Homework #6: Seam Carving
CIS 121—Fall 2017

Due: Friday, November 3rd, 2017

1 Analysis of seam carving
10 points

1. Analysis of running time: Estimate empirically the running times (in seconds) to remove one row and one column from a W-by-H image as a function of W and H. Use tilde notation to simplify your answer. Both should be functions of W and H. Removal should involve exactly one call to the appropriate find method and one call to the appropriate remove method.

2. Empirical analysis of your algorithm. Justify your answer with sufficient data using large enough W and H values. To dampen system effects, you may wish to perform many trials for a given value of W and H and average the results.
   • First, keep W constant and then estimate the Row removal time (seconds) and the Column removal time (seconds) for different values of H.
   • Next, keep H constant and then estimate the Row removal time (seconds) and the Column removal time (seconds) for different values of W.

Solution Follows from homework implementation

Suggested rubric:
1. 4 points
2. 6 points (3 for H, 3 for W)

2 Heteronormative graphs
10 points

A former senator from Pennsylvania is really interested in a particular kind of graph. He wants to color each node in a graph as either pink or blue. He is very concerned that the edges in the graph should only connect pink nodes to blue nodes, and that no edges connect a blue node to another blue node, or a pink node to another pink node. He’s fine if a blue node connects to multiple pink nodes. He knows one blue node that has connected with at least three pink nodes, and there have been reports that it has connected to many other pink nodes. In a sign of surprising progressiveness, he’s also OK with a pink connecting with multiple blues. Write an algorithm to check a graph for the former senator, so that it tells him whether it conforms to his constraints or not.

Remember to prove that your algorithm is correct and remember to give its running time. Senators love efficiency.

Solution
This is an instance of determining whether a graph is bipartite or not.

Algorithm: Pick a node $u$ and color $u$ pink. Do a BFS starting at $u$. Whenever you pull a node $v$ off the BFS queue, consider the color of each neighbor $w$ of $v$. If $w$ has a color and $v.color = w.color$ for any neighbor, output “false”, i.e. this graph is not "heteronormative". Otherwise, make $w.color$ the opposite of $v.color$ and continue the BFS appropriately.

Running time: $O(n + m)$, well established BFS running time.

Justification of correctness: for any node $v$ that was pulled off of the BFS queue, I know BFS will examine all the neighbors of $v$. So I know that I’ll be able to check whether $v$ violates the 2-coloring condition with any of its neighbors. Since every node in the graph gets enqueued, this will hold for all nodes.

**Suggested rubric:** Score full points for:
- Recognizing this as an instance of the bipartite graph problem (2)
- Correct BFS or DFS implementation (3)
- Correct running time (2)
- Sketch of correctness proof (3)

### 3 Unique Minimum Spanning Tree

**10 points**

Give a proof or a counterexample for the following assertion: An edge-weighted graph has a unique MST only if its edge weights are distinct.

**Solution** Counterexample:

![Graph 1](image1)

![Graph 2](image2)

**Suggested rubric:** Full points for correct counterexample.
4 Graphs With Negative Weights
10 points

1. Given the graph above, show the shortest path tree that Dijkstra’s algorithm would produce for the graph starting at vertex 0. Would Dijkstra’s algorithm produce the correct shortest path tree for the graph? Why or why not?

2. You could convert a graph with negative edge weights to one that has only positive weights by adding the value of the most negative edge weight to all edges in the graph. Would this cause Dijkstra’s algorithm to find the correct shortest path tree? Prove your answer.

3. Explain how the Bellman-Ford algorithm is able to find the shortest path even when there are negative weights. When does Bellman-Ford fail to output the shortest path tree?

Solution

1. Dijkstra produces the incorrect shortest path tree from 0:

   ![Graph Diagram]

<table>
<thead>
<tr>
<th>Vertex</th>
<th>distance to</th>
<th>edge to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>8→3</td>
</tr>
<tr>
<td>4</td>
<td>-15</td>
<td>9→4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0→5</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>7→6</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>3→7</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>9→8</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>2→9</td>
</tr>
</tbody>
</table>

   Here is the correct Shortest Path Tree:
2. Below is a contradiction to simply adding the weight of the most negative edge to all edge weights to make it positive, and then run Djikstra.
Note that the path to get from 1 to 4 that will be returned in the second graph will be the direct path from 1 to 4, when it should actually be the path from 1 to 2 to 3 to 4.

