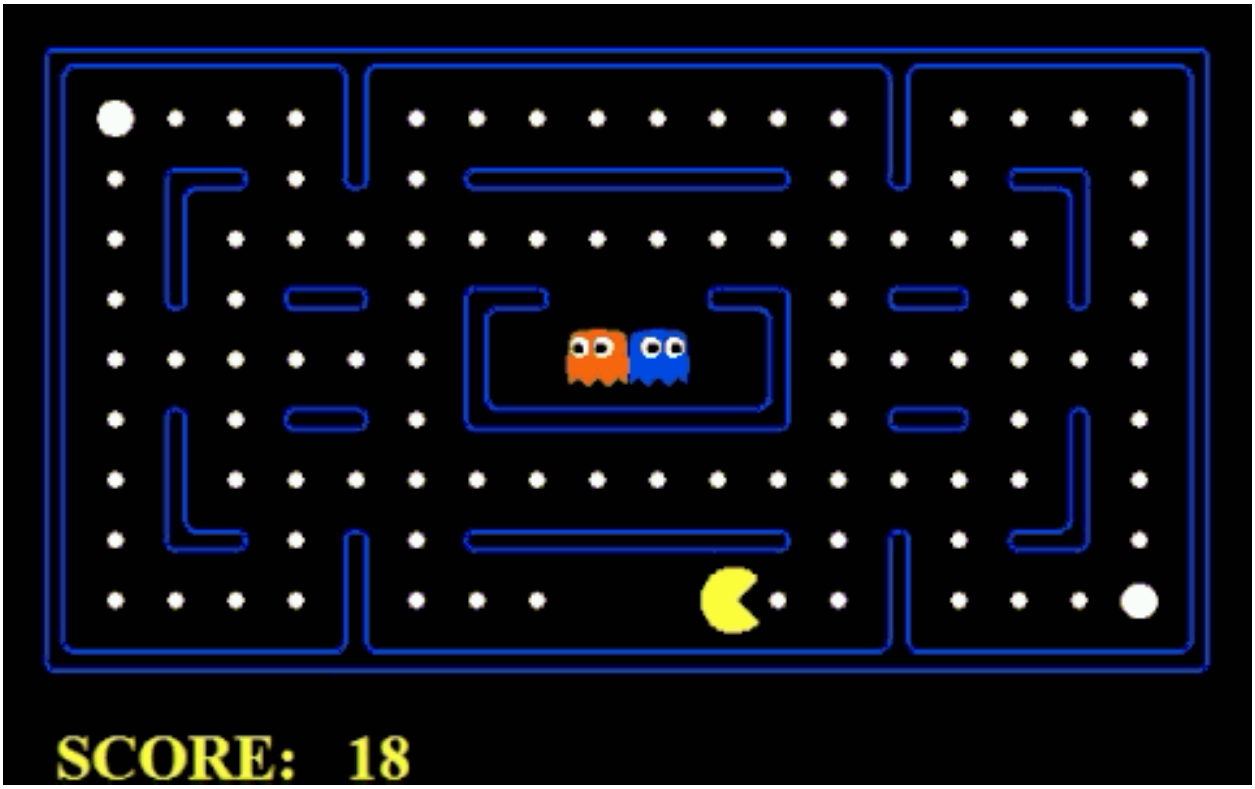
Parallel Functional Programming Final Project Report

Jonathan Chen (jyc2183), Kevin Wang (kjw2169)

Fall 2025

1 Introduction

Pac-Man is a classic arcade game which was developed by Namco in 1980. The game consists of a single player looking to navigate a board and consume all of the "pellets" present, while avoiding up to four ghosts that chase the player around the grid. Once a board is cleared of all "pellets", then the current level is considered "cleared" and the ghosts progressively become faster and faster. There are also power ups that allow the player to eat the ghosts, and special fruit on the board that are worth additional points for the player's score.



1.1 Game Modifications

For this project, we have made the following simplifications:

- **Objective:** Collect all pellets on the board while avoiding ghosts
- **Lives:** Pac-Man only has 1 life.
- **Powerups:** Removed all powerups.

