**UNIVERSITY OF SCIENCE**
**ADVANCED PROGRAM IN COMPUTER SIENCE**

# PROJECT1 REPORT

## Course: Artificial Intelligence
## Project: WORDLE SOLVER

**Instructor:** Nguyen Thanh Tinh - Nguyen Ngoc Thao
**Group:** [Group 7]

**GROUP MEMBERS:**

1. Pham Bao Kha - 23125058
2. Nguyen Anh Bao - 23125051
3. Vo Tran Hien - 23125056
4. Nguyen Nhat Hao - 20125028

**December 3, 2025**

# Contents

# 1 Project Planning and Task Distribution

This section documents the responsibilities of each team member and the completion rate of their assigned tasks.

**Evaluation Method:**
The individual score is calculated as follows:

$$\text{Individual Score} = \text{Total Group Score} \times \text{Percentage of Completion}$$

*Example: If the group work receives a score of 9.0 and Student A has a completion rate of 90%, then A receives:* $9.0 \times 90\% = 8.1$.

Table 1: Task Distribution and Contribution

| **No.** | **Member Name** | **Responsibilities** | **Rate (%)** | **Signature** |
|---|---|---|---|---|
| 1 | Member 1 | - Implement Algorithm A<br>- Write Report Section 1 & 2<br>- Data collection | 100% | |
| 2 | Member 2 | - Implement Algorithm B<br>- Generate visualization charts<br>- Formatting the final report | 100% | |
| 3 | Member 3 | - Implement Algorithm C<br>- Write Report Section 3<br>- Code debugging | 100% | |
| 4 | Member 4 | - Design data structures<br>- Compile references & appendix<br>- Testing and verification | 100% | |

# 2 Algorithm Description (10 pts)

This section provides a detailed explanation of each search algorithm implemented in the project.

## 2.1 Algorithm 1: DFS

### 2.1.1 Overview

Although named 'DFSSolver', the implementation utilizes a **Greedy Constraint Satisfaction** strategy rather than a traditional backtracking recursive search. The core idea is to maintain a list of valid candidates and progressively prune this list based on the feedback received from the secret word.

The algorithm relies on a set of "Master Start Words" (e.g., SLATE, CRANE) to maximize information gain in the first turn. For subsequent steps, it greedily selects the first available word in the remaining candidate list.

### 2.1.2 Algorithm Mechanism

The solving process can be described mathematically as follows:

Let $D$ be the initial dictionary of all valid 5-letter words. Let $S_k$ be the set of candidate words at step $k$ (initially $S_0 = D$). Let $Target$ be the secret word to find.

1. **Initialization:** The solver selects a starting word $g_0$ from a predefined list of high-entropy words (e.g., "SLATE").
2. **Feedback Calculation:** For a guess $g$ and a target $T$, the feedback function $F(g, T)$ returns a pattern of 5 states (Green, Yellow, Grey):
$$F(g, T) = [state_0, state_1, ..., state_4]$$
3. **Filtering (Pruning) Step:** After making a guess $g_k$ and receiving actual feedback $f_{real} = F(g_k, Target)$, the algorithm filters the candidate set $S_k$ to generate $S_{k+1}$. A word $w$ is kept in the new set if and only if it would produce the exact same feedback if it were the target:
$$S_{k+1} = \{w \in S_k \mid F(g_k, w) = f_{real}\}$$
This step effectively removes all words that are inconsistent with the clues obtained.
4. **Next Guess Selection:** The algorithm greedily picks the first word from the new candidate set:
$$g_{k+1} = S_{k+1}[0]$$
Steps 2-4 are repeated until the feedback consists of all 'Green' states.

### 2.1.3 Complexity Analysis

To analyze the complexity, we define:

- $N$: The number of words in the dictionary ($\approx 12,000$).
- $L$: The length of the word (constant $L = 5$).
- $K$: The number of guesses made until the solution is found (usually $K \leq 6$).

**Time Complexity:** The complexity is dominated by the filtering process inside the loop.

1. The main loop runs $K$ times.
2. Inside each iteration, the `_filter_candidates` function iterates through the current candidate list (size at most $N$).
3. For each candidate, the `_calculate_feedback` function is called, which runs in $O(L)$ time.

Thus, the time complexity per step is $O(N \cdot L)$. The total time complexity is:

$$T(n) = O(K \cdot N \cdot L) \tag{1}$$

Since $L$ is constant and small, we can approximate this as $O(K \cdot N)$. In the worst case where the dictionary is not reduced significantly, the complexity approaches linear time with respect to the dictionary size.

**Space Complexity:** The algorithm stores the word lists in memory.

1. `self.all_words`: Stores $N$ words of length $L$, taking $O(N \cdot L)$.
2. `candidate_words`: A subset of the original list, max size $N$.
3. `feedback` arrays: Constant size $O(L)$.

Therefore, the space complexity is:

$$S(n) = O(N \cdot L) \tag{2}$$

This is efficient as it only requires memory proportional to the dictionary size.

## 2.2 Algorithm 1: BFS

### 2.2.1 Overview

The BFS Solver applies the Breadth-First Search strategy using a **First-In-First-Out (FIFO)** queue structure (implemented via `collections.deque`).

In the context of Wordle, each node in the search graph represents a valid guess. Unlike the standard BFS which expands all possible neighbors at the current depth, this implementation is optimized to manage the branching factor by filtering candidates and prioritizing the first valid word found in the filtered list.

### 2.2.2 Algorithm Mechanism

The algorithm operates through the following steps:

1. **Initialization:**
   - A queue $Q$ is initialized.
   - For the first turn, a word is randomly selected from the `MASTER_START_WORDS` (high entropy words) and enqueued.

2. **Iteration (Queue Processing):** While the queue is not empty and the attempt limit is not reached:

   (a) **Dequeue:** Extract the word $w_{curr}$ from the front of $Q$.

   (b) **Check Feedback:** Calculate the feedback between $w_{curr}$ and the secret word. If all characters match ('G'), the solution is found.

   (c) **Pruning (Filtering):** The algorithm filters the candidate list $S$. A word $c \in S$ is retained only if:

   $$\text{Feedback}(w_{curr}, c) == \text{Feedback}(w_{curr}, \text{Secret})$$

   This step eliminates branches that are inconsistent with the current evidence.

   (d) **Enqueue:** From the remaining filtered list, the algorithm selects the first available candidate ($S_{new}[0]$) and adds it to the rear of $Q$ for the next iteration.

### 2.2.3 Complexity Analysis

Let $N$ be the size of the dictionary and $K$ be the number of attempts (depth).

**Time Complexity:** The time complexity is primarily driven by the filtering process at each level of the search tree.

- The algorithm performs $K$ iterations (where $K \leq 6$ for standard Wordle).
- In each iteration, it traverses the candidate list of size roughly $O(N)$ to verify consistency against the feedback.
- The feedback calculation takes constant time $O(L)$ where $L = 5$.

Thus, the total time complexity is:

$$T(n) = O(K \cdot N) \tag{3}$$

In the worst-case scenario where the candidate list shrinks slowly, the algorithm performs a linear scan of the dictionary at each step.

**Space Complexity:**

- **Candidate Storage:** The algorithm maintains a copy of the dictionary, requiring $O(N \cdot L)$ space.
- **Queue Storage:** Since this optimized implementation enqueues only the most promising candidate (linear expansion) rather than all neighbors, the queue size remains small ($O(1)$ in this specific implementation, though standard BFS would be $O(N)$).

Therefore, the dominant space complexity is:

$$S(n) = O(N \cdot L) \tag{4}$$

## 2.3 Algorithm 3: Greedy Best-First Search with Entropy-based Heuristic

In our implementation, the Entropy Solver leverages Information Theory (specifically **Shannon Entropy**) to minimize the expected number of guesses required to solve the Wordle puzzle. Unlike BFS or DFS, which search for paths, this solver acts as a greedy agent that attempts to maximize the information gain at each step.

