Details on the brute-force tool

Jelle Hellings

December 26, 2018

Abstract

This document describes the brute-force process to determine differences in the expressive power of specific fragments of the relation algebra. We also describe some implementation details we used to speed-up the described brute-force procedures.

1 Introduction

The *graph data model*, in which data is represented by labeled binary relations, is a versatile and natural data model for representing RDF data, social networks, gene and protein network, and other types of data. Many practical query languages for graph data are based on fragments of Tarski's *relation algebra*. Examples include XPath, SPARQL, the RPQs, and GXPath. In our study of the expressive power of fragments of the relation algebra, we have used two *brute-force techniques* that we detail in this note. Before we do so, we introduce the necessary terminology and notations.[1]

In this note, a *graph* is a triple $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$, with $\mathcal{V}$ a finite set of nodes, Σ a finite set of edge labels, and $\mathbf{E} : \Sigma \rightarrow 2^{\mathcal{V}\times\mathcal{V}}$ a function mapping edge labels to edge relations. We write $[\![q]\!]_{\mathcal{G}}$ to denote the *evaluation* of query q on graph $\mathcal{G}$. We can interpret a query q as a *Boolean query*, in which case $[\![q]\!]_{\mathcal{G}} \neq \emptyset$ represents `True`. For simplicity, we assume that queries always yield binary relations (sets of node pairs, $[\![q]\!]_{\mathcal{G}} \subseteq \mathcal{V} \times \mathcal{V}$). If R is a binary relation, then $\mathrm{R}|_1 = \{m \mid \exists n\ \langle m, n\rangle \in \mathrm{R}\}$ and $\mathrm{R}|_2 = \{n \mid \exists m\ \langle m, n\rangle \in \mathrm{R}\}$ denote the first and second column, respectively, of R.

Definition 1. The *relation algebra* is defined by the grammar

$$e := \emptyset \mid \mathrm{id} \mid \mathrm{di} \mid \ell \mid \ell^{\smallfrown} \mid \pi_j[e] \mid \overline{\pi}_j[e] \mid e \circ e \mid e \cup e \mid e \cap e \mid e - e,$$

in which $\ell \in \Sigma$ and $j \in \{1, 2\}$. Let $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ be a graph and let e be an expression.[2] The semantics of evaluation is defined as follows:

$$\begin{aligned}
[\![\emptyset]\!]_{\mathcal{G}} &= \emptyset;\\
[\![\mathrm{id}]\!]_{\mathcal{G}} &= \{\langle m, m\rangle \mid m \in \mathcal{V}\};\\
[\![\mathrm{di}]\!]_{\mathcal{G}} &= \{\langle m, n\rangle \mid m, n \in \mathcal{V} \wedge m \neq n\};\\
[\![\ell]\!]_{\mathcal{G}} &= \mathbf{E}(\ell);
\end{aligned}$$

[1] The terminology and notation used in this note is based on the work of Hellings [4].

[2] Usually we also consider the Kleene-star operator. As the brute-force techniques are always with respect to given finite graphs, the Kleene-star operator does not add expressive power over composition and union, and, hence, does not need to be considered.

$$
\begin{aligned}
[\![\ell^{\frown}]\!]_{\mathcal{G}} &= \{\langle n, m\rangle \mid \langle m, n\rangle \in \mathbf{E}(\ell)\};\\
[\![\pi_j[e]]\!]_{\mathcal{G}} &= \{\langle m, m\rangle \mid m \in [\![e]\!]_{\mathcal{G}}|_j\};\\
[\![\overline{\pi}_j[e]]\!]_{\mathcal{G}} &= [\![\mathrm{id}]\!]_{\mathcal{G}} - [\![\pi_j[e]]\!]_{\mathcal{G}};\\
[\![e_1 \circ e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \circ [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 \cup e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \cup [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 \cap e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} \cap [\![e_2]\!]_{\mathcal{G}};\\
[\![e_1 - e_2]\!]_{\mathcal{G}} &= [\![e_1]\!]_{\mathcal{G}} - [\![e_2]\!]_{\mathcal{G}},
\end{aligned}
$$

in which $\mathrm{R}_1 \circ \mathrm{R}_2 = \{\langle m, n\rangle \mid \exists z\ (\langle m, z\rangle \in \mathrm{R}_1 \wedge \langle z, n\rangle \in \mathrm{R}_2)\}$. We write $\mathcal{F} \subseteq \{\mathrm{di}, ^{\frown}, \pi, \overline{\pi}, \cap, -\}$ to denote a set of operators in which π represents both π_1 and π_2 and, likewise, $\overline{\pi}$ represents both $\overline{\pi}_1$ and $\overline{\pi}_2$. By $\mathcal{N}(\mathcal{F})$ we denote the fragment of $\mathcal{N}$ that only allows $\emptyset$, id, $\ell \in \Sigma$, $\circ$, $\cup$, and all operators in $\mathcal{F}$.

The brute-force techniques are used to study the expressive power of query languages: in specific, they can be used to determine whether query languages differ in expressive power. To do so, we introduce two notions of query language equivalence: path equivalence and Boolean equivalence.

Definition 2. We say that queries q_1 and q_2 are *path-equivalent*, denoted by $q_1 \equiv_{\mathrm{path}} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}} = [\![q_2]\!]_{\mathcal{G}}$ and are *Boolean-equivalent*, denoted by $q_1 \equiv_{\mathrm{bool}} q_2$, if, for every graph $\mathcal{G}$, $[\![q_1]\!]_{\mathcal{G}} = \emptyset$ if and only if $[\![q_2]\!]_{\mathcal{G}} = \emptyset$. Let $z \in \{\mathrm{path}, \mathrm{bool}\}$. We say that the class of queries $\mathcal{L}_1$ is *z-subsumed* by the class of queries $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \preceq_z \mathcal{L}_2$, if every query in $\mathcal{L}_1$ is z-equivalent to a query in $\mathcal{L}_2$. We say that the classes of queries $\mathcal{L}_1$ and $\mathcal{L}_2$ are *z-equivalent*, denoted by $\mathcal{L}_1 \equiv_z \mathcal{L}_2$, if $\mathcal{L}_1 \preceq_z \mathcal{L}_2$ and $\mathcal{L}_2 \preceq_z \mathcal{L}_1$.

2 Determining language inequivalence

To determine whether $\mathcal{L}_1$ and $\mathcal{L}_2$ do not have the same expressive power, we can prove that they are either not path equivalent or not Boolean equivalent. In some cases, we can do so using brute-force methods. The brute-force method determining path-inequivalence is the simplest of the two.

2.1 Determining path-inequivalence

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages. By Definition 2, we have $\mathcal{L}_1 \npreceq_{\mathrm{path}} \mathcal{L}_2$ if we can find a graph and a query q in $\mathcal{L}_1$ such that, for every query q', we have $[\![q_1]\!]_{\mathcal{G}} \neq [\![q_2]\!]_{\mathcal{G}}$.

