CS2110 Assignment 5 — Graphs and GUIs, Summer 2008
Due August 1, 2008, 11:59PM

0 Introduction

0.1 Goals

The goal of this assignment is to help you get familiar with graph algorithms, and some basic concepts involved in developing graphical user interfaces (GUIs).

0.2 Submission

Follow the Submission Requirements on the course website. The last section of this assignment summarizes the files that you need to submit on CMS.

0.3 Partners

As usual, you may work in groups of 2 for this assignment.

0.4 Grading

As before, solutions will be graded on both correctness and style. For correctness, at the very least your program must compile without errors or warnings. For style, we like concise, clear, easy-to-read code. Use mnemonic variable names. Use proper indentation. Comment where necessary for clarification, but do not overcomment. Be concise; a two-page solution to a problem that can be done in a few lines will lose points, even if correct. The assignment will be graded out of 100 points and is worth 10% of your course grade.

1 Ticket to Ride

Points: 100

In this assignment, we will develop a computer-based version of the popular board game 'Ticket to Ride' (abbreviated T2R below). This promises to be a fun assignment, with sufficient opportunity for you to exercise your creativity. Get ready to ride!

1.1 The Rules of the Game

As a first step, we would like you to get familiar with the rules of the game, available online at:
`http://static.ticket2ridegame.com/lang/english/images/tt_rules_en.pdf`.
For the most part, we will follow these rules, with the following exceptions/differences:

- Our game will accommodate 4 human players and 1 computer player.
- Players are dealt only 2 ticket cards to start the game, and must keep both.

To give you an idea of what our game would look like, we have developed a 'demo' version of the game. You can get the demo on CMS. To run the demo from a command line, make sure you are in the directory containing all the files (i.e. the `.class` files). If you are using the Windows command prompt, type:

```
java -cp .;bailey.jar TicketToRide
```

If your computer does not use Windows, replace the semi-colon with a colon.

1.2 Program Overview

The T2R program is significantly larger than the other programs you've written for this class. In fact, it is probably too large a project even for mighty CS211 students to finish in a week. As a result, we have provided the code for the demo for you to build on top of. This will let you focus on the interesting parts without getting buried by the quantity of code needed.[1]

The provided code handles the bulk of the GUI creation and setup, and also contains the definition and implementation of certain classes that represent objects and concepts in the game. The rest of this section describes the program structure and highlights the parts you need to implement to complete the T2R game. A margin note, as shown here, accompanies text that describes what you need to do. Section 3 lists resources you can use for this assignment.

Your Responsibility

1.2.1 The View from 10,000 Feet

This section outlines the overall structure of the TicketToRide program and summarizes the classes we have provided. One clarification on terminology. The game rules use **route** to describe a single rail line between two cities. Here we will instead use **connection** to refer to a single rail line connecting a pair of cities. To simplify modeling (and enforcing some game rules), we (usually) use **route** to refer to *all* the connections between two cities. So a route consists of 1 or 2 connections.

Game Object The heart of the program is the `TicketToRide` class. It performs two roles. First, it implements all the game rules and appropriately updates game state variables in response to game actions. Second, it is the glue code that puts all the other pieces together. It uses and depends on most of the other classes.

Presentation Layer The `GameUI` class is the front end, or presentation layer, for the program. It displays the game information to the user and gets the user input on what actions to take. Displaying the railroad map is delegated to `MapPanel`, a custom panel that knows how to draw routes between cities on top of a map. The `GameUI` object provides methods that allows the Game Object to react to user actions and to change what is shown to the user. The presentation layer uses the railroad network as input, so it has a dependency on the data structure storing this information (see below).

T2R Game Concepts Several small classes are used to represent game concepts and pieces. The primary role of most of these classes is to store necessary data values while abstracting the details of how the information is stored.