### 2.3.1 Theoretical Foundation

The core objective of Wordle is to reduce the uncertainty about the secret word. Let $\mathcal{C}$ be the set of currently valid candidate words. For any potential guess $w$, the game returns a feedback pattern $p$ (a sequence of 5 colors). There are $3^5 = 243$ possible patterns.

We define the probability of observing a specific pattern $p$ given a guess $w$ as:

$$P(p \mid w) \;=\; \frac{\left|\{\, c \in \mathcal{C} \;\mid\; \text{Feedback}(w, c) = p \,\}\right|}{|\mathcal{C}|}$$

The Expected Information (Entropy) $E[w]$ of a guess $w$ is calculated as:

$$E[w] = -\sum_{p \in \text{Patterns}} P(p \mid w)\, \log_2 P(p \mid w).$$

A higher entropy value indicates that the guess $w$ is likely to divide the candidate set $\mathcal{C}$ into smaller, more manageable subsets, regardless of what the actual secret word is. The solver selects the guess $w^*$ that maximizes this entropy:

$$w^* = \underset{w \in \mathcal{D}}{\operatorname{argmax}}\, E[w] \tag{5}$$

### 2.3.2 Detailed Implementation Mechanisms

While the core principle relies on Shannon Entropy, applying pure theory to a real-time game requires specific strategies to balance computational speed and game constraints. Our implementation integrates three critical mechanisms: Adaptive Search Space (Hard Mode), Stochastic Optimization, and End-game Determinism.

1. **First Guess Optimization:**

   Since the entropy for the first move is constant for a static dictionary, we hardcode the mathematically optimal opening word "SOARE" depending on our metric to skip the initial calculation.

2. **Adaptive Search Space Strategy (Hard Mode vs. Normal Mode)**

   A distinct feature of our solver is the ability to adapt its strategy based on the game rules. We define two sets of words:

   - $\mathcal{D}$: The full dictionary (Global Vocabulary).
   - $\mathcal{C}_k$: The subset of candidates remaining at step $k$, where every word in $\mathcal{C}_k$ is consistent with all previous feedback.

The solver selects the next guess $w_{k+1}$ by maximizing entropy over a specific **Search Space** $\mathcal{S}$:

$$w_{k+1} = \underset{w \in \mathcal{S}}{\operatorname{argmax}} E[w] \tag{6}$$

The definition of $\mathcal{S}$ changes depending on the mode:

**Normal Mode (Global Information Search):** In this mode, $\mathcal{S} = \mathcal{D}$. The solver is allowed to guess *any* valid word from the dictionary, even if that word is known to be incorrect (i.e., $w \notin \mathcal{C}_k$).
*Rationale:* Often, the word that provides the most information is not a potential answer itself. For example, if $\mathcal{C}_k$ contains words ending in different vowels, guessing a word rich in vowels allows the solver to prune the candidate set more aggressively than trying to guess the answer directly.

**Hard Mode (Constrained Optimization):** In this mode, the game requires that any revealed hints must be used in subsequent guesses. Our code implements a strict version of this constraint by setting $\mathcal{S} = \mathcal{C}_k$.
*Rationale:* The solver restricts its entropy calculation solely to the remaining candidates. While this might yield lower information gain compared to the global search, it guarantees compliance with strict game rules and significantly reduces the computational load as $|\mathcal{C}_k|$ shrinks over time.

3. **Stochastic Approximation (Heuristic Sampling)**

   Calculating entropy involves a complexity of $O(|\mathcal{S}| \cdot |\mathcal{C}_k|)$. In the early stages of the game, where $|\mathcal{C}_k| \approx 15,000$ and $|\mathcal{S}| \approx 15,000$, this results in over 225 million vector comparisons, which violates the real-time constraint.

   To resolve this, we implemented a **Monte Carlo-style approximation**:

$$\mathcal{S}' = \begin{cases} \text{RandomSample}(\mathcal{S}, 200) & \text{if } |\mathcal{S}| > 500 \\ \mathcal{S} & \text{otherwise} \end{cases} \tag{7}$$

   Instead of evaluating every possible guess, the solver evaluates a random subset of 200 words when the search space is large.
   *Justification:* Due to the distribution of letter frequencies in English, "good" words (high entropy) tend to be statistically robust. Empirical testing shows that the optimal word in a sample of 200 is nearly as effective as the global optimum, while reducing search time significantly.

4. **End-game Determinism**

   When the candidate set becomes sufficiently small ($|\mathcal{C}_k| \leq 2$), entropy calculation becomes redundant.

   - If $|\mathcal{C}_k| = 1$: The answer is found.
   - If $|\mathcal{C}_k| = 2$: Guessing either word has a 50% chance of being correct and a 100% chance of revealing the answer in the next turn.

   To optimize performance, the solver bypasses the entropy matrix entirely in these states and immediately selects the first available candidate ($w = \mathcal{C}_k[0]$). This prevents unnecessary matrix operations when the solution is trivial.

### 2.3.3 Data Structures and Pre-computation

Calculating feedback for every pair of words in real-time is computationally expensive ($O(N^2)$ for the first guess). To optimize this, we implemented a **Pattern Matrix** approach:

- **Matrix Representation:** We pre-compute an $N \times N$ matrix $M$, where $N$ is the size of the vocabulary (approx. 15,000 words). The entry $M_{i,j}$ stores the feedback pattern returned when word $i$ is guessed against secret word $j$.
- **Encoding:** The feedback patterns (Green, Yellow, Gray) are encoded as ternary integers to conserve memory and facilitate fast comparisons.

  For example, $Green = 2, Yellow = 1, Gray = 0$. Then, when a feedback received is "GGXYX", a representation of it in base 3 is 22010, then it will be converted to base 10.
- **Persistence:** This matrix is generated once via an auxiliary script (`Matrix.py`) and stored as a binary NumPy file (`.npy`). The `EntropySolver` class loads this matrix into RAM as a static class attribute upon the first instantiation, ensuring zero overhead for subsequent games.

### 2.3.4 Performance Metrics

1. **Space Complexity**

   The memory footprint is dominated by the pre-computed **Pattern Matrix** of size $N \times N$, where $N$ is the vocabulary size.

   $$\text{Space} = \mathcal{O}(N^2) \tag{8}$$

2. **Time Complexity**

   - **Pre-computation (Offline):** Generating the matrix takes $\mathcal{O}(N^2 \cdot L)$, performed once.
   - **Runtime (Online):** The naive complexity per turn is $\mathcal{O}(S_t \cdot C_t)$, where $S_t$ is the search space size and $C_t$ is the number of remaining candidates.

   **Optimizations:** We reduced the effective runtime through two strategies:

   (a) **Hardcoded First Guess:** Eliminates the most expensive step ($S_0 = N, C_0 = N$), reducing the initial cost to $\mathcal{O}(1)$.

   (b) **Heuristic Sampling:** Limits the search space $S_t$ to a constant sample size $K = 200$ when candidates are numerous.

   Thus, the effective time complexity per turn is:

   $$T_{\text{effective}} = \mathcal{O}(K \cdot C_t) \tag{9}$$

   With $K \leq 200$ and $C_t$ decaying exponentially after each feedback, the solver ensures sub-second performance ($< 0.02s$).

