Data Structures

BLG 223E

Project 3

Res. Assist. Enes Erdoğan
erdogane@itu.edu.tr

Instructors:
Prof. Dr. Tolga Ovatman
Asst. Prof. Dr. Yusuf Hüseyin Şahin

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 17.12.2024

1. Simple Bot for Connect-Four Game

In this homework, you are expected to implement a simple bot for the popular Connect Four game that is a simple turn-based two-player game. The core challenge lies in leveraging a *generic tree structure* to systematically explore and analyze all possible moves and their potential outcomes (up to a certain depth). By constructing such a tree, a bot can select the best move at any given board state that is most likely to lead to victory.

Let's first introduce the rules of the game, then the generic tree structure will be discussed to implement the game tree. Finally, we will combine these ideas to design a game bot.

1.1. Connect-Four Game

Connect-Four is a classic two-player strategy board game where two players take turns dropping colored discs into a vertically suspended grid as shown in the right figure. The grid is typically 7 columns wide and 6 rows high (different grid sizes are also possible). The objective is to be the first player to form a horizontal, vertical, or diagonal line of four of their own colored discs.

Thanks to its simplicity, this game is a good way of practicing the generic tree structure via implementing a simple algorithm that finds the best move.

1.2. Beyond the Binary Trees: Generic Trees

Previous discussions throughout this course mostly focused on binary trees, where each node has exactly two children. However, many real-world problems require more flexible tree structures. Generic trees allow a node to have an arbitrary number of children, making them more versatile for complex scenarios.

```c
typedef struct TreeNode {
    void* data;
    int num_children;
    struct TreeNode **children; // To keep the array of TreeNode pointers
} TreeNode;
```

This structure replaces fixed "left and "right" pointers with a flexible array of child pointers, enabling the representation of more complex hierarchical relationships. The functions you have learned for binary trees are easily expandable for generic trees. In this assignment, you will only need a subset of those functions.

It is obvious that a game tree for the Connect 4 game requires the tree to be generic since the branching factor for the game is not just more than 2, but also it is not a constant number. A classic Connect 4 board has seven columns, which means each game state can potentially branch into seven different subsequent states, and if some columns are full then the branching factor will be less than seven.

1.3. Designing a Bot

In developing a bot for the Connect 4 game, the primary challenge is identifying the best move given the current game state. Let's first define the game state and the notion of the next state.

The game state represents the complete configuration of the game board at any given moment. This includes the current position of all discs on the board and which player's turn it is. The term "next state" refers to all possible game states that can be reached by making a legal move from the current state.

At any given game state, we can construct a tree, where the root node keeps the current game state and its child nodes represent all possible next states resulting from legal moves. This tree can be systematically expanded to explore potential board configurations after multiple moves, alternating between players.

An example of such a game tree is shown in figure 1.1. As can be seen, different layers represent moves by alternating players i.e. the first layer is for yellow player moves, the second layer is for red player moves, and the final layer is again for the yellow player moves.

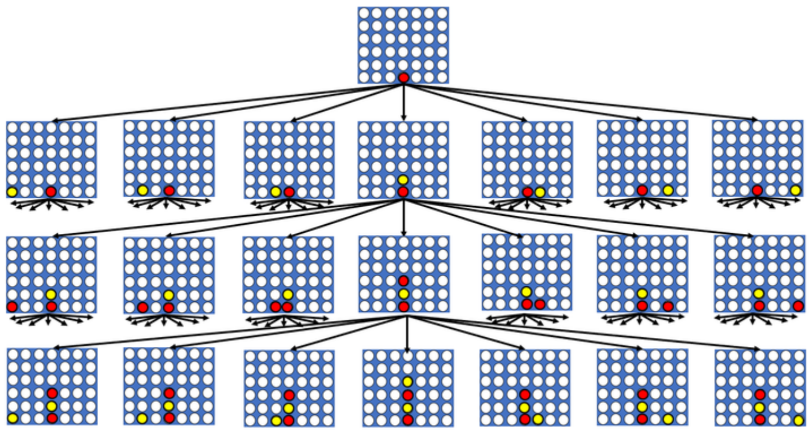

Figure 1.1: An example game tree

However, exploring every possible game state is computationally infeasible due to the game's relatively high branching factor i.e. it has an exponential complexity in terms of memory, and it is approximately 7^N where N is the depth of the tree. Thus, constructing and evaluating a complete game tree that explores all possible move sequences would require too much memory.

