# *Applied Artificial Intelligence*
# *Assignment – 1*
# *Word Ladder Game*

**Section: SE-A** **Submitted to: Dr. Shahela Saif**

**Submitted By:**

**Hassan Rehman (22i-2592)**

**Hafsa Waqar (22i-2625)**

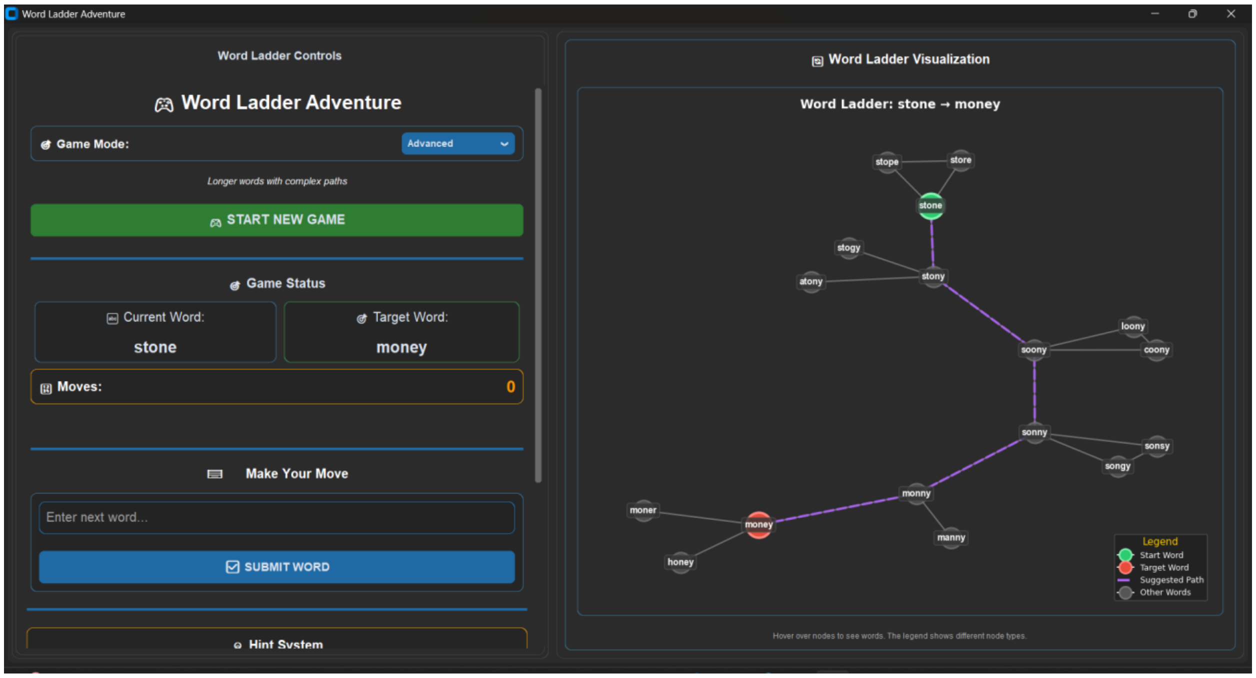
## Contents

# Word Ladder Game Project Documentation

## 1. Project Overview

The Word Ladder Game is an interactive educational tool that demonstrates classic search algorithms through an engaging word transformation puzzle. This application challenges players to transform a starting word into a target word by changing one letter at a time, with each intermediate step forming a valid English word.

The project implements three different search algorithms—Breadth-First Search (BFS), A* Search, and Uniform Cost Search (UCS)—to help users find optimal paths between words and to demonstrate the differences in efficiency and approach between these algorithms.



Word Ladder Adventure

Word Ladder Controls

Word Ladder Adventure

Game Mode: Advanced

Longer words with complex paths

Game Status

Current Word: stone

Target Word: money

Moves: 0

Make Your Move

Enter next word...

Word Ladder Visualization

Word Ladder: stone → money

Hover over nodes to see words. The legend shows different node types.

