# Optimal Initial Settlement Placement in Catan: A Game-Theoretic Approach

15.C57 Optimization
Christenson, Lobon, Martino

## 1 Problem Statement

The goal of this project is to determine the optimal placement of the two initial settlements in Catan assuming that all players act optimally and symmetrically. In Catan, players place settlements in the order 1–2–3–4–4–3–2–1, meaning early moves constrain future options. The first player's choice shapes the entire game, since each subsequent placement reduces the feasible set for everyone else, including their own second move. This sequential dependency naturally forms a game tree, where each node represents a partial configuration of placed settlements and each branch corresponds to a response by the next player. Our goal is to identify an optimal root-to-leaf path that preserves optimal play for every player at each decision point. However, explicitly constructing the entire game tree quickly becomes computationally ineficient due to its exponential growth in the number of possible placements. We will focus on developing computational techniques to efficiently solve the game tree of optimal placements under the stated assumptions. In this simplified version, we will not consider roads, ports, monopoly strategies, or adversarial positioning aimed at harming other players. These additional strategic elements could be incorporated as future extensions to the model.

### 1.1 Quick Primer on Catan Setup

- **Board**: a fixed arrangement of 19 hex tiles. These are five rows organized like 3–4–5–4–3. (See Figure 5 for a board setup example). Each hex tile represents the resources brick, wood, ore, wheat, and sheep plus a desert. Each hex is labeled with a dice number from 2 to 12 (excluding 7) and produces its resource when that number is rolled.
- **Vertices (intersections)**: settlements can be placed on the 54 intersections formed by the hex grid. Each vertex touches up to three hexes; resource production for a settlement equals the sum of contributions from all adjacent hexes.
- **Dice probabilities**: probability of each number is $\text{Prob}(k) = \text{combinations}(k)/36$, e.g., $\text{Prob}(6) = \text{Prob}(8) = 5/36$, $\text{Prob}(4) = \text{Prob}(10) = 3/36$, etc. These probabilities translate into expected resource inflows.
- **Distance rule (feasibility)**: two settlements cannot be adjacent; at least two edges must separate any pair.
- **Receiving Cards**: A player receives a card when the dice number rolled is on an adjacent tile to one of their settlements.

## 2 Player objective function

We define the following composite score that every player is assumed to maximize optimally and symmetrically. Given the two vertices $S_p = \{s_1, s_2\}$ owned by player $p$, we compute three interpretable metrics and combine them with chosen weights $(\alpha, \beta, \gamma)$:

1. **Material diversity**: counts how many distinct resource types the two settlements can produce (maximum of five). This rewards coverage across brick, wood, ore, wheat, and sheep.

2. **Expected cards per turn**: sums all adjacent dice probabilities to estimate the expected number of resource cards the player receives on each roll.

3. **Chance of receiving at least one card**: aggregates the unique dice probabilities (ignoring repeated numbers) to capture how often any of the player's numbers are likely to hit, favoring a spread of dice values.

The objective used by the solver is $\text{Objective}_p = \alpha \cdot \text{Prod}(S_p) + \beta \cdot \text{Bal}(S_p) + \gamma \cdot \text{Scar}(S_p)$.

# 3 Backward Induction

This is a deterministic dynamic-programming problem that we solve via backward induction. Because the tree is shallow but extremely wide, we traverse it with a depth-first search. We provide pseudo code of our solve function below.

### Pseudocode

```
function Solve(board, player):
    if player > num_players:
        return board

    best_board = None
    best_objective = -inf

    for vertex in feasible_vertices(board):
        new_board = Clone(board)
        PlaceSettlement(new_board, player, vertex)

        new_board = Solve(new_board, player + 1)

        // Now place this player's second settlement optimally
        // (given their first settlement and all later players' placements)
        PlaceSettlement(new_board, player, best_second_vertex(vertex,board))

        objective_value = Objective(new_board, player)

        if objective_value > best_objective:
            best_objective = objective_value
            best_board = new_board

    return best_board
```

The function `Solve(board, player)` implements backward induction by recursively solving from the perspective of `player` who is about to place their first settlement. The key insight is that each player's optimal decision depends on how later players will respond.

