Bisimulation partitioning and partition maintenance

On very large directed acyclic graphs

Jelle Hellings

April 28, 2011
Overview

Introduction

Preliminaries

Bisimulation partitioning

Experimental results

Conclusion
The objectives

- Directed acyclic graphs
- External memory
- Bisimulation
  - Partition refinement algorithm
  - Partition maintenance
Example

- Query: is the path root/a/b reachable?
- Query: give all nodes reachable by path root/a/b
Example

- Query: is the path root/a/b reachable?
- Query: give all nodes reachable by path root/a/b
- Answer queries using an 1-index

```
| root |
| a    |
| b    |
| a    |
| b    |
| a    |
| c    |
| c    |
| c    |

| root |
| a    |
| b    |
| c    |
```
Example

- Query: is the path root/a/b reachable?
- Query: give all nodes reachable by path root/a/b
- Answer queries using an 1-index
Example

- Query: is the path root/a/b reachable?
- Query: give all nodes reachable by path root/a/b
- Answer queries using an 1-index
Example

- Query: is the path root/a/b reachable?
- Query: give all nodes reachable by path root/a/b
- Answer queries using an 1-index
Example

- Query: is the path root/a/b reachable?
- Query: give all nodes reachable by path root/a/b
- Answer queries using an 1-index
Overview

Introduction

Preliminaries

Bisimulation partitioning

Experimental results

Conclusion
Node labeled graph

Definition
A (node labeled) graph is represented by a triple $G = \langle N, E, l \rangle$:

- $N$ is a set of nodes,
- $E \subseteq N \times N$ is a directed edge relation,
- $l : N \to \mathcal{D}$ is a node-label function
Node bisimilarity

Definition
Two nodes $n, m$ are bisimilar; denoted as $n \approx m$; if and only if:

- The nodes have the same label; $l(n) = l(m)$,
- For every child $n' \in E(n)$: $\exists m' \in E(m)$ such that $n' \approx m'$,
- For every child $m' \in E(m)$: $\exists n' \in E(n)$ such that $m' \approx n'$
Graph bisimularity

Definition
Two graphs $G_1 = < N_1, E_1, l_1 >$, $G_2 = < N_2, E_2, l_2 >$ are bisimilar; denoted as $G_1 \approx_G G_2$; if and only if:

- For every node $n \in N_1$: $\exists m \in N_2$ such that $n \approx m$,
- For every node $m \in N_2$: $\exists n \in N_1$ such that $n \approx m$
Maximum bisimulation graph

Definition
Graph $G_{\downarrow} = \langle N_{\downarrow}, E_{\downarrow}, l_{\downarrow} \rangle$ is a maximum bisimulation graph of $G = \langle N, E, l \rangle$ if

- $G \approx_G G_{\downarrow}$,
- For every $G' = \langle N', E', l' \rangle$, $G' \approx_G G$ we have $|N_{\downarrow}| \leq |N'|$
Definition
A partition block is a set of nodes

Bisimulation partition blocks
All nodes in the partition block are bisimilar equivalent
Partition

Definition
A partition $P = \{p_1, \ldots, p_n\}$ of a set of nodes $N$ is a set of partition blocks such that:

- The partition blocks contain all nodes from the set $N$: $N = \bigcup_{1 \leq i \leq n} p_i$,
- Each node from the set $N$ is present in only one partition block: $\forall i,j (p_i \cap p_j = \emptyset)$

Bisimulation partition
All bisimilar nodes in the same partition block
Partition refinement

Definition
If we have two different partitions $P_1$ and $P_2$ of a set of nodes $N$ then $P_1$ is a refinement of $P_2$ if and only if:

- For every $p \in P_1$ there is a $p' \in P_2$ such that $p \subseteq p'$,
- For every $p' \in P_2$ there is a set of partition blocks $p_1 \in P_1, \ldots, p_n \in P_1$ such that $p' = \bigcup_{1 \leq i \leq n} p_i$.
Index graph

