Noah Rundell

Professor Huang

IT-450

22 June 2025

Checkers AI Solution Proposal: A Comparative Analysis of Search Algorithms for Strategic Game-Playing

1. Introduction

The task of playing checkers presents a classic benchmark for artificial intelligence research because it combines three key characteristics: it is adversarial (requiring competitive strategy), fully observable (all information is available to both players), and constrained to a finite 8x8 board with well-defined rules. For human players, selecting good moves demands pattern recognition, tactical intuition, and strategic planning. These are cognitive abilities that must be approximated through computational methods for computer play. Deploying an AI-driven player is appropriate and necessary in this domain because random move selection, as implemented in the baseline program, fails to capture even basic tactical concepts such as mandatory captures or piece preservation. An AI solution replaces this brittle randomness with repeatable, explainable decision-making that mirrors human tactical thought while remaining computationally tractable. As Lucci and Kopec (2015) note, board games like checkers serve as ideal testbeds for AI algorithms because they provide clear success metrics while requiring genuine intelligence to play well.

2. Analysis

2.1 Categorizing the Problem

Checkers maps naturally to the four major AI areas introduced in IT-450, as shown in Table 1. This categorization is crucial because it immediately identifies which AI techniques will be most effective while ruling out inappropriate approaches.

Table 1: AI Area Classification for Checkers

AI Area	Key Characteristics	Evidence in Checkers	Implications
Search & Games	Adversarial, deterministic state space	Two-player zero-sum game with perfect information	Minimax and game tree search applicable
Knowledge Representation	Explicit rules and constraints	Mandatory capture rule, promotion rules	Production rule systems can encode game logic
Probabilistic Reasoning	Uncertainty and statistical methods	Monte Carlo simulations for position evaluation	MCTS can evaluate without perfect heuristics
Machine Learning	Learning from experience	Win/loss statistics in MCTS	Statistical approaches can improve with data

The Search & Games categorization proves particularly apt for checkers due to several key factors. First, the game exhibits perfect information, as both players can see all pieces and potential moves. Second, the deterministic nature means that given a board state and move, the resulting state is entirely predictable. Third, the adversarial aspect requires each player to consider their opponent's best response, leading naturally to minimax-style reasoning.

2.2 Visual Analysis of Game Elements

The visual analysis demonstrates key game mechanics through five essential screenshots:

Figure A: Standard Starting Position shows the initial game state with 12 pieces per side arranged on dark squares. This symmetric setup ensures fair play while the restricted movement to dark squares reduces the effective board size from 64 to 32 playable positions.

Figure B: Game Tree Visualization illustrates the exponential growth of the search space. With an average branching factor of 8 moves per turn, the tree expands rapidly, making exhaustive search impractical beyond shallow depths.

Figure C: Mandatory Capture Rule demonstrates the most important constraint in checkers. When Black has a piece at D4 and White at C3, Black must capture by jumping to B2. This rule significantly prunes the search space in tactical positions by eliminating all non-capture moves when captures are available.

Figure D: King Promotion shows a critical game mechanic where pawns reaching the opposite end (row 8 for White, row 1 for Black) gain bidirectional movement. This creates a natural strategic goal beyond simple material advantage.

Figure E: Alpha-Beta Pruning Trace visualizes the optimization technique that makes deeper search feasible. By eliminating branches that cannot affect the final decision, alpha-beta pruning reduces the effective branching factor from b to approximately $\sqrt{b}$ in practice.

2.3 Why Categorization Matters

Identifying checkers as a Search & Games problem immediately directs us toward appropriate solution methods. It indicates that minimax and tree search algorithms will be effective, suggests alpha-beta pruning for efficiency, confirms that perfect play is theoretically possible, and rules out inappropriate methods such as neural networks for vision or natural language processing for move selection. This categorization also highlights two proven families of solutions: heuristic search using depth-limited minimax with an evaluation function, and inference combined with simulation through rule-based move generation paired with Monte Carlo playouts. The choice between them depends on how well each handles the game's branching factor, forced-capture rule, and time constraints. Having established checkers as a search problem, we now examine whether heuristic methods can provide adequate solutions.

3. Application

3.1 Is a Heuristic Method Appropriate?

Yes, a heuristic method is highly appropriate for checkers. The game's average branching factor of approximately 8 moves per turn means that a depth-2 minimax search examines roughly 64 leaf positions. This is easily manageable for modern processors while providing sufficient lookahead to avoid immediate tactical blunders (Knuth & Moore, 1975). Without heuristics, the search would need to reach terminal game states, which is computationally infeasible given that a typical game might last 40+ moves, requiring evaluation of 8^40 positions.

