An approach to algorithmizing the problem of vertex minimization of nondeterministic automata. Part I. Problem statement and the brief description of the basis methods

M E Abramyan 1 and B F Melnikov 2

1 Southern Federal University, 105/42 Bolshaya Sadovaya str., 344006, Russia
2 Shenzhen MSU – BIT University, 1 International University Park Road, Dayun New Town, Longgang District, Shenzhen, Guangdong Province, 518172, China

E-mail: bormel@mail.ru

Abstract. We consider our approach to the problem of vertex minimization of nondeterministic finite automata. It comes down to constructing a lot of grids and knocking out on its basis an optimal (or pseudo-optimal) subset. However, our problem is much more complicated than the classical problem of optimal coverage of a set, since we need to succeed (to build pseudo-optimal solutions in an acceptable time) in its particular case; in this particular case, we are considering has many specific features, which are not characteristic of most other variants of the problem of optimal coverage of a set. In the first part of the paper, we consider issues related to the brief description of the basic branch-and-bound method, testing approach to the basic method, and some modifications the basic algorithm.

1. Introduction

This work continues the study of the algorithm for solving the problem of vertex minimization of nondeterministic finite automata (NFA), based on the branch-and-bound method (BBM).

We consider our approach to the problem of vertex minimization of nondeterministic finite automata. It comes down to constructing a lot of grids and knocking out on its basis an optimal (or pseudo-optimal) subset. However, we shall not consider issues related to so called Waterloo automaton [1, 2, 3], since, according to the authors of this paper, the basic algorithmic questions of minimizing automata do not apply to Waterloo-like automata (so called “Walibad”, from “Wa(terloo-)Li(ke) Bad(ness)”, see [2]).

However, despite our remark about such automata, the considered problem is much more complicated than the classical problem of optimal coverage of a set [4, 5, 6], with the help of which, of course, all this can be solved in principle. This is because we need to succeed (to build pseudo-optimal solutions in an acceptable time) in its particular case. However, “the devil is in the details!”: in this case, we are considering has many specific features, which are not characteristic of most other variants of the problem of optimal coverage of a set.

In Part I of the paper, we consider issues related to the brief description of the basic branch-and-bound method, testing approach to the basic method, and some modifications the basic algorithm.
2. Problem statement and brief description of the basic branch-and-bound method

We start with the statement of the problem, immediately formulating it in matrix terms (the problem is discussed in more detail in [3, 7, 8]).

We consider a matrix with logical values (true and false, or 1 and 0), not containing zero and matching rows and columns. We shall call a grid a set of rows and columns of a given matrix, at the intersection of which only units are contained. We emphasize that a grid can include non-adjacent rows and / or columns of the original matrix. A grid is called complete if it cannot be expanded by adding a new row or column. If the set of full grids includes all the single elements of the original matrix, then we shall call such a set the coating of the matrix, and consider the number of full grids, i.e. the size of the coating.

The problem is to find the coverage of the initial matrix having a minimum size.

The algorithm for solving the problem is based on the branch-and-bound method [8, 9, 10, 11] of and is implemented in C# for the .NET Framework. A detailed description of the algorithm, which is accompanied by a review of the corresponding C# classes, is given in [8, 12]. In this paper, additional heuristics are considered that make it possible to improve the properties of the basic algorithm. However, first we describe the basic algorithm itself.

Its first component is the MakeGridRnd() method, which is designed to generate a full grid of the original matrix containing a unit element with a fixed row and column number. (Note once again, that brief descriptions of some types and methods were given in [8]; here, we shall not repeat these descriptions.) After including the corresponding row and column in the grid under construction, a cycle is launched in which an attempt is made to expand the grid by adding a new row or column. The search for a suitable line begins with a line with a randomly selected number, after which all lines are sorted in cycles until the desired line is found or all lines are analyzed. The columns of the matrix are processed in the same way, while processing rows and columns alternates. Thus, the implemented full grid search algorithm is randomized and is determined by three initial parameters: the numbers of the initial row and column included in the grid, and the random number sensor (an object of type Random).

