Breadth First Search Demystified

A lot of students are facing difficulties in implementing the BFS. The goal of this document is to give complete details of algorithm for Ghost Chasing.

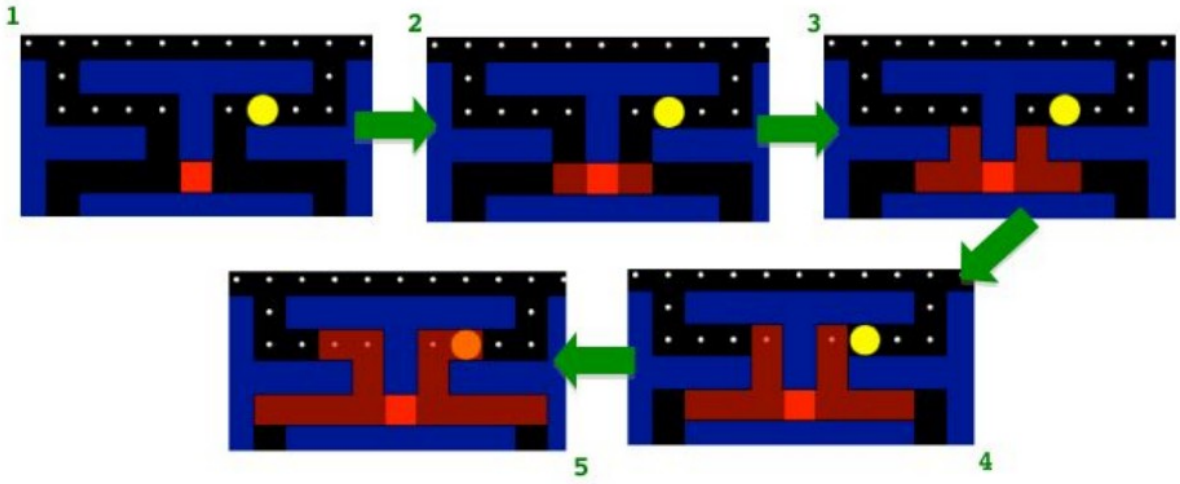
Lets first define an enum this will help us to keep the track of direction to move.

```
enum Direction {
UP=0, DOWN=1, LEFT=2, RIGHT=2;
};
```

When a Ghost needs to make a decision about where to turn, it must choose the direction that will bring it to its target the fastest. You will be implementing a BreadthFirst Search (BFS) algorithm to determine this optimal direction. We will be using STL deque class for implementation.

The general idea behind BFS is to first search all the squares neighboring you, then expand and search the squares neighboring your neighbors, then expand again, and so on, until the entire maze (or board) has been searched. **A BFS guarantees that a target will be reached first by the shortest path**. Note that we need to somehow mark which squares have already been visited, so we don't loop around the maze forever.

One possible way is what I discussed in the class that is label all the grid cells with numbers 0 (can't visit), 1 (already visited) and (-1 remained to be visited) but this technique require moving from pacman to ghost instead of reverse direction to find the direction in which the ghost must move.

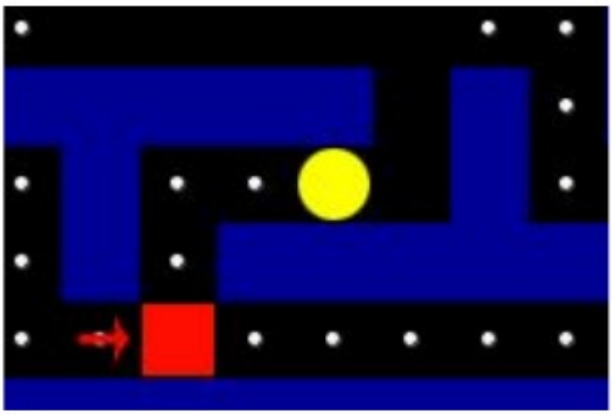


Example of BFS path starting at the ghost's location and searching for Pacman's location, examining all possible neighbors at each step.

