CPSC 221, 2015W2
Programming Project 1
A MAZE(ing) Solver

Early Bird Bonus Due: 9PM Mon 2016-Mar-7

Final Submission Due: 9PM Mon 2016-Mar-14

Note that both 9pm deadlines are subject to the same late penalty as on the written assignments: divide the number of minutes late (past 9pm) your assignment is by 5, take the floor function, and then raise 2 to the resulting power. That number out of 64 is the fraction of points you'll lose. E.g. if you're 10 minutes late, you lose 6.25%, but if you're 25 minutes late, you lose 50%.

Objectives

- Get used to building C++ programs in our UBC environment.
- Work through the details of stack, queue, and heap implementations.
- Understand depth-first-search (DFS), breadth-first-search (BFS), and best-first-search (BestFS).
- Be aMAZEd while your program solves mazes, and more!

Overview

Searching is an extremely general approach to solving problems: we systematically explore possible solutions until we find one that works. Thanks to the speed of the computer (and efficient algorithms like what you learn in 221!), search is able to solve a wide range of important, practical problems. In this project (and the next one), we'll develop some reusable code that lets you solve mazes. Plus, with just a bit of coding, you can think of many other problems as mazes, so the same code can solve other problems, too. To give you some fun examples, we've provided code for Sudoku and slider puzzles as if they were mazes. (If you're unfamiliar with these puzzles, there are many on-line implementations. Here's one for the sliding puzzles, and one for Sudoku. But, beware of spending too much time playing the puzzles!) With only a small amount of coding, you can make the code solve other puzzles and problems, too.

Of course, these are just simple mazes and fun puzzles, but the same basic approaches are used in real life to solve problems like robot motion planning or automatically finding bugs in security protocols (among many others).

In this project, you will implement the stack, queue, and priority queue code at the heart of three of the most fundamental and important search strategies in computer science: depth-first search (DFS), breadth-first search (BFS), and best-first search (BestFS).

- DFS involves following a sequence of moves as far as we can go, until we get stuck. When this occurs, we go back to the most recent place where we had a choice and try a different move instead. For example, if you were looking for a washroom in a big shopping mall, you might just pick a direction and start walking. When you get to an intersection, you'd pick one direction and keep going. If you ever hit a dead-end, you'd go back to the most recent intersection you were at, and try a different direction.
- BFS proceeds differently: it tries moves in order of their discovery, beginning at the starting point of the search. First it tries all moves available from the starting point, then it tries all moves available starting from where those first moves end, i.e., "two steps away". It continues in the same fashion from there until a solution

is found. Because of this, BFS has the nice property that it will discover the shortest path to a solution, in terms of the number of moves taken to reach the solution. (It has the nasty property that it will visit every closer state of the puzzle along the way.) Returning to the shopping mall analogy, BFS would be like searching everything within 10 meters of your starting point, then everything within 20 meters of your starting point, etc. When you find a washroom, it will be the closest one to where you started.

- BestFS uses a function that assigns a priority or quality to each place in the maze that you've been. It explores the places that are one step away from the previously explored state that is the most promising (best priority value). This hopefully helps guide the search to explore possibilities that are more likely to lead to a solution quickly. Again, back to the shopping mall, suppose you are hungry and are looking for a restaurant. If you follow the smell of food, you can always search from wherever the smells are the strongest and most appetizing. This is likely to help you find the food court much faster. Note, however, that following the smell (or in general, the heuristic priority function) works well only if the heuristic accurately reflects a path to your goal. E.g., following the food smells in a mall might not lead you to the food court, but to a staff entrance to the kitchen, or even to the exhaust vent outside! But with a good heuristic, BestFS works very well.

If you think about it, DFS, BFS, and BestFS are actually all doing the exact same thing, except with different data structures: each algorithm keeps track of places its been, where there are still unexplored paths to try, and whenever the algorithm gets stuck in a dead-end, it goes to one of those places and tries a different direction. DFS is based on a stack, because you're always returning to the most recent choice you made (so it's last-in-first-out); BFS is based on a queue, because you are always exploring states in the order in which you discover them (first-in-first-out), and BestFS is based on a priority queue, because you always go to the state that has the best priority value. Let's use "bag" to refer to the stack, queue, or priority queue, and we'll use "add " to refer to the operation that adds something into the bag: "push", "enqueue", or "insert"; and we'll use "remove" to refer to the operation that takes something out of the bag: "pop", "dequeue", or "deleteMin". In this way, we can code up DFS, BFS, and BestFS using the same code.