The main part of the basic algorithm is the creation and processing of a sequence of subtasks, each of which is a set of complete grids for the original matrix (which, possibly, is not yet a matrix covering). Along with the construction of a set of subtasks, subtasks, a set of full grids is also being built, from which elements are taken to build new subtasks based on existing ones.

In accordance with the main idea of BBM, two sets of full grids are associated with each subtask: the first set (Yes) contains the grids included in this subtask, the second set (No) contains grids that cannot be included in this subtask and in the subtasks built on its basis. Accounting for the set No allows avoiding duplication of subtasks in the process of their generation.

New subtasks are created by splitting the existing subtask into two new ones. This action is the main step of the basic algorithm and is implemented as the MainStep() method. The splitting is performed based on a new full grid, which thus plays the role of a separating element of BBM. The separating element is selected from the current set of full grids so that it contains the maximum and nonzero number of new (i.e., not yet contained in the original subtask) unit elements of the original matrix. For the first of the two created subtasks, the selected grid is included in the Yes set, that is, it is connected to the set of full grids associated with this subtask, for the second new subtask, this grid is included in the No set, that is, it is excluded from the associated her a set of full grids. If it was not possible to select the separating element in the existing grid set, then the Substep() helper method is launched, in which an attempt is made to obtain the separating element by constructing a new full grid (using the MakeGridRnd() method). If this attempt is successful, the above splitting of the original subtask into two new ones is performed and the created grid is included in the grids set, but if the separating element fails, the original task is excluded from consideration (i.e., we have cutting).

Upon successful splitting of the initial subtask, the second of them (for which the separating element was included in set No) is returned to the subtasks set of subtasks waiting for processing, and for the first subtask (the Yes set of which increased by adding a separating element) is checked, whether Yes grid set forms the full coverage of the original matrix. If it forms, then this coverage is analyzed for
minimality and if the coverage is the minimum of the previously obtained, then it is considered as the next pseudo-optimal solution to the original problem. If the Yes set does not yet form a complete coverage, then the subtask is returned to the subtasks set for further processing.

It should be emphasized that even for the initial small-sized matrices, it turns out to be impossible to carry out processing of all arising subtasks in a reasonable time. Therefore, an important aspect of BBM is the method of choosing the next task for processing, which increases the chances of finding the next pseudo-optimal solution. This method is based on the concept of the “weight” of a subtask: at each moment in time, the subtask with the smallest weight is selected from the set of subtasks. As weight components, it is natural to consider the following three characteristics of the subtask:

- the number of single elements of the original matrix, not yet covered with full grids from the Yes set of this subtask;
- the set size of the subtask Yes;
- the set size of the subtask number.

Indeed, the small value of the characteristic (1) means that the subtask can be quickly completed to any solution to the original problem, the small value (2) increases the chances of getting a small solution, and the small value (3) means that there are more options for completing the subtask to the solution. Numerical experiments presented in [8] showed that the best results are achieved if characteristic (3) is used as the weight (or, if all the characteristics are included in the weight, a larger weight coefficient is used for characteristic (3)). Therefore, when calculating the weight of the subtask, the algorithm used one of the following two sets of weighting coefficients: (0, 0, 1) and (1, 1, 10).

3. The testing approach to the basic method

When testing the implemented algorithm, two methods were used: StepRun(steps) and TimeRun(seconds). The StepRun() method executes the number of calls to the MainStep() method specified in the steps parameter. The duration of the TimeRun() method (and thus the number of calls to the MainStep() method) is determined by the seconds parameter as follows: the TimeRun() method stops working if no new pseudo-optimal solutions are found in the specified number of seconds.

The basic algorithm and its modifications were tested on two sets of 100 matrices. The first set contained matrices of size 15 by 25, which were randomly generated by adding 20 initial grids (grids were constructed in such a way that none of them was embedded in another). The second set contained 30 by 40 matrices based on 35 random grids. Thus, for the first set, the size of the optimal solution was estimated from above by a value of 20, and for the second set, by a value of 35. Each of the matrices was processed using the following methods: StepRun(500), StepRun(5000), TimeRun(10). Thus, to find a pseudo-optimal solution, either a fixed number of calls to the MainStep() method (500 or 5000) was used, or the time (10 seconds) was determined, after which the matrix processing was completed if no new pseudo-optimal ones were found during this time solutions.