- **Movement:** Pac-Man can move in four cardinal directions (up, down, left, right)
- **Scoring:** Each pellet collected awards 10 points
- **Loss Condition:** Collision with any ghost results in game over

2 Project Overview

Our objective was to develop an efficient AI solver for Pac-Man that:

1. Parses a text file as a Pac-Man board
2. Clears all pellets from that board configuration
3. Avoids ghost collisions through intelligent pathfinding
4. Demonstrates performance improvements through parallelization

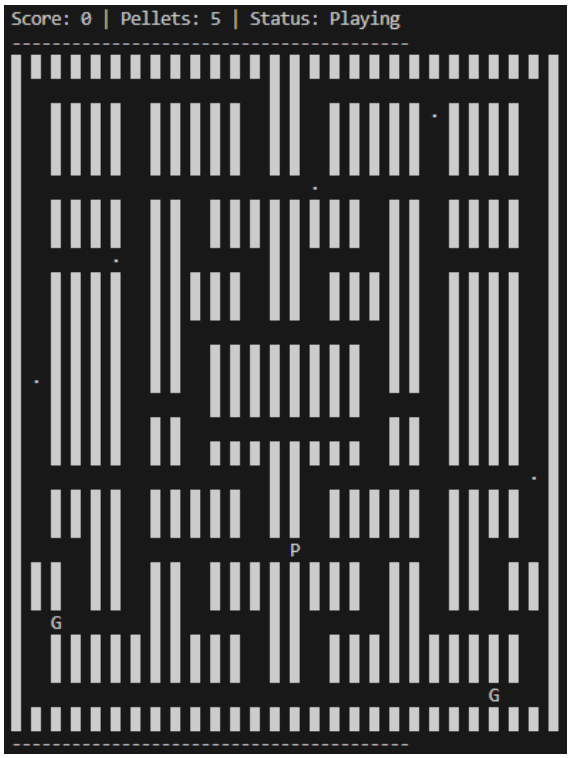
3 Parsing Input

Our input data consists of a text file containing grid (maze) information, Pac-Man's starting location, the ghosts' starting locations, and pellet locations.

Input File Format

```
1111111111111111111111111111
1000000000000110000000000001
1011110111110110111111*111101
1011110111110110110111110111101
1011110111110110110111110111101
1000000000000000*000000000001
1011110110111111111011011101
1011110110111111111011011101
10000*0110000110000110000001
1011110111110110111111011101
1011110111110110111111011101
1011110110000000000110111101
1011110110111111111011011101
1*1111011011111111011011101
1011110000111111111000011101
1011110110000000000110111101
1011110110111111111011011101
1000000000000110000000000*1
1011110111110110111111011101
1011110111110110111111011101
1000110000000P0000000110001
1110110110111111111011011011
1110110110111111111011011011
10G0000110000110000110000001
1011111111111011011111111101
1011111111111011011111111101
1000000000000000000000000G001
1111111111111111111111111111
```

Terminal Visualization



1=Wall 0=Empty *=Pellet P=Pac-Man G=Ghost

We implement this parsing in Game.hs as follows:

```haskell
parseGrid :: [String] -> ([[Cell]], Position, [Position], Int)
parseGrid lns = go lns 0 (0, 0) [] 0 []
    where
        go [] _ pacPos ghosts pellets gridRows =
            (reverse gridRows, pacPos, reverse ghosts, pellets)
        go (line:rest) y pacPos ghosts pellets gridRows =
            let (row, pacPos', ghosts', pellets') =
                parseLine line y 0 pacPos ghosts pellets []
            in go rest (y+1) pacPos' ghosts' pellets' (row:gridRows)

        parseLine [] _ _ pacPos ghosts pellets rowAcc =
            (reverse rowAcc, pacPos, ghosts, pellets)
        parseLine (c:cs) y x pacPos ghosts pellets rowAcc =
            case c of
                '1'->parseLine cs y (x+1) pacPos ghosts pellets (Wall:rowAcc)
                '0'->parseLine cs y (x+1) pacPos ghosts pellets (Empty:rowAcc)
                '*'->parseLine cs y (x+1) pacPos ghosts (pellets+1) (Pellet:rowAcc)
                'P'->parseLine cs y (x+1) (x,y) ghosts pellets (Empty:rowAcc)
                'G'->parseLine cs y (x+1) pacPos ((x,y):ghosts) pellets (Empty:rowAcc)
                ' '->parseLine cs y (x+1) pacPos ghosts pellets (Empty:rowAcc)
                _ ->parseLine cs y (x+1) pacPos ghosts pellets (Empty:rowAcc)
```

4 Minimax Algorithm for Multi-Agent Games

The minimax algorithm simulates all possible future moves and counter-moves in a game, constructing a decision tree where:

- **Max nodes:** Represent Pac-Man's turn (maximizing game state score)
- **Min nodes:** Represent the ghosts' turn (minimizing game state score)
- **Leaf nodes:** Terminal states or depth-limited states evaluated by a heuristic

In our implementation, Pac-Man acts as the maximizing player seeking to collect pellets and avoid ghosts, while the ghosts collectively act as minimizing players attempting to capture Pac-Man.

