:

undefined

Animation captions:

1. A depth-first search at vertex A first pushes vertex A onto the stack. The first loop iteration then pops vertex A off the stack and assigns currentV with that vertex.
2. Vertex A is visited and added to the visited set. Each vertex adjacent to A is pushed onto the stack.
3. Vertex B is popped off the stack and processed as the next currentV.
4. Vertices F and E are processed similarly.
5. Vertices F and B are in the visited set and are skipped after being popped off the stack.
6. Vertex C is popped off the stack and visited. All remaining vertices except D are in the visited set.
7. Vertex D is the last vertex visited.

PARTICIPATION ACTIVITY	14.6.4: DFS algorithm.

DFS is run on the following graph. Assume C is the starting vertex.
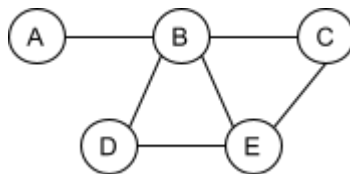
1) Which vertices are in the stack before the first iteration of the while loop?

   ○ A

   ○ C

   ○ A, B, C, D, E

2) Which vertices are in the stack after the first iteration of the while loop?

   ○ B, E, C

   ○ B, E

   ○ B

3) Which vertices are in visitedSet after the first iteration of the while loop?

    ○ B, E, C

    ○ C

    ○ B

4) In the second iteration, currentV = B. Which vertices are in the stack after the second iteration of the while loop?

    ○ C

    ○ A, C, D, E

    ○ A, C, D, E, E

5) Which vertices are in visitedSet after the second iteration of the while loop?

    ○ C, B

    ○ C

    ○ B

6) DFS terminates once all vertices are added to visitedSet.

    ○ True

    ○ False

Recursive DFS algorithm

A recursive DFS can be implemented using the program stack instead of an explicit stack. The recursive DFS algorithm is first called with the starting vertex. If the vertex has not already been visited, the recursive algorithm visits the vertex and performs a recursive DFS call for each adjacent vertex.

Figure 14.6.1: Recursive depth-first search.

```
RecursiveDFS(currentV) {
    if ( currentV is not in visitedSet ) {
        Add currentV to visitedSet
        "Visit" currentV
        for each vertex adjV adjacent to
    currentV {
            RecursiveDFS(adjV)
        }
    }
}
```
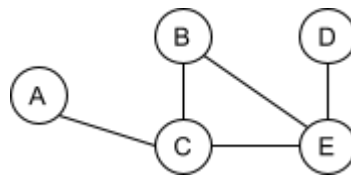

**PARTICIPATION
ACTIVITY**   14.6.5: Recursive DFS algorithm.

The recursive DFS algorithm is run on the following graph. Assume D is the starting
vertex.

1) The recursive DFS algorithm uses a
   queue to determine which vertices to
   visit.

   ○ True

   ○ False

2) DFS begins with the function call
   RecursiveDFS(D).

   ○ True

   ○ False

3) If B is not yet visited, RecursiveDFS(B)
   will make subsequent calls to
   RecursiveDFS(C) and
   RecursiveDFS(E).

   ○ True

   ○ False

CHALLENGE ACTIVITY | 14.6.1: Graphs: Depth-first search.

422352.2723990.qx3zqy7
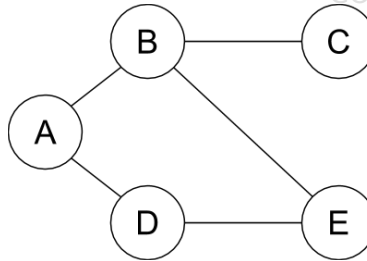
Start

Enter a valid depth-first search traversal when A is the starting vertex.

Ex: A, B, C, D, E (commas between values)

1	2	3	4	5

Check   Next

14.7 Directed graphs

Directed graphs

A ***directed graph***, or ***digraph***, consists of vertices connected by directed edges. A ***directed edge*** is a connection between a starting vertex and a terminating vertex. In a directed graph, a vertex Y is ***adjacent*** to a vertex X, if there is an edge from X to Y.

Many graphs are directed, like those representing links between web pages, maps for navigation, or college course prerequisites.

Figure 14.7.1: Directed graph examples: Process flow diagram, airline routes, and college course prerequisites.

Source: Fluid catalyic cracker (Mbeychok / Public Domain via Wikimedia Commons), Florida airline routes 1974 (Geez-oz (Own work) / CC-BY-SA-3.0 via Wikimedia Commons), college course prerequisites (zyBooks)

PARTICIPATION ACTIVITY	14.7.1: A directed graph represents connections among items, like links between web pages, or airline routes.

Animation captions:

1. Items in the world may have directed connections, like links on a website.
2. A directed graph's vertices represent items.
3. Vertices are connected by directed edges.
4. A directed edge represents a connection from a starting vertex to a terminating vertex; the terminating vertex is adjacent to the starting vertex. B is adjacent to A, but A is not adjacent to B.
5. A directed graph can represent many things, like airline connections between cities. A flight is available from Los Angeles to Tucson, but not Tucson to Los Angeles.

PARTICIPATION ACTIVITY	14.7.2: Directed graph basics.

Refer to the above graphs.

1) E is a _____ in the directed graph.

   ○ vertex

   ○ directed edge

2) A directed edge connects vertices A

and D. D is the _____ vertex.

  ◯ starting

  ◯ terminating

3) The airline routes graph is a digraph.

  ◯ True

  ◯ False

4) Tucson is adjacent to _____.

  ◯ San Francisco

  ◯ Los Angeles

  ◯ Dallas

Paths and cycles

In a directed graph:

- A **path** is a sequence of directed edges leading from a source (starting) vertex to a destination (ending) vertex.
- A **cycle** is a path that starts and ends at the same vertex. A directed graph is **cyclic** if the graph contains a cycle, and **acyclic** if the graph does not contain a cycle.

PARTICIPATION ACTIVITY	14.7.3: Directed graph: Paths and cycles.

Animation captions:

1. A path is a sequence of directed edges leading from a source (starting) vertex to a destination (ending) vertex.
2. A cycle is a path that starts and ends at the same vertex. A graph can have more than one cycle.
3. A cyclic graph contains a cycle. An acyclic graph contains no cycles.

Example 14.7.1: Cycles in directed graphs: Kidney transplants.

A patient needing a kidney transplant may have a family member willing to donate a kidney but is incompatible. That family member is willing to donate a kidney to someone else, as long as their family member also receives a kidney donation. Suppose Gregory needs a kidney. Gregory's wife, Eleanor, is willing to donate a kidney but is not compatible with Gregory. However, Eleanor is compatible with another patient Joanna, and Joanna's husband Darrell is compatible with Gregory. So, Eleanor donates a kidney to Joanna, and Darrell donates a kidney to Gregory, which is an example of a 2-way kidney transplant. In 2015, a 9-way kidney transplant involving 18 patients was performed within 36 hours (Source: SF Gate). Multiple-patient kidney transplants can be represented as cycles within a directed graph.
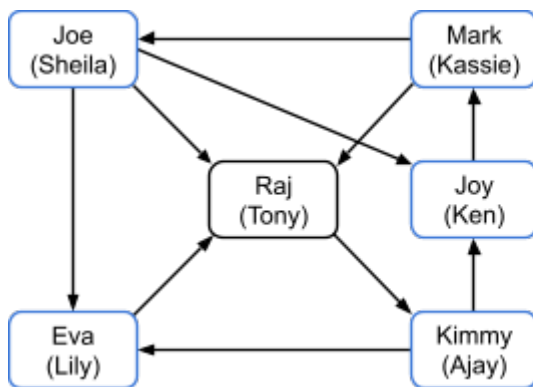
3-way transplant: Eva (Lily), Raj (Tony), Kimmy (Ajay)

4-way transplant: Raj (Tony), Kimmy (Ajay), Joy (Ken), Mark (Kassie)

In this graph, vertices represent patients, and edges represent compatibility between a patient's family member (shown in parentheses) and another patient. An N-way kidney transplant is represented as a cycle with N edges. Due to the complexity of coordinating multiple simultaneous surgeries, hospitals and doctors typically try to find the shortest possible cycle.

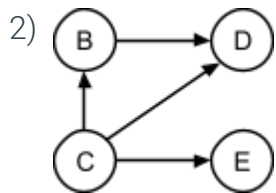**PARTICIPATION ACTIVITY**   14.7.4: Directed graphs: Cyclic and acyclic.

Determine if each of the following graphs is cyclic or acyclic.
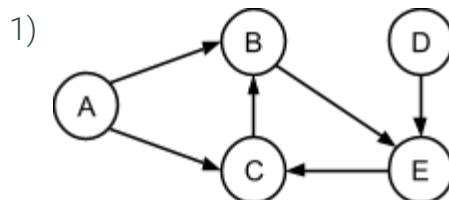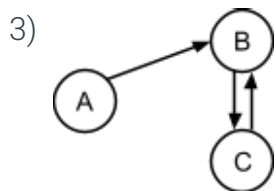
1)

- ○ Cyclic
- ○ Acyclic

2)

○ Cyclic

○ Acyclic

3)

○ Cyclic

○ Acyclic

4)
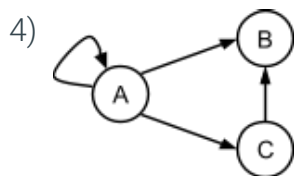
○ Cyclic

○ Acyclic

CHALLENGE ACTIVITY   14.7.1: Directed graphs.