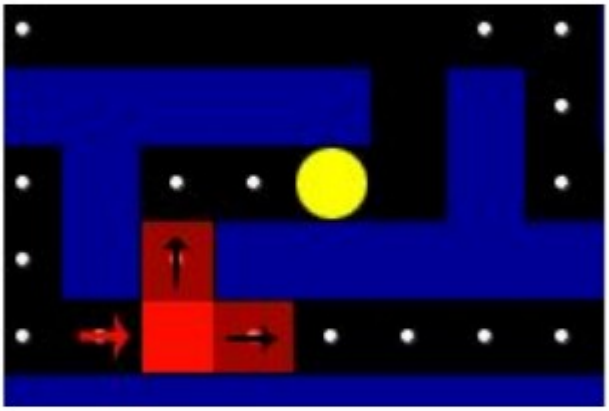
Here I will be presenting an alternative representation, where a visited grid cell will be

represented by the direction (up, down, left, right) ghost took initially to arrive at this cell.

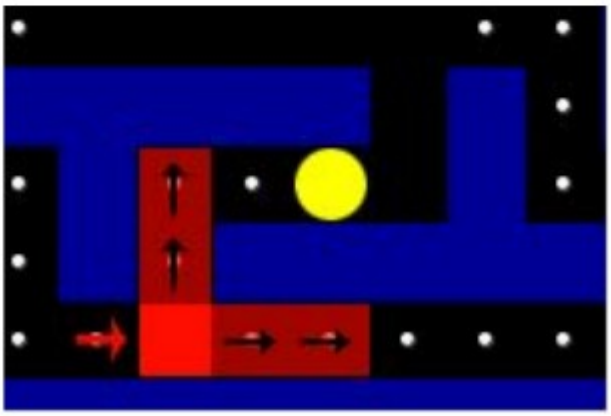
In our this implementation of BFS, we will mark each square as "visited" with the **initial** direction the search took to reach that square. That way, when we finally reach our target, it will be conveniently marked with the optimal direction the ghost should take!



Here is a simplified simulation of our BFS, with Pacman as our target. The ghost starts out moving right.

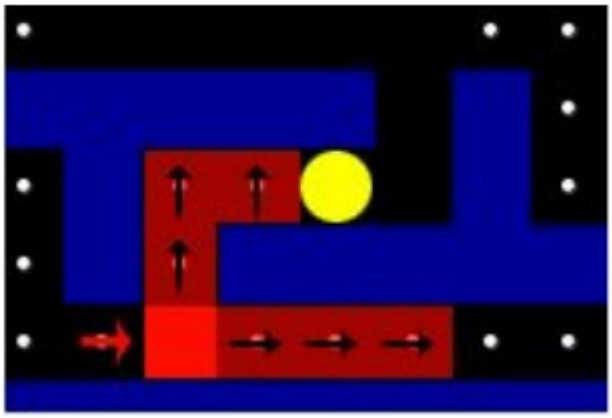


First, we find all the directions in which the ghost can move. Remember that ghosts cannot make a 180 degree turn, nor can they turn into walls. In this case, the ghost can either move up or right. We launch our search in each of these valid directions, marking those squares neighboring the ghost with their respective directions.

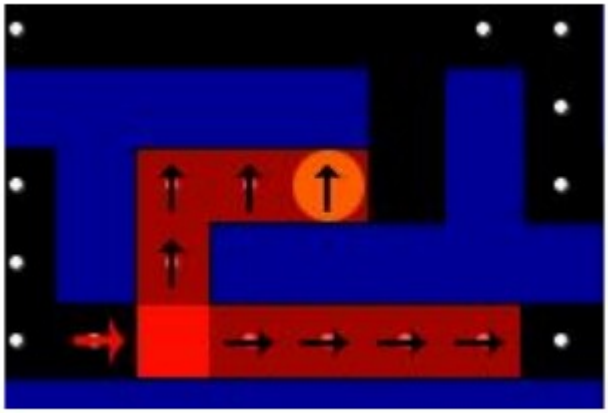


Next, we check the neighbors of the neighbors. The square that was marked as "up" checks all its unmarked neighbors (in this case there is only one) for the target, marking them all with "up."
Likewise the square that was marked "right" checks all of its unmarked neighbors and marks them with "right" to indicate that they have been visited.

