

a fg: AI for Games

Hollis Lehv, Hans Montero, Jillian Ross

cs4995.006 Final Project

Spring 2021

Abstract

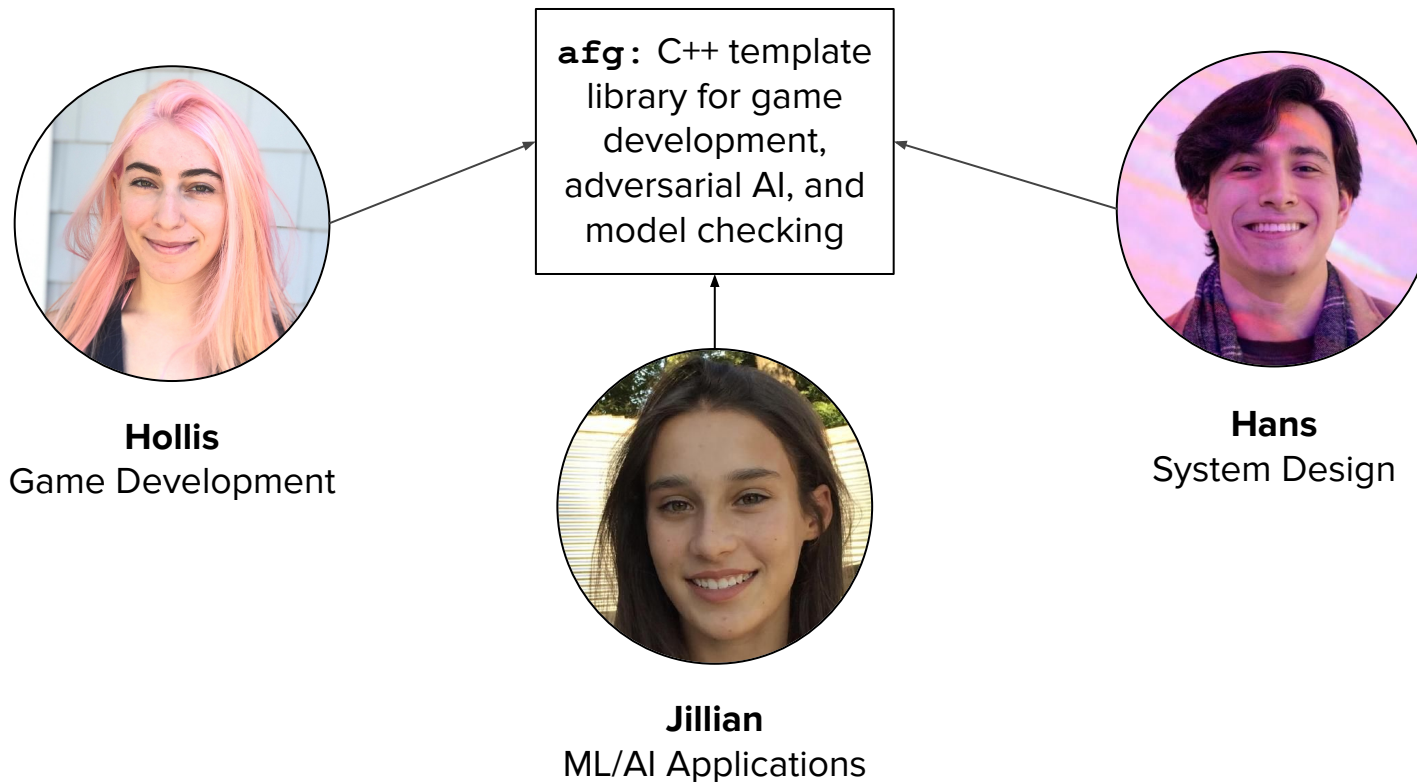
`afg` is a template library that aims to make game development in C++ more enjoyable and feasible. It offers powerful code abstractions from the fields of game development, adversarial AI, and model checking to enable game devs to write games from scratch faster with rich player and testing functionality. `afg` embraces compile-time polymorphism and presents a range of concepts for users to satisfy as opposed to offering abstract base classes for run-time polymorphism. Combined with modern C++ features, these design principles are meant to encourage developers to learn and try to use these elegant C++ constructs in their own implementations!

