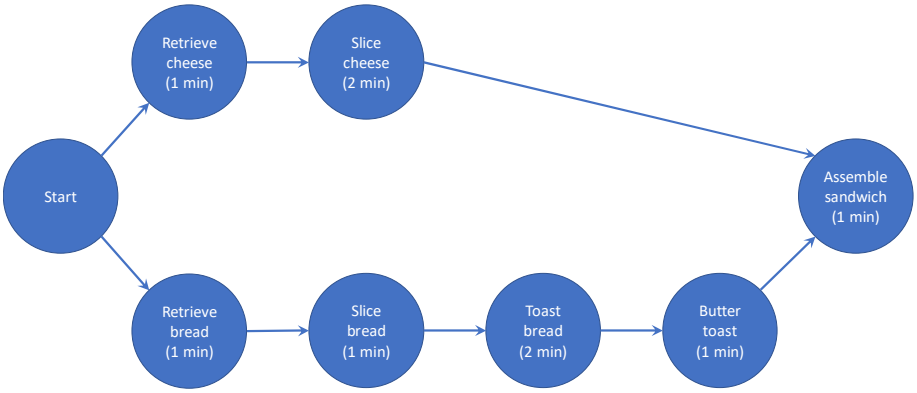
CMPT 381 Assignment 3
MVC, Multiple Views, and Interactive 2D Graphics

Due: Friday, November 6, 11:59pm

Overview

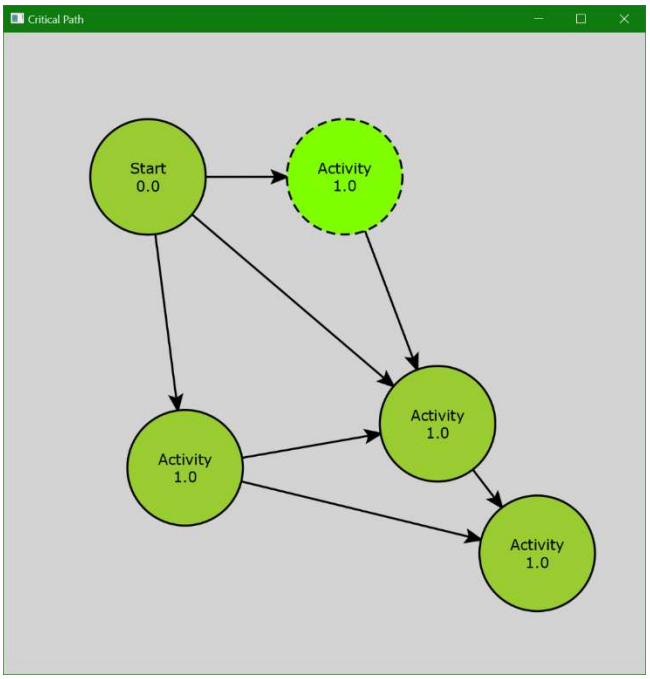
In this assignment you will build a JavaFX system that demonstrates your skills with Model-View-Controller and multiple views, 2D graphics, and interaction with graphical objects. The application, called CriticalPath, allows users to create a dependency graph for a project and see the critical path from project start to project completion. A dependency graph is a directed acyclic graph (DAG) where the nodes are activities in the project, and edges connect activities in terms of their time dependencies. For example, a dependency graph for the task of making a cheese sandwich might be as follows (note that we assume unlimited workers to carry out the tasks, meaning that all branches can be carried out in parallel):



The critical path is the path through the graph that requires the most total time, and so indicates the total time that will be needed to complete the project. Since the "toast" path in the graph requires 6 minutes (and the "cheese" path requires only 4 minutes), the toast path is the critical path. (Note that critical path is calculated only based on time, and does not care about the number of nodes in the path).

The assignment has 3 parts: Part 1 involves the main UI for creating and working with the graph, Part 2 involves additional views and navigation, and Part 3 involves graph operations.

Part 1. Main graph UI



The main panel for the system shows the graph and allows the user to interact with it (i.e., creating and deleting nodes and edges, dragging nodes on the workspace). The system uses an MVC architecture as specified below.

Interaction requirements:

- The graph view shows a graphical representation of the graph, with nodes as coloured circles (with titles and time costs), and edges as arrows between the nodes
- The user can create a new node by right-clicking the mouse (pressing + releasing) on the background. When a new node is created, it has the default title "Activity" and the default time cost 1.0
- The user can select a node by pressing the left mouse button on the node; selection is indicated in the view by drawing the node with a dashed outline and a different fill colour.