Definition
An index graph of graph $G = \langle N, E, l \rangle$ with maximum bisimulation graph $G_{\downarrow} = \langle N_{\downarrow}, E_{\downarrow}, l_{\downarrow} \rangle$ is represented by a quadruple $\mathcal{I} = \langle N_{\downarrow}, E_{\downarrow}, l_{\downarrow}, p \rangle$ where:

- $p : N_{\downarrow} \to \mathcal{P}(N)$, a bisimulation partition function,
- For every $m \in P(n)$ we have $m \approx n$
External Memory Algorithms

- Internal memory size $M$
- Block size $B$
External Memory Algorithms

- Internal memory size $M$
- Block size $B$
- $O(\text{SCAN}(N)) = O\left(\frac{N}{B}\right)$ I/Os
- $O(\text{SORT}(N)) = O\left(\frac{N}{B} \log \frac{M}{B} \left(\frac{N}{B}\right)\right)$ I/Os
- $O(\text{PQ}(N)) = O\left(\frac{N}{B} \log \frac{M}{B} \left(\frac{N}{B}\right)\right)$ I/Os
Overview

Introduction

Preliminaries

Bisimulation partitioning

Experimental results

Conclusion
Adapting existing algorithms

Problem

IO efficient bisimulation partitioning algorithm
Adapting existing algorithms

Problem
IO efficient bisimulation partitioning algorithm

Given
- Runtime efficient algorithms for Directed (Acyclic) Graphs
  - $O(|E| \log(|N|))$ by Robert Paige and Robert E. Tarjan (1987)
  - $O(|N| + |E|)$ ‘refinement’ for directed acyclic graphs
- These algorithms ‘access’ all parts of the graph continuously
Adapting existing algorithms

Problem
IO efficient bisimulation partitioning algorithm

Given
- Runtime efficient algorithms for Directed (Acyclic) Graphs
  - $O(|E| \log(|N|))$ by Robert Paige and Robert E. Tarjan (1987)
  - $O(|N| + |E|)$ ‘refinement’ for directed acyclic graphs
- These algorithms ‘access’ all parts of the graph continuously

Solution
- No straight forward adaption of internal memory algorithms
- We start with designing a basic ‘online’ approach
Oracle

Assume we have an oracle

- We can present nodes to this oracle
- Oracle responds with a bisimulation partition block
Oracle

Assume we have an oracle

- We can present nodes to this oracle
- Oracle responds with a bisimulation partition block

Oracle is implementable if

- It knows the label of the node
- For every child of the node:
  - The partition block wherein this child is placed
Partition decision structure

Definition
A partition decision structure $pds$ is a mapping $\mathcal{D} \times \mathcal{P}(N_{\downarrow}) \rightarrow N_{\downarrow}$ providing a single operation $\text{QUERY}(pds, (I, S))$
Definition
A partition decision structure \( pds \) is a mapping \( \mathcal{D} \times \mathcal{P}(N_{\downarrow}) \rightarrow N_{\downarrow} \) providing a single operation \( \text{QUERY}(pds, (l, S)) \)

- The \( pds \) represents \( G_{\downarrow} \)
- \( \text{QUERY} \) is non-trivial to implement
Basic algorithm

1: \(pds \leftarrow \) empty partition decision structure
2: \(M \leftarrow \) empty mapping between nodes and partition blocks
3: \textbf{for each} \(n \in N\), in reverse-topological order \textbf{do}
4: \(k \leftarrow (l(n), \{p : \exists m \in E(n) p = M[m]\})\)
5: \(M[n] \leftarrow \text{QUERY}(pds, k)\)
6: \textbf{print} \((M[n], n)\)
7: \textbf{end for}
Analysis

- Reverse-topological order
- Runtime cost is $O(|N| + |E| + \text{QUERY}(|N|))$
- Storage cost is $O(|N|)$ for $M$
- Storage cost is at least $O(|N_{\downarrow}| + |E_{\downarrow}|)$ for $pds$
- Algorithm is not IO efficient
Analysis

