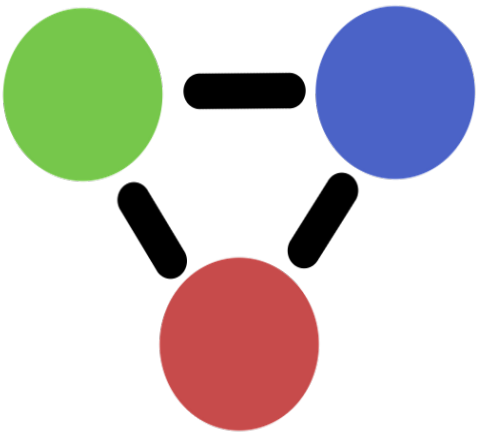
Graph Puzzle Game



BA3 Programming Assignment Documentation

by Arthur Kehrwald

Introduction

This is the game concept I wrote for my application to CGL. I always thought it was a neat idea, but never got around to programming a prototype. The assignment asks for a text adventure, but given this concept's inherent connection to graphs, I think it is just as good an excuse to practice the use of data structures. I also decided to program in C++ rather than C#, because the other obligatory assignment this semester is creating a point and click engine in C++ and I wanted something easier than that for my first major project in the new language.

Description

"Graph Puzzle Game" is implemented as a console application for Windows. The player chooses a difficulty and is then presented a procedurally generated graph. Due to the limitations of console output and because the user interface wasn't a focus, it is visualized as a table (fig. 1), with crosses indicating edges between the coloured nodes. The nodes can be thought of as buttons that change the colour of all connected nodes -but not their own- when pressed. Red nodes turn green, green nodes turn blue, and blue ones go back to red. The goal is to manipulate the graph such that all nodes have the same colour.

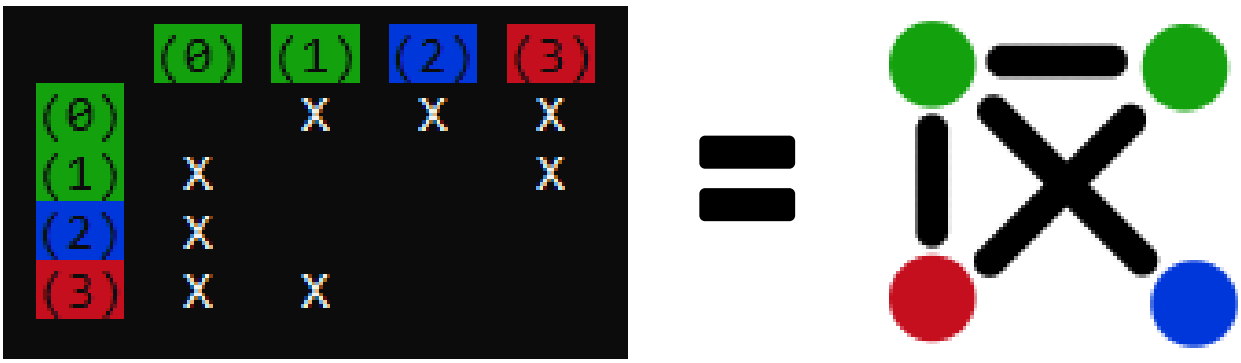


fig. 1 - Ingame matrix visuals and equivalent graph

Class Structure

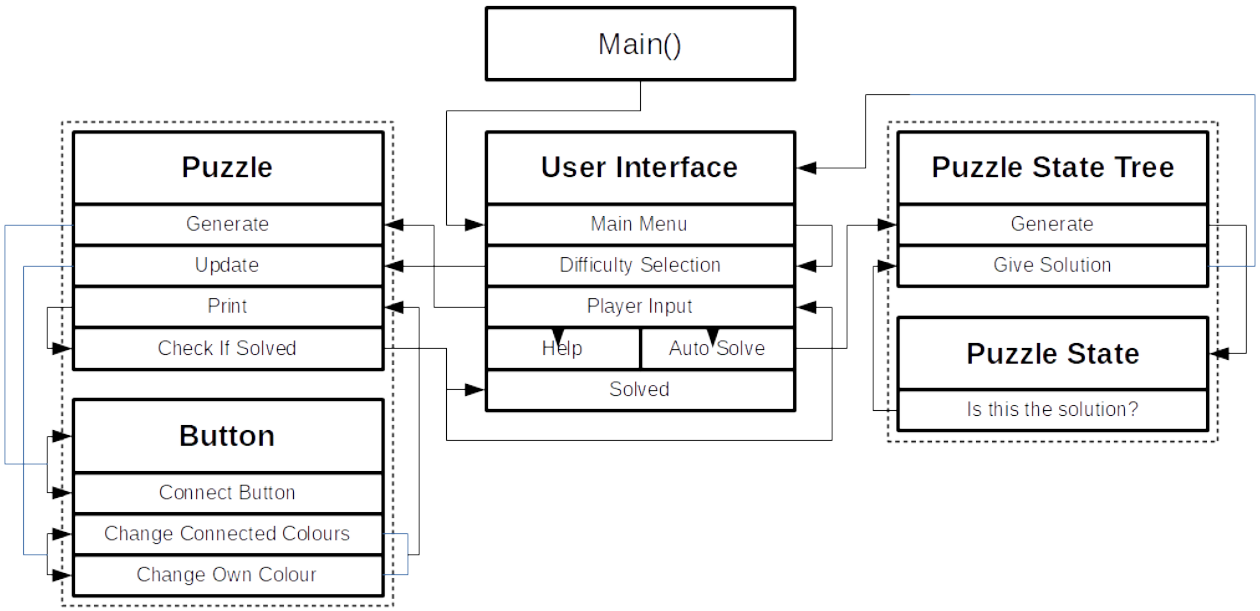


fig. 2 - Class Structure Diagram

User Interface

This is the class control is passed to from the main function in the beginning. It displays the main menu, then allows customization of the difficulty. Once the player has made a choice it passes the according parameters to the constructor of the puzzle class to create a puzzle. It then handles the input output loop while the player tries to find a solution, offering the options to skip or solve the puzzle automatically, as well as to display the games rules and adjust the difficulty.

