Integration Guide

Combining Direct QPU and Decomposition-Based Scenarios
Unified Benchmarking Framework

OQI-UC002-DWave Project

December 11, 2025

Abstract

This guide provides a comprehensive framework for integrating small-scale scenarios (suitable for direct QPU embedding) with large-scale scenarios (requiring decomposition strategies) within a unified benchmarking pipeline. We present the architectural design, implementation strategy, and best practices for handling heterogeneous problem scales (6-900 variables) across different formulations (portfolio, graph MWIS, single-period, multi-period rotation) using appropriate solving strategies (direct QPU, clique embedding, spatial-temporal decomposition).

Contents

1 Problem Space Overview

1.1 Scale Categories

Based on our benchmarking results, we identify three distinct scale categories:

Category	Variables	QPU Strategy	Embedding	Use Case
Micro	6-30	Direct QPU	Standard	Alternative formulations
Small	30-100	Clique / Direct	Clique-aware	Rotation (5 farms)
Medium	100-300	Decomposition	Zero overhead	Rotation (10-15 farms)
Large	300-900	Decomposition	Zero overhead	Rotation (20-50 farms)

Table 1: Problem scale categories and appropriate solving strategies

1.2 Formulation Types

Formulation	Variables	Structure	Classical Difficulty
Portfolio Selection	27	Sparse, synergy	Easy (instant)
Graph MWIS	30	Graph topology	Easy (instant)
Single Period	30-150	Assignment	Easy-Moderate
Penalty Rotation	90-900	Frustrated, dense	Hard (timeout)

Table 2: Formulation characteristics

2 Architectural Design

2.1 Unified Scenario Format

2.1.1 Common Data Structure

All scenarios, regardless of scale or formulation, should provide:

```
{
    'scenario_name': str,
    'formulation_type': str,  # 'portfolio', 'mwis', 'single_period', '
       rotation'
    'n_variables': int,
    'scale_category': str,  # 'micro', 'small', 'medium', 'large'
    'recommended_strategy': str,  # 'direct', 'clique', 'decomposition'

    # Problem-specific data
    'farms': {...},
    'crops': {...},
    'benefits': {...},
    'constraints': {...},

    # Metadata
    'description': str,
    'created_date': str,
    'expected_difficulty': str,  # 'easy', 'moderate', 'hard'
}
```

2.1.2 Strategy Selection Logic

Algorithm 1 Automatic Strategy Selection

Require: Scenario data with `n_variables`, `formulation_type`
Ensure: Selected strategy

if $n_{vars} \leq 30$ AND formulation *not* rotation **then**

  return `direct_qpu`
else if $n_{vars} \leq 20$ **then**

  return `clique_sampler`
else if $30 < n_{vars} \leq 100$ AND formulation = rotation **then**

  return `clique_decomposition`
else if $n_{vars} > 100$ **then**

  return `spatial_temporal_decomposition`
else

  return `clique_sampler` // Default fallback
end if

2.2 Solver Interface Abstraction

2.2.1 Base Solver Interface

```
class BaseSolver:
    """Abstract base class for all solvers"""

    def solve(self, data: Dict, **kwargs) -> Dict:
        """
        Solve the problem.

        Returns:
            {
                'method': str,
                'objective': float,
                'wall_time': float,
                'qpu_time': float,  # 0 for classical
                'violations': int,
                'success': bool,
                'solution': Dict,
            }
        """
        raise NotImplementedError

    def can_handle(self, data: Dict) -> bool:
        """Check if this solver can handle the given problem"""
        raise NotImplementedError
```

2.2.2 Concrete Solver Implementations

1. **DirectQPUSolver**: For micro-scale problems (6-30 vars)

```python
class DirectQPUSolver(BaseSolver):
    """Direct␣QPU␣embedding␣for␣small␣problems"""

    def can_handle(self, data: Dict) -> bool:
        n_vars = data['n_variables']
        return n_vars <= 30 and data['formulation_type'] != '
            rotation'

    def solve(self, data: Dict, **kwargs) -> Dict:
        # Convert CQM to BQM
        # Use DWaveSampler + EmbeddingComposite
        # Return result
```

2. **CliqueSolver**: For small-scale problems fitting cliques

```python
class CliqueSolver(BaseSolver):
    """DWaveCliqueSampler␣for␣problems␣<=␣20␣vars"""

    def can_handle(self, data: Dict) -> bool:
        return data['n_variables'] <= 20

    def solve(self, data: Dict, **kwargs) -> Dict:
        # Use DWaveCliqueSampler directly
        # Zero embedding overhead
```