Estimated time: 30min + 15min Q&A

Overview

- Motivation
- afg Features
- Design
 - Game
 - Adversarial Artificial Intelligence
 - Model Checking
- Benchmarks
- Demo: Game of the Amazons
 - Implementation
 - AI Heuristics
 - Testing scenarios
- Future Work and Acknowledgements

Motivation



afg Features

- Abstract away boilerplate code and mechanics from two-player games
- Offer adversarial AI algorithms for use in creating intelligent opponents
- Provide simple model checking tools to verify implementation correctness

Design Principles

- Zero cost abstraction
- Compile-time polymorphism and concepts
- Header-only library
- Informed decision making
- Modern C++

Design (1/3): Game

- Supply **boilerplate** code for game logic, and allow players to fill in the details
- Understand that adversarial games have similar structure and try to avoid repeated code
- Use templates + concepts instead of inheritance to avoid run-time polymorphism
- Define **play()** function:
 - Calls setup defined by game programmer
 - Prompts for player moves and keeps track of allotted time using **chrono** library
 - Checks validity of players move using **isValid()** function
 - Makes move if valid using **makeMove()**
 - Once a terminal state is reached (**isTerminal()**), check for a winner using **isWinner()**

Game

- We require `afg::game::Playable`

```
template <class G>
concept Playable = requires(G m, G::move_t mv, ostream& os, istream& is) {
    { m.isTerminal() } -> same_as<bool>;
    { m.isWinner() } -> same_as<bool>;
    { m.getTurnCount() } -> same_as<int>;
    { m.getTurnParity() } -> same_as<int>;
    { m.getAvailableMoves() } -> same_as<vector<typename G::move_t>>;
    { m.makeMove(mv) } -> same_as<void>;
    { m.isValid(mv) } -> same_as<bool>;
    { m.setup() } -> same_as<bool>;
    { os << m };
    { os << mv };
    { is >> mv };
};
```


Design (2/3): AI

- Apply **minimax** with a **heuristic** over **state space** to find optimal **move** for a **player**
- **Minimax**: an adversarial recursive algorithm with optimality guarantees
- **Heuristic**: a game-specific evaluation function of a game state
- **State space**: a “playable” object capable of simulating all possible moves
- **Move**: a generic type defined by the game developer when creating the game
- **Player**: an “intelligent player” object with a heuristic function that evaluate state

Player

- For any game, we require `afg::game::Player`

```
template <class T, class G>
concept Player = requires(T player, G game) {
    Playable<G>;
    { player.getStrategy(game) } -> same_as<typename G::move_t>;
    { player.getTimeout() } -> same_as<double>;
    { player.getParity() } -> same_as<int>;
};
```

Player

- For a game that uses AI, we require `afg::game::IntelligentPlayer`

```
template <class T, class G>
concept IntelligentPlayer = Player<T, G> requires(T player, G game) {
    { player.heuristic(game) } -> same_as<int>;
};
```

- Relies on previously defined concept, `afg::game::Player`
- Allows seamless use of `afg::AI`

Minimax

- We require `afg::game::IntelligentPlayer` and `afg::game::Playable` for minimax

```
template <Playable GameType, IntelligentPlayer<GameType> P>
GameType::move_t minimax(const GameType& state, P player, int depth) {
    GameType::move_t bestMove;
    int alpha = std::numeric_limits<int>::min();
    int beta = std::numeric_limits<int>::max();
    GameType stateCopy = state;
    if (/* Player maximizes */) {
        maximizer(...);
    }
    else {
        minimizer(...);
    }
    return bestMove;
}
```

AI in Tic-Tac-Toe

```
const int MINIMIZER = -1;  
const int MAXIMIZER = 1;  
const int NEUTRAL = 0;
```

```
template<>  
int SmartPlayer<TicTacToe>::heuristic(const TicTacToe& state) {  
    if (state.isWinner()) {  
        if (/* is Maximizer */)   
            return MAXIMIZER;  
  
        return MINIMIZER;  
    }  
  
    return NEUTRAL;  
}
```