1. `Ticket` objects correspond to destination ticket cards in the game.
2. The enumeration `CardColor` defines what colors train cards can be. If you are not familiar with enumerations in Java, think of an enumeration as a set of named constants.[2] This set of colors also corresponds to the colors of the connections / routes between cities in the railroad network, so `CardColor` is also used to represent colors of connections. (This makes sense, since the train cards are used to "purchase" connections with a matching color.) Note that `CardColor.WILD` represents wild train cards (locomotives), but grey connections. Since grey connections can be purchased using card sets of any color, combining grey with wild into one enum value seems like an acceptable overloading. In practice, the context in which `CardColor` is used makes it clear what interpretation is appropriate.
3. `Route` objects contain information about the connection(s) between a pair of cities. If you think of the railroad network as a graph, these objects are the labels on the graph edges.

[1] There is nothing magical about the code we've provided though. All the necessary tools, data structures, and algorithms are topics we have covered in class. We encourage you to read through all the code and see for yourself how it all works and uses things you know. Given more time for this project, we would point you at the game rules and tell you to implement the game. And these instructions would be *much* shorter.

[2] Under the covers enumerations in Java are classes, and the particular values of the enumeration are read-only, final instances of the class. So there is a `CardColor` class, with 9 instances named PURPLE, WHITE, etc. However, because they are read-only and final, we can almost think of an enumeration as another kind of primitive type.

4. `Owner` is an enumeration for the possible owners of a connection. Legal values are any of the 5 players, or OPEN (indicating the connection has not been claimed yet).

When necessary, cities on the map are identified by name and represented using `String` objects.

Players `Player` objects take care of storing and updating information about each player, such as what cards they have in hand and their current score. `ComputerPlayer` objects have the extra responsibility of knowing how to choose an action when their turn arrives.

Railroad Map/Network The most important part of the game state is the network of rail lines connecting cities and who has purchased which portions. The interface `RailMap` specifies how the rest of the program can query information about the railroad network. This is the most central data structure in the game, and where most (if not all) of the interesting algorithms take place. The presentation layer uses the `RailMap` to draw a picture of the network and to show which connections are not yet claimed. The main game object needs it to enforce some game rules and to do the scoring for destination tickets at the end of the game. Finally, `ComputerPlayer`s use the `RailMap` while deciding which action to take.

Toolbox Utility helper methods that did not fit well elsewhere. Feel free to add more methods here if it makes sense to you.

The following subsections describe the important program modules in more detail, with an emphasis on what you need to do.

1.2.2 `RailMap` Interface

Your Responsibility

Finish implementing the class `RailMapGraph`, specifically the methods relating to graph algorithms. The rest of the implementation is given to you (and frankly, is fairly straightforward). Take note of the other methods though: you might find them useful to completing the rest of the assignment. Also, if you are unsure how to use iterators in other parts of the program, there some examples in the implementation of this class.

A little explanation about the methods you need to implement. First, the method `findPath()` checks if there is a path from city1 to city2 in the graph *only* using connections claimed by the specified player. Conceptually, this is like filtering the railroad map and ignoring every edge not owned by the player. In other words, you need to find a path in the player's subgraph.[3] Of course, this might not be possible—in which case you should return a null path. This method is called at the end of the game to check if a player completed a destination ticket so that points can be awarded or deducted. The returned path does not need to be the shortest path. We just need to know whether city1 is connected to city2 in the owner's subgraph.

Second, the method `shortestPathPossible()` needs to find the shortest path in the subgraph containing OPEN edges and the owner's edges. This will be useful to a computer player deciding how to complete destination tickets in an efficient manner. Our view of shortest path (or least cost path) follows an economics perspective. The least cost way to complete a destination ticket, given some existing edges claimed by the player, is to find the shortest edges that can connect city1 to city2 OR can connect the cities to a common component in the player's subgraph. One way to do this is to apply Dijkstra's algorithm and pretend that the length of all edges already owned by the player have 0 length. Basically, connections already claimed are already paid for, and there is no future penalty for using them.

