PROJECT REPORT

DECEMBER 22, 2025

Title:

Word Ladder Adventure Game using Artificial Intelligence Search Algorithms

Group members

Name	Roll no
Talha khan	242210
Ahsan khan	242206
Muhammad Rehan Shoukat	242212

Submitted to: Aisha Sattar Choudhry

Air University Islamabad

Department of Creative Technology

Table of Content

Word Ladder Adventure Game using Artificial Intelligence Search Algorithms

Introduction

Word Ladder Adventure Game is an Artificial Intelligence based project that focuses on solving word transformation problems using classic AI search algorithms.
In this game, a player or an AI transforms a **start word** into a **target word** by changing **only one letter at a time**, while ensuring that every intermediate word is a valid dictionary word.

This project demonstrates practical implementation of **uninformed and informed search algorithms** such as BFS, DFS, UCS, Greedy Best First Search, and A* Search. The system also includes **custom gameplay**, **AI-assisted mode**, **difficulty levels**, and **visual graph generation** to better understand the search process.

Problem Statement

Finding the shortest and valid sequence of word transformations between two given words is a computationally expensive task when the dictionary size is large.
Manual solving is difficult and inefficient, especially when constraints such as restricted letters, banned words, and limited moves are applied.

The challenge is to:

- Validate each word transformation
- Ensure only one-letter changes
- Avoid invalid or repeated words
- Find an optimal path efficiently

Objectives

The main objectives of this project are:

- To design an interactive Word Ladder game
- To apply Artificial Intelligence search algorithms in a real-world problem
- To compare different search strategies for efficiency
- To implement both **manual (Custom)** and **AI-based** gameplay modes
- To visualize word transitions using graph representation
- To enhance understanding of AI problem-solving techniques

Problem Solution

The problem is solved by modeling the Word Ladder as a **state space search problem**, where:

- Each word represents a **state**
- Each valid one-letter transformation represents an **action**
- The goal is to reach the target word from the start word

The solution uses:

- Datamuse API for dictionary validation
- Graph-based traversal techniques
- AI search algorithms to explore possible paths
- Constraints handling (restricted letters, banned words, move limits)

Importance of Project:

This project is important because it provides a practical understanding of Artificial Intelligence search algorithms through an interactive Word Ladder game. It helps students learn how algorithms like BFS, DFS, UCS, GBFS, and A* work in real problem-solving situations.The project also demonstrates real-world API usage, algorithm optimization, and visualization of AI decision making, making complex concepts easy to understand.

.

Methodology

Step 1: Data Collection

- Datamuse API
- Python requests library

How it is implemented:

The Datamuse API is used to collect valid English words of a specific length. In the function fetch_word_set(length), a wildcard query (????) is generated based on the selected word length. The API returns a list of words, which are converted to uppercase and stored in a Python set to ensure uniqueness and fast lookup

Step 2: Word Validation

- Datamuse API
- API-based dictionary verification

How it is implemented:
Each user-entered word is validated using the validate_word(word) function.
A direct API query checks whether the word exists in the English dictionary. Only words confirmed by the API are accepted, preventing invalid or fake inputs.

Step 3: Transition Validation

- Character comparison logic
- Restricted letters and banned words filtering

How it is implemented:
The function is_valid_transition(current, next_word) ensures that a move is legal.
A transition is allowed only when:

- Word lengths are equal
- Exactly **one letter** is different
- Restricted letters are not used
- Banned words are excluded

Step 4: Neighbor Generation

- Alphabet iteration (A–Z)
- Transition validation logic

How it is implemented:
The neighbours() function generates all possible next words by changing one letter at

each position**.**
Each generated word is checked against:

- The dictionary (word_set)
- Transition validity rules

Step 5: Search Algorithms Implementation

The following algorithms are implemented:

• **BFS**: Finds shortest path using level-order traversal
• **DFS**: Explores deeply but may not give optimal path
• **UCS**: Uses uniform cost for optimal path
• **GBFS**: Uses heuristic for faster decision making
• **A***: Combines cost and heuristic for best performance

Step 6: Game Modes

Custom Mode:

- User manually enters each word
- Every move is validated
- No hints provided

AI Mode:

- User selects AI algorithm
- AI calculates solution path
- User can request hints

Step 7: Difficulty Levels

Conditional logic in main ()

Level	Word Length	Moves	Restrictions
Beginner	3	10	None
Advanced	4	15	None
Challenge	5	20	Banned & Restricted letters

restricted_letters = {'X', 'Z'}

banned_words = {"CRANE", "PLANE"}

Step 8: Visualization

What is used:

- Graphviz library
- Tree-based visualization

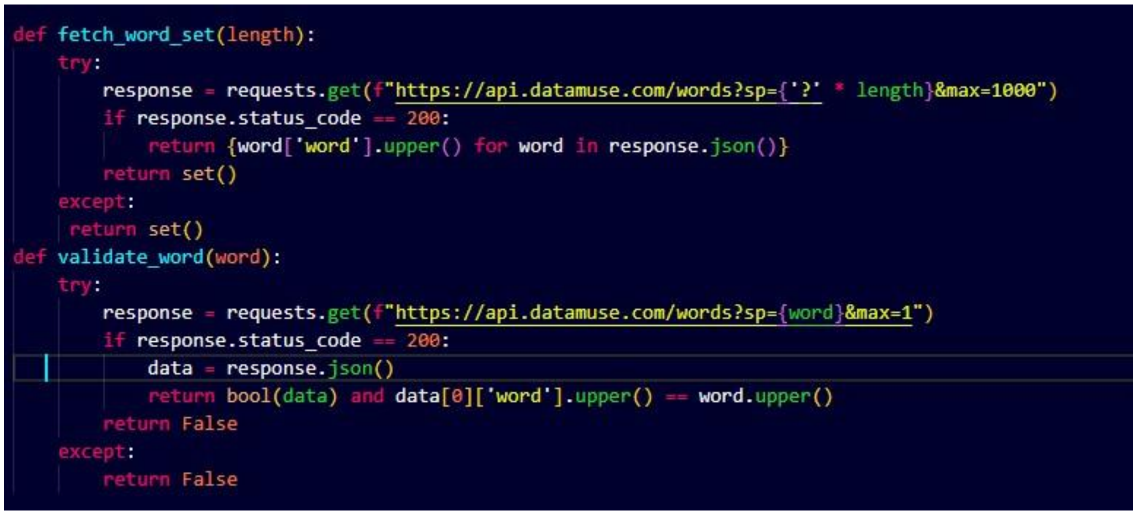
How it is implemented:
The visualize_tree() function draws:

- Correct path in blue solid lines
- Alternative paths in red dashed lines
- Player name and score

This helps users visually understand the Word Ladder solution.