For each feasible first position, the algorithm: (1) clones the board and places the first settlement, (2) recursively calls `Solve` for the next player, which eventually solves for all remaining players, and (3) after the recursion returns, places this player's second settlement optimally given the constraints imposed by all later players' optimal placements.

The recursion naturally implements the snake order $1 \to 2 \to 3 \to 4 \to 4 \to 3 \to 2 \to 1$.

# 4 Memoization

Many game states share identical configurations of occupied vertices and available feasible options, even if reached through different placement sequences. For each player $p$ and a given set of feasible vertices (available options), the optimal resulting board configuration is the same regardless of how we arrived at that state. By caching the mapping $(player, available\_vertices) \rightarrow best\_resulting\_board$, we avoid recomputing entire subtrees when the same configuration is encountered again.

The memoization key combines the current player and the set of available vertices. This allows us to recognize equivalent states even when reached via different paths through the game tree. Recall our tree is extremely wide, so we expect frequent repeats. When we recognize a memoization, we can immediately return the cached optimal board configuration without any recursive exploration.

### Pseudocode

```
memo = {}  // Maps (player, available_vertices) → best_board

function Solve(board, player):
    if player > num_players:
        return board

    memo_key = (player, available_vertices)

    if memo_key in memo:
        return memo[memo_key]

    best_board = None
    best_objective = -inf

    for vertex in feasible_vertices(board, player):
        new_board = Clone(board)
        PlaceSettlement(new_board, player, vertex)

        new_board = Solve(new_board, player + 1)

        if new_board is None:
            continue

        PlaceSettlement(new_board, player, best_second_vertex(vertex, new_board))

        objective_value = Objective(new_board, player)

        if objective_value > best_objective:
            best_objective = objective_value
            best_board = new_board

    memo[memo_key] = best_board
    return best_board
```

## 5 Pruning Strategies

- **Lower bound (LB)**: At each recursive node, we maintain a local `best_objective` value representing the best complete solution found so far for the current player at this node. This acts as a lower bound: any branch that cannot exceed this value is pruned. The LB is updated dynamically as we explore branches and discover better solutions.
- **Upper bound (UB)**: Before making a recursive call for a candidate first settlement position, we compute an upper bound on the maximum objective value achievable from that branch. The UB assumes an optimistic relaxation: given the first settlement at position $v_1$, we calculate the maximum pair_quality$(v_1, v_2)$ over all currently feasible second positions $v_2$, ignoring that future players may take some of these positions. If $UB \leq LB$, we prune the entire branch without recursion, since even in the best-case scenario (no interference from other players), this branch cannot improve upon the current best solution. To make UB computation extremely efficient, we precompute a pair_quality matrix of size $n \times n$ (where $n$ is the number of vertices) at initialization, storing the objective function value for every possible vertex pair $(v_1, v_2)$. This precomputation takes $\mathcal{O}(n^2)$ time once, making each UB query during search a simple lookup: for a given first vertex $v_1$, the UB is just the maximum value in row $v_1$ of this precomputed matrix over all currently feasible second vertices.
- **Move ordering**: We sort candidate first positions by their individual vertex quality (single-settlement objective) in descending order before exploration. This ensures we explore the most promising branches first, which helps establish a strong LB early in the search. A strong LB enables more aggressive pruning of remaining branches.

