PROJECT 1

Artificial Intelligence

Wordle Solver

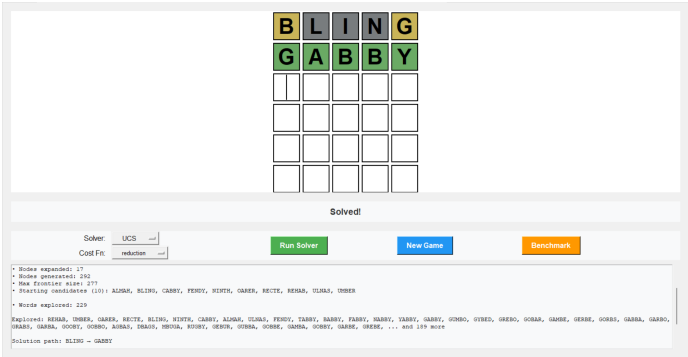
23125029 – Nguyễn Tiến Đạt
23125089 – Đoàn Công Pho
23125092 – Nguyễn Đức Thịnh
23125033 – Trần Phước Hải

December 3, 2025

Contents

1 Introduction

Wordle is a popular word-guessing puzzle in which players attempt to identify a hidden five-letter word within a maximum of six guesses. After each guess, the game provides feedback by coloring each letter to indicate its correctness: green for correct letter in the correct position, yellow for correct letter in the wrong position, and gray for letters not present in the target word. In the implementation used in this repository[1], the same rules apply, and the game interface is illustrated in Figure 1.



Graphical interface of the Wordle game implementation used in this study.

This document presents an empirical study of classical search algorithms when applied to the Wordle-solving problem. The objective is to examine the trade-offs between search time, memory usage, node expansion.

Our demo is availabled here: https://youtu.be/byle3x37xtk

2 Problem Modeling

The standard approach to solving Wordle involves a Minimax-style search or a greedy search. However, treating it as a shortest-path problem on a graph allows us to use classical algorithms like BFS and A*. Hence, we model Wordle as a deterministic single-agent search problem where a state represents the solver's knowledge about possible answers (the set of remaining candidate words and the history of guesses with feedback). Formally:

- **State**: Represented by `CompactState`, a hashable data structure that uniquely identifies a position in the search space:
  - `history`: A tuple of (guess, feedback) pairs encoding all guesses made and their corresponding feedback patterns
  - `remaining_count`: The number of candidate words still consistent with all observed feedback

  Example state after one guess:

[1] https://github.com/TienDat8605/wordle.git

```
CompactState(
    history=(("crane", (MISS, PRESENT, CORRECT, MISS, CORRECT)),),
    remaining_count=47
)
```

This represents a state where the guess "CRANE" received feedback [-YG-G] (gray, yellow, green, gray, green), narrowing down the 14,856 initial words to 47 candidates that satisfy the constraints: position 2 = 'A', position 4 = 'E', 'R' present but not at position 1, and no 'C' or 'N' anywhere.

- **Initial State**: The root state with an empty history and all words from the word pool as candidates:

  ```
  CompactState(history=(), remaining_count=14856)
  ```

  The word pool contains 14,856 five-letter words loaded from `valid_solutions.csv`.

- **Goal State**: A state where the most recent guess matches the hidden target word. This is detected when:

  ```
  if history and word_to_idx[history[-1][0]] == answer_idx:
      return success
  ```

  Equivalently, the goal is reached when the feedback pattern is all greens: (CORRECT, CORRECT, CORRECT, CORRECT, CORRECT).

- **Actions**: Making a guess from the word pool. To maintain computational tractability, the branching factor is limited:
  - At depth 0 (root state): Try up to 30 words from a random sample of `starting_candidates`
  - At depth > 0: Try up to 30 words from the `remaining_count` candidates of the current state

  Example actions from the root state:

  ```
  Actions = {CRANE, SLATE, PRIDE, STARE, ..., AUDIO}  # 30 words
  ```

- **Cost**: Algorithm-dependent. For BFS-like behavior a constant step cost is used; UCS variants experiment with information-driven costs (e.g. penalizing large remaining sets and rewarding high-information guesses).

- **State Transitions**: Applying a guess produces deterministic feedback (green/yellow/gray) that filters the candidate set to form the next state. Transitions are computed using the project's FeedbackTable, which provides cached (sparse) lookups and falls back to on-the-fly feedback computation when necessary. When an action (guess) is executed, the state transitions as follows:

  Transition Function: $\delta(\text{state}, \text{guess}) \rightarrow \text{new\_state}$

1. **Get Feedback**: Simulate the feedback pattern by comparing the guess against the target answer:

```
feedback = evaluate_guess(answer="stare", guess="crane")
# Returns: [MISS, PRESENT, CORRECT, MISS, CORRECT]
```

2. **Filter Candidates**: For each word w in the current `possible_indices`, check if guessing `guess` with w as the answer would produce the same feedback:

```
new_possible = {w | evaluate_guess(w, guess) == feedback}
```

   This ensures only words consistent with *all* observed feedback remain.

3. **Update History**: Append the (guess, feedback) pair:

```
new_history = history + ((guess, feedback),)
```

4. **Create New State**:

```
new_state = CompactState(
    history=new_history,
    remaining_count=len(new_possible)
)
```

Concrete transition example:

Before transition:

```
state_0 = CompactState(history=(), remaining_count=14856)
```

Action: Guess "CRANE" (target answer is "STARE")

Step 1 - Feedback:

- C $\neq$ S $\rightarrow$ MISS
- R $\in$ "STARE" but R $\neq$ T $\rightarrow$ PRESENT
- A = A $\rightarrow$ CORRECT
- N $\notin$ "STARE" $\rightarrow$ MISS
- E = E $\rightarrow$ CORRECT

Feedback = [MISS, PRESENT, CORRECT, MISS, CORRECT]

Step 2 - Filter: From 14,856 words, keep only those where:

- Position 2 = 'A' (green)
- Position 4 = 'E' (green)
- Contains 'R' but not at position 1 (yellow)

– No 'C' anywhere (gray)

– No 'N' anywhere (gray)

Filtering all 14,856 words, maybe 47 of them satisfy these constraints. Hence, there are 47 candidate words for this state. Hence, the candidate list might look like this: {STARE, SHARE, SPARE, SNARE, ...}
After transition:

```
state_1 = CompactState(
    history=(("crane", [MISS, PRESENT, CORRECT, MISS, CORRECT]),),
    remaining_count=47
)
```

This new state along with the candidate list are then added to the frontier with priority $f(n) = g(n) + h(n)$ for further exploration.

3 Optimization Techniques

A naive implementation would fail due to the massive branching factor (14,856 possible guesses initially) and exponential state space. We implemented four key optimizations to make graph search tractable:

3.1 Sparse Feedback Table (Memory Optimization)

3.1.1 What is a Feedback Table?

A **feedback table** is a precomputed lookup dictionary that stores Wordle feedback patterns for pairs of words. Given a guess word g and a target word t, the feedback is a 5-tuple of marks (e.g. CORRECT, PRESENT, MISS, MISS, PRESENT) indicating the correctness of each letter position.

Example feedback table entry:

```
Key: ("crane", "stare")
Value: [MISS, PRESENT, CORRECT, MISS, CORRECT]
         (CS,  R"stare", A=A,  N"stare", E=E)
Symbol representation: [-YG-G]
```

This feedback is computed by `evaluate_guess(target, guess)` in `feedback.py`.

3.1.2 Standard (Dense) Feedback Table

A **complete feedback table** stores feedback for *all* possible $(guess, target)$ pairs:

- Total entries: $N \times N = 14{,}856 \times 14{,}856 \approx 220{,}698{,}736$ entries
- Memory requirement: Each feedback is 5 marks $\times$ 1 byte $\approx$ 5 bytes
- Total memory: $220{,}698{,}736 \times 5 \approx 1.1$ GB (uncompressed)

- With Python object overhead: ∼17 GB of RAM
- **Problem:** Causes Out-Of-Memory (OOM) errors on standard hardware

3.1.3 Sparse Feedback Table Solution

We implemented a **sparse graph structure** that stores a maximum of 200 connections per word instead of all N connections:

Mechanism:

1. For each word w in the word pool:
   - Always store the self-loop: $(w, w) \rightarrow$ [GGGGG] (all correct)
   - Randomly sample 199 other words as targets (deterministic seed=42)
   - Store feedback for these 200 pairs only
2. Total entries: $14{,}856 \times 200 = 2{,}971{,}200$ entries
3. Memory reduction: $\frac{2{,}971{,}200}{220{,}698{,}736} \approx 1.35\%$ (98.65% reduction)
4. Actual memory usage: ∼180 MB vs. ∼17 GB

Fallback mechanism:
When a requested pair (g, t) is not in the cache, the system computes feedback on-the-fly:

```
def get_feedback(self, guess: str, target: str) -> Feedback:
    key = (guess.lower(), target.lower())
    if key in self._table:
        return self._table[key]  # O(1) cache hit
    return evaluate_guess(target, guess)  # O(1) fallback computation
```

Example sparse table entries:
For the word "CRANE", the table stores:

```
("crane", "crane") → [GGGGG]           # Self-loop (always cached)
("crane", "stare") → [-YG-G]           # Sampled connection
("crane", "share") → [--G-G]           # Sampled connection
...                                    # ~197 more sampled pairs
("crane", "audio") → not cached        # Computed on-the-fly if needed
```

Impact: Reduces memory by ∼98.6% while maintaining O(1) lookup speed for cached pairs and O(1) computation for cache misses. This makes the solver viable on standard hardware.

