Anaash Game

Group Information

Group Designation

- **Group Name**: Anaash_9

Group Members

1. **Lara Inês Alves Cunha**
   - **Student Number**: up202108876
   - **Contribution**: 50%
     - **Tasks Performed**:
       * Implementation of the `initial_state/2` predicate for initializing the game state.
       * Development of the `display_game/1` predicate for visualizing the board and game state.
       * Creation of the `game_over/2` predicate to determine when the game ends and the winner.
       * Auxiliary predicates for game initialization and visualization.
2. **Bernardo Marques Soares da Costa**
   - **Student Number**: up202207579
   - **Contribution**: 50%
     - **Tasks Performed**:
       * Implementation of the `move/3` predicate to validate and execute game moves.
       * Development of the `valid_moves/2` predicate to generate all valid moves for the current game state.
       * Creation of the `value/3` predicate to evaluate game states using heuristics.
       * Auxiliary predicates for move execution and game evaluation.

Installation and Execution

Prerequisites

Before running the game, ensure the following are installed:

1. **SICStus Prolog 4.9**
   - Download and install from the official website.
   - Follow the installation instructions for your operating system (Linux/Windows).

Steps for Execution on Linux

1. **Install SICStus Prolog**:
   - Follow the official instructions for installing SICStus Prolog on Linux. Use the terminal to verify the installation:
     `sicstus --version`
2. **Download the Project**:
   - Download the game files to your desired directory:
     `cd /path/to/anaash`
3. **Start the Game**:
   - Open the terminal and navigate to the project directory.
   - Launch SICStus Prolog:
     `sicstus`
   - Load the game file:
     `?- [game].`
   - Start the game by typing:
     `?- play.`

Steps for Execution on Windows

1. **Install SICStus Prolog**:
   - Download the SICStus Prolog installer for Windows from the official website.
   - Run the installer and follow the setup wizard to complete the installation.
2. **Download the Project**:
   - Download the ZIP file of the project to your desired directory.
3. **Start the Game**:
   - Open SICStus Prolog via the Start Menu.
   - Use the `cd/1` predicate to navigate to the directory containing the game files:
     `?- cd('C:/path/to/anaash').`
   - Load the game file:
     `?- [game].`
   - Start the game by typing:
     `?- play.`

Description of the Game

Overview

Anaash is a two-player strategy game designed by **Mark Steere** in February 2021. The game is played on a square checkerboard (typically 6x6 or 8x8), where

players control stacks of checkers of their respective colors, Red and Blue. The objective is to eliminate all the opponent's checkers by making positional moves, stacking friendly checkers, or capturing enemy stacks.

The name "Anaash" comes from the Mongolian word for "giraffe."

Rules

Initial Setup

- The board is filled with a checkered pattern of Red and Blue stacks.
- Each stack starts with a height of **1** (called a singleton).
- Red begins the game, followed by Blue, and players alternate turns.

Types of Moves Players can move one stack per turn. There are three types of moves:

1. **Positional Moves**:
   - Move a stack to an orthogonally adjacent, unoccupied square.
   - The destination must be one square closer (Manhattan distance) to the nearest stack, regardless of its color or height.
   - Only stacks with no orthogonal adjacencies (isolated stacks) can make positional moves.
   - Example: Refer to Figure 2 in the rulebook.
2. **Stacking Moves**:
   - Move a stack onto an orthogonally adjacent friendly stack of equal or larger height.
   - Example: Refer to Figure 3 in the rulebook.
3. **Capturing Moves**:
   - Capture an orthogonally adjacent enemy stack of equal or smaller height.
   - Example: Refer to Figure 4 in the rulebook.

Objective

- The game ends when one player captures all enemy checkers.
- There are no draws in Anaash, as at least one player will always have a valid move.

Additional Rules

- If a player has no available moves, they must pass their turn until a move becomes available.

