Graph Representations

Let $G = (V, E)$ with $|V| = n$, $|E| = m$. In other words, for some graph $G$, it contains $n$ vertices and $m$ edges.

**Adjacency Matrix**

One way to represent $G$ is with an $n \times n$ matrix $A$ where $A[i][j] = 1$ if there is an edge from vertex $i$ to vertex $j$ and 0 otherwise. The primary advantage of this approach is that you can check whether or not there is an edge connecting two vertices in $O(1)$ time. The disadvantage, however, is that this representation takes up $O(n^2)$ space. When $n$ is large, this might become untenable.

Two things worth noting:

- If $G$ is undirected, then its adjacency matrix is symmetric. That is, flipping the matrix along its main diagonal will produce the same matrix.
- Entries along the diagonal of an adjacency matrix (technically representing the presence of edges from vertices to themselves) are 0 by convention, as our graphs are simple. Non-simple graphs have self-loops, where vertices contain edges to themselves (these will not be dealt with in this course).

**Adjacency List**

Another way to represent $G$ is to use an adjacency list. Each vertex $u$ is associated to a list $\text{neighbors}(v)$ which contains the nodes $v$ such that $(u, v) \in E$. The advantage of this representation is that we use less space, $O(n + m)$, which is better than $O(n^2)$ of adjacency matrices as long as $m \ll n^2$. The disadvantage, though, is that checking whether $(u, v) \in E$ takes (potentially) linear time.

Graph Traversals

We now look at two ways to traverse a graph.

**BFS (Breadth First Search)**

In BFS, we begin at a node $v$ (level 0) and explore the graph in “layers.” First we would explore all children of $v$ (level 1), then the children of the nodes in level 1 (these would make up level 2), etc. The key point here is that we explore all nodes at level $i$ before exploring any nodes at level $i + 1$. The output of BFS is called a BFS tree. We typically use a queue to implement this algorithm. For implementation details, see [https://en.wikipedia.org/wiki/Breadth-first_search](https://en.wikipedia.org/wiki/Breadth-first_search).

The running time of BFS is $O(n + m)$, because each vertex is added and removed from the queue once and, in the worst case, we need to traverse every edge to visit each node.

**DFS (Depth First Search)**

In DFS, we begin at a node $v$ and examine its neighbors. As soon as we encounter a neighbor that hasn’t been visited, visit it. Once we arrive at a node for which all of its neighbors have been visited, we “backtrack” until we reach a node that has still unvisited neighbors (in the form of returning from recursive visit calls). We typically use a stack. There is also a recursive method to implement this algorithm. Please see both implementation methods in the link below.
The running time analysis for DFS is similar to that of BFS, giving a running time of $O(n + m)$.

### Dijkstra’s Algorithm

**Definition 1** (Greedy algorithm). A greedy algorithm is one which always makes the choice that looks best at the moment—the *locally optimal* choice—in order to find the best *globally optimal* solution. Greedy algorithms do not always yield optimal solutions, but for many problems they do.

**Definition 2** (Shortest path). A shortest path from vertex $s$ to vertex $t$ is a directed path from $s$ to $t$ with the property that no other such path has a lower total edge weight.

### Overview

Dijkstra’s algorithm finds the shortest path between two given vertices in a weighted graph, assuming that the graph’s edge weights are non-negative. The running time of the algorithm is $O((E + V) \log V)$ when the graph is implemented using adjacency lists. With a special transformation (use of Fibonacci heaps) this can be reduced to $O(E + V \log V)$, which is the fastest version of this algorithm. The pseudo-code for the algorithm is given below.

### Pseudocode

**Dijkstra**($G, s$)

1. for each vertex $v \in V_G$
2. 
3. 
4. $dist[v] = \infty$
5. $parent[v] = \text{NIL}$
6. $dist[s] = 0$
7. $Q = V_G$
8. while $Q \neq \emptyset$
9. 
10. $u = \text{EXTRACT-MIN}(Q)$
11. for each vertex $v \in G.\text{Adj}[u]$
12. 
13. if $dist[v] > dist[u] + w(u, v)$
14. 
15. $dist[v] = dist[u] + w(u, v)$
16. 
17. $parent[v] = u$

### Runtime

**Question:** Given this pseudocode, explain the running time of Dijkstra’s algorithm?

The running time of Dijkstra’s algorithm has two components, $E \log V$ and $V \log V$. Let us first consider the $V \log V$ term: this component derives from the maximum size ($V$) of the heap used to store vertices, and the running time of heap operations such as INSERT and REMOVE_MIN is $O(\log V)$.

The $E \log V$ term has to do with the relaxation step of Dijkstra’s algorithm. Each edge examined may result in a relaxation of the neighboring node in the heap; in other words, an update key operation that is $O(\log V)$. We know that the number of vertices examined in line 8 above is bounded by the total degree of all vertices, as each vertex is added and popped exactly once from the min-heap. This value is $2|E|$ by the Handshake lemma, so in the worst case we have $2|E|$ decrease-key operations, for a total of $O(E \log V)$.