O	X	
	O	
	X	O

Design (3/3): Model Checking

- At its simplest: deploy **search agent** in **state space** to find **goal(s)**
- Search agent: an algorithm capable of traversing a state space
- State space: a “checkable” object capable of supplying neighboring states
- Goals: predicate functions that indicate whether a state is a goal

State Space

- We only require a subset of `afg::game::Playable`

```
template <class T>
concept Checkable = requires(T m, T other, T::move_t mv) {
    { m.isTerminal() } -> same_as<bool>;
    { m.getTurnCount() } -> same_as<int>;
    { m.getAvailableMoves() } -> same_as<vector<typename T::move_t>>;
    { m.makeMove(mv) } -> same_as<void>;
    { m == other } -> same_as<bool>;
    { std::hash<T>{}(m) } -> same_as<size_t>;
};
```

- `Checkable` concept gives us a way of generating a state space
- Explicit specialization of `std::Hash<T>`

Goal Functions

- A predicate can be represented as a function object

```
template <class Function, class Model>
concept Predicate = Checkable<Model> && requires (Function f, Model m) {
    { f(m) } -> same_as<bool>;
};
```

- More specifically, a constrained function object (better type checking!)
- Lambdas make more sense than function pointers here
- `std::function<T>` was fussy

Search Agents

- Depth-limited BFS searches

```
template<Checkable GameType, Predicate<GameType> Function>  
SearchResult<GameType> bfsFind(const GameType& initState, Function isGoal, int depthLimit);
```

```
template<Checkable GameType, Predicate<GameType> Function>  
SearchResult<GameType> pathExists(const GameType& initState,  
                                const vector<Function>& predicates, int depthLimit);
```

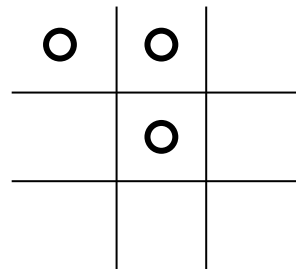
- Use `bfsFind()` to test the existence of a particular configuration
- Use `pathExists()` to test the existence of scenarios (sequence of configurations)
- Generators using C++ coroutines – not enough support

Model Checking Tic-Tac-Toe

```
/* Predicate #1: Board Config */
predicates.push_back(
    [](const TicTacToe& st) {
        return (st.board[0][0] == 'o'
                && st.board[0][1] == 'o'
                && st.board[1][1] == 'o');
    }
);

/* Predicate #2: Winner */
predicates.push_back(
    [](const TicTacToe& st) {
        return (/* Player 1 Won? */);
    }
);

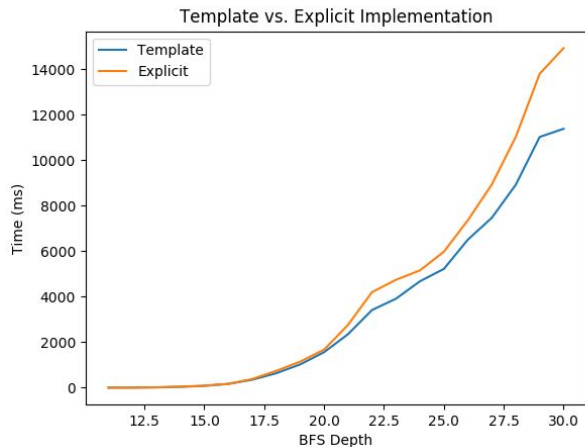
/* Check if it's possible to satisfy predicates #1 and #2 in 7 moves */
cout << Model::pathExists(ttt, predicates, 7) << endl;
```