4.1 Game State Representation

Our implementation uses Haskell's type system to represent game states:

```haskell
data Cell = Wall | Empty | Pellet
    deriving (Eq, Show)

type Position = (Int, Int)

data Action = MoveUp | MoveDown | MoveLeft | MoveRight
    deriving (Eq, Ord, Show, Enum, Bounded)

data GameState = GameState {
    grid :: Vector (Vector Cell),
    Pac–ManPos :: Position,
    ghostPositions :: [Position],
    score :: Int,
    pelletsRemaining :: Int,
    isTerminal :: Bool,
```

```
    deathPos :: Maybe Position
} deriving (Eq, Show)
```

The board is represented as a 2D vector for efficient indexed access. This immutable structure allows safe sharing across parallel computations without synchronization concerns.

4.2 Heuristic Evaluation Function

Since exhaustive search is computationally infeasible for most Pac-Man boards, we rely on a heuristic evaluation function at depth-limited nodes:

```
evaluateGameState :: GameState -> Int
evaluateGameState gs
    | isTerminal gs = terminalPenalty gs
    | otherwise =
        let baseScore = score gs * 100
            pelletFactor = if pelletsRemaining gs == 0
                           then 5000 else 0
            distToPellet = case nearestPelletDistance gs of
                Nothing -> 0
                Just d -> negate (10 * d)
            ghostProxPenalty = negate (sum
                (map dangerFromGhost (ghostDistances gs)))
        in baseScore + pelletFactor
           + distToPellet + ghostProxPenalty
```

The evaluation function considers:

- **Base score:** Current score multiplied by 100
- **Pellet completion bonus:** 5000 points for clearing all pellets
- **Pellet proximity:** Negative weight proportional to distance to nearest pellet (encourages collecting nearby pellets)
- **Ghost danger:** Exponentially increasing penalty as ghosts get closer:
  - Distance 0 (same cell): -5000 (immediate death)
  - Distance 1 (adjacent): -500
  - Distance 2: -150
  - Distance 3: -50
  - Distance > 3: -10

This heuristic balances offensive play (collecting pellets) with defensive play (avoiding ghosts).

5 Sequential Solution

5.1 Minimax with Alpha-Beta Pruning

Our baseline implementation uses minimax with alpha-beta pruning to efficiently explore the game tree:

```
minimax :: GameState -> Int -> Int -> Int -> Bool -> Int
minimax gs depth alpha beta maximizingPlayer
    | isTerminal gs || depth <= 0 = evaluateGameState gs
    | maximizingPlayer =
        let actions = legalActionsPacman gs
        in if null actions
```

```
            then evaluateGameState gs
            else goMax alpha actions
  | otherwise =
      let jointGhostMoves = legalJointGhostActions gs
      in if null jointGhostMoves
            then evaluateGameState gs
            else goMin beta jointGhostMoves
where
  goMax a [] = a
  goMax a (act:rest) =
      let child = applyPacmanAction gs act
          val = minimax child (depth - 1) a beta False
          a' = max a val
      in if beta <= a' then a' else goMax a' rest

  goMin b [] = b
  goMin b (joint:rest) =
      let child = applyGhostJointActions gs joint
          val = minimax child (depth - 1) alpha b True
          b' = min b val
      in if b' <= alpha then b' else goMin b' rest
```

Key features:

- Alpha-beta pruning eliminates branches that cannot influence the final decision
- Pac-Man's turn generates 4 possible actions (up, down, left, right)
- Ghosts' turn generates joint actions (Cartesian product of each ghost's legal moves)
- Pruning occurs when $\beta \leq \alpha$, indicating no better move can be found in that branch

