11.1 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 11.1.1: 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 | 11.1.1: 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 11.1.2: 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
   - 16

©zyBooks 11/20/23 11:09 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

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.

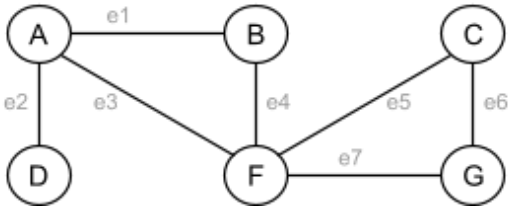
PARTICIPATION ACTIVITY 11.1.3: Graphs: adjacency, paths, and distance.

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 11.1.4: 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

11.2 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 11.2.1: 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 11.2.2: 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 11.2.3: 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 11.2.4: 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 11.2.5: 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 11.2.6: 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 11.2.7: 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

11.3 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 11.3.1: Adjacency list graph representation.

Animation captions:

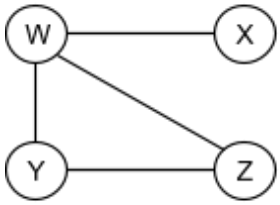
1. Each vertex has a list of adjacent vertices for edges. The edge connecting A and B appears in A's list and also in B's list.
2. Each edge appears in the lists of the edge's two vertices.

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 11.3.2: Adjacency lists.



Vertices	Adjacent vertices
W	?
X	W
Y	?
Z	?

1) What are vertex W's adjacent vertices?

- Y, Z
- X, Y, Z

2) What are vertex Y's adjacent vertices?

- W, X, Z
- W, Z
- Z

3) What are vertex Z's adjacent vertices?

- W
- W, X
- W, Y

PARTICIPATION ACTIVITY 11.3.3: 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

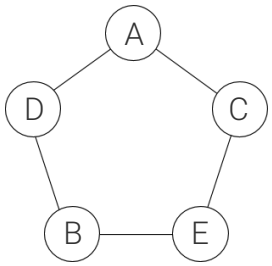
4) Is the following a path from Livermore to Marin City? Type: Yes or No
Livermore, Hillsborough, Sonoma, Marin City

Check Show answer

CHALLENGE ACTIVITY 11.3.1: Adjacency lists.

Start

Select the missing vertices to complete the adjacency list representation of the given graph.



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

(1):
(2):
(3):
(4):

1 2

Check Next

11.4 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 11.4.1: 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).

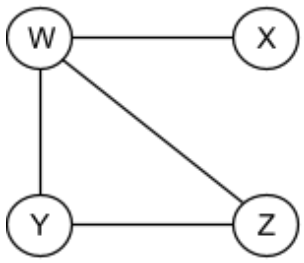
Analysis of adjacency matrices

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) 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 11.4.2: 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 11.4.3: 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

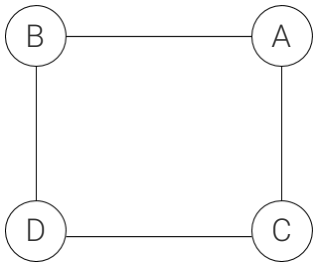
Check Show answer

CHALLENGE ACTIVITY 11.4.1: Adjacency matrices.

502696.2096894.qx3zqy7

Start

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



	A	B	C	D
A	0	1	(w)	0
B	1	0	(x)	1
C	1	(y)	0	1
D	0	(z)	1	0

(w):
(x):
(y):
(z):

1 2

Check Next

11.5 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.

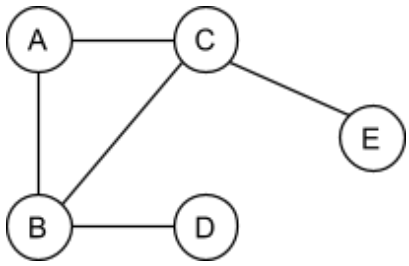
PARTICIPATION ACTIVITY 11.5.1: Breadth-first search.

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 11.5.2: 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

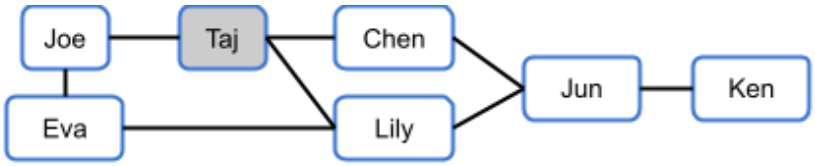
6) The BFS traversal of a graph is unique. Type: Yes or No

Check Show answer

Example: Social networking connection recommender

Example 11.5.1: 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 11.5.3: 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

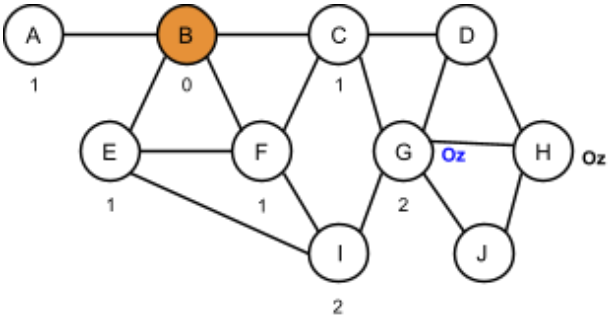
2) People with a distance of 2 are recommended before people with a distance of 3.
   - True
   - False

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 11.5.2: 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).



Visit order:	B	A	E	F	C	I	G
	0	1	1	1	1	2	2

Oz found; terminates

PARTICIPATION ACTIVITY 11.5.4: 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 11.5.5: BFS algorithm.

Animation content:

undefined

Animation captions:

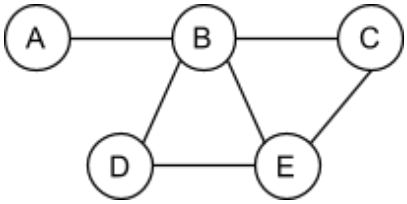
1. BFS enqueues the start vertex (in this case A) in frontierQueue, and adds A to discoveredSet.
2. Vertex A is dequeued from frontierQueue and visited.
3. Undiscovered vertices adjacent to A are enqueued in frontierQueue and added to discoveredSet.
4. Vertex A's visit is complete. The process continues 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 11.5.6: 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.

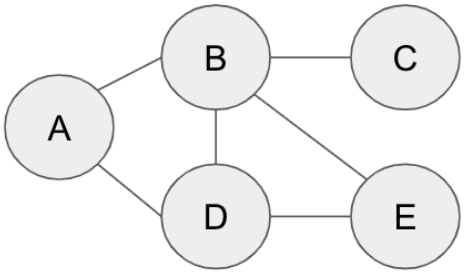
- True
- False

CHALLENGE ACTIVITY 11.5.1: Breadth-first search.

502696.2096894.qx3zqy7

Start

Given the graph below and the starting vertex E:



What is E's distance? Ex: 5

What is D's distance?

What is A's distance?

What is C's distance?

What is B's distance?

1 2 3 4

Check Next

11.6 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.

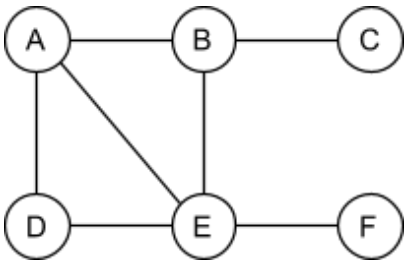
PARTICIPATION ACTIVITY 11.6.1: Depth-first search.

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 11.6.2: 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 11.6.3: Depth-first search traversal with starting vertex A.

Animation content:

undefined

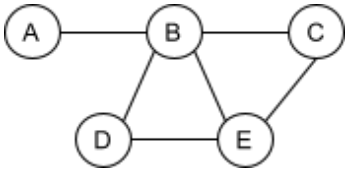
Animation captions:

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 11.6.4: 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

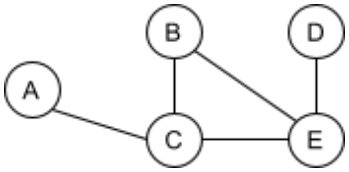
A recursive DFS can be implemented using the program stack instead of an explicit stack. The recursive DFS algorithm is first called with the starting vertex. If the vertex has not already been visited, the recursive algorithm visits the vertex and performs a recursive DFS call for each adjacent vertex.

Figure 11.6.1: 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 11.6.5: 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

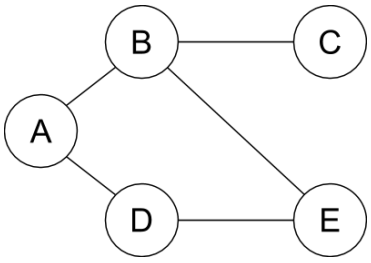
3) If B is not yet visited, RecursiveDFS(B) will make subsequent calls to RecursiveDFS(C) and RecursiveDFS(E).

- True
- False

CHALLENGE ACTIVITY 11.6.1: Graphs: Depth-first search.

502696.2096894.qx3zqy7

Start



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

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

1 2 3 4 5

Check Next

11.7 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 11.7.1: Directed graph examples: Process flow diagram, airline routes, and college course prerequisites.

Source: Fluid catalytic 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 11.7.1: 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 11.7.2: 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 11.7.3: Directed graph: Paths and cycles.

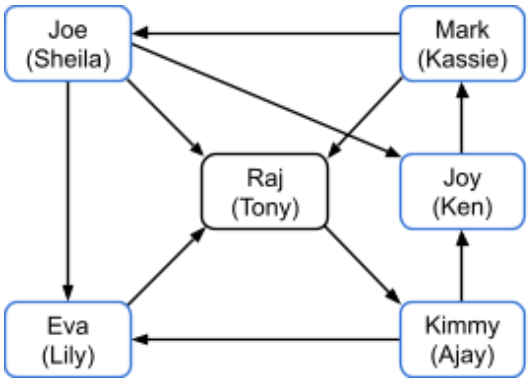
Animation captions:

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 11.7.1: 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)

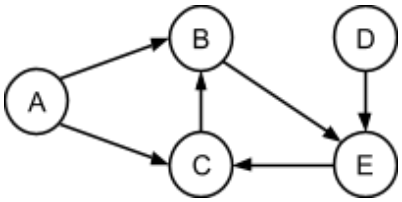
4-way transplant: Raj (Tony), Kimmy (Ajay), Joy (Ken), Mark (Kassie)

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 11.7.4: Directed graphs: Cyclic and acyclic.

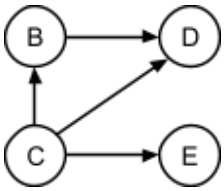
Determine if each of the following graphs is cyclic or acyclic.

1)



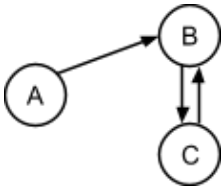
- Cyclic
- Acyclic

2)



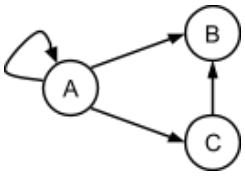
- Cyclic
- Acyclic

3)



- Cyclic
- Acyclic

4)



- Cyclic
- Acyclic

CHALLENGE ACTIVITY 11.7.1: Directed graphs.

