Experiment 2
State Space Search

***Abstract*—In this lab, we design a graph search agent to solve two problems: detecting plagiarism in text documents and solving the 8-puzzle. For plagiarism detection, we use the A* search algorithm to align sentences between two documents by minimizing edit distances, such as Levenshtein distance, to find similar or identical text. In the 8-puzzle problem, we apply state-space search techniques, such as A* and Iterative Deepening Search, to find the shortest sequence of moves to reach the goal configuration. The lab explores memory and time requirements for both tasks, showcasing the versatility of search algorithms.**

I. Introduction

This lab report consists of graph search algorithms to solve two distinct problems: plagiarism detection in text documents and the 8-puzzle. The goal is to design a graph search agent using techniques like A* and Iterative Deepening Search. By exploring these algorithms, we analyze their effectiveness in navigating problem spaces and their implications for memory and time efficiency.

The 8-puzzle is a classic AI problem that consists of a 3x3 grid with eight numbered tiles and one empty space. The objective is to rearrange the tiles from an initial configuration to a goal state by sliding them one at a time into the empty space. To solve this, we apply state-space search algorithms, such as A* and Iterative Deepening Search, to find the optimal sequence of moves that achieves the goal in the fewest steps.

The plagiarism detection task involves comparing two documents to identify similar or identical sequences of text, which may indicate copying. We implement the A* search algorithm to align sentences between the documents by minimizing the edit distance, such as Levenshtein distance. This approach helps to detect potential instances of plagiarism by finding closely matching text segments.

II. Puzzle-8

A. Problem Statement

1) Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.
2) Write a collection of functions imitating the environment for Puzzle-8.
3) Describe what is Iterative Deepening Search.
4) Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.
5) Generate Puzzle-8 instances with the goal state at depth "d".
6) Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

B. Solution

1) The pseudocode for the following problem can be accessed through this link and the flowchart of the following is:

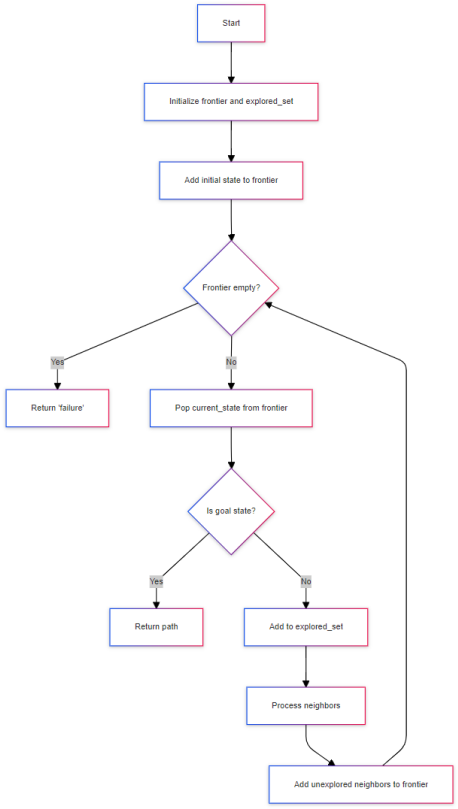

Figure 1: Flowchart for the Graph Search Agent

2) Collection of functions are:
   - **Initial State:** Any state can be the initial state where the configuration of blocks is stored in a 3×3 matrix.
   - **Actions:** Instead of moving the numbered blocks, we can consider the action as the blank space moving up, down, right, and left.
   - **Goal State:** Although any state could be the goal, the correct configuration of blocks (the expected final state) is considered the goal state.
   - **Goal Test:** Checking whether the new state is the same as the goal state or not.
   - **Transitions:** Switching the positions of moved and empty blocks, resulting in a new state.
   - **Path Cost:** Every step will have a cost, which will be added to the total cost.

