# SWEN 601
# *Software Construction*

*Depth-First Search, & Dijkstra's Algorithm*

# Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
   a. *The project **may** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

***Do not*** submit code that ***does not compile***. Comment it out if necessary.

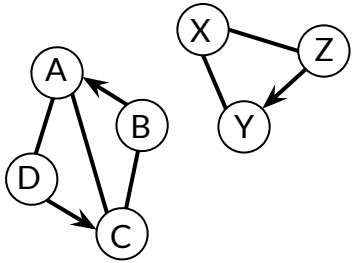
# Next Two Weeks

| WEEK 11 | SUN | MON | TUES | WEDS | THURS | FRI | SAT |
|---|---|---|---|---|---|---|---|
| **QUIZ** | | | Quiz #16 | | Quiz #17 | | |
| **LECTURE** | | | Graphs & BFS | | DFS & Dijkstra's Algorithm | | |
| **HOMEWORK** | Hwk 16 Due (**11:30PM**) | | *Hwk 17 Assigned* | | *Hwk 18 Assigned* | Hwk 17 Due (**11:30pm**) | |

| WEEK 12 | SUN | MON | TUES | WEDS | THURS | FRI | SAT |
|---|---|---|---|---|---|---|---|
| **QUIZ** | | | Quiz #18 | | | | |
| **LECTURE** | | | Study Day | | Practicum 3 | | |
| **HOMEWORK** | Hwk 18 Due (**11:30pm**) | | *Hwk 19 Assigned* | | | | |

# Recap: Graphs

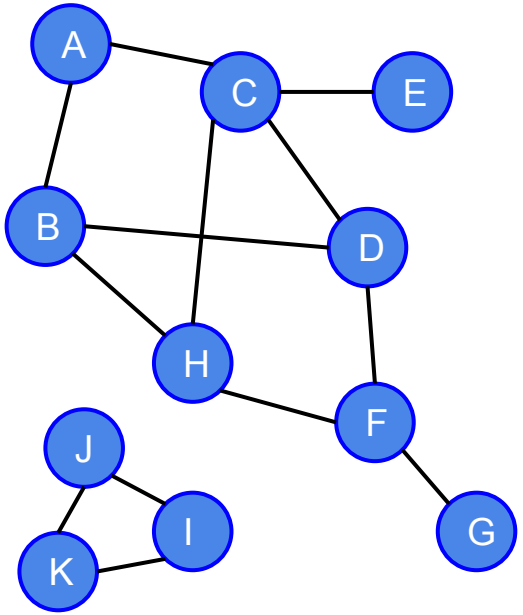
- A ***graph*** comprises vertices.
  - Each ***vertex*** holds a ***value*** of some generic type.
- The connection between two vertices is an ***edge***.
  - An edge may be ***directed*** or ***undirected***.
  - We refer to two vertices that are connected by an edge as ***neighbors***.
- The series of edges that connect two vertices together is a ***path***.
- An algorithm for finding a path between two vertices in a graph is a ***search***.
- A ***connected component*** is a set of vertices such that, given any two, there is a path between them.
  - Two vertices that are not connected to each other by one or more edges are in separate connected components.




| Vertex | Adjacency List |
|---|---|
| A | C, D |
| B | A, C |
| C | A, B |
| D | A, C |
| X | Y, Z |
| Y | X |
| Z | X, Y |

# Breadth-First Search (BFS)

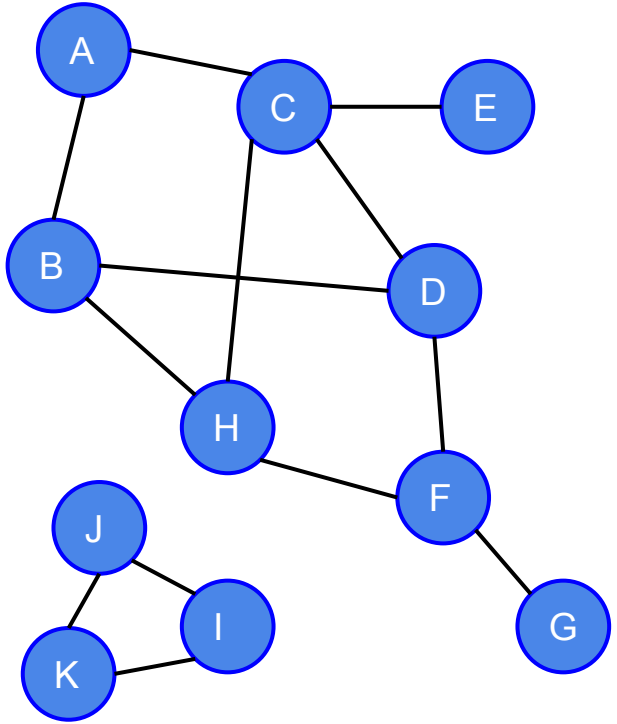
- An algorithm that attempts to find a path between two vertices in a graph is called a ***search***.
- A ***breadth-first search*** (***BFS***) is one such algorithm.
- A BFS examines all of the ***immediate*** neighbors of a vertex first before going any ***deeper*** into the graph.
  - It searches all of the neighbors that are ***one*** edge from the ***starting vertex*** first.
  - Then the neighbors that are ***two*** edges.
  - And so on until it finds the end or runs out of neighbors.
- A BFS uses a ***queue*** to insure that the vertices are visited in breadth-first order.
  - It uses a ***set*** or a ***hashmap*** to keep track of visited vertices.




