Word Blazer Algorithm report

Mohamed Ben Hadj Nasr
mmed.benhadjnasr@gmail.com

University of Tunis El Manar — Higher Institute of Computer Science
February 19, 2025

Introduction

This report presents a detailed analysis of the algorithms implemented in the Word Blazer game project, a fun word-formation game where players navigate through a grid of letters to create valid words while maximizing their score.
Our implementation focuses on optimizing several key algorithmic components to ensure smooth gameplay and reliable performance. Throughout this report, we will examine the fundamental algorithms that power the game's core features, including:

- Optimized pattern matching using the Aho-Corasick algorithm.
- Building the labyrinth and checking the validity of said labyrinth using DFS.

1 Aho-Corasick algorithm

To determine whether the player has created a valid word from the dictionary, we will employ the Aho-Corasick algorithm, which leverages graph-based automata to optimize the checking process. This approach reduces the time complexity of checking if we found a word when we add a character from $O(n*m)$ to $O(1)$, where n is the length of the input text, m is the total number of dictionary words.

1.1 Theoretical viewpoint

The algorithm constructs a trie (prefix tree) from the dictionary words and augments it with failure links, transforming it into a finite state machine. This allows us to efficiently traverse the graph and validate the player's word in near-linear time, significantly improving performance over brute-force methods.

1.2 Trie

A trie (also called a prefix tree) is a tree-like data structure used to store and retrieve strings efficiently. It is particularly useful for tasks like dictionary lookups, autocomplete, and pattern matching. Here's a detailed explanation of how a trie is constructed:

Structure of a Trie

- Nodes: Each node in the trie represents a single character of a string.
- Edges: The edges between nodes represent the connections between characters in the strings.
- Root Node: The trie starts with an empty root node, which does not represent any character.
- End-of-Word Marker: Nodes that represent the end of a valid word are marked (e.g., with a boolean flag or a special symbol).

Construction Process

1. Start with the Root:
    - Begin with an empty root node. This node has no character associated with it and serves as the starting point for all words.
2. Insert Words:
    - For each word in the dictionary, start at the root node and process the word character by character.
    - For each character in the word:
        - Check if the current node has a child node corresponding to the character.
        - If the child node exists, move to that node.
        - If the child node does not exist, create a new node for the character, add it as a child of the current node, and move to the new node.
    - After processing all characters of the word, mark the last node as the end of a valid word.

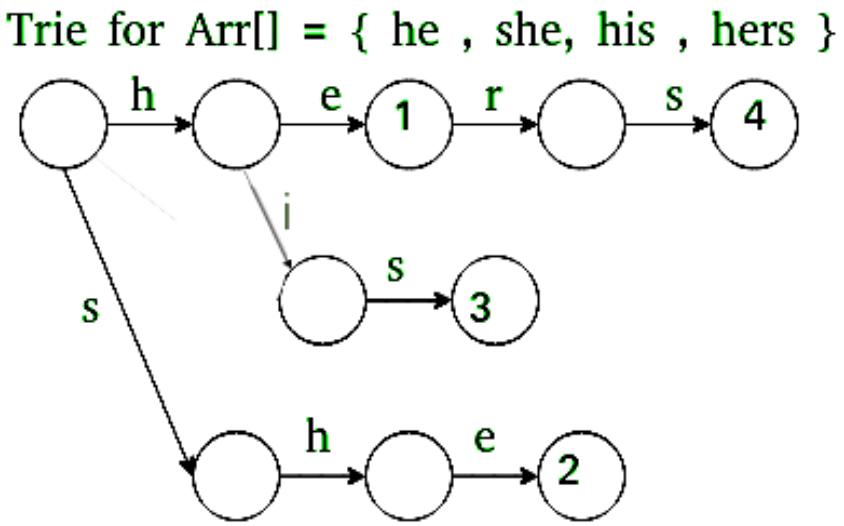

Figure 1: Trie

1.3 Failure Links

A failure link in the Aho-Corasick algorithm is a pointer from one node to another, representing the longest proper suffix of the current node's path that is also a prefix of another pattern in the trie. If a mismatch occurs, the algorithm follows the failure link to continue searching from the new node.

Construction process

1. Initialize the Root:
    - The root node has no failure link.
    - All children of the root node initially have their failure links pointing back to the root.
2. Use Breadth-First Search (BFS):
    - Traverse the trie level by level using BFS.
    - For each node, compute its failure link based on its parent's failure link.
3. Compute Failure Links For a given node u with character c:

- Let v be the parent of u.
- Follow the failure link of v to a node f.
- Check if f has a child node w corresponding to c.
  - If w exists, set the failure link of u to w.
  - If w does not exist, recursively follow the failure link of f until you reach the root or find a node with a child w.
- If no such node is found, set the failure link of u to the root.

