Term Project Proposal Report

Group Information

- Ibrahim Atatri – atatri@umich.edu
- Evan Wilkin – ewilkin@umich.edu

Both team members will contribute equally to the project. Ibrahim Atatri will take the lead on presentations and implementation, while Evan Wilkin will take the lead on writing and documentation. All design, implementation, and testing decisions will be made collaboratively.

Title

An Algorithmic Wordle Solver Using Custom Data Structures and Search Techniques

Problem Statement

Wordle is a popular word guessing game where a player must determine a hidden five letter word within six attempts using feedback provided after each guess. While the rules of the game are simple, developing an automated solver that consistently performs well requires careful handling of candidate word filtering, efficient data management, and algorithmic decision making.

The goal of this project is to design and implement a Wordle solving program that systematically narrows the search space of possible solutions and selects guesses intelligently. This problem is important because it provides a practical and intuitive way to apply core concepts from data structures and algorithm design, including search strategies, filtering mechanisms, and performance evaluation.

Our work lies in implementing all required data structures from scratch (without using the C++ standard library equivalents) and combining multiple algorithmic techniques to guide the guessing strategy. Unlike many existing Wordle solvers that rely on built in data structures, our solver will explicitly demonstrate how custom hashing and tree based structures interact with greedy and backtracking algorithms to improve guess selection and solution efficiency.

Related Work

Several Wordle solvers and analyses have been developed since the game's release, most of which focus on optimizing guess selection through heuristics, entropy based scoring, or brute force search. Some approaches emphasize information gain to minimize the number of remaining candidates, while others simulate all possible game states to determine optimal guesses.

Previous work has explored greedy strategies that prioritize guesses eliminating the largest number of candidate words, as well as recursive approaches that explore multiple guess paths. Additionally, some implementations use tree based data structures or hash based filtering to efficiently manage large word lists.

Our project is inspired by these approaches but differs in that it focuses on a clear educational implementation using custom built data structures and explicitly selected algorithm design techniques from this course. Rather than maximizing complexity, we aim to clearly demonstrate correctness, efficiency, and trade offs between algorithmic strategies in a way that aligns with CSC 375 learning objectives.

References:

1. R. Reddy, "Building a Smart Wordle Solver with Java," *Medium*, May 17, 2022. [Online]. Available: https://medium.com/data-science/building-a-smart-wordle-solver-with-java-cb734d4a9635. [Accessed: Jan. 30, 2026].
2. A. Shrestha, "Solving Wordle Using Information Theory," *Towards Data Science*, 2022.
3. D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, 1998.
4. R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
5. S. S. Skiena, The Algorithm Design Manual, 2nd ed. London, UK: Springer, 2008.

PDF versions of all references will be included as separate attachments.

Approach

The project will be implemented in C++ and will include a menu driven command line interface, which is well suited for algorithm focused projects and aligns with the expectations of the course. This interface will allow users to input feedback from Wordle guesses and observe how the solver updates its candidate pool and selects future guesses.

Data Structures

Two custom data structures will be implemented from scratch:

1. Hash Table - Used to store and quickly filter candidate words based on feedback constraints.

2. Tree Structure (Decision Tree / Tree-like structure) - Used to represent possible guessing paths and support recursive exploration of guess sequences.

Algorithm Design Techniques

Two algorithm design techniques will be used:

1. Greedy Algorithm – Selects guesses that eliminate the largest number of remaining candidate words.
2. Backtracking – Explores alternative guess paths when greedy choices do not immediately lead to a solution.

Project Timeline (Gantt Summary Table) - Full Gantt readable excel chart chart can be found in this link since I couldn't upload it to canvas or render it in a doc. https://docs.google.com/spreadsheets/d/1uGSD-sJ9pDaIO79viygTgeXoEuCPyjsG/edit?usp=sharing&ouid=117902256434401619392&rtpof=true&sd=true

Task ID	Task Description	Start Date	End Date	Owner	Deliverable
T1	Finalize project requirements and design decisions	Feb 5	Feb 7	Ibrahim & Evan	Design notes
T2	Implement custom hash table (insert, search, delete)	Feb 8	Feb 14	Evan	HashTable.cpp
T3	Implement word filtering logic using feedback rules	Feb 10	Feb 18	Ibrahim	Filter module
T4	Build command-line menu interface	Feb 15	Feb 20	Ibrahim	CLI prototype

T5	Unit testing for hash table (30 test cases)	Feb 18	Feb 24	Evan	Test table
T6	Implement tree structure for guess paths	Feb 26	Mar 5	Evan	Tree.cpp
T7	Implement greedy guess selection algorithm	Mar 1	Mar 10	Ibrahim	GreedySolver.cpp
T8	Integration testing and debugging	Mar 8	Mar 20	Both	Updated test results
T9	Implement backtracking algorithm	Mar 15	Mar 25	Evan	BacktrackingSolver.cpp
T10	Final testing (100+ test cases)	Mar 26	Apr 10	Both	Final test report
T11	Final report writing	Apr 10	Apr 20	Evan	Final paper
T12	Presentation slides and demo prep	Apr 15	Apr 22	Ibrahim	Slides + demo

Task	Deliverable	Start Date	End Date	Assigned	05/02/2026	06/02/2026	07/02/2026	08/02/2026	09/02/2026	10/02/2026	11/02/2026	12/02/2026	13/02/2026	14/02/2026	15/02/2026	16/02/2026	17/02/2026	18/02/2026	19/02/2026	20/02/2026	21/02/2026	22/02/2026	23/02/2026
Finalize project requirements and design decisions	Design notes	05-Feb	07-Feb	Ibrahim & Evan																			
Implement custom hash table (insert, search, delete)	HashTable.cpp	08-Feb	14-Feb	Evan																			
Implement word filtering logic using feedback rules	Filter module	10-Feb	18-Feb	Ibrahim																			
Build command-line menu interface	CLI prototype	15-Feb	20-Feb	Ibrahim																			
Unit testing for hash table (30 test cases)	Test table	18-Feb	24-Feb	Evan																			
Implement tree structure for guess paths	Tree.cpp	26-Feb	05-Mar	Evan																			
Implement greedy guess selection algorithm	GreedySolver.cpp	01-Mar	10-Mar	Ibrahim																			
Integration testing and debugging	Updated test results	08-Mar	20-Mar	Ibrahim & Evan																			
Implement backtracking algorithm	BacktrackingSolver.cpp	15-Mar	25-Mar	Evan																			
Final testing (100+ test cases)	Final test report	26-Mar	10-Apr	Ibrahim & Evan																			
Final report writing	Final paper	10-Apr	20-Apr	Evan																			
Presentation slides and demo prep	Slides + demo	15-Apr	22-Apr	Ibrahim																			

Task assignments will be split evenly between team members, with shared responsibility for coding, debugging, and testing.

Evaluation

The success of the project will be evaluated by measuring the solver’s ability to correctly identify the target word within the standard six guesses for a wide range of test cases. Correctness of the custom data structures will be verified through extensive unit testing, while algorithm performance will be evaluated by tracking guess counts and candidate reduction rates.

At least 30 test cases will be reported for Progress Report I, 70 test cases for Progress Report II, and 100 test cases for the final submission. Each test case will document inputs, expected outputs, actual outputs, and pass/fail status.

Potential challenges include managing edge cases where feedback combinations severely limit candidate words and ensuring efficiency without relying on standard library structures. These challenges will be addressed through incremental testing, careful debugging, and algorithm refinement throughout the semester.