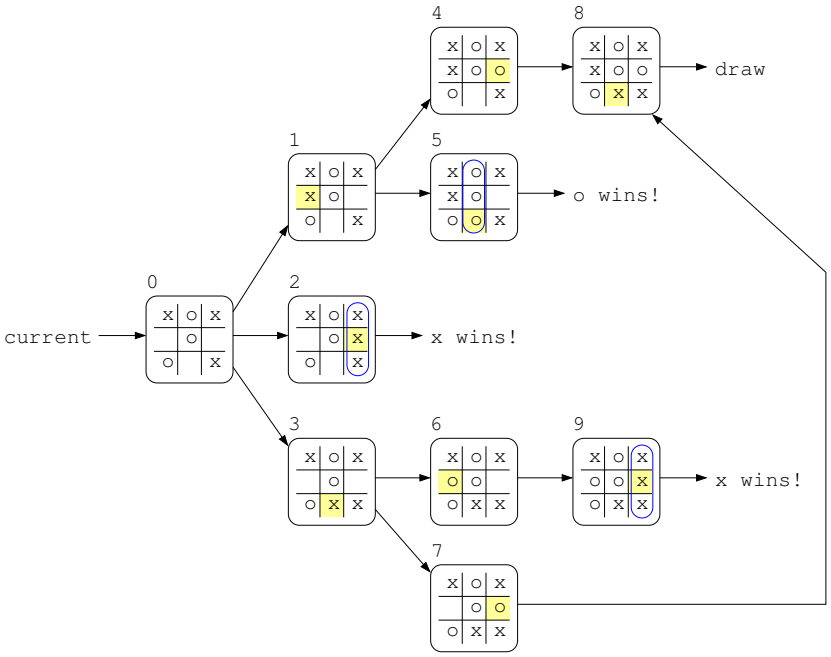
CSCI-1200 Data Structures — Fall 2011
Homework 8 — Tic-tac-toe Graphs

This assignment explores a generalization of linked lists and trees called graphs. Thus far we have seen nodes in linked lists storing one or two pointers to other nodes. In graphs, nodes may have any number of pointers to other nodes. Moreover, instead of having a head pointer (and perhaps a tail pointer) through which the rest of the data are accessed, we might access the contents of the graph starting at any node. We will build a graph data structure to represent how a game a Tic-tac-toe evolves as players take turns placing 'x's and 'o's on the board. If you are not familiar with the rules and gameplay, check out an online reference: e.g., http://en.wikipedia.org/wiki/Tic-tac-toe. *Please read the entire handout before you begin.*



Input and Output

Your program will read input from `cin` and write output to `cout`. You are encouraged to test your program using *file redirection*. Your program will not expect any command line arguments. The input file consists of six different types of commands (plus two additional commands for extra credit), described below.

`new_board` ***n*** : initialize an empty board for the *current game state*. The integer n indicates the size of the (square) board. For standard Tic-tac-toe, the size is 3x3.

`place` ***c i j*** : An 'x' or 'o' (specified by c) is added to the board at the position (i, j). Note that the two players alternate placing tokens on the board and 'x' always goes first. Thus, given the current game board, we can easily verify whose turn it is by simply counting the number of 'x's and 'o's placed so far.

`create_graph` ***d*** : Starting at the *current game state*, construct the graph of all possible future states, considering a maximum of d additional moves. The diagram above shows a graph of $d = 3$. For that particular starting configuration, this is also the complete graph of all future states (no more than 3 moves are necessary to complete the game). Higher values of d will produce the same graph.

`print_graph` : Output the necessary information about the nodes in the graph, to allow a diagram of the graph to be drawn as above. Please follow the format of the example output as closely as possible. All nodes in the graph are printed. Each node is identified by a *node id* (the number at the upper left corner of each game state in the diagram

above) to allow them to be easily connected together in a hand-drawn diagram. Note that depending on your implementation, the nodes may be created in a different order and be assigned different id numbers. Therefore, the output of the print graph command may be somewhat different. In addition to the node id and the board contents, the output for each node includes the game result ("x wins", "o wins", or "draw") if the game is over, and the nodes that point to this node (*links from*) and the nodes that follow this node (*links to*).

`merge_rotations_and_flips` : Many Tic-tac-toe boards are actually duplicates. The board can be rotated 90°, 180°, or 270°, and for each rotation, the board can be mirrored. By specifying this command the graph will only include one copy of these duplicate boards. In the example above, node #4 and node #7 are the same board. When this flag is specified your graph should contain only the unique game states, and should contain all connections between these states.

`evaluate` : For the *current game state*, output a summary of the choices for the current player. In the example above, if 'x' chooses to place a token at $(1, 0)$, he cannot win. There are 2 terminal states on this path: one ends with 'o' winning and one ends in a draw. If 'x' chooses to place a token at $(1, 2)$, there is a single terminal state, where 'x' wins. And if 'x' chooses to place a token at $(2, 1)$, there are two terminal states: one ends with 'x' winning and one ends in a draw. If the graph is not fully expanded, the unknown outcome paths are also included in the summary (see the example output).

`maximum_tokens` $\boldsymbol{m}$: This command is used for the extra credit variation limited to m tokens for each player, called *Achi*, *Tapatan*, or *Three Men's Morris*.

`move` $\boldsymbol{c}$ $\boldsymbol{i_1}$ $\boldsymbol{j_1}$ $\boldsymbol{i_2}$ $\boldsymbol{j_2}$: This command is used for the extra credit variation with limited tokens. Once a player has placed all tokens from their limited supply, they instead move one of their own tokens from (i_1, j_1) to (i_2, j_2).

There will not be any formatting errors in the input file, but make sure you do simple error checking to ensure that the operations are allowed (as this will help you debug).

The Graph Class

The main task in this assignment is to implement the `Graph` and `Node` classes. A `Graph` object (your program will have only one instance of the graph at a time) stores a pointer to the *current* game state `Node`, and an STL container of *pointers* to all `Node` objects. Each `Node` object will store its id, the board state, and two STL containers of *pointers* to the other `Node` objects it is linked to (one container stores *links from* other `Node`s, the other container stores *links to* other `Node`s).

Note that when a `place` (or `move`) command follows the `create_graph` command, the `Graph` object should be *updated* as follows. The *current game state* `Node` pointer should point to the `Node` representing the newly chosen game state. The `Graph` object should be expanded as necessary, so that all possible board states reachable in d moves are included in the graph (where d is the number of moves specified in the `create_graph` command). Finally, all `Node`s and *links between* `Node`s that cannot be reached using a forward traversal of the graph from the *current game state* `Node` are properly removed from the `Graph` and deleted as appropriate.

For the initial version of your code, do *not* use a `std::map` or `std::set` anywhere in your code. Instead use an STL `vector` and/or `list` (whichever is more appropriate) for the STL containers described above. This is to help you focus on the important aspects of the `Graph` and `Node` classes and interconnectivity between the `Node`s. Once the basic implementation is complete, and you have analyzed the performance of your implementation, you may replace these containers with an STL `set` or `map` and re-analyze the performance for extra credit.

We have provided a number of files from our solution to get you started (`main.cpp`, `board.h`, `board.cpp`). Study this code carefully as you work. You may modify the provided code as needed.

Pointers to Objects and Vectors of Pointers

There are several challenges of note in this assignment:

- The `Graph` and `Node` classes each store STL containers (`vector` or `list`) of pointers to `Node` objects. This may cause some confusion in the syntax when iterating through the container and accessing pointers. Here is sample code to print the *node ids* of all `Node`s linked to from a particular `Node`:

```
ostr << "has links to: ";
for (std::list<Node*>::const_iterator itr = links_to.begin();
     itr != links_to.end(); itr++) {
  ostr << " " << (*itr)->getID();
}
ostr << "\n";
```

  The iterator `itr` refers to a pointer to a `Node` object and `(*itr)` uses the iterator to access the `Node*` (the pointer). The `->` follows the pointer to the `Node` object and calls its `getID` member function. The parentheses are required here to ensure that the operators are applied in the correct order. You may need to use the `(*itr)->` idiom at several places throughout your code.

- To receive full credit you must deallocate (with `delete`) all memory that was dynamically allocated (with `new`), even if you are exiting the program and know that the operating system will clean it up for you. You are encouraged to use Valgrind or Dr. Memory on your local machine to find memory errors and memory leaks in your code. The submission server will run Valgrind on your program to help you check for problems.

- When an object (e.g., a `Node` object) has multiple pointers to it, a question arises about when it is appropriate to delete the object. In this case it should occur when the entire `Graph` is deleted (the `Graph` destructor is called), or when that `Node` is no longer accessible by following *forward* pointers from the current game state. It is not appropriate to delete the `Node` when only some of the incoming links (pointers) to it are being eliminated.

Performance Analysis

In your `README.txt` file analyze the running time order notation for each command, and the memory usage order notation for the graph in terms of n = the size of the board, d = the number of moves explored by the graph, and b = the average number of choices available to each player on each turn (the braching factor).

Extra Credit

For extra credit, change your implementation to use `set` or `map` to improve the running time performance. Analyze the original and improved implementations. Another option for extra credit is to implement a variation of Tic-tac-toe limiting each player to just m tokens (when $m = 3$ this is called *Achi*). After the tokens have been placed, the player instead moves one of their tokens to a new location on the board during their turn. Discuss your implementation and testing in your README.txt.

Submission

Do all of your work in a new folder named `hw8` inside of your Data Structures homeworks directory. Please use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.** When you are finished please zip up your folder exactly as instructed for the previous assignments and submit it through the course webpage.