Foundations of AI
Yunyu Guo
Assignment 3

- ***For each search algorithm:***
- ***Is the first path that it finds guaranteed to be the shortest path?***

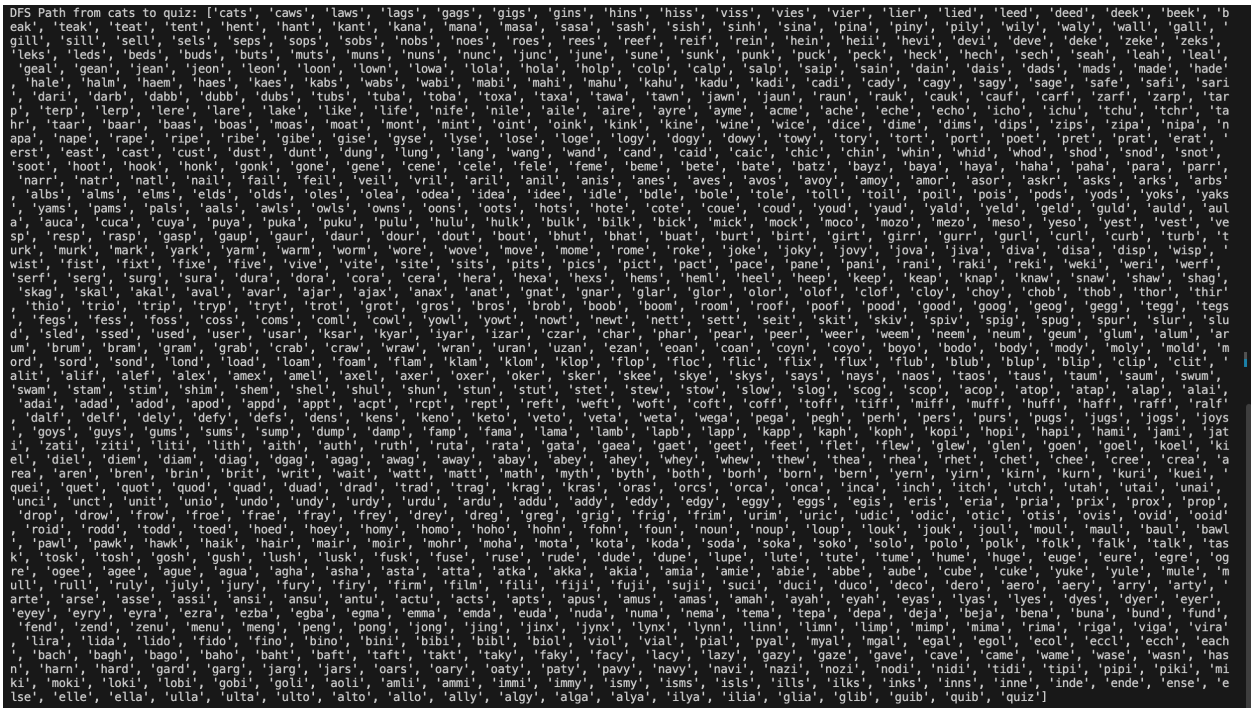
Breath first search (BFD):
No, the first path that BFS finds is not guaranteed to be the shortest path in all cases. BFS does guarantee finding the shortest path in unweighted graphs or graphs where all edge weights are equal.

In this specific word characters comparison problem, BFS will find the shortest path because:
The graph is unweighted - each edge (word change) has an implicit weight of 1.
The graph is undirected – the direction of each character in a word can be changed.

```
mac@Emmas-Laptop HW3 % /usr/local/bin/python3 /Users/mac/Desktop/5100/HW3/HW3.py
Please enter the start word: cats
Please enter the end word (same length as start word): quiz
Graph adjacency list created with 7186 entries.

BFS Path from cats to quiz: ['cats', 'tats', 'tuts', 'tuis', 'quis', 'quiz']
```

Depth first search (DFS):
No, the first path that DFS finds is not guaranteed to be the shortest path. DFS does not guarantee finding the shortest in general graph traversal. DFS explores as far as possible along each branch before backtracking. It goes deep into the graph before exploring other possibilities.
The first path DFS finds to the target word is not necessarily the shortest. It might find a longer path first, depending on the order of exploration.



Iterative deepening search (IDS):

Yes, in iterative deepening search, the first path it finds is guaranteed to be the shortest path. It combines DFS and BFS strategies. It performs a series of depth-limited searches, increasing the depth limit after each iteration. It explores all paths of length 1 before moving to length 2, all paths of length 2 before length 3, and so on. The first time it reaches the end word, it will be through the shortest possible path.

```
Iterative Deepening Path from cats to quiz: ['cats', 'caws', 'laws', 'lags', 'lugs', 'luis', 'quis', 'quiz']
```

A* search:

Yes, the first path it finds is guaranteed to be the shortest path. I use hamming distance between 2 words as heuristic and it is admissible which does not overestimate the cost to the goal. Nodes are ordered by their total_cost = actual_cost + h(n) in priority queue. Actual cost is the cost traveled so far and heuristic cost is the estimated remaining cost. A* search uses the heuristic to make informed decisions about which nodes to explore next. It prioritizes nodes that appear closer to the goal according to the heuristic. A* search can skip exploring some nodes that the heuristic suggests are less promising.

```
A* Path from cats to quiz: ['cats', 'cuts', 'tuts', 'tuis', 'quis', 'quiz']
```

- ***How efficient (time and space) is this algorithm for finding paths between words that are fairly similar (>50% characters in common)?***

Depth first search:
Time complexity: O(b^m), where b is branching factor and m is the maximum path length
Space complexity: O(bm)
For similar words: May not be efficient as it can explore deep paths before finding the solution.

