COMP 2012 Object-Oriented Programming and Data Structures

Assignment 3 Weighted Tic-Tac-Toe with Minimax

A game of tic-tac-toe. Source: Wikipedia.

Introduction

The minimax algorithm is an algorithm used in a two-player turn-based game to determine the best move each player can make at each turn. The algorithm essentially explores every moves the player can make, then every responses the opponent can make for said move, and so on. Ideally, if the algorithm searches far enough, every possible games can be considered and thus an optimal play can be found to force a win, if any. Otherwise, intermediate/inconclusive game states can also be given a score that estimates how "good" the current player is playing, that can help to suggest a promising move.

In this assignment, you will design a minimax algorithm to play a game of "Weighted" Tic-Tac-Toe. The game Tic-Tac-Toe is already heavily explored by scientists and computers, and is known to be a drawn game if players play optimally, which is not a hard issue for most people. Therefore, we will modify the game a bit by adding "weights" to each cell, and players gain scores proportional to said weights when playing on each cells. In this way, if all cells have been filled, the player with the higher total score will win, so players will also need to account for the scores of each cell when playing.

The minimax algorithm involves exploring possible moves from a given game state, as well as possible responses of the opponent. Therefore, you will need to design a tree to store the board states and its "children" - the possible future board states of the current one. Unlike the binary tree learned during lecture, a tree node can now have more than 2 children, but the structure is still mostly the same.

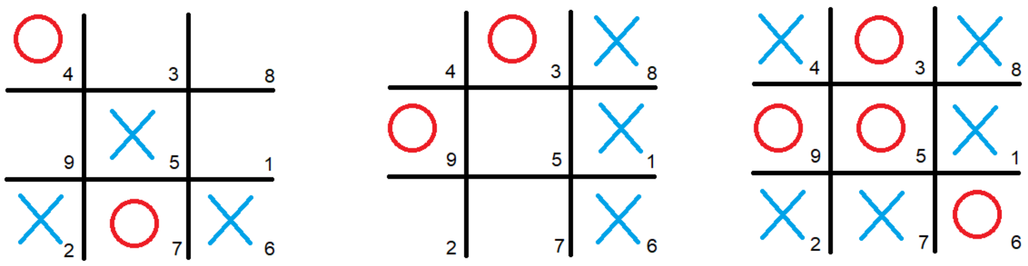
You will also find that pure minimax can be very time-consuming the deeper you search - after all, minimax is simply brute-forcing through all possible moves and board states. We will also implement some optimizations to the algorithm: a transposition table implemented as a hash table to store calculated board states, and alpha-beta pruning to reduce the number of states we need to calculate.

Important warning

Using a high search depth and/or large board size can not only take a long time, but may also use up a large amount of memory. **Running out of memory can crash your computer and force a hard reset (e.g holding the power button)**. It is recommended that you use board size 3 and a low search depth when starting the assignment or doing early testing. When you want to try a higher depth or board size, monitor the memory usage such as with Task Manager. If the program is not responding for ~30 seconds and the memory usage is reaching your machine's limit, **terminate the program immediately** (e.g. Ctrl+C in the terminal). Board size 3 should generally be fine for search depth up to 9 (which is sufficient to weakly-solve the game), but board size 4 will make the tree grow extrememly quickly and you are recommended to only try up to depth 8.

Gameplay overview

The game is played in a similar way as Tic-Tac-Toe. Player X starts by placing X on any of the cell on the 3x3 board. The player O then does the same, but on a cell that has not been filled. The game proceeds until one player gets a row/column/diagonal line with all of that player's symbol (called a *match*), at which point that player wins the game. Otherwise, when the board is completely filled, the player with higher total score associated with their chosen cells wins the game (if the score is equal, the game is a draw). A player's score is calculated as the sum of that player's cell scores, **multiplied by the number of moves the opponent makes**. For example, in a 3x3 board, player X's score is multiplied by 4, while O's score is multiplied by 5. This is to compensate player X being able to fill more cells than O. Another way is to subtract the total score of O from X - if the final score is negative, O wins, and X wins if it is positive (non-zero).



Here is an example of the score system. On the left example, the score is `(2+5+6)x4 - (4+7)x5 = -3`, although the game has not ended yet. On the middle example, X has a match on the rightmost column, so the score is `10000` and X wins the game. On the right example, all cells are filled so the score will determine the winner. The score is `(1+2+4+7+8)x4 - (3+5+6+9)x5 =`

-27, so O wins the game.

The game can also work with an arbitrary board size (at least 3). However the minimax tree can become extremely large with a bigger board, so you are recommended to test your code with board size of 3 or 4 when the minimax algorithm is implemented.

Classes overview

The following section describes the given classes and structs, as well as additional constants and helper functions.

const.h

This file contains several constants and helper functions:

- `BOARD_SIZE`: The size of the board, which is 3. If you would like to test your algorithm with a different board size, you may try 4.
- `TABLE_SIZE`: The size of the hash table. Recommended size for 3x3 board is 101, and 4x4 board is 200009.
- `WIN_SCORE`: The score of the board if player X has gotten a row/column/diagonal match (`-WIN_SCORE` if player O instead). **You can assume the regular score will never exceed this number**.
- `ILLEGAL`: A constant to specify the score of an "illegal" board state.
- `SCORE_PRESET`: The default score grid. The executable can be supplied an argument to enable manual score input.
- `DEPTH_PRESET`: The default search depth. The executable can be supplied an argument to set a custom search depth.
- `enum Cell`: An enum for the possible values on a board cell.
- `int getCellWeight(const Cell c)`: Helper function to get the score scaling for each player.

