CHANDIGARH UNIVERSITY

PROJECT CROSSWORD PUZZLE GAME

Subject: 24CAP-152

Data Structure

SUBMITED BY
GROUP-9

SUBMITED TO
DR. SHILPI GARG

GROUP DETAILS

SARTHAK KHANNA	24BCA20024
ADITYA MAHAJAN	24BCU10009
AKASH	24BCA20008

Crossword Puzzle Game Using Data Structures

Abstract:

This project presents a crossword puzzle game designed to enhance vocabulary, cognitive skills, and problem-solving abilities. It utilizes efficient data structures and algorithms to generate a dynamic crossword grid from a given word list.

Key data structures include Tries for word storage, a 2D Grid for representation, Backtracking for word placement, a Stack for undo/redo, a

Queue for hint management, and a Binary Search Tree (BST) for fast word searches. The generation process optimizes word interconnectivity while ensuring computational efficiency.

This project highlights the practical application of data structures in real-world scenarios, demonstrating improved algorithmic efficiency and problemsolving techniques for an engaging user experience.

Introduction:

Crossword puzzles enhance vocabulary, memory, and cognitive skills while providing entertainment. With the rise of digital crossword games, efficient generation methods are essential for maintaining interactivity and performance.

Data Structures and Algorithms (DSA) play a vital role in optimizing puzzle generation. A Trie enables fast word retrieval, a 2D grid represents the layout, and backtracking ensures correct word placement. Additionally, a Stack manages undo/redo, a Queue provides hints, and a Binary Search Tree (BST) speeds up word searches.

Beyond entertainment, crossword puzzles have educational applications in language learning and cognitive training. AI-driven systems can leverage these techniques for automated puzzle generation and adaptive difficulty. By integrating efficient data structures, this project creates a dynamic and engaging crossword puzzle experience.

Problem Statement

Designing an efficient crossword puzzle game involves computational challenges, requiring precise algorithms to generate a structured grid, ensure valid word placement, and enhance user interaction.

1 Optimized Crossword Grid

- **The grid must minimize empty spaces while ensuring meaningful word intersections.**
- **The algorithm should optimize space usage, prevent isolated letters, and adapt to different grid sizes and word lists.**

2 Valid Word Placement

- **Words must fit within boundaries, align correctly in intersections, and avoid conflicts.**
- **A validation mechanism ensures all words exist in the predefined dictionary.**
- **The grid should form a fully connected structure without isolated sections.**

3 Interactive Features

- **Hints: A queue-based system suggests valid words based on the grid.**
- **Undo/Redo: A stack-based approach allows users to reverse or redo moves.**
- **Real-time Validation: Incorrect placements trigger immediate feedback.**
- **Word Suggestions: The system suggests words based on partially filled letters.**
- **Adaptive Difficulty: Users can adjust puzzle complexity based on skill level.**

By leveraging data structures and algorithms, this project creates an engaging, dynamic crossword puzzle system that enhances usability and playability.

Data Structures Used

Developing an efficient crossword puzzle game requires the strategic use of multiple data structures, each serving a specific purpose. These structures optimize word storage, placement, validation, game mechanics, and user interactions. Below is a detailed breakdown of the core data structures used in the project, along with their roles, visual representations, and operational logic.

7.1 Trie Data Structure for Word Storage Why

Trie?

A Trie (Prefix Tree) is a specialized tree structure that enables fast word lookup, validation, and auto-completion. Unlike arrays or linked lists, which require O(n) search time, a Trie allows O(k) lookup, where k is the length of the word. This makes it ideal for storing and retrieving large word lists efficiently.

Role in the Crossword Game:

- **Efficient word validation: Ensures that only valid words are placed in the crossword grid.**
- **Auto-completion for hints: Helps generate word suggestions based on user input.**
- **Quick searches: Allows the system to instantly check whether a given word exists in the predefined dictionary.**

Trie Structure Diagram:

A visual representation of how words are stored in a Trie:

Example: The words stored here are cat, car, dog, star, stop. Each word follows a prefix-based hierarchy, allowing for efficient lookup.

7.2 2D Grid for Crossword Representation Why

2D Array?

