Automated Debugging system using Multi-Agent Architecture

s330457 - Noman Rafiq
s328433 - Kaan Sadik Aslan
s321940 - Antony Davi
s342320 - Javaria Babar

Efficient Approach for code Optimization : https://github.com/k0raty/IBEC_hackhathon/

Purpose of system

This system automates the debugging process, saving developers time effort and human error

Enhance code quality by identifying and fixing issues systematically.

An automated system ensures consistency, efficiency ,and scalability.

Core components of the debugging system

- **Syntax checker agent**(Detect and report syntax error)
- **Logic Error Agent**(identifies logical flaws)
- **Performance optimizer Agent**(Flags inefficient code)
- **Integration Agent**(Applies fixes back into the code)

Graph Theory and Nash Equilibrium

- Graph theory helps model complex code structures and dependencies between functions and modules in the system.
- Nash equilibrium refers to the scenario where no participant can benefit by changing strategies. This can be applies to optimize code corrections and decision in AI agent.
- Example: When two optimization approaches are competing for the same resources Nash equilibrium help identify the best balance.

Market Analysis

Market Overview

Growing demand for AI-assisted tools to speed up software development and reduce human error.

Competitive Landscape

Your AI tools with other Debugging tools(Static analyzer,IDE-based debuggers)

Target Audience

Software developers large tech companies

Simplified case : Word correction via multi-agent communication

Main Components:

- `debugger_agent.py`: Proposes permutations.
- `evaluator_agent.py`: Scores permutations.
- `plotting.py`: Visualizes progress.

Requirements: Python 3.x, `matplotlib`, `networkx`.

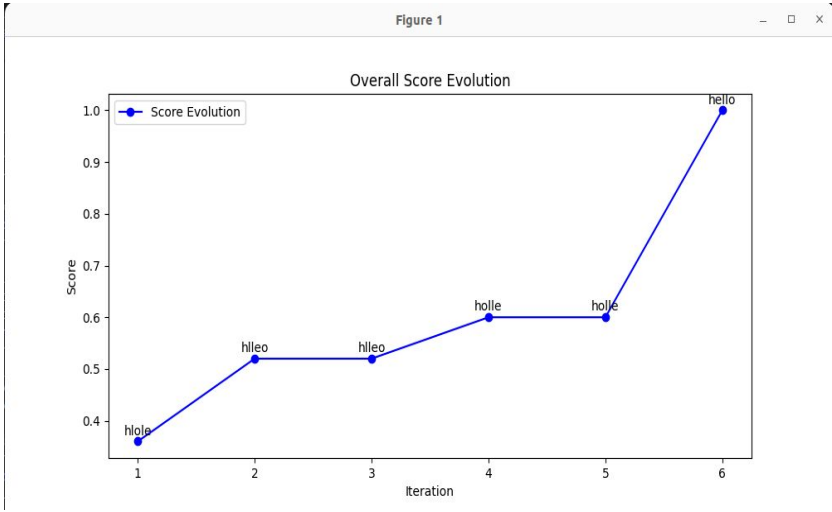
Overview

- Implements a **reinforcement learning (RL)**-based multi-agent system.
- **Debugger agents** propose letter permutations of a word.
- **Evaluator agent** scores permutations by similarity and positional distance.
- Goal: Achieve **Nash equilibrium** where the word is correctly spelled.

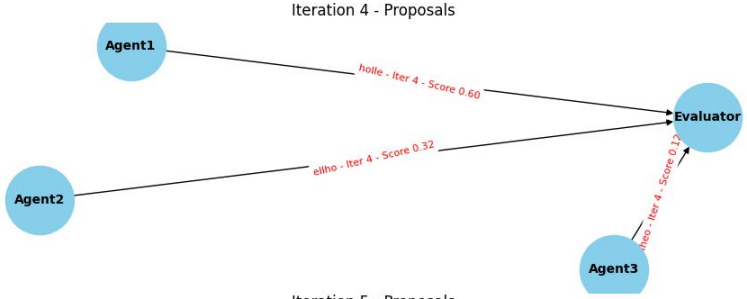
```
WORD_prototype/
├── debugger_agent.py
├── evaluator_agent.py
├── main.py
├── plotting.py
└── README.md
```

