Choose Your Own Adventure Game
Design Report

John Allard, ID : jhallard

December 5th, 2014

Contents

1 Overview

This document discusses the use and design of the cyoa (choose your own adventure) game that I have created. This game is created as part of assignment #4 for CS12 at UCSC. This program is designed to run any general game that follows a specific format, knows as the `.adventure` file format. More info on this can be found in Section 2.1, Making Your Own Game. All of these games revolve around the same principal, you start in a place (called a room), and from that room you can choose to go to other rooms. From those other rooms, you can go to more rooms, ad infinitum. All of the rooms come with descriptions and options to go to other places. A room can be anything the user defines it to be, but it just needs to represent one 'state' of the game, and it must include the option to jump to other game 'states'.

Internally, the game is represented using a directed graph data structure, with the vertices of this graph consisting of a custom Room data structure that I define. The input file is parsed and the rooms are loaded into the map. We only need to keep track of the current room, which makes game execution very simple. More about the implementation can be found in Section 3.

2 User Guide

2.1 Making Your Own Game

One of the best parts about this game is that you design it yourself! A game is defined by a `.adventure`file, which will be passed in as a command line argument when you run the cyoa program. These files are just ASCII text files that define a series of rooms and connections between rooms. This file must follow a specific design format, explained below.

All (non-comment) lines of the `.adventure` file must look like the following :

```
$Command $Argument
```

Where `Command` is one of the single characters in the table below, and `Argument` is a string to apply to that command. Any line not following the format above will be treated as a comment line.

Command	**Argument**	**Meaning**
r	Name of room	Insert a new room into the map
d	Arbitray String	Add a description to the current room
o	Arbitray String	Add an option to go to a new room
t	Valid Room Name	Set the destination corresponding to the last option

You can add as many descriptions as you like, and you can add up to 12 options per room, corresponding to the letters a-l. Note that a room must have at least one description or option, it cannot be completely empty (you can't have two lines with r commands in a row). The room can be defined in any order, and you can add a destination tag to a room that has not yet been defined. Any lines that do not start with a valid command will be ignored as comment lines. If you add a destination tag to a room that is not defined by the time the end of the file is reached, an error will be shown to the screen and the program will exit.

2.2 Playing the Game

To build the game, navigate the the directory containing all of the game source files (including the `Makefile`). Type :

```
Make
```

This will build the program (assuming all goes well). To run the program, from the same directory, type :

```
cyoa filename
```

Where `filename` is the name of the adventure file that you wish to play the game with. If everything goes well, you will begin the game in the first room defined in the adventure file that you are running. You will see a few lines of descriptions about the room, then a list of options for you to take (with anywhere from 1-12 options (`a-l`)) to other rooms. If you choose an invalid option, you will simply be prompted to choose again. If at any time you wish to go back a step, press the `z` key. This can take you all the way back to the first room in 'stack-like' fashion. To restart back at the first room (losing all data in the process), press the `r` key. To see a complete layout of the map, press the `y` key.

I have included a demo adventure file with this program, `demo.adventure`. This is a pretty large adventure file, consisting of 16 unique room and over 60 total connections between rooms! You start off at your home in Santa Cruz, CA, and make your way around the city, state, country, and even the solar system!

3 Design

3.1 High-Level

The bulk of my implementation is built around a home-made directed graph data structure (`dGraph`). This data structure is a templated, directed graph that can be either weighted or unweighted (by default all weights are 1, user can set the weight of edges if they wish), implemented in the C++ programming language. Because this graph is templated over the vertices, I was able to create a custom `Room` class to use as the vertices of the graph. This gave me an almost literal representation of the game map gathered from an adventure file, with 'rooms' connected to another 'rooms' via edges in the graph. This made traversal of the map during game play extremely simple, and allowed the actual game class (`CYOA`) very small.

The game is controlled by the `CYOA` class, which is just a simple finite state machine with a sentinel-based while-loop and a switch statement enumerating the possible user inputs. To start the game, a user first needs to pass in an input file for the class to parse. If successful, the user can call the `startGame()` function to begin gameplay. Finally, I had some other helper data-structures and algorithms that I also implemented, these are all discussed in the Section 3.3.