The functions `cell2chr` and `printBoard` are helper functions for printing the board, but you don't need to use them as the print functions are already implemented.

BoardCoordinate

```
// board.h
struct BoardCoordinate {
  int row, col;

  BoardCoordinate(int row, int col): row(row), col(col) {}
  bool isValid() const { return row >= 0 && row < BOARD_SIZE && col >= 0 && col < BOARD_SIZE;
};
```

This is a simple struct holding the coordinates of a move on a board (which row and which column). There is also a member function to check if the row/column numbers are valid.

BoardOptimalMove

```
// board.h
struct BoardOptimalMove {
  int score;
  BoardCoordinate coords;

  // Default constructor returns an "illegal" move
  BoardOptimalMove(): score(ILLEGAL), coords(BoardCoordinate(-1, -1)) {}
  BoardOptimalMove(const int score, const BoardCoordinate& coords): score(score), coords(coord
};
```

This is a another struct containing the information of a "move" in the minimax algorithm: the coordinates and its associated score. The minimax algorithm needs to generate every possible moves at a node, and compare the estimated scores of each move. The default constructor creates an "illegal" move, which can be used for default cases or when the move is illegal/cannot be made.

Board

```
// board.h
class Board {
  friend std::ostream& operator<<(std::ostream& os, const Board& board) {
    printBoard(os, board.cells, cell2chr);
    return os;
  }

  private:
    Cell cells[BOARD_SIZE][BOARD_SIZE];
    int score[BOARD_SIZE][BOARD_SIZE];
    Cell curPlayer;
    unsigned long long id;

  public:
    // Initialize cells as all empty, score with the given parameter, curPlayer as X and id as
    Board(const int score[][BOARD_SIZE]);

    // Return true if all cells are not EMPTY.
    bool isFull() const;

    // Return true if the game has finished (a match is found or board is full).
    bool isFinished() const;

    // Calculate the board's current score.
    int getBoardScore() const;

    // Play the next move at coords. If successful, update the data members and return true. O
    bool play(const BoardCoordinate& coords);

    // Implemented
    Cell getCurPlayer() const { return curPlayer; }
    unsigned long long getID() const { return id; }
};
```

This class represents a board state. It has the following data members:

- `cells`: A 2D array of Cell representing the board configuration.
- `score`: A 2D array of int containing the score grid. This is set when the Board instance is constructed.
- `curPlayer`: The next player to make the move. Note that because the first player is always X and the second player is O, the player to play can always be determined from any given board state. But for convenience, we keep track of the player and update it when we make changes to the board.
- `id`: A number representing the board state. Each possible board state is given a unique number calculated from its configuration - we also just keep track of it and update when the board configuration is changed. Note that the type of `id` is `unsigned long long`, meaning that we can support `id` as high as 2^64 - 1.

The following constructor and member functions need to be implemented:

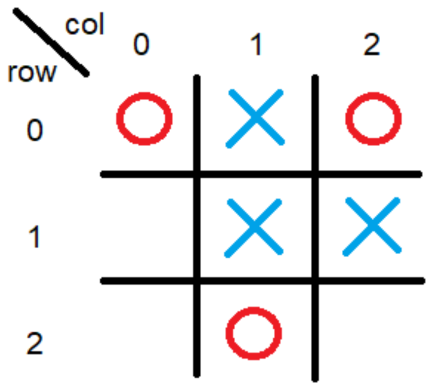
- `Board(const int score[][BOARD_SIZE])`: Creates an empty board with the specified score grid. Note that all data members are non-dynamic, so the default memberwise copy constructor is sufficient.
- `bool isFull() const`: Returns true if all cells of the board are non-empty.
- `bool isFinished() const`: Returns true if the game is finished - in other words, the board configuration cannot be continued, either because it is full or one player has made a match.
- `int getBoardScore() const`: Returns the current board's score by adding all of X's cell scores and subtracting all of O's cell scores, scaled by `getCellWeight()`. If a player has a match, returns `WIN_SCORE` or `-WIN_SCORE` instead.
- `bool play(const BoardCoordinate& coords)`: Try to make the next move at the given coordinates. If the move is not valid (out of bounds or cell is non-empty), returns false. Otherwise, update the board, `id`, `curPlayer` accordingly and return true.

Explanation of the ID system: The game rules imply that the state of the game can be uniquely determined only on the current board configuration - the score does not depend on what order the prior moves were played in. We can therefore optimize the search algorithm by keeping track of which board states have already been calculated, since there can be multiple paths leading to the same board configuration. We can "encode" the board configuration into a number for use in the hash table. This can be achieved by considering each cell as a "ternary" digit (base 3), as each cell can be either empty, has an X or has an O, and treating the board as a `BOARD_SIZE * BOARD_SIZE` ternary number (e.g. for a 3x3 board, we encode it as a 9-ternary-digit number, which is at most 3^9 - 1).

