Theoretical Computer Science 562 (2015) 165–176

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Exact algorithms for Kayles ☆

Hans L. Bodlaender [a,*], Dieter Kratsch [b], Sjoerd T. Timmer [a]

[a] Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[b] Université de Lorraine – Metz, LITA, 57045 Metz Cedex 01, France

A R T I C L E   I N F O

Article history:
Received 16 January 2012
Received in revised form 26 August 2013
Accepted 21 September 2014
Available online 28 September 2014
Communicated by G. Ausiello

Keywords:
Graph algorithms
Exact algorithms
Combinatorial games
Analysis of algorithms
Moderately exponential time algorithms
Kayles
Independent sets

A B S T R A C T

In the game of Kayles, two players select alternatingly a vertex from a given graph G, but may never choose a vertex that is adjacent or equal to an already chosen vertex. The last player that can select a vertex wins the game. In this paper, we give an exact algorithm to determine which player has a winning strategy in this game. To analyze the running time of the algorithm, we introduce the notion of a K-set: a nonempty set of vertices $W \subseteq V$ is a *K-set* in a graph $G = (V, E)$, if $G[W]$ is connected and there exists an independent set X such that $W = V - N[X]$. The running time of the algorithm is bounded by a polynomial factor times the number of K-sets in G. We prove that the number of K-sets in a graph with n vertices is bounded by $O(1.6052^n)$. A computer-generated case analysis improves this bound to $O(1.6031^n)$ K-sets, and thus we have an upper bound of $O(1.6031^n)$ on the running time of the algorithm for Kayles. We also show that the number of K-sets in a tree is bounded by $n \cdot 3^{n/3}$ and thus Kayles can be solved on trees in $O(1.4423^n)$ time. We show that apart from a polynomial factor, the number of K-sets in a tree is sharp.

As corollaries, we obtain that determining which player has a winning strategy in the games $G_{\text{avoid}}(\text{POSDNF2})$ and $G_{\text{seek}}(\text{POSDNF3})$ can also be determined in $O(1.6031^n)$ time. In $G_{\text{avoid}}(\text{POSDNF2})$, we have a positive formula F on n Boolean variables in Disjunctive Normal Form with two variables per clause. Initially, all variables are false, and players alternately set a variable from false to true; the first player that makes F true loses the game. The game $G_{\text{seek}}(\text{POSDNF3})$ is similar, but now there are three variables per clause, and the first player that makes F true wins the game.

1. Introduction

When a problem is computationally hard, then there still are many situations in which the need can arise to solve it exactly. This motivates the field of exact algorithms, in which we look for faster exact, exponential-time algorithms. Many such exact algorithms have been designed and analyzed for problems that are NP-complete or $\#P$-complete, see [7]. Of course, also problems that are complete for a 'harder' complexity class, such as PSPACE-complete, often ask for exact solutions. Many PSPACE-complete problems arrive from the question which player has a winning strategy for a given position

☆ The work of the second author was supported by the French Agence Nationale de la Recherche (ANR AGAPE ANR-09-BLAN-0159-03). An earlier version [4] of this paper appeared in the Proceedings of the 37th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2011. That version has one case missing in the proof of the upper bound for general graphs. The current paper has a corrected proof, and gives in addition an improved computer-generated bound.

* Corresponding author.
E-mail addresses: h.l.bodlaender@uu.nl (H.L. Bodlaender), kratsch@univ-metz.fr (D. Kratsch), s.t.timmer@uu.nl (S.T. Timmer).

http://dx.doi.org/10.1016/j.tcs.2014.09.042
0304-3975/

in a combinatorial game. Exact algorithms are of great relevance here, e.g., a program could use a heuristic to find a move, but once a position is simple enough, it switches to an exact algorithm to give optimal play in the endgame.

In this paper, we study exact algorithms for one such PSPACE-complete problem, namely the problem to determine which player has a winning strategy in a given instance of the game KAYLES. KAYLES is a two-player game that is played on a graph $G = (V, E)$. Alternatingly, the players choose a vertex from the graph, but players are not allowed to choose a vertex that already has been chosen or is adjacent to a vertex that already has been chosen. Thus, the players build together an independent set in G. The last player that chooses a vertex (i.e., turns the independent set into a maximal independent set) wins the game. Alternatively, one can describe the game as follows: the chosen vertex and its neighbors are removed and a player wins when his move empties the graph. The problem to determine the winning player for a given instance of the game is also called KAYLES. This problem was shown to be PSPACE-complete by Schaefer [12]. In an earlier paper [3], a subset of the authors showed that by exploiting Sprague–Grundy theory, KAYLES can be solved in polynomial time on several special graph classes, in particular graphs with a bounded asteroidal number (which includes well known classes of graphs like interval graphs, cocomparability graphs and cographs). Fleischer and Trippen [6] showed that KAYLES can be solved in polynomial time on stars of bounded degree, and also analyzed this special case experimentally. For general trees, the complexity of KAYLES is a long standing open problem. Variants of the game on paths were studied and shown to be linear time solvable by Guignard and Sopena [9]. For more background, the reader can consult [1,2,5].

It is not hard to find an algorithm that solves KAYLES in $O^*(2^n)$ time,[1] by tabulating for each induced subgraph of G which player has a winning strategy from that position. In this paper, we improve upon this trivial algorithm, and give an algorithm that uses $O(1.6031^n)$ time. The algorithm uses ideas from [3], exploiting results from Sprague–Grundy theory (also known as the theory of *nimbers*). To analyze the running time of the algorithm, we introduce the notion of a K-set: a set of nonempty vertices $W \subseteq V$ is a K-set in a graph $G = (V, E)$, if $G[W]$ is connected and there exists an independent set X such that $W = V - N[X]$. In this paper, we give two upper bounds on the number of K-sets of a graph. For the first one, an upper bound of $O(1.6052^n)$, we give a proof based on a nontrivial case analysis. With the help of a computer-generated case analysis, we obtained an improved bound of $O(1.6031^n)$ for the number of K-sets in a graph. We also show that if G is a tree, then G has at most $n \cdot 3^{n/3}$ K-sets, and thus, KAYLES can be solved in $O^*(3^{n/3}) = O(1.4423^n)$ time on trees (and forests). We also give lower bounds for the number of K-sets. In particular, our bound of $3^{n/3}$ K-sets for trees is sharp except for polynomial terms.

This paper is organized as follows. In Section 2, some preliminary notions and results from graph theory and Sprague–Grundy theory are given. In Section 3, we give an algorithm for determining the winning player for KAYLES played on a given graph G, and show that the running time is bounded by a polynomial in the number of vertices n times the number of K-sets in G. In Section 4, we present the proof that the number of K-sets in a graph is bounded by $O(1.6052^n)$. We explain how the computer-generated bound of $O(1.6031^n)$ is obtained in Section 5. In Section 6, we give an upper bound on the number of K-sets in a tree. Section 7 discusses lower bounds for the number of K-sets in graphs; our bounds for trees are sharp up to a polynomial factor. Some final remarks are made in Section 8, including a simple corollary that shows that for the games G_{avoid}(POSDNF2) and G_{seek}(POSDNF3), one can determine in $O(1.6031^n)$ time which player has a winning strategy.

2. Preliminaries

Graph terminology Throughout this paper all graphs $G = (V, E)$ are undirected and simple. Let $S \subseteq V$. Then $N[S] = \bigcup_{s \in S} N[s]$ is the *closed neighborhood* of S, $N(S) = N[S] \setminus S$ is the *open neighborhood* of S, and $G[S]$ denotes the *subgraph of G induced by S*.

Definition 1. A nonempty set of vertices $W \subseteq V$ of a graph $G = (V, E)$ is called a *K-set* (*Kayles set*) of G, if it fulfills each of the following criteria:

- $G[W]$ is connected
- there exists an independent set $X \subseteq V$ such that $W = V - N[X]$

Sprague–Grundy theory Next, we review some notions and results from Sprague–Grundy theory, and give some preliminary results on how this theory can be used for KAYLES. For a good introduction to Sprague–Grundy theory, the reader is referred to [1,5].

A *nimber* is an integer belonging to $\mathbf{N} = \{0, 1, 2, \ldots\}$. For a finite set of nimbers $S \subseteq \mathbf{N}$, define the minimum excluded nimber of S as $mex(S) = \min\{i \in \mathbf{N} \mid i \notin S\}$.

To each position in a two-player game that is finite, deterministic, full-information, impartial, and with 'last player wins'-rule, one can associate a nimber in the following way. If no move is possible in the position (and hence the player that must move loses the game), the position gets nimber 0. Otherwise the nimber is the minimum excluded nimber of the set of nimbers of positions that can be reached in one move.

[1] We use the so-called O^* notation: $f(n) = O^*(g(n))$ if $f(n) = O(g(n)p(n))$ for some polynomial $p(n)$. See also [7].

```
Procedure compute_nimber(G[W]).
  if nb(W) already computed then
  |  return nb(W)
  else
  |  M := ∅;
  |  for all w ∈ W do
  |  |  let Z_1, Z_2, ..., Z_r (r ≥ 1) be the components of G − N[w];
  |  |  nim := 0;
  |  |  for i ← 1 to r do
  |  |  |  nim := nim ⊕ compute_nimber(G[Z_i]);
  |  |  |  M := M ∪ {nim}
  |  |  answer := mex(M);
  |  nb(W) := answer;
  |  return answer
```

Algorithm 1: Procedure `compute_nimber`.

Theorem 1. *(See [1,5].) There is a winning strategy for player 1 from a position, if and only if the nimber of that position is at least 1.*

Denote the nimber of a position p by $nb(p)$. Given two (finite, deterministic, impartial, last player wins) games $\mathcal{G}_1$, $\mathcal{G}_2$, the *sum* of $\mathcal{G}_1$ and $\mathcal{G}_2$, denoted $\mathcal{G}_1 + \mathcal{G}_2$ is the game where a move consists of choosing $\mathcal{G}_1$ or $\mathcal{G}_2$ and then making a move in that game. A player that cannot make a move in $\mathcal{G}_1$ nor in $\mathcal{G}_2$ looses the game $\mathcal{G}_1 + \mathcal{G}_2$. With (p_1, p_2) we denote the position in $\mathcal{G}_1 + \mathcal{G}_2$, where the position in $\mathcal{G}_i$ is p_i $(i = 1, 2)$.

The binary XOR operation is denoted by $\oplus$, i.e., for nimbers i_1, i_2, $i_1 \oplus i_2 = \sum\{2^j \mid (\lfloor i_1/2^j \rfloor \text{ is odd}) \Leftrightarrow (\lfloor i_2/2^j \rfloor \text{ is even})\}$.

Theorem 2. *(See [1,5].) Let p_1 be a position in $\mathcal{G}_1$, p_2 a position in $\mathcal{G}_2$. The nimber of position (p_1, p_2) in $\mathcal{G}_1 + \mathcal{G}_2$ equals $nb((p_1, p_2)) = nb(p_1) \oplus nb(p_2)$.*

As Kayles is an impartial, deterministic, finite, full-information, two-player game with the rule that the last player that moves wins the game, we can apply Sprague–Grundy theory to Kayles, and we can associate with every graph G the nimber of the start position of the game Kayles, played on G. We denote this nimber $nb(G)$, and call it the nimber of G.

An important observation is the following: when $G = G_1 \cup G_2$ for disjoint graphs G_1 and G_2, then the game Kayles, played on G is the sum of the game Kayles, played on G_1, and the game Kayles, played on G_2. Hence, by Theorem 2, we have the following result.

Lemma 3. $nb(G_1 \cup G_2) = nb(G_1) \oplus nb(G_2)$.

Note that G_1 and G_2 might be disconnected graphs.

Our second observation shows how to express the nimber of a graph G in the nimbers of some subgraphs of G. Consider Kayles, played on $G = (V, E)$, and suppose that a vertex $v \in V$ is played. Then, the nimber of the resulting position is the same as the nimber of $G - N[v]$, as the effect of playing on v is the same as the effect of removing v and its neighbors from the graph. As the nimber of a position is the minimum nimber that is not in the set of nimbers of positions that can be reached in one move, we have:

Lemma 4. *Let $G = (V, E)$ be a graph.*

(i) *If $V = \emptyset$, then $nb(G) = 0$.*
(ii) *If $V \neq \emptyset$, then $nb(G) = mex(nb(\{G - N[v] \mid v \in V\}))$.*

3. An exact algorithm for Kayles

In this section we present our exact exponential-time algorithm solving Kayles. The algorithm starts with a call to the procedure `compute_nimber` shown as Algorithm 1, with input $G = (V, E)$. If it returns a nimber that is at least one, then Player 1 has a winning strategy on G; if it otherwise returns nimber zero, then Player 2 has a winning strategy. Correctness of the procedure directly follows from the discussion in Section 1.

Note that the procedure `compute_nimber`$(G[W])$ is only called for K-sets, and thus $G[W]$ is always connected, with one possible exception: if G is not connected, then the first call to the procedure is for $G[V]$ with V not a K-set. As the overhead per recursive call is polynomial, the running time is a polynomial factor times the number of K-sets in G. The procedure computes the nimber $nb(W)$ of $G[W]$ for all K-sets W of G and stores the value in a table using memoization, i.e., computed values are stored in a table, and by look-up no value $nb(W)$ is computed more than once. It follows that the running time of the algorithm is $O^*(|\mathcal{K}(G)|)$ where $\mathcal{K}(G)$ is the set of K-sets of G.

In Section 4, we give a combinatorial proof that the number of K-sets in a graph with n vertices is bounded by $O(1.6052^n)$. With a computer-generated analysis, this bound can be reduced to $O(1.6031^n)$, as discussed in Section 5. In Section 6, we show that the number of K-sets in a tree with n vertices is bounded by $O(1.4423^n)$. If we combine these bounds with the algorithm of this section, we establish the following result. (The result for forests follows directly from the result for trees, as each K-set in a forest belongs to one connected component.)

Theorem 5. Kayles *can be solved in time* $O(1.6031^n)$ *for graphs with n vertices.* Kayles *can be solved in time* $O(1.4423^n)$ *for trees and forests with n nodes.*

4. An upper bound on the number of K-sets

In this section, we give an explicit proof for an upper bound on the number of K-sets in a graph. Note that we give a slightly better, but computer-generated bound in Section 5. As discussed in Section 3, this bound gives an upper bound on the running time of our algorithm for Kayles on arbitrary graphs.

Theorem 6. *Let G be a graph with n vertices. Then G has* $O(1.6052^n)$ *K-sets.*

The proof of Theorem 6 is algorithmic: we give a branching procedure that generates all K-sets. By distinguishing different types of vertices, assigning these different weights, and considering the different branching vectors, we obtain a set of recurrences, whose solution gives us the desired bound. For information on branching algorithms and their analysis, in particular branching vectors and the corresponding recurrences we refer to [7].

We say that a K-set is *nontrivial*, if it has at least three vertices; otherwise we call it trivial. As each trivial set either consists of a single vertex or the two endpoints of an edge, the number of trivial K-sets is at most $n + m$, where m is the number of edges of the graph.

During our branching process, we decide at some points to put some vertices in an independent set X and forbid for some vertices to put them in the independent set. When placing a vertex in X, we say we *select* the vertex. The vertices in G are of four types, each with a different weight:

- **White** or free vertices. Originally all vertices in G are white. We have not made any decision yet for a white vertex. All white vertices have weight one.
- **Red** vertices. Red vertices may not be placed in the independent set X: i.e., we already decided this during the branching. It still is possible that a red vertex becomes deleted later, however. Red vertices have a weight $\alpha = 0.5685$.
- **Green** vertices. A green vertex will be a vertex of any K-set obtained: it never will be removed. I.e., we cannot place the green vertex nor any of its neighbors in the independent set X. Green vertices have weight zero.
- **Removed** vertices. These are either placed in the independent set or are a neighbor of a vertex in the independent set. All removed vertices have weight zero. Removed vertices are considered not existing, i.e., when discussing the neighbors of a vertex, these neighbors will be white, red, or green.

The *measure* of an instance G is the total weight of all vertices, and the difference in the measure from an instance to one of a subproblem often called *gain* is used to analyse the branching algorithm via branching vectors. Our branching process may be overcounting the number of K-sets (in particular, in some cases, we will not detect that a generated set is not connected), but the obtained bound nevertheless is valid as an upper bound.

The semantics of the colors implies that we can always perform the following actions:

- **Rule 1**: If a red vertex v has no white neighbors, we can color it green. This is valid, as we can no longer place a neighbor of v in X.
- **Rule 2**: If a green vertex v has a white neighbor w, we can color w red. This is valid, as placing w in X would remove v, which we are not allowed by the green color of v.

Rules 1 and 2 will always decrease the measure. They ensure that each red vertex will have a white neighbor, and that white vertices have no green neighbors.

The following action also can always be performed; the removed vertices can no longer be part of a nontrivial K-set.

- **Rule 3**: If $W \subseteq V$ is a set of one or two neighboring white vertices that have no other (nondeleted) neighbors, then remove all vertices of W.

Before starting the main recursive branching, we first fix one vertex $v_0 \in V$, of which we will assume that it is an element of the K-set. In terms of colors, this means that we color v_0 green and all neighbors of v_0 red. Clearly, the total number of K-sets will be at most n times the bound on the number of K-sets that contain a specific vertex.

We obtain a fourth rule.

- **Rule 4**: If G has more than one connected component, then remove all vertices from components that do not contain v_0.

As a consequence, we have that when no rule can be applied and there is at least one white vertex, then there exists a white vertex that is adjacent to a red vertex.

We consider two main types of branching. The first type of branching is a *vertex branch*. Let $v \in V$ be a white vertex. We consider two cases: v is placed in X (called select v), and v is not placed in X. In the former case, we remove and decrease the measure by the total weight of all white and red vertices in the closed neighborhood of v. In the latter case, we color v red and have a measure decrease of $1-\alpha$. In some cases, we gain more by applications of Rules 1, 2, and 3.

In the second type of branching, we consider a number of cases, of which one must apply. Again, in some cases, we can gain more by applications of Rules 1, 2, and 3.

In the sequel we present all branching rules in a preference order. Hence when Case i branching is applied to an instance all earlier cases do not apply.

Case 1: There is a white vertex with at least three white neighbors If v has three white neighbors, we can perform a vertex branch on v. The branching vector in this case will be $(4, 1-\alpha)$, i.e., in one case, we decrease the measure by at least four, and in the other case, we decrease the measure by $1-\alpha$.

Case 2: There is a white vertex with two white neighbors and at least one red neighbor If v has two white neighbors and at least one red neighbor, then a vertex branch on v gives a branching vector of $(3+\alpha, 1-\alpha)$.

Suppose Cases 1 and 2 cannot be applied anymore. Then all white vertices have at most two white neighbors. Moreover, there cannot be a cycle of white vertices, as such a cycle would either be removed by Rule 4 or contains a vertex to which Case 2 applies. With similar arguments, only the endpoints of a path of white vertices can be adjacent to a red vertex, and at least one endpoint is adjacent to a red vertex.

Case 3: The subgraph induced by white vertices contains a path of length at least two, with both endpoints incident to at least one red vertex Suppose now we have a path of white vertices $v_1, \ldots, v_r$, $r \geq 2$, with v_1 and v_r incident to a red vertex. As Case 2 no longer applies, we can assume that $v_2, \ldots, v_{r-1}$ have no nondeleted neighbors outside the path.

Let R be the set of red vertices that are adjacent to v_1 and/or v_r.

Case 3.1: $r = 2$ We must either select v_1, or select v_2, or select neither v_1 nor v_2. In the latter case, both v_1 and v_2 can be colored green, so the measure decreases by two in this case. Hence, we have a branching vector $(2+\alpha, 2+\alpha, 2)$.

Case 3.2: $r = 3$ and $|R| \geq 2$ We consider all cases of placing vertices from $\{v_1, v_2, v_3\}$ in X:

- Select v_1 and v_3: we decrease the measure by $3 + 2\cdot\alpha$.
- Select v_1: we decrease the measure by $3+\alpha$.
- Select v_2: we decrease the measure by 3.
- Select v_3: we decrease the measure by $3+\alpha$.
- Choose none: we decrease the measure by 3. (All three vertices can be colored green.)

So, in this case, we obtain a branching vector $(3+2\cdot\alpha, 3+\alpha, 3, 3+\alpha, 3)$.

Case 3.3: $r = 3$ and $|R| = 1$ The vertices v_1 and v_3 have a common red neighbor. Now, we can perform a vertex branch on v_1. If we select v_1, then v_3 becomes an isolated vertex and will be removed by applying Rule 3. Thus we have a branching vector of $(3+\alpha, 1-\alpha)$.

Case 3.4: $r = 4$ and $|R| \geq 2$ Like in Case 3.2, we consider all cases of placing vertices from $\{v_1, v_2, v_3, v_4\}$ in X, and obtain a somewhat tedious case analysis. In each case, each vertex in $\{v_1, v_2, v_3, v_4\}$ either is removed or is green. If v_1 or v_4 is placed in X, we gain an additional α for the removal of the red neighbor of this vertex. In case we select both v_1 and v_4, we gain $2\cdot\alpha$; here we use that $|R| \geq 2$. This gives a branching vector of $(4+\alpha, 4+2\cdot\alpha, 4+\alpha, 4+\alpha, 4, 4, 4+\alpha, 4)$, corresponding to selecting $\{v_1, v_3\}$, $\{v_1, v_4\}$, $\{v_2, v_4\}$, $\{v_1\}$, $\{v_2\}$, $\{v_3\}$, $\{v_4\}$ or no vertex from this path for inclusion in X.

Case 3.5: $r = 4$ and $|R| = 1$ We do a vertex branch on v_1: if we select v_1, then Rule 3 will remove v_3 and v_4. So the branching vector is $(4+\alpha, 1-\alpha)$.

Case 3.6: $r \geq 5$ We branch as follows:

- v_1 is placed in X: we decrease the measure by $2+\alpha$.
- v_2 is placed in X: we decrease the measure by 3.

- v_3 is placed in X and v_1 is not placed in X. v_1 can be colored green, and thus we decrease the measure by 4.
- v_4 is placed in X and v_1 and v_2 are not placed in X. v_1 and v_2 can be colored green, and thus we decrease the measure by 5.
- None of v_1, v_2, v_3, v_4 is placed in X. v_1, v_2, v_3 become green, and v_4 becomes red: a measure decrease of $4-\alpha$.

Thus, the branching vector is $(2+\alpha, 3, 4, 5, 4-\alpha)$.

Case 4: v_1 is a white vertex with no white but at least two red neighbors We do a vertex branch on v_1. If we do not select v_1, it can be colored green, by Rule 1. So we obtain a branching vector $(1+2\cdot\alpha, 1)$.

Case 5: v_1 is a white vertex with exactly one neighbor, which is red Let w be the red neighbor of v_1.

Let K be any nontrivial K-set which can be obtained from the current instance. The white vertex v_1 has only one red neighbor w, and neither white nor green neighbors. We claim that it suffices to study two branches:

- v_1 belongs to K and w belongs to K
- v_1 does not belong to K and w does not belong to K

Let us prove by contradiction that it suffices to explore those two branches by showing that the other two are superfluous.

First, if v_1 belongs to K and w does not belong to K, then v_1 is a trivial K-set; and will be removed by Rule 3. Hence v_1 does not belong to K, contradiction. Now let us assume that v_1 does not belong to K but w belongs to K. Since the only neighbor of v_1 is the red vertex w which belongs to K, v_1 cannot be removed by being a vertex of the independent set X or one of its neighbors. Since w belongs to K we also do not remove v_1 by Rule 3 or Rule 4. Hence v_1 belongs to K, contradiction.

Consequently, in the first branch v_1 and w are colored green and we gain $1+\alpha$. In the second branch v_1 and w are removed and we gain $1+\alpha$. So, the branching vector is $(1+\alpha, 1+\alpha)$.

If Case 1 cannot be applied, then there is no white vertex with three or more white neighbors. If Cases 4 and 5 can also not be applied then there is no white vertex without white neighbors. Thus we may assume that each white vertex has either one or two white neighbors. I.e., the white vertices induce paths and cycles in the graph, each of length at least two. By the arguments following Case 2, we cannot have internal vertices of these cycles having a red neighbor, but as there are vertices with red neighbors, we cannot have a cycle, and thus have a path of white vertices and a red vertex incident to an endpoint. Since Case 3 cannot be applied, we have a path of white vertices with exactly one endpoint adjacent to one or more red vertices. The remaining Case 6 is now discussed below, with several subcases.

Case 6: The subgraph induced by white vertices contains a path of length at least two, with exactly one endpoint incident to a red vertex Suppose $v_1, \dots, v_r$ is a path of white vertices, and suppose $r \geq 2$ is maximal. Assume without loss of generality that v_1 has a red neighbor, say w.

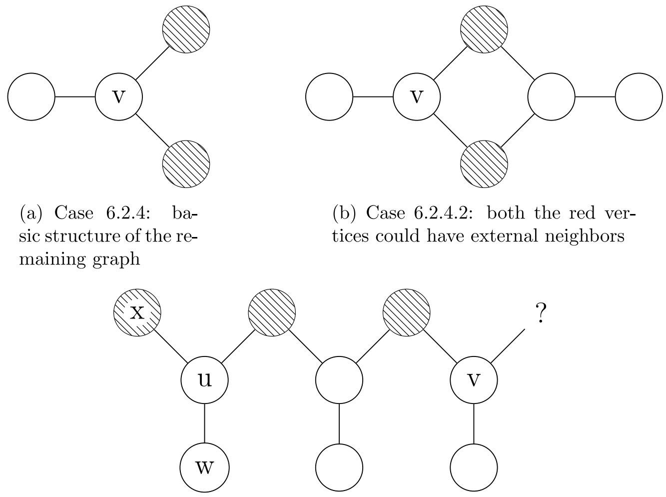
Case 6.1: $r \geq 3$ We do a vertex branch on v_{r-2}. If we select v_{r-2} then we gain at least $3+\alpha$: if $r \geq 4$, then v_{r-2} has two white neighbors, and if $r = 3$, then v_{r-2} has the white neighbor v_{r-1} and the red neighbor w. Moreover, v_r becomes an isolated vertex after v_{r-2} is placed in the independent set, and thus is removed by Rule 3. If we do not select v_{r-2}, we gain $1-\alpha$, and thus we have a branching vector of $(3+\alpha, 1-\alpha)$.

Case 6.2: $r = 2$ There is a number of possibilities:

Case 6.2.1: w is the unique red neighbor of v_1 and has at least one other white neighbor that we will call x We can now perform a vertex branch on x. If we place x in the independent set, then w and x are removed, but also v_1 and v_2 as Rule 3 can be applied: they form a connected component of at most two white vertices. So, the measure is decreased by at least $3+\alpha$. If we do not select x, we color x red to obtain a measure decrease of $1-\alpha$. So, this case gives a $(3+\alpha, 1-\alpha)$ branching vector.

Case 6.2.2: w is the unique red neighbor of v_1 and has no other white neighbor We either select v_1, or we select v_2, or we select neither v_1 nor v_2. If we select v_2, then w can be colored green, as its only white neighbor v_1 is removed. If we select neither v_1 nor v_2, then w, v_1 and v_2 can be colored green, so we decrease the measure $2+\alpha$ in this case. So we obtain a branching vector of $(2+\alpha, 2+\alpha, 2+\alpha)$.

Case 6.2.3: v_1 has ≥ 3 red neighbors We can do a vertex branch on v_1. If we select v, the measure is reduced by $2+3\alpha$. So this case gives a $(2+3\alpha, 1-\alpha)$ branching.



(a) Case 6.2.4: basic structure of the remaining graph

(b) Case 6.2.4.2: both the red vertices could have external neighbors

(c) Case 6.2.4.3: the structure could be longer, but v could be connected to x

Fig. 1. Overview of the possibilities of Case 6.2.4. Red vertices are hatched.

Case 6.2.4: Each white vertex has exactly one white neighbor; each path of two white vertices consists of one white vertex without red neighbors and one white vertex with exactly two red neighbors When none of the above cases applies, there is only one very specific structure for the white vertices. This structure is shown in Fig. 1(a). We now distinguish the following three possibilities. Throughout the analysis of Case 6.2.4, we denote with v a white vertex with two red neighbors.

Case 6.2.4.1: One (or both) of the red vertices has no other white neighbors We perform a vertex branch on v. If v is selected, then the measure is reduced by $2+2\alpha$. If v is not selected, we can color it red. The red neighbors of v that do not have white neighbors can then be colored green by Rule 1. We obtain a measure decrease of at least 1. The branching vector for this case becomes $(2+2\alpha, 1)$.

If the above case does not apply, then all red vertices have at least two white neighbors. The structures thus must form cycles of alternating red and white vertices. The smallest possible cycle is of length four, when two of these structures are connected.

Case 6.2.4.2: Two of the described structures share the same pair of red neighbors This situation is depicted in Fig. 1(b). In this case selecting v removes both of its red neighbors and therefore changes the two vertices on the other side into a connected component. Rule 3 will remove this connected component and therefore selecting v results in a measure decrease of $4+2\alpha$. If v is not placed in the independent set X then the measure decreases by $1-\alpha$. The branching vector is therefore $(4+2\alpha, 1-\alpha)$ in this case.

Case 6.2.4.3: There is no cycle of length four Since there must be a cycle, it has length at least six. Part of the graph is drawn in Fig. 1(c). Note that v may or may not be adjacent to x. For the branching we distinguish four possibilities; note that at least one of these cases must apply:

Select u and v: the remaining white vertices are disconnected and removed by Rule 3, so the measure decreases by $6+3\cdot\alpha$
Select u but do not select v: the measure decreases by $3+\alpha$
Select v but do not select u: the measure decreases by 2
Select neither u nor w: both u and w are first colored red, but then they can be removed by Rule 1. Thus the measure decreases by 2.

The branching vector for this case amounts to $(6+3\cdot\alpha, 3+\alpha, 2, 2)$.

If no case applies, then there are no white, and hence also no red vertices left, so we found one (or zero, in case the green vertices are not connected) K-set. Our choice of $\alpha = 0.5685$ gives the best value for the base of the exponent for the given branching vectors, namely the claimed 1.6052. Thus, it follows that there are $O(1.6052^n)$ nontrivial K-sets that contain v_0. As the value 1.6052 is obtained by rounding, and there are at most $n+m$ trivial K-sets, the result follows.

5. A bound obtained with an automated analysis

The case analysis that was presented in Section 4 can be extended further to reach a branching number of 1.6031. As the number of cases that need to be considered quickly becomes very large, we did not carry out this analysis by hand, but instead used a computer-generated case analysis. We will now outline the architecture of the program.

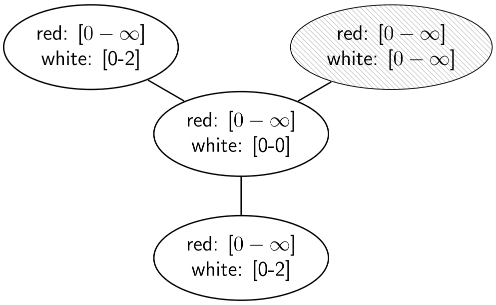


Fig. 2. Example of a configuration.

The main idea is to maintain a collection of cases. The program computes branching vectors for these cases, and from these the corresponding bound on the number of K-sets. Initially the cases as presented above are present in this collection. At runtime the algorithm will 'expand' cases, which means that a case is replaced by a larger collection of cases that together cover the replaced case. More precise details are given below.

Modeling a case Every case can be described by a 'configuration' or 'pattern' in the input graph. A configuration is modeled with a partial or incomplete graph. I.e. we have a colored graph that is assumed to be a subgraph of the input graph; vertices are not only labeled with a color, but also with intervals. These intervals describe how many white and how many red neighbors the vertex may have outside the subgraph. The neighbors of a vertex outside of the configuration will be called its 'external' neighbors. In general a vertex may have zero to infinite external white neighbors and zero to infinite external red neighbors, which simply represents the fact that nothing is known about the external neighbors. During the expansion of cases these intervals may become smaller. An example configuration is drawn in Fig. 2. This particular configuration corresponds to Case 2 from the above analysis. Since Case 1 no longer applies, each white vertex has at most two white neighbors.

Modeling a proof The analysis given in Section 4 is already three levels deep, but we can go even further. We have created a mechanism to automatically explore a certain configuration and generate a set of new configurations that covers all possibilities in which the old case could occur. The generation of these subcases will from here on be referred to as 'expansion'.

A very important aspect of the case analysis is the fact that some of these cases rely on the fact that all earlier cases do not apply. More specifically, Cases 1 and 2 are used in the analysis of the other cases, Case 4 is used in the analysis of Case 5 and Case 6.2 relies on the fact that all other cases have been removed. To use or infer these dependencies automatically is generally hard. It was however not required to do this because we can start with the cases that we distinguished by hand and treat every other case as if it was the first and nothing special was known.

Expansion strategies There are two ways to automatically expand a case: we can either look at the red neighborhood, or at the white neighborhood. When we look for instance at the white neighborhood of a vertex that can still have zero to infinity external white neighbors, we can split this into one configuration where it has exactly zero, one where it has exactly one and one configuration where it has two or more white neighbors. It is then guaranteed that the original case is covered, and we can ignore it in the analysis from that point onwards. A similar argument also holds for the expansion of the red neighborhood of the vertex. This means the algorithm has to make an automated decision with respect to the vertex and the type of neighborhood to expand on. We will introduce a very efficient heuristic for this purpose later on.

Analysis After the generation of a new configuration we recalculate the branching number. When doing a branching by hand there was always a good intuition for the vertex to branch on, but now that we automate the process it is not that easy to know which vertex this is. The solution is to branch on all vertices that do not have external white neighbors (we will call these 'inner vertices' from now on). We simply test all possible selections of these inner vertices, and determine the reduction that would follow from selecting those vertices. When two or more selections result in the same coloring afterward, we only have to consider it once for the branching vector. The calculation of the optimal α value can also be automated such that this procedure can be repeated without intervention.

Choosing the right expansion We have seen that there are always multiple options for the expansion. The question that arises is: "How do you choose the vertex to expand and the neighborhood to expand on?" In order to make progress (improve the branching vector) it is import to choose the right expansion, since choosing a disadvantageous expansion might lead to a lot more work later on, or even to a worse solution. One approach would be to expand in all possible directions and use a backtracking technique to find the optimum sequence of expansions. Since this is computationally not feasible we revert to a heuristic. The heuristic follows in a natural way from the different forms of expansion, and our experiments showed

```
while True do
    if there is a white vertex with external white neighbors then
        expand the white neighborhood of that vertex
    else if there is a white vertex with external red neighbors then
        expand the red neighborhood of that vertex
    else
        expand on the white neighborhood of a red vertex
```

Algorithm 2: The heuristic to choose how to expand a case.

that it is very effective. To understand this heuristic, we should take a look at the four possible kinds of expansions that we could perform:

1. Expand the white neighborhood of a white vertex; this is a good expansion, because afterwards this vertex has become an inner vertex and inner vertices can have either many neighbors which gives a good branch because they disappear if the vertex gets selected. They may, on the other hand, also have few white neighbors, but this also gives a good branch because while the structure gets larger this makes parts of the structure disconnected.
2. Expand the red neighborhood of a white vertex; this is also good to do, but less advantageous because you only reduce the weight by α for each neighbor if you select the vertex, and you do reduce α at all if this is not an inner vertex, so the above expansion should be preferred.
3. Expand the white neighborhood of a red vertex; this is called 'jumping' over the red vertex. If both of the above expansions cannot be applied and there is one red vertex with external white neighbors, selecting any of those white vertices disconnects the whole structure. This is a trick that was also applied manually in the paper to reduce the weight of cases 5 and 6. This is, of course, mainly a good expansion if the above two expansions cannot be applied.
4. Expand the red neighborhood of a red vertex; this is a very disadvantageous expansion because you do not reduce the weight in any of the resulting configurations.

From the above reasoning we deduce the heuristic, given in Algorithm 2.

We note that in our experiments the third case occurred only very rarely, and only if it was really necessary in order to make progress.

Automated result We applied our program to the case analysis given in Section 4. This helped us to identify a missing case in the original proof (given in [4]), and helped to obtain the corrected proof, given here in Section 4. In addition, the program showed that Case 3.4 of the proof given in Section 4 can be replaced by a collection of 95 new cases. This new case analysis is twelve expansions deep; the new collection of cases (all cases from Section 4 except Case 3.4 and the 95 new cases) lead to an upper bound of $O(1.6031^n)$ on the number of K-sets in graphs.

All sources of the automated analysis can be downloaded from https://bitbucket.org/sjoerdtimmer/kayles. Due to space constraints, we do not give the full case analysis here; the interested reader can however reproduce the results with the instructions accompanying the program.

6. A bound on the number of K-sets in trees

In this section, we establish an upper bound on the number of K-sets in a tree. This bound was used in Section 3 to show a bound on the running time of our algorithm, when the input graph is a tree or a forest.

Theorem 7. *Let $T(n)$ be the maximum number of K-sets in a tree with n nodes. Then $T(n) \le n \cdot 3^{n/3}$.*

Proof. We denote as a *rooted K-set* of a rooted tree T any K-set of T containing r, where r denotes the root of T. Let $R(n)$ be the maximum number of rooted K-sets in any rooted tree with n nodes. We claim that $R(n) \le 3^{n/3} - 1$ for all $n \ge 2$.

We are going to prove this claim by induction. To see that the claim is true for the base case $n = 2$, note that the only K-set containing r is the one containing both nodes of the tree, and that $3^{2/3} - 1 > 1.08$.

As induction hypothesis let us assume that the claim is true for all $n' < n$ and consider any rooted tree T on $n > 2$ nodes. Let r be the root of the tree and $u_1, u_2, \dots, u_p$ be the children of v. For every $i = 1, 2, \dots, p$, let T_i be the subtree of T rooted at u_i. Furthermore for all $i = 1, 2, \dots, p$, we denote by n_i the number of nodes of T_i.

Let W be any K-set of T containing its root r. Then for every i, the intersection of W with T_i is either empty or a K-set of T_i containing its root u_i. Note that $n_i = 1$ implies that W also contains u_i since $r \in W$ (and thus r cannot be taken into the independent set X generating W). Using the induction hypothesis and $\sum_{i=1}^{p} n_i = n - 1$, we establish the following upper bound for the number of rooted K-sets of a rooted tree with n nodes

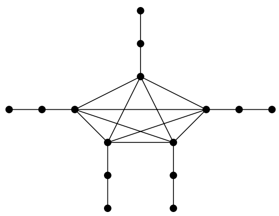


Fig. 3. Example of the construction of Theorem 8, with $n = 5$.

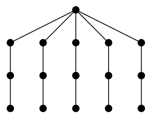


Fig. 4. Example of the construction of Theorem 9, with $n = 5$.

$$R(n) \leq \prod_{i:n_i \geq 2} \left(R(n_i) + 1\right) \leq \prod_{i:n_i \geq 2} 3^{n_i/3}$$
$$\leq \prod 3^{(n-1)/3} \leq 3^{n/3} - 1.$$

This completes the proof of our claim.

To complete the proof of the theorem simply note that any K-set is counted at least once as a rooted K-set for some vertex v chosen to be the root, and thus $T(n) \leq n \cdot R(n)$. □

The above proof can be used to obtain an algorithm to enumerate all K-sets of a tree in time $O^*(3^{n/3})$. This algorithm chooses any vertex r of maximum degree and branches into two subproblems: in one r is taken into W and in the other one r is removed from W and thus all neighbors of r are discarded from S.

7. Lower bounds

In this section we present graphs with n vertices having $\Theta(3^{n/3})$ different K-sets. This implies a lower bound on the maximum number of K-sets of any graph with n vertices as well as a lower bound on the running time of any exact algorithm solving Kayles by using all K-sets of the input graph.

Theorem 8. *There are graphs with n vertices with* $3^{n/3} + 2n/3$ *different K-sets.*

Proof. Consider the following family of (chordal) graphs G_n for all positive integers n on the vertex set $\{1, 2, \ldots, 3n\}$. The edge set of G_n is constructed as follows (Fig. 3):

- $\{3i : i = 1, 2, \ldots, n\}$ is a clique of G_n, and
- for all $i = 1, 2, \ldots, n$, the vertex set $\{3i - 2, 3i - 1, 3i\}$ induces a path and vertex $3i - 2$ has degree 1 in G_n.

Let us count the K-sets W of G_n.

Case 1: $W \cap \{3i \ : \ i = 1, 2, \ldots, n\} = \emptyset$, which implies $|S \cap \{3i \ : \ i = 1, 2, \ldots, n\}| = 1$. Say $S \cap \{3i \ : \ i = 1, 2, \ldots, n\} = \{3i_0\}$. Hence $W \subseteq \{3i - 1, 3i - 2\}$ for some i. Thus if $i \neq i_0$ then $W = \{3i - 1, 3i - 2\}$; and if $i = i_0$ then $W = \{3i - 2\}$. Thus there are $2n$ different K-sets in this case.

Case 2: $W \cap \{3i \ : \ i = 1, 2, \ldots, n\} \neq \emptyset$, which implies $S \cap \{3i \ : \ i = 1, 2, \ldots, n\} = \emptyset$. Then $W \subseteq \{3i - 2, 3i - 1, 3i\}$ may be any of the following sets $\{3i - 2, 3i - 1, 3i\}$, $\{3i\}$, $\emptyset$. Thus there are $3^n - 1$ different K-sets W in this case.

In total the graph G_n has at least $3^n + 2n$ K-sets. □

Theorem 9. *There are trees with n nodes with* $3^{(n-1)/3} + 4(n - 1)/3$ *different K-sets.*

Proof. Consider the following family of trees T_n for all positive integers n. The node set of T_n is the set $\{0, 1, 2, \ldots, 3n + 1\}$. The edgeset is constructed as follows (Fig. 4):

- For all $i = 1, 2, \ldots, n$, the vertex set $\{3i - 2, 3i - 1, 3i\}$ induces a path such that vertex $3i - 2$ has degree 1 in T_n, and
- the node 0 is adjacent to all nodes in the set $\{3i \ : \ i = 1, 2, \ldots, n\}$ and no others.

To count the K-sets W of T_n we distinguish two cases.

Case 1: $0 \notin W$. Then $S \cap \{0, 3, 6, \ldots, 3n\} \neq \emptyset$. Hence $W \subseteq \{3i, 3i-1, 3i-2\}$ for some i. Thus $W = \{3i, 3i-1, 3i-2\}$, $W = \{3i-1, 3i-2\}$, $W = \{3i\}$, or $W = \{3i-2\}$. Thus there are $4n$ different K-sets.
Case 2: $0 \in W$. Then $S \cap \{0, 3, 6, \ldots, 3n\} = \emptyset$. For every i, consider $W \cap \{3i-2, 3i-1, 3i\}$. By connectedness of $G[W]$ and $0 \in W$, we obtain that $W \cap \{3i-2, 3i-1, 3i\}$ is any of the following sets $\{3i-2, 3i-1, 3i\}$, $\{3i\}$, $\emptyset$. Thus there are $3^n - 1$ different K-sets W in this case.

Summarizing, the tree T_n has at least $3^n + 4n$ K-sets. $\square$

8. Conclusions

In this paper, we gave an algorithm to determine which player has a winning strategy for the game Kayles. To analyse the running time, we introduced the notion of K-sets, and obtained upper and lower bounds on the maximum number of K-sets that a graph can have. We also obtained such bounds for trees; up to a polynomial factor, the bounds are sharp for trees. We obtained an upper bound with a hand-made proof, but also a somewhat better upper bound with a computer-generated proof. We expect that for an even better bound, a different method of analysis should be used, e.g., it would be interesting to see if the bound can be improved using a measure and conquer analysis with a larger number of different vertex weights [8], or with the new potential method by Iwata [10].

A number of other interesting directions for further research remain. The complexity of Kayles on trees remains a long standing open problem. But one can also ask if there exists a subexponential-time algorithm for Kayles on trees, e.g., with running time of the form $O(c^{\sqrt{n}})$.

Our algorithm uses exponential memory. It also is open if there exists a polynomial space algorithm with a running time of $O^*(2^n)$, and this may well be hard to obtain.

Our paper is a first example of exact algorithms for problems that are PSPACE-complete. It would be interesting to study such algorithms for other PSPACE-complete problems, e.g., for other combinatorial games, or for a problem like Quantified 3-Satisfiability [11]. An algorithm that solves Quantified (3-)Satisfiability in $O^*(2^n)$ time is not hard to find, but it seems very hard (or impossible) to find an algorithm with a running time $O^*(c^n)$ with $c < 2$ for this problem.

For two of the combinatorial games, introduced by Schaefer [12], we obtain as a simple corollary, that determining which player has a winning strategy can also be determined in $O(1.6031^n)$ time. The first is the game G_{avoid}(POSDNF2). In this game, we are given a positive formula F on n Boolean variables in Disjunctive Normal Form with at most two variables per clause. Initially, all variables are false, and players alternately set a variable from false to true; the first player that makes F true loses the game. The G_{avoid}(POSDNF2) problem is to determine for a given positive DNF formula F with at most two literals per clause, if there is a winning strategy for the first player.

The second game we can handle is G_{seek}(POSDNF3). Here, we are given a positive formula F on n Boolean variables in Disjunctive Normal Form with at most three variables per clause. Again, initially all variables are false, and alternatively, players switch a variable from false to true. Now, the first player to make F true wins the game. G_{seek}(POSDNF3) is the problem to determine for given positive F with at most three literals per clause, if there is a winning strategy for the first player.

It was shown by Schaefer [12] that both G_{avoid}(POSDNF2) and G_{seek}(POSDNF3) are PSPACE-complete, using the PSPACE-completeness of Kayles as starting point for the transformations. Simple transformations in the other direction gives directly a time bound for solving G_{avoid}(POSDNF2) and G_{seek}(POSDNF3).

Corollary 10.
(i) G_{avoid}(*POSDNF*2) *for formulas with n variables can be solved in* $O(1.6031^n)$ *time.*
(ii) G_{seek}(*POSDNF*3) *for formulas with n variables can be solved in* $O(1.6031^n)$ *time.*

Proof. (i) Given a positive DNF formula F with at most two variables per clause, build a graph as follows: take a vertex for each variable in F, and for each clause with two variables $x_i \wedge x_j$ ($i \neq j$), take an edge $\{x_i, x_j\}$. For each clause with exactly one variable x_i, remove the vertex x_i and all incident edges from the graph. (Note that a player choosing x_i directly loses the game.) Let G_F be the resulting graph. Playing Kayles on G_F is equivalent to playing G_{avoid}(POSDNF2) on F: playing the second variable in a clause makes the player directly lose the game, and thus vertices in G_F not incident with a chosen vertices have a one-to-one correspondence to variables that do not give a direct (i.e., without further play) loss in the game G_{avoid}(POSDNF2).

Thus, to determine which player has a winning strategy on F for G_{avoid}(POSDNF2), we can equivalently determine which player has a winning strategy on G_F for Kayles.

(ii) Suppose we are given a positive DNF formula F with at most three variables per clause. If there is a clause with one variable, then a player choosing this variable directly wins the game, so player 1 has a winning strategy in this case. Suppose now that all clauses have at least two variables. Build a graph in the following way: take a vertex for each variable in F, and for each pair of distinct variables x_i, x_j, take an edge $\{x_i, x_j\}$ if there is a clause that contains both x_i as x_j. Now, for each clause that contains exactly two variables i.e., is of the form $x_i \wedge x_j$, remove x_i and x_j and all incident edges from the graph. Let G_F be the resulting graph.

In a winning strategy, a player never plays a variable that appears in a clause with two variables, or in a clause with three variables, one of which has been chosen, as the opponent then can directly win the game by choosing the remaining variable from the clause. Now, it is easy to see that there is a winning strategy for KAYLES on G_F if and only if there is a winning strategy for G_{seek}(POSDNF3) on F. $\square$

References

[1] E.R. Berlekamp, J.H. Conway, R.K. Guy, Winning Ways for Your Mathematical Plays, Volume 1: Games in General, Academic Press, New York, 1982.
[2] E.R. Berlekamp, J.H. Conway, R.K. Guy, Winning Ways for Your Mathematical Plays, Volume 2: Games in Particular, Academic Press, New York, 1982.
[3] H.L. Bodlaender, D. Kratsch, Kayles and nimbers, J. Algorithms 43 (2002) 106–119.
[4] H.L. Bodlaender, D. Kratsch, On exact algorithms for Kayles, in: P. Kolman, J. Kratochvil (Eds.), Proceedings of the 37th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2011, in: Lecture Notes in Computer Science, vol. 6986, Springer Verlag, 2011, pp. 59–70.
[5] J.H. Conway, On Numbers and Games, Academic Press, London, 1976.
[6] R. Fleischer, G. Trippen, Kayles on the way to the stars, in: H.J. van den Herik, Y. Björnsson, N.S. Netanyahu (Eds.), Proceedings of the 4th International Conference on Computers and Games, CG 2004, in: Lecture Notes in Computer Science, vol. 3846, Springer Verlag, 2006, pp. 232–245.
[7] F.V. Fomin, D. Kratsch, Exact Exponential Algorithms, Springer, 2010.
[8] F.V. Fomin, F. Grandoni, D. Kratsch, A measure & conquer approach for the analysis of exact algorithms, J. ACM 56 (5) (2009).
[9] A. Guignard, Éric Sopena, Compound Node–Kayles on paths, Theoret. Comput. Sci. 410 (2009) 2033–2044.
[10] Y. Iwata, A faster algorithm for dominating set analyzed by the potential method, in: D. Marx, P. Rossmanith (Eds.), Proceedings of the 6th International Symposium on Parameterized and Exact Computation, IPEC 2011, in: Lecture Notes in Computer Science, vol. 7112, Springer Verlag, 2012, pp. 41–54.
[11] T.J. Schaefer, The complexity of satisfiability problems, in: Proceedings of the 10th Annual Symposium on Theory of Computing, STOC'78, 1978, pp. 216–226.
[12] T.J. Schaefer, On the complexity of some two-person perfect-information games, J. Comput. System Sci. 16 (1978) 185–225.