3.2 Gameplay

I wanted to keep gameplay as simple as possible for the `main` function. To do, I encapsulated different aspects of the game play into a nested series of classes, which in the end allows the game to be played with two simple function calls from main onto a CYOA object. Inside the CYOA object is a small set of functions that mostly deal with parsing and validating the input file, and building up the graph data structure. There is one function that controls the gameplay, and this just consists of a while-loop and switch-statement over the user input cases. All of the cases are handled in just a few lines of code, because all complicated functionality is encapsulated inside of the dGraph class or one of the other classes that I defined.

3.3 Classes

3.3.1 OOP Design Overview

I tried to solve the problem of implementing this game with an object oriented approach, and I believe it worked out quite well. The program consists of 8 custom-made classes, 5 of which

I imported from my personal Data Structures project[1]. The relationship between the various classes can be seen in the table below.

Class Name	File(s)	Description
GraphInterface	`GraphInterface.h`	Abstract Interface for a graph data structure.
dGraph	`dGraph.h/cpp`	Templated, directed graph data structure.
AdjList	`AdjList.h/cpp`	Adjacency list data structure, a linked list of vertices.
Edge	`Edge.h/cpp`	Represent an edge (link) between two graph vertices.
Vertex	`Vertex.h/cpp`	Templated vertex class, represents one node in a graph.
Stack	`Stack.h/cpp`	Simple stack class that works with the Room data structure.
Room	`Room.h/cpp`	Represents a single game room, template arg. for Vertex class.
CYOA	`CYOA.h/cpp`	The class that encapsulates all of the gameplay.

The first 5 listed above are brought in from a personal project, the other 3 I made specifically for assignment 4. I also had to go into the dGraph class that I had made and change some functionality around to fit the specifications. All of the classes listed above will be discussed in more detail in the following sections.

3.3.2 Data Structures Project

This is a project I started sometime around early November of this year to practice for upcoming technical interviews. The project can be seen at `https://github.com/jhallard/DataStructures`. As of right now I am only working on the graph section of the data structures project, this includes working on both a directed and undirected graphs in both an adjacency list and adjacency matrix representation. As of right now, only the adjacency list representations of both the directed and undirected graphs are finished (well, 95% finished, there are always more tests to run). For the CYOA program, I am borrowing the dGraph class, it's parent (GraphInterface), and all of the other self-defined classes contained inside of it.

3.3.3 GraphInterface

File(s) : `GraphInterface.h`
The GraphInterface class is exactly that, an interface that the real graph objects in this project must implement. This is done to promote good design, and guarentee a consistent and well-planned interface throughout the many flavors of graphs implemented. This is an pure virtual class, which is C++'s way of supplying interface functionality. This means that this class will not compile on it's own, it contains functions that must be implemented by a child class. Overall, this class consists of the basic graph functions : `insertVertex, deleteVertex, findVertex, insertEdge, deleteEdge`, as well as other functions like `breadthFirst, depthFirst, isConnected, minimumCut, minimumSpanningTree, dijkstras` which implement some of the more common graph algorithm for the user.

3.3.4 dGraph

File(s) : `dGraph.h, dGraph.cpp`
This is the directed graph class. It publically inherits from the GraphInterface class, as do all other graph flavors in this project. This class implements all of the functions contained in the GraphInterface class, plus a few extra ones (implementation specific private functions). This class uses a series of adjacency lists to represent a graph data structure. An adjacency list consista vertex and a list of edges that eminate from this vertex to the other vertices in the map, along with the weight associated with those edges. Thus if some vertex v had edges

[1] `https://github.com/jhallard/DataStructures/tree/master/Graphs/GraphAdjList`

connected it with vertices a, b, c, d, and e, the adj list for vertex v would look like :

$$v \rightarrow a^* \rightarrow b^* \rightarrow c^* \rightarrow d^* \rightarrow \text{null}$$