422352.2723990.qx3zqy7

Start

Consider the following directed graph.
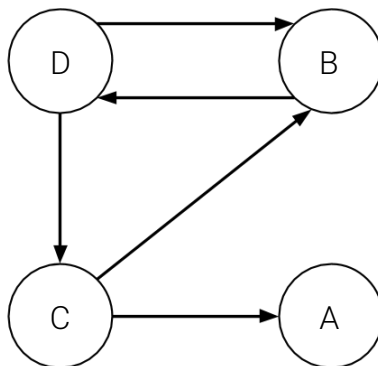
Identify the vertices (if any) that are adjacent to:

A   [Ex: A, B]   Enter values separated by commas.
                     If none, enter: none

B   [   ]

1	2	3

[Check]   [Next]

14.8 Weighted graphs

ℹ This section has been set as optional by your instructor.

Weighted graphs

A **weighted graph** associates a weight with each edge. A graph edge's **weight**, or **cost**, represents some numerical value between vertex items, such as flight cost between airports, connection speed between computers, or travel time between cities. A weighted graph may be directed or undirected.

PARTICIPATION ACTIVITY   14.8.1: Weighted graphs associate weight with each edge.

Animation captions:

1. A weighted graph associates a numerical weight, or cost, with each edge. Ex: Edge weights may indicate connection speed (Mbps) between computers.
2. Weighted graphs can be directed. Ex: Edge weights may indicate travel time (hours) between cities; travel times may vary by direction.
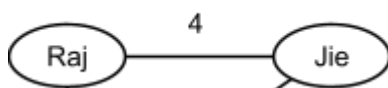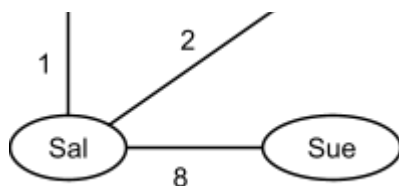

PARTICIPATION ACTIVITY   14.8.2: Weighted graphs.

1) This graph's weights indicate the number of times coworkers have collaborated on projects. How many times have Raj and Jie teamed up?
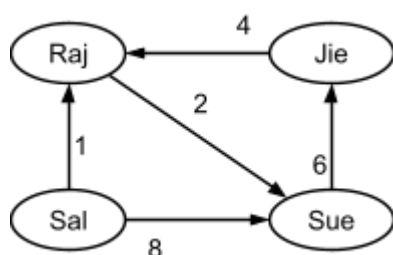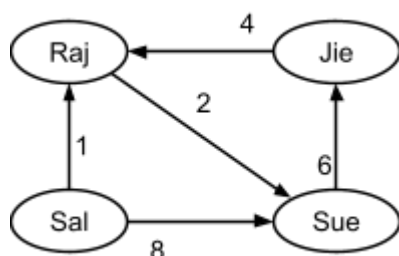
- ○ 1
- ○ 4

2) This graph indicates the number of times a worker has nominated a coworker for an award. How many times has Sal nominated Sue?

- ○ 0
- ○ 8

3) This graph indicates the number of times a worker has nominated a coworker for an award. How many times has Raj nominated Jie?
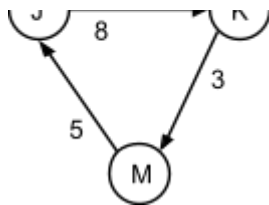

- ○ 4
- ○ 1
- ○ 0

4) The weight of the edge from K to J is _____.

○ 7

○ 8

Path length in weighted graphs

In a weighted graph, the ***path length*** is the sum of the edge weights in the path.

| PARTICIPATION ACTIVITY	14.8.3: Path length is the sum of edge weights.	

Animation captions:

1. A path is a sequence of edges from a source vertex to a destination vertex.
2. The path length is the sum of the edge weights in the path.
3. The shortest path is the path yielding the lowest sum of edge weights. Ex: The shortest path from Paris to Marseille is 6.

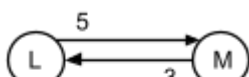| PARTICIPATION ACTIVITY	14.8.4: Path length and shortest path.	

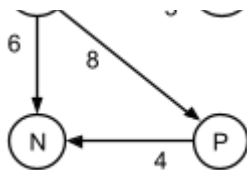1) Given a path A, C, D, the path length is _____.
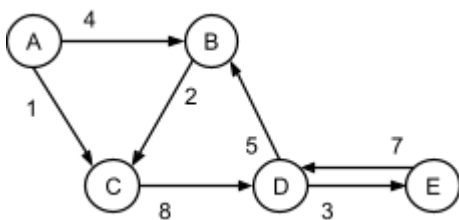
[   ]

Check   **Show answer**

2) The shortest path from M to N has a length of _____.

[Check]   Show answer

3) The shortest path from S to U
   has a length of _____.
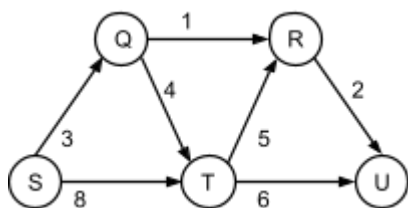
[Check]   Show answer

4) Given a path H, G, J, F, the path
   length is _____.
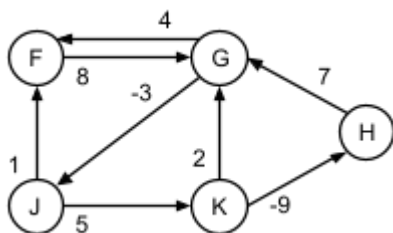
[Check]   Show answer

Negative edge weight cycles

The **cycle length** is the sum of the edge weights in a cycle. A **negative edge weight cycle** has a
cycle length less than 0. A shortest path does not exist in a graph with a negative edge weight
cycle, because each loop around the negative edge weight cycle further decreases the cycle length,
so no minimum exists.

Figure 14.8.1: A shortest path from A to D does not exist, because the cycle B, C can be repeatedly taken to further reduce the cycle length.
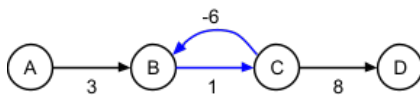
Path 1: A, B, C, B, C, D   Length = 3 + 1 + -6 + 1 + 8 = 7
Path 2: A, B, C, B, C, B, C, D   Length = 3 + 1 + -6 + 1 + -6 + 1 + 8 = 2
Path 3: A, B, C, B, C, B, C, B, C, D   Length = 3 + 1 + -6 + 1 + -6 + 1 + -6 + 1 + 8 = -3
and so on...

PARTICIPATION ACTIVITY	14.8.5: Negative edge weight cycles.

1) The cycle length for F, G, H, F is _____.

[   ]
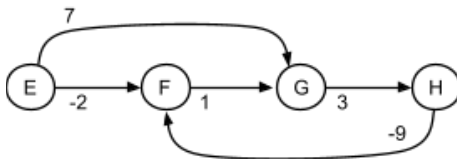
Check   **Show answer**

2) Is A, C, D, A a negative edge weight cycle? Type Yes or No.

[   ]
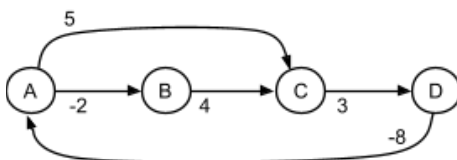
Check   **Show answer**

3) The graph contains _____ negative edge weight cycles.
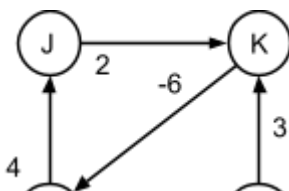
[   ]

Check   **Show answer**

CHALLENGE ACTIVITY	14.8.1: Weighted graphs.

422352.2723990.qx3zqy7

Start
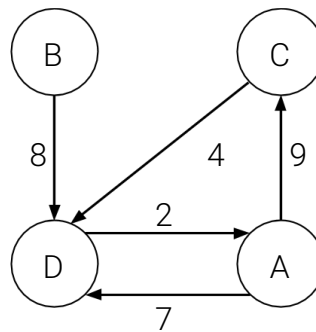
Find the weight of each of the following edges.

Weight of edge from A to C: [Ex: 1 or undefined]   Enter "undefined" if the edge is not defined.

Weight of edge from A to D: [   ]

Weight of edge from D to C: [   ]

1	2	3	4	5

Check   **Next**

14.9 Algorithm: Dijkstra's shortest path

ⓘ   This section has been set as optional by your instructor.

Dijkstra's shortest path algorithm

Finding the shortest path between vertices in a graph has many applications. Ex: Finding the shortest driving route between two intersections can be solved by finding the shortest path in a directed graph where vertices are intersections and edge weights are distances. If edge weights instead are expected travel times (possibly based on real-time traffic data), finding the shortest path will provide the fastest driving route.

