Tic-Tac-Toe AI: Search Algorithms in Action

This project implements a classic Tic-Tac-Toe game where you can play against an AI opponent that uses different search algorithms. Choose between BFS, DFS, or A* to see how various algorithms approach the same problem.

👥 Our Team

1

Saif Eldeen Ahmed Lotfy

ID: 23030241

2

Mohamed Waleed Abdelgwad

ID: 23030176

3

Youssef Ahmed Hassan

ID: 23030218

4

Khaled Atef Mahmoud

ID: 23030064

5

Omar Ahmed Omar

ID: 23030122

6

Saeed Mamdouh Mohamed

ID: 23030086

7

Mayada Salah

ID: 23030198

🎯 Project Overview: Key Features

Three AI Algorithms

Choose between BFS, DFS, or A* to play against.

Simple UI

Clean Tkinter interface with easy controls.

Rematch Option

Quick rematch with the same algorithm.

Visual Feedback

Color-coded moves and game status for clarity.

🎮 Board Structure

The game uses a 1D array with 9 positions (0-8) to represent the 3x3 board.

0	1	2
3	4	5
6	7	8

🔵 BFS Algorithm (Breadth-First Search)

BFS explores all possibilities at the current level before moving deeper. In Tic-Tac-Toe, it systematically checks for winning moves, then blocking moves, and finally makes strategic choices.

01

Level 1: Immediate Win

Checks all 9 positions for an immediate winning move.

02

Level 2: Block Player

Checks all 9 positions to block an opponent's winning move.

03

Level 3: Strategic Play

Prioritizes center, then corners, then edges for optimal positioning.

💡 Key Insight

BFS explores systematically, ensuring no winning or blocking opportunity is missed by checking all positions at each decision level.

BFS Implementation

```
def bfs(self):
 """BFS: Check all moves one by one (breadth-first)"""

 # LEVEL 1: Check all positions for immediate win
 for i in range(9):
 if self.board[i] == '':
 self.board[i] = 'O' # Try AI move
 if self.is_winner('O'):
 self.board[i] = '' # Undo temporary move
 return i # Found winning move!
 self.board[i] = '' # Not a win, undo

 # LEVEL 2: Check all positions to block player
 for i in range(9):
 if self.board[i] == '':
 self.board[i] = 'X' # Simulate player move
 if self.is_winner('X'):
 self.board[i] = '' # Undo
 return i # Must block here!
 self.board[i] = ''

 # LEVEL 3: Strategic positioning
 # Take center if available (most valuable)
 if self.board[4] == '':
 return 4

 # Take a corner (second most valuable)
 for corner in [0, 2, 6, 8]:
 if self.board[corner] == '':
 return corner

 # Take any remaining position
 for i in range(9):
 if self.board[i] == '':
 return i
```

🟢 DFS Algorithm (Depth-First Search)

DFS follows a specific priority path deeply before exploring alternatives. It checks positions in a predetermined order: center first, then corners, then edges, creating a "depth-first" exploration pattern.

1

Priority Order

[4, 0, 2, 6, 8, 1, 3, 5, 7]

2

Depth 1: Win Check

Checks priority positions for an immediate win.

3

Depth 2: Block Check

Checks priority positions to block the opponent.

4

Depth 3: Strategic Move

Takes the first available position in the priority list.

💡 Key Insight

DFS prioritizes strategic positions like the center and corners, following a "depth-first" approach rather than a broad scan.

DFS Implementation

```
def dfs(self):
 """DFS: Check moves with priority (depth-first approach)"""

 # Define priority order: center > corners > edges
 # This creates the "depth" in depth-first search
 priority = [4, 0, 2, 6, 8, 1, 3, 5, 7]

 # DEPTH 1: Check priority positions for immediate win
 for i in priority:
 if self.board[i] == '':
 self.board[i] = 'O'
 if self.is_winner('O'):
 self.board[i] = ''
 return i # Winning move found
 self.board[i] = ''

 # DEPTH 2: Check priority positions to block player
 for i in priority:
 if self.board[i] == '':
 self.board[i] = 'X'
 if self.is_winner('X'):
 self.board[i] = ''
 return i # Block this move
 self.board[i] = ''

 # DEPTH 3: Follow priority order for strategic move
 for i in priority:
 if self.board[i] == '':
 return i # Take first available priority position
```

🟡 A* Algorithm (A-Star Search)

A* is an informed search algorithm that uses a heuristic function to evaluate move quality. Each position gets a score based on its strategic value, and the algorithm chooses the highest-scoring move, making it "smarter" than blind search algorithms.

Heuristic Scoring

Center (position 4): 4 points

Corners

(0, 2, 6, 8): 3 points

Edges

(1, 3, 5, 7): 2 points

💡 Key Insight

A* is "informed" by domain knowledge. It assigns scores based on how many winning lines each position contributes to, ensuring optimal play.

A* Implementation

```
def astar(self):
 """A*: Score each move and pick the best"""

 # STEP 1: Check for immediate win (highest priority)
 for i in range(9):
 if self.board[i] == '':
 self.board[i] = 'O'
 if self.is_winner('O'):
 self.board[i] = ''
 return i
 self.board[i] = ''

 # STEP 2: Check for blocking moves
 for i in range(9):
 if self.board[i] == '':
 self.board[i] = 'X'
 if self.is_winner('X'):
 self.board[i] = ''
 return i
 self.board[i] = ''

 # STEP 3: Use heuristic scoring
 scores = []

 # Define heuristic values for each position
 # Index: 0 1 2 3 4 5 6 7 8
 # Score: 3 2 3 2 4 2 3 2 3
 position_value = [3, 2, 3, 2, 4, 2, 3, 2, 3]

 # Calculate score for each empty position
 for i in range(9):
 if self.board[i] == '':
 score = position_value[i]
 scores.append((score, i)) # (score, position)

 # Sort by score (highest first)
 if scores:
 scores.sort(reverse=True)
 return scores[0][1] # Return position with highest score
```

📊 Algorithm Comparison & How to Run

All three algorithms play optimally for Tic-Tac-Toe, meaning they won't lose if played perfectly. However, they differ in their approach:

BFS

- Systematic exploration
- Never misses opportunities
- Checks all positions

DFS

- Follows smart priority order
- Efficient exploration path
- Good strategic positioning

A*

- Uses informed heuristic
- Mathematically optimal
- Adapts to board state

🚀 How to Run the Game

1. Save the code as `simple_tic_tac_toe.py`
2. Run: `python simple_tic_tac_toe.py`
3. Select your AI opponent (BFS, DFS, or A*)
4. Click any cell to make your move
5. Use "Rematch" or "New Game" for replays