3) Iterative deepening search involves applying depth-limited search multiple times, incrementing the depth limit with each iteration to efficiently locate a solution. In every iteration, there are three potential outcomes: finding a solution node, encountering failure if no solution exists within the current depth, or hitting a cutoff point that suggests a possible solution at a deeper level. This method does not keep track of previously reached states, which means the same state might be revisited multiple times along different paths. The optimal depth limit is determined by progressively testing limits starting from 0, then 1, and so on, until a solution is found. This ensures a comprehensive exploration of the state space. The time complexity of iterative deepening search depends on whether a solution is present. If a solution is found, the time complexity is $O(b^d)$, where b is the branching factor and d is the solution depth. If no solution exists, the complexity becomes $O(b^m)$, where m is the maximum explored depth.
4) The pseudo-code for the following problem can be accessed through link
5) The pseudo-code for the following problem can be accessed through link
6) The table is as follows:

Depth	Nodes Explored	Time Elapsed (s)	Memory Used (bytes)
1	2.00	0.0000	0
2	2.60	0.0000	819
3	2.80	0.0000	0
4	3.00	0.0003	0
5	2.00	0.0000	0
6	3.20	0.0000	0
7	3.60	0.0002	0
8	7.80	0.0000	2458
9	4.40	0.0000	0
10	3.80	0.0002	0
11	6.60	0.0000	0
12	6.40	0.0002	0
13	7.00	0.0000	0
14	29.00	0.0002	12288
15	6.20	0.0002	0
16	7.40	0.0000	0
17	14.00	0.0002	0
18	12.40	0.0002	0
19	9.00	0.0000	0
20	31.40	0.0002	0

Figure 2: Table indicating memory and time requirements

III. Plagiarism Detection System

A. *Problem Statement:*

Develop a plagiarism detection system that identifies similar or identical sequences of text between two documents by aligning their sentences or paragraphs. The system will leverage the A* search algorithm to efficiently find optimal text alignments based on edit distance, indicating potential instances of plagiarism.

B. *Levenshstein Distance*

Levenshtein distance is a similarity measure between two strings, representing the minimum number of edit operations (insertions, deletions, or substitutions) required to transform one string into another. For example, changing the word "test" into "tent" requires one substitution, so the Levenshtein distance is 1.

In the context of this lab, the Levenshtein distance is used to compute the cost of aligning sentences from two documents, which is key to detecting plagiarism. By aligning text based on minimal edit distance, the system identifies similar or identical sequences of text.

Algorithm and Use in Plagiarism Detection

Given two text documents, the system aligns their sentences using the A* search algorithm. The alignment minimizes the Levenshtein distance between corresponding sentences. The Levenshtein distance provides the following advantages in this context:

- **Cost Function (g(n)):** The total edit distance between aligned sentences is used to compute the cost. Smaller distances indicate higher similarity and possible plagiarism.
- **Heuristic Function (h(n)):** The remaining sentences' minimum edit distances are estimated to guide the A* search algorithm to an optimal alignment efficiently.

In this system, a lower Levenshtein distance between two sentences indicates a higher degree of similarity, which may be indicative of plagiarism. Conversely, a high edit distance suggests that the sentences differ significantly, reducing the likelihood of plagiarism.

Example of Levenshtein Distance

Consider two words: "GUMBO" and "GAMBOL". The Levenshtein distance is 2 because transforming "GUMBO" into "GAMBOL" requires one substitution ("U" to "A") and one insertion ("L"). This concept scales to entire sentences in plagiarism detection, where the number of edits required to transform one sentence into another helps identify potentially copied content.

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2

Figure 3: Levenshtein Distance Matrix for "GUMBO" and "GAMBOL"

C. *Solution*

The code for this problem statement can be accessed through this link.

Here's an overview of the steps taken to arrive at the solution:

1) Text Preprocessing: A 'preprocess_text' function was created that:

- Splits the input text into sentences using regular expressions.
- Converts each sentence to lowercase.
- Removes leading and trailing whitespace from each sentence.

2) Similarity Calculation: To calculate similarity between sentences, the Levenshtein distance was utilized:
   - A 'levenshtein_distance' function was implemented to compute the edit distance between two strings.
   - A 'calculate_similarity' function was developed that normalizes the Levenshtein distance to a percentage similarity.
3) Plagiarism Detection: The 'detect_plagiarism' function was created to tie everything together:
   - Finds the best pairs of sentences between the documents using A* algorithm to align sentences in both documents.
   - Calculates an overall similarity percentage for the entire documents.

The Levenshtein distance between two strings s_1 and s_2 is given by the following recurrence relation:

$$lev_{s_1,s_2}(i,j) = \begin{cases} \max(i,j) \\ \min \begin{cases} lev_{s_1,s_2}(i-1,j)+1 \\ lev_{s_1,s_2}(i,j-1)+1 \\ lev_{s_1,s_2}(i-1,j-1)+1_{(s_1[i]\neq s_2[j])} \end{cases} \end{cases}$$

where $1_{(s_1[i]\neq s_2[j])}$ is the indicator function equal to 0 when $s_1[i] = s_2[j]$ and equal to 1 otherwise.