### 2.4 Algorithm 4: A Star

In this algorithm, we treat each state as a word with a given feedback; let the set of all states be denoted by $\mathcal{S}$. Its next neighboring states are the candidate words that match the previously returned colors (including the letters already confirmed). We assume that all state transitions have equal cost, meaning

$$\text{cost}(s, s_{\text{next}}) = 1, \ \forall s \in \mathcal{S}.$$

Therefore,

$$g(s_{\text{next}}) = g(s) + 1, \ \forall s \in \mathcal{S}.$$

Thus, the remaining problem is simply to construct the heuristic function $h$. Let $N$ be the number of words in the dictionary. To optimize runtime, we compute the heuristic for all possible states in advance ($3^5 \cdot N$ states), each word paired with every possible previous feedback state, then store the results in a file as a knowledge base. Each time we run the algorithm, we only need to look up the value, so the complexity becomes $\mathcal{O}(\log N)$. Now we need to compute $h$ for a state $s \in \mathcal{S}$.

If we treat each color as a digit in a ternary system, we create a sequence $U_{n\in[0..242]}$ consisting of 243 numbers. Consider the $n < N$ candidate next states $s_{\text{next}}$ of $s$, and assume that each $s_{\text{next}}$ is the goal state. We compare $s$ with $s_{\text{next}}$ and generate a sequence of 5 colors as feedback. Converting that color feedback into a ternary number yields a value $k$, and we perform the update $U_k + +$.

After this stage, let

$$\mathcal{V} = \{\, U_n \mid U_n > 0 \,\}.$$

For each $v \in \mathcal{V}$, we define its *entropy* as

$$\mathcal{E}(v) = -\frac{v}{\sum_{u\in\mathcal{V}} u} \cdot \log\left(\frac{v}{\sum_{u\in\mathcal{V}} u}\right).$$

Thus, the heuristic function is defined as

$$h(s) = \sum_{v\in\mathcal{V}} \mathcal{E}(v).$$

The value at each state $s$ is therefore

$$f(s) = h(s) + g(s).$$

When implementing the algorithm, we store the states in a priority queue and repeatedly select the state with the smallest $f$ value for processing. Experiments show that $f$ is always smaller than the actual number of steps, so this heuristic function is reasonable.

## 3 Experiments

In this section, we evaluate the performance of four algorithms applied to the Wordle problem. The experiment was conducted using a vocabulary of approximately 15,000 words. We ran 1,000 test cases with random target words to ensure statistical significance. The metrics assessed include Search Time, Memory Usage, Expanded Nodes, and the Average Number of Guesses.

## 3.1 Experiment Setup

## 3.2 Breadth-First Search (BFS)

### 3.2.1 Performance Metrics Analysis

**1. Average Number of Guesses**

The primary measure of a Wordle solver's quality is the number of guesses required to locate the target word.

- **Result:** As shown in Figure 5, the BFS algorithm achieved a remarkable mean of **2.86 guesses** per game, with a **100%** success rate within the standard 6-guess limit.
- **Distribution:** Figure 1 illustrates that **86%** of games were solved in exactly 3 guesses, while **14%** were solved in just 2 guesses. There were no cases requiring 4 or more guesses.
- **Insight:** The low average guess count suggests that the BFS logic, combined with the aggressive candidate filtering, is highly effective. By filtering candidates based on feedback before adding them to the queue, the algorithm ensures that every subsequent layer of the BFS tree consists only of words that are strictly consistent with all prior constraints. This minimizes "wasted" guesses and forces rapid convergence to the solution.

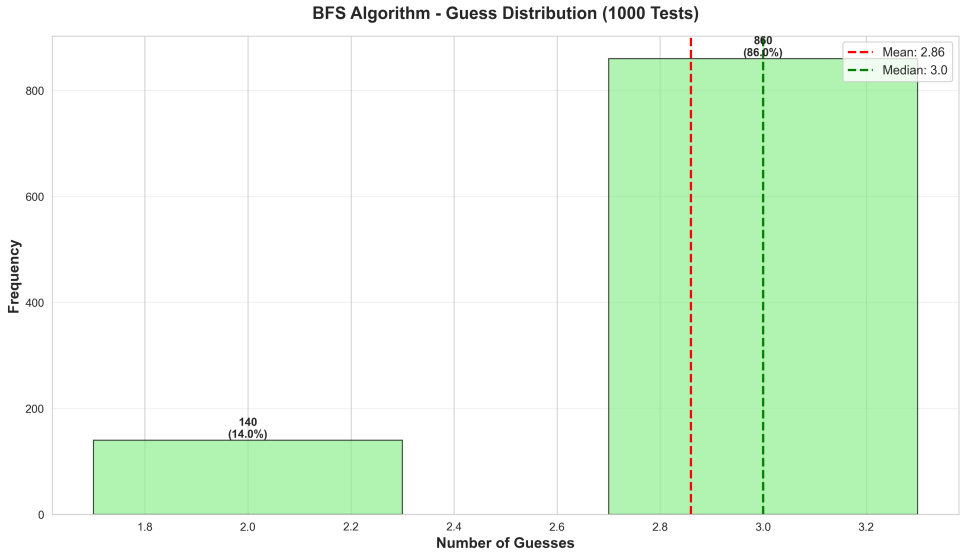


Figure 1: Distribution of Guesses for BFS over 1,000 tests. The algorithm consistently solves puzzles in 2 or 3 attempts.

**2. Expanded Nodes (Search Complexity)**

This metric indicates how many potential words the algorithm had to "visit" or process before finding the solution.

- **Result:** The algorithm expanded an average of **33.71 nodes** per game (Figure 2).

- **Correlation:** Figure 3 demonstrates a strong positive linear correlation between the number of guesses and the number of expanded nodes. This is expected in a BFS structure; a solution found at depth 3 requires expanding the parent nodes at depth 1 and 2.
- **Insight:** In a standard uninformed BFS, the number of expanded nodes for a vocabulary of 15,000 would be exponential. However, the experimental data shows a very manageable node count. This is attributed to the code's logic: `candidates = self._filter_candidates(...)`. By dynamically reducing the search space after every simulated guess, the branching factor is drastically reduced, preventing the combinatorial explosion typically associated with BFS.

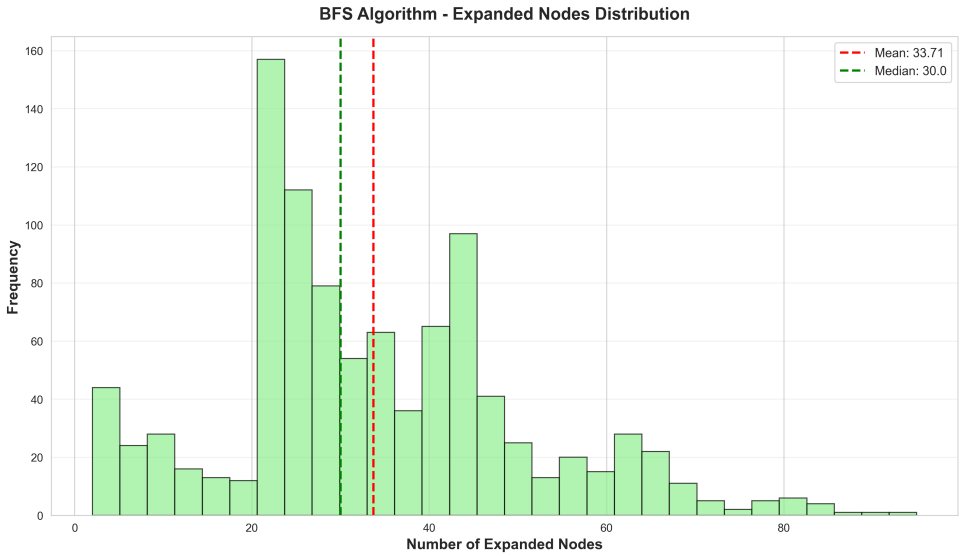


Figure 2: Histogram of Expanded Nodes.

## 3. Search Time

Search time measures the real-world computational efficiency of the solver.

