

# Engineering Specification: Static Constant-Time Wordle Solver

Version: 1.0.0

Status: Approved for Implementation

Target Architecture: C++17 (Builder/Solver) + CUDA (Optional Acceleration)

---

## 1. System Overview

### 1.1 Objective

Construct a static artifact (binary file) containing a complete solution tree for the game of Wordle. The solver must guarantee a win within 6 guesses for every word in the standard 2,315-word solution set, starting with the fixed word **SALET**.

### 1.2 Core Constraints

1. **Runtime Complexity:**  $O(1)$  time per move (array lookup).
  2. **Max Depth:** Strictly  $\leq 6$  for all leaf nodes.
  3. **Correctness:** Completeness must be verified for all 2,315 solutions.
  4. **Hardware:** Build process uses CPU (AVX2/512) and/or GPU (CUDA). Runtime runs on any standard CPU.
- 

## 2. Mathematical Strategy

### 2.1 The "Remaining Guesses" ( $R$ ) Logic

The tree construction is guided by the number of guesses remaining, defined as  $R = 6 - \text{current\_depth}$ . The selection heuristic adapts based on  $R$  to strictly enforce the depth limit.

Depth	R	Heuristic Strategy	Constraints
0 (Root)	6	Max Entropy	Fixed start word: <b>SALET</b>
1	5	Max Entropy	None
2	4	Max Entropy	None
3	3	Hybrid	Penalize if max bucket size $> 5\$$ .
4	2	Minimax	<b>Hard Constraint:</b> Max bucket size must be $1\$$ . (Every remaining candidate must be uniquely identified by the next feedback).
5	1	Solve	Guess the only remaining word.

## 2.2 Algorithm: Iterative Deepening Beam Search

Since finding the optimal tree is NP-Complete, we use a satisfying approach. We guarantee correctness (finding a tree, not necessarily the *smallest*) via eventual exhaustive search.

**Procedure BuildNode(Candidates, Depth):**

1. **Check Cache:** If Candidates (normalized) exists in VisitedStates, return cached result.
2. **Base Case:** If  $|Candidates| = 1\$$ , return Leaf.
3. **Failure Case:** If  $Depth == 6\$$  and  $|Candidates| > 1\$$ , return FAILURE.
4. **Beam Expansion Loop:**
  - o Generate heuristics for all valid guesses.
  - o Iterate through Beam Widths  $K \in \{5, 50, \text{ALL}\}$ .
  - o **For each  $K$ :**
    - Select top  $K$  guesses.
    - **Parallel Attempt:** specific guesses are assigned to a thread pool.
    - First thread to build a valid subtree sets atomic Success flag.
    - Other threads abort work immediately.
  - o If success, return Node.
  - o If all  $K$  fail, expand to next  $K$ .
  - o If  $K=\text{ALL}$  fails, return FAILURE (Backtrack).

**Completeness Argument:** Because the final beam width is ALL, the algorithm performs a depth-limited DFS. If a solution tree of depth  $\leq 6$  exists rooted at SALET (which is mathematically proven), this algorithm will find it.

---

## 3. Architecture & Data Structures

### 3.1 State Representation (The "Bitset")

To optimize memory and cache locality, a game state (subset of solutions) is represented as a fixed-size bitset.

- **Definition:** std::bitset (or std::array<uint64\_t, 37>).
- **Mapping:** Index  $i$  corresponds to the  $i$ -th word in the sorted solutions.txt.
- **Memoization Key:** XXHash64 of the bitset data.

### 3.2 Global Memoization Map

- **Structure:** ConcurrentHashMap<uint64\_t, NodeOffset>
- **Purpose:** Prevents re-solving identical states (transpositions).
- **Memory Estimate:** 200,000 states  $\times$  (8 byte key + 4 byte val + overhead)  $\approx$  50-100 MB. Fits easily in RAM.

### 3.3 The Pattern Matrix (GPU/CPU Shared)

A precomputed matrix  $P$  of dimensions  $[12972 \times 2315]$  stored as uint8\_t.

- $P[g][s] =$  Pattern ID (0-242) for guess  $g$  against solution  $s$ .
- **GPU Residency:** If CUDA is enabled,  $P$  is loaded once into VRAM. It is **never** transferred back and forth.
- **Usage:**
  - **CPU (AVX):** Load 64 bytes (solutions) into ZMM registers, compute patterns, update histogram.
  - **GPU (CUDA):** Kernel receives a list of active solution indices (sparse). Threads iterate over guesses, performing gather-scatter to compute entropy.

