Definitions

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.

Dijkstra’s Algorithm

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 $O(E \log V + V \log V)$ 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

\[
\text{Dijkstra}(G, s)
\]

1. for each vertex $v \in V_G$
2. \hspace{1cm} $\text{dist}[v] = \infty$
3. \hspace{1cm} $\text{parent}[v] = \text{NIL}$
4. \hspace{1cm} $\text{dist}[s] = 0$
5. \hspace{1cm} $Q = V_G$
6. while $Q \neq \emptyset$
7. \hspace{1cm} $u = \text{Extract-Min}(Q)$
8. \hspace{1cm} for each vertex $v \in G.\text{Adj}[u]$
9. \hspace{2cm} if $\text{dist}[v] > \text{dist}[u] + w(u, v)$
10. \hspace{2cm} \hspace{1cm} $\text{dist}[v] = \text{dist}[u] + w(u, v)$
11. \hspace{2cm} \hspace{1cm} $\text{parent}[v] = u$

Edge-Weighted DAGs (Directed Acyclic Graphs)

The algorithm for shortest path on edge weighted DAGs is simpler and faster than Dijkstra’s algorithm. However, instead of considering vertices by priority of their distance estimates, we consider the vertices of the DAG in a topological order. (Why must a DAG always have a topological order?) Then we just relax each vertex in the topological ordering. Running time: $O(|V| + |E|)$. 
Dijkstra Questions

Problem 1. Find the shortest path between vertices E and G in the graph provided

Solution. Dijkstra’s algorithm produces the following state:

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

<table>
<thead>
<tr>
<th>Node</th>
<th>Parent node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>NULL</td>
</tr>
<tr>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>I</td>
<td>F</td>
</tr>
<tr>
<td>J</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 2. 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 3. 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 $dist[u]$ 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, $dist[y] = \delta(s, y) \leq \delta(s, u) \leq dist[u]$. But since both $u$ and $y$ were in $V - S$ when $u$ was taken off of the priority queue, it must be that $dist[u] \leq dist[y]$. So $u$ is in fact the vertex with its distance estimate $dist[u]$ exactly equal to the shortest path distance $\delta(s, u)$.

Problem 4. 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 5. True or false: The shortest path algorithm in an edge weighted DAG works even with negative edge weights.

Solution. True. First, because we are considering a DAG, we do not have to worry about negative weight cycles. Also notice that because we consider vertices in topological order, no ancestor of $v$ will be relaxed after $v$ itself is relaxed.

Problem 6. How could you modify Dijkstra’s algorithm to find all shortest paths?

Solution. Dijkstra’s algorithm produces the shortest paths to all nodes in the graph from a single source. In order to find all shortest paths (i.e., the shortest path between any pair of nodes in the graph), you can simply run Dijkstra’s from each node in the graph, for a resulting running time of $O(V(|E| + |V|) \log V)$.

Problem 7. How could you modify Dijkstra’s algorithm to stop once it’s found the shortest path to a particular node?

Solution. Dijkstra’s algorithm produces the shortest paths to all nodes in the graph from a single source. If you are only interested in finding the shortest path from $s$ to $t$, you can stop the algorithm once $t$ is removed from the priority queue.

Problem 8. Explain the running time of Dijkstra’s algorithm.

Solution. 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 REMOVEMIN 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 $O(\log V)$.

Problem 9. 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 scalar multiplication on edge 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.
**Problem 10.** Say we are given a graph $G$ where all edges are positively weighted. Construct graph $G'$ where for all edges $e$ with weight $w(e)$ and endpoints $u$ and $v$, we replace $e$ with $w(e)$ edges of weight 1 in $G'$, such that the path from $u$ to $v$ in $G'$ consists of $w(e) - 1$ middle nodes.

How could you use this method to find the shortest path between two vertices in $G$? What problem do you see with this approach?

