

## Algorithms for Relations

Klaus Sutner  
Carnegie Mellon University  
www.cs.cmu.edu/~sutner

## Battleplan

- Implementing Relations
- Composition
- Kernel Relations
- Closures
- Single Pair

## Implementing Relations

## Data Structures for Relations

So how do we compute with relations? First off, we have to fix data structures to represent relations, at least in the case where the carrier set is finite and, in fact, reasonably small.

To simplify matters we will often identify the finite carrier set  $A$  of cardinality  $n$  with  $[n] = \{1, 2, \dots, n\}$ . In other words, we fix an enumeration  $A = \{a_1, \dots, a_n\}$ . But note that there are  $n!$  enumerations.

Our various representations of binary relations translate quite directly into data structures:

$\rho \subseteq A \times A$	list of ordered pairs
$\rho : A \times A \rightarrow \mathbb{B}$	Boolean matrix
$\rho : A \rightarrow A \rightarrow \mathbb{B}$	list of lists of objects

The last one requires a bit of interpretation, the others are quite straightforward.

## Déjà Vu All Over Again

As mentioned previously, we can think of a binary relation as a digraph. The representations from above correspond to the standard data models for digraph.

- edge lists
- adjacency matrices
- adjacency lists

Still, we will keep things separate since many of the computational problems typically considered in graph-theory are somewhat different from the ones we are interested in here.

But, if you prefer, you can always translate everything into graph-theory. Bad idea.

## An Example

List of pairs:

$((1,1), (1,3), (1,4), (2,3), (2,5), (4,1), (5,1), (5,5))$

List of lists:

1: 1, 3, 4  
2: 3, 5  
3:  
4: 1  
5: 1, 5

Boolean matrix:

1	0	1	1	0
0	0	1	0	1
0	0	0	0	0
1	0	0	0	0
1	0	0	0	1

### Comparison of the 3 Representations

Suppose the carrier set has size  $n$ , and let  $m = |\rho|$  be the number of related pairs. Note that  $m = O(n^2)$ .

The sizes of the data structures are:

- pair list:  $O(m)$
- list of lists:  $O(n + m)$
- Boolean matrix:  $\Theta(n^2)$

Note we assume a uniform cost function here: in the list representations we actually need some  $\log n$  bits to store a single object. This is perfectly reasonable for realistic values of  $n$ , but note that the constants are larger than for the Boolean matrix method.

### Digression: Matrices Are Not Always Bad

It is clear from the size estimates that for large  $n$  Boolean matrices are unacceptable. However, for small  $n$  they can be the method of choice even when the relation is sparse, i.e. when  $m \ll n^2$ .

The reason is that we can exploit bit-parallelism to speed up operations: a singly unsigned integer can represent 32 (or even 64) bits, and we can manipulate them in "one step". As a result, some of the algorithms here have faster real world implementations using Boolean matrices for values of  $n$  up to several hundred.

Another advantage of Boolean matrices is that they make it easy to insert and delete a pair  $(a, b)$  from the relation: in the list-based implementations we have to search.

list of ordered pairs	$O(m)$
Boolean matrix	$O(m_a)$
list of lists of objects	$O(1)$

Here  $m_a = |\{b \in A \mid a \rho b\}|$ .

### Sparse Matrices

As an aside: matrix implementations can be competitive even if the carrier set is large provided that

- the relation itself is sparse, and
- the implementation is based on sparse matrices.

A sparse matrix implementation does not require  $\Theta(n^2)$  storage but  $\Theta(m)$ : only the non-zero entries in the matrix are associated with storage. This approach requires fairly messy pointer-based data structures and is quite difficult to implement.

**Example 1.** In *Mathematica*, a  $100000 \times 100000$  sparse 0/1 matrix can be represented by just 2800656 bytes (as opposed to the  $16 \cdot 10^{10}$  bytes needed by an ordinary matrix this size; 16 byte integers).

### Large Carrier Sets

If the carrier  $A$  is infinite, or just very large, none of the classical data structures will suffice. We can still use a Boolean function, something along the lines of

```
bool rho( const A& x, const A& y );
```

to test whether  $a \rho b$  holds. This is used e.g. in built-in comparison operators in most programming languages.

