Breadth First Search and Depth First Search

Our goal is to start from some vertex **s** in a connected graph G and systematically visit every other vertex in G. One reason to do this may be to look for a particular vertex in G and find a path from your start vertex **s** to the target vertex. There are two main approaches to searching through the graph: breadth-first and depth-first. The two algorithms follow the same outline.

1. Initialize a **seen** set and a **to-visit** list both containing just the starting vertex **s**.
2. While the **to-visit** list is not empty:
   (a) Remove the first vertex **n** from **to-visit**
   (b) For each neighbor vertex of **n**, if it is not in **seen**, add it to **seen** and **to-visit** (Optionally add parent pointers back to **n** if you want to recover path)

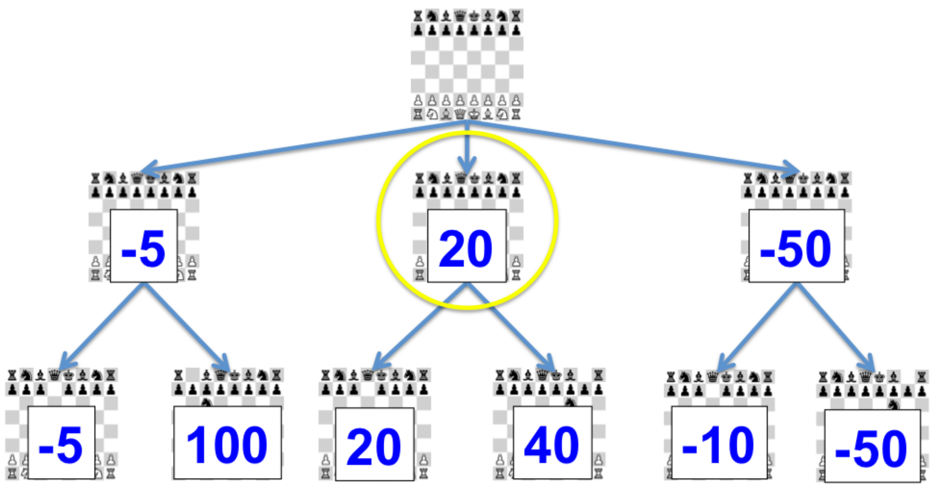
When the **to-visit** list is empty, we will have visited all the vertices in the graph. In addition to visiting every vertex, we also checked every vertex's neighbors, or in other words every edge coming out of each vertex. The total runtime of this search is $O(|V|+|E|)$ since we visit every edge and vertex and the other operations (inserting and removing from seen and to-visit) are constant time and do not affect the asymptotic running time.

The difference between depth first search (DFS) and breadth first search (BFS) is simply where in the **to-visit** list we insert the neighbor vertices (step . In BFS, we want to expand (visit all the neighbors of) all the vertices we have seen already before expanding any newer vertices, so we insert the new vertices to the end of **to-visit**. In DFS, we want to expand vertices that we have just expanded to go deeper into the graph, so we insert the new vertices to the beginning of **to-visit**.

Another way to think about the difference between BFS and DFS is to consider **to-visit** as a stack in DFS and as a queue in BFS. A stack is a data structure where elements are inserted and removed in a first-in-last-out order, like inserting cards on top of a deck and removing cards only off the top of the deck. A queue is a data structure where elements are inserted and removed in a first-in-first-out order, like a ticket line.

BFS and Games: Minimax

A two-player game can be represented as a graph where the game states are represented as vertices and transitions from one game state to another are represented as edges. We can use BFS to find all the game states that can be obtained in up to x moves from some starting state. We can also use some assessment function to quantify how good of a position a game state is for us.



Using BFS, we can figure out all the possible sequences of x moves from some starting position and using our assessment function we can figure out approximately how good of a situation we'll be in after each sequence of x moves. Given this information, the next question is figuring out which move to make from our starting position. Intuitively, one may want to pick the move that leads to the highest value game state. However, you have to keep in mind that the opponent is also looking to to play his best move as well. Thus, what you actually want to do is play to minimize your maximum loss, i.e. play so that you minimize your opponent's gain even if he plays perfectly.

The minimax algorithm follows this outline:

1. From some starting game state **s**, use BFS to find all reachable game states in x moves. The game states reachable in x moves will be at the xth level of the game state tree rooted at **s**.
2. Starting at the $(x-1)$th level and moving up a level each time until you reach the root:
   (a) If this level represents your turn, assign the value of each game state to be the maximum of their children states (your best move is to maximize state value)
   (b) Else, if this level represents the opponent's turn, assign the value of each game state to be the minimum of their children states (opponent's best move is to minimize state value).

By the end of this algorithm, you will have game state values of each of the neighboring game states of **s** that factors in the opponent's optimal moves and your best move is the edge that leads to the highest value neighbor.