Grade 7/8 Math Circles

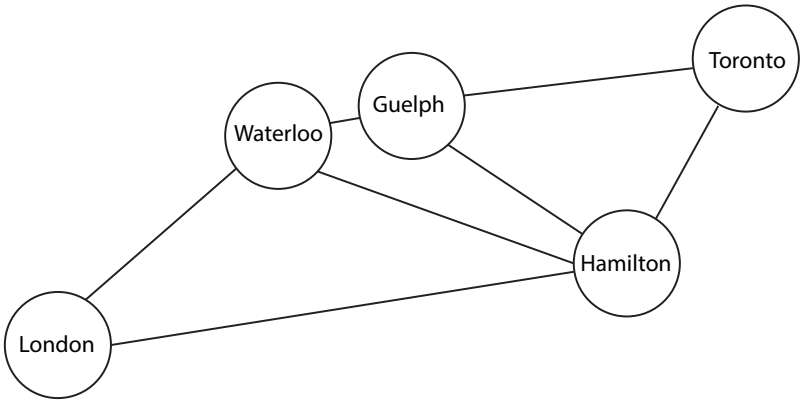
March 9th, 2022

Graph Theory and Search Algorithms

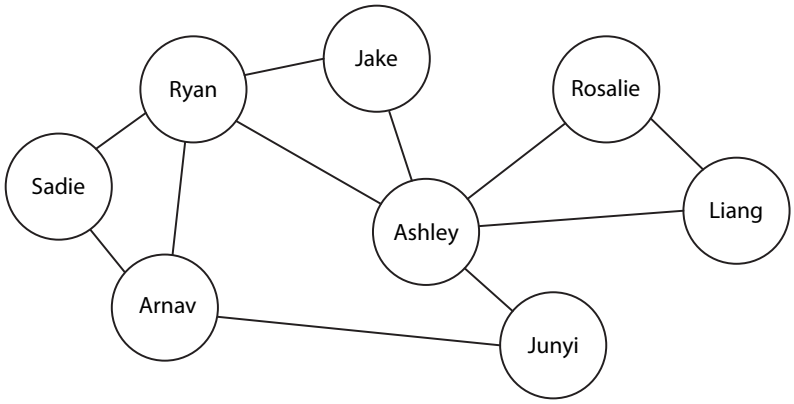
Introduction

Graph theory is a field of mathematics that studies a variety of graphs. But, instead of graphs being bar graphs or line graphs, we are referring to graphs that look like networks.

Looking at these networks, we can discover many different ways to model real-life scenarios and solve problems. For example, you could think about a map as being a network of cities, where you could try to figure out the shortest path between all the cities. Or, you could think about your social media and the network of who follows who, and in that network you could search to find a friend.



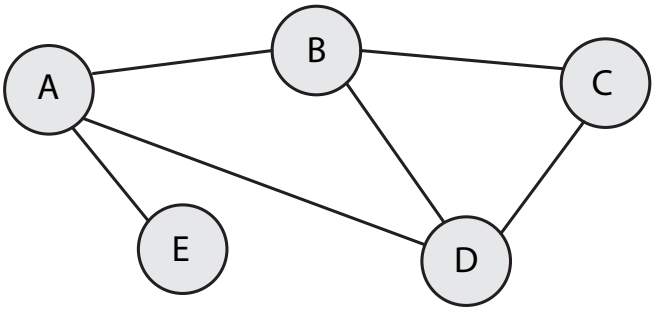
City network example—each circle is a city and the lines are roads that connect them



Friend network example—each circle is a person and the lines represent friend connections

In this lesson, we will define graphs in the perspective of graph theory and introduce two types of searching algorithms. These search algorithms are particularly important for when you can't see all of the information at once. For example, if you wanted to program a computer to go through the data points, or perhaps you're clicking through people's social media accounts in search of a person. The two algorithms we will discuss are quite effective and useful when you want to search a network.

Definitions in Graph Theory



A **graph** is a collection of vertices and edges that create a network, like above.