Dijkstra's shortest path algorithm, created by Edsger Dijkstra, determines the shortest path from a start vertex to each vertex in a graph. For each vertex, Dijkstra's algorithm determines the vertex's distance and predecessor pointer. A vertex's **distance** is the shortest path distance from the start vertex. A vertex's **predecessor pointer** points to the previous vertex along the shortest path from the start vertex.

Dijkstra's algorithm initializes all vertices' distances to infinity (∞), initializes all vertices' predecessors to null, and enqueues all vertices into a queue of unvisited vertices. The algorithm then assigns the start vertex's distance with 0. While the queue is not empty, the algorithm dequeues the vertex with the shortest distance. For each adjacent vertex, the algorithm computes the distance of the path from the start vertex to the current vertex and continuing on to the adjacent vertex. If that path's distance is shorter than the adjacent vertex's current distance, a shorter path has been found. The adjacent vertex's current distance is updated to the distance of the newly found shorter path's distance, and vertex's predecessor pointer is pointed to the current vertex.

PARTICIPATION ACTIVITY	14.9.1: Dijkstra's algorithm finds the shortest path from a start vertex to each vertex in a graph.

Animation content:

undefined

Animation captions:

1. Each vertex is initialized with distance set to Infinity and the predecessor pointer set to null. Each vertex is enqueued into unvisitedQueue.
2. The start vertex's distance is 0. The algorithm visits the start vertex first.
3. For each adjacent vertex, if a shorter path from the start vertex to the adjacent vertex is found, the vertex's distance and predecessor pointer are updated.
4. B has the shortest path distance, and is dequeued from the queue. The path through B to C is not shorter, so no update occurs. The path through B to D is shorter, so D's distance and predecessor are updated.
5. D is then dequeued. The path through D to C is shorter, so C's distance and predecessor

pointer are updated.

6. C is then dequeued. The path through C to D is not shorter, so no update occurs.

7. The algorithm terminates when all vertices are visited. Each vertex's distance is the shortest path distance from the start vertex. The vertex's predecessor pointer points to the previous vertex in the shortest path.

PARTICIPATION ACTIVITY	14.9.2: Dijkstra's shortest path traversal.

Perform Dijkstra's shortest path algorithm on the graph below with starting vertex B.

1) Which vertex is visited first?

[   ]
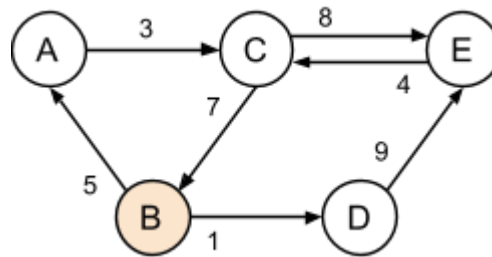
Check   **Show answer**

2) A's distance after the algorithm's first while loop iteration is ＿＿＿. Type inf for infinity.

[   ]

Check   **Show answer**

3) D's distance after the first while loop iteration is ＿＿＿. Type inf for infinity.

[   ]

Check   **Show answer**

4) C's distance after the first iteration of the while loop is

＿＿＿.

Type inf for infinity.

[   ]

Check   **Show answer**

5) Which vertex is visited second?

[   ]

Check   **Show answer**

6) E's distance after the second
   while loop iteration is _____.

[   ]

Check   **Show answer**

Finding shortest path from start vertex to destination vertex

After running Dijkstra's algorithm, the shortest path from the start vertex to a destination vertex can be determined using the vertices' predecessor pointers. If the destination vertex's predecessor pointer is not 0, the shortest path is traversed in reverse by following the predecessor pointers until the start vertex is reached. If the destination vertex's predecessor pointer is null, then a path from the start vertex to the destination vertex does not exist.

PARTICIPATION ACTIVITY	14.9.3: Determining the shortest path from Dijkstra's algorithm.

Animation content:

undefined

Animation captions:

1. The vertex's predecessor pointer points to the previous vertex in the shortest path.
2. Starting with the destination vertex, the predecessor pointer is followed until the start vertex is reached.
3. The vertex's distance is the shortest path distance from the start vertex.

PARTICIPATION ACTIVITY	14.9.4: Shortest path based on vertex predecessor.

Type path as: A, B, C
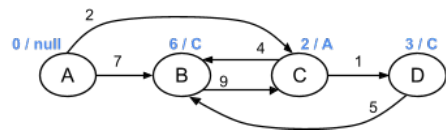If path does not exist, type: None

1) After executing
   DijkstraShortestPath(A), what is
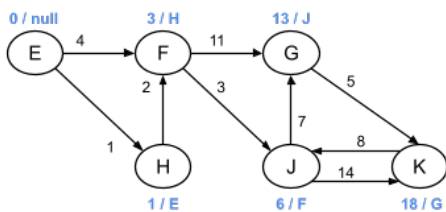   the shortest path from A to B?

Check   **Show answer**

2) After executing
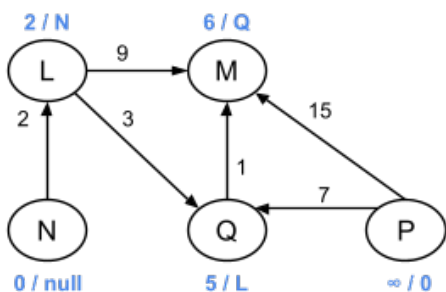   DijkstraShortestPath(E), what is
   the shortest path from E to G?

Check   **Show answer**

3) After executing
   DijkstraShortestPath(N), what is
   the shortest path from N to P?

Check   **Show answer**

Algorithm efficiency

If the unvisited vertex queue is implemented using a list, the runtime for Dijkstra's shortest path algorithm is $O(V^2)$. The outer loop executes V times to visit all vertices. In each outer loop execution, dequeuing the vertex from the queue requires searching all vertices in the list, which has a runtime of O(V). For each vertex, the algorithm follows the subset of edges to adjacent vertices; following a total of E edges across all loop executions. Given $E < V^2$, the runtime is $O(V*V + E) = O(V^2 + E) = O(V^2)$. Implementing the unvisited vertex queue using a standard binary heap reduces the runtime to $O((E + V) \log V)$, and using a Fibonacci heap data structure (not discussed in this material) reduces the runtime to $O(E + V \log V)$.

Negative edge weights

Dijkstra's shortest path algorithm can be used for unweighted graphs (using a uniform edge weight of 1) and weighted graphs with non-negative edges weights. For a directed graph with negative edge weights, Dijkstra's algorithm may not find the shortest path for some vertices, so the algorithm should not be used if a negative edge weight exists.

PARTICIPATION ACTIVITY	14.9.5: Dijkstra's algorithm may not find the shortest path for a graph with negative edge weights.	

Animation content:

undefined

Animation captions:

1. A is the start vertex. Adjacent vertices resulting in a shorter path are updated.
2. Path through B to D results in a shorter path. Vertex D is updated.
3. D has no adjacent vertices.
4. A path through C to B results in a shorter path. Vertex B is updated.
5. Vertex B has already been visited, and will not be visited again. So, D's distance and predecessor are not for the shortest path from A to D.

PARTICIPATION ACTIVITY	14.9.6: Dijkstra's shortest path algorithm: Supported graph types.

Indicate if Dijkstra's algorithm will find the shortest path for the following graphs.

1)

- ○ Yes
- ○ No

2)

- ○ Yes
- ○ No

3)
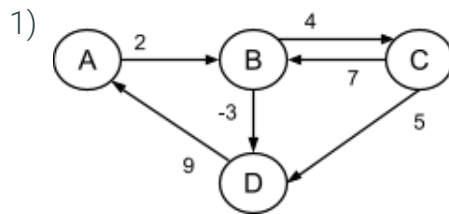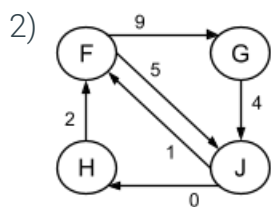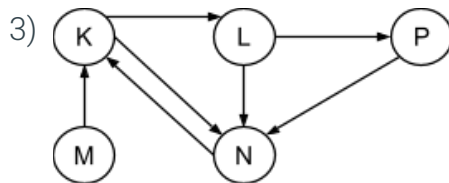
- ○ Yes
- ○ No

CHALLENGE ACTIVITY	14.9.1: Dijkstra's shortest path.

422352.2723990.qx3zqy7

14.10 Algorithm: Bellman-Ford's shortest path

> ℹ   This section has been set as optional by your instructor.

Bellman-Ford shortest path algorithm

The **Bellman-Ford shortest path algorithm**, created by Richard Bellman and Lester Ford, Jr., determines the shortest path from a start vertex to each vertex in a graph. For each vertex, the Bellman-Ford algorithm determines the vertex's distance and predecessor pointer. A vertex's **distance** is the shortest path distance from the start vertex. A vertex's **predecessor pointer** points to the previous vertex along the shortest path from the start vertex.