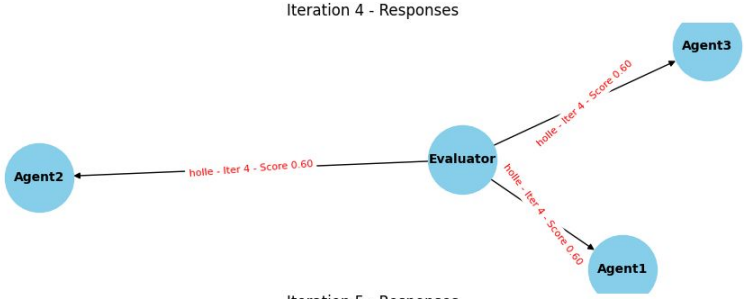
Simplified case : Results

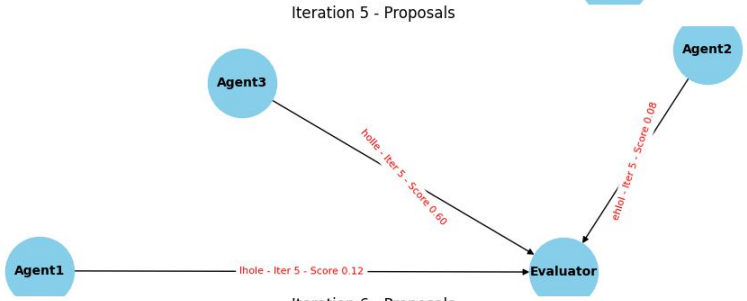
- At each iteration , each agents propose a modified word
- The word with the best scored is kept
- Then each agents from this word starts again the process.
- We loop until equilibrium (score = 1)
- Code working in one day see associated files.

