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

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

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 \text{INSERT} and \text{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.
Example

Trace through Dijkstra’s on this graph, starting at A.

![Graph Image]

Dijkstra’s algorithm produces the following state:

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance from A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Parent node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NULL</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
</tr>
</tbody>
</table>

Timestamped DFS

Recall the DFS graph traversal algorithm from last week’s recitation. This week, we will extend this algorithm to include timestamps for every vertex. Each vertex $v$ now has two timestamps. The first timestamp, $v.d$, records when $v$ is first discovered in the search. The second timestamp, $v.f$, records when $v$ is finished, that is, when its adjacency list has been examined completely. Below is the pseudocode for a timestamped algorithm built upon the recursive version of DFS. Note that this timestamp modification can also be applied to the iterative version of DFS (you will have to do this in Homework 8).

DFS($G$)
1 \hspace{1em} \text{time} = 0
2 \hspace{1em} \text{for each vertex } v \in G.V
3 \hspace{2em} v.visited = false
4 \hspace{1em} \text{for each vertex } v \in G.V
5 \hspace{2em} \text{if } v.visited == false
6 \hspace{3em} DFS-VISIT(v)

DFS-VISIT($v$)
1 \hspace{1em} \text{time} = \text{time} + 1
2 \hspace{1em} v.d = \text{time}
3 \hspace{1em} v.visited = true
4 \hspace{1em} \text{for each vertex } w \in G.Adj(v)
5 \hspace{2em} \text{if } w.visited == false
6 \hspace{3em} DFS-VISIT(w)
7 \hspace{1em} \text{time} = \text{time} + 1
8 \hspace{1em} v.f = \text{time}
**Strongly Connected Components**

**Definition 3** (Strongly connected component). Given a directed graph \( G = (V, E) \), a *strongly connected component* (SCC) is a maximal set \( S \subseteq V \) such that for all \( u, v \in S \), there exists a path \( u \leadsto v \) and a path \( v \leadsto u \).

Note: We consider only directed graphs here because in undirected graphs, every connected component is trivially strongly connected.

**Definition 4** (Transpose graph). For a graph \( G = (V, E) \), the transpose of \( G \) is defined to be \( G^T = (V, E^T) \), where \( E^T = \{(u, v) : (v, u) \in E\} \). That is, \( E^T \) consists of the edges of \( G \) with their directions reversed. Given an adjacency-list representation of \( G \), the time to create \( G^T \) is \( O(V + E) \).

**Definition 5** (Component graph (kernel graph)). The *strongly connected component graph* of a directed graph \( G \) is the directed graph \( G^{SCC} = (V^{SCC}, E^{SCC}) \) where each vertex of \( V^{SCC} \) represents a strongly connected component of \( G \), and the new edges \( E^{SCC} \) consist of the directed edges between the SCCs of \( G \).

Formally, the definition of the component graph is as follows: suppose that \( G \) has strongly connected components, \( C_1, C_2, ..., C_k \). The vertex set \( V^{SCC} \) is \( \{v_1, v_2, ..., v_k\} \), and it contains a vertex \( v_i \) for each strongly connected component \( C_i \) of \( G \). There is an edge \( (v_i, v_j) \in E^{SCC} \) if \( G \) contains a directed edge \( (x, y) \) for some \( x \in C_i \) and some \( y \in C_j \). Looked at another way, by contracting all edges whose incident vertices are within the same strongly connected component of \( G \), the resulting graph is \( G^{SCC} \). Figure 2(c) shows the component graph of the graph in Figure 2(a). In other words, we can contract every edge whose incident vertices are in the same SCC to produce the component graph.

Notice that the component graph is a directed, acyclic graph. This is useful because we can now we can topologically order its vertices. This idea is crucial to many linear time graph algorithms.