Starting from vertex A, the BFS search order in this graph would be ***A, B, C, D, H, E, F, G***.

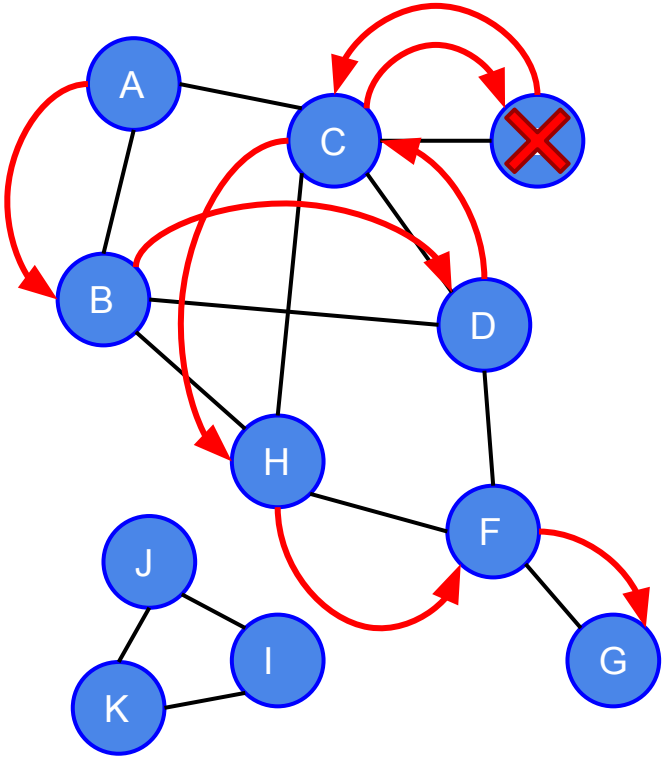
# Depth-First Search (DFS)

- A ***depth-first search*** (***DFS***) is an alternative to BFS.
- Rather than explore all of the neighbors of a node first, a DFS follows edges along a path ***deeper*** into the graph.
- If it arrives at the ***goal***, it ***stops***.
- If it arrives at a vertex that is not the goal and does not have any ***unexplored*** neighbors, it returns back along its path ***only as far as necessary*** to find a vertex with unexplored neighbors.
- A DFS can be implemented using a ***stack*** or by using ***recursion***.
  - Each alternative implementation will find a ***different*** path!



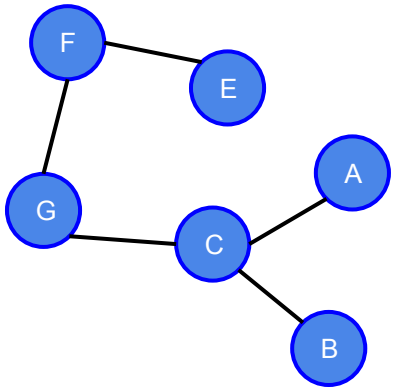

Let's practice a DFS on this example graph to find a path from A to G.

# Illustrated DFS Example

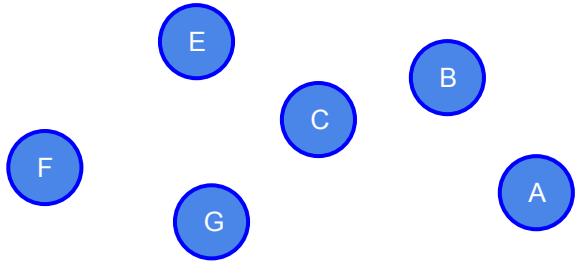


# The DFS Algorithm

- Like BFS, a Depth-First Search can be used to determine whether or not a path ***exists*** between two nodes, ***S*** and ***E***.
- We will need a simple helper function to ***visit*** a vertex.
  - It will take a ***vertex*** and a ***set*** as parameters.
  - For each of the vertex's ***neighbors*** that is not already in the set:
    - ***Add*** it to the set.
    - ***Visit*** the neighbor (make a recursive call).
- Our main DFS function will then simply:
  - ***Create*** the empty set.
  - Add ***S*** to the set.
  - Call the helper function with ***S*** and the ***set***.
  - When the function returns, if ***E*** is in the set, then a path from ***S*** to ***E exists***.

