ASSIGNMENT 1, AREEBA RIAZ, 21I-1736, DS-N

Algorithm: Depth-First Search (DFS)

The algorithm used to solve cryptarithmetic puzzles is primarily based on Depth-First Search (DFS). Here's how it works:

Initialization: Initialize the puzzle, extract unique letters, and initialize a mapping for each letter to None.

DFS Function: Define a DFS function to explore possible solutions recursively. At each step, check the validity of the current mapping to ensure it adheres to puzzle constraints. If the current mapping satisfies the equation, return it as a potential solution; otherwise, continue exploration.

Backtracking: If no solution is found with the current mapping, backtrack to the previous state and explore alternative mappings. Backtracking continues until a valid solution is found or all possible combinations are exhausted.

Evaluation: Implement an evaluation function to compute the value of an expression using the current mapping of letters to digits.

Solution Verification: Once a potential solution is found, verify its correctness by evaluating the expressions using the current mapping. If the equations hold true, the solution is valid; otherwise, continue searching for alternative mappings.

Termination: The algorithm terminates when a valid solution is found or when all possible mappings have been explored without success.

Implementation

PuzzleSolverDFS Class:

- **Constructor:** Initializes the puzzle by storing it and extracting unique letters from it. Initializes a mapping for each unique letter, setting initial values to None.

- **solve Method:** Splits the puzzle into left and right operands and the result. Implements a DFS algorithm using an inner function to explore possible solutions recursively. Checks the validity of the current mapping at each step to ensure it adheres to puzzle constraints, such as no leading zeros. If a valid solution is found, it returns the mapping; otherwise, it continues exploring. Backtracks, when necessary, by reverting to the previous state and exploring alternative mappings.

- **evaluate Method:** Evaluates an expression using the current mapping of letters to digits. Converts the expression into a numerical value by substituting digits for letters according to the mapping.

SolutionPrinter Class:

- **print_solution Method:** Takes the puzzle and the solution as input. Prints the solution by substituting letters with their corresponding digits. If no solution is found, it indicates that no solution exists for the given puzzle.

CryptarithmeticSolver Class:

- **Constructor:** Initializes the puzzle.
- **solve_and_print Method:** Instantiates a PuzzleSolverDFS object to solve the puzzle. Prints the solution using SolutionPrinter.

Efficient Backtracking:

Backtracking is achieved by reverting to the previous state of the mapping when a solution violation is encountered. This is accomplished by resetting the value of the current letter to None and continuing exploration.

Modular Design:

Encapsulation of Logic:

The solving process is encapsulated within separate classes (PuzzleSolverDFS, SolutionPrinter, and CryptarithmeticSolver), enhancing code organization and readability . Each class focuses on a specific aspect of the solving process, facilitating maintenance and extension of the codebase.

Efficient Exploration:

The algorithm avoids exploring branches that are likely to lead to invalid solutions, such as assigning digits already used in the current mapping.

where N is the number of unique letters in the puzzle.

Time Complexity: O(10^N)

Space Complexity: O(N)

EXAMPLE = > BREAD + WINE = DINER

Initialization:

Puzzle: BREAD + WINE = DINER

Unique Letters: B, R, E, A, D, W, I, N

Initial Mapping: {B: None, R: None, E: None, A: None, D: None, W: None, I: None, N: None}

DFS Exploration:

Start with an empty mapping: {B: None, R: None, E: None, A: None, D: None, W: None, I: None, N: None}.Assign a digit to the first unassigned letter (e.g., B: 2).Recursively explore possible mappings until a valid solution is found or backtracking is required.

Step 1: Assigning B (B: 2) = Mapping: {B: 2, R: None, E: None, A: None, D: None, W: None, I: None, N: None}

STEP 2 = Mapping: {B: 2, R: 8, E: None, A: None, D: None, W: None, I: None, N: None}

STEP3 = Mapping: {B: 2, R: 8, E: 5, A: None, D: None, W: None, I: None, N: None}

STEP4 = Mapping: {B: 2, R: 8, E: 5, A: 4, D: None, W: None, I: None, N: None}

STEP5 = Mapping: {B: 2, R: 8, E: 5, A: 4, D: 3, W: None, I: None, N: None}

STEP6 = Mapping: {B: 2, R: 8, E: 5, A: 4, D: 3, W: 7, I: None, N: None}

STEP7 = Mapping: {B: 2, R: 8, E: 5, A: 4, D: 3, W: 7, I: 6, N: None}

STEP8 = Mapping: {B: 2, R: 8, E: 5, A: 4, D: 3, W: 7, I: 6, N: 1}

All letters are assigned. Evaluate the equation.

Equation Evaluation:

BREAD: 28543 + WINE: 7615 = DINER: 36158

The equation holds true, so the solution is valid:

A: 4, D: 3, +: 0, R: 8, W: 7, B: 2, N: 1, =: 9, I: 6, E: 5

The final solution is verified, with each letter correctly substituted by a digit, resulting in a valid mathematical expression.