A **vertex** is a point on a graph. The plural of vertex is vertices. Typically, vertices are represented on graphs using circles. They sometimes also have labels, and we can use these labels to write them. The vertices in the graph above are A, B, C, D, E.

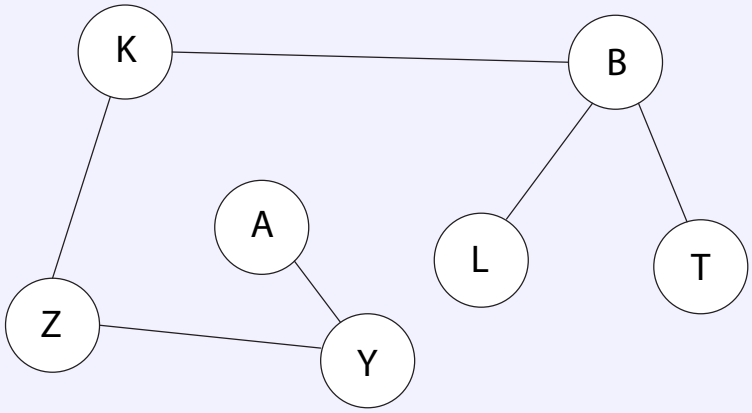
An **edge** is a line that connects two vertices. The **endpoints** of an edge are the vertices that it connects. We can write an edge as the pair of its endpoints. The edges in the graph above are $\{A, B\}, \{A, D\}, \{A, E\}, \{B, C\}, \{B, D\}$. As you can see, we usually list the vertices/endpoints in alphabetical order.

Two vertices, u and v, are called **adjacent** if $\{u, v\}$ is in the collection of edges. For example, in the graph above, A and B are adjacent, and E and C are *not* adjacent.

When we have that a vertex is adjacent to u, we can call it a **neighbour** of u. Furthermore, the collection of vertices that are adjacent to u is called the collection of **neighbours** of u or the **neighbourhood** of u. For example, in the graph above, the neighbourhood of D is A, B, C and we'd say that A, B, and C are neighbours of D.

Exercise 1

What are the vertices and edges in the graph below? What is the neighbourhood of B?



Search Algorithms

Sometimes when we have a graph, we might want to search through it to see what vertices we can reach from a start point or to try to find a vertex.

In this lesson, we will go over two search algorithms called "Breadth-First Search" (BFS) and "Depth-First Search" (DFS). These are different but both use a similar concept of colouring vertices.

As said earlier, we can add labels to vertices. Unlike our examples, they don't always have to be letters, and vertices can have more than one label. For our search algorithms, it'll make it much easier if we add a colour label to our vertices. In this case, we can pair the labels for vertices, similar to how we pair endpoints to represent edges. For example, if we wanted to colour vertex A purple, we would write this as $\{A, \text{purple}\}$.

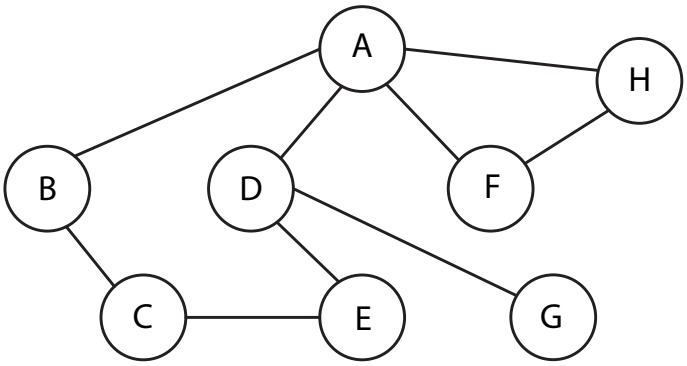
Breadth-First Search

When we do BFS, it'll be important to understand what a queue is. For our purposes, it's best to describe that a **queue** (pronounced like "cue") is like a line-up, where the people at the front of the line get served first (like at a bank). We will use the letter Q to represent our queue of vertices.

