Graph Theory

Richard Zhan

October 2019

1 What is a Graph?

A graph is a set of nodes that are connected together by edges. 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, which are nodes, respectively.

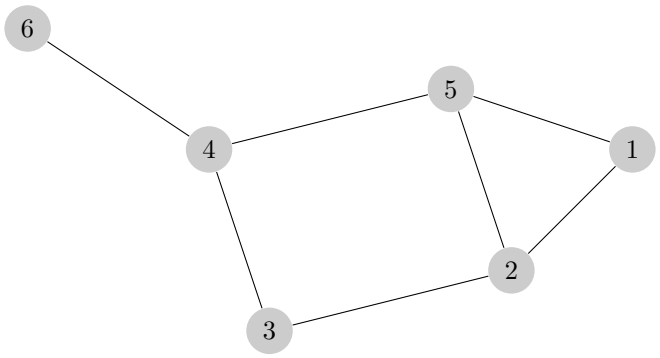


Figure 1: Example of a graph

1.1 Undirected Graphs

Undirected graphs consist of a series of nodes with line segments in between them. These line segments, called edges, determine which nodes are connected. In an undirected graph, if vertices u and v are connected by an edge, then a traversal can be made in both ways. In other words, one can go from u to v and from v to u. When a problem statements says "graphs with bidirectional edges" or something of that sort, you can interpret it as an undirected graph because bidirected graphs and undirected graphs basically have same properties.

1.2 Directed Graphs

Certain graphs only allow for one way movement. We call these graphs directed graphs. Typically, when we draw a graph, we place all of the nodes and then draw line segments in between the points. In a directed graph, these line segments are turned into rays that represent movement. If a directed graph has a path that allows two-way movement, then we draw arrows on both end of the line segment.

1.3 Weighted Graphs

In most cases, we often give some weight to each edge. This weight represents the value associated with it. For example, in the case of creating an airplane network, we know that going from DC to LA is much further than DC to NYC. Therefore, we can give each line segment a different weight, such as the distance, to represent the value of each edge.

2 Searches

Often, we want to navigate through these graphs to look for a specific thing. To do this, we have two primary approaches: breadth first search (BFS) and depth first search (DFS).

2.1 BFS

The first approach we use is searching by breadth. At each node, we recursively call each of its neighbors, expanding outward. This way, our region of inspection expands our circularly and fairly evenly. Essentially, its as if at each fork in the graph, we clone ourselves and one clone goes down each direction. This is very useful expanding out steadily and making sure we don't skip over regions.

2.2 DFS

Another approach that is used is by searching by depth. At each node, we simply pick one direction and keep following this path all the way down till we hit a base case. Then we trek back to the last fork in the road and take that path till we hit a base case. This continues till the objective is reached or the entire graph is traversed. In some cases, this can be more effective, however it risks the fallacy of skipping over potentially very rewarding nodes that are very close simply because they aren't explored early on.

3 Shortest Path

We want to find the shortest path between any two points. We consider that each edge has some weight or penalty to traverse. This weight represents the cost needed to traverse the connection. This graph can be either directed or undirected.

3.1 Dijkstra

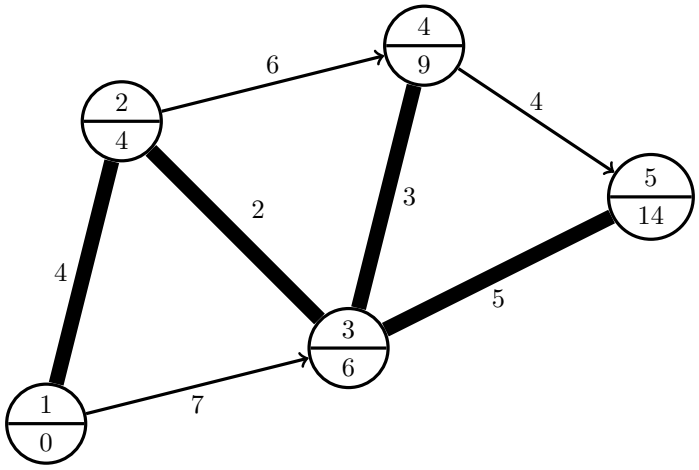
Dijkstra's algorithm is a common approach to finding the shortest path between any two points. This is done through a greedy approach.

For every vertex v in the graph, we keep track of the shortest known distance $dist(v)$ from the source to v, a boolean $visited(v)$ to keep track of which nodes we "visited," and a pointer to the previous node in the shortest known path $prev(v)$ so that we can trace the shortest path once the algorithm finishes.

Dijkstra iteratively "visits" the next nearest vertex, updating the distances to that vertex's neighbors if necessary. The nearest vertex is simply the one that is closest to what we have already explored, aka has the lowest weight on its edge.

Algorithm 1 Dijkstra

```
for all vertices v do
    dist(v) ← ∞
    visited(v) ← 0
    prev(v) ← −1
dist(src) ← 0
while ∃v s.t. visited(v) = 0 do
    v ≡ v s.t. visited(v) = 0 with min dist(v)
    visited(v) ← 1
    for all neighbors u of v do
        if visited(u) = 0 then
            alt ← dist(v) + weight(v, u)
            if alt < dist(u) then
                dist(u) ← alt
                prev(u) ← v
```



3.2 Bellman-Ford

Bellman-Ford is a single-source $O(VE)$ shortest path algorithm that works when edge weights can be negative. The key observation here is that the shortest path, assuming no negative cycles, has length at most $V-1$. If we loop over all edges and consider if they form a new shortest path to their endpoint we can find all shortest paths of length 1. If we repeat this loop $V-1$ times we discover all paths of length up to $V-1$. To detect a negative cycle we simply run the loop one last time. If any shorter paths are discovered, then the optimal path is longer than $V-1$ nodes. The only way this is possible is if there is a negative cycle.

Algorithm 2 Bellman-Ford

```
for all vertices v do
    dist(v) ← ∞
    prev(v) ← −1
dist(src) ← 0
for i = 1 to V − 1 do
    for all edges (u, v) do
        if dist(u) + weight(u, v) < dist(v) then
            dist(v) ← dist(u) + weight(u, v)
            prev(v) ← u
for all edges (u, v) do                          ▷ check for negative cycles
    if dist(u) + weight(u, v) < dist(v) then
        negative cycle detected
```

3.3 Floyd-Warshall

Floyd-Warshall solves the shortest path problem for all pairs of vertices in $O(V^3)$ time, which is faster than V single-source Dijkstra runs on a dense graph. Floyd-Warshall works even if some edge weights are negative but not if the graph has a negative cycle. Essentially, it computes the path from every node to every node.

Algorithm 3 Floyd-Warshall

```
for all vertices v do
    dist(v, v) = 0
for all edges (u, v) do
    dist(u, v) = weight(u, v)
for all vertices k do
    for all vertices i do
        for all vertices j do
            if dist(i, j) > dist(i, k) + dist(k, j) then
                dist(i, j) ← dist(i, k) + dist(k, j)
```