### Pseudocode

```
memo = {}

function Solve(board, player):
    if player > num_players:
        return board

    memo_key = (player, GetAvailableVertices(board))

    if memo_key in memo:
        return memo[memo_key]

    best_board = None
    best_objective = -inf  // Local lower bound

    for vertex in sorted(feasible_vertices(board, player)):

        ub = UpperBound(board, player, vertex)
        if ub <= best_objective:
            continue  // Prune this branch

        new_board = Clone(board)
        PlaceSettlement(new_board, player, first_vertex)

        new_board = Solve(new_board, player + 1)

        if new_board is None:
```

```
            continue

        PlaceSettlement(new_board, player, BestSecondVertex(vertex, new_board))

        objective_value = Objective(new_board, player)

        if objective_value > best_objective:
            best_objective = objective_value
            best_board = new_board

    memo[memo_key] = best_board
    return best_board
```

Memoization avoids redundant computation of identical subtrees, while pruning eliminates hopeless branches before expensive recursive exploration. Move ordering amplifies both effects by finding strong solutions early, which tightens LBs and enables more aggressive pruning.

## 6 Experimental Setup

To quantify the performance benefits of each optimization technique, we conducted a comprehensive experimental evaluation. We generated 30 random Catan boards and evaluated each board using three solver modalities:

1. **Feasibility Pruning Only**: Baseline DFS that only filters out infeasible moves (violations of distance rule and occupancy constraints). This serves as our reference baseline.
2. **Feasibility + Memoization**: Adds memoization to cache results for identical game states, avoiding redundant subtree exploration.
3. **All Prunings (Feasibility + Upper Bound + Memoization)**: The complete solver with all optimizations enabled, including upper bound pruning and move ordering.

For each board and modality, we measured:

- **Execution time**: Total wall-clock time to find the optimal solution
- **Recursive calls**: Total number of DFS recursive invocations, which directly reflects the size of the search space explored
- **Solution verification**: We compared the optimal solutions found by all three modalities to ensure correctness. All modalities must agree on the optimal placement. This confirms that pruning does not eliminate optimal branches.

This experimental design allows us to isolate and quantify the contribution of each optimization technique, demonstrating both the correctness (all modalities find identical optimal solutions) and the dramatic performance improvements achieved through memoization and pruning.

## 7 Results and Benchmarks

The experimental results on 30 random boards demonstrate the substantial performance gains achieved by each optimization technique:

### Performance Metrics

| Modality | Avg. Time | Min Time | Max Time | Avg. Calls | Speedup |
|---|---|---|---|---|---|
| Feasibility Pruning Only | 444.16 s | 441.64 s | 454.26 s | 5,634,937 | 1.00× |
| Feasibility + Memoization | 79.20 s | 78.61 s | 80.28 s | 983,839 | 5.61× |
| All Prunings (Full Solver) | 0.33 s | 0.05 s | 0.69 s | 569 | 1,361.88× |

### Key Findings

- **Memoization impact**: Adding memoization reduces execution time by a factor of 5.61 (from 444s to 79s) and recursive calls by 82.5% (from 5.6M to 984K). This demonstrates that many game states are reached through multiple paths, and caching these results eliminates massive redundant computation.
- **Upper bound pruning impact**: The complete solver with all optimizations achieves a **1,361× speedup** over the baseline, reducing average execution time from 444 seconds to just 0.33 seconds. Recursive calls drop by 99.99% (from 5.6M to 569 on average), showing that upper bound pruning eliminates the vast majority of hopeless branches before expensive recursive exploration.
- **Solution correctness**: All three modalities found identical optimal solutions for all 30 boards, confirming that neither memoization nor pruning eliminates optimal branches. This validates the correctness of our optimization techniques.
- **Consistency**: The baseline modality shows remarkably consistent performance across all boards (5,634,937 recursive calls for every board), indicating that the search space size is largely independent of board configuration when only feasibility pruning is used. In contrast, the full solver shows variability (65 to 1,195 recursive calls), reflecting how different board configurations affect the effectiveness of upper bound pruning.