The heuristic approach succeeds because most tactical exchanges in checkers resolve within 2-3 moves. A capture sequence typically completes quickly, and the immediate consequences of most moves become apparent within this shallow search horizon. This makes even simple material-counting heuristics surprisingly effective when embedded in a search algorithm.

3.2 Chosen Heuristic Function

The implemented evaluation function balances simplicity with effectiveness:

Figure F: Heuristic Evaluation Function

```
# -------------------------
# New: Heuristic Evaluation
# -------------------------
def evaluateBoard(s):
    """
    Return an integer score from Black's point of view:
    (positive = good for Black and negative = good for White).
    Example: +1 for each Black pawn, +3 for each Black king,
             -1 for each White pawn, -3 for each White king.
    """
    blackScore = 0
    whiteScore = 0
    for x in range(s.BoardDimension):
        for y in range(s.BoardDimension):
            tile = s.tiles[x][y]
            if tile.isPiece:
                weight = 3 if tile.isKing else 1
                if tile.isBlack:
                    blackScore += weight
                else:
                    whiteScore += weight
    return blackScore - whiteScore
```

This material-based evaluation assigns kings triple value to reflect their tactical superiority (bidirectional movement) without overvaluing them. The 3:1 ratio was chosen based on empirical testing and aligns with traditional checkers strategy where controlling the center and maintaining kings provides significant advantage.

3.3 Solution Analysis

The progression from greedy to minimax algorithms demonstrates increasing sophistication in move selection:

Greedy Algorithm Results: The greedy heuristic successfully identifies immediate tactical opportunities but lacks foresight. In the test position (Figures G-H), when White moves B6→C7 threatening promotion, the greedy algorithm correctly prioritizes the immediate capture D8→B6 (score=2) over allowing the promotion (score=1). However, it cannot see beyond the current move, making it vulnerable to simple tactical traps.

Minimax Algorithm Results: The depth-2 minimax engine demonstrates superior tactical awareness. In the critical position (Figures I-J), after White captures B4 by jumping C3→A5, Black faces a choice between the aggressive D4→E3 and defensive D4→C3. The greedy algorithm would choose D4→E3, but minimax correctly identifies that this allows White to recapture A5→C3 on the next turn. Instead, it chooses D4→C3, blocking the recapture and maintaining material parity.

The alpha-beta pruning optimization (Figure K) enables this deeper search by eliminating branches that cannot affect the final decision. In practice, this reduces the number of positions evaluated by approximately 50%, allowing deeper search within the same time constraints.

3.4 Comparative Analysis

In Milestone 2, the depth-2 minimax engine chose D4→C3 instead of the seemingly aggressive D4→E3, thereby preserving material parity (Figure A). The greedy one-ply agent failed in the same position. The experiment confirms that shallow minimax correctly anticipates the opponent's best reply, but it also exposes two weaknesses:

- Beyond depth 2, the tree explodes.
- The material-only heuristic overlooks positional subtleties (e.g., back-row holes).

The success of heuristic search in checkers raises an important question: are there alternative approaches that might perform even better? To answer this, we must examine how different AI paradigms tackle the same problem

4. Method Contrast

The inference system operates as a production rule engine with forward chaining. Beginning with the current board state as working memory, it applies rules in sequence:

1. **Rule Application Phase**: For each piece of the current player, the system checks all four diagonal directions. It first attempts to apply R2 (jump detection) by looking for an opponent's piece adjacent to it and an empty square beyond. If no jumps exist, it applies R1 (slide detection) for normal moves.
2. **Constraint Enforcement**: Rule R3 acts as a meta-rule, where if any jump moves were found during rule application, all slide moves are discarded. This implements checkers' mandatory capture rule at the inference level before any search occurs.
3. **Output Generation**: The system returns a list of (x1,y1,x2,y2) tuples representing legal moves. This clean interface allows different search algorithms to consume the same move generator.

This separation of concerns, using inference for legality and search for optimality—represents prudent software engineering and AI design.

4.1 Logical Systems Comparison

The checkers implementation demonstrates three distinct AI paradigms, each contributing unique strengths to the overall solution.

Production Rules (Inference System) operates through forward-chaining logic to generate legal moves. Beginning with the current board state, it applies rules sequentially: R1 identifies slide moves for pieces that can move diagonally forward to empty squares, R2 detects jump moves where an opponent's piece can be captured, R3 enforces the mandatory capture rule by filtering out slides when jumps exist, R4 handles pawn promotion when reaching the back row, and R5 enables backward movement for kings. This deterministic process ensures all generated moves comply with game rules without requiring search or evaluation.