Considerations for Game Extensions

Variable-Sized Boards

The game design has been implemented with support for variable-sized boards, allowing players to choose board dimensions between 5x5 and 10x10 during the initial game configuration. This flexibility was achieved by dynamically generating the board and adjusting the rules to accommodate different board sizes without impacting the gameplay mechanics. The `initial_state/2` predicate generates the board and ensures it adheres to the selected size, with all initial stacks properly positioned in the checkered pattern.

Optional Rules

To enhance accessibility for players of different skill levels, the game design includes optional rules that can be toggled based on player preference. Examples include: - **Simplified Rules for Novice Players**: These rules reduce the complexity of valid moves by limiting move options to basic positional moves. Capturing and stacking moves can be disabled to help new players familiarize themselves with the core mechanics. - **Additional Rules for Expert Players**: Advanced players can enable strategic constraints, such as requiring specific conditions to be met before performing a stacking or capturing move. For example, expert rules could introduce restrictions based on stack heights or limit the number of consecutive captures.

These optional rules can be seamlessly integrated by extending the `valid_move/3` and `choose_move/3` predicates to account for the chosen difficulty level or rule set.

AI Difficulty Levels

The game includes two AI difficulty levels: 1. **Level 1 (Random Moves)**: The AI selects valid moves randomly, providing a basic challenge for beginner players. 2. **Level 2 (Greedy Algorithm)**: The AI evaluates moves using the `value/3` predicate and chooses the move that maximizes its advantage, offering a more strategic challenge.

This modular design allows additional difficulty levels to be introduced in the future, such as implementing advanced AI algorithms (e.g., minimax or Monte Carlo Tree Search).

Enhanced User Experience

Several design considerations were made to improve the overall user experience: - **Interactive Menu**: The main menu provides intuitive navigation for game setup, including board size selection, game mode configuration, and difficulty level selection. - **Dynamic Feedback**: Players receive real-time feedback on invalid moves, game states, and when turns are passed due to unavailable

moves. - **Visual Enhancements**: The board is displayed with labeled rows and columns, and stack information is clearly represented, making the game state easy to interpret.

Future Extensions

The current design allows for the following potential extensions: - **Online Multiplayer**: Integrating network play for human-vs-human matches over the internet. - **Customizable Rules**: Adding a rule editor to enable players to create and test their own variations of the game. - **Scenario-Based Gameplay**: Introducing predefined scenarios where players start with specific board configurations and objectives, such as capturing a certain number of opponent stacks within a limited number of moves. - **Statistics and Leaderboards**: Tracking player performance, win rates, and AI difficulty success metrics to encourage competition and replayability.

These considerations ensure that the game design remains modular, scalable, and adaptable to future development needs.

Game Logic

The implementation of this game is structured around a modular and dynamic approach.

1. **Game Configuration Representation**

The game configuration encapsulates information about the board size, player types, and initial game conditions.This is managed as a list of attributes that include:

- Board Size: Defined by an integer between 5 and 10, chosen by the user at the start. This determines the number of rows and columns on the board.
- Player Types: Represented as a pair, such as human or computer(Level), where the level specifies the AI difficulty. The predicate `initial_state/2` initializes the game state based on these configurations. It generates the board structure and assigns player types and scores:

```
initial_state([size(Size), player_types(PlayerRed, PlayerBlue)], game_state(Board, red, Conf
```

2. **Internal Game State Representation**

The game state is encapsulated in the game_state/3 structure:

- Board: A 2D list representing rows of the board. Each cell is an atom (e.g., red(1), blue(2), or empty), indicating the owner and stack height.
- Current Player: Indicates whose turn it is, either red or blue.
- Configuration Details: Includes the board size, player types, and score for both players.

Example representations:

- Initial State:

```
game_state(
  [[red(1), blue(1)], [blue(1), red(1)]],  % 2x2 board
  red,
  config(2, [human, computer(1)], red(2)-blue(2))
).
```

- Intermediate State: Reflects a game in progress with updated stacks and player turns.
- Final State: Occurs when one player's stacks are entirely removed.

