

Breadth First Search Algorithm Tutorial with Java



Elliot Forbes

6 Minutes

Apr 15, 2017



ai

java

Table Of Contents

- Uninformed Search vs Informed / Heuristic Search
- Breadth First Search
- How it Works
- Our Breadth First Search Class
- The Results
- Our Driver Class
- Conclusion



Subscribe on YouTube!

Join the thousands of developers already subscribed!
<https://youtube.com/tutorialedge>

Subscribe for the Latest Content!

* indicates required

Email Address *

Subscribe

JAVA

[Breadth First Search Algorithm Tutorial with Java](#)

[Depth First Search in Java](#)

[Depth Limited Search in Java](#)

This lesson is part of the course: [Artificial Intelligence](#)

Uninformed Search vs Informed / Heuristic Search

The next couple of algorithms we will be covering in this Artificial Intelligence course can be classed as either:

uninformed or blind searches: in which our algorithms have no additional information about states beyond that provided in the problem definition.

Informed or Heuristic searches: in which our algorithms have some extra knowledge about the problem domain and can distinguish whether or not one non-goal state is “more promising” than another.

Breadth First Search

BFS is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors and so on until the best possible path has been found. Due to the fact that this strategy for graph traversal has no additional information about states beyond that provided in the problem definition, Breadth First Search is classed as an uninformed or blind search.

Breadth First Search Utilizes the queue data structure as opposed to the stack that Depth First Search uses.

BFS uses a queue data structure which is a 'First in, First Out' or FIFO data structure. This queue stores all the nodes that we have to explore and each time a node is explored it is added to our set of visited nodes.

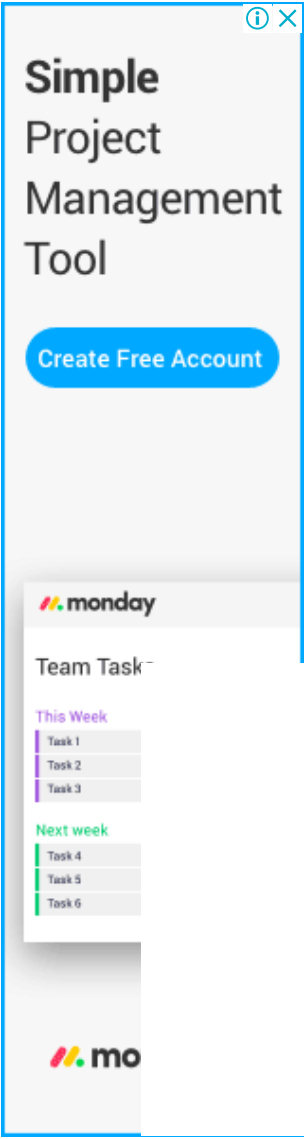
If we were to conduct a breadth first search on the binary tree above then it would do the following:

1. Set Node 1 as the start Node
2. Add this Node to the Queue
3. Add this Node to the visited set
4. If this node is our goal node then return true, else add Node 2 and Node 3 to our Queue
5. Check Node 2 and if it isn't add both Node 4 and Node 5 to our Queue.
6. Take the next node from our Queue which should be Node 3 and check that.
7. If Node 3 isn't our goal node add Node 6 and Node 7 to our Queue.
8. Repeat until goal Node is found.

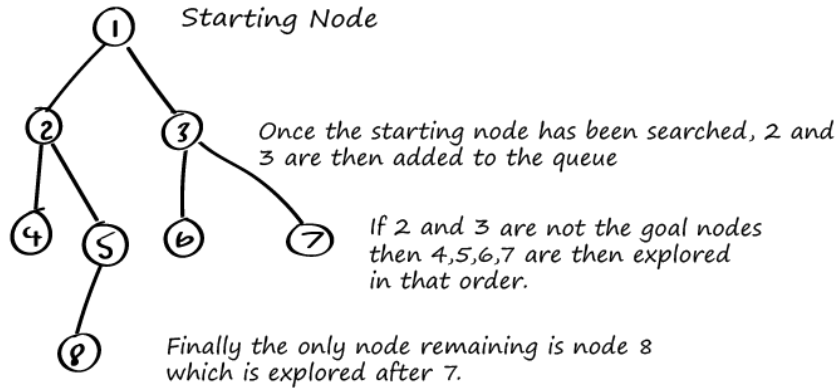
If we stopped execution after Node 3 was checked then our Queue would look like this:

Node 4, Node 5, Node 7, Node 8.

As you can see, if you follow this algorithm through then you will recursively search every level of the binary tree going deeper and deeper until you find the shortest possible path.



Breadth First Search



How it Works

Say we had a map of the London Underground, each station would represent a node which would itself have a smaller list of stations that are directly connected to it. The entire map of the London Underground represents our Graph and each of the stations on that graph represent a node.

For example, take Westminster station for example. This station could be represented as a node which would have: STATION 1, STATION 2 and STATION 3 in its list of child nodes.