3.2 Fast Candidate Filtering

After receiving feedback for a guess, the solver must filter the remaining candidates to those consistent with the observed feedback. This is the computational bottleneck of the search.

3.2.1 Naive Approach: Word-by-Word Constraint Checking

A naive implementation would check each candidate against the learned constraints letter-by-letter:

```
# For each remaining candidate word:
for candidate in remaining_words:
    # Check if position 2 == 'A'
    if candidate[2] != 'a':
        continue
    # Check if position 4 == 'E'
    if candidate[4] != 'e':
        continue
    # Check if 'R' is present but not at position 1
    if 'r' not in candidate or candidate[1] == 'r':
        continue
    # Check if 'C' and 'N' are absent
    if 'c' in candidate or 'n' in candidate:
        continue
    # ... more constraint checks
    filtered_candidates.append(candidate)
```

Cost: O(candidates × constraints × word_length) operations per filtering step.

3.2.2 Optimized Approach: Feedback Table Lookup

Instead of checking constraints individually, the solver uses the sparse feedback table:

```
def _filter_candidates_fast(
    possible_indices: Set[int],
    guess_idx: int,
    observed_feedback: Feedback,
    word_list: List[str],
    feedback_table: FeedbackTable,
) -> Set[int]:
    result = set()
    for idx in possible_indices:
        candidate = word_list[idx]
        guess = word_list[guess_idx]
        # Single lookup: if this candidate were the answer,
        # would it produce the same feedback?
        if feedback_table.get_feedback(guess, candidate) == observed_feedback:
            result.add(idx)
    return result
```

Concrete example:
Scenario: After guessing "CRANE" with feedback [-YG-G], filter 500 remaining candidates.
Naive approach: 500 words × 5 constraints × 5 letters = 12,500+ operations
Optimized approach:

- For each of 500 candidates:
  - Lookup: `get_feedback("crane", candidate)`
  - If cached: O(1) dictionary lookup
  - If not cached: O(1) `evaluate_guess` computation
  - Compare: Does it equal [-YG-G]?
- Total (best case - when all of the candidate words have a matching feedback in the feedback table calculated above) : 500 lookups $\times$ O(1) = 500 operations

Example filtering:

```
observed_feedback = [MISS, PRESENT, CORRECT, MISS, CORRECT]  # [-YG-G]

# Candidate 1: "STARE"
feedback_table.get_feedback("crane", "stare")
  → [MISS, PRESENT, CORRECT, MISS, CORRECT]
  == observed_feedback? YES → Keep "STARE"

# Candidate 2: "BRAIN"
feedback_table.get_feedback("crane", "brain")
  → [MISS, CORRECT, CORRECT, MISS, MISS]  # Different!
  == observed_feedback? NO → Discard "BRAIN"

# Candidate 3: "SPARE"
feedback_table.get_feedback("crane", "spare")
  → [MISS, MISS, CORRECT, MISS, CORRECT]
  == observed_feedback? NO → Discard "SPARE" (no 'R' present)
```

Benefit: This method greatly accelerates candidate filtering by replacing per-letter constraint evaluation with a single feedback comparison. When the required (guess, candidate) pair is already present in the sparse feedback table, the lookup is an $O(1)$ dictionary access, resulting in extremely fast filtering. However, because the table stores only ~ 200 preselected feedback entries per word, the actual speedup depends on whether the solver is "lucky"—i.e., whether the candidates encountered during search happen to use precomputed pairs. In these cases, filtering can be orders of magnitude faster than the naive approach. When a pair is not cached, the system falls back to computing the feedback on demand, which is still inexpensive ($O(5)$ operations). Overall, the approach substantially reduces the cost of the filtering step, which is executed thousands of times during the search and would otherwise dominate the runtime.

3.3 Compact State Representation

3.3.1 Problem

Storing the full list of remaining candidates (potentially thousands of words) in every frontier node consumes excessive memory and slows down hashing/equality checks for the visited set in graph search.

Naive approach memory cost per state (assuming each state results in about 1000 candidate words):

- List of candidate words: ∼1,000 candidates × 5 bytes/word = 5 KB
- Plus Python list overhead: ∼50 KB per state
- With 10,000 states in frontier: ∼500 MB

3.3.2 Solution: CompactState

We use a lightweight `CompactState` dataclass:

```
@dataclass(frozen=True)
class CompactState:
    history: Tuple[Tuple[str, Tuple[Mark, ...]], ...]
    remaining_count: int
```

What it stores:

- `history`: Compressed history signature (guesses + feedback patterns)
- `remaining_count`: Just the count, not the actual word list

Memory cost per state:

- History tuple: ∼100 bytes (for typical 3-4 guess history)
- Remaining count: 8 bytes (integer)
- Total: ∼108 bytes vs. ∼50 KB (∼450× reduction)

Example CompactState representations:

1. **Initial state (root):**

   ```
   CompactState(
       history=(),
       remaining_count=14856
   )
   ```

   Memory: 8 bytes

2. **After one guess "CRANE":**

   ```
   CompactState(
       history=(("crane", (MISS, PRESENT, CORRECT, MISS, CORRECT)),),
       remaining_count=47
   )
   ```

   Memory: ∼100 bytes

3. **After three guesses:**

```
CompactState(
    history=(
        ("crane", (MISS, PRESENT, CORRECT, MISS, CORRECT)),
        ("stare", (CORRECT, MISS, CORRECT, MISS, CORRECT)),
        ("share", (CORRECT, CORRECT, CORRECT, CORRECT, MISS))
    ),
    remaining_count=3
)
```

Memory: ∼180 bytes

Key insight: The actual candidate list (`possible_indices`) is stored separately in the frontier tuple and is only used during expansion:

```
# When pushing to frontier:
heapq.heappush(frontier, (
    priority,           # f(n) for A*
    sequence,           # Tiebreaker
    compact_state,      # Lightweight CompactState (~100 bytes)
    history,            # Full history tuple
    possible_indices,   # Actual candidate indices (not in CompactState)
    depth               # Accumulated cost g(n)
))
```

Benefit: Fast hashing for the visited set, minimal memory overhead, and efficient state comparison while keeping candidate lists available when needed.

3.4 Branching Factor Control

3.4.1 Problem

At the root of the search tree, all 14,856 words are valid guesses. Without branching control, the state space grows exponentially:

- **Depth 0 (root):** 1 state (initial state with all 14,856 candidates possible)
- **Depth 1:** ∼14,856 child states
  - Each of the 14,856 possible first guesses creates one child state
  - Each child state represents a different guess-feedback combination
  - Example: Guessing "CRANE" creates $\text{state}_{\text{CRANE}}$, guessing "SLATE" creates $\text{state}_{\text{SLATE}}$, etc.
- **Depth 2:** Potentially millions of states
  - From each depth-1 state, we can make another guess

- If each depth-1 state still has $\bar{k}$ remaining candidates on average (e.g., $\bar{k} \approx 100$), we expand each state into $\sim$100 child states
- Total states at depth 2: $14{,}856 \times \bar{k} \approx 14{,}856 \times 100 = 1{,}485{,}600$ states
- *Concrete example:*
  - State after "CRANE": 47 remaining candidates $\rightarrow$ 47 child states
  - State after "SLATE": 23 remaining candidates $\rightarrow$ 23 child states
  - State after "AUDIO": 150 remaining candidates $\rightarrow$ 150 child states
  - Summing across all 14,856 depth-1 states yields millions of depth-2 states

- **Depth 3 and beyond:** Combinatorial explosion continues
  - Even with reducing candidate sets, the total number of states grows exponentially
  - Estimated depth-3 states: tens of millions
  - Memory requirement: $\sim$50 KB per state $\times$ millions of states = hundreds of GB
  - Time requirement: Expanding millions of nodes becomes computationally prohibitive

Why the explosion occurs:
The branching factor remains high because:

1. At depth 0, we have 14,856 possible actions (guesses)
2. At depth 1, each state typically still has 50-200 valid remaining candidates
3. The product rule: Total states = (states at level $d-1$) $\times$ (avg. branching per state)
4. This results in exponential growth: $\mathcal{O}(b^d)$ where b is the average branching factor

Without branching control, memory and time requirements become infeasible for depth ≥ 3.

