Graphviz Dot Layout Algorithm - PureScript Port Feasibility Analysis

Date: 2025-11-17 **Branch:** `graphviz` **Source:** https://gitlab.com/graphviz/graphviz/-/tree/main/lib/dotgen

---

Executive Summary

Verdict: ✅ **FEASIBLE BUT SUBSTANTIAL** - The Graphviz Dot layout algorithm can be ported to PureScript, but represents a significant undertaking (~5,000 LOC across 150+ functions). A rigorous, test-driven incremental approach is strongly recommended.

Recommended Strategy: Port the algorithm in phases, starting with the simplest components (ranking), building up to the most complex (crossing minimization), with comprehensive test harness at each step.

---

1. Overview: What is Dot Layout?

The Dot layout algorithm creates hierarchical directed graph layouts using the **Sugiyama method** (layered graph drawing). It arranges nodes in horizontal ranks/layers with edges flowing predominantly top-to-bottom.

Algorithm Phases

The algorithm proceeds through **four distinct phases**:

1. **Ranking (dot_rank)** - Assigns nodes to horizontal layers
   - Files: `acyclic.c`, `class1.c`, `class2.c`, `decomp.c`, `fastgr.c`
2. **Crossing Minimization (dot_mincross)** - Orders nodes within layers to minimize edge crossings
   - Files: `mincross.c`, `flat.c`
3. **Positioning (dot_position)** - Computes final (x,y) coordinates
   - Files: `position.c`
4. **Edge Routing (dot_splines)** - Computes spline/polyline paths for edges
   - Files: `dotsplines.c`

Additional support: `cluster.c`, `compound.c`, `aspect.c`, `conc.c`

---

2. Code Size Analysis

Total Codebase

Metric	Count	Notes

Metric	Count	Notes
Total C files	20 files	Core algorithm + infrastructure
Total LOC	~6,500 lines	Including comments, headers
Executable LOC	~4,800 lines	Estimated actual code
Total Functions	~150 functions	Across all files

Breakdown by Phase

Phase 1: Ranking (~1,200 LOC, ~35 functions)

- `acyclic.c` - Cycle breaking via DFS (~100 LOC, 3 functions)
- `class1.c` - Edge classification (~85 LOC, 3 functions)
- `class2.c` - Second classification pass (~200 LOC, ~8 functions, estimated)
- `decomp.c` - Graph decomposition (~400 LOC, ~12 functions, estimated)
- `fastgr.c` - Fast graph operations (~340 LOC, 19 functions)

Complexity: O(V + E) for DFS, O(E × V) for ranking overall

Phase 2: Crossing Minimization (~1,850 LOC, ~55 functions)

- `mincross.c` - Main crossing minimization (~1,850 LOC, ~55 functions)
- `flat.c` - Flat edge handling (~300 LOC, ~10 functions, estimated)

Complexity: O(P × I × N × M) where:

- P = passes (2-3)
- I = iterations (~24)
- N = nodes per rank
- M = edges
- Crossing calculation: $O(N^2)$ per iteration

This is the **most complex phase** - heavy optimization, heuristics, adjacency matrices.

Phase 3: Positioning (~1,100 LOC, ~30 functions)

- `position.c` - Node coordinate assignment (~1,100 LOC, ~30 functions)

Complexity: O(V + E) for constraint generation + network simplex $O(V^3)$ worst case (typically much better)

Uses **network simplex algorithm** for constraint solving - sophisticated optimization.

Phase 4: Edge Routing (~1,800 LOC, ~40 functions)

- `dotsplines.c` - Spline/polyline routing (~1,800 LOC, ~40 functions)

Complexity: O(E log E) for sorting + O(E × V) for path construction

Involves curve generation, bounding boxes, parallel edge spacing.