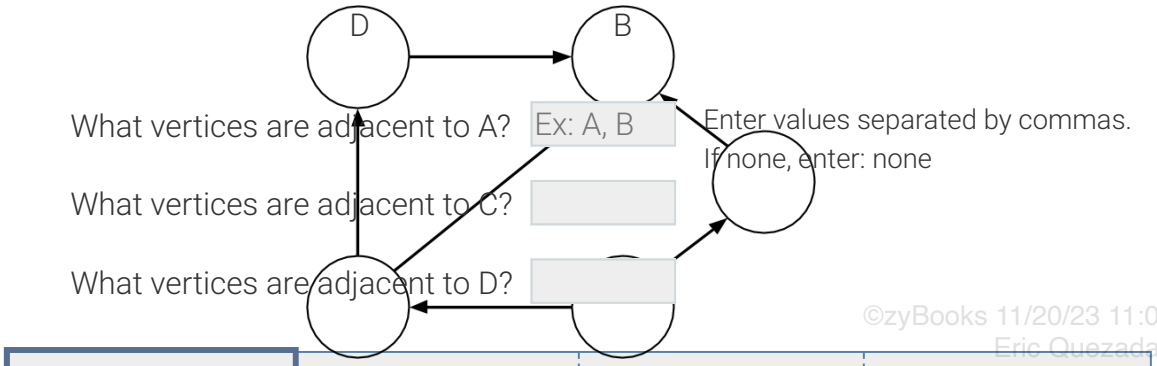
502696.2096894.qx3zqy7

Start

Consider the following directed graph.

A E

C



Check Next

11.8 Weighted graphs

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.

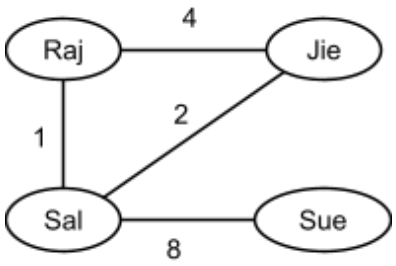
PARTICIPATION ACTIVITY 11.8.1: Weighted graphs associate weight with each edge.

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.

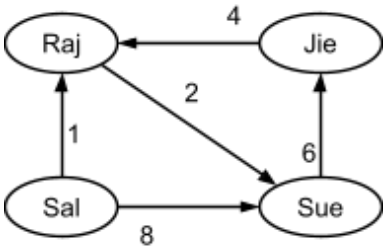
PARTICIPATION ACTIVITY 11.8.2: Weighted graphs.

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



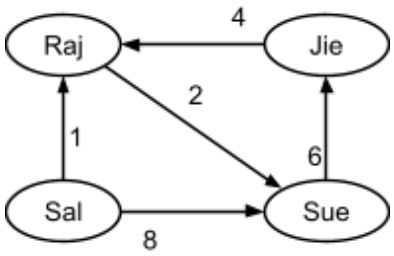
- 1
- 4

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



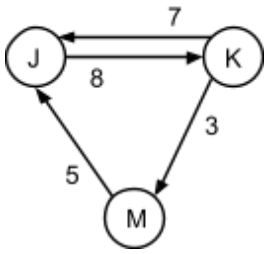
- 0
- 8

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.

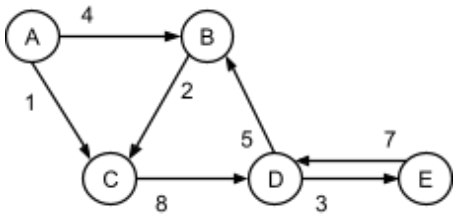
PARTICIPATION ACTIVITY 11.8.3: Path length is the sum of edge weights.

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.

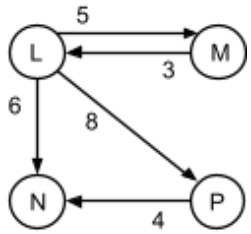
PARTICIPATION ACTIVITY 11.8.4: Path length and shortest path.

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



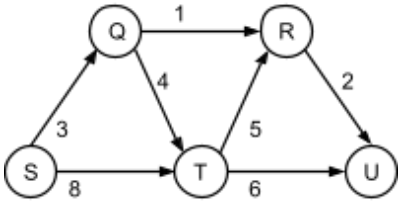
Check Show answer

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



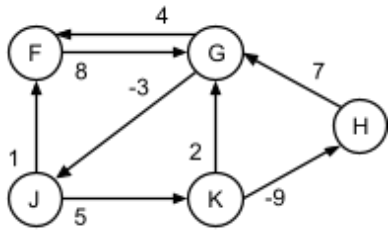
Check Show answer

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 _____.

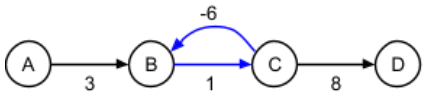


Check Show answer

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.

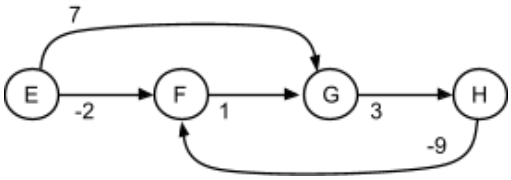
Figure 11.8.1: A shortest path from A to D does not exist, because the cycle B, C can be repeatedly taken to further reduce the cycle length.



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...	

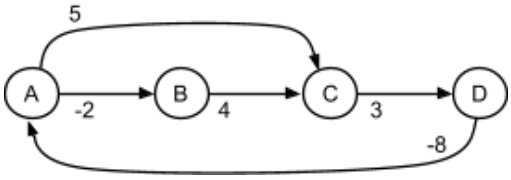
PARTICIPATION ACTIVITY 11.8.5: Negative edge weight cycles.

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



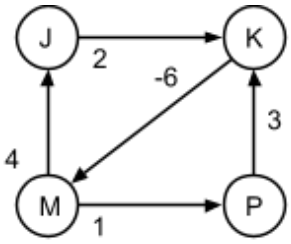
Check Show answer

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.

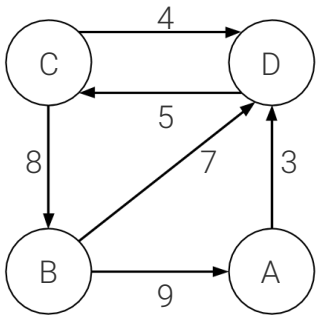


Check Show answer

CHALLENGE ACTIVITY 11.8.1: Weighted graphs.

502696.2096894.qx3zqy7

Start



Find the weight of each of the following edges.

Weight of edge from B 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 C to D:

1 2 3 4 5

Check Next

11.9 Python: Graphs

Building the Graph and Vertex classes

The Graph class holds a vertex adjacency list using a dictionary that maps a Vertex object to a list of adjacent Vertex objects. The Vertex class contains a label, but can be augmented by graph algorithms to contain additional data if required.

A Graph object is initialized with an empty adjacency list. Vertex objects are created and added to the Graph using the add_vertex() method.

Figure 11.9.1: Graph and Vertex classes.

Graph and Vertex classes	Example adding vertices to a graph

```
class Vertex:
    def __init__(self, label):
        self.label = label

class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_vertex(self, new_vertex):

self.adjacency_list[new_vertex] = []
```

```
# Program to create and populate a
Graph object.
g = Graph()
vertex_a = Vertex("New York")
vertex_b = Vertex("Tokyo")
vertex_c = Vertex("London")

g.add_vertex(vertex_a)
g.add_vertex(vertex_b)
g.add_vertex(vertex_c)
```


Graph edges and edge weights

Edges are represented as vertex *pairs*, using a 2-item tuple. Ex: (vertex_a, vertex_b) is an edge that goes from vertex_a to vertex_b. Undirected edges are two symmetric vertex pairs: (vertex_a, vertex_b) and (vertex_b, vertex_a). Edges also have numeric *weights*. By default, an edge is assigned with weight 1.0. Edge weights are stored in the dictionary edge_weights where the vertex pair is the key and the edge weight is the value. Ex: The edge (vertex_a, vertex_b) is assigned with weight 3.7 by `self.edge_weights[(vertex_a, vertex_b)] = 3.7`.

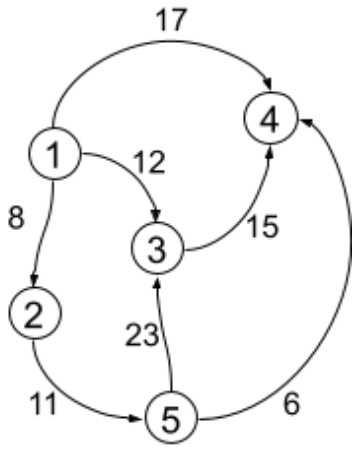
The method add_directed_edge() is used to add edges to a graph. To store the edge (vertex_a, vertex_b), vertex_b is appended to self.adjacency_list[vertex_a]. The method add_undirected_edge() calls add_directed_edge() twice to add both of the edge's symmetric versions. Both add_directed_edge() and add_undirected_edge() use a default parameter for the edge's weight, assigned with 1.0 if the method is called without the last parameter.

Figure 11.9.2: Graph class methods: add_directed_edge() and add_undirected_edge().

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}
        self.edge_weights = {}

    def add_vertex(self, new_vertex):
        self.adjacency_list[new_vertex] = []

    def add_directed_edge(self, from_vertex, to_vertex, weight =
1.0):
        self.edge_weights[(from_vertex, to_vertex)] = weight
        self.adjacency_list[from_vertex].append(to_vertex)

    def add_undirected_edge(self, vertex_a, vertex_b, weight =
1.0):
        self.add_directed_edge(vertex_a, vertex_b, weight)
        self.add_directed_edge(vertex_b, vertex_a, weight)
```

Figure 11.9.3: A directed graph showing water flow through a system of pumping stations and a program to create the graph.

Graph with directed edges: Vertices (1, 2, 3, 4, 5) represent water pump stations. Edges represent water flow from one station to another, with edge weights indicating the capacity of flow that is possible along that route.




```
g = Graph()
vertex_a = Vertex("1")
vertex_b = Vertex("2")
vertex_c = Vertex("3")
vertex_d = Vertex("4")
vertex_e = Vertex("5")
g.add_vertex(vertex_a)
g.add_vertex(vertex_b)
g.add_vertex(vertex_c)
g.add_vertex(vertex_d)
g.add_vertex(vertex_e)

g.add_directed_edge(vertex_a, vertex_b, 8)
g.add_directed_edge(vertex_a, vertex_c, 12)
g.add_directed_edge(vertex_a, vertex_d, 17)
g.add_directed_edge(vertex_b, vertex_e, 11)
g.add_directed_edge(vertex_e, vertex_c, 23)
g.add_directed_edge(vertex_c, vertex_d, 15)
g.add_directed_edge(vertex_e, vertex_d, 6)
```


Figure 11.9.4: An undirected graph of flight distances between cities.

Graph with undirected edges: The vertices are cities, and the edges represent flights between the cities, with distance in miles as the edge weights.