3.4.2 Solution: Adaptive Branching Limit

We limit the **branching factor** to a maximum of 30 guesses per state (configurable via `max_branching` parameter):

```
def _select_guesses(self, possible_indices: Set[int],
                    depth: float) -> List[int]:
    if depth == 0:
        # Root: use pre-selected starting candidates
        candidates = list(self.starting_candidates_indices
                          & possible_indices)
    else:
        # Subsequent depths: use remaining possible words
        candidates = list(possible_indices)

    if len(candidates) <= self.max_branching:
        return candidates

    # Limit to first max_branching candidates
    return candidates[:self.max_branching]
```

Strategy:

- **Depth 0 (root):** Use 30 pre-selected diverse starting words (randomly sampled from word pool)
- **Depth > 0:** Consider the first 30 valid candidates from the remaining set

Concrete example:
Scenario: Solving with target word "STARE"

1. **At root (depth 0):**
   - Total possible guesses: 14,856 words
   - Branching limit: 30
   - Selected guesses: [CRANE, SLATE, AUDIO, PRIDE, ..., STALE] (30 words)
   - States generated: 30 child states instead of 14,856
2. **After guessing "CRANE" (depth 1):**
   - Remaining candidates: 47 words
   - Branching limit: 30
   - Selected guesses: First 30 from the 47 candidates
   - States generated: 30 child states
3. **After guessing "STARE" (depth 2):**
   - Remaining candidates: 5 words (SHARE, SNARE, STALE, STAVE, SCARE)
   - Branching limit: 30
   - Selected guesses: All 5 (less than limit)
   - States generated: 5 child states

Impact analysis:
Without branching control:

$$\text{Total states at depth 2} \approx 14{,}856 \times \bar{f}_1 \times \bar{f}_2 \approx \text{millions}$$

With branching factor = 30:

$$\text{Total states at depth 2} \approx 30 \times 30 = 900 \text{ states}$$

Benefit: Prevents combinatorial explosion while maintaining solution quality. The solver remains focused on promising paths, making the search tractable without exhaustive exploration.

4 Cost and Heuristic Functions

The performance of UCS and A* search algorithms heavily depends on the choice of cost and heuristic functions. These functions guide the search toward optimal solutions while maintaining computational efficiency.

4.1 Cost Functions $g(n)$

Cost functions determine the accumulated path cost from the root to a given node. The solver implements multiple cost functions, each optimizing for different objectives. We focus on two primary implementations:

4.1.1 Constant Cost Function

Formula:

$$c(n) = 1.0$$

Description:
The constant cost function assigns a uniform cost of 1.0 to every guess, regardless of its effectiveness in reducing the candidate space. This treats the problem as finding the shortest path measured purely by the number of guesses.

Behavior in search algorithms:

- **UCS:** Reduces to Breadth-First Search (BFS), expanding nodes level-by-level
- **A*:** The heuristic $h(n)$ becomes the primary guide, with $g(n)$ serving only to track depth

Concrete example:
Target word: "STARE", Word pool: 14,856 words

1. **First guess: "CRANE"**
    - Before: 14,856 candidates
    - Feedback: [-YG-G] (C miss, R present, A correct, N miss, E correct)
    - After: 47 candidates
    - Step cost: $c_1 = 1.0$
    - Accumulated cost: $g(n_1) = 1.0$
2. **Second guess: "STARE"**
    - Before: 47 candidates
    - Feedback: [GGGGG] (all correct)
    - After: 1 candidate (goal reached)
    - Step cost: $c_2 = 1.0$
    - Accumulated cost: $g(n_2) = 1.0 + 1.0 = 2.0$

Result: The path cost is exactly 2.0, representing a 2-guess solution. This cost function treats a highly effective guess (14,856 $\rightarrow$ 47) the same as a moderately effective guess.

Use case: Finding solutions with the minimum number of guesses, without regard to information efficiency.

4.1.2 Candidate Reduction Cost Function

Formula:

$$c(n) = 1.0 + \frac{|\text{candidates after}|}{|\text{candidates before}|}$$

Description:

The reduction cost function penalizes guesses that fail to eliminate many candidates. The ratio $\frac{\text{after}}{\text{before}}$ measures the fraction of candidates remaining after the guess. Lower cost favors aggressive elimination strategies.

Mathematical properties:

- **Range:** $c(n) \in [1.0, 2.0)$
  - Best case: Eliminates all but 1 candidate $\to c(n) \approx 1.0$
  - Worst case: Eliminates nothing $\to c(n) \to 2.0$
- **Gradient:** The function is continuous and monotonic in the reduction ratio

Concrete example:

Target word: "STARE", Word pool: 14,856 words

1. **First guess: "CRANE"**
   - Before: 14,856 candidates
   - After: 47 candidates
   - Reduction ratio: $\frac{47}{14{,}856} \approx 0.00316$
   - Step cost:
   $$c_1 = 1.0 + 0.00316 = 1.00316$$
   - Accumulated cost: $g(n_1) = 1.00316$
   - *Interpretation:* Extremely effective guess (eliminated 99.68% of candidates)
2. **Second guess: "STARE"**
   - Before: 47 candidates
   - After: 1 candidate
   - Reduction ratio: $\frac{1}{47} \approx 0.02128$
   - Step cost:
   $$c_2 = 1.0 + 0.02128 = 1.02128$$
   - Accumulated cost: $g(n_2) = 1.00316 + 1.02128 = 2.02444$
   - *Interpretation:* Very effective guess (eliminated 97.87% of remaining candidates)

Result: The total path cost is 2.02444, slightly higher than the constant cost of 2.0, but the small penalty reflects the high information gain of both guesses.

Comparison of guess effectiveness:

Consider two alternative first guesses:

- **Guess A: "CRANE"** (as above)
  - 14,856 $\rightarrow$ 47 candidates
  - $c_A = 1.0 + \frac{47}{14,856} = 1.00316$
- **Guess B: "XYZZY"** (hypothetical poor guess)
  - 14,856 $\rightarrow$ 10,000 candidates (only eliminated 33%)
  - $c_B = 1.0 + \frac{10,000}{14,856} = 1.0 + 0.673 = 1.673$

The search algorithm will prefer paths through Guess A ($c_A = 1.00316 < c_B = 1.673$), naturally favoring information-rich guesses.

Use case: Optimizing for solutions that maximize information gain per guess, leading to efficient elimination strategies.

4.2 Heuristic Function $h(n)$ (A* Only)

Heuristic functions estimate the cost from the current node to the goal. For A* to guarantee optimal solutions, the heuristic must be **admissible** (never overestimate the true cost).

4.2.1 Log2 Heuristic (Information-Theoretic Optimal)

Formula:

$$h(n) = \log_2(\max(1, |\text{remaining candidates}|))$$

Description:

The log2 heuristic represents the information-theoretic minimum: the number of binary questions (yes/no) needed to uniquely identify one item from N possibilities. This is derived from information theory, where each perfect guess ideally provides 1 bit of information by bisecting the search space.

Admissibility proof:

To prove $h(n)$ never overestimates:

1. In the *best case*, each guess perfectly bisects the candidate space
2. Starting with N candidates, after k perfect bisections:
$$N \rightarrow \frac{N}{2} \rightarrow \frac{N}{4} \rightarrow \cdots \rightarrow \frac{N}{2^k}$$
3. To reach 1 candidate: $\frac{N}{2^k} = 1 \Rightarrow k = \log_2(N)$
4. Therefore, the true minimum cost to goal is at least $\lceil \log_2(N) \rceil$ guesses
5. Since $h(n) = \log_2(N) \leq \lceil \log_2(N) \rceil$, the heuristic never overestimates

Concrete example:

Target word: "STARE", Word pool: 14,856 words

1. **Initial state (root):**

- Remaining candidates: 14,856
- Heuristic:
$$h(n_0) = \log_2(14{,}856) \approx 13.86$$
- *Interpretation:* At least ∼14 perfect guesses needed in theory

2. **After first guess "CRANE":**
   - Remaining candidates: 47
   - Path cost: $g(n_1) = 1.00316$ (with reduction cost function)
   - Heuristic:
$$h(n_1) = \log_2(47) \approx 5.55$$
   - Priority (A*):
$$f(n_1) = g(n_1) + h(n_1) = 1.00316 + 5.55 = 6.55316$$
   - *Interpretation:* Estimated ∼5-6 more guesses needed
3. **After second guess "STARE":**
   - Remaining candidates: 1 (goal!)
   - Path cost: $g(n_2) = 2.02444$
   - Heuristic:
$$h(n_2) = \log_2(1) = 0.0$$
   - Priority:
$$f(n_2) = 2.02444 + 0.0 = 2.02444$$
   - *Interpretation:* Goal reached, no more guesses needed

Heuristic accuracy analysis:

- **Initial estimate:** $h(n_0) = 13.86$ guesses
- **Actual solution:** 2 guesses
- **Underestimation:** Yes, by a factor of ∼7
- **Why acceptable:** Admissibility only requires $h(n) \leq h^*(n)$ (true cost), not tight bounds. The actual Wordle feedback provides much more than 1 bit of information per guess due to the 5-position color-coded response (up to $3^5 = 243$ possible feedback patterns).