3. **CliqueDecompositionSolver**: For small-medium rotation

```python
class CliqueDecompositionSolver(BaseSolver):
    """Farm-by-farm␣decomposition␣with␣clique␣embedding"""

    def can_handle(self, data: Dict) -> bool:
        is_rotation = data['formulation_type'] == 'rotation'
        n_vars = data['n_variables']
        return is_rotation and 30 <= n_vars <= 100

    def solve(self, data: Dict, **kwargs) -> Dict:
        # Decompose by farm (18 vars each)
        # Solve each with DWaveCliqueSampler
        # Coordinate across iterations
```

4. **SpatialTemporalSolver**: For medium-large rotation

```python
class SpatialTemporalSolver(BaseSolver):
    """Spatial-temporal␣decomposition␣for␣large␣problems"""

    def can_handle(self, data: Dict) -> bool:
        is_rotation = data['formulation_type'] == 'rotation'
        return is_rotation and data['n_variables'] > 100

    def solve(self, data: Dict, **kwargs) -> Dict:
        # Cluster farms spatially (2-3 per cluster)
        # Solve temporal periods sequentially
        # 12-variable subproblems with clique embedding
```

5. **GurobiSolver**: Classical ground truth (all scales)

```python
class GurobiSolver(BaseSolver):
    """Optimally-configured␣Gurobi␣for␣all␣problem␣types"""

    def can_handle(self, data: Dict) -> bool:
        return True  # Can handle any problem

    def solve(self, data: Dict, timeout=300, **kwargs) -> Dict:
        # Build MIQP with hard constraints
        # Configure: MIPFocus=1, Presolve=2, Threads=0
        # Return result with timeout handling
```

2.3 Unified Benchmark Runner

```python
class UnifiedBenchmark:
    """
␣␣␣␣Unified␣benchmark␣runner␣for␣all␣scenario␣types␣and␣scales.
␣␣␣␣"""

    def __init__(self):
        self.solvers = {
            'gurobi': GurobiSolver(),
            'direct_qpu': DirectQPUSolver(),
            'clique': CliqueSolver(),
            'clique_decomp': CliqueDecompositionSolver(),
            'spatial_temporal': SpatialTemporalSolver(),
        }

    def run_benchmark(self, scenarios: List[Dict],
                      methods: List[str] = None) -> Dict:
        """
␣␣␣␣␣␣␣␣Run␣benchmark␣across␣multiple␣scenarios.

␣␣␣␣␣␣␣␣Args:
␣␣␣␣␣␣␣␣␣␣␣␣scenarios:␣List␣of␣scenario␣data␣dictionaries
␣␣␣␣␣␣␣␣␣␣␣␣methods:␣List␣of␣method␣names␣to␣test␣(None␣=␣auto-select)

␣␣␣␣␣␣␣␣Returns:
␣␣␣␣␣␣␣␣␣␣␣␣Complete␣results␣dictionary
␣␣␣␣␣␣␣␣"""
        results = {}

        for scenario in scenarios:
            scenario_name = scenario['scenario_name']
            print(f"\n{'='*80}")
            print(f"SCENARIO:␣{scenario_name}")
            print(f"Variables:␣{scenario['n_variables']}")
            print(f"Formulation:␣{scenario['formulation_type']}")
            print(f"{'='*80}\n")

            scenario_results = {}

            # Auto-select methods if not specified
            if methods is None:
                selected_methods = self._select_methods(scenario)
            else:
                selected_methods = methods
```

```
        # Run each method
        for method_name in selected_methods:
            solver = self.solvers.get(method_name)

            if solver is None:
                print(f"␣␣[{method_name}]␣Solver␣not␣found")
                continue

            if not solver.can_handle(scenario):
                print(f"␣␣[{method_name}]␣Cannot␣handle␣this␣
                   problem")
                continue

            print(f"␣␣[{method_name}]␣Running...")
            try:
                result = solver.solve(scenario)
                scenario_results[method_name] = result

                if result['success']:
                    print(f"␣␣␣␣   ␣obj={result['objective']:.4f},␣
                       "
                          f"time={result['wall_time']:.2f}s,␣"
                          f"violations={result['violations']}")
                else:
                    print(f"␣␣␣␣   ␣Failed:␣{result.get('error',␣'
                       unknown')}")

            except Exception as e:
                print(f"␣␣␣␣   ␣Exception:␣{e}")
                scenario_results[method_name] = {
                    'success': False,
                    'error': str(e)
                }

        results[scenario_name] = scenario_results

    return results

def _select_methods(self, scenario: Dict) -> List[str]:
    """Auto-select␣appropriate␣methods␣based␣on␣scenario"""
    n_vars = scenario['n_variables']
    formulation = scenario['formulation_type']

    methods = ['gurobi']  # Always include ground truth

    if n_vars <= 30 and formulation != 'rotation':
        methods.append('direct_qpu')

    if n_vars <= 20:
        methods.append('clique')

    if formulation == 'rotation':
        if 30 <= n_vars <= 100:
            methods.append('clique_decomp')
        if n_vars > 100:
            methods.append('spatial_temporal')
```

```
        return methods
```