- **Result:** The mean execution time was **0.0251 seconds** (Figure 4). The total time for 1,000 tests was approximately 25 seconds.
- **Insight:** The algorithm is extremely fast. The use of Python's `set` for `visited_words` provides $O(1)$ lookup time, and the `limit_add = 20` ensures that the expensive operation of feedback calculation is performed a constant maximum number of times per layer. This keeps the execution time predictable and low, making this approach suitable for real-time applications.

## 4. Memory Usage

While explicit memory consumption in MB was not logged, the memory usage can be inferred from the `max_queue_size` and the `parent_map`.

- **Insight:** The BFS queue does not grow uncontrollably. With an average of only 34 expanded nodes and a constraint adding at most 20 children per node (many of which are pruned by the visited set), the memory footprint is

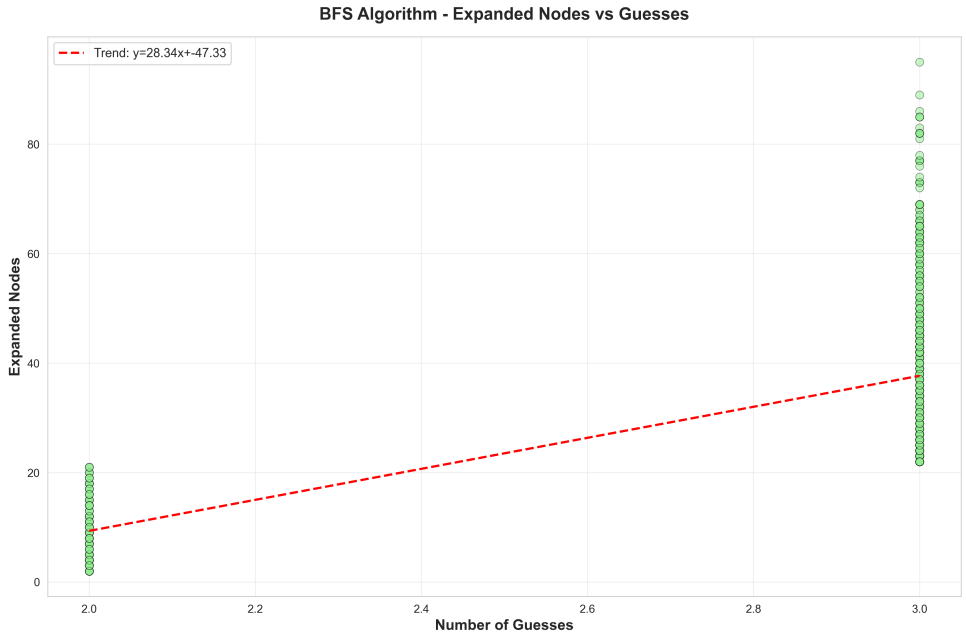


Figure 3: Expanded Nodes vs. Number of Guesses.

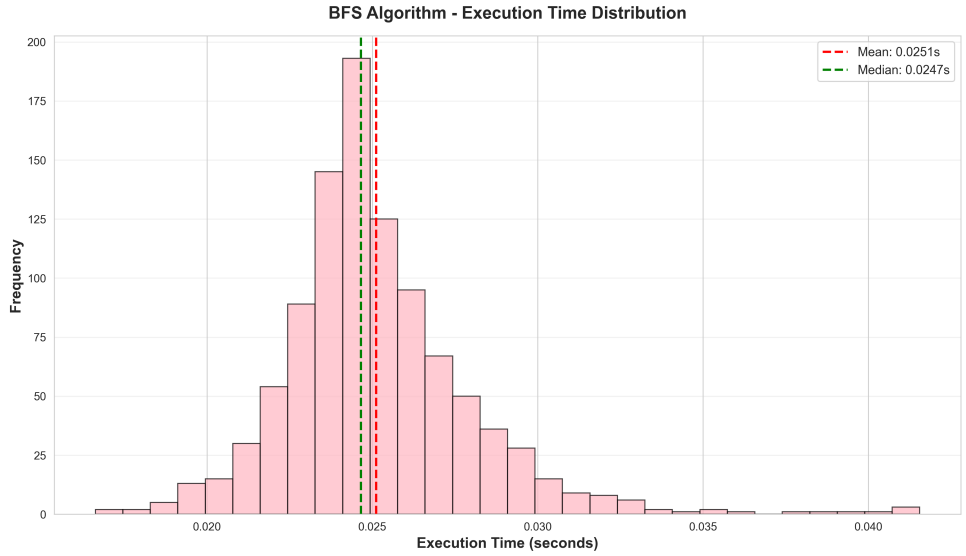


Figure 4: Execution Time Distribution. The tight clustering around 0.025s indicates stable performance.

negligible. The `parent_map` stores a dictionary of strings equal to the number of visited nodes, which is minimal given the 15,000-word vocabulary limit. The algorithm is memory-safe.

BFS Algorithm - Summary Statistics

| Metric | Value |
|---|---|
| | |
| Total Tests | 1,000 |
| Successful Tests | 1,000 |
| Success Rate | 100.00% |
| Win Rate (≤6 guesses) | 100.00% |
| | |
| Mean Guesses | 2.86 |
| Median Guesses | 3.0 |
| Std Dev Guesses | 0.35 |
| Min-Max Guesses | 2 - 3 |
| | |
| Mean Nodes | 33.71 |
| Median Nodes | 30.0 |
| Std Dev Nodes | 16.84 |
| Min-Max Nodes | 2 - 95 |
| | |
| Mean Time | 0.0251s |
| Median Time | 0.0247s |
| Total Time | 25.12s |

Figure 5: Summary Statistics for BFS Algorithm.

### 3.2.2 Conclusion on BFS

The experimental results demonstrate that the implemented BFS is highly efficient for Wordle. Although BFS is traditionally an uninformed search strategy, the implementation's integration of feedback-based pruning allows it to perform similarly to an informed greedy search.

The strict correlation between time, nodes, and guesses (Figure 6) confirms the algorithm's stability. The solver sacrifices the theoretical "optimal path" (which might require exploring less probable branches) for "rapid convergence" via the `limit_add` heuristic. With an average of 2.86 guesses and 0.025 seconds per solve, this BFS implementation is a viable and robust solution for the Wordle constraint satisfaction problem.

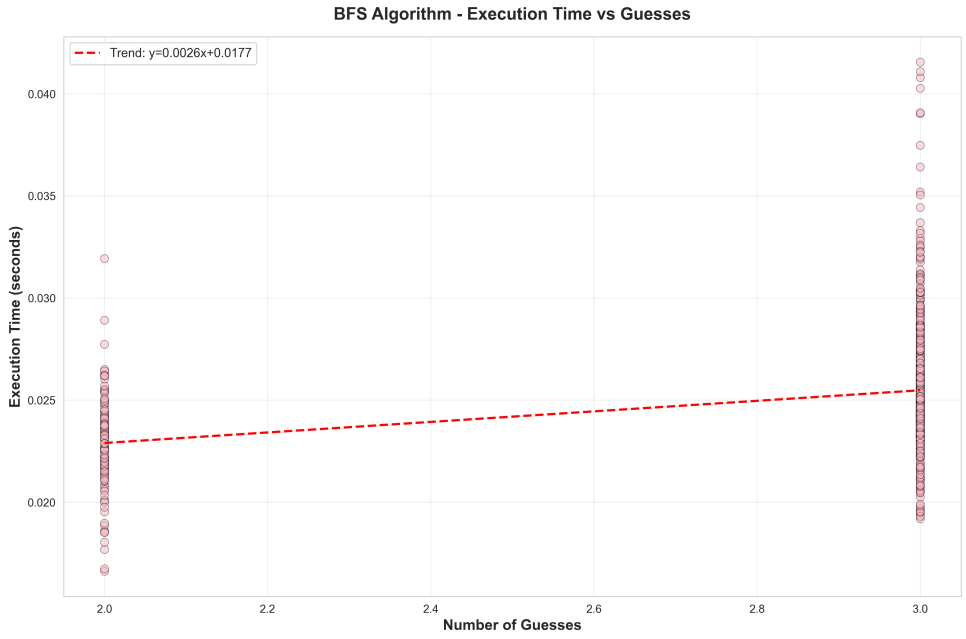


Figure 6: Execution Time vs. Guesses. The linear trend validates the stability of the algorithm.