One way to determine if this condition holds for a given graph $\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E})$ and languages $\mathcal{L}_1$ and $\mathcal{L}_2$ is by computing the sets of all possible query results of queries in $\mathcal{L}_1$ and $\mathcal{L}_2$ when evaluated on $\mathcal{G}$. Let $\mathcal{F} \subseteq \{\mathrm{di}, ^{\frown}, \pi, \overline{\pi}, \cap, -\}$. We write

$$
X(\mathcal{F}, \mathcal{G}) \equiv \{[\![e]\!]_{\mathcal{G}} \mid e \text{ a query in } \mathcal{N}(\mathcal{F})\}
$$

to denote the set of all query results of queries in $\mathcal{N}(\mathcal{F})$ when evaluated on $\mathcal{G}$. Observe that $[\![e]\!]_{\mathcal{G}} \subseteq \mathcal{V} \times \mathcal{V}$ for every expression in $\mathcal{N}(\mathcal{F})$. Hence, the set $X(\mathcal{F}, \mathcal{G})$ is finite and $\|X(\mathcal{F}, \mathcal{G})\| \leq 2^{\|\mathcal{V}\|^2}$ in the worst case.

Now, to decide $\mathcal{N}(\mathcal{F}_1) \npreceq_{\mathrm{path}} \mathcal{N}(\mathcal{F}_2)$, for $\mathcal{F}_1, \mathcal{F}_2 \subseteq \{\mathrm{di}, ^{\frown}, \pi, \overline{\pi}, \cap, -\}$, we only need to find a graph $\mathcal{G}$ such that $X(\mathcal{F}_1, \mathcal{G}) - X(\mathcal{F}_2, \mathcal{G})$. To gain more insight in the differences in the query languages $\mathcal{N}(\mathcal{F}_1)$ and $\mathcal{N}(\mathcal{F}_2)$, it is also useful to find an expression q such that $[\![q]\!]_{\mathcal{G}} \in (X(\mathcal{F}_1, \mathcal{G}) - X(\mathcal{F}_2, \mathcal{G}))$.

Example 1. Consider a graph $\mathcal{G}$ with two nodes m and n and a single edge $\langle m, n\rangle$ labeled ℓ. We have $[\![\ell^\frown]\!]_{\mathcal{G}} = \{\langle n, m\rangle\}$. Now consider the query language $\mathcal{N}()$. We have

$$X(\emptyset, \mathcal{G}) = \{\emptyset, \{\langle m, n\rangle\}, \{\langle m, m\rangle, \langle n, n\rangle\}, \{\langle m, m\rangle, \langle m, n\rangle, \langle n, n\rangle\}\}.$$

As $[\![\ell^\frown]\!]_{\mathcal{G}} \notin X(\emptyset, \mathcal{G})$, we conclude $\mathcal{N}(^\frown) \npreceq_{\text{path}} \mathcal{N}()$.

Next, we show how to compute the set $X(\mathcal{F}, \mathcal{G})$ together with, for every $\text{R} \in X(\mathcal{F}, \mathcal{G})$, an expression e in $\mathcal{N}(\mathcal{F})$ with $[\![e]\!]_{\mathcal{G}} = \text{R}$. We refer to Algorithm Bruteforce for details.

We observe that Algorithm Bruteforce follows a simple iterative bottom-up process. The while-loop of the algorithm satisfies the following invariants:

1. If $\langle \text{R}, e\rangle \in L$, then e in $\mathcal{N}(\mathcal{F})$ and $[\![e]\!]_{\mathcal{G}} = \text{R}$.
2. $M = \{\text{R} \mid \exists e\ \langle \text{R}, e\rangle \in L\}$.
3. If $\text{R} \in (X(\mathcal{F}, \mathcal{G}) - M)$, then there exists an expression e in $\mathcal{N}(\mathcal{F})$ with $[\![e]\!]_{\mathcal{G}} = \text{R}$ such that there exists a subexpression e' of e with $[\![e']\!]_{\mathcal{G}} = \text{R}'$ and there exists an expression q in $\mathcal{N}(\mathcal{F})$ such that $\langle \text{R}', q\rangle \in \mathcal{L}[i \ldots \|\mathcal{L}\|)$.

2.2 Determining Boolean-inequivalence

We say that $\mathcal{L}$ can *distinguish* between graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ if we can find a query q in $\mathcal{L}$ such that $[\![q]\!]_{\mathcal{G}_1} = \emptyset$ and $[\![q]\!]_{\mathcal{G}_2} \neq \emptyset$.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages. By Definition 2, we have $\mathcal{L}_1 \npreceq_{\text{bool}} \mathcal{L}_2$ if we can find graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $\mathcal{L}_1$ can distinguish between $\mathcal{G}_1$ and $\mathcal{G}_2$, while $\mathcal{L}_2$ cannot distinguish between $\mathcal{G}_1$ and $\mathcal{G}_2$.

Example 2 (Fletcher et al. [3, Proposition 11]). Consider the graphs $\mathcal{G}_3$ and $\mathcal{G}_{\bowtie}$ of Figure 2. The expression $e = (\mathcal{E} \circ \mathcal{E}) - (\mathcal{E} \cup \text{id})$ in $\mathcal{N}(-)$ can distinguish between $\mathcal{G}_3$ and $\mathcal{G}_{\bowtie}$: we have $[\![e]\!]_{\mathcal{G}_3} = \emptyset$ and $[\![e]\!]_{\mathcal{G}_{\bowtie}} \neq \emptyset$.

One way to determine whether $\mathcal{L}$ can distinguish between graphs $\mathcal{G}_1 = (\mathcal{V}_1, \Sigma_1, \mathbf{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \Sigma_2, \mathbf{E}_2)$ is by effectively computing the set of all possible query results of queries in $\mathcal{L}$ when evaluated on both $\mathcal{G}_1$ and $\mathcal{G}_2$. Let $\mathcal{F} \subseteq \{\text{di}, ^\frown, \pi, \overline{\pi}, \cap, -\}$. We write

$$Y(\mathcal{F}, \mathcal{G}_1, \mathcal{G}_2) \equiv \{\langle [\![e]\!]_{\mathcal{G}_1}, [\![e]\!]_{\mathcal{G}_2}\rangle \mid e \text{ a query in } \mathcal{N}(\mathcal{F})\}$$

to denote the set of all query result-pairs of queries in $\mathcal{N}(\mathcal{F})$ when evaluated on $\mathcal{G}_1$ and $\mathcal{G}_2$. Observe that $\langle [\![e]\!]_{\mathcal{G}_1}, [\![e]\!]_{\mathcal{G}_2}\rangle \subseteq (\mathcal{V}_1 \times \mathcal{V}_1) \times (\mathcal{V}_2 \times \mathcal{V}_2)$ for every expression in $\mathcal{N}(\mathcal{F})$. Hence, the set $Y(\mathcal{F}, \mathcal{G}_1, \mathcal{G}_2)$ is finite and $\|X(\mathcal{F}, \mathcal{G})\| \leq 2^{\|\mathcal{V}_1\|^2 \cdot \|\mathcal{V}_2\|^2}$ in the worst case. We can adapt Algorithm Bruteforce in a straightforward way to compute $Y(\mathcal{F}, \mathcal{G}_1, \mathcal{G}_2)$. To determine whether $\mathcal{N}(\mathcal{F})$ can distinguish between $\mathcal{G}_1$ and $\mathcal{G}_2$, we do not have to fully compute $Y(\mathcal{F}, \mathcal{G}_1, \mathcal{G}_2)$: we can stop as soon as we find a pair $\langle \text{R}_1, \text{R}_2\rangle$ with $\text{R}_1 = \emptyset$ and $\text{R}_2 \neq \emptyset$.