Progressive refinement example:
Tracking how the heuristic updates as search progresses:

State	Remaining	$h(n)$	Estimate
Root	14,856	13.86	∼14 guesses
After CRANE	47	5.55	∼6 guesses
After STARE	1	0.00	0 guesses (goal)

The heuristic provides increasingly accurate estimates as the search narrows down candidates.
Use case: Default heuristic for A* search, providing theoretically grounded estimates that guarantee optimal solutions while efficiently guiding the search.

5 Algorithm Description

5.1 BFS Solver

5.1.1 Algorithm

Breadth-First Search (BFS) explores the search space level-by-level using a FIFO (First-In-First-Out) queue. At each depth level, all states are expanded before proceeding to the next level. This systematic exploration guarantees finding the solution with the minimum number of guesses.

The algorithm maintains a frontier (queue) and a visited set to avoid exploring duplicate states.

5.1.2 Search Execution Example

Consider searching for the answer "STALE" with the starting guess "CRANE":

Depth 0 (Initial State):

- State: s_0 with 14,856 possible words
- History: [] (empty)
- Frontier: $[s_0]$

Depth 1 (After "CRANE"):

- Pop s_0 from frontier
- Select up to 30 guesses from starting candidates
- Guess "CRANE" against answer "STALE" $\rightarrow$ feedback: `[-,Y,Y,-,Y]`
- Filter 14,856 words to 47 candidates (containing C, R, A, E but not in those positions)
- Create new state: s_1 with history=[("CRANE", `[-,Y,Y,-,Y]`)], remaining=47
- Add s_1 to end of frontier
- Similarly expand other 29 starting guesses $\rightarrow$ frontier contains $\sim$30 states

Depth 2 (After "STARE"):

- Pop next state s_1 from frontier (FIFO order)
- Select up to 30 guesses from the 47 remaining candidates
- Guess "STARE" against "STALE" $\rightarrow$ feedback: `[G,G,G,G,Y]`
- Filter 47 words to 1 candidate: "STALE"
- Next expansion will guess "STALE" $\rightarrow$ all correct $\rightarrow$ **goal found!**

BFS expands all states at depth 1 before any states at depth 2, ensuring the solution path with minimum guesses is found first.

5.1.3 Properties

- **Complete**: Yes — BFS will find a solution if one exists within the maximum attempt limit (6 guesses), since the search space is finite (14,856 words).
- **Optimal**: Yes — BFS guarantees finding the solution with the minimum number of guesses because it explores all depth-d states before exploring any depth-$(d+1)$ states.
- **Time Complexity**: $O(b^d)$ where b is the branching factor ($\sim$30–100 per state) and d is the solution depth ($\sim$2–6 guesses). With branching control, typical expansions: $\sim$100–10,000 nodes.
- **Space Complexity**: $O(b^d)$ — highest among all algorithms. BFS must store the entire frontier at each level. At depth 2 with average branching $b = 100$: $100^2 = 10,000$ states in memory simultaneously.
- **Drawback**: High memory usage makes BFS impractical for deep searches or large branching factors without aggressive pruning (`max_branching=30`).

5.2 DFS Solver

5.2.1 Algorithm

Depth-First Search (DFS) explores the search space by diving deep into one branch before backtracking. It uses a LIFO (Last-In-First-Out) stack as the frontier, making it memory-efficient but potentially exploring many irrelevant branches before finding the solution.

5.2.2 Search Execution Example

Consider searching for "STALE" with starting guess "CRANE":

Depth 0:

- State: s_0 with 14,856 words
- Frontier: $[s_0]$

Depth 1 (After "CRANE"):

- Pop s_0
- Expand with 30 starting guesses
- Last guess added: "CRANE" $\rightarrow$ creates s_1 (47 remaining) — pushed to top of stack
- First guess added: "ADIEU" $\rightarrow$ creates s_{30} — at bottom of stack
- Frontier (top to bottom): $[s_1, s_2, \ldots, s_{30}]$

Depth 2 (DFS dives deep immediately):

- Pop s_1 from top (most recently added state)
- Expand with guesses from 47 remaining candidates

- If "STARE" is selected → filter to 1 candidate
- Push new state $s_{1,1}$ to stack top
- Continue diving: guess "STALE" at depth 3 → **goal found!**

Key difference from BFS: DFS immediately explores depth 2, 3, ... states from the "CRANE" branch before exploring other depth-1 branches (like "ADIEU"). This can find solutions quickly if the first branch is lucky, but may waste time exploring unpromising deep branches.

5.2.3 Properties

- **Complete**: No — DFS can get stuck exploring infinite paths or very deep branches. With depth limiting (max 6 guesses in Wordle), it becomes complete but still inefficient.
- **Optimal**: No — DFS returns the first solution found, which may require many more guesses than necessary. It does not explore states in order of depth.
- **Time Complexity**: $O(b^d)$ in the worst case, but can be faster or slower than BFS depending on solution location. May explore many deep, fruitless branches.
- **Space Complexity**: $O(bd)$ — much lower than BFS. Only stores nodes along the current path from root to frontier node. At depth $d = 6$ with branching $b = 30$: only ∼180 states vs. BFS's thousands.
- **Advantage**: Memory-efficient — suitable for deep searches with limited RAM.
- **Drawback**: Unreliable for Wordle — may find poor solutions (e.g., 5–6 guesses) when better solutions exist (2–3 guesses). Exploration order depends heavily on guess ordering.

5.3 Uniform-Cost Search (UCS)

5.3.1 Algorithm

Uniform-Cost Search (UCS) expands the state with the lowest accumulated path cost $g(n)$. Unlike BFS (which treats all steps equally), UCS uses a priority queue (min-heap) to prioritize states with lower total cost. This enables finding optimal solutions when guesses have different costs.

5.3.2 Search Execution Example

Consider searching for "STALE" with `cost_fn='reduction'`:
Cost function: $c(n) = 1 + \frac{\text{after}}{\text{before}}$ (rewards guesses that eliminate candidates)
Initial State:

- State: s_0, remaining: 14,856
- $g(s_0) = 0$
- Priority queue: $[(0.0, s_0)]$

Expand s_0 (guess "CRANE"):

- Guess "CRANE" against "STALE" → feedback: `[-,Y,Y,-,Y]`
- Filter: 14,856 → 47 candidates
- Step cost: $c_1 = 1 + \frac{47}{14856} = 1 + 0.00316 = 1.00316$
- New state s_1: $g(s_1) = 0 + 1.00316 = 1.00316$
- Add to queue: $[(1.00316, s_1), \ldots]$

Expand other starting guesses:

- Guess "ADIEU" → 92 remaining → $c = 1.00619$ → $g = 1.00619$
- Guess "SLATE" → 35 remaining → $c = 1.00236$ → $g = 1.00236$ (best so far!)
- Queue after all 30 starting guesses: states sorted by $g(n)$

UCS pops minimum $g(n)$ first:

- Pop state from "SLATE" guess ($g = 1.00236$, 35 remaining)
- Expand with guesses from 35 candidates
- If this branch reaches goal quickly, UCS finds it first

Eventually expand s_1 (from "CRANE"):

- $g(s_1) = 1.00316$, 47 remaining
- Guess "STARE" → filter 47 → 1 candidate
- Step cost: $c_2 = 1 + \frac{1}{47} = 1.02128$
- $g(s_2) = 1.00316 + 1.02128 = 2.02444$
- Next expansion: guess "STALE" → goal!

UCS explores states in order of increasing path cost, preferring branches that efficiently eliminate candidates.

5.3.3 Properties

- **Complete**: Yes — UCS will find a solution if one exists, since the search space is finite and all step costs are positive.
- **Optimal**: Yes — UCS guarantees finding the minimum-cost solution path when all step costs are non-negative (which holds for all cost functions: `constant`, `reduction`, `partition`, `entropy`).
- **Time Complexity**: $O(b^{C^*/\epsilon})$ where C^* is the optimal solution cost and ϵ is the minimum step cost. With typical costs ∼1.0–2.0 per step, complexity is similar to BFS but expands fewer nodes due to cost-based prioritization.

- **Space Complexity**: $O(b^{C^*/\epsilon})$ — similar to BFS, must maintain frontier of all generated states.
- **Advantage over BFS**: Finds optimal solutions with respect to path cost (not just number of guesses). With `cost_fn='reduction'`, prioritizes guesses that maximize information gain.
- **Disadvantage**: Slower than A* — lacks heuristic guidance to focus search toward the goal. Expands more nodes than A* but fewer than uninformed BFS when using informative cost functions.

5.4 A* Search

5.4.1 Algorithm

A* Search combines the path cost $g(n)$ from UCS with a heuristic estimate $h(n)$ of the remaining cost to the goal. States are expanded in order of $f(n) = g(n) + h(n)$, ensuring optimal solutions when the heuristic is admissible (never overestimates).