For consistency, you should calculate the ID according to the following algorithm:

```
id = 0;
for each cell {
  cellValue = 0 if cell is empty, 1 if cell is X, 2 if cell is O;
  id += cellValue * (3 ^ (row * 3 + column));
}
```

The following is an example of the ID calculation. A program **id2board.exe** is provided if you would like to verify your board ID.



$$ID = 2 * 3^0 + 1 * 3^1 + 2 * 3^2 + 1 * 3^4 + 1 * 3^5 + 2 * 3^7 = 4721$$

Note that you don't need to perform the loop to calculate the ID everytime. As we keep track of the board ID, when a player makes a move, you only need to add the id value corresponding to that move's coordinates to the current value.

BoardTree

```
// boardtree.h
struct BoardNode;

class BoardTree {
  private:
    BoardNode* root {nullptr};

  public:
    // Default empty constructor
    BoardTree() = default;
    // Initialize root using the provided board
    BoardTree(const Board& board);
    // Destructor
    ~BoardTree();

    // We don't need to copy tree in this case
    BoardTree(const BoardTree&) = delete;
    BoardTree& operator=(BoardTree&) = delete;

    bool isEmpty() const { return root == nullptr; }

    // Return a pointer to the subtree at the given coordinates. Build the tree if it is empty
    BoardTree* getSubTree(const BoardCoordinate& coords);

    // Calculate the best move by searching the tree up to depth moves ahead
    BoardOptimalMove getOptimalMove(const unsigned int depth);

    // Same as above but utilizes alpha-beta pruning
    BoardOptimalMove getOptimalMoveAlphaBeta(const unsigned int depth, int alpha, int beta);
};

struct BoardNode {
  const Board board; // Current board state
  BoardTree subTree[BOARD_SIZE][BOARD_SIZE]; // One sub-tree for each possible next move

  BoardNode(const Board& board): board(board) {}
  BoardNode(const BoardNode& node) = delete;
  BoardNode& operator=(const BoardNode& node) = delete;
};
```

This class represents the search tree of the minimax algorithm. Its structure is similar to the BT and BST examples given in the course materials: the class contains a single data member which is a pointer to a struct `BoardNode`. This struct contains a **const** Board instance and a 2D array of BoardTree representing the subtrees of the current node. A leaf node is an empty node (`root == nullptr`), which can either mean the node has not been explored, or the board/move represented by that node is illegal.

Example: Consider a BoardTree representing a 3x3 board with an X played in the middle. This means `root->board` is the board configuration, and `root->subtree` is a 3x3 array of BoardTree representing the 9 possible moves of O. If the next step has not been explored, all 9 array elements are default initialized and will be empty leaf nodes. Once we explore this node fully, all array elements will be non-empty, **except** `root->subtree[1][1]`, since O cannot make a move in the middle cell occupied by X.

The following constructor, destructor and member functions need to be implemented:

- `BoardTree(const Board& board)`: Initializes a non-empty BoardTree using the provided Board.
- `~BoardTree()`: Destroys the object and deallocate dynamic memories accordingly.
- `BoardTree* getSubTree(const BoardCoordinate& coords)`: Returns a pointer to the subtree corresponding to the given coordinates. If that subtree has not been explored, it should be explored first. This function is mostly for the `main()` function to keep track of the current board.
- `BoardOptimalMove getOptimalMove(const unsigned int depth)`: Finds the optimal move according to the minimax algorithm, using the provided depth.
- `BoardOptimalMove getOptimalMoveAlphaBeta(const unsigned int depth, int alpha, int beta)`: Same as above, but with alpha-beta pruning. The two functions are split so that you can focus on making sure the non-optimized version works first.

Notice that unlike the `BT`/`BST` examples, we don't want to create copies of the tree or node since each node should correspond to a unique board configuration. Therefore the copy constructors and assignment operators are all `deleted`. In addition, there is no `insert` or `remove` member functions. The tree can only be expanded by calling `getOptimalMove()` (or `getOptimalMoveAlphaBeta()`) to explore unexplored nodes, or via `getSubTree()` if the subtree is unexplored.

Also note that BoardNode is not a private struct of BoardTree unlike BT/BST since BoardTree is not a template class. Even so, you should not have to create any BoardNode instances directly in any other classes/functions other than BoardTree. Another way to resolve this is to have the definition of BoardNode in **boardtree.cpp**.

BoardHashTable

```
// hashtable.h
class BoardHashTable {
  private:
    struct BoardHashNode {
      unsigned long long id;
      int depth;
      BoardOptimalMove optimalMove;
      BoardHashNode* next;

      BoardHashNode(const unsigned long long id, const int depth, const BoardOptimalMove& opti
        id(id), depth(depth), optimalMove(optimalMove), next(nullptr) {}
    };

    BoardHashNode* table[TABLE_SIZE] {nullptr};

    BoardHashTable() = default;
    ~BoardHashTable() { clearTable(); }

  public:
    BoardHashTable(const BoardHashTable&) = delete;
    BoardHashTable& operator=(const BoardHashTable&) = delete;

    // Return the only instance of BoardHashTable
    static BoardHashTable& getInstance()
    {
      static BoardHashTable instance;
      return instance;
    }

    // Return the stored BoardOptimalMove for the given id and depth. If it is not stored, ret
    BoardOptimalMove getHashedMove(const unsigned long long id, const int depth);

    // Update the table with the optimal move for the given id and depth.
    // If id does not exist, create a new linked list node at (id % TABLE_SIZE).
    // Else, if stored depth is lower, update the optimal move with the parameter. Otherwise,
    void updateTable(const unsigned long long id, const int depth, const BoardOptimalMove& opt

    // Clear all dynamic memory and reset table array to all nullptr.
    void clearTableble();
};
```

This is a hash table for storing the calculated board configurations. It is designed as an array of linked lists, using the separate chaining method. The size of the table is defined in **const.h**. For your reference, there are 5478 possible board states for a 3x3 board, and 9722011 possible states for a 4x4 board. The given table sizes are sufficient to hold the number of calculated states so that the length of linked lists is not large and slows down calculation time.