The big problem is: this representations makes it very hard to implement any kind of operation on  $\rho$ .

For example, suppose we are given  $a, b \in A$  and want to determine whether  $a \rho^2 b$ . This involves a huge search for all possible witnesses and is inefficient at best, and impossible in the infinite case.

### Composition

### Relational Composition

Consider the following computational problem.

**Problem: Relational Composition**  
**Instance:** Two relations  $\rho$  and  $\sigma$  on  $[n]$ .  
**Solution:** The relation  $\tau = \rho \bullet \sigma$ .

Of course, there is a more general version where we have not necessarily square relations, but we will ignore this generalization.

**Claim 1.** In the list representations we can compute  $\tau$  in time linear in  $n + |\tau|$ .

**Exercise 1.** Give precise descriptions of the corresponding algorithms.

## Composition and Boolean Matrices

So suppose input and output are both expected to be Boolean matrices.

In this case we can exploit matrix algebra: composition of relations comes down to multiplication of matrices.

The product  $C = A \cdot B$  of two Boolean matrices is defined by

$$C(i, j) = \bigvee_k A(i, k) \wedge B(k, j)$$

Thus,  $k$  is none other than the witness we have to find in relational composition. Logical  $\wedge$  represents the search over all possible choices of  $k$ .

Hence we can compute the matrix of  $\tau = \rho \bullet \sigma$  in  $\Theta(n^3)$  steps.

## An Isomorphism

The last method bears some closer scrutiny. We are trying to compute in the monoid

$$\langle \text{Rel}_n, \bullet, I \rangle$$

But this monoid is isomorphic to

$$\langle \mathbb{B}^{n,n}, \cdot, I \rangle$$

where  $\cdot$  is matrix multiplication and  $I$  represents the identity matrix.

This is exactly the same trick that allows one to translate multiplication into addition using logarithms.

## Squaring

An important special case is when we compose a relation with itself.

**Problem: Squaring**

Instance: A relation  $\rho$  on  $[n]$ .

Solution: The relation  $\rho^2 = \rho \bullet \rho$ .

If we represent input and output as Boolean matrices this comes down to simple matrix multiplication.

Complexity of the standard algorithm is  $\Theta(n^3)$ .

Is there any way to speed this up?

## Strassen's Algorithm

This is a text-book example of a sophisticated divide-and-conquer approach: first, we subdivide the input matrices into 4 blocks of size  $n/2 \times n/2$  each.

$$P = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Then we compute a number of temporary matrices:

$$\begin{aligned} T_1 &= (A + D)(E + H) & T_5 &= (C + D)E \\ T_2 &= D(G - E) & T_6 &= A(F - H) \\ T_3 &= (B - D)(G + H) & T_7 &= (C - A)(E + F) \\ T_4 &= (A + B)H \end{aligned}$$

Yes, and so?

## Combine the Results

A little calculation shows that the product matrix we are trying to compute is none other than

$$P = \begin{pmatrix} T_1 + T_2 + T_3 - T_4 & T_4 + T_6 \\ T_2 + T_5 & T_1 - T_5 + T_6 + T_7 \end{pmatrix}$$

We have used only 7 multiplications of  $n/2 \times n/2$  matrices and a number of additions and subtractions which are quadratic and don't matter here.

**Theorem 1.** Strassen 1969

Matrix multiplication requires no more than  $O(n^{\log_2 7}) \approx O(n^{2.81})$  steps.

## Some Caveats

The first important constraint is that Strassen's method requires a ring, not just a semiring: we need to be able to subtract. For our application this is actually not an issue, we can think of the Boolean matrices as 0/1-matrices over the integers.

What we are computing, then, are multiplicities:

$$M(a, b) = |\{z \mid a \rho z \rho b\}|$$

but there is no harm in that, we can always cast back down to Booleans.

More importantly, it seems that Strassen's method is not really practical, the overhead is large and only matrices of substantial size benefit from the method.

Note that there are even faster methods such as Coppersmith and Winograd which runs in time  $O(n^{2.37})$ , alas, they do not seem to be practical either.