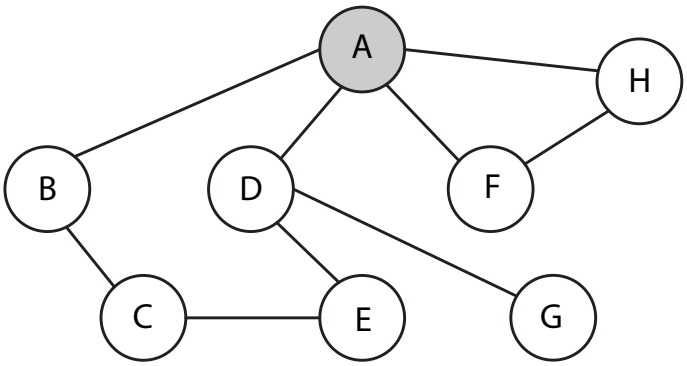
To help demonstrate this algorithm, we will do BFS on the example graph below. Note that if we are trying to find a vertex in particular, we can stop the search once we get to it. However, for the

sake of showing how the algorithm goes through the whole graph, we will continue the search until we have throughly searched each vertex.

To start, we have that all of the vertices are coloured white. When a vertex is in the queue, it will be coloured grey to help us visualize where we've been. Lastly, when we are done with a vertex, we will colour it black. You can feel free to use your own colours for this when you do it!



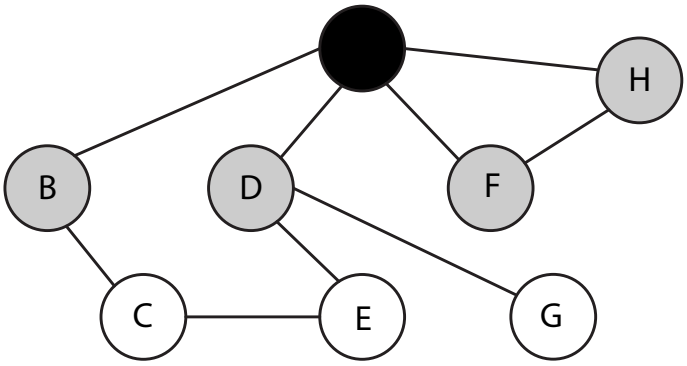
To begin the search, we want to have a starting point. You may be given a starting point or be able to choose it yourself. Let's pick our starting point to be A. Immediately, we will add A to the queue and colour it grey. Furthermore, we can write that $Q = A$.



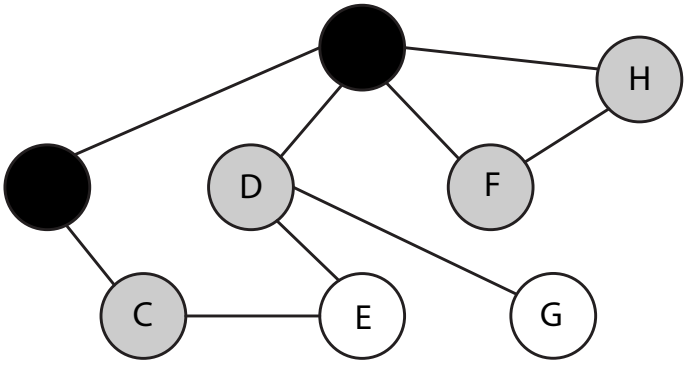
Whenever the queue is not empty, we should look the vertex at the front of the queue. In this looking stage, if we were searching for a vertex and the current vertex is the vertex we are looking for, we'd stop the algorithm. However, we are not searching for a vertex in this example, so we continue.

