Narative of Graph Theorist's Sketchpad

Jadin Sadler
Date 12/13/22

Introduction

I developed a sketchpad for graph theory using C# and WPF. By separating the front-end visuals from the back-end logic, I was able to efficiently implement all of the required features and some additional recommended features. The internal structure of the code includes various class types that interact with a main class that handles the main operations.

Here is a list that I will use to go into detail about the features that I implemented and explain how I was able to do it with the code that I wrote.

Required functionality

Graphical display of vertices and edges & Input of vertices and edges

This one was a fun one to implement, so I have a canvas which sits and listens for mouse down events on it. I also have a enum that stores what toolset is currently selected this corresponds to the buttons on the side of the interface. When the canvas, called DrawingCanvas, receives a left mouse down event, it checks if the vertex tool is currently selected. If so, it checks if the user has clicked on an existing vertex or a blank space. If you clicked on a blank space then it creates an ellipse at that location and creates a new vertex object to be associated with it and stores them together in a dictionary.

When you have the edge tool selected then it waits for you to select a vertex that is currently drawn on the DrawingCanvas and if it's the first one you have selected then it selects it and remembers your selection in the backend. If you already have a vertex selected and select another one then it draws a path composed of two arcs between the two vertices. This process also creates an edge object that gets associated with this path object in a dictionary like the vertices do.

Able to reposition vertices while maintaining adjacencies

I implemented this by first storing in each vertex object where its ellipse is drawn on the canvas. Second I created a way for each Edge object to store its starting point and end point so that I could update them according to the change in its starting vertex and ending vertex. Third I Created in each Vertex object a way for it to signal when its position has been changed which is an event. Each Edge object subscribes to this event in both its starting vertex and ending vertex and associates its starting point and ending point to the same location as its vertices. When the vertex is moved with the move tool then the event is fired updating the edges points and then the edge is redrawn to reflect this.

Deletion of Vertices and Edges

When you click on an edge or a vertex while having the delete tool selected it removes that vertex or edge from the canvas by updating the canvas's children objects. If you delete an edge then the path is deleted from the dictionary that stores its association to a edge object, then that edge object is deconstructed so that no vertexes contain references to it. If you delete a vertex then you have to first delete all edges that are connected to that vertex then you can safely remove the vertex from the vertex dictionary without any form of artifacts being left over.

Parallel Edges

I solved this problem by drawing arcs instead of lines for my edges, this does mean that if you try to make two edges that go from the same first node to the same second node as another edge that exists then they will overlap.

Loops

When using the edge tool and clicking the same vertex twice you will get a loop. I implemented this by just making the path that the edge is drawn from a ellipse instead of two arcs.

Ability to color or label vertices

I implemented this with a really handy nuget package found here. Effectively WPF does not have a inbuilt color picker so I used the one there, it contains on the control a variable called current color. When you have the color tool selected and you click on a vertex it updates that vertexes color to the one selected as the selected color on the color picker. I did have this working with edges but I removed the functionality because it did not function perfectly. Funny enough though I still have the same bug even after disabling it so I might have been able to get extra points if I had left it in.

Recommended Features

Information about numbers of vertices and edges

How I implemented this was by storing each vertex and edge that I create into a dictionary. When I draw or delete either I update the dictionary to reflect the change. The interface contains two labels that just update based on the size of the dictionary in the back end.

Things that I almost got working

This section will be dedicated to the things that I almost had working but never displayed in a meaningful way to the user.

Information about degrees of vertices

I had this working in the back end, and the plan was to have a new tool called the info tool. When you would click a vertex with this new tool it would have displayed the number of edges currently connected to it. I had each vertex keeping track of each edge that was connected to it already I would have just needed to implement the info tool.

Display of directed arcs (for directed graphs)

Each edge already knew what direction it was drawn from so all I would have had to do would have been draw an arrow pointer on it somewhere pointing in the direction that it was drew.

Information about components

I would have had to just go through the list of vertices and check off all that are connected to each other then move to the next one in the list that was not checked off and rinse repeat until all vertexes have been counted.