# Exact algorithms for coloring graphs while avoiding monochromatic cycles[*]

Fabrice Talla Nobibon[†], Cor Hurkens[‡], Roel Leus[§], Frits C.R. Spieksma[¶]

**Abstract.** We consider the problem of deciding whether a given directed graph can be vertex partitioned into two acyclic subgraphs. Applications of this problem include testing rationality of collective consumption behavior, a subject in micro-economics. We prove that the problem is NP-complete even for oriented graphs and argue that the existence of a constant-factor approximation algorithm is unlikely for an optimization version which maximizes the number of vertices that can be colored using two colors while avoiding monochromatic cycles. We present three exact algorithms, namely an integer-programming algorithm based on cycle identification, a backtracking algorithm, and a branch-and-check algorithm. We compare these three algorithms both on real-life instances and on randomly generated graphs. We find that for the latter set of graphs, every algorithm solves instances of considerable size within few seconds; however, the CPU time of the integer-programming algorithm increases with the number of vertices in the graph more clearly than the the CPU time of the two other procedures. For real-life instances, the integer-programming algorithm solves the largest instance in about a half hour while the branch-and-check algorithm takes about ten minutes and the backtracking algorithm less than five minutes. Finally, for every algorithm, we also study empirically the transition from a high to a low probability of a YES answer as function of the number of arcs divided by the number of vertices.

**Keywords:** directed graph; undirected graph; bipartite graph; acyclic graph; phase transition; NP-complete.

## 1. Introduction

Consider the following problem. Given is a finite, directed graph $G = (V, A)$. The goal is to partition the vertices of $G$ into two subsets such that each subset induces an acyclic subgraph. Since the problem can be equivalently phrased as coloring the vertices of $G$ using two colors such that no monochromatic cycle occurs, we refer to this problem as the *acyclic 2-coloring problem.* Notice that the acyclic 2-coloring problem is defined for a directed graph. The counterpart for undirected graphs is named *partition into two forests* and is known to

[*]A preliminary version of some of the results in this paper has appeared in [Proc. 6th Internat. Conference on Algorithmic Aspects in Information and Management, AAIM 2010, in: LNCS 6124, pp. 229–242, 2010].

[†]University of Leuven, Operations Research Group, Naamsestraat 69, B-3000 Leuven, Belgium. E-mail: Fabrice.TallaNobibon@econ.kuleuven.be

[‡]Eindhoven University of Technology, Department of Mathematics and Computer Science. P.O. Box 513, 5600 MB Eindhoven, the Netherlands. E-mail: wscor@win.tue.nl

[§]University of Leuven, Operations Research Group, Naamsestraat 69, B-3000 Leuven, Belgium. E-mail: Roel.Leus@econ.kuleuven.be

[¶]University of Leuven, Operations Research Group, Naamsestraat 69, B-3000 Leuven, Belgium. E-mail: Frits.Spieksma@econ.kuleuven.be be NP-complete [29]. The problem defined for directed graphs seems to be neither a special case nor a generalization of the problem for undirected graphs; in other words, an algorithm for solving one problem cannot directly be used to solve the other problem and vice versa. Notice also that the acyclic 2-coloring problem is different from the standard graph coloring problem on an undirected graph because two adjacent vertices can have the same color; a directed acyclic graph, for instance, can be colored using a single color.

In this paper, we describe applications of the acyclic 2-coloring problem. We prove that the problem is NP-complete, even for oriented graphs. We also show that it is unlikely to find a constant-factor approximation algorithm for solving an optimization formulation which maximizes the number of vertices that can be colored using two colors while avoiding monochromatic cycles. Further, we identify classes of directed graphs for which the problem is easy. We develop and implement three exact algorithms, namely an integer-programming (IP) algorithm based on cycle identification (in the rest of this text, we also refer to this algorithm as cycle-identification algorithm), a backtracking algorithm and a branch-and-check algorithm. We compare these algorithms based on their CPU time, both on real-life instances coming from micro-economics and on randomly generated graphs. We find that every algorithm solves random graphs of considerable size within few seconds. The CPU time of the cycle-identification algorithm increases with the number of vertices in the graph more clearly than the CPU times of both the backtracking algorithm and the branch-and-check algorithm. Further, for every algorithm we study empirically the phase transition of the problem as function of the number of arcs divided by the number of vertices. When applying the three algorithms to real-life instances stemming from a micro-economics application, however, we find that the cycle-identification algorithm usually takes more time than the two other procedures: the largest instance with 4 384 vertices takes about a half hour, while the branch-and-check algorithm solves that instance in about ten minutes and the backtracking algorithm in less than five minutes.

The contributions of this paper include:

(1) The proof of the complexity status of the acyclic 2-coloring problem for oriented graphs and the establishment of the non-approximability of the optimization formulation which maximizes the number of vertices that can be colored using two colors while avoiding monochromatic cycles.

(2) The identification of some classes of easy graphs.

(3) The development and the implementation of three exact algorithms for solving the acyclic 2-coloring problem.

(4) The empirical study of the phase transition of the acyclic 2-coloring problem.

This paper is organized as follows. In Section 2, we motivate this problem and present a brief literature review. In Section 3, we prove the complexity and the non-approximability results and present some properties of the acyclic 2-coloring problem. In Section 4, we describe the three exact algorithms, present some refinements and identify classes of directed graphs for which the acyclic 2-coloring problem is easy. Section 5 presents some issues related to the implementation of the algorithms. In Section 6, we comment computational results and study empirically the phase transition of the problem. We conclude in Section 7.

## 2. Motivation and notation

In this section, we first explain our motivation for studying this problem and describe some notation and definitions that will be used throughout this paper. Subsequently, we present a brief literature review.

### 2.1 Motivation

Our motivation to consider this problem comes from an application in the *study of rationality* of consumption behavior, a field in micro-economics. We now shortly elaborate on this application. Suppose that there is an economy with $k$ goods, and that we are given a dataset $S$ consisting of $\ell$ observations. Each observation $i$ consists of a pair $(p^i, x^i)$ of (positive) prices $p^i = (p^i_1, \dots, p^i_k)$ and (non-negative) quantities (also called *bundle*) $x^i = (x^i_1, \dots, x^i_k)$, with $i = 1, \dots, \ell$. A single observation may, for instance, describe the expenditures of an economic entity such as a household, at a given moment in time; hence at time $i$, $p^i_j$ $(x^i_j)$ is the price (demand) of good $j$. The dataset $S$ then describes the expenditures over time. Notice that the scalar product $p^i x^i$ corresponds to the amount of money spent by the household in observation $i$. Informally put, *revealed preference* now says that the household directly prefers the bundle $x^i$ over another bundle $x$ if $x^i$ was chosen while $x$ was affordable (and could have been chosen); this translates into $p^i x \leq p^i x^i$.