The Bellman-Ford algorithm initializes all vertices' current distances to infinity (∞) and predecessors to null, and assigns the start vertex with a distance of 0. The algorithm performs V-1 main iterations, visiting all vertices in the graph during each iteration. Each time a vertex is visited, the algorithm follows all edges to adjacent vertices. For each adjacent vertex, the algorithm computes the distance of the path from the start vertex to the current vertex and continuing on to the adjacent vertex. If that path's distance is shorter than the adjacent vertex's current distance, a shorter path has been found. The adjacent vertex's current distance is updated to the newly found shorter path's distance, and the vertex's predecessor pointer is pointed to the current vertex.

The Bellman-Ford algorithm does not require a specific order for visiting vertices during each main iteration. So after each iteration, a vertex's current distance and predecessor may not yet be the shortest path from the start vertex. The shortest path may propagate to only one vertex each iteration, requiring V-1 iterations to propagate from the start vertex to all other vertices.

PARTICIPATION ACTIVITY	14.10.1: The Bellman-Ford algorithm finds the shortest path from a source vertex to all other vertices in a graph.

Animation content:

undefined

Animation captions:

1. The Bellman-Ford algorithm initializes each vertex's distance to Infinity and each vertex's predecessor to null, and assigns the start vertex's distance with 0.
2. For each vertex in the graph, if a shorter path from the currentV to the adjacent vertex is found, the adjacent vertex's distance and predecessor pointer are updated.
3. The path through B to D results in a shorter path to D than is currently known. So vertex D is updated.
4. The path through C to B results in a shorter path to B than is currently known. So vertex B is updated.
5. D has no adjacent vertices. After each iteration, a vertex's distance and predecessor may not yet be the shortest path from the start vertex.
6. In each main iteration, the algorithm visits all vertices. This time, the path through B to D results in an even shorter path. So vertex D is updated again.
7. During the third main iteration, no shorter paths are found, so no vertices are updated. V-1 iterations may be required to propagate the shortest path from the start vertex to all other

8. When done, each vertex's distance is the shortest path distance from the start vertex, and the vertex's predecessor pointer points to the previous vertex in the shortest path.

PARTICIPATION ACTIVITY   14.10.2: Bellman-Ford shortest path traversal.

The start vertex is A. In each main iteration, vertices in the graph are visited in the following order: A, B, C, D.
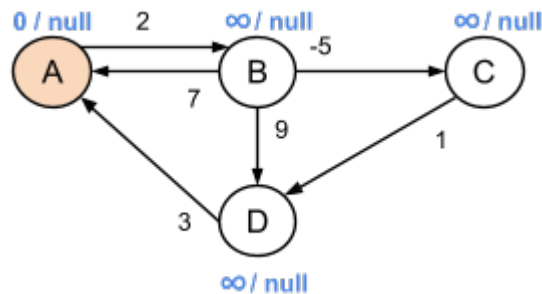
1) What are B's values after the first iteration?

  ○ 2 / A

  ○ ∞ / null

2) What are C's values after the first iteration?

  ○ -5 / B

  ○ -3 / B

  ○ ∞ / null

3) What are D's values after the first iteration?

  ○ 3 / A

  ○ -2 / C

  ○ ∞ / null

PARTICIPATION ACTIVITY   14.10.3: Bellman-Ford: Distance and predecessor values.

The start vertex is J. Vertices in the graph are processed in the following order: E, F, G, H, J.
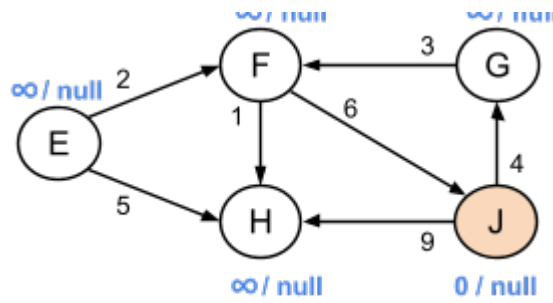
Firefox

1) How many iterations are executed?

○ 5

○ 4

○ 6

2) What are F's values after the first
   iteration?

○ ∞ / null

○ 7 / G

3) What are G's values after the first
   iteration?

○ 4 / J

○ ∞ / null

4) What are E's values after the final
   iteration?

○ 9 / F

○ ∞ / null

Algorithm Efficiency

The runtime for the Bellman-Ford shortest path algorithm is O(VE). The outer loop
(the main iterations) executes V-1 times. In each outer loop execution, the algorithm
visits each vertex and follows the subset of edges to adjacent vertices, following a
total of E edges across all loop executions.

Checking for negative edge weight cycles

The Bellman-Ford algorithm supports graphs with negative edge weights. However, if a negative edge weight cycle exists, a shortest path does not exist. After visiting all vertices V-1 times, the algorithm checks for negative edge weight cycles. If a negative edge weight cycle does not exist, the algorithm returns true (shortest path exists), otherwise returns false.

PARTICIPATION ACTIVITY	14.10.4: Bellman-Ford: Checking for negative edge weight cycles.

Animation content:

undefined

Animation captions:

1. After visiting all vertices V-1 times, the Bellman-Ford algorithm checks for negative edge weight cycles.
2. For each vertex in the graph, adjacent vertices are checked for a shorter path. No such path exists through A to B.
3. But, a shorter path is still found through B to C, so a negative edge weight cycle exists.
4. The algorithm returns false, indicating a shortest path does not exist.

Figure 14.10.1: Bellman-Ford shortest path algorithm.

```
BellmanFord(startV) {
    for each vertex currentV in graph {
        currentV→distance = Infinity
        currentV→predV = null
    }

    // startV has a distance of 0 from itself
    startV→distance = 0

    for i = 1 to number of vertices - 1 { // Main iterations
        for each vertex currentV in graph {
            for each vertex adjV adjacent to currentV {
                edgeWeight = weight of edge from currentV to adjV
                alternativePathDistance = currentV→distance +
edgeWeight

                // If shorter path from startV to adjV is found,
                // update adjV's distance and predecessor
                if (alternativePathDistance < adjV→distance) {
                    adjV→distance = alternativePathDistance
                    adjV→predV = currentV
                }
            }
        }
    }

    // Check for a negative edge weight cycle
    for each vertex currentV in graph {
        for each vertex adjV adjacent to currentV {
            edgeWeight = weight of edge from currentV to adjV
            alternativePathDistance = currentV→distance +
edgeWeight

            // If shorter path from startV to adjV is still found,
            // a negative edge weight cycle exists
            if (alternativePathDistance < adjV→distance) {
                return false
            }
        }
    }

    return true
}
```

PARTICIPATION ACTIVITY | 14.10.5: Bellman-Ford algorithm: Checking for negative edge weight cycles.

1) Given the following result from the

Bellman-Ford algorithm for a start vertex of A, a negative edge weight cycle is found when checking adjacent vertices of vertex _____.
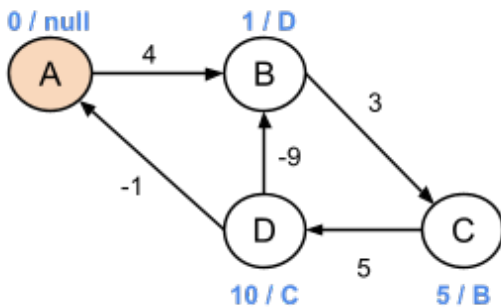
○ B

○ C

○ D

2) What does the Bellman-Ford algorithm return for the following graph with a start vertex of H?
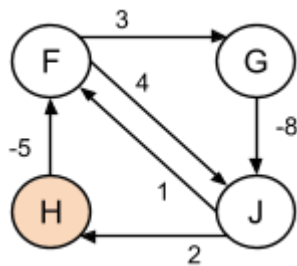
○ True

○ False

CHALLENGE ACTIVITY   14.10.1: Bellman-Ford's shortest path.

422352.2723990.qx3zqy7

14.11 Topological sort

Overview

A **topological sort** of a directed, acyclic graph produces a list of the graph's vertices such that for every edge from a vertex X to a vertex Y, X comes before Y in the list.

PARTICIPATION ACTIVITY	14.11.1: Topological sort.

Animation captions:

1. Analysis of each edge in the graph determines if an ordering of vertices is a valid topological sort.
2. If an edge from X to Y exists, X must appear before Y in a valid topological sort. C, D, A, F, B, E is not valid because this requirements is violated for three edges.
3. Ordering D, A, F, E, C, B has 1 edge violating the requirement, so the ordering is not a valid topological sort.
4. For ordering D, A, F, E, B, C, the requirement holds for all edges, so the ordering is a valid topological sort.
5. A graph can have more than 1 valid topological sort. Another valid ordering is D, A, F, B, E, C.

PARTICIPATION ACTIVITY	14.11.2: Topological sort.

1) In the example above, D, A, B, F, E, C is a valid topological sort.

   ○ True

   ○ False

2) Which of the following is NOT a requirement of the graph for topological sorting?

   ○ The graph must be acyclic.

   ○ The graph must be directed.

   ○ The graph must be weighted.

3) For a directed, acyclic graph, only one possible topological sort output exists.

   ○ True

   ○ False

PARTICIPATION ACTIVITY	14.11.3: Identifying valid topological sorts.