Third, the method `longestPath()` searches for the longest path in the graph for a given owner. This is used at the end of the game to award 10 points to the player with the longest path. Figuring out how to do this will require some thought, since we have not discussed how to do this in class. Some form of graph traversal (like depth-first search or breadth-first search) will probably be needed. We are being intentionally vague about how to solve this to give you a chance to solve the problem on your own. Be sure to include comments in your code that explains the idea of whatever algorithm you use.

Please do not change the interface `RailMap`. If we find a problem with your implementation, we will swap in our own implementation to test the rest of your program. If you feel like you need to add something, please speak to a TA or the instructor first.

[3] At the beginning of the game, before any connections are claimed, calling `findPath()` for `Owner.OPEN` should be just like finding a path in the full graph, ignoring the notion of people owning edges.

1.2.3 Presentation Layer

Most of the presentation layer is already implemented in the files given to you. As stated above, the `MapPanel` class takes care of drawing the railroad network on top of the USA map. You do not need to change anything in this class.

The `GameUI` class defines the top level window and adds the appropriate controls and informational components to the window. The class constructor does all the setup work for the visual elements. Various subparts of the UI are created by private helper methods called from the constructor.

The GUI dynamics are handled at two levels. `GameUI` is responsible for reacting to non-permanent user input events (i.e. anything that does not need to change the game state). The presentation layer also enforces the game policy that a user can only take a single action per turn. To do this it uses a group of radio buttons; whichever button is selected determines which group of controls is enabled. Further, it also listens to permanent input events (e.g. drawing a card) so it knows when to disallow changing to a different action. The class uses private inner classes to implement event listeners responsible for UI behavior that is self-contained in the presentation layer.

The second level of dynamics happens in the game object (see below for details). The game object needs to know about any permanent events so it can validate the action (that is, make sure it is allowed according to the rules of the game) and update the game state appropriately. `GameUI` provides public methods that can be called to get references to the controls that should cause permanent game state changes (e.g. the confirm claim button can be accessed through `getClaimButton()`). This allows other modules to register event listeners so they can be notified appropriately.

`GameUI` also provides public methods that allow the game object to get information about inputs the user selected, while hiding the control details. For example, `getSelectedRoute()` returns the route selected by the user. Typically these methods are called by the game object while handling potentially permanent events.

The final part of `GameUI`'s public interface are methods that can be called to change what is shown to the user and to prepare the GUI for the next player's turn.

Your Responsibility

For this assignment, you will add two pieces of functionality to the presentation layer. The first is finishing the implementation of drawing extra destination tickets. The second is getting rid of the dialog prompts that get player names (coded in `TicketToRide`) and adding controls to the top right corner of the main window to collect this information AND the type of each player (human or computer).

To finish the ticket functionality, you need to implement `getTicketsInput()` so the main game object can access the information it needs. To reduce problems from user input errors, you also need to add code to disable the "This is my final answer" button when none of the check boxes are marked. That way the confirmation button can only be pressed when there are a legal number of tickets kept by the user. All of this code can be put in the `GameUI` class.

For the second task, you need to add appropriate controls to the top right corner of the main window to collect player configuration info (name and player type). You may use any controls you feel are appropriate. These controls should only be enabled before the game begins; similarly, when they are enabled the rest of the controls in the window should be disabled (the game cannot start without this information). You will need to create appropriate event listeners to achieve this, and add public methods to `GameUI` so the main object can register listeners and get the player configuration information. You may also need to add a method so the main game object can tell the UI that it is okay to start the game. Feel free to add any member variables to `GameUI` that you need.

1.2.4 The Main Game Object

The `TicketToRide` class has roughly 3 sections of code. First, the constructor is responsible for setting up all of the game state (creating the players, getting the railroad network initialized, dealing out starting hands, etc.), constructing the GUI object, and registering event listeners for events the game object cares about. All the main method in the class does is call the constructor; after that, the program is driven from the GUI by the user.

Second, there are public methods `handleClaim()` and `handleDraw()` that alter the game state in response to claim and draw (train card) actions, or return an error string if there is a problem with the requested action. You should add a `handleTickets()` method to implement the game logic related to a player drawing

Your Responsibility

extra destination tickets.

