

Scaling the *MCL* algorithm

The complexity of the *MCL* algorithm, if nothing special is done, is $\mathcal{O}(N^3)$ where N is the number of nodes of the input graph. The factor N^3 corresponds to the cost of one matrix multiplication on two matrices of dimension N . The inflation step can be done in $\mathcal{O}(N^2)$ time. I will leave the issue aside here of how many steps are required before the algorithm converges to a doubly idempotent matrix. In practice, this number lies typically somewhere between 10 and 100, but only a small number of steps (in a corresponding range of approximately 3 to 10) in the beginning correspond with matrix iterands that are not extremely sparse. The only way to cut down the complexity of the algorithm is to keep the matrices sparse. Fortunately, the *MCL* process is by its very nature susceptible to such modification. This issue is discussed in the first of the two sections in this chapter. The last section contains a brief description of the implementation with which the experiments in this thesis were carried out.

11.1 Complexity and scalability

The limits of an *MCL* process are in general extremely sparse. All current evidence suggests that overlap or attractor systems of cardinality greater than one correspond with certain automorphisms of the input graph (Sections 10.1 and 10.2 in the previous chapter).

The working of the *MCL* process with respect to finding cluster structure is mainly based on two phenomena. First, the disappearance of flow on edges between sparsely connected dense regions, in particular the edges in the input graph. Second, the creation of new flow within dense regions, corresponding with edges in the limit graph not existing in the input graph.

Typically, the average number of nonzero elements in a column of a limit matrix is equal to or very close to one, and the intermediate iterands are sparse *in a weighted sense*. The expansion operator causes successive iterands to fill very rapidly, but if natural cluster structure is present and the cluster diameters are not too large (cf. Section 10.6) then the inflation operator ensures that the majority of the matrix entries stays very small, and that for each column the deviation in the size of its entries is large. A small cluster diameter implies that the equalizing of probability distributions is relatively easy as flow need not be transferred over long distances before it eventually stabilizes. This fact is exploited in various proposals for matrix pruning schemes made below.

REMARK. Before introducing these schemes a remark on the justification of pruning is in place. I will not attempt a numerical or perturbation analysis of pruning. Rather, I will stick to heuristic reasoning in higher-level terms of cluster structure and random walks when discussing the viability of pruning, and this reasoning will be put to the test by experimenting with randomly generated testgraphs in the next chapter.

11.1.1 Pruning schemes. If it is assumed that the probabilities of intermediate random walks are indeed distributed inhomogeneously per column, then this leads naturally to the idea that it will do no harm to remove intermediate random walks (i.e. setting matrix entries to zero) which have very small probability. The interpretation of the process then enforces obvious constraints on such pruning:

- The magnitude of a transition probability is only relevant in relationship to the other transition probabilities of the associated tail node. Pruning must be done locally rather than globally, that is, column-wise.
- Pruning should only remove a small part of the overall weight of a column; the corresponding entries should ideally have large (downward) deviation from the column average (for a suitable notion of column average).
- In order to maintain the stochastic interpretation, columns are rescaled after pruning.

Together these form the the key to an efficient implementation of the *MCL* algorithm. Three different pruning schemes have been considered and implemented. Let M be a sparse column stochastic matrix. Suppose a column c of the square M^2 has been computed with full precision. The three schemes are respectively:

- Exact pruning — the k largest entries of the column are computed. Ties are broken arbitrarily or are allowed to increase the bound k . This computation becomes increasingly expensive for larger values of k and increasing deviation between k and the number of nonzero entries of c .
- Threshold pruning — a threshold value f is computed in terms of the mass centre $\text{ctr}(c)$ of order two of c . All values greater than f are kept, the rest is discarded. A typical candidate for such a threshold value is of the form $a \text{ctr}(c)(1 - b[\max_i(c_i) - \text{ctr}(c)])$, where $0 < a \leq 1$ and b is chosen in the range $1 \dots 8$; another one is $a[\text{ctr}(c)]^b$, where $0 < a \leq 1 \leq b$. The motivation for the first depends on the fact that if $\max_i(c_i)$ is close to $\text{ctr}(c)$ then the (large) nonzero entries of the vector c are rather homogeneously distributed.
- A combination of the above, where threshold pruning is applied first in order to lower the cost of exact pruning. It is either allowed or disallowed for threshold pruning to leave a number of nonzero entries smaller than k .