### Higher Powers

**Definition 1.** Let  $\rho$  be a relation on  $A$ . The powers  $\rho^i$ ,  $i \geq 0$ , of  $\rho$  are defined by

$$\begin{aligned} \rho^0 &= I_A \\ \rho^{k+1} &= \rho \bullet \rho^k \end{aligned}$$

Here  $I_A$  denotes the identity relation on  $A$ .

Composition on the left is arbitrary, we could also have chosen composition on the right.

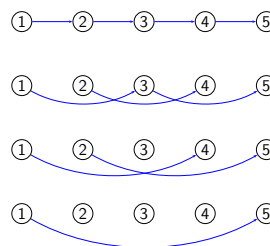
Thus  $x \rho^k y$  if, and only if, there is a chain of length  $k$  in  $\rho$ :

$$\exists x_0, x_1, \dots, x_k (x = x_0 \rho x_1 \rho x_2 \rho \dots x_{k-1} \rho x_k = y)$$

### Power Example

The picture below shows the relation  $x \rho y \iff y = x + 1$  on  $[5]$ .

It is easy to see that in this case,  $\rho^k = \emptyset$  for all  $k \geq 5$ .



Note that the powers of the relation are not subsets of each other, in fact, in this particular case they are pairwise disjoint.

### Computing Powers

The associated computational problem is this:

**Problem: Relational Power**

Instance: A relation  $\rho$  on  $[n]$ , an integer  $k \geq 0$ .

Solution: The relation  $\rho^k$ .

Again suppose input and output are both Boolean matrices.

Of course we can use  $k - 1$  matrix multiplications to do this, yielding an unattractive running time of  $\Theta(kM)$  where  $M$  is the cost of matrix multiplication and seems in practice to be no better than  $n^3$ .

Can we speed this up a bit?

### Repeated Squaring

Better is to use fast exponentiation, aka repeated squaring, a method that work in any semigroup (we are here interested in the multiplicative semigroup of matrices).

$$\begin{aligned} x^{2k} &= (x^k)^2 \\ x^{2k+1} &= x(x^k)^2 \end{aligned}$$

It is clear that this approach requires only  $O(\log k)$  multiplications to determine  $x^k$ .

Using the standard matrix multiplication algorithm the total cost is  $O(n^3 \log k)$ .

Other improvements?

### Kernel Relations

### Representing Equivalence Relations

For equivalence relations there is a very compact representation that exploits functions.

**Definition.** Consider a function  $f : A \rightarrow B$ . The kernel relation  $K_f$  of  $f$  is defined by

$$x K_f y \iff f(x) = f(y)$$

**Claim 2.**  $K_f$  is an equivalence relation for any function  $f$ .

We have see several examples already (same area, same parents, . . .)

One might suppose that equivalence relation are more general than kernel relations, but it turns out that every equivalence relation is already a kernel relation.

## A Proof

**Theorem.** Every equivalence relation  $\rho$  on  $A$  is a kernel relation. In fact, we can choose a function  $f : A \rightarrow A$  such that  $K_f = \rho$ .

*Proof.*

We must define a function  $f : A \rightarrow A$  such that  $\rho = K_f$ .

In other words, we need  $x \rho y \iff f(x) = f(y)$ .

For each equivalence class  $[x]$  pick one particular element  $x_0 \in [x]$ .

Set  $f(z) = x_0$  for all  $z \in [x]$ . □

The reader may notice that this argument requires the Axiom of Choice;  $A$  is an abstract set, so we have to be careful about "picking" an element in each class.

## Representing Kernel Relations

Clearly  $f : A \rightarrow A$  can be represented by a simple array of length  $n$ , requiring only  $\Theta(n)$  space.

Question: How should one choose  $x_0$ ?

Standard Answer: Pick the least element in  $[x]$ .

$$f(x) = \min(z \mid z \rho x)$$

This is the *canonical representation*.

