Halma Game

Luana Liao LL3637, Catherine Lyu hl3553

Introduction - What is Halma?

Figure 1

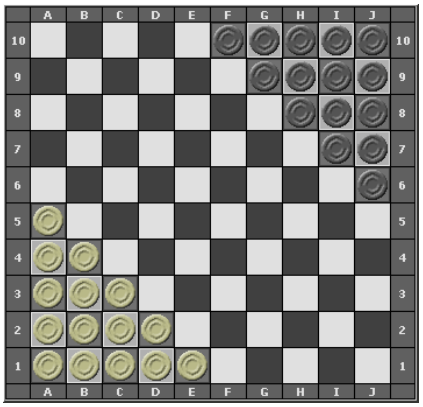

Figure 2

Halma is a strategic board game played on a checkers-like board involving moving and jumping pieces. Unlike Chinese Checkers, which is a variant played on a hexagonal board (Figure 1), Halma involves square tiles and two players with the goal of moving pieces across the board. In the two-player version we'll be implementing, each player has 10 pieces located at opposing corners (Figure 2 shows an example setup). The objective is to transfer all pieces from the starting "camp" to the opponent's "camp" before the other player. Players move one piece per turn, either by moving it to an adjacent empty square or by "jumping" over an adjacent piece, landing in the empty square directly on the other side. Multiple jumps in a single turn are allowed if they continue to land in open spaces. (Play Halma online against a computer here!)

This project was inspired by our interest in strategic games like Chinese Checkers, with Halma's grid layout making it easier to represent in code. Additionally, our familiarity with the minimax algorithm from a previous artificial intelligence class made it an exciting algorithm to parallelize in Haskell, especially as it presents opportunities for performance improvements due to Haskell's concurrent capabilities.

Project Overview

Our goal is to create a parallelized version of the minimax algorithm to determine the optimal moves in Halma. Minimax is a recursive algorithm widely used in two-player games. The algorithm evaluates moves in terms of maximizing the advantage for one player while minimizing it for the opponent. Due to the exponential growth of possible moves, the search space can become massive, making minimax computationally intensive. We would like to

integrate alpha-beta pruning into minimax to reduce unnecessary computations, pruning parts of the search tree that cannot possibly influence the final decision.

Algorithm / Parallelism Approach

Minimax Basics

The minimax algorithm alternates between “maximizer” and “minimizer” nodes, representing the two players. It assigns scores to each possible move based on how advantageous it is for the current player.

Alpha-beta pruning can be added to the algorithm to cut off branches that do not need to be evaluated, enhancing the algorithm’s efficiency by reducing the number of nodes in the search tree. Halma’s 10x10 grid and large search space mean a high branching factor. Parallelizing the minimax search will allow us to explore multiple move options concurrently.

Parallelization

We will parallelize the minimax search by splitting the computation at the higher levels of the search tree. Each immediate child node of the root (representing each initial move) will be processed in parallel, allowing concurrent evaluation of possible moves.

We will likely use Haskell’s Par monad and Strategies for handling parallel computations. The Par monad allows us to control parallel execution, while the Strategies library gives us more flexibility in forcing evaluation when required.

We will first implement a basic minimax algorithm with alpha-beta pruning in Haskell. Then, we’ll adapt this algorithm to distribute each move of the game tree across separate threads.

The input data for the algorithm consists of the current board state and available moves for each piece. Board states will be represented as matrices, with each position indicating the presence of a player’s piece or an empty space.

The initial setup, as shown in Figure 2, will be hardcoded for simplicity, though moves will be generated dynamically.