University of Cyprus

MAI611 AI Fundamentals Assignment 2 – Solving Gogen Puzzles

Chrysis Andreou
(UC1366020)

Solving Gogen Puzzles Using Depth-First Search: A Constraint Satisfaction Approach

1. Introduction

Gogen puzzles present an engaging challenge where a set of words must be formed within a 5x5 grid containing a mix of pre-filled and empty cells. The objective is to strategically place the remaining letters in the grid such that all specified words are formed by traversing adjacent cells horizontally, vertically, or diagonally. This report details the implementation of a Python-based solver for Gogen puzzles, utilizing depth-first search (DFS) within a constraint satisfaction framework to efficiently navigate the solution space. The problem is modeled as a Constraint Satisfaction Problem (CSP), where the goal is to assign values to a set of variables under specific constraints. In the context of Gogen puzzles:

- **Variables**: The 16 empty cells on the grid that need to be filled with letters.
- **Domains**: The 16 remaining letters of the alphabet (excluding the initial nine and the letter Z).
- **Global Constraint**: All letters placed on the grid must be unique.
- **Local Constraint (Look-Ahead)**: Before placing a letter, the algorithm checks whether the current partial assignment still allows all words to be formed. If not, it backtracks immediately, reducing unnecessary exploration. By incorporating both global and local constraints, the solver efficiently navigates the search space, pruning infeasible paths early and preventing redundant state explorations. The depth-first search approach systematically explores potential letter placements, ensuring that all words can be successfully traced on the grid.

2. Overview of the Implementation

The solution is structured around several interconnected classes designed to model the puzzle state, manage the search process, and execute the depth-first search algorithm. The primary components include:

- `Search_State`: An abstract base class defining the interface for puzzle states.

- **`Search_Node`**: Represents nodes within the search tree, each holding a specific state.
- **`Search`**: A base class implementing generic search algorithms.
- **`Gogen_Search`**: Extends `Search` to handle Gogen-specific search logic.
- **`Gogen_State`**: Subclass of `Search_State` that encapsulates the state of the Gogen puzzle.
- **`Run_Gogen_Search`**: The driver class responsible for initializing the search and producing the final solution. The program operates by reading a list of words and initial letters, initializing the puzzle state, and then systematically exploring possible letter placements using DFS to find a configuration that satisfies all constraints.

3 Base Classes

`Search_State`

The `Search_State` class serves as an abstract foundation for representing different states within a search problem. It defines essential methods that any subclass must implement:

- **`goalP(searcher)`**: Determines if the current state meets the goal criteria.
- **`get_Successors(searcher)`**: Generates and returns a list of possible successor states.
- **`same_State(other_state)`**: Checks if another state is equivalent to the current state.
- **`cost_from(from_state)`**: Calculates the cost of transitioning from another state (not utilized in DFS).
- **`difference(goal_state)`**: Measures the difference from a goal state (not utilized in DFS).

These methods provide a standardized interface for evaluating states, generating successors, and determining goal attainment, enabling the search algorithm to operate generically across different problem domains.

`Search_Node`

Each `Search_Node` instance represents a node in the search tree, holding a specific state and a reference to its parent node.

Key functionalities include:

- **`expand(searcher)`**: Generates successor nodes by invoking `get_Successors` on the current state.
- **`goalP(searcher)`**: Checks if the node's state is a goal state.

This structure facilitates the construction of the search tree, enabling traversal and exploration of different states systematically.

`Search`

The `Search` class encapsulates the generic search algorithm. Its primary method, **`run_Search(init_state, searcher, search_method)`**, executes the search process by exploring nodes in a depth-first manner. It maintains an `open` list (acting as a stack) and a `closed` set to track visited states. Upon finding a goal state, it constructs and reports the solution path.

Key aspects of the DFS implementation include:

- **Open List**: Functions as a stack (Last-In-First-Out) to manage the nodes to be explored.
- **Closed Set**: Keeps track of visited states to prevent redundant exploration.
- **Successor Generation**: Successor states are generated by placing letters in the next available empty cell, ensuring consistency with puzzle constraints.

4 Gogen-Specific Components

4.1 `Gogen_State` Class

The `Gogen_State` class is pivotal in modeling the current state of a Gogen puzzle within the constraint satisfaction framework. As a subclass of the abstract `Search_State` class, it encapsulates all necessary information about the puzzle's current configuration and provides methods to manipulate and evaluate this state during the search process. This section analyses each function within the `Gogen_State` class, elucidating their roles, inputs, outputs, and the rationale behind their implementation.