Indicate whether each vertex ordering is a valid topological sort of the graph below.
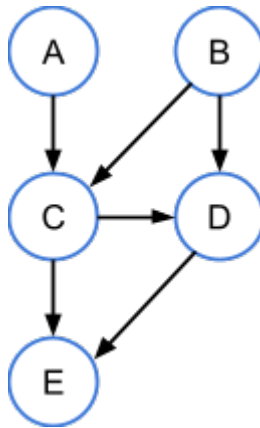
1) A, B, C, D, E

   ○ Valid

   ○ Invalid

2) E, D, C, B, A

   ○ Valid

   ○ Invalid

3) D, E, A, B, C

   ○ Valid

   ○ Invalid

4) B, A, C, D, E

   ○ Valid

   ○ Invalid

5) B, A, C, E, D

   ○ Valid

   ○ Invalid

Example: course prerequisites

Graphs can be used to indicate a sequence of steps, where an edge from X to Y indicates that X must be done before Y. A topological sort of such a graph provides one possible ordering for

performing the steps. Ex: Given a graph representing course prerequisites, a topological sort of the graph provides an ordering in which the courses can be taken.

PARTICIPATION ACTIVITY | 14.11.4: Topological sorting can be used to order course prerequisites.

Animation captions:

1. For a graph representing course prerequisites, the vertices represent courses, and the edges represent the prerequisites. CS 101 must be taken before CS 102, and CS 102 before CS 103.
2. CS 103 is "Robotics Programming" and has a physics course (Phys 101) as a prerequisite. Phys 101 also has a math prerequisite (Math 101).
3. The graph's valid topological sorts provide possible orders in which to take the courses.

PARTICIPATION ACTIVITY | 14.11.5: Course prerequisites.

1) The "Math 101" and "CS 101" vertices have no incoming edges, and therefore one of these two vertices must be the first vertex in any topological sort.

  ○ True

  ○ False

2) Every topological sort ends with the "CS 103" vertex because this vertex has no outgoing edges.

  ○ True

  ○ False

Topological sort algorithm

The topological sort algorithm uses three lists: a results list that will contain a topological sort of vertices, a no-incoming-edges list of vertices with no incoming edges, and a remaining-edges list. The result list starts as an empty list of vertices. The no-incoming-edges vertex list starts as a list of all vertices in the graph with no incoming edges. The remaining-edges list starts as a list of all edges in the graph.

The algorithm executes while the no-incoming-edges vertex list is not empty. For each iteration, a

vertex is removed from the no-incoming-edges list and added to the result list. Next, a temporary list is built by removing all edges in the remaining-edges list that are outgoing from the removed vertex. For each edge currentE in the temporary list, the number of edges in the remaining-edges list that are incoming to currentE's terminating vertex are counted. If the incoming edge count is 0, then currentE's terminating vertex is added to the no-incoming-edges vertex list.

Because each loop iteration can remove any vertex from the no-incoming-edges list, the algorithm's output is not guaranteed to be the graph's only possible topological sort.

PARTICIPATION ACTIVITY	14.11.6: Topological sort algorithm.

Animation content:

undefined

Animation captions:

1. The topological sort algorithm begins by initializing an empty result list, a list of all vertices with no incoming edges, and a "remaining edges" list with all edges in the graph.
2. Vertex E is removed from the list of vertices with no incoming edges and added to resultList. Outgoing edges from E are removed from remainingEdges and added to outgoingEdges.
3. Edge EF goes to vertex F, which still has 2 incoming edges. Edge EG goes to vertex G, which still has 1 incoming edge.
4. Vertex A is removed and added to resultList. Outgoing edges from A are removed from remainingEdges. Vertices B and C are added to noIncoming.
5. Vertices C and D are processed, each with 1 outgoing edge.
6. Vertices B and F are processed, each also with 1 outgoing edge.
7. Vertex G is processed last. No outgoing edges remain. The final result is E, A, C, D, B, F, G.

Figure 14.11.1: GraphGetIncomingEdgeCount function.

```
GraphGetIncomingEdgeCount(edgeList,
vertex) {
    count = 0
    for each edge currentE in edgeList {
        if (edge⇢toVertex == vertex) {
            count = count + 1
        }
    }
    return count
}
```

PARTICIPATION ACTIVITY	14.11.7: Topological sort algorithm.

Consider calling GraphTopologicalSort on the graph below.
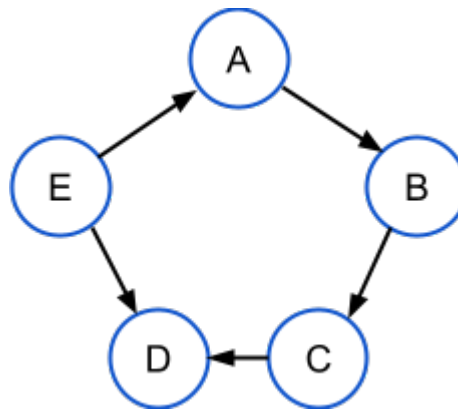
1) In the first iteration of the while loop, what is assigned to currentV?

   ○ Vertex A

   ○ Vertex E

   ○ Undefined

2) In the first iteration of the while loop, what is the contents of outgoingEdges right before the for-each loop begins?

○ Edge from E to A and edge
3) When currentV becomes vertex C, from E to D
what is the contents of resultList?
○ Edge from A to B

○ A, B
○ No edges
○ E, A, B

○ E, D

4) What is the final contents of
resultList?

○ A, B, C, D, E

○ E, A, B, C, D

○ E, A, B, D, C

**PARTICIPATION
ACTIVITY** | 14.11.8: Topological sort matching.
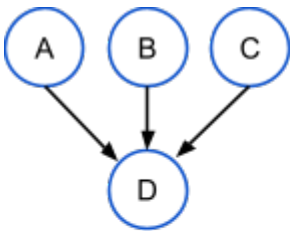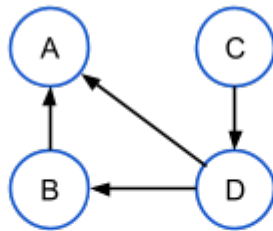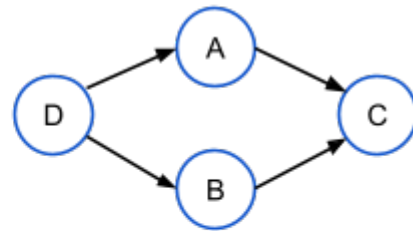
1   2   3

If unable to drag and drop, refresh the page.

Graph 1   **Graph 3**   **Graph 2**

D, B, A, C

C, D, B, A

B, C, A, D

Reset

PARTICIPATION ACTIVITY

14.11.9: Topological sort algorithm.

1) What does
   `GraphTopologicalSort` return?

   ○ A list of vertices.

   ○ A list of edges.

   ○ A list of indices.

2) GraphTopologicalSort will not work
   on a graph with a positive number of
   vertices but no edges.

   ○ True

   ○ False

3) If a graph implementation stores incoming and outgoing
   edge counts in each vertex, then the statement
   `GraphGetIncomingEdgeCount(remainingEdges,`
   `edge⇢to)` can be replaced with
   `currentE⇢toVertex⇢incomingEdgeCount`.

   ○ True

   ○ False

## Algorithm efficiency

The two vertex lists used in the topological sort algorithm will at most contain all
the vertices in the graph. The remaining-edge list will at most contain all edges in
the graph. Therefore, for a graph with a set of vertices V and a set of edges E, the
space complexity of topological sorting is $O(|V| + |E|)$. If a graph
implementation allows for retrieval of a vertex's incoming and outgoing edges in
constant time, then the time complexity of topological sorting is also
$O(|V| + |E|)$.

14.12 Minimum spanning tree

Overview

A graph's **minimum spanning tree** is a subset of the graph's edges that connect all vertices in the graph together with the minimum sum of edge weights. The graph must be weighted and connected. A **connected** graph contains a path between every pair of vertices.

PARTICIPATION ACTIVITY	14.12.1: Using the minimum spanning to minimize total length of power lines connecting cities.

Animation captions:

1. A minimum spanning tree can be used to find the minimal amount of power lines needed to connect cities. Each vertex represents a city. Edges represent roads between cities. The city P has a power plant.
2. Power lines are along roads, such that each city is connected to a powered city. But power lines along every road would be excessive.
3. The minimum spanning tree, shown in red, is the set of edges that connect all cities to power with minimal total power line length.
4. The resulting minimum spanning tree can be viewed as a tree with the power plant city as the root.

PARTICIPATION ACTIVITY	14.12.2: Minimum spanning tree.

1) If no path exists between 2 vertices in a weighted and undirected graph, then no minimum spanning tree exists for the graph.

   ○ True

   ○ False

2) A minimum spanning tree is a set of vertices.

   ○ True

   ○ False

3) The "minimum" in "minimum spanning tree" refers to the sum of edge weights.

○ True

4) A minimum spanning tree can only be
   built for an undirected graph.