A 2D array is the most efficient way to represent the crossword puzzle grid, as it allows direct indexing for accessing and modifying letters in O(1) time. Each cell represents a character or an empty space, and words are arranged horizontally and vertically.

Role in the Crossword Game:

- **Stores the crossword structure where words are placed.**
- **Quick access to any cell for validation and word placement.**

- **Enables real-time updates when a player enters a word.**

Crossword Grid Representation:

Example of a 5×5 crossword grid stored as a 2D array:

```
[['C', 'A', 'T', '#', '#'],
 ['#', '#', 'A', '#', '#'],
 ['D', 'O', 'G', '#', '#'],
 ['#', '#', 'R', '#', '#'],
 ['#', '#', 'T', 'O', 'P']]
```

Here, # represents empty spaces, and each letter is positioned strategically to form valid words.

7.3 Backtracking for Word Placement Why

Backtracking?

Backtracking is a recursive algorithm that systematically tries different placements for words while backtracking when conflicts arise. This approach ensures that all possible placements are explored until an optimal configuration is found.

Role in the Crossword Game:

- **Ensures correct word fitting by recursively testing word placements.**
  - **Eliminates invalid configurations and backtracks when needed.**
- **Finds the best possible arrangement to maximize word intersections.**

Backtracking Process Flowchart:

1 Place a word in the grid.
2 Check validity (Does it fit? Does it intersect correctly?) 3
If valid, move to the next word.
4 If invalid, backtrack and try a different placement.
5 Repeat until all words are placed correctly.

```
Start
  ↓
Place Word
  ↓
Is Placement Valid?
  ├── Yes → Place Next Word → Repeat
  └── No  → Backtrack → Try Another Placement → Repeat
```

7.4 Stack for Undo/Redo Feature

Why Stack?

A stack (LIFO - Last In, First Out) structure is ideal for managing Undo/Redo operations. The most recent action (last added word) is stored at the top of the stack, making it easy to reverse moves in O(1) time.

Role in the Crossword Game:

- **Undo feature: Removes the last added word when a player wants to correct an error.**
- **Redo feature: Restores the last removed word if the player reverts the undo.**
- **Efficient tracking of player moves.**

Undo/Redo Stack Operations:

```
Stack (Undo):
[ "CAT added" ]
[ "DOG added" ]
[ "TOP added" ]  <-- Last Action (can be undone)

Undo Operation: Remove "TOP"
Stack after Undo:
[ "CAT added" ]
[ "DOG added" ]
```

Using a second stack for redo:

```
Stack (Redo):
[ "TOP removed" ] <-- Last Undo action (can be redone)
```

If the player chooses Redo, "TOP" is restored.

7.5 Queue for Hint System Why

Queue?

A queue (FIFO - First In, First Out) is ideal for managing hints, as it processes requests in order. Players request a hint, and the oldest request is served first.

Role in the Crossword Game:

- **Ensures fair hint distribution by providing words in the order they are requested.**
- **Prevents overloading the system with repeated hint requests.**
- **Allows timed hint generation, ensuring an interactive and engaging experience.**

Queue Representation:

```
Hint Queue:
[ "Hint for Word 1" ] <-- First Hint (served first)
[ "Hint for Word 2" ] <-- Second Hint
[ "Hint for Word 3" ] <-- Third Hint
```

When a hint is provided, it is dequeued, and the next request moves up.

7.6 Binary Search Tree (BST) for Word Lookup Why

BST?

A Binary Search Tree (BST) provides faster word searches (O(log n)) compared to linear searches in arrays. It is used instead of a Hash Table because BSTs allow ordered traversal, making it easier to suggest related words.

Role in the Crossword Game:

- **Fast word validation: Allows quick lookup of words.**
- **Efficient word suggestions: Finds related words based on partial input.**
- **Reduces search time compared to linear or unsorted data structures.**

BST Visualization:

Here, a word search follows the BST rule:

- **Searching for "E": Compare with "M" → Go left to "C" → Go right to "E" (Found!).**
- **Searching for "Z": Compare with "M" → Go right to "S" → Go right to "T" → No more nodes (Not Found).**

Algorithm Explanation:

Step-by-Step Breakdown of Crossword Generation

The crossword puzzle generation follows a structured algorithm to efficiently place words while ensuring correctness. Below is a detailed breakdown of each step:

- **Step 1: Input Word List**

  The system takes a predefined list of words from the user or a dictionary.