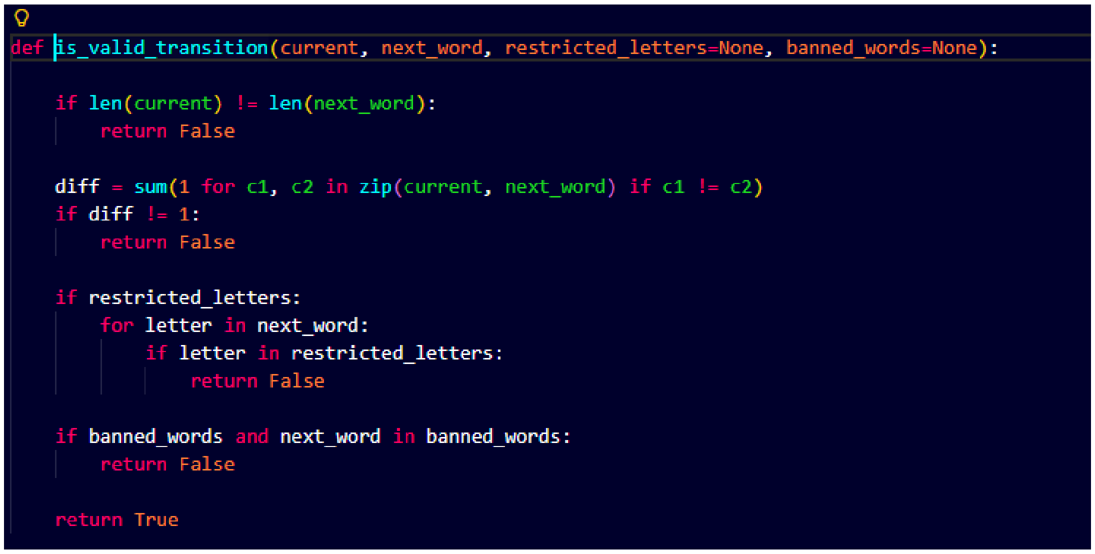
Main Modules:

Datamuse API Function:

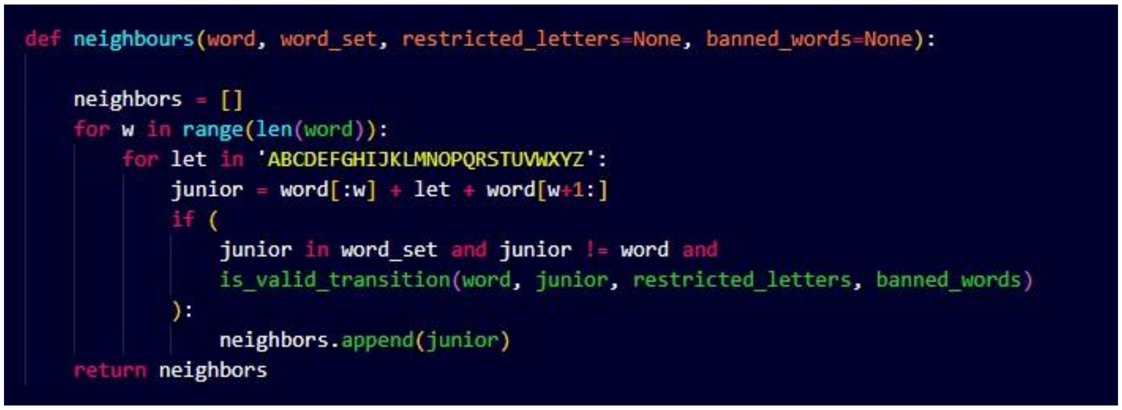


```python
def fetch_word_set(length):
    try:
        response = requests.get(f"https://api.datamuse.com/words?sp={'?' * length}&max=1000")
        if response.status_code == 200:
            return {word['word'].upper() for word in response.json()}
        return set()
    except:
     return set()
def validate_word(word):
    try:
        response = requests.get(f"https://api.datamuse.com/words?sp={word}&max=1")
        if response.status_code == 200:
            data = response.json()
            return bool(data) and data[0]['word'].upper() == word.upper()
        return False
    except:
        return False
```

Condition for Transition:

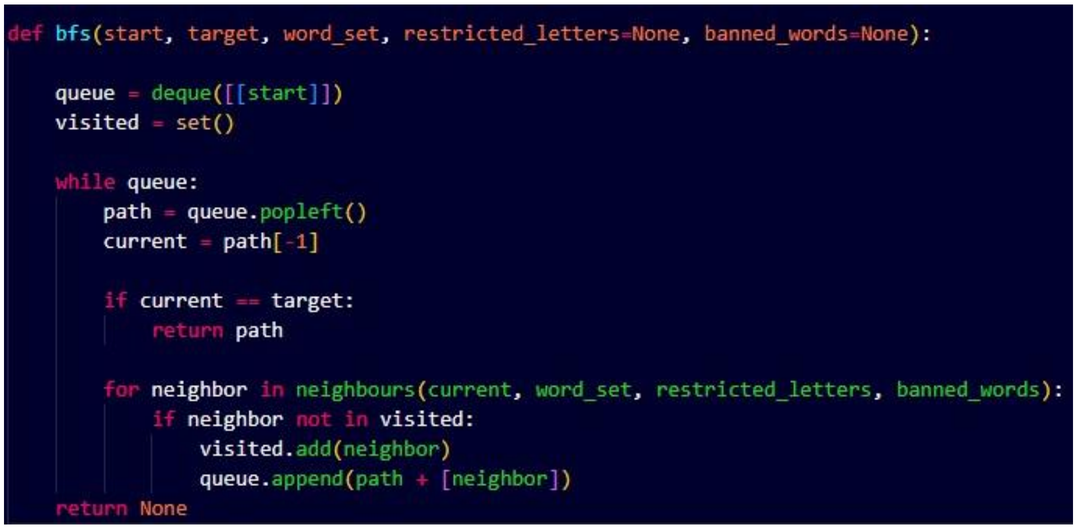


```
def is_valid_transition(current, next_word, restricted_letters=None, banned_words=None):

    if len(current) != len(next_word):
        return False

    diff = sum(1 for c1, c2 in zip(current, next_word) if c1 != c2)
    if diff != 1:
        return False

    if restricted_letters:
        for letter in next_word:
            if letter in restricted_letters:
                return False

    if banned_words and next_word in banned_words:
        return False

    return True
```



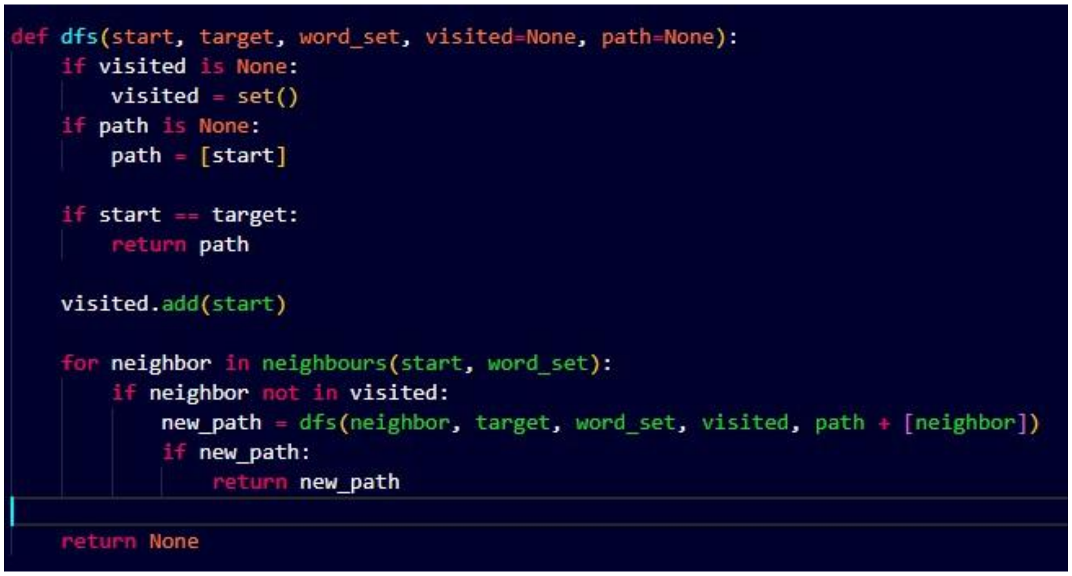
```
def neighbours(word, word_set, restricted_letters=None, banned_words=None):

    neighbors = []
    for w in range(len(word)):
        for let in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
            junior = word[:w] + let + word[w+1:]
            if (
                junior in word_set and junior != word and
                is_valid_transition(word, junior, restricted_letters, banned_words)
            ):
                neighbors.append(junior)
    return neighbors
```