- Reverse-topological order
- Runtime cost is $O(|N| + |E| + \text{QUERY}(|N|))$
- Storage cost is $O(|N|)$ for $M$
- Storage cost is at least $O(|N\downarrow| + |E\downarrow|)$ for $pds$
- Algorithm is not IO efficient
  - Implementation details $pds$
  - Structure $M$ is randomly accessed
Time-forward processing

Idea: send partition blocks from child node to parents

- Nodes are reverse-topological ordered
- Give nodes numeric identifiers $i$
- Give partition blocks numeric identifiers $p$
- Per node $n$: add $(i, p)$ to a queue for every parent $i$
- Order queue on identifier: use priority queue
Time-forward processing algorithm

1: \( P \leftarrow \) empty partition decision structure
2: \( Q \leftarrow \) empty minimum-priority queue
3: \textbf{for each} \( n \in N \), in reverse-topological order \textbf{do}
4: \( k \leftarrow (l(n), \{ c : \text{TOP}(Q) = (n, c) \}) \)
5: \( p \leftarrow \text{QUERY}(P, k) \)
6: \textbf{print} \ ((p, n))
7: \textbf{for each} \( m \in E'(n) \) \textbf{do}
8: \( \text{ADD}(Q, (m, p)) \)
9: \textbf{end for}
10: \textbf{end for}
Time-forward processing algorithm

1: \( P \leftarrow \) empty partition decision structure
2: \( Q \leftarrow \) empty minimum-priority queue
3: \textbf{for each} \( n \in N \), in reverse-topological order \textbf{do}
4: \( k \leftarrow (l(n), \{ c : \text{TOP}(Q) = (n, c) \}) \)
5: \( p \leftarrow \text{QUERY}(P, k) \)
6: \textbf{print} \( (p, n) \)
7: \textbf{for each} \( m \in E'(n) \) \textbf{do}
8: \( \text{ADD}(Q, (m, p)) \)
9: \textbf{end for}
10: \textbf{end for}

\textbf{Analysis}

IO cost is \( O(\text{SCAN}(|N| + |E|) + \text{PQ}(|E|) + \text{QUERY}(|N|)) \)
pds: queries in unpredictable order

Reverse-topological orderings

- \( c <_{rt} b <_{rt} a <_{rt} c <_{rt} b <_{rt} a <_{rt} c <_{rt} b <_{rt} a \)
- \( c <_{rt} c <_{rt} c <_{rt} b <_{rt} b <_{rt} b <_{rt} a <_{rt} a <_{rt} a \)
Initial partition

Idea
Use initial partition \( P_i \) of \( N \); bisimulation partition of \( N \) refines \( P_i \)
### Initial partition

#### Idea

Use initial partition $P_i$ of $N$; bisimulation partition of $N$ refines $P_i$

- Label-equivalence partition
  - How to maintain reverse-topological order?
Initial partition

Idea
Use initial partition $P_i$ of $N$; bisimulation partition of $N$ refines $P_i$

- Label-equivalence partition
  - How to maintain reverse-topological order?
- Rank-label equivalence partition
  - Rank of node $n$ is length of longest path from $n$ to leaf
  - Order blocks on increasing rank: reverse-topologically sorted
Definition
Express bisimulation equivalence as equivalence of a recursively defined node value $\nu$

- For leaf node $n$: value $\nu(n) = (l(n), \emptyset)$
- For non-leaf node $n$: value $\nu(n) = (l(n), \{\nu(m) : m \in E(n)\})$
Node value $v$ is ‘arbitrary large’
- Node value $v$ is ‘arbitrary large’
- Map $v = (l, S)$ to value:
  - Depth of nested sets: rank
  - (Perfect) hash function: structural summary
• Node value $v$ is ‘arbitrary large’
• Map $v = (l, S)$ to value:
  • Depth of nested sets: rank
  • (Perfect) hash function: structural summary
• Partition on (rank, label, structural summary)
  • We need rank for reverse-topological ordering
Annotation algorithm