`BoardHashNode` is a node in a linked list. It contains the following data members:

- `id`: The board ID.
- `depth`: The depth this board has been calculated to. If we would like to calculate this board to a higher depth, the hashed value is no longer sufficient.
- `optimalMove`: The `BoardOptimalMove` calculated for this board. It contains the associated optimal score and move coordinates.
- `next`: Pointer to the next BoardHashNode in the linked list.

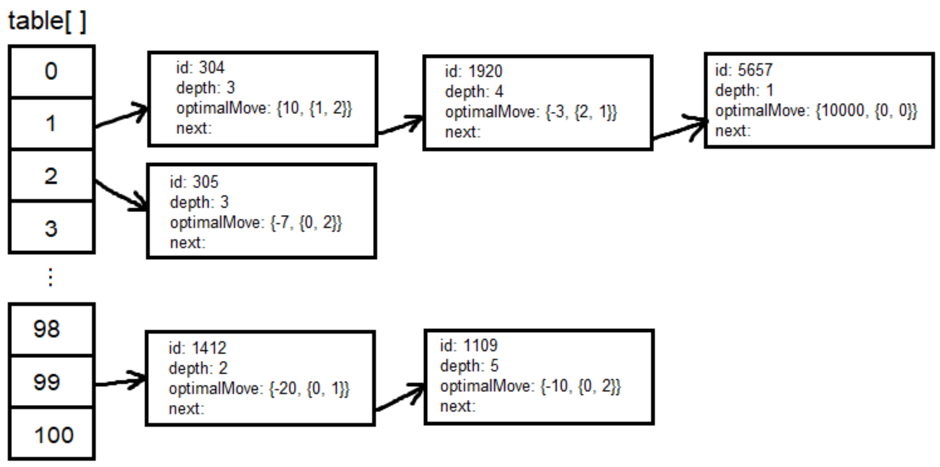
Board ID `id` is stored in the linked list at index `id % TABLE_SIZE`. The linked list can be in any order, and during searching you can simply loop through the linked list at the corresponding index until the board ID is found.

The private default constructor returns an empty hash table with all array elements `nullptr`. The copy constructor and assignment operator are deleted, since we also don't need to create copies of the table. However, you won't need to create any BoardHashTable instances, as there is a **static** function `getInstance()`. This class is designed using the Singleton pattern, meaning that there is guaranteed to be one and only one instance of BoardHashTable throughout the program. This way, all BoardTable nodes can access the shared hash table directly by retrieving the static table instance.

The following member functions need to be implemented:

- `BoardOptimalMove getHashedMove(...)`: Get the hashed BoardOptimalMove for the given board ID and depth. If the board ID does not exist or has not been calculated to the given depth, this returns an "illegal" move that can be ignored.
- `void updateTable(...)`: Updates the table with the provided board ID, depth and optimal move. This can either insert a new node to the table, or updates an existing node with a newer optimal move associated with higher search depth.
- `void clearTable()`: Delete all dynamic memories and reset the table to all `nullptr`. This is only needed in `main()` in case you would like to reset the board for testing. Note that the destructor simply calls this function.

The following is an example of the hash table. Some nodes are currently stored in the table.


table[ ]
0
1
2
3
⋮
98
99
100
id: 304
depth: 3
optimalMove: {10, {1, 2}}
next:
id: 1920
depth: 4
optimalMove: {-3, {2, 1}}
next:
id: 5657
depth: 1
optimalMove: {10000, {0, 0}}
next:
id: 305
depth: 3
optimalMove: {-7, {0, 2}}
next:
id: 1412
depth: 2
optimalMove: {-20, {0, 1}}
next:
id: 1109
depth: 5
optimalMove: {-10, {0, 2}}
next:

Here are the expected result of some operations on the table:

- `getHashedMove(306, 3)`: The table index is `306 % 101 = 3`, which points to `nullptr`. Returns an illegal move.
- `getHashedMove(402, 3)`: The table index is `402 % 101 = 99`. The linked list at index 99 is non-empty, but contains no node with id 402. Returns an illegal move.
- `getHashedMove(1920, 4)`: The table index is `1920 % 101 = 1`. The linked list at index 1 is non-empty, and contains a node with id 1920 and depth 4. Returns `BoardOptimalMove{-3, {2, 1}}`.
- `getHashedMove(1920, 3)`: The linked list at index 1 is non-empty, and contains a node with id 1920 and depth 4, so the stored move is at least as good as what is being searched for. Returns `BoardOptimalMove{-3, {2, 1}}`.
- `getHashedMove(1920, 5)`: The linked list at index 1 is non-empty, and contains a node with id 1920 and depth 4, which is not sufficiently deep. Returns an illegal move.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to ensure you understand what is considered plagiarism and the penalties. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, including expulsion.

