Departmental Chess: A Chess Game with search algorithms as a game mechanic

Neelay Kamat IMS23174

February 16, 2026

Abstract

This report details the design and implementation of a unique chess variant that integrates classical computer science search algorithms as playable game mechanics. Unlike traditional chess, where movement is strictly defined by piece geometry, this project introduces ”Algorithm Cards” (BFS, DFS, A*) that allow players to override standard rules. These cards enable pieces to traverse the board using pathfinding logic to capture targets or reposition strategically. The report covers the architectural design of the Grid World, the specific implementation of Breadth-First Search (BFS) and Depth-First Search (DFS) as asynchronous visualizers, and the integration of the A* algorithm for optimal pathfinding. It also discusses current limitations, such as the ”King Protection” filter, game loop synchronization, and future development plans for heuristic-driven gameplay.

Contents

1 Introduction

The intersection of Game Theory and Artificial Intelligence often focuses on agents playing games. However, this project inverts that relationship by making the AI algorithms themselves the mechanics of the game.

The environment is a discrete 8x8 Grid World (a standard Chess board) populated with distinct entities (Pawns, Knights, Rooks, etc.). The core objective is to defeat the opponent's King. However, players are equipped with a "Hand" of algorithm cards. When played, these cards trigger a search algorithm starting from a selected piece, visualizing the search process in real-time, and executing a move based on the first valid target found.

This approach transforms the abstract concepts of graph traversal into tangible gameplay elements, providing a visual and interactive method for understanding how different search strategies explore a state space.

2 System Architecture and Game Logic

The project is built on a modular architecture separating the Game Loop, Board State, and UI Interaction.

2.1 The Round Manager

The game operates on a strictly phased round system managed by a central `RoundManager`.

1. **Card Selection Phase:** Players draft their "Algorithm Hand" from a deck.
2. **Reveal Phase:** Hands are revealed, and Initiative (AP) is calculated.
3. **Play Phase:** Players take turns spending Action Points (AP) to move pieces normally or cast Algorithm Cards.

2.2 The Board and Representation

The board is represented as a dictionary of coordinates mapping to objects (chess pieces). This sparse representation allows for efficient lookup $O(1)$ and easy validity checks.

- **Grid:** An 8×8 coordinate system.
- **State:** Each square holds information about occupancy, piece type, and ownership (White/Black).
- **Visualization:** The board handles the instantiation of "Ghost Pieces"—semi-transparent visual markers that appear during an algorithm's execution to visualize the search tree.

2.3 Card System Integration

Each card carries data regarding its Algorithm Type (BFS, DFS, A*), AP Cost, and AP gain (only valid for point cards). When a card is played:

1. The board enters a `Targeting` state.

2. The player selects a source piece.
3. The specific algorithm runs a loop, visually scanning the board.
4. Upon completion, the card is consumed (removed from hand and UI), and the move is executed, where the piece takes the first piece visited or goes to the last blank space it visited if the proximity had no capturable pieces.
5. The algorithm also makes sure it does not visit the same square again.

3 Search Algorithms: Implementation and Mechanics

3.1 Uninformed Search Strategies

Uninformed strategies traverse the board without heuristic knowledge of the goal's location.

3.1.1 Breadth-First Search (BFS)

Concept: BFS explores the neighbor nodes first, effectively scanning the board in expanding concentric circles (or diamond shapes in Manhattan geometry). It guarantees finding the shallowest goal.

Game Implementation:

- **Visuals:** The search expands uniformly from the source piece.
- **Behavior:** The piece moves to the *closest* enemy unit. It is ideal for short-range tactical strikes where the nearest threat must be neutralized.
- **Constraint:** Max Depth = 2. This limits the BFS to a local area scan, balancing its power.

3.1.2 Depth-First Search (DFS)

Concept: DFS explores as deep as possible along each branch before backtracking. It is non-optimal for path length but can find distant targets quickly.