## 8 Possible Extensions

- **Competitiveness-aware objectives**: Instead of maximizing their own score $v_p$, each player could maximize $v_p - \max_{q \neq p} v_q$ (their advantage over the best opponent). This is more realistic because players care about winning, not just absolute value—they're motivated to both maximize their own benefit and limit opponents' advantage. This increases tree depth: player 4's second move now affects how players 1–3 respond in their second placements (since they care about relative performance), creating order $1 \to 2 \to 3 \to 4 \to 4 \to 3 \to 2 \to 1$. The UB/LB pruning extends naturally: $UB_{\text{competitive}}(p) = UB_{\text{self}}(p) - LB_{\text{maxOthers}}$.
- **Stochastic opponents**: Instead of assuming all players act optimally, we can model realistic error behavior for players 2–4 (e.g., using a logit choice model where players are more likely to choose better moves but can still make mistakes). Player 1 then solves a stochastic dynamic program, maximizing expected utility while integrating over the probability distribution of other players' moves. Real players make suboptimal decisions, and player 1's optimal strategy should account for these uncertainties.
- **New objective functions**: incorporate monopoly bonuses ("maximize wheat dominance"), port leverage, positioning to obtain longest road, specific resource placement to obtain more development cards or build cities, or robber resilience.
- **Full game features**: integrate roads, ports, or road-building constraints, expanding the state but still capturable by the memoized DFS with carefully designed feasibility checks.

# 9 Conclusions

- The solver demonstrates how backward induction, memoization, and pruning convert an explosive game tree into a tractable optimization pipeline, quantifiably outperforming naive DFS at each incremental enhancement.
- Because each acceleration is modular (caching layer, UB/LB estimators, move ordering), the framework adapts quickly to richer rulesets—making it a strong foundation for increasingly realistic Catan simulators.
- The methodology generalizes to sequential competitive placement problems where each decision constrains future feasibility. It showcases the tight integration of game theory, optimization, and algorithmic design to compute exact equilibria efficiently.

# Appendix

## A.1 Example Optimal Placement Visualization

The following visualization shows an example of optimal settlement placement computed by the solver. This placement was generated with the following objective function weights:

- **Resource diversity weight** ($\alpha$): 0.500
- **Expected cards per turn weight** ($\beta$): 0.300
- **Probability of at least one card weight** ($\gamma$): 0.200

These weights emphasize material diversity (50% weight) while still valuing expected resource production (30%) and dice number spread (20%).

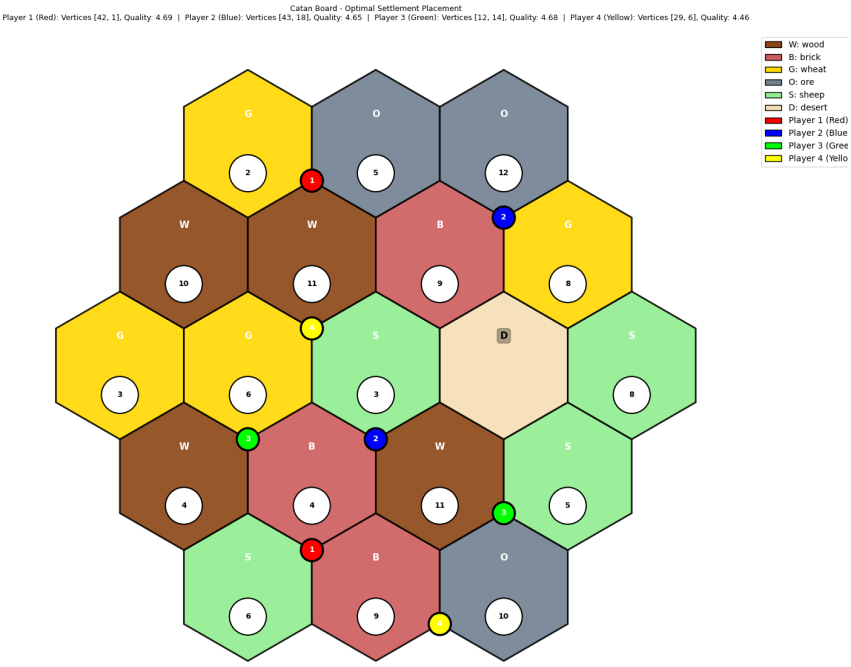


Figure 1: A solved Catan board with optimal placements for each player

## A.2 Detailed Board-by-Board Results

The following tables report the detailed per-board performance for each pruning modality.