We can represent this sort of structure like so in Java:

```

1  import java.lang.reflect.Array;
2  import java.util.ArrayList;
3
4  /**
5   * The Node class represents a station
6   * in this tutorial and will as such have
7   * a string representing the station's name.
8   * As well as an ArrayList of nodes that will store
9   * any instantiated nodes children.
10 */
11 public class Node {
12
13     //    A Unique Identifier for our node
14     public String stationName;
15     //    An arraylist containing a list of Nodes that
16     //    This node is directly connected to - It's child nodes.
17     Node leftChild;
18     Node rightChild;
19
20     public Node(String stationName, Node firstChild, Node
secondChild){
21
22         this.stationName = stationName;
23         this.leftChild = firstChild;
24         this.rightChild = secondChild;
25     }
26
27     public ArrayList<node> getChildren(){
28         ArrayList<node> childNodes = new ArrayList<>();
29         if(this.leftChild != null)
30         {
31             childNodes.add(leftChild);
32         }
33         if(this.rightChild != null) {
34             childNodes.add(rightChild);
35         }
36         return childNodes;
37     }
38
39     //    An auxiliary function which allows
40     //    us to remove any child nodes from
41     //    our list of child nodes.
42     public boolean removeChild(Node n){
43         return false;
44     }
45
46 }

```

Our Breadth First Search Class

In this tutorial I will be implementing the breadth first searching algorithm as a class as this makes it far easier to swap in and out different graph traversal algorithms later on.

```

1  import java.util.ArrayList;
2  import java.util.LinkedList;
3  import java.util.Queue;
4
5  /**
6   * basic breadth first search in java
7   */
8  public class BreadthFirstSearch {
9
10     Node startNode;
11     Node goalNode;
12
13     public BreadthFirstSearch(Node start, Node goalNode){
14         this.startNode = start;
15         this.goalNode = goalNode;
16     }
17
18     public boolean compute(){
19
20         if(this.startNode.equals(goalNode)){
21             System.out.println("Goal Node Found!");
22             System.out.println(startNode);
23         }
24
25         Queue<node> queue = new LinkedList<>();
26         ArrayList<node> explored = new ArrayList<>();
27         queue.add(this.startNode);
28         explored.add(startNode);
29
30         while(!queue.isEmpty()){
31             Node current = queue.remove();
32             if(current.equals(this.goalNode)) {
33                 System.out.println(explored);
34                 return true;
35             }
36             else{
37                 if(current.getChildren().isEmpty())
38                     return false;
39                 else
40                     queue.addAll(current.getChildren());
41             }
42             explored.add(current);
43         }
44
45         return false;
46
47     }
48
49 }

```

The Results

Whilst Breadth First Search can be useful in graph traversal algorithms, one of its flaws is that it finds the shallowest goal node or station which doesn't necessarily mean it's the most optimal solution. Breadth First Search is only every optimal if for instance you happen to be in a scenario where all actions have the same cost.

Breadth First graph traversal algorithms also happen to be very computationally demanding in the way that they calculate the shortest path. Take for instance if we have a binary tree of depth 10. The binary tree contains nodes which contain a maximum of 2 child nodes each, this is otherwise known as having a branching factor equal to 2. if we wanted to compute the optimal path for this graph then we would have to traverse, in a worst case scenario, 512 distinct nodes. Given that on modern machines this isn't exactly what we would consider

demanding, imagine if we had a new graph that had 3 child nodes for every node and the same depth of 10. With this new graph we would have to traverse, in a worst case scenario, 19,683 different nodes. And given that this is only at depth 10 with 3 child nodes, you can easily extrapolate the numbers for yourself. With a branching factor of 10 and a depth of 16, it would take 350 years to compute the solution on an ordinary personal computer, give or take.

Our Driver Class

```
1  /**
2   * Our main driver class which instantiates some example nodes
3   * and then performs the breadth first search upon these newly
4   created
5   * nodes.
6   */
7  public class Driver {
8
9      public static void main(String args[]){
10         Node station1 = new Node("Westminster", null, null);
11         Node station2 = new Node("Waterloo", station1, null);
12         Node station3 = new Node("Trafalgar Square", station1,
13 station2);
14         Node station4 = new Node("Canary Wharf", station2,
15 station3);
16         Node station5 = new Node("London Bridge", station4,
17 station3);
18         Node station6 = new Node("Tottenham Court Road",
19 station5, station4);
20
21         BreadthFirstSearch bfs = new
BreadthFirstSearch(station6, station1);
22
23         if(bfs.compute())
24             System.out.print("Path Found!");
25     }
26 }
```


Conclusion

If you found this tutorial useful or require further assistance then please let me know in the comments section below!

Was This Post Helpful?

✓ Yes

✗ No

 Submit a PR: [Edit on Github](#) ▶

Do you have any suggestions as to how we can make it better?

Submit Feedback



Courses

- [Python](#)
- [Angular](#)
- [Golang](#)
- [Artificial Intelligence](#)
- [Web Development](#)

Quick Links

- [Search Page](#)
- [About](#)
- [Get Involved](#)

Suggested Resources

- [Top Angular Programming Books](#)
- [Top Python Programming Books](#)
- [Top Golang Programming Books](#)

Follow Me

- [@Elliot_f](#)
- [TutorialEdge](#)
- [elliotforbes](#)



TutorialEdge.net

TutorialEdge.net is a totally free programming tutorial site that has been built with the intention of helping as many people as possible learn to program. The site is also completely opensourced and you can contribue here: [elliotforbes/tutorialedge-v2](#), any pull requests will be greatly appreciated!