Game Implementation:

- **Visuals:** The search shoots out in long tendrils across the board.
- **Behavior:** The piece tends to bypass nearby enemies to strike at targets deep in enemy lines. It introduces chaos and unpredictability.
- **Randomization:** The order of neighbor expansion is shuffled `moves.shuffle()` to prevent deterministic behavior, making each DFS card play feel unique.
- **Constraint:** Max Depth = 4. Allows for cross-board snipes.

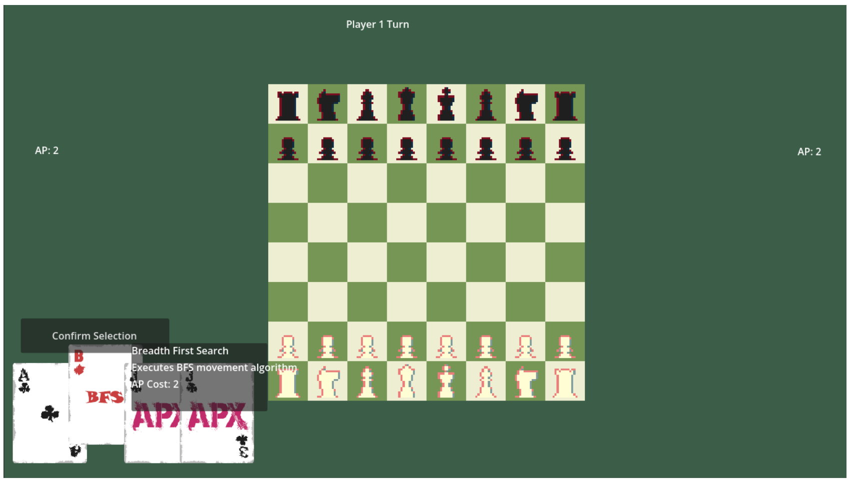

Figure 1: White choosing which cards to play onto active hand.

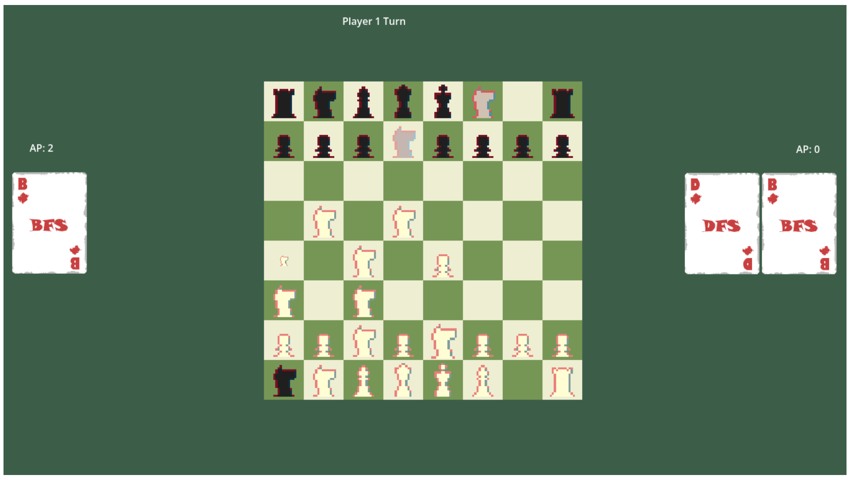

Figure 2: White Knight Executing a successful BFS move.