- The user can move a node by pressing the left mouse button on a node and then dragging the mouse. (This also selects the node)
- Selection is persistent (i.e., the node stays selected even after a move is completed)
- The user can delete a node by selecting it and then pressing the Delete key on the keyboard
- The user can create an edge by right-clicking on the start node, dragging to the end node, and releasing the mouse button. As the user drags the mouse, a temporary line is shown from the start node to the mouse cursor; when the mouse is released on the end node, the edge is created and shown. If the user releases when the cursor is not on a node, the temporary line is discarded and no edge is created.
- Edges show arrowheads to indicate the direction of the edge
- The user can select an edge by left-clicking on it (within a small "close enough" region as described in lectures); a selected edge is shown using a dashed line
- The user can delete an edge by selecting it and then pressing the Delete key on the keyboard
- Selecting any node or edge removes any previous selection. There is no multiple selection.
- When the application starts, the graph contains a single node titled "Start", with a time cost of 0.0. This is the root node for calculation of the critical path, and cannot be deleted or edited (however, it can be dragged).

Code requirements:

- Create class MainGraphView to show the graph
    - The view must use immediate-mode graphics in JavaFX (i.e., a Canvas)
    - The view should use 2D transforms to position the graphics reference frame before drawing nodes, edges, and arrowheads
- Create class GraphModel to store the graph
    - Create classes Node and Edge that will be used by the model
    - GraphModel must use publish-subscribe communication to the view
    - GraphModel must have a public API that can be called by the controller
    - The model must store locations as normalized coordinates
- Create class GraphViewController to handle user events from the MainGraphView
    - The controller must implement a state machine to handle mouse events from the MainGraphView
    - Note: when you set up event handling in MainGraphView, normalize the mouse coordinates before they are sent to the controller (as shown in the lab example).
- Create class InteractionModel to store view state including node and edge selection

Resources for Part 1

- The BoxDemo code and the graphics demos available in the Code Examples folder on the course website
- JavaFX APIs for Canvas and GraphicsContext: openjfx.io/javadoc/15/

Result for Part 1: a zipped IDEA project that meets the interaction and code requirements above. To export your project, choose File → Export → Project to Zip file. NOTE: if you have fully completed Part 2, do not hand in a project for Part 1.

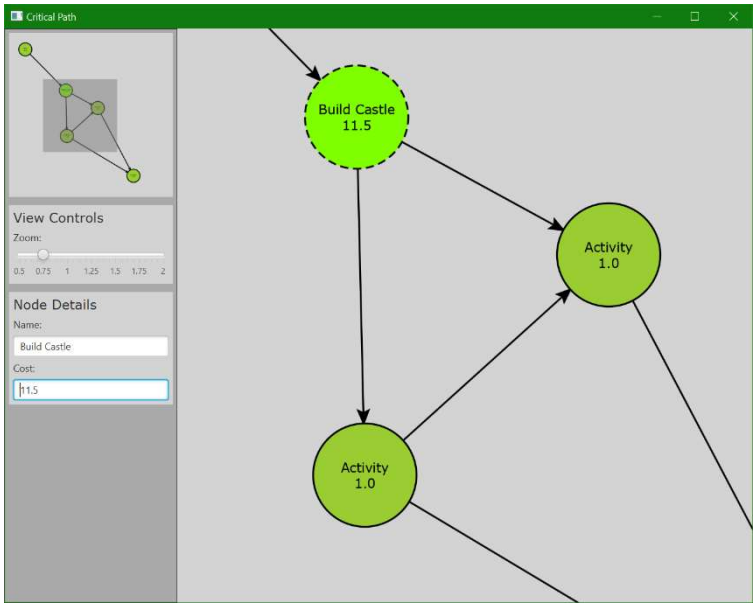
Part 2. Additional views and viewport navigation

In Part 2 you will create additional views for the CriticalPath app, and will implement panning and zooming capability. There will be two new views of the graph (a view of a single node's details, and an overview of the entire workspace), and you will combine these and your existing graph view into a new composite view that holds the entire UI.