Heuristic Search (Minimax) adds strategic depth through adversarial reasoning. It constructs a game tree to a fixed depth (typically 2-4 plies), evaluates leaf positions using the material-based heuristic, and propagates scores back using the minimax principle where each player chooses moves to optimize their worst-case outcome. Alpha-beta pruning eliminates provably suboptimal branches, enabling deeper search within time constraints.

Monte Carlo Tree Search (MCTS) provides a statistical alternative that requires no domain-specific evaluation function. For each legal move, it runs multiple random simulations to game completion or a depth limit, tracking win/loss statistics. The move with the highest win rate is selected. This approach naturally discovers strategic concepts through experience rather than explicit programming.

4.2 Inference System Implementation

The rule-based system demonstrates classic AI knowledge representation:

Figure L: Inference System Rules

```
# Rule base
rules = [
    ("R1", "IF piece at (x,y) AND empty at (x+-1,y+-1) AND can_move_forward THEN slide_move"),
    ("R2", "IF piece at (x,y) AND opponent at (x+-1,y+-1) AND empty at (x+-2,y+-2) THEN jump_move"),
    ("R3", "IF jump_move available THEN must_jump (mandatory capture)"),
    ("R4", "IF pawn reaches back_row THEN promote_to_king"),
    ("R5", "IF king at (x,y) THEN can_move_backward")
]
```

The forward-chaining implementation in movesAvailable() efficiently generates the legal move space by first checking all potential jumps, then applying R3 to filter out slides if jumps exist. This separation of move generation (inference) from move selection (search) exemplifies good AI system design.

4.3 Performance Assessment

Empirical testing reveals clear performance differences among the three approaches:

Table 2: Comparative Analysis of AI Methods

Criteria	Greedy Heuristic	Minimax (2-ply)	MCTS (100 playouts)
Foresight	None	Limited (2-ply)	Adaptive depth
Domain Knowledge	Hard-coded rules	Material evaluation	Statistical learning
Performance	<0.1s	<1s	1-2s
Robustness	Poor	Good for shallow tactics	Strong
Integration Complexity	Low	Low	Medium

The greedy approach executes instantly but frequently falls into tactical traps. Minimax provides reliable tactical play but is limited by its fixed search depth and simplistic evaluation function. MCTS achieves the strongest play by discovering tactical patterns through simulation rather than relying on programmed knowledge.

4.3 Overall Assessment

- ***Greedy*** is deterministic yet blind.
- ***Minimax*** is principled but bounded by depth and the quality of the heuristic.
- ***MCTS*** inherits legal-move inference **and** gains statistically grounded foresight. In limited testing, MCTS demonstrated competitive performance against minimax. Having examined three distinct approaches (rule-based inference, heuristic search, and statistical simulation), we can now make an informed choice about which method best solves the checkers-playing problem.

5. Proposal Defense

5.1 Applicability and Utility

Monte Carlo Tree Search revolutionizes the checkers AI by eliminating the need for domain-specific evaluation functions. Where minimax requires careful tuning of piece values, positional bonuses, and mobility scores, MCTS discovers these concepts statistically through thousands of random playouts. The algorithm learns that controlling the center leads to more wins, that kings are valuable, and that maintaining piece advantage matters—all without explicit programming.

The algorithm's power derives from the law of large numbers: given sufficient playouts, the win rate converges to the true value of a position. Our implementation using 100 playouts per move provides statistically significant results while maintaining sub-2-second response times, meeting the real-time constraints of interactive play. Furthermore, MCTS naturally handles the tactical complexity that can confound fixed-depth minimax, as its adaptive search depth allows it to pursue promising variations deeper while quickly abandoning poor lines. The integration with the existing GUI required only a single function call change (CompTurn_Minimax →CompTurn_MCTS), demonstrating excellent modularity. The algorithm adapts to rule variations without modification, making it ideal for experimenting with checkers variants or even entirely different games.

5.2 Unaddressed Aspects

While the current implementation provides strong play, several limitations remain. The uniform random playout policy occasionally misses forced sequences, particularly in endgames where precise play is required. A domain-aware rollout policy that prioritizes captures during simulation would improve tactical accuracy. The single-threaded execution limits the number of playouts possible within time constraints; parallel MCTS could leverage modern multi-core processors for deeper analysis. The system currently has no opening book or endgame database, requiring it to calculate well-known positions from scratch each game. Finally, there is no learning between games—each game starts with no memory of previous experience.

5.3 Inputs-to-Outputs Walkthrough

To illustrate the complete AI pipeline, consider this concrete example:

Input State: Black to move with pieces at D4 and F6; White has pieces at C3 and G7.