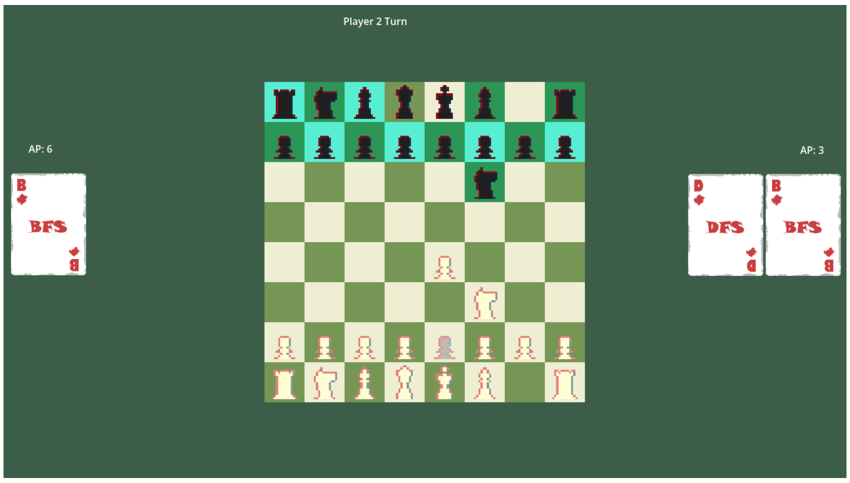

Figure 3: Black Chooosing a valid piece to enhance.

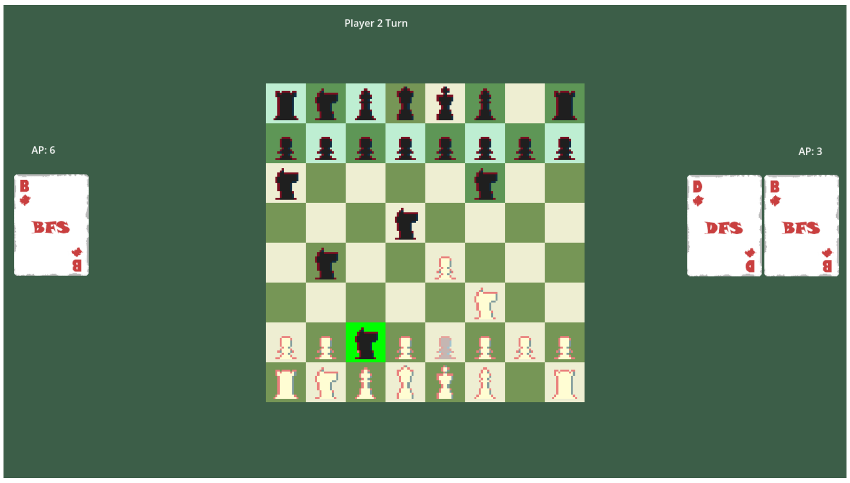

Figure 4: Black Knight Executing a successful DFS move.

3.2 Informed Search Strategies

3.2.1 A* Search (Planned Integration)

Concept: A* uses a heuristic function $f(n) = g(n)+h(n)$ to guide the search, where $g(n)$ is the cost so far and $h(n)$ is the estimated distance to the goal (Manhattan Distance).

Game Mechanics:

- Unlike BFS/DFS which search for *any* target, A* will be user-targeted. The player selects a specific destination square.
- The algorithm calculates the optimal path avoiding obstacles.
- **Cost:** High AP cost due to its precision and optimality.