The notion of preference has allowed economic theory to develop a number of properties that reflect rationality of the dataset (see Varian [28] for an overview). As example of such a property, we mention the Strong Axiom of Revealed Preference (SARP); a dataset $S$ may or may not satisfy SARP. By definition, SARP says that for two observations $s$ and $t$, if there exists a sequence (possibly empty) of observations $i$, $j$, $\dots$, $r$ such that $p^s x^s \geq p^s x^i$, $p^i x^i \geq p^i x^j$, $\dots$, $p^r x^r \geq p^r x^t$, then $p^t x^t < p^t x^s$; observe that the first series of inequalities reflects a direct preference of $x^s$ over $x^i$, of $x^i$ over $x^j$, $\dots$, of $x^r$ over $x^t$ (and we say that $x^s$ is preferred over $x^t$), while the latter inequality reflects that $x^t$ is not directly preferred over $x^s$. Clearly, a relevant question is how to test whether a given dataset $S$ satisfies SARP. It has been shown (in [28]) that this question can be answered using graph theory. A directed graph $G$ with $\ell$ vertices is built by considering each observation $i$ as a vertex. Further, there is an arc from vertex $i$ to vertex $j$ if and only if $p^i x^i \geq p^i x^j$. The dataset $S$ satisfies SARP if and only if $G$ is acyclic.

Recently, testing rationality of observed consumption behavior has been extended to households consisting of multiple members or decision makers (see Cherchye et al. [9]). Deb [13] shows that the problem of testing whether observed data of two-member household consumption behavior satisfies the so-called Generalized Axiom of Revealed Preference (GARP) is NP-complete and in fact is equivalent to an acyclic 2-coloring problem for a specific directed graph built from the data. The problem of testing whether observed data of two-member household consumption behavior satisfies the so-called Collective Axiom of Revealed Preference (CARP) is proved to be NP-complete by Talla Nobibon and Spieksma [24]. In order to find out whether a given dataset satisfies CARP, integer-programming models are proposed in [10] and heuristic approaches, based on acyclic 2-coloring problems for specific directed graphs, are described in [23]. The methods described in this paper can be used to color graphs arising either from testing GARP or from testing CARP.

## 2.2 Notation and definitions

We denote by $G = (V, A)$ a finite directed graph with $|V| = n$ vertices and $|A| = m$ arcs. In this paper, we are only interested in directed graphs without loops, which are arcs for which start and end vertex are the same. For a vertex $p \in V$, the *outdegree* of $p$ is the number of arcs leaving $p$ while the *indegree* of $p$ is the number of incoming arcs to $p$. The *degree* of $p$ is the sum of its outdegree and its indegree. For ease of exposition, we will use $pq$ to represent the arc $p \to q$. If $G$ is such that there are no vertices $p$ and $q$ in $V$ with $pq \in A$ and $qp \in A$ then $G$ is an *oriented* graph. An undirected graph that can be drawn in the plane without any of its edges intersecting is called undirected *planar* graph; such graph is also said to be *embedded in the plane. If a planar graph can be embedded in the plane such that all vertices are incident to the unbounded face of the embedding, then it is called outerplanar graph.* An oriented graph is also obtained by choosing an orientation for each edge of an undirected graph. If the undirected graph is planar (outerplanar) then the obtained oriented graph is also planar (outerplanar). A sequence of vertices $[v_0, v_1, \ldots, v_\ell]$ is called a *chain* of length $\ell$ if $v_{i-1}v_i \in A$ or $v_iv_{i-1} \in A$ for $i = 1, \ldots, \ell$. $G$ is *connected* if between any two vertices there exists a chain in $G$ joining them. In the rest of this paper, we consider only connected graphs. A sequence of vertices $[v_0, v_1, \ldots, v_\ell]$ is called a *path* from $v_0$ to $v_\ell$ if $v_{i-1}v_i \in A$ for $i = 1, \ldots, \ell$. A *vertex-induced subgraph* (subsequently called *induced subgraph* in this text) is a subset of vertices of $G$ together with all arcs whose endpoints are both in that subset. An *arc-induced subgraph* is a subset of arcs of $G$ together with any vertices that are their endpoints. A *strongly connected component* (SCC) of $G$ is a maximal induced subgraph $S = (V(S), A(S))$ where for every pair of vertices $p, q \in V(S)$, there is a path from $p$ to $q$ and a path from $q$ to $p$. A sequence of vertices $[v_0, v_1, \ldots, v_\ell, v_0]$ is called a *cycle* of length $\ell + 1$ in $G = (V, A)$ if $v_{i-1}v_i \in A$ for $i = 1, \ldots, \ell$ and $v_\ell v_0 \in A$. A graph is *acyclic* if it contains no cycle; otherwise it is *cyclic*. A *k-coloring* of the vertices of $G$ is a partition $V_1, V_2, \ldots, V_k$ of $V$; the sets $V_j$ $(j = 1, \ldots, k)$ are called *color classes*. Given a $k$-coloring of $G$, a cycle $[v_0, v_1, \ldots, v_\ell, v_0]$ in $G$ is *monochromatic* if there exists $i \in \{1, \ldots, k\}$ such that $v_0, v_1, \ldots, v_\ell \in V_i$. In this paper, we use the notions vertex coloring and vertex partition of a graph interchangeably.

Given an integer $k$, an *acyclic k-coloring* of $G$ is a $k$-coloring in which the subgraph induced by each color class is acyclic. The *acyclic chromatic number* $a(G)$ of $G$ is the smallest $k$ for which $G$ has an acyclic $k$-coloring. The *directed line graph* $LG$ of $G$ has $V(LG) \equiv A(G)$ and a vertex $(u, v)$ is adjacent to a vertex $(w, z)$ if $v = w$. An arc $pq \in A$ is called a *single arc* if the arc $qp \notin A$. We define the 2-*undirected graph* $G_2 = (V, E)$ associated with $G$ as the undirected graph obtained from $G$ by deleting all single arcs and transforming a pair of arcs forming a cycle of length 2 into an edge (undirected arc); more precisely, $\{v_1, v_2\} \in E$ if and only if $v_1v_2 \in A$ and $v_2v_1 \in A$. We define the *single directed graph* $G_s = (V, A_s)$ of $G$ as the subgraph of $G$ containing only single arcs; more precisely, for a given pair of vertices $v_1$ and $v_2$ in $V$, $v_1v_2 \in A_s$ if and only if $v_1v_2 \in A$ and $v_2v_1 \notin A$.