BFS:

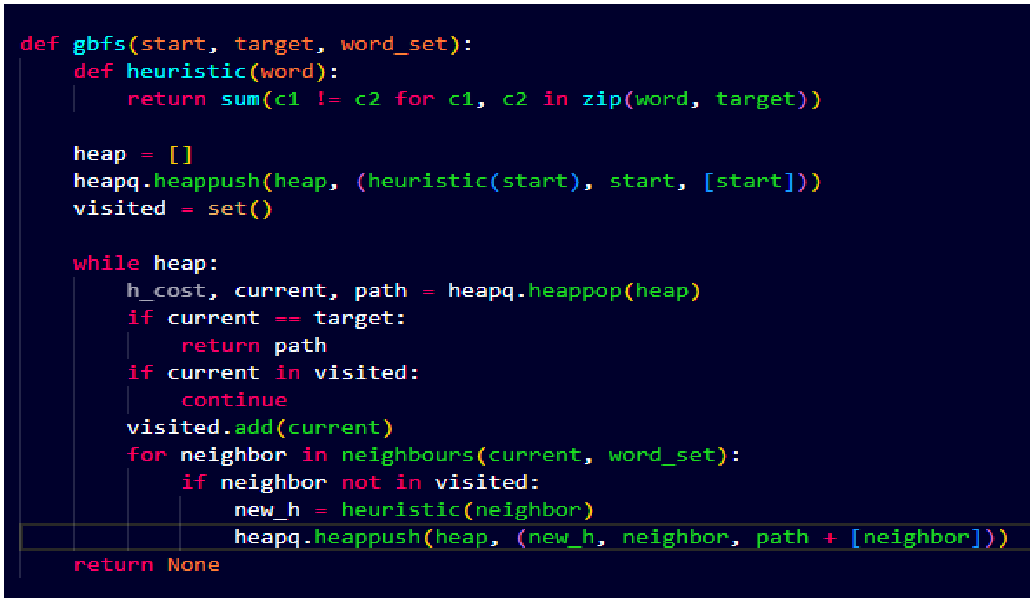


```
def bfs(start, target, word_set, restricted_letters=None, banned_words=None):

    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        current = path[-1]

        if current == target:
            return path

        for neighbor in neighbours(current, word_set, restricted_letters, banned_words):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])
    return None
```

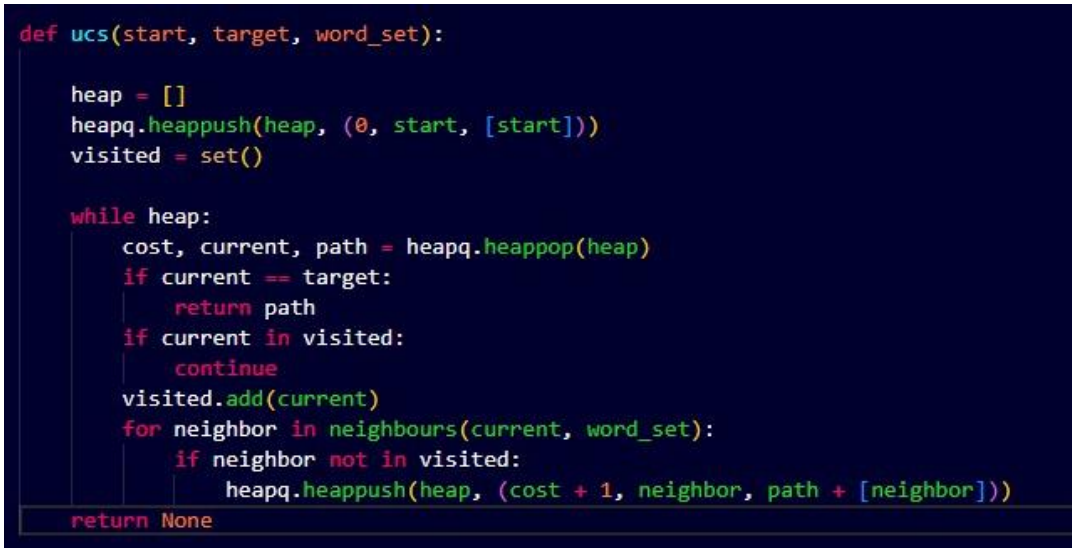
DFS:


```python
def dfs(start, target, word_set, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = [start]

    if start == target:
        return path

    visited.add(start)

    for neighbor in neighbours(start, word_set):
        if neighbor not in visited:
            new_path = dfs(neighbor, target, word_set, visited, path + [neighbor])
            if new_path:
                return new_path

    return None
```

GBFS:

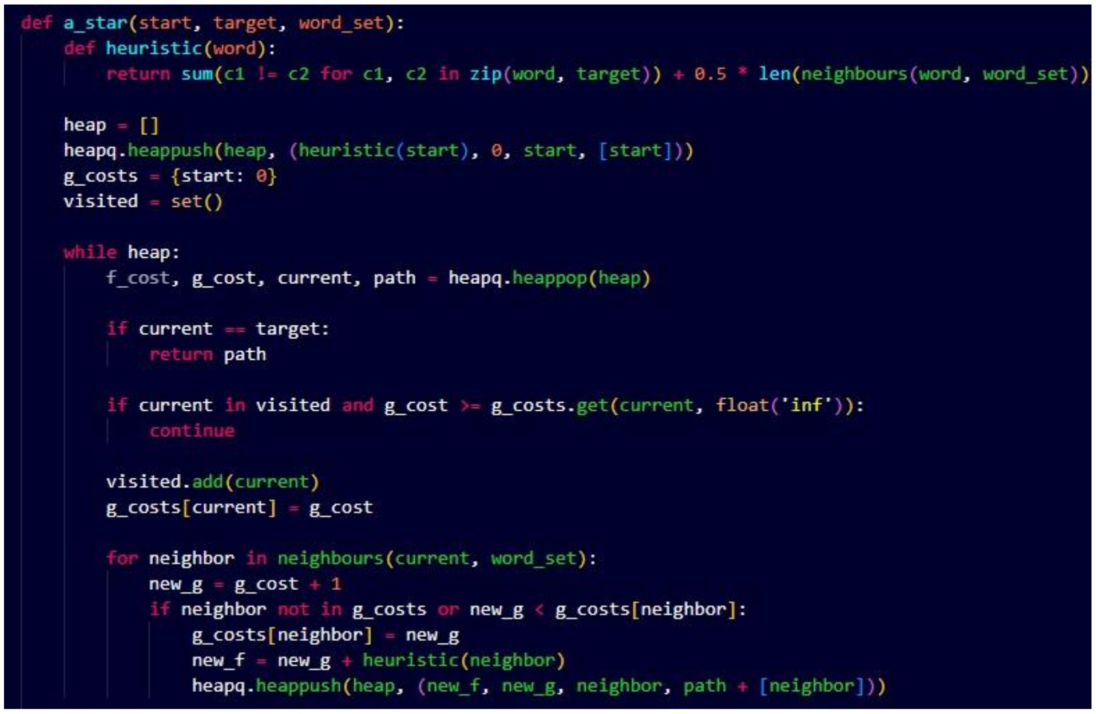

```python
def gbfs(start, target, word_set):
    def heuristic(word):
        return sum(c1 != c2 for c1, c2 in zip(word, target))

    heap = []
    heapq.heappush(heap, (heuristic(start), start, [start]))
    visited = set()

    while heap:
        h_cost, current, path = heapq.heappop(heap)
        if current == target:
            return path
        if current in visited:
            continue
        visited.add(current)
        for neighbor in neighbours(current, word_set):
            if neighbor not in visited:
                new_h = heuristic(neighbor)
                heapq.heappush(heap, (new_h, neighbor, path + [neighbor]))
    return None
```