4.1.1. Initialization and State Setup

`__init__(self, starting_letters)`

Purpose: Initializes a new instance of the `Gogen_State` class with the provided starting letters, setting up the initial grid configuration.

Arguments:

- `starting_letters` (str): A nine-character string representing the initial letters to be placed on the grid.

Functionality:

- **Grid Initialization**: Creates a 5x5 grid (`self.board`) where each cell is initially set to `None`.
- **Placement of Starting Letters**: Places the nine provided starting letters (`starting_letters`) at predefined positions on the grid. These positions are consistent across all puzzles, ensuring a standard starting point.
- **Remaining Letters Calculation**: Determines the set of remaining letters (`self.remaining_letters`) by excluding the starting letters and the letter 'Z' from the complete English alphabet.
- **Empty Cells Identification**: Identifies and stores the coordinates of the 16 empty cells (`self.empty_cells`) that need to be filled with the remaining letters.

Rationale: This constructor establishes the foundational state of the puzzle, ensuring that the initial grid is correctly populated and that the solver is aware of which letters and cells are available for subsequent placements.

4.1.2. Successor Generation

`get_Successors(self, searcher)`

Purpose: Generates all possible valid successor states from the current state by placing one of the remaining letters into the next available empty cell.

Arguments:

- `searcher` (`Gogen_Search`): An instance of the `Gogen_Search` class, providing access to the list of words to be formed.

Functionality:

- **Empty Cell Check**: If there are no empty cells left, it returns an empty list, indicating no further successors.
- **Next Cell Selection**: Selects the first empty cell (`target_row`, `target_col`) from `self.empty_cells` for letter placement.
- **Candidate Letters**: Iterates over each letter in `self.remaining_letters` as potential candidates for the selected empty cell.
- **State Copying**: For each candidate letter, creates a deep copy of the current state (`next_state`) to avoid mutating the original state.
- **Letter Placement**: Places the candidate letter in the selected cell of the copied state.
- **Constraint Checking**: Verifies that the placement of the letter does not violate any constraints by ensuring that all words can still potentially be formed with the current partial assignment.
- **Successor Accumulation**: If the placement is consistent, the new state is added to the list of successors.

Rationale: This method systematically explores all feasible letter placements in the next empty cell, ensuring that each potential move adheres to the puzzle's constraints. By generating only consistent successors, it optimizes the search process by pruning infeasible paths early.

4.1.3. Goal State Verification

`goalP(self, searcher)`

Purpose: Determines whether the current state is a goal state, meaning all cells are filled and all specified words are successfully formed on the grid.

Arguments:

- `searcher` (`Gogen_Search`): An instance of the `Gogen_Search` class, providing access to the list of words to be validated.

Functionality:

- **Empty Cell Check**: Immediately returns `False` if there are still empty cells, as the puzzle is not yet complete.
- **Word Formation Validation**: Iterates through each word in the provided word list and checks if it can be formed on the current grid using the `can_form_word` method.

- **Word Display**: If all words can be formed, it prints each word's placement on the grid for validation purposes.
- **Goal Confirmation**: Returns `True` if all words are successfully formed, signaling that the current state is a goal state.

4.1.4. Word Formation Feasibility

`can_form_word(self, word, is_partial)`

The `can_form_word` function is a pivotal component of the Gogen puzzle solver. Its primary responsibility is to ascertain whether a specified word can be constructed on the current 5x5 grid configuration, adhering strictly to the puzzle's adjacency rules. This ensures that each letter of the word is placed in cells that are horizontally, vertically, or diagonally adjacent to one another, without reusing any cell within the same word formation path.

Purpose and Functionality

The `can_form_word` function serves to validate the feasibility of forming a given word based on the current state of the grid. It operates by verifying two main criteria:

1. **Letter Availability:** Ensures that all letters required for the word are either already present on the grid or can be placed in the remaining empty cells.
2. **Adjacency Compliance:** Confirms that the letters of the word can be connected sequentially through adjacent cells, maintaining the integrity of the puzzle's rules.

Detailed Workflow

1. **Mapping Letter Positions:**
    - **Objective:** Identify all potential grid positions for each letter in the target word.
    - **Process:**
        - The function begins by creating a dictionary that maps each letter of the word to a set of grid positions where that letter is either already placed or can potentially be placed (if the `is_partial` flag is `True` and the letter is available in the remaining letters).

