Implementation of a strategy improvement algorithm for finite-state parity games

Dominik Schmitz and Jens Vöge*

Lehrstuhl für Informatik VII
RWTH Aachen, Germany

April 2000

Abstract

An implementation of a recent algorithm (Vöge, Jurdzinski, to appear in CAV 2000) for the construction of winning strategies in infinite games is presented. The games under consideration are "finite-state parity games", i.e. games over finite graphs where the winning condition is inherited from ω-automata with parity acceptance. The emphasis of the paper is the development of a user interface which supports the researcher in case studies for algorithms of ω-automata theory. Examples of such case studies are provided which might help in evaluating the (so far open) asymptotic runtime of the presented algorithm.

1 Introduction

The algorithmic approach to automata theory, which has been so successful in the theory of automata over finite words, has not yet produced results of similar strength in the theory of ω-automata, i.e., automata over infinite sequences. Although many intriguing and mathematically powerful constructions exist in ω-automata theory, the high complexity of most algorithms has so far prevented practical applications. Examples of such constructions are the Safra construction for determinizing ω-automata, the complementation of Büchi automata, and the construction of winning strategies in infinite games.

The package OMEGA of algorithms in ω-automata theory, presently under development at RWTH Aachen, should support the researcher in getting experience with the performance of these algorithms and in carrying out case studies. In the present paper we address a problem from ω-automata theory which belongs to the theory of infinite games: the construction of winning strategies in parity games. These games have a direct connection to the model-checking problem for the modal μ-calculus (cf. [EJS93]). One of the central open questions in the field asks whether this model-checking problem or, equivalently, the construction of winning strategies in parity games is possible in polynomial time.

In [VJ00] a new algorithm was developed, which has the potential of being more efficient than previously known procedures (e.g., [EJ91, McN93, Jur00]). The new algorithm follows the idea of strategy improvement as devised by Puri [Pur95], but now in a completely discrete setting rather than over a dense value domain. The algorithm is hard to analyze; indeed, it is open whether the running time of this concrete algorithm can be bounded by a polynomial. Furthermore, the execution of the algorithm is hard to follow since it induces in each step a global change over the given game graph. In this situation, it is helpful to study the algorithm with an implementation and some auxiliary functions which help in assessing the main characteristics of its behaviour.

In the present paper we report on this implementation work, and on the user interface, and we discuss the output of the program in some simple example cases. In this way, we prepare a platform also for model-checking in the full μ-calculus (this issue is, however, not treated in the present work).

*Address: RWTH Aachen, Lehrstuhl für Informatik VII, Ahornstraße 55, D-52056 Aachen, Germany. Email: `schmitz@i7.informatik.rwth-aachen.de`, `voege@informatik.rwth-aachen.de`. Supported by the Deutsche Forschungsgemeinschaft (DFG), project Th 352/5-3.

The paper is structured as follows: We present, in section 2, a short outline of the algorithm (for more detailed pseudocode see [VJ00]) and describe some improvements which make an implementation more efficient. Based on these considerations, we have implemented the algorithm in C.

The focus of this paper is, however, the user's view of the program and the discussion of some case studies. These issues are treated in sections 3 and 4.

In section 3, we first present a Lisp–based input language for describing large game graphs. Secondly, the output format is discussed, including some features which support the user in analyzing the performance of the algorithm. Three essential parameters are isolated: The number of iterations of the main loop of the algorithm, the number of changes of strategy for each vertex, and (regarding the global running time) the number of edge traversals in the search procedures of the algorithm. For the second parameter a two-dimensional output (in matrix format) is developed.

Section 4 illustrates the use of the program. From the case studies considered so far we only discuss here an example taken from [BLV96], where a classical (exponential-time) solution of parity games was given.

2 Parity Games and the Construction of Winning Strategies

A parity game is an infinite game played on a finite colored graph (the "arena" of the game). The game graph is of the form $G = (V_0, V_1, E, c)$. The set V of vertices is the union of the two disjoint sets vertex V_0 and V_1, and we have $E \subseteq (V_0 \times V_1) \cup (V_1 \times V_0)$, i.e., an edge leads from a V_0-vertex to a V_1-vertex or vice versa. We assume $vE \neq \emptyset$ for all vertices v.

