

An Efficient Algorithm for Dynamic Implicit Strict Posets with Conflict Resolution

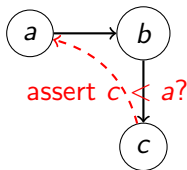
Andrew Chen

Minnesota State University Moorhead

March 12, 2026

The Problem: Dynamic Poset Maintenance

- **Strict Partial Order:** Irreflexive, asymmetric, transitive relation $<$
- **Digraph:** Way of visualizing a strict partial order - will treat terminology as interchangeable here
- **Dynamic Environment:** New assertions arrive over time
- **Challenge:** Ensure new assertion $a < b$ doesn't create a cycle (violation of transitive implication)



Applications: Task scheduling, dependency resolution, constraint systems, food preferences

Existing Approaches

Full Transitive Closure

- Store all pairs (u, v) where $u < v$
- Query: $O(1)$
- Space: $O(V^2)$
- Update: $O(V^2)$
- Unsuitable for large, sparse graphs

Standard DFS Cycle Detection

- Only check for cycles
- Space: $O(V + E)$
- Binary failure (which edge to remove?)
- Typically reports binary success/failure

We need: Efficient updates + cycle identification + space efficiency

Our Approach: Implicit Poset with Conflict Resolution

Our Approach:

- 1 **Implicit representation:** Store only explicit edges, derive transitive closure on-demand
- 2 **Arena-based traversal:** Contiguous memory for cache efficiency
- 3 **Cached timestamps:** Enable $O(L)$ conflict identification (could be other meta-data besides timestamps)

While individual techniques are well-known, their combination for implicit poset maintenance with practical conflict resolution is our contribution.

Complexity:

- Space: $O(V + E)$ vs. $O(V^2)$ for full TC
- Query: $O(V + E)$ worst-case
- Assert: $O(V + E)$ with $O(L)$ conflict identification ($L = \text{cycle length}$)

Our contribution: Efficient implementation combining implicit representation with conflict resolution support

EdgeStore (Red-Black Tree)

- Stores (u, v, t) for each edge
- Dual indexing: forward + reverse
- $O(\log E)$ insertion/lookup
- Clusters outgoing edges

Arena (Vector)

- Temporary workspace per search
- TraceNode: (node_id, parent_index, timestamp)
- Append-only: $O(1)$ allocation
- Contiguous $\rightarrow$ cache-friendly

Contiguous memory $\rightarrow$ CPU prefetcher works efficiently

Arena: $[(a, \perp, 0) \text{ --- } (b, 0, t_1) \text{ --- } (c, 1, t_2) \text{ --- } \dots]$

Algorithm: Path Search with Caching

Goal: Find path from b to a (check if $b < a$ already holds)

```
function FINDPATH( $b, a$ )  
  Arena  $\leftarrow [ (b, \perp, 0) ]$   
  visited  $\leftarrow \{b\}$   
  return RECURSIVESHARECH( $0, a$ )
```

Algorithm: Path Search with Caching

Goal: Find path from b to a (check if $b < a$ already holds)

```
function RECURSIVESEARCH( $index, target$ )  
   $u \leftarrow \text{Arena}[index].\text{node\_id}$   
  for each edge  $(u, v, t)$  do  
    if  $v = target$  then  
      Arena.push( $v, index, t$ )  
      return TRUE  
    if  $v \notin \text{visited}$  then  
      visited  $\leftarrow \text{visited} \cup \{v\}$   
      next_index  $\leftarrow \text{Arena.push}(v, index, t)$   
      if RECURSIVESEARCH( $next\_index, target$ ) then  
        return TRUE  
return FALSE
```

// Cache timestamp!

Extracting Conflict Information via Traceback

If cycle detected: Walk backwards through Arena to extract conflicting edges

```
function IDENTIFYCONFLICTS(Arena)
```

```
  candidates  $\leftarrow \emptyset$ 
```

```
  idx  $\leftarrow$  Arena.size - 1
```

// Start at target

```
  while Arena[idx].parent  $\neq \perp$  do
```

```
    parent_idx  $\leftarrow$  Arena[idx].parent
```

```
    u  $\leftarrow$  Arena[parent_idx].node_id
```

```
    v  $\leftarrow$  Arena[idx].node_id
```

```
    t  $\leftarrow$  Arena[idx].timestamp
```

// Cached!

```
    candidates  $\leftarrow$  candidates  $\cup \{(u, v, t)\}$ 
```

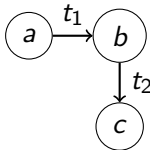
```
    idx  $\leftarrow$  parent_idx
```

```
  return candidates
```

Benefit: Timestamps cached during search enable efficient traceback without re-traversal

Example: Detecting and Identifying Conflict

Existing graph:



Try to assert: $c < a$

Check: Does $a < c$ already hold (implicitly)?

