**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.

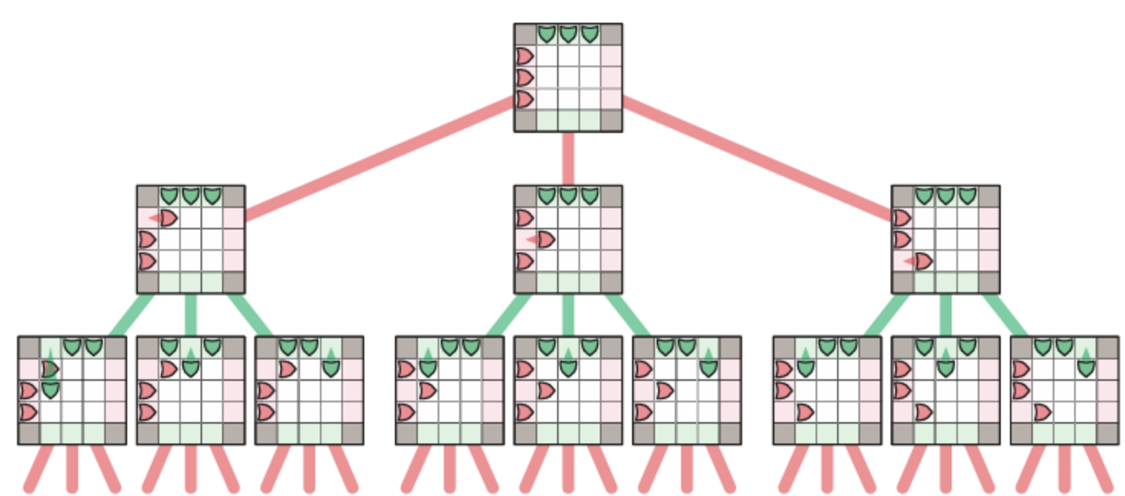
**Backtracking and Game Tree Algorithm**

The computer’s decision-making in this game is handled through a recursive backtracking algorithm that explores possible moves by building a game tree. Each state in the tree represents the current board layout and the player whose turn it is. The algorithm explores future states using depth-first search, aiming to find a path that leads to a win.

To construct the game tree, the algorithm generates all legal moves for the current player and applies each move to create a new game state. These new states are then explored recursively in the same way. After a state is evaluated, the algorithm backtracks to the previous one and continues exploring other possible moves. This process continues until a win condition is found or the maximum search depth is reached.

The goal of the algorithm is to identify the move that leads to the most favorable outcome. A move is considered good if it leads to a win or puts the opponent in a difficult position. The evaluation function assigns scores to non-terminal states by considering factors such as how close tokens are to the goal zone, whether a jump move is possible, and whether the opponent is close to winning.

When multiple moves result in similar evaluations, the algorithm uses additional criteria to choose the best one. It prefers moves that advance a token into the goal zone, enable a jump over an opponent’s piece, or block the opponent’s movement. These rules help the AI play more effectively and provide a more challenging experience for the human player.

**Game Tree Analysis**

The AI uses a recursive backtracking algorithm to explore all possible moves within the game. This function systematically generates each potential move, applies it to the current game state, and evaluates the resulting configuration. It continues this process recursively, diving deeper into the game tree until it reaches either a winning or a losing state or identifies the most optimal move based on the evaluation criteria. To manage the sequence of moves and enable reversal when necessary, the AI maintains a stack that records previous board states. This allows it to efficiently backtrack to earlier positions after simulating and analyzing each move, ensuring a comprehensive search through the solution space.

**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 generates and returns a list of all valid moves for the given player based on the current game state. It checks each of the player's tokens and adds moves that follow the movement rules, including normal steps and valid jump moves, while ignoring tokens already in the goal zone.

function getAllPossibleMoves(state, player):

moves = []

for each cell (row, col) on the board:

if cell contains player's piece and is not in the goal zone:

if player is Red:

if right cell is empty:

add move to the right

if can jump over Green to an empty cell:

add jump move to the right

else if player is Green:

if cell below is empty:

add move downward

if can jump over Red to an empty cell:

add jump move downward

return moves

**Function: isLosingState ()**

checks if the opponent has won the game. It simply calls isWinningState for the opposing player..

function isLosingState(state, player):

return isWinningState(state, opponent of player)

**Function: applyMove()**

It updates the game state by performing the given move for the specified player. It moves the player's token to the new position, clears the old position, switches the turn to the opponent, and saves the current state to a stack for easy backtracking.

function applyMove(state, move, player):

save current state on a stack

place player's token at the target position

remove the token from its original position

switch the turn to the opponent

**Function: findBestMove()**

It evaluates all possible legal moves for the current player and selects the one that leads to the most favorable outcome. It uses recursive backtracking to simulate future game states and assigns a score to each move based on how likely it is to lead to a win. The move with the highest score is returned as the best choice.