---

## 4. Implementation Details (The Builder)

### 4.1 Hybrid Dispatcher

To balance overhead vs. throughput:

- **Tier 1 (Heavy):**  $|Candidates| > 1024$ . Dispatch entropy calculation to **GPU**.
- **Tier 2 (Medium):**  $64 < |Candidates| \leq 1024$ . Dispatch to **CPU (AVX-512/AVX2)** multithreaded.
- **Tier 3 (Light):**  $|Candidates| \leq 64$ . Scalar CPU execution (overhead of threading/GPU too high).

## 4.2 Module: libwordle\_core

- **WordList:** Loads solutions/guesses. Computes SHA-256 checksum of the list.
- **PatternTable:** Generates/Loads the  $[N_g \times N_s]$  matrix.
- **Feedback:** Canonical function calc\_pattern(guess, solution).

## 4.3 Module: builder

- **Thread Pool:** std::thread pool size = hardware\_concurrency.
- **Atomic Control:** std::atomic<bool> node\_solved passed to worker threads to trigger early exit.
- **Output:** Writes to solver\_data.bin.

---

# 5. Runtime Artifact & Solver

## 5.1 Binary Format (solver\_data.bin)

Little-endian, packed struct.

Offset	Type	Field	Description
<b>Header</b>			
0x00	u32	Magic	0x5752444C ("WRDL")
0x04	u32	Version	1
0x08	u64	ListChecksum	XXHash64 of sorted word list.
0x10	u32	NumNodes	Total nodes in tree.
0x14	u32	RootIndex	Index of root node

			(usually 0).
<b>Data</b>			
0x20	Array	Nodes	NumNodes instances of Node.
...	Array	Children	NumNodes * 243 instances of u32.

## 5.2 Node Struct

C++

```
struct Node {
    uint16_t guess_index; // Index into guess list
    uint16_t flags;     // Bit 0: IsLeaf, Bit 1: IsSolution
    // Note: Children offset is implicit: (node_index * 243)
    // The Children Array follows the Node Array in the file.
};
```

- **Children Lookup:** `next_node_idx = children_array[current_node_idx * 243 + pattern_id]`
- **Memory Footprint:** 200k nodes  $\times$  (4 bytes node + 243  $\times$  4 bytes children)  $\approx$  195 MB.

## 5.3 Solver Logic

1. **Load:** mmap the binary. Verify Magic and ListChecksum.
2. **Input:** User types 5 colors (e.g., G Y B B G).
3. **Parse:** Convert string to base-3 integer (0-242).
4. **Transition:** `current_node = children_table[current_node * 243 + pattern].`
5. **Output:** `word_list[nodes[current_node].guess_index].`

---

## 6. Verification & Quality Assurance

## 6.1 The --verify Flag

The builder must **not** produce the final binary until a verification pass succeeds.

- **Logic:**

1. Load the generated tree into memory.
2. Iterate  $\$s\$$  from  $\$0\$$  to  $\$2314\$$  (all solutions).
3. Simulate a game with  $\$s\$$  as the secret.
4. Generate feedback using libwordle\_core.
5. Traverse tree.
6. **Assert:**
  - Leaf reached in  $\leq 6$  steps.
  - Leaf word matches  $\$s\$$ .
  - Transition is valid (no "impossible" moves).

## 6.2 Unit Tests ("Nasty Cases")

Specifically test the builder against known traps during development:

- **\_IGHT Trap:** Solution NIGHT. Ensure solver does not waste guesses on RIGHT, MIGHT, SIGHT sequentially.
- **\_ATCH Trap:** Solution WATCH.
- **Assertion:** The solver must pick a "burner" word (e.g., FLOWN) to distinguish candidates when  $\$R\$$  is tight.

---

# 7. Execution Plan for Coding Agent

## Phase 1: Core Library & Environment

1. Setup CMake project with C++17.
2. Implement WordList loader (parse solutions.txt, guesses.txt).
3. Implement PatternTable generation (naive CPU first, then AVX optimized).
4. Implement BitSet state representation.

## Phase 2: The Builder (Logic)

1. Implement Entropy calculation (Scalar).
2. Implement RecursiveBuild with std::map memoization (no threading yet).
3. Validate small trees (e.g., 50-word dictionary).
4. Add CUDA kernel for entropy (optional, verify availability via CMake).
5. Implement ThreadPool and BeamSearch logic.

## Phase 3: Serialization & Verification

1. Implement binary writer.
2. Implement verification traversal.
3. Run full build on standard word lists.

## Phase 4: The Solver (Runtime)

1. Implement mmap loader.
2. Implement simple CLI loop.
3. Add Checksum validation.

## Phase 5: Optimization

1. Profile memory usage (ensure BitSets are efficient).
2. Tune Beam Width thresholds (\$K\$ values).
3. Verify GPU/CPU handoff thresholds (default 1024).

---

*(End of Specification)*