5.2 Ghost AI Strategy

Ghosts use greedy BFS pathfinding to pursue Pac-Man:

```
stepGhostsGreedy :: GameState -> GameState
stepGhostsGreedy gs =
    let distGrid = bfsDistances (grid gs) (pacmanPos gs)
        choose pos =
            let acts = filter (isLegalMove (grid gs) pos)
                              allActions
                nexts = [(movePosition pos a, a) | a <- acts]
                score np = maybe maxBound id
                                 (distanceAt distGrid np)
                best = minimumBy (comparing score) nexts
            in fst best
        movedGhosts = map choose (ghostPositions gs)
        collided = (pacmanPos gs) `elem` movedGhosts
    in gs { ghostPositions = movedGhosts
          , isTerminal = collided
                      || pelletsRemaining gs <= 0 }
```

This implementation:

1. Computes BFS distances from Pac-Man to all reachable cells
2. Each ghost independently selects the adjacent cell with minimum distance to Pac-Man

3. Checks for collision after all ghosts move simultaneously

BFS pathfinding provides more realistic and challenging ghost behavior than simple Manhattan distance, as it accounts for walls and actual navigable paths.

6 Parallel Solutions

While alpha-beta pruning significantly improves sequential performance, we further reduced runtime through parallelization. Haskell's pure functional paradigm and immutable data structures make parallelization straightforward, as there are no shared mutable states to synchronize.

6.1 Parallelization Strategy

We implemented parallel evaluation at the top levels of the minimax tree using Haskell's `Control.Parallel.Strategies` library:

```
minimaxPar :: GameState -> Int -> Int -> Int -> Int
           -> Bool -> Int
minimaxPar gs depth parLevels alpha beta maximizingPlayer
    | isTerminal gs || depth <= 0 = evaluateGameState gs
    | parLevels >= 0 && maximizingPlayer =
        let actions = legalActionsPacman gs
        in case actions of
            [] -> evaluateGameState gs
            _ ->
                let children = [applyPacmanAction gs a
                                | a <- actions]
                    vals = parMap rdeepseq
                           (\c -> minimaxPar c (depth - 1)
                            (parLevels - 1) alpha beta False)
                           children
                in maximum vals
    | parLevels >= 0 && not maximizingPlayer =
        let jointGhostMoves = legalJointGhostActions gs
        in case jointGhostMoves of
            [] -> evaluateGameState gs
            _ ->
                let children = [applyGhostJointActions gs jm
                                | jm <- jointGhostMoves]
                    vals = parMap rdeepseq
                           (\c -> minimaxPar c (depth - 1)
                            (parLevels - 1) alpha beta True)
                           children
                in minimum vals
    | otherwise = minimax gs depth alpha beta maximizingPlayer
```

Implementation details:

- `parMap rdeepseq`: Evaluates child states in parallel, forcing full evaluation before aggregating results
- **Configurable parallelism depth:** The `parLevels` parameter controls how many tree levels are parallelized
- **Sequential fallback:** Once `parLevels < 0`, the algorithm falls back to sequential minimax with alpha-beta pruning
- **Data parallelism:** Sibling nodes in the game tree are evaluated concurrently

6.2 Parallelism Levels

Our implementation uses 3 levels of parallelism for optimal performance:

- **Level 0 (root):** Pac-Man's 4 immediate action choices evaluated in parallel
- **Level 1:** Ghost responses to each Pac-Man action evaluated in parallel
- **Beyond Level 1:** Sequential minimax with alpha-beta pruning

This configuration balances parallelism benefits with overhead costs. Deeper parallelism provides diminishing returns due to:

- Thread creation/management overhead
- Reduced granularity at deeper levels
- Less effective alpha-beta pruning in parallel contexts

7 Performance Evaluation

We evaluated our parallel implementation against the sequential baseline using various board configurations.