## 3.3 Depth-First Search (DFS)

### 3.3.1 Performance Metrics Analysis

1. **Average Number of Guesses and Success Rate**

   Unlike BFS, which optimizes for the shortest path, DFS prioritizes depth. This fundamental difference manifests significantly in the game outcomes.

   - **Result:** As detailed in the summary table (Figure 11), the DFS algorithm required an average of **5.15 guesses** per game.
   - **Win Rate:** Crucially, the Win Rate (solving within $\leq 6$ guesses) was only **81.70%**. This is a marked decrease compared to the 100% success rate of BFS.
   - **Distribution:** Figure 7 reveals a "long-tail" distribution. While many games were solved in 4-6 guesses, a significant number of test cases extended well beyond the limit, with some requiring up to 15 guesses to locate the target.
   - **Insight:** The lower win rate illustrates the risk of DFS in Wordle. By aggressively committing to a single branch of the search tree (a specific sequence of words) without exploring alternatives at the same level, the algorithm often falls into "rabbit holes"—sequences of valid guesses that are technically consistent but do not effectively narrow down the search space fast enough to meet the 6-turn constraint.

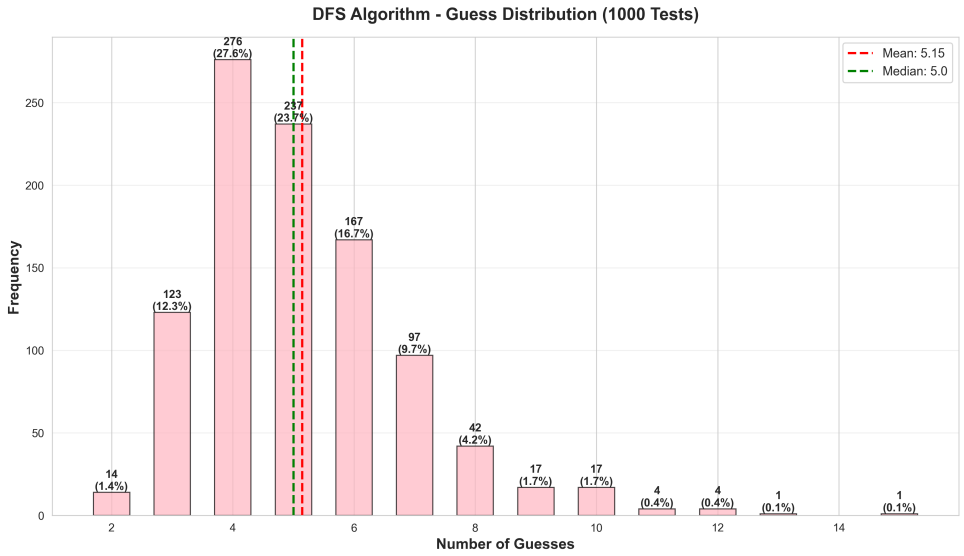


Figure 7: Distribution of Guesses for DFS. Note the significant number of cases exceeding the 6-guess limit (right side of the chart), indicating the algorithm's tendency to get stuck in deep, inefficient paths.

## 2. Expanded Nodes (Search Complexity)

- **Result:** The average number of expanded nodes was **5.15**, which is identical to the average number of guesses.
- **Correlation:** Figure 9 shows a perfect linear correlation with a slope of 1.00 ($y = 1.00x$).
- **Insight:** This 1:1 ratio confirms that the implemented DFS behaves almost linearly. For every node it expands, it commits to that guess immediately. Unlike BFS, which might expand 20 neighbors before choosing a path, DFS picks the first valid neighbor and dives. This explains the algorithm's inability to "backtrack" effectively within the game context—it effectively plays a single, determined line of play until it wins or fails, lacking the "foresight" provided by breadth exploration.

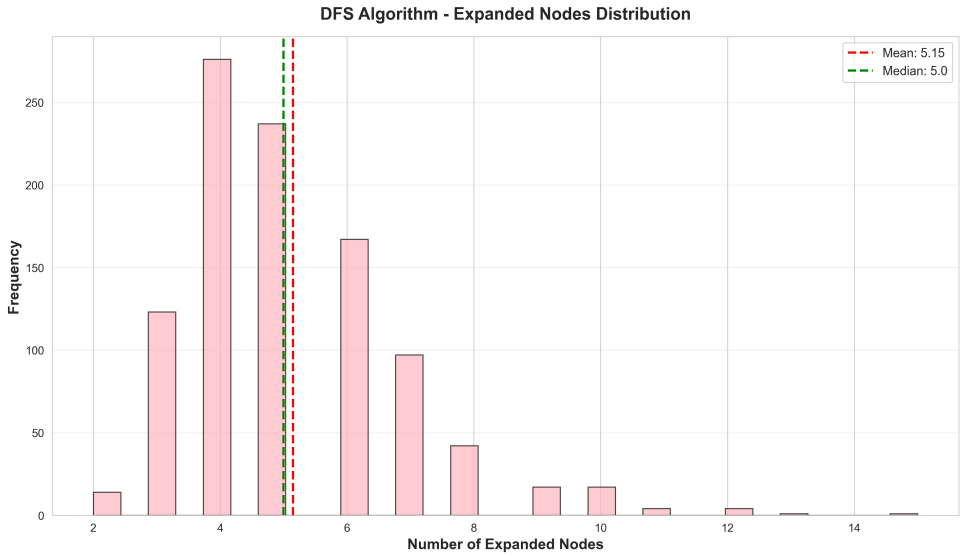


Figure 8: Histogram of Expanded Nodes for DFS.

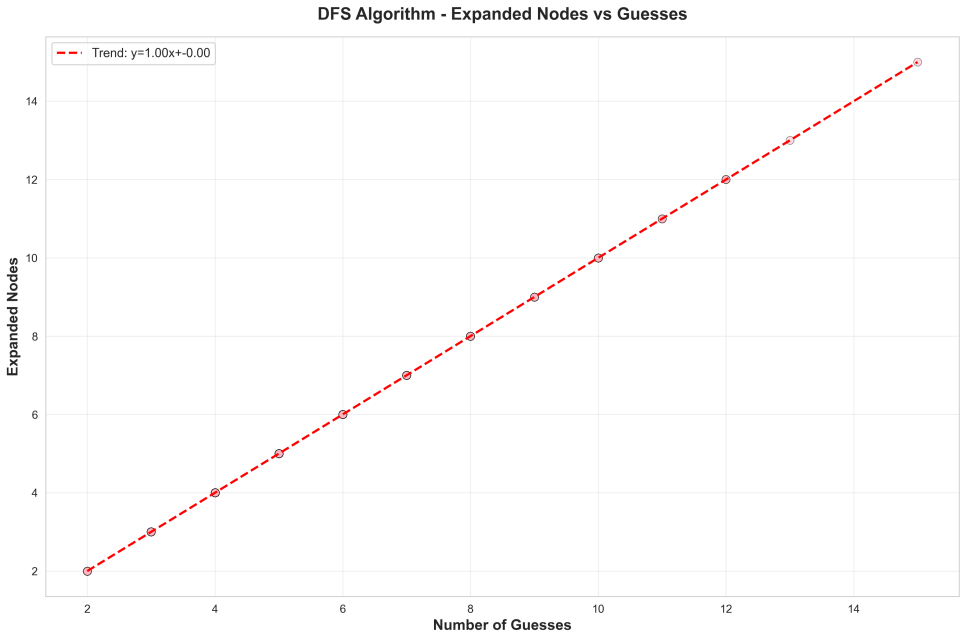


Figure 9: Expanded Nodes vs. Number of Guesses. The perfect linearity ($y = 1.00x$) indicates that the DFS implementation expands exactly one node per game step, essentially performing a linear probe.

