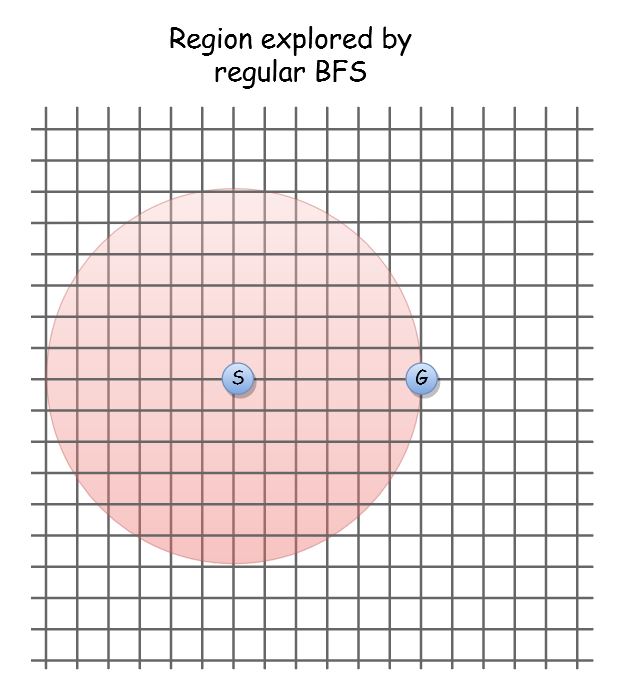
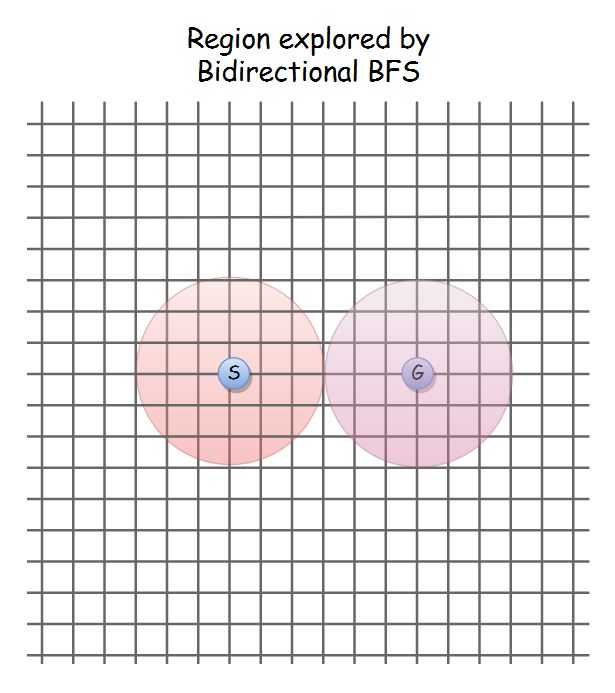
**Bidirectional Search**

POSTED [JANUARY 21, 2018](http://theoryofprogramming.com/2018/01/21/bidirectional-search/) [VAMSI SANGAM](http://theoryofprogramming.com/author/vamsi-sangam/)

Hello people! In this post we will talk about Bidirectional Search, a powerful search strategy which can be used if you have a goal directed agent in a (possibly infinite) search space. The idea behind Bidirectional search is to run two simultaneous searches, one forward from the source state and the other backward from the goal, hoping that the two searches meet somewhere in the middle.

In this post we will talk about Bidirectional Breadth First Search, where we apply BFS from the source and goal simultaneously until search routes intersect. Visually, the way regular BFS would progress would be something like this –

[](http://theoryofprogramming.azurewebsites.net/wp-content/uploads/2018/01/bidirectional-search-1.jpg)But if we applied BFS simultaneously from source (S) and goal (G), visually it would look something like this –

[](http://theoryofprogramming.azurewebsites.net/wp-content/uploads/2018/01/bidirectional-search-2.jpg)We can already start to seem some improvement as the number of vertices visited would be lesser.

In terms of complexity, if branching factor is b, and distance from source to goal is d, then, BFS would take O(bd), but Bidirectional search would take O(bd/2 + bd/2) ⇒ O(bd/2), which is quite better than O(bd). So in many cases a bidirectional search is faster as the amount of exploration done is lesser.

**Properties of Bidirectional search**

* Our goal in Bidirectional search is to find a path from source to goal.
* We can use any search algorithm on both sides, Bidirectional search is more of a strategy than a fixed algorithm. Using BFS on both sides is the most popular option as it guarantees an optimal path.
* Bidirectional search must be used only when your goal is well defined. Eg. for a Rubik’s cube, your goal is to fill each side with one color, which is your goal and your current configuration of cube is your source state. We could use Bidirectional search in this scenario. But in chess, your goal is to checkmate the other player, but a checkmate can happen in thousands of ways. Bidirectional search isn’t feasible in chess.
* Bidirectional search using BFS needs the edge weights to be same or non-existent. So usually Bidirectional BFS is used in undirected unweighted graphs.

**Code**

Writing the code for Bidirectional BFS is easier if you have already written the code for [Breadth First Search using queue](http://theoryofprogramming.azurewebsites.net/breadth-first-search-algorithm-using-queue/). Basically, all the variables we used in BFS get doubled here. That is, for Bidirectional BFS, we will use 2 queues and 2 parent arrays. One set will be used for applying BFS from source and other set will be used for applying BFS from goal. So we can call them –

* sourceParent – the parent array to construct paths for BFS from source vertex.
* sourceQueue – the queue for BFS from source vertex.
* goalParent – the parent array to construct paths for BFS from goal vertex.
* goalQueue – the queue for BFS from goal vertex.

Although we are using a separate set of variables for BFS from source and goal, the way in which they operate will be the same, i.e., the steps like picking first element in queue, enqueuing its adjacent vertices, etc. We can put that logic in a separate method called *BFSExplore()*.

So how will we know if the BFS from source and goal have an intersecting node? After expanding a layer on either side, we will check if a vertex has been marked as visited in both of the BFS. If yes, then that is our intersection node. Recall, that we can check if a vertex has been visited if its parent value has been set.

Now the only thing left is to return the final path. There’s no genius logic for this, just do it in a brute force way  . Add all vertices from source → *intersecting\_vertex* using *sourceParent* into a collection. Then add all the vertices from *intersecting\_vertex* → goal using *goalParent* into the same collection. Be careful, don’t add the *intersecting\_vertex* twice in the path

I have given you all the theory you need to write the code for Bidirectional BFS. It may take some time, but do try your best to write the code yourself. You can refer to my code below if you get stuck –