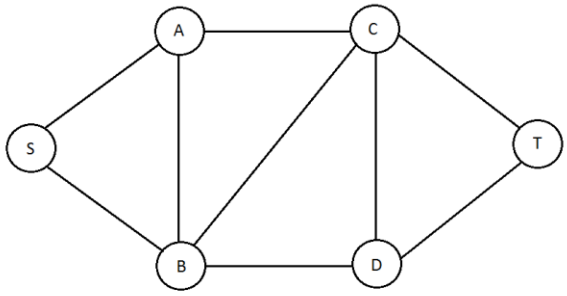
DEPTH FIRST SEARCH OF A GRAPH

Here's a graph along with its edge vector representation; for simplicity we assign all edges a cost of 1 for now. I'm using an adjacency matrix here (so there are zeros for missing edges). Your Graph.java class uses an adjacency map, but this matrix is easier to draw or print out.



[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

We want a method that automatically explores the graph and finds a path from vertex S to vertex T, if that's feasible.

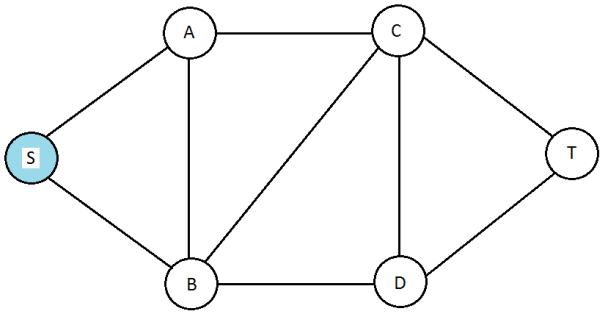
To prime the search, we take our starting node S and add it to a particular kind of vector, called toExplore. We also initialize another vector of boolean values, called visited, to all-false except for S. Since we're starting the search there, we can consider S as visited.

	vertices		visited		yetToExplore
[0]	S	[0]	true	[0]	S
[1]	A	[1]	false		
[2]	B	[2]	false		
[3]	C	[3]	false		
[4]	D	[4]	false		
[5]	T	[5]	false		

Now, we repeat these steps:

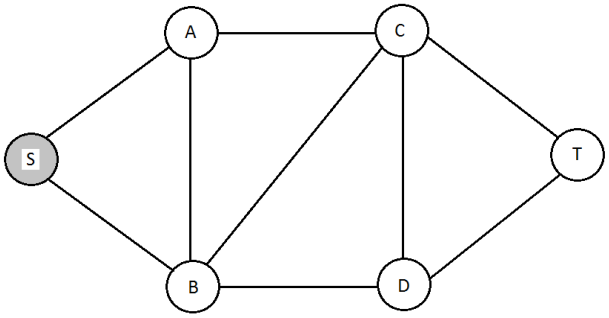
1) While the stack is not empty,
2) We pop v, the top vertex in yetToExplore, out of the stack
3) If v is our destination, then we're done; exit the loop
4) Else,
5) For each neighboring vertex that v has an edge to,
6) If we have not visited this neighbor,
7) Then add it to the top of yetToExplore and set its visited to true

The first time through, the vertex we added to yetToExplore is S. We take that off the yetToExplore stack. S is not the destination, T. So we look at the places we can go from S.



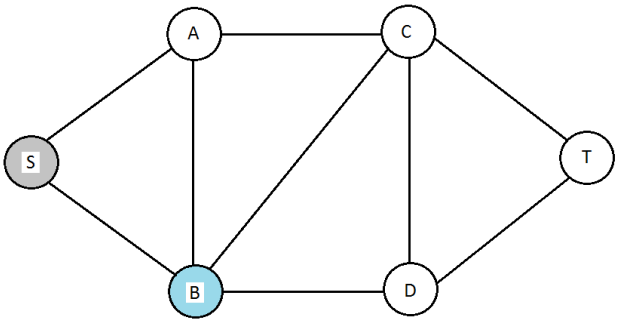
	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

From S, we can get to A and B; once S is off yetToExplore, and A and B are on it, we'll have this.



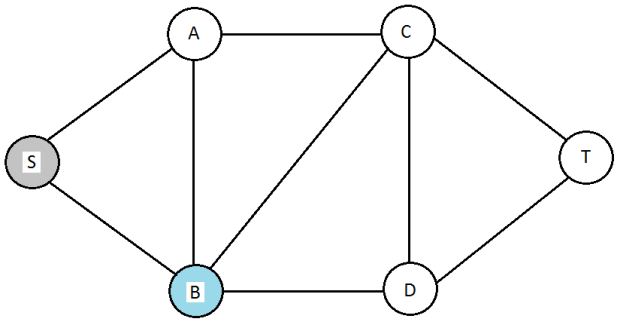
	vertices	visited	yetToExplore
[0]	S	true	A
[1]	A	true	B
[2]	B	true	
[3]	C	false	
[4]	D	false	
[5]	T	false	

In the next round, we'll take B off yetToExplore. B is also not the vertex we are trying to reach. So then, we look at B's neighbors:



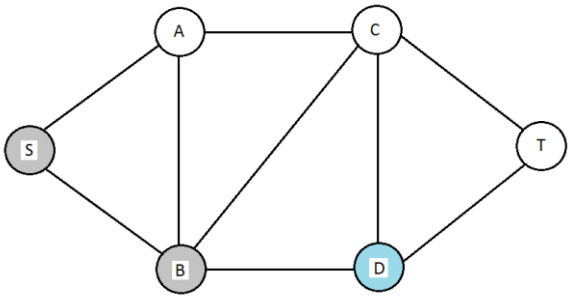
	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

From B, we can get to S and A, but we've already visited those. The unvisited neighbors we still have are C and D, so they go in the yetToExplore vector on top of A, and their visited gets marked as true.