2.3 Implementation of the Bruteforce Algorithm

Implementing the Bruteforce Algorithm is rather straightforward. From the description of the algorithm, it already follows that the search space explored by the algorithm can quickly become extremely large, even when operating on very small graphs. To make the tool feasible for practical usages, we have looked for a practical implementation in which each facet of the algorithm is implemented as efficient as possible. Below,

Algorithm BRUTEFORCE($\mathcal{G} = (\mathcal{V}, \Sigma, \mathbf{E}), \mathcal{F}$)**:**

```
1:  𝓛, X := [ ], ∅
2:  ID := {⟨m, m⟩ | m ∈ 𝓥}

3:  #(Add all atomic expressions to 𝓛).
4:  Add(𝓛, X, ∅, ∅)
5:  Add(𝓛, X, id, ID)
6:  Add(𝓛, X, di, {⟨m, n⟩ | m, n ∈ 𝓥 ∧ m ≠ n}) if di ∈ 𝓕
7:  for ℓ ∈ Σ do
8:     Add(𝓛, X, ℓ, E(ℓ))
9:     Add(𝓛, X, ℓ⌢, {⟨n, m⟩ | ⟨m, n⟩ ∈ E(ℓ)}) if ⌢ ∈ 𝓕
10: end for

11: #(Add non-atomic expressions to 𝓛).
12: i := 0
13: while i ≤ ‖𝓛‖ do
14:    ⟨R, e⟩ := 𝓛[i]
15:    Add(𝓛, X, e ∘ e, R ∘ R)
16:    Add(𝓛, X, π₁[e], {⟨m, m⟩ | m ∈ R|₁}) if π ∈ 𝓕
17:    Add(𝓛, X, π₂[e], {⟨m, m⟩ | m ∈ R|₂}) if π ∈ 𝓕
18:    Add(𝓛, X, π̄₁[e], ID − {⟨m, m⟩ | m ∈ R|₁}) if π̄ ∈ 𝓕
19:    Add(𝓛, X, π̄₂[e], ID − {⟨m, m⟩ | m ∈ R|₂}) if π̄ ∈ 𝓕
20:    for (S, q) ∈ 𝓛[0 . . . i) do
21:       Add(𝓛, X, e ∘ q, R ∘ S)
22:       Add(𝓛, X, q ∘ e, S ∘ R)
23:       Add(𝓛, X, e ∪ q, R ∪ S)
24:       Add(𝓛, X, e ∩ q, R ∩ S) if ∩ ∈ 𝓕
25:       Add(𝓛, X, e − q, R − S) if − ∈ 𝓕
26:       Add(𝓛, X, q − e, S − R) if − ∈ 𝓕
27:    end for
28:    i := i + 1
29: end while
30: return S

31: Algorithm Add(𝓛, X, e, R):
32: if R ∉ X then
33:    𝓛 := 𝓛 + [⟨R, e⟩]
34:    X := X ∪ {R}
35: end if
```

Figure 1: Return $X(\mathcal{F}, \mathcal{G})$ with $\mathcal{F} \subseteq \{\text{di}, {}^\frown, \pi, \overline{\pi}, \cap, -\}$.

Figure 2: The 3-clique graph $\mathcal{G}_3$ and the bow-tie graph $\mathcal{G}_{\bowtie}$.

we shall discuss the details and considerations that went into our efforts to implement the BRUTEFORCE Algorithm in C++. We have made the following basic decisions

concerning the algorithm: the list $\mathcal{L}$ is implemented as an `std::vector` of (relation, expression)-pairs, and the set M is implemented as an `std::set` of relation-keys.[3]

The most important aspect of the implementation are the binary relations and of the relation algebra operations on binary relations. We notice that the relation implementation must meet the following criteria:

1. Relation algebra operations on relations must be highly efficient.
2. Relations must have a small memory footprint (to accommodate storing many of them).
3. Relations must have a strict ordering defined on them (to allow their usage in `std::set`).

We have also found that the cost of memory allocation and deallocation contribute significantly to the overall running time of the implementation.

We consider two distinct implementations, each with their own benefits and weaknesses. First, in Section 3, we describe the general-purpose array-based edge-list implementation. Then, in Section 4,we describe specialized matrix-based implementations that can only deal with graphs of up to 16 nodes.

3 Binary relations as arrays

In our setting, we can simply represent nodes by integers and binary relations by arrays of integer-pairs. In such a *relation-array*, we enforce that the node pairs are sorted on lexicographical order and that there are no duplicate node pairs and that the. Hence, if $\mathcal{R} = [\langle v_1, w_1\rangle, \dots, \langle v_{\|\mathcal{R}\|}, w_{\|\mathcal{R}\|}\rangle]$ is a relation-array and $\langle v_i, w_i\rangle, \langle v_j, w_j\rangle \in \mathcal{R}$ with $1 \leq i < j \leq n$, then we enforce that $v_j < v_j$ or $(v_i = v_i) \wedge (w_i < w_j)$.

These structural properties will aid in implementing the necessary operators on binary relations efficiently. The union, intersection, and difference operators ($\cup$, $\cap$, and $-$) can be implemented straightforward by using the highly efficient standard library algorithms `std::set_union`, `std::set_intersect`, and `std::set_difference`, respectively.[4] We refer to Table 1 for details. Hence, we only need to provide efficient implementations for the composition and the projections. Both projection operators are straightforward to implement.

Operator:	Implementation:	Worst-case complexity:	
		Running time:	Buffer size:
$\mathcal{A} \cup \mathcal{B}$	`std::set_union`	$O(a + b)$	k
$\mathcal{A} \cap \mathcal{B}$	`std::set_intersect`	$O(a + b)$	k
$\mathcal{A} - \mathcal{B}$	`std::set_difference`	$O(a + b)$	k

Table 1: Details on each operator in the array-implementation of binary relations. In this table, $a = \|\mathcal{A}\|$, $b = \|\mathcal{B}\|$, and $k = \|\mathcal{R}\|$ with $\mathcal{R}$ the output list.

[3] The `std::set` is usually implemented using binary search trees. Instead, one can also use `std::unordered_set`, which is usually implemented using hash tables, but this set-implementation had difficulties dealing with large amounts of keys.

[4] These standard `C++` algorithms are part of the standard library and are defined in `<algorithm>`.

3.1 Computing compositions

Next, we look at the composition of two lists, $\mathcal{A} \circ \mathcal{B}$. If we have arbitrary lists of node pairs, then the composition can be computed using a simple *nested loop* algorithm. We refer to Figure 3 for details.

Algorithm Compose-NestedLoops($\mathcal{A}$, $\mathcal{B}$)**:**

```
1:  L := [ ]
2:  for ⟨m, n⟩ ∈ A do
3:    for ⟨v, w⟩ ∈ B do
4:      if n = v then
5:        L := L + [⟨m, w⟩]
6:      end if
7:    end for
8:  end for
9:  Sort(L, 0)
10: Unique(L, 0)
```

Figure 3: Return $\mathcal{A} \circ \mathcal{B}$ as a sorted list of edges.