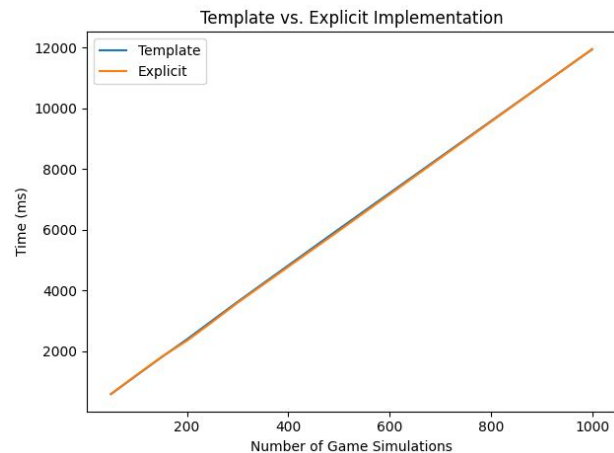
O	O	
	O	

Benchmarks (1/3): Zero Cost Abstraction

Exhaustive search during model check

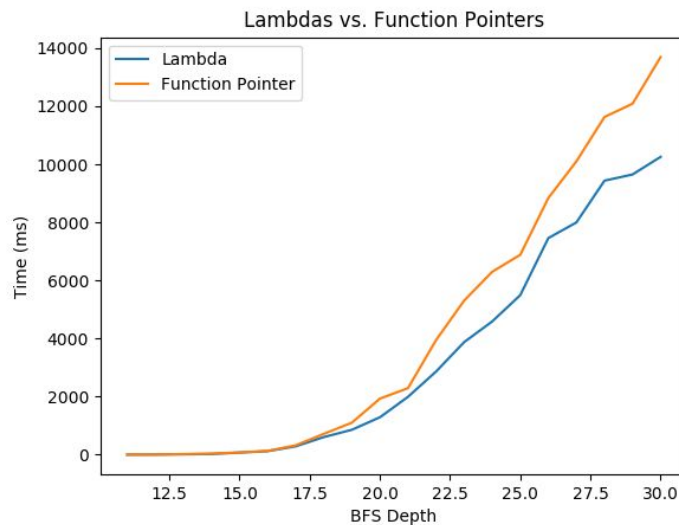


Game simulation for minimax



Benchmarks (2/3): Lambdas and Inlining

Predicates are called $O(2^d)$ times