MainGraphView: new interaction requirements

- The graph is now shown on a square workspace that may be larger than the MainGraphView's canvas, and therefore the MainGraphView now shows a viewport onto the workspace. For example, the workspace might be 2000x2000, and a 500x500 MainGraphView will show one quarter of the workspace.
- The viewport can now be panned: when the user presses on the background (either mouse button) and drags, the view will pan by the amount of the drag.
- Panning is restricted to the size of the workspace (i.e., the left/top of the MainGraphView can never go past 0,0 in the workspace, and the right/bottom of the MainGraphView can never go past the workspace extents)

- If the window is resized, the size of the MainGraphView canvas changes, but not the size of the workspace.



MainGraphView: new code requirements

- Add variables to MainGraphView to store the size of a logical workspace that can be different from the size of the MainGraphView. (You may wish to also store the workspace size in the InteractionModel)
- Use the width and height of the workspace as the extents for normalizing coordinates (i.e., divide the mouse X and Y by the workspace width and height, rather than the MainGraphView width and height)
- Add viewportLeft and viewportTop variables to the InteractionModel to store the left and top coordinates of the viewport within the workspace
- When you draw the graph in the MainGraphView, subtract viewportLeft and viewportTop from your translations in order to draw the correct content in the MainGraphView. For example, assume the workspace is 2000x2000 and the MainGraphView is 500x500. Assume a Node has location 0.5, 0.5 in the workspace, and that the viewport is at viewportLeft 0.4 and viewportTop 0.4. The Node should be therefore be drawn in the MainGraphView at x = (0.5 - 0.4) * 500 and y = (0.5 - 0.4) * 500.

NodeDetailView:

- Create class NodeDetailView to show the name and time cost for a selected node
- Lay out the components of NodeDetailView as shown in the picture
- If no node is selected, this view should show blanks for both name and time
- The text fields in the NodeDetailView are editable: if the user enters a new name and presses Return, or enters a new cost and presses Return, the selected node will be updated with the new values. (Note that using Return to generate an event in a TextField can be done with the setOnAction() method.)
- If the user attempts to edit the Start node, the values return to their defaults (“Start” and 0.0)

MiniGraphView:

- Create class MiniGraphView to show an overview of the entire workspace
- The mini view should be very similar to the MainGraphView, with the following changes:
    - The mini view is 200x200 pixels, and does not change size
    - The mini view always shows the entire workspace
    - The mini view shows the MainGraphView’s viewport as a transparent grey rectangle
    - There are no user interactions with the mini view.
- You may wish to create an inheritance hierarchy to avoid duplicating code for the two graph views (e.g., MainGraphView and MiniGraphView both extend abstract GraphView)

Composite app view:

- Create class MainAppView that will hold your other views and additional controls
- Lay out the views inside MainAppView as shown in the picture
- Add a panel “View Controls” between the mini view and the node view
    - The view controls should show a slider with extents [0.25..2.0]
    - Dragging the slider changes the zoom level in the MainGraphView (and also changes the size of the viewport rectangle in the mini view). Add a variable zoomLevel to InteractionModel to store this value.

Resources for Part 2

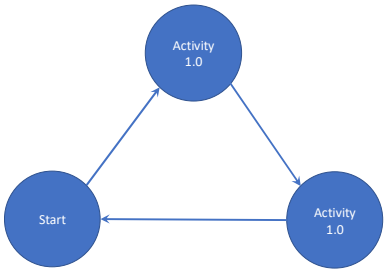
- The lab demonstration of composite views, and the view-control demo code available in the Code Examples folder on the course website

Result for Part 2: a zipped IDEA project that meets the interaction and code requirements above. To export your project, choose File → Export → Project to Zip file. NOTE: if you have fully completed Part 3, do not hand in a project for Part 2.

Part 3. Graph operations: detecting cycles and finding the critical path

Add methods to your CriticalPath app to check for cycles and to calculate the critical path, and add to your view classes to highlight the critical path through the graph.