        - This mapping accounts for multiple occurrences of the same letter and multiple possible positions for each letter, providing a comprehensive overview of where each letter can be located on the grid.

2. **Early Termination Check:**
    - **Objective:** Optimize performance by quickly identifying impossible scenarios.
    - **Process:**
        - The function iterates through each letter in the word to verify that there is at least one available position for it on the grid.
        - If any letter lacks available positions, the function immediately returns `False`, indicating that the word cannot be formed under the current grid configuration.
3. **Depth-First Search (DFS) for Adjacency Verification:**
    - **Objective:** Ensure that the letters of the word can be connected in a valid sequence through adjacent cells.
    - **Process:**
        - The function employs a recursive depth-first search (DFS) strategy to explore all possible paths for forming the word.
        - **`dfs(current_position, letter_index, visited_positions)`**
            - **Purpose:** Implements the recursive search to verify the possibility of forming the word by traversing adjacent cells.
            - **Functionality:**
                - **Base Case:** If the `letter_index` equals the length of the word, the entire word has been successfully formed, and the function returns `True`.
                - **Recursive Case:**
                    - Identifies the next letter to be placed based on the `letter_index`.
                    - Iterates through all potential positions for this next letter.
                    - For each potential position, checks if it is adjacent to the current position and has not been visited yet.
                    - If both conditions are met, marks the position as visited and recursively attempts to place the subsequent letter.

        - If the recursive call returns `True`, the word formation is successful.
        - If not, backtracks by unmarking the position and continues searching.
      - **Termination:**
        - If all paths from the current position fail to form the word, the function returns `False`.
    - **Role in `can_form_word`:**
      - Serves as the core mechanism for exploring all possible paths to form the word, ensuring both letter availability and adjacency compliance.
  - **Adjacency Determination:**
    - **Objective:** Define the criteria for cells to be considered adjacent.
    - **Process:**
      - Two cells are deemed adjacent if they are next to each other horizontally, vertically, or diagonally.
      - The function ensures that the difference in both row and column indices between two positions does not exceed one, and that the positions are not identical.
  - **Initiating DFS from All Starting Positions:**
    - **Objective:** Explore all possible starting points for forming the word.
    - **Process:**
      - The function iterates through each available position of the first letter in the word.
      - For each starting position, it invokes the DFS method to attempt forming the rest of the word.
      - If any invocation of DFS successfully forms the word, the function concludes that the word can be formed and returns `True`.
      - If none of the starting positions lead to a successful formation, the function ultimately returns `False`.

Conclusion

The `can_form_word` function is integral to the Gogen puzzle solver's ability to accurately and efficiently determine the feasibility of forming specified words within the given grid constraints. By meticulously mapping letter positions, employing a robust DFS algorithm for adjacency verification, and optimizing performance through early termination checks, the function ensures both correctness and efficiency. This comprehensive approach not only adheres to the puzzle's rules but also enhances the solver's capability to navigate complex grid configurations, ultimately contributing to the successful resolution of the Gogen puzzle.

4.1.5. Visualizing Word Placement

`show_word_on_grid(self, target_word)`

Purpose: Displays the current grid with the specified word highlighted, providing a visual confirmation of the word's placement.

Arguments:

- `target_word` (str): The word to be highlighted on the grid.

Functionality:

- **Grid Duplication**: Creates a copy of the current grid (`grid_copy`) where only the letters of the `target_word` are retained, and all other cells are set to a blank space.
- **Grid Printing**: Formats and prints the grid with horizontal and vertical separators, clearly indicating the placement of the specified word.

4.1.6. State Duplication

`copy(self)`

Purpose: Creates and returns a deep copy of the current `Gogen_State` instance, ensuring that modifications to the copy do not affect the original state.

Functionality:

- **State Cloning**: Initializes a new `Gogen_State` instance without invoking the constructor (`__new__`), then duplicates the current grid (`self.board`), remaining letters (`self.remaining_letters`), and list of empty cells (`self.empty_cells`).

- **Return Value**: Returns the newly created deep copy of the state.

Rationale: This method is essential for generating successor states during the search process. By providing an independent copy of the current state, the solver can safely explore different letter placements without risking unintended side effects on the original state.