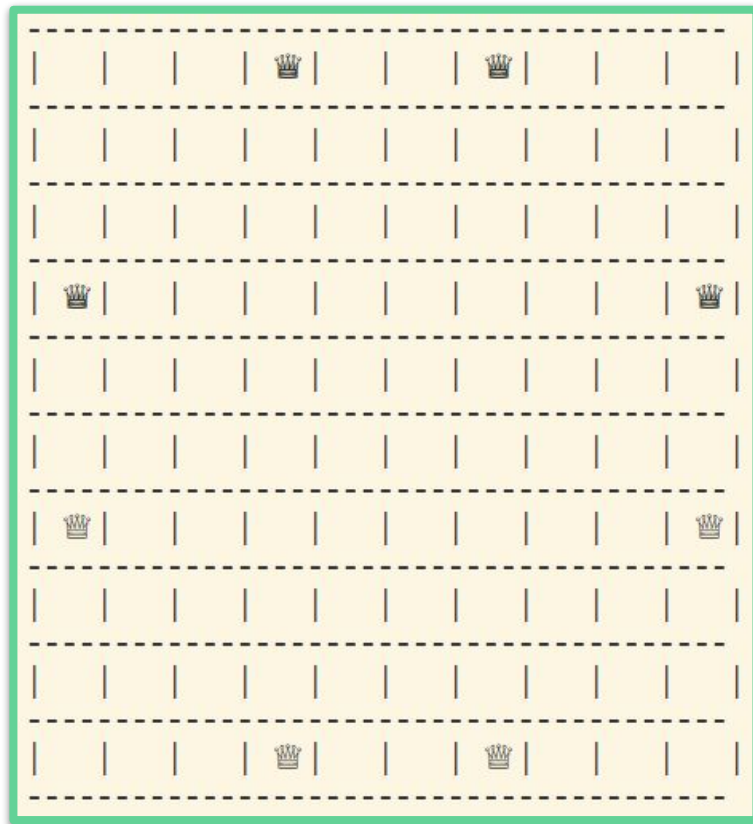
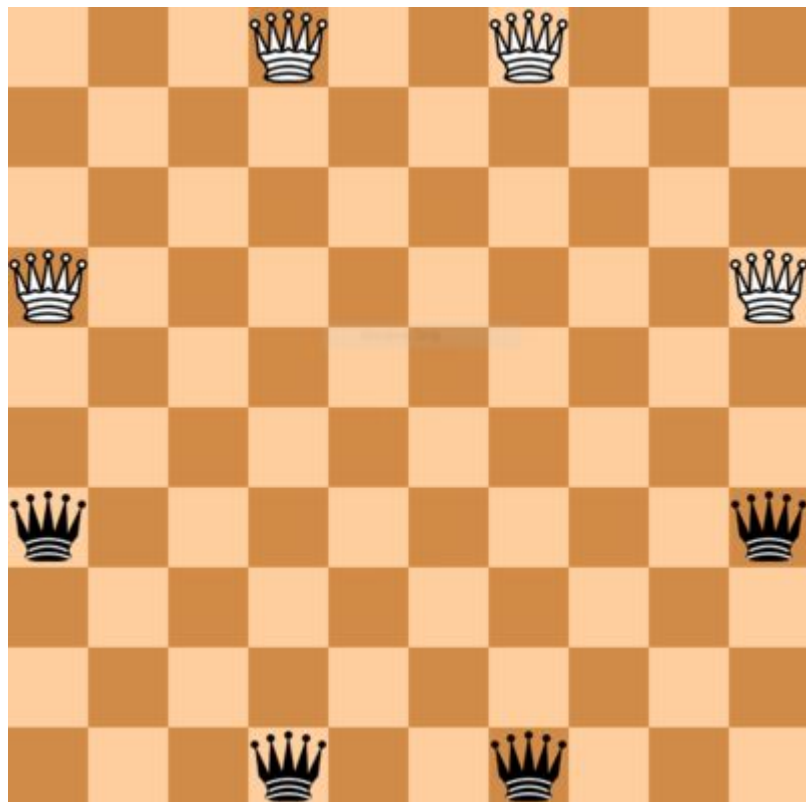


Benchmarks (3/3): TicTacToe LOC

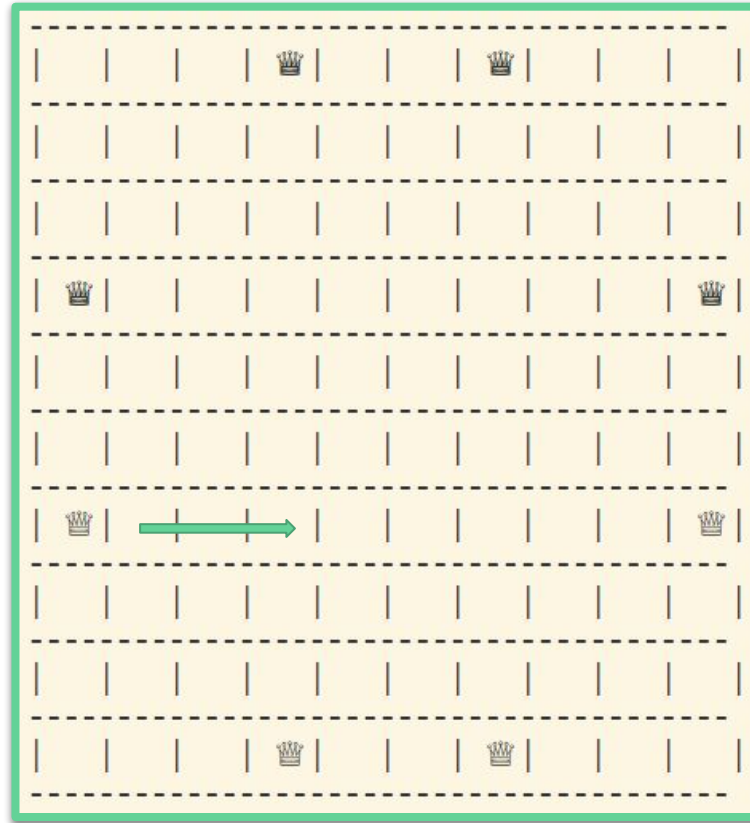
Implementation	Game LOC	AI LOC
GeorgeSeif/ic-Tac-Toe-AI	261	70
lukechu10/TicTacToe-Minimax	180	190
Prajwal-P/TicTacToe-with-AI	200	97
afg/TicTacToe	189	<10