Search from a to c :

- ① Arena = [$(a, \perp, 0)$]
- ② Visit b : Arena = [$(a, \perp, 0), (b, 0, t_1)$]
- ③ Visit c : Arena = [..., $(c, 1, t_2)$]
- ④ **Found c !** Path exists: $a \rightarrow b \rightarrow c$

Traceback:

- Start at c (index 2)
- Edge: $(b, c, t_2) \leftarrow$ candidate
- Edge: $(a, b, t_1) \leftarrow$ candidate

Return: $\{(a, b, t_1), (b, c, t_2)\}$

Complexity Analysis Summary

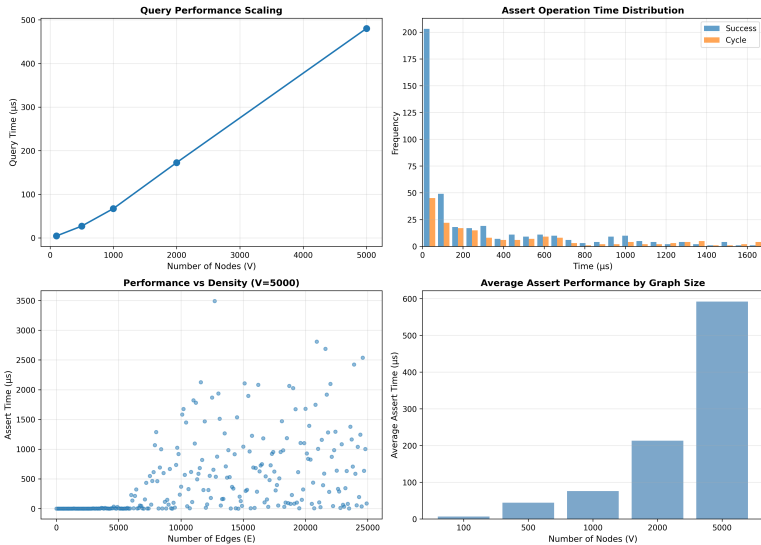
Approach	Space	Query	Assert
Full TC Matrix	$O(V^2)$	$O(1)$	$O(V^2)$
Our Algorithm	$O(V + E)$	$O(V + E)$	$O(V + E)$

Additional costs:

- Conflict identification: $O(L)$ where $L \leq V$ is cycle length
- **Zero additional EdgeStore lookups** (timestamps cached)
- Typical case: $L \ll V$ in sparse graphs

Multiple cycles: If diamond structures exist, may need multiple iterations (each $O(V + E)$)

Figure: Experimental Performance Results



Experimental Results

Implementation: C++17, compiled with `-O3 -march=native`, Apple M2 Pro

Query Scaling:

- $V=100$: $7\ \mu s$
- $V=5000$: $491\ \mu s$
- Linear scaling confirms $O(V + E)$ behavior

Assert Performance:

- Success median: $85\ \mu s$
- Cycle detection median: $280\ \mu s$
- $\sim 3\times$ overhead for conflict identification
- Practical for graphs up to 5K nodes, 25K edges

Key finding: Performance degrades gracefully as density increases
Sparse graphs: single-digit μs — Dense graphs: sub-millisecond

- ➊ **Space-efficient dynamic poset:** $O(V + E)$ space vs. $O(V^2)$ for TC
- ➋ **Built-in conflict identification:** Traceback mechanism provides candidate edges for resolution
- ➌ **Flexible resolution framework:** Returns candidate edges, not just binary failure
- ➍ **Practical implementation:** Demonstrated performance on graphs up to 25K edges
- ➎ **Working implementation:** available at <https://github.com/andrew-chen/cpp-poset>

Applications:

- Package dependency resolution
- Any domain requiring dynamic acyclic ordering

Future Work & Open Questions

Possible future work (I am not territorial, so if you want any of this, go for it, just please let me know - thanks!)

- **Parallelization:** Concurrent queries on very large graphs ($V > 10^6$)
- **Incremental TC:** Hybrid approach for frequently queried subgraphs
- **Optimized resolution policies:** Machine learning to predict best edge removals
- **Comparison study:** Direct empirical comparison with Italiano-style dynamic TC algorithms
- **Alternative memory strategies:** Compare arena vs. custom allocators

Code available: <https://github.com/andrew-chen/cpp-poset>

Efficient dynamic implicit strict posets with constructive conflict resolution

$O(V + E)$ space and time

$O(L)$ conflict identification with zero additional lookups

Practical performance demonstrated empirically

Thank you!

Questions?



Andrew Chen — chen@mnstate.edu
Minnesota State University Moorhead