## 2.3 Literature review

To the best of our knowledge, Deb [12, 13] is the first to explicitly address the acyclic 2-coloring problem. He proves that the problem is NP-complete and extends the results of

Chen [8] for undirected graphs by computing an upper bound on the acyclic chromatic number $a(G)$. Talla Nobibon et al. [23] propose heuristics for maximizing the number of vertices that can be colored using two colors while avoiding monochromatic cycles; these heuristics are based on greedily coloring the vertices.

The literature on acyclic $k$-coloring for undirected graphs, however, is more elaborate. For $k = 2$, Wu et al. [29] study the partition of a graph into two induced forests. Thomassen [26] studies 2-list-coloring planar graphs without monochromatic triangles. Broersma et al. [7] investigate the coloring problem on planar graphs while avoiding monochromatic subgraphs. Several authors have studied the acyclic coloring problem for planar graphs [2, 16, 21, 22]. For a general $k$, Chen [8] gives an efficient algorithm for computing an upper bound of $a(G)$. Theoretical results on acyclic $k$-coloring for undirected graphs are contained in the framework of the generalized graph coloring problem [3]. Applications of acyclic $k$-coloring for undirected graphs include wireless spectrum estimation [18], game theory [5] and logic [6].

## 3. Complexity and properties of the problem

In this section, we study the complexity of the acyclic 2-coloring problem and derive some properties that we use in the next section to build exact algorithms.

### 3.1 Complexity results

We prove that the acyclic 2-coloring problem is NP-complete even for oriented graphs and we argue that it is unlikely to find a constant-factor approximation algorithm for an optimization version which maximizes the number of vertices that can be colored using two colors while avoiding monochromatic cycles.

The acyclic 2-coloring problem is explicitly defined as the following decision problem.
INSTANCE: A finite directed graph $G = (V, A)$.
QUESTION: Does $G$ have an acyclic 2-coloring?

Notice that the acyclic 2-coloring problem is defined as a vertex partition problem. A different problem can be similarly defined by considering arc partitioning of $G$ into two subsets such that each arc-induced subgraph is acyclic. This variant of the problem can be decided in polynomial time; in fact every directed graph is a YES instance. This argument comes from the fact that by building the corresponding line graph, the problem becomes equivalent to partitioning the vertices of the line graph into two subsets such that each subset induces an acyclic subgraph. The latter is identified later in this paper as a YES instance of acyclic 2-coloring problem (see Section 4.5).

Notice that the acyclic 2-coloring problem is in the class NP. In fact suppose that we are given a coloring of the vertices of $G$ using two colors. We consider each subgraph induced by a color class separately. We conclude that we have an acyclic coloring of $G$ if and only if both subgraphs are acyclic (this can be checked in linear time using the topological ordering algorithm [1]). The following theorem shows that the acyclic 2-coloring problem is NP-complete, even for oriented graphs.

**Theorem 1.** *The acyclic 2-coloring problem is NP-complete for oriented graphs.*

**Proof:** See Appendix. □

An optimization version of the acyclic 2-coloring problem maximizes the number of vertices of $G$ that can be colored using two colors such that the subgraph induced by each color class is acyclic. We refer to this problem as Max-A2C. We next prove that Max-A2C contains the *maximum bipartite subgraph problem* defined for undirected graphs as a special case. The maximum bipartite subgraph problem is defined a follows: given an undirected graph $K$, find a bipartite subgraph of $K$ with the maximum number of vertices.

**Lemma 2.** *Max-A2C contains the maximum bipartite subgraph problem as a special case.*

**Proof:** Consider a given instance of the maximum bipartite subgraph problem for a given undirected graph $K = (V, E)$. We build a directed graph $G = (V, A)$ from $K$ as follows: given two vertices $p$, $q \in V$, if there is an edge between $p$ and $q$ in $E$ then both the arc from $p$ to $q$ and the arc from $q$ to $p$ are present in $A$. Observe that a bipartite subgraph in $K$ containing $k$ vertices corresponds to a 2-coloring of the $k$ vertices in the corresponding directed graph $G$ that is acyclic, and vice versa. Therefore, the problem Max-A2C is at least as hard as the maximum bipartite subgraph problem. □

Lund and Yannakakis [19] prove a non-approximability result for the maximum bipartite subgraph problem. Lemma 2, together with their result, implies the following corollary.

**Corollary 3.** *There exists an $\epsilon > 0$ such that Max-A2C cannot be approximated in polynomial time with ratio $n^\epsilon$ unless $P = NP$.*

## 3.2 Properties of the acyclic 2-coloring problem

We derive two properties of the acyclic 2-coloring problem that are used in the next section to build exact algorithms. Let $G = (V, A)$ be a given directed graph, $G_2$ its associated 2-undirected graph and $G_s$ its single directed graph.

**Proposition 4.** *If the set $V$ of vertices of $G$ can be partitioned into two subsets, RED and BLUE, such that $G_2$ is bipartite with all the vertices in RED on one side and those in BLUE on the other side; and the single directed graphs induced by RED, $G_s(RED)$, and by BLUE, $G_s(BLUE)$, respectively, are acyclic then $G$ is a YES instance of the acyclic 2-coloring problem; otherwise $G$ is a NO instance.*

**Proof:** The YES part follows from the fact that RED and BLUE form an acyclic coloring of $G$ while the NO part is immediate. □

**Proposition 5.** *If $G_2$ is not bipartite then $G$ is a NO instance of the acyclic 2-coloring problem, while if $G_2$ is bipartite and $G_s$ is acyclic, then $G$ is a YES instance.*

**Proof:** Immediate. □

Notice that Proposition 5 implies Proposition 4 since if $G_2$ is not bipartite, then there are no two subsets RED and BLUE satisfying the hypothesis of Proposition 4. On the other hand, if $G_2$ is bipartite and $G_s$ is acyclic then there exists two subsets RED and BLUE satisfying the hypothesis of Proposition 4. The converse is not true.

