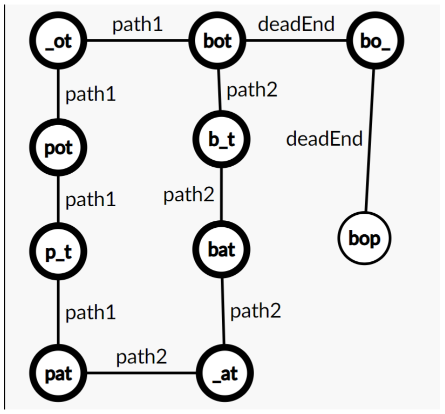
Brute Force Solution:
For the most basic solution, we can loop through every letter of our starting word, changing them one by one, checking if the word is in the dictionary, and checking if the end result is equal to the ending word. We can use an amount of nested for-loops equal to the length of our word, changing a letter at each position, and checking with the ending string at every step.

This will have a time complexity of O(26^n), where n is the length of the word. This exponential time complexity is extremely slow, and we should find a way to make the solution more efficient.

Optimized Solution Idea:
We can think of a series of words as a bidirectional graph, with each word being a node, and each edge representing that two words can be transformed into each other. To optimize the amount of operations, we can have a word with a blank letter as a parent (e.g c_st, _ate, lo_), and each word that can arise from filling in the blank with one letter as the children of said parent. We can store the parent as a key in hashmap, and our children as the value of the key (in an arraylist)

Key = “_ost”, Value = [“host”, “post”, “cost”, “fost”, “jost”, “lost”]

Our graph will look like this:
Start → Bot, End → Pat



Here, we can clearly identify paths that would lead to an optimal solution, and those that will not lead to an optimal solution or any solution at all.
To search for a path to the ending word, we should find all of the partitions/variations of our starting word with one letter replaced, so that we can locate these keys in the map.

Start → bank
Variations → _ank, b_nk, ba_k, ban_

These can be generated by looping through the word’s indices, replacing one letter with an underscore, and then adding the string into the map with a new arraylist as the value.

To perform the actual search, we can start at each one of the variations of our start word, look through the values of the variation to find other words, and then generate variations of each of those words. Notice that the graphical representation of the problem shows that we should continue searching along the graph until we reach our solution, which is a representation of a graph/tree traversal algorithm, such as Breadth First Search. We can perform a breadth first search to get from the start to the end. To get the explicit path, we can store the parent of each word in the graph that the BFS traverses through in a map, and we can recursively search through the keys from end to start in the map (since the BFS search can be represented as a tree, each word will only have one parent).

The overall time complexity of the algorithm is O(nm^2), where n is the number of words in the dictionary, and m is the length of the start/end words.

Pseudocode:

```
Create all variations of words in the given dictionary by looping through each letter and removing it
    Create a key of the variation with an array of words that apply to it as a value
    Create new arraylist if variation has not been created before
    Add the word which follows the variation into the variation's list
Output error if graph cannot be built
Check all variations of the starting word
    Check if any of the words in each of the variation lists includes the end word
        Add end word with last word into a map
        Add end word into a stack
        Add parent word of end word into stack
        Repeat for each parent of current word until start word is added into stack
        Print out path by taking each word from the top of the stack until empty
    Add the start word and next words into a map if valid and not already checked
    Repeat process with each word in the variation lists until end word is found
Print error message if end word cannot be found
```

Data Structure and Algorithm Types:

Collection of all words and their connections is a **non-directed graph.**

- Each word is a node, and they share an edge if one can be changed into the other

Connections stored after pruning words using BFS model a **tree.**

- Each node now only has one parent, the word which it was changed from. There is only one instance of this throughout the structure.

HashMaps were used to store connections between words

- One map was used to store adjacency between an individual word and all other words in the dictionary
- Another map was used to word to word connections from a BFS originating from the start word

A **HashSet** was used to store words in the dictionary of the same length as the start/end words

- Each word is unique, often checking that a new word is in the dictionary

A **Stack** was used to store the reversed path from traversing the tree's map backwards, which was used to output the answer

- Since the path is inversely traced, we must order the words such that the end will be at the bottom, and the start will be on top. The LIFO structure of the stack allows this

The main algorithm used was a **breadth-first search of a graph**. This is a tree-traversal algorithm.

