Intro to Graphs

Neal Bayya - Based on Samuel Hsiang's Guide

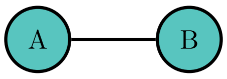
January 5, 2018

1 Introduction

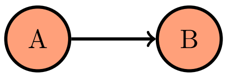
A graph is a set of vertices (V) and edges (E; storing connectivity information between vertices in V). Graphs are used to represent a variety of computer science problems. For example, we can generate a graph with airports and draw lines between each airport to indicate flight paths. This structure can be used to solve problems such as connectivity and shortest path. When using big O notation, instead of N, we use E and V to represent edges and vertices, respectively. We will cover some basic terminology and representations of graphs, as Java and C++ do not provide a library for them.

1.1 Terminology

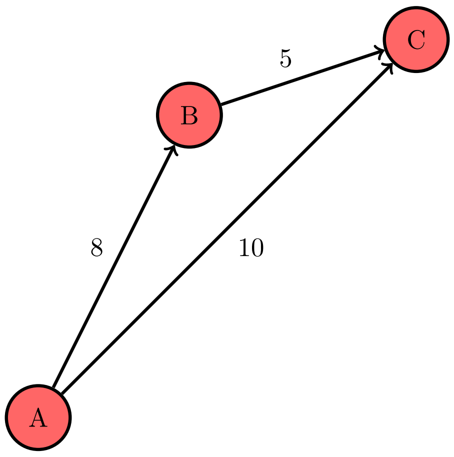
An **undirected graph** is a graph that in which edge connections go both ways. For example, if two cities, A and B, have a railroad connection, then you can travel from A to B or B to A.



A **directed graph**, also known as a digraph, is one in which edges point only one way. If a directed graph has a path that allows two-way movement, then we draw arrows on both end of the line segment.



A **weighted graph** is a graph in which edges are unequal. We assign a weight to each edge representing the cost of traversing it in context of the problem. For example, within the set of vertices: $\{Alexandria, DC, LosAngeles\}$, the edge between Alexandria and DC will be significantly smaller than that of DC and Los Angeles.



1.2 Adjacency Matrix

A static 2D array serves as a connectivity table between vertices: **int adjMat[V][V]**. adjMat[i][j] is a 1 or 0 depending on if there is a connection for unweighted graphs and the edge weight or 0 for weighted graphs. An interesting property of adjacency matrices in unweighted graphs is that if you want to find the number of paths between two vertices of length n, simply raise the matrix to the nth power. For example, for depth 2 paths, matrix multiply the adjacency matrix by itself. The space complexity of storing an adjacency matrix is O(V^2), meaning it is acceptable for small and dense graphs, but impractical for large and sparse ones.

1.3 Adjacency List

A list of lists of integer pairs: **ArrayList<ArrayList<Integer Pair» adjList**. Each element in this structure is a list of integer pairs that represent neighboring elements. The first element of the integer pair is the neighbor's index, and the second element is the edge weight. If the graph is unweighted, you only need one integer, as opposed to a pair. You may additionally need to hash given vertex names to indices.
Alternatively, you can create a class Vertex, in which you store two parallel lists: **ArrayList<Vertex> neighbors, and ArrayList<Integer> weights**. In this representation, weight.get(i) should represent the weight between neigbors.get(i) and the current vertex.
In terms of space complexity, this representation is O(V+E) because we are no longer storing non-existant edge connections.

1.4 Edge List

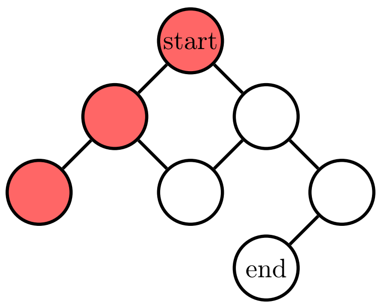
A list that stores an integer triple: **ArrayList<Integer Triple> edgeList**. Each element in the list is a triplet in the form: (vertex 1 index, vertex 2 index, edge weight). The space complexity of this representation is O(E).

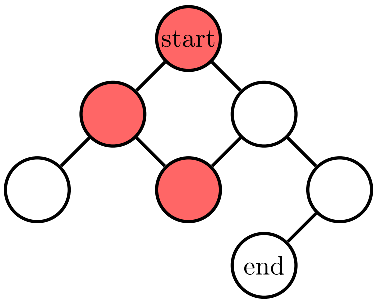
2 Graph Traversals

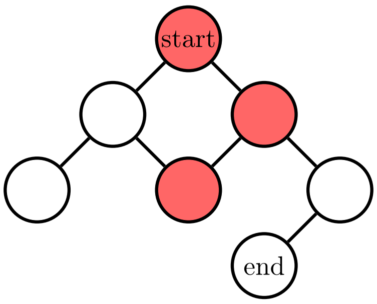
In an unweighted graph, you are given a starting vertex and are asked to find a path to some vertex *end*. DFS and BFS are two approaches to finding such a path. These two search techniques eliminate the possibility of cycles by maintaining a list of already processed vertices (seen), so the diagrams you will see in the followup sections are simplified graphs (trees).

