David’s Dynamic Graph - Solution Development Conversations

Overview

This document tracks the development of various solution approaches for the David’s Dynamic Graph problem, simulating conversations with different AI assistants to explore multiple algorithmic strategies.

Conversation 1: Multi-Phase Strategy Development

Human: How should I approach this multi-phase interactive graph optimization problem?

Assistant: For David’s Dynamic Graph problem, I recommend a three-phase strategy:

Phase 1: Discovery (Efficient Structure Learning)

- Use STRUCTURE queries strategically on high-degree vertices first
- Build adjacency representation incrementally
- Estimate graph properties (density, connectivity)
- Budget: ~n queries (cost n)

Phase 2: Adversarial (Prediction and Adaptation)

- Use ADVERSARY_PREDICT sparingly but strategically
- Track query patterns that trigger adversarial moves
- Maintain multiple graph state hypotheses
- Adapt query strategy based on hints
- Budget: ~n/2 queries (cost varies)

Phase 3: Optimization (Multi-Objective Solving)

- Solve k-coloring using COLOR_CHECK with binary search
- Find dominating set using DOMINATION_WEIGHT with greedy approach
- Optimize resource allocation with dynamic programming
- Budget: remaining queries

Human: What about the multi-objective optimization aspect?

Assistant: The multi-objective nature requires careful balancing:

1. **Weighted Scoring Approach**: Convert to single objective
   - Score = 1000 - 50*(k-χ) - 10*W_dom - 5*R_waste - 100*P_violations
   - Prioritize avoiding violations (highest penalty)
   - Balance coloring efficiency vs domination weight

2. **Sequential Optimization**:
   - First ensure feasibility (no violations)
   - Then minimize k (coloring)
   - Then minimize domination weight
   - Finally optimize resource allocation

3. **Pareto Frontier**: For advanced approach
    - Generate multiple solutions
    - Select best compromise solution

Conversation 2: Game Theory and Adversarial Strategy

Human: How do I handle the adversarial component effectively?

Assistant: The adversarial phase requires game-theoretic thinking:

Adversary Behavior Modeling

```
class AdversaryPredictor {
    map<string, int> query_patterns;
    vector<string> hints_received;

    string predict_next_move(int round, vector<int> recent_queries) {
        // Analyze patterns in query history
        // Cross-reference with previous hints
        // Predict likely edge modifications
    }
};
```

Counter-Strategies

1. **Diversification**: Avoid predictable query patterns
2. **Deception**: Make queries that mislead adversary about your strategy
3. **Robustness**: Build solutions that work under multiple graph states
4. **Information Theory**: Maximize information gain per query

Hint Utilization

- Parse ADVERSARY_PREDICT responses carefully
- Build decision trees based on conditional probabilities
- Use hints to guide query selection in subsequent rounds

Conversation 3: Dynamic Programming Approaches

Human: Can dynamic programming help with the resource allocation subproblem?

Assistant: Absolutely! DP is perfect for the resource allocation component:

Resource Allocation DP

```
// dp[i][r] = maximum utility using first i vertices with r resources
vector<vector<int>> dp(n+1, vector<int>(R+1, 0));

for (int i = 1; i <= n; i++) {
    for (int r = 0; r <= R; r++) {
        // Don't allocate to vertex i-1
        dp[i][r] = dp[i-1][r];

        // Try allocating k resources to vertex i-1
        for (int k = 1; k <= min(r, capacity[i-1]); k++) {
            int utility = calculate_utility(i-1, k);
            dp[i][r] = max(dp[i][r], dp[i-1][r-k] + utility);
        }
    }
}
```

Utility Function Design

Consider multiple factors:
- Vertex weight (inverse relationship)
- Capacity utilization
- Graph centrality measures
- Domination contribution

DP for Dominating Set (Small Graphs)

```
// For n ≤ 15, use bitmask DP
// dp[mask] = minimum weight to dominate vertices in mask
vector<int> dp(1 << n, INF);
dp[0] = 0;

for (int mask = 0; mask < (1 << n); mask++) {
    for (int v = 0; v < n; v++) {
        if (!(mask & (1 << v))) {
            int new_mask = mask | (1 << v);
            // Add all vertices dominated by v
            for (int u : adj[v]) new_mask |= (1 << u);
            dp[new_mask] = min(dp[new_mask], dp[mask] + weight[v]);
        }
    }
}
```

Conversation 4: Approximation Algorithms

Human: The problem has NP-hard components. What approximation algorithms should I use?

Assistant: For the NP-hard subproblems, use these approximation strategies:

k-Coloring Approximation

1. **Greedy Coloring**: O(Δ+1) approximation where Δ is max degree
2. **Welsh-Powell**: Sort by degree, color greedily
3. **DSATUR**: Dynamic saturation-based coloring

Dominating Set Approximation

1. **Greedy Algorithm**: ln(Δ)+1 approximation

```
vector<int> greedy_dominating_set() {
    vector<bool> dominated(n, false);
    vector<int> domset;

    while (true) {
        int best_vertex = -1;
        int max_new_dominated = 0;

        for (int v = 0; v < n; v++) {
            if (find(domset.begin(), domset.end(), v) != domset.end()) continue;

            int new_dominated = 0;
            if (!dominated[v]) new_dominated++;
            for (int u : adj[v]) {
                if (!dominated[u]) new_dominated++;
            }

            if (new_dominated > max_new_dominated) {
                max_new_dominated = new_dominated;
                best_vertex = v;
            }
        }

        if (max_new_dominated == 0) break;

        domset.push_back(best_vertex);
        dominated[best_vertex] = true;
        for (int u : adj[best_vertex]) dominated[u] = true;
    }

    return domset;
}
```

Multi-Objective Approximation

1. **ε-constraint method**: Fix all but one objective
2. **Weighted sum**: Convert to single objective
3. **NSGA-II**: For population-based approaches (if time permits)

Conversation 5: Performance Optimization

Human: How can I optimize the performance of my solution?

Assistant: Several optimization strategies:

Query Efficiency

1. **Batch Information**: Maximize info per query
2. **Caching**: Store all received information
3. **Inference**: Deduce information without querying

Algorithmic Optimizations

```
// Fast adjacency checking
vector<vector<bool>> adj_matrix(n, vector<bool>(n, false));

// Efficient set operations for domination checking
bitset<MAX_N> dominated;
bitset<MAX_N> dominating_set;

// Memoization for repeated subproblems
map<pair<int, vector<int>>, int> memo;
```

Memory Management

- Use bitsets for large sets
- Compress graph representation
- Lazy evaluation of expensive computations

Time Complexity Analysis

- Discovery: O(n) queries, $O(n^2)$ processing
- Adversarial: O(n) queries, O(n log n) prediction
- Optimization: O(2^n) for small n, $O(n^3)$ approximation for large n

Solution Implementation Status

Completed Solutions

1. **solution_correct.cpp**: Multi-phase strategy with game theory
2. **solution_greedy.cpp**: Fast approximation algorithms
3. **solution_dp.cpp**: Dynamic programming for small instances

Performance Comparison

- Correct solution: High accuracy, moderate speed
- Greedy solution: Fast execution, good approximation
- DP solution: Optimal for small graphs, exponential scaling

Future Improvements

1. Hybrid approaches combining multiple strategies
2. Machine learning for adversary prediction
3. Advanced multi-objective optimization techniques
4. Parallel processing for independent subproblems