4.1.7. State Comparison

`same_State(self, other_state)`

Purpose: Determines whether another `Gogen_State` instance (`other_state`) is equivalent to the current state.

Rationale: This method facilitates the detection of duplicate states during the search process. By identifying equivalent states, the solver can avoid redundant explorations, thereby optimizing the search efficiency.

4.1.8. Grid Representation

`__str__(self)`

Purpose: Provides a human-readable string representation of the current grid state, suitable for printing and visualization.

4.1.9. Summary of `Gogen_State` Functions

The `Gogen_State` class comprises a cohesive set of functions designed to manage and evaluate the state of a Gogen puzzle effectively. Here's a summary of the implemented functions and their interrelationships:

Function	Purpose
`__init__`	Initializes the grid with starting letters and identifies remaining letters and empty cells.
`get_Successors`	Generates all possible valid successor states by placing remaining letters.
`goalP`	Determines if the current state is a goal state by checking word formation and grid completion.
`can_form_word`	Verifies whether a specific word can be formed on the current grid configuration.

Function	Purpose
`show_word_on_grid`	Provides a visual representation of a word's placement on the grid.
`copy`	Creates a deep copy of the current state for successor generation.
`same_State`	Compares two states for equivalence based on their grid configurations.
`__str__`	Returns a formatted string representation of the grid for visualization.

4.2 `Gogen_Search` subclass

The `Gogen_Search` class extends the `Search` base class to handle Gogen-specific search operations. It manages the list of words to be placed on the grid and orchestrates the search process.

Methods:

- **`__init__(words_file)`**:
  - Initializes the `Gogen_Search` instance by reading the list of words from the specified file.
  - Ensures the number of words read matches the provided count in the file for consistency.
- **`read_words(filename)`**:
  - Parses the input file to extract the number of words and the word list, which are used to initialize the search.
- **`getWords()`**:
  - Provides access to the full list of words.
  - Useful for retrieving all words for search-related tasks or verifying grid constraints.
- **`nthWord(i)`**:
  - Retrieves the ith word from the word list, making it accessible during the search process.
  - This method supports tasks such as validating word placement on the grid.
- **`run_Search(init_state, search_method)`**:

- Uses the generic search algorithm defined in the `Search` base class to perform the search.
- Takes the initial state and search method as arguments, passing them to the base class's method to initiate the search process.

4.3 `Run_Gogen_Search` subclass

The `Run_Gogen_Search` class serves as the entry point for executing the Gogen puzzle solver. It initializes the search process and handles user inputs.

Methods:

- **`__init__(words_file, starting_letters)`**: Sets up the `Gogen_Search` instance with the word list and initializes the `Gogen_State` with the starting letters.
- **`run()`**: Executes the search and displays the result, including the filled grid and efficiency metrics.

5. Execution Flow

The program operates through the following sequence of steps:

1. **Input Handling**:
   - The user provides two command-line arguments: the name of the text file containing the list of words and a nine-character string representing the initial letters on the grid.
   - The program validates that the starting letters string is exactly nine characters long.
2. **Initialization**:
   - An instance of `Run_Gogen_Search` is created with the provided inputs.
   - `Gogen_Search` reads the words from the specified file.
   - `Gogen_State` initializes the grid with the starting letters placed at their predefined positions.
3. **Search Execution**:
   - The `run_Search` method of `Gogen_Search` is invoked, initiating the depth-first search algorithm.
   - The search explores possible letter placements, generating successor states while adhering to the constraints that ensure all words can be formed.

4. **Solution Detection**:
    - When a goal state is reached—where all cells are filled and all words are successfully formed—the search halts.
    - The program constructs the solution path, calculates efficiency metrics, and displays the filled grid with all words highlighted for validation.
5. **Output**:
    - The final grid is printed, showcasing the solution to the Gogen puzzle.
    - Efficiency metrics and the sequence of states leading to the solution are also displayed for analysis.

6. Conclusion

The implemented Python program effectively solves Gogen puzzles by modeling them as constraint satisfaction problems and employing a depth-first search strategy. The `Gogen_State` class plays a pivotal role in managing the grid configuration, tracking remaining letters, and enforcing constraints that ensure all words can be formed. By incorporating both global and local constraints, the solver efficiently navigates the search space, pruning infeasible paths early and preventing redundant state explorations. The depth-first search approach, complemented by strategic constraint checking, enables the program to find valid solutions systematically. Overall, the implementation demonstrates a robust and efficient method for solving complex Gogen puzzles within the defined framework.