**Solution.** Say we are trying to find the shortest path in $G$ between two vertices $x$ and $y$. Perform BFS in $G'$ starting at $x$ and stopping once we see $y$.

For graphs with large edge weights, this approach takes much longer than using Dijkstra

---

**Topological Sort**

A few weeks ago we covered an algorithm called topological sort. This is motivated by many problems encountered in the real world. For example, you are running an assembly line where there are a number of tasks required to create a product. Some of the tasks must come before others. You want to maximize the amount of parallel tasks you can complete at once. How can you obtain an ordering of these tasks to make sure the product is assembled properly? The answer is a topological sort!

**Definition 3** (Topological ordering). A topological ordering of a directed acyclic graph $G = (V,E)$ is a linear ordering of $V$ such that whenever $G$ contains a directed edge $(u,v)$, then $u$ appears before $v$ in the ordering.

There are two canonical algorithms for this. It is good for you to understand both of them.

**Using depth-first search (Tarjan’s algorithm)**

- Call DFS and compute finish times for each vertex $v$.
- As each vertex finishes, push each onto a stack.
- Return the stack.

From most recently pushed to the eldest element, the stack contains the nodes in order of decreasing finishing times.

This is equivalent to a reverse postorder traversal.

You should think carefully about the correctness of this algorithm!

**Kahn’s algorithm**

- Maintain a set $S$ of nodes with in-degree 0.
- While $S$ is not empty, remove a node from $S$ and add to the end of ordering.
- Remove all edges going out of that node and update $S$ accordingly.

This is perhaps the more intuitive algorithm based on your understanding of topo sort.

---

**Problems**

**Problem 1**

Conceptual questions:

1. (True/False) Every DAG has exactly one topological ordering.

   **Solution.** False. Take for example, a graph with no edges. Any ordering is valid.

2. (True/False) A preorder traversal always produces a topological ordering on a tree.
Solution. True. Recall that in a tree, every vertex has exactly one path to every other vertex.

3. If a graph has a topological ordering, then a depth-first traversal of the same graph will not see any back edges.

Solution. True. If we are able to topologically sort it, then it must be a DAG. That means there will not be any back edges.

Problem 2

Problem (CLRS 22.4-2). Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V,E)$ and two vertices $s$ and $t$, and returns the number of simple paths from $s$ to $t$ in $G$. You only need count the simple paths, not list them. (An example can be found in the textbook.)

Solution. First, we see it is a DAG, so we should immediately think of topological sort. In this case, they ask for linear time, so we know that asymptotically this is fine. We can now reason about the graph in a more reasonable way.

We make the observation that the number of paths from $s$ to $t$ can be counted by using intermediate nodes. For each $u$ that has an edge $e = (u,t)$, we can count the paths to $t$ as the sum of the number of paths to each of the $u$ nodes. We know this because we ended up at each of those $u$ nodes by some number of paths, then took the last edge $e$ to get to $t$. Therefore, we only need to consider how we got to $u$.

From this observation, we can now build an algorithm.

```
function pathCount(G):
    Topologically sort the vertices, $v_1 \ldots v_n$
    return pathCount(v_n, 0)

function pathCountHelper(v, accumulator):
    for incoming edge $e = (u, v)$
        accumulator += pathCountHelper(u, accumulator)
    return accumulator
```

Now we look at this algorithm and you should be able to reason that the running time is not optimal! We are doing a lot of overlapping work on the recursive call. It seems very likely that we will be rerunning the same recursive call multiple times (for all nodes that have edges from that node), so let’s try to eliminate doing that work again.

```
function pathCount(G):
    Topologically sort the vertices, $v_1 \ldots v_n$
    arr = new array of size n
    arr[0] = 1
    for each $i$ from 1 to n-1
        for each $e = (v_k, v_i)$
            arr[i] += arr[k]
```

This works going from ‘left to right’ on the topological ordering, counting up the paths based on the observation we made. Note that you can also go the other direction—can you figure that out?