A play from vertex v is an infinite sequence $v_0v_1v_2\ldots$ of vertices with $(v_i, v_{i+1}) \in E$ for all $i \geq 0$; one imagines players 0 and 1 moving a token along edges through the graph in a nonterminating sequence of steps. If $v_i \in V_0$ it is the turn of player 0 to decide for an edge leaving v_i, otherwise player 1 has to choose an edge leaving v_i.

The winner of a play $v_0v_1v_2\ldots$ is defined with a reference to the component c of the game graph, which is a function (called coloring) $c : V \to \{0, \ldots, k\}$ for some k. The play $v_0v_1v_2\ldots$ is won by player 0 if in the associated sequence $c(v_0)c(v_1)c(v_2)\ldots$ of colors the maximal color occurring infinitely often is even; otherwise player 1 wins the play.

It is a well-known theorem of the theory of infinite games (see e.g. [EJ91, Tho97]) that parity games (even over infinite game graphs) are "determined" in the following strong sense: From each vertex either player 0 or player 1 can force a win by a *positional winning strategy*. More precisely, the vertex set is partitioned in two *winning regions* W_0, W_1 such that player 0 wins starting from any vertex in W_0 by a fixed choice of outgoing edges for the V_0-vertices in W_0, and analogously player 1 wins starting from any vertex in W_1 by a fixed choice of outgoing edges for the V_1-vertices in W_1. The "solution of the game" is given by the sets W_0, W_1 and corresponding edge sets $E_0 \subseteq (V_0 \cap W_0) \times V_1$ and $E_1 \subseteq (V_1 \cap W_1) \times V_0$. As shown by [EJS93], the problem of solving parity games over finite graphs is in NP $\cap$ co-NP and polynomially equivalent to the model-checking problem for the full modal μ-calculus. All known algorithms [EJ91, McN93, Jur00] have an exponential worst case running time. A key problem of the theory of program verification is to settle the question whether parity games can be solved in polynomial time.

In [VJ00], a new algorithm was developed, based on the idea of strategy improvement (which originates in work on stochastic games and mean pay-off games). The running time of this algorithm can be trivially bounded by the number of possible strategies (say of player 0), which gives an exponential upper bound, but it is an open question whether this can be sharpened to a polynomial bound. This motivates an experimental study of the algorithm as initiated in this paper.

Let us sketch the algorithm of [VJ00]. The key idea is a discrete valuation of plays and a process which successively modifies strategies of player 0 to guarantee higher and higher values of associated plays. To define this valuation of plays, we use a "relevance order" $<$ of the vertex set, which is a total order refining the order according to the coloring (so higher colors signal higher relevance). The vertices of the highest even color are the most valuable for player 0, while the vertices of the highest odd color are most valuable for the opponent, player 1. This defines the "reward ordering" $\prec$ (for player 0), obtained by concatenating the reversed relevance ordering of the odd-colored vertices and the relevance ordering of the even-colored vertices. (So a higher position of a vertex in the reward ordering signals an advantage for player 0.)

Assume a strategy pair (σ, τ) of the two players is given, where σ is defined by the choice of one out-edge for each vertex in V_0, similarly τ by the choice of one out-edge for each vertex in V_1. The strategy pair induces a well-defined play once an initial vertex is given; moreover, this play will end in a loop (which is completed when the first repetition of a vertex occurs). Given (σ, τ), each vertex v will get a value (also called *play profile*) which is extracted from the associated play starting in v. Given this valuation on the whole game graph (and an ordering of the value set), a strategy improvement step of player 0 will then consist in locally redefining the choice of his out-edges: he will pick, for each vertex u, an edge to a neighbour vertex v of highest value.

The play profiles are composed of three components, referring to the relevance order introduced above. Recall that for a given strategy pair (σ, τ) and vertex v the resulting play will end in a loop L. The three components r, P, m of the play profile of v are, in their order of significance, defined as follows: The first component is the most relevant vertex r of L, and called positive if its color is even, otherwise negative. Similarly, a play profile (r, P, m) is called positive (resp. negative) iff r is. We call r shortly "the most relevant loop vertex associated to v". The second component P of the play profile is the set of vertices more relevant than r and visited before entering the loop L (from v), and the third component is the number m of vertices visited before reaching r.