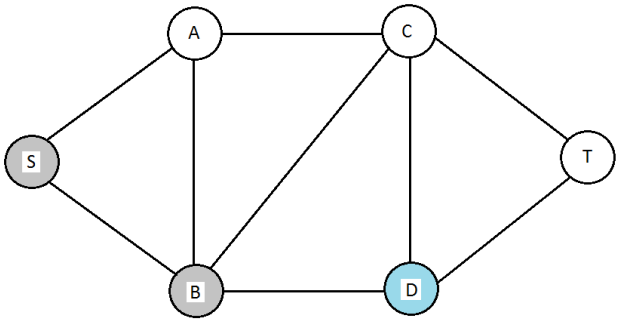
	vertices	visited	yetToExplore
[0]	S	true	A
[1]	A	true	C
[2]	B	true	D
[3]	C	true	
[4]	D	true	
[5]	T	false	

Now in the next round, we'll take D off yetToExplore. D is also not our destination T. Now, we look at D's neighbors:



	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

The only unvisited vertex left is T, so it goes into yetToExplore and gets marked as visited.

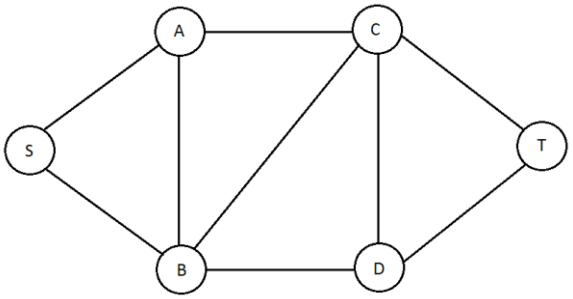


	vertices	visited	yetToExplore
[0]	S	true	A
[1]	A	true	C
[2]	B	true	T
[3]	C	true	
[4]	D	true	
[5]	T	true	

In the final round, we pop T off the yetToExplore stack. T is our destination and we have reached it. So we're done; our search is successful.

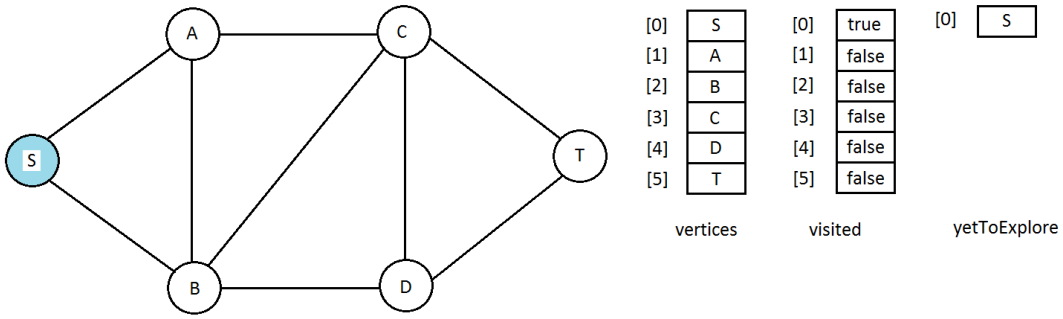
The good thing about depth first search is that it will always explore the graph completely. The bad thing about it is that it will not care about finding vertices near the starting vertex before vertices that are farther away; in fact, depth first search is kind of biased towards finding more distant vertices first, due to that stack.

A second search algorithm, breadth first search, is more efficient at exploring the graph in order from the starting vertex. This search algorithm is almost the same, except that *we add to one end of yetToExplore and remove from the other*. This biases the search toward finding vertices close to the starting one first. We begin with the same graph and the same edges:

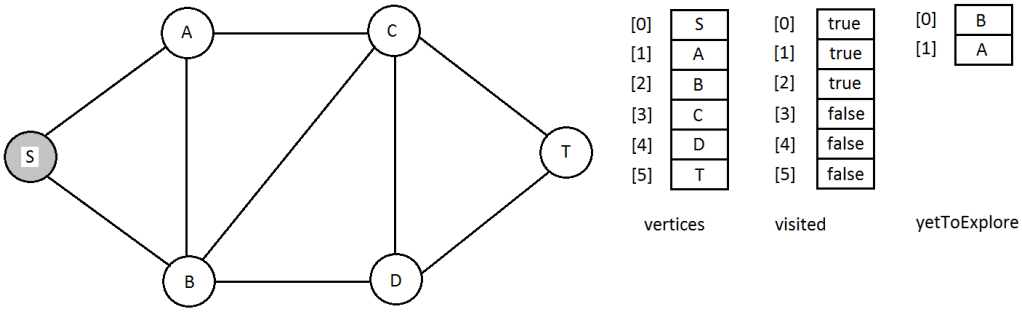


	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

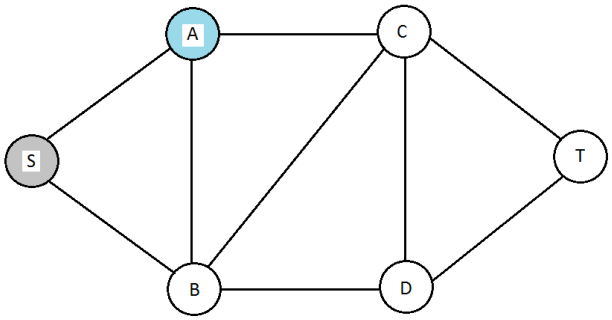
We even start out the same way, with S added to yetToExplore and marked as visited:



When we remove S from the end of the vector, we push its neighbors onto the front of yetToExplore, as before:



Now, the next time through, we'll inspect A. A has only one unvisited neighbor, C, so C goes in the front of yetToExplore and gets marked as visited:

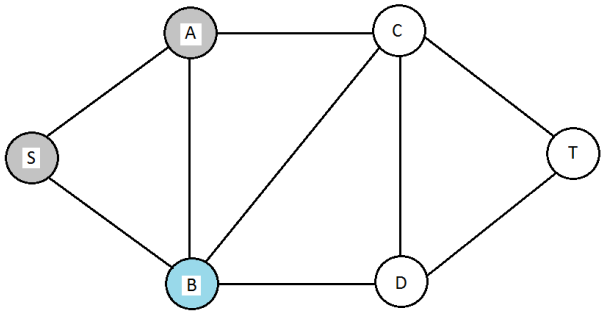


	vertices
[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

	visited
[0]	true
[1]	true
[2]	true
[3]	true
[4]	false
[5]	false

	yetToExplore
[0]	C
[1]	B

Then, we'll take B off and inspect it. B has one unvisited neighbor, D, so D goes onto the front of yetToExplore:

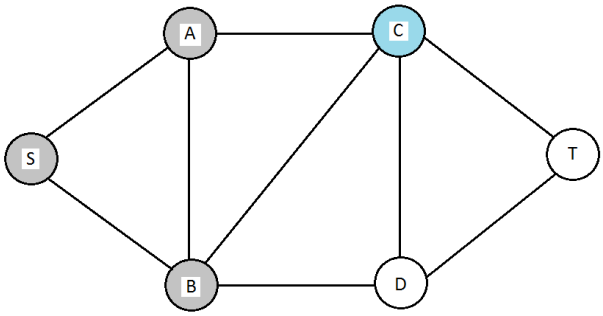


	vertices
[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

	visited
[0]	true
[1]	true
[2]	true
[3]	true
[4]	true
[5]	false

	yetToExplore
[0]	D
[1]	C

Now, we take C off the end of yetToExplore. From C, we can reach the unvisited vertex T, so we mark T as visited, and add it to the front of yetToExplore:

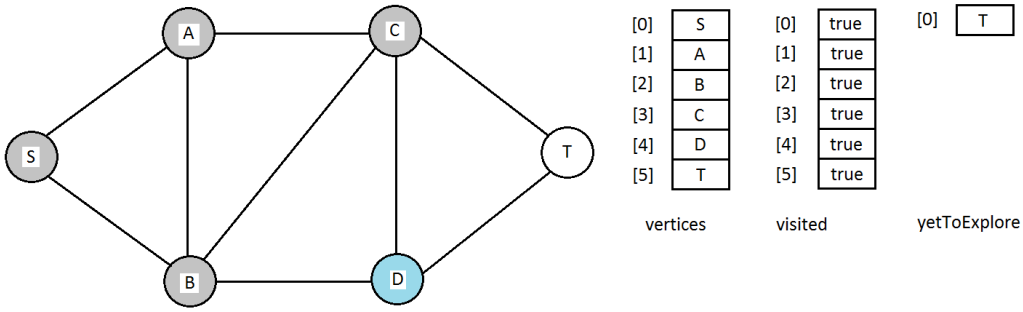


	vertices
[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

	visited
[0]	true
[1]	true
[2]	true
[3]	true
[4]	true
[5]	true

	yetToExplore
[0]	T
[1]	D

When we remove D from the end, nothing gets added to yetToExplore, because D has no unvisited neighbors:



Finally, we'll remove the T from yetToExplore, and after that, the stack will be empty. Again, this means we can stop.

Unlike depth first search, breadth first search always finds vertices close to the starting path first, in terms of hops from one vertex to another. It may take longer to do that, though. Getting an optimal solution (the shortest path) is usually more computationally expensive than just getting any solution (any path at all), so this is not surprising.