CISC 471 - HW 6

Hershil Devnani (20001045)

April 9, 2021

1 Programming

Please refer to the submitted python files and *README.md* for the solutions to Part 1.

2 Theory

1. Lesson 5.6: Exercise Break: Find the length of the longest path from source to sink in the graph in Figure 1. You may use your algorithm.

In order to find the longest path from source to sink, based on Figure 1 given in the assignment outline, we can work backwards and start with finding the lengths of the longest paths from (0, 0) to every node in the graph. As we build this out, from (0, 0), we will eventually be able to find the longest path from source to sink, which will just be the subsequent accumulation of the longest path we have found so far to every node in the graph.

The figure below shows the sum of the maximum sum of the weights to get to each node, as well as the path taken to reach this maximum length highlighted in blue. We essentially traverse every node in the graph, starting at (0, 0), and at each step, for every node, select the largest weight from every previous node. This way, we eventually reach the sink node, (4, 4), with the length of the longest path being 35.

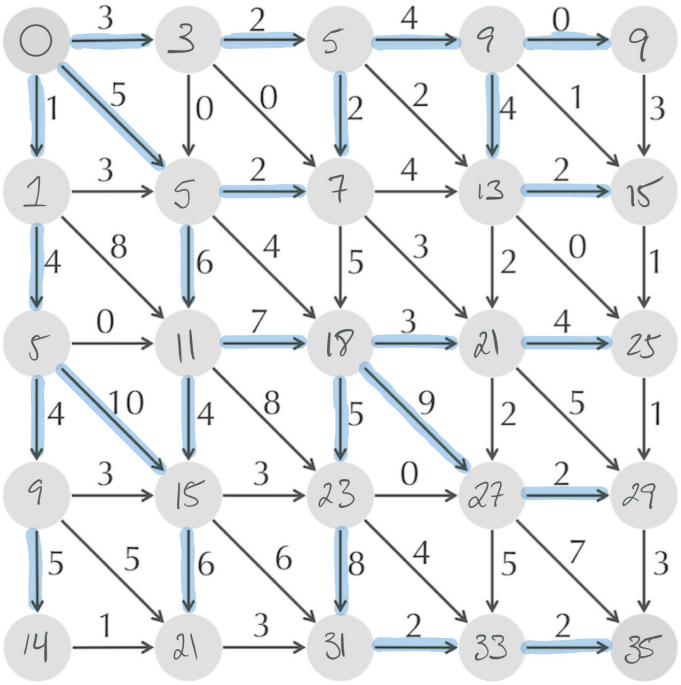

Figure 1: Representation of longest path to any node in the graph and the value of the longest path in each node. The blue highlights represent the path taken to get to the longest path to any particular chosen node.

2. Lesson 5.12: Exercise Break: Design an algorithm for computing optimal local (rather than global) alignment with affine gap penalties.

Note: Please see the submitted file titled *5.12_pseudocode.py* for the pseudocode for this implementation.

The modification made to computing an optimal local, rather than a global, alignment with affine gap penalties included adjusting the scoring mechanisms for each node to include a zero-weighted path into each node form the source as well as a zero-weighted path from each node to the sink node. Additionally, the pseudocode for the algorithm was constructed as a progression from writing pseudo code for global alignment then to adjusting it for local alignment. Therefore, the pseudocode contains function definitions that allow dynamic switching between local and global alignment as well as for using affine scoring gaps. In addition, modification have to be made to model the problem using three different graphs to factor for affine scoring in local alignment.

The key to the local alignment of the algorithm is the three different level for dynamic programming. Combining this with the concepts for local alignment,

such as every node having an incoming edge from the source node with weight of zero and every node also having an outgoing node to the sink node with weight of zero, allows us to create an algorithm that can be implemented with affine gap penalty scoring for local alignment.

3. Lesson 5.13: Exercise Break: Design a space-efficient algorithm for local sequence alignment.

Note: Please see the submitted file titled *5.13_pseudocode.py* for the pseudocode for this implementation.

In order to design a space-efficient algorithm for local sequence alignment, they key changes that need to be made, when compared to a global alignment, include discarding nodes that result in a zero edge score. The key to the space-efficiency of the algorithm is segmenting and dividing up the alignment sequencing on either side of the middle node by a half over each iteration, or each subsequent sequenced node added, i.e. from source to middle and middle to sink. On each pass through for sequencing, the total search space area gets halved. We start with two halves, one from the source to middle and one from the middle to sink. Each node added to the sequence allows us to further cut the next search space by another half. So the total search space is less than 2 times the area of the graph.

Combining this with the concepts for local alignment, such as every node having an incoming edge from the source node with weight of zero and every node also having an outgoing node to the sink node with weight of zero, we can create an algorithm that can be implement with space-efficiency. We can add a step during the alignment to check whether the alignment step scored a zero, up to a point where we reach positive scoring in the alignment. If we do this, we don't have to keep track of the nodes that scored zero and can move along to the next column until we reach a node we can add to our sequence. This way, we continue to reduce our search space by 1/2 each subsequent addition to the alignment, and until we reach the local alignment, we can disregard all leading zero weighted nodes in the top left half of the graph, and all disregard all trailing zero weighted nodes in the bottom left half of the graph.

The runtime of sequencing algorithms is proportional to the number of edges which is quadratic. The memory consumption of alignment algorithms is proportional to number of nodes which is also quadratic. Memory is usually the bottleneck, since increasing runtime operations require additional time, but to store billions of values in memory, you need many gigabytes of memory, which is not something that is readily available on most machines, and this memory requirement grows as the sequences being aligned get larger. We can usually wait for runtime operations, but we don't usually have large memory often needed for the algorithms when the input size grows very large.