3. **Move Representation**

Moves are represented by the move(SRow, SCol, TRow, TCol) structure, where:

- SRow, SCol: Source row and column of the move.
- TRow, TCol: Target row and column of the move.

The `move/3` predicate validates and executes a move. Validity checks include: - Source cell belongs to the current player. - Target cell is within board bounds and reachable (Manhattan distance 1). - Move does not target the same cell as the source.

The board is updated with `execute_move/3` to reflect the move: - Stacks are adjusted based on interaction (e.g., merging or capturing). - Ownership of stacks is recalculated.

4. **User Interaction**

The game features an interactive menu system for configuration and play:

Menu Options: - Start a game (1), specifying board size and game type. - Exit the game (2).

Input Validation: - Ensures valid numerical inputs for board size, game mode, and moves. - Invalid inputs trigger a prompt for re-entry.

During gameplay: - Human players enter move coordinates interactively. - Computer players choose moves automatically, using strategies like random selection (Level 1) or greedy evaluation (Level 2).

Example user interaction for moves:

```
write('Enter source row: '), read(SRow),
write('Enter source column: '), read(SCol),
write('Enter target row: '), read(TRow),
write('Enter target column: '), read(TCol),
Move = move(SRow, SCol, TRow, TCol).
```

Input errors (e.g., out-of-bounds moves) prompt retry:

```
interactive_move(GameState, NewGameState) :-
    get_move_inputs(GameState, move(SRow, SCol, TRow, TCol)),
    move(GameState, move(SRow, SCol, TRow, TCol), NewGameState).
interactive_move(GameState, NewGameState) :-
    nl, write('Invalid move. Try again.'), nl,
    interactive_move(GameState, NewGameState).
```

This design balances flexibility and modularity, making it easy to extend or modify gameplay elements. User interaction is prioritized for ease of use, while Prolog's logical inference capabilities streamline game state management and AI decision-making.

Conclusion

The development of Anaash successfully brought the board game's strategic complexity to life in Prolog, leveraging the language's declarative nature for representing game states, validating moves, and implementing strategies. Key achievements include dynamic game configuration, user-friendly interaction, and modular design, making the game scalable for future enhancements. The inclusion of two AI difficulty levels—random and greedy algorithms—offers challenges tailored to different player skill levels, while Prolog's logical inference facilitates seamless state management.

However, the game has limitations that present opportunities for improvement. Input handling, though functional, can be cumbersome for users unfamiliar with command-line interfaces, and the AI strategies lack deeper predictive capabilities. Visualization is limited to text-based outputs (due to the prolog version via sicstus), which constrain accessibility and aesthetic appeal. Additionally, some edge cases in move validation and turn handling may require further refinement to ensure robustness.

Future developments like implementing advanced AI (e.g., minimax or Monte Carlo Tree Search) and creating a graphical user interface for enhanced player experience could come into view. Other potential extensions include customizable rules, leaderboards for competitive play, and dynamic AI difficulty adjustments.

With all these enhancements, Anaash could evolve into an engaging and polished game for a broader audience.

Bibliography

The following resources were used during the development of this assignment:

1. **Anaash Official Rulebook**
    - Source: https://www.marksteeregames.com/Anaash_rules.pdf

- Description: Official rulebook for the Anaash game, providing detailed information about the game's rules, setup, and move mechanics.

2. **SICStus Prolog Documentation**
   - Source: https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/Writing-Efficient-Programs.html

   - Description: Official documentation for SICStus Prolog, covering installation, usage, and best practices for writing efficient Prolog programs.

3. **AI and Prolog Game Strategies**
   - Source: https://arxiv.org/pdf/0911.2899

   - Description: Research paper on AI techniques for turn-based games, focusing on heuristic evaluations and game tree exploration in declarative languages like Prolog.