UNIVERSITY OF
BIRMINGHAM

An Interactive Algorithms Platfrom: From Maze Traversal to Minimax and NNUE Engines

Yan Chervonyi
Student ID: 2535602

A dissertation submitted to the University of Birmingham
for the degree of BSc Computer Science

Project Supervisor: Dr Sagnik Mukhopadhyay
Word Count: 10504
Date: 16th April 2025

Contents

Abstract

Artificial intelligence has influenced chess ever since early computational engines showcased the power of combinatorial search. Recent breakthroughs, such as NNUE (Efficiently Updatable Neural Network), further expand these capabilities, enabling engines to discover strategies that go beyond traditional human understanding. Rather than simply enhancing an existing Python engine like Sunfish, this project also provides a comprehensive educational platform that introduces users to NNUE in a clear, approachable way before they dive into official documentation. In addition, it offers visualizations of fundamental algorithms such as pathfinding, minimax, and binary search trees (BST), along with a custom-built chess engine for hands-on exploration. Step-by-step animations and explanations make complex concepts more accessible, while the interactive chess environment helps learners connect theory to real-world applications. Through these engaging, real-time demonstrations, the platform shows how neural-network evaluations can combine both learning and strategic decision-making.

1 Introduction

1.1 Literature Review

1.1.1 Overview of Chess Engines and Evaluation Approaches

Early chess programs relied primarily on brute-force search, examining every possible move and position, combined with basic heuristics for evaluation. While conceptually straightforward, brute-force methods scale poorly in a complex game like chess, leading to exponential growth in the search tree. To address this, techniques such as alpha-beta pruning, introduced by Knuth and Moore [1], significantly reduced the number of positions requiring evaluation. Iterative deepening, explained by Korf [2], further enhanced efficiency by incrementally deepening the search while reusing previously gathered moves. Classical chess engines, such as IBM's Deep Blue [3], heavily relied on these strategies.

Modern engines build on these foundational ideas, combining clever search algorithms with sophisticated evaluation functions. The following subsections outline both the principal *engine search approaches* and the *evaluation techniques* that have shaped classical and contemporary chess programs.

1.1.2 Engine Search Approaches

- **Brute-force Search**: Checks every possible move, often without skipping any branch. Although it guarantees finding the optimal move given enough time, it

becomes prohibitively slow for deeper search depths [4].

- **Minimax Algorithm**: Provides a fundamental framework for two-player games, where each player aims to maximize their own advantage while minimizing the opponent's. Originally formalized by von Neumann and Morgenstern [5], it systematically evaluates future positions assuming perfect play from both sides.
- **Alpha-Beta Pruning**: An enhancement to minimax that prunes branches guaranteed not to affect the final decision, significantly cutting down the search space [1]. This pruning allows the engine to investigate deeper lines within the same time budget.
- **Iterative Deepening Search**: Gradually expands the search depth in discrete layers [2]. It reuses information (like best moves) discovered at shallower depths to prioritize deeper exploration, balancing speed and thoroughness in time-constrained environments.
- **Monte Carlo Tree Search (MCTS)**: Rather than exhaustively exploring each branch, MCTS employs statistical sampling to simulate potential move sequences [6]. Modern systems like AlphaZero incorporate a variant of MCTS, guided by neural-network-based evaluations.

These search approaches collectively form the "decision engine," determining which moves to explore and in what order. Top chess engines still use iterative deepening and alpha-beta pruning, which let them examine far fewer positions yet remain just as strong.

1.1.3 Evaluation Approaches

A major part of playing chess for AI is the *problem of proper board evaluation*. Throughout the history of chess engines, multiple approaches have been introduced to translate a raw position into a numerical score that a search algorithm (e.g., alpha-beta) can use to compare moves. Even with powerful search techniques (Section 1.1.2), an engine's ultimate performance hinges on how effectively it judges each board. This subsection outlines the key evaluation techniques developed over time.

- **Basic Heuristics and Piece-Square Tables**: Early engines primarily measured material balance (assigning point values to each piece type). *Piece-square tables* added simple positional considerations by awarding bonuses or penalties for placing a piece on a certain square. For instance, a knight might get extra points near the center, reflecting greater mobility. Although computationally cheap and easier to hand-tune, these heuristics may overlook deeper positional or strategic aspects, such as pawn chains or long-term king safety [4].
- **Transposition Tables**: Over a typical chess game, identical positions can be

achieved through different move sequences. *Transposition tables* store the evaluation results of these positions once computed, so the engine can recall them instantly instead of recalculating. This technique, discussed by Slate and Atkin [7], saves an enormous amount of time, particularly in the midgame when the branching factors are high. By caching previously evaluated boards, engines can redirect precious computation toward unexplored or more critical lines.

- **Handcrafted Positional Features**: As engines evolved, developers introduced additional factors capturing subtler positional characteristics. For example, king safety checks whether the king has protective pawns or easy lines of attack, pawn structure highlights advanced or isolated pawns, and control of open files rewards rooks on half-open or open files. Each factor is generally weighted, and developers iteratively refine these weights through testing. Although more powerful than basic heuristics, they still rely on human insight to identify which positional traits matter most [8].
- **Neural Network Evaluations**: Modern engines have begun incorporating *learned* evaluations. Instead of relying solely on programmer-crafted rules, a neural network trains on large datasets (or self-play) to automatically discover critical patterns. NNUE (Efficiently Updatable Neural Network) is a notable example, using incremental updates to evaluate small board changes without recomputing the entire network output [9]. This design aligns well with alpha-beta search, allowing the engine to preserve speed while benefiting from position assessment. Engines like Stockfish have demonstrated significant performance gains by combining classical search with NNUE evaluation, frequently outperforming engines using purely hand-tuned heuristics.

1.1.4 Existing Approaches

While much of the cutting-edge research and development in chess engines is driven by projects written in C++ (e.g., the open-source Stockfish engine [10]), there also exist purely Python-based engines that have gained recognition for their simplicity and educational value. One notable example is *Sunfish* [11], which uses a **120-character board** with **invisible padding** to detect out-of-bounds moves easily, along with a **piece-square table** (PST) for basic evaluation and an *alpha-beta* search approach. Its small codebase makes it accessible to beginners who want to learn how a chess engine works without the complexity of a large project. Another Python resource is the *python-chess* library [12], which provides robust functionality for **board representation**, **legal move generation** using pure Python, and a *very basic* position evaluation that primarily considers material balance. It can also interface with external UCI engines like Stockfish for stronger analysis when needed.

Although these Python engines are highly accessible for experimentation and rapid prototyping, they often lack advanced evaluation mechanisms and low-level optimizations found in more established C++ engines. Notably, Stockfish has introduced an *NNUE* (Efficiently Updatable Neural Network) evaluation [13] that significantly enhances position scoring by integrating neural-network-based insights into a traditional alpha-beta search framework.

1.1.5 NNUE in Traditional Engines

The *Efficiently Updatable Neural Network (NNUE)* approach, originating in the Shogi community, enables a fast, incremental way to update neural evaluations [9]. Unlike fully neural engines (e.g. AlphaZero, Leela Chess Zero), NNUE is designed for standard CPUs, making it practical for engines such as *Stockfish* when combined with alpha-beta search [14].

Matrix Multiplication in a Linear Layer A key step in NNUE is a **linear** (fully-connected) layer, defined by:

$$\text{output} = (W \times \text{input}) + b,$$

where W and b are the weight and bias parameters, and input is a feature vector encoding board information. Figure 1 illustrates how this multiplication is split into colored "bands," each chunk of the input multiplied by a corresponding slice of the weights. Because most moves alter only a small subset of features (e.g. one piece moving), engines can update a few relevant indices rather than recomputing all multiplications each move.

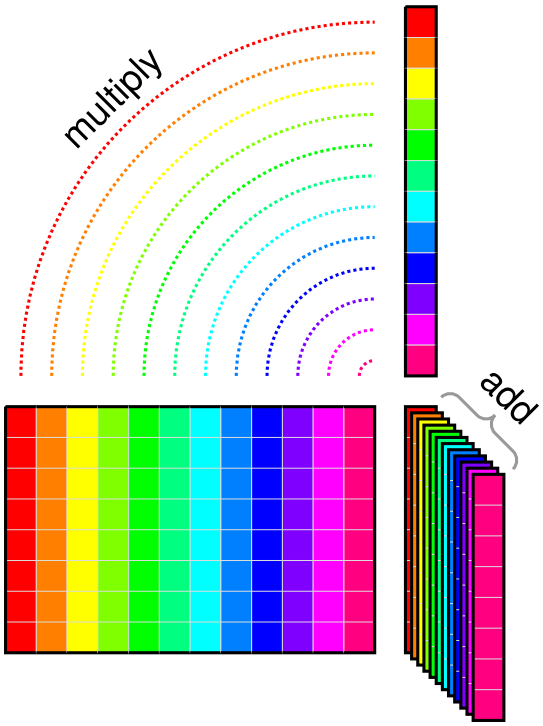


Figure 1: A conceptual look at the linear layer. Each coloured band ("rainbow") handles a slice of the input multiplied by the matching slice of the weights [15].

One can further simplify the multiplication $\mathbf{Ax}$ by noting that *if* $\mathbf{x}[i]$ *is zero, we skip column* i *entirely.* In other words, $\mathbf{Ax}$ becomes:

$$\mathbf{Ax} \;=\; \sum_{i\,:\,\mathbf{x}[i]\neq 0} \big[\text{column } i \text{ of } \mathbf{A}\big] \cdot \mathbf{x}[i],$$

which is extremely useful for a sparse $\mathbf{x}$. Although W (or $\mathbf{A}$) may have tens of thousands of columns, engines only process those columns for features that are active in the current position.

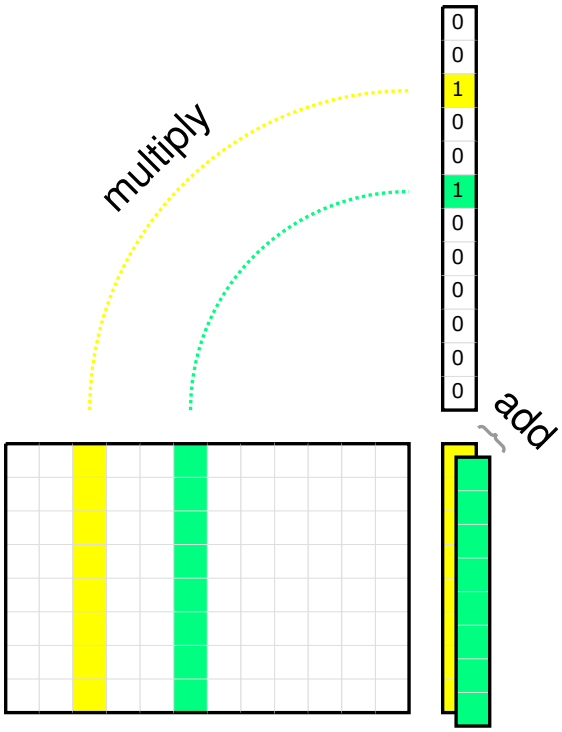


Figure 2: Skipping columns of **A** for zero-valued inputs. Only the columns corresponding to nonzero features in **x** are processed [15].

Clipped ReLU and Partial Updates After multiplying by W, many NNUE implementations apply a *clipped ReLU* activation to introduce non-linearity. In integer-based engines, this often includes a *shift and clamp* so that intermediate sums remain within valid numeric ranges (e.g., 8-bit or 16-bit accumulators). By selectively updating only the neurons corresponding to changed squares, the engine avoids recalculating every feature on every move. In practice, two *accumulators* store hidden-layer values for each side's perspective (e.g., White vs. Black). When a piece moves from one square to another, the old location's contribution is subtracted, the new location's contribution is added, and the clipped/shifted ReLU ensures the layer remains both bounded and non-linear.

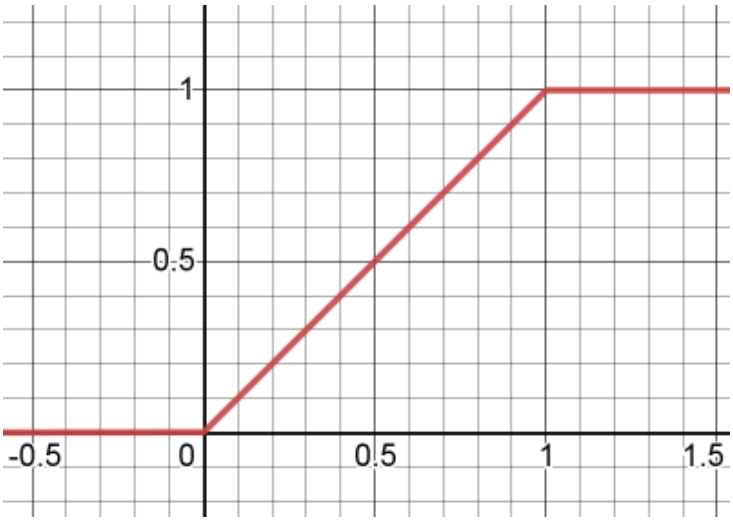


Figure 3: After the linear layer, a clipped or shifted activation keeps integer operations within limits, with only changed squares prompting recalculations [15].

Overall Pipeline Figure 4 shows an incremental NNUE flow. Each green node represents a piece-square feature, most of which remain constant on a given move. Hidden layers (yellow) combine weighted sums and clipped ReLU activations, while the final red output node yields a position evaluation. By performing small, targeted updates instead of a full forward pass every move, alpha-beta engines maintain high speed despite a neural-based evaluation.

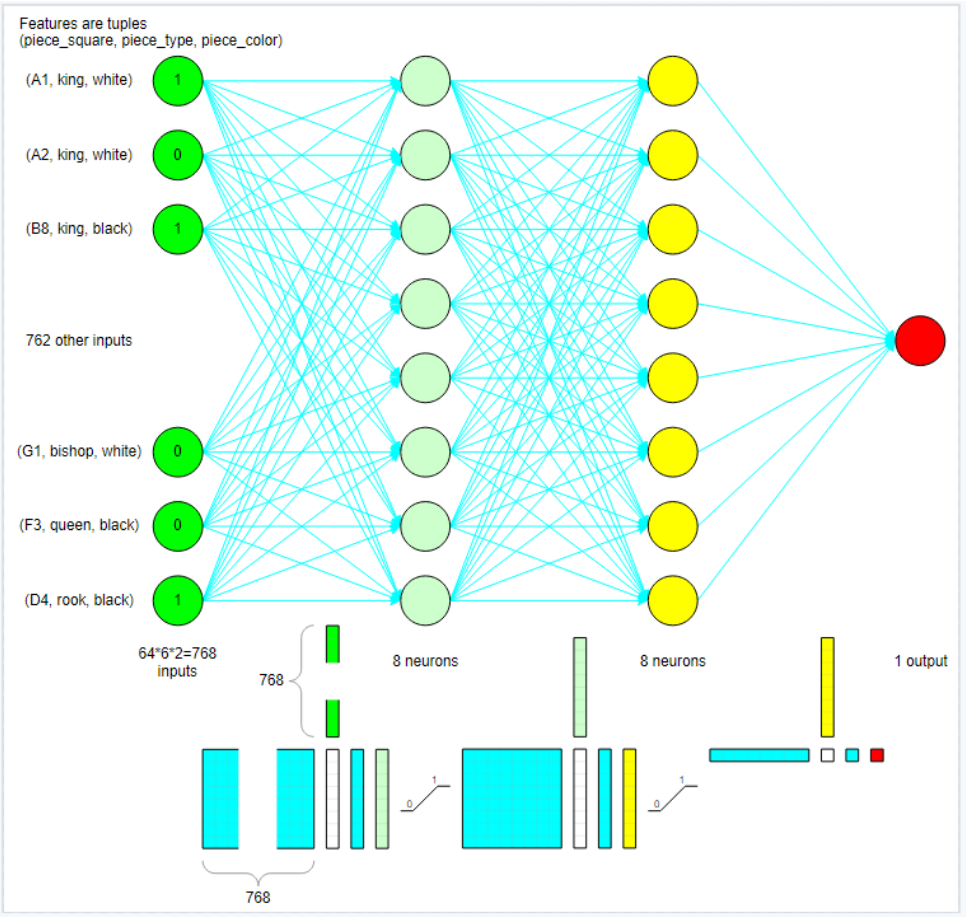


Figure 4: A simplified NNUE pipeline: piece-square inputs (green) flow through fully-connected hidden layers (yellow), yielding a single evaluation output (red) [15].

HalfKP: Basic Piece-Square Mapping One of the earliest NNUE variants, `HalfKP`, focuses on *piece-square pairs* relative to each king's position. It maintains two "perspectives": one for the side to move and another for the opposing side. Each perspective can hold tens of thousands of binary features (for example, `40960`), which feed into a `40960`→`256` linear layer (the `256` here can be any chosen number of neurons), followed by a clipped ReLU activation.

