# QUESTION’S IMPLEMENTATION

# 1. System Architecture

The application follows a modular architecture, separating the User Interface (UI) from the Logic Core.

## 1.1. Technology Stack

- **Frontend/Controller:** Python 3.12 + **Streamlit**. chosen for its efficiency in creating data-driven web interfaces.
- **Computational Core (Backend): C++**. Used for implementing the heavy AI algorithms (N-Queens, Hanoi, Minimax, etc.) to ensure high performance and precise solution verification.
- **Data Storage:**
    - data/cpp/questions.json: Stores the templates and parameters for questions.
    - data/submissions.xlsx: Logs all user attempts and scores for evaluation.
- **Integration:** The Python subprocess library acts as a bridge, compiling and executing C++ binaries on demand based on the user's selected question.

## 1.2. Directory Structure

- app.py: Main entry point. Handles the Streamlit UI, session state, and C++ execution logic.
- data/cpp/: Contains the source code (.cpp) for all AI algorithms.
- requirements.txt: Python dependencies.

---

# 2. Implementation of Question Types

The core innovation of SmarTest is that answers are **calculated**, not stored. Below is the implementation strategy for the mandatory question types:

## 2.1. Search Problem Identification

- **Objective:** Determine the most suitable search strategy for a given problem instance (e.g., N-Queens, Knight's Tour).
- **Implementation:** We implemented comparative scripts (e.g., Compilado_Knight.cpp, N-queens_N*.cpp). These scripts run multiple algorithms (Backtracking, BFS, A*, Hill Climbing) against the same instance.
- **Logic:** The system measures execution time and node expansion. The "correct answer" is dynamically determined by identifying the algorithm with the best performance metric (e.g., finding a solution for Knight's Tour N=30 using Hill Climbing where BFS fails).

## 2.2. Constraint Satisfaction Problems (CSP)

- **Objective:** Solve variable assignment problems using Backtracking with optimizations.
- **Implementation:** Files such as CSP_Logic_MRV.cpp and graphcoloring*.cpp implement the constraint graph.
- **Logic:** The C++ code models variables, domains, and binary constraints. It implements **Forward Checking** and the **Minimum Remaining Values (MRV)** heuristic to prune the search tree. When a user submits an answer (e.g., A=1, B=2), the system runs the C++ solver to verify if that assignment is valid within the constraint graph.

### 2.3. Game Theory (Normal Form)

- **Objective:** Identify Nash Equilibria in a payoff matrix.
- **Implementation:** Standard matrix analysis algorithms in C++.
- **Logic:** The code iterates through the payoff matrix (e.g., Stag Hunt, Prisoner's Dilemma). It checks for dominant strategies and best responses for both players. If a cell $(r, c)$ represents a best response for Player A given Player B's choice, and vice-versa, it is flagged as a Nash Equilibrium.

### 2.4. Adversarial Search (Minimax)

- **Objective:** Calculate the root value and identify pruned nodes in a game tree.
- **Implementation:** A recursive Minimax algorithm with **Alpha-Beta Pruning**.
- **Logic:** The system constructs a tree based on the problem instance (branching factor and depth). It traverses the tree, updating alpha and beta values. The C++ program outputs not just the final score, but also a log of which specific leaf nodes were skipped (pruned) during the process, allowing for precise validation of the user's understanding of the pruning mechanism.

## 3. Team Contributions

The project was developed collaboratively using an Agile methodology. Tasks were distributed to balance the workload between architectural design, algorithm implementation, and data management.

| Member | Role | Key Contributions |
|---|---|---|
| **Víctor** | **Lead Architect & Frontend** | Design of the Streamlit interface (app.py). Implementation of the Python-C++ bridge (subprocess logic). Handling of UTF-8 encoding for output parsing and UI state management. |
| **Daniel** | **Search Algorithms** | Implementation of Search Problem algorithms (Knight's Tour, N-Queens). Developed the comparative logic (BFS vs A* vs Hill Climbing) and benchmarking metrics in C++. |
| **Javier Peñalver** | **CSP & Logic Implementation** | Focused on Constraint Satisfaction Problems. Coded CSP_Logic_MRV.cpp and Graph Coloring solvers. Implemented the MRV and Forward Checking heuristics. |

| | | |
|---|---|---|
| **Javier Sánchez** | **Game Theory & Data Structure** | Structured the questions.json database. Implemented Game Theory algorithms (Nash Equilibrium detection) and ensured correct parsing of matrix inputs. |
| **Diego** | **Adversarial Search & QA** | Implementation of Minimax and Alpha-Beta pruning logic in C++. Developed the Logging System (submissions.xlsx) and performed Quality Assurance (debugging simulation errors). |

# 4. Interactions with Conversational Agents

As per project requirements, we utilized AI assistants (Gemini, ChatGPT) to accelerate development.

- **Usage**: Agents were used to generate boilerplate code for standard algorithms (e.g., "Generate a C++ template for N-Queens"), debug compilation errors (e.g., fixing UnicodeDecodeError in Python), and optimize CSS for Streamlit.
- **Documentation**: All prompts and responses are compiled in the attached file Chat_Documentation_AI.pdf.