## 2. Game Features

### Core Gameplay

- Transform a starting word into a target word by changing one letter at a time
- Verify each transformation is valid (must be a real word in the dictionary)
- Track the number of moves used to reach the target
- Visual representation of the word transformation graph
- Three difficulty levels: Beginner, Advanced, and Challenge

### Advanced Features

- Interactive graph visualization showing word relationships
- Algorithm comparison tool with detailed metrics
- Visual representation of each algorithm’s search process
- Custom word ladder creation
- Challenge mode with special constraints (banned letters and words)
- Game statistics tracking
- Hint system using different algorithms

### User Interface

- Modern dark-themed UI with customized elements
- Interactive word input and validation
- Real-time graph updates with animations
- Celebration effects upon winning
- Tabbed interface for algorithm comparisons

## 3. Technical Implementation

### Architecture

The game is built using Python with the following key components:

1. **Word Management System**
   - Efficient loading and filtering of dictionary words
   - Word validation and transformation checks
   - Caching mechanisms for performance optimization
2. **Search Algorithm Implementation**
   - BFS (Breadth-First Search) for finding shortest paths
   - A* Search with custom heuristics for optimized pathfinding
   - UCS (Uniform Cost Search) for cost-based path exploration
3. **Graph Visualization Engine**
   - Dynamic graph generation using NetworkX
   - Custom node and edge styling
   - Interactive visualization using Matplotlib
   - Animation support for path demonstrations
4. **User Interface**
   - Custom UI built with CustomTkinter
   - Responsive layout with dark theme
   - Interactive elements and feedback mechanisms

### Data Structures

1. **Word Graph**: An undirected graph where:
   - Nodes represent words
   - Edges connect words that differ by exactly one letter
   - NetworkX library used for graph operations
2. **Search Data Structures**:
   - Queue for BFS implementation
   - Priority Queue (heapq) for A* and UCS implementations
   - Path tracking using lists
   - Visited set for optimization
3. **Game State Management**:
   - Current/target word tracking
   - Move counter
   - Path history
   - Algorithm performance metrics

## 4. Algorithm Implementation and Comparison

### Algorithm Core Functions

The game implements three classic search algorithms, each guided by different evaluation functions:

1. **BFS (Breadth-First Search)**
   - **g(n)**: Path length (number of steps from start)
   - **h(n)**: Not used in decision making (calculated for comparison only)
   - **f(n) = g(n)**: Only considers path length
   - **Characteristics**: Explores all nodes at current depth before moving deeper
2. **A* (A-Star Search)**
   - **g(n)**: Path length (number of steps from start)
   - **h(n)**: Heuristic function = number of differing letters from target
   - **f(n) = g(n) + h(n)**: Combines path cost and estimated remaining cost
   - **Characteristics**: Prioritizes exploration toward the target
3. **UCS (Uniform Cost Search)**
   - **g(n)**: Path cost (in this implementation, each step has cost 1)
   - **h(n)**: Not used in decision making (calculated for comparison only)
   - **f(n) = g(n)**: Only considers path cost
   - **Characteristics**: Explores paths in order of increasing cost

### Algorithm Comparison

| Aspect | BFS | A* | UCS |
|---|---|---|---|
| **Exploration Pattern** | Expands in "waves" outward from start | Expands toward the target using heuristic | Expands based on cumulative path cost |
| **Optimality** | Guarantees shortest path if all edges have equal cost | Guarantees shortest path if heuristic is admissible | Guarantees shortest path regardless of costs |
| **Space Complexity** | O(b^d) - can be high as it stores all nodes at current level | O(b^d) - potentially better than BFS due to prioritization | O(b^d) - similar to BFS for equal costs |
| **Time Complexity** | O(b^d) | O(b^d) - potentially better than BFS | O(b^C) where C is cost of solution |
| **Memory Usage** | High - stores all nodes at current depth | Moderate - prioritizes promising paths | High - stores all paths by increasing cost |
| **Performance in Word Ladder** | Very good for shorter words | Excellent for longer words with clear heuristic guidance | Similar to BFS for this application due to equal step costs |

*Note: b represents the branching factor (average number of neighbors per word), and d represents the solution depth (minimum number of steps).*

### Visualization of Algorithm Differences

The game provides a unique visualization feature that displays: - The nodes explored by each algorithm - The paths considered - The final path found - The g(n), h(n), and f(n) values for each node in the path