Algorithm: A* SEARCH

Approach

Initialization: *State Initialization:* Unique placeholder values are assigned to each letter in the expression.

Goal State Extraction: *Extract Goal State:* The goal state is extracted from the expression.

Heuristic Function: *Heuristic Estimation:* Define a heuristic function to estimate the cost to reach the goal state from any given state.

Search Space States:

Initial State: The initial state represents the starting point of the search with placeholder values for each letter.

Goal State: The goal state is the desired solution where the expression evaluates to true.

Intermediate States: These are the states generated during the search process by assigning digits to letters and evaluating the expression.

Generate Permutations: *Permutations:* Generate all possible permutations of digits (0-9).

Frontier: *Priority Queue:* Initialize a priority queue (frontier) to store nodes. Nodes are prioritized based on their estimated cost to reach the goal.

Explore Nodes: *Node Expansion:* Iterate through each permutation and create a node for each state.

Evaluate Expression: *Expression Evaluation:* Evaluate the expression using the current state to check if it satisfies the cryptarithm.

Next States Generation: *Generate Next States:* Generate the next possible states by assigning digits to letters.

Update Frontier: Add next states to the frontier if they have not been explored.

Repeat: *Search Iteration:* Continue exploring nodes until a solution is found or the frontier is empty.

Implementation

Cryptarithm Class

- **Initialization:** The Cryptarithm class initializes the puzzle with the expression. State Initialization: Unique placeholder values are assigned to each letter in the expression.
- **Goal State Extraction:** The goal state is extracted from the expression.
- **Getter Methods:** Getter methods are provided to access the initial and goal states.

PuzzleNode Class

- **Initialization:** The *PuzzleNode* class represents nodes in the search space.
- **Comparison:** Nodes are compared based on their costs for priority queue operations.

CryptarithmSolver Class

- **Initialization:** The *CryptarithmSolver* class initializes the solver with the cryptarithm expression.
- **Solve Method:** The solve method executes the A* algorithm to find the solution.
- **Heuristic Function:** A heuristic function estimates the cost to reach the goal state from any given state.
- **Expression Evaluation:** The *evaluate_expression* method evaluates the expression using the current state.
- **Next States Generation:** Possible next states are generated by assigning digits to letters.
- **Solution Retrieval:** Once a solution is found, it is returned.

AStarCryptarithmSolver Class

- **Initialization:** The AStarCryptarithmSolver class is a wrapper for CryptarithmSolver.
- **Solve Method:** It is called the solve method of CryptarithmSolver.

Printing Solution: The *print_solution function* prints the solution with the assigned digits for each letter and the evaluated expression.

where n is the number of unique digits (10 in this case).

Time complexity = **O(n!)**

Space Complexity = **O(n!)**

Example: STAR + STAR = NOVA

1. Initialization: **Expression:** STAR + STAR = NOVA

Initial State: Placeholder values assigned to each unique letter: {'S': -1, 'T': -1, 'A': -1, 'R': -1, 'N': -1, 'O': -1, 'V': -1}

2. Heuristic Function: The heuristic function estimates the cost to reach the goal state from any given state. In this case, the heuristic function calculates the difference between the target value of NOVA and the sum of STAR + STAR. Since each letter corresponds to a digit, the difference indicates how close the current state is to the goal state.

3. Generate Permutations: All possible permutations of digits (0-9) are generated. These permutations represent potential assignments of digits to letters.

4. Frontier Initialization: A priority queue (frontier) is initialized to store nodes. Each node represents a potential state in the search space. Nodes are prioritized based on their estimated cost to reach the goal, calculated using the heuristic function.

5. Explore Nodes

A) permutation: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Mapping: S: 0, T: 1, A: 2, R: 3, N: 4, O: 5, V: 6 , Evaluate: 0123 + 0123 = 0124 (Not a solution). Add node to frontier with cost estimated by the heuristic function.

B) Permutation: (0, 1, 2, 3, 4, 5, 6, 7, 9, 8)

Mapping: S: 0, T: 1, A: 2, R: 3, N: 4, O: 5, V: 6, Evaluate: 0123 + 0123 = 0124 (Not a solution)

Add node to frontier with cost estimated by the heuristic function, keep on doing this. Iterating through permutations, nodes are created for each state.

6. Evaluate Expression: Each node's state is evaluated against the expression to check if it satisfies the cryptarithm.

Example evaluation:

STAR: 4863 + STAR: 4863 = NOVA: 9726

7. Next States Generation: Possible next states are generated by assigning digits to letters.

Example next state: {'S': 4, 'T': 8, 'A': 6, 'R': 3, 'N': 9, 'O': 7, 'V': 2}

8. Update Frontier: Next states are added to the frontier if they have not been explored.

9. Repeat: The search continues until a solution is found or the frontier is empty.

Solution: **Solution found:**

A: 6, R: 3, O: 7, T: 8, N: 9, S: 4, V: 2, STAR [4863] + STAR [4863] =   NOVA [9726]