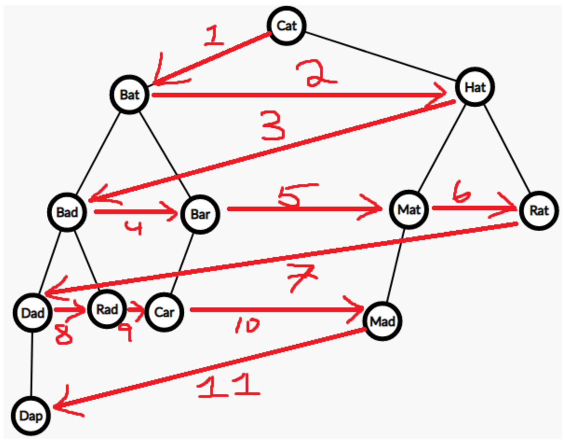
- Explained in solution idea

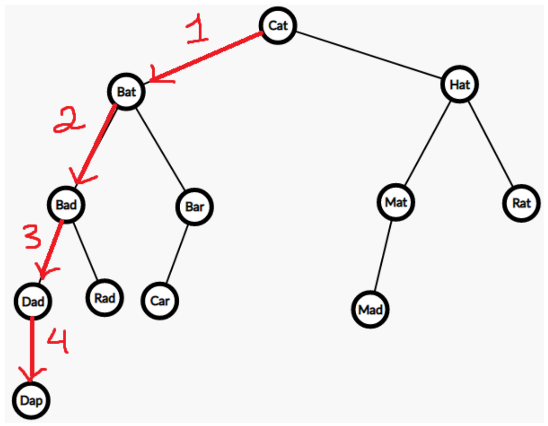
A **simple recursive algorithm** was used to trace the path of the graph.

- Repeated traversal through static elements allows for simple recursion implementation.

Implementation/Compile-time/Run-time Issues:

- Creating 2 variations of each word takes quadratic time, thus this is the slowest operation in the program
    - This can be optimized by a constant factor if a list of already existing variations is memoized, however this is a large waste of memory space
- Breadth First search will take each word row by row, whereas Depth First Search will check path by path
    - Using BFS will give more optimal solutions for a large majority of cases, but there are some rare cases where DFS will be faster
    - Example:





(BFS: Left, DFS: Right. DFS finds a lucky pathway that goes straight to the end with 4 iterations, whereas BFS searches all 11 nodes of the graph)

- An additional data structure was necessary to print the path of the search
    - The path was retraced using the knowledge that the BFS would prune the graph into a tree, thus each word would only have one parent
    - We cannot start from the beginning word since it has no parents, thus we must start from the end word, which means that the path will be reversed

- To orientate the path in the correct order, we must store it in some sort of collection. A stack is the best way to do this, since the earlier elements are at the end/bottom of the structure.

Test Case Outputs:

FILE INPUT (dictionary): LetterMutationTest.txt

Word Length	Start	End	Expected Output
3	Cat	Rat	CAT -> RAT -> RAD -> RED
4	Heat	Crab	HEAT -> PEAT -> PRAT -> DRAT -> DRAB -> CRAB
5	Cream	Sweet	CREAM -> CREAK -> CREEK -> CHEEK -> CHEEP -> SHEEP -> SWEEP -> SWEET
	Start	Enter	START -> STARS -> SOARS -> SOAKS -> SOCKS -> POCKS -> PACKS -> PACES -> PATES -> PATER -> EATER -> ENTER
	Angel	Devil	No Possible Solution!
6	Canyon	Wicked	CANYON -> CANTON -> CANTOR -> CANTER -> BANTER -> BANKER -> BACKER -> BICKER -> WICKER -> WICKED
	Ladder	Player	LADDER -> LANDER -> LENDER -> BENDER -> BENDED -> BEADED -> BLADED -> BLARED -> FLARED -> FLAYED -> PLAYED -> PLAYER
	Orange	Faster	No Possible Solution!
7	Ballous	Callous	BALLOUS -> CALLOUS
	Backers	Vulgate	No Possible Solution!