Efficient Algorithms For Coherent Configurations

Algebraic and Extremal Graph Theory
University of Delaware

Sven Reichard

TU Dresden

2017-08-10

Outline

Models of computation

Coherent configurations

Implementations

Improvements

- There are different models of computation.
- Most common:
  - Central processing unit
  - Random memory with uniform access
  - Program and data stored in the same memory.
- Other model: Turing machine.

- More realistic model of current hardware:
  - Network of processing units
  - Each processing unit can process several pieces of data at a time
  - Varying distances between the units
  - Hierarchy of memory modules of increasing size and latency.
  - Parts of the memory may be exclusive to (groups of) processors

- These models allow us to investigate the complexity of algorithms.
- The different models are equivalent in the following sense:
- Bounds on the complexity of a given algorithm in different models are the same up to a constant factor.
- So if we are interested in the asymptotic behaviour we choose the most convenient model.

- If we write and use programs in practice, we have a different point of view.
- Knuth: “The size of the constant does matter.”
- Hence it is good to keep the more realistic model in mind.

- We will look at one particular problem from graph theory.
- Several implementations of the same basic idea.
- The best asymptotic implementation is actually slower for all practical examples.

Outline

- Let Ω be a finite set.
- Let $n = |\Omega|$, let $k \in \mathbb{N}$.
- The group $S(\Omega)$ acts on Ω^k componentwise:

$$g(x_1, \ldots, x_k) = (g(x_1), \ldots, g(x_k))$$

- On the other hand, the group S_k acts on Ω^k.
- For $\sigma \in S_k$ and $x \in \Omega^k$ we have

$$x_\sigma = (x_{\sigma(1)}, \ldots, x_{\sigma(k)}).$$

- The two group actions commute:

$$g(x_\sigma) = g(x)_\sigma.$$

- Let $G \leq S(\Omega)$.
- The orbits of G on Ω^k are the k-orbits of G.
- We get the following properties:
  - If x, y are in the same orbit, then

$$x_i = x_j \Rightarrow y_i = y_j;$$

  - if x, y are in the same orbit, then so are x_σ, y_σ.
- These are the defining properties of a configuration.

- A *configuration consists of a finite set* Ω*, a set of colours* C *and a coloring*
$$c : \Omega^k \to C$$
such that
  - *If for* $x, y \in \Omega^k$ *we have* $c(x) = c(y)$*, then for* $0 \leq i, j < k$ *we have*
$$x_i = x_j \Rightarrow y_i = y_j.$$
  - *For* $\sigma \in S_k$ *and* $x, y \in \Omega^k$ *we have*
$$c(x) = c(y) \Rightarrow c(x_\sigma) = c(y_\sigma).$$
  This gives an action of S_k *on* C*.*

Automorphisms

- A permutation $\phi \in S(\Omega)$ is an automorphism of $W = (\Omega, c)$ if $c(x) = c(\phi(x))$ for all $x \in \Omega^k$.
- More generally, ϕ is a colour automorphism if it permutes colours.
- In other words, there is a $\psi \in S_k$ such that

$$\psi \circ c = c \circ \phi$$

Dimension 2

A 2-dimensional configuration corresponds to a set $\mathcal{R}$ of binary relations on Ω such that

- the relations partition Ω^2;
- each relation is either reflexive or antireflexive;
- if $R \in \mathcal{R}$, then $R^{-1} \in \mathcal{R}$.

This implies that each relation is either symmetric or antisymmetric.

Substitution

- If $x \in \Omega^k$, $y \in \Omega$, and $0 \leq i < k$, we denote by x_i^y the result of replacing the i-th coordinate of x by y.
- So, $(x_i^y)_i = y$, and $(x_i^y)_j = x_j$ for $i \neq j$.

WL refinement

- A configuration c' is a refinement of a configuration c if for $x, y \in \Omega^k$, $c'(x) = c'(y)$ implies $c(x) = c(y)$.
- Given a configuration c we define its WL-refinement as follows:

$$c'(x) = (c(x), [(c(x_1^y), \ldots, c(x_k^y)) \mid y \in \Omega])$$

  Here, the second component is a *multiset* of vectors obtained by picking a point y and substituting it for all components of x in turn.
- Since $c(x)$ appears as the first component of $c'(x)$, the latter is in fact a refinement.
- We get that $Aut(c) = Aut(c')$.

Coherent configurations

- A configuration is coherent, if c' is equivalent to c.
- Any configuration c has a unique coarsest coherent refinement, its coherent closure $\langle\langle c \rangle\rangle$.
- $$Aut(c) = Aut\left(\langle\langle c \rangle\rangle\right)$$
- The procedure of finding the coherent closure by successive refinement is known as the k-dimensional Weisfeiler-Leman algorithm (WL_k).

Reformulation WL_2: Graphs

- Given an edge-colouring of a complete graph.
- Given an edge (x,y) of colour k, and two colours i, j.
- Count the number of points z such that $c(x, z) = i$, $c(z, y) = j$.
- Use these counts to distinguish edges of colour k.
- When no new colours appear we have a coherent graph.

Reformulation WL_2: Matrices

- A two-dimensional configuration is basically a matrix.
- Replace all distinct entry values by non-commuting indeterminates.
- Replace the matrix by its square.
- Repeat as long as the number of distinct entries grows.
- This is Weisfeiler's original formulation.
- Can be extended to higher dimensions by defining an appropriate product of tensors.