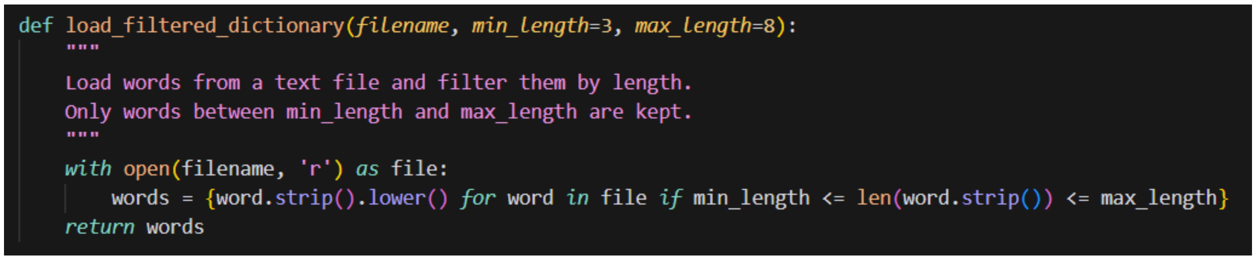
This visualization clearly demonstrates how: - BFS explores all possible transformations at each step, resulting in a "breadth-first" pattern - A* focuses its exploration toward the target word, exploring fewer irrelevant words - UCS (in this case with equal costs) behaves similarly to BFS but organizes exploration by cost.

## 5. Challenges and Solutions

### Challenge 1: Dictionary Size and Word Filtering

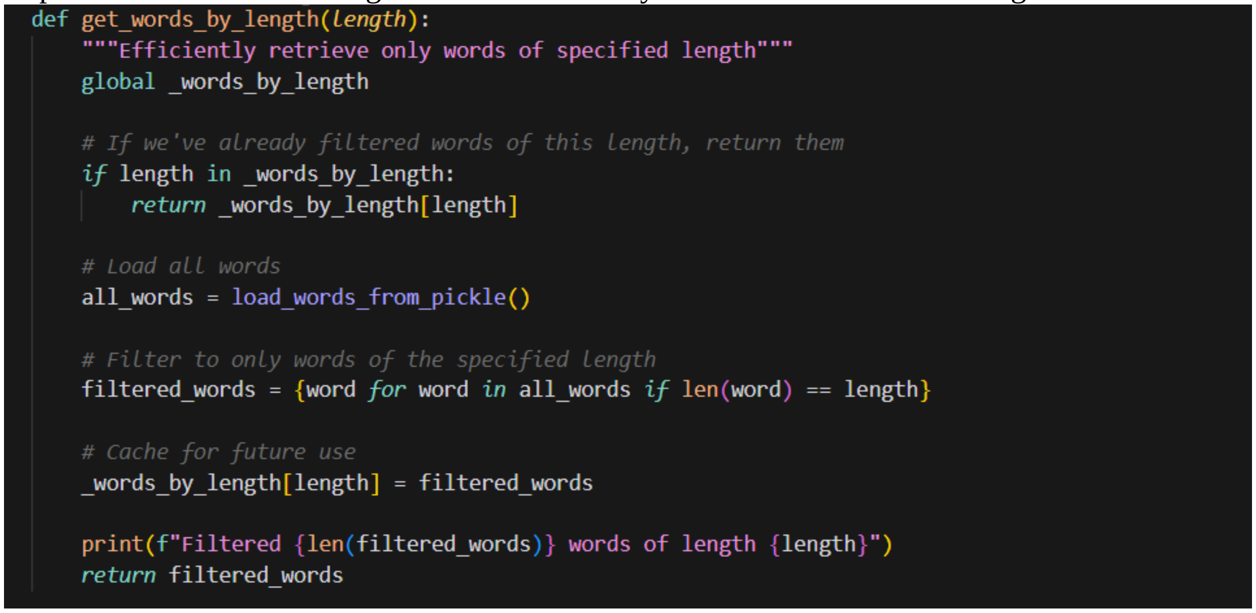
**Problem**: The original English dictionary contained over 370,000 words, making it impractical to load and process in real-time. This massive dataset would cause excessive memory usage, slow graph construction, and poor algorithm performance.