To address this complexity, a game bot typically employs a more nuanced approach. Instead of exhaustively mapping every possible game outcome, a board evaluation function can be used. This function assigns a numerical score to a given board state, indicating its potential favorability for a specific player. So, after constructing the game tree with a specified depth, boards at the leaf nodes can be evaluated so that program can decide which moves can lead to a winning position.

1.3.1. Rational Opponent

The key detail here is that opponent's strategic reasoning must be accounted, i.e. the bot should always assume that the opponent plays the most rational move in any situation. This introduces the need for the minimax algorithm, a decision-making strategy that assumes both players are rational and will make the best possible moves given their current understanding of the game state.

Minimax operates on a fundamental principle: while one player (the maximizing player) seeks to maximize their potential score, the opponent (the minimizing player) simultaneously attempts to minimize the first player's advantage.

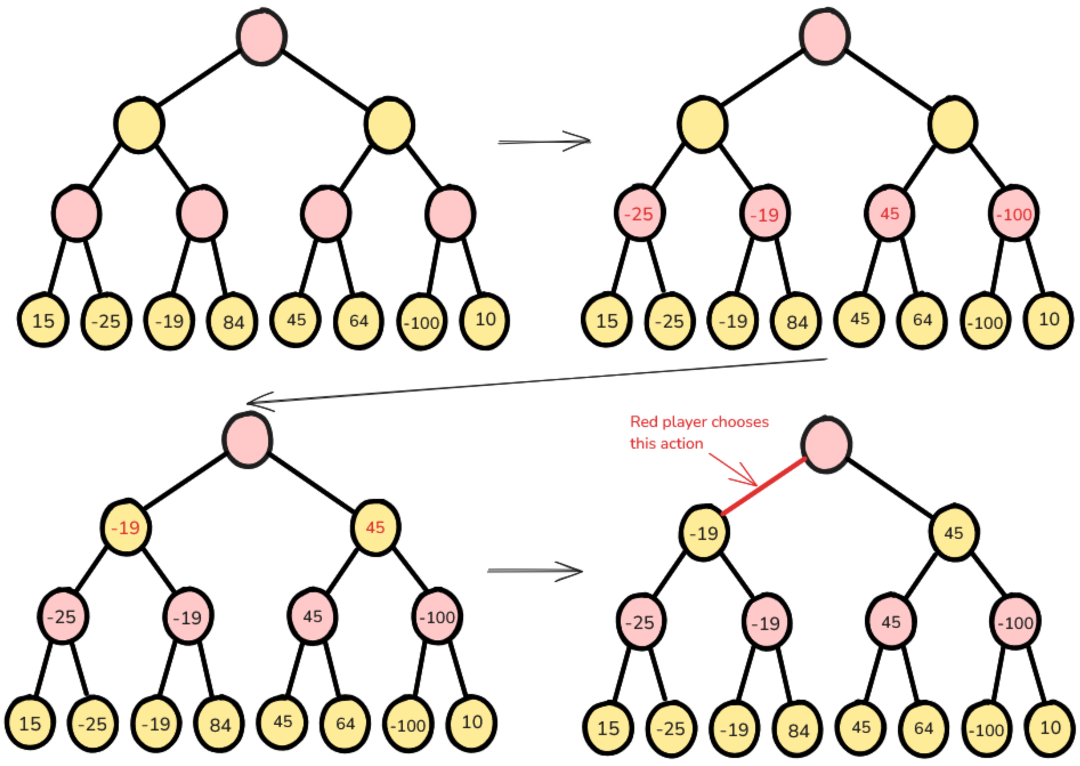

Figure 1.2: Step-by-step minimax evaluation for choosing the best action

An example of minimax algorithm is illustrated in figure 1.2. In the first step, the tree is constructed up to a specified depth, then the leaf nodes are evaluated. Positive evaluations indicate the advantage of the yellow player, and negative evaluations indicate the advantage of the red player. Basically, the magnitude of the number represents the degree of advantage, while its sign indicates which player holds that advantage. In this figure, red is the minimizer, and yellow is the maximizer. So in the next step, red nodes prefers the minimizing moves. Then in the third step, yellow nodes takes the maximizing moves among their children. Finally, root chooses the left move.