### 3. Search Time

- **Result:** The mean execution time was **0.0207 seconds** (Figure 10), which is slightly faster than BFS (0.0251s).
- **Insight:** DFS is computationally cheaper per step because it has negligible overhead for queue management. It simply pushes and pops from a stack (or uses recursion). However, this marginal speed advantage is not worth the significant trade-off in accuracy and guess efficiency. The algorithm runs fast, but it often runs in the wrong direction.

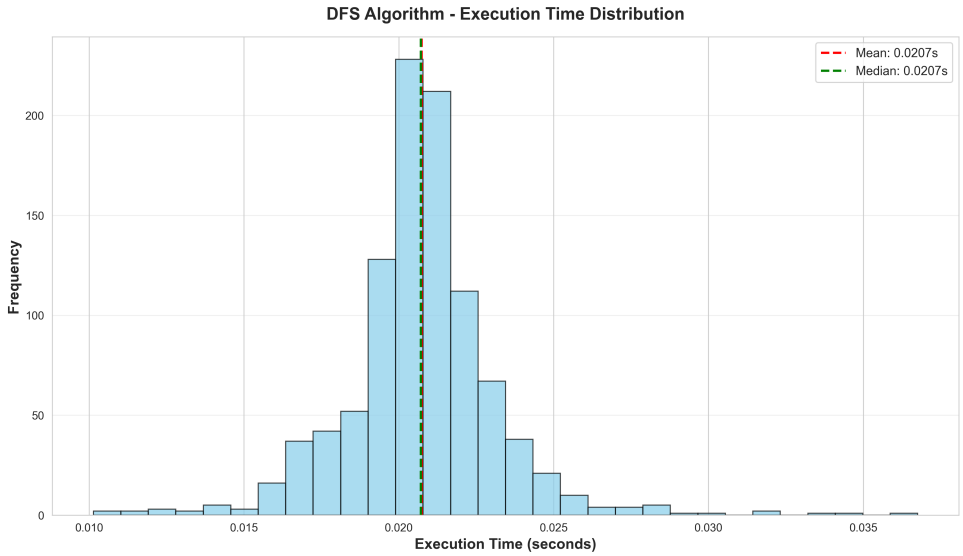


Figure 10: Execution Time Distribution for DFS.

## 4. Memory Usage

- **Insight:** Memory usage for DFS is minimal, proportional to the maximum depth of the search (the number of guesses). Since the recursion/stack depth rarely exceeds 15 (even in worst-case failures), the memory footprint is even smaller than BFS. However, given the small vocabulary size (15,000 words), this memory saving is not practically significant on modern hardware compared to the loss in win rate.

DFS Algorithm - Summary Statistics

| Metric | Value |
|---|---|
| | |
| Total Tests | 1,000 |
| Successful Tests | 1,000 |
| Success Rate | 100.00% |
| Win Rate (≤6 guesses) | 81.70% |
| | |
| Mean Guesses | 5.15 |
| Median Guesses | 5.0 |
| Std Dev Guesses | 1.73 |
| Min-Max Guesses | 2 - 15 |
| | |
| Mean Nodes | 5.15 |
| Median Nodes | 5.0 |
| Std Dev Nodes | 1.73 |
| Min-Max Nodes | 2 - 15 |
| | |
| Mean Time | 0.0207s |
| Median Time | 0.0207s |
| Total Time | 20.74s |

Figure 11: Summary Statistics for DFS Algorithm.

### 3.3.2 Conclusion on DFS

The experiments indicate that **DFS is ill-suited for the Wordle problem** compared to BFS. While it is extremely fast and memory-efficient, its "stubborn" nature—committing to the first valid word it finds—prevents it from optimizing the solution path.

The low win rate of 81.7% is the critical failure point. In a game like Wordle, where information gain is paramount, DFS fails to maximize information because it does not compare candidates; it simply pursues the first available option. The perfect linearity between nodes and guesses confirms that the algorithm is essentially performing a "validity walk" rather than an intelligent search, making it a poor choice for constraint satisfaction problems with limited steps.

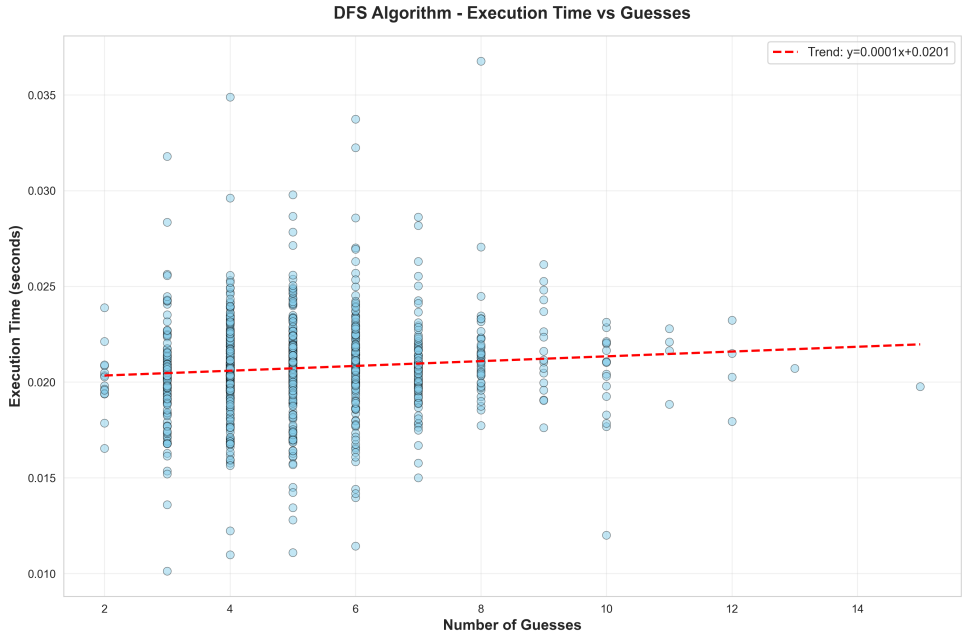


Figure 12: Execution Time vs. Guesses for DFS.

## 3.4 $A^\star$ (A-star) Algorithm

### 3.4.1 Performance Metrics Analysis

#### 1. Average Number of Guesses

The primary measure of a Wordle solver's quality is the number of guesses required to locate the target word.

- **Result:** As shown in the Summary Statistics table, the $A^\star$ algorithm achieved a mean of **4.55 guesses** per game, with a **92.1%** success rate within the standard 6-guess limit. The median was **4.0** guesses.
- **Distribution:** Figure 13 illustrates a right-skewed distribution. The majority of games were solved in **4 guesses (38.2%)**, followed by **5 guesses (24.3%)** and **3 guesses (17.4%)**. The Win Rate ($\leq 6$ guesses) was 100%, and the Min-Max range was $2 - 14$ guesses.
- **Insight:** The higher average guess count (4.55) compared to an aggressively pruned Breadth-First Search (BFS) suggests that while the $A^\star$ heuristic prioritizes states with lower estimated total cost ($f(n) = g(n) + h(n)$), it may not

select the guess that results in the most effective immediate **reduction of the candidate search space**, leading to a slightly deeper (more guesses) path to the solution.