The play profiles are linearly ordered by an ordering $\prec$ as follows:

$$(r, P, m) \prec (r', P', m') \iff \begin{cases} \quad r \prec r' \\ \vee \quad (r = r' \wedge P \prec P') \\ \vee \quad (r = r' \wedge P = P' \wedge r' \in V_- \wedge m < m') \\ \vee \quad (r = r' \wedge P = P' \wedge r' \in V_+ \wedge m > m') \end{cases}$$

where $P \prec P'$ holds if there is a most relevant vertex in the symmetric difference of P and P' that is either positive and in P' or negative and in P.

The algorithm is outlined in Figure 1.

Strategy Improvement Algorithm

1. Choose an arbitrary strategy σ for player 0 (by picking an out-edge for each vertex in V_0).
2. Evaluate this strategy for player 0 (by computing an "optimal response" strategy τ of player 1, which induces together with σ a valuation of the game graph with play profiles).
3. By picking a neighbour vertex of highest value (play profile) for each vertex in V_0, construct a new strategy σ for player 0 and continue with step 2; if the new strategy σ was assumed already before, proceed with step 4.
4. The winning region W_0 (resp. W_1) consists of the vertices with positive (resp. negative) play profile, and the desired winning strategies of player 0 and 1 are extracted from the computed strategies σ and τ on the sets W_0 and W_1.

Figure 1: Strategy Improvement Algorithm

The crucial step in this algorithm is the computation of the "optimal response strategy" of player 1 in step 2. For these details we have to refer the reader to the paper [VJ00]. In our implementation, written in C, some algorithmic improvements over [VJ00] are realized, which significantly affect the running time. For the sake of completeness we list them here (but skip the algorithmic justification):

1. Only play profiles of vertices v are computed where the associated most relevant loop vertex r is negative.
2. A strategy σ is not changed at vertex v when the associated most relevant loop vertex is positive.
3. The second component of the play profile is represented in tree form, which gives a linear memory bound for these components over all vertices.

In the sequel, we concentrate on the second component of the implementation, a user interface written in Common Lisp. This user interface allows to name vertices by arbitrary symbols or tuples of symbols. Using the interactive environment of Lisp, one can easily generate scalable game graphs, as needed for parametrized case studies.

3 User Interface

3.1 Input language

The input language is embedded in Common Lisp, extended by some functions and macros that allow a convenient description of parity game graphs.

The synthesis algorithm can be called from within a Common Lisp environment by

(strategy vertices−of−player0 vertices−of−player1 edge−list colors order)

This function call starts with its name *strategy*. The parameters *vertices−of−player0* and *vertices−of−player1* are the lists of vertices associated with player 0 resp. 1. The parameter *edge−list* is a list of edges; each edge is itself a list of the form *(source target)* where *source* and *target* are vertices. The parameter *colors* is a list *(color0 color1 ... colorn)* where *colori* is the list of vertices of color i, usually in the order of increasing relevance.

The parameter *order* can be omitted; in this case (which corresponds to the option *:last−best*) the colors are considered to be ordered by increasing relevance. If the parameter *order* is set to be *:first−best*, then the colors are considered to be listed in decreasing relevance, as well as the vertices in each list *colori*.

A vertex can be any Lisp object. The equality of two vertices is determined by the function ***equal***.

Example

Consider the parity game graph $G_2 = (V_0, V_1, E, c)$ with $V_0 = \{2, 4\}$, $V_1 = \{1, 3\}$, $E = \{(1, 2), (2, 1), (2, 3), (3, 4), (4, 3)\}$ and $c(1) = 1$, $c(2) = 1$, $c(3) = 1$, $c(4) = 0$ as presented in Figure 1. We indicate vertices in V_0 by circles and vertices in V_1 by boxes.

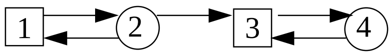

Figure 2: Example graph G_2.

The procedure for computing the winning regions and corresponding winning strategies is called as follows:[1]

```
(strategy '(2 4)
          '(1 3)
          '((1 2) (2 1) (2 3) (3 4) (4 3))
          '((4) (3 2 1))
          :first-best )
```

Instead of listing the elements for the parameters of such a function call explicitly, we shall use functions that generate the parameters for a call of *strategy*.