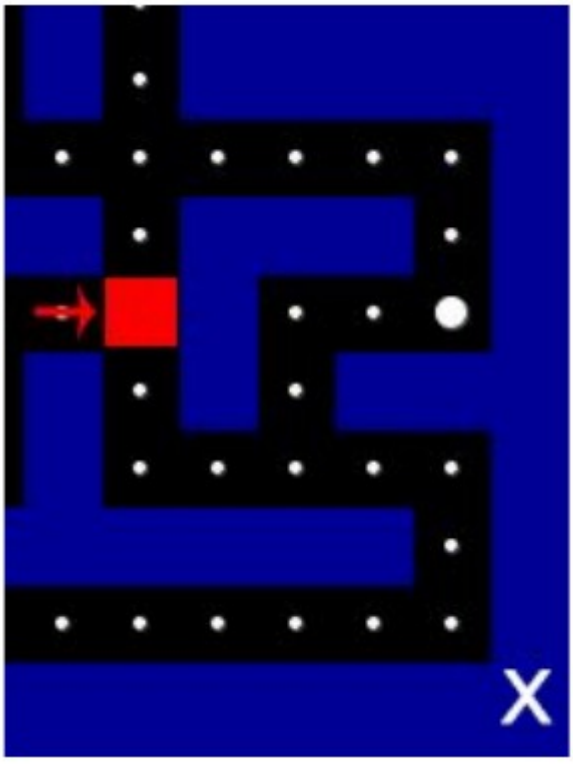
Note that each square has four neighbors, but we only check those to which the ghost can move. That means we don't check squares that are walls. What about making sure the ghosts never turn 180 degrees? Fortunately, we have already accounted for this rule by never checking a square that has already been visited. This makes it impossible for any trail to double back on itself. Please take time to think about this and understand how this works.



We continue expanding our search, looking for the target.
Note that even though the path taken by the initial "up" direction turns to the right, all the squares along the path are still marked "up".



Here, our search has located the target! We've reached the square containing Pacman and marked it with "up", because that's the direction with which its preceding square was marked.
Now we know that moving up will take the ghost to Pacman fastest. If our target was always Pacman, we could be done with this algorithm now...



However, what if the target is not **reachable?** With our targeting scheme, it may be that the target square (e.g. two squares in front of Pacman) is actually a wall, which our current BFS would never reach.

Oh no! What then?!

There is a solution, but before you continue reading, make sure you have a concrete grasp of how the current BFS simulation works.

Okay, ready?

Because we need to account for unreachable targets, we can no longer end the BFS when the target has been found. Rather, we are going to make a "breadth first traversal of the entire map", looking for the open square closest to the target. The process of visiting our neighbors first, then our

neighbors' neighbors, etc. remains exactly the same. We will also use the same technique of marking squares as *"visited"* with directions.

Here's where the search changes: Every time we visit a square, we will compute the distance between that square and the target square. Throughout the entire traversal, we will keep track of the current minimum distance between a square and the target (a simple way is which I discussed in the class). At the end of the traversal, we will return the direction with which the absolute minimum-distance square is marked, which will be the correct direction to take!

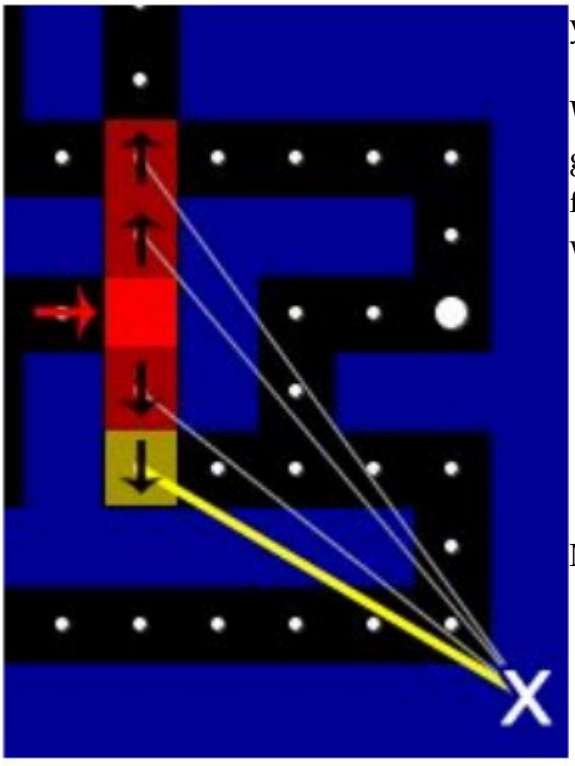
Here is a simulation of our final algorithm. Now our target is off of the board, marked by an X.