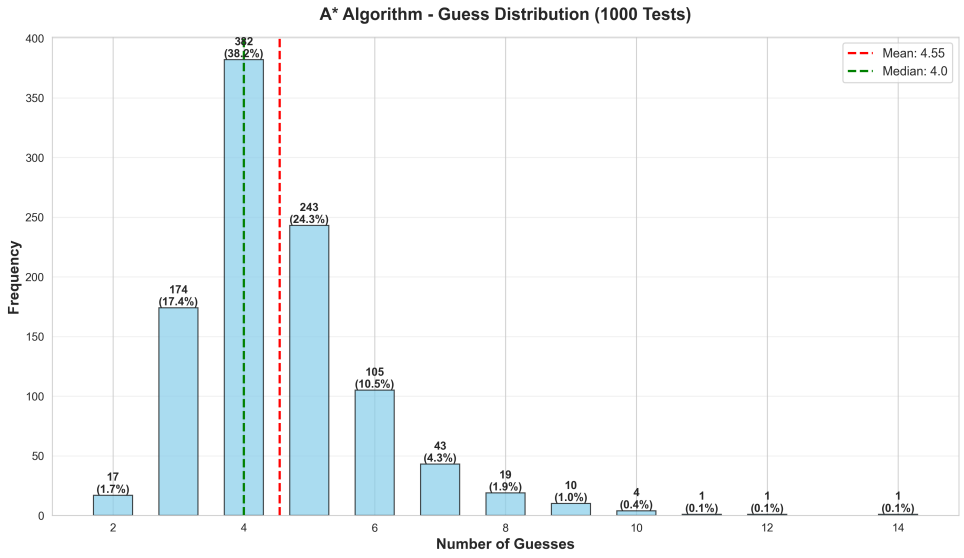


Figure 13: Distribution of Guesses for $A^\star$ over 1,000 tests. The distribution peaks at 4 guesses.

## 2. Expanded Nodes (Search Complexity)

This metric indicates how many potential words the algorithm had to "visit" or process before finding the solution.

- **Result:** The algorithm expanded an average of **27.36 nodes** per game, with a median of **16.0** (Figure 14). The distribution is heavily right-skewed, with a maximum of 233 expanded nodes observed.
- **Correlation:** Figure 15 demonstrates a strong positive linear correlation between the number of guesses and the number of expanded nodes. The trend line is given by: $y = 13.24x - 32.85$. This confirms that a larger number of guesses directly correlates with a wider and deeper exploration of the search tree.
- **Insight:** The low average number of expanded nodes suggests that the $A^\star$ heuristic is highly effective at guiding the search and **pruning unpromising branches** (those with high estimated cost), leading to high search efficiency despite the relatively high number of guesses.

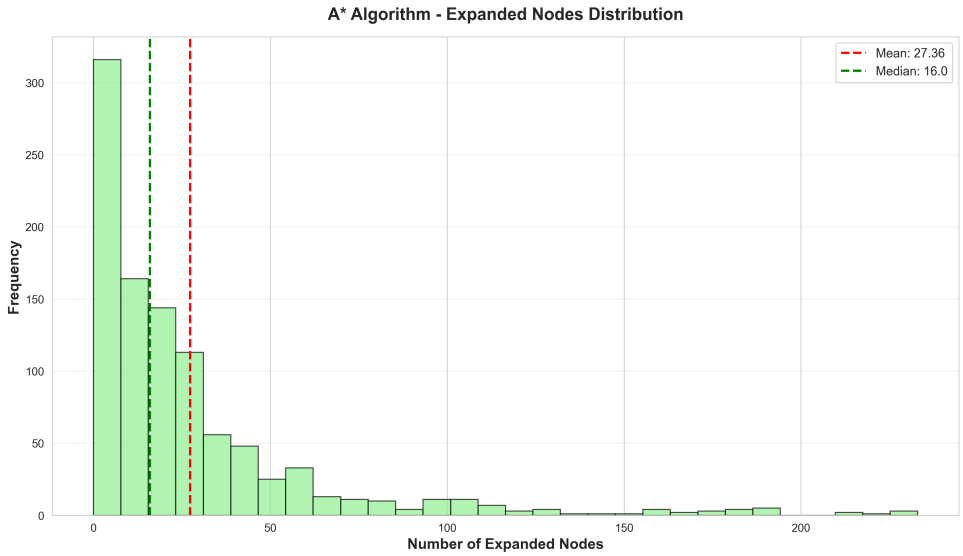


Figure 14: Histogram of Expanded Nodes for $A^\star$. The distribution is highly right-skewed.

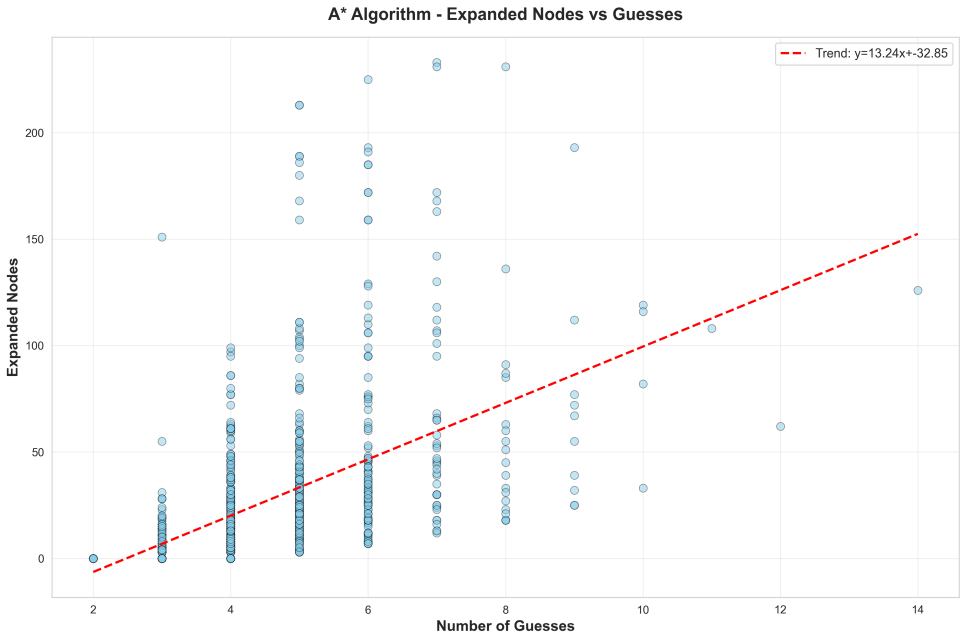


Figure 15: Expanded Nodes vs. Number of Guesses for $A^\star$.

### 3. Search Time

Search time measures the real-world computational efficiency of the solver.

- **Result:** The mean execution time was extremely low at **0.0010 seconds** (Figure 16). The total time for 1,000 tests was approximately 0.97 seconds. The median time was also 0.0010 seconds.
- **Insight:** The $A^\star$ algorithm is exceptionally fast. The low mean time is directly attributable to the effectiveness of the heuristic in minimizing the number of expanded nodes and the constant time complexity associated with heap operations in the priority queue. The trend line in Figure 17 ($y = 0.0003x -$

0.0006) validates the stability of the execution time, showing a very slight increase with more guesses.