7.1 Test Configuration

- **Hardware:** Apple M1 MacBook Pro (8 performance cores)
- **Test board:** `large.txt` (medium complexity maze)
- **Search depth:** 10 plies
- **Max planning steps:** 200
- **Parallel levels:** 3
- **Compilation:** GHC 9.10.3
- **Runtime flags:** +RTS -N (use all available cores)

7.2 Performance Results

Implementation	Runtime (s)	Speedup
Sequential	14.012s	1.0x
Parallel (2 Cores)	12.158	1.152492186
Parallel (3 Cores)	9.544	1.468147527
Parallel (4 Cores)	7.227	1.938840459
Parallel (5 Cores)	5.48	2.556934307
Parallel (6 Cores)	4.987	2.809705234
Parallel (7 Cores)	4.761	2.943079185
Parallel (8 Cores)	4.733	2.960490175

Table 1: Performance comparison of various levels of parallelism

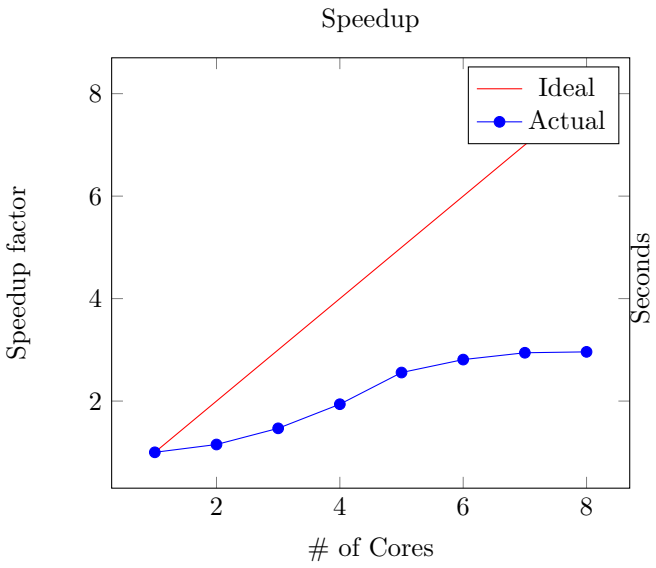


Figure 1: Observed speedup plot.

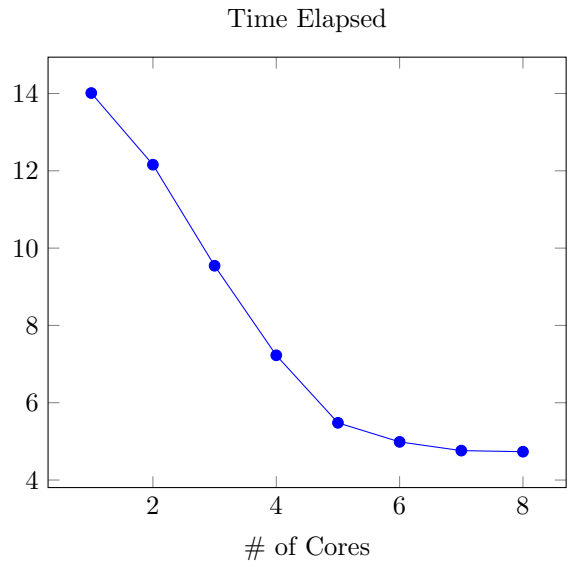


Figure 2: Time elapsed plot.

7.3 Expected Results and Analysis

Based on the parallel architecture, we expect an average 3x speedup on a 8-core M1 chip when compared to sequential minimax with alpha-beta pruning.

The actual speedup depends on several factors:

- **Branching factor:** Boards with more legal moves benefit more from parallelism
- **Search depth:** Deeper searches expose more parallelism opportunities
- **Pruning effectiveness:** Heavy pruning may reduce available parallel work
- **Hardware:** Core count and memory bandwidth affect scaling

7.4 ThreadScope Analysis

ThreadScope provides visualization of parallel execution:
Expected patterns in ThreadScope output:

- **Burst activity:** Spikes of parallel work at each Pac-Man decision point
- **Multi-core utilization:** All 8 cores active during parallel levels
- **Sequential phases:** Single-thread activity during deeper tree levels
- **GC events:** Periodic garbage collection (typically minor impact)
- **Idle periods:** Brief synchronization points between parallel phases

To generate ThreadScope data:

```
stack exec pacman-minimax-exe -- --grid grids/large.txt --par +RTS -N8 -l
threadscope pacman-minimax-exe.eventlog
```