Feature Index Computation. In our implementation, each feature index is computed by:

$$p\_idx = (\text{piece\_type}) \times 2 \; + \; \text{piece\_color},$$

$$\text{halfkp\_idx} = \text{piece\_square} \; + \; \big(p\_idx + (\text{king\_square} \times 10)\big) \times 64.$$

The special case is when the king itself moves. Since king_square appears in many feature indices, *all* those features change and we must perform an accumulator refresh. Although more costly for king moves, this remains a rare event and keeps the overall number of updates per evaluation low.

Partial Updates for Efficiency. Although only a few input features change from move

to move, we can exploit this sparsity by storing partial sums in the engine's state (the "accumulators"). Recall that a linear layer effectively adds columns of a weight matrix based on which features are active. Instead of recomputing the entire first hidden layer, we do:

- **Removed feature** $(1 \to 0)$: *subtract* that feature's column from the accumulator.
- **Added feature** $(0 \to 1)$: *add* that feature's column to the accumulator.

A single move usually affects only one or two pieces (e.g. a capture, promotion, or normal displacement), so identifying changed features is straightforward. As a result, the incremental approach remains efficient even when combined with alpha-beta search.

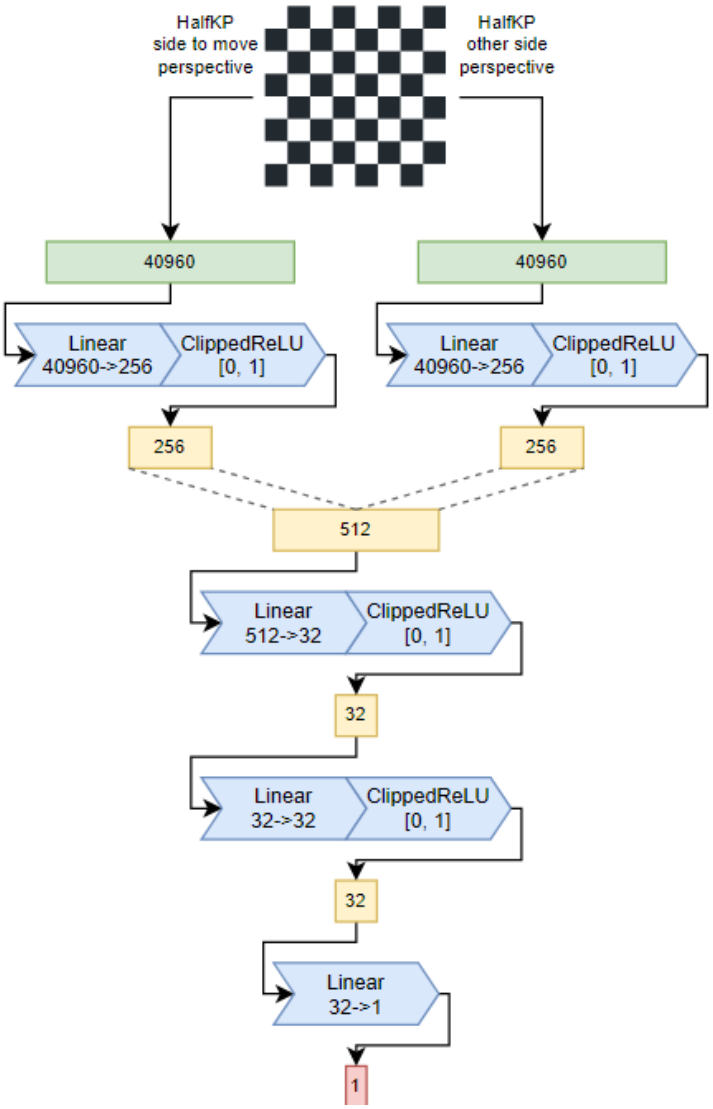


Figure 5: An example `HalfKP` architecture [15].

In Figure 5, the network's *left block* corresponds to the accumulator for the side to move, while the *right block* corresponds to the opposing side. Each block begins with a large binary input (e.g. `40960` features), encoding piece-square pairs relative to that side's king.

These features pass through a linear layer (`40960→256`) and a clipped ReLU, producing a 256-dimensional output. The two 256-dimensional outputs (one per accumulator) are then concatenated into a 512-dimensional vector, which flows through additional fully-connected layers (e.g. `512→32→32→1`) to yield a single scalar evaluation.

By updating only the features that changed (e.g. a knight moving from c3 to e4), `HalfKP` greatly reduces overhead compared to a naive feed-forward pass. This incremental strategy is especially effective on CPU-based chess engines.

HalfKAv2: Refined Input Organization `HalfKAv2` rearranges `HalfKP`'s piece-square indices to reduce collisions, where different squares might otherwise map to the same feature index. It uses *buckets*, or separate weight sets, based on how many pieces remain on the board (e.g. special "endgame" weights). These refinements improve partial updates and adapt evaluations across different game phases.

As shown in Figure 6, the indexing layer is reorganized so that each piece-square combination is mapped more distinctly. The diagram also illustrates *optional buckets*—in essence, separate weight matrices specialized for different material levels (e.g., midgame versus endgame). When the engine detects a transition (typically calculated via $(\text{piece\_count} - 1) \div 4$), it activates the relevant bucket to make use of more specialized evaluation weights. This approach allows the engine to adapt its heuristics based on how many pieces remain on the board without altering the underlying feature index.

Revised Feature Index Formula. A typical `HalfKAv2` approach avoids a separate king plane by merging white and black king categories. As shown below, the formula maps each non-king piece to one of 11 planes, flips squares for black's perspective, and combines piece type, square, and king-square into a final index:

$$
\begin{aligned}
&\textbf{If } (\text{piece\_type} = 6) \;\rightarrow\; \text{index} = -1 \quad (\text{skip king}),\\
&p\_idx = (\text{piece\_type} - 1) \times 2 \;+\; \delta(\text{piece\_color\_white} \neq \text{is\_white\_pov}),\\
&\textbf{if } p\_idx = 11 \;\rightarrow\; p\_idx = 10 \quad (\text{merge king plane}),\\
&sq\_o = \begin{cases} \text{sq}, & \text{if is\_white\_pov} = \text{true},\\ 56 \oplus \text{sq}, & \text{if is\_white\_pov} = \text{false}, \end{cases}\\
&k\_sq\_o = \begin{cases} \text{king\_sq}, & \text{if is\_white\_pov} = \text{true},\\ 56 \oplus \text{king\_sq}, & \text{if is\_white\_pov} = \text{false}, \end{cases}\\
&\text{index} = sq\_o \;+\; (p\_idx \times 64) \;+\; (k\_sq\_o \times 64 \times 11).
\end{aligned}
$$

Explanation and Buckets. If the piece type equals *king*, the formula returns -1 to skip it. Otherwise, p_idx ranges from 0 to 10 after merging potential king categories. A

perspective flip occurs when is_white_pov = false, using $56 \oplus \text{square}$ to orient the board from Black's viewpoint.

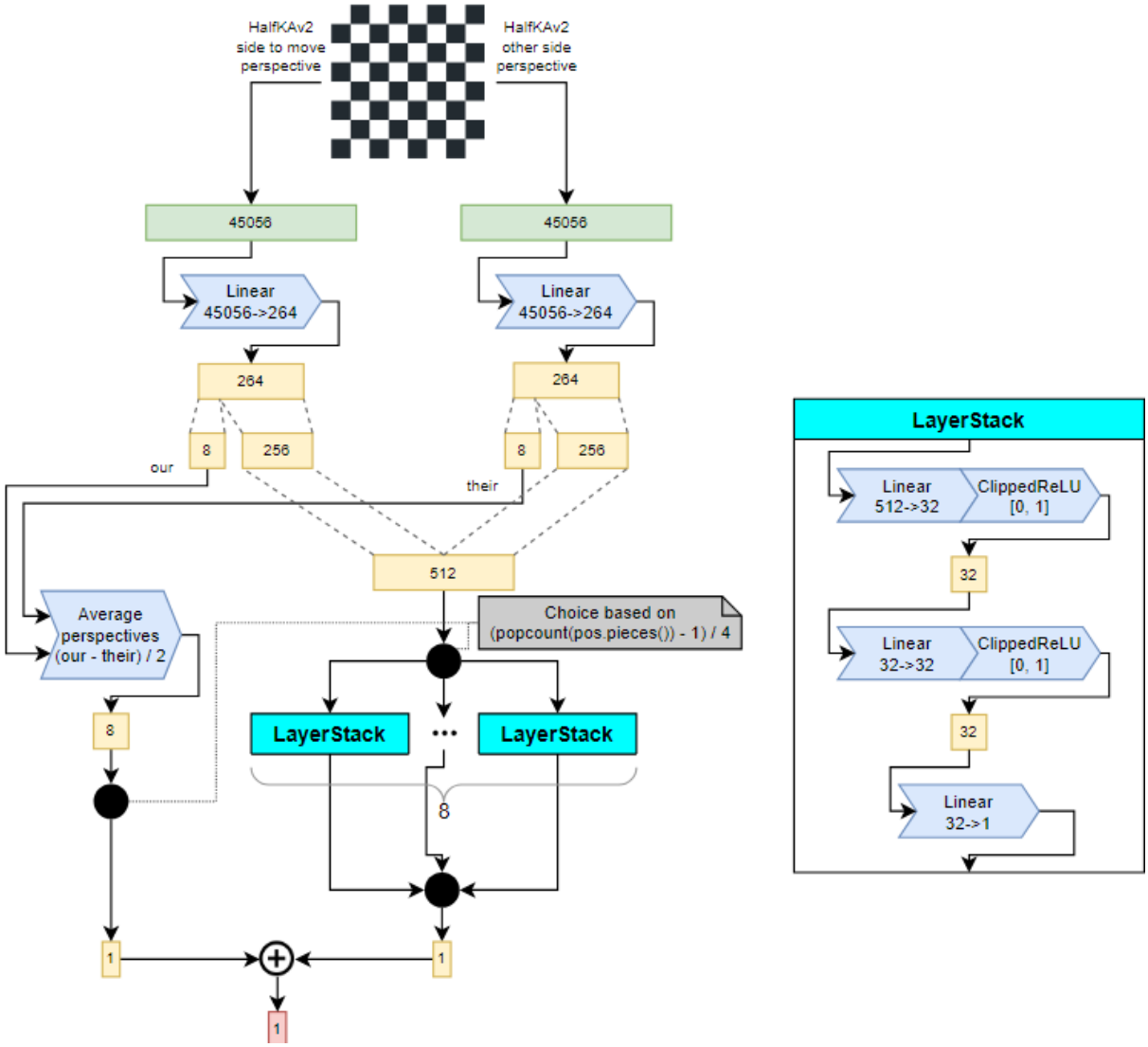


Figure 6: `HalfKAv2`: rearranged indices to minimize collisions and optional buckets for variable piece counts. Each bucket (e.g. midgame, endgame) can hold its own specialized weight set, enabling more accurate evaluations in different phases [15].

HalfKAv2_hm: Mirrored or Hash-Map Variant `HalfKAv2_hm` (sometimes referred to as "mirrored" or "hash map") refines indexing further by flipping or mirroring ranks and files. As illustrated in Figure 7, the board is mapped to feature indices in a more compact way, effectively cutting the total number of indices in half.

Horizontal mirroring reflects each rank left-to-right around the central vertical axis (files D/E). Squares on the a-file and h-file, for example, are treated as "mirrors" of each other, yet still receive unique identifiers. Compared to earlier *board flipping* (inverting ranks top-to-bottom), this method not only captures symmetrical positions but also yields *fewer overall indices*, reducing the size of any accumulators or tables that track these features.

As a result, incremental updates are much faster, making *horizontal mirroring* the more efficient choice for handling symmetries.

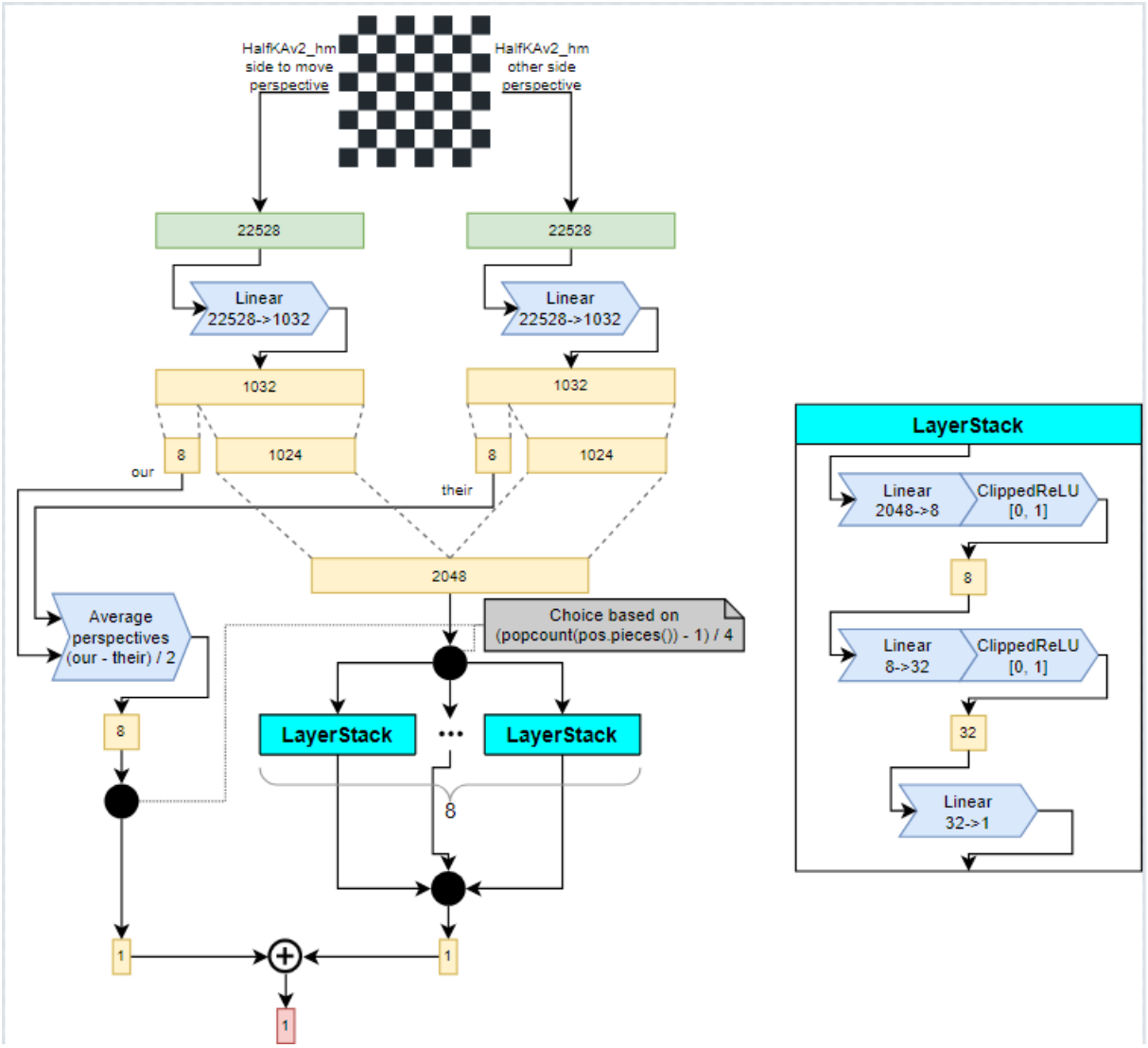


Figure 7: `HalfKAv2_hm` uses mirrored or hashed rank/file references to reduce collisions further. Here, horizontal mirroring reflects the board left-to-right, while flipping inverts ranks top-to-bottom, improving partial updates for symmetric positions [15].

Practical Impact on Engines Historically, chess engines relied on hand-tuned heuristics (e.g. piece-square tables, king safety checks). NNUE replaces many of these with a learned approach, often discovering patterns humans did not explicitly encode. By combining incremental NNUE layers with alpha-beta, engines like *Stockfish* achieve strong results in tournaments. This blend of sparse, partial updates and a machine-learned evaluation offers a robust yet efficient pipeline on regular CPUs.

1.2 Chess Platforms and Visualization Tools

Most popular chess engines, such as Stockfish or Lc0, run inside graphical interfaces (e.g., Arena, Cute Chess, SCID) that display the engine's recommended move and a numerical evaluation [16, 17, 18, 19, 20]. While this setup helps users see which move is best, it does not show *how* the engine arrives at that decision. Typically, there is no option to watch the internal search tree expand or see how each candidate move is evaluated at various depths.