5.4.2 Priority Function $f(n)$ in A*

A* combines path cost and heuristic estimate:

$$f(n) = g(n) + h(n)$$

Complete example continuing from the CRANE guess:
Configuration: `cost_fn='reduction'`, `heuristic_fn='log2'`

1. **After guessing "CRANE":**
   - Remaining: 47 candidates
   - $g(n_1) = 1.003$ (path cost)
   - $h(n_1) = \log_2(47) = 5.55$ (heuristic)
   - $f(n_1) = 1.003 + 5.55 = 6.55$ (priority)
2. **After guessing "STARE":**
   - Remaining: 1 candidate (goal!)
   - $g(n_2) = 1.003 + 1.021 = 2.024$
   - $h(n_2) = \log_2(1) = 0.0$ (at goal)
   - $f(n_2) = 2.024 + 0.0 = 2.024$

States are expanded in order of increasing $f(n)$, ensuring optimal solutions when the heuristic is admissible.

5.4.3 Search Execution

The A* search proceeds as follows:

1. Initialize a priority queue (min-heap) with the root state
2. While the frontier is not empty:
   (a) Pop the state with minimum $f(n) = g(n) + h(n)$
   (b) If this state is visited, skip it (graph search duplicate detection)
   (c) Mark state as visited and increment expanded node count
   (d) If the last guess equals the answer, return success
   (e) If depth exceeds maximum attempts (6), skip this branch
   (f) Otherwise, expand: select up to 30 promising guesses
   (g) For each guess:
      - Simulate feedback using precomputed `FeedbackTable`
      - Filter candidates to those consistent with the feedback
      - Compute step cost c and new depth $g(n') = g(n) + c$
      - Compute $h(n')$ for the new state
      - Push new state to frontier with priority $f(n') = g(n') + h(n')$
3. If frontier is exhausted without finding the answer, return failure

5.4.4 Properties

- **Complete**: Yes — A* will find a solution if one exists within the maximum attempt limit
- **Optimal**: Yes — guaranteed to find the minimum-cost solution path because:
  - The heuristic function (`log2`) is admissible (never overestimate)
  - The graph search variant with consistent heuristics ensures optimality
- **Time Complexity**: $O(b^d)$ in the worst case, but typically much better due to heuristic guidance. Expands significantly fewer nodes than BFS/UCS.
- **Space Complexity**: $O(b^d)$ — maintains visited set and frontier, but typically smaller than BFS due to focused search.
- **Efficiency**: Significantly faster than BFS/DFS/UCS due to informed search; the heuristic guides expansion toward promising states

6 Experiments

6.1 Experimental Setup

All experiments were conducted using the GUI-based benchmark tool included in the Wordle AI Studio application. The benchmark configuration can be customized via two key parameters:

- **samples**: Number of random words to test each solver against
  - *Meaning*: Controls how many test cases are run
  - *Impact*: Higher values (e.g., 100+) provide more reliable statistics but increase runtime proportionally. Lower values (e.g., 3–10) enable rapid iteration during development
- **seed**: Random seed for word selection
  - *Meaning*: Initializes the random number generator to deterministically select test words
  - *Impact*: Identical seeds produce identical test sets, ensuring reproducibility. Different seeds test different word combinations, validating solver robustness across varied scenarios

Each selected answer uses the same randomly selected set of 30 starting candidates (also determined by the seed) across all solvers to ensure fair comparisons. This controlled setup isolates solver performance differences from random variation in test conditions.

6.1.1 Benchmarking Procedure

The benchmark was executed through the graphical interface in three steps:

Step 1: Launch Benchmark Dialog

Click the "Benchmark" button (orange button in the control panel) to open the benchmark configuration dialog.

Step 2: Configure Parameters

In the benchmark dialog, configure the following settings:

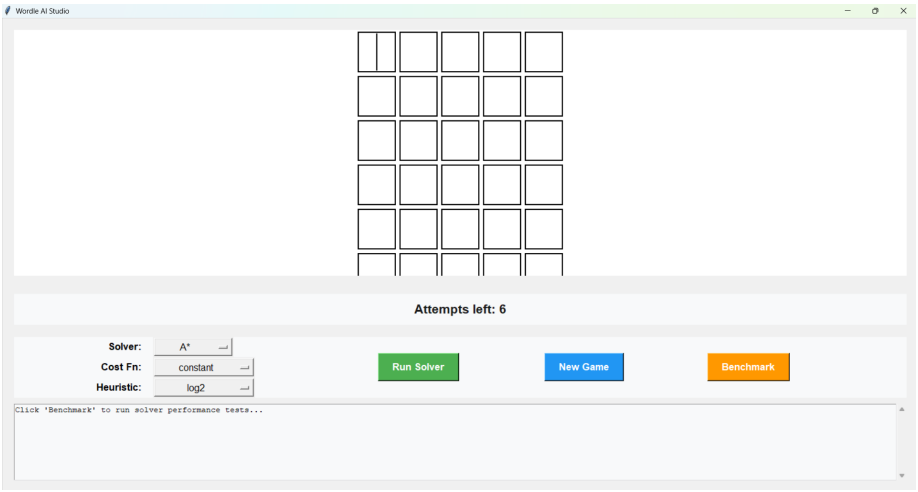
- Set **Number of samples** to the desired value (e.g., 50 for this report)
- Set **Random seed** to ensure reproducibility (e.g., 23125092 for this report)
- Select which solvers to benchmark using the checkboxes
- For UCS, select the cost function from the dropdown menu
- For A*, select both the cost function and heuristic function
- Click the "Run" button to start the benchmark

Step 3: View Results

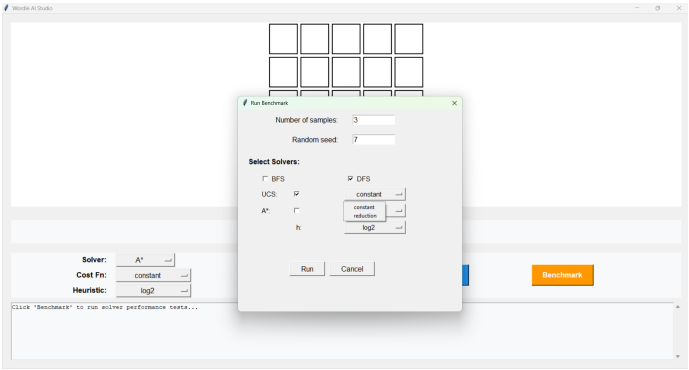
After completion, the benchmark results are displayed in a formatted table in the results area at the bottom of the application window. The table shows comprehensive performance metrics for each selected solver.

Alternative: Command-Line Benchmarking

For automated or batch testing, benchmarks can also be run from the command line (PowerShell):



Step 1: Click the Benchmark button to open the configuration dialog



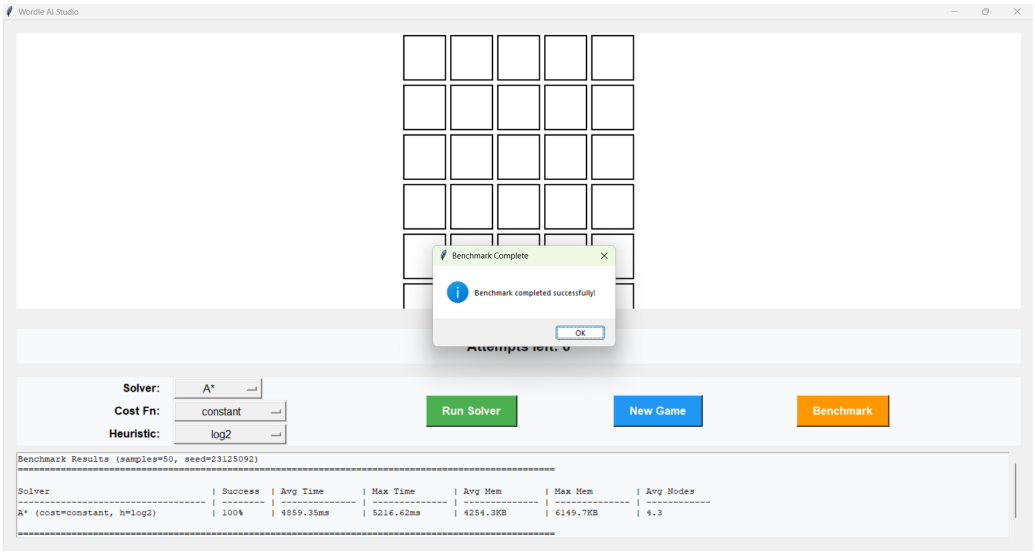
Step 2: Configure benchmark parameters (samples=50, seed=23125033) and select solvers

```
# Quick benchmark with default settings
$env:PYTHONPATH = (Get-Location).Path; python .\run_benchmarks.py

# Custom configuration (modify run_benchmarks.py):
print_benchmarks(samples=50, seed=23125092)
```

Notes:

- The first run builds a sparse feedback table (200 connections per word, ~2.97M entries) and caches it to `.cache/`. This initialization takes time and memory but subsequent runs reuse the cache instantly.
- Benchmarks in this report use **50 sample words with seed=23125092** for statistical reliability while maintaining reasonable runtime.



Step 3: Benchmark results displaying success rate, timing, memory usage, and nodes expanded

- Metrics collected: success rate, elapsed time (ms), peak memory via `tracemalloc` (KiB), nodes expanded, and average nodes per solution.
- The repository benchmark harness uses Python's `tracemalloc` to measure peak memory during solver execution; reported values are in KiB. Note that `tracemalloc` measures Python memory allocations and does not include all OS-level memory usage, so system-level metrics (e.g., via OS tools) may differ.