### Feasibility Pruning Only

| Board | Time (s) | Positions (v1,v2) | Obj. value | Recursive calls |
|---|---|---|---|---|
| 0 | 442.0174 | (42, 31) | 6.7667 | 5,634,937 |
| 1 | 445.5643 | (20, 1) | 6.7917 | 5,634,937 |
| 2 | 445.0003 | (24, 41) | 6.7778 | 5,634,937 |
| 3 | 445.4573 | ( 2, 21) | 6.7361 | 5,634,937 |
| 4 | 444.2690 | (33, 7) | 6.7917 | 5,634,937 |
| 5 | 442.6132 | (25, 1) | 6.7778 | 5,634,937 |
| 6 | 441.6423 | (29, 7) | 6.7472 | 5,634,937 |
| 7 | 444.9940 | (30, 44) | 6.7639 | 5,634,937 |

| | | | | |
|---|---|---|---|---|
| 8 | 444.3487 | (41, 43) | 6.6667 | 5,634,937 |
| 9 | 443.4941 | (19, 33) | 6.7667 | 5,634,937 |
| 10 | 444.5225 | (29, 33) | 6.8194 | 5,634,937 |
| 11 | 454.2635 | ( 2, 41) | 6.7778 | 5,634,937 |
| 12 | 445.4887 | (44, 33) | 6.7639 | 5,634,937 |
| 13 | 443.1874 | (32, 12) | 6.7500 | 5,634,937 |
| 14 | 444.2226 | (34, 1) | 6.7556 | 5,634,937 |
| 15 | 443.5160 | (15, 42) | 6.7778 | 5,634,937 |
| 16 | 444.5709 | (15, 21) | 6.7778 | 5,634,937 |
| 17 | 443.6889 | (21, 44) | 6.7917 | 5,634,937 |
| 18 | 443.3624 | (12, 42) | 6.7917 | 5,634,937 |
| 19 | 442.9552 | (30, 31) | 6.7639 | 5,634,937 |
| 20 | 443.8549 | (33, 29) | 6.7917 | 5,634,937 |
| 21 | 442.7369 | (12, 32) | 6.7528 | 5,634,937 |
| 22 | 443.2766 | ( 1, 25) | 6.7778 | 5,634,937 |
| 23 | 443.3843 | ( 7, 24) | 6.8111 | 5,634,937 |
| 24 | 444.0095 | ( 1, 41) | 6.7889 | 5,634,937 |
| 25 | 442.8969 | ( 1, 29) | 6.7778 | 5,634,937 |
| 26 | 443.8668 | (12, 14) | 6.7528 | 5,634,937 |
| 27 | 444.6593 | (43, 19) | 6.8056 | 5,634,937 |
| 28 | 442.7565 | (41, 15) | 6.7972 | 5,634,937 |
| 29 | 444.2616 | (38, 43) | 6.7917 | 5,634,937 |

