

Problem Set 7

Due before midnight on Tuesday, April 1, 2008 (no kidding!)

1 Assignment Goals

1. Use the partition data structure (union-find) in a real application.
 2. Gain experience in dealing with a program of substantial size and several interacting parts.
-

2 Assignment

This assignment is an extension of Problem Set 6 in which you implemented a graphics-based Game of Hex. In that assignment, the player could make and retract moves by clicking with the mouse. The machine would check the move for legality and then take the appropriate action, updating the board accordingly.

In this assignment, you are to complete the game implementation so that the machine will also detect when one player or the other has won the game, announce that fact, and then terminate the game.

I have extended the controller module to display a dialog box announcing the winner of the game and then quit. In order for it to know when the game is over, I have augmented `game.h` with a new function `getStateGame()`. It returns a value of type `GameState` that indicates whether the game is still in progress or not, and if the game is over, who the winner is. You will need to add a definition for `getStateGame()` to your implementation file `game.c`. Of course, in order for this function to produce the correct answer, you will have to make several other modifications to your code which I have outlined in the next section.

3 Method

The game of Hex is over when a player succeeds in connecting the player's two goal sides with an unbroken path of tokens of that player's color. The left and right sides of the board are Red's goals; the top and bottom are Blue's goals. Because a winning path bisects the board, it is not possible for both players to have simultaneously won. A more complex argument allows one to establish that every Hex game ends in a win for one player or the other; it is not possible to fill the Hex board such that neither player has a winning path.

It is not obvious how best to detect when the game is over. While one could run some sort of backtracking algorithm to search for winning paths after every move, this would be quite inefficient. Each play makes only small changes to the connectivity of the board. We can exploit this locality to greatly improve the efficiency of detecting game termination.

3.1 Connected components

Two hexagons are said to be *adjacent* if they share a side. Two hexagons of the same color are *connected* if there is a path of adjacent cells from the first to the second such that all cells on the path are that same color. A *component* is a maximal subset of cells such that every pair of cells in the component is connected. We do not consider empty cells to have a color, so no empty cell can be connected to any other cell, and every empty cell is its own singleton component.

Any partially-played Hex board can be uniquely decomposed into a collection of connected components. Red wins if there is a Red component that touches the left and right borders. Similarly, Blue wins if there is a Blue component that touches the top and bottom borders.

Our strategy for detecting game termination is to keep track of the connected components as the game progresses and the borders of the board that they touch. The game is over when a Red component touches both of Red's goals, or a Blue component touches both of Blue's goals.

Initially, all of the cells are empty, so each is a singleton component. When a cell is played on, it acquires a color. At that point, it and all of its neighboring components of the same color are joined together into a new bigger component.

We associate with each component four Boolean flags that are true or false depending on whether that component touches the left, right, top, or bottom borders, respectively. Initially, the components of the cells touching the left border have their "left" flags set, and similarly for the top, right, and bottom borders. Since the four corner cells each touch two borders, they each have two flags set. The flags of the new component that results from joining two components together is just the Boolean "or" of the corresponding flags of the two constituent components.

3.2 Partitions and union-find

We now turn to the problem of finding a data structure for representing and updating the components as the game progresses.

The connected components partition the cells of the hex board. Abstractly, a partition P is a collection of disjoint subsets of a finite set U (called the universe) that covers U , that is, the union of the sets in P is exactly U .

A *union-find* data structure supports two basic operations on partitions:

$find(x)$ Finds and returns the set $S \in P$ such that $x \in S$.

$union(S_1, S_2)$ Assumes that sets $S_1, S_2 \in P$. Removes S_1 and S_2 from P and replaces them by their union. In other words, if P is the partition before the union operation and P' is the partition afterwards, then $P' = P - \{S_1, S_2\} \cup S_1 \cup S_2$.

3.3 Implementation of union-find

The basic union-find algorithm represents a partition P by a forest (= set of trees) whose nodes correspond to the elements of the universe U . Each set $S \in P$ is represented by a tree in the forest whose nodes correspond exactly to the elements in S . We take the root of the tree as a representative for the entire tree (and for the set $S \in P$ to which it corresponds). Thus, two elements are in the same set S if they belong to the same tree, and two elements are in the same tree if they are represented by the same root. The root is also a natural place to store additional information pertaining to the entire component (such as the border flags in the Hex Game application).

To perform $find(x)$, one starts at the node of the forest corresponding to x and walks up the tree to the root, which represents x . Now assume that x_1 and x_2 are roots of distinct trees representing

sets S_1 and S_2 respectively. To perform $\text{union}(S_1, S_2)$, we can either attach x_2 as a son of x_1 and x_1 becomes the root of the union, or we can attach x_1 as a son of x_2 and x_2 becomes the root of the union. Either way, whichever node becomes the son of the new root is itself no longer a root, and any information about the component that it formerly represented must be propagated to the new root.