Detect cycles on edge creation:



- Dependency graphs that calculate critical paths must be acyclic (an example of a cycle is shown in the picture at right)
- Add code to check for cycles whenever an edge is added to the graph. If adding the new edge results in a cycle, undo the addition of the edge
- Use recursive depth-first traversal to check for cycles:
    - In class Node, add boolean instance variables *visited* and *exploring*
    - In class GraphModel, add method boolean checkForCycles() that first sets the visited and exploring flags of all Nodes to false, and then calls findCycles() on the start node.
    - In class Node, add method findCycles() that implements the following algorithm:
        - if this node's visited flag is true, return false
        - if this node's exploring flag is true, return true
        - set this node's exploring flag to true
        - for each outgoing edge:
            - if findCycles() on the end Node of this edge is true, return true
        - set this node's visited flag to true
        - return false

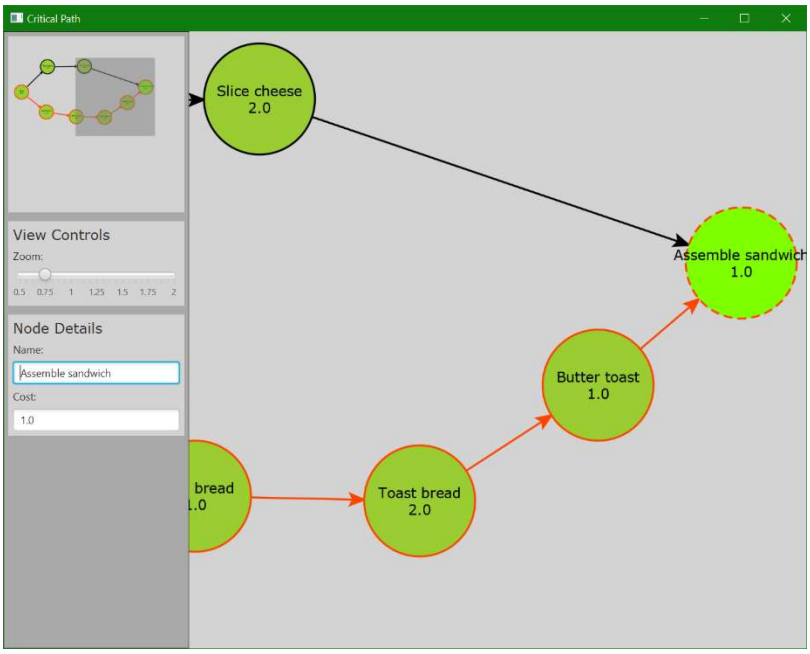
Calculate the critical path:

- Whenever an edge is added or a time cost is edited, recalculate the graph's critical path
- Create class GraphPath to store a path through the graph
    - This class should have instance variables to store a list of Nodes, a list of Edges, and a total cost
- In class GraphModel, create method findCriticalPath() that does the following:
    - Create a new Path *current* and add the Start node to the path
    - Create a list of Path objects *allPaths*
    - Call findPaths(current, allPaths) on the Start node
    - Iterate through the paths in allPaths and calculate the total cost for each path
    - The path with the highest cost is the critical path; store this path in the GraphModel
- In class Node, create method findPaths(Path currentPath, List<Path> allPaths) that implements the following:
    - if this node has no outgoing edges, add the current path to allPaths and return
    - for each outgoing edge:
        - create a new Path *continuePath* and copy the nodes and edges from currentPath
        - add the current edge's end node to continuePath's nodes
        - add the current edge to continuePath's edges
        - call findPaths on the edge's end node, passing in continuePath and allPaths
- In your view classes, when you draw nodes and edges, check whether they are in the critical path; if they are, draw them with a different stroke colour

Resources for Part 3

- medium.com/@trykv/algorithms-on-graphs-directed-graphs-and-cycle-detection-3982dfbd11f5

Result for Part 3: a zipped IDEA project that meets the interaction and code requirements above. To export your project, choose File → Export → Project to Zip file.



What to hand in

Note that this assignment is to be done individually; each student will hand in an assignment.

- A zip file of your IDEA project folder and a readme.txt file that indicates exactly what the marker needs to do to run your code. (Systems for 381 should never require the marker to install external libraries, other than JavaFX).
- You only need to hand in one project if that project represents everything you have completed on the assignment. However, if you have completed different elements from different parts of the assignment, or if there are bugs in one part of your solution, you can hand in multiple zipfiles to demonstrate the parts that you have completed. (If this is the case, state in your readme.txt file which features are available in which projects)

Where to hand in

Hand in your files to the link on the course website.

Evaluation

Marks will be given for: producing a system that runs without errors; demonstrating that you can develop MVC-based systems with 2D interactive graphics; demonstrating that you can develop multiple views and composite views; demonstrating that you can work with view geometry and navigation at multiple scales.

Weighting of the parts in the overall grade: Part 1=35%, Part 2=50%, Part 3=15%.

Note that no late assignments will be allowed, and no extensions will be given, without medical reasons.