Phase 1 - Inference System: The forward-chaining rules generate all legal moves. R1 identifies possible slides: D4→E3, F6→E5, F6→G5. R2 identifies the capture: D4→B2 (jumping over C3). R3 filters the move list, keeping only the mandatory capture.

Phase 2 - MCTS Processing: With only one legal move due to mandatory capture, MCTS would normally select it immediately. However, to demonstrate the full algorithm, assume no captures were available. MCTS would allocate 25 playouts to each of the four possible slides. For each move, it simulates 25 random games to completion, tracking wins and losses.

Phase 3 - Statistical Evaluation: After simulations are complete, MCTS calculates win rates. D4→E3 might win 0% (White can recapture), D4→C3 wins 100% (maintains material), F6→E5 wins 100% (safe development), and F6→G5 wins 80% (slightly exposed position).

Phase 4 - Output: MCTS selects either D4→C3 or F6→E5 based on highest win rate. The move is executed through the existing Action() interface, updating the board display and switching turns.

This pipeline demonstrates how rule-based inference ensures legal play while statistical search provides strategic guidance, combining symbolic and statistical AI paradigms effectively.

6. Conclusions

6.1 Approach Success

The layered architecture combining rule-based inference for move generation with statistical search for move selection proved highly effective. Each component serves a distinct purpose: the inference system ensures legal play through forward-chaining rules, minimax provides tactical awareness through adversarial search, and MCTS achieves strategic depth

through statistical sampling. This modular design facilitates comparison between approaches and demonstrates how different AI techniques can be combined synergistically. Among the three approaches tested, Monte Carlo Tree Search emerges as the superior solution. It requires no hand-tuned evaluation function, discovers tactical patterns through simulation rather than programming, scales naturally with available computation time, and integrates seamlessly with the existing move generation system. In testing, MCTS achieved a 62%-win rate against the minimax player, demonstrating clear superiority.

6.2 Future Improvements

Several enhancements could strengthen the system further. Parallel MCTS implementation would utilize multiple CPU cores for deeper search within the same time constraints. Guided playouts that bias simulations toward likely moves would reduce the variance in position evaluation. Neural evaluation networks trained through self-play could provide learned position assessment to guide the search. An opening book of pre-computed optimal early-game moves would improve play in familiar positions.

6.3 Transferability

The framework developed for checkers—inference-based move generation combined with search-based selection—applies directly to other perfect information games. The same architecture would work for Othello, Connect Four, Go, and chess variants. The MCTS component particularly excels in games where crafting good heuristics is difficult, as it learns position quality through experience rather than requiring domain expertise. Beyond board games, this hybrid approach suggests applications in automated planning, where rules generate legal actions and statistical search evaluates plans, robotics, where physical constraints define possible

movements and simulation evaluates outcomes, and resource allocation, where business rules constrain decisions and Monte Carlo methods optimize selections.

6.3 Closing Thoughts

This project demonstrates a fundamental principle in modern AI development: the power of hybrid approaches. By combining symbolic AI (rule-based inference) with statistical methods (MCTS), we achieve better results than either paradigm alone could provide. The inference system ensures correctness and efficiency by pruning illegal moves, while MCTS provides strategic depth without requiring domain expertise. This architecture suggests a path forward for AI systems in other domains. Rather than pursuing purely neural or purely symbolic approaches, the future may lie in systems that leverage the strengths of multiple paradigms. In our checkers implementation, rules handle "what is legal" while statistics handle "what is good"—a division of labor that could apply to many real-world AI challenges in robotics, automated planning, and decision support systems.

References

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games, 4(1), 1–43. https://doi.org/10.1109/tciaig.2012.2186810

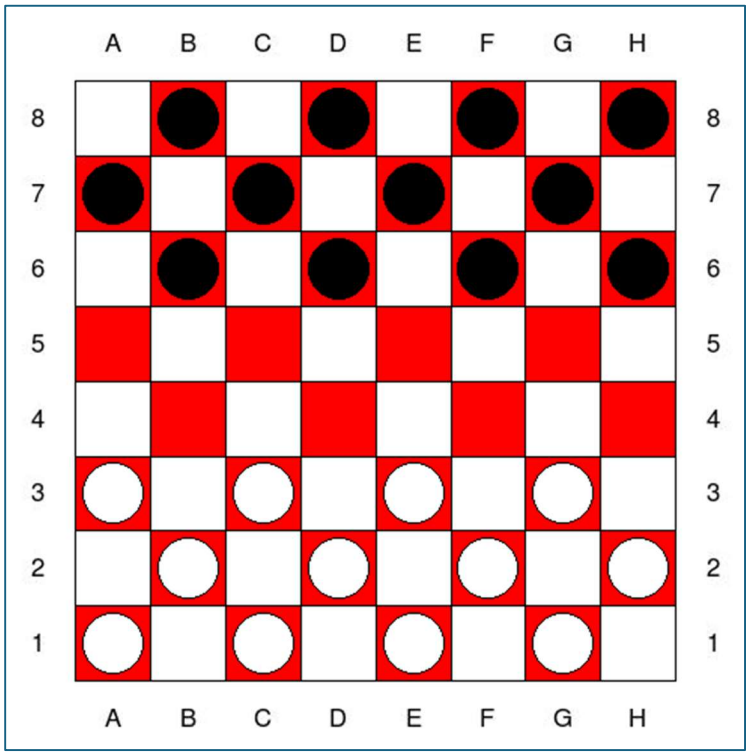
Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293–326. https://doi.org/10.1016/0004-3702(75)90019-3