2. Implementation

You are given the skeleton code for this project. It consists of four components:

- `connect4.h/connect4.c`: Game-logic related functions
- `tree.h/tree.c`: Tree-related functions
- `game_bot.h/game_bot.c`: Board evaluation function, minimax algorithm, etc.
- `interface.h/interface.c`: Game loop related functions

Please read all header files carefully, and try to understand their functionality and their purpose before starting to code. Also, do not change the signature of the available functions in any files. However, you can declare new functions as much as you need. Some part

2.1. The Game

The Connect-Four game is **already implemented for you**. So, you should not modify `connect.h` and `connect.c` files. You are not required to check the implementation of the game, but you need to understand what each function does.

The GameState is one of the central definitions in this project, and as its name implies it is for keeping the state of the game.

```
typedef struct GameState {
    char *board;
    bool next_turn;
    int evaluation;
    int width;
    int height;
} GameState;
```

The important functions are the followings:

```
// From the connect4.h file
...
// Initialize the game state with and empty board
GameState *init_game_state(int width, int height);

// Given a board, modifies the "moves" array with
// the available moves and return the number of available moves.
int available_moves(GameState *gs, bool *moves);

// Given a board and a move, allocates a new GameState
// object with move applied and returns it.
GameState *make_move(GameState *gs, int move);

// Given a board, return the game status (it returns an enum object)
GameStatus get_game_status(GameState *gs);
...
```

For further details, please check the `connect.h` file. The `main.c` is prepared so that you can play the game by yourself.

2.2. Game Tree

The node for the game tree defined as the following:

```
typedef struct TreeNode {
    int num_children;
    GameState *game_state;
    struct TreeNode **children;
} TreeNode;
```

As a data, it keeps an `GameState` object. As mentioned earlier, this is a node for a generic tree, so you need to keep track of the number child of nodes. Important functions for the tree implementation are the following:

```
// From the tree.h file:
TreeNode *init_node(GameState *gs);
TreeNode *init_tree(GameState *gs, int depth);
void expand_tree(TreeNode *root);
void free_tree(TreeNode *root);
...
```

`init_tree` function constructs a game tree to the specified depth by generating all possible moves at each node, then returns the root node You can assume that the depth is equal or greater than 2. For an already initialized tree, it is sometimes needed to expand all the leaf nodes one more layer. To this end, implement the `expand_tree` function.

During the implementation, you should comply with the natural ordering of the children, in other words, the first legal move corresponds to the first child, the nth legal move corresponds to the nth child so on and so forth.

2.3. Minimax Algorithm

The minimax algorithm is intuitively explained in the previous section. More concretely, to implement the minimax, you have two functions called `get_min` and `get_max`. They are recursive functions that explore the game tree to find the best move for minimizing and maximizing players respectively:

`get_max` tries to maximize the score for the maximizing player by:

- Checking if the game is over (returning the terminal value)
- Recursively calling `get_min` on child nodes
- Selecting the maximum score from possible moves

`get_min` does the opposite, trying to minimize the score for the minimizing player by:

- Checking if the game is over (returning the terminal value)
- Recursively calling `get_max` on child nodes
- Selecting the minimum score from possible moves

These functions work in tandem, switching between maximizing and minimizing perspectives with each recursive call. When a leaf node is reached, the function returns the final evaluation score. During the tree traversal, each function calls the opposite function to simulate an opponent making the most strategically beneficial move.

2.4. User Interface

Until this point, each piece for the game is described separately. To make this implementation a fully functioning game, it is required to call those functions in a loop. In this section, the overall game loop is explained.

First, initialize the tree with an empty GameState, then in a loop that will go until the termination of the game:

- If it is the bot's turn, find the best move via minimax, or else play a random legal move

- Apply the move to the game tree (update the root)
- Prune the unnecessary subtrees (subtrees for not selected moves)
- Expand the tree one more layer. If the node count is lower than the threshold, expand one more layer.

In the `interface.c` file, you only need to implement the `apply_move_to_tree` function, which is responsible for the last three items in the list.

