LikeGT: Simplified Graph-based Genotyping in Rust

Overview

LikeGT is a simplified reimplementation of cosigt's graph-based genotyping approach, written in Rust for improved performance and maintainability. The system performs genotyping by comparing read coverage patterns against known haplotypes in a pangenome graph.

Architecture

Core Components

1. **Graph Construction Pipeline** (`graph` module)
   - Integrates with existing tools (allwave, seqwish, odgi) via command execution
   - Manages graph construction with varying k-mer parameters
   - Handles graph visualization and format conversions
2. **Coverage Analysis** (`coverage` module)
   - Parses gafpack output (gzip-compressed TSV format)
   - Extracts node coverage vectors for both reference paths and sample reads
   - Implements mask support for selective node consideration
3. **Genotyping Engine** (`genotype` module)
   - Computes cosine similarity between coverage vectors
   - Generates haplotype combinations based on ploidy
   - Supports blacklisting of specific paths
   - Outputs ranked genotype predictions
4. **Experiment Framework** (`experiment` module)
   - Hold-two-out validation workflow
   - Read simulation using wgsim
   - Mapping pipeline integration (bwa mem → gfainject → gafpack)
   - Performance evaluation metrics

Data Flow

```
Input FASTA
    ↓
[allwave] → PAF alignment
    ↓
[seqwish] → Initial GFA (multiple k values)
    ↓
[odgi sort] → Sorted GFA
    ↓
[odgi viz] → Visualization
    ↓
Graph Ready for Genotyping
    ↓
Sample Reads → [bwa mem] → SAM
    ↓
[gfainject] → GAF
    ↓
[gafpack] → Coverage vectors
    ↓
[LikeGT genotype] → Genotype predictions
```

Key Algorithms

Cosine Similarity Calculation

The core genotyping algorithm uses cosine similarity to compare coverage patterns:

```rust
pub fn cosine_similarity(a: &[f64], b: &[f64], mask: Option<&[bool]>) -> f64 {
    let dot_product = compute_dot_product(a, b, mask);
    let magnitude = compute_magnitude(a, b, mask);

    if magnitude > 0.0 {
        dot_product / magnitude
    } else {
        0.0
    }
}
```

Haplotype Combination Generation

For a given ploidy level, the system generates all possible combinations of haplotypes:

```rust
pub struct HaplotypeCombinator {
    haplotypes: Vec<String>,
    ploidy: usize,
}

impl HaplotypeCombinator {
    pub fn generate_combinations(&self) -> Vec<Vec<String>> {
        // Generate all k-combinations from n haplotypes
        // Include homozygous combinations (same haplotype repeated)
    }
}
```

Coverage Vector Aggregation

When evaluating multi-haplotype genotypes, coverage vectors are summed:

```rust
pub fn aggregate_coverage(haplotypes: &[Vec<f64>]) -> Vec<f64> {
    let mut sum = vec![0.0; haplotypes[0].len()];
    for hap in haplotypes {
        for (i, val) in hap.iter().enumerate() {
            sum[i] += val;
        }
    }
    sum
}
```

Module Structure

```
src/
  main.rs           # CLI interface and argument parsing
  graph.rs          # Graph construction pipeline
  coverage.rs       # Coverage vector extraction and parsing
  genotype.rs       # Genotyping algorithm implementation
  experiment.rs     # Hold-two-out validation framework
```

```
io.rs            # File I/O utilities (gzip, TSV, JSON)
math.rs          # Mathematical operations (cosine similarity)
utils.rs         # Common utilities and helpers
```

Configuration

Command-line Interface

```
likegt genotype \
    --paths graph.paths.gz \      # Reference path coverage
    --gaf sample.gaf.gz \         # Sample read alignments
    --output results/ \           # Output directory
    --ploidy 2 \                 # Ploidy level
    --blacklist exclude.txt \     # Optional path exclusions
    --mask nodes.mask \           # Optional node mask
    --threads 8                   # Parallelization
```

Hold-Two-Out Experiment

```
likegt experiment \
    --graph input.gfa \           # Input graph
    --fasta sequences.fa \        # Reference sequences
    --holdout 2 \                # Number of paths to hold out
    --coverage 30 \              # Simulated read coverage
    --output experiment/ \        # Output directory
    --threads 8
```

Performance Optimizations

1. **Parallel Processing**
   - Rayon for parallel haplotype combination evaluation
   - Concurrent file I/O where applicable
2. **Memory Efficiency**
   - Streaming parsing of large gzip files
   - Lazy evaluation of combinations
   - Efficient vector operations using SIMD where available
3. **Caching**
   - Memoization of homozygous combinations
   - Reusable coverage vector buffers

Integration Points

External Tool Dependencies

- `allwave`: All-vs-all alignment
- `seqwish`: Graph induction
- `odgi`: Graph manipulation and visualization
- `bwa mem`: Read mapping
- `gfainject`: SAM to GAF conversion
- `gafpack`: Coverage calculation
- `wgsim`: Read simulation

Input Formats

- **FASTA**: Reference sequences
- **GFA**: Graph representation
- **GAF**: Graph alignments
- **TSV.gz**: Coverage vectors from gafpack
- **JSON**: Cluster assignments (optional)

Output Formats

- **TSV**: Genotype predictions with cosine similarities
- **TSV.gz**: All evaluated combinations (sorted)
- **JSON**: Experiment metrics and evaluation results

Testing Strategy

1. **Unit Tests**
   - Math operations (cosine similarity)
   - Combination generation
   - File parsing
2. **Integration Tests**
   - End-to-end pipeline execution
   - Tool integration verification
3. **Validation**
   - Hold-two-out experiments
   - Comparison with original cosigt results
   - Performance benchmarking

Future Enhancements

1. **Algorithm Improvements**
   - Alternative similarity metrics (Jaccard, Euclidean)
   - Machine learning-based genotype scoring
   - Adaptive masking strategies
2. **Performance**
   - GPU acceleration for similarity computations
   - Distributed processing for large datasets
   - Incremental updates for streaming data
3. **Features**
   - Multi-sample joint genotyping
   - Structural variant genotyping
   - Quality score integration
   - Interactive visualization

Development Phases

Phase 1: Core Implementation

- Basic CLI structure
- Coverage vector parsing
- Cosine similarity calculation
- Simple genotyping output

Phase 2: Pipeline Integration

- External tool execution
- Graph construction automation
- Read mapping pipeline

Phase 3: Validation Framework

- Hold-two-out experiments
- Performance metrics
- Comparison tools

Phase 4: Optimization

- Parallelization
- Memory optimization
- Algorithm refinements

Phase 5: Advanced Features

- Clustering support
- Masking capabilities
- Extended output formats