3 Scenario Definitions

3.1 Micro-Scale Scenarios (Direct QPU)

3.1.1 Alternative Formulations

```
MICRO_SCENARIOS = [
    {
        'scenario_name': 'portfolio_27crops',
        'formulation_type': 'portfolio',
        'n_variables': 27,
        'scale_category': 'micro',
        'recommended_strategy': 'direct_qpu',
        'description': 'Crop␣portfolio␣selection␣with␣synergies',
        'expected_difficulty': 'easy',
        # Data generation function
        'generator': generate_portfolio_data,
        'generator_args': {'n_crops': 27, 'target_selection': 15},
    },
    {
        'scenario_name': 'graph_mwis_30vars',
        'formulation_type': 'mwis',
        'n_variables': 30,
        'scale_category': 'micro',
        'recommended_strategy': 'direct_qpu',
        'description': 'Maximum␣weighted␣independent␣set',
        'expected_difficulty': 'easy',
        'generator': generate_graph_mwis_data,
        'generator_args': {'n_farms': 5, 'n_crops': 6},
    },
    {
        'scenario_name': 'single_period_30vars',
        'formulation_type': 'single_period',
        'n_variables': 30,
        'scale_category': 'micro',
        'recommended_strategy': 'direct_qpu',
        'description': 'Single-period␣assignment',
        'expected_difficulty': 'easy',
        'generator': generate_single_period_data,
        'generator_args': {'n_farms': 5, 'n_crops': 6},
    },
]
```

3.2 Small-Scale Scenarios (Clique / Decomposition)

3.2.1 Rotation Problems

```
SMALL_SCENARIOS = [
    {
        'scenario_name': 'rotation_micro_25',
        'formulation_type': 'rotation',
        'n_variables': 90,  # 5 farms    6 crops    3 periods
        'scale_category': 'small',
```

```
        'recommended_strategy': 'clique_decomposition',
        'description': 'Multi-period␣rotation␣(5␣farms)',
        'expected_difficulty': 'hard',
        'data_source': 'scenarios/rotation_micro_25.json',
    },
]
```

3.3 Medium-Scale Scenarios (Spatial-Temporal Decomposition)

```
MEDIUM_SCENARIOS = [
    {
        'scenario_name': 'rotation_small_50',
        'formulation_type': 'rotation',
        'n_variables': 180,  # 10 farms    6 crops    3 periods
        'scale_category': 'medium',
        'recommended_strategy': 'spatial_temporal',
        'description': 'Multi-period␣rotation␣(10␣farms)',
        'expected_difficulty': 'hard',
        'data_source': 'scenarios/rotation_small_50.json',
    },
    {
        'scenario_name': 'rotation_medium_100',
        'formulation_type': 'rotation',
        'n_variables': 270,  # 15 farms    6 crops    3 periods
        'scale_category': 'medium',
        'recommended_strategy': 'spatial_temporal',
        'description': 'Multi-period␣rotation␣(15␣farms)',
        'expected_difficulty': 'hard',
        'data_source': 'scenarios/rotation_medium_100.json',
    },
]
```

3.4 Large-Scale Scenarios (Advanced Decomposition)

```
LARGE_SCENARIOS = [
    {
        'scenario_name': 'rotation_large_200',
        'formulation_type': 'rotation',
        'n_variables': 360,  # 20 farms    6 crops    3 periods
        'scale_category': 'large',
        'recommended_strategy': 'spatial_temporal',
        'description': 'Multi-period␣rotation␣(20␣farms)',
        'expected_difficulty': 'hard',
        'data_source': 'scenarios/rotation_large_200.json',
    },
]
```