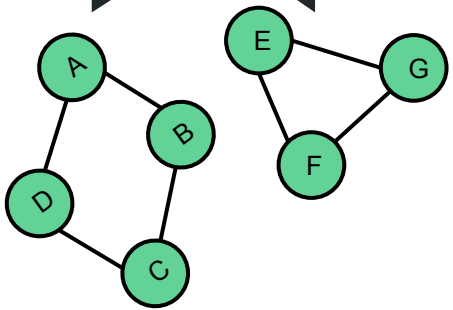



Set of Previously Seen Vertices




Using the above graph, perform a DFS from F to B. Always visit neighbors in alphabetical order.

# Activity: Implement DFS




Remember, from the perspective of the user, there is no such thing as a Vertex.

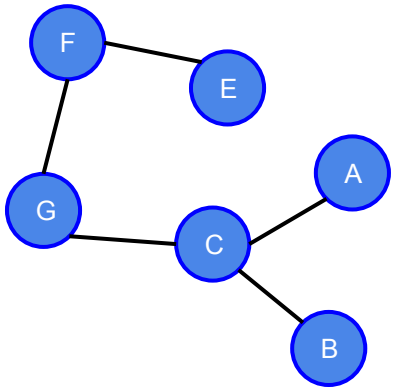
And so the user specifies the ***values*** at the `start` and `end` of the path.

Now, we'll implement the DFS algorithm.

1. First, you will need to copy your `Graph` interface and `AdjacencyGraph` class from the previous session.
2. Modify your Graph interface to include a new method:
   a. `public boolean depthFirstSearch(T start, T end)`
3. In your `AdjacencyGraph` class:
   a. Stub out the `depthFirstSearch` method.
   b. Add the helper function that recursively visits every vertex that is reachable from `start`.
      i. `private void visitDFS(Vertex<T> vertex, Set<Vertex<T>> visited)`
   c. Finally, implement the `depthFirstSearch` method.

# DFS Path Building

- Like our initial BFS implementation, the current version of DFS doesn't determine what the path ***actually is***.
  - Also like BFS, we will need to ***modify*** it to do so.
- The DFS Path algorithm takes ***S***, ***E***, and a ***set*** of ***visited*** nodes as parameters:
  - If ***S == E***, return a path containing only ***S***.
  - Otherwise, for each ***neighbor N*** of ***S*** that is not in the ***set***:
    - ***Add*** it to the set.
    - ***Recursively*** call DFS Path with ***N***, ***E***, and the ***set***.
    - If a ***path*** is returned, add ***S*** to the front and ***return*** the path.
  - If ***none*** of the neighbors returns a path, return `null` (to indicate that there is no path).




Set of Previously Seen Vertices

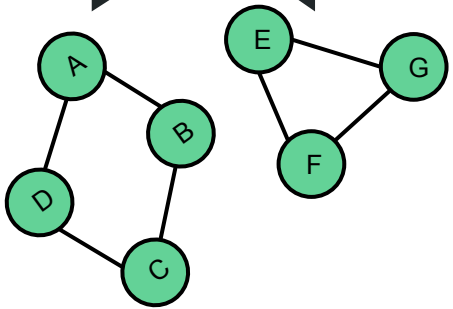
B C A G F E

| V | B | C | A | G | F | E |
|---|---|---|---|---|---|---|
| P | - | B | C | C | G | F |

Path from B to E:

B C G F E

# Activity: Implement DFS Path




Once again you will need a `private` helper method into which you can pass a set.

Call this helper from your public search method.

Now, we'll implement the DFS path algorithm.

1. Modify your Graph interface to include a new method:
   a. `public List<T> depthFirstPath(T start, T end)`
2. In your `AdjacencyGraph` class, implement a `depthFirstPath` helper method:
   a. If `S == E`, return a `LinkedList` containing S.
   b. Otherwise for each neighbor N of S:
      i. Add N to the set of `visited` nodes.
      ii. Recursively call the helper method with N and E.
      iii. If a path is returned (non-`null`), add S.
   c. If no path is returned, return `null`.

# BFS vs. DFS

| | Breadth-First Search (BFS) | Depth-First Search (DFS) |
|---|---|---|
| Order of Visits | NEIGHBORS FIRST | NEIGHBORS-OF-NEIGHBORS FIRST |
| Data Structures Used | QUEUE, MAP | STACK (RECURSION) |
| Guarantees Shortest Path | YES (FEWEST EDGES) | NO |
| Time Complexity | O(V + E) | O(V + E) |

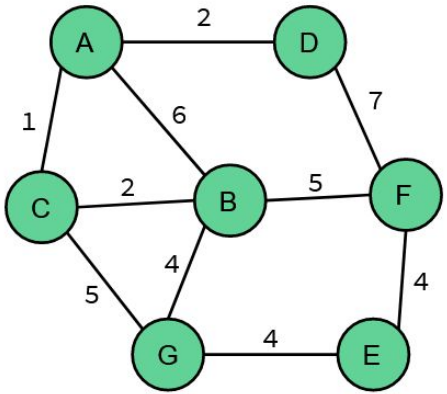
# What Does “Shortest” Mean?