1: \( N', E' \leftarrow \) empty list, empty list
2: \( Q \leftarrow \) empty minimum priority queue
3: for each \( n \in N \), in reverse-topological order do
4: \( S \leftarrow \{(r + 1, h) : \text{TOP}(Q) = (n, r, h)\} \)
5: \( r, h \leftarrow \max(\{r : (r, h) \in S\}), \text{HASH}(\{h : (r, h) \in S\}) \)
6: \( \text{ADD}(N', (r, l(n), h, n)) \)
7: for each \( m \in E(n) \) do
8: \( \text{ADD}(E', (n, m)) \)
9: \( \text{ADD}(Q, (m, r, h)) \)
10: end for
11: end for
12: \( \text{SORT}(N') \)
13: \( \text{REORDER}(E') \)
Analysis

- \( O(\text{scan}(|N| + |E|) + \text{sort}(|N|) + \text{sort}(|E|) + \text{PQ}(|E|)) \)
- Incremental hash function
- Do we need bisimulation after annotation?
Analysis

- $O(\text{SCAN}(|N| + |E|) + \text{SORT}(|N|) + \text{SORT}(|E|) + \text{PQ}(|E|))$
- Incremental hash function
- Do we need bisimulation after annotation?
  - Hash introduces probability on collisions
    - Different values $v \neq v'$ get same hash value $h(v) = h(v')$
Analysis

- $O(\text{SCAN}(|N| + |E|) + \text{SORT}(|N|) + \text{SORT}(|E|) + \text{PQ}(|E|))$
- Incremental hash function
- Do we need bisimulation after annotation?
  - Hash introduces probability on collisions
    - Different values $v \neq v'$ get same hash value $h(v) = h(v')$
  - Thus: structural summaries is not full bisimulation
  - Structural summaries expected to provide good initial partition
Collision propagation

\[
\begin{array}{cc}
\text{a} & (1 + 2) \\
\downarrow & \\
\text{b} & (2 + 0) \\
\downarrow & \\
\text{c} & (0) \\
\end{array}
\quad 
\begin{array}{cc}
\text{a} & (1 + 2) \\
\downarrow & \\
\text{b} & (2 + 0) \\
\downarrow & \\
\text{d} & (0) \\
\end{array}
\]
Handling collisions and collision propagation

- Problem:
  - Partitions can have collisions
  - Collisions propagate
- When processing partition block $p$:
  - Children of nodes in $p$ are processed
Handling collisions and collision propagation

- **Problem:**
  - Partitions can have collisions
  - Collisions propagate

- **When processing partition block $p$:**
  - Children of nodes in $p$ are processed

- **Locally refine each initial partition block**
  - We have full $pds$ keys to refine on
Handling collisions and collision propagation

- Problem:
  - Partitions can have collisions
  - Collisions propagate

- When processing partition block $p$:
  - Children of nodes in $p$ are processed

- Locally refine each initial partition block
  - We have full $pds$ keys to refine on
  - We can refine on hash of this key
Handling collisions and collision propagation

- **Problem:**
  - Partitions can have collisions
  - Collisions propagate

- **When processing partition block** $p$:
  - Children of nodes in $p$ are processed

- **Locally refine each initial partition block**
  - We have full $pds$ keys to refine on
  - We can refine on hash of this key
  - Refine on size of key
External memory bisimulation

- Initial partition: using structural summaries
- Refine each initial partition; gives local partitions
External memory bisimulation

- Initial partition: using structural summaries
- Refine each initial partition; gives local partitions
- Local collisions still possible
  - Low probability: ‘constant factor’
External memory bisimulation

- Initial partition: using structural summaries
- Refine each initial partition; gives local partitions
- Local collisions still possible
  - Low probability: ‘constant factor’
- Implement locally needed \textit{pds} as list of (key, partition block)
  - Keys have a fixed size
  - ‘constant number of keys per local partition’
Analysis