function findBestMove(initialState, player):

clear the game state stack

set workingState to the initial state

get all possible moves for the player

set bestScore to the lowest possible value

set bestMove to an invalid/default move

for each move in possible moves:

save current state on stack

apply the move

if this move results in a win:

restore the previous state from stack

return this move immediately

evaluate the move using backtracking

restore the previous state from stack

if the score is better than the current best:

update bestScore and bestMove

return the move with the best score

**Function: isInGoalZone ()**

checks if a given position is inside the goal zone for the specified player. Red’s goal zone is the rightmost column (column 4, rows 1–3), and Green’s goal zone is the bottom row (row 4, columns 1–3).

function isInGoalZone(row, col, player):

if player is Red:

return true if column is 4 and row is between 1 and 3

else: // player is Green

return true if row is 4 and column is between 1 and 3

**Function: evaluateGameState()**

This calculates a score that reflects how good the board is for the current player. It adds points for the player's progress and goal zone pieces, and subtracts points for the opponent’s progress, possible jumps, and goal threats. This score helps the AI decide which move is best.

function evaluateState(state, player):

opponent = getOpponent(player)

score = 0

for each cell on the board:

if it's player's piece:

add progress points (based on how far it moved)

add bonus if it's in the goal zone

if it's opponent's piece:

subtract their progress points

subtract if they're in the goal zone

get opponent's possible moves:

if they can jump:

subtract jump penalty

if they can reach their goal zone:

subtract goal threat penalty

return score

**Function: backtrack()**

A helper function that prepares for the recursive search by clearing the game state stack, then calls the main backtracking function to evaluate the best possible outcome for the current state.

function backtrack(state, depth, originalPlayer):

clear the game state stack

return the result of calling backtrack\_impl with the given state, depth, and player

**Function: isWinningState()**

checks if a player has at least three tokens inside their goal zone. If they do, the player is considered to have won the game.

function isWinningState(state, player):

count = 0

for each cell (row, col) on the board:

if the cell has player's piece and is in their goal zone:

increase count

return true if count is 3 or more

**Function: backtrack\_impl()**

This is a recursive function that searches through possible future game states using depth-first backtracking. It checks for winning or losing conditions, evaluates the board at a certain depth, and tries every legal move to find the best possible outcome. It returns a score that helps the AI choose the strongest move.

function backtrack\_impl(state, depth, originalPlayer):

save current state on stack

if originalPlayer has won:

remove saved state and return high score

if originalPlayer has lost:

remove saved state and return low score

if depth limit reached:

score = evaluate the state

remove saved state and return score

get all possible moves for current player

if it's originalPlayer's turn:

bestValue = very low

else:

bestValue = very high

for each move:

copy the current state

apply the move

result = call backtrack\_impl recursively with deeper depth

restore previous state from stack

if it's originalPlayer's turn:

bestValue = max(bestValue, result)

else:

bestValue = min(bestValue, result)

remove current state from stack

return bestValue

**Time Complexity Analysis**

1. **opponent(player) & isInGoalZone(row, column, player) & isJumpMove(move, player)**

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

1. **isWinningState(state, player) & isLosingState(state, player)**
   * Loops over the board: n x n cells
   * For each cell:
     + Checks if each token is in the goal zone and counts them
   * Time Complexity:
2. **getAllPossibleMoves(state, player)**

* Loop over the board: n x 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)**

* Move one token and switch turn and push current board state to the stack
* Time Complexity:

1. **evaluateState(state, player)**
   * Loop over the board: n x n cells
   * For each cell:
     + Checks if each move reaches the goal zone or can jump
   * Then it calls **getAllPossibleMoves()** and tests possible moves as well :
   * Time Complexity =
2. **backtrack\_impl (state, depth, player)**

* This is the recursive backtracking function.
* Each call:
  + Gets possible using **getAllPossibleMoves()** → O(n²)
  + For each of O(n) moves:
  + Applies the move → O(1)
  + 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 using **getAllPossibleMoves()** → O(n²)
* For each of O(n) moves:
* Apply move → O(1)
* Call **backtrack\_impl()** →
* Time Complexity:

**Total Time Complexity:**

**Space Complexity Analysis**

1. **opponent(player) & isInGoalZone(row, column, player) & isJumpMove(move, player)**

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

1. **isWinningState(state, player) & isLosingState(state, player)**
   1. updates one variable “counter”
   2. For each cell: Time Complexity:
2. **getAllPossibleMoves(state, player)**

* Stores all legal moves in a list
* Space Complexity:

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

* Pushes the current game state to the stack and consider the size of the board →
* Moves a token and switches the turn →
* Space Complexity:

1. **evaluateState(state, player)**
   1. calls the **getAllPossibleMoves()** function to generate moves to evaluate
   2. Space Complexity:
2. **findBestMove(state, player)**

* Gets the list of legal moves via **getAllPossibleMoves()** →
* Calles **applyMove()** which creates a copy of the state →
* Call **backtrack\_impl()** which has a complexity of →
* Space Complexity:

1. **backtrack\_impl (state, depth, player)**

* This is the recursive backtracking function.
* Each call:
  + It pushes the current board state into the stack →
  + Stores the legal moves in a list using **getAllPossibleMoves()** →
  + It calls the **applyMove()** functioncreating a new copy →
  + It calls the **evaluateState()** function →
  + 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:

**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.