**A black and white logo

AI-generated content may be incorrect.Game Tree Implementation with Backtracking for a Two-Player Strategy Game**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Ahmed Sameh  236322 | Ahmed Khaled  236664 | Hassan Ahmed  229591 |  |

Group: 33

Analysis Of Algorithms  
24CSCI01I

Faculty of Informatics & Computer Science, The British University In Egypt  
Cairo, Egypt

Submitted to Dr Mostafa Salama

2025

**Introduction**

This project is based on a two-player board game designed to apply the concepts of game trees and backtracking as part of the Analysis of Algorithms course. The game takes place on a 5×5 square grid and involves two players: one controlling the Red tokens (Player A) and the other controlling the Green tokens (Player B). Each player starts with three tokens, and the objective is to move these tokens across the board to reach the goal zone. Red pieces begin on the left and move horizontally to the right, while Green pieces start at the top and move vertically downward. The first player to get three tokens into their respective goal zone wins.

The game takes alternating turns between the players, allowing them to either make a normal move into an empty adjacent cell or jump over an opponent’s piece into a free square two spaces ahead. If there are two tokens of the same color standing next to each other, the other player cannot make a move. If no legal move is available, the player’s turn is skipped. The project allows a human player to play against the computer, and the game includes a graphical interface built using Windows Forms in C++/CLI. It also visually tracks player turns and checks win conditions.

**Backtracking and Game Tree Algorithm**

The computer's decision-making in this game is powered by a recursive backtracking algorithm based on the concept of a game tree. Each game state is defined by the positions of all pieces and the current player’s turn. The algorithm explores possible future states using depth-first search, aiming to find a path that leads to a win.

The game tree is built recursively by generating all legal moves for the current player and applying each one to create a new game state. These new states are then explored in the same way. After evaluating a state, the algorithm returns to the previous one to explore other move options, which is the essence of backtracking.

To avoid recalculating the same state multiple times, the algorithm keeps track of already visited states using a map called memoMinimax. Each state is converted into a string that uniquely represents the board and the current player. This helps reduce unnecessary computation and speeds up the process of finding the best move.

The goal is to find a good move, meaning a move that leads to a win or to a situation where the opponent has no winning options. If the current player can move into a state that is bad for the opponent (where all their options are good for the current player), then the state is considered good. The algorithm uses this logic to evaluate the quality of each move and recommend the best one.

When multiple moves have the same evaluation score, the algorithm uses extra criteria to decide. It prefers moves that advance a piece to the goal zone, allow jumping over an opponent’s token, or block the opponent from moving forward. These rules help the AI play more strategically and make it harder to beat.

**Pseudocode for Backtracking Algorithm**

**Function: opponent()**

This function switches turns between players. 'R' becomes 'G' and vice versa. This is used during the recursion when alternating turns.

function opponent(player):

if player == 'R':

return 'G'

else:

return 'R'

**Function: getAllPossibleMoves()**

This function scans the board to find all of the player's tokens and determines what legal moves they can make (either a step or a jump).

function getAllPossibleMoves(state, player):

moves = []

for each row in state.board:

for each column in row:

if the cell has player’s token and not in goal zone:

if next cell is empty:

add regular move to moves

else if next cell has opponent and the cell after is empty:

add jump move to moves

return moves

**Function: applyMove()**

Creates a new game state by applying the move and changing the turn to the opponent. This is used for simulating the next step in the game tree.

function applyMove(state, move, player):

newState = copy of state

move player’s token from move.from to move.to

newState.currentPlayer = opponent(player)

return newState

**Function: findBestMove()**

This is the heart of the recursive backtracking algorithm. It simulates all possible moves, alternating players, and classifies states as:

* 'g' if the current player can force the opponent into a losing position
* 'b' if all paths lead to the opponent winning
* 'n' if depth limit is reached and no clear result is found

function findBestMove(state, player):

bestMove = null

for each move in getAllPossibleMoves(state, player):

newState = applyMove(state, move, player)

result = evaluateGameState(newState, opponent(player), 1)

if result == 'b':

return move // this move guarantees a win

return first move in list // fallback if no winning move

**Function: evaluateGameState()**

This is the decision maker. It tries all moves and checks if any lead to a guaranteed win (evaluateGameState returns 'b' for opponent = good for us). If so, it returns that move. If not, it defaults to any legal move.