Game of the Amazons

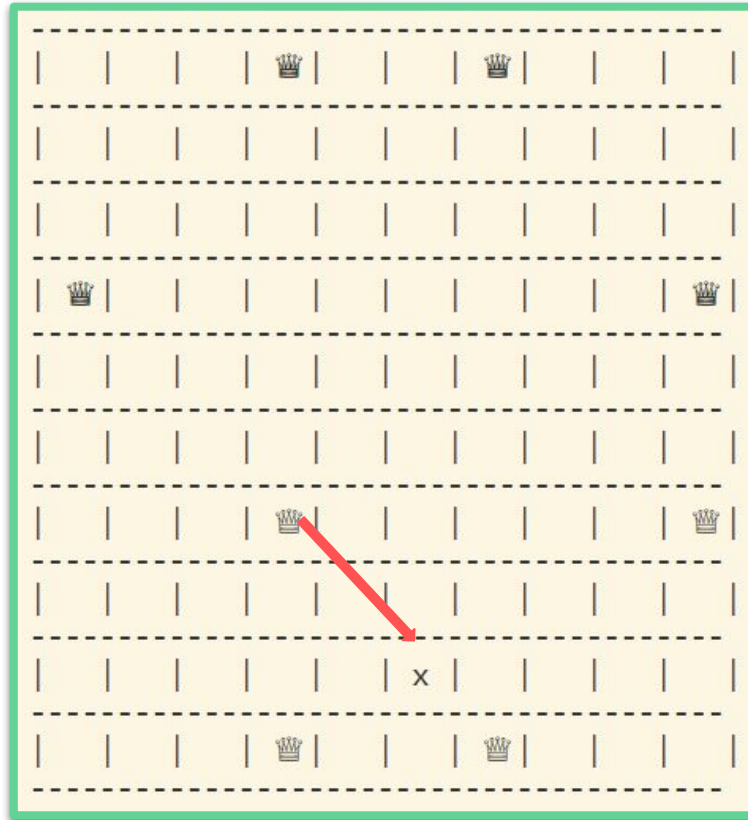
Game of the Amazons: Gameplay



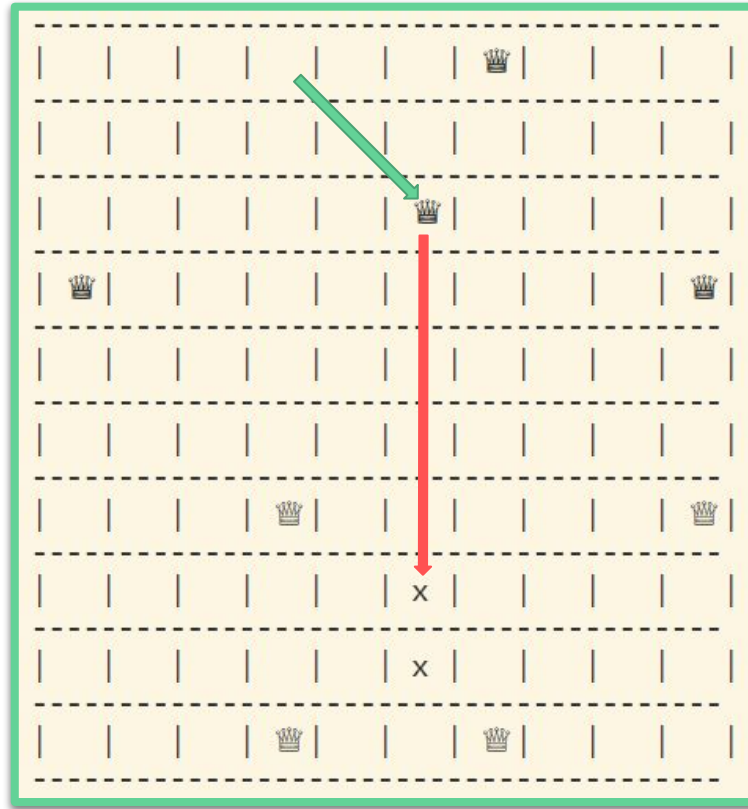
Game of the Amazons: Gameplay



Game of the Amazons: Gameplay



Game of the Amazons: Gameplay



Game of the Amazons: Code

```
template <class G>
concept Playable = requires(G m, G::move_t mv, ostream& os, istream& is) {
    { m.isTerminal() } -> same_as<bool>;
    { m.isWinner() } -> same_as<bool>;
    { m.getTurnCount() } -> same_as<int>;
    { m.getTurnParity() } -> same_as<int>;
    { m.getAvailableMoves() } -> same_as<vector<typename G::move_t>>;
    { m.makeMove(mv) } -> same_as<void>;
    { m.isValid(mv) } -> same_as<bool>;
    { m.setup() } -> same_as<bool>;
    { os << m };
    { os << mv };
    { is >> mv };
};
```

Game of the Amazons: Board

```
/* Subset of code */  
class Board {  
    public:  
        void print(bool instructions) const;  
        bool isValid(Move move, int turn) const;  
        void makeMove(Move move, int turn);  
        bool isWinner(int turn) const;  
        vector<Move> getAvailableMoves(int turn) const;  
}
```

Game of the Amazons: isValid()

```
bool Board::isValid(Move move, int turn) const
{
    /* Check that the move is not outside the bounds of the board */

    /* Check that queen starting position must indeed be occupied by a queen */

    /* Check that white cannot move black's queen and visa versa */

    /* End and firing positions must be empty (except firing equal to starting) */

    /* Starting and ending positions cannot be equal, queen cannot fire on a spot she's on */

    /* Check that the queen move is valid */