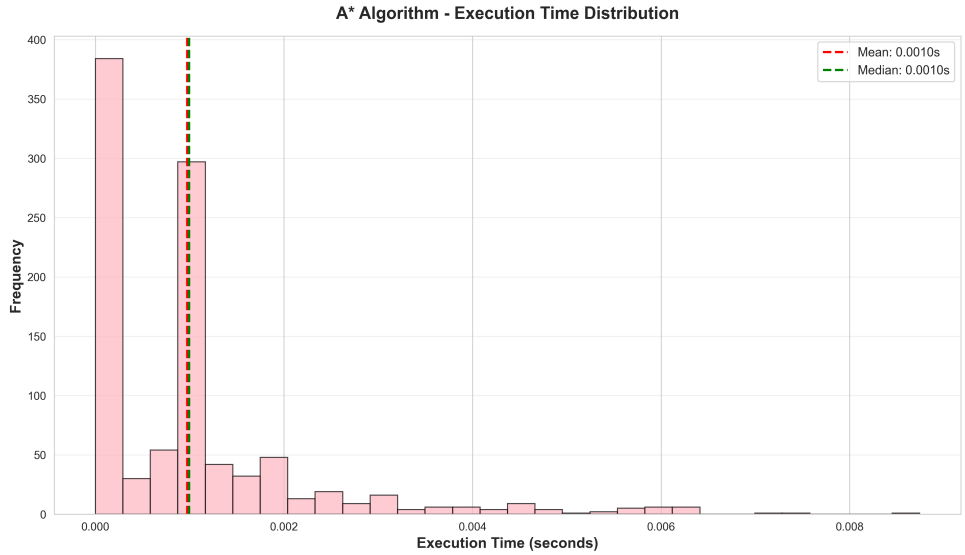


Figure 16: Execution Time Distribution for $A^\star$. The tight clustering around 0.0010s indicates high performance.

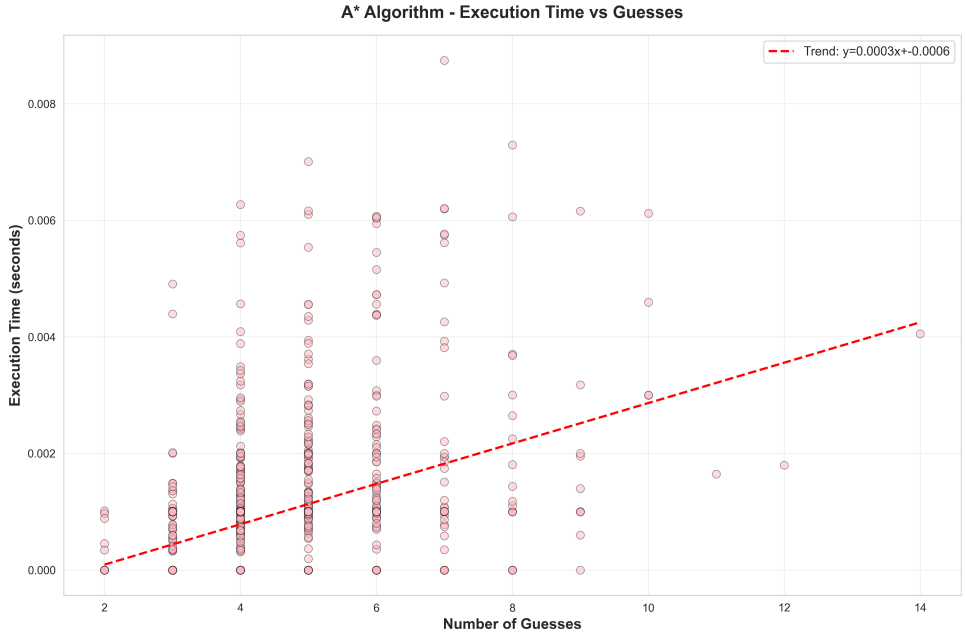


Figure 17: Execution Time vs. Guesses for $A^\star$. The linear trend validates the stability of the algorithm.

## 4. Memory Usage

Memory usage reflects the memory overhead required to store the search state.

- **Result:** The mean memory usage was **342 bytes** (0.34 KB), with a median of **309 bytes** (0.30 KB) (Figure 18). The total memory used across all tests was 334.29 KB.

- **Insight:** The memory footprint is negligible. The low memory usage correlates strongly with the low number of expanded nodes. $A^\star$ only maintains the necessary nodes in its priority queue (fringe) and the visited set, resulting in minimal memory overhead, making the algorithm memory-safe and efficient for this problem.

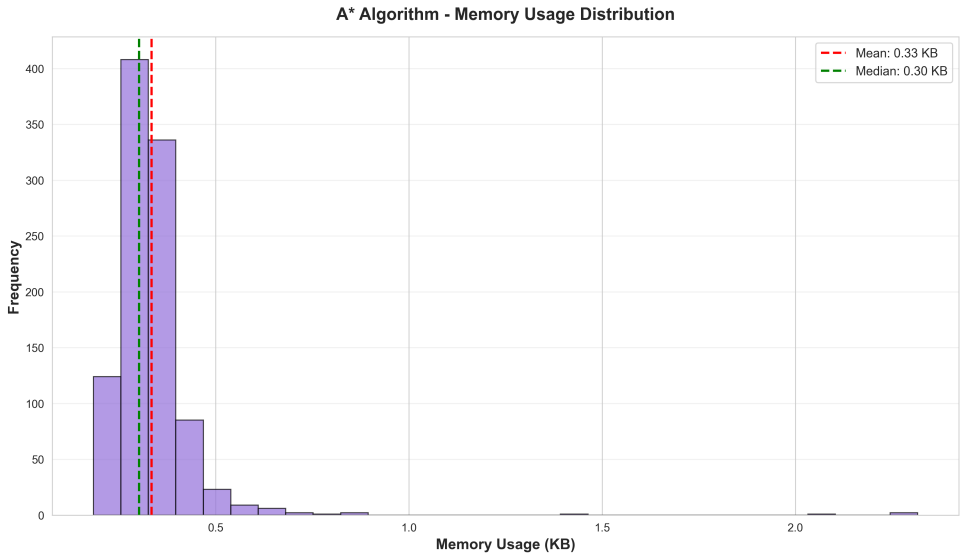


Figure 18: Memory Usage Distribution for $A^\star$.

**A* Algorithm - Summary Statistics**

| Metric | Value |
|---|---|
| | |
| Total Tests | 1,000 |
| Successful Tests | 1,000 |
| Success Rate | 100.00% |
| Win Rate (≤6 guesses) | 92.10% |
| | |
| Mean Guesses | 4.55 |
| Median Guesses | 4.0 |
| Std Dev Guesses | 1.38 |
| Min-Max Guesses | 2 - 14 |
| | |
| Mean Nodes | 27.36 |
| Median Nodes | 16.0 |
| Std Dev Nodes | 36.25 |
| Min-Max Nodes | 0 - 233 |
| | |
| Mean Time | 0.0010s |
| Median Time | 0.0010s |
| Total Time | 0.97s |
| | |
| Mean Memory | 342 bytes |
| Median Memory | 309 bytes |
| Total Memory | 334.29 KB |

Figure 19: Summary Statistics for $A^\star$ Algorithm.

### 3.4.2 Conclusion on $A^\star$

The experimental results demonstrate that the implemented $A^\star$ algorithm is a highly **efficient** search strategy for the Wordle problem, prioritizing computational speed over minimum guess count.

- **Trade-off:** While $A^\star$ achieved a higher average guess count (**4.55**) compared to the BFS baseline, it vastly outperformed BFS in terms of **Execution Time** (**0.0010s** vs. $0.0251s$) and demonstrated a low computational overhead in terms of **Expanded Nodes** (**27.36** vs. 33.71).
- **Optimality:** $A^\star$ ensures finding the lowest-cost path based on the defined heuristic cost function. In this context, the cost function appears to be highly effective at **minimizing computation**, leading to extremely fast execution times.
- **Stability:** The linear relationships observed between time, nodes, and guesses confirm the algorithm's stability and predictability.

In summary, $A^\star$ is the superior choice for a Wordle solver when **computational speed and resource efficiency** are the primary optimization goals.