MATH 580A Assignment 3

1 Checking Bipartite-ness

In class, we saw how to use DFS in order to compute whether or not a graph is k-colorable. A graph that is 2-colorable is called ***bipartite***. In the first program, you will write a function to check if a graph is bipartite, using BFS. Using a BFS approach, you can color all the vertices at distance d away from the root vertex with color 1 if d is odd and color 2 if d is even.

After you color a node, you should check to see if it has been colored the same color as any of its neighbors. If so, then a contradiction has been reached and the graph is not bipartite. If you manage to color all the nodes successfully without contradictions, then the graph is bipartite.

2 Topological Sort

A real world application of graphs is for deciding on an order to do tasks. Perhaps you have a project that has some set of tasks $\mathcal{T}$ and each task $t_i \in \mathcal{T}$ depends on first finishing some other tasks $t_{i_1}, \ldots, t_{i_k}$. For example, your "project" may be throw a birthday party. One task might be to bake a birthday cake but this is dependent on a task where you shop for ingredients. Sending invitations to the party is independent of both of the aforementioned tasks. The problem to solve here is to find an ordering of the tasks such that you can do them in order without worrying about dependencies.

This problem can be modeled using graphs and the resulting algorithm is known as ***topological sort***. Each task t_i is a node in a directed graph and if task t_i depends on task t_j, then draw a directed edge from t_j to t_i, indicating that t_j has to be finished first. If there are no cycles in the resulting graph (otherwise you could not finish any of the tasks in the cycle!), then the graph is called a ***directed acyclic graph***, or DAG for short. The problem is then to compute an ordering of the nodes $[t_{i_1}, t_{i_2}, \ldots, t_{i_n}]$ such that if there is a path from t_i to t_j in the graph, then t_i appears before t_j in the ordering. In the real life application, this is saying that by the time we do task t_j, any tasks t_i that are a requirement has already been done first.

You will implement an algorithm for this known as Kahn's algorithm. The idea is as follows: first find all nodes in the graph with no incoming edges. These nodes can done in any order since they have no dependencies so do them all first. Delete these nodes from the graph and repeat. Similar to BFS, you will have a list of nodes to process at each "depth" in the algorithm. The order in which nodes at the same depth are processed do not matter so you will return a list of lists, where the ith lists are the nodes at "depth" i in this algorithm.

3 Intelligent Scissors

Introduction

There is a selection tool in digital image programs known under various names such as *intelligent scissors* or *magnetic lasso*. The tool helps with selecting a specific object from a picture to cut out. To use these tools, you start off clicking on a point in the image and start moving the mouse around the contour of the object. The selection path will approximately follow the the path traced out by your mouse, but it will attempt to intelligently "snap" to the edges of objects in the image and automatically correct for any human inaccuracies. You can see an example of the magnetic lasso tool in action at `https://youtu.be/o-m3loHVbJw?t=126`. In this program, you will implement a variant of Dijkstra's algorithm that we saw in class to implement the intelligent scissors tool. The resulting tool will be able to cutout contours of objects as in Fig. 1. The approach here is based off a 1995 paper by Eric N. Mortensen and William A. Barrett titled "Interactive Segmentation with Intelligent Scissors".

Figure 1: Intelligent Scissors Tracing Out a Bluebird Contour

Modeling an Image as a Graph

The function `image_data_to_graph()` is provided to you and computes a weighted graph representation of an image useful for applying Dijkstra's algorithm on. We would like the shortest path from the initial point to the current position of the mouse to follow the contours of the object in the image. This subsection will give a brief explanation of how this graph representation is computed under the hood.

The graph structure contains one node per pixel in the image. Orthogonally adjacent pixels in the image have edges joining them. Thus, if our image has width w and height h, then our resulting graph is the $w \times h$ grid graph. Now, to each edge, we would like to associate a weight such that pixels of a similar color have a high weight and pixels of differing colors have a low weight. This weight assignment implies that our shortest path will try to avoid the interior of objects and instead follow their contours. See Fig. 2 for an example 6x6 image, its associated weighted graph, and the shortest path on this graph. Observe that this is not the shortest Euclidean distance between the top-left and top-right pixels. Instead, it roughly

approximates the edge by first stepping down one pixel.