4 Implementation Example

4.1 Complete Usage Example

```
#!/usr/bin/env python3
"""
Unified␣benchmark␣runner␣example
```

```
"""

# Setup
benchmark = UnifiedBenchmark()

# Load all scenarios
all_scenarios = (
    MICRO_SCENARIOS +
    SMALL_SCENARIOS +
    MEDIUM_SCENARIOS +
    LARGE_SCENARIOS
)

# Run benchmark with auto-selected methods
results = benchmark.run_benchmark(all_scenarios)

# Generate comparison report
generate_unified_report(results)
```

4.2 Custom Method Selection

```
# Test specific combinations
custom_config = {
    'portfolio_27crops': ['gurobi', 'direct_qpu', 'clique'],
    'rotation_micro_25': ['gurobi', 'clique_decomposition'],
    'rotation_small_50': ['gurobi', 'spatial_temporal'],
}

for scenario_name, methods in custom_config.items():
    scenario = get_scenario_by_name(scenario_name)
    results = benchmark.run_benchmark([scenario], methods=methods)
```

5 Results Analysis Framework

5.1 Unified Metrics

For consistent comparison across all scenarios and methods:

Metric	Description
`objective`	Objective function value
`wall_time`	Total execution time (s)
`qpu_time`	Pure QPU execution time (0 for classical)
`violations`	Number of constraint violations
`feasible`	Boolean: zero violations
`gap`	Optimality gap vs. ground truth (%)
`speedup`	Wall time speedup vs. Gurobi
`qpu_efficiency`	`qpu_time / wall_time` ratio

5.2 Cross-Scale Comparison

```
def analyze_scaling(results: Dict) -> pd.DataFrame:
    """Analyze␣how␣performance␣scales␣with␣problem␣size"""
```

```
    rows = []
    for scenario_name, scenario_results in results.items():
        scenario = get_scenario_by_name(scenario_name)
        n_vars = scenario['n_variables']

        for method, result in scenario_results.items():
            if result['success']:
                rows.append({
                    'scenario': scenario_name,
                    'n_variables': n_vars,
                    'formulation': scenario['formulation_type'],
                    'method': method,
                    'objective': result['objective'],
                    'wall_time': result['wall_time'],
                    'qpu_time': result.get('qpu_time', 0),
                    'violations': result['violations'],
                })

    df = pd.DataFrame(rows)

    # Analyze scaling
    for method in df['method'].unique():
        method_df = df[df['method'] == method]
        # Fit power law: time ~ n_vars^alpha
        # Plot scaling curves
        # Generate summary statistics

    return df
```

6 Best Practices

6.1 When to Use Each Strategy

Strategy	Use When
Direct QPU	Variables 30, non-rotation, testing alternative formulations
Clique Sampler	Variables 20, any formulation, benchmark baseline
Clique Decomp	Rotation with 30-100 vars (5 farms), farm-by-farm independence
Spatial-Temporal	Rotation with ¿100 vars (10+ farms), need coordination
Gurobi	Always run as ground truth, optimal settings required

6.2 Common Pitfalls to Avoid

1. **Don't use direct QPU for rotation**: 87% gap due to embedding overhead
2. **Don't skip Gurobi ground truth**: Essential for validating quantum results
3. **Don't compare wall times across methods**: Use QPU-only time for fair comparison
4. **Don't ignore constraint violations**: Feasibility is as important as optimality
5. **Don't use penalty BQM for Gurobi**: Use MIQP with hard constraints

6.3 Reporting Standards

Always report:

- Problem size (variables, constraints)
- Formulation type and structure
- Solver configuration (especially Gurobi parameters)
- Both wall time and QPU-only time
- Optimality gap and constraint violations
- Hardware details (QPU topology, solver version)

7 Conclusion

This integration framework enables seamless benchmarking across:

- Multiple problem scales (6-900 variables)
- Different formulations (portfolio, MWIS, single-period, rotation)
- Various solving strategies (direct, clique, decomposition)
- Classical and quantum approaches

Key principles:

1. **Automatic strategy selection**: Let problem characteristics drive method choice
2. **Unified interface**: Consistent API across all solvers
3. **Comprehensive metrics**: Compare fairly across all dimensions
4. **Scalable design**: Easy to add new scenarios and methods

Implementation Checklist

To implement this framework:

1. Create `BaseSolver` interface with `solve()` and `can_handle()`
2. Implement concrete solvers for each strategy
3. Define scenario dictionaries with metadata
4. Create `UnifiedBenchmark` runner class
5. Add automatic strategy selection logic
6. Generate unified reports with cross-scale analysis
7. Document Gurobi configuration for reproducibility