End of Introduction

Read the FAQ page for some common clarifications. You should check that a day before the deadline to ensure you don't miss any clarification, even if you have already submitted your work.

void cluttering th
before posting y

Task description

You will need to complete the following tasks in 3 files: **board.cpp**, **boardtree.cpp** and **hashtable.cpp**. Complete the relevant tasks in the relevant files. In addition, **you are not allowed to include additional libraries unless specified**. You may only `#include` any given header files (`board.h`, `boardtree.h`, `hashtable.h` and `const.h`). The libraries `<iostream>` and `<cmath>` have been included in **const.h**.

Task 1: Implement the game

For the first task, you need to implement the member functions of `Board` in **board.cpp**:

```
Board::Board(const int score[][BOARD_SIZE])
```

The constructor initializes the `cells` board as all empty, the `score` grid using the provided parameter, the current player `curPlayer` as X since X starts first, and the `id` as 0.

```
bool Board::isFull() const
```

Returns true if all cells of the board are non-empty, and false otherwise.

```
bool Board::isFinished() const
```

Returns true if all cells of the board are non-empty or a player has gotten a match.

```
int Board::getBoardScore() const
```

Returns the board's score. You need to first check if any player has gotten a match: a row, column or diagonal with all cells of that player. If yes, return `WIN_SCORE` if player X has a match, or `-WIN_SCORE` if player O has a match. Otherwise, calculate the score based on the current cells on the board.

Note: You can assume there will never be a case where both players have a match, since it would be an unreachable board state.

```
bool play(const BoardCoordinate& coords);
```

If the move is not valid (out of bounds or the cell is filled), return false. Otherwise, update the corresponding `cells` array element, change `curPlayer` to the other player and update `id` according to the algorithm described above.

Upon completing this task, you can uncomment the first function in **main.cpp** and test if you can play a full game of Weighted Tic-Tac-Toe in your program. For each move, the player inputs the row and column number (starting from 0), and the game is played until a player gets a match or no more moves can be made, at which point the score is used to determine the winner. The intermediate score and board ID for each move is also displayed.

Task 2: Implement the Minimax algorithm

For the second task, you need to implement the member functions of `BoardTree` in **boardtree.cpp**:

```
BoardTree::BoardTree(const Board& board)
```

The constructor initializes the `root` BoardNode with the provided board. This constructor is used to create the actual "root" of the search tree, with the children nodes constructed via default constructor and will be properly initialized as the tree is explored.

```
BoardTree::~BoardTree()
```

Free any dynamic memory accordingly. *Hint:* BoardTree is similar to BT/BST introduced in the course materials.

```
BoardTree* BoardTree::getSubTree(const BoardCoordinate &coords)
```

Returns a pointer to the sub-tree indicated by the given coordinates. This represents the child node corresponding to the move with the given coordinates. If the sub-tree is empty, **you should try to build the sub-tree first** (hint: if `play()` returns true, the new board is valid and can be build) by initializing its `root` data member accordingly.

```
BoardOptimalMove BoardTree::getOptimalMove(const unsigned int depth)
```

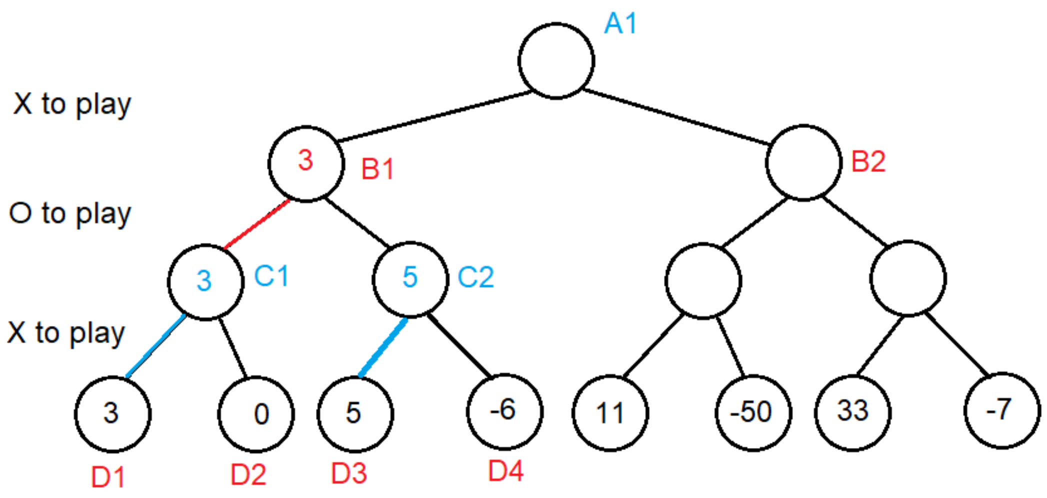
The core minimax algorithm function. Returns a BoardOptimalMove struct representing the best possible score the current player can reach, and the corresponding move that can reach that score, as calculated up to the given depth. You may refer to the following pseudocode:

```
if (board is empty) {
  return BoardOptimalMove(); // Returns a dummy illegal move
}

if (depth == 0 || board is finished) {
  // If depth is 0 or if the game has already finished, we cannot search further
  // return the score of this board with any move coordinate since we will not use it
  return BoardOptimalMove(board score, any coordinates);
}

// Else, we find the estimated score and optimal move of this node by calculating the score of
// Player X is trying to maximize the score, so the estimated score is the maximum of children
// Vice versa, player O is trying to minimize the score
int estimatedScore = -∞ if current player is X, else +∞;
BoardOptimalMove bestMove;
for each subtree {
  build the subtree if it is empty;

  BoardOptimalMove childMove = subtree.getOptimalMove(depth - 1);

  if (childMove.score == ILLEGAL) {
    // If the move is illegal, the subtree corresponds to an invalid move/board, simply skip t
    continue;
  }

  if (childMove.score is better than estimatedScore) {
    estimatedScore = childMove.score;
    bestMove = BoardOptimalMove(estimatedScore, move corresponding to subtree);
  }
}

return bestMove;
```