All three algorithms begin by adding the initial puzzle state to the bag of active states. Then, until the bag is empty or a solution is found, they take a state out of the bag. If that state has not already been explored (i.e., previously removed from the bag), they find the possible next states (after one move) and put each one of these children into the bag.

Note that except in special cases (like Sudoku), we need a second data structure in addition to the bag. This is a dictionary ADT that keeps track of what states we have explored during our search. We don't want the algorithm to re-explore any state (this wastes effort and might lead to infinite loops). In addition, we usually want more than just knowing there is a solution; we also want to see a sequence of moves that led from the initial state to the solution. If we use the dictionary to store for each state the predecessor state from which we arrived at the state, then once we find a solution, we can follow these predecessors backwards and find the sequence of states that led to the solution. Note that this dictionary will be used a LOT during the search, so it should be efficient.

BTW, to see the connection of puzzles to mazes, you can think of a maze as a puzzle, where you start somewhere, and have to find your way to the goal. Conversely, you can think of a puzzle as an abstract maze: every possible configuration of the puzzle is a "place" in the maze, and one "place" is connected to another "place" if the puzzle lets you make that move. E.g., in Sudoku, filling in a number in a square is basically a step that moves from one "place" (the board before you wrote the number in) to another "place" (the board after you wrote the number in. Similarly, in the Slider Puzzle, sliding a tile into the hole is like moving from one "place" (one configuration of the tiles) to another "place" (the new configuration of the tiles).

Assignment

Note that this assignment has an Early Bird Bonus! These are just some simple questions, which are meant to be easy, but they require you to get the files and get them to compile and run, which can sometimes take a long time if you run into technical difficulties. So be sure to get started as soon as possible, so you can get help if needed! Directions for what and how to submit your Early Bird Bonus and final submission are below. (Why the Early Bird Bonus? Because in the past, a lot of folks waited until so late that they never managed to compile the code in time, and they ended up getting zeroes on the assignment!)

For this assignment, you will try solving mazes with the active states bag being a stack, queue, and priority queue. We have provided two implementations of the stack ADT (which you can use as examples to help you with your programming). You must implement the queue ADT using a circular array (and supporting dynamic resizing) as well as using a linked list. And you must implement the priority queue ADT using a binary heap

You will use your implementations, and the ones that we have supplied, to experiment with solving some mazes and puzzles. In particular, you can try out your code by using the supplied `solve.cpp` program, along with its associated files. (However, note that just as you were taught in 110 and 210, it's important that your code actually implement the Queue ADT and Priority Queue ADT properly, so you should write your own unit tests. In previous terms, some fraction of students always gets very low marks even though their code seemed to work OK with solve.cpp, because their code failed the torture tests we use for marking. Do your best to make sure your code really works correctly, even for corner cases!)

You also must answer some questions in the HANDIN.txt file.

If you choose to do the Early Bird Bonus (and you should as it's an easy way to get points, plus it helps make sure you get started on the project earlier), you need to answer some questions in the EARLYBIRD.txt file, and hand them in by the Early Bird Bonus deadline.

Important implementation constraints: In your circular array queue implementation, you must use regular C++ arrays. You may not use `std::vector` or other classes that directly support resizing. (We want you to learn how to do it yourself!) We have provided a simple, bare-bones linked-list node type and an example of using it in the LinkedListStack implementation. You are free to use this for LinkedListQueue, or you may implement your own linked list structure. You may not use `std::list` (which is implemented as a linked list) or any other linked list libraries. For the priority queue, you may not use `std::priority_queue` or any other heap implementations; you must implement your own. However, you are allowed (and strongly encouraged) to use `std::vector` in your heap implementation to make resizing the heap easy. (However, you may not use the make_heap, push_heap, pop_heap, etc. methods on std::vector.)

Important marking constraints: We will mark your code semi-automatically. Therefore, you must ensure that your code will compile and link correctly by simply running `make` on the ugrad Linux servers, with our supplied Makefile. Also, your code must respect the supplied interface types, with no modifications.

You are responsible for deallocating any memory that you allocate. (The rule of thumb is: if you create something with `new`, you should destroy it with `delete` when you've finished with it. If you used `new` with brackets to make an array, use `delete []` rather than just `delete`.)

