# Word Ladder Game Report

## 1. Approach & Objectives

### Game Overview

The **Word Ladder Game** challenges players to transform a **start word** into a **target word** by changing **one letter at a time**. Each intermediate word must be valid (i.e., found in a dictionary).

To assist players, the game includes **AI-powered search algorithms** (**BFS, UCS, and A***), which provide hints and compute optimal paths.

---

### Key Features

- **Three Difficulty Modes:**
  - Beginner Mode – Simple word transformations.
  - Advanced Mode – Medium difficulty, longer words.
  - Challenge Mode – Includes banned letters and banned words for added complexity.
- **Real-time Visualization:**
  - The game renders a transformation graph using Graphviz.
  - Shows player's path, AI-suggested paths, and banned words/letters.
- **AI-Powered Word Solving:**
  - Players can request hints from BFS, UCS, and A* algorithms.
  - AI solutions are color-coded and visually represented.

---

## 2. Search Algorithm Implementation & Comparisons

### 1. Breadth-First Search (BFS)

**Purpose:** Finds the shortest path from the start word to the target word, measured in the number of moves (not cost).

**How it Works:**

- Explores all possible words at depth  before moving to .

n

n+1

- Uses a **FIFO queue (First-In, First-Out)**.
- Ensures the shortest transformation sequence is found.

```
def bfs(start_word, target_word, word_dict, banned_characters=None, mov
e_limit=None):
    queue = deque([[start_word]])
    visited = set([start_word])
    while queue:
        path = queue.popleft()
        current_word = path[-1]

        for neighbor in get_neighbors(current_word, word_dict, banned_chara
cters):
            if neighbor == target_word:
                return path + [neighbor], visited
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])
    return None, visited
```

**Complexity:** O(b^d), where:

- b = Branching factor (number of valid word transformations per step).
- d = Depth of the shortest path.

**Pros:**

- Guarantees the shortest path.
- Simple to implement.

**Cons:**

- Inefficient for large word spaces because it explores all possibilities at each level.

---

## 2. Uniform Cost Search (UCS)

**Purpose:** Finds the lowest-cost path, where the cost = number of moves.

**How it Works:**

- Uses a **priority queue**, where the path with the lowest cumulative cost is expanded first.
- Always expands the cheapest path first (i.e., the fewest transformations).
- Uses a cost function  to track the path cost.

  g(n)

```
def ucs(start_word, target_word, word_dict, banned_letters=None, move_li
mit=None):
    frontier = []
    heapq.heappush(frontier, (0, start_word, [start_word]))

    while frontier:
        current_word_cost, current_word, path = heapq.heappop(frontier)
        if current_word == target_word:
            return path, visited

        for neighbor in get_neighbors(current_word, word_dict, banned_letter
s):
            new_cost = current_word_cost + 1
            heapq.heappush(frontier, (new_cost, neighbor, path + [neighbor]))
    return None, visited
```

**Complexity:** O(b^d) – Similar to BFS but optimized for cost-awareness.

**Pros:**

- Guarantees the optimal path in terms of cost.

**Cons:**

- Explores more nodes than A* since it does not use heuristics.

---

## *3. A Search Algorithm*

**Purpose:** Balances shortest path search (BFS) with cost optimization (UCS) by using heuristics.

**How it Works:**

- Uses a **priority queue**, where paths are sorted by:

f(n)=g(n)+h(n)

- **g(n)** = Cost from the start word.
- **h(n)** = Heuristic: Number of **letter mismatches** between the current word and the target word.

```
def heuristic(word):
    return sum(c1 != c2 for c1, c2 in zip(word, target_word))

def a_star(start_word, target_word, word_dict, banned_letters=None, move_limit=None):
    frontier = []
    heapq.heappush(frontier, (heuristic(start_word), 0, start_word, [start_word]))

    while frontier:
        f, g, current_word, path = heapq.heappop(frontier)
        if current_word == target_word:
            return path, visited

        for neighbor in get_neighbors(current_word, word_dict, banned_letters):
            new_g = g + 1
            new_f = new_g + heuristic(neighbor)
            heapq.heappush(frontier, (new_f, new_g, neighbor, path + [neighbor]))
    return None, visited
```

**Complexity:** O(b^d) but much faster than UCS in practice due to heuristics.

**Pros:**

- Best balance between BFS and UCS.
- Faster than UCS due to heuristic-based search.

**Cons:**

- Requires tuning of heuristics for optimal performance.

---

# 3. Comparative Analysis of Search Algorithms

| Algorithm | Shortest Path? | Cost Optimized? | Performance (Speed) | Space Complexity |
|---|---|---|---|---|
| **BFS** | Yes | No | Slow for large words | O(b^d) |
| **UCS** | Yes | Yes | Slower than A* | O(b^d) |
| **A*** | Yes | Yes | Fastest & Most Efficient | O(b^d) |

**Key Takeaways:**

1. **BFS** is fast for small words, but inefficient for large transformations.
2. **UCS** ensures the optimal path but expands too many nodes.
3. **A*** is the best approach as it combines speed and optimality.

---

# 4. Visualization & UI

**Graph Representation:**

- **Green Nodes** → Player's current path.
- **Pink Nodes** → AI-suggested path.
- **Red Nodes** → Words with banned letters.

**Terminal UI Enhancements:**

- PyFiglet for ASCII art headers.
- Color-coded blocks using `colorama`.

---

# 5. Conclusion & Recommendations

- *A is the best algorithm*as it balances search efficiency and optimal cost.
- **Graph-based visualization** improves user experience.
- **Challenge Mode makes the game more interactive** with banned words/letters.

**Future Improvements:**

- Implement **bidirectional search** for faster solutions.
- Add **machine learning-based heuristics** for better AI hints.

This report provides a detailed analysis and justification for AI-assisted word transformation.