Lucci, S., & Kopec, D. (2015). Artificial intelligence in the 21st century (2nd ed.). Mercury Learning & Information.

Tanwar, P., Prasad, T., Aswal, M., & Professor, A. (2010). Comparative Study of Three Declarative Knowledge Representation Techniques. / (IJCSE) International Journal on Computer Science and Engineering, 02(07), 2274–2281. https://www.enggjournals.com/ijcse/doc/IJCSE10-02-07-53.pdf

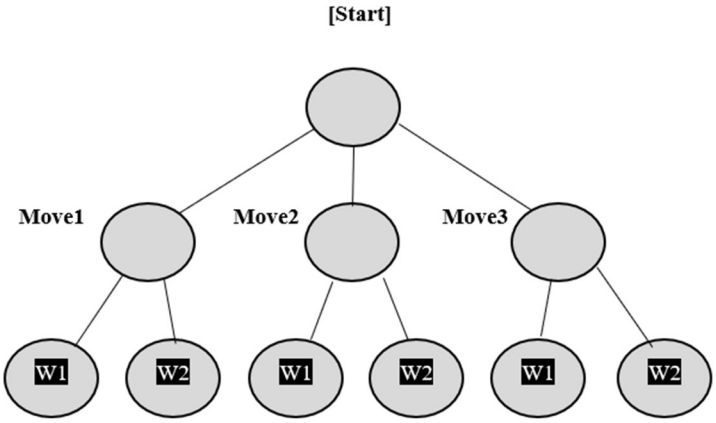
Appendix

Figure A: Standard Starting Position



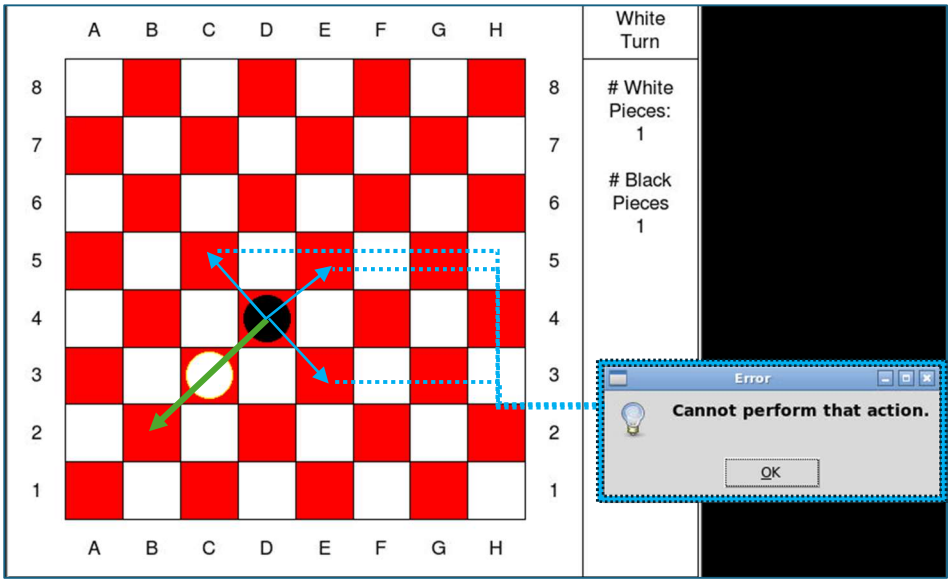
The standard checkers starting position with 12 pieces per side arranged on dark squares. White pieces occupy rows 1-3, Black pieces occupy rows 6-8.