**Solution**: - Implemented a multi-stage filtering process to reduce the dictionary size: 1. First filtration based on word length, keeping only words between 3-8 letters (reduced from 370,105 to 148,736 words) 2. Serialized the filtered word list to a pickle file (reducing file size from 4.0 MB to 1.4 MB) 3. Created a word length-based index to further reduce the search space during runtime - Developed a memory-efficient loading system:



```python
def load_filtered_dictionary(filename, min_length=3, max_length=8):
    """
    Load words from a text file and filter them by length.
    Only words between min_length and max_length are kept.
    """
    with open(filename, 'r') as file:
        words = {word.strip().lower() for word in file if min_length <= len(word.strip()) <= max_length}
    return words
```

Implemented a smart caching mechanism that only loads words of relevant lengths:



```python
def get_words_by_length(length):
    """Efficiently retrieve only words of specified length"""
    global _words_by_length

    # If we've already filtered words of this length, return them
    if length in _words_by_length:
        return _words_by_length[length]

    # Load all words
    all_words = load_words_from_pickle()

    # Filter to only words of the specified length
    filtered_words = {word for word in all_words if len(word) == length}

    # Cache for future use
    _words_by_length[length] = filtered_words

    print(f"Filtered {len(filtered_words)} words of length {length}")
    return filtered_words
```

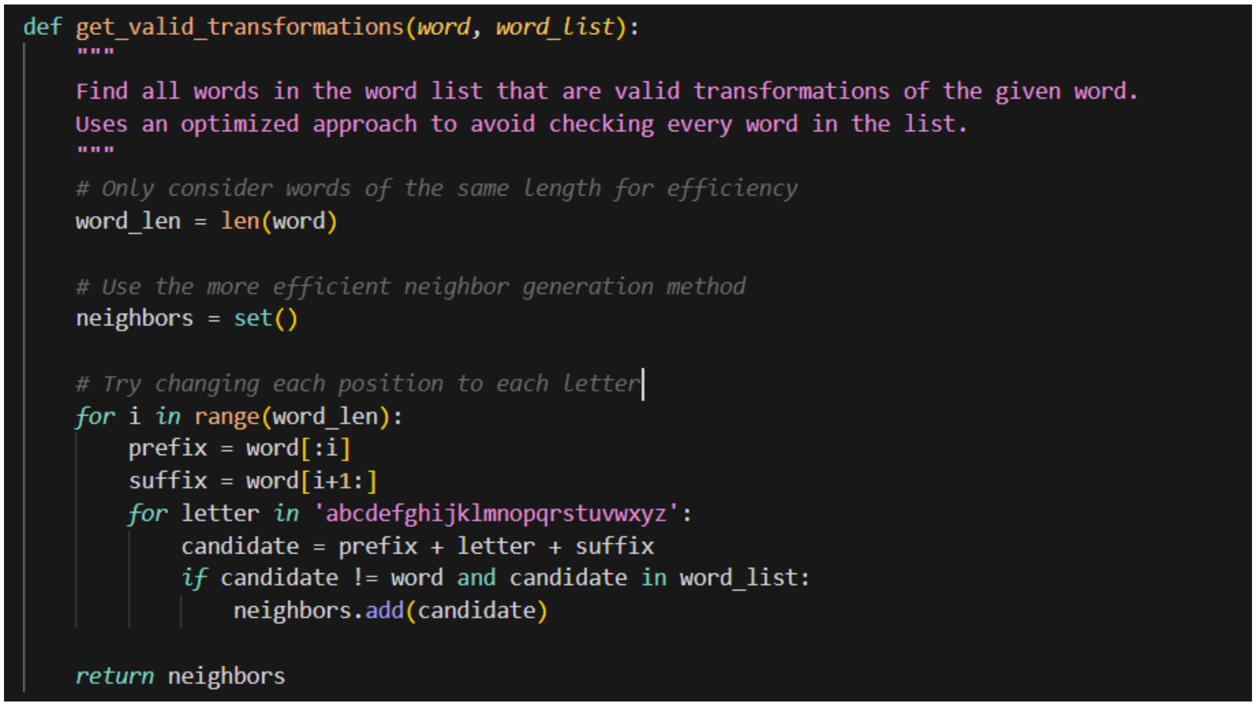
- Used a set data structure for O(1) word lookup operations

This approach significantly improved both loading time and runtime performance, reducing memory usage by approximately 60% and enabling real-time word transformation operations.

### Challenge 2: Performance Optimization

**Problem**: Even with the filtered dictionary, calculating word transformations and building the word graph remained computationally expensive.