- DFS is ***never*** guaranteed to find the shortest path.
  - It is good for finding a path that connects all of the vertices in one connected component.
- BFS finds the shortest path only in terms of the ***number of edges*** along the path.
- But ***neither*** algorithm considers the ***weight*** of the edges in a graph, and therefore neither will necessarily find the path with the lowest ***cost***.


Keeping in mind that the path with the lowest total cost may include more edges, which algorithms *can* find the shortest path?

Let’s take a look at a few example algorithms for finding paths with a low cost (but not necessarily *the lowest* cost).

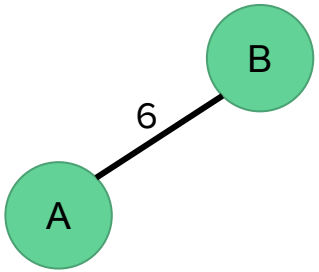
# Activity: A Weighted Graph 1




Our current graph implementation classes do not support weighted edges. That will need to change. Begin by creating a new package: `session.weighted`.

1. Create a new ***generic*** interface called `WeightedGraph` with the following methods:
   a. `void addValue(T value)`
   b. `void connect(T value, T neighbor, int weight)`
   c. `List<T> dijkstrasShortestPath()`
2. Create a new class called `WeightedAdjacencyGraph` that implements `WeightedGraph`.
   a. Stub out the methods for now.
3. ***Copy*** the `Vertex` class into the package (***copy***, do not *move*).

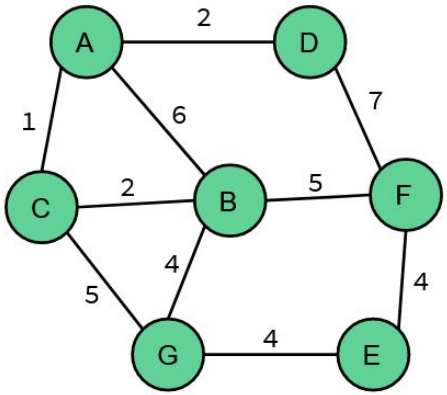
# Activity: A Weighted Edge




Next, we will need an Edge class. It should be include the following:

1. It should be ***generic***, e.g. `Edge<T>`
2. A `from` `Vertex<T>` field.
3. A `to` `Vertex<T>` field.
4. A `weight` field.
5. A constructor to set all three fields.
6. Getters for all three fields.

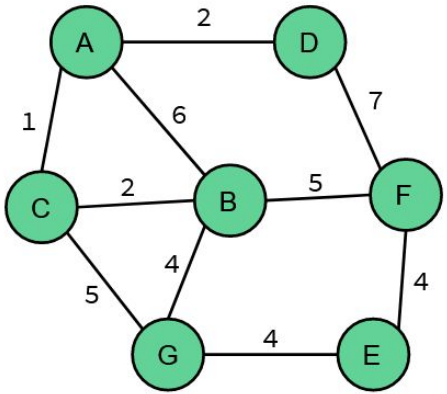
# Activity: A Weighted Vertex




Next, we will need to modify the `Vertex` class that works with edges. Make sure that you are changing the `Vertex` in your `session.weighted` package!

1. Change the collection of `neighbors` to hold edges rather than vertices.
2. Change the `addNeighbor()` method:
   a. Add a `weight` parameter.
   b. Create an `Edge` and add it to the collection.
3. Change the `getNeighbors()` method to copy all of the from vertices from each edge into a collection and return it.
4. Add a `getEdges()` method that returns the edges.

# Activity: A Weighted Graph II




Let's take a look at how Dijkstra's Algorithm works in theory.

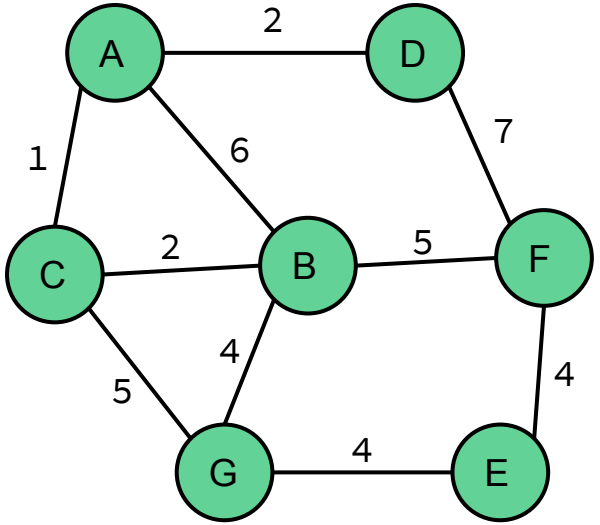
You will then implement it in your weighted graph.