The Compose-NestedLoops Algorithm is very inefficient. Fortunately, we can easily use the lexicographical ordering of relation-arrays to our advantage. Let $\mathcal{A} = [(m_1, n_1), \ldots, (m_{\|\mathcal{A}\|}, n_{\|\mathcal{A}\|})]$ be a relation-array, let m be a node, and let (m_i, n_i), (m_j, n_j), $1 \leq i \leq j \leq \|\mathcal{A}\|$, be the first and the last node pairs in $\mathcal{A}$ with $m_i = m_j = m$. Due to the lexicographical ordering on $\mathcal{B}$, the first node pair $(n, w) \in \mathcal{B}$ with $n_k \leq n$, $i \leq k \leq j$ will be found after the last node pair $(n', w') \in \mathcal{B}$ with $n' < n_k$. Using these observations leads to Algorithm Compose-PartialMerge, see Figure 4 for details.

Algorithm Compose-PartialMerge($\mathcal{A}$, $\mathcal{B}$)**:**

```
1:  L, i := [ ], 0
2:  while i < ‖A‖ do
3:    m, j, k := A[i]_1, 0, ‖L‖
4:    #(Produce pairs ⟨m, w⟩ with w ∈ B|_2).
5:    while i < ‖A‖ and A_1 = m do
6:      n := A[i]_2
7:      #(Search the position of the first pair ⟨v, w⟩ ∈ B with n ≤ v).
8:      j := search(B, j, ‖B‖, n)
9:      #(Produce pairs ⟨m, w⟩ with ⟨m, n⟩ ∈ A, ⟨v, w⟩ ∈ B|_2, and n = v).
10:     while j < ‖B‖ and B[j]_1 = n do
11:       L, j := L + [⟨m, B[j]_2⟩], j + 1
12:     end while
13:     i := i + 1
14:   end while
15: end while
16: return L
```

Figure 4: Return $\mathcal{A} \circ \mathcal{B}$ as a sorted list of edges. For brevity, we have omitted the necessary sort and deduplication steps.

3.2 Other implementation details

In Algorithm Compose-PartialMerge, we have not specified how to search for the first node pair $(n, w) \in \mathcal{B}$ with $n_k \leq n, i \leq k \leq j$. We can use basic well-known search techniques:

Linear search. We can simply traverse $\mathcal{B}$ for every node m. In the worst case, this is optimal. If, however, most pairs in $\mathcal{B}$ do not join with pairs in $\mathcal{A}$, then the traversal will do unnecessary work.

Binary search. Due to the lexicographical ordering, we can also use binary search to search in $\mathcal{B}$. On the one hand, binary search will be fast if only a few pairs in $\mathcal{B}$ join with pairs in $\mathcal{A}$: in this case, binary search will skip most of $\mathcal{B}$. On the other hand, if most pairs in $\mathcal{B}$ join with pairs in $\mathcal{A}$, then straightforward linear traversal will be much more efficient.

Exponential search. To combine the benefits of linear search and binary search, we can also use exponential search [1], as detailed in Figure 5, which will perform acceptable in all cases.

Table 2 lists the complexity of these three search algorithms.

Algorithm Search-Exponential($\mathcal{L}$, *lo*, *hi*, *n*)**:**

1: $lo, step, hi := j, 1, \|\mathcal{L}\|$
2: **while** $lo + step < hi$ **and** $\mathcal{L}[lo + step]_1 < n$ **do**
3: $step := 2 \cdot step$
4: **end while**
5: **return** Search-Binary($\mathcal{L}$, $lo + (step \div 2)$, $\min(lo + step, hi)$)

Figure 5: Algorithm Search-Exponential returns the index of the first pair $\langle v, w \rangle \in \mathcal{L}[lo : hi]$ with $n \leq v$, returns hi if no such pair is found.

Algorithm:	Complexity:
Search-Linear($\mathcal{L}$, *lo*, *hi*, *n*)	$\Theta(k)$
Search-Binary($\mathcal{L}$, *lo*, *hi*, *n*)	$\Theta(\log(hi - lo))$
Search-Exponential($\mathcal{L}$, *lo*, *hi*, *n*)	$\Theta(\log(k))$

Table 2: Complexity of the search algorithms that returns the index $lo + k$ of the first element in $\mathcal{L}[lo, \ldots, hi)$ equal-or-greater than the specified value n.

Using either linear search or exponential search, we have the following (when we omit the necessary sort and deduplication steps):

Theorem 1. *Algorithm Compose-PartialMerge computes $\mathcal{A} \circ \mathcal{B}$ in $O(a + a_1 \times b)$, in which $a = \|\mathcal{A}\|$, $a_1 = \|\mathcal{A}|_1\|$, and $b = \|\mathcal{B}\|$.*

We notice that other join algorithms have better theoretical complexities: in this setting, however, we usually deal with small relations for which this algorithm suffices.

In our implementation we have made several low-level decisions which each increased performance of the relation-arrays significantly. These decisions are:

1. We represent relation-arrays by fixed-size dynamic arrays (a struct holding a pair of pointers to the begin and end of the array in heap memory) instead of `std::vector`, as fixed-size dynamic arrays have a slightly smaller memory footprint.

2. To reduce memory allocations and deallocations, all algorithms use a shared fixed-size buffer and use this buffer to construct all intermediate query results in. After a query result is constructed in this buffer, at which point we know the exact size, we copy the query result to a freshly created relation array of exactly the right size.

3. We can define the strict ordering on relation-arrays $\mathcal{A} < \mathcal{B}$ by using the lexicographical comparison of lists $\mathcal{A} \prec \mathcal{B}$, of which an efficient implementation is provided by `std::lexicographical_compare`. As relation-array comparisons are very common due to the usage of `std::set`, we have opted for the following strict-ordering instead:

$$\mathcal{A} < \mathcal{B} \text{ if } (\|\mathcal{A}\| < \|\mathcal{B}\|) \text{ or } (\|\mathcal{A}\| = \|\mathcal{B}\| \wedge \mathcal{A} \prec \mathcal{B}).$$

   This ordering has the benefit that most relation-array comparisons can be performed without looking up the content of the arrays (which would involve following an additional pointer).

4. We do not store node pairs of x-bit integer nodes as `std::pair<X, X>`, but instead we encode node pairs as a single $2x$-bit value of which the x most significant bits represent the first node, and the remaining bits represent the second node. E.g. node pairs of 8-bit nodes are stored in single `std::uint16_t` values. This will make node pair comparisons much faster. (Notice that node pair comparisons play a crucial role in almost all operations).

4 Binary relations as matrices

Assume we have at most n nodes. It is well known that in this setting a set of node pairs can be represented by a boolean $n \times n$ matrix. We only introduce the minimum notation and background necessary. For more details on matrices and their role in graph representations and graph algorithms, we refer to standard textbooks [2]. Consider the following $n \times n$ matrix:

$$\mathsf{m} = \begin{pmatrix} \mathsf{m}_{1,1} & \mathsf{m}_{1,2} & \dots & \mathsf{m}_{1,n} \\ \mathsf{m}_{2,1} & \mathsf{m}_{2,2} & \dots & \mathsf{m}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{m}_{n,1} & \mathsf{m}_{n,2} & \dots & \mathsf{m}_{n,n} \end{pmatrix}.$$

We, again, represent nodes by unique integers. Let i and j be nodes, $1 \leq i, j \leq n$. We say that the node pair $\langle i, j \rangle$ is in the binary relation represented by m ($\langle i, j \rangle \in \mathsf{m}$) if and only if $\mathsf{m}_{i,j}$ is `True`.