Table 1. The size of the best pseudo-optimal solution for various modes.

|          | StepRun(500) | StepRun(5000) | TimeRun(10) |
|----------|--------------|---------------|-------------|
| 15 × 25  | 17,38 (14-22)| 15,85 (13-19)| 15,11 (12-18)|
| 15 × 25, set (1, 1, 10)| 20,47 (17-25)| 19,04 (15-24)| 17,93 (14-22)|
| 30 × 40  | 58,04 (43-74)| 38,92 (30-47)| 34,86 (27-46)|
| 30 × 40, set (1, 1, 10)| 54,63 (44-69)| 51,49 (41-66)| 49,03 (40-63)|

Table 2. Matrix processing time (in seconds) for various modes.

|          | StepRun(500) | StepRun(5000) | TimeRun(10) |
|----------|--------------|---------------|-------------|
| 15 × 25  | 0,06 (0,05-0,11)| 0,83 (0,53-1,37)| 12,35 (10,02-22,25)|
| 15 × 25, set (1, 1, 10)| 0,10 (0,05-0,20)| 1,32 (0,51-2,29)| 14,48 (10,02-22,37)|
| 30 × 40  | 0,13 (0,11-0,17)| 2,04 (1,53-2,50)| 18,55 (10,55-29,94)|
The basic version of the algorithm for solving the main problem described Section 2 allows several modifications. To study the effectiveness of such modifications, one could generate, based on the Task class, which contains the implementation of the basic algorithm and derived classes. However, this method makes it difficult to analyze options based on combinations of modifications. Therefore, a different approach was used, in which all possible modifications of the algorithm are introduced into the methods of the original Task class, and its constructor provides the algOptions parameter, which determines the combination of those modifications that should be used at the current start. The algOptions parameter has an enumerated type AlgOptions, equipped with the [Flags] attribute, which allows to use this parameter to specify a set of bit flags.

In the basic version of the algorithm, the subtask was cut off in the only case: if in the Subtask() method for the specified number of attempts it was not possible to form a new grid that could be added to the current subtask. It should be noted that for large matrices a similar situation arises relatively rarely. At the same time, a part arises when, in the process of constructing a subtask, the number of grids added to it (i.e., the size of the Yes set) becomes equal to or larger than the already found size of the pseudo-optimal solution. Obviously, in this situation it does not make sense to analyze this subproblem (and generate new subproblems on its basis). Moreover, you can immediately interrupt the processing of a subtask extracted from the subtasks set if it contains OptSize-1 grid, where OptSize is the size of the pseudo-optimal solution already found. Cutting off the subtask should also be performed in the case when, after adding another grid to it, the size of the Yes set becomes OptSize-1, while the subtask is not yet a solution to the original problem, i.e., it does not determine the coverage of the matrix. Thus, the clipping actions should be performed not only in the Substep() method, but also at the beginning of the MainStep() method (when extracting the next subtask from the subtasks set), as well as at the end of this method when performing the final actions after adding a new grid to the subtask. For the described heuristic, we shall use the notation CutOff in the next parts of this paper.

| Table 3. Matrix processing time (in seconds). |
|---------------------------------------------|
|                           | StepRun(500) | StepRun(5000) |
|---------------------------|--------------|---------------|
| 15 × 25                   | 0.06 (0.05-0.11) | 0.83 (0.53-1.37) |
| 15 × 25, CutOff           | 0.04 (0.03-0.09) | 0.30 (0.25-0.39) |
| 30 × 40                   | 0.13 (0.11-0.17) | 2.04 (1.53-2.50) |
| 30 × 40, CutOff           | 0.12 (0.09-0.16) | 1.09 (0.83-1.39) |
It is natural to expect that using the CutOff heuristic, a greater number of subtasks will be processed, and this will increase the chances of finding the best pseudo-optimal solutions. For the StepRun mode, an acceleration of the algorithm is observed (table 3), and for the TimeRun mode, an increase in the number of processed subtasks (table 4).