Support Code (~1,000 LOC, ~30 functions)

- `cluster.c` - Cluster/subgraph handling (~425 LOC, 14 functions)
- `compound.c` - Compound edges (~200 LOC, ~8 functions, estimated)
- `aspect.c` - Aspect ratio adjustment (~150 LOC, ~5 functions, estimated)
- `conc.c` - Edge concentration (~150 LOC, ~5 functions, estimated)
- `dotinit.c` - Initialization/cleanup (~200 LOC, ~5 functions, estimated)

---

3. Dependencies & Data Structures

External Dependencies

The dotgen library depends on other Graphviz libraries:

1. **cgraph** - Core graph data structure (`Agraph_t`, `Agnode_t`, `Agedge_t`)
2. **common** - Utility functions (lists, memory, rendering primitives)
3. **pathplan** - Path planning/routing algorithms
4. **gvc** - Graph visualization context
5. **cdt** - Container data types (dictionaries, lists)

Impact: We must either:

- Port these dependencies (adds ~10,000+ LOC)
- Create PureScript equivalents (significant design work)
- Use simplified versions for our use case

Key Data Structures

```
// Graph structures (from cgraph)
typedef struct Agraph_t { ... } Agraph_t;    // Graph
typedef struct Agnode_t { ... } Agnode_t;    // Node/vertex
typedef struct Agedge_t { ... } Agedge_t;    // Edge

// Layout-specific (from dot.h)
typedef struct rank_t {
    int n;              // Number of nodes in rank
    node_t **v;         // Array of nodes
    int r;              // Rank number
    // ... more fields
} rank_t;

typedef struct graph_t {
    rank_t *rank;       // Array of ranks
    int n_rank;         // Number of ranks
    // ... cluster info, edge lists, etc.
} graph_t;

// Node layout data
typedef struct {
    int rank;           // Node's rank/layer
```

```
    int order;          // Position within rank
    double coord_x;     // Final x coordinate
    double coord_y;     // Final y coordinate
    // ... more fields
} node_layout_t;
```

PureScript Translation Strategy:

- Use records for structs
- Use `Array` for C arrays
- Use `Map` for associative lookups
- Use `ST` monad for mutable algorithms (or pure functional equivalents)

---

4. Key Challenges for PureScript Port

4.1 Imperative Algorithms with Mutation

Challenge: C code heavily uses in-place mutation of graph structures.

Example from mincross.c:

```
static void reorder(graph_t *g, int r, bool reverse, bool hasfixed) {
    int changed = 0, nelt;
    bool muststay;
    node_t **vlist = GD_rank(g)[r].v;  // Mutable array
    node_t **lp, **rp, **ep = vlist + GD_rank(g)[r].n;

    for (nelt = GD_rank(g)[r].n - 1; nelt >= 0; nelt--) {
        lp = vlist;
        while (lp < ep) {
            // Swap nodes in place
            rp = lp + 1;
            if (left2right(g, *lp, *rp)) {
                node_t *tmp = *lp;
                *lp = *rp;
                *rp = tmp;
                changed++;
            }
            lp = rp;
        }
    }
}
```

PureScript Solutions:

- **Option A:** Use `ST` monad with mutable arrays (`ST.Array`)
- **Option B:** Pure functional with persistent data structures (may be slower)
- **Recommendation:** Use `ST` monad to match C performance characteristics

4.2 Complex State Management

Challenge: Graph state is scattered across multiple mutable fields.

Example: Node state includes:

- Rank assignment
- Order within rank
- Coordinates (x, y)
- In/out edge lists
- Cluster membership
- Virtual node flags
- Layout-specific temp data

PureScript Solution:

- Define comprehensive record types
- Use lenses for nested updates
- Separate immutable graph structure from mutable layout state

4.3 Network Simplex Algorithm

Challenge: Position computation uses sophisticated **network simplex** optimization.

Complexity: This is a specialized linear programming algorithm - one of the hardest pieces.

Options:

- Port the network simplex implementation (~500-1000 LOC, non-trivial)
- Use a simpler approximation (loses exactness guarantee)
- Use existing PureScript linear programming library (if available - unlikely)

Recommendation: Port the network simplex - critical for exact equivalence.

4.4 Floating Point Precision

Challenge: Different FP implementations may produce slightly different results.