Macrolanguage

For larger examples, an algorithmic definition of game graphs is appropriate. We develop a notation for such definitions, again in the form of a Lisp macro, now with the name *parity−game*. It provides an environment in which the three macros *vertex*, *edge* and *dedge* are available. These introduce the vertices, the edges, and for convenience also double edges ("dedges") which are considered present in both directions.

[1]The quote symbol in front of a parameter tells the interpreter that it is constant, otherwise the following list would be also interpreted as a function call.

Example

We shall explain its use for the example graphs G_n given in Figure 3.

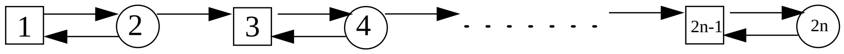

Figure 3: Example graph G_n.

A specification in Pascal–like pseudo code is given below as well as the corresponding Lisp-Code.

```
for v:= 1 to n ∗ 2 do
  v is vertex with:
    if even then v in V0
           else v in V1
    if v = 2 ∗ n then color(v) = 0
               else color(v) = 1
for i:= 1 to n do
  (2 ∗ i − 1, 2 ∗ i) is d−edge
for i:= 1 to n − 1 do
  (2 ∗ i , 2 ∗ i + 1) is edge
```

```
(parity−game
 (for (v 1 (∗ n 2))
   (vertex (v)
           (if (evenp v) 0 1)
           (if (= v (∗ 2 n)) 0 1)))

 (for (i 1 n)
   (dedge ((− (∗ 2 i ) 1)) ((∗ 2 i)) ))
 (for (i 1 (− n 1))
   (edge ((∗ 2 i)) ((+ (∗ 2 i ) 1)) )))
```

The introduction of the edges is self-explanatory. For the vertices, one has to declare, besides the name, the player to which it belongs, and its color (in the example the even vertices belong to player 0, the other to player 1, and the color is set 0 for vertex $2n$, otherwise 1). In this environment, a vertex is always a list. The conditionals are written as usual, with the if-clause and the two values for the yes- resp. no-case. We introduced also a for-loop with the obvious meaning.

For the examples discussed below we also use definitions of the same format (not given in this extended abstract).

3.2 Output

The output contains the following data: the computed winning regions of players 0 and 1, optimal strategies for the two players, and the number of iterations used (improvement steps). The latter is the critical parameter for separating polynomial from exponential behaviour of the algorithm.

The listed strategies are winning on the respective winning region. On the complement the computed strategy is of course not winning, but just as good as possible with respect to the reward order (in the sense of section 2).

Example

In the example considered above, one sees the following output (which says that player 0 wins from each vertex by choosing the listed edges from vertices in V_0):

```
(4 3 2 1)       ; winning region for 0
NIL             ; winning region for 1
((2 3) (4 3))   ; winning, resp. optimal strategy for 0
((1 2) (3 4))   ; winning, resp. optimal strategy for 1
2               ; number of iterations
```

For a more detailed analysis of the algorithm we found the following parameters to be significant (and accessible by separate calls of corresponding functions):

The overall running time is measured in the *number of edge traversals*. By this we mean the number of all edges traversed for computing the valuations. The number of executions of each other basic operation is bounded by the number of edge traversals. That means runtime is approximately linear in the number of edge traversals; and the number of edge traversals is a system independent performance measure.

The complicated feature of the algorithm is the possibility of changing the strategy for each vertex of player 0 in each step. An interesting overview of the execution is thus obtained by a matrix in which each row describes the result of a given iteration for the vertices of player 0. We call this matrix the *improvement trace*. Each vertex is associated to one column, in the order of increasing relevance. The i-th row records the result for the i-th iteration. A mark at position (i, j) of the matrix indicates that the strategy for vertex j (in the given order) is changed in iteration i. We distinguish two kinds of marks, star and dot. By this we record whether in the corresponding improvement step the most relevant loop vertex (see section 2 above) of vertex j is changed during iteration i or not. If this happens then the mark is taken to be a star, otherwise a dot.

An important observation in this connection is the following: If n is the number of vertices, there may by only n^2 number of iterations containing a star. Hence the crucial point in deciding whether the algorithm is polynomial time is to bound the number of iterations without star entries. (In the present extended abstract we have to skip the examples which illustrate this.)