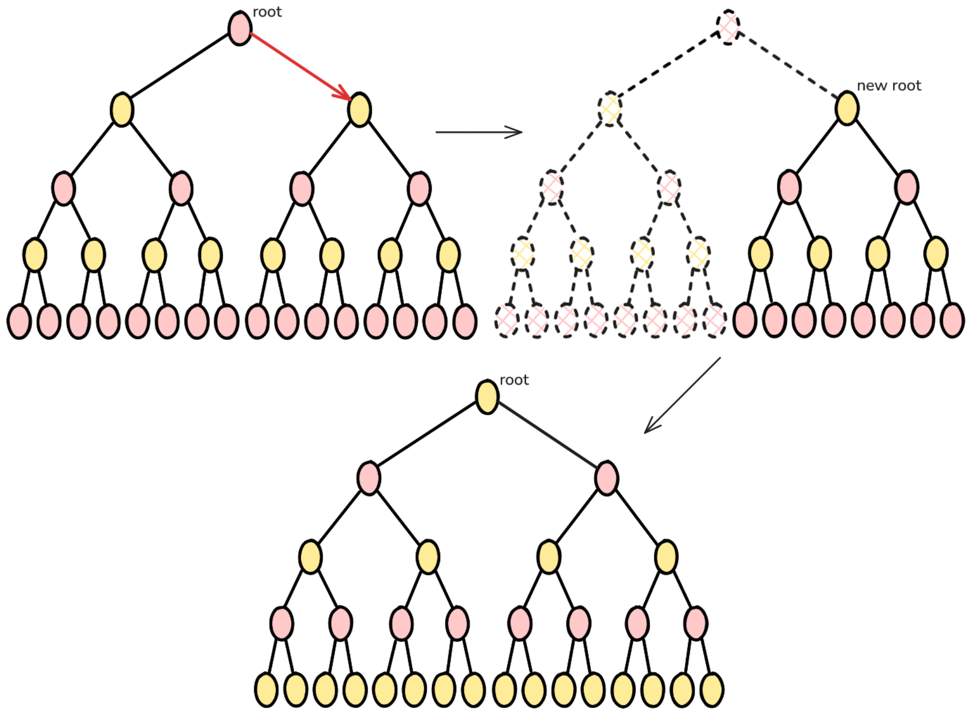

Figure 2.1: `apply_move_to_tree` illustrated

This function is illustrated in the figure 2.1. First, there is a tree, and the selected move is right. The root node must be updated and irrelevant parts of the tree must be freed. Finally, the expand tree function is supposed to be called.

3. Compiling and Debugging

To make the coding experience smoother, Makefile and VS Code's JSON files are updated. We added a make rule called `valgrind`. So, after completing the implementation, you can now call `make valgrind` and it will create a `valgrind-out.txt` file that shows your memory leaks and errors if there are any. Also, it suggested for you to look for the other make rules in the file.

Additionally, to find potential bug sources, it is suggested to compile with `-Wall -Werror` flags in the Makefile, but this is completely optional.

Also, you can now directly debug the test files. Select the "Test (build and debug)" option in the debug menu in the VS Code interface.

4. Submission

As mentioned earlier, you have already provided the skeleton code, so you shouldn't submit the whole project. We prepared a **submission script** for you that creates the zip folder automatically. You must use this script:

```
sh submission_script.sh
```

Then directly upload the created zip file to Ninova.

Also, note that the provided test files will not be the only ones used for grading. Your code will be evaluated on additional test cases, so make sure your implementation is robust and can handle various scenarios.

Good luck! For any questions, feel free to e-mail me (erdogane@itu.edu.tr)

IMPORTANT NOTE:

- You must stick with the container environment that is provided to you. This is crucial for the grading of your work. If your code can not be compiled in the container, you will receive a zero grade.
- Copying code fragments from any source, including books, websites, or classmates, is considered plagiarism.
- You are free to conduct research on the topics of the assignment to gain a better understanding of the concepts at an algorithmic level.
- Refrain from posting your code or report on any public platform (e.g., GitHub) before the deadline of the assignment.
- You are **NOT** permitted to use the STL for any purposes in this assignment.
- Only C language is allowed. Do not use C++ in anyway.
- Your code will be analyzed via Valgrind for memory leaks. Check all of the memory allocations and make sure you don't have any memory leaks. If you have any, it will be penalized severely.
- Additionally, please refrain from using any AI tools to help you complete your work. While I cannot directly detect AI-generated code, relying on such tools is not in your best interest. Everything you learn in this class forms the foundation for future courses, and if you do not build a strong foundation now, you will likely struggle in the semesters to come.