    if (!isValidMovement(move.queenStartingPos, move.queenEndingPos))
    {
        return false;
    }

    /* Check that firing move is valid */
    if (!isValidMovement(move.queenEndingPos, move.firePos))
    {
        return false;
    }

    return true;
}
```

Game of the Amazons: makeMove()

```
bool Board::makeMove(Move move, int turn)
{
    if (isValid(move, turn))
    {
        /* Add the correct colored queen to the ending square */

        /* Clear the starting square */

        /* Add the obstruction to the board */
    }
}
```

Game of the Amazons: Play!

```
int main(int argc, char **argv)
{
    HumanPlayer<Amazons> p1(/*timeout=*/0);
    HumanPlayer<Amazons> p2(/*timeout=*/0);

    Amazons amz;

    TPGame<Amazons, HumanPlayer<Amazons>, HumanPlayer<Amazons>> game(amz, p1, p2);

    game.play();

    return 0;
}
```

Game of the Amazons - Demo

AI in Game of the Amazons

```
template<>
int QuickPlayer<TicTacToe>::heuristic(const Amazons& state) {
    vector<Move> playerMoves = state.getAvailableMoves( /* Player */ );
    vector<Move> opponentMoves = state.getAvailableMoves( /* Opponent */ );

    return playerMoves.size() - opponentMoves.size();
}
```

Game of the Amazons - Demo

Game of the Amazons - Demo

Future Work and Acknowledgements

- Expand AI library
 - Include additional variants of minimax, such as expectiminimax and negamax
- Faster model checking
 - Allow for heuristics/costs to guide search and cut down execution time
 - C++ coroutines and generators
- cs4701: Artificial Intelligence (Minimax)
- cs4113: Distributed Systems (Model Checking)
- <https://github.com/rossjillian/afg>

Thank you!