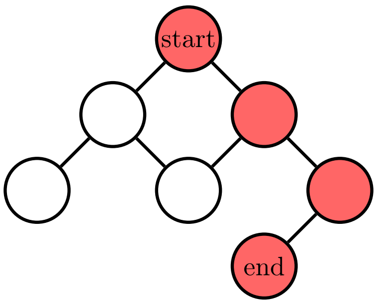
2.1 Depth-First Search (DFS)

The idea behind depth first search is to recursively search each neighbor until a solution is found. Essentially, we fully explore one neighbor before moving onto the next one. The time complexity of DFS is O($V + E$), because in the worst case, we will have to traverse everything.





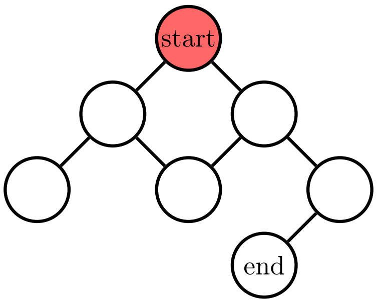


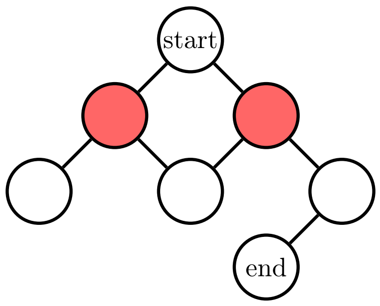


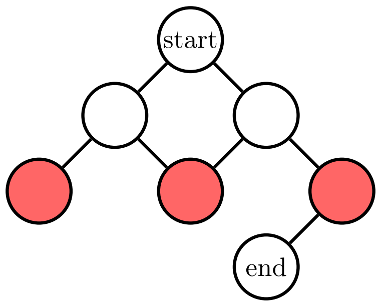
2.2 Breadth-First Search (BFS)

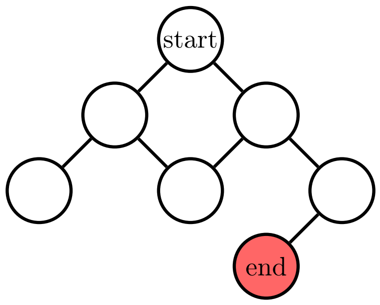
The idea behind breadth-first search is to search the space level by level. Instead of recursion (which was used in DFS), the first-in-last-out property of a queue is utilized to store and process the vertices. First, we visit the neighbors of the source vertex, then the neighbors of the neighbors, and so on. BFS starts by inserting the start vertex into a queue. Then, while the queue is not empty, BFS pops vertex v from the queue, marks it as visited, and pushes all unvisited neighbors of v to the queue.

Since we are not skipping any vertices before moving to a subsequent level, BFS is guaranteed to find the shortest path (which DFS does not guarantee). The time complexity of BFS is $O(V + E)$. The use of the queue makes BFS worse than DFS in terms of space complexity, as in DFS, you only have to keep track of one path at a time.









3 Problems

- (What's Up With Gravity, USACO Silver 2013) Captain Bovidian is on an adventure to rescue her crew member, Doctor Beefalo in a two dimensional N by M grid (1 <= N, M <= 500). Some grid cells are empty while others are blocked and cannot be traversed.

  Captain Bovidian must obey the following rules of physics while traversing her world.

  1) If there is no cell directly underneath Captain Bovidian (that is, if she is at the edge of the grid), then she flies out into space and fails her mission.
  2) If the cell directly underneath Captain Bovidian is empty, then she falls into that cell.
  3) Otherwise: a) Captain Bovidian may move left or right if the corresponding cell exists and is empty. b) Or, Captain Bovidian may flip the direction of gravity.

  When Captain Bovidian changes the direction of gravity, the board is flipped vertically and she falls to what used to be the top. Doctor Beefalo is lost somewhere in this world. Help Captain Bovidian arrive at her cell using the least number of gravity flips as possible. If it is impossible to reach Doctor Beefalo, please output -1.

- (UVa 11902, Dominator) A vertex X dominates a vertex Y if all paths to Y must go through X. If Y is not reachable from the start node then node Y does not have any dominator. By definition, every node reachable from the start node dominates itself. You are given a graph adjacency matrix, and have to find the dominators of every node where the 0-th node is the start node.

- January 2015 Contest - Silver (Problems 2 and 3)