```
g = Graph()
vertex_a = Vertex("Tokyo")
vertex_b = Vertex("New York")
vertex_c = Vertex("London")
vertex_d = Vertex("Sydney")
g.add_vertex(vertex_a)
g.add_vertex(vertex_b)
g.add_vertex(vertex_c)
g.add_vertex(vertex_d)

g.add_undirected_edge(vertex_a, vertex_b, 6743)
g.add_undirected_edge(vertex_a, vertex_c, 5941)
g.add_undirected_edge(vertex_a, vertex_d, 4863)
g.add_undirected_edge(vertex_b, vertex_c, 3425)
g.add_undirected_edge(vertex_b, vertex_d, 9868)
g.add_undirected_edge(vertex_c, vertex_d, 10562)
```

PARTICIPATION ACTIVITY 11.9.1: Python Graph class.

1) The Graph class' adjacency_list data member is:

   - a list
   - a matrix
   - a dictionary

2) An undirected edge (a, b) is represented as two directed edges, (a, b) and (b, a).

   - True
   - False

3) Which statement is false?

- Edges are stored in the Graph class' edges data member.
- Two adjacency_list entries are modified when an undirected edge is added to the graph.
- All edges have weights. If a weight is not explicitly given when added to the graph, the edge is assigned with a default weight of 1.0.

11.10 Python: Breadth-first search

Breadth-first search algorithm

Breadth-first search traverses a graph by starting at a specific vertex and visiting the vertex's adjacent vertices before visiting the next closest vertices. No vertex is re-visited.

The breadth-first search function has three parameters:

- graph is the graph to search.
- start_vertex is the starting vertex of the breadth-first search.
- distances is an optional dictionary to store the distance of each visited vertex from the start vertex.

The breadth-first search function defines three container variables:

- discovered_set is a set of all discovered vertices.
- frontier_queue is a Queue variable (Queue class defined in a previous section) that stores vertices that have been discovered, but not yet visited.
- visited_list is an ordered list of visited vertices.

Initially, the start vertex is added to both frontier_queue and discovered_set. start_vertex is dequeued from the queue and added to the visited list. Then, undiscovered vertices adjacent to the current dequeued vertex are added to discovered_set and frontier_queue. The distances from the adjacent vertices are set to the current dequeued vertex's distance + 1.

The process of dequeuing a vertex from frontier_queue and adding the vertex's adjacent vertices to discovered_set continues until frontier_queue is empty. In the end, visited_list contains the order of the breadth-first search's traversal.

Figure 11.10.1: Breadth-first search algorithm.

```
# Breadth-first search function
def breadth_first_search(graph, start_vertex, distances=dict()):
    discovered_set = set()
    frontier_queue = Queue()
    visited_list = []

    # start_vertex has a distance of 0 from itself
    distances[start_vertex] = 0

    frontier_queue.enqueue(start_vertex) # Enqueue start_vertex in
frontier_queue
    discovered_set.add(start_vertex)     # Add start_vertex to
discovered_set

    while (frontier_queue.list.head != None):
        current_vertex = frontier_queue.dequeue()
        visited_list.append(current_vertex)
        for adjacent_vertex in graph.adjacency_list[current_vertex]:
            if adjacent_vertex not in discovered_set:
                frontier_queue.enqueue(adjacent_vertex)
                discovered_set.add(adjacent_vertex)

                # Distance of adjacent_vertex is 1 more than current_vertex
                distances[adjacent_vertex] = distances[current_vertex] + 1
    return visited_list
```


PARTICIPATION ACTIVITY 11.10.1: Breadth-first search algorithm.

1) A vertex with a distance of 1 is

- adjacent to the start vertex
- the start vertex

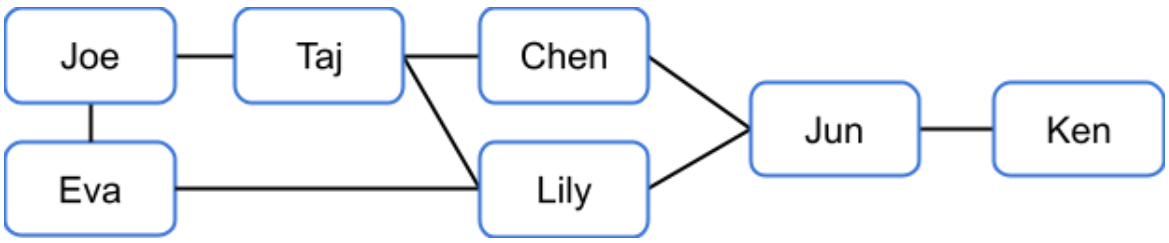
2) The equation to calculate a vertex's distance from the start vertex is

- `distances[current_vertex] = distances[start_vertex] + 1`
- `distances[adjacent_vertex] = distances[current_vertex] + 1`
- `distances[adjacent_vertex] = distances[start_vertex] + 1`

3) What method is used to add a vertex to frontier_queue?

- push()
- enqueue()
- append()

Figure 11.10.2: Social networking graph for breadth-first search example.



zyDE 11.10.1: Breadth-first search example.

The following program sets up a Graph object to represent the above social networking g and then runs a breadth-first search algorithm to find the "friendship" distance between a starting person and everyone else in the graph. Try changing the name of the starting per

Current file: main.py Load default templat

```
1 from Graph import Vertex, Graph
2 from Queue import Queue
3
4 # Breadth-first search function
```

```
def breadth_first_search(graph, start_vertex, distances=dict()):
    discovered_set = set()
    frontier_queue = Queue()
    visited_list = []

    # start_vertex has a distance of 0 from itself
    distances[start_vertex] = 0

    frontier_queue.enqueue(start_vertex) # Enqueue start_vertex in fr
    discovered_set.add(start_vertex)     # Add start_vertex to discov
```


Taj

Run

11.11 Python: Depth-first search

Depth-first search algorithm

Depth-first search traverses a graph by first visiting a specific starting vertex, and then visiting every vertex along each path originating from the starting vertex. Each path is traversed to the path's end before backtracking.

To perform depth-first search, the Graph and Vertex classes are used as well as a depth-first search function. depth_first_search() takes three arguments (the current graph, the starting vertex, and a function to mark visited vertices) and defines two container variables:

- vertex_stack is a list used as the stack of yet-to-be-visited vertices.
- visited_set is a set used to keep track of which vertices have been visited.

Initially, visited_set is empty and the starting vertex is pushed onto vertex_stack. So long as vertex_stack is not empty, the loop pops a vertex from vertex_stack and assigns current_vertex with the popped vertex. If current_vertex is not in visited_set, then current_vertex is visited, added to visited_set, and each adjacent vertex is pushed onto vertex_stack.

depth_first_search()'s visit_function parameter is a function that takes a single vertex as an argument. The function is called once for each vertex traversed during the depth-first search.

Figure 11.11.1: Depth-first search algorithm.

```
# Depth-first search function
def depth_first_search(graph, start_vertex, visit_function):
    vertex_stack = [start_vertex]
    visited_set = set()

    while len(vertex_stack) > 0:
        current_vertex = vertex_stack.pop()
        if current_vertex not in visited_set:
            visit_function(current_vertex)
            visited_set.add(current_vertex)
            for adjacent_vertex in
graph.adjacency_list[current_vertex]:
                vertex_stack.append(adjacent_vertex)
```

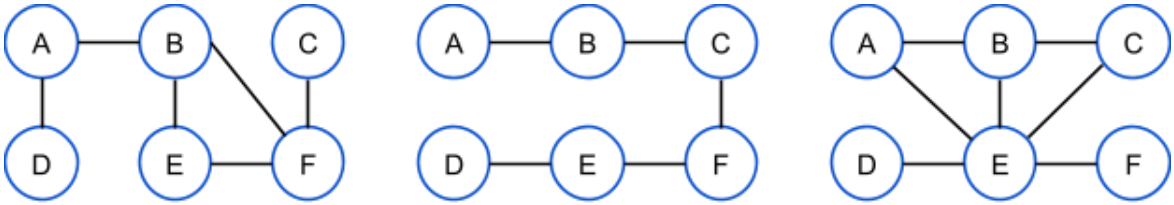

PARTICIPATION ACTIVITY 11.11.1: Depth-first search algorithm.

1) The loop in depth_first_search() operates as long as visited_set is not empty.
   - True
   - False

2) Without the use of visited_set to keep track of visited vertices, depth_first_search() would enter an infinite loop for any graph with a cycle.
   - True
   - False

3) If vertex_stack was initialized to an empty list, depth_first_search() would never visit any vertices.
   - True
   - False

Figure 11.11.2: Graphs for depth-first search example.



Graph 1 Graph 2 Graph 3

zyDE 11.11.1: Depth-first search example.

The following program creates the three graphs represented in the above figure, and then a depth-first search algorithm on each. The order of vertex traversal for each graph is displayed.

Before running the code, try to determine the traversal order for each graph. After running code, try to answer the following questions:

- Would traversal order change if current_vertex were visited after pushing current_vertex's adjacent vertices onto the stack?
- If a traversal ordering was missing one or more vertices, what must be true of the g
- How could a visitor function be used to make an ordered list of vertices visited?

Current file: main.py Load default templat

```
from graph import Vertex, Graph

# Depth-first search function
def depth_first_search(graph, start_vertex, visit_function):
    vertex_stack = [start_vertex]
    visited_set = set()

    while len(vertex_stack) > 0:
        current_vertex = vertex_stack.pop()
        if current_vertex not in visited_set:
            visit_function(current_vertex)
            visited_set.add(current_vertex)
            for adjacent_vertex in graph.adjacency_list[current_verte
                vertex_stack.append(adjacent_vertex)

```


Run

11.12 Python: Dijkstra's shortest path

Dijkstra's shortest path

Dijkstra's algorithm computes the shortest path from a given starting vertex to all other vertices in the graph.

To perform Dijjkstra's algorithm, the Graph and Vertex classes are used. The Vertex class is extended to include two additional data members:

- distance - The total sum of the edge weights on a path from some start vertex to the vertex.
- pred_vertex - A reference to the vertex that occurs immediately before the vertex, on a path from some start vertex to the vertex.

Figure 11.12.1: Dijkstra's shortest path algorithm.