Next, we take a look at how to implement the relation algebra operators we support. First, in Sections 4.1, we present these operators at an abstract level (as matrix operations). Then, in Section 4.2, we look at the details on how to efficiently implement these operators for 64-bit 8×8 matrices using MMX/SSE instructions. Next, in Section 4.3, we briefly look at how to efficiently implement these operators for 64-bit

8×8 matrices using 128-bit SSE2 instructions. Finally, in Section 4.4, we look at the details on how to efficiently implement these operators for 256-bit 16×16 matrices using AVX2 instructions.

4.1 Relation algebra operators on matrices

We look at how to perform each relation operator on matrices.

Union, intersection, and difference. Let u and v be $n \times n$ matrices. If we assume that these matrices are stored as a consecutive sequence of n^2-bits, then the union $\mathsf{u} \cup \mathsf{v}$ can be computed by taking the bitwise-or of u and v and the intersection $\mathsf{u} \cap \mathsf{v}$ can be computed by taking the bitwise-and of u and v.

For the difference $\mathsf{u} - \mathsf{v}$, we first need the *complement* of v. If c is the complement of v, then $\langle i, j \rangle \in \mathsf{c}$ if and only if $\langle i, j \rangle \notin \mathsf{v}$. The complement c can be computed by taking the bitwise-not of c. Them, we simply compute $\mathsf{u} \cap \mathsf{c}$ by taking the bitwise-and of u and c.

Converse. On matrices, the converse operator can be evaluated using the standard *matrix transpose* operator. Let m be a $n \times n$ matrix. The transpose m^T is defined by

$$\begin{pmatrix} \mathsf{m}_{1,1} & \mathsf{m}_{1,2} & \dots & \mathsf{m}_{1,n} \\ \mathsf{m}_{2,1} & \mathsf{m}_{2,2} & \dots & \mathsf{m}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{m}_{n,1} & \mathsf{m}_{n,2} & \dots & \mathsf{m}_{n,n} \end{pmatrix}^\mathsf{T} = \begin{pmatrix} \mathsf{m}_{1,1} & \mathsf{m}_{2,1} & \dots & \mathsf{m}_{n,1} \\ \mathsf{m}_{1,2} & \mathsf{m}_{2,2} & \dots & \mathsf{m}_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{m}_{1,n} & \mathsf{m}_{2,n} & \dots & \mathsf{m}_{n,n} \end{pmatrix}.$$

The equivalence of the converse operator and transposition is straightforward to see. We have $\langle i, j \rangle \in \mathsf{m}^{-1}$ if and only if $\langle j, i \rangle \in \mathsf{m}$ if and only if $\langle i, j \rangle \in \mathsf{m}^\mathsf{T}$.

Projections. Let m be a $n \times n$ matrix. We have

$$\pi_1[\begin{pmatrix} \mathsf{m}_{1,1} & \mathsf{m}_{1,2} & \dots & \mathsf{m}_{1,n} \\ \mathsf{m}_{2,1} & \mathsf{m}_{2,2} & \dots & \mathsf{m}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{m}_{n,1} & \mathsf{m}_{n,2} & \dots & \mathsf{m}_{n,n} \end{pmatrix}] = \begin{pmatrix} p_1 & \bot & \dots & \bot \\ \bot & p_2 & \dots & \bot \\ \vdots & \vdots & \ddots & \vdots \\ \bot & \bot & \dots & p_m \end{pmatrix},$$

in which $\bot = \texttt{False}$ and $p_i = (\mathsf{m}_{1,1} \lor \mathsf{m}_{1,2} \lor \dots \lor \mathsf{m}_{1,n})$, $1 \leq i \leq n$. We can compute $\pi_2[\mathsf{m}]$ by computing $\pi_1[\mathsf{m}^\mathsf{T}]$.

Composition. On matrices, the composition operator can be evaluated using the standard *Boolean matrix multiplication* operator. The Boolean matrix multiplication

$\mathsf{w} = \mathsf{u} \cdot \mathsf{v}$ is defined by

$$\begin{pmatrix} \mathsf{u}_{1,1} & \mathsf{u}_{1,2} & \dots & \mathsf{u}_{1,n} \\ \mathsf{u}_{2,1} & \mathsf{u}_{2,2} & \dots & \mathsf{u}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{u}_{n,1} & \mathsf{u}_{n,2} & \dots & \mathsf{u}_{n,n} \end{pmatrix} \cdot \begin{pmatrix} \mathsf{v}_{1,1} & \mathsf{v}_{1,2} & \dots & \mathsf{v}_{1,n} \\ \mathsf{v}_{2,1} & \mathsf{v}_{2,2} & \dots & \mathsf{v}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{v}_{n,1} & \mathsf{v}_{n,2} & \dots & \mathsf{v}_{n,n} \end{pmatrix} =$$

$$\begin{pmatrix} \bigvee_{1 \le k \le n} (\mathsf{u}_{1,k} \wedge \mathsf{v}_{k,1}) & \bigvee_{1 \le k \le n} (\mathsf{u}_{1,k} \wedge \mathsf{v}_{k,2}) & \dots & \bigvee_{1 \le k \le n} (\mathsf{u}_{1,k} \wedge \mathsf{v}_{k,n}) \\ \bigvee_{1 \le k \le n} (\mathsf{u}_{2,k} \wedge \mathsf{v}_{k,1}) & \bigvee_{1 \le k \le n} (\mathsf{u}_{2,k} \wedge \mathsf{v}_{k,2}) & \dots & \bigvee_{1 \le k \le n} (\mathsf{u}_{2,k} \wedge \mathsf{v}_{k,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \bigvee_{1 \le k \le n} (\mathsf{u}_{n,k} \wedge \mathsf{v}_{k,1}) & \bigvee_{1 \le k \le n} (\mathsf{u}_{n,k} \wedge \mathsf{v}_{k,2}) & \dots & \bigvee_{1 \le k \le n} (\mathsf{u}_{n,k} \wedge \mathsf{v}_{k,n}) \end{pmatrix},$$

with $\mathsf{w}_{i,j} = \bigvee_{1 \le k \le n} (\mathsf{u}_{i,k} \wedge \mathsf{v}_{k,j}) = (\mathsf{u}_{i,1} \wedge \mathsf{v}_{1,j}) \vee (\mathsf{u}_{i,2} \wedge \mathsf{v}_{2,j}) \vee \dots (\mathsf{u}_{i,n} \wedge \mathsf{v}_{n,j})$ for every $1 \le i, j \le n$. The relationship between composition and Boolean matrix multiplication is straightforward. We have $\langle i, j \rangle \in (\mathsf{u} \circ \mathsf{v})$ if and only if there exists a k such that $\langle i, k \rangle \in \mathsf{u}$ and $\langle k, j \rangle \in \mathsf{v}$. Observe that $k \in \{1, \dots, n\}$. Hence, we have $\langle i, j \rangle \in (\mathsf{u} \circ \mathsf{v})$ if and only if $(\langle i, 1 \rangle \in \mathsf{u}) \wedge (\langle 1, j \rangle \in \mathsf{v})$, or $(\langle i, 2 \rangle \in \mathsf{u}) \wedge (\langle 2, j \rangle \in \mathsf{v})$, …, or $(\langle i, n \rangle \in \mathsf{u}) \wedge (\langle n, j \rangle \in \mathsf{v})$ holds. This condition is equivalent to the condition expressed by $\mathsf{w}_{i,j}$. We conclude $\langle i, j \rangle \in (\mathsf{u} \circ \mathsf{v})$ if and only if $\langle i, j \rangle \in \mathsf{w}$.