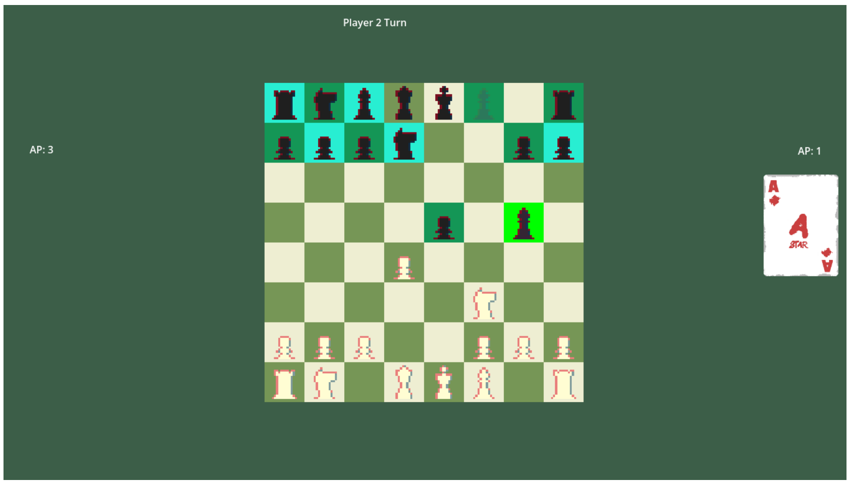

Figure 5: Black successfully executing A* algorithm to capture a piece

4 Technical Implementation Details

4.1 Asynchronous Visualization

To ensure the player understands the algorithm, the search does not happen instantly. It is implemented using Coroutines (via `await` and timers).

```
func visualize_chess_bfs(start_coord):
    queue = [start_coord]
    while queue:
        current = queue.pop_front()

        spawn_persistent_ghost(current, COLOR_RED)
        await timer(0.2)

        if is_enemy(current):
            if current == KING: continue
            return current

        neighbors = get_valid_moves(current)
        for n in neighbors:
            queue.push(n)
```

Listing 1: BFS Coroutine Logic

4.2 King Protection Logic

A critical game balance decision was the "King Protection" filter.

- **Problem:** Algorithms would often find the King immediately if he was within range, leading to anti-climactic game endings.
- **Solution:** The search loop explicitly checks ‘if target.type == KING: continue‘. This treats the King as an obstacle rather than a valid capture node. The algorithm flows around the King, simulating a "guard" mechanic where the King cannot be targeted by automated systems.

4.3 UI and State Management

The UI is tightly coupled with the game state to prevent illegal actions.

- **Phase Locking:** The ‘HandArea‘ is hidden during the ‘PLAY‘ phase to prevent players from playing multiple cards simultaneously.
- **Card Consumption:** A card is only consumed after the algorithm successfully completes a move.

5 Bugs, Limitations, and Challenges

5.1 Current Limitations

- **Turn Synchronization:** Initially, players could make moves while an algorithm was still visualizing, leading to race conditions. This was solved by adding a global locked condition and centralizing the decision (earlier decided by board instead of **The Round Manager**

- **Memory Management:** Ghost pieces must be manually tracked and cleared. Failing to await the cleanup coroutine resulted in "stuck" ghosts overlaying the board.

6 Future Work

6.1 New Algorithms

- **Greedy Best-First Search:** A variant that moves purely based on heuristic proximity to the enemy King, regardless of obstacles (likely getting stuck in traps).
- **Bi-Directional Search:** Two search trees starting from the player's piece and the enemy piece simultaneously, meeting in the middle.

6.2 Enemy AI

- Adding an enemy AI to make informed decisions to checkmate the opponent while considering the card game mechanic

6.3 Heuristic Improvements

- Implementing different heuristics for A* such as "Safety Heuristic" (avoid squares attacked by enemy) vs "Aggression Heuristic" (move toward high-value pieces).

6.4 Visual Polish

- **Path Highlighting:** Once a target is found, draw a distinct line connecting the start and end nodes to show the final path taken.
- **Camera Shake:** Add screen shake effects when an algorithm captures a piece to emphasize the impact.

7 Conclusion

By treating search algorithms as abilities, the game provides an intuitive understanding of how BFS expands uniformly versus how DFS probes deeply. The implementation challenges—specifically regarding asynchronous state management and game balance—highlight the complexity of adapting pure algorithms into a user-friendly interactive experience. The resulting system is a robust framework for experimenting with further AI concepts in a competitive environment.

Additional Info

- Game Engine: Godot Engine

- Programmed In: GDScript
- Current Game Size: 3.3 Mb
- Estimate Lines of Code: Aprox. 1600 lines
- Learning Resources: Godot Documentation, Youtube (Godot with me (for inital chess setup), Clear Code (godot shaders setup))
- Assets Used: Chess Pieces (Godot with me), Intial Cards setup (Simple-Cards by twdoor)