The edge weights between graphs are calculated by first computing the gradient in the x and y directions and the Laplacian of the image at each pixel. Images are first typically converted to grayscale as opposed to handling the red, green, and blue channels separately. To convert to grayscale, we compute the luma value given by the weighted sum $L = 0.299R + 0.587G + 0.114B$, where R, G, B are the values of the red, green, and blue values respectively. In order to approximate the gradient and the Laplacian at each pixel, we convolve with the following 3x3 kernels:

$$g_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad g_y = \begin{pmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \qquad \Delta = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Next, if G_x and G_y are the gradients in the x and y direction for a given pixel, we compute the magnitude of its gradient via $G = \sqrt{G_x^2 + G_y^2}$. We also compute a value called the zero-crossing Z at a pixel. For a pixel at position (x, y) we look at its Laplacian $\Delta_{x,y}$ and if it differs in sign from the Laplacian of *some* neighboring pixel $\Delta_{x-1,y}, \Delta_{x+1,y}, \Delta_{x,y-1}, \Delta_{x,y+1}$ *and* its absolute value is smaller than the Laplacian of that neighboring pixel, then $Z = 1$. Otherwise, $Z = 0$.

Finally, for each pixel, we define its cost to be $C = 0.5Z + 0.5G$. To compute the edge weight between a pixel at position (x, y) and a pixel at position (x', y') we average the sum of their costs. This is a simplified model of the edge weight function described in Mortensen and Barrett's paper and will suffice for our intelligent scissors application to perform well.

Dijkstra's Algorithm Variant

For this program, you will only need to code up the variant of Dijkstra's algorithm used to find the shortest path between the initial anchor node that the user clicks on and the current node that the mouse cursor is on. The version of Dijkstra's we saw in class has two issues that we need to address.

First, it only outputs the length of the shortest path from the source node to other nodes. For our application we also need the actual path itself. Fortunately, this can be obtained fairly easily. For each node, we record not only the length of the shortest path to it, but also the parent node that precedes it in its shortest path. In Dijkstra's algorithm, when we "lock in" a node v, we look at each of its neighbors and see if we can compute a path length that is shorter than our tentative upper bound. For any neighboring node n whose upper bound we update, we also update its parent node to be v, the current node we're locking in. This is because the shortest path we've found so far for n involves first taking the shortest path to v and then taking the edge $v \to n$.

At the end of the algorithm, to compute the shortest path from the source to a node v, we start with v, follow its parent node, then follow its parent's parent, and so on until we reach the source node. This is the shortest path from v back to the source, so we can reverse this path to go from the source to v.

The second issue is that the implementation we saw in class is too slow! The naïve version's runtime was $\mathcal{O}(|V|^2)$ where $|V|$ is the number of vertices in the graph. Our sample image has dimensions 308×200 pixels and the resulting graph has $308 \cdot 200 = 61600$ vertices. Therefore, our in-class implementation of Dijkstra would take around $61600^2 \approx 3.8 \cdot 10^9$ operations. This is acceptable (on the order of 10 seconds) if we needed to only compute the shortest path once, but is far too slow for recomputing the shortest path every time the user moves their mouse. We will