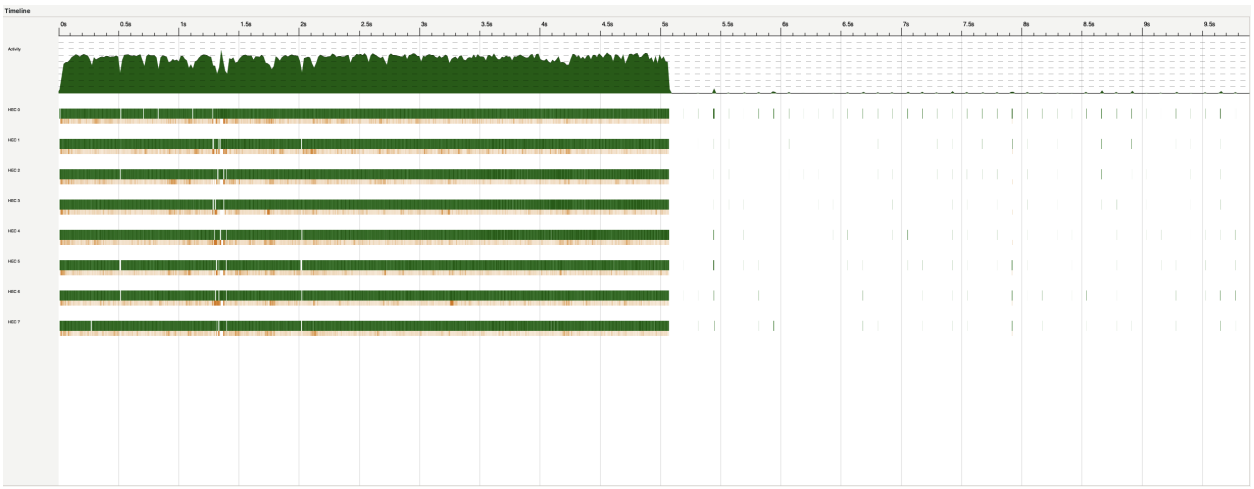


Figure 3: ThreadScope visualization showing parallel activity across cores. The period of high utilization is during the core minimax calculation, while the remaining portion of the chart is during the visualization run directly after Pac-Man's path is calculated. The visualization portion *appears* to have zero utilization, but in reality it is simply sleeping between frame renders.

8 Discussion and Further Considerations

8.1 Alpha-Beta Pruning in Parallel Contexts

One challenge of parallelizing minimax is that alpha-beta pruning becomes less effective when evaluating nodes concurrently. In sequential evaluation, early results can prune later branches, but parallel evaluation commits to exploring all sibling nodes simultaneously.

Our two-level parallelism strategy mitigates this issue by:

1. Parallelizing only the top levels where pruning is less effective
2. Falling back to sequential evaluation with aggressive pruning at deeper levels
3. Accepting reduced pruning at shallow levels in exchange for parallelism benefits

8.2 Ghost Coordination

In minimax, ghosts are modeled as a single adversarial player making joint decisions. This represents perfect coordination, making the problem harder than realistic Pac-Man where ghosts have independent AI patterns.

Alternative approaches could include:

- **Independent ghost moves:** Each ghost moves greedily without coordination (easier problem)
- **Limited lookahead:** Ghosts plan only 1-2 moves ahead
- **Stochastic behavior:** Add randomness to ghost movement

8.3 Heuristic Improvements

Our current heuristic could be enhanced with:

- **Pellet density analysis:** Favor regions with dense pellet clusters
- **Dead-end detection:** Strongly penalize positions that lead to dead ends

- **Ghost avoidance paths:** Compute escape routes and favor positions with multiple exits
- **Temporal reasoning:** Consider not just positions but timing of moves

8.4 Scalability Considerations

For larger boards or deeper search depths, additional optimizations could include:

- **Transposition tables:** Cache evaluated states to avoid redundant computation
- **Iterative deepening:** Gradually increase search depth, using previous results
- **Dynamic parallelism:** Adjust parallel levels based on available work

9 Next Steps

While our current implementation performs well on medium-sized boards, future work could explore more sophisticated parallelization strategies, improved heuristics, and optimizations for larger state spaces. Nevertheless, the project successfully demonstrates the viability of parallel minimax for solving complex adversarial games like Pac-Man.

References

[1] Science Museum - Pac-Man Turns 45 [online] Available at `https://blog.sciencemuseum.org.uk/pac-man-turns-45/` Accessed 16 Dec. 2025

[2] Wikipedia - Monte Carlo tree search [online] Available at `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search` [Accessed 15 Nov. 2025]

[3] Wikipedia - Pac-Man [online] Available at `https://en.wikipedia.org/wiki/Pac-Man` [Accessed 15 Nov. 2025]