```
def dijkstra_shortest_path(g, start_vertex):
    # Put all vertices in an unvisited queue.
    unvisited_queue = []
    for current_vertex in g.adjacency_list:
        unvisited_queue.append(current_vertex)

    # start_vertex has a distance of 0 from itself
    start_vertex.distance = 0

    # One vertex is removed with each iteration; repeat until the list is
    # empty.
    while len(unvisited_queue) > 0:

        # Visit vertex with minimum distance from start_vertex
        smallest_index = 0
        for i in range(1, len(unvisited_queue)):
            if unvisited_queue[i].distance <
unvisited_queue[smallest_index].distance:
                smallest_index = i
        current_vertex = unvisited_queue.pop(smallest_index)

        # Check potential path lengths from the current vertex to all
neighbors.
        for adj_vertex in g.adjacency_list[current_vertex]:
            edge_weight = g.edge_weights[(current_vertex, adj_vertex)]
            alternative_path_distance = current_vertex.distance +
edge_weight

            # If shorter path from start_vertex to adj_vertex is found,
            # update adj_vertex's distance and predecessor
            if alternative_path_distance < adj_vertex.distance:
                adj_vertex.distance = alternative_path_distance
                adj_vertex.pred_vertex = current_vertex
```

After calling the dijkstra_shortest_path() function, the shortest path from the starting vertex to any other given vertex can be built by following the pred_vertex *backwards* from the destination vertex to the starting vertex.

Figure 11.12.2: Getting the shortest path and distance between two vertices.


```
def get_shortest_path(start_vertex, end_vertex):
    # Start from end_vertex and build the path
backwards.
    path = ''
    current_vertex = end_vertex
    while current_vertex is not start_vertex:
        path = ' -> ' + str(current_vertex.label) +
path
        current_vertex = current_vertex.pred_vertex
    path = start_vertex.label + path
    return path
```

PARTICIPATION ACTIVITY 11.12.1: Dijkstra's shortest path algorithm.

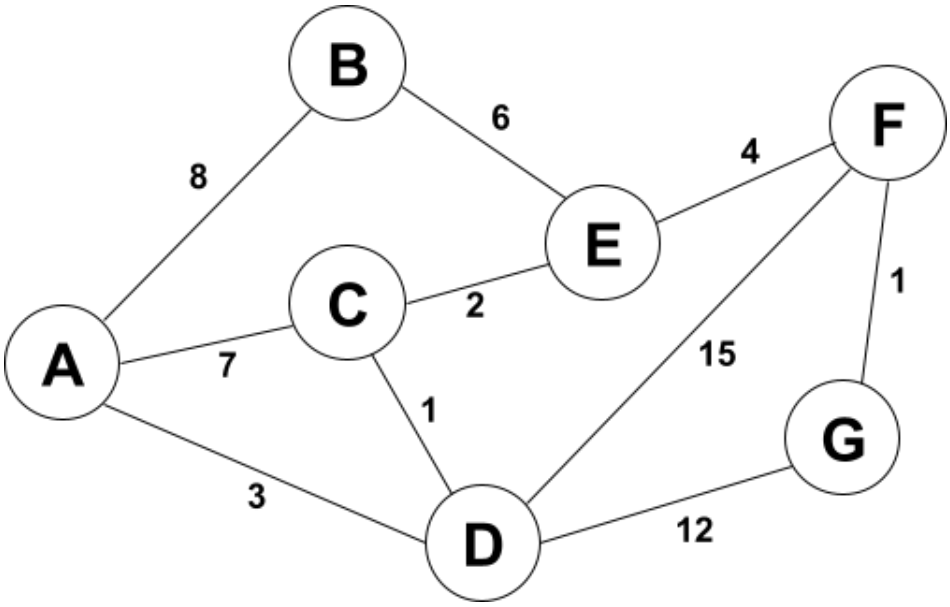
1) When initializing the list of unvisited vertices, all vertex distances (except the source vertex distance) are assigned with the value:
   - 0
   - -1
   - float('inf')

2) The shortest path from A to G is: A → D → B → G. What vertex is B's pred_vertex data member assigned with?
   - D
   - G
   - A

3) Can a vertex, V, ever have the distance data member assigned with infinity at the end of dijkstra's algorithm?

- No. All vertex distance values will be assigned with non-infinity values during the algorithm.
- Yes, if no path exists from the source vertex to V.
- Yes, if a negative edge weight exists in the graph.

Figure 11.12.3: Graph for Dijkstra's shortest path example.



zyDE 11.12.1: Dijkstra's shortest path example.

The following program sets up a Graph object to represent the above graph, and then run Dijkstra's shortest path algorithm to find the shortest path from vertex A to all other vertic

Try changing the weights of some of the edges to see the result in the shortest paths. In e case, can you anticipate which shortest paths will change?

- change (A, C) from 7 to 2
- change (F, G) from 1 to 10
- change (B, E) from 6 to 20

Load default templat

Current file: main.py

```
import operator
from graph import Graph, Vertex

def dijkstra_shortest_path(g, start_vertex):
    # Put all vertices in an unvisited queue.
    unvisited_queue = []
    for current_vertex in g.adjacency_list:
        unvisited_queue.append(current_vertex)

    # Start_vertex has a distance of 0 from itself
    start_vertex.distance = 0

    # One vertex is removed with each iteration; repeat until the lis
    # empty.
    while len(unvisited_queue) > 0:
```


Run

11.13 Algorithm: Dijkstra's shortest path

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

Animation content:

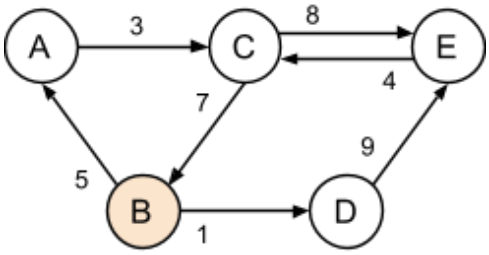
undefined

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 11.13.2: 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 11.13.3: Determining the shortest path from Dijkstra's algorithm.

Animation content:

undefined

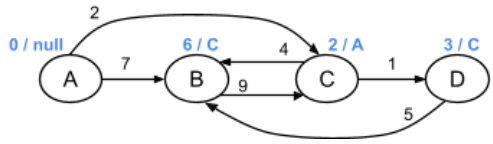
Animation captions:

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 11.13.4: Shortest path based on vertex predecessor.

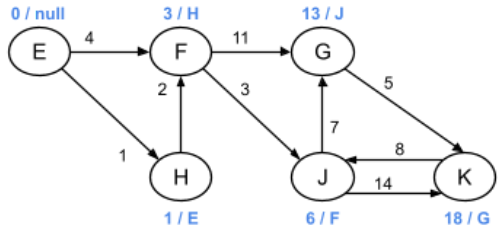
Type path as: A, B, C
If path does not exist, type: None

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



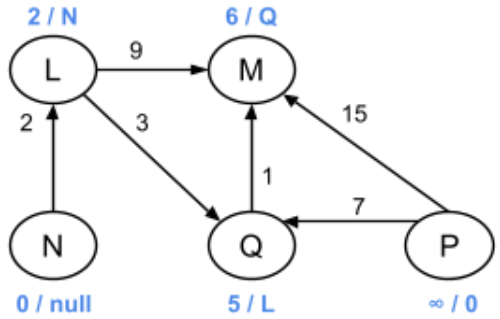
Check Show answer

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). 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 , the runtime is O(V*V + E) = O(V + E) = O(V). 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 11.13.5: Dijkstra's algorithm may not find the shortest path for a graph with negative edge weights.

Animation content:

undefined

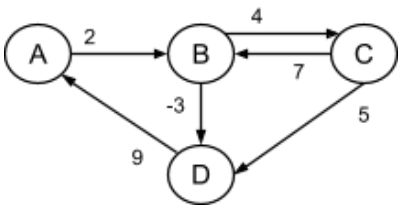
Animation captions:

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 11.13.6: Dijkstra's shortest path algorithm: Supported graph types.

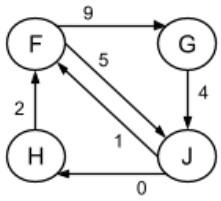
Indicate if Dijkstra's algorithm will find the shortest path for the following graphs.

1)



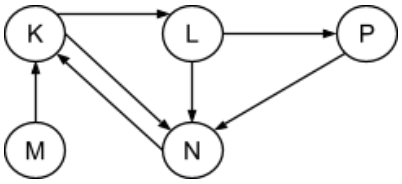
- Yes
- No

2)



- Yes
- No

3)

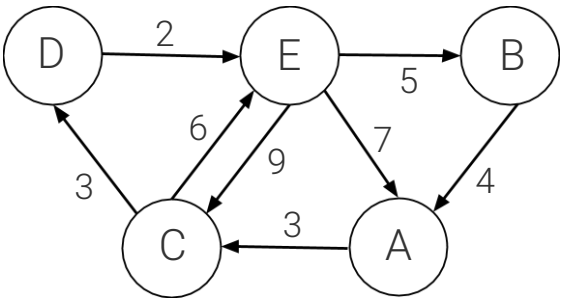


- Yes
- No

CHALLENGE ACTIVITY 11.13.1: Dijkstra's shortest path.

502696.2096894.qx3zqy7

Start



Starting at vertex A, what is the shortest path length to each vertex?

A: Ex: 1
B:
C:
D:
E:

1 2 3

Check Next

11.14 Algorithm: Bellman-Ford's shortest path

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 | 11.14.1: The Bellman-Ford algorithm finds the shortest path from a source vertex to all other vertices in a graph.

Animation content:

undefined

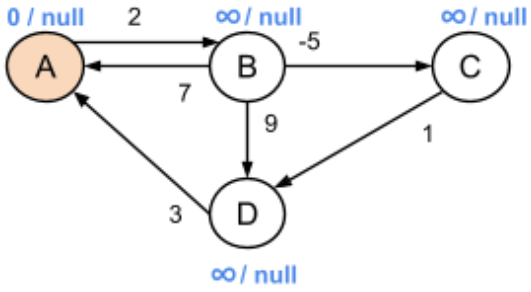
Animation captions:

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 vertices.
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 11.14.2: 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

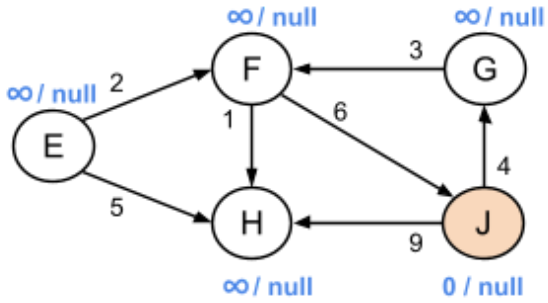
2) What are C's values after the first iteration?
   - -5 / B
   - -3 / B
   - ∞ / null

3) What are D's values after the first iteration?
   - 3 / A
   - -2 / C
   - ∞ / null

PARTICIPATION ACTIVITY

11.14.3: 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.



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 11.14.4: Bellman-Ford: Checking for negative edge weight cycles.

Animation content:

undefined

Animation captions:

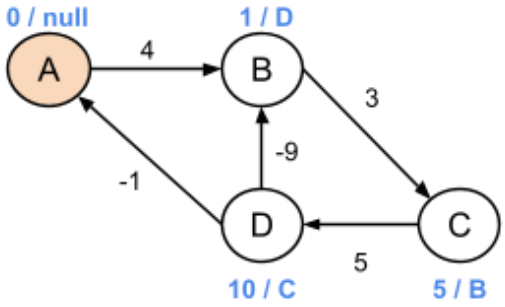
1. After visiting all vertices V-1 times, the Bellman-Ford algorithm checks for negative edge weight cycles.
2. For each vertex in the graph, adjacent vertices are checked for a shorter path. No such path exists through A to B.
3. But, a shorter path is still found through B to C, so a negative edge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

Figure 11.14.1: 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
}
```

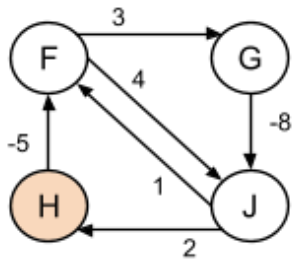

PARTICIPATION ACTIVITY 11.14.5: Bellman-Ford algorithm: Checking for negative edge weight cycles.