Outside of chess, some interactive tools focus on *pathfinding* rather than board-game search. For instance, *MazeSolver* [21] allows users to place walls on a grid and then run algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS), observing how the chosen algorithm explores the maze cell by cell. Similarly, *PathfindOut* [22] provides a web interface for experimenting with various pathfinding methods, including Dijkstra's Algorithm and A* (A-star). Although both sites offer a clear, step-by-step illustration of how nodes are visited and paths are formed, neither provides *preloaded mazes* or *standardized benchmark layouts* that could help users measure and compare algorithm performance in a consistent manner [23]. Instead, users must hand-draw obstacles or maze patterns each time, making quick demonstrations or controlled testing more difficult.

Meanwhile, typical chess interfaces focus on final engine outputs (for example, a numeric evaluation in centipawns) without revealing the move-by-move search process [24]. While these outputs are sufficient for practical gameplay, they do not help learners or developers who want a deeper look at the algorithm's decision-making. Although pathfinding visualizers demonstrate the *benefit* of interactive, stepwise feedback, they often lack built-in benchmarks, and common chess GUIs omit visualization of the decision tree.

Summarizing these observations highlights two main gaps:

1. **Absence of Benchmarks and Preloaded Mazes in Pathfinding Tools:** MazeSolver [21] and PathfindOut [22] rely on user-defined grids and lack standardized layouts or metrics, limiting consistent, repeatable tests for performance comparison.
2. **No minimax visualizations:** Most chess platforms show only a final score or best move, without revealing how each node in the search tree is evaluated or pruned.

Addressing these gaps could provide more accessible, instructive interfaces for both algorithmic learning and engine development. In subsequent sections, we will consider approaches that enable ready-made maze scenarios for consistent testing and expose the step-by-step searching logic in a chess engine.

1.3 Hypothesis

Many maze-generation platforms lack strong visualization features, and they are rarely combined with chess engines. Inspired by NNUE's success in other projects, this work aims to build a complete platform that merges maze generation and visualization with an NNUE-enhanced Sunfish engine. The following questions guide this study:

(a) Does adding an NNUE-style network to the Sunfish engine lead to a significant improvement in playing strength, compared to the original heuristic-based method?

(b) Can creating a more advanced maze visualization tool boost clarity, user engagement, and feature completeness beyond what is currently offered?

(c) Could we develop a new tool that interactively visualizes minimax, making the algorithm more understandable?

(d) Is it feasible and beneficial to combine NNUE, enhanced maze visualization, and a minimax visualizer into one platform, offering a unified set of features to users?

These questions will be examined by measuring how much NNUE augmentation improves the engine's performance, along with evaluating user feedback on the new visualization features.

2 Background

This section presents the core technologies and frameworks used in developing our application, spanning both the front-end and back-end. We also highlight the underlying chess engine algorithms and our chosen containerization strategy.

2.1 Languages, Frameworks, and Tools

2.1.1 Front-end

When developing modern web applications, the primary front-end languages must run in the browser, which generally narrows the choices to JavaScript or TypeScript (a typed superset of JavaScript) [25]. However, front-end development also relies on foundational technologies such as HTML and CSS, standardized by the W3C, to structure and style web pages [26, 27].

In recent years, various *front-end frameworks* (e.g. React, Angular, Vue) and build tools (e.g. Webpack, Babel) have emerged to simplify development and enhance performance [28, 29]. These tools enable developers to build *single-page applications* (SPAs) that provide better user experience by dynamically rendering and updating page content without full page reloads. Additionally, package managers like *npm* or *yarn* streamline

the process of installing and managing external libraries, making it easier to integrate third-party code [30, 31].

JavaScript

JavaScript is a high-level, dynamically typed language originally designed for adding interactivity to web pages [25, 32]. It runs natively in the browser, making it a leading choice for front-end development. Over time, it has evolved through the ECMAScript standards (often referred to as ES6 or ES2015 and beyond), which introduced modern features such as arrow functions, classes, and modules. These enhancements enable more maintainable and modular code.

JavaScript uses an event-driven, single-threaded execution model, managed by the event loop. This design allows for asynchronous operations via callbacks, promises, or `async/await`, ensuring responsive user interfaces [33]. Although traditionally a client-side language, JavaScript is equally capable on the server side through platforms like Node.js, enabling full-stack development in a single language [34].

React

React is a market-leading JavaScript library (though some developers call it a "framework" due to its structured approach) created by Facebook (now Meta) [35]. Its primary strength lies in its *component-based architecture*, where UI elements are split into reusable components that manage their own `state`. When this state changes, React efficiently re-renders the affected components by leveraging a *virtual DOM*. In contrast to "vanilla" JavaScript, where developers manually update the DOM, React's virtual DOM automatically determines which parts of the real DOM need updating, minimizing unnecessary manipulations.

Additionally, React offers various *Hooks* (such as `useState` and `useEffect`) that trigger re-renders under specific conditions [36]. These Hooks reduce the boilerplate typically found in class-based components, making React both flexible and powerful for building dynamic, responsive interfaces.

Tailwind CSS

Tailwind CSS is a *utility-first* framework that provides small, reusable classes for layout, spacing, colours, and more [37]. Unlike older frameworks like Bootstrap, it does not offer pre-built components or a default theme. Instead, developers can combine individual utility classes to build custom designs.

This approach works especially well in React applications, where you can place classes directly in JSX, keeping your styles clear and consistent. Another benefit is the reduced

risk of naming conflicts across different React components, since Tailwind's classes are purpose-built for utilities rather than general styles. Tailwind also includes a configuration file that lets you enable only the classes you need, reducing your final CSS size and speeding up page loads.

Front-end Third-Party Libraries

Chess.js Chess.js is a JavaScript library that provides chess rules, move generation, and board validation [38]. It supports features like check, checkmate, and move history tracking, simplifying the implementation of core game logic for browser-based applications.

React Flow React Flow is a library designed for interactive node-based UIs [39]. By abstracting away low-level details of draggable, zoomable views, it allows developers to quickly build complex flow diagrams. In this project, React Flow supports visual representations of algorithms or game-state transitions, enhancing user understanding.

2.1.2 Back-end

FastAPI

FastAPI is a modern Python framework designed for building high-performance web APIs [40]. It leverages Python's *async* features, enabling it to handle many simultaneous requests efficiently. One of FastAPI's standout traits is its use of *type hints*, which allow it to automatically generate detailed documentation and interactive Swagger/Redoc interfaces. This makes it straightforward to develop and test endpoints without writing extra code. In this project, FastAPI provides the back-end services for receiving user requests, handling logic, and returning responses in a clean, maintainable way.

To serve these applications, FastAPI typically runs on an ASGI server, such as `uvicorn`, which takes advantage of asynchronous execution to handle many simultaneous connections. This synergy between FastAPI and `uvicorn` makes it an excellent choice for back-end services in Python projects [41].

PyTorch

PyTorch is a popular Python library for deep learning, known for its *dynamic computation graph* and ease of use [42]. It integrates smoothly with NumPy, allowing developers to switch between NumPy arrays and PyTorch tensors with minimal overhead. This flexibility makes PyTorch especially convenient for experimenting with different neural network architectures.

In this project, PyTorch is used to implement and train an **NNUE** (Efficiently Updatable Neural Network), a specialized network structure originally designed for chess and

shogi engines. NNUE leverages feature inputs that can be incrementally updated, reducing the computational cost when board states change. PyTorch's built-in `autograd` feature automates backpropagation, while its robust *GPU acceleration* shortens training time. Combined with a large community and extensive documentation, PyTorch offers a practical foundation for integrating NNUE into this AI-focused application.

Back-end Thirt-Party Libraries

NumPy NumPy is a fundamental Python library for numerical computing, providing a powerful *multi-dimensional array* object along with tools for fast operations [43]. It supports a wide range of data formats, making it straightforward to import and export datasets in formats like CSV, TXT, and more specialized binary formats. This compatibility helps streamline workflows by allowing different parts of a project—such as data preprocessing and model training—to share the same array structures. Additionally, many scientific and machine learning libraries (including PyTorch and Pandas) integrate seamlessly with NumPy, further simplifying the development of data-driven applications.

python-chess The *python-chess* library [12] provides robust functionality for chess-specific tasks such as board representation, legal move generation, and simple evaluation routines. It can also interface with stronger engines via UCI, making it a convenient tool for building, testing, or analyzing chess programs in Python.

collections The *collections* module [44] is part of the Python standard library and offers specialized container datatypes such as `deque`, `Counter`, and `defaultdict`. These structures can improve both performance and code clarity in data manipulation tasks. For instance, `deque` enables efficient appends and pops from both ends of a list, while `defaultdict` simplifies handling of dictionary keys that may not yet be defined.

2.2 Docker

Docker is a containerization platform that allows applications to run in isolated environments called *containers* [45]. Docker uses *images*, which serve as blueprints detailing all essential components and dependencies a container needs, forming the foundation from which containers are created. A typical image is essentially a block of information containing the operating system base, required libraries, and the application code. When an image is executed, it creates a container (a lightweight instance) that runs the application consistently across different systems.

In addition to images and containers, Docker provides a feature called *volumes* to handle persistent storage. Volumes let you store and manage data outside the container's filesystem, simplifying updates or container replacement without losing data. By utilizing this

functionality, Docker ensures that the application is agnostic of the underlying machine. This approach greatly enhances portability, as Docker images can be pushed to a registry or packaged into a `.tar` file. These advantages are key factors making Docker one of the most popular choices for software development and deployment.

3 Design

Design is a key factor in building a practical, visually pleasing, and *trustworthy* chess application. Good design improves how users see, understand, and interact with features. It can strengthen confidence in the system, ensure a smooth workflow, and reduce the cognitive load on users. In essence, a well-crafted design promotes **usability**, **engagement**, and **clarity**.

3.1 Importance of a Clear Layout

A simple, repetitive layout can quickly lose a reader's attention. Conversely, **dynamic layouts** that balance text and images break large blocks of information into more manageable pieces, making the user experience more welcoming. Research indicates [46] that alternating text and visuals (often called a *zigzag* pattern) can reduce visual fatigue and boost content retention. Following ideas from Nielsen [47], this project also uses **keyword highlights**—for instance, marking key phrases (`NNUE` or `Minimax`) in `green` or `red`—to draw attention and guide the user's eye.

3.2 Layout-Driven Approach and Zigzag Highlights

Tsumura et al. [48] emphasize a *layout-driven approach* to webpage reading, in which the structure, visual emphasis, and text-image placement guide how users (and screen readers) interpret content. Within this approach, the *zigzag pattern* is especially useful for preventing visual monotony and directing attention . On larger screens, text typically appears on one side while an image or illustration occupies the opposite side. On smaller screens, these elements stack vertically to preserve readability.

Moreover, **color-coded keywords** (e.g., `text-green-500`) help emphasize terms like `NNUE` or `Minimax`, drawing the user's focus immediately. These design choices reduce visual fatigue and highlight important concepts, aligning with the layout-driven principle of conveying structure through both text and visual arrangement.

Example of a Zigzag Layout in Code. Listing 8 illustrates how `md:grid md:grid-cols-2` sets up a two-column layout for medium-sized screens, with classes such as `text-green-500` to emphasize crucial terms.

```jsx
<section className="md:grid␣md:grid-cols-2␣md:gap-12␣items-
    center">
  <div>
    <h2 className="text-2xl␣font-bold␣mb-4">
      Understanding <span className="text-green-500">NNUE</span
         >
    </h2>
    <p className="text-gray-300␣leading-relaxed">
      The <span className="text-green-500">NNUE</span> approach
      offers more flexible and precise board evaluations...
    </p>
  </div>
  <div className="bg-gray-900␣p-6␣rounded-md␣shadow-lg">
    {/* Additional content or an image can appear here */}
  </div>
</section>
```

Figure 8: A React snippet showing a two-column layout (zigzag) and highlighted keywords (e.g., `NNUE` in green).

3.3 Screenshot and Visual Confirmation

Figure 9 shows how this layout appears in the actual interface. If only one main image is available (or if multiple images look nearly the same), using a single best screenshot can effectively demonstrate how text and images alternate, preventing a dull or uniform look. Notably, highlighted keywords improve visibility of key features.

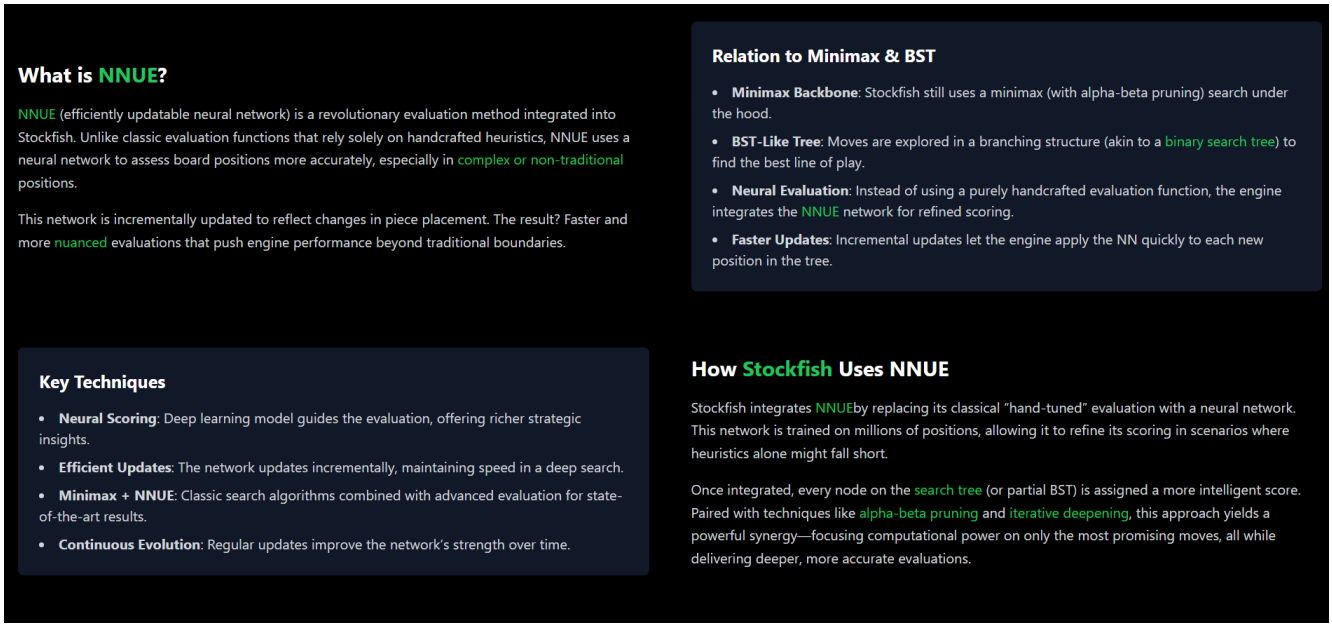


Figure 9: An example of the partial zigzag design. Text is on one side, an image on the other, while keywords such as `NNUE` appear in green to attract focus.

3.4 Conclusion

In summary, this project's design approach centers on:

- **Avoiding Repetition**: Alternating (zigzag) or partially alternating text and images to keep viewers interested.
- **Emphasizing Key Terms**: Applying color highlights to important labels like "`NNUE`."
- **Responsive Layout**: Ensuring the layout scales gracefully across different screen sizes.
- **Building User Trust**: Presenting information in a structured, polished manner so users feel confident in the application.

By adhering to this *layout-driven approach*—balancing visual variety and carefully placed highlights—the design encourages consistent user focus, improves comprehension of crucial features, and fosters a more positive interaction experience.