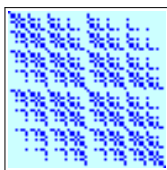
**Example 2.** Congruence modulo 4 on  $[10]$  produces

$x$	1	2	3	4	5	6	7	8	9	10
$f(x)$	1	2	3	4	1	2	3	4	1	2

## Comparison: Boolean Matrices

As we have seen, another very straightforward implementation of (arbitrary) relations is given by Boolean matrices

$$i \rho j \iff R[i, j] = 1.$$



Binary lists of length 6, "same number of 1's"

But: this representation requires  $\Theta(n^2)$  bits, even for the identity relation, the smallest equivalence relation.

## Cost of Operations

Note that in either implementation we can check equivalence of  $x$  and  $y$  in  $O(1)$ : simple array lookup.

How about the following computational problem:

**Problem: Equivalence Relation Meet**

Instance: Two equivalence relations  $\rho, \sigma$  on  $[n]$ .

Solution: The relation  $\rho \sqcap \sigma$ .

If we are interested in detailed complexity analysis, we have to specify the data structures used for input and output.

## Intersections

With Boolean matrices we could do this easily in  $\Theta(n^2)$  steps by masking. How about the kernel representation?

Suppose  $\sigma$  and  $\rho$  are represented by  $f$  and  $g$ .

	1	2	3	...	$p$	...	$n$
$f(1)$	$f(2)$	$f(3)$	...	$f(p)$	...	$f(n)$	
$g(1)$	$g(2)$	$g(3)$	...	$g(p)$	...	$g(n)$	

So the function  $H(x) = (f(x), g(x))$  represents  $\rho \sqcap \sigma$  but it's not the canonical representation  $h$ .

To compute the canonical representation  $h$  we traverse the list, and look for new pairs  $(i, j)$  in the  $f/g$  rows, and enumerate them as we go.

## Pseudo Code

```
// meet( f, g )
for( p = 1; p <= n; p++ )
{
    i = f[p];
    j = g[p];
    if( (i, j) is new pair )           // ****
        T[p] = val(i, j) = p;
    else
        T[p] = val(i, j);
}
```

To implement `val` we can use a hash table, yielding expected performance  $\Theta(n)$ .

Alternatively, we can use an auxiliary array `valarr[n][n]`, initialized to 0. Disregarding the initialization cost,  $T$  can be computed in  $\Theta(n)$  steps. Note that we can reset `valarr` in linear time after  $T$  has been computed. Thus, we can perform  $m$  meet operations in time  $O(n^2 + m \cdot n)$ .

## Closures

## Dynamic Equivalence Relations

What if we do not know all of  $\rho$  but only certain relationships

$$(a_i, b_i) \quad i = 1, \dots, m$$

In particular assume that these pairs are given to us one at a time and we want to update the equivalence relation as more information becomes available.

More formally: We want to compute the coarsest equivalence relation  $\rho_k$  such that  $a_i \rho b_i$  for all  $i \leq k$  and there should be an easy way to get from  $\rho_k$  to  $\rho_{k+1}$ .

Coarsest simply expresses the idea that we declare elements to be equivalent only when forced to do so by the data. Otherwise,  $U_A$  would be a cheap and useless answer.

For example, we may want to determine connectivity in a network: every time a new link  $(a, b)$  becomes active, we have to recompute the blocks.

## Example

Let  $n = 6$ .

Here are the partitions resulting from 5 pairs.

—	(1), (2), (3), (4), (5), (6)
(1, 3)	(1, 3), (2), (4), (5), (6)
(4, 6)	(1, 3), (2), (4, 6), (5)
(5, 3)	(1, 3, 5), (2), (4, 6)
(5, 1)	(1, 3, 5), (2), (4, 6)
(3, 4)	(1, 3, 4, 5, 6), (2)

## Equivalential Closure

**Definition 2.** Let  $\rho \subseteq A \times A$ . The **equivalential closure** of  $\rho$  is the least equivalence relation  $\rho'$  on  $A$  such that  $\rho \subseteq \rho'$ .

Least here is meant in the sense of  $\sqsubseteq$ .

So we are looking for the  $\sqsubseteq$ -smallest relation

- contains  $\rho$ , and
- is reflexive, symmetric and transitive.

This is a typical example of a **closure operation**, also called a **hull operation**: add as little as possible, but just enough to satisfy certain conditions.