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

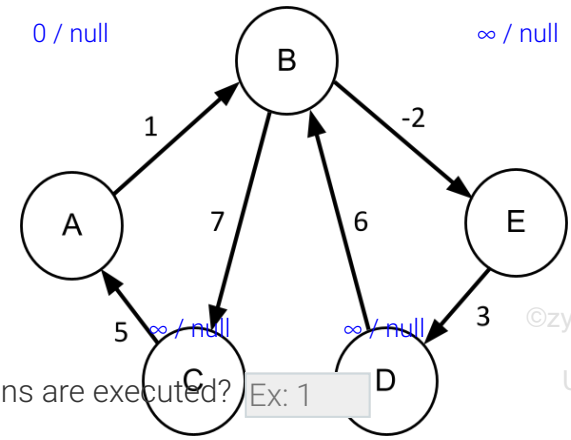
CHALLENGE ACTIVITY 11.14.1: Bellman-Ford's shortest path.

502696.2096894.qx3zqy7

Start

The Bellman-Ford algorithm is run on the following graph. The start vertex is A. Assume each l of the algorithm visits vertices in the graph in the following order: A, B, C, D, E.

∞ / null



How many main loop iterations are executed? Ex: 1

What are B's values after the first iteration? Ex: 0 or inf / Enter inf for ∞.

What are C's values after the first iteration? /

What are D's values after the first iteration? /

What are E's values after the first iteration? /

1 2 3 4

Check Next

11.15 Python: Bellman-Ford's shortest path

Bellman-Ford's shortest path algorithm

The Bellman-Ford algorithm computes the shortest path from a given starting vertex to all other vertices in the graph. The algorithm works by first finding the minimum distances for all vertices that are reachable from the start vertex in one step, then in two steps, then in three steps, etc. until minimum distances for all reachable vertices have been calculated. Note that if the graph has |V| vertices, then all minimum distances must be found after at most |V|-1 iterations of the main loop.

In the main loop, each edge is examined to see if a shorter distance to the start vertex than was previously known is found. After |V|-1 iterations are complete, a final iteration is performed to see if any distance decreases further; if a distance decreases again, then a negative edge weight cycle must exist in the graph and the algorithm fails.

Each Vertex object is assigned two extra data members to help with the Bellman-Ford algorithm:

- distance - The minimum distance found so far from the start vertex. The data member is assigned with an initial value of `float('inf')`, which is the Python representation of .

- pred_vertex - The previous vertex in the shortest path from the start vertex. The data member is assigned with an initial value of None, indicating that at the start of the algorithm, no known path exists from the start vertex.

Figure 11.15.1: The bellman_ford() function.


```
def bellman_ford(graph, start_vertex):
    # Initialize all vertex distances to infinity and
    # and predecessor vertices to None.
    for current_vertex in graph.adjacency_list:
      current_vertex.distance = float('inf') # Infinity
      current_vertex.pred_vertex = None

    # start_vertex has a distance of 0 from itself
    start_vertex.distance = 0

    # Main loop is executed |V|-1 times to guarantee minimum distances.
    for i in range(len(graph.adjacency_list)-1):
        # The main loop.
        for current_vertex in graph.adjacency_list:
            for adj_vertex in graph.adjacency_list[current_vertex]:
                edge_weight = graph.edge_weights[(current_vertex,
adj_vertex)]
                alternative_path_distance = current_vertex.distance +
edge_weight

                # If shorter path from start_vertex to adj_vertex is found,
                # update adj_vertex's distance and predecessor
                if alternative_path_distance < adj_vertex.distance:
                   adj_vertex.distance = alternative_path_distance
                   adj_vertex.pred_vertex = current_vertex

    # Check for a negative edge weight cycle
    for current_vertex in graph.adjacency_list:
        for adj_vertex in graph.adjacency_list[current_vertex]:
             edge_weight = graph.edge_weights[(current_vertex, adj_vertex)]
             alternative_path_distance = current_vertex.distance +
edge_weight

             # If shorter path from start_vertex to adj_vertex is still
found,
             # a negative edge weight cycle exists
             if alternative_path_distance < adj_vertex.distance:
                return False

    return True
```


PARTICIPATION ACTIVITY 11.15.1: Bellman-Ford's shortest path algorithm.

1) The bellman_ford() function returns a list of Vertex objects that represent the shortest path from the start vertex to the end vertex.

   - True
   - False

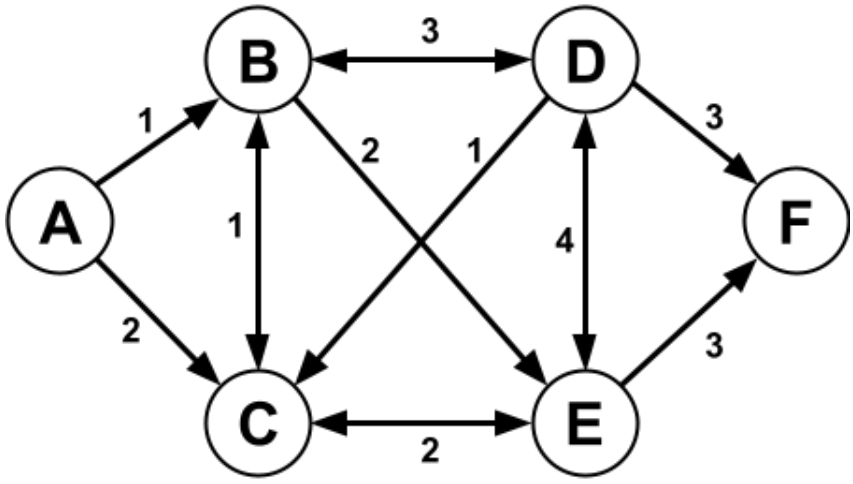
2) The algorithm can return True even if a negative edge weight exists in the graph.

   - True
   - False

3) The algorithm returns False if any negative edges are in an undirected graph.

   - True
   - False

Figure 11.15.2: Graph for testing bellman_ford() function.



zyDE 11.15.1: Using the bellman_ford() function.

The following program runs the Bellman-Ford shortest path algorithm on the above graph with vertex A as the starting point.

Run the Bellman-Ford algorithm by hand, and use the program to verify the results. Then t the following:

- Change the weight of the (B, E) edge to -2. How does this affect the algorithm's resu
- Change the weight of the (B, E) edge to -4. How does this affect the algorithm's resu

Current file: main.py Load default template...

Run

```
from graph import Vertex, Graph

def bellman_ford(graph, start_verte
    # Initialize all vertex distanc
    # and predecessor vertices to N
    for current_vertex in graph.adj
        current_vertex.distance = flo
        current_vertex.pred_vertex =

    # start_vertex has a distance o
    start_vertex.distance = 0

    # Main loop is executed |V|-1 t
    for i in range(len(graph.adjace
        # The main loop.
```


11.16 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 11.16.1: 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 11.16.2: Topological sort.

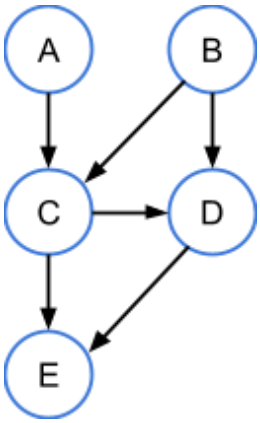
1) In the example above, D, A, B, F, E, C is a valid topological sort.
   - True
   - False

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 11.16.3: 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 11.16.4: 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 11.16.5: 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 11.16.6: 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.

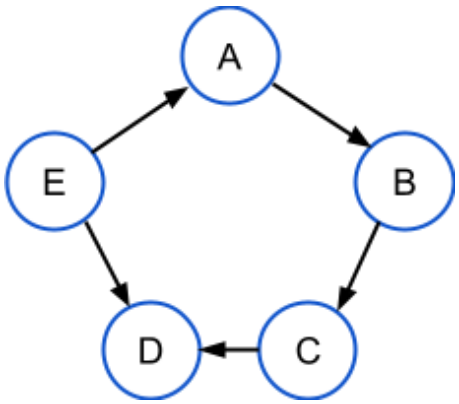
3. Edge EF goes to vertex F, which still has 2 incoming edges. Edge EG goes to vertex G, which still has 1 incoming edge.
4. Vertex A is removed and added to resultList. Outgoing edges from A are removed from remainingEdges. Vertices B and C are added to noIncoming.
5. Vertices C and D are processed, 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 11.16.1: GraphGetIncomingEdgeCount function.

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

PARTICIPATION ACTIVITY 11.16.7: 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 from E to D
- Edge from A to B
- No edges

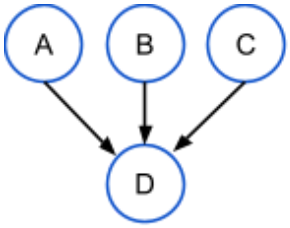
3) When currentV becomes vertex C, what is the contents of resultList?

- A, B
- E, A, B
- E, D

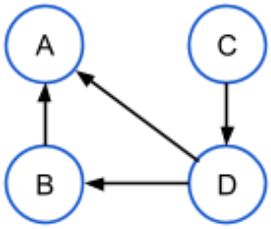
4) What is the final contents of resultList?

- A, B, C, D, E
- E, A, B, C, D
- E, A, B, D, C

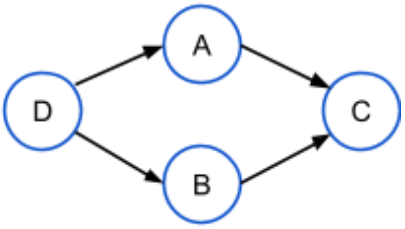
PARTICIPATION ACTIVITY 11.16.8: Topological sort matching.



1



2



3

If unable to drag and drop, refresh the page.

Graph 3 Graph 1 Graph 2

	D, B, A, C
	C, D, B, A

B, C, A, D

Reset

PARTICIPATION ACTIVITY 11.16.9: 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 . 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 .