**Feasibility + Memoization**

| Board | Time (s) | Positions (v1,v2) | Obj. value | Recursive calls |
|---|---|---|---|---|
| 0 | 79.0538 | (42, 31) | 6.7667 | 983,839 |
| 1 | 79.3012 | (20, 1) | 6.7917 | 983,839 |
| 2 | 79.2471 | (24, 41) | 6.7778 | 983,839 |
| 3 | 80.2107 | ( 2, 21) | 6.7361 | 983,839 |
| 4 | 79.4050 | (33, 7) | 6.7917 | 983,839 |
| 5 | 79.3719 | (25, 1) | 6.7778 | 983,839 |
| 6 | 79.2980 | (29, 7) | 6.7472 | 983,839 |
| 7 | 79.0294 | (30, 44) | 6.7639 | 983,839 |
| 8 | 80.2757 | (41, 43) | 6.6667 | 983,839 |
| 9 | 79.1570 | (19, 33) | 6.7667 | 983,839 |
| 10 | 79.1125 | (29, 33) | 6.8194 | 983,839 |
| 11 | 78.8134 | ( 2, 41) | 6.7778 | 983,839 |
| 12 | 79.6239 | (44, 33) | 6.7639 | 983,839 |
| 13 | 79.1707 | (32, 12) | 6.7500 | 983,839 |
| 14 | 78.7598 | (34, 1) | 6.7556 | 983,839 |
| 15 | 79.1261 | (15, 42) | 6.7778 | 983,839 |
| 16 | 78.8883 | (15, 21) | 6.7778 | 983,839 |
| 17 | 80.0107 | (21, 44) | 6.7917 | 983,839 |
| 18 | 78.6079 | (12, 42) | 6.7917 | 983,839 |
| 19 | 78.9985 | (30, 31) | 6.7639 | 983,839 |
| 20 | 78.7038 | (33, 29) | 6.7917 | 983,839 |
| 21 | 79.9333 | (12, 32) | 6.7528 | 983,839 |
| 22 | 78.6525 | ( 1, 25) | 6.7778 | 983,839 |
| 23 | 79.1416 | ( 7, 24) | 6.8111 | 983,839 |
| 24 | 78.7174 | ( 1, 41) | 6.7889 | 983,839 |
| 25 | 78.9081 | ( 1, 29) | 6.7778 | 983,839 |

| 26 | 79.6396 | (12, 14) | 6.7528 | 983,839 |
|---|---|---|---|---|
| 27 | 78.9915 | (43, 19) | 6.8056 | 983,839 |
| 28 | 78.7992 | (41, 15) | 6.7972 | 983,839 |
| 29 | 79.0307 | (38, 43) | 6.7917 | 983,839 |

**All Prunings (Feasibility + Upper Bound + Memo)**

| Board | Time (s) | Positions (v1,v2) | Obj. value | Recursive calls |
|---|---|---|---|---|
| 0 | 0.4945 | (42, 31) | 6.7667 | 808 |
| 1 | 0.1670 | (20, 1) | 6.7917 | 307 |
| 2 | 0.1578 | (24, 41) | 6.7778 | 263 |
| 3 | 0.6914 | ( 2, 21) | 6.7361 | 1,195 |
| 4 | 0.1095 | (33, 7) | 6.7917 | 206 |
| 5 | 0.2161 | (25, 1) | 6.7778 | 366 |
| 6 | 0.2815 | (29, 7) | 6.7472 | 463 |
| 7 | 0.5058 | (30, 44) | 6.7639 | 1,033 |
| 8 | 0.4778 | (41, 43) | 6.6667 | 786 |
| 9 | 0.4428 | (19, 33) | 6.7667 | 805 |
| 10 | 0.2294 | (29, 33) | 6.8194 | 394 |
| 11 | 0.3785 | ( 2, 41) | 6.7778 | 608 |
| 12 | 0.3272 | (44, 33) | 6.7639 | 623 |
| 13 | 0.2454 | (32, 12) | 6.7500 | 404 |
| 14 | 0.3412 | (34, 1) | 6.7556 | 554 |
| 15 | 0.1936 | (15, 42) | 6.7778 | 349 |
| 16 | 0.4707 | (15, 21) | 6.7778 | 836 |
| 17 | 0.2948 | (21, 44) | 6.7917 | 487 |
| 18 | 0.2776 | (12, 42) | 6.7917 | 487 |
| 19 | 0.3105 | (30, 31) | 6.7639 | 529 |
| 20 | 0.2449 | (33, 29) | 6.7917 | 435 |
| 21 | 0.4322 | (12, 32) | 6.7528 | 767 |
| 22 | 0.3469 | ( 1, 25) | 6.7778 | 604 |
| 23 | 0.1327 | ( 7, 24) | 6.8111 | 232 |
| 24 | 0.3594 | ( 1, 41) | 6.7889 | 555 |
| 25 | 0.0454 | ( 1, 29) | 6.7778 | 65 |
| 26 | 0.3619 | (12, 14) | 6.7528 | 671 |
| 27 | 0.4358 | (43, 19) | 6.8056 | 726 |
| 28 | 0.2693 | (41, 15) | 6.7972 | 420 |
| 29 | 0.5425 | (38, 43) | 6.7917 | 1,090 |