To continue, we remove our current vertex, A, from the queue and we add all of our current vertex's neighbours to the queue. When we remove a vertex from the queue, it means we're done with it,

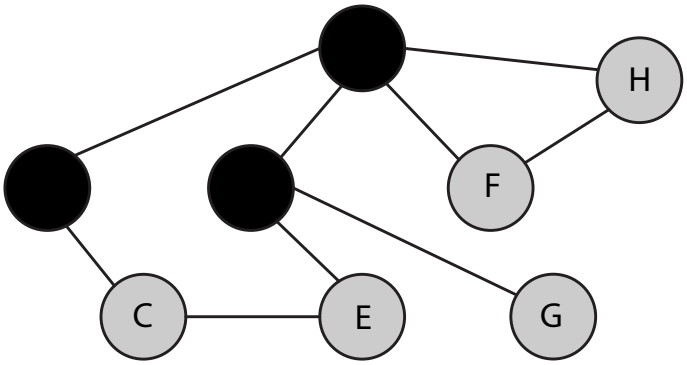
so we colour it black. Then, since we are adding the neighbours to the queue, we colour them grey. Since the neighbours of A are B, D, F, H, we colour those vertices grey. We also add them to the queue at the same time, and the order of adding them can be defined by us. We could add them alphabetically, or from left-to-right in the graph, or even from right-to-left. Let's add the vertices to the queue alphabetically for this example. Therefore, $Q = B, D, F, H$.



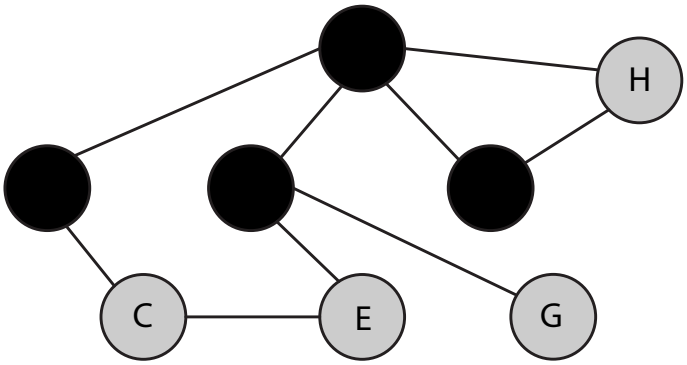
Now, since our queue isn't empty, we take the first vertex, which is B this time. When we are looking at B, we remove it from the queue. We also add the neighbours of B to the end of the queue. This is where the colouring helps us. We only want to add neighbours that are coloured **white** to the queue, because we haven't looked at them yet and they're not already in the queue. Therefore, since C is the only neighbour of B that's coloured white, our queue is now $Q = D, F, H, C$ and we have the following graph.

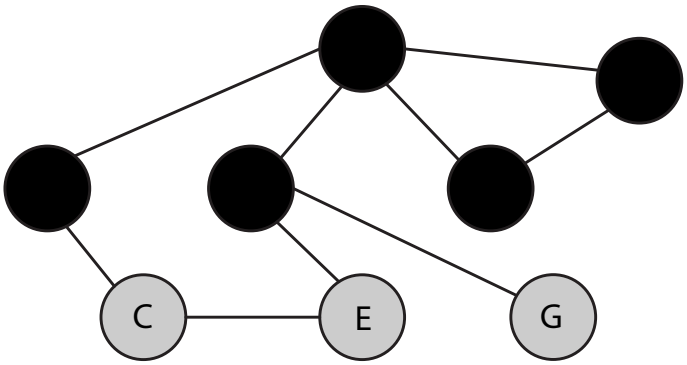


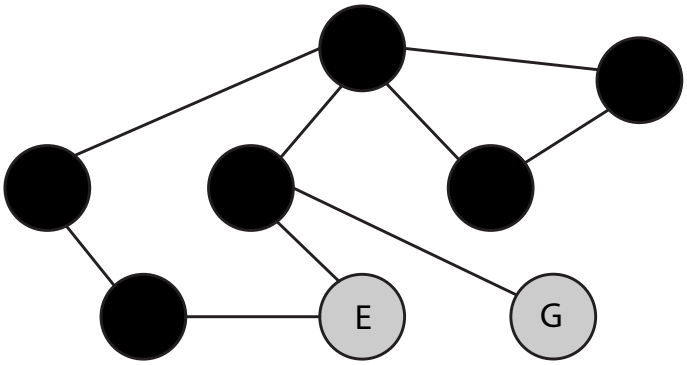
The pattern continues while the queue is not empty. Next, we would look at D since it's at the front of our queue. We remove it from the queue and add all its neighbours that are coloured white to the end of the queue (in alphabetical order). So, we have $Q = F, H, C, E, G$ and the following graph.