Breadth first search:
Time complexity: O(b^(d+1)), d is the steps to goal
Space complexity: O(b^(d+1))
For similar words: when words has more than 50% characters in common, the solution path is likely short. BFS excels at finding such short paths quickly. In this case, it is likely to be more efficient than IDS because BFS does not repeat work. It explores each word exactly once.

Iterative deepening search:
Time complexity: O(b^d)
Space complexity: O(bd)
For similar words: Better than DFS, but still explores unnecessary paths in early iterations.
For example: In the first iteration (depth = 0), IDS explores only the start word.
In the second iteration (depth = 1), it explores the start word again, plus its immediate neighbors. In the third iteration (depth = 2), it explores the start word and its neighbors again, plus words two edits away.

A* search:
Time complexity: O(b^d) in worst case, but often better in practice
Space complexity: O(b^d) in worst case
For similar words: Most efficient among the mentioned searches above. Because it uses a heuristic (e.g., Hamming distance) that directly relates to word similarity. Prioritizing words more similar to the target, reducing unnecessary exploration.
Comparison for finding paths between similar words:

Time efficiency: A* *> BFS > IDS > DFS*

- ***How efficient (time and space) is this algorithm for finding paths between words that are not similar (<50% characters in common)?***

Breadth first search:
Guaranteed to find the shortest path, but may explore a large number of irrelevant paths first. Also high memory usage for not similar words.

Depth first search:
May get stuck exploring deep in irrelevant paths. Space-efficient, but time-inefficient for not similar words.

Iterative deepening:
More space-efficient than BFS. Can be more time-efficient than BFS for longer paths, as it doesn't store all nodes at each level.

A* search:
Potentially most efficient if the heuristic can guide the search effectively. However, for very dissimilar words, the heuristic might not provide as much benefit.

Time efficiency: A* > IDS > BFS > DFS

- ***How efficient (time and space) is this algorithm for determining that there is no path between the two words?***

  Breath firs search:
  BFS explores all possibilities in order of edit distance. It guarantees to exhaust all possibilities in the shortest number of steps.

  Depth first search:
  Like BFS, it explores all possibilities. It may get stuck exploring deep in irrelevant paths before concluding no path exists.

  Iterative deepening:
  It will eventually explore all possibilities, but with some repeated work as it iterates from the root each time. More space-efficient than BFS. It has a good balance of completeness and space efficiency.

  A* search:
  Potentially as efficient as BFS. It may be more efficient if the heuristic can quickly identify dead-ends.

- ***For the A* search, how did you choose an appropriate heuristic?***
  I choose the hamming distance, it counts the number of positions where the characters differ between 2 words in equal length. The heuristic is admissible, it does not overestimate the number of steps needed to reach the goal word. This heuristic is also consistent. The estimated cost from any word to the goal is no greater than the cost of changing one letter plus the estimated cost from the resulting word.

- ***Suppose there is a set of words that you must include at some point in the path between the start_word and the end_word. The order of the words does not matter. How would you implement an algorithm that finds the shortest path from the start_word to the end_word which includes every word in the given set of words? You may describe the algorithm or provide pseudocode.***

  Modify the heuristic function in A* search to consider both the distance to the end word and the number of remaining required words.
  def heuristic(start_word: str, end_word: str, remaining_required: set[str]) -> int:
      return hamming_distance(start_word, end_word) + len(remaining_required)
  Initialize the search with the start word and all required words.
  As exploring neighbors, remove them from the set of remaining required words if they're in that set.
  Reach the goal state when it finds the end word and there is no remaining required words.
  Use a tuple of the current word and sorted remaining required words as the visited state to avoid revisiting equivalent states.

- ***How long did this assignment take you? (1 sentence)***
  2 days

- ***Whom did you work with, and how? (1 sentence each)***
  I discuss the NodePathWord class property, how to use the node eg. in priority queue, and in the get_path_to_root() method to move the node pointer to the parent in order to get the path from goal to start word.

- *Which resources did you use? (1 sentence each)*
  Dictionary: https://github.com/dwyl/english-words/blob/master/words_alpha.txt

Text distance:
https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7

- ***What was the most difficult part of the assignment?***

How to define the heuristic cost function in the A* search; I also encounter the issue: error occurs in the comparison: if neighbor not in actual_costs or tentative_actual_cost < actual_costs[neighbor]
This error occurs because the heapq module requires the elements being pushed onto the heap to be comparable. By default, Python does not know how to compare instances of custom classes like WordPathNode.
How to fix: define how WordPathNodes should be compared for less-than, add this method in the WordPathNodes class.

```
  def __lt__(self, other):
     return node_cost(self) < node_cost(other)
```

define a nod_cost method: def node_cost(node):

```
   return len(node.get_path_to_root()) - 1 + heuristic(node.word, end_word)
```

```
# Instead of directly comparing nodes, use the node_cost function as a key
heapq.heappush(priority_queue, (node_cost(WordPathNode(neighbor, current_node)),
WordPathNode(neighbor, current_node)))
```

- ***What was the most rewarding part of the assignment?***
  Compare BFS, DFS, IDS, and A* search efficiency using characters comparison case, better understand how to choose heuristic

- ***What did you learn doing the assignment?***
  The difference between these searching algorithms, how to use different data structure in the search, eg. queue in BFS and A* search, stack in DFS and IDS.

- ***Constructive and actionable suggestions for improving assignments, office hours, and class time are always welcome***.
  It would be nice to go over all the slides at class, could use some extra virtual classes to make up for the holidays.