**Kosaraju’s algorithm**

**Description**

Kosaraju’s is a linear time algorithm for finding the strongly connected components of a graph. The algorithm is described below:
KOSARAJU’S\((G)\)

1. call DFS\((G)\) to compute the finishing times \(u.f\) for each vertex \(u\)
2. compute \(G^T\)
3. call DFS\((G^T)\), but in the main loop of DFS consider the vertices in order of decreasing \(u.f\) as computed in line 1
4. output the vertices of each tree in the depth-first forest formed in line 4 as a separate strongly connected component

Notes: We call DFS on an arbitrary initial node in line 1; which node we pick does not affect the correctness of the algorithm. In line 3, we would initially call DFS on the node with the highest finish time, and whenever we must restart our call to DFS we start on the unvisited node which has the highest finish time out of all unvisited nodes.

Running time

DFS takes \(O(n + m)\) time. We perform it twice, for a total of \(O(2(n + m)) = O(n + m)\). Computing \(G^T\) requires simply iterating over \(G\)’s adjacency list once, which takes \(O(n + m)\) time. Thus, the total running time is \(O(n + m)\).

Correctness sketch

In the DFS of \(G\), after we visit a node \(x\), we visit its SCC \(C\) and some edges out of \(C\). We observe that if there is a path from \(x \rightsquigarrow u\) in \(G\), then \(u\) and \(x\) are strongly connected only if there is also a path from \(x \rightsquigarrow u\) in \(G^T\). Because \(G\) and \(G^T\) have the same strongly connected components, there will be a path in \(G^T\) from \(x\) to every vertex in \(C\) but the edges out of \(C\) will have been reversed and they will not be followed before the algorithm finishes processing \(C\). When \(C\) is finished, the part of the DFS starting from the vertex with the next highest finish time will, by logic similar to the above, only reach vertices in its own component. Continuing to apply this logic, we see that the output of the algorithm is a forest of DFS trees, each of which is a strongly connected component of \(G\).
Figure 2: (a) represents the run of DFS($G, c$) with the start and end times labeled. The SCCs are shaded. The edges traversed in the DFS call are highlighted in grey. (b) is the transpose $G, G^T$, with the vertices of highest end time in (a) shaded in dark grey. The SCCs are also shaded. The edges traversed in the DFS call are highlighted in grey. (c) represents $G^{SCC}$, which is the strongly connected component graph of $G$.

**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 6** (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. Both of these algorithms are $O(|V| + |E|)$. Tarjan’s naturally follows this, while it takes more thought to see why this holds true for Kahn’s.

Problems

Topological Sorting

Problem 1. Conceptual questions:
1. (True/False) Every DAG has exactly one topological ordering.
2. (True/False) A preorder traversal always produces a topological ordering on a tree.
3. If a graph has a topological ordering, then a depth-first traversal of the same graph will not see any back edges.

Almost Strongly Connected

Problem 2. Consider a graph $G = (V, E)$ 'almost strongly connected' if adding a single edge could make the entire graph strongly connected. Design an algorithm to determine whether a graph is almost strongly connected.

Dijkstra’s Questions

Problem 3. Does Dijkstra’s Algorithm work with negative weights? Why or why not?

Problem 4. True or false: Dijkstra’s algorithm will not terminate if run on a graph with negative edge weights.

Problem 5. 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.

Additional Problems

Conceptual SCC Questions

Problem 6.
1. (True/False) The finish times of all vertices in a SCC $s$ must be greater than the finish times of other SCCs reachable from $s$ during the first DFS.
2. How does the number of SCCs of a graph change if a new edge is added?
3. (CLRS 22.5) Professor Bacon claims that Kosaraju’s algorithm would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?

**Problem 7.** True or false: The shortest path algorithm in an edge weighted DAG works even with negative edge weights.

**Problem 8.** How could you modify Dijkstra’s algorithm to find all shortest paths?

**Problem 9.** How could you modify Dijkstra’s algorithm to stop once it’s found the shortest path to a particular node?

**Problem 10.** Say we are given a graph $G$ where all edges are positively weighted integers. 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?

**Problem 11.** Find the shortest path between vertices $E$ and $G$ in the graph provided

**Problem 12** (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.)