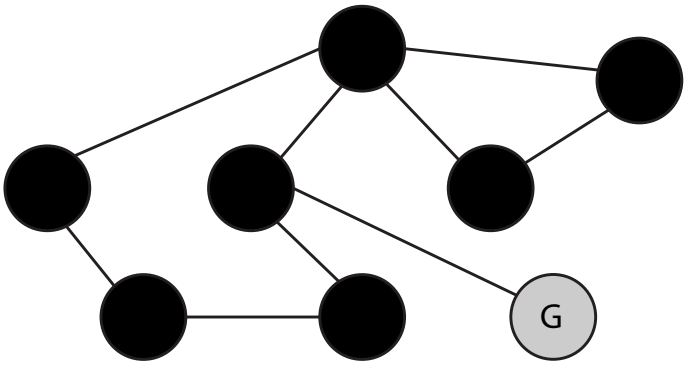


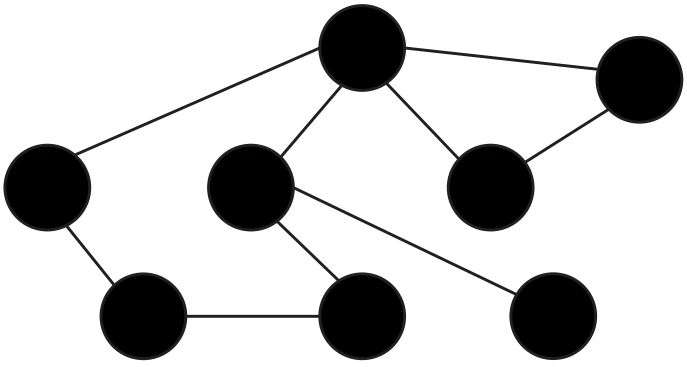
Now, as you can see, there are no neighbours left that are coloured white. Therefore, we are just going to start emptying the queue without adding any more vertices. Whenever we look at a vertex, we remove it from the queue and colour it black. Since the queue is currently $Q = F, H, C, E, G$, we will first look at F, then H, then C, then E, and then finally G. Therefore, the following sequence of graphs will occur.











Exercise 2

What is the queue for each of the five graphs above?

As you can see, drawing each graph takes up a lot of space! Therefore, if we feel confident, we can simply list out the order that we look at each vertex to show how we completed the steps too. For example, for our BFS of the graph above, we looked at the vertices in the following order: A, B, D, F, H, C, E, G. This will be fun to compare to the order that we do DFS!

Furthermore, as an example, if we were looking for vertex C in the graph and started with A again, then we would have looked at the vertices in the same order, but not continuing past C. In other words, we would have looked at the vertices in the following order: A, B, D, F, H, C. Another example could be that if we were looking for vertex F, we would have looked at the vertices in the order A, B, D, F, which is even shorter than looking for C.

Lastly, now that we've gone through the algorithm, we can see that it searches the closest points to the starting point as its first steps, going from side to side. In other words, we start by stretching the search area to be wide or side-to-side and close to the start point. This is why it's named *breadth*-first search!

Exercise 3

For the same graph as above, use BFS to do the following searches. Add neighbours to the queue in alphabetical order for both searches and list the order that you looked at the vertices.

(a) Search for vertex H starting from vertex B.

(b) Search for vertex E starting from vertex F.