function evaluateGameState(state, player, depth):

if player has won in state:

return 'g' // good outcome

if opponent has won in state:

return 'b' // bad outcome

if depth > MAX\_DEPTH:

return 'n' // neutral outcome (depth limit reached)

for each move in getAllPossibleMoves(state, player):

newState = applyMove(state, move, player)

result = evaluateGameState(newState, opponent(player), depth + 1)

if result == 'b':

return 'g' // found a winning path

return 'b' // all moves lead to bad outcome

**Time Complexity Analysis**

1. **opponent(player)**

* One conditional check.
* Time Complexity**:**

1. **getAllPossibleMoves(state, player)**

* Loop over the board: n × n cells
* For each cell:
  + If it has a player's token and it's not in the goal zone, check possible directions (at most 2: step or jump)
* Time Complexity:

1. **applyMove(state, move, player)**

* Make a copy of the board: O(n²)
* Move one token and switch turn: O(1)
* Time Complexity:

1. **evaluateGameState(state, player, depth)**

* This is the recursive backtracking function.
* Each call:
  + Gets possible moves → O(n²)
  + For each of O(n) moves:
  + Applies the move → O(n²)
  + Makes a recursive call to itself with depth + 1
* So, we define the recurrence:
  + Now we will solve using backward substitution:
  + → equation 1
  + → equation 2
  + By Substitution 2 in 1
  + → equation 3
  + → equation 4
  + By Substitution 4 in 3
  + solving this recurrence:
* Time Complexity:

1. **findBestMove(state, player)**

* Get all possible moves → O(n²)
* For each of O(n) moves:
* Apply move → O(n²)
* Call evaluateGameState →

**Total Time Complexity:**

**Space Complexity Analysis**

1. **opponent(player)**

* Only returns one char variable.
* Space Complexity**:**

1. **getAllPossibleMoves(state, player)**

* Stores all legal moves in a list:
  + Maximum number of moves can reach up to 2 moves per token which means that the complexity is , which can be simplified into.
* Space Complexity:

1. **applyMove(state, move, player)**

* Make a new copy of the board each time it’s called which has a complexity of
* Moves a token and checks turns which is takes about O(2) per each call which is
  + Total complexity is which can be simplified into.
* Space Complexity:

1. **evaluateGameState(state, player, depth)**

* This is the recursive backtracking function.
* Each call:
  + It calls the applyMove again creating a new copy →
  + Stores the legal moves in a list using getAllPossibleMoves →
  + Makes a recursive call to itself with depth + 1
* So, we define the recurrence:
* Now we will solve using backward substitution:
  + → equation 1
  + → equation 2
  + By Substitution 2 in 1
  + → equation 3
  + → equation 4
  + By Substitution 4 in 3
  + solving this recurrence:
* Space Complexity:

1. **findBestMove(state, player)**

* Gets the list of legal moves →
* Call evaluateGameState which has a complexity of →
* Space Complexity:

**Total Space Complexity:**

**Game Screenshots and Descriptions**

A screenshot of a game

AI-generated content may be incorrect.

Initial game state: Red (R) tokens on the left, Green (G) tokens on top. Ready to begin.  
Notice that the current turn is set a “None”, This means that the player is free to choose whichever color they desire.

A screenshot of a game

AI-generated content may be incorrect.

After the first Human move: The AI player (R) starts and moves one token forward. Notice how the current turn changed to “G” symbolizing that the human picked the green tokens to play with. Additionally, notice how the computer-generated move appears in the logs.

A screenshot of a game

AI-generated content may be incorrect.

Blocking Conditions: After some moves it is now green’s (Human) turn, and the AI has reached a state where it blocked green’s only available move. In this case the game should recognize that green does not have a legal move and skip their turn. See the next picture for more clarification.

A screenshot of a game

AI-generated content may be incorrect.

Human turn skipped: Human had no legal moves and was forced to skip. Then the computer played their next move.

A screenshot of a game

AI-generated content may be incorrect.

AI wins: The Red player (AI) has reached the goal area with all tokens.