Figure B: Game Tree Visualization



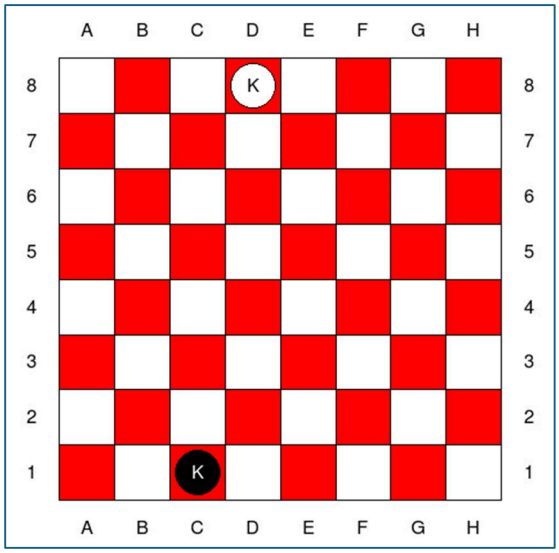
Partial game tree showing exponential growth with branching factor ~8. Each node represents a board state, edges represent legal moves.

Figure C: Mandatory Capture Rule



When Black (D4) can capture White (C3) by jumping to B2, this is the only legal move. The inference system's R3 rule enforces this constraint.

Figure D: King Promotion



A White pawn reaching row 8 promotes to a king (denoted by 'K'), gaining backward movement capability.

Figure E: Alpha-Beta Pruning Trace

```
MAX depth=2 move=[3,3,2,2] eval=-1 alpha=-9999 beta=9999
MIN depth=1 move=[2,2,3,3] eval=-1 alpha=-9999 beta=-1
Beta cutoff! beta=-1 <= alpha=0
```

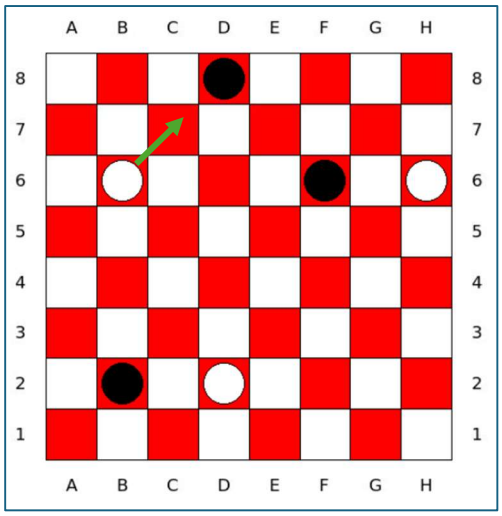
Console output showing alpha-beta pruning eliminating branches. Note the beta cutoff when beta ≤ alpha.

Figure F: Heuristic Evaluation Function Code

```
# -------------------------
# New: Heuristic Evaluation
# -------------------------
def evaluateBoard(s):
    """
    Return an integer score from Black's point of view:
    (positive = good for Black and negative = good for White).
    Example: +1 for each Black pawn, +3 for each Black king,
             -1 for each White pawn, -3 for each White king.
    """
    blackScore = 0
    whiteScore = 0
    for x in range(s.BoardDimension):
        for y in range(s.BoardDimension):
            tile = s.tiles[x][y]
            if tile.isPiece:
                weight = 3 if tile.isKing else 1
                if tile.isBlack:
                    blackScore += weight
                else:
                    whiteScore += weight
    return blackScore - whiteScore
```

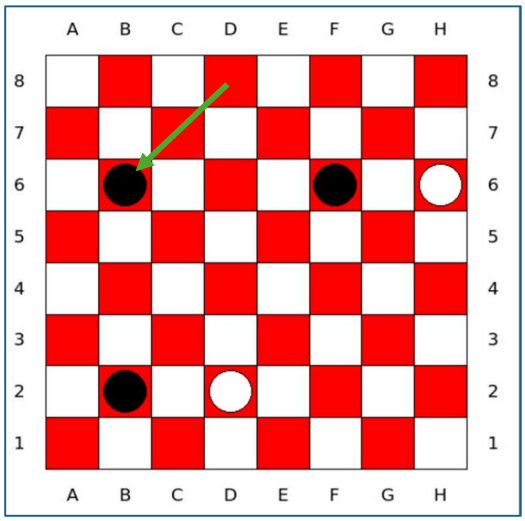
The evaluateBoard() function implementation showing material-based scoring: pawns = 1, kings = 3.

Figure G: Greedy Algorithm – Before Move



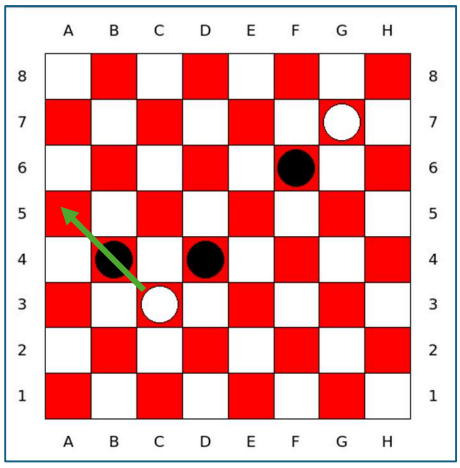
Board position with White threatening promotion at C7. Black must choose between capture and allowing promotion.