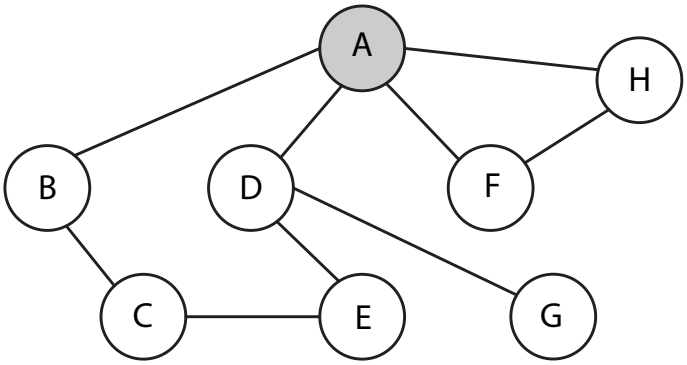
Depth-First Search

When we do DFS, we search as far as we can, and then backtrack when we reach a dead-end.

We can use the same colour scheme as with BFS, except this time, we don't have a queue and instead we will colour vertices grey when we've looked at them but are not done with them yet. White will still mean we haven't interacted with that vertex yet and black will still mean we are completely done with that vertex.

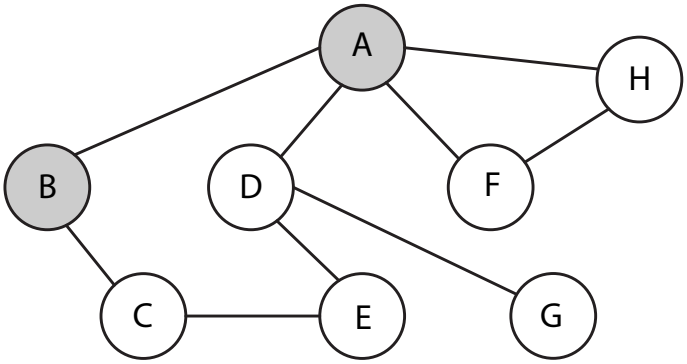
We will use the same graph as with BFS to help show the differences between the two search methods. Note again that if we are trying to find a vertex in particular, we can stop the search once we get to it. However, for the sake of showing how the DFS algorithm goes through the whole graph, we will continue the search until we have throughly searched each vertex.

To begin the search, we want to have a starting point. You may be given a starting point or be able to choose it yourself. Let's pick our starting point to be A again. We immediately look at A and colour it grey.

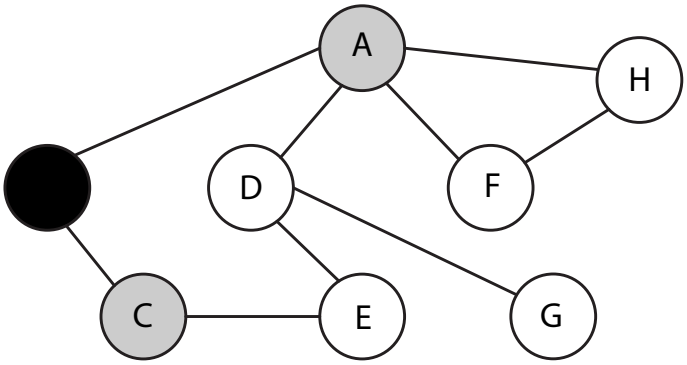


In this looking stage, if we were searching for a vertex and the current vertex is the vertex we are looking for, we'd stop the algorithm. However, we are not searching for a vertex in this example, so we continue.

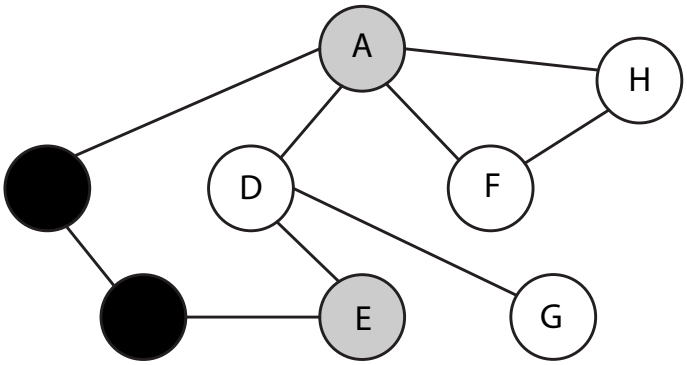
The next step in DFS is to look at a single neighbour of our current vertex that's coloured white. We can pick a neighbour in a variety of ways, like alphabetically, from left-to-right on the graph, or from right-to-left. Let's choose neighbours alphabetically. So, the neighbour we will look at next is B. The key to this search method is that we only colour vertices black if they have no more neighbours that are coloured white to look at, so we leave A grey, and also colour B grey since we are looking at it now.