Third, there are private inner classes at the end of the file that are event listeners for game-level events like claiming routes. These are the glue between the GUI and the game logic implemented in the rest of the class. They perform some simple error checking and then call the appropriate `handleXXXXX()` method to do the real work. If there is an error they tell the GUI to show the message; otherwise, they tell the GUI it is okay to let the next player go (i.e. enable the next turn button). You need to complete the listener `TicketListenerHelper` to call the `handleTickets()` method you write. Your Responsibility

In addition to handling some of the game logic as mentioned above, we ask you to improve the user experience by giving more feedback for all the actions. Currently, there is minimal visible record showing what actions players took. For all actions, you should write action summary messages to the message pane at the bottom of the GUI. For example, this might be something like "Dijkstra picked up a RED card," or "Jay Gould claimed Seattle-Vancouver (length 1)." The messages should correspond to information everyone would have while playing the original board game. Your Responsibility

To finish adding the feature of collecting player names and types in the main window, you need to remove the code that gets player names using dialog boxes and replace it with code that gets the values from the main window. When it is collected in the main window, the information is only available after the window is shown—so storing the names and types will have to be done using an event handler. Look at how input events game actions (e.g. drawing cards) are handled by event listeners and then processed to update the game state. You probably need to use a similar programmatic structure for storing player names and types. Your Responsibility

Finally, we ask that you write the end of game code that adds (or subtracts) points for destination tickets to the player scores, awards the 10 points for longest path, and declares a winner. The hard parts of this job should be taken care of in your `RailMap` implementation. Look at the `nextPlayersTurn()` method to see where this functionality fits into the program.[4]

1.2.5 Computer Player

]

The code we provide contains nothing to implement computer players. You are free to design the computer player functionality as you see fit. Inheritance might be useful in your solution, since a `ComputerPlayer` is a `Player`. Your Responsibility

You will need to make changes to `TicketToRide` to better support the needs of a computer player. We have tried to structure the code to make this addition relatively painless, but it is unlikely we anticipated every last requirement. You should be able to reuse the `handleXXXXX()` methods from the main game object to update the game state with the computer player's action. We recommend that you never ignore non-null error strings returned from a `handleXXXXX()`. The computer never makes foolish input mistakes; it has moved past that human flaw . . . unless there is a bug in its program.

To get full credit for this part, you should implement the following computer strategy.

Dijkstra 's strategy is to complete as many destination tickets as possible using the shortest available paths in the graph. More precisely, each turn he checks the map to see what is the shortest possible path to complete each ticket. If one or more ticket is incomplete, he sees if he has enough cards to claim any of the missing connections on the paths he needs. If he does, he claims the shortest connection he can afford;[5] otherwise, he draws train cards. To decide which color to pick up, he considers the missing connections and picks up cards for connections he is closest to completing. Again, if there are ties he breaks them using the same color ordering as above . . . with one exception. Dijkstra never picks up WILD cards; if he doesn't need any of the face-up choices, he draws blind.

If all of his tickets are completed, Dijkstra's action depends on how many cars he has left. If he has at least 15 cars, he draws more destination tickets and keeps any that he has already completed (rare, but it can happen). In addition, he keeps a single uncompleted ticket—the one that has the shortest possible path needed for completion. If he has less than 15 cars, he buys the longest unclaimed connection *with length less than or equal to his remaining cars* (or draws cards to buy it).

[4] Testing hint: change the number of cars that players start with to a small number, so you can get to the end of game condition quickly.

[5] Ties are broken by giving priority to route colors that occur later in the `CardColor` enumeration. For example, grey connections (represented as WILD) are purchased over any other color.

The computer player depends on the `RailMap` to handle the graph algorithm details they need for their strategies.