Here is a simulation of our final algorithm, with implementation specifics. Now our target is off of the board, marked by an X.

We make a copy of 2D board array to store directions. We will use this array to mark squares as "visited" and allow the neighbors to know what direction was "taken" to reach it. The value of a cell should be NULL when it is unvisited.

During the traversal, each time we dequeue (remove from queue) a square to check its distance to the target, we will mark its neighbors as visited and add them to the end of the queue to be checked later. This ensures that we always visit our neighbors before we visit our neighbors' neighbors, and so on. This approach puts the name "breadth-first" in BFS, so make sure you understand how it works!

Now let's start. After visiting the first two squares, we update the minimum distance to be the length of the line in yellow. The current minimum-distance square is likewise shown in yellow.



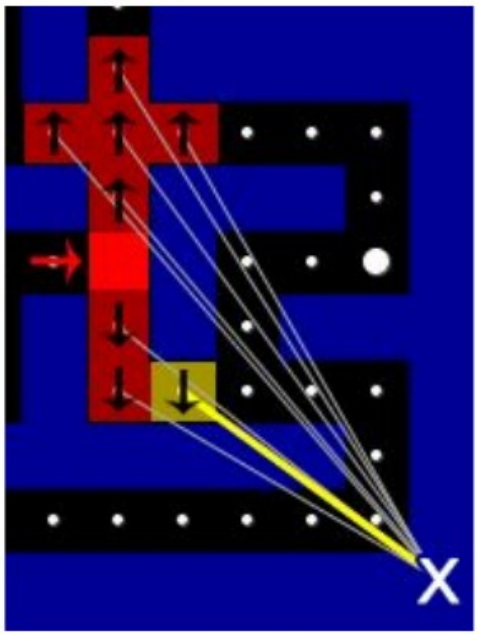
We continue the traversal as before. Remember, to get the next square to examine, we simply dequeue from the queue. Each time we dequeue a square, we do the following:

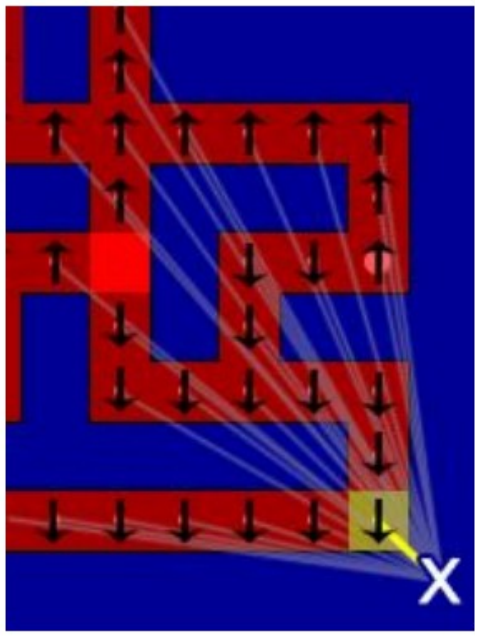
1. Calculate the distance to the target square.
2. Update the minimum distance if necessary.
3. Find the square's unvisited neighbors, mark them with the current square's direction, and stick them at the end of the queue.

Note that all the squares in the queue have already

been marked with a direction. This means that when you dequeue square, it can tell you the direction with which to mark its unvisited neighbors and update the direction we are keeping track of!

Eventually, the entire maze will have been traversed. The 2D array will be fully marked with directions (except for the wall squares), and we will have the absolute minimum distance as well as the direction that is marked in the minimum-distance square. This is the direction we need! In this example, we find that the ghost should move down to get to the target.





High Level Pseudocode

```
Make a 2D array (i.e. make copy of board array)of directions.
for each of the ghost's valid neighboring squares:
   add the square's location (the square's position in the 2D
   array) to the queue and storethe square's direction in the 2D
   array, indexed by its location

while the queue isn't empty:
   dequeue the next square location
   update the shortest distance and board coordinate
   for each valid neighbor of the current square
      if the neighbor has not been visited, add its location to the
      queue and store the current square's direction in the 2D
      array at the neighbor's x-y coordinates.

return the direction of the closest square.
```

Follow the same procedure for other ghosts as well.