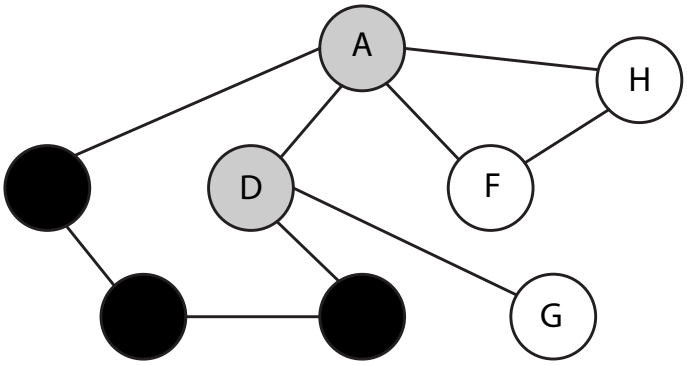
We again choose a neighbour of our current vertex that's coloured white. Since our current vertex is B, we look at C next. Furthermore, since B has no other neighbours that are coloured white, we can colour it black.



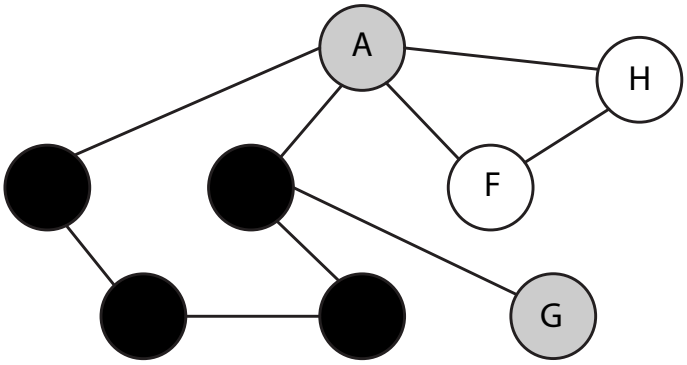
Our current vertex is C, and the only neighbour of C that's coloured white is E, therefore we look at E next and can colour C black.



A similar outcome occurs when we look at E, where D is their only neighbour that's coloured white so we look at D next and colour E black.

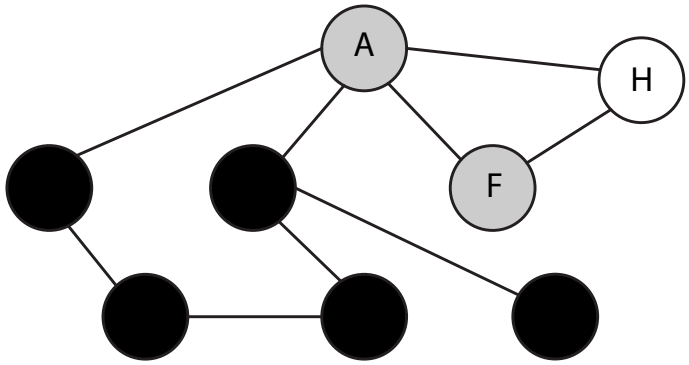


Now, looking at D, we see that G is it's only neighbour that's coloured white so we can look at G and colour D black.