- $O(\text{SCAN}(|N| + |E|) + \text{SORT}(|N|) + \text{SORT}(|E|) + \text{PQ}(|E|))$
- Do we need structural summaries?
Analysis

- $O(\text{SCAN}(|N| + |E|) + \text{SORT}(|N|) + \text{SORT}(|E|) + \text{PQ}(|E|))$
- Do we need structural summaries?
  - Worst case: $O(\cdots + \frac{\max_{p \in P} |p||E(p)|}{B})$
  - $p$ is a local partition block, $|p| \leq |N|$
Analysis

- $O(\text{SCAN}(|N| + |E|) + \text{SORT}(|N|) + \text{SORT}(|E|) + \text{PQ}(|E|))$
- Do we need structural summaries?
  - Worst case: $O(\cdots + \frac{\max_{p \in P} |p||E(p)|}{B})$
  - $p$ is a local partition block, $|p| \leq |N|$
  - Structural summaries: more initial partition blocks
Overview

Introduction

Preliminaries

Bisimulation partitioning

Experimental results

Conclusion
Datasets

- Random generated graphs
- Transitive closure graphs
Running times for random graphs

- gen
- dagdagfp
- dagfpdagfps
Running times for transitive closure graphs

- gen
- dagdagfp
- dagfpdagfps
Running times for transitive closure graphs
## Partitions for random graphs

<table>
<thead>
<tr>
<th>nodes</th>
<th>structural summaries</th>
<th>without structural summaries</th>
<th>collisions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial</td>
<td>local</td>
<td>initial</td>
</tr>
<tr>
<td>100 · 10^6</td>
<td>99.86%</td>
<td>100.00%</td>
<td>0.02%</td>
</tr>
<tr>
<td>250 · 10^6</td>
<td>99.83%</td>
<td>100.00%</td>
<td>0.01%</td>
</tr>
<tr>
<td>400 · 10^6</td>
<td>99.83%</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>550 · 10^6</td>
<td>99.82%</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>700 · 10^6</td>
<td>99.81%</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>850 · 10^6</td>
<td>99.81%</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>1000 · 10^6</td>
<td>99.80%</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
### Partitions for transitive closure graphs

<table>
<thead>
<tr>
<th>nodes</th>
<th>structural summaries</th>
<th>without structural summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial</td>
<td>local</td>
</tr>
<tr>
<td>$20 \cdot 10^3$</td>
<td>99.62%</td>
<td>100.00%</td>
</tr>
<tr>
<td>$50 \cdot 10^3$</td>
<td>99.89%</td>
<td>100.00%</td>
</tr>
<tr>
<td>$80 \cdot 10^3$</td>
<td>99.91%</td>
<td>100.00%</td>
</tr>
<tr>
<td>$110 \cdot 10^3$</td>
<td>99.95%</td>
<td>100.00%</td>
</tr>
<tr>
<td>$140 \cdot 10^3$</td>
<td>99.39%</td>
<td>100.00%</td>
</tr>
<tr>
<td>$170 \cdot 10^3$</td>
<td>99.97%</td>
<td>100.00%</td>
</tr>
<tr>
<td>$200 \cdot 10^3$</td>
<td>99.96%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>
Overview

Introduction

Preliminaries

Bisimulation partitioning

Experimental results

Conclusion
Graph maintenance

Goal
Techniques for indexing graph datasets
Graph maintenance

Goal
Techniques for indexing graph datasets

What if data changes?
Can we update this graph index efficiently when:

- We add or remove a subgraph
- We add or remove edges
Summary

- Fast (expected) IO efficient bisimulation partitioning
  - Partition decision structure
  - Structural summaries
- Small scale experimental verification
  - Efficient on structured data
  - Efficient on unstructured data
  - Even ‘efficient’ for non-sparse graphs
<table>
<thead>
<tr>
<th>Introduction</th>
<th>Preliminaries</th>
<th>Bisimulation partitioning</th>
<th>Experimental results</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>