ПЛОВДИВСКИ УНИВЕРСИТЕТ "ПАИСИЙ ХИЛЕНДАРСКИ", БЪЛГАРИЯ
НАУЧНИ ТРУДОВЕ, ТОМ 36, КН. 3, 2009 – МАТЕМАТИКА
PLOVDIV UNIVERSITY "PAISSII HILENDARSKI", BULGARIA
SCIENTIFIC WORKS, VOL. 36, BOOK 3, 2009 – MATHEMATICS

A REFLEXIVE ALGORITHM FOR EXISTENCE OF NULL-SUBMATRICES

Dobromir P. Kralchev, Dimcho S. Dimov, Alexander P. Penev

Abstract. We propose a heuristic algorithm for existence of null-submatrices in big sparse matrices. The algorithm is reflexive: it examines its own memory consumption, which is in correlation with its output.

Mathematics Subject Classification 2000: Primary 68T20, Secondary 60G35, 68R05

Key words: binary matrices, matchings, heuristics, random processes, reflexive (self-monitoring) algorithms

1. Introduction

Given an M x N binary matrix and a positive integer $G \geq 2$, can you find a P x Q null-submatrix such that $P + Q = G$?

This problem is connected with many other combinatorial problems. For instance, searching an unoriented graph for cliques is equivalent to searching $A = (a_{ij})$ for a (symmetrically positioned) null-submatrix, where $a_{ij} = 0$ if and only if the vertices i and j are connected or $i = j$.

Consider also the matching problem. You have a bipartite unoriented graph with $M + N$ vertices. Given a positive integer K, can you choose K arcs such that no two of their vertices coincide? A *matching* is any set of arcs that has this property. In the special case $M = N = K$ the matching is *perfect*. An algorithm for finding perfect matchings is given in [1]. Many real-world problems can be reduced to this one: assigning jobs to workers or classrooms to teachers, etc. A theorem belonging to F. Hall (cf. [2]) can be used to prove the following proposition:

There is a perfect matching in an $N + N$ bipartite graph iff its adjacency matrix does not contain a $P \times Q$ null-submatrix such that $P + Q = N$.

Our problem has three parts:

a) Does at least one null-submatrix exist?

b) How many null-submatrices exist?

c) Find at least one null-submatrix, if any.

At first sight, the third part seems most interesting. On the contrary, we shall pay attention to the first one, which is especially important for this reason: many problems are OR-compositions of null-matrix subproblems and can be solved following the next schemes:

— examine the subproblems one by one trying to find a null-submatrix until you find one or there are no subproblems left;

— examine the subproblems one by one: if a null-submatrix exists, then find it and stop searching, else go to the next subproblem.

If negative answers are more probable, the second scheme is faster. That is why, it is important to have a quick algorithm for existence of null-submatrices.

Moreover, in many problems the search can be implemented through backtracking; then the existence algorithm ensures a correct decision at each step, which accelerates the search.

2. Construction of the algorithm

2.1. Analysis of old algorithms

We shall accelerate the algorithm from [3]. (In fact, [3] does not contain an explicit formulation of the algorithm, but the theorems managing different cases are arranged in the same way as the steps of the algorithm. An explicit formulation of the algorithm is given in [4] and [5].)

Strictly speaking, the algorithm from [3] is not an existence algorithm: it *searches* for a null-submatrix. However, the heuristic we use makes it an existence algorithm.

Searching for a null-submatrix, the algorithm checks each zero for a possible participation in such a submatrix. To do so, the algorithm explores some submatrices (according to the position of the zero being checked) and makes recursive calls when necessary.

The details of the algorithm are unimportant for the current study. You can find them in [3], [4] and [5].

The average running-time of the algorithm is about half a second even for 1000 x 1000 matrices. However, for a special case of sparse matrices

(N x N binary matrices, $110 \leq N \leq 140$, density: 10%) and for special positions of the submatrix the running-time is unacceptable — more than 30 min. The *density* is the percentage of the units of the matrix. (The tests were run on Pentium III, 450 MHz, 128 MB RAM.)

A powerful idea applicable in difficult combinatorial problems is to examine the actions of the implementation of the algorithm itself. It is often the case that some easily recognizable feature (such as running-time or memory consumption) is in correlation with the output. Such a feature can be used to predict the answer long before the algorithm terminates normally. If the correlation is strong enough, the execution of the algorithm can be interrupted thus saving running-time.