4.2 The 8-node MMX/SSE implementation

Our first specialized matrix implementation supports graphs with up to 8 nodes and provides high-performance implementations of the operations of Section 4.1 using MMX/SSE instructions. Instead of low-level assembly instructions, we shall use intrinsics to specify specific instructions.[5] Before we discuss the details of each operator, we first look at how we represent 8×8 matrices in memory. Let

$$\mathsf{m} = \begin{pmatrix} \mathsf{m}_{1,1} & \mathsf{m}_{1,2} & \dots & \mathsf{m}_{1,8} \\ \mathsf{m}_{2,1} & \mathsf{m}_{2,2} & \dots & \mathsf{m}_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{m}_{8,1} & \mathsf{m}_{8,2} & \dots & \mathsf{m}_{8,8} \end{pmatrix}$$

be a 8×8 matrix. In memory, this matrix is represented by a 8-byte (64-bit) sequence with the following layout:

$$\underbrace{\overbrace{\mathsf{m}_{1,1}}^{1}\ \overbrace{\mathsf{m}_{1,2}}^{2} \dots \overbrace{\mathsf{m}_{1,8}}^{8}}_{\text{1st byte}}\ \underbrace{\overbrace{\mathsf{m}_{2,1}}^{9}\ \overbrace{\mathsf{m}_{2,2}}^{10} \dots \overbrace{\mathsf{m}_{2,8}}^{16}}_{\text{2nd byte}} \dots \underbrace{\overbrace{\mathsf{m}_{8,1}}^{57}\ \overbrace{\mathsf{m}_{8,2}}^{58} \dots \overbrace{\mathsf{m}_{8,8}}^{64}}_{\text{8th byte}}.$$

Next, we look at how to implement the converse, first projection, and composition efficiently (Section 4.1 already provided efficient implementations for the set operators).

Converse. We will compute the transpose of an 8×8 matrix by extracting each column of the matrix efficiently. To do so, we need two MMX/SSE instructions. First, the SSE instruction

```
int _mm_movemask_pi8(__m64 a)
```

[5]For details on each of the intrinsics used in this document. we refer to the Intel Intrinsics Guide at https://software.intel.com/sites/landingpage/IntrinsicsGuide.

interprets the 64-bit value a as a sequence of eight bytes, and returns the most significant bit in each byte. Interpreted in matrix terms, we have

$$\texttt{_mm_movemask_pi8}(\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,8} \\ a_{2,1} & a_{2,2} & \dots & a_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ a_{8,1} & a_{8,2} & \dots & a_{8,8} \end{pmatrix}) = [a_{1,8} \quad a_{2,8} \quad \dots \quad a_{8,8}].$$

Hence, a single call of _mm_movemask_pi8 will return the last column of matrix a. Next, the MMX instruction

```
__m64 _mm_slli_pi16(__m64 a, int imm8)
```

interprets the 64-bit value a as a sequence of four 16-bit integers, and shifts the bits in each integer imm8 positions to the left (filling with zeros). For our usage, we will only use this instruction with imm8 = 1. Interpreted in matrix terms, we have

$$\texttt{_mm_slli_pi16}(\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,8} \\ a_{2,1} & a_{2,2} & \dots & a_{2,8} \\ a_{3,1} & a_{3,2} & \dots & a_{3,8} \\ \vdots & \vdots & \ddots & \vdots \\ a_{8,1} & a_{8,2} & \dots & a_{8,8} \end{pmatrix}, 1) = \begin{pmatrix} \bot & a_{1,1} & \dots & a_{1,7} \\ a_{1,8} & a_{2,2} & \dots & a_{2,7} \\ \bot & a_{2,1} & \dots & a_{2,7} \\ \vdots & \vdots & \ddots & \vdots \\ a_{7,8} & a_{8,1} & \dots & a_{8,7} \end{pmatrix},$$

in which $\bot$ = False. Hence, a single call of _mm_slli_pi16 will shift every column in the matrix to the right (for our purpose, the newly created first column can be ignored entirely).[6]

Using these two MMX/SSE instructions, the transpose of an 8×8 matrix m can be constructed by Algorithm MatrixTranspose-8 of Figure 6. This algorithm assumes that m is already stored in a variable of type __m64. If this is not the case, then the MMX instruction

```
__m64 _mm_cvtsi64_m64(__int64 a)
```

can be used to load m in a variable of type __m64.

Algorithm MatrixTranspose-8(m)**:**

```
1: r := [0, 0, 0, 0, 0, 0, 0, 0]                 #(r is a sequence of eight bytes).
2: for i := 7 down to 0 do
3:     r[i] := _mm_movemask_epi8(m)             #(Obtain and set the ith row of b^T).
4:     m := _mm_slli_epi16(m, 1)
5: end for
6: return r
```

Figure 6: Return m^T, the converse of the binary relation m.

First projection. Let m be a 8×8 matrix. The ith row $[m_{i,1} \quad m_{i,2} \quad \dots \quad m_{i,8}]$, $1 \le i \le 8$, is stored as the ith byte of m. Hence, we have $m[i] \neq 0$ if and only if $\langle i, i \rangle \in \pi_1[m]$, and we simply set the corresponding bit accordingly.

[6]The x86 and x86-64 architectures are Little-Endian. Hence, shifting the bits in a 16-bit integer to the left, will shift bits to higher addresses in memory ('to the right' in memory).

Composition. We will compute the Boolean matrix product of two 8×8 matrices one entire row at a time. To do so, we will need four additional MMX instructions. First, the MMX instruction

```
__m64 _mm_setzero_si64(void)
```

will return a 64-bit value with all bits set to zero. Interpreted in matrix terms, we have

$$\texttt{_mm_setzero_si64}() = \begin{pmatrix} \bot & \bot & \dots & \bot \\ \bot & \bot & \dots & \bot \\ \vdots & \vdots & \ddots & \vdots \\ \bot & \bot & \dots & \bot \end{pmatrix}.$$

in which $\bot$ = `False`. Second, the MMX instruction

```
__m64 _mm_set1_pi8(char a)
```

will repeat a byte eight times to make a 64-bit value. Interpreted in matrix terms, we have

$$\texttt{_ma_set1_pi8}([a_1 \quad a_2 \quad \dots \quad a_8]) = \begin{pmatrix} a_1 & a_2 & \dots & a_8 \\ a_1 & a_2 & \dots & a_8 \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \dots & a_8 \end{pmatrix}.$$

Third, the MMX instruction

```
__m64 _m_pand(__m64 a, __m64 b)
```

returns the bitwise-and of the two arguments. Interpreted in matrix terms, we have

$$\texttt{_m_pand}(\begin{pmatrix} \mathsf{a}_{1,1} & \mathsf{a}_{1,2} & \dots & \mathsf{a}_{1,8} \\ \mathsf{a}_{2,1} & \mathsf{a}_{2,2} & \dots & \mathsf{a}_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{a}_{8,1} & \mathsf{a}_{8,2} & \dots & \mathsf{a}_{8,8} \end{pmatrix}, \begin{pmatrix} \mathsf{b}_{1,1} & \mathsf{b}_{1,2} & \dots & \mathsf{b}_{1,8} \\ \mathsf{b}_{2,1} & \mathsf{b}_{2,2} & \dots & \mathsf{b}_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{b}_{8,1} & \mathsf{b}_{8,2} & \dots & \mathsf{b}_{8,8} \end{pmatrix}) =$$

$$\begin{pmatrix} \mathsf{a}_{1,1} \land \mathsf{b}_{1,1} & \mathsf{a}_{1,2} \land \mathsf{b}_{1,2} & \dots & \mathsf{a}_{1,8} \land \mathsf{b}_{1,8} \\ \mathsf{a}_{2,1} \land \mathsf{b}_{2,1} & \mathsf{a}_{2,2} \land \mathsf{b}_{2,2} & \dots & \mathsf{a}_{2,8} \land \mathsf{b}_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{a}_{8,1} \land \mathsf{b}_{8,1} & \mathsf{a}_{8,2} \land \mathsf{b}_{8,2} & \dots & \mathsf{a}_{8,8} \land \mathsf{b}_{8,8} \end{pmatrix}.$$