Mitigation:

- Use tolerance-based comparison (e.g., `abs(a - b) < 1e-6`)
- Test with multiple input graphs
- Document acceptable precision bounds
- Consider fixed-point arithmetic for critical calculations

4.5 Performance

Challenge: PureScript may be slower than optimized C.

Mitigation:

- Use `ST` monad for mutable algorithms

- Profile and optimize hot paths
- Consider FFI to C for performance-critical sections (defeats purpose of port)
- Accept 2-5x slowdown as reasonable for initial version

---

5. Testing Strategy for Exact Equivalence

5.1 Test Harness Architecture

```
For each C function f(input) → output:
  1. Create test case: input graph/data
  2. Run C version: c_output = f_c(input)
  3. Run PureScript version: ps_output = f_ps(input)
  4. Compare: assert(c_output ≈ ps_output within tolerance)
```

5.2 Test Pyramid

Level 1: Unit Tests (Per Function)

- Test each ported function independently
- Compare C vs PureScript output
- ~150 test suites (one per function)

Level 2: Phase Tests

- Test each phase end-to-end
- Compare intermediate state after each phase
- 4 test suites (one per phase)

Level 3: Integration Tests

- Full layout on test graphs
- Compare final node coordinates and edge paths
- 20-50 test graphs of varying complexity

Level 4: Regression Tests

- Standard Graphviz test suite (if available)
- Real-world graph layouts
- Performance benchmarks

5.3 Test Data Generation

Approach 1: Instrumented C Code

- Add logging to C implementation
- Capture inputs/outputs for every function call
- Use as test oracle for PureScript

Approach 2: Known Test Cases

- Small hand-crafted graphs
- Known correct layouts
- Edge cases (cycles, disconnected components, etc.)

Recommendation: Combine both approaches.

5.4 Comparison Mechanism

```
type Tolerance = Number

-- Compare two numbers with tolerance
approxEqual :: Tolerance -> Number -> Number -> Boolean
approxEqual tol a b = abs (a - b) < tol

-- Compare two points
approxEqualPoint :: Tolerance -> {x :: Number, y :: Number} -> {x ::
Number, y :: Number} -> Boolean
approxEqualPoint tol p1 p2 =
  approxEqual tol p1.x p2.x && approxEqual tol p1.y p2.y

-- Compare graph layouts
compareLayouts :: Tolerance -> Layout -> Layout -> Array LayoutDifference
compareLayouts tol l1 l2 = ...
```

Tolerances:

- Coordinates: ±0.01 (1/100th pixel)
- Edge weights/costs: ±0.001
- Ranks: exact (integer)
- Orders: exact (integer)

---

6. Phased Implementation Plan

Phase 0: Infrastructure (2-3 weeks)

Goal: Set up testing framework and basic graph data structures.

Deliverables:

- PureScript graph types (Graph, Node, Edge)
- Test harness infrastructure
- C instrumentation for test oracle
- Basic graph construction/manipulation functions

Estimated LOC: ~500 lines PureScript + infrastructure

Phase 1: Ranking (3-4 weeks)

Goal: Port ranking phase - assigns nodes to layers.

Files to port:

- `acyclic.c` - Cycle breaking (~100 LOC)
- `class1.c` - Edge classification (~85 LOC)
- `class2.c` - Classification pass 2 (~200 LOC)
- `fastgr.c` - Fast graph ops (~340 LOC)
- `decomp.c` - Decomposition (~400 LOC)

Total: ~1,200 LOC, ~35 functions

Test Strategy:

- Unit test each function (5-10 tests per function)
- Integration test: verify ranks match C output
- Test on ~20 graphs

Challenges:

- DFS with mutable marks (use `ST`)
- Edge merging logic
- Network simplex for ranking (simplified version first)

Success Criteria:

- All unit tests pass
- Ranks exactly match C output (integer, so exact comparison)

Phase 2A: Position (Simplified) (2-3 weeks)

Goal: Port positioning WITHOUT network simplex (use simpler heuristic).