4 Architecture

4.1 Overview

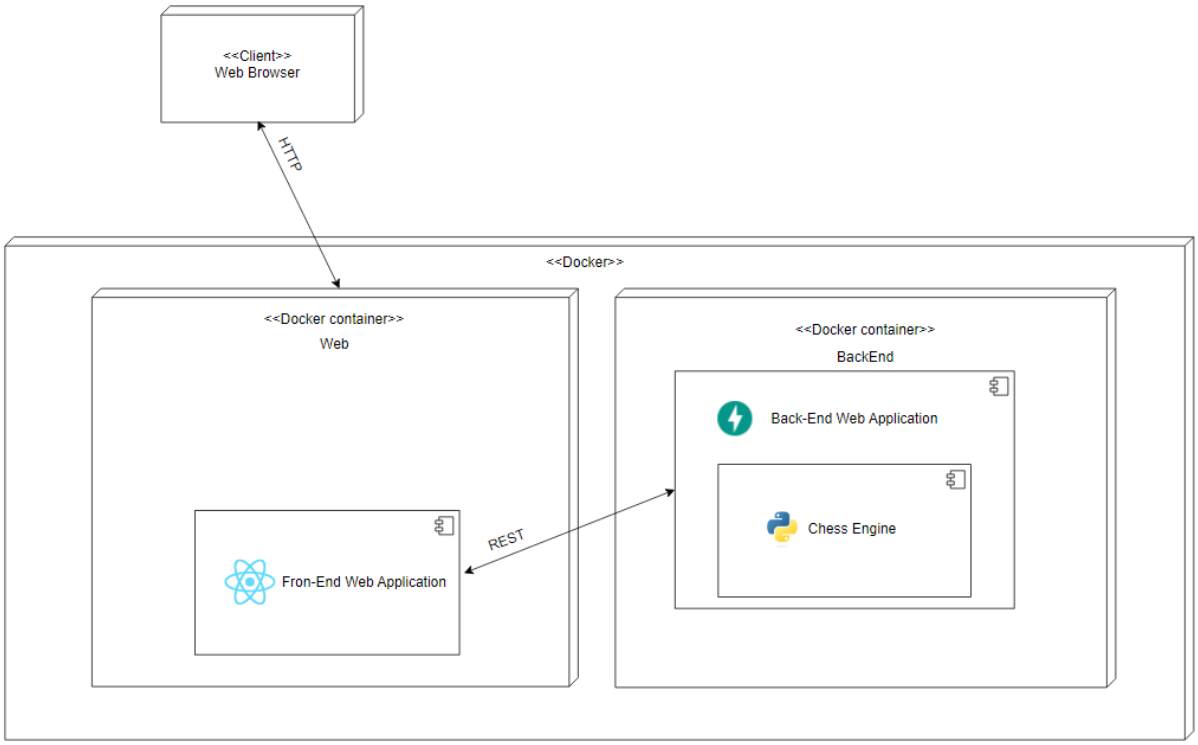


Figure 10: diagram illustrating the user's browser interacting with a front-end container and a back-end container, which houses the Chess Engine.

As shown in Figure 10, the system is composed of two Docker containers— one for the front-end (React) and another for the back-end (FastAPI + NNUE). The user accesses the application through a standard web browser, sending HTTP requests to the front-end, which in turn communicates with the back-end via REST.

4.2 Communication Flow

Communication between the React front end and the FastAPI back end follows a straightforward RESTful request–response cycle over HTTP. Figure 10 provides a high-level overview, while this subsection details how each endpoint fits into the application's workflow. Generally, users interact with buttons or pages in the React interface, triggering specific HTTP requests to the FastAPI server. The server processes these requests and returns the necessary data, which the front end uses to update the user interface.

Maze Endpoint

- `GET /api/maze`:
  - **Purpose**: Generates a maze based on specified dimensions (`width`, `height`, and `tile`), then returns a text representation, with `0` indicating free space and `1` indicating a wall.
  - **Usage**: The React client (e.g., a Maze page or component) requests a maze whenever the user clicks a "Generate Maze" button.
  - **Flow**:
    1. **Front end** sends `GET /api/maze?width=40&height=20&tile=2`.
    2. **Back end** calls `generate_maze` and `convert_maze_to_array`, then converts the resulting 2D array into a string.
    3. **Response** is returned as plain text, which the React client parses to display the maze layout.

Chess Endpoints

- `POST /chess/makemove`:
  - **Purpose**: Applies a user-submitted move (in UCI format) to the current chess position, then optionally runs a server-side chess engine for its own move suggestion.
  - **Usage**: In the React chess page, the user selects and submits a move (e.g., `e2e4`), and the front end sends a JSON payload like {`"move": "e2e4"`}.

- **Flow**:
    1. **Front end** sends `POST /chess/makemove` with the chosen move in JSON.
    2. **Back end** processes the move using `process_move`, updating the `hist` (game history). It then calls the embedded chess engine, which accepts a UCI-style board representation to evaluate and possibly generate a response move.
    3. **Response** includes a status flag, any engine-generated move, and the updated FEN string so the front end can refresh the board state accordingly.

- `POST /chess/reset`:
    - **Purpose**: Restores the chessboard to the initial opening position.
    - **Usage**: Triggered by a "Reset" or "New Game" button on the chess page.
    - **Flow**:
        1. **Front end** sends `POST /chess/reset`.
        2. **Back end** reinitialises the `hist` array to the default start position.
        3. **Response** returns a status flag plus the FEN of the new board, allowing the React client to render the reset position.

CORS and Deployment Notes To enable local development, `CORSMiddleware` in FastAPI allows requests from `http://localhost:3000`, ensuring the React front end can communicate with the back end without cross-origin errors. For production, both parts can be containerised via Docker, allowing consistent environments across different machines.

Response Handling in React Regardless of the endpoint, the React client:

1. Makes an HTTP request (with parameters or JSON data).
2. Receives a response (plain text for the maze, JSON for chess moves).
3. Updates the relevant component (e.g., Maze or ChessBoard) with the new state.
4. Displays errors or invalid inputs using UI notifications if necessary.

By keeping the client-side focused on user interaction and display, and offloading maze generation and chess logic (including engine moves) to the server, this approach fosters a clear division of responsibilities in the application.

4.3 User Interface Structure

To facilitate both learning and direct experimentation, the web application offers two approaches: a *guided path* for step-by-step exploration of concepts, and the option to *jump directly* into specific visualization or gameplay features. Four main pages organize these capabilities:

1. **Home Page:** Introduces the project, providing general information and navigation links to the algorithm demos. It also lets users follow a guided path consisting of three detailed pages, each explaining:
   - What Breadth-First Search (BFS) is and how it applies to maze traversal.
   - How to build a basic NNUE (Neural Network Updatable Engine).
   - How the minimax algorithm works, including its alpha–beta pruning variant.

   This guided path gives novices a structured way to learn each concept before experimenting with live demos. Alternatively, users can skip the path and directly access any feature of interest.
2. **Maze Visualization Page:** Demonstrates different pathfinding algorithms (including BFS) on a maze. Users can watch the process, adjust maze dimensions or generation parameters, *or even construct a custom maze* by defining walls and starting points.
3. **Minimax Traversal Page:** Allows users to step through a minimax search with alpha–beta pruning on a specific chess problem, illustrating how the algorithm chooses optimal moves.
4. **Chess Page:** Lets users play against multiple engines such as Stockfish and Yanfish (the custom engine implementation).

4.3.1 Home Page

Serving as the application's entry point, the Home Page briefly explains the project's goals and functionalities. It also provides clear navigation links (e.g., a top menu or sidebar) to the other three pages. By offering a concise overview, this page ensures visitors can quickly choose between following the guided path or diving straight into any tool.

4.3.2 Maze Visualization Page

This page visualizes various pathfinding algorithms within randomly generated or custom-built mazes. It includes:

- **Algorithm Selection:** Users can choose BFS, DFS, Dijkstra, or A*.

- **Custom Maze Building:** Walls and pathways can be placed manually, allowing tailored scenarios.
- **Adjustable Maze Settings:** Parameters such as maze width or height can be changed to produce new layouts.
- **Interactive Controls:** Start, pause, or reset the pathfinding process; observe how each cell is explored.
- **Visual Feedback:** The traversal is animated step by step, helping users understand how these algorithms systematically uncover paths.

4.3.3 Minimax Traversal Page

Here, users can explore how the minimax algorithm (with alpha–beta pruning) handles a specific chess problem:

- **Depth Selection:** Users can set the number of moves before running the algorithm.
- **Step-by-Step Explanation:** Each move can be examined using a control panel.
- **Interactive Visualization:** A visualized decision tree where users can double-click on a node to expand it, or fully expand the entire tree in one step.

4.3.4 Chess Page

On the Chess Page, users can play interactive games against multiple engines:

- **Engine Selection:** Switch between Stockfish and Yanfish (the custom NNUE-enhanced engine) to compare playing styles.
- **Interface:** Displays the chessboard along with a move history, time controls, and a record of captured pieces.

Each of these four pages is accessible via a navigation bar at the top of the website. By separating functionality into modular subpages, users can easily focus on the specific algorithm or chess feature they wish to explore, while the underlying code remains cleaner and more maintainable. For further screenshots illustrating the interface, Appendix A.

4.4 Build

To standardise environments and simplify deployment, we containerise both the front end and back end using Docker. Figure 10 shows that each component runs in its own container:

- **Front end (React):** We use a `Dockerfile` based on an Alpine-based Node image. After installing dependencies and building the production bundle, the container

serves the compiled files. Alpine Linux provides a smaller footprint, reducing overall image size.

- **Back end (FastAPI + NNUE):** We use a `Dockerfile` based on an Alpine-based Python image. It installs libraries such as FastAPI, PyTorch, and any NNUE-specific packages, then launches `uvicorn`. Using Alpine reduces image bloat and streamlines resource usage.

Docker Compose is a YAML-based configuration tool that allows you to define and manage multiple containers for a single application. Instead of running separate Docker commands, you place all relevant settings—such as build instructions, environment variables, and port mappings—into a single file, typically called `docker-compose.yml`. This setup simplifies both development and deployment by letting you use one command (`docker-compose up`) to start all services at once.

```yaml
version: "3.8"

services:
  BackEnd:
    build: .
    container_name: "fastapi_app"
    restart: "always"
    ports:
      - "8000:8000"
  Web:
    build:
      context: "./my-project"
    container_name: "frontend_app"
    restart: "always"
    ports:
      - "3000:3000"
    depends_on:
      - "BackEnd"
    environment:
      - REACT_APP_API_BASE_URL="http://localhost:8000"
```

Listing 1: docker-compose.yml

5 Implementation

Having established the theoretical underpinnings and high-level design in previous chapters, we now turn to the practical side: the code. This section is divided into three parts. First,

we walk through the neural network (NNUE) evaluation framework and its role in enhancing the chess engine's strength. Second, we unveil the backend, which orchestrates data flow and enforces game rules. Finally, we showcase the frontend implementation, illustrating how users interact with the engine's capabilities through a clean and responsive interface.

5.1 NNUE Model

The NNUE (Neural Network-based Underestimated Evaluation) model is central to this project, as noted in the hypothesis section 1.3. The core idea is to develop a platform that brings together various visualization tools and integrates a chess engine enhanced by NNUE. By leveraging NNUE, the goal is to achieve more accurate position evaluations and stronger overall gameplay compared to traditional heuristic-based methods.

5.1.1 NNUE Implementation

This project employs the HalfKP architecture, one of the earliest NNUE-based designs used in Stockfish. Because training a model to reach grandmaster-level performance is time-consuming and requires extensive resources, we utilize pre-existing weights from a previously established neural network. To ensure consistency and correctness, we replicate the same operations that were used in the original training pipeline (Figure 5). The key steps are:

- Implement a feature transformer that calculates indices for each board position and generates an accumulator, enabling efficient updates.
- Construct a neural network that adheres exactly to the original HalfKP design.
- Integrate the neural network into the existing Sunfish chess engine.

This approach allows the project to achieve a competitive level of strength in position evaluation and move generation without incurring the high costs of training a new network from scratch.

Accumulator

The HalfKP approach (Section 1.1.5) relies on an accumulator to handle feature transformations efficiently. By storing intermediate computations, the accumulator enables faster updates whenever a board state changes. This is particularly useful for searching through a large number of positions in the course of a chess engine's operation.

To implement this functionality, a `MyAccumulator` class was created in Python. This class leverages several libraries:

- `chess` to represent and manipulate chess positions,
- `numpy` for efficient numerical operations, and
- `pytorch` for neural network integration.

```python
class NNUEAccumulator:
    def __init__(self,
                 weights_file="engines/weights/transformer_weights.npy",
                 bias_file="engines/weights/transformer_bias.npy"):
        # Load the transformer's embedding weights and bias
        self.weights = np.load(weights_file)  # shape (41024, 256)
        self.bias = np.load(bias_file)        # shape (256,)

        # Validate shapes to ensure they match the expected dimensions
        assert self.weights.shape == (DEFAULT_NUM_FEATURES, FEATURE_TRANSFORMER_HALF_DIMENSIONS)
        assert self.bias.shape == (FEATURE_TRANSFORMER_HALF_DIMENSIONS,)

        # Initialize accumulator storage (batch dimension of 1, size of 512)
        self.accum = np.zeros((1, 512))
```

Listing 2: Part 1: Initialization of NNUEAccumulator

Explanation In Listing 2, the constructor method loads the pre-trained weights and biases, checks their shapes against constants (such as `DEFAULT_NUM_FEATURES` and `FEATURE_TRANSFORMER_`) and initializes a zeroed accumulator that will be updated incrementally after each move.

```python
    def update(self, accum, turn, move, kings):
        """Incrementally update the accumulator for a new position."""
        from_sq = move[0]
        from_pc = move[1]
        to_sq   = move[2]
        to_pc   = move[3]

        our_king, opp_king = (kings[0], kings[1]) if turn else (kings[1], kings[0])

        # Create a new array matching the shape of the existing accumulator
        self.accum = np.empty_like(accum)

        # Calculate indices for the piece being moved
```

```
14          index_from_w_perspective = self.make_halfkp_index(
15              turn, self.orient(turn, our_king), from_sq, from_pc
16          )
17          index_from_b_perspective = self.make_halfkp_index(
18              not turn, self.orient(not turn, opp_king), from_sq,
                    from_pc
19          )
20
21          # If a capture or promotion occurs, calculate indices for
                those scenarios
22          if to_pc is not None:
23              index_to_w_perspective = self.make_halfkp_index(
24                  turn, self.orient(turn, our_king), to_sq, to_pc
25              )
26              index_to_b_perspective = self.make_halfkp_index(
27                  not turn, self.orient(not turn, opp_king), to_sq,
                        to_pc
28              )
29
30          # Handle promotion cases for pawns
31
32
33          ...
34          # Update accumulators for both color perspectives
35          self.accum[0][:256] = accum[0][256:512] \
36                                + self.weights[
                                      index_to_b_perspective_new] \
37                                - self.weights[
                                      index_from_b_perspective]
38          self.accum[0][256:512] = accum[0][0:256] \
39                                   + self.weights[
                                         index_to_w_perspective_new] \
40                                   - self.weights[
                                         index_from_w_perspective]
41
42          # Subtract weights for captured piece if applicable
43          if to_pc is not None:
44              self.accum[0][:256] -= self.weights[
                    index_to_b_perspective]
45              self.accum[0][256:512] -= self.weights[
                    index_to_w_perspective]
46
47          return self.accum
```

Listing 3: Part 2: The update method snippet

Explanation In Listing 3, the `update` method performs an incremental adjustment of the accumulator to reflect the latest move on the chessboard. First, it extracts the relevant details of the move:

- `from_sq`: The square the piece is moving from
- `from_pc`: The piece on `from_sq`
- `to_sq`: The square the piece is moving to
- `to_pc`: The piece on `to_sq` (if any), used to detect captures or promotions

Next, it determines whose move it is by setting `our_king` to the current mover's king and `opp_king` to the opponent's king. A fresh `accum` array is created with `np.empty_like` to store updated feature values. The method then calculates the feature indices for the moved piece (from both White's and Black's perspective) using `make_halfkp_index` 4, which converts a piece-specified square into a unique index for the HalfKP feature set.

If a capture or promotion occurs, the code computes additional indices to account for the new piece occupying the square (`to_pc` or a promoted piece), as well as to remove any influence of the captured piece. Pawn promotions involve substituting the moved pawn with a queen piece (unless you alter it to handle other promotion cases) and calculating corresponding indices for both color perspectives.

The accumulator is updated for each half of its array: indices `[0:256]` represent Black's perspective, and indices `[256:512]` represent White's. In both ranges, the method:

1. **Adds** the weights of any newly occupied square (promotions or regular moves),
2. **Subtracts** the weights of the old square if the piece moved away or was captured,
3. **Repositions** the piece by shifting its contribution from the source index to the destination index.

Finally, if `to_pc` is not `None`, meaning a capture happened, the contribution of the captured piece is removed from both perspectives by subtracting its corresponding weights.