Puzzle and Button

These classes form the core graph data structure of the game. The graph contains a vector of button objects that each hold a vector of connected buttons. I chose the vector for this purpose because I need fast access to specific buttons to handle user input and can't use a traditional array because the number of buttons is unknown at compile time.

The puzzles are generated using two parameters received within a struct from the user interface after difficulty selection. There are three preset difficulties, but the player can customize each parameter individually. Once the parameters are passed to the constructor, the puzzle is generated in four steps:

1. Create the desired number (first parameter, 2-10) of button objects and randomly set all of them to either red, green, or blue.
2. Connect all buttons in a line to ensure that the resulting graph is connected (the vector is shuffled before to hide this repetitive pattern from the player).

3. Loop through each pair of buttons that is not already connected once and create an amount of connections relative to the connection probability parameter (second parameter, 0–1).
4. Randomly press each button either once, twice, or not at all. (Since any button press can be reverted, the resulting puzzle is guaranteed to be solvable.)

Puzzle State Tree / Auto Solve

This class can solve any graph puzzle (at least up to the biggest allowed size in this implementation, 10 nodes). It now uses a mixture of brute force and predictive behaviour to do so. I started by generating a tree in level order containing states of the puzzle along with a series of inputs that lead to each state until a solution was found. Because loops are possible, I needed a way to avoid following paths that have already been tried before. To do this, I added each state to a list, and compared each newly created state to every other state in that list. If a match was found, the state was discarded. This worked, but was too slow to solve big puzzles in an acceptable amount of time, so I started looking for ways to optimize the process. The most expensive part of the algorithm, even more so than the sheer number of generated states, was the comparison of states, because it had to be done many times for each generated state. After following my algorithm by hand a few times, I made two useful observations:

- Sometimes, two or more nodes are connected to the same set of nodes. This causes them to have the same effect when “pressed”
- The order in which buttons are pressed does not change the outcome. Pressing a button effectively adds one to a set of values and since the result of an addition is not affected by order, neither is this puzzle.

Based on these observations, I set up three conditions by which redundant states can be identified simply by the series of inputs that leads to them, rather than direct comparison, eliminating the need to even create these states to begin with. I also changed the structure in which inputs are stored for each state from a vector to a dictionary containing the number of times each button has been pressed since I am no longer interested in the order.

- Only simulate an input if there is no button with a lower index that has the same effect.
- Don’t press any button more than twice. (Since the order doesn’t matter, as soon as we’ve pressed a button twice in a series, pressing it again would be redundant.)
- Don’t press buttons with an index lower than that of the button with

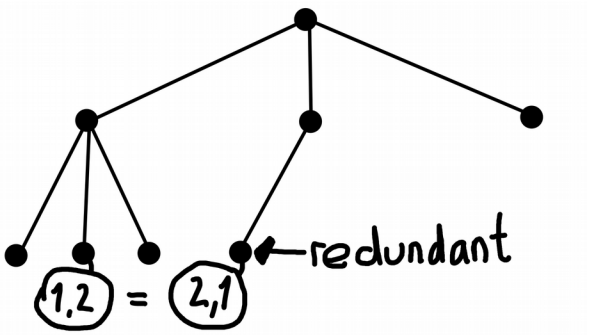


fig. 3 - example of condition three finding a redundant state

the highest index in the series of previously pressed nodes. In other words, press buttons in ascending order. This works because the tree is created in level order so buttons with lower indices are always checked first. If for example (fig. 3), 2 was pressed, and the algorithms is about to check what happens when 1 is pressed after that, we know that the sequence 1,2 will have already been checked, and the simulation of input 1 can be skipped for this state.

Together, these conditions eliminate all redundant states for most graphs. In some graphs, there is a more complicated kind of node equivalence where not two nodes have the same effect, but a set of nodes have the same effect as another set of nodes. For example, the combined connections of buttons 1 and 2 may be the same as the combined connections of buttons 3 and 4, making the input sequence 1,2 equivalent to 3,4. If you uncomment the "#define DOCOMPARISON" line at the top of my state tree code file, you can see for yourself when this happens. Start the game, generate a graph, and auto solve it. You will get debug information about how many redundant states were discarded by the aforementioned prediction rules and how many passed those conditions and were picked up by direct comparison. When solving large puzzles, you will also notice the algorithm is much slower. It turns out that unnecessarily generating a handful of states in some graphs is still much faster than making sure it doesn't happen using comparison. Compared to the amount of unique states generated, there are not enough of these states to significantly slow down the algorithm anyway, so I have decided to leave them in for now.

Conclusion

Apart from the above minor issue with the auto solver, I only had problems with one other task: The graph generator does not produce puzzles of consistent difficulty. Despite writing an additional class that generates and solves graphs in bulk rather than one by one and puts out averages and mean deviations of the amount of steps it took for each, I was only able to control the average complexity of my puzzles through parameters, but couldn't bring down the deviation.
The rest of my work went very well, taught me a lot about data structures and produced a game that I enjoy playing. I also don't regret working in C++ as I badly needed the practice for the upcoming engine assignment.