Switch back to your `WeightedAdjacencyGraph` class.

1. Add a map of values (`T`) to vertices (`Vertex<T>`) to the class.
2. Implement a parameterless constructor to create the empty map.
3. Implement the `add` method.
4. Implement the `connect` method to create ***undirected*** connections.
5. You will implement `dijkstrasShortestPath` for your homework.

# Lowest Cost Edges

- One alternative is a ***greedy algorithm*** algorithm that chooses the ***lowest cost edges***.
  - All of the edges are examined, and the edge with the lowest cost is placed into a ***set***.
  - The process is repeated until some combination of edges forms a complete path from from ***start*** to ***end***.
- Is this guaranteed to find the shortest path?




SET OF EDGES

$E_{AC}(1)$ $E_{CB}(2)$ $E_{FE}(4)$

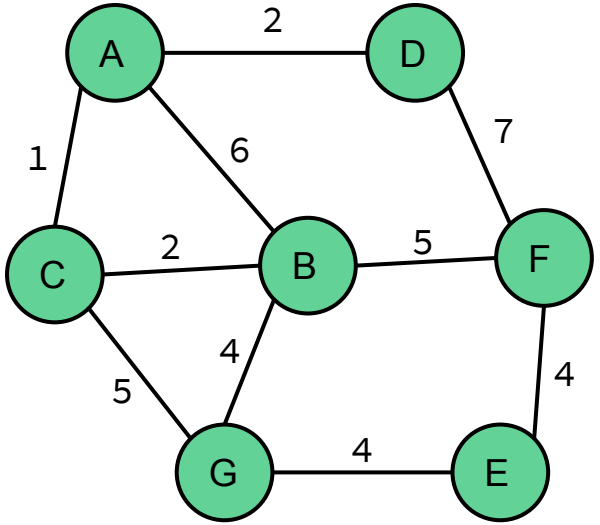
$E_{AD}(2)$ $E_{BG}(4)$ $E_{GE}(4)$

Path from A to E:

$E_{AC}(1)$ $E_{CB}(2)$ $E_{BG}(4)$ $E_{GE}(4)$

# Nearest Neighbor

- Another alternative is yet another ***greedy algorithm*** that always chooses the ***nearest neighbor***.
  - The ***nearest*** neighbor is the one connected by the edge with the ***lowest cost***.
  - ***Repeat*** until arriving at the ***destination***.
- This is one solution to the Traveling Salesman Problem.
- Is it guaranteed to find the shortest path?



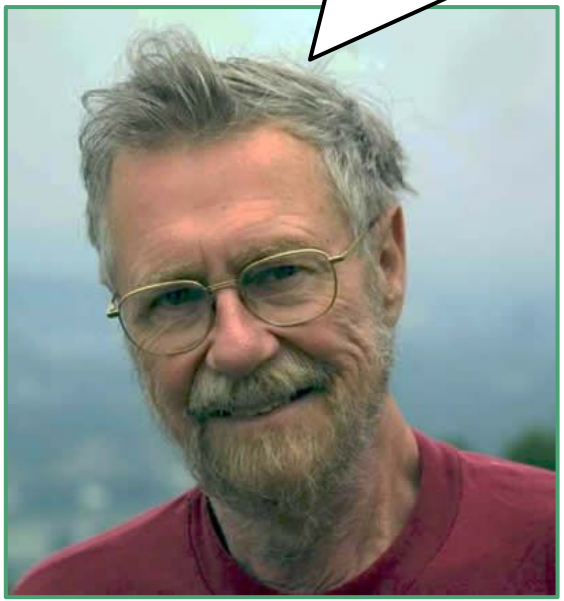

Path from A to E:

$E_{AC}(1)$ $E_{CB}(2)$ $E_{BG}(4)$ $E_{GE}(4)$

# Dijkstra's Algorithm

- Edsger Dijkstra observed the following:
  - A path ***P*** between two vertices ***S*** and ***E*** comprises a sequence of edges that may be numbered ***0-to-N***.
  - If *P* is ***optimal***, then it must also be true that the path ***P'***, that comprises the edges numbered ***0-to-N-1*** and ends at some vertex ***E'***, must also be optimal.
    - That is to say that there cannot be a shorter path from ***S*** to ***E'*** or else the path ***P*** would not be optimal.
  - And if ***P'*** is optimal, then the path ***P''***, that comprises the edges ***0-to-N-2*** that ends are some vertex ***E''***, must also be optimal.
  - And so on.
- But what does this give us?
  - Essentially, by keeping track of the ***optimal paths*** from ***S*** to every other vertex, we can rule out ***sub-optimal*** paths and only explore those that are potentially optimal.