This incremental approach ensures that only the portions of the feature vector changed by the move are recalculated, saving computation time. Instead of rebuilding the entire feature set from scratch, the method selectively applies additions or subtractions to reflect the new board state. The updated `accum` is returned for use in subsequent neural network evaluations, allowing the chess engine to efficiently evaluate each move in the game with minimal overhead.

```
1     def get_halfkp_indices(self, board: chess.Board):
2         """Compute␣indices␣for␣all␣pieces␣on␣the␣board␣from␣both␣
              perspectives."""
```

```
3         result = []
4         is_white_pov = board.turn
5
6         for turn_state in [board.turn, not board.turn]:
7             indices = []
8             for sq, piece in board.piece_map().items():
9                 if piece.piece_type == chess.KING:
10                    continue
11                indices.append(
12                    self.make_halfkp_index(turn_state,
13                                           self.orient(turn_state,
                                                board.king(turn_state)
                                                ),
14                                           sq, piece)
15                )
16            result.append(indices)
17
18        return np.array(result, dtype=np.intp)
19
20    @staticmethod
21    def orient(is_white_pov: bool, sq: int) -> int:
22        """Flip or retain square indexing based on side to move."""
23        return (63 * (not is_white_pov)) ^ sq
24
25    @staticmethod
26    def make_halfkp_index(is_white_pov: bool, king_sq: int, sq: int,
          p: chess.Piece) -> int:
27        """Combine orientation, piece-square encoding, and king-
              square indexing for HalfKP."""
28        return NNUEAccumulator.orient(is_white_pov, sq) \
29               + PieceSquare.from_piece(p, is_white_pov) \
30               + PieceSquare.END * king_sq
```

Listing 4: Part 3: Utility methods for orientation and index computation

In essence, `MyAccumulator` captures the relevant board features and transforms them into a format suitable for the neural network. Because each move in a chess game typically alters only a few squares, this approach allows updates to be carried out incrementally rather than computing all features from scratch.

NNUE

While the accumulator focuses on efficiently tracking board features, the NNUE network itself leverages these features to evaluate positions. This network is built using the PyTorch library, allowing for straightforward model definition, training, and inference. By combining the HalfKP feature representation with a compact yet expressive feed-forward

architecture, the network can efficiently process positional information and deliver accurate evaluations.

```python
class MyNNUE(nn.Module):
    def __init__(self):
        super(MyNNUE, self).__init__()
        self.layer1 = nn.Linear(512, 32)
        self.layer2 = nn.Linear(32, 32)
        self.output_layer = nn.Linear(32, 1)
        self.relu = nn.ReLU()
        self.load_extracted_weights()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.shift_and_clamp_torch(x)
        x = self.relu(self.layer2(x))
        x = self.shift_and_clamp_torch(x)
        x = self.output_layer(x)
        return x  # shape (batch_size, 1)
```

Listing 5: A simplified PyTorch implementation of the NNUE network structure

Explanation The `MyNNUE` class follows a design similar to one of the early Stockfish neural networks. It includes two hidden layers, each with 32 neurons, and a final output layer that produces a single evaluation score. Between each layer, the `shift_and_clamp_torch` function (discussed in Section 6) ensures intermediate activations remain within a limited integer range, aligning with the integer-based arithmetic common in many NNUE engines. Additionally, the method `load_extracted_weights` inserts pretrained weights and biases directly into each layer, allowing this network to immediately leverage a previously optimized model. By structuring the network in this way, we can achieve a balance of simplicity, speed, and accuracy while still preserving the benefits of an NNUE-style incremental update system.

Shift and Clamp Logic

Clamping prevents activations from growing too large or falling below a minimum value, while shifting can help align outputs with subsequent layers in an integer-based or limited-precision environment.

```python
def shift_and_clamp_torch(self, x: torch.Tensor) -> torch.Tensor:
        """Bit-shifts␣and␣clamps␣values␣to␣[0,␣127]␣while␣ensuring␣
           correct␣data␣types."""
        x = x.to(torch.int32)  # Convert to int32 before bit-
           shifting
        x = x >> SHIFT  # Apply bit-shift
```

```
5         x = torch.clamp(x, CLAMP_MIN, CLAMP_MAX)  # Clamp to [0,
              127]
6         return x.to(torch.float32)  # Convert back to float32 for
              the neural network
```

Listing 6: Example of a shift and clamp step

Explanation Listing 6 demonstrates how a "shift and clamp" operation might look in a PyTorch-based NNUE setting. The `tensor` (which could be the output of a hidden layer or an accumulation step) is first shifted by a fixed amount to reduce its magnitude. This is often done in environments where integer arithmetic is used for performance or hardware constraints. Next, `torch.clamp` enforces a minimum and maximum value to prevent numerical overflow or excessively negative values. By maintaining outputs within a controlled range, we reduce precision loss and ensure the network's evaluations remain stable.

5.1.2 Evaluation in the Search Loop

This project integrates an NNUE-based evaluation into the Sunfish chess engine in a manner reminiscent of older Stockfish versions that initially relied on NNUE only at leaf nodes. Specifically, the `MyNNUE` class is invoked to generate evaluations at depth zero, leaving the deeper layers of the search to use Sunfish's traditional heuristic evaluation and pruning mechanisms. This hybrid approach provides several benefits:

- **Incremental NNUE Adoption:** By limiting NNUE calls to leaf nodes, the engine avoids performing expensive neural network evaluations at every depth, thus keeping computational overhead manageable.
- **Preservation of Proven Heuristics:** Deeper search layers still rely on well-tested heuristic rules, ensuring that pruning logic and move ordering—key components of Sunfish's design—remain stable and effective.
- **Accumulator-Based Updates:** The `MyNNUE` class also employs an accumulator, which constructs and incrementally updates a 512-dimensional feature vector. When moving from one leaf node to another, only the parts of the board state that have changed get recomputed, reducing the cost of recalculating features from scratch.

In practice, when the engine explores a position at a leaf node (i.e., a position where depth = 0 or quiescence search ends), it instructs the accumulator to provide a current feature vector. This vector is then passed to the neural network in `MyNNUE`, which returns an evaluation score. That score is incorporated into the search's alpha-beta logic to finalize a leaf evaluation. At deeper search levels, Sunfish's existing, well-understood material/positional heuristics continue to guide move ordering, pruning, and extensions. This

incremental, leaf-focused adoption of NNUE allows the engine to benefit from more accurate evaluation where it matters most—at terminal nodes—while avoiding the overhead of a full-scale neural network call on every position in the search tree.

```
# Inside the Searcher class

# Instantiating the NNUE network
nnue = MyNNUE()

# Helper function to clamp and convert our accumulator for NNUE
def process_accum_for_nn(accum: np.ndarray) -> torch.Tensor:
    processed = np.clip(accum, CLAMP_MIN, CLAMP_MAX)
    return torch.tensor(processed, dtype=torch.float32)

# ...
# Later in the 'bound' method, we compute the leaf evaluation.
if depth == 0:
    if not self.use_classical:
        with torch.no_grad():
            x = process_accum_for_nn(cur_accum)  # shape: [1, 512] float32
            score_tensor = nnue(x)
        raw_score = int(score_tensor.item())
        score = (((raw_score // 16) * 100) // 208)
        print(score)
    else:
        score = pos.score

    # Return the leaf evaluation
    yield None, score

# ...
# Setting up and maintaining the accumulator
nnue_accum = NNUEAccumulator()
cur_accum = np.empty([1, 512], dtype=np.float32)  # Avoid direct changes to root

if not self.use_classical:
    if accum_up:
        # Recalculate entire accumulator if the king has moved or there's a special update needed.
        board = chess.Board(renderFEN(pos))
        turn = board.turn
        kings = (board.king(turn), board.king(not turn)) if turn else (board.king(not turn), board.king(turn))
        ind = nnue_accum.get_halfkp_indices(board)

```

```
40          cur_accum[0][:256] = np.sum(nnue_accum.weights[ind[0]], axis
                =0)
41          cur_accum[0][256:] = np.sum(nnue_accum.weights[ind[1]], axis
                =0)
42          cur_accum[0][:256] += nnue_accum.bias
43          cur_accum[0][256:] += nnue_accum.bias
44
45          accum_up = False
46      else:
47          # Incremental update if only one piece has moved.
48          if move_prev:
49              move_chess = chess_move_from_to(pos_prev, move_prev)
50              turn = False if pos_prev.board.startswith('\n') else
                    True
51              cur_accum = nnue_accum.update(accum_root, turn,
                    move_chess, kings)
52          else:
53              # Handle null moves by swapping the two halves of the
                    accumulator
54              cur_accum[0][0:256] = accum_root[0][256:]
55              cur_accum[0][256:] = accum_root[0][0:256]
```

Listing 7: Refined Sunfish Searcher Code snippet with NNUE Integration

Explanation In Listing 7, the `bound` method checks if `depth == 0` and, if so, uses `MyNNUE` to evaluate the position. By limiting neural network inferences to leaf nodes, the engine avoids excessive overhead. Meanwhile, `NNUEAccumulator` maintains a 512-dimensional feature vector. Depending on whether a king move or null move has occurred, the code either rebuilds the entire feature vector or uses `update` to adjust only the parts affected by the last move. This incremental approach reduces unnecessary recalculations and preserves the efficiency of Sunfish's search, while still benefiting from NNUE's strong evaluation at critical leaf positions.

5.2 Backend Implementation

The backend server in this project is designed to be straightforward yet versatile. As outlined in Section 4, it relies on `FastAPI` to manage server-side logic and client communications. One of the key design principles is to keep configuration minimal. Since the system does not require user authentication or persistent database storage, its core functionality depends almost entirely on `FastAPI` and a simple CORS setup, with no additional middleware or specialized libraries needed.

Several endpoints handle specific operations: Chess Move Submission, Maze Generation, and Game Reset. When a user submits a chess move in UCI format (for example, `e2e4`),

the server updates the internal game state and returns the resulting position in FEN notation. This approach keeps the backend flexible by using well-known chess notation standards. The Maze Generation endpoint employs a depth-first search (DFS) with backtracking to produce a random maze, then serves the final result as either a text-based grid or a PNG image. By providing multiple content types in its responses, the backend demonstrates how `FastAPI` can serve diverse data formats with minimal overhead. A dedicated Game Reset endpoint reinitializes the chess position, allowing new games to start without restarting the server.

A notable component of this design is the DFS-based Maze Generation. Every cell in the maze starts with all four walls intact. A random cell is selected as the starting point and marked as visited. The algorithm uses a stack to track the path: at each step, it picks a random neighbor that has not been visited, removes the shared wall between the current cell and that neighbor, and proceeds until every cell has been visited exactly once. This guarantees a single connected path through all cells without cycles, ensuring that each maze remains unique. The final structure is then translated into text or a PNG image for convenient retrieval by the client.

Central to the maze generation is the `Cell` class, which encapsulates the logic for each grid location. A `Cell` stores its coordinates `(x, y)` as well as a dictionary of walls indicating whether each side (`top`, `right`, `bottom`, `left`) is still intact. It also features a boolean flag, `visited`, to indicate whether the maze-generation process has already included that cell in the path. Two methods, `check_cell` and `check_neighbors`, handle neighbor validity: `check_cell` returns `None` if given invalid coordinates or the corresponding cell otherwise; `check_neighbors` collects all valid, unvisited neighbors and then randomly selects one to continue carving passages. Once a path is established between two neighboring cells, their shared walls are removed, and both cells become part of the same contiguous corridor.

The backend itself runs on `uvicorn`, a high-performance ASGI server. By default, it listens on `127.0.0.1:8000`, which makes local testing straightforward and deployment to ASGI-compatible hosts simple. For chess functionality, a global list, `hist`, keeps track of all positions, updating whenever a move is submitted. If the user requests an engine move, the backend calculates and returns it in standard notation. By employing established formats such as FEN for board positions and UCI for moves, the backend ensures seamless compatibility with diverse client applications.

In summary, the combination of `FastAPI` and a straightforward DFS-based maze-generation algorithm creates a simple yet flexible backend. The `Cell` class ensures the maze construction process is well-structured, while returning data in multiple formats (JSON, text, images) highlights the adaptability of the system. This design is particularly suited to a lightweight frontend that needs immediate, clear data responses and minimal setup on

the server side.

```
backend
|   endpoint.py
|   maze.png
|   maze.txt
|   __init__.py
|   ...
+--- engines
|   |   1.py
|   |   ...
|   \--- __pycache__
|       |   ...
|       \---
   numpy_weights
|                ...
\--- tests
    |   test_sunfish.py
    \--- __pycache__
            ...
```

Figure 11: Simplified project directory structure

5.3 Frontend Implementation

The user interface is built with React (JavaScript) as the core framework, and Tailwind CSS for styling. Each page is implemented as a standalone React component consisting of multiple sub-components that collaborate to provide desired functionalities while maintaining clarity and clean code. We also leverage well-established libraries such as react-router (for seamless navigation) and `react-flow` (for interactive flow diagrams), installing and managing these dependencies via `npm`.

5.3.1 Navigation

Navigation in the frontend uses `react-router-dom` to allow users to switch between different parts of the application without reloading the entire page. This setup makes transitions fast and enables the interface to update its content seamlessly. A single file, `App.js`, maps each URL path to a React component under `components/`.

Each major feature or page has its own folder and entry point, keeping the code organized. For example:

- `components/HomePage.jsx` holds the homepage layout.
- `components/MazeSolvingPage/MazeSolvingPage.jsx` includes maze-related features and visuals.
- `components/ChessGamePage/ChessGamePage.jsx` manages the chess interface and coordinates with the backend.

```
.
|   Dockerfile
|   README.md
|   package-lock.json
|   package.json
+--- src
|   |   App.css
|   |   App.js
|   |   App.test.js
|   |
|   +--- components
|   |   |   ChessGamePage
|   |   |   DecisionTreePage
|   |   |   Header.jsx
|   |   |   HomePage.css
|   |   |   HomePage.jsx
|   |   |   MazeSolvingPage
|   |   |   MinimaxPage
|   |   |   NnnuePage
|   |   |   bstPage
|   |   |   engine
|   |   \--- toberemoved
|   |
|   |   index.css
|   |   index.js
|   \--- setupTests.js
\--- tailwind.config.js
```

Figure 12: A simplified version of the project's directory structure.

From a user's perspective, each page is accessed by a unique path (e.g., `/maze` for the maze solver, `/chess` for the chess interface). `react-router-dom` checks the requested path and seamlessly loads the correct component. Because the routing is handled on the client side, the experience feels smooth compared to a full page reload.

A shared header at the top level maintains a consistent layout across all pages. As the application grows, it can be split into sub-routes or nested routes to handle larger sections. Keeping the routing logic in `App.js` also makes adding or modifying pages straightforward.

5.3.2 Maze Visualization

Our maze is presented as a grid of cells, each of which can be a wall (`1`) or an open space (`0`). When we fetch the raw maze data from the backend, we split it into lines and transform it into a two-dimensional array, which we refer to as *grid*. This *grid* helps the frontend recognize walls versus walkable paths.

To show the internal steps of a search algorithm (like DFS or BFS), we record a series of "events" in a dedicated array, which we call *dfsEvents* for Depth-First Search or *bfsEvents* for Breadth-First Search. Each entry in this array includes the row and column of the cell, plus whether the search is *'visiting'* or *'backtracking'*. We store a *step* index in React

state, indicating how many events have been processed so far. By incrementing *step* one unit at a time (via a small time delay), the visualization updates cell by cell, letting us see the search progress in "real time."

For coloring, walls remain black, while visited path cells are highlighted in distinct colors depending on their status. If a cell is on the current search path, it appears more vividly (for instance, red), and if it's already explored and removed from the path (*'backtracked'*), it might appear in a lighter shade (like light blue). We also mark the *start* cell with an `S` and the *end* cell with an `E`. This color-coding helps illustrate the search's forward movement and backtracking phases, offering a clear, step-by-step look at how the algorithm finds its route through the maze.

By combining this event-based approach with incremental state updates in React, we provide an easily understood animation of how the algorithm explores each corridor, hits dead ends, and eventually reaches the goal. It also makes it simple to switch among different pathfinding algorithms, such as DFS, BFS, Dijkstra's, or A*—each one can generate its own events while reusing the same visualization logic.

5.3.3 Minimax Visualization

Our minimax visualization relies on building a tree of potential moves, called the *candidateTree*. Each node in this tree represents a specific board position, while its children represent the moves that can arise from that position. To avoid immediately displaying every branch, these nodes are stored in a breadth-first-search queue (*bfsQueue*), which we reveal step by step, giving a clearer understanding of how the algorithm expands its search.

As the algorithm finds promising sequences, we also generate a list of directional markers or *arrows*, collected in an *arrowTraversalQueue*. Each arrow points from the origin square to the destination square of a move. By incrementing through *arrowTraversalQueue*, the board re-creates each move in sequence, illustrating how minimax (or alpha-beta) works through possible lines of play.

Whenever the visualization is updated, our React components use the current step in *arrowTraversalQueue* to determine which moves to highlight. This step-by-step animation, combined with incremental node expansion in the *candidateTree*, offers an interactive way to see how the algorithm seeks winning moves and prunes weaker options, ultimately guiding users toward a clear picture of how minimax arrives at its final decision.