4. Handle Output Links:
5. If a node represents the end of a valid word, store the word in the node.
6. If a failure link points to a node that also represents the end of a valid word, link the output of the current node to that node (to capture all matches).

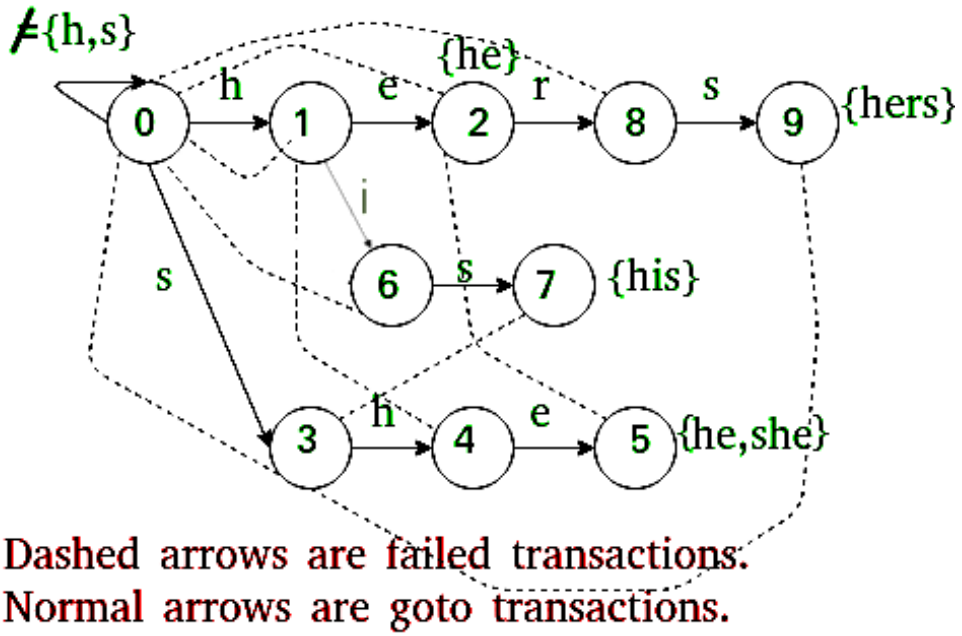

Figure 2: failure links example

2 Labyrinth Building

The algorithm generates a word search puzzle by randomly placing dictionary words in various directions across a grid. It uses an Aho-Corasick automaton to ensure that all character placements can potentially form valid words from the dictionary.

2.1 Word Generation Process

The process begins with an empty grid of the desired dimensions. Starting from a randomly selected empty cell, the algorithm systematically fills the grid while maintaining word validity. At each cell, the process begins at the root state of the Aho-Corasick automaton. The algorithm obtains the set of valid characters from the current automaton state and randomly selects one character from this set. This selected character is then placed in the current cell.

After placing a character, the algorithm randomly selects one of the eight possible directions, including horizontal, vertical, and diagonal orientations. It then moves to the adjacent cell in the chosen direction and updates the automaton state based on the character that was just placed. This process continues cell by cell until the entire grid has been filled.

this is a generated maze using only the words [fire,love,find,word] we can clearly see how it's fille with these words:

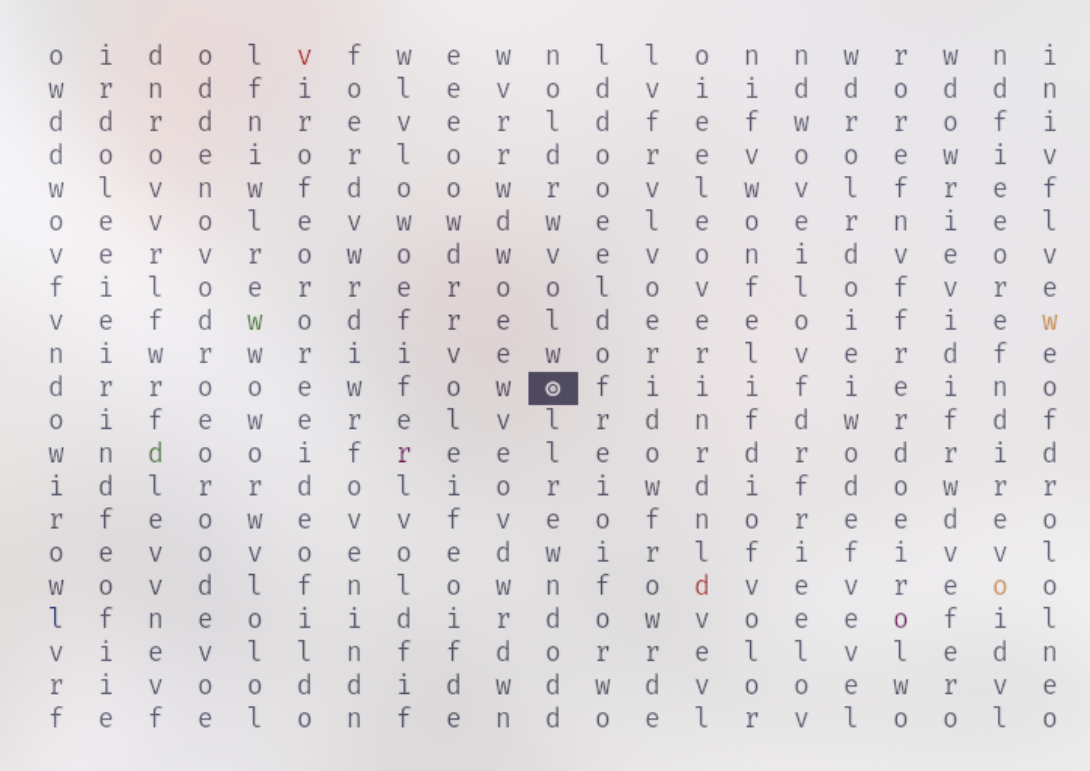

Figure 3: example of maze generation

2.2 Wall genearation

After the word grid is complete, the algorithm adds complexity through wall generation. Starting from a selected node in the grid, the wall generation process simultaneously extends in two opposite directions. At each step of wall construction, there is a 50% probability of continuing straight in the current direction. Additionally, there is a 25% chance of slightly tilting the wall to either side, creating more organic and unpredictable barriers. and then I use DFS to check if a path exists between the player and exit.

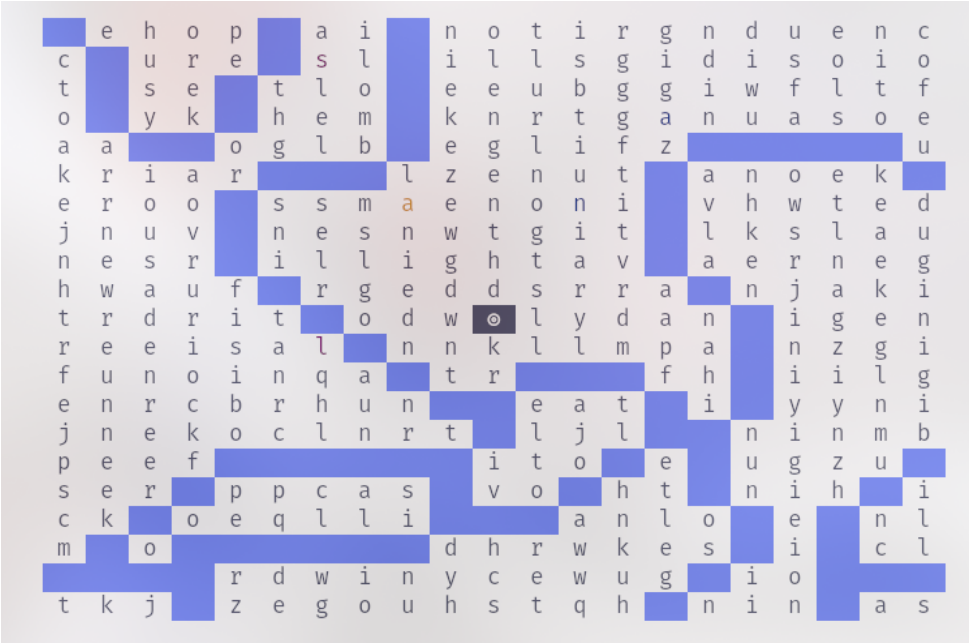

Figure 4: example of how the walls are built

3 Implementation Benefits

- The Aho-Corasick automaton ensures fast and reliable character validation during puzzle generation.
- Every puzzle is guaranteed to be solvable while maintaining unique and engaging challenges.
- High randomization in both word placement and wall patterns creates virtually unlimited unique puzzles.
- The bidirectional wall system creates complex mazes while guaranteeing all areas remain accessible.
- Difficulty scaling is achieved through both word placement complexity and wall density adjustments.