11.17 Python: Topological sort

Topological sort algorithm

The topological sort algorithm finds an ordered list of vertices from a directed graph where no vertex has an outgoing edge to a preceding vertex in the list. Topological sorting is useful in many real-world situations such as manufacturing (where some components must be built before other components) or course scheduling (where some courses must be completed as prerequisites to other courses). When the graph is drawn with the vertices lined up left to right in the order of a topological sort, then all edges will point in the direction from left to right.

Figure 11.17.1: An acyclic directed graph.

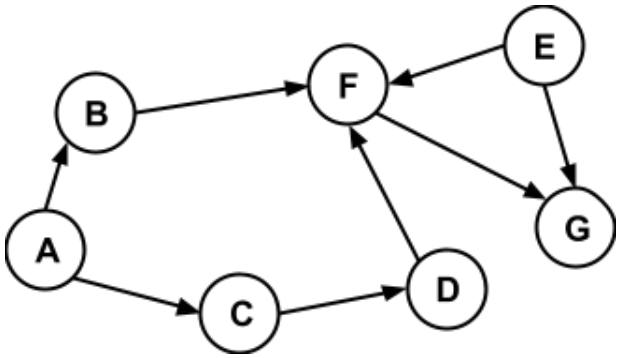
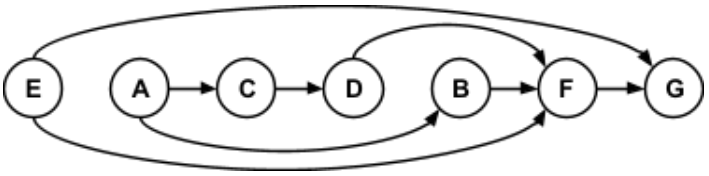


Figure 11.17.2: The same graph with a left-to-right topological ordering.



PARTICIPATION ACTIVITY 11.17.1: Topological Sort.

1) No topological ordering exists if a graph contains a cycle.
   - ○ True
   - ○ False

2) Undirected graphs can't be used in the topological sort algorithm.

- True
- False

3) There can only be one unique topological sort for any graph.

- True
- False

4) The vertices' labels can be sorted alphabetically to produce a topological sort.

- True
- False

Iterating through a collection of edges

The topological sorting algorithm iterates through lists of edges. In Python, the Graph class represents edges using a 2-tuple in the form: (from_vertex, to_vertex), and a for loop can be used to iterate through a list or set of edges. Ex. Edges are stored in a data member dictionary called edge_weights, so all of the edges in a graph can be iterated through using a loop like:

```
for (from_vertex, to_vertex) in g.edge_weights.keys():
```

The above code defines two variables that can be used inside the loop body: from_vertex (the start of the edge) and to_vertex (the end of the edge). Any collection of edges (ex: a list or set) can be used, not just the edge_weights dictionary in a graph.

The get_incoming_edge_count() function uses a similar for loop. The parameter edge_list can be assigned with any type of edge collection. The topological_sort() function uses get_incoming_edge_count() as part of the algorithm.

Figure 11.17.3: The get_incoming_edge_count() function.


```
def get_incoming_edge_count(edge_list,
vertex):
   count = 0
   for (from_vertex, to_vertex) in
edge_list:
      if to_vertex is vertex:
         count = count + 1
   return count
```

The for loop iterates through each edge in edge_list, assigning from_vertex and to_vertex variables with the edges' start and end vertices. The loop checks to see if the to_vertex from each edge matches the argument vertex, incrementing a counter each time a match is found.

The topological_sort() function

topological_sort() starts by creating a list of all vertices that have no incoming edges. A for loop is used to check each vertex in the graph, calling get_incoming_edge_count() for each vertex using the entire set of edges in the graph.

Figure 11.17.4: Initializing the no_incoming list at the beginning of topological_sort().

```
# make list of vertices with no incoming edges.
no_incoming = []
for vertex in graph.adjacency_list.keys():
    if get_incoming_edge_count(graph.edge_weights.keys(), vertex)
== 0:
        no_incoming.append(vertex)
```

topological_sort()'s main loop starts by selecting a vertex from the no_incoming list. Since any vertex from the list can be used, the most efficient choice is the last vertex, which can be removed using the pop() method. The main method repeats, selecting one vertex from no_incoming at a time, until the no_incoming list is empty.

Each selected vertex is added to the final topological sort result, then all outgoing edges from the vertex are removed from future consideration. A set object called edges_remaining is initialized before the main loop with all edges from the graph; a set is used so edges can be removed in time.

Finally, after all of the vertex's outgoing edges have been removed from the edges_remaining set, the no_incoming list is updated to include any new vertices that now have no incoming edges.

Figure 11.17.5: The main loop of topological_sort().

```
# remaining_edges starts with all edges in the graph.
# A set is used for its efficient remove() method.
remaining_edges = set(graph.edge_weights.keys())
while len(no_incoming) != 0:
    # select the next vertex for the final result.
    current_vertex = no_incoming.pop()
    result_list.append(current_vertex)

    # remove current_vertex's outgoing edges from remaining_edges.
    outgoing_edges = []
    for to_vertex in graph.adjacency_list[current_vertex]:
        outgoing_edge = (current_vertex, to_vertex)
        if outgoing_edge in remaining_edges:
            outgoing_edges.append(outgoing_edge)
            remaining_edges.remove(outgoing_edge)

    # see if removing the outgoing edges creates any new vertices
    # with no incoming edges.
    for (from_vertex, to_vertex) in outgoing_edges:
        in_count = get_incoming_edge_count(remaining_edges,
to_vertex)
        if in_count == 0:
            no_incoming.append(to_vertex)
```


The full topological_sort() function is shown below.

Figure 11.17.6: The topological_sort() function.

```
def topological_sort(graph):
    result_list = []

    # make list of vertices with no incoming edges.
    no_incoming = []
    for vertex in graph.adjacency_list.keys():
        if get_incoming_edge_count(graph.edge_weights.keys(), vertex)
== 0:
            no_incoming.append(vertex)

    # remaining_edges starts with all edges in the graph.
    # A set is used for its efficient remove() method.
    remaining_edges = set(graph.edge_weights.keys())
    while len(no_incoming) != 0:
        # select the next vertex for the final result.
        current_vertex = no_incoming.pop()
        result_list.append(current_vertex)
        outgoing_edges = []

        # remove current_vertex's outgoing edges from remaining_edges.
        for to_vertex in graph.adjacency_list[current_vertex]:
            outgoing_edge = (current_vertex, to_vertex)
            if outgoing_edge in remaining_edges:
                outgoing_edges.append(outgoing_edge)
                remaining_edges.remove(outgoing_edge)

        # see if removing the outgoing edges creates any new vertices
        # with no incoming edges.
        for (from_vertex, to_vertex) in outgoing_edges:
            in_count = get_incoming_edge_count(remaining_edges,
to_vertex)
            if in_count == 0:
                no_incoming.append(to_vertex)

    return result_list
```


PARTICIPATION ACTIVITY 11.17.2: The topological_sort() function.

1) What is the result of calling topological_sort() with a graph that contains a cycle?

- ○ The function returns False.
- ○ A runtime error occurs.
- ○ The function returns a list that does not contain all of the vertices in the graph.
- ○ The function gets trapped in an infinite loop and never returns.

2) Why is remaining_edges assigned with a set instead of a list?

- Sets contain a fast ( ) remove() method.
- Sets are easier to iterate through than lists.
- The edges in the graph are already stored in a set object.

3) What is the purpose of the for loop at the bottom of the main while loop?

- The loop checks for cycles.
- The loop checks to see if any of the vertices have no incoming edges, after the selected vertex's outgoing edges are removed.
- The loop cleans up the no_incoming list by removing any vertices that have an incoming edge.

zyDE 11.17.1: Using the topological_sort() function.

The following program creates the graph shown earlier in this section and prints the resul the topological_sort() function. Verify that the program outputs a valid topological orderin the graph.

Experiment with the algorithm by adding and removing some edges. See if you can predic results before trying the program. (Make each modification starting from the original grap not from the graph resulting from the previous step.)

- Add an edge from E to A.
- Add an edge from C to G.
- Remove the edge from A to B.
- Add an edge from F to C.

Current file: main.py Load default template...

Run

```
1
2 from graph import Graph, Vertex
3
4 def get_incoming_edge_count(edge_li
5     count = 0
```

```
    for (from_vertex, to_vertex) in
      if to_vertex is vertex:
        count = count + 1
    return count

def topological_sort(graph):
    result_list = []

    # make list of vertices with no
    no_incoming = []
```


11.18 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 11.18.1: 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 11.18.2: 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
   - False

4) A minimum spanning tree can only be built for an undirected graph.
   - 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 11.18.3: Minimum spanning tree algorithm.

Animation content:

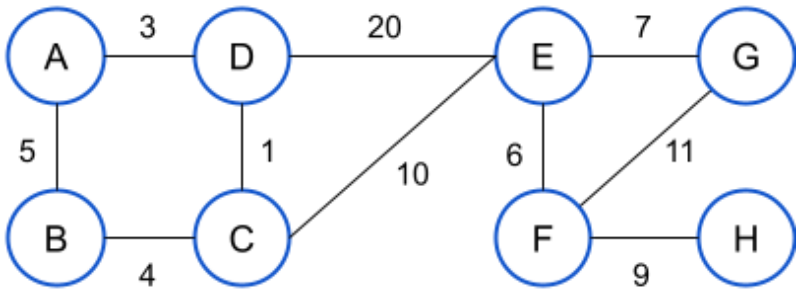
undefined

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.

PARTICIPATION ACTIVITY 11.18.4: 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
- 7
- 9
- 10

PARTICIPATION ACTIVITY 11.18.5: 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

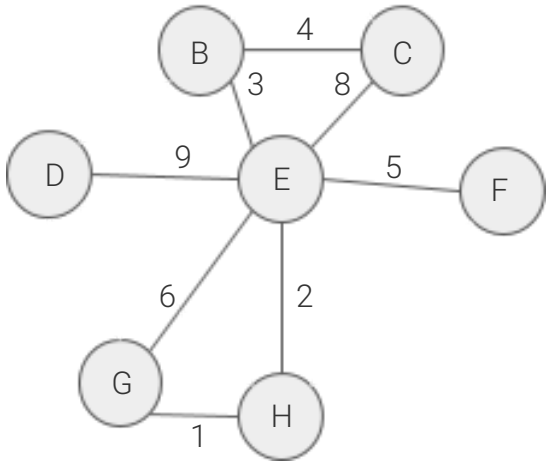
Kruskal's minimum spanning tree algorithm's use of the edge list, collection of vertex sets, and resulting edge list results in a space complexity of . If the edge list is sorted at the beginning, then the minimum edge can be removed in constant time within the loop. Combined with a mechanism to map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

CHALLENGE ACTIVITY 11.18.1: Minimum spanning tree.

502696.2096894.qx3zqy7

Start

Kruskal's minimum spanning tree algorithm is executed on the following graph.