Note: If there are multiple moves with the "best" estimated score, you should return the first move found (as following the above algorithm, `bestMove` is only updated if a "better" score is found). To ensure consistency, the search order is **loop through each row, and in each row loop through each column**. For example, in a 3x3 board, this is the order of subtrees being searched:

1	2	3
4	5	6
7	8	9

Here is an illustration of the minimax algorithm to get yourself more familiar:



Assume we are calculating the score of board state A1, which is X's turn, by performing minimax search with depth 3. For simplicity, if in each move each player can only go left or right, we will need to calculate the score of 8 board states, which are indicated in black. As the minimax algorithm performs depth-first search, we will calculate the score of state D1 and D2 directly via `getBoardScore()`. Then, as C1 corresponds to the board state where X is making a move, the score of C1 is the maximum of its children nodes, thus state C1's optimal move is X moving left with a score of 3. Similarly, C2's optimal move is X moving left with a score of 5. Then, B1 corresponds to O's move, who is trying to minimize the score. Here, O would need to go left, since O can assume X will not move right at C2 and making O win. Therefore, B1's optimal move is O moving left with a score of 3.

Tip: Try to fill in the remaining nodes yourself, and convince yourself that A1's score is 11, and the optimal move for X is to move right.

The last member function, where you will implement an optimization of the algorithm known as alpha-beta pruning, will be covered in Task 4. After you have finished Task 2, you can uncomment the relevant function in **main.cpp**, and the program should be able to automatically run a full game of Weighted Tic-Tac-Toe, playing the optimal move for each player (calculated up to defined `DEPTH_PRESET`). The program will also output the time it takes for the game to run, which may be slow at the moment as we have not added any optimization.

IMPORTANT: You are recommended to make sure your code works properly with board size 3 and a small search depth first (e.g. 1, then 3 or 4). Trying a larger search depth may not only take a long time, but can also make it hard for you to pin-point the error in your code.

Task 3: Implement the transposition hash table

For the third task, you need to implement the member functions of `BoardHashTable` in **hashtable.cpp**:

```
BoardOptimalMove BoardHashTable::getHashedMove(const unsigned long long id, const int depth)
```

Returns the BoardOptimalMove stored in the hash table for the given board ID and search depth. If the board ID does not exist in the table, or the ID is stored but the search depth is lower than the parameter, then return an illegal move indicating that the table does not contain the queried board ID or the stored calculation is not sufficient (if the search depth parameter is equal to or lower than the stored search depth, return the stored move).

```
void BoardHashTable::updateTable(const unsigned long long id, const int depth, const BoardOpti
```

Updates the table with the optimal move for the given board ID and search depth. If the board ID does not exist in the table, create a new node at the corresponding linked list (the node can be inserted anywhere in the list). Otherwise, if the stored depth of the board ID is lower than the depth parameter, update the node's `depth` and `optimalMove`.

```
void BoardHashTable::clearTable()
```

Free all dynamic memories in the hash table and reset the linked lists to `nullptr`.

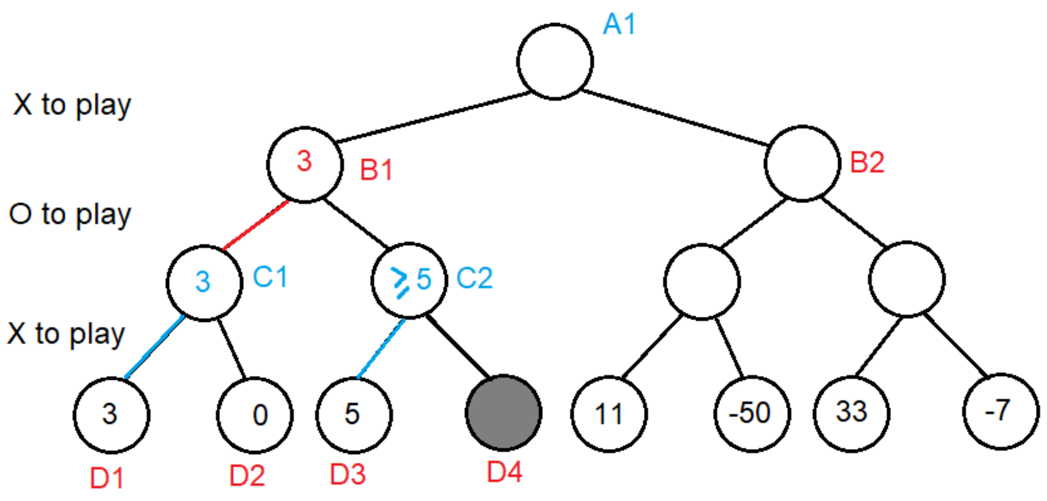
Once finished, you can start optimizing the minimax algorithm using the transposition table. By storing the optimal move calculated at each board positions, you can save a lot of computation time not having to calculate the same board state multiple time. The pseudocode for the minimax algorithm above can be modified as:

```
if (board is empty) {
  return BoardOptimalMove(); // Returns a dummy illegal move
}

if (depth == 0 || board is finished) {
  // ADDED: save the move to the table before returning
  BoardOptimalMove move = BoardOptimalMove(board score, any coordinates);
  updateTable(board id, depth, move);
  return move;
}

// ADDED: check if the board is already calculated to this depth
if (board and depth is stored in table) {
  return the optimal move stored in table;
}

// Else, we find the estimated score and optimal move of this node by calculating the score of
...
for each subtree {
  ...
}

// ADDED: save the move to the table before returning
updateTable(board id, depth, bestMove);
return bestMove;
```