Outline

Previous implementations

- Two implementations of WL_2 were described in a 1990's paper (Babel et al): a "Russian" and a "German" program
- Focus on practical vs. theoretical complexity.
- Input of size n^2.
- The German algorithm has a running time of $O(n^3 \log n)$ and a space requirement of $O(n^3)$.
- The Russian algorithm has a running time bounded by $O(n^6)$ and a space requirement $O(n^2)$.
- The latter is faster for most practical instances

Outline

We will describe a few improvements to these classical implementations.

Using values instead of polynomials

- During the algorithm we need to compute a matrix product.
- The actual values of the entries is relevant only for the determination of structure constants; during the stabilization we are interested only in the classes of equal entries.
- The entries are dot products of the form $f = \sum_{k=1}^{n} X_{i_k} X_{j_k}$, where the X_i are non-commuting indeterminates over the integers.
- Computation in this ring can become expensive, in the sense that basic operations such as comparison, addition and multiplication cannot be done in constant time.

- To distinguish two expressions it is sufficient to find a point where they evaluate differently.
- For a ring R and $x, y \in R^r$, let $f(x, y) = \sum_k x_{i_k} y_{j_k}$.
- Then the matrix product can be computed in R.
- However it is possible that we fail to distinguish some expressions.

- Let R be a ring, $U = R^*$ the set of units.
- Let $f \neq 0$ be an "expression" with small coefficients:

$$f = \sum \alpha_{ij} x_i y_j$$

- Let $x, y \in U^r$ be uniformly distributed.
- Then $f(x, y) \neq 0$ with high probability.

It follows that if $f(x, y) = g(x, y)$, then $f = g$ with high probability.
For ease of implementation we choose $R = \mathbb{Z}_q$, $q = 2^{32}$.

Fast matrix multiplication

- The problem is reduced to $n \times n$ matrix multiplication over the integers mod p.
- The naive algorithm uses $O(n^3)$ operations.
- A lower bound is $O(n^2)$.
- The fastest known methods have an exponent of about 2.35. However, those become worthwhile only for very large n.
- Strassen's method uses the fact that 2×2 products can be computed with seven multiplications (instead of eight).
- This gives an exponent of $\log_2 7 = 2.81$.

- So far, fast matrix multiplication has not led to practical improvement.
- One reason is that we do not get good bounds on the number of iterations.
- Following an idea of Babel's, we take a different approach.

Reusing results

- We can give a bound on the number of times each triangle is considered.
- If there are m new colors in one iteration we can choose the recoloring in such a way that at least n/m arcs retain their color.
- An ordered triangle (x, y, z) contributes to the product (x, z).
- If the color of the arc (x, y) is changed from i to i', the product has to be recomputed.
- At most half of the arcs of colour i is recoloured to i'.
- Hence each arc is recoloured at most $\log_2(n^2)$ times (very rough estimate).
- Each arc contributes to $2n$ products, so we need to perform $4n \log_2 n$ updates of products.
- If we keep all products in memory we do not need essentially more memory to perform the updates.

Memory layout

- The algorithm is not very compute intensive.
- Memory access actually dominates it.
- Hence it it beneficial to optimize memory access patterns.

- We basically need to compute a matrix product.
- Each individual product involves a row and a column of the matrix.
- If we store the matrix row by row, the elements of one row are located close together.
- However, the entries of a column are spread out.
- This leads to bad cache usage.

- The usual way around this is an alternative storage pattern.
- For example z-order.
- Complicated to implement for general n.
- Another way around the cache problem is theory.

Lemma

The algorithm remains correct if the square M^2 is replaced by $M\dot{M}^T$.

Proof.

Since after preprocessing we always have a configuration we get that

$$M_{ij} = M_{kl}$$

if and only if

$$M_{ji} = M_{lk}.$$

It follows that two row-column products are equal only if the corresponding row-row-columns are equal.
And we are only interested in equality/inequality of the entries of the product. □

- The previous lemma allows us to replace the columns in the algorithms with rows.
- This alone led to a five-fold speedup, highlighting the importance of memory access.

Parallelization

- We need to compute inner products over the ring of integers mod 2^{32}.
- Common processors can compute several (4-8) integer products simultaneously.
- The various inner products are independent and can be computed by different cores.
- It remains to be seen if parallelization across CPU's is worth the communication overhead.

Algorithm outline

- The input is given in the matrix M.
- Preprocessing to distinguish the diagonal and make the algebra symmetric.
- Select random numbers x_i, y_i, where i runs through all colors.
- Compute the product $P = M(x) * M(y)$ over R; use fast multiplication.
- Repeat the following until no new colors appear.
  - Collect the set of pairs $(M[x][y], P[x][y])$, for $x, y \in \Omega$
  - Decide for each orignal color which class of arcs will retain that color.
  - Extend x and y by adding additional values for all new colors.
  - For each arc (x, z) that changes its color from i to i'
  - Update all relevant products

Benchmarks

- The three algorithms were tested on three classes of examples
  - A finite set of small chemical compounds.
  - Benzene stacks.
  - Möbius ladders.
- These may not be the best test cases, for various reasons.
- However, the latter two give examples with known results which are arbitrarily scalable.

Benchmarks

Möbius ladders

order	RU	DE	S
200	3	2	0.3
400			2
800			15
1600			127

Benchmarks

Benzene stacks

order	RU	DE	S
60	0	2	0
150	2	35	0
198	6		0
300			1
600			7
1200			67