Finally, the MMX instruction

```
__m64 _mm_cmpeq_pi8(__m64 a, __m64 b)
```

will interpret the 64-bit values `a` and `b` as sequences of eight bytes, compare corresponding bytes in these sequences, and return a sequence of eight bytes holding the result. Let $1 \le i \le 8$. In the result, the ith byte is set to 255 (all bits set) if the ith bytes in `a` and `b` are equivalent, and the ith byte is set to 0 (no bits set) if the ith bytes in `a`

and b are not equivalent. Interpreted in matrix terms, we have

$$\texttt{_mm_cmpeq_pi8}\left(\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,8} \\ a_{2,1} & a_{2,2} & \dots & a_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ a_{8,1} & a_{8,2} & \dots & a_{8,8} \end{pmatrix}, \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,8} \\ b_{2,1} & b_{2,2} & \dots & b_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ b_{8,1} & b_{8,2} & \dots & b_{8,8} \end{pmatrix}\right) = \begin{pmatrix} r_1 & r_1 & \dots & r_1 \\ r_2 & r_2 & \dots & r_2 \\ \vdots & \vdots & \ddots & \vdots \\ r_8 & r_8 & \dots r_8 & \end{pmatrix}$$

with r_i = `True` if and only if $[a_{i,1} \quad a_{i,2} \quad \dots \quad a_{i,8}] = [b_{i,1} \quad b_{i,2} \quad \dots \quad b_{i,8}]$.

Using these MMX instructions, the composition of 8×8 matrices can be constructed by Algorithm MatrixMultiply-8 of Figure 7. As with Algorithm MatrixTranspose-8, Algorithm MatrixMultiply-8 assumes that the input, a and b, are already stored in a variable of type `__m64`. If this is not the case, then `_mm_cvtsi64_m64(__int64)` can be used accordingly.

Algorithm MatrixMultiply-8(a, b)**:**

```
1:  r := [0, 0, 0, 0, 0, 0, 0, 0]                #(r is a sequence of eight bytes).
2:  t := b^T
3:  z := _mm_set1_pi8(0)                        #(Matrix with all values set to False).
4:  for i := 0 up to 7 do
5:      i := _mm_set1_pi8(ith byte in a)         #(Every row is the ith row in a).

6:      #(Next, we will compute the entire ith row of r = a · b).
7:      i := _m_pand(i, t)                      #(First, perform all ∧-operations necessary).
8:      i := _mm_cmpeq_pi8(i, z)                #(Next, perform all ∨-operations).

9:      #(The value i_{j,8}, 1 ≤ j ≤ 8, is True if and only if (a · b)_{i,j} is False).
10:     r[i] := ¬(_mm_movemask_epi8(i))          #(Bitwise operators).
11: end for
12: return r
```

Figure 7: Return a · b, the composition of the binary relations a and b.

4.3 The 8-node SSE2 implementation

Certain compilers do not support the 64-bit `__m64` data type or the MMX/SSE instructions used in Algorithms MatrixTranspose-8 and MatrixMultiply-8 when compiling 64-bit binaries. In these cases, it is straightforward to port the code above to use 128-bit SSE2 instructions instead. See Table 3 for details.

4.4 The 16-node AVX2 implementation

Our second specialized matrix implementation supports graphs with up to 16 nodes and provides high-performance implementations of the operations of Section 4.1 using

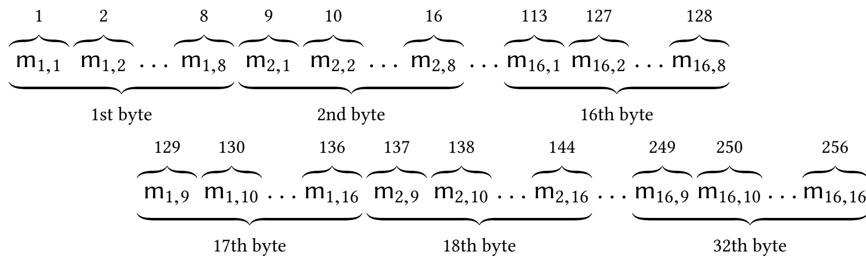
Original instruction:	128-bit SSE2 instruction:
`_mm_movemask_pi8`	`int _mm_movemask_epi8(__m128i a)`
`_mm_slli_pi16`	`__m128i _mm_slli_epi16(__m128i a, int imm8)`
`_mm_cvtsi64_m64`	`__m128i _mm_cvtsi64_si128(__int64 a)`
`_mm_set1_pi8`	`__m128i _mm_set1_epi8 (char a)`
`_m_pand`	`__m128i _mm_and_si128 (__m128i a, __m128i b)`
`_mm_cmpeq_pi`	`__m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)`

Table 3: Mapping between 64-bit MMX/SSE instructions used in Algorithms MatrixTranspose-8 and MatrixMultiply-8 and the corresponding 128-bit SSE2 instructions.

AVX2 instructions. Before we discuss the details of each operator, we first look at how we represent 16×16 matrices in memory. Let

$$\mathsf{m} = \begin{pmatrix} \mathsf{m}_{1,1} & \mathsf{m}_{1,2} & \cdots & \mathsf{m}_{1,8} & \mathsf{m}_{1,9} & \mathsf{m}_{1,10} & \cdots & \mathsf{m}_{1,16} \\ \mathsf{m}_{2,1} & \mathsf{m}_{1,2} & \cdots & \mathsf{m}_{2,8} & \mathsf{m}_{2,9} & \mathsf{m}_{2,10} & \cdots & \mathsf{m}_{2,16} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathsf{m}_{16,1} & \mathsf{m}_{16,2} & \cdots & \mathsf{m}_{16,8} & \mathsf{m}_{16,9} & \mathsf{m}_{16,10} & \cdots & \mathsf{m}_{16,16} \end{pmatrix}$$

be a 16×16 matrix. In memory, this matrix is represented by a 32-byte (256-bit) sequence with the following layout:



Observe that the 256-bit sequence consisting of two 128-bit parts, each encoding eight columns. Next, we look at how to implement the converse, first projection, and composition efficiently (Section 4.1 already provided efficient implementations for the set operators and the second projection).