5.3.4 Interactive Chess Gameplay

Our chess interface allows users to play against themselves or challenge an engine. We store the ongoing game within *gameRef*, which holds a *Chess.js* instance representing the

board state. Each time the player clicks a square, the *handleSquareClick* function checks whether the move is legal and updates both the board display and the *moveHistory*.

If an engine is chosen (either *Stockfish* or our custom *Yunfish*), the application determines whose turn it is and, if necessary, requests a move by sending the current *FEN* (Forsyth–Edwards Notation) to the selected engine. For *Stockfish*, we spawn a web worker pointing to */stockfish.js* and parse its *bestmove* line. For *Yunfish*, we send the user's last move to a backend endpoint, which replies with the engine's move. In either case, the chosen move is processed through *handleEngineMove* to make sure it follows the same rules as the user's moves (e.g., promotions or captures).

We also track time for both sides (white and black), plus any pieces captured along the way (*whiteCaptures* and *blackCaptures*). Whenever a check, stalemate, or checkmate condition arises, a modal pops up showing the result. By refreshing *moveHistory* and board position after each move, the interface remains synchronized, ensuring users always see the correct board state and remain engaged with the engine in real time.

6 Evaluation & Testing

6.1 Chess Testing

Assessing chess engine performance can be complex due to the many factors that influence the outcome. This project's engine builds on Sunfish 1, which provides a `quick-tests.sh` script for testing core scenarios such as checkmates, stalemates, and the Bratko–Kopec test.

The primary modification is an improved evaluation function 5.1.2, while the core search mechanism remains unchanged. Several methods were used to confirm that the new system functions correctly and to compare its strength against the original Sunfish engine:

1. **Quick Tests**: Running `quick-tests.sh` on both versions showed that each passed the same 7 out of 103 test positions. Since both engines rely heavily on CPU resources, the available hardware could not handle deeper or more extensive searches in most test cases, limiting overall success. Identical outcomes here indicate that the changes did not break original functionality.
2. **Head-to-Head Arena**: An `arena.py` tool was used to compare the updated engine with the original Sunfish under the UCI protocol. The tests allowed for different parameters, such as:
    - A fixed time of 3 seconds per move for both sides.
    - A fixed search depth of 1 for both sides.

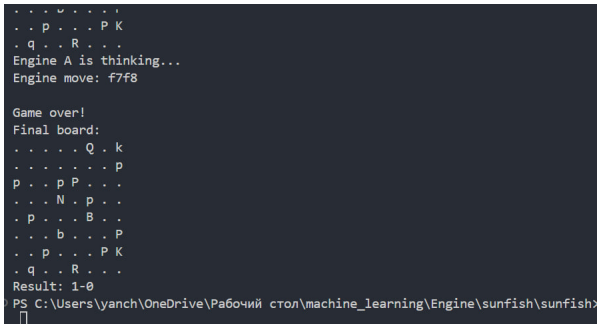
- A fixed search depth of 2 for both sides.

In most matchups, the new engine achieved stronger results, suggesting that a more accurate evaluation often outweighs any potential speed disadvantage.

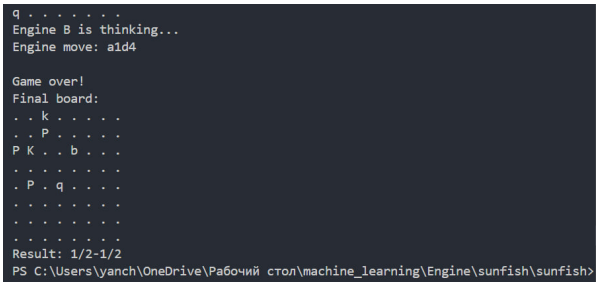
6.1.1 Performance Considerations

Although the improved evaluation can be two to three times slower than the original method, it often achieves higher success rates under typical time or depth constraints. Because these engines rely heavily on CPU resources, hardware limitations may also restrict deep searching or advanced analysis. Nevertheless, tests suggest that a stronger evaluation can outperform a faster yet less accurate approach. Granting extra time to the original Sunfish may allow it to win some games, but the updated system generally prevails in most scenarios.

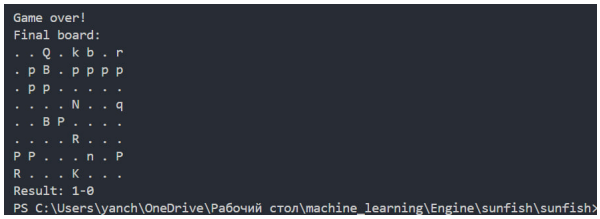
6.1.2 Screenshots and Observations



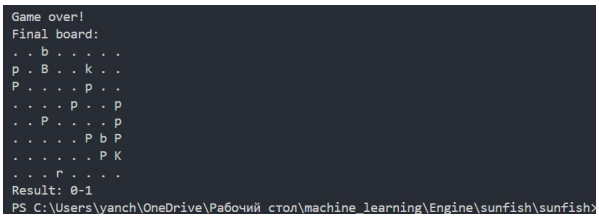
(a) Updated engine (White) wins at 3 s/move.



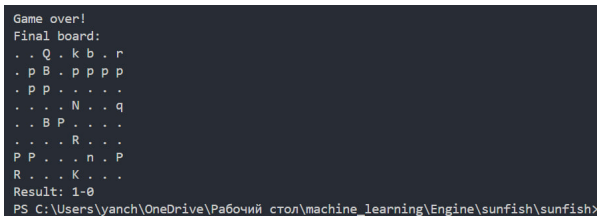
(b) Updated engine (Black) draws at 3 s/move.



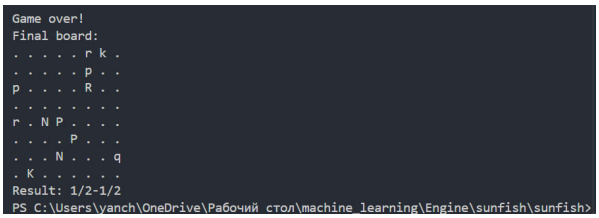
(c) Updated engine (White) wins at depth 1.



(d) Updated engine (Black) wins at depth 1.



(e) Updated engine (White) at fixed depth 2.



(f) Updated engine (Black) draws at fixed depth 2.

Figure 13: Six head-to-head match examples between the updated engine and the original Sunfish (two subfigures per row).

6.1.3 Extended Match Results

A series of **20 head-to-head games** were played at a move time of **3 seconds or more** to further examine the updated engine's performance with longer time controls. Table 1 summarizes the outcomes:

Engine	Wins	Win %
Updated Engine	13	65%
Original Sunfish	3	15%
Draws	4	20%

Table 1: Results from 20 games at 3+ seconds per move.

With more time to analyze each position, the updated engine leverages its improved evaluation more effectively, winning 65% of the games against the original Sunfish. Although Sunfish secured 15% of the wins, the updated approach maintained an advantage in the majority of positions. This outcome suggests that, at slower time controls, the benefit of a stronger evaluation outweighs raw speed, as both engines have enough time to search multiple lines in depth.

6.2 User Feedback

User feedback was gathered through a series of informal interviews conducted both in person and via voice calls. These discussions aimed to assess how easily participants could navigate the interface and whether they found the articles enjoyable to read.

From these interviews, three main themes emerged:

- **Navigation Challenges:** A majority of participants reported difficulty navigating the interface, which led to revisions in the final UI design.
- **Complex Content:** Several users felt the articles were difficult to read, prompting further UI refinements to improve clarity and presentation.

- **Positive Responses:** Approximately 30% of respondents expressed overall satisfaction with the interface, noting no major concerns.

These findings influenced multiple aspects of the final version, ensuring that both usability and readability received careful attention.

6.3 Visualization

6.3.1 Maze Visualization Evaluation

The maze visualization was tested with grid sizes ranging from 20×20 to 40×40. In each configuration, the interface successfully displayed the incremental steps of the selected pathfinding algorithm (*e.g.*, BFS, DFS, Dijkstra, or A*) and highlighted the final path from the start to the goal cell.

For smaller mazes (20×20), computations finished almost instantly, allowing the animation to play smoothly. As the maze size increased to 40×40 or more, a slight delay became noticeable, reflecting the added computational load. Nevertheless, users reported the visualization remained helpful in understanding how each algorithm expands and prioritizes cells during the search. No instances of crashes or incorrect pathfinding outcomes were observed, indicating a robust implementation for typical use cases.

6.3.2 Minimax Traversal Evaluation

The minimax traversal was evaluated using a series of chess positions with varying complexity. In simpler puzzles featuring only a few possible moves per position, the algorithm performed quickly and accurately, often finding the optimal move sequence within a second.

However, in more complex scenarios with a larger branching factor (*e.g.*, mid-game positions containing multiple pieces and move options), the exponential growth of the search tree led to noticeable slowdowns at greater depths. To address this, **alpha–beta pruning** was integrated into the minimax routine, significantly reducing the total nodes evaluated. In tests at a depth of 3 or more, the alpha–beta version evaluated up to 35% fewer nodes compared to the naive minimax, speeding up move discovery.

7 Discussion

7.1 Hypothesis

The main goal of this project was to create a software tool that addresses the four questions listed in Section 1.3. Each point is reviewed here with an overview of how well it was

achieved:

(a) *Does adding an NNUE-style network to the Sunfish engine lead to a significant improvement in playing strength, compared to the original heuristic-based method?*

(b) *Can creating a more advanced maze visualization tool boost clarity, user engagement, and feature completeness beyond what is currently offered?*

(c) *"Could we develop a new tool that interactively visualizes minimax, making the algorithm more understandable?"*

(d) *Is it feasible and beneficial to combine NNUE, enhanced maze visualization, and a minimax visualizer into one platform, offering a unified set of features to users?*

Improvement in Chess Engine Strength (Question (a)).
Tests showed that adding an NNUE-style network to the Sunfish engine did indeed increase its playing strength. Multiple matches confirmed that the NNUE-enhanced engine outperforms the original version. This outcome directly answers question (a) with a clear "yes," as the improved evaluation led to more accurate move choices and better overall results.

Maze Visualization Enhancements (Question (b)).
The new maze visualization tool includes both random generation and custom options, which fills the gap mentioned in the literature review. Early versions received mixed feedback, but subsequent improvements made the interface more intuitive and comprehensive, thus addressing question (b). The tool now allows users to explore different maze layouts, track benchmarks, and visually follow the paths generated by the system.

Minimax Visualizer (Question (c)).
The project also introduced a minimax visualizer that operates independently of the NNUE engine. In its current form, it utilizes a simple engine to illustrate how search trees and evaluations are generated in real time. This approach demonstrates that the visualizer can be adapted to various algorithms or engines, thus confirming that a real-time minimax view is both feasible and helpful for understanding decision-making processes.

Combined Platform (Question (d)).

Finally, the updated user interface merges NNUE-based chess analysis, maze visualization, and the new minimax view into one platform. Although initial UI designs faced some negative feedback regarding layout and ease of use, the latest version demonstrates a more unified approach and better user experience. Therefore, question (d) has largely been met, showing it is both possible and beneficial to offer all these features together in a single tool.

7.2 Future Work

Although this platform offers several useful features, there are many ways to improve it further:

- **Engine Performance:** Currently, the engine uses a halfkp architecture. The original plan was to implement a halfkv2 architecture, but it was not successful. It remains unclear whether the issue lies in the code or the weights themselves. Investigating this problem and moving to halfkv2 in the future could significantly boost engine strength.
- **Maze Input and Analysis:** In the maze-solving component, allowing users to upload images that an AI agent converts into mazes could be very helpful, especially for individuals working on robotics or pathfinding research. This feature would allow greater flexibility and more real-world applications.
- **Minimax Visualizer Integration:** At present, the minimax visualizer uses a simple engine to demonstrate how search trees and evaluations are generated. Integrating this visualizer with a more advanced engine (such as the NNUE-enhanced version) would give users deeper insight into the decision-making process during actual gameplay.
- **Platform Expansion:** While the current platform supports essential features, there is potential to extend its capabilities even further. Future updates might include more comprehensive visualizations for both maze generation and the chess engine. This could involve showing additional metrics, more interactive controls, or support for other AI techniques, allowing the platform to grow into a fully featured research and learning tool.

By pursuing these directions, the platform can continue to evolve, serving as both a more powerful chess engine and a more flexible maze and decision-making visualization environment.

7.3 Conclusion

This project combined three important components: a stronger chess engine, an improved maze visualization tool, and a minimax visualizer. The chess engine was enhanced by incorporating a neural network (NNUE-style), leading to better overall play. The maze tool allows users to generate random or custom layouts, simplifying the study of various paths and designs. Meanwhile, the minimax visualizer displays the engine's decision-making process step by step, giving a clear look into how the search works. Having these features all in one platform offers a more engaging learning environment for both chess and pathfinding. Future goals include further strengthening the chess engine, introducing

additional maze customization options, and integrating the minimax visualizer with the advanced engine to deliver an even more comprehensive experience.

References

[1] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[2] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985, pp. 1034–1036.

[3] M. Campbell, A. J. H. Jr., and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[4] C. E. Shannon, "Programming a computer for playing chess," Philosophical Magazine, 41(314), pp.256–275, 1950.

[5] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[6] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games*, 2006, pp. 72–83.

[7] D. J. Slate and L. R. Atkin, "Chess 4.5 - the northwestern university chess program," Tech. Report, Northwestern University, 1977.

[8] D. N. L. Levy and M. Newborn, *How Computers Play Chess*. W. H. Freeman and Company, 1991.

[9] Y. Nasu, "Nnue: Efficiently updatable neural network for shogi," Presentation, CSA Shogi forum, 2018.

[10] S. Community, "Stockfish chess engine," https://stockfishchess.org/, 2023, [Online; accessed May 2023].

[11] J. K. Månsson, "Sunfish chess engine," https://github.com/thomasahle/sunfish, 2015, [Online; accessed May 2023].

[12] N. Fiekas, "python-chess: A pure Python chess library," https://python-chess.readthedocs.io/, 2023, [Online; accessed May 2023].

[13] S. Contributors, "Stockfish with nnue (neural network-based universal evaluator)," https://github.com/official-stockfish/Stockfish, 2020, [Online; accessed May 2023].

[14] S. Developers, "Nnue integration in stockfish," GitHub Pull Request, Stockfish engine repository, 2020.

[15] E. Author(s), "Nnue implementation in halfkp," https://github.com/official-stockfish/nnue-pytorch/blob/master/docs/nnue.md, accessed: YYYY-MM-DD.

[16] "Stockfish: Official documentation," https://stockfishchess.org/, accessed: 2025-04-10.

[17] "Lichess engine integration guide," https://lichess.org/blog/XyzEngineIntegration, accessed: 2025-04-10.

[18] "Arena chess gui documentation," http://www.playwitharena.de/, accessed: 2025-04-10.

[19] "Cute chess: Official docs," https://github.com/cutechess/cutechess, accessed: 2025-04-10.

[20] "Scid: Shane's chess information database," http://scid.sourceforge.net/, accessed: 2025-04-10.

[21] "Maze Solver by ESStudio," https://esstudio.site/maze-solver/, accessed: 2025-04-10.

[22] "Pathfindout," https://pathfindout.com/, accessed: 2025-04-10.

[23] A. or Forum Handle, "Discussion on the importance of standardized pathfinding benchmarks," https://some-forum-or-article.com, accessed: 2025-04-10.

[24] "Alpha-beta pruning: A search algorithm tutorial," https://www.example.com/alpha-beta-tutorial, accessed: 2025-04-10.

[25] "Ecmascript language specification," https://262.ecma-international.org/, accessed: 2025-04-11.

[26] "World wide web consortium (w3c)," https://www.w3.org/, accessed: 2025-04-11.

[27] "Mdn web docs," https://developer.mozilla.org/, accessed: 2025-04-11.

[28] "Webpack official documentation," https://webpack.js.org/, accessed: 2025-04-11.

[29] "Babel official documentation," https://babeljs.io/, accessed: 2025-04-11.

[30] "npm documentation," https://docs.npmjs.com/, accessed: 2025-04-11.

[31] "Yarn documentation," https://yarnpkg.com/, accessed: 2025-04-11.

[32] "History of javascript," https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript, accessed: 2025-04-11.

[33] K. Simpson, "You Don't Know JS: Async & Performance," https://github.com/getify/You-Dont-Know-JS, accessed: 2025-04-11.

[34] "Node.js documentation," https://nodejs.org/en/docs, accessed: 2025-04-11.

[35] "React official documentation," https://react.dev/, accessed: 2025-04-11.

[36] "Using the state hook - react official docs," https://react.dev/reference/react/useState, accessed: 2025-04-11.

[37] "Tailwind css official documentation," https://tailwindcss.com/, accessed: 2025-04-11.

[38] "Chess.js github repository," https://github.com/jhlywa/chess.js, accessed: 2025-04-11.