    ○ False

    ○ True

    ○ False

Kruskal's minimum spanning tree algorithm

Kruskal's minimum spanning tree algorithm determines subset of the graph's edges that connect all vertices in an undirected graph with the minimum sum of edge weights. Kruskal's minimum spanning tree algorithm uses 3 collections:

- An edge list initialized with all edges in the graph.
- A collection of vertex sets that represent the subsets of vertices connected by current set of edges in the minimum spanning tree. Initially, the vertex sets consists of one set for each vertex.
- A set of edges forming the resulting minimum spanning tree.

The algorithm executes while the collection of vertex sets has at least 2 sets and the edge list has at least 1 edge. In each iteration, the edge with the lowest weight is removed from the list of edges. If the removed edge connects two different vertex sets, then the edge is added to the resulting minimum spanning tree, and the two vertex sets are merged.

PARTICIPATION ACTIVITY	14.12.3: Minimum spanning tree algorithm.

Animation content:

undefined

Animation captions:

1. An edge list, a collection of vertex sets, and an empty result set are initialized. The edge list contains all edges from the graph.
2. Edge AD is removed from the edge list and added to resultList, which will contain the edges forming the minimum spanning tree.
3. The next 5 edges connect different vertex sets and are added to the result.
4. Edges AB and CE both connect 2 vertices that are in the same vertex set, and therefore are not added to the result.
5. Edge EF connects the 2 remaining vertex sets.
6. One vertex set remains, so the minimum spanning tree is complete.

14.12.4: Minimum spanning tree algorithm.

Consider executing Kruskal's minimum spanning tree algorithm on the following graph:
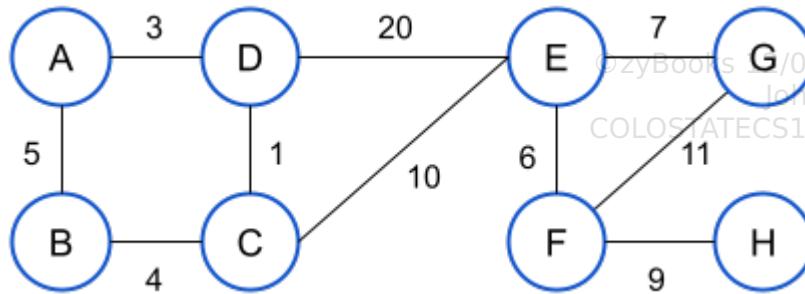
1) What is the first edge that will be
    added to the result?

    ○ AD

    ○ AB

    ○ BC

    ○ CD

2) What is the second edge that will be
    added to the result?

    ○ AD

    ○ AB

    ○ BC

    ○ CD

3) What is the first edge that will NOT be
    added to the result?

    ○ BC

    ○ AB

    ○ FG

    ○ DE

4) How many edges will be in the
    resulting minimum spanning tree?

○ 5

PARTICIPATION ACTIVITY | 14.12.5: Minimum spanning tree - critical thinking.

1) The edge with the lowest weight will always be in the minimum spanning tree.

○ True

○ False

2) The minimum spanning tree may contain all edges from the graph.

○ True

○ False

3) Only 1 minimum spanning tree exists for a graph that has no duplicate edge weights.

○ True

○ False

4) The edges from any minimum spanning tree can be used to create a path that goes through all vertices in the graph without ever encountering the same vertex twice.

○ True

○ False

Algorithm efficiency

Kruskal's minimum spanning tree algorithm's use of the edge list, collection of vertex sets, and resulting edge list results in a space complexity of $O(|E| + |V|)$. If the edge list is sorted at the beginning, then the minimum edge can be removed in constant time within the loop. Combined with a mechanism to map a vertex to the containing vertex set in constant time, the minimum spanning tree algorithm has a runtime complexity of $O(|E|log|E|)$.

CHALLENGE ACTIVITY	14.12.1: Minimum spanning tree.	

422352.2723990.qx3zqy7

14.13 All pairs shortest path

> ℹ This section has been set as optional by your instructor.

Overview and shortest paths matrix

An **all pairs shortest path** algorithm determines the shortest path between all possible pairs of vertices in a graph. For a graph with vertices V, a |V| x |V| matrix represents the shortest path lengths between all vertex pairs in the graph. Each row corresponds to a start vertex, and each column in the matrix corresponds to a terminating vertex for each path. Ex: The matrix entry at row F and column T represents the shortest path length from vertex F to vertex T.

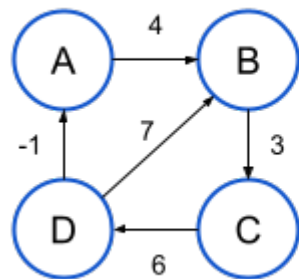| PARTICIPATION ACTIVITY	14.13.1: Shortest path lengths matrix.	

Animation captions:

1. For a graph with 4 vertices, the all-pairs-shortest-path matrix has 4 rows and 4 columns. An entry exists for every possible vertex pair.
2. An entry at row F and column T represents the shortest path length from vertex F to vertex

T.

3. The shortest path from vertex A to vertex D has length 9.
4. The shortest path from B to A has length 2.
5. Only total path lengths are stored in the matrix, not the actual sequence of edges for the corresponding path.

PARTICIPATION ACTIVITY   14.13.2: Shortest path lengths matrix.

	A	B	C	D
A	0	4	7	?
B	8	0	3	9
C	5	9	?	6
D	-1	?	6	0

1) What is the shortest path length from A to D?

    [ ]

Check   **Show answer**

2) What is the shortest path length from C to C?

    [ ]

Check   **Show answer**

3) What is the shortest path length from D to B?

    [ ]

Check   **Show answer**

PARTICIPATION ACTIVITY   14.13.3: Shortest path lengths matrix.

1) If a graph has 11 vertices and 7 edges, how many entries are in the all-pairs-shortest-path matrix?

   ○ 11

   ○ 7

   ○ 77

   ○ 121

2) An entry at row R and column C in the matrix represents the shortest path length from vertex C to vertex R.

   ○ True

   ○ False

3) If a graph contains a negative edge weight, the matrix of shortest path lengths will contain at least 1 negative value.

   ○ True

   ○ False

4) For a matrix entry representing path length from A to B, when no such path exists, a special-case value must be used to indicate that no path exists.

   ○ True

   ○ False

Floyd-Warshall algorithm

The **Floyd-Warshall all-pairs shortest path algorithm** generates a |V| x |V| matrix of values representing the shortest path lengths between all vertex pairs in a graph. Graphs with cycles and negative edge weights are supported, but the graph must not have any negative cycles. A **negative cycle** is a cycle with edge weights that sum to a negative value. Because a negative cycle could be traversed repeatedly, lowering the path length each time, determining a shortest path between 2 vertices in a negative cycle is not possible.

The Floyd-Warshall algorithm initializes the shortest path lengths matrix in 3 steps.

1. Every entry is assigned with infinity.
2. Each entry representing the path from a vertex to itself is assigned with 0.
3. For each edge from X to Y in the graph, the matrix entry for the path from X to Y is initialized
   with the edge's weight.

The algorithm then iterates through every vertex in the graph. For each vertex X, the shortest path
lengths for all vertex pairs are recomputed by considering vertex X as an intermediate vertex. For
each matrix entry representing A to B, existing matrix entries are used to compute the length of the
path from A through X to B. If this path length is less than the current shortest path length, then the
corresponding matrix entry is updated.

**PARTICIPATION
ACTIVITY** | 14.13.4: Floyd-Warshall algorithm.

## Animation content:

undefined

Animation captions:

1. All entries in the shortest path lengths matrix are first initialized with ∞. Vertex-to-same-
   vertex values are then initialized with 0. For each edge, the corresponding entry for from X
   to Y is initialized with the edge's weight.
2. The k = 0 iteration corresponds to vertex A. Each vertex pair X, Y is analyzed to see if the
   path from X through A to Y yields a shorter path. Shorter paths from C to B and from D to B
   are found.
3. During the k = 1 iteration, 4 shorter paths that pass through vertex B are found from path
   from A to C, from A to D, from C to D, and from D to C.
4. During the k = 2 iteration, 1 shorter path that passes through vertex C is found from the
   path from B to A.
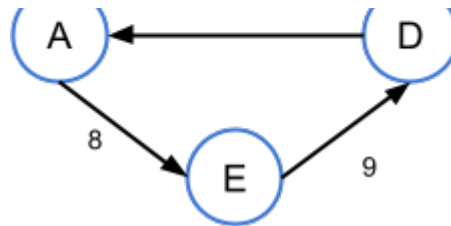5. During the k=3 iteration, no entries are updated. The shortest path lengths matrix is
   complete.

**PARTICIPATION
ACTIVITY** | 14.13.5: Floyd-Warshall algorithm.

Consider executing the Floyd-Warshall algorithm on the graph below.
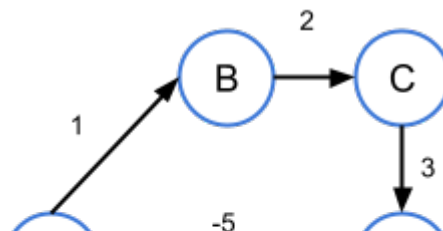
1) How many rows and columns are in the shortest path length matrix?