## A.3 Altering Weights and the Corresponding Optimal Boards

Given the same board, we alter the weights of our reward function to have an intuition and understanding that our solver is placing settlements logically.

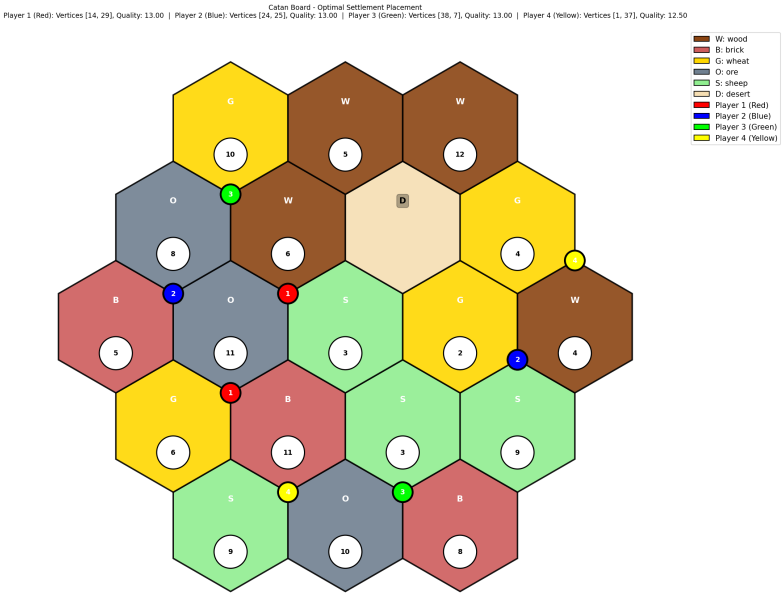


Figure 2: A solved Catan board with optimal placements for each player given weight = 1 for resource diversity

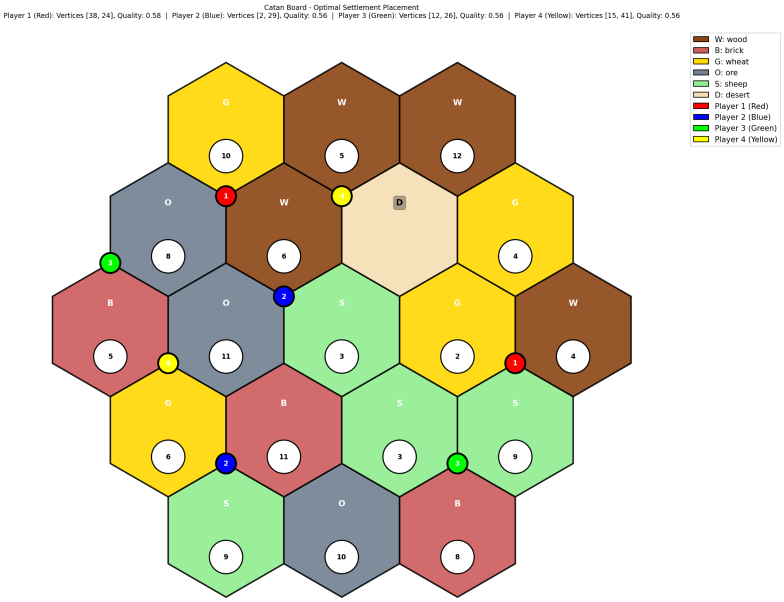


Figure 3: A solved Catan board with optimal placements for each player given weight = 1 for Expected number of cards per turn (value high probability numbers)