In **main.cpp**, you can uncomment the code for Task 3 to check your implementation of the hash table functions. If you have also modified the minimax algorithm, try to run Task 2 test function again - you should see the same result, but lower calculation time. If you get a different result compared to without the hash table, double check your implementation as it is not supposed to change the algorithm result.

Task 4: Implement alpha-beta pruning

The final task is to implement alpha-beta pruning to further reduce the calculation time. Here is an illustration of the idea behind alpha-beta pruning using the previous example:



With alpha-beta pruning, we also keep track of the "best score found by the opponent". We still calculate the score of D1, D2, C1 and D3 as normal. But now, we can know that the score of C2 must be at least 5 - if D4's score is higher, C2's score is equal to D4, otherwise C2's score is 5. Thus, B1's children scores are 3 and "at least 5", which must mean its estimated score is 3 and O should move left. Therefore, we have skipped calculating D4.

Tip: Try to perform the same calculations on the right branch and convince yourself state D8 (score -7) will also be skipped. If we are using a higher search depth, we are also skipping the calculation of the grey nodes' children, and thus lead to a great decrease in computation time.

```
BoardOptimalMove BoardTree::getOptimalMoveAlphaBeta(const unsigned int depth, int alpha, int b
```

Implement the alternative version of `getOptimalMove()` by also keeping track of 2 additional variables: `alpha` and `beta`. To start, you can simply copy-paste the code implemented in `getOptimalMove()` (make sure it works first!), then make the changes described in the pseudocode below.

Essentially, `alpha` is the "best score found by X" and `beta` is the "best score found by O". In the previous example, `beta` when starting calculation for node C2 is 3, which means the score of B1 is at most 3. Depending on the current player, if a children node is found to have a "worse" score for the opponent (e.g. D3 has score 5, which is worse for O, who is trying to minimize the score, compared to score 3), we can exit the loop early and return to the parent node.

Here is the modified minimax (with transposition table) pseudocode with alpha-beta pruning:

```
...

// Else, we find the estimated score and optimal move of this node by calculating the score of
int estimatedScore = -∞ if current player is X, else +∞;
BoardOptimalMove bestMove;
for each subtree {
  build the subtree if it is empty;

  // MODIFIED: Call the alpha-beta pruning variant
  BoardOptimalMove childMove = subtree.getOptimalMoveAlphaBeta(depth - 1, alpha, beta);

  ...

  // ADDED: Update alpha and beta according to newest child score found
  if (current player is X) {
    if (optimalScore > beta) {
      // Found a move that would be worse for O, so O will not consider this subtree
```

End of Description

```
    // Update the best move found by X
```

Resources & Sample Program

Skeleton code: skeleton.zip

In the given skeleton code, you will see there are 2 files: **main.cpp** and **main_program.cpp**. The first file is used for your early testing, as described at the end of each tasks above. If you are satisfied with the testing, you can edit the Makefile to compile using the second file instead, which will give you a program similar to the below demo program.

- Demo program (Windows): pa3_3.exe / pa3_4.exe

pa3_3.exe is compiled with board size 3, and **pa3_4.exe** is compiled with board size 4. The remaining default values can be modified using **command-line arguments**. The following arguments are supported:

- **-h** or **--help**: Display the help message.
- **-d [num]** or **--depth [num]**: Set a custom search depth. If not specified, `DEPTH_PRESET` is used.
- **-s** or **--score**: Enable manual score grid input. If not specified, `SCORE_PRESET` is used.
- **-ab** or **--alphabeta**: Enable alpha-beta pruning.
- **-a** or **--auto**: Selects auto mode.
- **-v** or **--versus**: Selects versus mode.
- **-m** or **--manual**: Selects manual mode.

Note that the options -a, -v and -m are exclusive. In total, you can choose one out of 4 program modes:
- Auto: Similar to **main.cpp** Task 2.
- Versus: Play against the algorithm, which plays automatically.
- Manual: Similar to **main.cpp** Task 1.
- Hinted (default): Similar to **main.cpp** Task 1, but with score evaluation and optimal move at every stage.

To execute the program, add the arguments after the program name, such as: `pa3_3.exe -ab -s -d 6` will run the program in Hinted mode with manual score input, alpha-beta pruning enabled and search depth of 6.

In addition, we provide the following program to convert a board ID to its board configuration as a helper program. It requires the following 2 arguments:

- **-s [num]** or **--size [num]**: Specify board size.
- **-i [num]** or **--id [num]**: Specify board id.

For example, executing `id2board.exe -s 3 -i 4721` outputs:

```
=============
| O | X | O |
=============
|   | X | X |
=============
|   | O |   |
=============
```

- Helper program (Windows): id2board.exe

Note: If your machine cannot run the executables above, you may consider using one of the Windows virtual machine to run the Windows demo program.