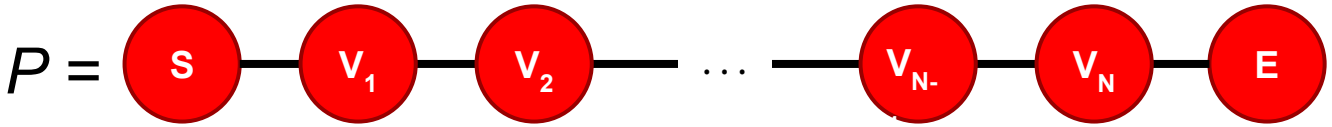
If debugging is the process of removing bugs, then programming must be the process of putting them in.



His last name is pronounced "Dike-stra."

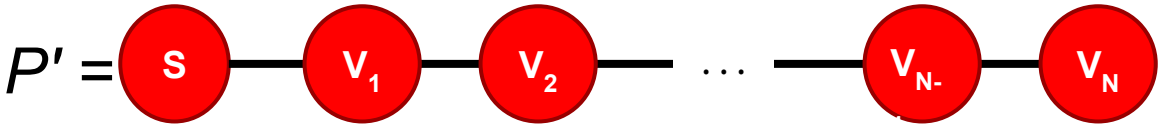
# Dijkstra's Algorithm in a Nutshell

Given two vertices, *S* and *E*, there exists an optimal path *P* that has the lowest cost.




We will label the vertices along this path $V_1$ to $V_N$.

If *P* is optimal, then the path from *S* to $V_N$, which we will call *P'*, must also be optimal.



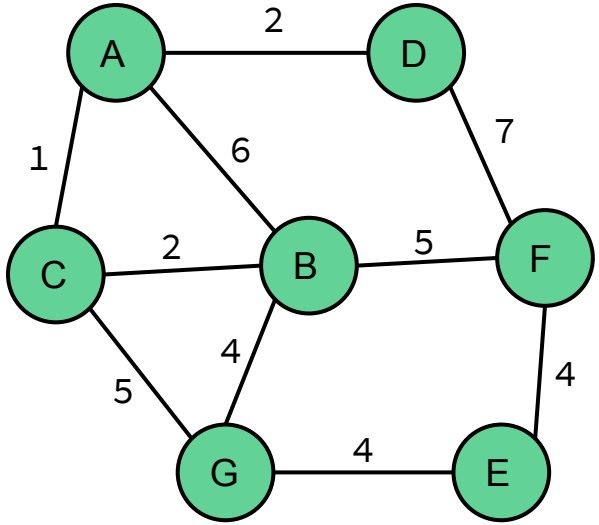

In other words, the path *P* cannot be optimal if there is a path to $V_N$ that is shorter than *P'*.

It must also be true that the path from *S* to $V_{N-1}$, and the path from *S* to $V_{N-2}$, and so on, including the path from *S* to $V_1$ are also optimal for the same reason.

Therefore, we can find the optimal path from S to E by finding the optimal subpaths to each vertex in the graph.

# Dijkstra Example

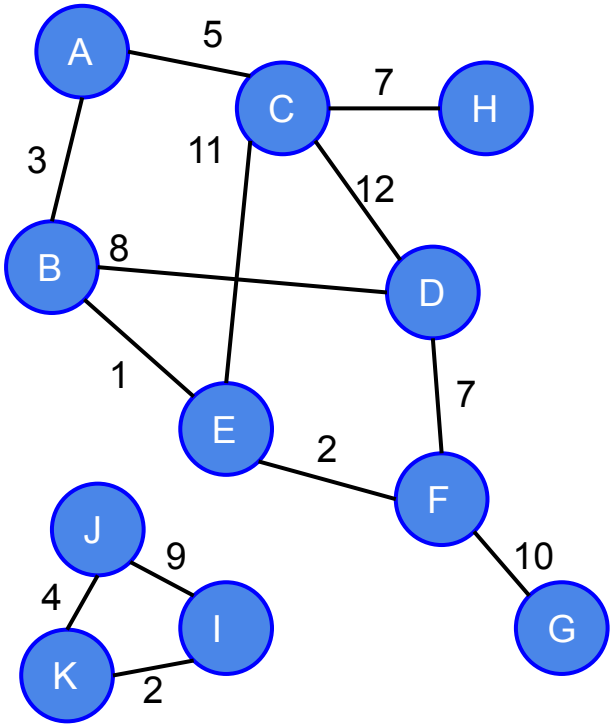
- Given an ***optimal*** path ***P***, all of its ***sub-paths*** must also be optimal.
- For example, given the graph to the right, the optimal path $P_{AE}$ comprises edges:
  - $E_{AC} - E_{CG} - E_{GE} = 10$
- If $P_{AE}$ truly is optimal, then it must also be true that the path $P_{AG}$ is also optimal.
  - $E_{AC} - E_{CG} = 6$
- And $P_{AC}$ is also optimal.
  - $E_{AC} = 1$




In other words, if we keep track of the shortest path from A to every other vertex as we explore the graph...