7. Appendix A: Sample Input and Output

Sample Input File (`gwords.txt`):

```
12
ACQUIRE
AXLE
CUT
DERV
FIR
JUST
KEG
MAN
PILE
PRY
```

```
ROB
WHAM
```

Starting Letters String:

```
MGDWLYSJB
```

Sample Output:

```
python3 solution.py gwords.txt MGDWLYSJB
Running Gogen Search with words file: gwords.txt and starting
letters: MGDWLYSJB
===============================
Word: ACQUIRE
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   | A |   | E |   |
---------------------
|   | C |   | R |   |
---------------------
| Q | U | I |   |   |
---------------------
|   |   |   |   |   |
---------------------

===============================
Word: AXLE
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   | A | X | E |   |
---------------------
|   |   | L |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------

===============================
Word: CUT
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   | C |   |   |   |
---------------------
|   | U |   |   |   |
---------------------
|   | T |   |   |   |
---------------------

===============================
Word: DERV
Grid:
---------------------
|   |   |   |   | D |
---------------------
|   |   |   | E | V |
---------------------
|   |   |   | R |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------

===============================
Word: FIR
Grid:
---------------------
```

```
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   | R |   |
---------------------
|   |   | I |   |   |
---------------------
|   |   |   | F |   |
---------------------

===============================
Word: JUST
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   | U |   |   |   |
---------------------
| S | T | J |   |   |
---------------------

===============================
Word: KEG
Grid:
---------------------
|   |   | G | K |   |
---------------------
|   |   |   | E |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
```

```
|   |   |   |   |   |
---------------------

===============================
Word: MAN
Grid:
---------------------
| M | N |   |   |   |
---------------------
|   | A |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------

===============================
Word: PILE
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   |   |   | E |   |
---------------------
|   |   | L |   |   |
---------------------
|   |   | I | P |   |
---------------------
|   |   |   |   |   |
---------------------

===============================
Word: PRY
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   | R | Y |
---------------------
|   |   |   | P |   |
---------------------
|   |   |   |   |   |
---------------------

===============================
Word: ROB
Grid:
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   | R |   |
---------------------
|   |   |   |   | O |
---------------------
|   |   |   |   | B |
---------------------

===============================
Word: WHAM
Grid:
---------------------
| M |   |   |   |   |
---------------------
| H | A |   |   |   |
---------------------
| W |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------

================================
Search Succeeds
Efficiency: 0.14 (Path length: 17 / Nodes visited: 125)
Nodes visited: 125
Solution Path:
Node 1:

---------------------
| M |   | G |   | D |
---------------------
|   |   |   |   |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 2:

---------------------
| M | N | G |   | D |
---------------------
|   |   |   |   |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 3:

---------------------
```

```
| M | N | G | K | D |
---------------------
|   |   |   |   |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 4:

---------------------
| M | N | G | K | D |
---------------------
| H |   |   |   |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 5:

---------------------
| M | N | G | K | D |
---------------------
| H | A |   |   |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------
```

```
Node 6:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X |   |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 7:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E |   |
---------------------
| W |   | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 8:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W |   | L |   | Y |
---------------------
```

```
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 9:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L |   | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 10:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
|   |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 11:

---------------------
| M | N | G | K | D |
---------------------
```

```
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q |   |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 12:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q | U |   |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 13:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q | U | I |   |   |
---------------------
| S |   | J |   | B |
---------------------

Node 14:
```

```
---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q | U | I | P |   |
---------------------
| S |   | J |   | B |
---------------------

Node 15:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q | U | I | P | O |
---------------------
| S |   | J |   | B |
---------------------

Node 16:

---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q | U | I | P | O |
---------------------
```

```
| S | T | J |   | B |
---------------------


Node 17:


---------------------
| M | N | G | K | D |
---------------------
| H | A | X | E | V |
---------------------
| W | C | L | R | Y |
---------------------
| Q | U | I | P | O |
---------------------
| S | T | J | F | B |
---------------------


Success
```