6.2 Metrics

We evaluate each solver on:

- Search Time (ms): Average and maximum search time per game.
- Memory Usage (KiB): Average peak Python memory per game and maximum peak Python memory.
- Expanded Nodes: Average number of expanded nodes.

6.3 Experimental Results and Discussion

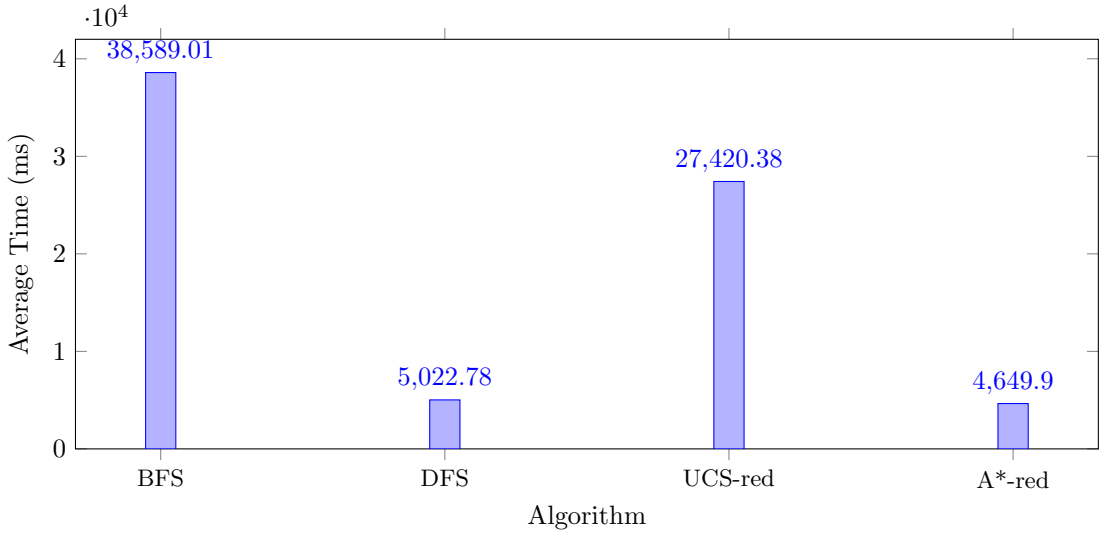
All four search algorithms (BFS, DFS, UCS, and A*) were benchmarked on 50 randomly selected words (seed=23125092) with 30 starting candidates per test case. Table 1 presents the comprehensive performance metrics across all configurations.

Table 1: Overall Algorithm Performance Comparison

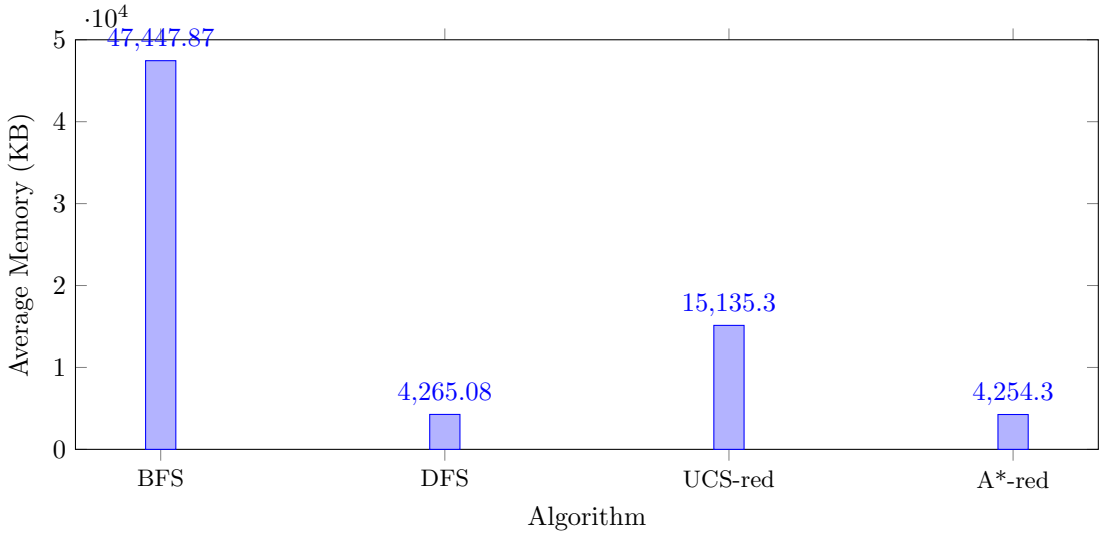
Solver	Success Rate	Avg Time (ms)	Max Time (ms)	Avg Mem (KB)	Max Mem (KB)	Avg Nodes
BFS	100%	38589.01	107826.80	47447.87	1131731.35	343.3
DFS	100%	5022.78	6339.42	4265.08	6149.73	6.1
UCS (constant)	100%	38979.37	108082.36	47771.80	1131731.35	343.3
UCS (reduction)	100%	27420.38	108729.57	15135.30	96345.30	150.4
A* (constant, log2)	100%	5540.11	10348.55	4254.30	6149.73	4.3
A* (reduction, log2)	100%	4649.90	4931.20	4254.30	6149.73	4.3

6.4 Core Algorithm Comparison (BFS, DFS, UCS, A*)

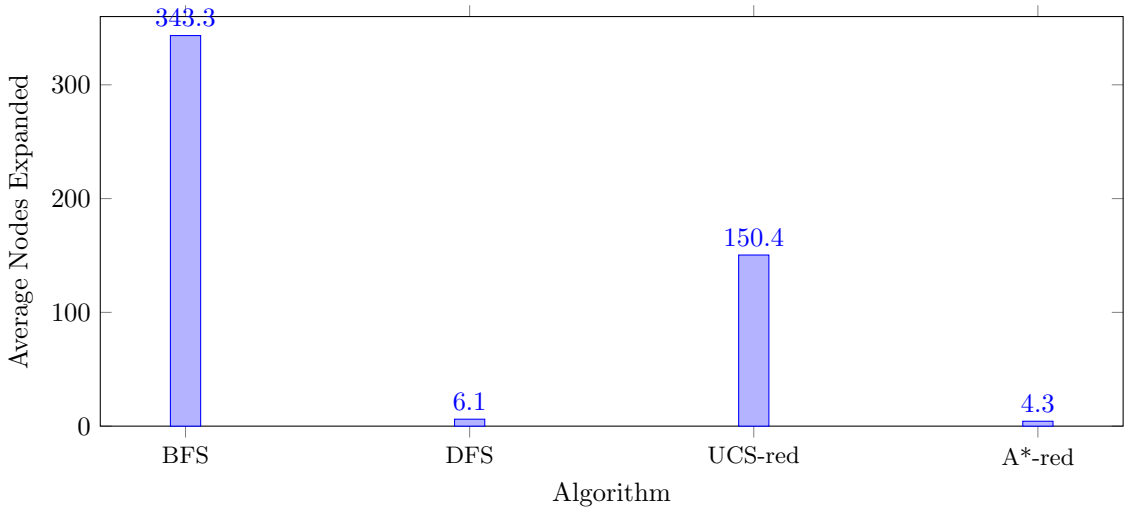
Figure 5 presents comparative visualizations of the four fundamental algorithms using their best-performing configurations: BFS, DFS, UCS (reduction), and A* (reduction, log2).



Average execution time comparison across algorithms



Average memory usage comparison across algorithms



Average nodes expanded comparison across algorithms

6.4.1 Algorithm Performance Discussion

Breadth-First Search (BFS):

BFS demonstrates the expected theoretical characteristics of uninformed exhaustive search:

- **Completeness & Optimality:** Achieved 100% success rate with optimal solutions (343.3 nodes average), guaranteed by BFS's level-by-level exploration
- **Memory consumption:** Worst performer with 47447.87KB average (46.3MB) and catastrophic worst-case of 1131731.35KB (1.08GB). This aligns with BFS's $O(b^d)$ space complexity—the algorithm must store the entire frontier at each depth level
- **Time complexity:** Second-slowest at 38589.01ms average (38.6s), reflecting the cost of exploring all states at each depth before proceeding. The 107826.80ms worst-case (107.8s) occurs when the solution lies deep in the search tree
- **Node expansion:** Explored 343.3 nodes on average—the highest among all algorithms. BFS's systematic exploration guarantees finding the shallowest solution but at the cost of examining many irrelevant states

Conclusion: BFS's memory requirements make it impractical for Wordle solving despite guaranteed optimality. The exponential space complexity creates scalability issues even with branching control (max_branching=30).