In our problem it is suitable to examine the memory consumption of the algorithm. Consider the memory consumption as a random process $(M_t)_{t \geq 0}$. The memory consumption can be measured in bytes but we shall use the recursion depth as a measurement unit (it is a non-negative integer proportional to the actual consumption measured in bytes).

Sparse binary matrices (density: 10%) of both kinds (containing a null-submatrix or not) were generated at random, then passed to the program with $M = N = G$. The average recursion depth D for each test was calculated according to the formula

$$D = \frac{1}{T - k + 1} \sum_{t=k}^{T} M_t \,. \tag{1}$$

Here T is the moment when the program was interrupted, i.e. we have the random process $(M_t)_{t=0}^{T}$ instead of $(M_t)_{t \geq 0}$. The integer k stands for the starting moment of calculation. It is greater than zero because M_t increases in the beginning, then starts fluctuating (the value k is intended for filtering out the initial increment).

Typical values of the average recursion depth are shown below.

Size	110	120	130	140	$\overline{x}$	s
'Yes'	7.9	7.8	7.5	7.7	7.7	0.2
'No'	8.6	8.2	9.7	8.9	8.9	0.6

Table 1. Average recursion depth

The size is the common value of M, N and G. 'Yes' means there is a null-submatrix, 'No' means there is not.

Obviously, the average depth is smaller for matrices containing a null-submatrix. This hypothesis has been confirmed with a p-value about 0.005.

The column marked with $\overline{x}$ contains the average recursion depth for all sizes. The column marked with s contains the standard deviation. To choose some boundary, divide the interval between 7.7 and 8.9 in a ratio 0.2 : 0.6 and you will get the value 8.0. This number means that with a great certainty the average recursion depth will be smaller than 8.0 if there is a null-submatrix; otherwise, it will be greater.

The algorithm can be made faster through implementing a new level examining the memory consumption of the algorithm. A new algorithm is thus obtained consisting of two levels: the lower level is the old algorithm, the higher level is a monitor that can stop the low-level algorithm whenever the average recursion depth becomes much smaller or much greater than 8.0. The levels are united by a common goal, so they form a single self-monitoring algorithm, hence the name 'reflexive'.

2.2. A reflexive algorithm for null-submatrices

```
Input:
  A: an M x N binary matrix;
  G: an integer greater than 1.
Question: Is there a P x Q null-submatrix B of A with P + Q = G?
Output: 'Yes' or 'No' — the answer to the question.
Const V: the 'yes-no' boundary.
Actions (of the higher level):
1. Run the low-level algorithm that searches A for B.
2. Wait until the recursion depth stops increasing monotonously.
3. Wait until the sample path becomes long enough.
4. Start monitoring the average recursion depth D:
   each time the recursion depth changes,
   calculate D according to formula (1).
5. Wait until the lower level has finished or D << V, or D >> V.
6. If the lower level has finished, return its answer.
7. If D << V, return 'Yes'.
8. If D >> V, return 'No'.
```

The details of the algorithm need some explanation.

The constant V has the value 8.0 calculated above (in fact, the value 7.975 was used but this is just a matter of precision).

Step 1 makes some initializations ($T = 0,\ M_0 = 0$) and starts monitoring the memory consumption. Each time the recursion depth changes, T takes on the next value $(1, 2, 3, \dots)$, then a new element M_T is appended to the array M_t .

Step 2 waits for $T > 0$ such that $M_T < M_{T-1}$, and sets $k = T$.

Step 3 waits until T becomes much greater than k. To be more precise, our implementation of the algorithm waits until $T = k + 99$, i.e. the sample path (the array M_t) has got at least 100 elements with indices $\geq k$.

Step 4 applies formula (1) to the array M_t each time T has increased, i.e. a new element M_T has been appended to the array.

Step 6 could return B instead of 'Yes', thus avoiding a subsequent search.

The inequalities $D \ll V$ and $D \gg V$ in steps 5, 7 and 8 can be made more precise by specifying a critical point. In our implementation the inequality $D \gg V$ is replaced with $D > V + Z_\gamma \cdot \frac{s}{\sqrt{n}}$ $\Big($and $D \ll V$ is replaced with $D < V - Z_\gamma \cdot \frac{s}{\sqrt{n}}\Big)$, where Z_γ is a quantile of the standard normal distribution. The confidence level γ must be chosen close to 1. However, there is no point in setting it to a value greater than 0.995 because of the p-value. It is natural to choose for s the greatest of the values 0.2 and 0.6 (see Table 1). Then $Z_\gamma \,.\, s = 0.6\, Z_{0.995} = 1.5$. Now we have $D > V + \frac{1.5}{\sqrt{n}}$ instead of $D \gg V$, and $D < V - \frac{1.5}{\sqrt{n}}$ instead of $D \ll V$. Here n is the length of the sample path, i.e. $n = T - k + 1$.