8. Appendix B: Python Code

```
import sys



# Base class for search states

class Search_State:

def goalP(self, searcher):

# Determines if the current state is a goal state.

raise NotImplementedError



def get_Successors(self, searcher):

# Generates and returns a list of successor states from the
```

```
current state.

raise NotImplementedError



def same_State(self, other_state):

# Compares the current state with another state to check for
equivalence.

raise NotImplementedError



def cost_from(self, from_state):

# Calculates the cost of transitioning from 'from_state' to the
current state.

# used for algorthms like A* not for depth first search

raise NotImplementedError



def difference(self, goal_state):

# Measures the difference between the current state and a goal
state.

# used for algorthms like A* not for depth first search

raise NotImplementedError



# Node representation in the search tree

class Search_Node:

def __init__(self, state, parent=None):

# Initializes the node with a state and an optional parent node.
```

```
self.state = state

self.parent = parent



def expand(self, searcher):

# Generates and returns a list of successor nodes.

successor_states = self.state.get_Successors(searcher)

return [Search_Node(s, self) for s in successor_states]



def goalP(self, searcher):

# Returns True if the node's state is a goal state, otherwise
False.

return self.state.goalP(searcher)



def __eq__(self, other):

# Returns True if the node's state is equal to another node's
state.

return self.state.same_State(other.state)



def __hash__(self):

# Returns a hash value for the node based on its state.

# hash allows for efficient lookup in sets and dictionaries

return hash(self.state)
```

```
# Base class for search algorithms

class Search:

def run_Search(self, init_state, searcher, search_method):

self.init_node = Search_Node(init_state)

self.open = [self.init_node]

self.closed = set()



while self.open:

current_node = self.open.pop() # Last In First Out for depth-
first search



if current_node.state in self.closed:

continue



self.closed.add(current_node.state)



if current_node.state.goalP(searcher):

return self.report_Success(current_node)



successor_nodes = current_node.expand(searcher)

for node in successor_nodes: # add successors to open list if not
in closed list or already in open list

if node.state not in self.closed and all(node.state != n.state
for n in self.open):

self.open.append(node) # Append to end for depth-first



return "Search Fails"



def report_Success(self, node):

# Construct the solution path

path = []

n = node

while n:

path.append(n.state)

n = n.parent

path.reverse() # reverse path to start from initial state



# Calculate efficiency

efficiency = len(path) / (len(self.closed) + 1)



print("================================")

print("Search Succeeds")

print(f"Efficiency: {efficiency:.2f} (Path length: {len(path)} /
Nodes visited: {len(self.closed) + 1})")

print(f"Nodes visited: {len(self.closed) + 1}")

print("Solution Path:")
```

```
for index, state in enumerate(path, start=1):

print(f"Node {index}:")

print() # Add an empty line between the node index and the state

print(state)

print()

return "Success"



# State representation for the Gogen puzzle

class Gogen_State(Search_State):

def __init__(self, starting_letters):

# Initialize the Gogen state with the starting letters

self.board = [[None for _ in range(5)] for _ in range(5)]

starting_positions = [

(0, 0), (0, 2), (0, 4),

(2, 0), (2, 2), (2, 4),

(4, 0), (4, 2), (4, 4)

]



for i, position in enumerate(starting_positions):

row, col = position

self.board[row][col] = starting_letters[i]
```

```
alphabet_set = set('ABCDEFGHIJKLMNOPQRSTUVWXYZ') - {'Z'}

self.remaining_letters = alphabet_set - set(starting_letters)

self.empty_cells = [(r, c) for r in range(5) for c in range(5) if
self.board[r][c] is None]



def get_Successors(self, searcher):

# Generate and return a list of successor states for the current
state

# check if the successors are consistent with the words

successors = []

if not self.empty_cells: # if no empty cells, return empty list
for successors

return successors



target_row, target_col = self.empty_cells[0]

candidate_letters = self.remaining_letters.copy()



for letter in candidate_letters:

# Create a copy of the next state

next_state = self.copy()

next_state.board[target_row][target_col] = letter

next_state.empty_cells = self.empty_cells[1:]

next_state.remaining_letters = self.remaining_letters - {letter}
```

```
# Check consistency

consistent = True

for word in searcher.getWords():

if not next_state.can_form_word(word, is_partial=True):

consistent = False

break



if consistent:

successors.append(next_state)



return successors



def goalP(self, searcher):

# Check if the current state is a goal state

if self.empty_cells:

return False



for word in searcher.getWords():