2 Task Summary and Point Break Down

- Finish implementing `RailMapGraph` *(35 points)*
- Finish implementing the draw extra destination tickets action *(10 points)*
- Remove dialog boxes prompting for user names, and replace with controls integrated into main program window. *(20 points)*
- Print messages summarize each user action to message pane. *(5 points)*
- Write end of game code: count points from destination tickets and longest path, and declare a winner. *(5 points)*
- Implement a computer player and integrate into the T2R program. *(25 points)*

3 Resources

In addition to the class notes, and course textbook, you may use any of the following that you find helpful.

1. The code given to you for this assignment. No, seriously. Read through it and note how things are done. For example, there are lots of examples of arranging user interface components and registering listeners. It is likely you can use similar techniques to complete the implementation. You might try running javadoc on the files we gave you to generate HTML documentation for the methods.
2. The Java API libraries. We encourage you to take full advantage of functionality in the Java API. Need a queue? Need a dictionary? Don't spend time writing your own, look for something that fits your needs that someone else has written (and debugged!).
3. Want to store data in a graph, but there isn't one in the Java API libraries? Don't worry, we're here to make your life as painless as possible. For this assignment, you can use the Java Structures package by Duane Bailey. You can find a Graph implementation (several, actually) there. The url is:
   `http://www.cs.williams.edu/JavaStructures/Welcome.html`
   In addition to javadoc documentation for the package, there is a free online book that you can refer to as a) an additional data structures reference, and b) a guide in understanding the design and implementation of the structure package in particular. You can download the file `bailey.jar` from the website, or grab it from the demo zipfile.
4. As always, the online Java tutorials are a great resource you can use. They are particularly helpful for learning about GUI controls.

Important: the course textbook has online files for many things, including graphs. DO NOT use the online materials from the textbook that relate to graphs.

4 Submission Requirements

Place all files needed to run your program in a directory called a5_¡netid¿, where ¡netid¿ is your netID. Add this directory and all its contents to a zip file called `t2ride.zip` and upload the zip file to CMS. Do not include the .class files in the zip file; we will compile your code so they will just take up useless space. We should be able to download your zip file, extract the contents, compile the code, and run the program without hassle.

5 Extra Credit, Extra Fun!

If you have finished everything above and are craving some more programming, there are lots of opportunities in this assignment. *We will award extra credit based on how well they are done. However, extra credit is always worth less than the main part of the homework. Therefore, your time is best spent finishing the homework well and only working on extra credit afterwards.*

Make sure you keep a pristine backup copy of the solutions before you start.

5.1 Design Critique

Points: 5

The first design of a program can usually be improved. Write a short critique (1 page or less) of the T2R program structure that focuses on what parts of the design could be improved. For example, are there any leaky abstractions? You will get the most extra credit by outlining design fixes for the flaws/shortcomings you identify.

5.2 Extra T2R Features

Points: variable

There is a lot of room for creativity and extra credit on this project. Use your imagination to think of cool extra features to add. We will award extra credit points accordingly.

If you add extra features, include a plain text file named `EXTRA.txt` in your submission that (briefly) describes the features.

To get you started brain-storming, here is a laundry list of possible ideas:

- Implement additional computer players with different (better?) strategies. For example, what would a Prim computer player use for a strategy? Could a computer player count which cards everyone else is picking up and guess which routes they intend to buy? Would this give an advantage?
- Add menu items for restarting a new game, maybe saving and loading a game in progres, ...
- The staff threw this GUI together as fast as we possibly could. Some of it looks nice, but admit it: you know you could make a GUI that looks better. We'd love to see what you can do!
- In the real Ticket to Ride game, anywhere from 2 to 5 players can play. Extend the program to allow variable number of players (human and/or computer). Pay special attention to the rule restrictions on claiming routes when there are only 2 or 3 players.
- Start the game properly: deal out 3 tickets to each player in the beginning, and give them the option of discarding one. Could be interesting to see how a computer player would decide.
- Generalize the `MapPanel` to work with different maps and configuration files. In other words, remove our assumption that the cities are already drawn on the map.
- Maybe there is a way to have the GUI be resizable—including the map!
- Animate the map? *Warning, this is significant work.*
- Anything else you can think of!

Start Early and Good Luck! Have Fun!