Note: Implementation details could be different from this description. For example, the array M_t is not necessary: step 2 needs only M_T and M_{T-1}; step 4 needs only M_T and the old value of D to calculate the new value of D.

3. Experimental results

The implementation of the reflexive algorithm was tested for reliability. Sparse matrices (density: 10%) were again generated at random, then passed to the program with $110 \leq M = N = G \leq 140$. These are some conditional probabilities calculated from the test results:

P(the output is 'Yes' | the correct answer is 'Yes') = 95.0%;

P(the output is 'No' | the correct answer is 'No') = 62.5%;

P(the correct answer is 'Yes' | the output is 'Yes') = 71.7%;

P(the correct answer is 'No' | the output is 'No') = 92.6%.

The reliability of the algorithm was estimated as follows:

$$P(\text{the output is correct}) = 78.75\%.$$

The average running-time of the reflexive algorithm is 20-30 sec., which is incomparably better than the running-time of the lower level without the higher one.

4. Possible improvements

A disadvantage of the reflexive algorithm described above is the fact that all the parameters necessary for its work are predetermined. Consequently, one must repeat the whole statistical analysis whenever the density or the size of the matrices goes out of the range investigated. To avoid this, a third level could be embedded that will analyse the running-time and reliability of the algorithm and will automatically configure its parameters.

The algorithm can be made even faster (and less reliable) if it does not wait for D to become much smaller or much greater than V but rather calculates the probabilities of these events. If one of these probabilities becomes close to 1, the monitoring level can stop the lower level and return the corresponding answer as if the event has happened. However, to implement this, one must know the distribution of the random process $(M_t)_{t \geq 0}$.

5. Conclusion

Reflexive algorithms are suitable when the output is in correlation with some easily recognizable feature of the algorithm. Memory consumption can be used in this part, but it is not the only feature of this kind. Other characteristics, such as running-time, may also be useful.

6. Acknoledgement

This research has been partially supported by the Bulgarian NSF under Contract No VU-MI 107/2005 and by project No IS-M-4/2008 of Department for Scientific Research, Plovdiv University "Paisii Hilendarski".

References

[1] Preslav Nakov, Fundamentals of computer algorithms, TopTeamCo, Sofia, 2001, ISBN: 954-8905-04-3, pp. 175–179 (in Bulgarian).

[2] V. Lipskii, Combinatorics for programmers, Moscow, "Mir", 1988 (in Russian).

[3] Dimcho Dimov, Dobromir Kralchev, Alexander Penev, Stanimir Stanchev, Existence of solutions to the assignment problem, *International Conference on Automatics and Informatics*, Sofia, May 30 – June 2, 2001, pp. I-81 – I-83.

[4] Dobromir Kralchev, Investigation of the rook problem, BSc diploma work, Plovdiv University "Paissii Hilendarski", Department of Mathematics and Informatics, Plovdiv, Bulgaria, 2001 (in Bulgarian).

[5] Dobromir Kralchev, Existence, generation and count of the assignments in the rook problem, MSc diploma work, Plovdiv University "Paissii Hilendarski", Dept. of Mathematics and Informatics, Plovdiv, Bulgaria, 2003 (in Bulgarian).

Dobromir P. Kralchev
University of Food Technologies
Dept. of Informatics & Statistics
26 Maritsa Str.
4000 Plovdiv, Bulgaria
dobromir_kralchev@abv.bg

Dimcho S. Dimov, Alexander P. Penev
Plovdiv University "Paissii Hilendarski"
Dept. of Mathematics and Informatics
236 Bulgaria Blvd.
4000 Plovdiv, Bulgaria
apenev@uni-plovdiv.bg

Received 06 January 2008

РЕФЛЕКСИВЕН АЛГОРИТЪМ ЗА СЪЩЕСТВУВАНЕ НА НУЛЕВИ ПОДМАТРИЦИ

Добромир Кралчев, Димчо Димов, Александър Пенев

Резюме. Предлагаме евристичен алгоритъм за съществуване на нулеви подматрици в големи разредени матрици. Алгоритъмът е рефлексивен: изследва количеството памет, използвано от самия него, а то се намира във взаимна връзка с изхода.