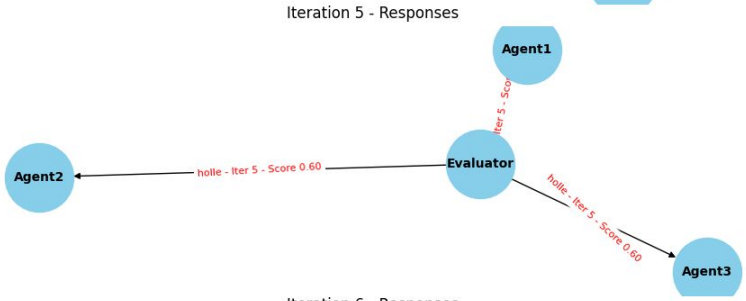


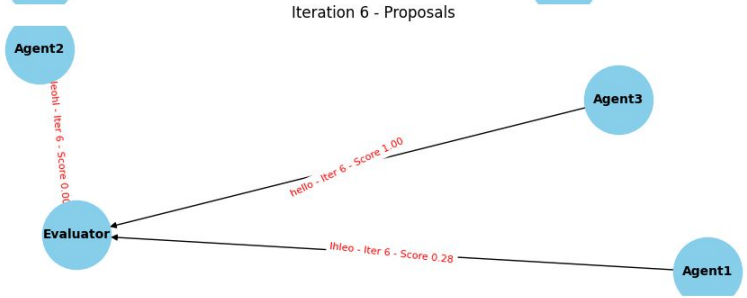
Communication process between agents

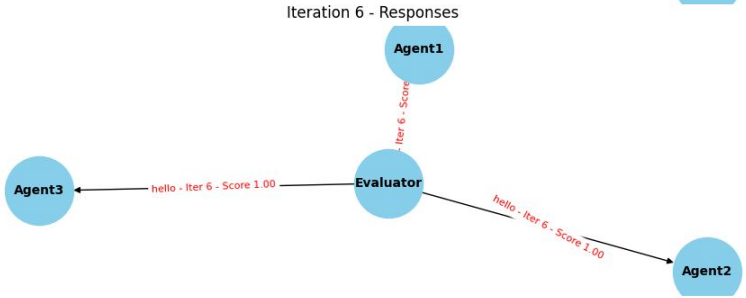












Comparing the two cases

Key Features

- **Game Theory Framework**: Agents compete to propose the best permutation.
- **Exploration Mechanism**: Avoids local optima by accepting suboptimal solutions probabilistically.
- **Visualization**: Tracks score evolution and agent interactions with communication graphs.
- We can adapt this algorithm : instead of a WORD , WE MODIFY CODE

```
Final_Prototype/
── Compiler_Agent.py
   ├── DebuggerAgent.py
   ├── Integrator_Agent.py
   ├── LICENSE
   ├── main.py
   ├── Optimizer_Agent.py
   ├── plotting.py
   ├── README.md
   └── sample_code.py
```

Prototype of Final Case :Autocorrective system

```
Multi-Agent System
│
├── DebuggerAgent
│   └── Proposes corrections to the code
│
├── OptimizerAgent
│   └── Optimizes the code
│
├── CompilerAgent
│   └── Compiles the code to check for syntax errors
│
├── IntegratorAgent
│   └── Ensures the code is unique and not previously seen
│
├── Communication
│   └── Agents communicate to propose corrections,
optimizations, and compilations based on a score function.
│
├── Iterative Improvement
│   └── The system iteratively improves the code until a Nash
equilibrium is reached
```

—--- SCORE_function : Designed on matlab , evaluate the complexity and efficiency of the algorithm

Rules :

- Multi-based system agent
- DebuggerAgent and Optimizer Agents are IA

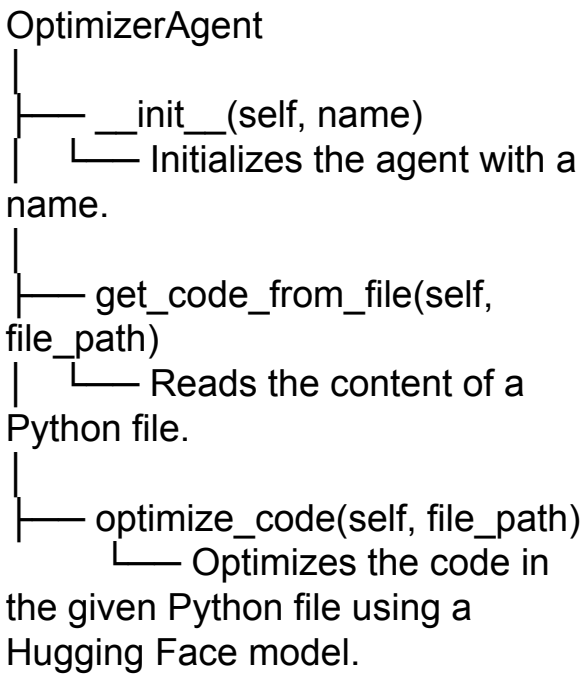
DebuggerAgent

- **Purpose:** Proposes corrections to the code using a Hugging Face model.
- **Methods:**
  - `__init__(self, name)`: Initializes the agent with a name.
  - `get_code_from_file(self, file_path)`: Reads the content of a Python file.
  - `debug_code(self, file_path)`: Debugs the code in the given Python file using a Hugging Face model.
- **Explanation:**
  - The `DebuggerAgent` class uses a Hugging Face model to propose corrections to the code. It reads the code from a file, sends it to the model for debugging, and returns the corrected code.

DebuggerAgent
│
├── __init__(self, name)
│ └── Initializes the agent with a name.
│
├── get_code_from_file(self, file_path)
│ └── Reads the content of a Python file.
│
├── debug_code(self, file_path)
└── Debugs the code in the given Python file using a Hugging Face model.