[39] "React flow official documentation," https://reactflow.dev/, accessed: 2025-04-11.

[40] "Fastapi official documentation," https://fastapi.tiangolo.com/, accessed: 2025-04-11.

[41] "Uvicorn official documentation," https://www.uvicorn.org/, accessed: 2025-04-11.

[42] "Pytorch official documentation," https://pytorch.org/, accessed: 2025-04-11.

[43] C. R. Harris, K. J. Millman, S. J. van der Walt, and et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[44] Python Software Foundation, "Python documentation: collections — container datatypes," https://docs.python.org/3/library/collections.html, accessed: 15-Apr-2025.

[45] "Docker documentation," https://docs.docker.com/, accessed: 2025-04-11.

[46] L. Dombrowski, "Layout strategies for text clarity and retention," in *Proceedings of the 2008 Conference on User Experience*, 2008.

[47] J. Nielsen, *Prioritizing Web Usability*. New Riders Press, 2006.

[48] K. Tsumura, J. Shirogane, H. Iwata, and Y. Fukazawa, "Layout-driven webpage reading," in *eSociety Conference*, Tokyo, Japan, 2022.

A Appendix

This appendix presents additional screenshots from the application's user interface. Each figure has a short caption, and the paragraphs below highlight key features or functionalities shown in each image.

Figure 14 shows the *main homepage layout*, including a welcome message and a navigation bar for exploring different features.

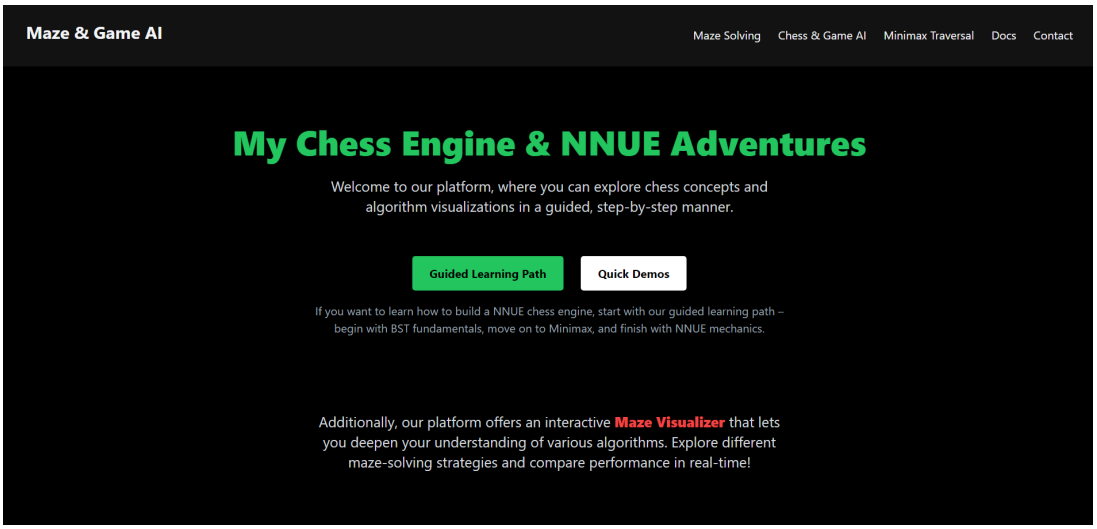


Figure 14: Main interface showing the navigation bar and welcome message (part of the `mainpage` component).

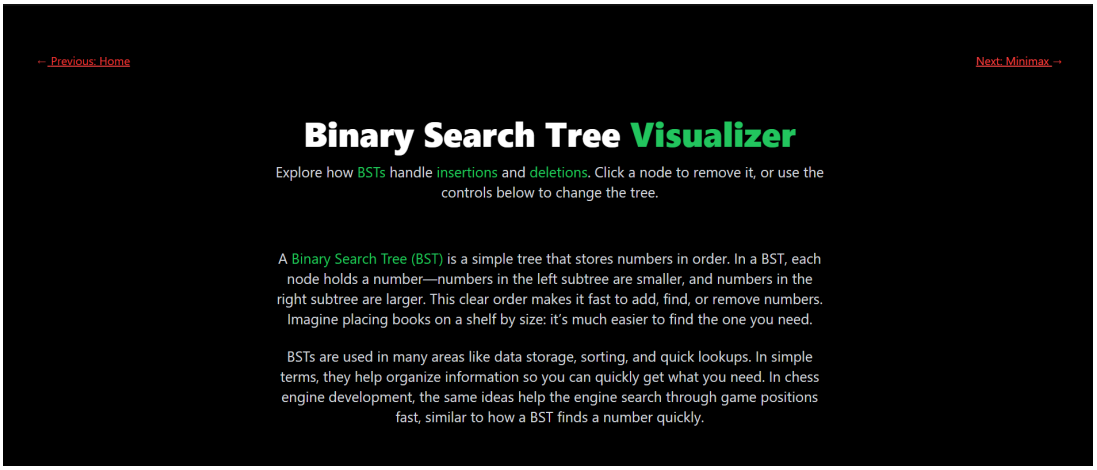


Figure 15: Guided path for users interested in building an NNUE (Neural Network Updatable Engine).

Figure 15 illustrates part of a guided path where beginners can follow step-by-step instructions to construct an NNUE-based chess engine.

Figures 16 and 17 capture other guided pages. The first focuses on step-by-step minimax explanations, while the second covers building an NNUE engine in greater depth.

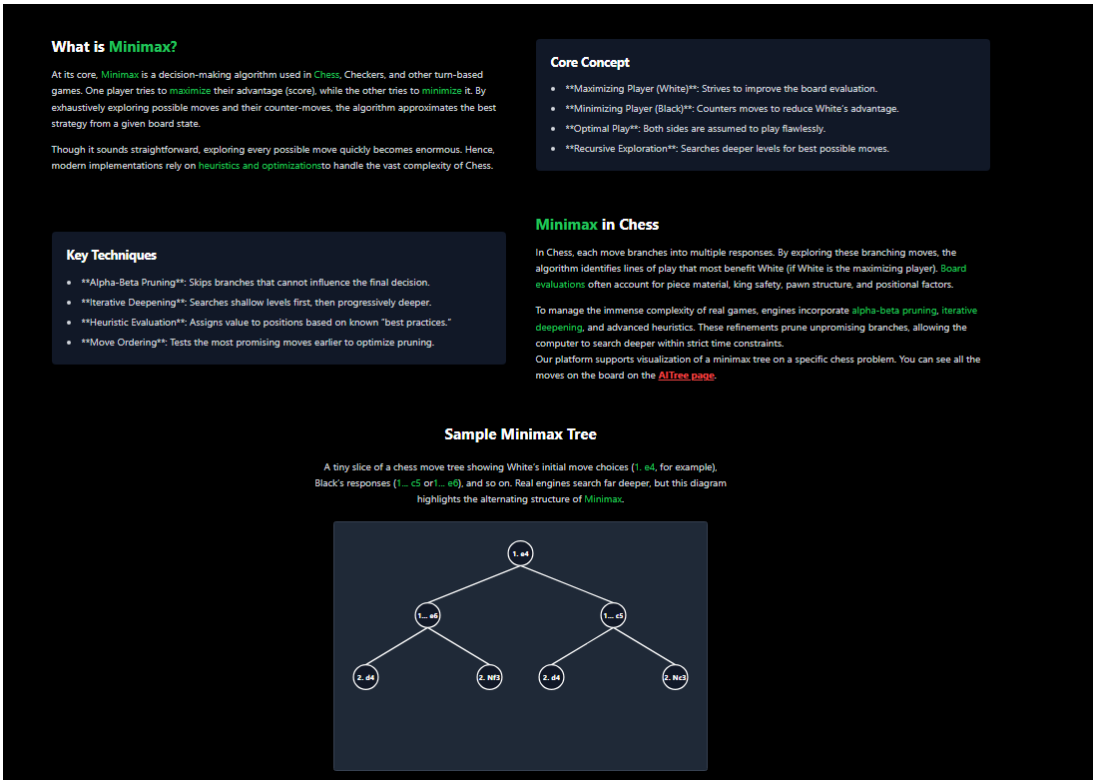


Figure 16: Minimax page from a guided path.

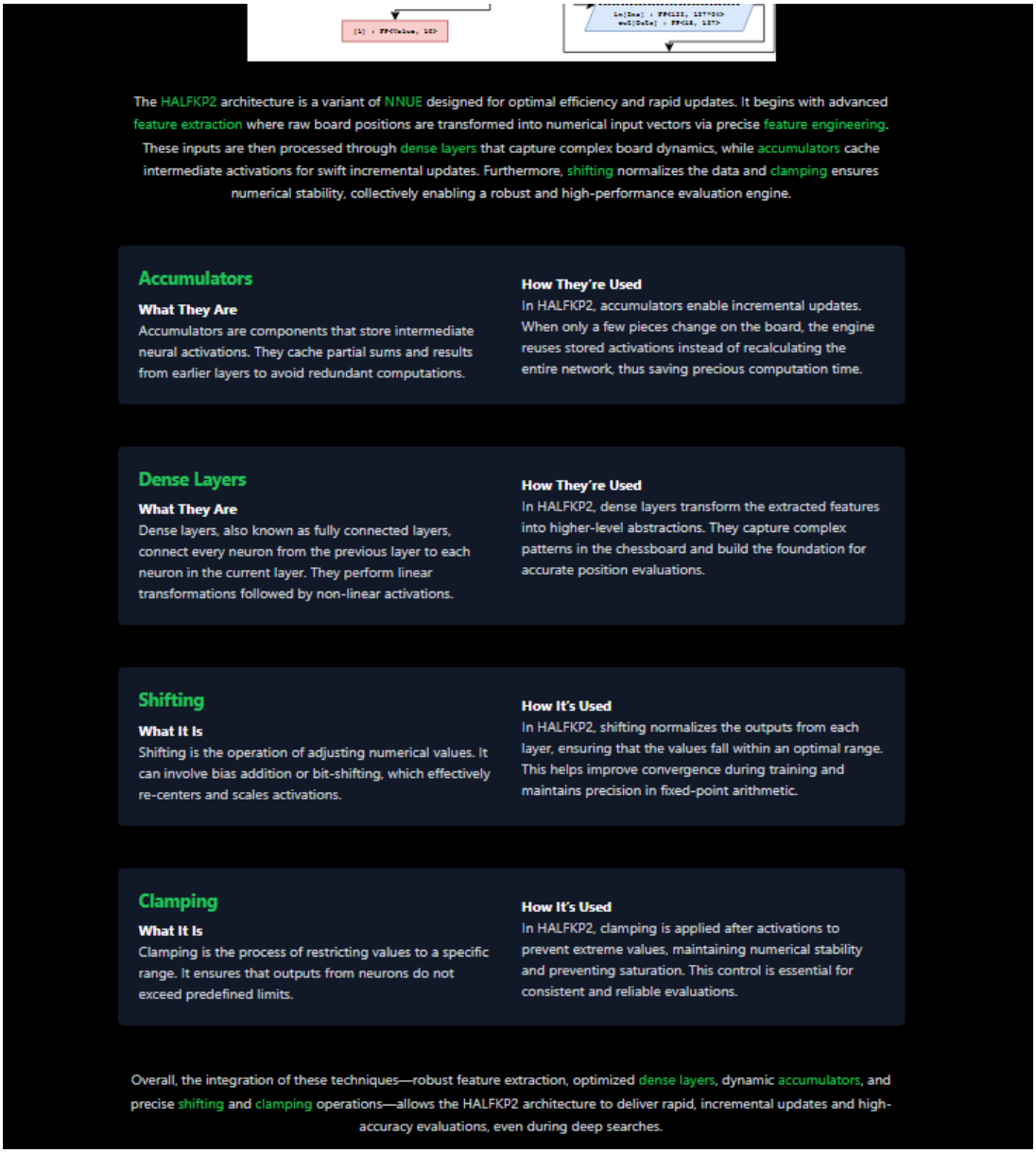


Figure 17: NNUE page from a guided path (fully zoomed out).

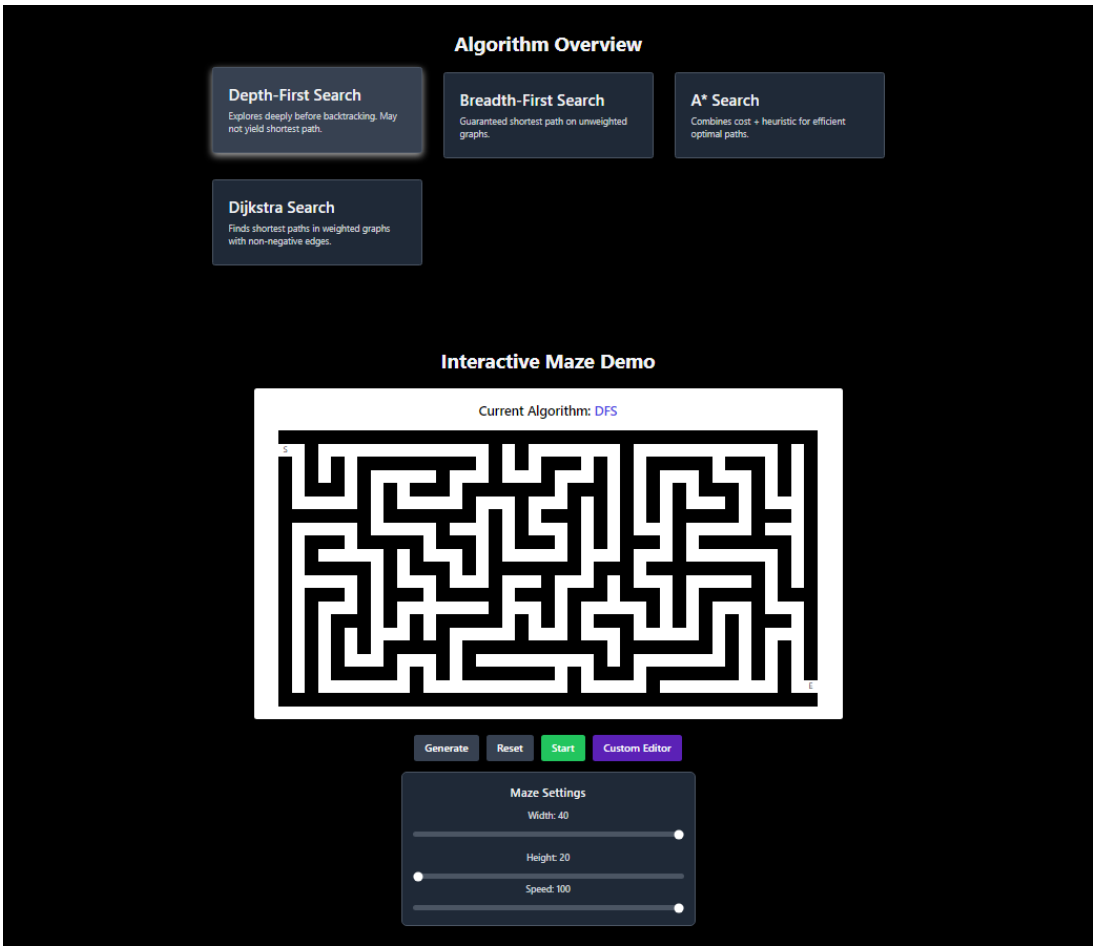


Figure 18: Top portion of the `mazesolving` component, showing algorithm selection, maze area, and a control panel with settings.

Figure 18 displays the upper section of the `mazesolving` component, including controls or parameters for selecting various pathfinding algorithms. Figure 19 shows how users can manually construct maze walls and pathways to explore unique pathfinding scenarios.