The percentage similarity between two strings can be calculated using the following formula:

$$\text{similarity}(s1,s2) = \left(1 - \frac{\text{levenshtein\_distance}(s1,s2)}{\max(|s1|,|s2|)}\right) \times 100$$

Where: - $\text{levenshtein\_distance}(s1,s2)$ is the Levenshtein distance between the two strings $s1$ and $s2$. - $|s1|$ and $|s2|$ are the lengths of the strings $s1$ and $s2$ respectively.

D. A* Algorithm for document alignment

Input: Two documents doc1 and doc2.
Output: Optimal alignment between sentences

1) **Initialization:**
   - Create the initial state with pos1 = 0, pos2 = 0, cost = 0, and an empty path.
   - Initialize a priority queue (min-heap) with the initial state.
   - Initialize an empty set to keep track of visited positions.
2) **Main Loop:**
   - While the priority queue is not empty:
     - Pop the state with the lowest cost from the queue.
     - If both pos1 = |doc1| and pos2 = |doc2|, return the alignment path (goal reached).
     - If (pos1, pos2) has already been visited, continue to the next iteration.
     - Mark (pos1, pos2) as visited.
3) **State Expansion:**
   - If both pos1 < |doc1| and pos2 < |doc2|
     - Compute the cost using the Levenshtein distance between doc1[pos1] and doc2[pos2].
     - Create a new state advancing both pos1 and pos2, update the total cost, and append the alignment to the path.
     - Push the new state onto the priority queue with priority equal to the cost plus a heuristic.
   - If pos1 < |doc1| (skip sentence in doc2):
     - Create a new state advancing pos1 only, with an additional cost of 1, and push it onto the queue.
   - If pos2 < |doc2| (skip sentence in doc1):
     - Create a new state advancing pos2 only, with an additional cost of 1, and push it onto the queue.
4) **Heuristic:**
   - Use a heuristic function to estimate the remaining alignment cost by taking the minimum of the remaining sentences between doc1 and doc2:

$$h(\text{state}) = \min(|\text{doc1}| - \text{pos1}, |\text{doc2}| - \text{pos2})$$

E. Result of Test-Cases

The result of test cases can be seen below:

```
Test Case: Identical Documents
Document 1: This is a test. It has multiple sentences. We want to detect plagiarism.
Document 2: This is a test. It has multiple sentences. We want to detect plagiarism.
Detected 3 plagiarism cases:
  Sentence 1 in doc1 matches sentence 1 in doc2 with 100.00% similarity
  Sentence 2 in doc1 matches sentence 2 in doc2 with 100.00% similarity
  Sentence 3 in doc1 matches sentence 3 in doc2 with 100.00% similarity
Overall similarity: 100.00%
Verdict: Surely copied.

Test Case: Slightly Modified Document
Document 1: This is a test. It has multiple sentences. We want to detect plagiarism.
Document 2: This is an exam. It contains several phrases. We aim to identify copying.
Detected 0 plagiarism cases:
Overall similarity: 46.23%
Verdict: Possibly copied.
```

```
Test Case: Completely Different Documents
Document 1: This is about cats. Cats are furry animals. They make good pets.
Document 2: Python is a programming language. It is widely used in data science.
Detected 0 plagiarism cases:
Overall similarity: 19.93%
Verdict: Not copied.

Test Case: Partial Overlap
Document 1: This is a test. It has multiple sentences. We want to detect plagiarism.
Document 2: This is different. We want to detect plagiarism. This is unique.
Detected 1 plagiarism cases:
  Sentence 3 in doc1 matches sentence 2 in doc2 with 100.00% similarity
Overall similarity: 65.24%
Verdict: Likely copied.
```

Figure 4: Test Cases Verdict

IV. REFERENCES

1) Masrat, A., Gawde, H., Makki, M.A., & Parekh, U. (2021). Document Plagiarism Detection Tool using Edit Distance Text Similarity Measure. International Research Journal of Engineering and Technology (IRJET), 08(09), 1395-1399.
2) Gardahadi, G. (2019). Plagiarism Detection Using Levenshtein Distance With Dynamic Programming.