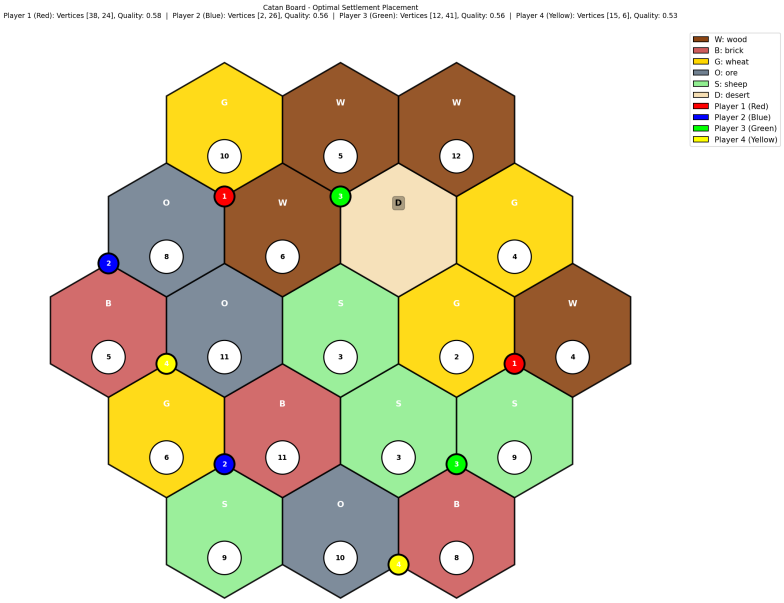


Figure 4: A solved Catan board with optimal placements for each player given weight = 1 for Expectation to obtain one card in a turn (number diversity)

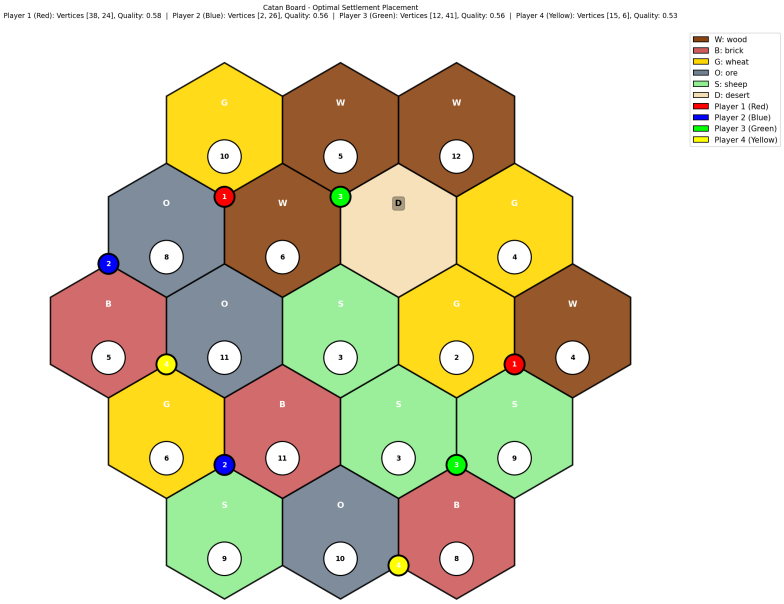


Figure 5: A solved Catan board with optimal placements for each player given weight = 0.5 for material diversity, 0.3 for Expected number of cards per turn (high probability numbers), and 0.2 for Expectation to obtain one card in a turn (number diversity)

## A.4 Code Repository

All code used for data generation, solver implementation, and visualization is available at the following GitHub repository:

`https://github.com/ericchristenson1/CatanPlacement`