Where x* is an edge that leads from vertex v to vertex x. Thus to find all of the edges that eminate from a given vertex v, you just need to traverse the list of edges on the AdjList that contains vertex v. This above is just one adjacency list, our graph will have a single adjacency list for each vertex in the graph. So our graph data structure will look more like this :

$$\begin{aligned}
&a \rightarrow c^* \rightarrow d^* \rightarrow \text{null} \\
&b \rightarrow v^* \rightarrow d^* \rightarrow e^* \rightarrow \text{null} \\
&c \rightarrow a^* \rightarrow v^* \rightarrow e^* \rightarrow \text{null} \\
&d \rightarrow a^* \rightarrow b^* \rightarrow \text{null} \\
&e \rightarrow b^* \rightarrow c^* \rightarrow v^* \rightarrow \text{null} \\
&v \rightarrow b^* \rightarrow c^* \rightarrow \text{null}
\end{aligned}$$

This implementation is accomplished through a series of nested compositions of classes. The dGraph class contains an array of adjacency lists. Each adjacency list contains a vertex, and a linked list of edges. Each edge structure contains a pointer to a vertex object, and an optional weighting value. Finally, a vertex is just a templated data structure meant to wrap around whatever type data the user wants the graph to contain.

3.3.5 AdjList

File(s) : `AdjList.h, AdjList.cpp`
The AdjList class is basically a convenient wrapper around a pair of objects; a vertex, and a first node in a linked list of edges that eminate from that vertex. All of the functions it contains serve to encapsulate this functionality and provide a clean interface for use by the dGraph class.

3.3.6 Edge

File(s) : `Edge.h, Edge.cpp`
The edge class is a simple class that represent a single node in the linked list of edges contained in AdjList. This class contains 3 main items, a pointer to a vertex (the vertex it leads to), a pointer to the next edge in the list of edges, and a weight value that can be optionally changed by the user but defaults to a value of 1.

3.3.7 Vertex

File(s) : `Vertex.h, Vertex.cpp`
This simple class serves as to wrap around whatever data data the user wishes the graph to contain. It encapsulates this data with standard get/set functions and that is pretty much the extent of it's use.

3.3.8 Stack

File(s) : `Stack.h, Stack.cpp`
This is a very simple stack implementation that I made specifically for the CYOA project. This stack works on data of the Room class type, which is another type defined by me (see next subsection). This stack contains the basic functions, `push, pop, peek, size`, and that's about it. It is implemented internally using a dynamically allocated array (using `new/delete`), that grows as the user demands more space. This stack is used by CYOA class to keep track of the history of room visits performed by the user. We can then perform backtracking with the stack, allowing the user to go back to previous rooms.

3.3.9 Room

Files(s) : `Room.h, Room.cpp`
This is a data structure that represents a single room in the CYOA game. This room consists of the following items : a name, an array of description strings, and an array of option strings. These arrays are dynamically declared and resized as needed internally. On top of this, in order to use this class as the vertex of the dGraph class, I also needed to implement a copy constructor, equals operator, and a hash function. These are all used internally by the dGraph class when it is moving around and manipulating the graph.

This class is instantiated during the file parsing phase of the CYOA program. When we find a room tag that we haven't seen before, we create a new Room object, and fill it with the appropriate data. Once the data is loaded, we insert it into the graph object as a new vertex, and add edges with all of the rooms enumerated with option tags.

3.3.10 CYOA

Files(s) : `CYOA.h, CYOA.cpp`
This class is the main game control class for the program. CYOA encapsulated the FSM[2]-like functionality of the game behind 2 function calls, `parseFile` and `startGame`. Before the game start, the `parseFile` function must be called with the file name of the `.adventure` file passed in as a parameter. If this function returns true, then the `startGame` function can be called to run the game.

This class contains only a few members, including an instance of the following classes : dGraph, Stack, Room. The purpose of this class is to build, traverse, and manipulate the graph object as requested by the user. The current room is kept track of, and states of the game are represented by going to different rooms. All of the control logic is handled inside of a while-loop and switch-statement in the `startGame` function. X

3.4 Bugs/Enhancements

[2]finite-state-machine