  Words are filtered based on length and relevance to ensure compatibility with the grid size.

  Each word is stored in a Trie for quick validation.

- **Step 2: Generate Crossword Grid Using Backtracking**

  A 2D Grid (Array) is initialized to represent the crossword board.

  The backtracking algorithm attempts to place each word in the grid.

  It checks for horizontal or vertical placement.

  Ensures no overlapping letters create invalid words.

  If a word cannot be placed, it backtracks and tries a different placement.

- **Step 3: Store Words in Trie for Validation**

  Once a word is successfully placed, it is stored in a Trie.

The Trie enables quick validation for subsequent word placements.

It prevents duplicate or invalid words from being added to the crossword.

- **Step 4: Use Stack for Undo/Redo**

  A Stack is used to record each move (word placement, removal).

  The LIFO (Last-In, First-Out) mechanism allows users to undo recent actions.

  If a player removes a word, the previous state is restored from the stack.

- **Step 5: Use Queue for Hint Management**

  A Queue is implemented to manage hint requests in FIFO (First-In, FirstOut) order.

  The system provides hints based on pre-stored words.

  Players receive the oldest hint request first to maintain fairness.

- **Step 6: Display Crossword to the User**

  The final crossword grid is displayed to the player.

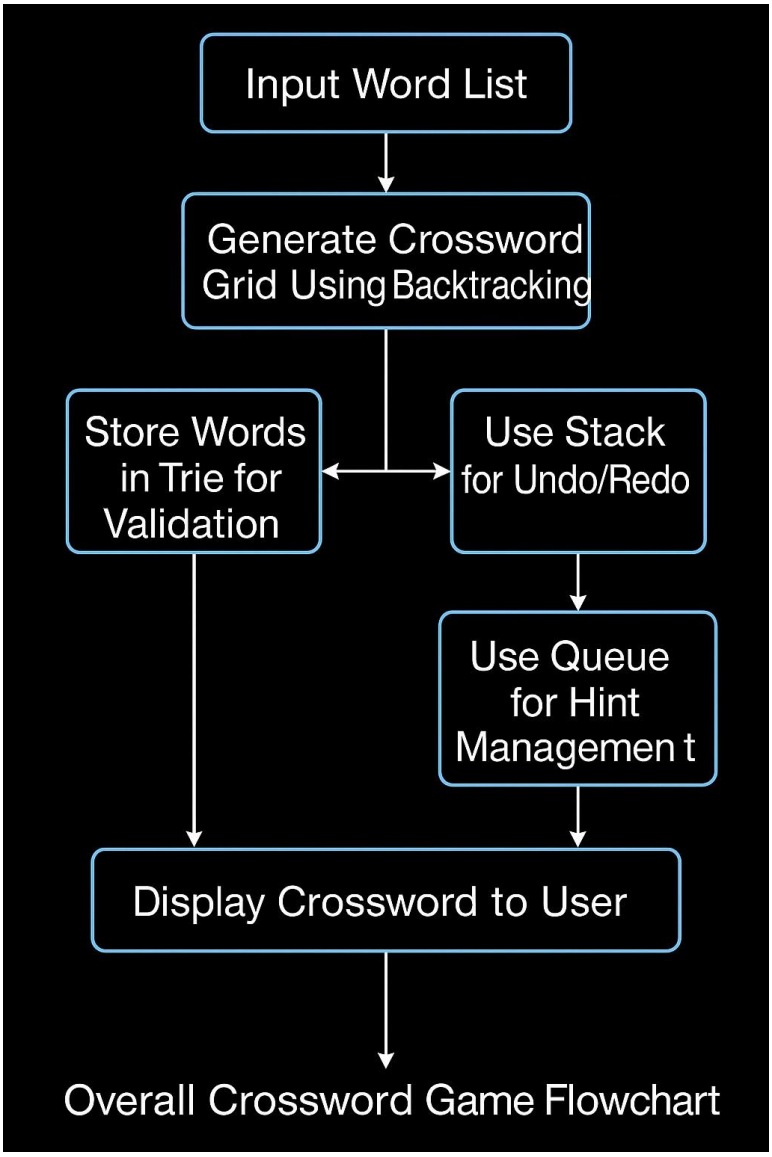
  The game checks for correctness by validating words in the Trie.