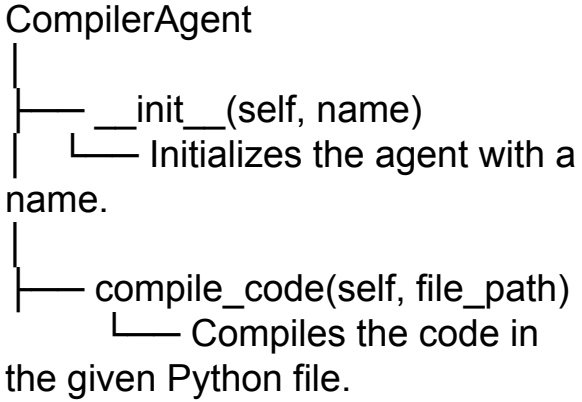
OptimizerAgent

- **Purpose:** Optimizes the code using a Hugging Face model.
- **Methods:**
  - `__init__(self, name)`: Initializes the agent with a name.
  - `get_code_from_file(self, file_path)`: Reads the content of a Python file.
  - `optimize_code(self, file_path)`: Optimizes the code in the given Python file using a Hugging Face model.
- **Explanation:**
  - The `OptimizerAgent` class uses a Hugging Face model to optimize the code. It reads the code from a file, sends it to the model for optimization, and returns the optimized code.



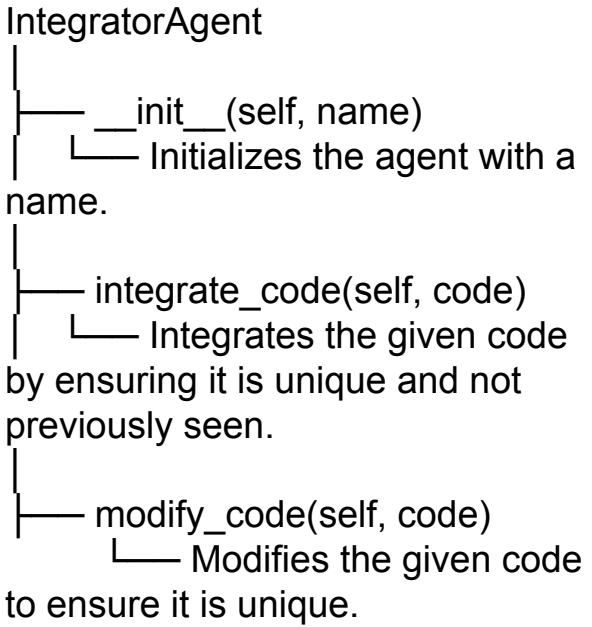
CompilerAgent

- **Purpose:** Compiles the code to check for syntax errors.
- **Methods:**
  - `__init__(self, name)`: Initializes the agent with a name.
  - `compile_code(self, file_path)`: Compiles the code in the given Python file.
- **Explanation:**
  - The `CompilerAgent` class compiles the code to check for syntax errors. It uses the `subprocess` module to run the Python file and capture the output. If the code compiles successfully, it returns `True`; otherwise, it returns `False`.



IntegratorAgent

- **Purpose:** Integrates the code by ensuring it is unique and not previously seen.
- **Methods:**
  - `__init__(self, name)`: Initializes the agent with a name.
  - `integrate_code(self, code)`: Integrates the given code by ensuring it is unique and not previously seen.
  - `modify_code(self, code)`: Modifies the given code to ensure it is unique.
- **Explanation:**
  - The `IntegratorAgent` class ensures that the code is unique and not previously seen. It keeps track of previously seen codes and modifies the code if necessary to ensure uniqueness.



Main function

- **Purpose:** Runs the code correction process using the multi-agent system.
- **Explanation:**
    - The main function initializes the agents, reads the code from a file, and iteratively proposes corrections, optimizations, and compilations. It keeps track of the scores and codes for plotting and visualizes the communication between agents.