Converse. We will compute the transpose of a 16×16 matrix in a similar way as how we computed the transpose of a 8×8 matrix. To do so, we need two AVX2 instructions. First, the instruction

```
int _mm256_movemask_epi8(__m256i a)
```

operates similar to `_movemask_pi8`: it interprets the 256-bit value `a` as an array of 32 bytes, and returns the most significant bit in each byte. Interpreted in matrix terms,

we have

$$\texttt{_mm256_movemask_epi8}(\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,16} \\ a_{2,1} & a_{2,2} & \dots & a_{2,16} \\ \vdots & \vdots & \ddots & \vdots \\ a_{16,1} & a_{16,2} & \dots & a_{16,16} \end{pmatrix}) =$$
$$[a_{1,8} \quad a_{2,8} \quad \dots \quad a_{16,8} \quad a_{1,16} \quad a_{2,16} \quad \dots \quad a_{16,16} \quad].$$

Observe that only due to the memory layout we use, the most significant bits in the first 16 bytes represents the 8-th column of the matrix, and the most significant bit in the last 16 bytes represents the 16-th column of the matrix. Hence, due to the memory layout we use, a single call of `_mm256_movemask_epi8` will return the 8th and 16th column of the matrix a. Next, the instruction

```
__m256i _mm256_slli_epi16(__m256i a, int imm8)
```

extends `_mm_slli_pi16` to operate on sixteen 16-bit integers in stead of four. Using these two AVX2 instructions, the transpose of an 16×16 matrix m can be constructed by Algorithm MatrixTranspose-16 of Figure 8. Algorithm MatrixTranspose-16 assumes that m is already stored in a variable of type `__m256i`.

Algorithm MatrixTranspose-16(m)**:**

```
1: r := [0, 0, ..., 0]                            #(r is a sequence of thirty-two bytes).
2: for i := 7 down to 0 do
3:     r := _mm256_movemask_epi8(m)               #(Obtain the ith and (i+8)th row of b^T).
4:     m := _mm256_slli_epi16(m, 1)
5:     r[i], r[i + 16] := 1st byte in r, 2nd byte in r
6:     r[i + 8], r[i + 24] := 3th byte in r, 4th byte in r
7: end for
8: return r
```

Figure 8: Return m^T, the converse of the binary relation m.

First projection. Let m be a 16×16 matrix. The ith row $[m_{i,1} \quad m_{i,2} \quad \dots \quad m_{i,16}]$, $1 \leq i \leq 16$, is stored in two parts. The part $[m_{i,1} \quad m_{i,2} \quad \dots \quad m_{i,8}]$ is stored in the ith byte of m and the part $[m_{i,9} \quad m_{i,2} \quad \dots \quad m_{i,16}]$ is stored in the $(i{+}16)$th byte of m. Hence, $(m[i] \neq 0) \lor (m[i + 16] \neq 0)$ if and only if $\langle i, i \rangle \in \pi_1[m]$.

To reduce the amount of work, we can construct a 16-byte (128-bit) sequence S consisting of the bitwise-or of the first 16-bytes and the last 16-bytes in m. With this sequence, we have $(S[i] \neq 0)$ if and only if $\langle i, i \rangle \in \pi_1[m]$, and we can simply set the corresponding bit accordingly. We construct S using Algorithm OrLowHigh-16 of Figure 9.

Composition. We will compute the Boolean matrix product of two 16×16 matrices one entire row at a time using Alorithm MatrixTranspose-16 of Figure 10. This Algorithm is based directly on Algorithm MatrixMultiply-8, while taking into account the necessary changes due to the memory layout of 16×16 matrices.

Algorithm OrLowHigh-16(m)**:**

```
1: low := _mm256_extractf128_si256(m, 0)        #(Return first 128-bit half of m).
2: high := _mm256_extractf128_si256(m, 1)       #(Return second 128-bit half of m).
3: return _mm_or_si128(low, high)               #(Bitwise-or of the two halves).
```

Figure 9: Return the sequence S obtained by the bitwise-or of the first 16-bytes and the last 16-bytes in m.

Algorithm MatrixMultiply-16(a, b)**:**

```
1: r := [0, 0, . . . , 0]                       #(r is a sequence of thirty-two bytes).
2: t := b^T
3: z := _mm256_setzero_si256(0)                 #(Matrix with all values set to False).
4: for i := 0 up to 15 do
5:     #(Make a matrix in which every row is the ith row in b^T).
6:     low := _mm_set1_epi8(ith byte in a)              #(Set the first eight columns).
7:     high := _mm_set1_epi8((i+16)th byte in a)        #(Set the last eight columns).
8:     i := _mm256_set_m128i(high, low)     #(Combine first and last eight columns).

9:     #(Next, we will compute the entire ith row of r = a · b).
10:    i := _mm256_and_si256(i, t)
11:    i := _mm256_cmpeq_epi8(i, z)
12:    z := _mm256_movemask_epi8(i)
13:    r[i] := ¬(1st byte in z) ∨ ¬(3rd byte in z)        #(Bitwise operators).
14:    r[i + 16] := ¬(2nd byte in z) ∨ ¬(4th byte in z)   #(Bitwise operators).
15: end for
16: return r
```

Figure 10: Return a · b, the composition of the binary relations a and b.

5 Relation-arrays versus relation-matrices

Usually, a matrix representation is space-efficient if the node pair set is *dense* (has close to the maximum of n^2 pairs), but not when the node pair set is *sparse* (in which case a relation-array implementation is more space-efficient). For small graphs with only a few nodes, this rule of thumb is misleading, however. If we have at most 8 nodes, then we can store any binary relation over these nodes already in a $8^2 = 64$-bit (8-byte) value. Likewise, if we have at most 16 nodes, then we can store any binary relation over these nodes already in a $16^2 = 256$-bit (32-byte) value.

Compare this to the relation-array implementation of Section 3: independent of the number of edges the relation-arrays uses storage for at least two pointers, which, on a 64-bit platform, costs $2 \cdot 64 = 128$-bit of storage.[7] Hence, for very small graphs (at most 8 nodes), the Boolean 8×8 matrix implementation is always smaller, and for small graphs (at most 16 nodes), the Boolean 16×16 matrix implementation is smaller whenever the binary relation holds at least a few node pairs (eight node pairs if we choose the 8-bit nodes).

We refer to Table 4 for an overview.

[7] This is a lower bound, as the process of dynamic memory allocation can have additional storage overheads.

Max. num. nodes:	Implementation:	Memory Requirements (bytes)		
		Base:	Per edge:	Total:
8	MMX/SSE	8		8
16	AVX2	32		32
$2^8 = 256$	Array	$16^\dagger$	2	$16^\dagger + 2e$
$2^{16} = 65536$	Array	$16^\dagger$	4	$16^\dagger + 4e$
$2^{32} = 4294967296$	Array	$16^\dagger$	8	$16^\dagger + 8e$

Table 4: Details on each binary relation implementation. In this table, e represents the number of node pairs in the binary relation, and † indicates that the size depends on the size of pointers on the platform (we have assumed a 64-bit architecture with 8-byte pointers).

References

[1] Jon Louis Bentley and Andrew Chi-Chih Yao. “An almost optimal algorithm for unbounded searching”. In: *Information Processing Letters* 5.3 (1976), pp. 82–87. DOI: 10.1016/0020-0190(76)90071-5.

[2] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. The MIT Press, 2009.

[3] George H. L. Fletcher et al. “Relative Expressive Power of Navigational Querying on Graphs”. In: *Proceedings of the 14th International Conference on Database Theory*. ACM, 2011, pp. 197–207. DOI: 10.1145/1938551.1938578.

[4] Jelle Hellings. “On Tarski’s Relation Algebra: querying trees and chains and the semi-join algebra”. PhD thesis. Hasselt University and transnational University of Limburg, 2018.