UCS:



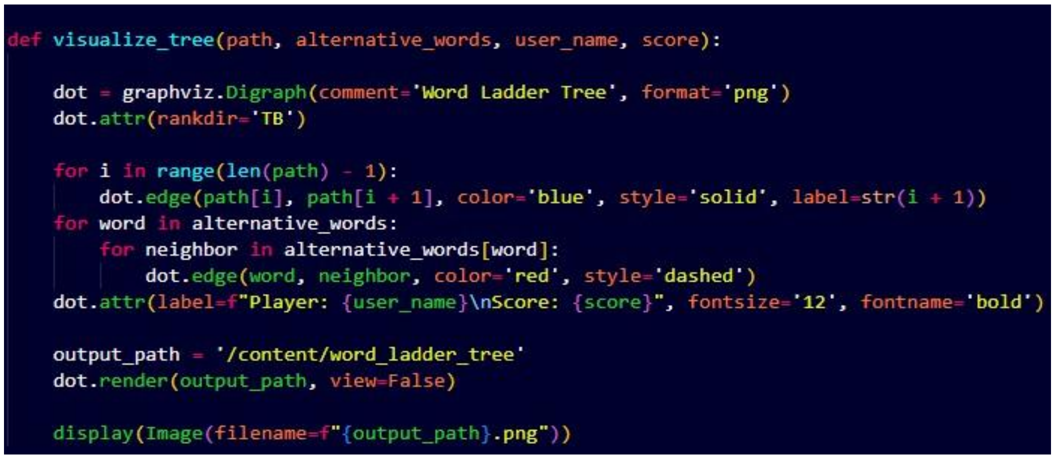
```python
def ucs(start, target, word_set):

    heap = []
    heapq.heappush(heap, (0, start, [start]))
    visited = set()

    while heap:
        cost, current, path = heapq.heappop(heap)
        if current == target:
            return path
        if current in visited:
            continue
        visited.add(current)
        for neighbor in neighbours(current, word_set):
            if neighbor not in visited:
                heapq.heappush(heap, (cost + 1, neighbor, path + [neighbor]))
    return None
```

A*:

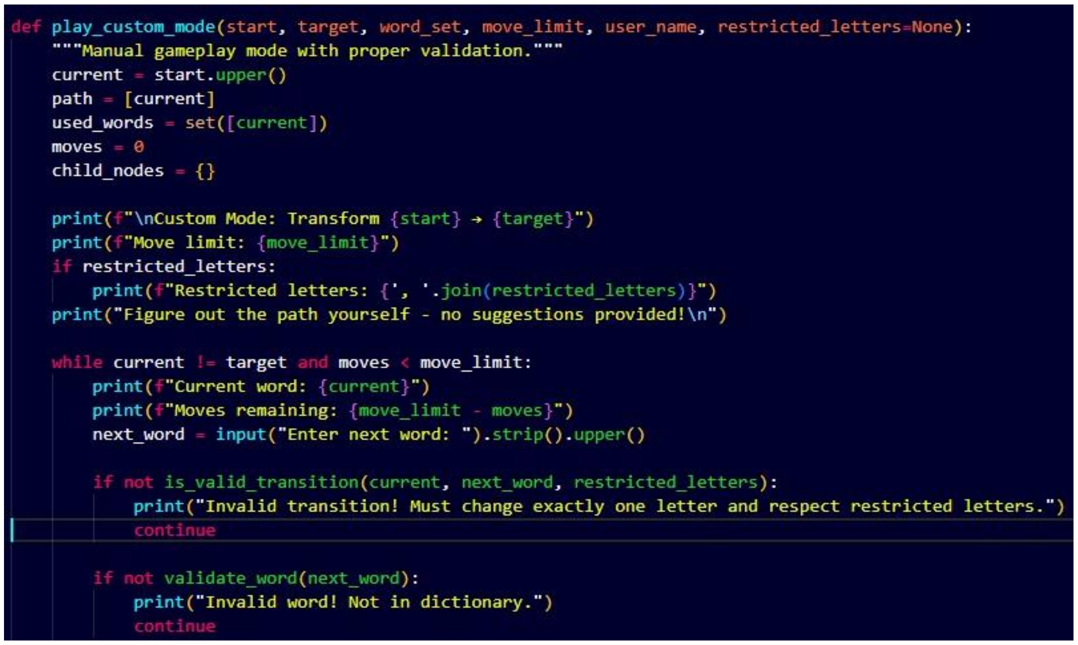


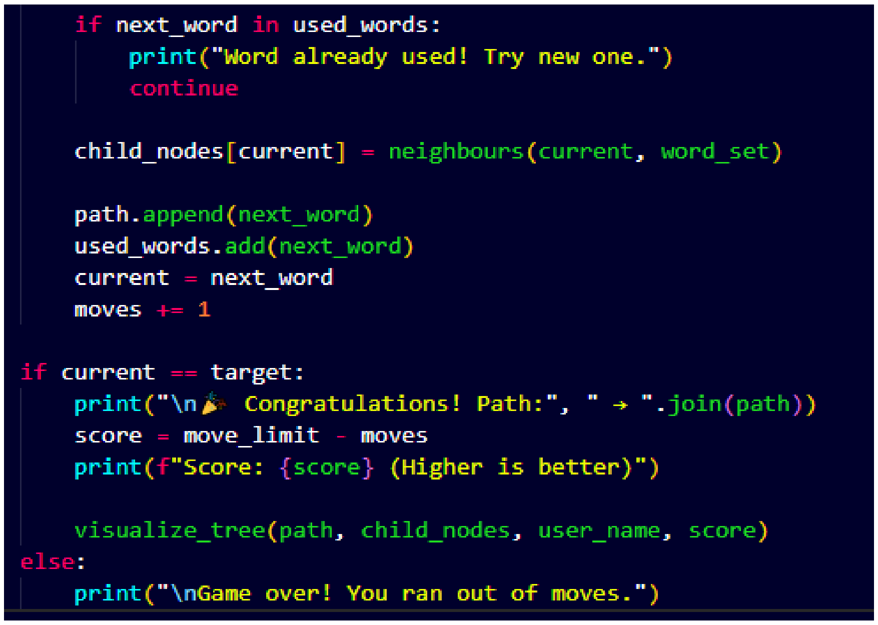
```python
def a_star(start, target, word_set):
    def heuristic(word):
        return sum(c1 != c2 for c1, c2 in zip(word, target)) + 0.5 * len(neighbours(word, word_set))

    heap = []
    heapq.heappush(heap, (heuristic(start), 0, start, [start]))
    g_costs = {start: 0}
    visited = set()

    while heap:
        f_cost, g_cost, current, path = heapq.heappop(heap)

        if current == target:
            return path

        if current in visited and g_cost >= g_costs.get(current, float('inf')):
            continue

        visited.add(current)
        g_costs[current] = g_cost

        for neighbor in neighbours(current, word_set):
            new_g = g_cost + 1
            if neighbor not in g_costs or new_g < g_costs[neighbor]:
                g_costs[neighbor] = new_g
                new_f = new_g + heuristic(neighbor)
                heapq.heappush(heap, (new_f, new_g, neighbor, path + [neighbor]))
```