  Users can interact using undo, redo, and hint features.

  A scoring system can be implemented based on correct words and time tak

---

- **Diagram: Overall Crossword Game Flowchart**



Implementation :

1. Word Placement (Backtracking Algorithm)

Uses recursion to place words while ensuring valid positioning.

Logic:

- **Attempt to place the word horizontally or vertically.**
- **Check for valid overlaps with existing letters.**
- **If placement is invalid, backtrack and try another position.**
- **Repeat until all words are placed successfully.**

2. Fast Word Lookup (Trie Data Structure)

A Trie stores words efficiently for validation and hint suggestions.

Logic:

- **Each letter is stored in a hierarchical structure.**
- **Enables quick validation and auto-completion.**

3. Undo/Redo Feature (Stack Implementation)

Tracks previous actions, allowing users to reverse or redo moves.

Logic:

- **Every word placement is pushed onto a stack.**
- **Undo: The last move is popped and reverted.**
- **Redo: The last undone move is restored.**

4. Hint System (Queue Implementation) Provides

hints in First-In-First-Out (FIFO) order.

Logic:

- **Hints are queued when requested.**
- **The oldest hint is retrieved first.**
- **Users can request multiple hints in sequence.**

Execution Screenshots:

```
1. Display Grid
2. Place Word
3. Exit
Enter your choice: 2
Enter word (in uppercase): APPLE
Enter row (0-4): 1
Enter col (0-4): 0
Enter direction (H for horizontal, V for vertical): H

Word placed successfully!

Enter your choice: 1

Crossword Grid:
 . . . . .
 A P P L E
 . . . . .
 . . . . .
 . . . . .
```

```
Crossword Grid:
 .  .  .  .  .
 A  P  P  L  E
 .  .  .  .  .
 .  .  .  .  .
 .  .  .  .  .

1. Display Grid
2. Place Word
3. Undo Last Move
4. Show Hint
5. Exit
Enter your choice: 3

Last move undone!

1. Display Grid
2. Place Word
3. Undo Last Move
4. Show Hint
5. Exit
Enter your choice: 1

Crossword Grid:
 .  .  .  .  .
 .  .  .  .  .
 .  .  .  .  .
 .  .  .  .  .
 .  .  .  .  .
```

Conclusion

This project successfully demonstrates the role of data structures in building an efficient and interactive crossword puzzle game. By utilizing Tries, 2D Grids, Backtracking, Stacks, Queues, and Binary Search Trees, the crossword generation process is optimized for speed, accuracy, and user experience.

Through this project, we have gained a deeper understanding of how different data structures can work together to solve real-world problems efficiently. The crossword puzzle game showcases how computational efficiency can be improved through algorithmic design, making it a valuable learning experience for data structure enthusiasts and game developers alike.

Github Links:

https://github.com/Sarthakk-D/DSA-PROJECT.git

https://github.com/Aditya005-mahajan/DSA-Project.git

https://github.com/AKASH00000007/DSA-PROJECT.git

References:

References for Crossword Puzzle Generation (Algorithms & Code)

- **Backtracking Algorithm for Crossword Generation**
  - **GeeksforGeeks: Backtracking Algorithm for Crossword (Can be adapted for crossword placement).**
- **Trie Data Structure for Word Storage ○ MIT OpenCourseWare: Trie Data Structure ○ GeeksforGeeks: Trie Implementation in Python**
- **Stack & Queue Implementation for Undo/Redo & Hints ○ GeeksforGeeks: Stack Operations ○ GeeksforGeeks: Queue Operations**
- **Binary Search Tree (BST) for Fast Word Lookup**

  GeeksforGeeks: BST Search Implementation