○ 5 rows, 5 columns

○ 5 rows, 6 columns

○ 6 rows, 5 columns

○ 6 rows, 6 columns

2) After matrix initialization, but before the k-loop starts, how many entries are set to non-infinite values?

○ 5

○ 11

○ 14

○ 25

3) After the algorithm completes, how many entries in the matrix are negative values?

○ 0

○ 3

○ 5

○ 7

4) After the algorithm completes, what is the shortest path length from vertex B to vertex E?

○ -4

○ 5

○ 8

○ Infinity (no path)

Path reconstruction

Although only shortest path lengths are computed by the Floyd-Warshall algorithm, the matrix can be used to reconstruct the path sequence. Given the shortest path length from a start vertex to an end vertex is L. An edge from vertex X to the ending vertex exists such that the shortest path length from the starting vertex to X, plus the edge weight, equals L. Each such edge is found, and the path is reconstructed in reverse order.

PARTICIPATION ACTIVITY	14.13.6: Path reconstruction.

Animation captions:

1. The matrix built by the Floyd-Warshall algorithm indicates that the shortest path from A to C is of length 3.
2. The path from A to C is determined by backtracking from C.
3. Since A is the path's starting point, shortest distances from A are relevant at each vertex.
4. Traversing backwards along the only incoming edge to C, the expected path length at B is 3 - 5 = -2.
5. The computation at the edge from A to B doesn't hold, so the edge is not part of the path.
6. The computation on the D to B edge holds.
7. The computation on the A to D edge holds, and brings the path back to the starting vertex. The final path is A to D to B to C.

Figure 14.13.1: FloydWarshallReconstructPath algorithm.

```
FloydWarshallReconstructPath(graph, startVertex, endVertex,
distMatrix) {
    path = new, empty path

    // Backtrack from the ending vertex
    currentV = endVertex
    while (currentV != startVertex) {
        incomingEdges = all edges in the graph incoming to current
vertex
        for each edge currentE in incomingEdges {
            expected = distMatrix[startVertex][currentV] -
currentE⇢weight
            actual = distMatrix[startVertex][currentE⇢fromVertex]
            if (expected == actual) {
                currentV = currentE⇢fromVertex
                Prepend currentE to path
                break
            }
        }
    }

    return path
}
```

PARTICIPATION
ACTIVITY   14.13.7: Path reconstruction.

1) Path reconstruction from vertex X to
   vertex Y is only possible if the matrix
   entry is non-infinite.

   ○ True

   ○ False

2) A path with positive length may
   include edges with negative weights.

   ○ True

   ○ False

3) More than 1 possible path sequence
   from vertex X to vertex Y may exists,
   even though the matrix will only store
   1 path length from X to Y.

  ○ True
  ○ False
4) Path reconstruction is not possible if
   the graph has a cycle.

  ○ True

  ○ False

Algorithm efficiency

The Floyd-Warshall algorithm builds a |V| x |V| matrix and therefore has a space complexity of $O(|V|^2)$. The matrix is constructed with a runtime complexity of $O(|V|^3)$.

14.14 PA12: Graph Manipulation

Objectives

Learn how to build and manipulate graph data structures

Getting Started

The following files are included with this assignment:

```
P8/
├── resources
│   ├── MileagesLarge.csv
│   ├── MileagesSmall.csv
│   └── MileagesExtraSmall.csv
└── src
    ├── GraphAbstract.java
    └── GraphImplementation.java
```

Part One

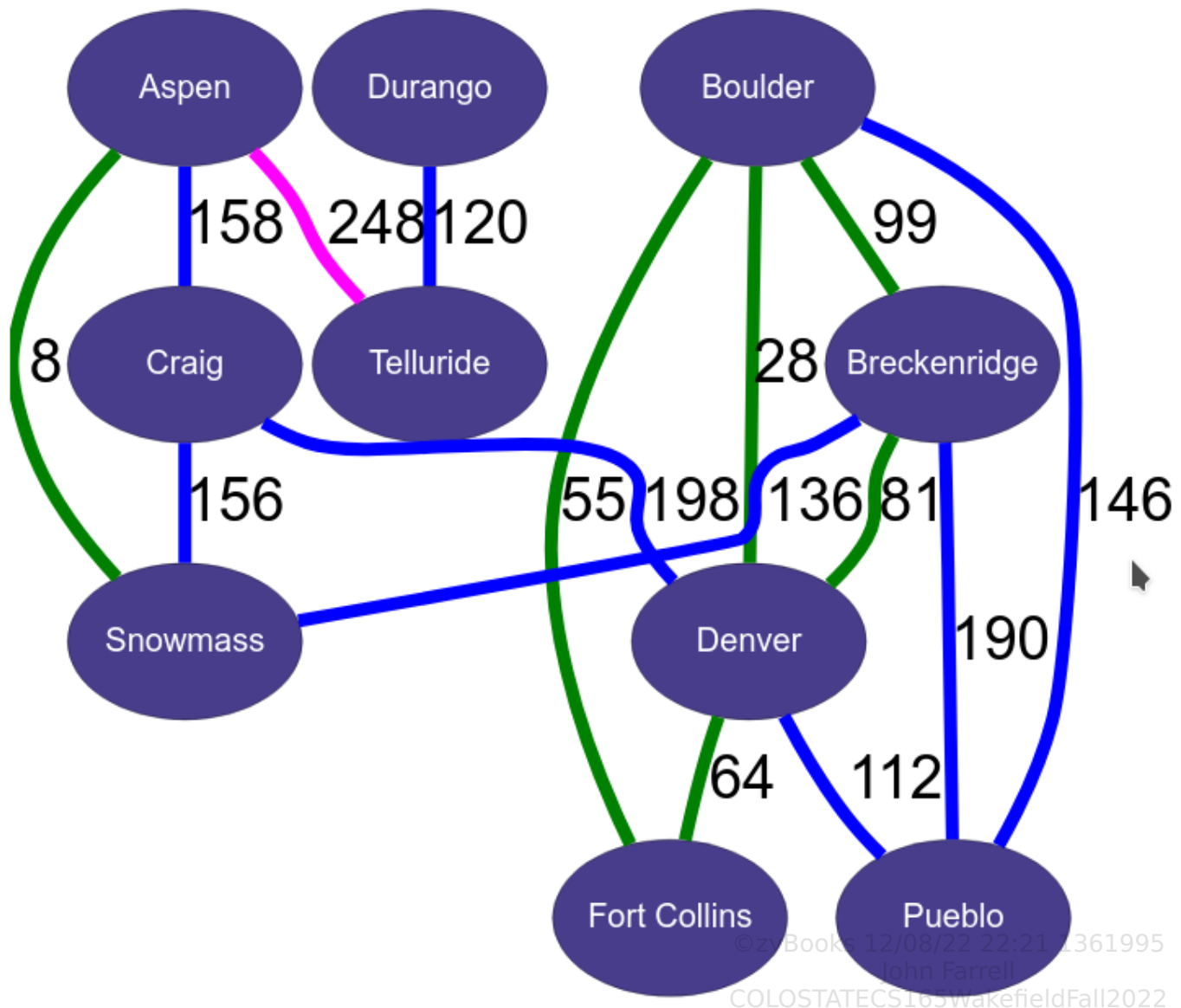Implement the readGraph() method

Testing

Enter an input mileages file and a starting city in your run configurations, such as below:

```
MileagesSmall.csv
Fort Collins
```

This is the graph of MileagesSmall.csv for reference:

Part Two

The second part of this assignment requires you to implement two graph algorithms:

1. Depth first search (DFS) starting at any arbitrary node.

2. `Breadth first search (BFS)` starting at any arbitrary node.

Check out the corresponding lecture slides for details on how to implement these algorithms. The cities are ordered alphabetically in each input file. Make sure to traverse through the cities based on their mileages. You should visit cities with a smaller distance before cities with a greater distance.

Here is the expected output from the supplied chart for the `depthFirst()` and `breadthFirst()` methods, called with Fort Collins and Aspen as the starting cities:

```
Depth First Search:
Visited Fort Collins
Visited Boulder
Visited Denver
Visited Breckenridge
Visited Snowmass
Visited Aspen
Visited Craig
Visited Telluride
Visited Durango
Visited Pueblo

Breadth First Search:
Visited Aspen
Visited Snowmass
Visited Craig
Visited Telluride
Visited Breckenridge
Visited Denver
Visited Durango
Visited Boulder
Visited Pueblo
Visited Fort Collins
```

Specifications

Your program must meet the following specifications:

- Work on your own, as always.

- Submit the file `GraphImplementation.java`.

- Assignments should be implemented using Java 1.8.

- Submit your program on zyBooks.

- Read the syllabus for the late policy.

- We will be checking programs for plagiarism, so please don't copy from anyone else.

Grading Criteria (Part A)

- 100 points for perfect submission.

- 0 points for no submission, will not compile, etc.

- Preliminary Tests: a compilation check. Use the information we have provided for MileageSmall.csv mileage chart.