3. Bellman-Ford will be able to find the shortest path even when there are negative edge weights because it does not assume that the node that one is currently on has the shortest possible distance to it, which Dijkstra does. Bellman-Ford constantly relaxes edges until the distances can no longer be decreased further, ensuring that no matter what the edge weights are, the distances are minimized.

Bellman-Ford fails when there are negative edge weight cycles, because the algorithm itself forces a fail. With such a negative edge weight cycle you can never find the set of shortest paths because continually going along this cycle will yield lower and lower path lengths.

Suggested Rubric:
1. 3 points (1 for correct answer, 2 for explanation or showing correct tree)
2. 4 points (2 for correct answer, 2 for correct contradiction)
3. 3 points (2 for correct explanation, 1 for instance when Bellman-Ford fails)

5 Topological Sort

10 points

You can topologically sort a DAG by repeatedly finding a node that doesn’t have any incoming edges, and then removing it and all of its outgoing edges from the graph.

1. Create an algorithm using this idea with running time $O(V + E)$.
2. Would your algorithm work if the graph was cyclic? Why or why not?

Solution

1. First have a set stored with all vertices with in-degree 0. Then, you remove this vertex (along with all of its edges) from the graph and put this into your output graph. When you’re removing edges, you check whether the vertex you’re removing the edge from now has an in-degree of 0. If so, then add this to the set of vertices that has indegree 0. You then repeat this process by popping off from this set of nodes with in-degree 0 any arbitrary node. Eventually, this set will be empty, and that is when the algorithm will terminate.

Runtime justification Note that finding all vertices with in-degree 0 in the beginning is simply an $O(V + E)$ operation. This is because we can simply run DFS on the graph and every time we encounter a node (via an edge) we increase its in-degree count by 0. Any node with an in-degree of 0 is added to the set of nodes with in-degree 0. This runs in $O(V + E)$ time.

From there, we then look at a node with in-degree 0 from this set and remove it from the graph along with all of its edges (and add it to the output graph). We then check each edge such that when it is removed, if the node it is connected to now has in-degree 0 we add it to the set of nodes with in-degree 0. Note that we will encounter each node exactly once and encounter each edge still exactly once, thus the runtime will be $O(V + E)$. Also, added to the set takes constant time (assume the set is a HashSet).

Proof of Correctness To prove that this algorithm works, let us use induction on the current step of execution of the algorithm (where a step is the removal of a node from the DAG, and inserted into the topologically sorted graph T).

Induction Hypothesis: Assume that the algorithm works for the first k steps. Let us call this execution time E.
Induction Step: Let us consider the first k iterations of the algorithm. If we use the induction hypothesis, this means that we have two graphs, our DAG, and T (which we hope to return at the end), which currently has k nodes. In our DAG, we have two cases, either there exists a node with in-degree 0, or there exists no node with in-degree 0. Let us consider the first case.

**Case 1: There exists a node, U with in-degree 0** In this case, we simply add this node to the end of the T, and remove all edges connected to U in the DAG. Note that although the node had in-degree 0 in the DAG, it may have an in-degree in T. Now, let us show that although it may have in-degree greater than 0, when it is added to T, it cannot have out-degree greater than zero at this point in execution in T. Let us assume for contradiction that U has an edge to some other node in T, V. Note that V was added to T before U was added to T. Therefore, when V was added, we know that it must have had in-degree of 0. However, there is an edge from U to V, and since U was added to T after V, this means that the edge from U to V would have existed in the DAG at execution time E. This is a contradiction, as V would then have an in-degree greater than 0 and not have been added to T. Thus, this means that U can be added to T without fear of creating a cycle.

**Case 2: There exists no node, U with in-degree 0** In this case, let us consider a maximal path in the graph. If we consider the first node in any maximal path, since this does not have in-degree 0, there exists some node V that has an edge to U. Thus, we have a cycle U...V-U, which is a contradiction that our graph is a DAG.

Thus, our algorithm is correct for the k + 1 th iteration of the algorithm.

2. No, you cannot topologically sort a graph that is cyclic. This is because if you take any cycle, by definition there must be a back-edge, thus breaking the invariant of a topologically sorted graph.

**Suggested Rubric:**

1. 5 points (minus two for slower runtime)
2. 5 points