What is the first edge that will be added to the result? Ex: YZ

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

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

What is the minimum spanning tree edge weight sum? Ex: 1

1 2

Check Next

11.19 Python: Minimum spanning tree

EdgeWeight class

The minimum spanning tree algorithm requires a priority queue for edges, where each edge's weight is the edge's priority. The EdgeWeight class implements a simple edge object that stores to and from vertices, as well as weight. Comparison operators are implemented such that edges are compared to each other using edge weights.

Figure 11.19.1: EdgeWeight class.

```
class EdgeWeight:
    def __init__(self, from_vertex, to_vertex, weight):
        self.from_vertex = from_vertex
        self.to_vertex = to_vertex
        self.weight = weight

    # Only edge weights are used in the comparisons that
follow

    def __eq__(self, other):
        return self.weight == other.weight

    def __ge__(self, other):
        return self.weight >= other.weight

    def __gt__(self, other):
        return self.weight > other.weight

    def __le__(self, other):
        return self.weight <= other.weight

    def __lt__(self, other):
        return self.weight < other.weight

    def __ne__(self, other):
        return self.weight != other.weight
```


PARTICIPATION ACTIVITY 11.19.1: EdgeWeight class.

1) Two EdgeWeight objects with equal weights are equal, even if the to_vertex or from_vertex members are not equal.

   - True
   - False

2) Syntax for constructing an EdgeWeight object is:
   `EdgeWeight(vertex1, vertex2, weight)`
   where vertex1 and vertex2 are graph vertices, and weight is a number.

   - True
   - False

3) Assuming vert1 and vert2 are graph vertices, the expression below evaluates to _____.

```
EdgeWeight(vert1, vert2, 12.0)
>= EdgeWeight(vert2, vert1,
8.0)
```

- True
- False

VertexSetCollection class

The minimum spanning tree algorithm uses a collection of vertex sets to keep track of what portions of the graph are connected by edges chosen so far. The VertexSetCollection class is implemented to store a collection of sets in a dictionary and simplify the minimum spanning tree algorithm. Each graph vertex exists in only one set. Collectively, all sets in the collection store all vertices from the graph.

The vertex_map data member is a dictionary that maps a vertex to the set containing the vertex. The get_set() method takes a vertex as an argument and returns the set containing the vertex.

The merge() method takes two different sets as arguments and merges the two together internally. Ex: Suppose 5 vertices, A, B, C, D, and E, are represented in 3 sets in a collection: {A, E}, {B, C}, and {D}. If merge() is called on sets {A, E} and {D}, then the resulting collection contains two sets: {A, E, D} and {B, C}.

Figure 11.19.2: VertexSetCollection class.

```
# Stores a collection of vertex sets, which collectively store all vertices
in a
# graph. Each vertex is in only one set in the collection.
class VertexSetCollection:
    def __init__(self, all_vertices):
        self.vertex_map = dict()
        for vertex in all_vertices:
            vertex_set = set()
            vertex_set.add(vertex)
            self.vertex_map[vertex] = vertex_set

    def __len__(self):
        return len(self.vertex_map)

    # Gets the set containing the specified vertex
    def get_set(self, vertex):
        return self.vertex_map[vertex]

    # Merges two vertex sets into one
    def merge(self, vertex_set1, vertex_set2):
        # First create the union
        merged = vertex_set1.union(vertex_set2)
        # Now remap all vertices in the merged set
        for vertex in merged:
            self.vertex_map[vertex] = merged
```


PARTICIPATION ACTIVITY 11.19.2: VertexSetCollection class.

1) The type of the vertex_map member is a list.
   - True
   - False

2) If a VertexSetCollection is initialized with all vertices from a graph, then the union of all sets in vertex_map is a set with all of the graph's vertices.
   - True
   - False

3) The number of key-value pairs in vertex_map decreases by one after calling merge().
   - True
   - False

Minimum spanning tree algorithm

Kruskal's minimum spanning tree algorithm is implemented in the minimum_spanning_tree() function. A graph is passed as the only argument, and a list of edges representing the graph's minimum spanning tree is returned.

First, edge_list is initialized as a priority queue containing all edges from the graph. Python's heapq module provides a heap-based priority queue implementation, but elements must support comparison. So each object in the queue is an EdgeWeight object.

Next, vertex_sets is initialized as a new VertexSetCollection and result_list is initialized as an empty list. Then the while loop runs as long as vertex_sets has more than one set and edge_list has at least one edge.

During each loop iteration, the edge with the lowest weight is removed from edge_list. set1 is assigned to the set containing the edge's from_vertex and set2 is assigned to the set containing the edge's to_vertex. If the two sets are not the same, the edge is added to result_list and set1 and set2 are merged within vertex_sets.

Figure 11.19.3: minimum_spanning_tree function.

```
# Returns a list of edges representing the graph's minimum spanning
tree.
# Uses Kruskal's minimum spanning tree algorithm.
def minimum_spanning_tree(graph):
    # edge_list starts as a list of all edges from the graph
    edge_list = []
    for edge in graph.edge_weights:
        edge_weight = EdgeWeight(edge[0], edge[1],
graph.edge_weights[edge])
        edge_list.append(edge_weight)
    # Turn edge_list into a priority queue (min heap)
    heapq.heapify(edge_list)

    # Initialize the collection of vertex sets
    vertex_sets = VertexSetCollection(graph.adjacency_list)

    # result_list is initially an empty list
    result_list = []

    while len(vertex_sets) > 1 and len(edge_list) > 0:
        # Remove edge with minimum weight from edge_list
        next_edge = heapq.heappop(edge_list)

        # set1 = set in vertex_sets containing next_edge's 'from' vertex
        set1 = vertex_sets.get_set(next_edge.from_vertex)
        # set2 = set in vertex_sets containing next_edge's 'to' vertex
        set2 = vertex_sets.get_set(next_edge.to_vertex)

        # If the 2 sets are distinct, then merge
        if set1 is not set2:
            # Add next_edge to result_list
            result_list.append(next_edge)
            # Merge the two sets within the collection
            vertex_sets.merge(set1, set2)

    return result_list
```


PARTICIPATION ACTIVITY 11.19.3: minimum_spanning_tree function.

1) Which variable serves as a priority queue for edges?

- edge_list
- vertex_sets
- result_list

2) Two sets are merged during a loop iteration _____.

- only if set1 and set2 refer to the same object
- only if set1 and set2 refer to different objects
- in all cases

3) result_list will be sorted by ascending edge weights.

- True
- False

zyDE 11.19.1: Minimum spanning tree example.

The following program creates two graphs, then computes and displays the minimum spanning tree of both. Try altering the graph declarations to compute minimum spanning of other graphs.

Current file: main.py Load default templat

```
from graph import Vertex, Graph
from EdgeWeight import EdgeWeight
from VertexSetCollection import VertexSetCollection
import heapq

# Returns a list of edges representing the graph's minimum spanning t
# Uses Kruskal's minimum spanning tree algorithm.
def minimum_spanning_tree(graph):
    # edge_list starts as a list of all edges from the graph
    edge_list = []
    for edge in graph.edge_weights:
        edge_weight = EdgeWeight(edge[0], edge[1], graph.edge_weights
        edge_list.append(edge_weight)
    # Turn edge_list into a priority queue (min heap)
    heapq.heapify(edge_list)
```


Run

11.20 All pairs shortest path

Overview and shortest paths matrix

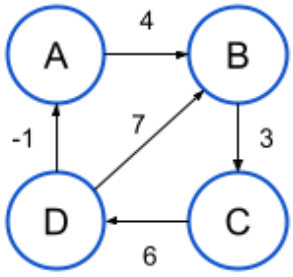
An ***all pairs shortest path*** algorithm determines the shortest path between all possible pairs of vertices in a graph. For a graph with vertices V, a |V| x |V| matrix represents the shortest path lengths between all vertex pairs in the graph. Each row corresponds to a start vertex, and each column 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 11.20.1: 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 11.20.2: 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 11.20.3: 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 11.20.4: Floyd-Warshall algorithm.

Animation content:

undefined

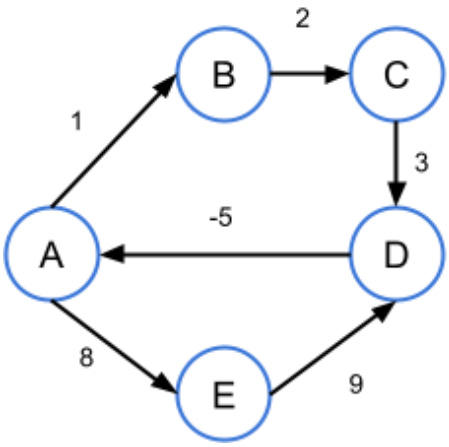
Animation captions:

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 11.20.5: 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 11.20.6: 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 11.20.1: 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 11.20.7: 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 exist, 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 . The matrix is constructed with a runtime complexity of .

11.21 Python: All pairs shortest path

Additional Graph class methods

The Graph class introduced in another section can be extended to assist with implementation of the all pairs shortest path algorithm. Three methods are added:

Table 11.21.1: Additional Graph class method descriptions.

Method name	Description
get_vertex()	Takes a vertex label as an argument and returns the vertex in the graph with the specified label, or None if no such vertex exists.
get_vertex_list()	Returns a list of all vertices in the graph. The all pairs shortest path algorithm commonly refers to vertices by index, so a linear list of all the graph's vertices is more convenient than the graph's adjacency_list dictionary.
get_incoming_edges()	Takes a vertex as an argument and returns a list of all edges from the graph that are incoming to that vertex.

Figure 11.21.1: Graph class with get_vertex(), get_vertex_list(), and get_incoming_edges() methods.

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}
        self.edge_weights = {}

    def add_vertex(self, new_vertex):
        self.adjacency_list[new_vertex] = []

    def add_directed_edge(self, from_vertex, to_vertex, weight =
1.0):
        self.edge_weights[(from_vertex, to_vertex)] = weight
        self.adjacency_list[from_vertex].append(to_vertex)

    def add_undirected_edge(self, vertex_a, vertex_b, weight = 1.0):
        self.add_directed_edge(vertex_a, vertex_b, weight)
        self.add_directed_edge(vertex_b, vertex_a, weight)

    # Returns the vertex in this graph with the specified label, or
None
    # if no such vertex exists.
    def get_vertex(self, vertex_label):
        for vertex in self.adjacency_list:
            if vertex.label == vertex_label:
                return vertex
        return None

    # Returns a list of all vertices in this graph
    def get_vertex_list(self):
        return list(self.adjacency_list)

    # Returns a list of all edges incoming to the specified vertex
    # Each edge is a tuple of the form (from_vertex, to_vertex)
    def get_incoming_edges(self, vertex):
        incoming = []
        for edge in self.edge_weights:
            if edge[1] is vertex:
                incoming.append(edge)
        return incoming
```


PARTICIPATION ACTIVITY 11.21.1: Graph class.

1) If multiple vertices in the graph had the same label, get_vertex() may return a list of vertices.

   - ○ True
   - ○ False

2) get_vertex_list() returns a list built from the collection of keys in the adjacency_list dictionary.

- True
- False

3) get_incoming_edges() iterates through all vertices in the graph.

- True
- False

Floyd-Warshall all pairs shortest path 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. The all_pairs_shortest_path() function returns a list of lists, representing the |V| x |V| matrix of shortest path lengths. The list's length equals the number of vertices in the graph. The length of each list in the list also equals the number of vertices in the graph.

Figure 11.21.2: all_pairs_shortest_path() function.