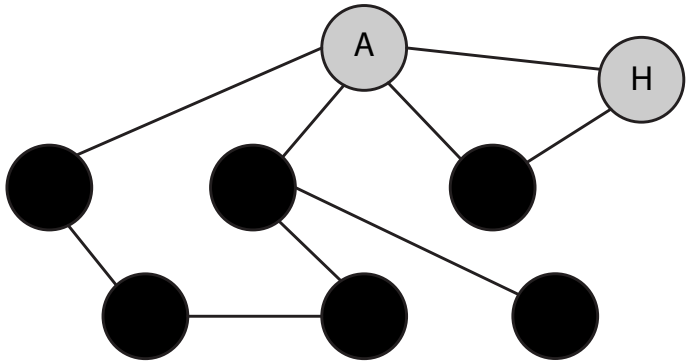


At this point, looking at G, we see that it has no neighbours that are coloured white. Therefore, we should colour G black. We've also hit a dead-end. So, we will need to backtrack to the most recent grey vertex. In this case, that vertex is A.

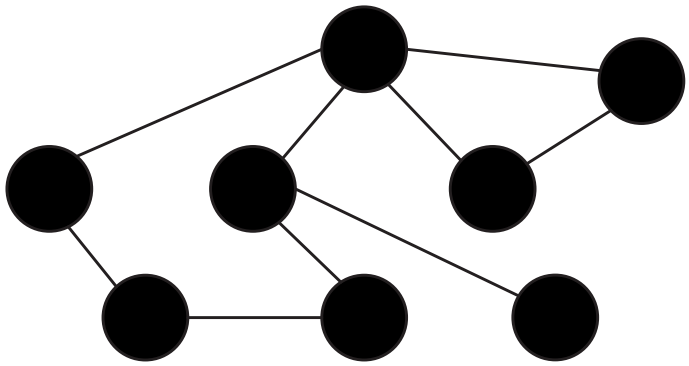
Looking at A again, we pick another neighbour that's coloured white according to alphabetical order. Therefore, we look at F and colour it grey. Note that we still keep A grey because it has another neighbour that's coloured white.



Now, looking at F, we see that it has H as its only neighbour that's coloured white, so we colour F black and H grey since we are looking at it next.



With H, we have hit another dead-end because it has no neighbours that are coloured white, so we colour H black. Then, we backtrack to the most recent grey vertex, which is A again. But now, A actually has no neighbours that are coloured white left, so we also colour A black, and we are done!



Similar to BFS, we needed to draw a lot of graphs to show our DFS process. Instead, again similar to BFS, we can create a list of what order we looked at each vertex to show our process. For our DFS of the graph, we looked at the vertices in the following order: $A, B, C, E, D, G, A, F, H, A$. This is very different from our BFS order!

Furthermore, as an example, if we were looking for vertex C in the graph and started with A again, then we would have looked at the vertices in the same order, but not continuing past C. In other words, we would have looked at the vertices in the following order: A, B, C. Notice how finding C took less steps for DFS than it did for BFS. Another example could be that if we were looking for vertex F, we would have looked at the vertices in the order A, B, C, E, D, G, A, F. Notice how finding F took more steps for DFS than it did for BFS.

Lastly, now that we've gone through the algorithm, we can see that it searches as far as possible away from the start point as its first steps. In other words, we start by stretching the search area to be longer and farther from the start point. This is why it's named *depth*-first search!

Exercise 4

For the same graph as above, use DFS to do the following searches. Choose neighbours in alphabetical order for both searches and list the order that you looked at the vertices.

(a) Search for vertex H starting from vertex B.

(b) Search for vertex E starting from vertex F.

How do the ordered lists using DFS in this exercise compare to the ordered lists using BFS in Exercise 3? Which search method is quicker for each search?

Conclusion

Graph theory is a very cool field of mathematics and has many different applications and problems to solve. In this lesson, we've looked at a couple of search algorithms that can help us when we want to search through a graph or network. Both BFS and DFS have pros and cons and it might be better to use one or the other in different situations! For example, in our example graph, we noticed that finding vertex C from vertex A was faster with DFS, but finding vertex F from vertex A was faster with BFS.

As an optional activity, try to think of other scenarios where BFS would be better, or alternatively where DFS would be better. An example where BFS would be better is if you're looking at a map and trying to find a store that's the closest to you. Each store could be a vertex and each edge could be a road! Clearly if you want a shorter distance, it's better to look at all the vertices closer to the start point first, which is exactly what we do with BFS.