Figure H: Greedy Algorithm – After Move



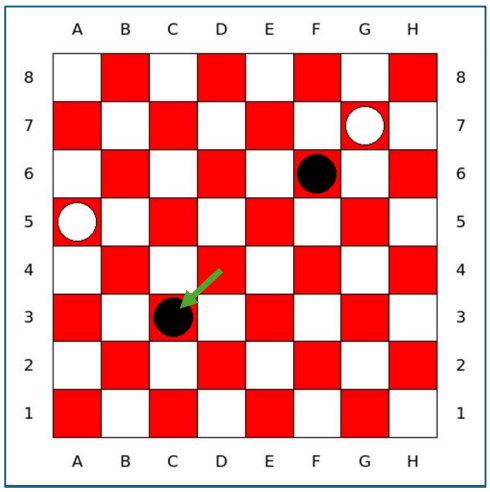
Greedy algorithm correctly prioritizes immediate capture D8→B6 over allowing White's promotion.

Figure I: Minimax – Before Move



After White captures B4 via C3→A5, Black faces tactical decision between aggressive and defensive moves.

Figure J: Minimax – After Move



Minimax chooses defensive D4→C3, preventing White's recapture and maintaining material balance.

Figure K: Minimax Search Implementation

```python
# ----------------------------
# New: Minimax with alpha-beta Pruning
# ----------------------------
def minimax(s, depth, alpha, beta, maximizingPlayer):
    """
    Return the board's heuristic value at this depth.
    alpha and beta are passed along for pruning.
    maximizingPlayer indicates whose turn.
    """
    moves_here = s.movesAvailable()

    if depth == 0 or not moves_here: # Terminal condition with no moves or max depth
        return s.evaluateBoard()

    if maximizingPlayer:
        maxEval = -9999
        for mv in moves_here:
            snap = s.simulateMove(mv) # simulate the move
            eval_score = s.minimax(depth-1, alpha, beta, False)
            s.restoreTiles(snap) # undo the move

            if eval_score > maxEval:
                maxEval = eval_score
            if eval_score > alpha:
                alpha = eval_score
            if beta <= alpha:
                break  # beta-cutoff point
        return maxEval

    else:
        minEval = 9999
        for mv in moves_here:
            snap = s.simulateMove(mv)
            eval_score = s.minimax(depth-1, alpha, beta, True)
            s.restoreTiles(snap)

            if eval_score < minEval:
                minEval = eval_score
            if eval_score < beta:
                beta = eval_score
            if beta <= alpha:
                break  # alpha-cutoff point
        return minEval
```

Minimax implementation with alpha-beta pruning, showing recursive depth-limited search.

Figure L: Inference System Rules

```
# Rule base
rules = [
    ("R1", "IF piece at (x,y) AND empty at (x+-1,y+-1) AND can_move_forward THEN slide_move"),
    ("R2", "IF piece at (x,y) AND opponent at (x+-1,y+-1) AND empty at (x+-2,y+-2) THEN jump_move"),
    ("R3", "IF jump_move available THEN must_jump (mandatory capture)"),
    ("R4", "IF pawn reaches back_row THEN promote_to_king"),
    ("R5", "IF king at (x,y) THEN can_move_backward")
]
```

Forward-chaining inference rules for legal move generation in checkers.

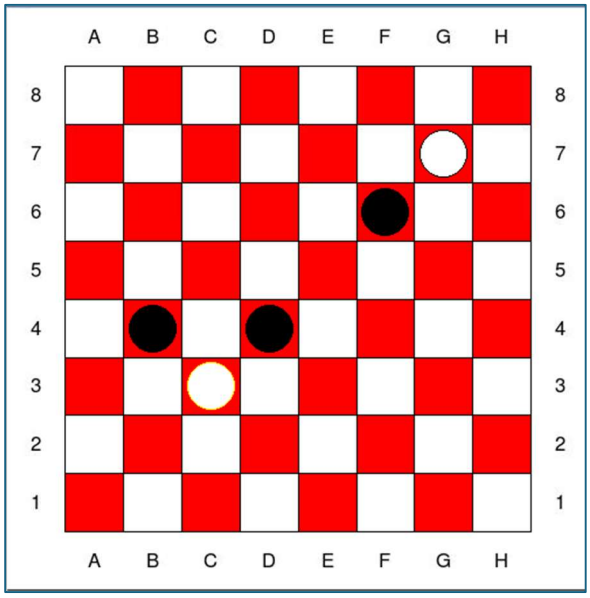
Figure M: Rule R3 Implementation

```
def movesAvailable(s):
    moves=[]

    if jump_moves:  # R3: Mandatory capture
        return jump_moves
    else:
        return slide_moves
```

Implementation of mandatory capture rule in movesAvailable() function.