Rationale: Get end-to-end layout working before tackling hardest optimization.

Files to port:

- `position.c` - Simplified version (~500 LOC of ~1,100 total)

Test Strategy:

- Coordinates within reasonable bounds
- No node overlaps
- Edges flow top-to-bottom

Success Criteria:

- Produces valid layouts (not necessarily optimal)
- Visual inspection looks reasonable

Phase 2B: Crossing Minimization (4-6 weeks)

Goal: Port crossing minimization - most complex phase.

Files to port:

- `mincross.c` - Main algorithm (~1,850 LOC, ~55 functions)
- `flat.c` - Flat edges (~300 LOC)

Total: ~2,150 LOC, ~65 functions

Challenges:

- Iterative heuristic optimization
- Adjacency matrix operations
- Median calculations
- Transposition logic
- Many edge cases

Test Strategy:

- Unit test each helper (median, transpose, etc.)
- Integration test: compare crossing counts with C
- May not match exactly (heuristic), but should be close

Success Criteria:

- Node orders produce similar crossing counts (within 10%?)
- No regressions on test graphs

Phase 3: Position (Full Network Simplex) (3-4 weeks)

Goal: Port full position optimization with network simplex.

Files to port:

- `position.c` - Complete (~600 LOC remaining + network simplex ~500 LOC)

Challenges:

- Network simplex is complex optimization algorithm
- Constraint generation
- Tight/loose edge handling

Test Strategy:

- Unit test constraint generation
- Compare network simplex intermediate state
- Final coordinates within tolerance (±0.01)

Success Criteria:

- Coordinates match C within ±0.01 pixels

Phase 4: Edge Routing (3-4 weeks)

Goal: Port spline/polyline edge routing.

Files to port:

- `dotsplines.c` - Full routing (~1,800 LOC, ~40 functions)

Challenges:

- Spline mathematics (Bézier curves)
- Bounding box computation
- Parallel edge spacing
- Self-loop routing

Test Strategy:

- Unit test spline generation
- Compare edge control points with C
- Visual comparison

Success Criteria:

- Edge paths match C within tolerance
- No edge-node overlaps

Phase 5: Advanced Features (2-3 weeks)

Goal: Port clusters, compound edges, aspect adjustment.

Files to port:

- `cluster.c` (~425 LOC)
- `compound.c` (~200 LOC)
- `aspect.c` (~150 LOC)
- `conc.c` (~150 LOC)

Total: ~925 LOC

Test Strategy:

- Test graphs with clusters
- Test compound edges
- Test aspect ratio constraints

Success Criteria:

- Clustered layouts match C output
- Compound edges route correctly

Phase 6: Optimization & Polish (2-3 weeks)

Goal: Performance tuning, documentation, API design.

Tasks:

- Profile and optimize hot paths
- Add comprehensive documentation
- Create user-friendly PureScript API
- Integration with PSD3 library
- Example visualizations

---

7. Effort Estimation

Time Estimates (Single Developer, Full-Time)

Phase	Weeks	LOC	Functions	Complexity
Phase 0: Infrastructure	2-3	~500	~10	Medium
Phase 1: Ranking	3-4	~1,200	~35	Medium
Phase 2A: Position (Simple)	2-3	~500	~15	Medium
Phase 2B: Crossing Min.	4-6	~2,150	~65	**High**
Phase 3: Position (Full)	3-4	~1,100	~30	**High**
Phase 4: Edge Routing	3-4	~1,800	~40	High
Phase 5: Advanced Features	2-3	~925	~30	Medium
Phase 6: Polish	2-3	~500	~10	Low
TOTAL	**21-30 weeks**	**~8,675**	**~235**	

Calendar Time: ~5-7 months full-time, or ~10-14 months part-time

Risk Factors

High Risk:

- Network simplex complexity (Phase 3)
- Crossing minimization edge cases (Phase 2B)
- Floating point precision issues
- Performance bottlenecks

Medium Risk:

- Test harness complexity
- C code instrumentation
- Edge routing mathematics

Low Risk:

- Basic graph structures (well-understood)

- Ranking phase (relatively straightforward)

---

8. Alternative Approaches

Option A: Full Port (Recommended Above)

Pros: Exact equivalence, full control, no dependencies **Cons:** Substantial effort (~6 months)

Option B: Simplified "Dot-Like" Layout

Port only essential algorithms, skip optimizations.