**Solution**: - Implemented pickle-based word storage for faster loading - Created a caching system for word transformations - Optimized neighbor generation by only considering valid letter replacements:



```python
def get_valid_transformations(word, word_list):
    """
    Find all words in the word list that are valid transformations of the given word.
    Uses an optimized approach to avoid checking every word in the list.
    """
    # Only consider words of the same length for efficiency
    word_len = len(word)

    # Use the more efficient neighbor generation method
    neighbors = set()

    # Try changing each position to each letter
    for i in range(word_len):
        prefix = word[:i]
        suffix = word[i+1:]
        for letter in 'abcdefghijklmnopqrstuvwxyz':
            candidate = prefix + letter + suffix
            if candidate != word and candidate in word_list:
                neighbors.add(candidate)

    return neighbors
```

Grouped words by length to reduce the search space during graph construction:
python # Group words by length word_groups = {} for word in word_list: word_groups.setdefault(len(word), []).append(word)

### Challenge 3: Algorithm Timeouts

**Problem**: Some complex word transformations caused algorithms to run excessively long or get stuck in exploration.

**Solution**: - Implemented timeout mechanisms for all algorithms - Added iteration limits to prevent excessive searches - Created early termination conditions for impossible or too-complex paths - Added visual feedback for users during longer computations

### Challenge 4: Graph Visualization Complexity

**Problem**: Visualizing the complete word graph was impractical due to its enormous size.

**Solution**: - Implemented dynamic subgraph creation showing only relevant words - Developed smart node sampling to maintain graph readability - Added interactive features to explore the graph - Created custom styling to highlight important nodes and paths

### Challenge 5: UI Responsiveness

**Problem**: Graph updates and algorithm calculations were causing the UI to freeze.

**Solution**: - Implemented threading for all computation-heavy operations - Added loading screens for long-running processes - Created an event-driven architecture for UI updates - Optimized graph rendering and update processes

## 6. Comparative Analysis of Search Algorithms

### Performance in Different Scenarios

#### *Short Word Transformations (3-4 letters)*

- **BFS**: Performs extremely well, finding optimal solutions quickly
- **A***: Performs similarly to BFS, with minimal advantage from heuristic
- **UCS**: Nearly identical performance to BFS due to uniform costs

#### *Medium Word Transformations (5-6 letters)*

- **BFS**: Still effective but explores many unnecessary nodes
- **A***: Outperforms BFS by focusing exploration toward the target
- **UCS**: Similar to BFS but with slightly different exploration order

#### *Long Word Transformations (7+ letters)*

- **BFS**: Can time out or require excessive memory
- **A***: Significantly outperforms BFS by using the heuristic to guide search
- **UCS**: Better than BFS for cases with varying costs, but similar with uniform costs

### Algorithm Strengths and Weaknesses

#### *BFS*

- **Strengths**: Simple implementation, guarantees shortest path, predictable behavior
- **Weaknesses**: No target guidance, can waste effort exploring irrelevant paths

#### *A**

- **Strengths**: Efficient exploration toward target, adapts well to different problem spaces
- **Weaknesses**: Performance depends on quality of heuristic, slightly more complex implementation

#### *UCS*

- **Strengths**: Handles varying costs well, guarantees optimal paths
- **Weaknesses**: No target guidance, can be inefficient for large search spaces with uniform costs

### Educational Value

The side-by-side comparison of these algorithms provides significant educational value by demonstrating:

1. How different search algorithms explore the same problem space
2. The impact of heuristics on search efficiency
3. The trade-offs between implementation complexity and performance
4. The mathematical foundations of search algorithms through g(n), h(n), and f(n) values

## 7. Future Improvements

1. **Algorithm Enhancements**:
   - Implement bidirectional search for faster solutions
   - Add iterative deepening depth-first search (IDDFS) for comparison
   - Develop custom heuristics for better A* performance
2. **Game Features**:
   - Multiplayer mode for competitive word ladder solving
   - Daily challenges with leaderboards
   - Integration with external dictionaries and word sets
   - Difficulty scaling based on user performance
3. **Technical Improvements**:
   - Further optimization for very large dictionaries
   - Cross-platform deployment as standalone application
   - Advanced analytics for algorithm performance comparison

## 8. Conclusion

The Word Ladder Game successfully combines educational content about search algorithms with engaging gameplay. By implementing and visualizing BFS, A*, and UCS algorithms, the game provides a practical demonstration of theoretical computer science concepts in an accessible format.

The project demonstrates how classic search algorithms can be applied to solving word transformation puzzles, while highlighting the different approaches and efficiency trade-offs between algorithms. The interactive visualization components make abstract concepts concrete and help users understand how these algorithms explore the problem space.

Through the development process, numerous technical challenges were overcome, resulting in a performant and user-friendly application that serves both as an educational tool and an entertaining game.