Graph Generation:


```python
def visualize_tree(path, alternative_words, user_name, score):

    dot = graphviz.Digraph(comment='Word Ladder Tree', format='png')
    dot.attr(rankdir='TB')

    for i in range(len(path) - 1):
        dot.edge(path[i], path[i + 1], color='blue', style='solid', label=str(i + 1))
    for word in alternative_words:
        for neighbor in alternative_words[word]:
            dot.edge(word, neighbor, color='red', style='dashed')
    dot.attr(label=f"Player: {user_name}\nScore: {score}", fontsize='12', fontname='bold')

    output_path = '/content/word_ladder_tree'
    dot.render(output_path, view=False)

    display(Image(filename=f"{output_path}.png"))
```

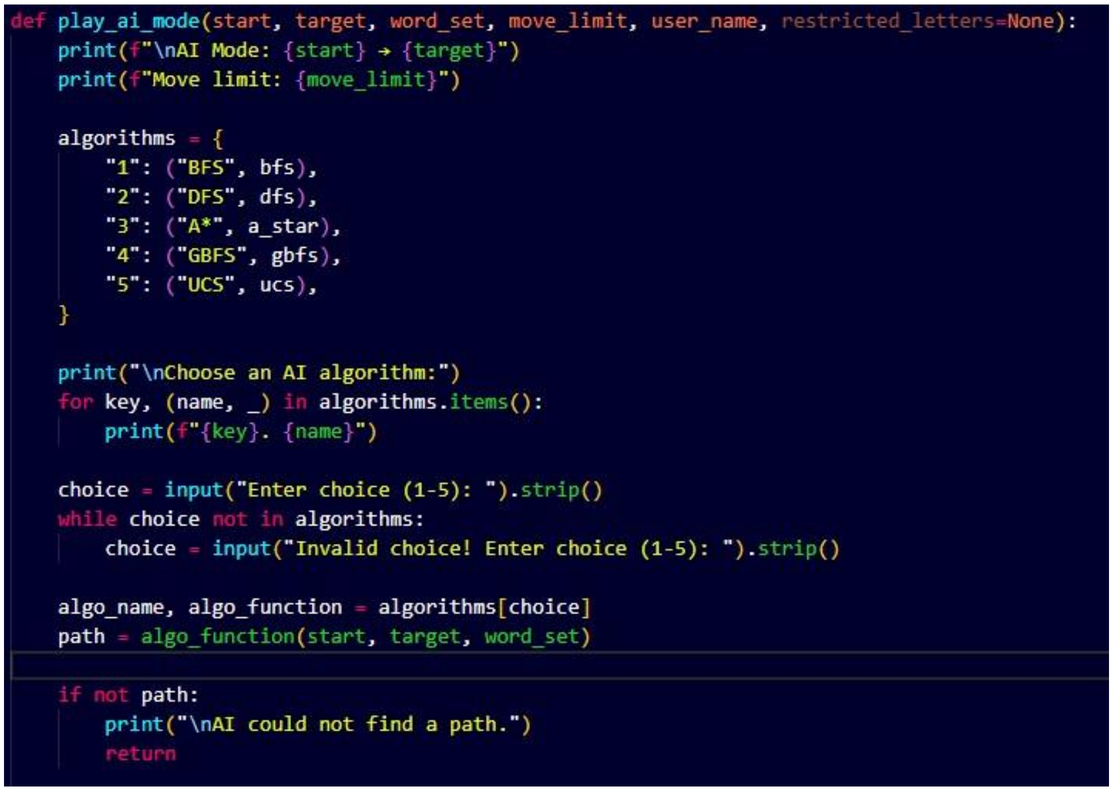
Custom Mode:


```python
def play_custom_mode(start, target, word_set, move_limit, user_name, restricted_letters=None):
    """Manual gameplay mode with proper validation."""
    current = start.upper()
    path = [current]
    used_words = set([current])
    moves = 0
    child_nodes = {}

    print(f"\nCustom Mode: Transform {start} → {target}")
    print(f"Move limit: {move_limit}")
    if restricted_letters:
        print(f"Restricted letters: {', '.join(restricted_letters)}")
    print("Figure out the path yourself - no suggestions provided!\n")

    while current != target and moves < move_limit:
        print(f"Current word: {current}")
        print(f"Moves remaining: {move_limit - moves}")
        next_word = input("Enter next word: ").strip().upper()

        if not is_valid_transition(current, next_word, restricted_letters):
            print("Invalid transition! Must change exactly one letter and respect restricted letters.")
            continue

        if not validate_word(next_word):
            print("Invalid word! Not in dictionary.")
            continue
```

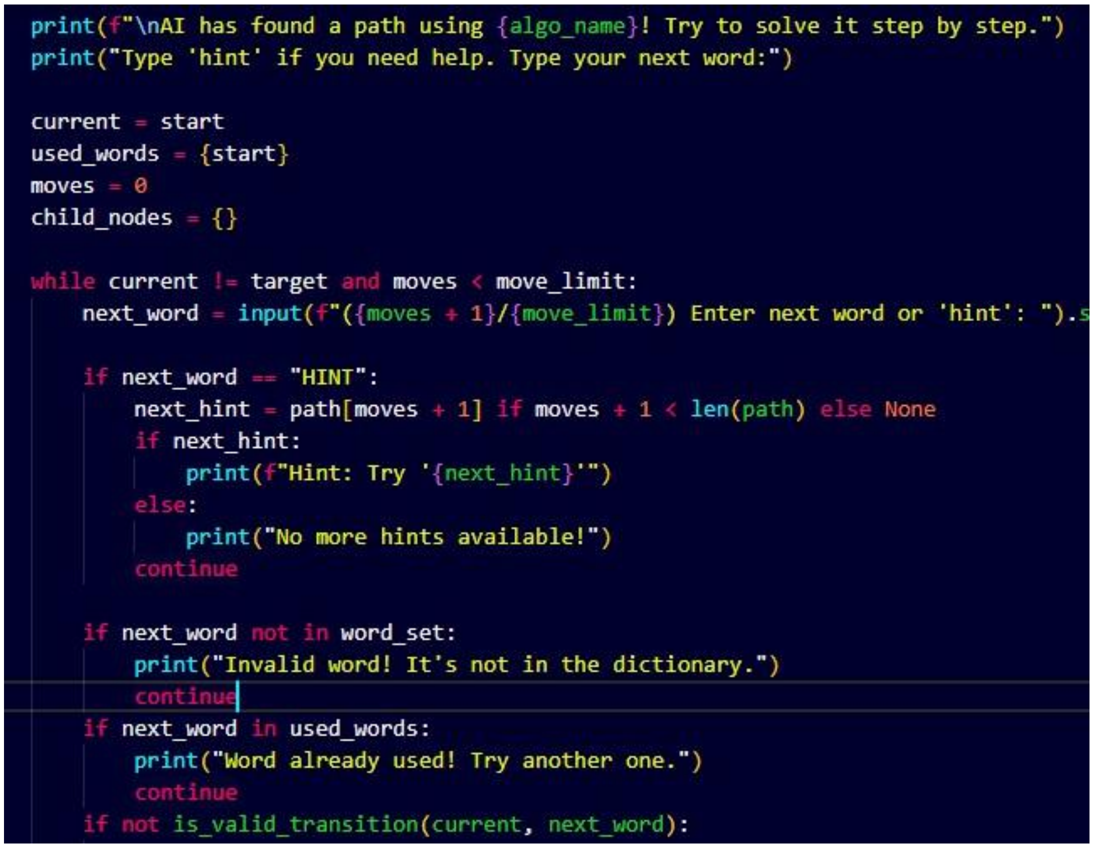


```
    if next_word in used_words:
        print("Word already used! Try new one.")
        continue

    child_nodes[current] = neighbours(current, word_set)

    path.append(next_word)
    used_words.add(next_word)
    current = next_word
    moves += 1

if current == target:
    print("\n🎉 Congratulations! Path:", " → ".join(path))
    score = move_limit - moves
    print(f"Score: {score} (Higher is better)")

    visualize_tree(path, child_nodes, user_name, score)
else:
    print("\nGame over! You ran out of moves.")
```