Teams

You may work on a team of at most two on this assignment, subject to the course's Academic Conduct policy, and we strongly encourage that you do so. We recommend that you work (literally) together as much as possible. You must: (1) document each team member's effort in the `HANDIN.txt` file; (2) work together on and *both* understand your `HANDIN.txt`; (3) understand how your team's code is structured and why it works. Remember to test your team's code as an integrated whole! Except in extreme cases, all team members will receive the same grade for the project. Check out All I Need to Know about Pair Programming I Learned in Kindergarten for advice on working in a pair.

What we've provided

As a starting point we've provided some files:

Makefile
: The command `make` will use this to build your program. You should not need to modify the makefile, even if you add new `.cpp` and `.hpp` files. DO NOT MISS the following commands: `make handin-earlybird` hands in your Early Bird Bonus. Run it early and often so you're not late. `make handin-proj1` hands in your final project 1 submission (run early and often). These commands must be executed on our undergrad linux servers, in a directory that contains the Makefile, and the relevant EARLYBIRD.txt, HANDIN.txt, and various .cpp and .hpp files you wish to hand in.

EARLYBIRD.txt, HANDIN.txt
: Key documentation files for your early bird and final submissions. Read these and fill in the TODO items! Also, answer the questions there.

solve.cpp
: This is the main code that solves mazes (and puzzles). You will edit this code to choose which maze or puzzle to solve, set up the initial configuration, and select which search strategy and dictionary implementation to use. Initially, you'll just make a few changes to it to explore the different search algorithms. Later, you'll alter it so it uses your data structures rather than ours.

BagOfMazeStates.hpp
: This defines the abstract type which your stack, queue, and prioritie queue must implement. You can look at the provided implementations of the ArrayStack and LinkedListStack to see how to do this. Your queue, and priority queue classes must be subclasses of BagOfMazeStates.
Warning: Don't go about inserting logic for depth-first, breadth-first, or best-first search into your code. It's entirely unnecessary. The searches use the *same* algorithm; the only difference is a different underlying data structure for the active states.

ArrayStack.hpp, ArrayStack.cpp
: An implementation of a stack using a (resizable) array. Study it carefully and use it as a guide to implement the rest of your data structures! It will especially be useful for your array-based queue.
Warning: your `ensure_capacity` for the queue will necessarily look somewhat different from the one in this file!

LinkedListStack.hpp, LinkedListStack.cpp
: An implementation of a stack using a linked list. Study it carefully and use it as a guide to implement the rest of your data structures! It will especially be useful for your linked-list queue.

PredDict.hpp
: This defines the abstract type which the dictionary must implement.

LinkedListDict.hpp, LinkedListDict.cpp
: A naive implementation of a dictionary that scans the list for each `find` operation.

LinkedListQueue.hpp, LinkedListQueue.cpp
: Skeleton code for the linked list queue files. Implement it!

ArrayQueue.hpp, ArrayQueue.cpp
: Skeleton code for an array-based queue class. Implement it!

MazeState.hpp
: An abstract class to represent a maze position.

Sudoku.*, SliderPuzzle.*
: Code that solves these puzzles by implementing the `MazeState` interface, and lets us solve different puzzles using the same code for the rest of the program.

All of the above files are included in this zip archive: project1-files.zip

Type `make` in that directory, and then run the `solve` program. Start playing with `solve.cpp`.

Early Bird Bonus Handin

This project includes an Early Bird Bonus, due before the final deadline. For this bonus, submit your completed `EARLYBIRD.txt` file, as directed in that file.

Turn your file in electronically using `handin`, by running make handin-earlybird in a directory that contains the relevant file (EARLYBIRD.txt). (As noted above, you must run this on our ugrad Linux servers, in the appropriate directory.)

Final Submission

You are required to hand in your project1 directory with your implementation of the ADTs. Included in this directory should be at least these files:

- LinkedListQueue.{hpp,cpp}, ArrayQueue.{hpp,cpp}, HeapPriorityQueue.{hpp,cpp}
- Your completed HANDIN.txt file

You should not need to hand in changes to any other files in the project.

Turn your files in electronically using `handin`, by running make handin-proj1. (As noted above, you must run this on our ugrad Linux servers, in the appropriate directory.)

Last updated: 2016-02-28 18:13:39 cs221