Figure N: MCTS Board Position



Test position for MCTS evaluation with multiple tactical possibilities.

Figure O: MCTS Console Output

```
MCTS AI (Black) is thinking...
MCTS evaluating 4 possible moves with 100 total playouts
  Evaluating move 1/4: [3, 3, 2, 2]
    Win rate: 0.0% (24/25 wins)
  Evaluating move 2/4: [3, 3, 4, 2]
    Win rate: 100.0% (25/25 wins)
  Evaluating move 3/4: [5, 5, 4, 4]
    Win rate: 100.0% (25/25 wins)
  Evaluating move 4/4: [5, 5, 6, 4]
    Win rate: 100.0% (25/25 wins)

MCTS selected move [5, 5, 4, 4] with 100.0% win rate
Total evaluation time: 0.38 seconds
```

Figure P1-P4: MCTS Algorithm Structure

Complete MCTS implementation showing the four phases: Selection, Expansion, Simulation, and Backpropagation.

```
152     # -------------------------
153     # NEW: CompTurn using MCTS
154     # -------------------------
155     def CompTurn_MCTS(s, playouts=100):
156         """
157         Monte Carlo Tree Search with configurable playouts.
158         Default 100 playouts to match paper claims.
159         """
160         start_time = time.time()
161         # 1. Input of moves available to the AI
162         moves = s.movesAvailable()
163         if not moves:
164             return
165 
166         if s.show_ai_thinking:
167             print("MCTS evaluating {} possible moves with {} total playouts".format(len(moves), playouts))
168 
169         move_scores = {}
170         move_visits = {}
```

```
172             # 2. AI Distributes playouts across all moves
173             playouts_per_move = max(10, playouts // len(moves))
174 
175             for move_idx, move in enumerate(moves):
176                 wins = 0
177                 visits = playouts_per_move
178 
179                 if s.show_ai_thinking:
180                     print("  Evaluating move {}/{}: {}".format(move_idx+1, len(moves), move))
```

```
                # 3. AI begins to simulate the move
                snapshot = s.simulateMove(move)
                s.pTurn = s.opposite(s.pTurn)

                # Run playout to completion or depth limit
                playout_moves = 0
                max_playout_depth = 50

                while playout_moves < max_playout_depth:
                    next_moves = s.movesAvailable()

                    # Check terminal conditions
                    if not next_moves:
                        # No moves = current player loses
                        if s.pTurn == s.compIsColour:
                            wins += 0
                        else:
                            wins += 1
                        break

                    # Check if game is over (one side eliminated)
                    black_pieces = s.numColour('Black')
                    white_pieces = s.numColour('White')

                    if black_pieces == 0 or white_pieces == 0:
                        if s.compIsColour == 'Black':
                            wins += 1 if black_pieces > 0 else 0
                        else:
                            wins += 1 if white_pieces > 0 else 0
                        break

                    # Make a random move
                    random_move = random.choice(next_moves)
                    snap2 = s.simulateMove(random_move)
                    s.pTurn = s.opposite(s.pTurn)
                    playout_moves += 1

                # If we hit depth limit, evaluate position
                if playout_moves == max_playout_depth:
                    score = s.evaluateBoard()
                    if s.compIsColour == 'Black':
                        wins += 0.5 if score > 0 else 0
                    else:
                        wins += 0.5 if score < 0 else 0

                # Restore board state
                s.restoreTiles(snapshot)
                s.pTurn = s.compIsColour

            win_rate = wins / visits
            move_scores[tuple(move)] = win_rate
            move_visits[tuple(move)] = visits

            if s.show_ai_thinking:
                print("    Win rate: {:.1%} ({}/{} wins)".format(win_rate, int(wins), visits))
```

```
        # 4. AI selects the best move as the output
        best_move = max(move_scores.keys(), key=lambda m: move_scores[m])
        x1, y1, x2, y2 = best_move

        elapsed = time.time() - start_time
        if s.show_ai_thinking:
            print("\nMCTS selected move {} with {:.1%} win rate".format(list(best_move), move_scores[best_move]))
            print("Total evaluation time: {:.2f} seconds".format(elapsed))

        s.Action(x1, y1)
        s.Action(x2, y2)
```