Pros: Much faster (~2-3 months) **Cons:** Not exact Dot, may produce worse layouts

Option C: FFI to C Graphviz

Use PureScript FFI to call existing C library.

Pros: Minimal effort (~2 weeks), exact results **Cons:** Requires C dependency, not browser-compatible, defeats learning goal

Option D: Port Subset for Specific Use Case

Port only what's needed for your graph types (e.g., no clusters, no splines).

Pros: Reduced scope (~3-4 months) **Cons:** Not general-purpose

---

9. Recommendation

Recommended Approach: **Incremental Full Port**

Rationale:

1. **Educational Value:** Deep understanding of graph layout algorithms
2. **Browser Compatibility:** Pure PureScript runs in browser
3. **Integration:** Clean integration with PSD3 library
4. **Control:** Full control over algorithm and optimizations
5. **Test-Driven:** Rigorous testing ensures correctness

Suggested Start: Begin with **Phase 0 + Phase 1** (5-7 weeks)

This delivers:

- Working test infrastructure
- Ranking algorithm (most fundamental piece)
- Proof of feasibility
- Foundation for remaining phases

Decision Point: After Phase 1, evaluate:

- Did testing approach work?

- Are results matching C?
- Is performance acceptable?
- Continue to Phase 2 or pivot?

Alternative Quick Win: **Phase 0 + Phase 1 + Phase 2A** (7-10 weeks)

This produces **end-to-end layouts** (albeit not fully optimized) and demonstrates full pipeline.

---

10. Open Questions

1. **Do we need exact Graphviz compatibility, or is "Dot-like" sufficient?**
   - Exact: Full port required
   - Dot-like: Can simplify significantly
2. **What graph sizes do we need to support?**
   - Small (<100 nodes): Any approach fine
   - Medium (100-1000): Need reasonable performance
   - Large (>1000): May need C FFI
3. **Browser vs. Node.js target?**
   - Browser: Pure PureScript required
   - Node.js: Could consider FFI
4. **What's the learning goal vs. production goal ratio?**
   - Learning-focused: Full port valuable
   - Production-focused: FFI might be pragmatic

---

11. Next Steps

Immediate (This Week)

1. **Validate test approach:**
   - Write sample C instrumentation
   - Create test harness prototype
   - Verify can capture C function I/O
2. **Prototype core graph structure:**
   - Define PureScript graph types
   - Implement basic graph operations
   - Test creation/manipulation
3. **Port one simple function:**
   - Pick simple function (e.g., `reverse_edge` from acyclic.c)
   - Port to PureScript

- Write test comparing C vs PS
- Validate testing methodology

If Proceeding (Next 2 Weeks)

4. **Phase 0 completion:**
   - Full test infrastructure
   - C instrumentation framework
   - Graph data structures
   - Basic graph algorithms
5. **Begin Phase 1:**
   - Port `acyclic.c` (cycle breaking)
   - Comprehensive unit tests
   - Validate exact matching

Conclusion

Porting Graphviz Dot layout to PureScript is **feasible and valuable**, but represents a **substantial engineering effort** (~6 months full-time). The rigorous test-driven approach you've outlined is exactly right for ensuring correctness.

Key Success Factors:

- ✅ Incremental, phased approach
- ✅ Comprehensive test harness from day one
- ✅ Test each function against C oracle
- ✅ Start with simplest components
- ✅ Build complexity gradually

Recommended Next Action: Prototype test infrastructure + port one simple function to validate approach (1 week effort).

If that succeeds, commit to Phase 0 + Phase 1 (5-7 weeks) to get ranking working with full test coverage. This provides strong foundation and clear go/no-go decision point for remaining phases.