if not self.can_form_word(word, is_partial=False):

return False
```

```
for word in searcher.getWords():

self.show_word_on_grid(word)

return True



def can_form_word(self, word, is_partial):

# Check if the current state can form a given word

letter_positions = {letter: set() for letter in word} #
dictionary to store the positions of the letters in the word



# iterate through the board to find the positions where the
letters of the word are located

# or if the word is partial, the empty cells where the letters
can be placed

# each letter can have multiple positions on the board

for i in range(5):

for j in range(5):

cell_value = self.board[i][j]

for letter in word:

# if the cell value is the letter or if the cell is empty and the
letter is in the remaining letters

if cell_value == letter or (is_partial and cell_value is None and
letter in self.remaining_letters):

letter_positions[letter].add((i, j))



for letter in word:
```

```
if not letter_positions[letter]:

return False


def dfs(current_position, letter_index, visited_positions):

# Base case: if the entire word is formed, return True

if letter_index == len(word):

return True


# Get the next letter to find

next_letter = word[letter_index]


# Explore all possible positions for the next letter

for potential_position in letter_positions[next_letter]:

# Check if the potential position is adjacent and not visited

if (potential_position not in visited_positions and

is_adjacent(current_position, potential_position)):


# Mark the position as visited

visited_positions.add(potential_position)


# Recursively attempt to form the rest of the word

if dfs(potential_position, letter_index + 1, visited_positions):
```

```
return True



# Backtrack: unmark the position as visited to try a different
path using the same letter

visited_positions.remove(potential_position)



return False



def is_adjacent(pos1, pos2):

# Check if two positions are adjacent on the grid

return (abs(pos1[0] - pos2[0]) <= 1 and

abs(pos1[1] - pos2[1]) <= 1 and

pos1 != pos2)



# Start DFS from each position of the first letter

for start_position in letter_positions[word[0]]:

if dfs(start_position, 1, {start_position}): # if the word can be
formed, return True

return True



return False



def show_word_on_grid(self, target_word):
```

```
grid_copy = [

[

self.board[i][j] if self.board[i][j] in target_word else ' '

for j in range(5)

]

for i in range(5)

]



print("================================")

print(f"Word: {target_word}")

print("Grid:")

print('-' * 21)

for row in grid_copy:

row_str = ' | '.join(row)

print(f"| {row_str} |")

print('-' * 21)

print()



def copy(self):

new_state = Gogen_State.__new__(Gogen_State)

new_state.board = [row[:] for row in self.board]

new_state.remaining_letters = self.remaining_letters.copy()
new_state.empty_cells = self.empty_cells[:]

return new_state


def same_State(self, other_state):

return self.board == other_state.board


def __eq__(self, other):

return self.same_State(other)


def __hash__(self):

return hash(str(self.board))


def __str__(self):

grid_str = ""

horizontal_line = '-' * 21

for row in self.board:

row_str = ' | '.join([c if c is not None else ' ' for c in row])

grid_str += f"{horizontal_line}\n| {row_str} |\n"

grid_str += horizontal_line

return grid_str


# Extends Search to solve the Gogen puzzle
```

```
class Gogen_Search(Search):

def __init__(self, words_file):

super().__init__()

self.words = self.read_words(words_file)



def read_words(self, filename):

with open(filename, 'r') as f:

lines = f.read().splitlines()

num_words = int(lines[0])

# Strip whitespace and convert to uppercase

words = [line.strip().upper() for line in lines[1:num_words + 1]]

return words



def getWords(self):

return self.words



def nthWord(self, i):

if 0 <= i < len(self.words):

return self.words[i]

else:

return ""
```

```
def run_Search(self, init_state, search_method):

return super().run_Search(init_state, self, search_method)



# Main class to execute the Gogen search algorithm

class Run_Gogen_Search:

def __init__(self, words_file, starting_letters):

# Initialize Gogen_Search with words_file

self.searcher = Gogen_Search(words_file)



# Set up initial state

self.initial_state = Gogen_State(starting_letters)



def run(self):

result = self.searcher.run_Search(self.initial_state,
"depth_first")

print(result)



# Main execution

if __name__ == "__main__":

if len(sys.argv) != 3:

print("Usage: python3 code.py <words_file> <starting_letters>")

sys.exit(1)
```

```
words_file = sys.argv[1]

starting_letters = sys.argv[2]



if len(starting_letters) != 9:

print("Error: Starting letters string must be exactly 9
characters long.")

sys.exit(1)



print(f"Running Gogen Search with words file: {words_file} and
starting letters: {starting_letters}")

runner = Run_Gogen_Search(words_file, starting_letters)

runner.run()
```