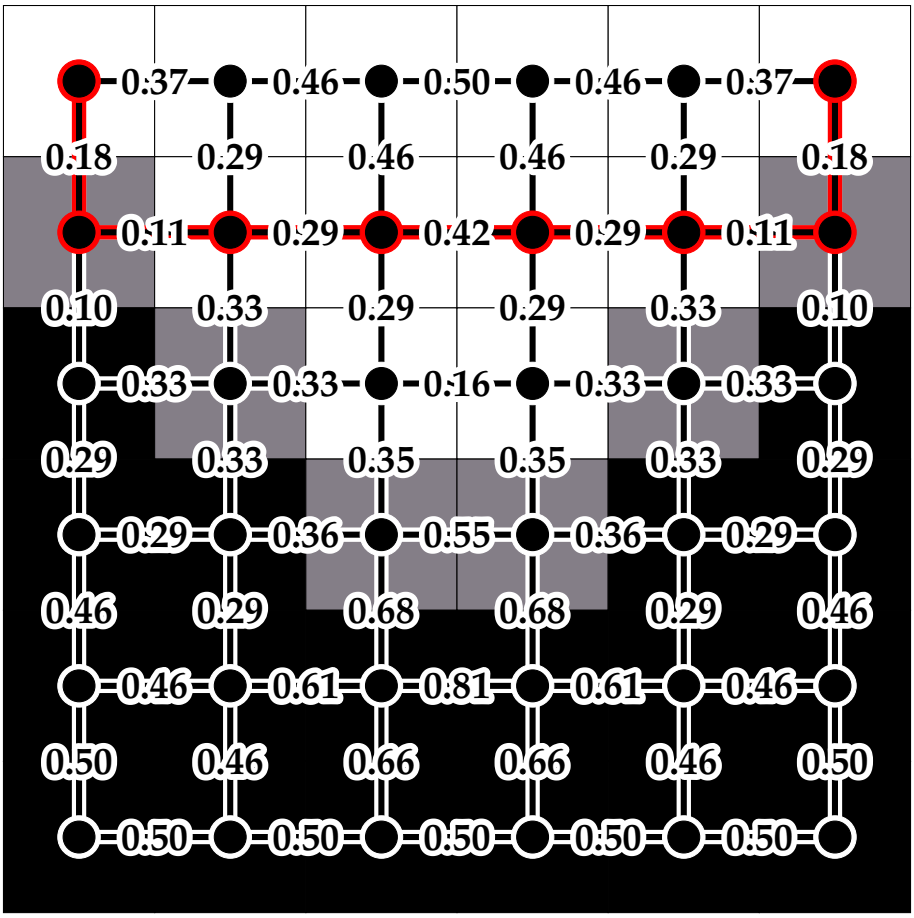

Figure 2: Example 6x6 Image, Graph and Shortest Path.

see how to code up an implementation that runs in $\mathcal{O}((|V| + |E|) \log |V|)$ time. For our graph, each vertex has degree at most 4 so $\mathcal{O}(|E|) = \mathcal{O}(|V|)$. This reduces the number of operations to about $61600 \cdot \log(61600) \approx 7 \cdot 10^5$ which is many orders of magnitude faster.

In order to obtain this speedup, we will optimize how to pick the next vertex to lock in. In class, we looped over every remaining vertex to find the one with the minimum value. We will use a data structure called a *heap* or *priority queue*. A heap acts like a list except that after sorting the elements in it initially in $\mathcal{O}(n \log n)$ time, you can add and remove elements from it in $\mathcal{O}(\log n)$ time and the heap will automatically stay sorted. We can keep a heap of our candidate vertices and it will stay sorted by the path length as we update the lengths when iterating over neighboring vertices. The vertex with the minimum value can then easily be found since the heap remains sorted. Python provides a heap data structure as part of its built-in library `heapq`.

In order to use `heapq`, first create a regular Python list to represent the actual heap. Then, instead of adding and removing elements to the list in the usual fashion, use `heapq.heappush(<heap>, <item>)` to add an element and `heapq.heappop(<heap>)` to pop off the smallest element. In order to implement Dijkstra's with Python's `heapq`, one possibility is to push on tuples of the form `(<path_length>, <node>)`. Tuples are compared by first comparing their first element, then breaking ties by comparing their second element, and so on. By placing `<path_length>` first, the heap will stay sorted according to the tentative path lengths.

One snag with Python's implementation of `heapq` is that there is no default method for updating the values of an element in the heap. In order to work around this, simply push another tuple `(<path_length>, <node>)` when updating a path length of a neighboring node. The heap may contain duplicate nodes as a result, and as you pop tuples off, you should check that the node hasn't already been added to `used`. Also, note that the use of a heap should be in addition to maintaining a `ret` dictionary that stores the shortest path length and the parent node.

There is one more slight optimization to for this particular application of Dijkstra's algorithm. Typically, one calculates the shortest path from the source node to *all* nodes in the graph. However, we only require the shortest path to a specific target node. Thus, once we've locked in the shortest path to the target node, we can stop Dijkstra's algorithm and return that path. The target node will be typically close to where the user initially clicked so this will substantially speed up computations.