AI Mode:

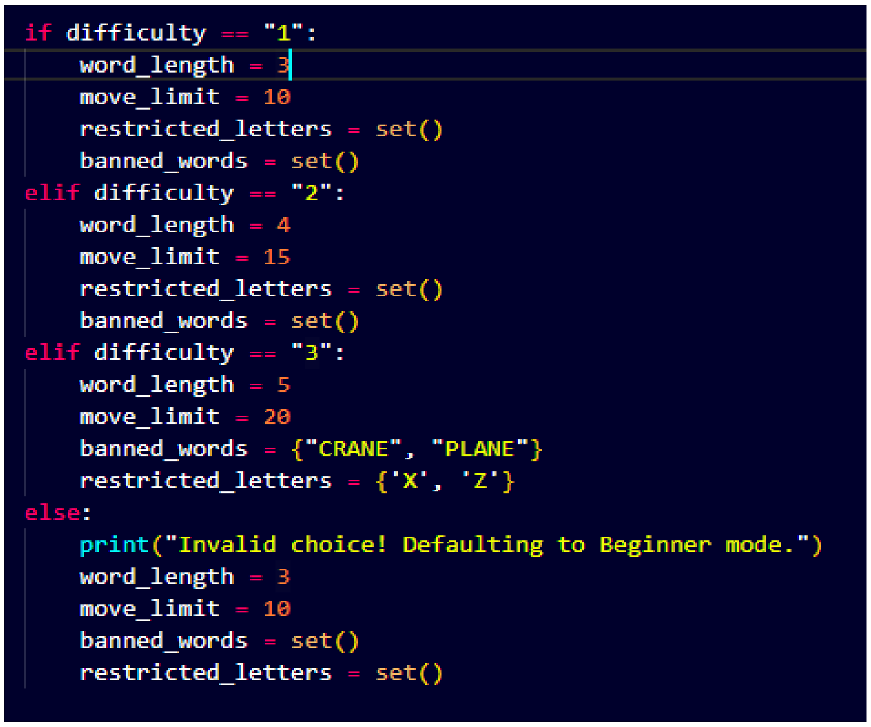


```
def play_ai_mode(start, target, word_set, move_limit, user_name, restricted_letters=None):
    print(f"\nAI Mode: {start} → {target}")
    print(f"Move limit: {move_limit}")

    algorithms = {
        "1": ("BFS", bfs),
        "2": ("DFS", dfs),
        "3": ("A*", a_star),
        "4": ("GBFS", gbfs),
        "5": ("UCS", ucs),
    }

    print("\nChoose an AI algorithm:")
    for key, (name, _) in algorithms.items():
        print(f"{key}. {name}")

    choice = input("Enter choice (1-5): ").strip()
    while choice not in algorithms:
        choice = input("Invalid choice! Enter choice (1-5): ").strip()

    algo_name, algo_function = algorithms[choice]
    path = algo_function(start, target, word_set)

    if not path:
        print("\nAI could not find a path.")
        return
```



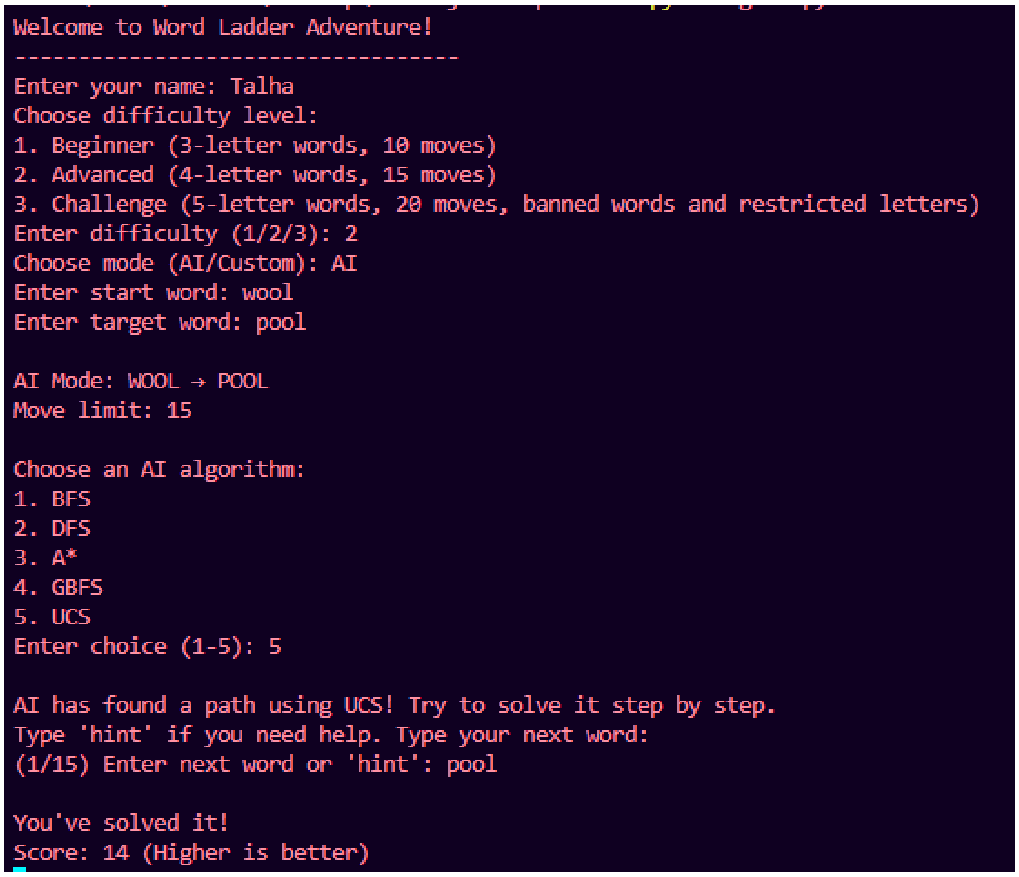
```
print(f"\nAI has found a path using {algo_name}! Try to solve it step by step.")
print("Type 'hint' if you need help. Type your next word:")

current = start
used_words = {start}
moves = 0
child_nodes = {}

while current != target and moves < move_limit:
    next_word = input(f"({moves + 1}/{move_limit}) Enter next word or 'hint': ").s

    if next_word == "HINT":
        next_hint = path[moves + 1] if moves + 1 < len(path) else None
        if next_hint:
            print(f"Hint: Try '{next_hint}'")
        else:
            print("No more hints available!")
        continue

    if next_word not in word_set:
        print("Invalid word! It's not in the dictionary.")
        continue
    if next_word in used_words:
        print("Word already used! Try another one.")
        continue
    if not is_valid_transition(current, next_word):
```