Again, "smallest" is important here, otherwise we could just set  $\rho' = U_A$ .

## Other Closures

In the same way we can define **reflexive closure**, **symmetric closure**, **transitive closure** and **reflexive transitive closure**.

Notation:

$$\text{eqcl}(\rho), \text{rcl}(\rho), \text{scl}(\rho), \text{tcl}(\rho), \text{rtcl}(\rho)$$

**Lemma 1.** All these operations  $F$  are idempotent:  $F(F(\rho)) = F(\rho)$ .

**Lemma 2.**

- $\text{scl}(\text{rcl}(\rho)) = \text{rcl}(\text{scl}(\rho))$
- $\text{rtcl}(\rho) = \text{tcl}(\text{rcl}(\rho)) = \text{rcl}(\text{tcl}(\rho))$
- $\text{eqcl}(\rho) = \text{tcl}(\text{scl}(\text{rcl}(\rho)))$

## Boolean Matrix Case

Suppose we use Boolean matrices as data-structure.

$\text{rcl}(\rho)$ ,  $\text{scl}(\rho)$  are easy:

- turn on all bits on diagonal,
- reflect matrix along diagonal and perform bit-wise or.

How about  $\text{tcl}(\rho)$ ?

This is much harder, but also much more interesting.

Transitive closure corresponds to path existence in the associated digraph and is used in many algorithms.

### Powers and Transitive Closure

Powers and transitive closure are very closely connected. We could have defined the closures like so:

$$\text{rtcl}(\rho) = \bigcup_{k \geq 0} \rho^k = I_A \sqcup \rho \sqcup \rho^2 \sqcup \rho^3 \sqcup \dots$$

$$\text{tcl}(\rho) = \bigcup_{k > 0} \rho^k = \rho \sqcup \rho^2 \sqcup \rho^3 \sqcup \dots$$

Here

**Definition 3.** The join or union  $\rho \sqcup \sigma$  of two relations  $\rho$  and  $\sigma$  is defined by

$$x(\rho \sqcup \sigma)y \iff x \rho y \vee x \sigma y.$$

### Truncating Chains

This characterization simply says that  $x \text{ tcl}(\rho) y$  iff

$$\exists k \geq 1 \exists x_0, x_1, \dots, x_k (x = x_0 \rho x_1 \rho \dots \rho x_{k-1} \rho x_k = y)$$

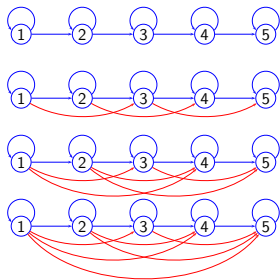
Chains can be of arbitrary length, but in order to compute the transitive closure we only need to concern ourselves with existence: any chain from  $x$  to  $y$  will do and is natural to consider shortest chains.

It is easy to see that  $x \text{ tcl}(\rho) y$  implies that there is a chain of length at most  $n - 1$  where  $n = |A|$ : we can remove any loop from the chain (a segment of the form  $x_i \rho x_{i+1} \rho \dots \rho x_{j-1} \rho x_j = x_i$ ).

So we only need to search for chains of length at most  $n - 1$  and this bound is tight.

### Small Example

The relation  $x \rho y \iff y = x + 1 \vee y = x$  on  $[5]$ .



Done after 3 steps:  $\rho^4 = \text{tcl}(\rho) = \text{rtcl}(\rho)$ .

### Applying Truncation

Lets get back to reflexive transitive closure. We have seen that

$$(I \sqcup \rho)^{n-1} = \text{rtcl}(\rho)$$

so we can determine the closure in  $\Theta(n^3 \log n)$  steps.

Since the bound  $n - 1$  is tight it is not entirely clear how to improve on this result (ignoring the possibility of faster matrix multiplication).

Yet there is a clever dynamic programming trick due to Warshall that performs the whole computation in just cubic time, the same as the cost of just one matrix multiplication.

### Warshall's Algorithm

We compute a three-dimensional Boolean matrix  $BB$ .