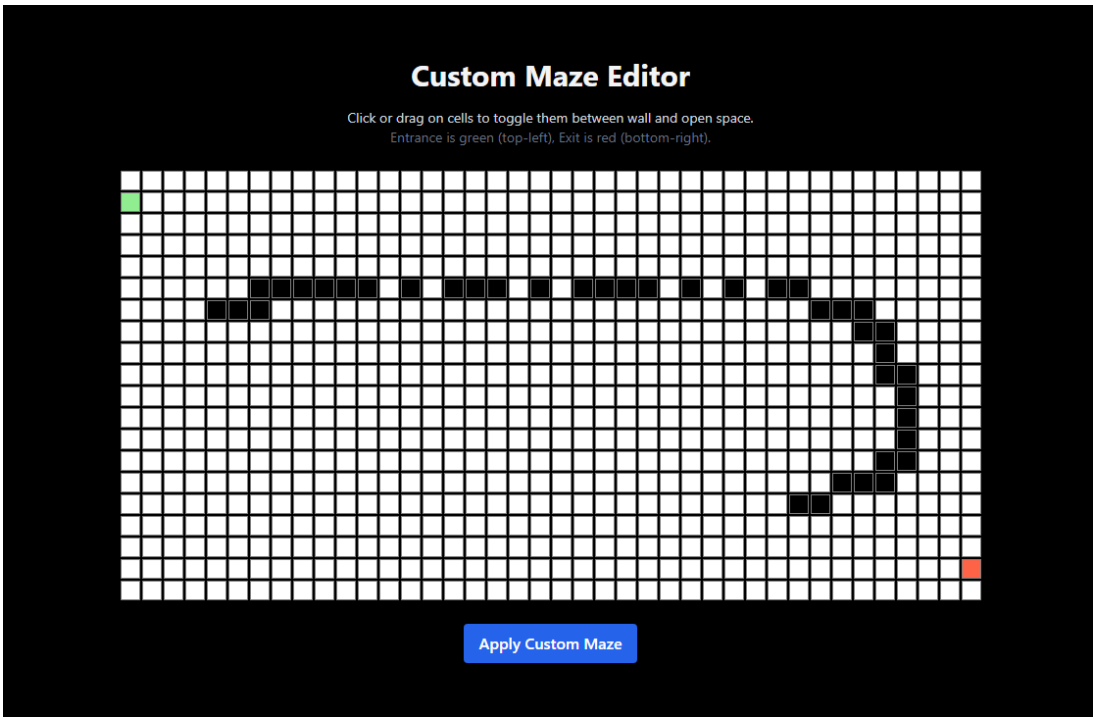


Figure 19: Section of the `mazesolving` component for building a custom maze.

Figure 20 shows benchmark results to help users compare the performance of different pathfinding methods under various conditions.

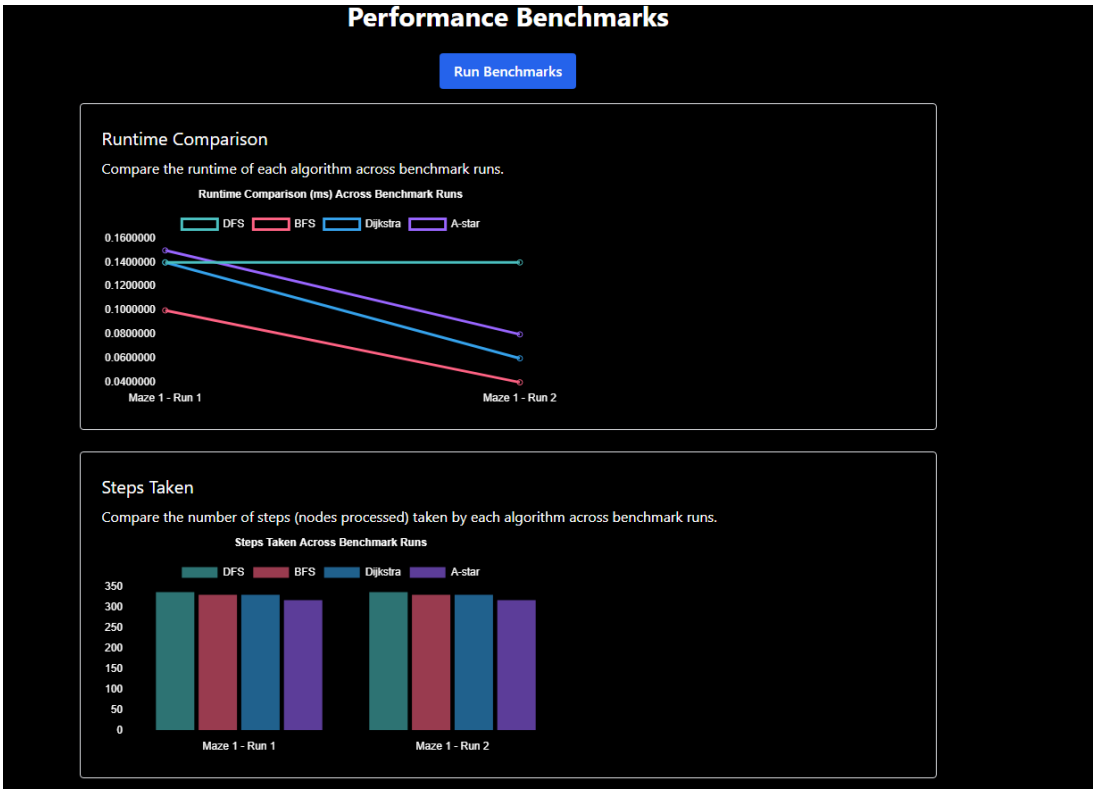


Figure 20: Benchmarks section within the `mazesolving` component.

Figure 21 depicts the `chessgame` interface, allowing users to switch between Stockfish or the custom NNUE-based engine, review a move history, manage time controls, and more.

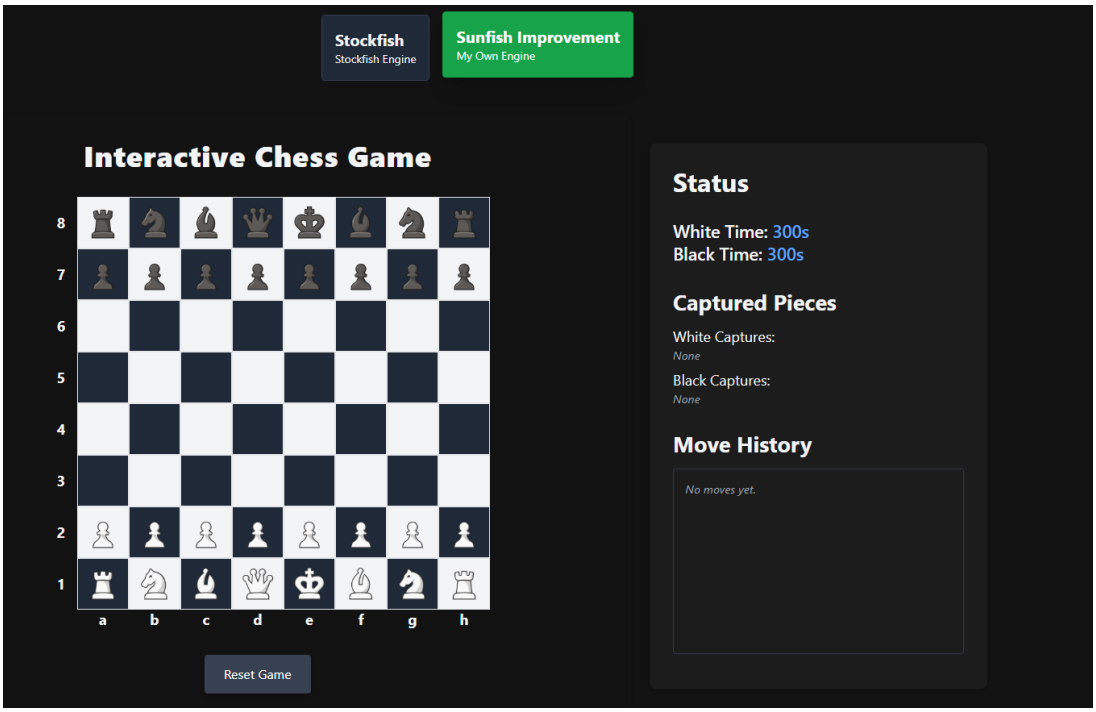


Figure 21: `chessgame` component where users can play against the updated engine or Stockfish.

Figure 22 shows how the `minimaxsolver` begins with a default chess position ready to be analyzed or changed by the user.

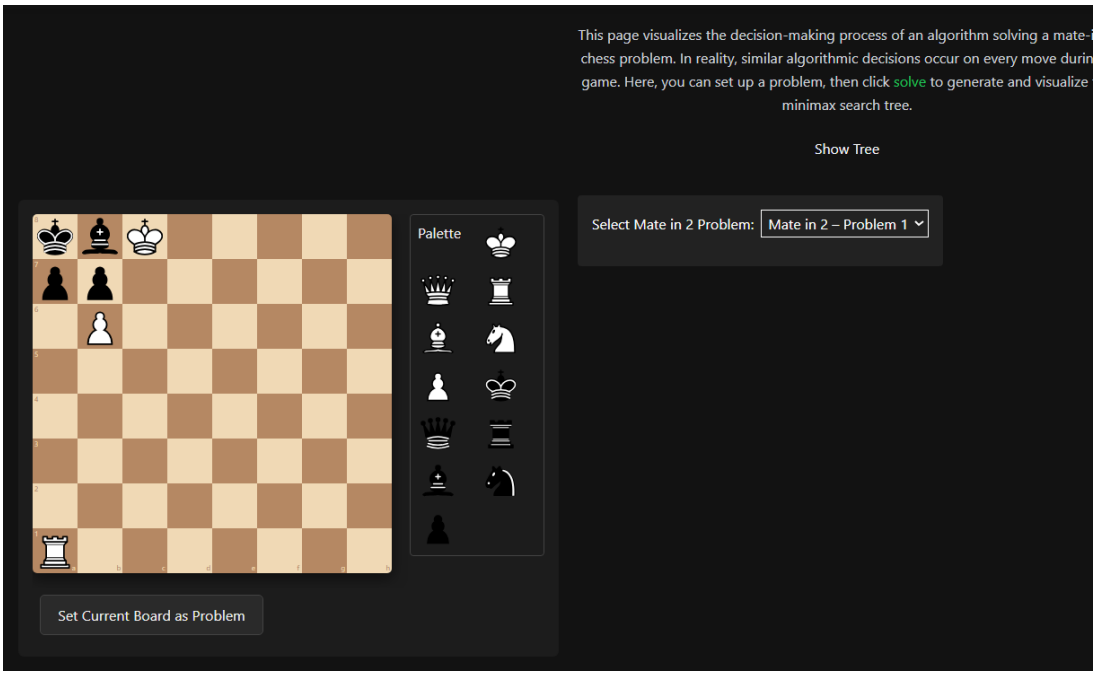


Figure 22: Initial state of the `minimaxsolver` component.

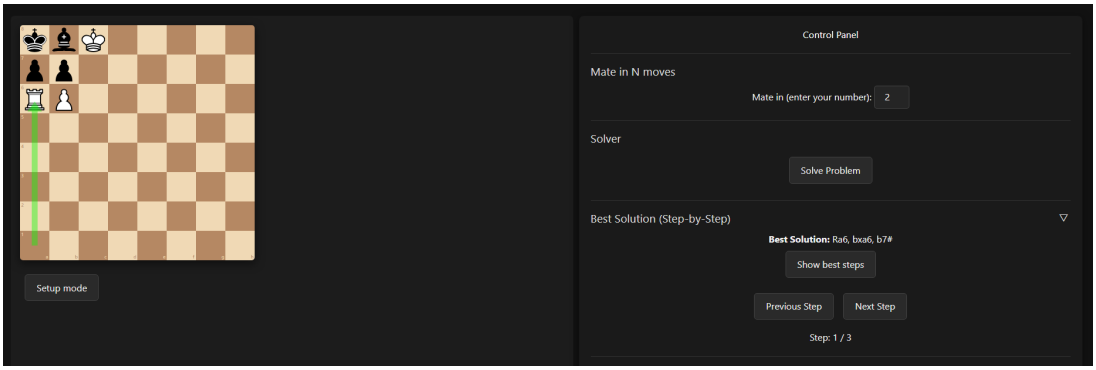


Figure 23: `minimaxsolver` component after a problem is set up and partially solved.

In Figure 23, the minimax algorithm is actively searching once the user defines a particular chess position or puzzle to solve.

Figure 24 depicts the state once minimax completes its search and outputs a final evaluation or sequence of moves.

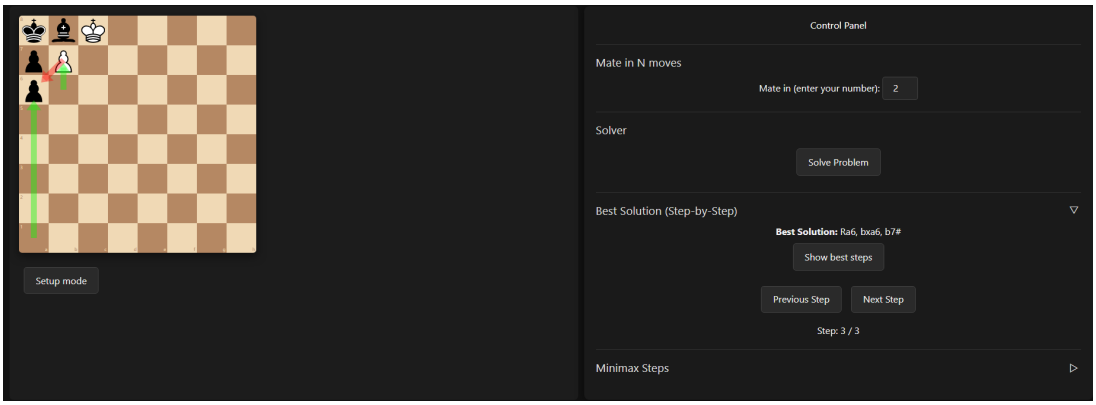


Figure 24: `minimaxsolver` component after the problem is fully traversed.

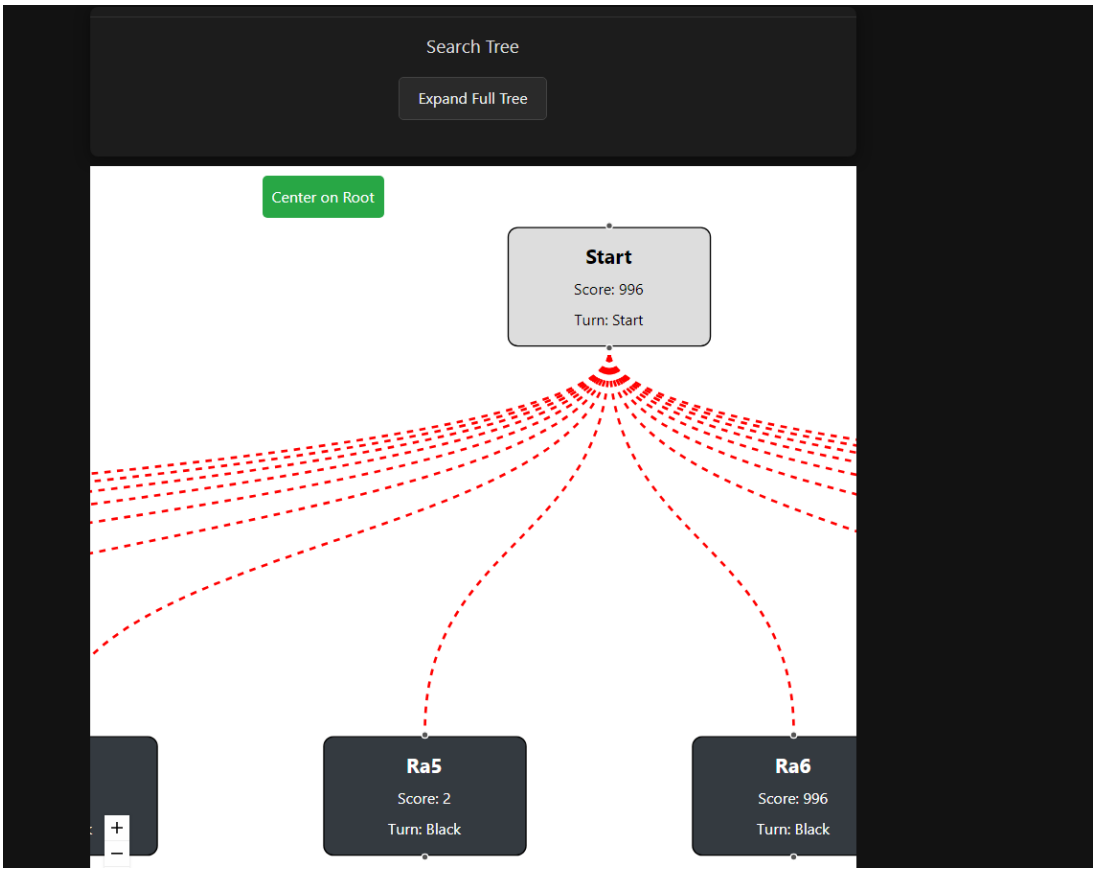


Figure 25: Example of the search tree generated by `minimaxsolver` for a given chess puzzle.

Figure 25 displays the expanded search tree where each node corresponds to a possible move or outcome during minimax exploration.

Finally, Figure 26 shows a zoomed-out perspective of the search tree, highlighting how nodes can be expanded or collapsed for closer inspection.

Figure 26: An expanded, zoomed-out view of the `minimaxsolver` tree.