# 4. Exact algorithms

In this section, we describe three exact algorithms for solving the acyclic 2-coloring problem, namely a *cycle-identification* algorithm, a *backtracking* algorithm and a *branch-and-check* (B&C) algorithm. The backtracking algorithm and the B&C algorithm are implicit enumeration algorithms built to solve the acyclic 2-coloring problem while the cycle-identification algorithm is based on an IP formulation of the problem. We also present two dominance rules which can be used to reduce the size of the considered graph. In the rest of this section, $G = (V, A)$ is a given directed graph, $G_2$ is its associated 2-undirected graph and $G_s$ its single directed graph.

## 4.1 Cycle-identification algorithm

We consider an IP formulation of the acyclic 2-coloring problem with binary variables $x_i$ $(i = 1, \ldots, n)$, each of which equals one if vertex $i$ is colored red and zero if it is colored blue. We are looking for a coloring $x_i$ $(i = 1, \ldots, n)$ for which there is no monochromatic cycle. We choose to maximize the number of red vertices. Notice that any other objective function can be chosen. We come back to this issue in Section 6.2. To complete the IP formulation, we add for each cycle $\mathcal{C}$ in $G$, the pair of constraints $1 \leq \sum_{i \in \mathcal{C}} x_i \leq |\mathcal{C}| - 1$, where $|\mathcal{C}|$ is the number of vertices in $\mathcal{C}$. Note that this IP formulation may have an exponential number of constraints.

A formal description of the cycle-identification algorithm is given by CycleId($G$). It works as follows. A relaxed IP instance containing only a subset of constraints is solved. If that instance is infeasible, we stop and output NO. Otherwise, we consider the subgraph induced by each color class separately and check whether there is a cycle. If both subgraphs are acyclic then we stop and output YES. On the other hand, if for at least one induced subgraph a cycle is found, we add to the relaxed IP instance the corresponding pair of constraints. The problem is solved again and the above procedure is repeated until either a YES or a NO answer is returned. Notice that the implementation of this algorithm does not need an optimal solution to the IP instances; a feasible solution is enough.

**CycleId**($G$)

```
1: solve a relaxed IP instance containing only a subset of constraints
2: if there exists a feasible solution
3:    for each subgraph induced by a color class, search for a monochromatic cycle
4:    if monochromatic cycle found
5:       add the corresponding pair of constraints to the relaxed IP instance
6:       solve the relaxed IP instance again and goto 2
7:    else return YES
8: else return NO
```

## 4.2 Backtracking algorithm

An "ordinary" backtracking algorithm for solving the acyclic 2-coloring problem is an adaptation of the well-known backtracking algorithm for graph coloring on undirected graphs. It would work as follows: successively color the vertices of $G$ either red or blue and each time a new vertex is colored, the subgraph induced by the corresponding color class is checked to see whether it is still acyclic; otherwise the color of the last vertex is switched and the subgraph induced by its new color class is then checked. If it is not acyclic, the algorithm backtracks.

In this paper, we propose a backtracking algorithm based on Proposition 4. This is an enumeration algorithm which explicitly colors every vertex of $G$. The key difference between our algorithm and an ordinary backtracking algorithm is that the backtracking algorithm described here can anticipate a NO conclusion earlier without having to color many vertices. This is due to the bipartiteness test included in the algorithm. Broadly speaking, this test consistently extends (if possible) the effect of colored vertices to (connected) uncolored vertices.

A formal description of the backtracking algorithm is given by BT($RED$, $BLUE$, $G$) with $RED = \emptyset$ and $BLUE = \emptyset$ at the beginning. In the description, the function bipartite($RED$, $BLUE$, $G_2$) returns YES if $G_2$ is bipartite given that the vertices in RED are on one side and those in BLUE are on the other side; otherwise it returns NO. We denote by $G_s(A)$ the single directed graph induced by a set $A$.

**BT**($RED$, $BLUE$, $G$)

```
1: if $V = RED \cup BLUE$, then return YES
2: choose a vertex $p$ in $V \setminus \{RED \cup BLUE\}$
3: $RED = RED \cup \{p\}$
4: if bipartite($RED$, $BLUE$, $G_2$) and $G_s(RED)$ acyclic then
5:     if BT($RED$, $BLUE$, $G$) then return YES
6: $RED = RED \setminus \{p\}$, $BLUE = BLUE \cup \{p\}$
7: if bipartite($RED$, $BLUE$, $G_2$) and $G_s(BLUE)$ acyclic then
8:     if BT($RED$, $BLUE$, $G$) then return YES
9: return NO
```

**Proposition 6.** *The backtracking algorithm terminates after a finite number of iterations. Further, upon termination, the output decision corresponds to the decision for the original graph $G$.*

**Proof:** This follows from the fact that there is a finite number of colorings (at most $2^n$) and in the worst case, the backtracking algorithm will enumerate all of them. □

## 4.3 Branch-and-check algorithm

This B&C algorithm is based on Proposition 5. Like the backtracking algorithm, it is an enumeration algorithm where at each node we check some conditions and decide whether to proceed or to stop. Unlike the backtracking algorithm, however, the B&C algorithm is an implicit coloring algorithm which branches on an arc, and the directed graph obtained at every child node is different from the graph at the parent node. The expression *branch-and-check* has also been used in the literature to refer to some algorithms that integrate mixed-integer programming and constraint logic programming [27].

We now explain how to construct two new graphs from a given arbitrary directed graph $G$. This construction is used in the branching step of the B&C algorithm. Let $p, q \in V$ be two adjacent vertices in $G_s$ such that there is a cycle in $G_s$ containing the arc $pq$. Consider the directed graphs $H^{pq} = (V'', A'')$ and $F^{pq} = (V', A')$ defined as follows.

The set of vertices of $H^{pq}$ is $V'' = V$ and the set of arcs $A'' = A \cup \{qp\}$. The set of vertices $V'$ of $F^{pq}$ contains $V$ and two additional vertices $(pq_1)$ and $(pq_2)$; that is $V' = V \cup \{(pq_1), (pq_2)\}$. The set of arcs $A'$ is built as follows.

1. Every arc in $A \setminus \{pq\}$ is an arc in $A'$.
2. For every **single** incoming arc $ap$ into $p$, add an arc $a(pq_2)$ in $A'$.
3. For every **single** outgoing arc $qa$ out of $q$, add an arc $(pq_2)a$ in $A'$.
4. Finally, add the arcs: $p(pq_1)$, $(pq_1)p$, $q(pq_1)$, $(pq_1)q$, $(pq_1)(pq_2)$, $(pq_2)(pq_1) \in A'$.