4 Remarks on Case Studies

The first case studies carried out with the system described above were concerned with example games discussed in previous literature on the subject, where different approaches to the problem of solving parity games were tried. Two such papers are [BLV96], based on a recursive algorithm of McNaughton [McN93], and [Jur00], where progress measures were applied. In both cases, suitable families of game graphs can be defined which prove exponential worst case behaviour for each algorithm. The analysis of the present algorithm with our implementation and user interface shows that here a linear number of iterations suffices to reach termination (which shows its polynomial time behaviour in these special cases).

In this extended abstract, we deal only with the case study of [BLV96].

Let us describe the additional output function of the implementation for the version of the graphs of [BLV96] with 136 vertices, where the vertex of player 0 are numbered $0, 2, \ldots, 134$. The numbers of vertices of player 0 are noted as names of vertices in the computed improvement trace (in decimal notation top-down).

The output (see Figure 4) gives information about the iterations 1 to 11. From the improvement trace one infers that the algorithm adjusts several choices simultaneously until the 6th iteration and then proceeds by adjusting the strategy at the single vertices $16, 12, 8, 4, 0$ in the remaining iterations. In this case, there are 12 iterations, the last one offering no change in strategy (which causes termination) and hence suppressed in the trace.

```
                                                     111111111111111111
        111112222233333444445555566666777778888899999000001111122222333
   02468024680246802468024680246802468024680246802468024680246802468024
--|--------------------------------------------------------------------
 1| * * * * * * * ** * * * * * * * * * * * * * * * *  * * * * * * * * *
 2|* * * * * * * *  * * * * * * * *******************
 3|                              *                                    *
 4|              * * * * * * * *
 5|*           *                    *                                 *
 6|*         *
 7|        *
 8|      *
 9|    *
10|  *
11|*
```

Figure 4: Example of Improvement Trace

Further information can be accessed in an additional file with the following data: no. of iteration (1.), presence of a strategy modification which leads to a change of the most relevant loop vertex (2.), number of vertices where player 0 changes strategy (3.), numbers of edge traversals in different function calls (4.-6.), and sum of all edge traversals (7.). For the example considered above the table is given in Figure 5.

```
#vertices: 136 belonging to player 0: 68
#edges: 272 starting from vertices belonging to player 0: 136
#colors: 68
   1    | 2 |     3     |     4     |     5     |     6     |     7
-------+---+-----------+-----------+-----------+-----------+-----------
     1 | x |         34|          1|        352|         47|        400
     2 | x |         34|         16|       1200|        164|       1380
     3 | x |          2|         86|       1186|        422|       1694
     4 | x |          8|         89|        659|         89|        837
     5 | x |          4|        156|       1192|        142|       1490
     6 | x |          2|        158|       1116|        139|       1413
     7 | x |          1|        160|       1164|        159|       1483
     8 | x |          1|        162|       1247|        149|       1558
     9 | x |          1|        164|       1336|        139|       1639
    10 | x |          1|        166|       1431|        129|       1726
    11 | x |          1|        168|       1532|        119|       1819
    12 |   |          0|          0|          0|          0|          0
```

Figure 5: Additional information

5 Conclusion

We have provided an implementation of the algorithm of [VJ00], extended it by a user interface, and applied it in several case studies. In ongoing work, more and larger examples are analyzed, and the program is connected to other procedures of the system OMEGA. Moreover, the algorithm is being applied to the model-checking problem of the modal μ-calculus, with a suitably extended user interface.

References

[BLV96] Nils Buhrke, Helmut Lescow, and Jens Vöge. Strategy construction in infinite games with Streett and Rabin chain winning conditions. In T. Magaria and B. Steffen, editors, *TACAS'96*, volume 1055 of *LNCS*, pages 207–225. Springer, 1996.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symp. on the Foundations of Computing*, pages 368 – 377, 1991.

[EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model checking for fragments of μ-calculus. In C. Courcoubetis, editor, *Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 385–396, Berlin, 1993. Springer Verlag.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 1770 of *LNCS*, pages 290–301, Lille, France, February 2000. Springer.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl Logic*, 65:149 – 184, 1993.

[Pur95] Anuj Puri. *Theory of hybrid systems and discrete event systems.* Ph.d. thesis, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, December 1995. Memorandum No. UCB/ERL M95/113.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, New York, 1997.

[VJ00] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *Computer Aided Verification (CAV)*, 2000.