**Table 4. Number of completed / catted subtasks.**

| TimeRun(10) |  |
|-------------|---|
| 15 × 25     | 7516 (1281-25559) / 2844 (0-7905) |
| 15 × 25, CutOff | 8 (5-12) / 58804 (28364-122771) |
| 30 × 40     | 2513 (1106-4352) / 49 (0-391) |
| 30 × 40, CutOff | 13 (8-20) / 5688 (2706-23972) |

5. Conclusion

We intend to consider the continuation of this topic in the following parts of this paper. The plans for the next research on this topic are as follows.

We intend to consider various options for choosing a separating element: it is obvious that several heuristics are possible for this. For different variants of the “simple” choice of this element, we shall present statistical data (already partially obtained by us by now) related to the quality of the implementation of the algorithms. We check the quality itself, first, based on the average working time.

In addition, we present our studies related to the random generation of finite automata [13] and the difference that arises for different versions of such generation.

After that, we intend to publish the complication of the “simple” choice of the separating element: based on several implemented algorithms for its generation, we show the possibility of using risk functions [14, 15] for the final selection of this element.

Preliminary calculations show that in this direction it is also possible to obtain a decrease in the running time of the entire algorithm of the branch and bound method.

And, of course, we intend to continue the parallel implementation of the algorithms we describe before. We started such a parallel implementation in [7].

References

[1] Kameda T and Weiner P 1970 On the state minimization of nondeterministic finite automata IEEE Transactions on Computers C-19 (7) 617-27

[2] Melnikov B F 2017 The complete finite automaton International Journal of Open Information Technologies 5 (10) 9-17

[3] Krivolapova A V, Melnikova E A, and Sofonova N V 2016 Some auxiliary algorithms for building Waterloo-like finite automata Bulletin of Voronezh State University. Series: System Analysis and Information Technology 4 20-8

[4] Lund C and Yannakakis M 1994 On the hardness of approximating minimization problems Journal of the ACM 41 (5) 960-81

[5] Cormen T H, Leiserson C E, Rivest R L, and Stein C 2001 Introduction to algorithms (Cambridge, Mass.: MIT Press) pp 1033–8

[6] Hromković J 2004 Algorithmics for Hard Problems. Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics (Berlin: Springer) p 429

[7] Melnikov B and Tsyganov A 2012 The state minimization problem for nondeterministic finite automata: the parallel implementation of the truncated branch and bound method Proceedings – 5th International Symposium on Parallel Architectures, Algorithms and Programming 194-201

[8] Melnikov B and Abramyan M 2019 About some features of the heuristic implementation of branch and bound method for the problem of NFA minimization Numerical algebra with applications – Proceedings of Eighth China-Russia Conference. Southern Federal University 102-6
[9] Melnikov B 2005 Discrete optimization problems – some new heuristic approaches Proceedings – Eighth International Conference on High-Performance Computing in Asia-Pacific Region 73-80

[10] Melnikov B F and Eirich S N 2010 An approach to combining an incomplete branch-and-bound method and a simulation normalization algorithm Bulletin of Voronezh State University. Series: System Analysis and Information Technology 1 35-8

[11] Abramyan M E, Melnikov B F, and Trenina M A 2019 Implementation of the branch-and-bound method for the problem of restoring the matrix of distances between DNA sequences Modern information technologies and IT education 15 (1) 81-91

[12] Melnikov B F, Nichiporchuk A V, Trenina M A, and Abramyan M E 2019 Clustering situations when solving applied optimization problems (by examples of the traveling salesman problem and the task of restoring the distances matrix) International Journal of Open Information Technologies 7 (5) 1-8

[13] Melnikov B F, Pivneva S V, and Rogova O A 2010 Representativeness of randomly generated non-deterministic finite automata in terms of the corresponding basic automata Stochastic optimization in computer science 6 74-82

[14] Melnikov B F and Radionov A N 1998 A choice of strategy in nondeterministic antagonistic games Programming and Computer Software 24 (5) 247-52

[15] Melnikov B F and Pivneva S V 2010 Decision making in applied problems using dynamically selected risk functions Bulletin of the Volga Region Transport 3 (23) 28a-33