**Example 1.** *Figure 1 illustrates the construction of $H^{13}$ and $F^{13}$ from the directed graph $G$ by branching on the arc $1 \to 3$.*

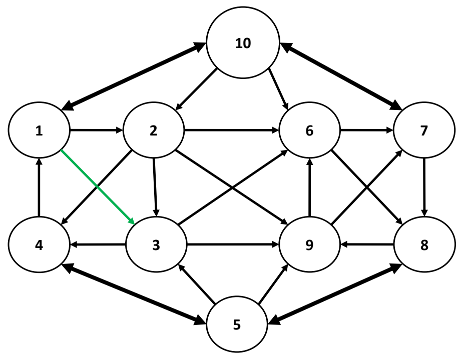
The graph $H^{pq}$ corresponds to a setting where $p$ and $q$ receive different colors, whereas the graph $F^{pq}$ represents the setting where $p$ and $q$ have the same color in any feasible coloring. Informally, the graph $H^{pq}$ arises from $G$ by adding the arc $qp$; the graph $F^{pq}$ arises from $G$ by replacing the arc $pq$ by a node $(pq_2)$, such that each single arc in $G$ entering $p$ (or leaving $q$) now enters $(pq_2)$ (or leaves $(pq_2)$). Further, we add a node $(pq_1)$ in $F^{pq}$ to enforce that the vertices $p$, $q$ and $(pq_2)$ have the same color. Remark that each cycle in $G$ containing the arc $pq$ corresponds to a cycle in $F^{pq}$ containing the vertex $(pq_2)$.

**Proposition 7.** *Let $p$ and $q$ be two adjacent vertices contained in a cycle in $G_s$. $F^{pq}$ or $H^{pq}$ is a YES instance of the acyclic 2-coloring problem if and only if $G$ is a YES instance.*

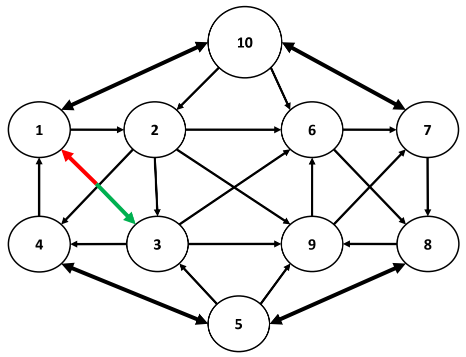
**Proof:** $\Leftarrow$) Assume that the graph $G$ can be partitioned into two acyclic subgraphs. There are two options: either the vertices $p$ and $q$ have the same color or they do not.

If $p$ and $q$ have different colors, then the directed graph $H^{pq}$ can be partitioned into two acyclic subgraphs according to the coloring of $G$; clearly, the 2-cycle $[p, q, p]$ is not monochromatic.

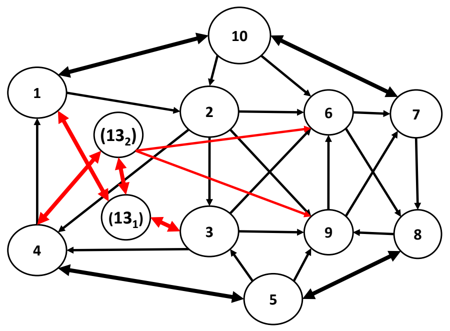
On the other hand, if $p$ and $q$ have the same color, we prove that the directed graph $F^{pq}$ can be partitioned into two acyclic subgraphs. Consider the following coloring of $V'$. Each vertex $a \in V$ receives the color obtained by the coloring of $G$. The vertex $(pq_2)$ is given the color of $p$ and $q$ while $(pq_1)$ receives the color different from that of $p$ and $q$. We next prove that the subgraphs induced by the color classes are acyclic. Suppose there exists a monochromatic cycle $\mathcal{C}$ in $F^{pq}$. $\mathcal{C}$ cannot contain $(pq_1)$ because all its neighbors have a different color. $\mathcal{C}$ must contain $(pq_2)$ because otherwise it would lie in $G$ as well. Consider the part of the cycle $x \to (pq_2) \to y$. Now change cycle $\mathcal{C}$ into cycle $\mathcal{C}'$ by replacing $x \to (pq_2) \to y$ by $x \to p \to q \to y$. This would be a monochromatic cycle in $G$.



(a) The initial graph $G$



(b) The graph $H^{13}$



(c) The graph $F^{13}$

Figure 1: Illustration of the construction of $H^{13}$ and $F^{13}$. In the graphs, a double-direction arc ($\leftrightarrow$) represents a cycle of length two between the considered vertices.

$\Rightarrow$) Suppose that $F^{pq}$ or $H^{pq}$ can be partitioned into two acyclic subgraphs. Clearly, a partition of $H^{pq}$ into two acyclic subgraphs immediately yields a partition of $G$ into two acyclic subgraphs. On the other hand, if $F^{pq}$ can be partitioned into two acyclic subgraphs, we consider the coloring of $G$ defined as follows: $p \in V$ receives the same color as in the coloring of $F^{pq}$. The partition of $F^{pq}$ induces a partition of $G \setminus \{pq\}$ (the graph $G$ minus the arc $pq$) into two acyclic subgraphs because $G \setminus \{pq\}$ is a subgraph of $F^{pq}$. Consequently, if there is a monochromatic cycle $\mathcal{C}$ in $G$, then $\mathcal{C}$ must use the arc $pq$. However, since a cycle in $G$ that uses the arc $pq$ corresponds to a cycle in $F^{pq}$ using $(pq_2)$, there would be a monochromatic cycle in $F^{pq}$: a contradiction. □

A formal description of the B&C algorithm for deciding $G$ is given by BnC($G$).

The branching strategy involves the selection of two adjacent vertices $p$ and $q$ in $G_s$ such that there is a cycle in $G_s$ containing the arc $pq$. The following result proves that using this branching strategy, the B&C algorithm terminates after a finite number of iterations.