End of Download

Testing

The skeleton files also come with 2 folders: **inputs** and **outputs**. These files are for you to compare your program output (compiled with **main.cpp**). They are named according to the task being tested, and the constants defined in **const.h**:

- **input1_{}.txt**: Testing task 1
  - **input1_3_draw.txt**: Compiled with `BOARD_SIZE = 3`. The game is played until the end.
  - **input1_3_win.txt**: Compiled with `BOARD_SIZE = 3`. The game ends early because O has a match.
  - **input1_4_draw.txt**: Compiled with `BOARD_SIZE = 4`. The game is played until the end.
  - **input1_4_win.txt**: Compiled with `BOARD_SIZE = 4`. The game ends early because X has a match.
- **input2.txt**: Testing task 2
  - **output2_3_4.txt**: Compiled with `BOARD_SIZE = 3` and `DEPTH_PRESET = 4`.
  - **output2_3_9.txt**: Compiled with `BOARD_SIZE = 3` and `DEPTH_PRESET = 9`.
  - **output2_4_4.txt**: Compiled with `BOARD_SIZE = 4` and `DEPTH_PRESET = 4`.
  - **output2_4_7.txt**: Compiled with `BOARD_SIZE = 4` and `DEPTH_PRESET = 7`.
- **input3.txt**: Testing task 3
- **input4.txt**: Testing task 4
  - **output4_3_4.txt**: Compiled with `BOARD_SIZE = 3` and `DEPTH_PRESET = 4`.
  - **output4_3_9.txt**: Compiled with `BOARD_SIZE = 3` and `DEPTH_PRESET = 9`.
  - **output4_4_4.txt**: Compiled with `BOARD_SIZE = 4` and `DEPTH_PRESET = 4`.
  - **output4_4_7.txt**: Compiled with `BOARD_SIZE = 4` and `DEPTH_PRESET = 7`.

You can ignore the time duration printed in tasks 2 and 4 as they are machine-dependent (they will not show up in ZINC testing), but the remaining components should be the same as we have specified how the functions should be implemented. If your code passes these test cases, feel free to make adjustments to main.cpp, or compile the **main_program.cpp** version and compare with the given demo programs.

On ZINC, we will use another main source file to compile and test your submitted code. Note that unlike the previous assignment, we will not have batch grading before the deadline, so you are expected to test your code locally with the above given files, as well as testing on the CS Lab 2 machine to make sure there is no memory leak/address issue/unpredictable behaviour.

End of Testing

Submission and Deadline

Deadline: 23:59:00 on *May 9, 2023*.

ZINC Submission

Please submit the following files to ZINC by zipping the following 3 files. ZINC usage instructions can be found here.

```
board.cpp
boardtree.cpp
hastable.cpp
```

Notes:

- The compiler used on ZINC is `g++11`. However, there shouldn't be any noticeable differences if you use other versions of the compiler to test on your local machine.
- **We will check for memory leak/address issues in the final grading**, so make sure your code does not have any memory leak. You are strongly recommended to test your code on the CS Lab 2 machine to ensure your code will work on ZINC.
- ZINC grading will allow 4 GB of memory usage. If your program uses more than allowed for some test case, it will be marked as fail. Note that the most "memory consuming" test case will not be more than the given 4x4 board with depth 7 test case.
- You may submit your file multiple times, but only the latest version will be graded.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.
- If you have encountered any server-side problems or webpage glitches with ZINC, you may post on the ZINC support forum to get attention and help from the ZINC team quickly and directly. If you post on Piazza, you may not get the fastest response as we need to forward your report to them, and then forward their reply to you, etc.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online autograder ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts you cannot finish, just put in a dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
  return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded()
{
}
```

Reminders

Make sure you actually upload the correct version of your source files - we only grade what you upload. Some students in the past submitted an empty file or a wrong file or an exe file which is worth zero mark. So **you must double-check the file you have submitted**.

Late Submission Policy

There will be a penalty of -1 point (out of a maximum 100 points) for every minute you are late. For instance, since the deadline for assignment 3 is 23:59:00 on *May 9*, if you submit your solution at 1:00:00 on *May 10*, there will be a penalty of -61 points for your

End of Submission and Deadline

Frequently Asked Questions

Q: My code doesn't work / there is an error. Here is the code. Can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own, and we should not finish the tasks for you. We might provide some very general hints, but we shall not fix the problem or debug it for you.

Q: Can I add extra helper / global functions?
A: You may do so in the files that you are allowed to submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?
A: No. All libraries have been included in the given header files.

Q: Can I use global or static variables such as `static int x`?
A: No.

Q: Can I use `auto`?
A: No.

End of Frequently Asked Questions

Changelog

23/4/2023: Added a notice about ZINC memory usage in the Submission & Deadline section.

End of Changelog

Further notes

There are many other optimizations that can be done to improve the performance of the minimax algorithm, such as looking at "good" moves first to further utilize alpha-beta pruning, or deleting old nodes to allow more memory space for deeper search. If you found this assignment fascinating, you may be interested in taking up COMP4451 - Game Programming, which will also introduce minimax along with other game algorithms. The topic of trees, hash tables and how to optimize them may also be covered in courses such as COMP3711 or COMP5711.

Menu

Page maintained by

DINH Anh Dung
Email: dzung@ust.hk
Last Modified: 04/23/2023 15:20:02

Homepage

Course Homepage