Main( ):keep in mind about banned words and letters:

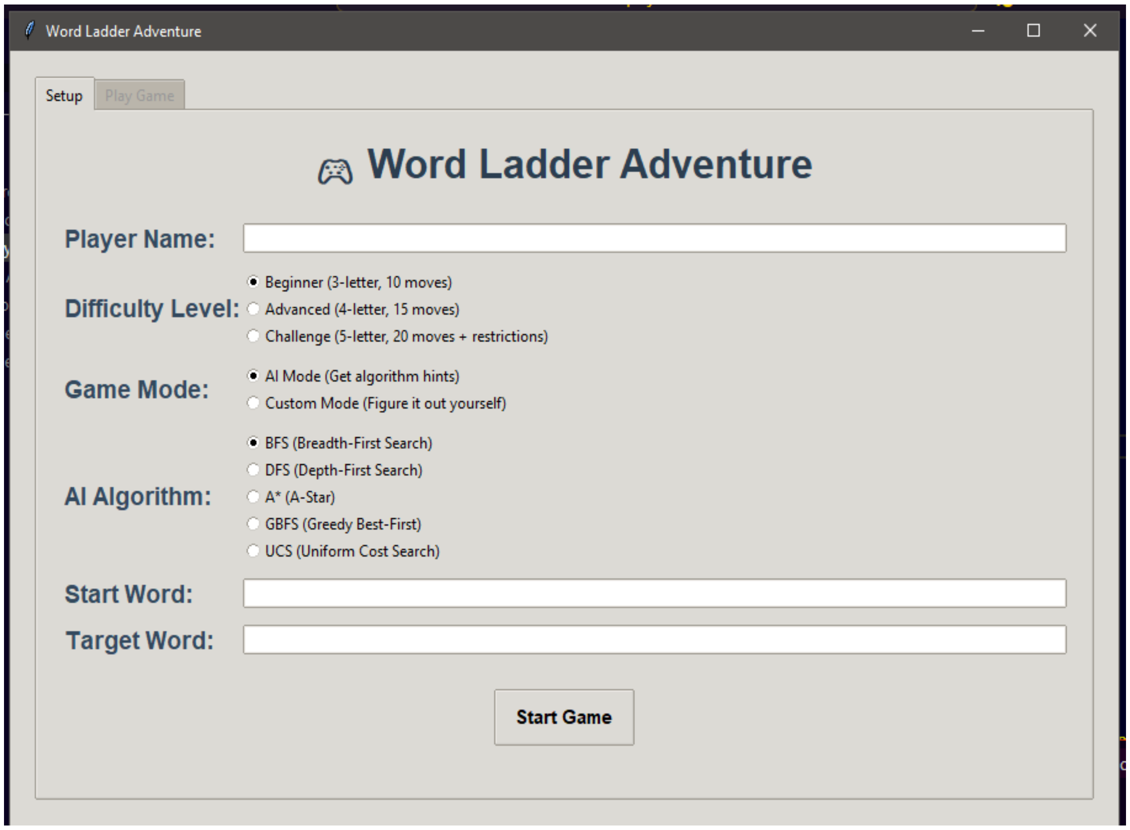


```
if difficulty == "1":
    word_length = 3
    move_limit = 10
    restricted_letters = set()
    banned_words = set()
elif difficulty == "2":
    word_length = 4
    move_limit = 15
    restricted_letters = set()
    banned_words = set()
elif difficulty == "3":
    word_length = 5
    move_limit = 20
    banned_words = {"CRANE", "PLANE"}
    restricted_letters = {'X', 'Z'}
else:
    print("Invalid choice! Defaulting to Beginner mode.")
    word_length = 3
    move_limit = 10
    banned_words = set()
    restricted_letters = set()
```

GUI:



```
Welcome to Word Ladder Adventure!
-----------------------------------
Enter your name: Talha
Choose difficulty level:
1. Beginner (3-letter words, 10 moves)
2. Advanced (4-letter words, 15 moves)
3. Challenge (5-letter words, 20 moves, banned words and restricted letters)
Enter difficulty (1/2/3): 2
Choose mode (AI/Custom): AI
Enter start word: wool
Enter target word: pool

AI Mode: WOOL → POOL
Move limit: 15

Choose an AI algorithm:
1. BFS
2. DFS
3. A*
4. GBFS
5. UCS
Enter choice (1-5): 5

AI has found a path using UCS! Try to solve it step by step.
Type 'hint' if you need help. Type your next word:
(1/15) Enter next word or 'hint': pool

You've solved it!
Score: 14 (Higher is better)
```


Word Ladder Adventure

Setup | Play Game

Word Ladder Adventure

Player Name:

Difficulty Level:
[✓ Beginner (3-letter, 10 moves)]
Advanced (4-letter, 15 moves)
Challenge (5-letter, 20 moves + restrictions)

Game Mode:
[✓ AI Mode (Get algorithm hints)]
Custom Mode (Figure it out yourself)

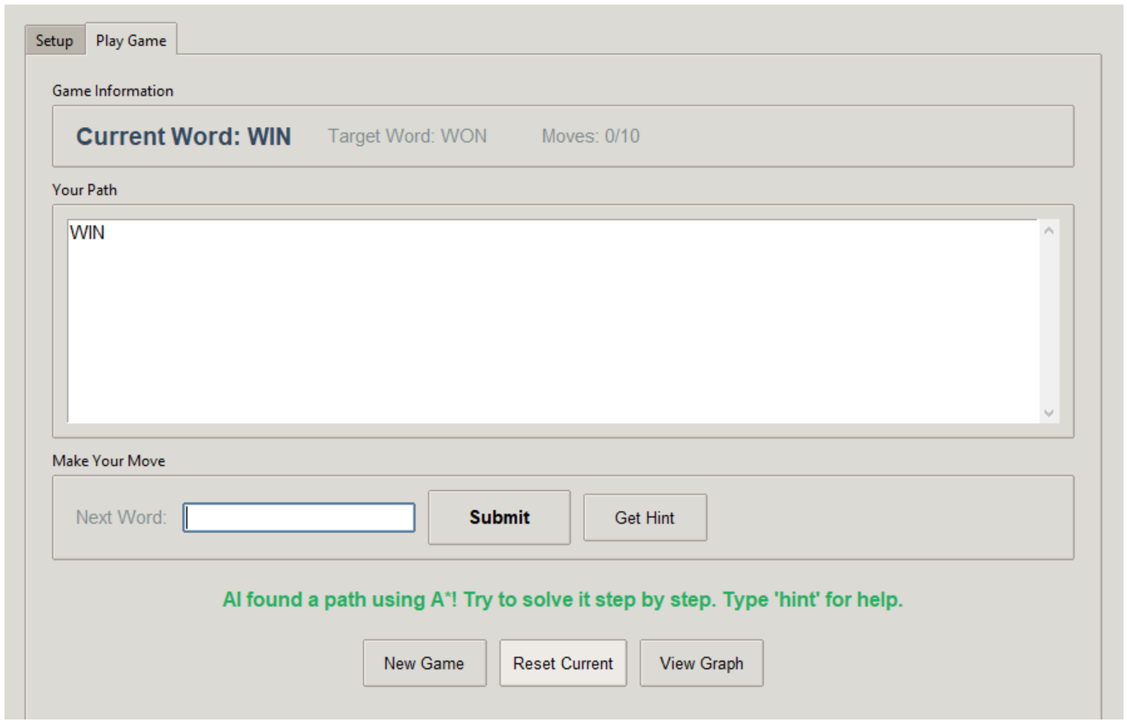
AI Algorithm:
[✓ BFS (Breadth-First Search)]
DFS (Depth-First Search)
A* (A-Star)
GBFS (Greedy Best-First)
UCS (Uniform Cost Search)

Start Word:

Target Word:

Start Game

Second Page where you write next word by changing one letter only!