**Proposition 8.** *The B&C algorithm terminates after a finite number of iterations.*

**Proof:** To prove this result we introduce the following parameter of a graph. Given a directed graph $G$ and its single directed graph $G_s$, we define the *total length of all distinct*

**BnC($G$)**

```
1: determine $G_2$, $G_s$
2: if $G_2$ is not bipartite, then return NO
3: if $G_s$ is acyclic, then return YES
4: choose an arc $pq$ on a cycle in $G_s$
5: determine $H^{pq}$, $F^{pq}$
6: if BnC($H^{pq}$) then return YES
7: else return BnC($F^{pq}$)
```

*cycles* in $G_s$, denoted $L(G)$, as the number of arcs in all distinct cycles in $G_s$. Notice that an arc is counted as many times as it appears in distinct cycles. We prove that for any two adjacent vertices $p, q \in G_s$ such that there is a cycle in $G_s$ containing the arc $pq$, $L(H^{pq}) < L(G)$ and $L(F^{pq}) < L(G)$. Clearly, $L(H^{pq}) < L(G)$ because at least one cycle in $G_s$ disappears in $H_s^{pq}$ since the arc $pq$ is not in $H_s^{pq}$. On the other hand, $L(F^{pq}) < L(G)$ because any cycle in $G_s$ that uses the arc $pq$ has become one arc shorter in the single directed graph $F_s^{pq}$ of $F^{pq}$. Every cycle in $G_s$ that does not use the arc $pq$ is still present in $F_s^{pq}$, and so has the same contribution to $L(G)$ and $L(F^{pq})$. □

**Theorem 9. Correctness of the branch-and-check algorithm**
*Suppose that the B&C algorithm is run on $G$. Then its execution terminates after a finite number of iterations and the decision corresponds to the decision for the original graph $G$.*

**Proof:** This follows from Proposition 5, Proposition 7 and Proposition 8. □

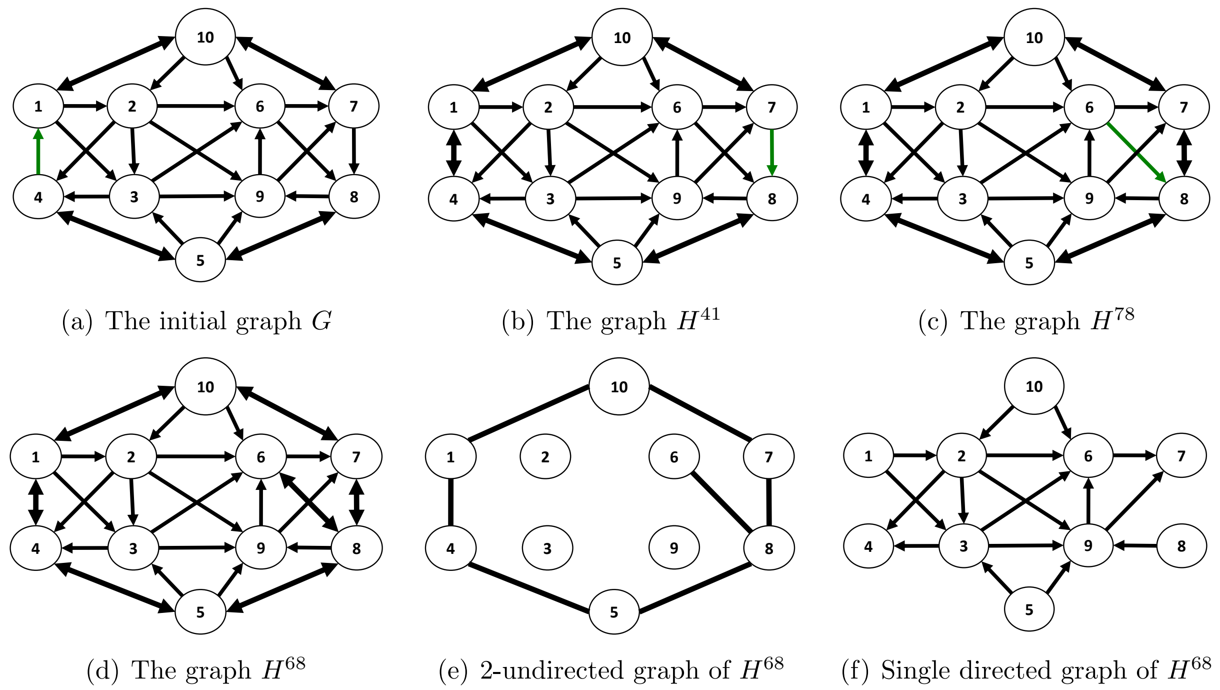
**Example 2.** *Figure 2 illustrates the application of the B&C algorithm. The initial graph $G$, Figure 2(a), is the graph in Figure 1(a). By branching on the arc $4 \to 1$, we obtain two graphs ($H^{41}$ and $F^{41}$) and the graph $H^{41}$, Figure 2(b), is selected as the next graph to investigate. In that graph, we choose to branch on the arc $7 \to 8$. The result is two new graphs, $H^{78}$ and $F^{78}$, and we select $H^{78}$ depicted by Figure 2(c) as the next graph. By branching on the arc $6 \to 8$ in $H^{78}$, we obtain the graphs $H^{68}$ and $F^{68}$. Considering the graph $H^{68}$ given by Figure 2(d), the associated 2-undirected graph depicted by Figure 2(e) is bipartite and the single directed $H_s^{68}$ depicted by Figure 2(f) is acyclic. Therefore, the initial graph $G$ is a YES instance of the acyclic 2-coloring problem. One acyclic 2-coloring of $G$ has color classes $\{1, 2, 3, 5, 6, 7, 9\}$ and $\{4, 8, 10\}$.*

## 4.4 Refinements

In this section, we present two dominance rules which can be used to reduce the size (the number of arcs and/or the number of vertices) of the directed graph $G$.

**Dominance rule 1:** This rule is characterized by the following lemma.

**Lemma 10.** *Given a vertex $p$ in $G$, if the outdegree or the indegree of $p$ is less than or equal to one then the vertex $p$ can be removed from $G$ without changes in the final outcome.*



(a) The initial graph $G$ (b) The graph $H^{41}$ (c) The graph $H^{78}$

(d) The graph $H^{68}$ (e) 2-undirected graph of $H^{68}$ (f) Single directed graph of $H^{68}$

Figure 2: Illustration of the B&C algorithm.

**Proof:** Let $G = (V, A)$ be the directed graph and $p$ be a vertex of $G$ with outdegree or indegree less than or equal to one. Let $G_p$ be the subgraph of $G$ obtained by removing the vertex $p$ and all incident arcs (arcs from $p$ and arcs entering $p$). Clearly, if $G_p$ cannot be partitioned into two acyclic subgraphs, then $G$ cannot be partitioned into two acyclic subgraphs.

