Dijkstra’s, Kosaraju’s, and Topological Sort—Monday, March 18 / Tuesday, March 19

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

**DIJKSTRA**($G, s$)

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

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.
Example

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

Dijkstra’s algorithm produces the following state:

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance from A</th>
<th>Parent node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>NULL</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>7</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 it’s 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   time = 0
2   for each vertex $v \in G.V$
3       $v.visited = false$
4   for each vertex $v \in G.V$
5       if $v.visited == false$
6           DFS-VISIT($v$)

DFS-VISIT($v$)
1   time = time + 1
2   $v.d = time$
3   $v.visited = true$
4   for each vertex $w \in G.Adj(v)$
5       if $w.visited == false$
6           DFS-VISIT($w$)
7   time = time + 1
8   $v.f = time$
Figure 1: An example graph for the timestamped DFS algorithm. Note that the source node, \( s \), has a start time of 1. Discovery and finish times are indicated inside each node. The edges traversed by DFS have been highlighted.

**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 $x \leadsto u$ in $G$, then $u$ and $x$ are strongly connected only if there is also a path from $x \leadsto u$ 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 $G$. 

4
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.

   **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.

**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.

**Solution.** First, use Kosaraju’s algorithm to create the graph of SCCs, $G_{SCC}$. Then topologically sort the graph, since it is a DAG.

If the graph is ‘almost strongly connected’, then adding a single edge will connect the graph.

Add an edge from the last component to the first, and check if the graph is now strongly connected using DFS/BFS.

**Correctness.** If the algorithm returns true, meaning our new graph was strongly connected, since we only added a single edge, it follows that the original graph was almost strongly connected.

In the case that our algorithm returns false:

For a graph to be almost strongly connected, every vertex must have a path to vertex $s$, the source of the edge to add, and a path from $t$, the second vertex in the new edge. If for contradiction it didn’t then the new graph must clearly have a vertex with no path to $s$, or no path from $t$, as adding an edge from a vertex does not affect its reachability.

$s$ must be in the first component in $G_{SCC}$, as it it wasn’t, then any vertex earlier in the topological order is clearly not reachable from $s$. 
must be in the last component in $G_{SCC}$, as if it wasn’t, then any vertex later in the topological order clearly can not reach $t$.

Running time.

Steps:
1. Creating SCC kernel graph: $O(|V| + |E|)$
2. Topological sort: $O(|V| + |E|)$
3. Checking if strongly connected: $O(|V| + |E|)$

Therefore, this algorithm is $O(|V| + |E|)$.

Dijkstra’s Questions

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: 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.

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.

Solution. False, consider the first vertex the DFS visits in $s$. Consider a path from that vertex within $s$ that only has edges to other vertices in that SCC. If DFS takes this path before taking an edge out of $s$, the vertices on the path will finish first. Since the SCC graph is a DAG, we will never revisit $s$ if we take an edge out. It is true though that at least one vertex must have a larger finish time than those SCCs reachable from $s$. 

□
2. How does the number of SCCs of a graph change if a new edge is added?

Solution. Consider a new directed edge \((u, v)\). We have two cases. Either \(u\) and \(v\) are in the same component, in which case the total number of components does not change and we are done; or \(u\) and \(v\) are in different components. Let \(u\) and \(v\) be in components \(C_u\) and \(C_v\) respectively. Consider the component graph. If \(C_u \sim C_v\), then \((u, v)\) does not change the total number of components, since it is redundant. But if instead \(C_v \sim C_u\), then via \((u, v)\) we have \(C_u \sim C_v\). Thus, all components reachable with a path starting at \(C_u\) and ending at \(C_v\) (including \(C_u\) and \(C_v\)) are contracted into a single component.

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?

Solution. No, consider the first connected connected component having the vertex with the smallest finish time (see first true/false). Then a DFS would start from this vertex and discover the whole graph, declaring it incorrectly as a single connected component.

Problem 7. 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 8. 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 9. 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 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?

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. In fact, the edge weights could be up to \(2^n\), giving us a very poor runtime of \(O(2^n)\).

Problem 11. 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>
<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 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.)

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.

```python
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.

```python
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?