Because we only need to walk up these trees and not down, we can represent them simply by placing a single “parent” link in each node. This link points to the node’s parent if it has one, and it is NULL if the node is a root. Performing a *union* is then just a matter of setting the parent link of one root to point to the other.

To keep trees from becoming tall and skinny, one prefers to attach the smaller tree to the larger one when performing a *union*. To implement this optimization, one stores the size (number of nodes) of each tree at the root. After a *union*, the new size is simply the sum of the sizes of the two trees that were unioned together, so this quantity is easily maintained as the algorithm progresses.

3.4 Undo/redo

Since the partition is updated incrementally after each play of the game, a way must be found to revert the partition and its associated border flags to its former state when taking back a move. A way to do this is to create a change record corresponding to each *union* operation that allows the operation to be undone. A *union* makes three changes to the partition data structure described above:

1. It sets the parent of some root x to point to another root y .
2. It updates the weight of y .
3. It updates the border flags associated with y .

To undo these three operations, one must know x and y as well as the former values of the three fields that are changed. Note however that some of these values can be reconstructed and do not need to be saved: the former value of the parent link of x is NULL since x was a root prior to the *union*, and the former weight of y is just its weight after the union less the weight of x .

These change records should be pushed onto a stack after each move and popped from the stack for each *union* that needs to be redone.

In addition, one needs to keep a change record for each move, as was done in Problem Set 6, that allows that move to be taken back. These records should also be pushed onto a stack after each play and popped off for each undo operation.

It’s convenient to have two different stacks, one for the union change records and one for the move change records, both because these records contain different information and also because they are not in sync—one move can cause several unions to be performed if the played-upon cell has several neighbors of the same color.

4 Programming details

I strongly suggest solving this problem in two stages. First, implement the union-find data structure and border flags and get the game termination logic correct. Then add the undo/redo code and get it working. Save the results of the first phase so that you have something to submit if you fail to get the second part debugged.

4.1 Representation issues

One might be tempted to represent the partition by adding fields to the existing `HexCell` data type. While this could be done, it would require you to modify `hexboard.h` and `hexboard.c`, which I do not want you to do. You should feel free to modify `game.c` and to add new modules to the project, but do not change the existing `hexboard`. I'm stipulating this for two reasons. First of all, one never wants to alter working code unless absolutely necessary. Secondly, the notion of a partition is logically quite separate from the structure of a Hex board, and it's cleaner to keep the code that deals with these two aspects of the game separate.

Instead, you should build a parallel data structure to represent the tree nodes in the union-find algorithm. There is a one-to-one correspondence between tree nodes and hexagons which you will need to maintain. The easiest way to do this is to define new types for `Node` and `Forest`, analogous to the types `HexCell` and `Hexboard` in the `hexboard` module. Then each element of the universe can be identified by its row and column numbers, and the tree nodes are represented by pointers to `Node` objects. To fully modularize the partition code, I suggest you create files `partition.c` and `partition.h` and make `Node` an opaque type (i.e., pointer to a node structure) rather than having it be a simple `struct` type like `HexCell` is.

4.2 Furnished files

You will find updated versions of `controller.c`, `game.h`, and the skeleton `game.c` file in the Problem Set 7 assignment directory `/c/cs223/assignments/ps7`. As with Problem Set 6, my controller can be linked with your `game.o` file to produce an interactive graphical version of Hex called `ghex`. I have included an executable file `reference` built from my solution so that you know what to expect to see when your code is working.

As a debugging aid, I have also included a file `tester.c` and an extended `Makefile` that will build an executable file `tester`. This program calls your functions in `game.o`, just as the graphical interface does, but its source of inputs is an input file rather than user interaction. Also, it displays the board using the `displayBoardHex()` function in `hexboard.c` so that it can be run from a command shell and does not require a windowing system. This allows you to prepare unit test scripts for use in debugging without having to make long sequences of mouse clicks on each trial. It also makes it easier to find your memory management errors with `valgrind` since you will not be distracted by the memory errors that `valgrind` reports deep in the `gtk` and `X` packages. I have supplied a few initial test scripts with names ending in `.in`.

`tester` takes the name of a game data file as a command line argument. The data file consists of a sequence of command lines. The commands are single letters: 'p' for play, 'u' for undo, 'r' for redo, and 'q' for quit. 'p' is followed by two real numbers representing the coordinates of a mouse click on the board in the form expected by `playGame()`. 'p' invokes `playGame()`, 'u' invokes `undoGame()`, 'r' invokes `redoGame()`, and 'q' terminates the program. The program also quits after `getStateGame()` reports that the game has been won. The board is displayed using character graphics after each call to `playGame()`, `undoGame()`, or `redoGame()` that returns `true`, indicating that the board has changed.

5 Submission

You should submit your source files and `Makefile` as usual along with unit test input files for `tester`. For this assignment, please submit all of the source files necessary to build your project, including copies of any of my files that you use but did not modify.