On the other hand, suppose that $G_p$ can be partitioned into two acyclic subgraphs. If the degree of $p$ equals zero, we simply add $p$ to any one of the subgraphs forming the partition of $G_p$, and the resulting partition is a partition of $G$ into two acyclic subgraphs. If the indegree (outdegree) of $p$ equals one, let $q$ be the vertex of $G_p$ such that the arc $qp$ ($pq$) exists in $G$. Then we add the vertex $p$ to the subgraph not containing $q$. Clearly, the resulting partition is a partition of $G$ into two acyclic subgraphs. □

**Dominance rule 2:** The aim of this rule is to identify and remove from the graph all single arcs not involved in any cycles in $G_s$. It proceeds as follows. The vertices of $G_s$ are partitioned into SCCs; notice that such a partition is unique. The arcs between two distinct SCCs are deleted since they are not part of any cycle in $G_s$.

Notice that if either Dominance rule 1 or Dominance rule 2 removes at least one arc or at least one vertex, then the repeated application of the other rule may further remove new arcs or vertices. For both the cycle-identification algorithm and the backtracking algorithm, these rules can be applied before starting the algorithm. For the branch-and-check algorithm, however, these rules can be applied both before starting the algorithm and at every node of the branching tree since a new directed graph (either $H^{pq}$ or $F^{pq}$) is built.

### 4.5 Classes of easy graphs

This subsection is devoted to the identification of classes of directed graphs for which the corresponding acyclic 2-coloring problem is always a YES instance. The first class is the class of directed acyclic graphs (DAG). The second class of graphs is the class of line graphs (LG). Talla Nobibon et al. [23] show that a line graph is always a YES instance of the acyclic 2-coloring problem. The third class of easy graphs is the class of partial directed line (PDL) graphs, see e.g. [4]. These are graphs obtained from line graphs by removing a set (possibly empty) of arcs. Clearly, the PDL class of graphs contains the class of directed line graphs. Combining the fact that a line graph is a YES instance of the acyclic 2-coloring problem and the fact that any subgraph of an acyclic graph is also acyclic, we conclude that each graph $G$ in the class of PDL graphs is a YES instance of the acyclic 2-coloring problem.

Let us define the following classes of directed graphs. The class $\mathcal{G}_i^<$ (with $i$ a positive integer) contains all connected directed graphs with each vertex having degree at most $i$; and there is at least one vertex with degree less than $i$. The following corollary follows from repeated application of Lemma 10.

**Corollary 11.** *Every graph in $\mathcal{G}_4^<$ is a YES instance of the acyclic 2-coloring problem.*

Further, some results obtained for undirected planar graphs can be extended to oriented planar graphs. These results are included in the following lemma.

**Lemma 12.**
1. *Each oriented planar graph of maximum degree 4 is a YES instance of the acyclic 2-coloring problem.*
2. *Each oriented outerplanar graph is a YES instance of the acyclic 2-coloring problem.*

**Proof:** This follows from the fact that a similar result is true for undirected planar graphs of maximum degree 4 [21] and for undirected outerplanar graphs [2, 16]. □

# 5. Implementation issues

In this section, we present several issues related to the implementation of every algorithm described in Section 4.

#### Bipartiteness, acyclicness and strongly connected components

An adapted breadth-first-search algorithm [11] is implemented to check whether $G_2$ is bipartite. The same algorithm is also adapted to verify for two given disjoint subsets of vertices, RED and BLUE, whether $G_2$ is bipartite given that all the vertices in RED are on one side and those in BLUE are on the other side. A topological ordering algorithm [1] is used for testing acyclicness of $G_s$ and any induced subgraph $G_s(A)$, where $A$ is a subset of vertices. Tarjan's algorithm [25] is used to identify the SCCs of a given graph.

## Cycle-identification algorithm

The intuition behind the implementation of this algorithm is that “large” cycles (cycles having many vertices) are likely to share some vertices and arcs with “small” cycles (cycles having few vertices). Therefore, feasibly coloring small cycles may lead to a feasible coloring of large cycles at the same time. In our implementation, we start by including only the smallest cycles and gradually add larger cycles.

Hence, the relaxed IP instance initially contains only constraints coming from cycles of length 2. Therefore, throughout the algorithm we search a monochromatic cycle only in the single directed graphs induced by the color classes. Given a color class, we use the Floyd-Warshall algorithm [1,11] to find (if there exist) monochromatic cycles which use the smallest number of vertices. If a monochromatic cycle is found, we add the corresponding pair of constraints to the IP, and the IP instance is solved again; this is an iteration of CycleId. The IP instances are solved using the MIP solver of CPLEX; once a feasible solution is found we stop the solver.

## Backtracking algorithm

**Branching strategy:** The branching strategy of the backtracking algorithm involves the selection of a vertex $p \in V$ which is neither in RED nor in BLUE. We investigate two choices: the first one is simply the first uncolored vertex found while the second choice is an uncolored vertex with the highest degree; ties are broken arbitrarily.

**Propagation rule:** This rule is applied any time that a new vertex $p$ is added either to RED or to BLUE. It works as follows: suppose a vertex $p$ is added to RED (BLUE). Then for any vertex $q$ which is such that the arcs $pq$ and $qp$ exist (this is equivalent to $p$ and $q$ being adjacent in the undirected graph $G_2$), if $q$ is not yet in BLUE (RED) then we add $q$ to BLUE (RED). The procedure is repeated for every new vertex added either to RED or to BLUE.

**Node selection:** Our main objective is to color all the vertices as soon as possible (provided such coloring is possible). Therefore, we use a *depth-first-search* strategy.

## Branch-and-check algorithm

**Branching strategy:** This branching strategy selects a single arc $pq$ which is such that there is a cycle in $G_s$ containing that arc. Prior to choosing an arc $pq$ for branching, we first reduce the single directed graph $G_s$ by proceeding as follows: first, we identify the strongly connected components $G_2^1$, $G_2^2$,. . . ,$G_2^\ell$ of $G_2$ assuming that it has $\ell$ such components. Next, since $G_2$ is bipartite, all vertices have a color either blue or red inferred from the bipartiteness test. For each strongly connected component $G_2^i$ ($i = 1, 2, \dots, \ell$), we delete all single arcs between two vertices of $G_2^i$ with different colors. Finally, any single arc between two vertices of $G_2^i$ with the same color is not considered for branching. We investigate two different choices of the arc $pq$. The first choice is the first arc $pq$ found that meets the above restriction. The second choice is an arc $pq$ with $p$ having the highest degree possible, breaking ties arbitrarily. In both cases, if in addition there is no path in $G_s$ from $p$ to $q$ other than the arc $pq$, we define a simplified version of $F^{pq} = (V', A')$ by merging $p$ and $q$. $V'$ contains a vertex $(pq)$ and all vertices in $V$ except $p$ and $q$ such that $|V'| = |V| - 1$ while $A'$ is built as follows. First, every arc $ab \in A$ with $a, b \notin \{p, q\}$ is an arc in $A'$. Second, for every single incoming arc $ax$ to $x$ with $x \in \{p, q\}$, (respectively every single outgoing arc $xa$ from $x$), add an arc $a(pq)$ (respectively $(pq)a$) in $A'$ while avoiding the repetition of arcs.