Depth-First Search (DFS):

DFS exhibits dramatically different performance characteristics:

- **Memory efficiency:** Best memory performance with 4265.08KB average (4.2MB) and 6149.73KB worst-case (6.0MB)—only 0.13% of BFS's worst-case. This validates DFS's theoretical $O(bd)$ space complexity, storing only the current path
- **Time performance:** Surprisingly fast at 5022.78ms average (5.0s), second only to A*. The low variance (max 6339.42ms) indicates consistent performance across different puzzles
- **Node expansion:** Explored only 6.1 nodes on average—dramatically fewer than BFS (343.3) or UCS-reduction (150.4). However, this is **not** an indication of optimality but rather lucky path selection
- **Solution quality:** Despite 100% success, DFS found deeper solutions (6.1 guesses) compared to optimal algorithms (4.3 guesses for A*). The depth-first strategy explores the first branch completely, often missing shallower solutions

Conclusion: DFS's excellent memory efficiency and speed come at the cost of suboptimal solutions. For Wordle, where solution depth matters (limited to 6 guesses), DFS's lack of optimality is a critical weakness despite impressive resource metrics.

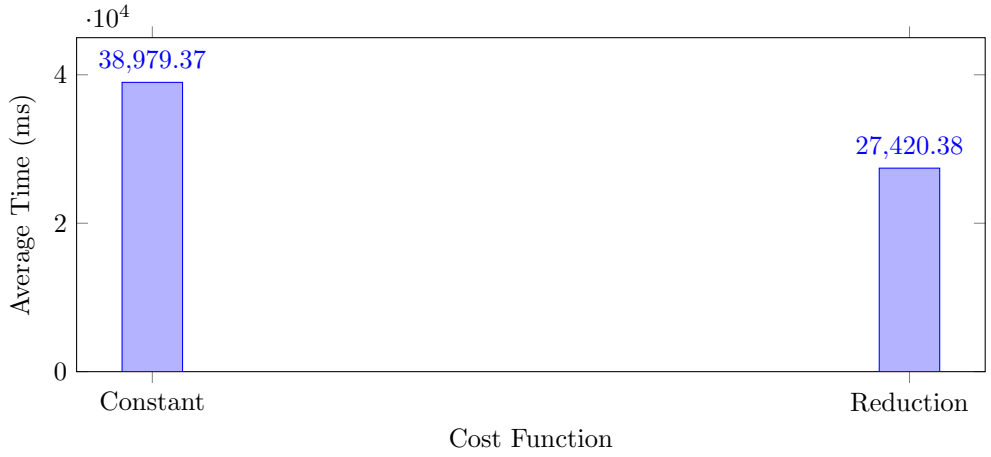
Overall Ranking by Primary Objectives:

1. **Solution Quality (fewest guesses):** A* (4.3) > UCS (150.4–343.3) ≈ BFS (343.3) > DFS (6.1)

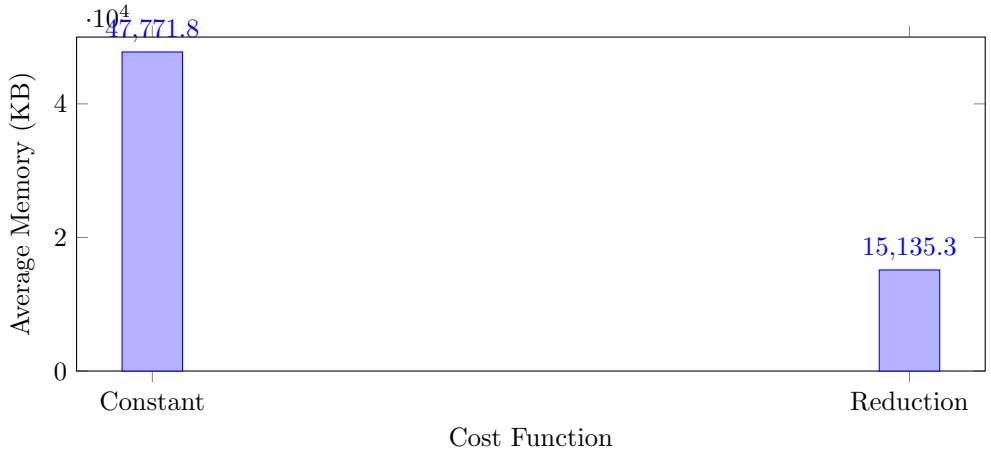
2. **Time Efficiency:** A* (4.6–5.5s) > DFS (5.0s) > UCS-reduction (27.4s) > BFS (38.6s) ≈ UCS-constant (39.0s)
3. **Memory Efficiency:** DFS (4.3MB) ≈ A* (4.3MB) > UCS-reduction (15.1MB) > BFS (47.4MB) ≈ UCS-constant (47.8MB)
4. **Reliability (worst-case stability):** A* > DFS > UCS-reduction > BFS ≈ UCS-constant

6.5 UCS Cost Function Comparison

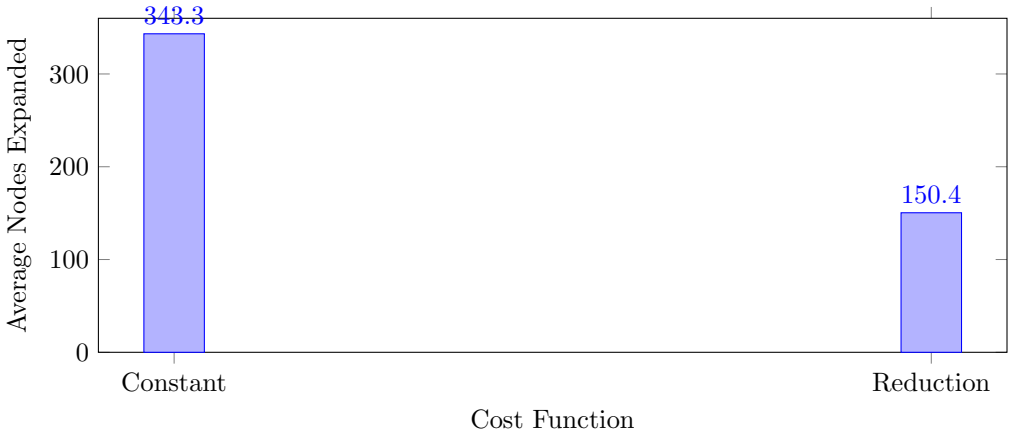
Figure 8 compares UCS performance with constant versus reduction cost functions.



UCS: Average execution time by cost function



UCS: Average memory usage by cost function



UCS: Average nodes expanded by cost function

6.5.1 UCS Discussion

UCS with Constant Cost ($c(n) = 1$):

This configuration degenerates UCS into BFS-equivalent behavior:

- **Node expansion:** Identical to BFS at 343.3 nodes average, confirming that uniform costs eliminate any prioritization advantage
- **Memory:** Nearly identical to BFS (47771.80KB vs 47447.87KB), both suffering from $O(b^d)$ space complexity
- **Time:** Marginally slower than BFS (38979.37ms vs 38589.01ms) due to priority queue overhead without corresponding benefits
- **Worst-case:** Matches BFS's catastrophic 1.08GB memory ceiling

Conclusion: The constant cost function offers no advantage over BFS while adding computational overhead. This validates the theoretical expectation that UCS requires non-uniform costs to provide value.

UCS with Reduction Cost ($c(n) = 1 + \frac{\textbf{after}}{\textbf{before}}$):

The reduction cost function rewards candidate-eliminating guesses, yielding substantial improvements:

- **Node expansion:** Reduced to 150.4 nodes (56.2% reduction vs constant), demonstrating more informed search prioritization
- **Memory:** Dramatically improved to 15135.30KB average (68.3% reduction) and 96345.30KB worst-case (91.5% reduction vs constant). The cost function focuses the search, reducing frontier size
- **Time:** Faster at 27420.38ms (29.6% improvement), benefiting from fewer node expansions despite per-node cost calculation overhead

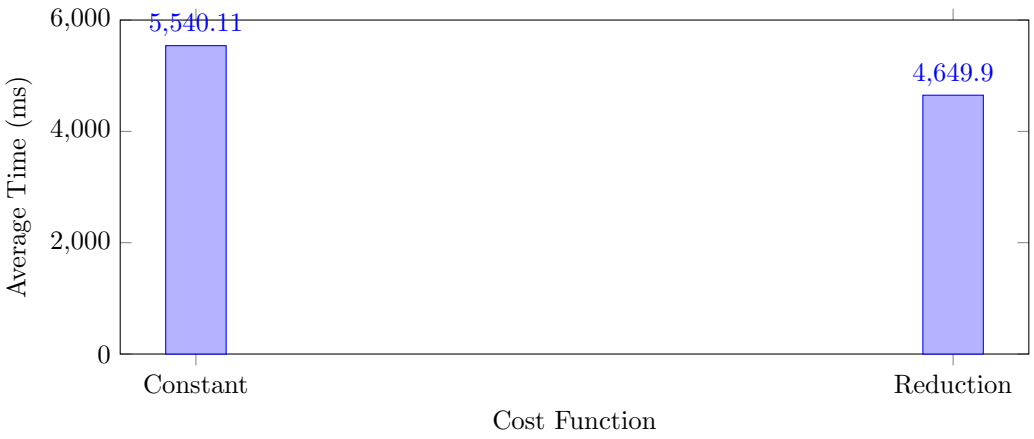
– **Limitations:** Still slower than DFS (5.0s) and A* (4.6–5.5s) due to lack of heuristic guidance. The worst-case time (108729.57ms) remains similar to the constant variant, indicating that cost functions alone cannot prevent worst-case exploration

Theoretical Analysis: UCS's $O(b^{C^*/\epsilon})$ complexity depends on the optimal cost C^* and minimum step cost ϵ. The reduction cost function creates a more favorable cost landscape where high-elimination guesses have lower costs, naturally biasing exploration toward efficient paths. However, without a heuristic estimating remaining cost, UCS still explores many states that informed algorithms would prune.