```
# Implementation of Floyd-Warshall all-pairs shortest path
def all_pairs_shortest_path(graph):
    vertices = graph.get_vertex_list()
    num_vertices = len(vertices)

    # Initialize dist_matrix to a num_vertices x num_vertices matrix
    # with all values set to infinity
    dist_matrix = []
    for i in range(0, num_vertices):
        dist_matrix.append([float("inf")] * num_vertices)

    # Set each distance for vertex to same vertex to 0
    for i in range(0, num_vertices):
        dist_matrix[i][i] = 0

    # Finish matrix initialization
    for edge in graph.edge_weights:
        dist_matrix[vertices.index(edge[0])][vertices.index(edge[1])] =
graph.edge_weights[edge]

    # Loop through vertices
    for k in range(0, num_vertices):
        for toIndex in range(0, num_vertices):
            for fromIndex in range(0, num_vertices):
                currentLength = dist_matrix[fromIndex][toIndex]
                possibleLength = dist_matrix[fromIndex][k] + dist_matrix[k]
[toIndex]
                if possibleLength < currentLength:
                    dist_matrix[fromIndex][toIndex] = possibleLength

    return dist_matrix
```


PARTICIPATION ACTIVITY 11.21.2: all_pairs_shortest_path() function.

1) Which expression represents the data type that all_pairs_shortest_path() might return for a graph with two vertices?

- [0, 7, 0, 4]
- [[0, 7], [0, 4]]
- [[0], [7], [0], [4]]

2) Each entry would be _____ in a matrix built for a graph with no edges, but two or more vertices.

- zero
- infinity
- either zero or infinity

3) Suppose graph.get_vertex_list() returns the list [VertexC, VertexA, VertexB] and the following code is run:

```
matrix =
all_pairs_shortest_path(graph)
```

Which expression is used to lookup the shortest path length from vertex C to vertex A?

- matrix[0][1]
- matrix[1][0]
- matrix[2][0]

Path reconstruction

The matrix returned from all_pairs_shortest_path() can be used to determine the list of edges in the shortest path between two vertices. The reconstruct_path() function takes 4 arguments:

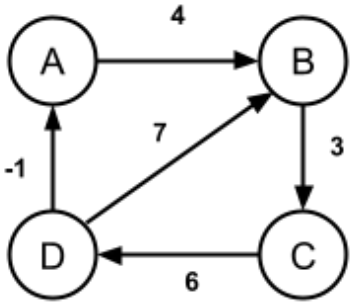
- graph: The graph that the all pairs shortest path matrix was constructed for.
- start_vertex: Vertex at the start of the path.
- end_vertex: Vertex at the end of the path.
- dist_matrix: The distances matrix built by all_pairs_shortest_path().

A list of edges comprising the shortest path is returned.

Figure 11.21.3: reconstruct_path() function.

```
def reconstruct_path(graph, start_vertex, end_vertex, dist_matrix):
    vertices = graph.get_vertex_list()
    start_index = vertices.index(start_vertex)
    path = []

    # Backtrack from the ending vertex
    current_index = vertices.index(end_vertex)
    while current_index != start_index:
        incoming_edges = graph.get_incoming_edges(vertices[current_index])

        found_next = False
        for current_edge in incoming_edges:
            expected = dist_matrix[start_index][current_index] -
graph.edge_weights[current_edge]
            actual = dist_matrix[start_index]
[vertices.index(current_edge[0])]
            if expected == actual:
                # Update current vertex index
                current_index = vertices.index(current_edge[0])

                # Prepend current_edge to path
                path = [current_edge] + path

                # The next vertex in the path was found
                found_next = True

                # The correct incoming edge was found, so break the inner
loop
                break

        if found_next == False:
            return None # no path exists

    return path
```


PARTICIPATION ACTIVITY 11.21.3: Path reconstruction.



Suppose graph.get_vertex_list() returns [Vertex A, VertexB, VertexC, VertexD] and dist_matrix is

```
[
  [  0, 4, 7, 13 ],
  [  8, 0, 3, 9  ],
  [  5, 9, 0, 6  ],
```

```
[ -1, 3, 6, 0  ]
```

]. The graph is shown above. Select the order of events when start_vertex is vertex D and end_vertex is vertex B.

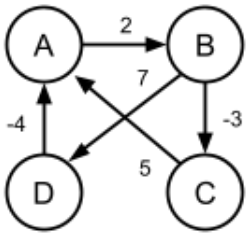
If unable to drag and drop, refresh the page.

Second First never

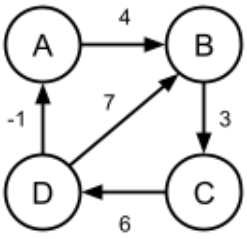
	The edge from D to B is prepended to the path.
	The edge from D to A is prepended to the path
	The edge from A to B is prepended to the path.

Reset

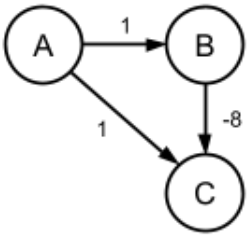
Figure 11.21.4: Graphs for testing all_pairs_shortest_path() and reconstruct_path() functions.



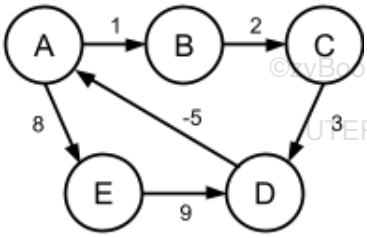
Graph 1



Graph 2



Graph 3



Graph 4

zyDE 11.21.1: All pairs shortest path example.

The following program creates graphs for the four graphs above, then computes and disp the all pairs shortest path matrix for each. A sample shortest path is also displayed for ea graph.

After running the code, try altering the graph declarations that start on line 70. Each declaration contains vertex and edge lists, followed by two vertex labels that specify verti to construct a shortest path for. Try altering the two vertices in the existing declarations t show different shortest path sequences. Try adding a new graph to the list as well.

Current file: main.py Load default templat

```
1  from graph import Vertex, Graph
2  from display_matrix import display_matrix
3
4  # Implementation of Floyd-Warshall all-pairs shortest path
5  def all_pairs_shortest_path(graph):
6      vertices = graph.get_vertex_list()
7      num_vertices = len(vertices)
8
9      # Initialize dist_matrix to a num_vertices x num_vertices matrix
10     # with all values set to infinity
11     dist_matrix = []
12     for i in range(0, num_vertices):
13         dist_matrix.append([float("inf")] * num_vertices)
14
15     # Set each distance for vertex to same vertex to 0
```

Run

11.22 LAB: Graph representations

Step 1: Inspect Vertex.py, Edge.py, and DirectedGraph.py

Inspect the Vertex class declaration in the Vertex.py file. Access Vertex.py by clicking on the orange arrow next to main.py at the top of the coding window. The Vertex class represents a graph vertex and

has a string for the vertex label.

Inspect the Edge class declaration in the Edge.py file. The edge class represents a directed graph edge and has references to a from-vertex and a to-vertex.

Inspect the DirectedGraph abstract class declaration in the DirectedGraph.py file. DirectedGraph is an abstract base class for a directed, unweighted graph.

Step 2: Inspect AdjacencyListGraph.py and AdjacencyListVertex.py

The AdjacencyListGraph class inherits from DirectedGraph and is declared in AdjacencyListGraph.py. The `vertices` attribute is a list of AdjacencyListVertex references. The list contains all the graph's vertices.

The AdjacencyListVertex class inherits from Vertex and is declared in the read only AdjacencyListVertex.py file. The `adjacent` attribute is a list of references to adjacent vertices.

Step 3: Inspect AdjacencyMatrixGraph.py

The AdjacencyMatrixGraph class inherits from DirectedGraph and is declared in AdjacencyMatrixGraph.py. The `vertices` attribute is a list of Vertex references. The list contains all the graph's vertices. The `matrix_rows` attribute is a list of matrix rows. Each row itself is a list of bool values. If matrix_rows[X][Y] is true, then an edge exists from vertices[X] to vertices[Y].

Indices in `vertices` correspond to indices in `matrix_rows`. So if vertex "C" exists at index 2 in `vertices`, then row 2 and column 2 in the matrix correspond to vertex "C".

Step 4: Implement the AdjacencyListGraph class

Implement the required methods in AdjacencyListGraph. Each method has a comment indicating the required functionality. The `vertices` list must be used to store the graph's vertices and must not be removed. New methods can be added, if needed, but existing method signatures must not change.

Step 5: Implement the AdjacencyMatrixGraph class

Implement the required methods in AdjacencyMatrixGraph. Each method has a comment indicating the required functionality. The `vertices` and `matrix_rows` lists must be used to store the graph's vertices and adjacency matrix, respectively. Both `vertices` and `matrix_rows` lists must not be removed. New methods can be added, if needed, but existing method signatures must not change.

Step 6: Test in develop mode, then submit

File main.py contains test cases for each graph operation. The test operations are first run on an AdjacencyListGraph. Then the same test operations are run on an AdjacencyMatrixGraph. Running code in develop mode displays the test results.

After each method is implemented and all tests pass in develop mode, submit the code. The unit tests run on submitted code are similar, but use different graphs and perform direct verification of the graphs internal attributes.

LAB ACTIVITY | 11.22.1: LAB: Graph representations | 2 / 10

Downloadable files

`main.py` , `Vertex.py` , `Edge.py` , `DirectedGraph.py` , `AdjacencyListVertex.py` , `AdjacencyListGraph.py` , `AdjacencyMatrixGraph.py` , and `GraphCommands.py`

Download

File is marked as read only Current file: **main.py**

```
1  import sys
2
3  from AdjacencyListGraph import AdjacencyListGraph
4  from AdjacencyMatrixGraph import AdjacencyMatrixGraph
5
6  from GraphCommands import *
7
8  def main():
9      graph1 =  AdjacencyListGraph()
10     graph2 =  AdjacencyMatrixGraph()
11
12
13     # Test AdjacencyListGraph first
14     print("AdjacencyListGraph: ")
15     adj_pass = test_graph(sys.stdout, graph1)
```


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) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

```
11/11 S-2 U-0------0-1--1,2,2 min:36
```

©zyBooks 11/20/23 11:09 1048447
Eric Quezada
UTEPCS2302ValeraFall2023