If pruning with pruning constant k is incorporated into the algorithm, the complexity is reduced to $\mathcal{O}(Nk^2)$ for a single matrix multiplication. This follows from the fact that any columns of the product of two k -pruned matrices has at most k^2 nonzero entries. It is assumed that pruning can be done in $\mathcal{O}(t)$ time for a vector with t nonzero entries. In the experiments in the next chapter this was ensured by using threshold pruning.

11.1.2 Factors affecting the viability of pruning. It is intuitively acceptable that pruning eats away the least probable walks, if they have large downward deviation from the column centre, and if the total number of pruned entries accounts for a relatively small percentage of the column mass, say somewhere in between 5 and 10 percent. If the distribution of a column c is rather homogeneous, with many entries approximately equal to the centre $\text{ctr}(c)$, and if pruning removes a sizeable fraction of the distribution, this will clearly disturb the *MCL* algorithm, rather than perturb. The examples in the previous chapter indicate that the latter will be the case if the diameter of the natural clusters is large. Those examples turned out to vex the the *MCL* algorithm in a much more fundamental way however. In the next chapter I report on experiments using randomly generated testgraphs in which both the graphs themselves and the natural clusters have in general small diameter, and on how the algorithm scales when scaling these dimensions.

11.1.3 Convergence in the presence of pruning. The convergence properties in the setting sketched above do not change noticeably, and the resulting clusterings are still very satisfactory. Clusterings of graphs with up to a thousand nodes resulting from both normal matrix computation and prune mode with otherwise identical parametrizations were compared. The respective clusterings sometimes differed slightly (e.g. a node moving from one cluster to another) and were often identical. The effect of varying the pruning parameter is investigated quantitatively in the following chapter in terms of performance criteria.

11.2 MCL implementation

The *MCL* algorithm was implemented at the *CWI* by the author. It is part of a library written in C with extensive support for matrix operations, mapping of matrices onto clusterings, comparison of clusterings, generation of statistics (e.g. for different pruning schemes), and facilities for random generation of partitions and cluster test matrices. Both Jan van der Steen and Annius Groenink have contributed significantly to the matrix section of the library in terms of rigor and elegance. The library will be made available under a public license in due course.

At the heart of the library lies the data structure implementing a matrix. A matrix is represented as an ordered array of vectors, and a vector is represented as an array of index/value pairs. Each index is unique in the array, and the index/value pairs are ordered on increasing index. This generic construction is used to represent a nonnegative vector by its positive entries only. The vector $(4.2, 0.0, 2.7, 3.1, 0.0, 0.0, 5.6)^T$ is thus represented as the array (indexing starts at zero)

[0|4.2][2|2.7][3|3.1][6|5.6]

There is a choice of representing a matrix via its rows or its columns. A column stochastic matrix M is naturally represented via its columns. Assuming that pruning is applied with pruning constant k , computing the square M^2 requires for each column of M^2 the computation of a weighted sum of at most k columns, resulting in a vector which may have k^2 entries. This vector is pruned down to at most k entries via either of the schemes

given in the previous section. For large k , say larger than 70, it is pertinent that threshold pruning is applied in order to ease the burden of exact pruning. This may lead to a pruned vector with less than k entries. It is easy to envision a looping process in which several thresholds are tried in order to obtain an optimum threshold value resulting in a vector with a number of entries close or even to k , or even a version of threshold pruning where the pruning regime depends on the weight distribution of the probability vector, so that nodes with a large homogeneous distribution are allowed to have more than k nodes. This was not tried for, but the experiments in Chapter 12 indicate that fine-tuning the pruning regime may result in considerably better performance.