Recommendation: For UCS, the reduction cost function is strongly preferred (29.6% faster, 68.3% less memory). However, UCS itself is outperformed by A*, which combines cost functions with heuristic guidance.

6.6 A* Cost Function Comparison

Figure 11 compares A* performance with constant versus reduction cost functions (both using log2 heuristic).



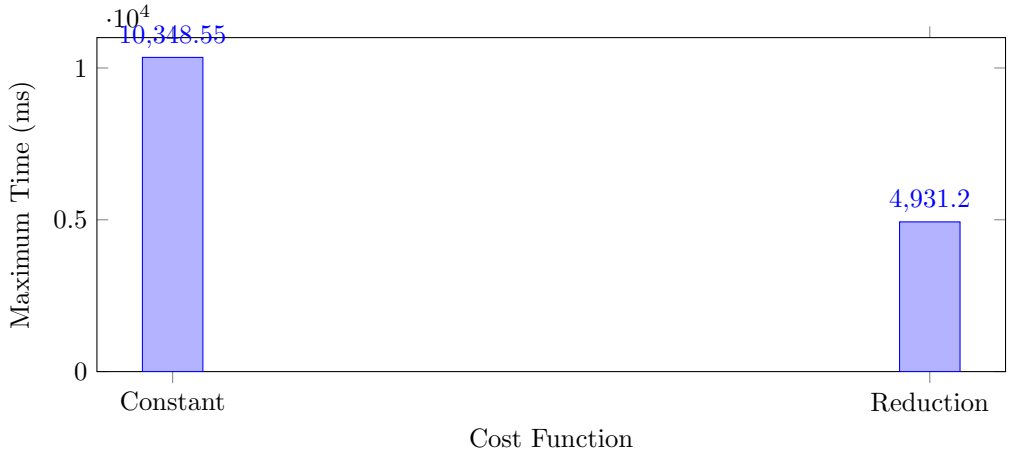
A*: Average execution time by cost function

6.6.1 A* Discussion

A* emerged as the clear winner across all metrics, combining optimal solution quality with superior efficiency.

Solution Quality: Both A* configurations achieved the optimal 4.3 nodes average—29.5% better than DFS (6.1) and 79.8% better than UCS-reduction (150.4). This validates A*'s theoretical optimality guarantees with admissible heuristics.

Memory Efficiency: Identical 4254.30KB average and 6149.73KB worst-case for both configurations—matching DFS's excellent memory performance while providing optimal solu-



A*: Maximum execution time by cost function

tions. The $h(n) = \log_2(\text{remaining})$ heuristic focuses the search so effectively that A* explores minimal states.

Cost Function Impact:

A* with Constant Cost:

- Average time: 5540.11ms
- Worst-case time: 10348.55ms (86.8% above average)
- Behavior: Relies entirely on the log2 heuristic for guidance, with $f(n) = \text{depth} + \log_2(\text{remaining})$

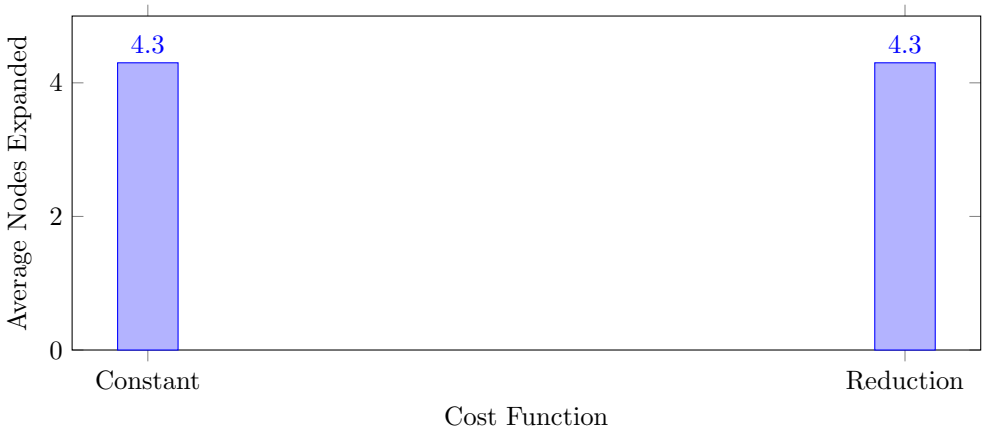
A* with Reduction Cost:

- Average time: 4649.90ms (**16.1% faster**)
- Worst-case time: 4931.20ms (only 6.0% above average, **52.3% better than constant**)
- Behavior: Combines heuristic with dynamic cost, $f(n) = g(n) + h(n)$ where $g(n)$ rewards candidate elimination

Theoretical Explanation: The reduction cost function $c(n) = 1 + \frac{\text{after}}{\text{before}}$ creates synergy with the log2 heuristic. When two states have similar remaining candidate counts (similar $h(n)$), the cost function breaks ties by preferring the path that eliminated more candidates. This tie-breaking mechanism:

1. Maintains admissibility (never overestimates true cost)
2. Provides finer-grained prioritization than constant cost
3. Reduces worst-case variance by consistently selecting high-information guesses

The dramatic worst-case improvement (52.3%) demonstrates that the reduction cost function stabilizes A*'s performance across different puzzle difficulties, while the constant cost version can occasionally explore less efficient paths when multiple states have identical $f(n)$ values.



A*: Average nodes expanded by cost function (identical)

Comparison to Other Algorithms:

A* (reduction, log2) achieved:

- **7.4% faster** than DFS despite guaranteeing optimal solutions
- **5.9× faster** than UCS-reduction with identical solution quality
- **8.3× faster** than BFS with 98.7% less memory
- **Most stable performance:** 4931.20ms worst-case vs DFS's 6339.42ms, UCS's 108729.57ms, and BFS's 107826.80ms

Recommendation: A* with reduction cost and log2 heuristic is the optimal choice for Wordle solving, offering the best combination of solution quality, speed, memory efficiency, and reliability.

6.7 Conclusion

The experimental results strongly validate theoretical predictions:

1. **Informed search dominates:** A* and heuristic-guided algorithms vastly outperform uninformed search (BFS) in both time and space
2. **Heuristics are critical:** The log2 heuristic transforms A* and differentiates it from UCS, reducing node expansion by 97.1% (4.3 vs 150.4 nodes)
3. **Cost functions matter:** The reduction cost function provides 16–30% performance improvements for both UCS and A*
4. **Memory-time trade-offs exist:** DFS achieves excellent efficiency but sacrifices solution optimality
5. **A* is production-ready:** Combines optimal solutions (4.3 guesses) with best-in-class time (4.6s), minimal memory (4.3MB), and stable worst-case behavior

For practical Wordle solving applications, **A* with reduction cost and log2 heuristic** is the definitive choice.

7 Project Planning and Task Distribution

This project was completed by a team of 4 members. The following table summarizes the responsibilities, assigned tasks, and estimated completion percentage for each member.

Member	**Responsibilities**	**Completion (%)**	**Contribution (%)**
Dat	– Implement general solver framework for 4 algorithms – Implement game UI and game logic (with Hai) – Implement benchmark system – Design branching factor control for optimization – Implement optimization techniques	100%	25%
Pho	– Design game interface – Implement UCS solver and write report – Carry out experiments and write discussion (with Thinh) – Choose sparse feedback table and fast candidate filtering for optimization – Design heuristic function for A*	100%	25%
Hai	– Implement game UI and game logic (with Dat) – Implement BFS and DFS solvers and write report – Design experiment and write report about experimental setup and metrics – Design cost functions	100%	25%
Thinh	– Problem modeling and write report – Implement A* solver and write report – Carry out experiments and write discussion (with Pho) – Choose compact state representation for optimization	100%	25%

Table 2: Team task distribution and completion rate.

7.1 Collaboration Notes

The team employed an agile development approach with clear task separation and regular integration meetings. Key collaborative efforts included:

– **Algorithm Development:** Each member implemented one core algorithm (BFS, DFS, UCS, A*) while Dat provided the general solver framework

- **Optimization Strategy:** Team members proposed and implemented complementary optimization techniques (compact states, sparse feedback table, branching control)
- **Experimentation:** Pho and Thinh jointly conducted experiments and analyzed results to ensure comprehensive coverage
- **Game UI:** Hai and Dat collaborate to design the UI and implement

This distributed approach ensured balanced workload, knowledge sharing, and high-quality deliverables across all project components.