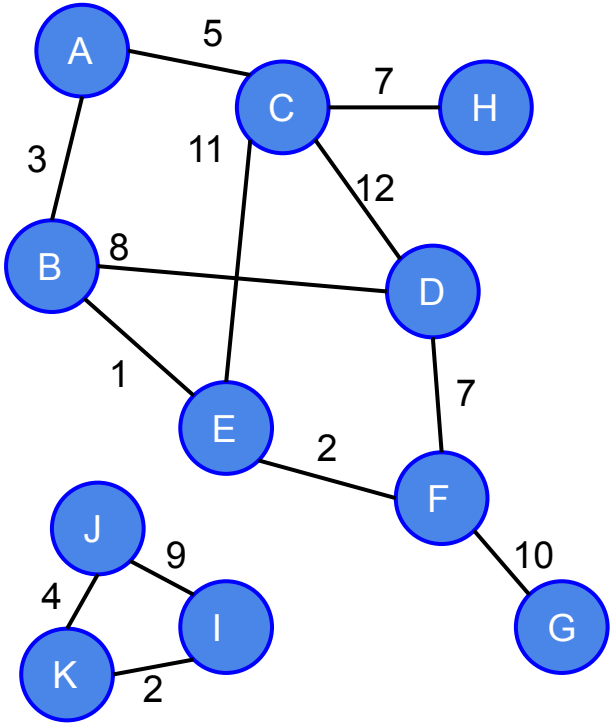
… then we will be able to reverse engineer the shortest path from A to *any* other vertex in the graph.

# An Example




- The BFS algorithm identified the path that traversed the smallest number of nodes as A→B→D→F→G, with a weight of 28.
- But this is not the shortest path if we are concerned about ***lowest cost***! The path A→B→E→F→G has a weight of 16.
- In the next segment, we will see how Dijkstra's Algorithm finds it...

# EXAMPLE GRAPH



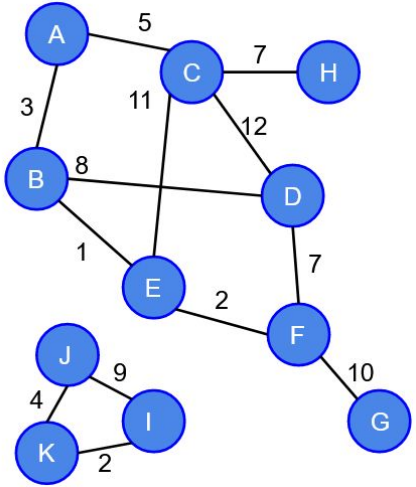
# A Path Tuple

- In implementing BFS, we used a map to keep track of each vertex and its predecessor.
- That is no longer sufficient; we need to know not only the predecessor, but the ***total distance*** from the starting vertex ***through*** that predecessor.
- This means that we will need a class to store both pieces of information, i.e. a ***path tuple***.
  - Again we will use a ***map***.
  - The starting vertex has a `null` predecessor and a distance of ***0*** (***zero***).
  - All other vertices have a `null` predecessor and a distance of ***infinity*** (`null`, ∞).

| Vertex | Path Tuple |
|---|---|
| A | (NULL, 0) |
| B | (NULL, ∞) |
| C | (NULL, ∞) |
| D | (NULL, ∞) |
| E | (NULL, ∞) |
| F | (NULL, ∞) |
| G | (NULL, ∞) |
| H | (NULL, ∞) |
| I | (NULL, ∞) |
| J | (NULL, ∞) |
| K | (NULL, ∞) |

# A Priority Queue

- Next we will need a ***priority queue*** to hold ***all*** of the vertices.
- It will be used to determine the ***order*** in which the vertices are ***visited***.
- The priority is based on ***distance*** from the starting vertex: the vertex with the ***shortest*** distance is always returned.
- Note that Java's built-in priority queue will ***not*** work!
  - Nor will your heap-based priority queue.
  - Neither will ***reprioritize*** the entries if they change (and they ***will*** change).




PRIORITY QUEUE

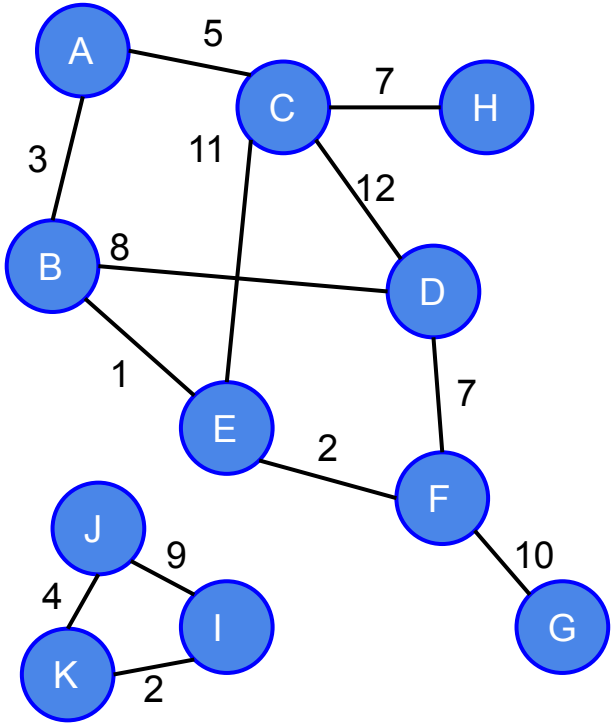
[A : 0] [E : ∞] [H : ∞] [G : ∞] [C : ∞] [I : ∞] [J : ∞] [D : ∞] [K : ∞] [F : ∞] [B : ∞]

To begin with, the only vertex with a distance less than infinity is the starting vertex, so it will be returned first.