- Final Tests:

  - testing DFS and BFS on MileagesExtraSmall.csv with two different start cities (20 points)
  - testing DFS and BFS on MileagesSmall.csv with four different start cities (40 points)
  - testing DFS and BFS on MileagesLarge.csv with four different start cities (40 points)

422352.2723990.qx3zqy7

LAB ACTIVITY	14.14.1: PA12: Graph Manipulation	0 / 100

Downloadable files

`MileagesLarge.csv` , `MileagesExtraSmall.csv` , and

`MileagesSmall.csv`

Download

Current file: **GraphImplementation.java** ▾   Load default template...

```
 1  // GraphImplementation.java - supplied code for graph assignment
 2
 3  import java.io.File;
 4  import java.io.IOException;
 5  import java.io.PrintWriter;
 6  import java.util.*;
 7
 8  public class GraphImplementation extends GraphAbstract {
 9
10      // Main entry point
11      public static void main(String[] args) {
12
13        // Instantiate code
14          GraphImplementation impl = new GraphImplementation();
15          Scanner input = new Scanner(System.in);
16          String fileName = input.nextLine();
17          String startCity = input.nextLine();
```

Develop mode	Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program   Input (from above) ⟶   **GraphImplementation.java**
(Your program)

Program output displayed here

Coding trail of your work   What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

14.15 Lab 25 - Graph: Adjacency List (BFS)

Lab 25: Graph - Adjacency List - BFS

Graphs

This lab includes the following .java files:

L25/
└── Graph.java
└── GraphTest.java*****

***** This is the main, and it is a read-only file in zyBooks.

Jar Link: L25.jar

Overview

Adjacency Lists are a fast and relatively easy way to store graphs. This lab should help you to grasp how they work. In the image below, there is an adjacency list for each vertex. For example, vertex 1 has edges connecting to vertices 2 and 3, so its adjacency list is [2, 3]. For vertex 2, the adjacency list is [1,5].

You will also be implementing a breadth first search which searches the graph from a given node in the order given by how many edges the node is away from the starting node. For example, it would search nodes 1 edge away first and then 2 edges away and so on. Below is an example of how to do a BFS using a queue:

First, we queue the start node. Then, we add all of the neighboring nodes that are *unvisited* and output the node at the front of the queue and repeat until the queue is empty.
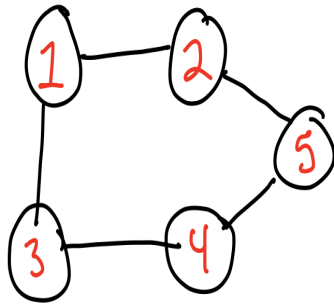
ADJACENCY LISTS:

1: [2,3]   [[2,3],
2: [1,5]   [1,5],
3: [1,4]   =>   [1,4],
4: [3,5]   [3,5],
5: [2,4]   [2,4]]

BREADTH FIRST SEARCH (BFS):

Starting at 1:

Queve	Output
1	–
2,3	1
3,5	1,2
5,4	1,2,3
4	1,2,3,5
–	1,2,3,4,5

Completing the Code

First complete the addEdge() method and then run the code in main() to make sure that your graph is being constructed correctly.

Once you are confident addEdge() is correct implement breadthFirst() and make sure that your output is in the same order as shown below under Testing BFS.

Testing addEdge

Run the main method and you should get this output if your code is correct:

Note: this also checks that you throw the exception when the node isn't in the graph.

```
Node 1: [8, 2, 8, 3, 8]
Node 2: [3, 6, 7, 3, 3, 4, 6, 7, 8]
Node 3: [0, 4, 5, 6, 5, 7, 9]
Node 4: [1, 0, 1, 1, 6]
Node 5: [2, 1, 6, 5, 6, 7]
Node 6: [2, 4, 2, 7, 6]
Node 7: [1, 2, 4, 5, 4, 3, 1, 8, 9, 9]
Node 8: [1, 5, 2, 4, 1]
Node 9: [0, 0, 6, 0, 1]
Node 10: [6, 6, 2]

You threw the exception. Yay!

BFS 1:
Visited Node 0
Visited Node 8
Visited Node 2
Visited Node 3
Visited Node 6
Visited Node 1
Visited Node 4
Visited Node 5
Visited Node 7
Visited Node 9

BFS 2:
Visited Node 6
Visited Node 1
Visited Node 2
Visited Node 4
Visited Node 5
Visited Node 3
```

```
Visited Node 8
Visited Node 9
Visited Node 7
Visited Node 0

BFS 3:
Visited Node 4
Visited Node 2
Visited Node 1
Visited Node 6
Visited Node 5
Visited Node 7
Visited Node 0
Visited Node 9
Visited Node 3
Visited Node 8
```

422352.2723990.qx3zqy7

LAB ACTIVITY	14.15.1: Lab 25 - Graph: Adjacency List (BFS)	0 / 7

Downloadable files

Graph.java   and   GraphTest.java   **Download**

Current file: **Graph.java** ▾

```
1 Loading latest submission...
```

Develop mode	Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

> If your code requires input values, provide them here.

Run program   Input (from above) ➡️   **Graph.java** ➡️   O
                                            (Your program)

Program output displayed here

Coding trail of your work   **What is this?**

⟳ Retrieving signature

14.16 Lab 26 - Graph: Link References (DFS)

Lab 26: Graph - Link References - DFS

Graphs - Link References and Depth First Search

This lab includes the following .java files:

L26/
└── Graph.java
└── GraphAbstract.java
└── GraphTest.java*****

* This is the main, and it is a read-only file in zyBooks.

Here's the jar: L26.jar

Link References

Link references are another way to represent edges among a graph's nodes. Unlike adjacency lists, link references rely on each individual node to determine which node it points to. In the code you will be working with, your graph will be made of GraphNodes, a subclass of Graph. These nodes will have an ArrayList called neighbors, which will contain all of the nodes that specific node points to. The image below is a graph you will be working with in the code for this lab.
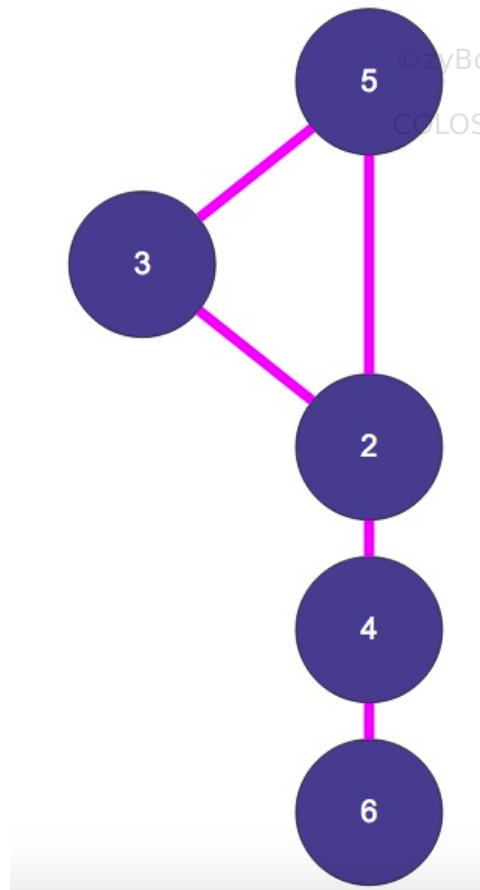
The node containing 5 will have an ArrayList of nodes called neighbors. It will contain {3, 2} because 5 connects to those two nodes. The node containing 6 will only have {4} as it only connects to that one node.

Depth First Search

The depth first search algorithm works in a way that is reasonable given its title. Basically, it works in that you start at a node, go to one of its neighbors, go one of that nodes neighbors, until you reach a node with no neighbors (that you haven't visited). This means that there are technically multiple possible DFSs for the above graph starting at node 5, 2 or 3. To display my point two possible DFSs starting at node 5 are {5, 3, 2, 4, 6} and {5, 2, 3, 4, 6}.

Sometimes, you will have weighted edges which can be used to determine which neighbor node is chosen, but not in this lab. In your code, if you write it to the specifications, there will be no ambiguity because you will simply go off of the order in which you added nodes and edges. Note: when you print the nodes out in DFS order, you should print a space between each node. (A trailing space after the last node is fine.)

422352.2723990.qx3zqy7

LAB ACTIVITY	14.16.1: Lab 26 - Graph: Link References (DFS)	0 / 7

Downloadable files

`Graph.java` , `GraphAbstract.java` , and

`GraphTest.java`

Download

Current file: **Graph.java** ▾   Load default template...

```
 1  /* Graph class
 2  * Template made by Toby Patterson
 3  * With the help of: https://www.cs.colostate.edu/~cs165/.Spring20/assignments/P8/doc/
 4  * Completed by: your name here
 5  * 6/2/2020
 6  * For cs165 at CSU
 7  *
 8  * A basic graph which has a depth first search print method. It uses
 9  * link references to keep track of edges.
10  * */
11
12  import java.io.IOException;
13  import java.io.PrintWriter;
14  import java.util.ArrayList;
15
16  public class Graph<E extends Comparable<E>> extends GraphAbstract<E> {
17      public Graph() {
```

Develop mode	**Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program   Input (from above) ➡️   **Graph.java**   ➡️ O
                                                         (Your program)

Program output displayed here

Coding trail of your work   What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```