This bound is good for easily proving our run-time, but it is not tight. Each edge $(u, v)$ can only cause one relaxation, not two as the handshake lemma suggests. This is because $(u, v)$ is explored only when node $u$ is popped from the min-heap. This means that when $(u, v)$ is explored from node $v$ node $u$ has already been removed, so it’s key cannot be decreased.
Graph Traversal Questions

Problem 1. Design an algorithm to determine whether or not a graph has a cycle.

Solution. You can perform a BFS or DFS and just keep track of which elements have been seen. For example, you can run a DFS and store vertices you have seen in a set and just track whether or not any previously seen node is encountered again by checking if it is in the set. Since we are simply doing a BFS or DFS, this algorithm runs in linear time.

Problem 2. Design an algorithm to determine whether or not a connected graph has a cycle in $O(n)$ time.

Solution. Perform the same algorithm as problem 1. However, terminate early if you explore at least $n$ edges. Recall that a tree has exactly $n - 1$ edges. An additional edge would signify that two nodes are connected by two independent paths. Thus, there is a cycle in the graph. This algorithm will take $O(n)$ time to check each vertex and $O(n)$ to check each edge (since we are checking at most $n$ edges). Thus, the running time is $O(n)$.

Note that a graph with $n$ edges must have a cycle regardless of whether it’s connected. This example is just simple to prove with BFS.

Problem 3. Design an algorithm to find the shortest path between nodes $u$ and $v$ in a connected, unweighted graph.

Solution. Since the graph is unweighted, we can just run BFS starting from $u$ and for each node $x$ that we visit, we just keep a pointer to its parent node (the node we visited $x$ from). When we reach $v$, we stop and find the shortest path by backtracking through the pointers that we kept (i.e. we could see that $v$’s parent was $d$, $d$’s parent was $c$, and $c$’s parent was $u$, so our path would be $u \rightarrow c \rightarrow d \rightarrow v$). We just do a BFS and backtrack no more than $O(n)$ times (the longest path in a graph is $n - 1$ edges), so this algorithm also runs in linear time.

Dijkstra’s Questions

Problem 4. Find the shortest path between vertices $E$ and $G$. 
Solution. Dijkstra’s algorithm produces the following state:

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance from $E$</th>
<th>Parent node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>17</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>22</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>13</td>
<td>I</td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>F</td>
</tr>
<tr>
<td>J</td>
<td>20</td>
<td>I</td>
</tr>
</tbody>
</table>

We can use the mapping from nodes to parent nodes to find the shortest path from $E$ to $G$, which is $E \rightarrow B \rightarrow F \rightarrow I \rightarrow H \rightarrow G$.  

**Problem 5.** Explain why Dijkstra’s algorithm is a *greedy algorithm*.

*Solution.* A greedy algorithm makes the best choice that is currently available. Dijkstra’s algorithm follows this paradigm by using a priority queue structure that, when polled, always produces the node with the shortest distance from the source node.

**Problem 6.** Does Dijkstra’s Algorithm work with negative weights? Why or why not?

*Solution.* No, Dijkstra’s Algorithm will not work on negative weighted graphs. First, if there exists a negative cycle, the concept of shortest path does not exist.
Secondly, a negative weight breaks an important assumption in the canonical proof of correctness for Dijkstra’s algorithm.

Proof (adapted from CLRS). Induct on the size of the shortest path tree $S$ with source $s$. Assume that Dijkstra’s algorithm correctly computes the shortest path for a tree of size $|S| = k$, for some $k \geq 1$. We must show that if $u$ is the $k+1$-st vertex brought into $S$, then $\text{dist}[u]$ is the weight of the shortest path from $s$ to $u$. Let $p$ be a shortest path from $s$ to $u$. Let $y$ be the first vertex along $p$ such that $y \in V - S$, and let $x \in S$ be the predecessor of $y$. Path $p$ can be deconstructed as $s \rightarrow x \rightarrow y \rightarrow u$. Let $\delta(\cdot, \cdot)$ represent the actual shortest path distance between two vertices. Because $y$ appears before $u$ and all edge-weights are non-negative, $\text{dist}[y] = \delta(s, y) \leq \delta(s, u) \leq \text{dist}[u]$. But since both $u$ and $y$ were in $V - S$ when $u$ was taken off of the priority queue, it must be that $\text{dist}[u] \leq \text{dist}[y]$. So $u$ is in fact the vertex with its distance estimate $\text{dist}[u]$ exactly equal to the shortest path distance $\delta(s, u)$.

Problem 7. True or false: Dijkstra’s algorithm will not terminate if run on a graph with negative edge weights.

Solution. False. The algorithm will terminate, but it will return a wrong answer.

Problem 8. True or false: If we double the weights of all the edges in a graph, then Dijkstra’s algorithm will produce the same shortest path.

Solution. True. Any linear transformation on the weights will not affect the calculation of shortest paths. You can think of it as unit-conversion. For instance, if you converted weights from expression in miles to kilometers, that would not affect the relative ordering of shortest paths.