Slice 0:  $BB[i, j, 0]$  is the reflexive closure of the given relation.

```
for( k = 1; k <= n; k++ )
  for( i = 1; i <= n; i++ )
    for( j = 1; j <= n; j++ )
      BB[i, j, k] = BB[i, j, k-1] || (BB[i, k, k-1] && BB[k, j, k-1] );
```

Upon completion, the last slice  $BB[i, j, n]$  represents the transitive reflexive closure. Takes  $\Theta(n^3)$  steps, constants small.

**Exercise 2.** Prove the correctness of Warshall's algorithm and show how to implement in space  $O(n^2)$ .

### Lowering the Bound

Is there any other way we can potentially speed up the computation of  $\text{rtcl}(\rho)$ ?

Note that

$$(I \sqcup \rho)^k = I \sqcup \rho \sqcup \rho^2 \sqcup \rho^3 \sqcup \dots \sqcup \rho^k$$

and therefore for  $\rho$  reflexive and some  $k$

$$I \subseteq \rho \subseteq \rho^2 \subseteq \rho^3 \subseteq \dots \subseteq \rho^k = \rho^{k+1} = \text{rtcl}(\rho)$$

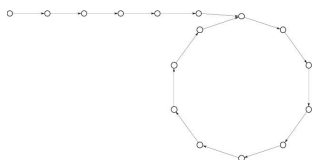
In general we may have  $k = n - 1$ , so computing the whole sequence of relations is not attractive.

But it may happen that the sequence is much shorter.

## The Lasso

This is the usual lasso situation: we are computing the orbit of point  $\rho$  under the operation  $R \mapsto \rho \bullet R$ . Since the carrier set is finite we must have

$$\rho^t = \rho^{t+p} \quad \text{where} \quad 0 \leq t, 1 \leq p$$



Because of monotonicity we must have  $p = 1$ : the orbit ends in a fixed point.

## Modified Lasso

Hence we can compute the reflexive transitive closure like this:

```
rhok = rho = rho + I;
while( rhok * rho != rhok ) {
    rhok = rhok * rho; // relational composition
}
return rhok;
```

In the worst case there will be  $n - 1$  rounds, but we may have far fewer rounds. Still, using matrix multiplication to implement composition we have worst case running time  $O(n^4)$  steps – no match for Warshall.

However, this algorithm is interesting from another perspective.

## The Algebra Underneath

It is worth taking a closer look at the last method from an algebraic point of view.

As mentioned previously, we are dealing with a monoid

$$\langle \text{Rel}_A, \bullet, I_A \rangle$$

consisting of all binary relations on  $A$ , with composition as operation and the identity relation as the neutral element.

We are computing the least submonoid containing  $\rho$ , also called the submonoid *generated* by  $\rho$ .

Computing subalgebras generated by some elements is a standard problem in computational algebra.

## Single Pair

## Single Queries

Consider a single query of the form

**Problem: Single Pair**

Instance: A relation  $\rho$  and two points  $a$  and  $b$ .

Question: Is  $a \text{ rtcl}(\rho) b$ ?

Of course we can compute the whole closure and then check if  $(a, b)$  lies in it, but this approach is clearly overkill. We compute much more information than we are required to.

Is there a method that answers only this one query, without producing the whole closure?

## Graphs to the Rescue

The answer is yes, we can solve a single query in time quadratic in  $n$ , the size of the underlying set.

Think of the associated digraph  $G$  and use a standard graph exploration algorithm such as DFS or BFS to check if there is a path in the graph from  $a$  to  $b$ .

**Theorem 2.** *One can solve the Single Pair problem in quadratic time.*

More precisely, the running time is  $O(n + |\rho|)$ . Also, graph exploration really solves the Single Source problem: we determine all points reachable from a fixed starting point. So if  $\rho$  is sparse repeated runs of the Single Source algorithm may be faster than Warshall.

Note that DFS and BFS also require linear space and it seems unlikely that there are better algorithms (say, using only logarithmic space).

## Summary

- Static equivalence relations can be represented efficiently using kernel functions.
- Works well in particular for computing meets.
- Boolean matrices only for very dense relations.
- Computing transitive closure can be handled using the lasso algorithm.
- Warshall's algorithm computes the transitive closure in cubic time.