Setup | Play Game

Game Information

Current Word: WIN   Target Word: WON   Moves: 0/10

Your Path

```
WIN
```

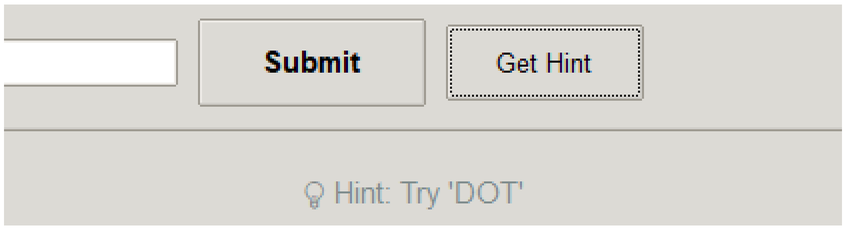
Make Your Move

Next Word: Submit   Get Hint

AI found a path using A*! Try to solve it step by step. Type 'hint' for help.

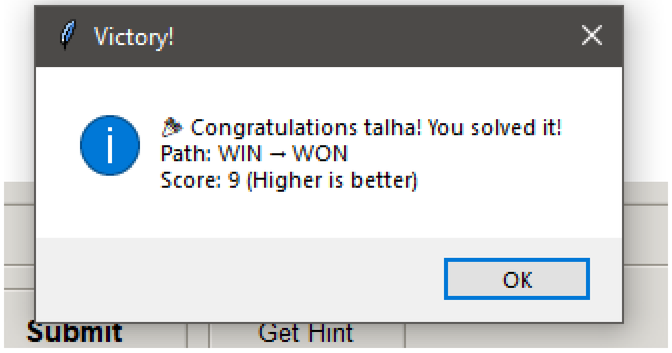
New Game   Reset Current   View Graph

If you click Hint Button: it will give you hint of next word according to rules using ai



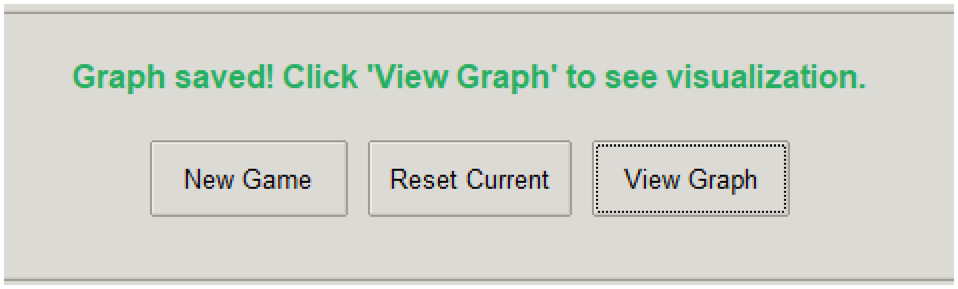
Hint: Try 'DOT'

Moves left will your winning score:



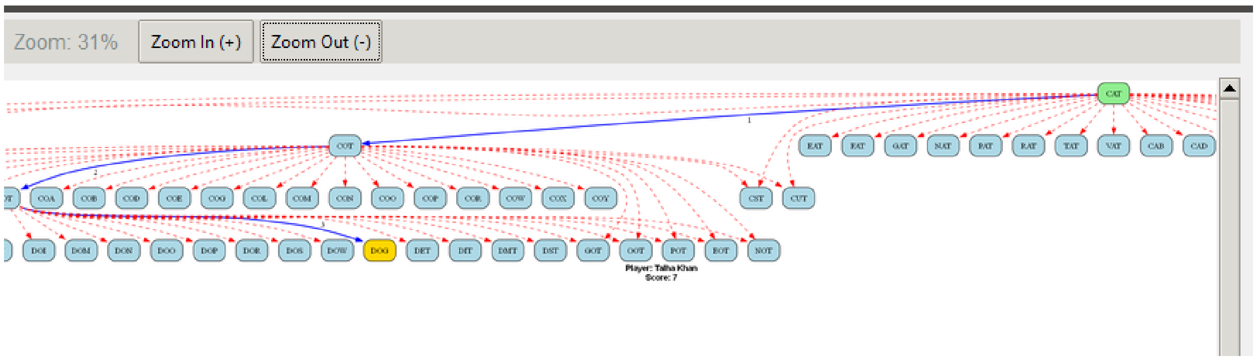
Congratulations talha! You solved it!
Path: WIN → WON
Score: 9 (Higher is better)

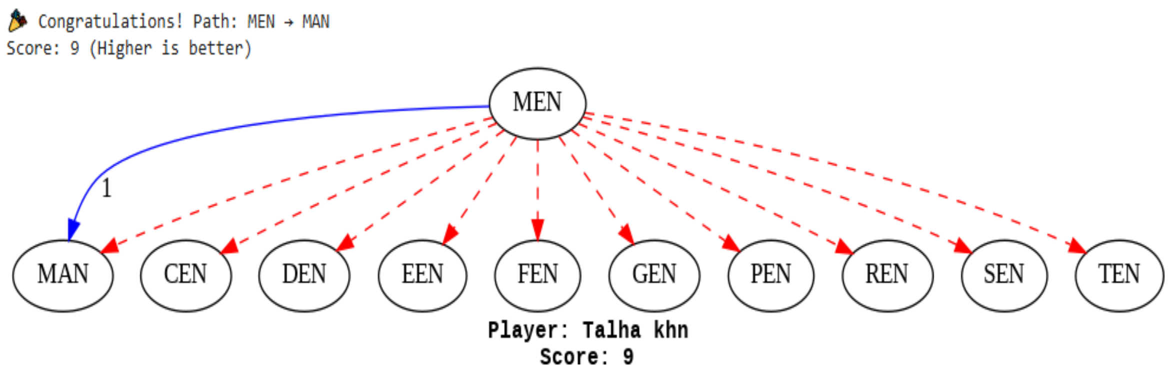
View Graph Button will show you Graph of applied Algothrim:



Graph saved! Click 'View Graph' to see visualization.

Graph Generation:




```
Congratulations! Path: MEN → MAN
Score: 9 (Higher is better)
```

Future Work

- Adding multiplayer mode
- Adding difficulty-based scoring system
- Improving Heuristic Functions
- Supporting different languages
- Mobile and web-based deployment
- Performance optimization for large dictionaries

Conclusion

The Word Ladder Adventure Game successfully demonstrates the application of Artificial Intelligence search algorithms in solving real-world problems.
The project enhances understanding of algorithm behavior, efficiency, and decision-making strategies.
With interactive gameplay, visualization, and multiple AI approaches, the system serves as both an educational and practical AI project.

Thank you!