# The Main Loop

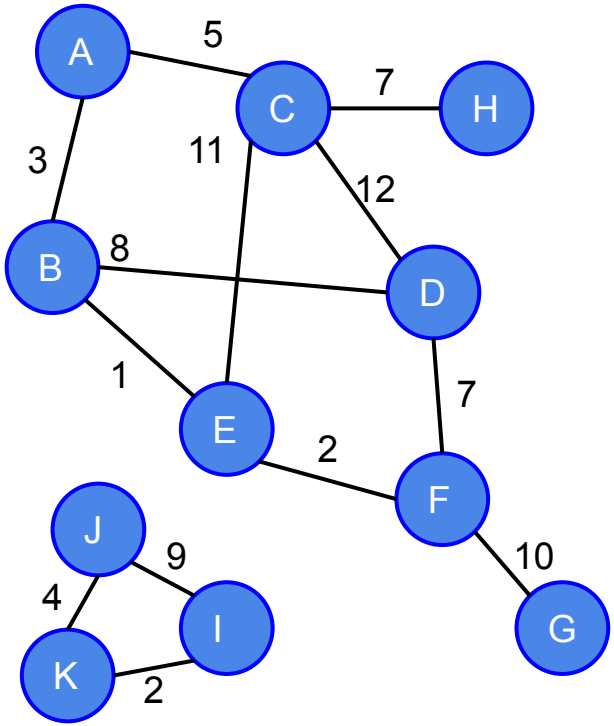
- To find the shortest path from ***S*** to ***E***:
  - Let ***V*** be the vertex in the priority queue with the shortest distance from ***S***. ***Remove*** it from the priority queue.
  - Let the distance from ***S*** to ***V*** be $D_{SV}$.
  - For each of V's neighbors ***N***:
    - Let $D_{SN}$ be the current known distance from ***S*** to ***N***.
      - Keep in mind that this starts as ***infinite*** for all but ***S***.
    - Let $D_{VN}$ be the cost of the edge between from ***V*** and ***N***.
    - Calculate the distance from ***S*** to ***N*** through ***V***:

      $D_{SVN} = D_{SV} + D_{VN}$
    - If $D_{SVN}$ is less than $D_{SN}$, update N's Path Tuple to $(V, D_{SVN})$.
- Stop when the priority queue is ***empty***, or $D_{SV}$ is ***infinity***.
  - If ***V*** is the vertex that is closest to ***S***, and the distance is ***infinity***, then there is no path from ***S*** to ***V*** or any other vertex that might be in the priority queue.




Let's practice using the example graph...

# An Example




| Vertex | Path Tuple |
|---|---|
| A ✔ | (null, 0) |
| B ✔ | ~~(null, ∞)~~ (A, 3) |
| C ✔ | ~~(null, ∞)~~ (A, 5) |
| D ✔ | ~~(null, ∞)~~ (B, 11) |
| E ✔ | ~~(null, ∞)~~ (B, 4) |
| F ✔ | ~~(null, ∞)~~ (E, 6) |
| G ✔ | ~~(null, ∞)~~ (F, 16) |
| H ✔ | ~~(null, ∞)~~ (C, 12) |
| I | (null, ∞) |
| J | (null, ∞) |
| K | (null, ∞) |

PRIORITY QUEUE

[A : 0] [K : ∞]
[H : 12]
[D : 11]
[F : 6]
[G : 16]
[B : 3]
[C : 5]
[E : 4]
[I : ∞]
[J : ∞]

# A Different View

Path (A to G): A B E F G

Cost: 16

| Vertex | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null |
| A | ✓ | 3, A | 5, A | | | | | | | | |
| B | | ✓ | | 11, B | 4, B | | | | | | |
| E | | | | | ✓ | 6, E | | | | | |
| C | | | ✓ | | | | | 12, C | | | |
| F | | | | | | ✓ | 16, F | | | | |
| D | | | | ✓ | | | | | | | |
| H | | | | | | | | ✓ | | | |
| G | | | | | | | ✓ | | | | |
| | 0 | 3, A | 5, A | 11, B | 4, B | 6, E | 16, F | 12, C | | | |

# A Different View

Path:

Cost:

| Vertex | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null | ∞, null |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |