Assignment 7

Version 1.02 (last update: Nov. 19, 16:00)
Changes highlighted in yellow
Due date: Thursday, Nov 26, 11:59 PM

Summary and Purpose

For this assignment, you will be working with both a hash-table and a graph data structure. You will be using these to create an unbeatable tic-tac-toe (noughts and crosses) player.

Provided Files

You can find the following files on CourseLink:

1) A file called **`ttt.h`** that contains some macro constants, a **`typedef`**, some external (global) variable definitions, a structure, and some function prototypes.
2) A file called **`ttt1.c`** that contains some global variables, and two helper functions.
3) A file called **`alphatoe.c`** that contains the code to play games of tic-tac-toe.

Do not include these files in your repository. Do not modify these files (since you are not including them in the repository and only the original files will be used during testing).

Deliverables

You will be submitting:

1) A file called **`ttt2.c`** that will be compiled to create an object file called **`ttt2.o`**.
2) A file called **`show_node.c`** that will be compiled to create an executable file called **`show_node`**.
3) A file called **`count_nodes.c`** that will be compiled to create a program called **`count_nodes`**.
4) A **`makefile`** that contains the instructions to compile your code.

You will submit all of your work via git to the School's gitlab server. (As per instructions in the labs.)

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate adjudicated using the University's Academic Integrity rules.

Data structures and global variables and constants

You will be using the following data structures in your code:

```
typedef char Board[HSIZE];
```

This typedef is used to represent a printable tic-tac-toe board. The initial board will be printed as:

```
0|1|2
-+-+-
3|4|5
-+-+-
6|7|8
```

This board is represented by the string:

```
Board START_BOARD = "0|1|2\n-+-+-\n3|4|5\n-+-+-\n6|7|8\n";
```

Once the game begins players select one of the places on the board (indicated by numbers) and place their markers (capital **X**s and capital **O**s) over top of the numbers. The length of the board string always stays the same (**HSIZE** including the null terminator character). E.g. if the player, **X**, decides to play a piece at position 4, then the new board string would look like:

```
"0|1|2\n-+-+-\n3|X|5\n-+-+-\n6|7|8\n";
```

In order to quickly figure out where in the string to draw the letter, I have provided an array of integers (pos2idx) which can be used to map a board position to an index in the board string. So, to play an X at position 4 you could write:

```
board_string[ pos2idx[4] ] = 'X';
```

The objective of the game of tic-tac-toe is to place three markers in one row, column or diagonal.

An array called **WINS** contains 8 rows corresponding to the possible winning configurations, with 3 position values in each row.

A structure called **BoardNode** represents a node in the graph of all possible tic-tac-toe games. Each node represents one board, some properties for that board, and the moves that can be applied to that board and the subsequent boards, based on those nodes.

```
struct BoardNode
{
  char init;            // 0=not initialized, or 1=initialized
  char turn;            // 'X', 'O', '-'=board is full (game over)
  int depth;            // depth of the node in the graph = moves played
  Board board;          // printable board file
  char winner;          // 'X', 'O', '-' (tie), '?' (game in progress)
  int move[9];          // board hashes for all move positions; -1=illegal
  int score;            // winner of the game if both players play smartly
                        // 1=X -1=O 0=tie
};
```

A global variable called **htable** is designed to describe all the possible boards in the game of tic-tac-toe (and a bunch of impossible boards as well).

```
struct BoardNode htable[HSIZE];
```

The functions and programs

void init_boards();
This is a function which you will create in the file **ttt2.c**. The function should go through the entire **htable** and set the **init** variable in each structure to **0**, to indicate that that element of the hash table is empty (hasn't been initialized).

int depth(Board board);
This is a function which you will create in the file **ttt2.c**. The function should return the number of markers (**X**s and **O**s) on the board.

char winner(Board board);
This is a function which you will create in the file **ttt2.c**. The function should return either:

- **'X'**, if **X** has won the game,
- **'O'**, if **O** has won the game,
- **'-'**, if the game is over and a draw (the board is filled and no-one has won), or
- **'?'**, if the game is incomplete and no-one has won.

You can use the **pos2idx** and **WINS** arrays to help to detect wins, and the **depth** function to determine whether the game is over.

char turn(Board board);
This is a function which you will create in the file **ttt2.c**. The function should return either:

- **'X'**, if it is **X**'s turn,
- **'O'**, if it is **O**'s turn,
- **'-'**, if the game is over a (the board is filled).

You can use the **depth** function and modular division to determine whose turn it is.

`void init_board( Board board );`
This is a function which you will create in the file **`ttt2.c`**. The function should compute the hash index of the provided board using the **`board_hash`** function in **`ttt1.c`**. It should set the variables of the structure at that index as follows:

- **`init`** should be set to **`1`**,
- **`turn`** should be set to the return value of the **`turn`** function for the given **`board`**,
- **`depth`** should be set to the return value of the **`depth`** function for the given **`board`**,
- **`board`** should be set to the argument **`board`** (by calling the **`strcpy`** function), and
- **`winner`** should be set to the return value of the **`winner`** function for the given **`board`**.

`void join_graph( Board board );`
This is a function which you will create in the file **`ttt2.c`**. This function should create and join all the nodes of all possible tic-tac-toe games. This can be done recursively by looping over all the possible move positions (0-8). For each position, if there is already a piece in that position (check the **`pos2idx`** index in the string for the board), then store a value of **`-1`** in the corresponding position in the **`move`** array in the appropriate structure of the **`htable`** (computed by calling the **`board_hash`** function). If the index in the string contains a number instead (the spot is empty), then you must make a copy of the **`board`**, place the appropriate marker (**`turn(…)`**) in the appropriate spot (**`pos2idx`**) in the string. Next you compute the **`hash`** value of the new board, and store it in the **`move`** array. Then, you will need to check if the hash table already contains an entry (**`init!=0`**) for the new board string. If it does, you can loop to the next move position. If there is no entry, you will need to create one by calling **`init_board`** and recursively calling **`join_graph`**.

`show_node`
This is a program that accepts 0 or more integer command line arguments. It will display the value of the board node corresponding to the hash values supplied on the command line. It will call **`init_board()`**, **`init_board(START_BOARD)`**, **`join_graph(START_BOARD)`**, and, if you have implemented it, **`compute_score`**(). Then, for each command line argument it will call **`print_node`** on the location in the **`htable`** corresponding to the argument (e.g. if you pass a single command line argument of **`0`**, it should print the opening board).

`count_nodes`
This program will call **`init_board()`**, **`init_board(START_BOARD)`**, and **`join_graph(START_BOARD)`**, and then count the number of entries in the **`htable`** that have **`init==1`**. This is nodes that can actually occur in real tic-tac-toe games.

The Last 20%

`void compute_score();`
This is a function which you will create in the file **`ttt2.c`**. This function should assign a score to each entry in the **`htable`** that has an **`init`** value of 1. The score to be assigned is defined by the following rules:

1) if the board results in a win for **`X`**, then the score is **`1`**, otherwise
2) if the board results in a win for **`O`**, then the score is **`-1`**, otherwise
3) if the board is a complete game that ends in a ***tie***, then the score is **`0`**, otherwise

4) if it is **x**'s turn, then the score is equal to the ***maximum*** of all the valid child nodes in the **move** array, otherwise
5) if it is **o**'s turn, then the score is equal to the ***minimum*** of all the valid child nodes in the **move** array.

Hints: You must evaluate the nodes in order of decreasing depth. You can do this in a loop that counts down from 9 to 0, or using a recursive function that assigns values "coming out of" the recursion.

```
int best_move( int board );
```

This is a function which you will create in the file **ttt2.c**. This function should return the best possible move position. Note that the best possible move position is the move leading to the child node with the ***highest*** score for **x**, and the ***lowest*** score for **o**. You can test this code by using the **alphatoe** program.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify. Place the functions only in ttt2.c (not ttt1.c).

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will contain some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the standard SoCS Virtualbox VM (http://socs.uoguelph.ca/SoCSVM.zip) which will be run using the Oracle Virtualbox software (https://www.virtualbox.org/wiki/Downloads). If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. We will NOT test your code on YOUR machine/environment.

Full instructions for using the SoCS Virtualbox VM can be found at: https://wiki.socs.uoguelph.ca/students/socsvm.

Makefile

You will create a makefile that supports the following targets:

`all:`

this target should generate the files **`ttt1.o`**, **`ttt2.o`**, **`show_node`**, **`count_nodes`**, **`alphatoe`**.

`clean:`

this target should delete all **`*.o`** and executable files.

Each, **`*.c`** file should correspond to a rule in the **`makefile`** that generates a corresponding **`*.o`** file.

Additionally, there should be an individual rule for each executable file.

All compilations and linking must be done with the **`-Wall -pedantic -std=c99`** flags and compile and link **without any warnings or errors**.

Git

You must submit your **`.c`** files, and **`makefile`** using git to the School's git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A2 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student's code, now allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

init	4
turn	4
depth	8
board	8
winner	8
moves	16
style	2
makefile	2
score	8
alphatoe	5
Total	65

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.