**Branch-pruning criterion:** This branch-pruning criterion considers each connected component of $G_2$ and the coloring of its vertices given by the bipartiteness test. If there exists a color class in a connected component which is such that the induced single directed graph is cyclic, then any graph built at a child node of that node is a NO instance of the acyclic 2-coloring problem. Therefore, that node is pruned.

**Node selection:** For the branch-and-check algorithm, we wish to reach a node with a YES answer as soon as possible (provided it exists). We again use a *depth-first-search* strategy.

## 6. Computational experiments

All algorithms have been coded in C using Visual Studio C++ 2005; all the experiments were run on a Dell Optiplex 760 personal computer with Pentium R processor with 3.16 GHz clock speed and 3.21 GB RAM, equipped with Windows XP. CPLEX 10.2 was used for solving the IP instances. Below, we first provide some details on the real-life instances and the generation of random datasets and subsequently, we discuss the computational results.

### 6.1 Data

The three algorithms were tested both on real-life graphs stemming from a micro-economics application and on randomly generated graphs. We first present the real-life instances and next we describe how random instances were generated. The instances described in this section can be found at http://www.econ.kuleuven.be/public/NDBAC96/acyclic_coloring.htm

#### 6.1.1 Real-life data

The graphs presented below come from the study of rationality of consumption behavior described in Section 2. We refer to Cherchye et al. [10] for more details about the datasets containing the prices and quantities describing the expenditures of the household and to Talla Nobibon et al. [23] for the translation of those datasets into directed graphs. Table 1 reports the properties of the real-life instances.

| Instance | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # vertices ($n$) | 22 | 48 | 68 | 95 | 118 | 139 | 226 | 279 | 294 | 410 | 755 | 4384 |
| # arcs ($m$) | 53 | 169 | 297 | 513 | 699 | 985 | 1979 | 2012 | 2427 | 3660 | 10113 | 124321 |
| # arcs/$n$ | 2.40 | 3.52 | 4.36 | 5.40 | 5.92 | 7.09 | 8.76 | 7.21 | 8.26 | 8.93 | 13.39 | 28.36 |

Table 1: Properties of the real-life instances

### 6.1.2 Random data

We have randomly generated directed graphs with $n$ vertices, where $n$ takes the values 50, 100, 200, 500, 1 000 and 5 000. These graphs are generated in such a way that they are connected and contain at least one cycle. To diversify as much as possible the instances, we vary the density $D$ of the graph, which equals the number $m$ of arcs present in the graph divided by the total number of possible arcs.

The graphs are generated using a two-phase procedure. During the first phase, for each value of $n$, 400 graphs are randomly generated with 40 different densities, starting from a lower bound of 2.5% for $n = 50$, 1.5% for $n = 100$, 1% for $n = 200$ and 0.5% for $n = 500$ and $n = 1\,000$; and increased with a step of 0.5%. For $n = 5\,000$ the lower density is 0.05 and the stepsize is 0.05. Thus each arc is present with a probability equal to the density, independently of other arcs. The lower bound is obtained by taking the first multiple of 0.5% greater than or equal to the smallest density for which a connected and cyclic graph can be built given the number $n$ of vertices. For every value of $D$, 10 directed graphs with $m = \lceil D \times (n^2 - n) \rceil$ arcs are generated. Therefore, in total we have $400 \times 6 = 2\,400$ test instances for the first phase.

After preliminary computation on the graphs obtained in the first phase, we identify for each value of $n$ a *critical interval* containing the densities for which we encountered at least one YES instance and at least one NO instance. We observe that densities in this critical interval are exactly those for which potentially hard graphs (requiring long running times) can be found. Notice that for each density not in the critical interval, we have obtained for the instances generated in first phase either always a YES or always a NO answer. This, however, does not mean that there is no density outside the critical interval for which both YES instances and NO instances exist. For a given $n$, we generate additional graphs with the densities given in Table 2.

| $n$ | density ($D$) | | | |
|---|---|---|---|---|
| | from | to | step | total |
| 50 | 8% | 15.75% | 0.25% | 32 |
| 100 | 3.05% | 8.95% | 0.05% | 119 |
| 200 | 2.01% | 3.99% | 0.01% | 199 |
| 500 | 0.8% | 1.498% | 0.002% | 350 |
| 1 000 | 0.3% | 1.2% | 0.002% | 451 |
| 5 000 | 0.03% | 0.8% | 0.002% | 385 |

Table 2: Densities of the graphs generated in the second phase

For every value of the density, 100 directed graphs are randomly generated following the procedure described above, leading to $1\,536 \times 100 = 153\,600$ additional graph instances for the second phase.

## 6.2 Computational results

In this section, we compare different implementations of each of the three algorithms for the set of 50-vertex graphs generated during the first phase (these are 400 graphs in total). We compare the best implementation of the three algorithms based on their CPU time on both

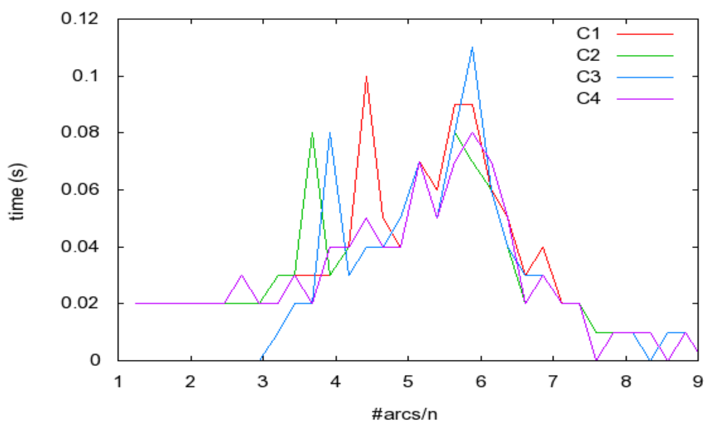


Figure 3: CPU time of four different implementations of the cycle identification algorithm for 50-vertex random graphs generated during the first phase.

randomly generated instances and real-life instances. For each algorithm, we study the phase transition [17, 20] of the acyclic 2-coloring problem as function of some problem parameters.

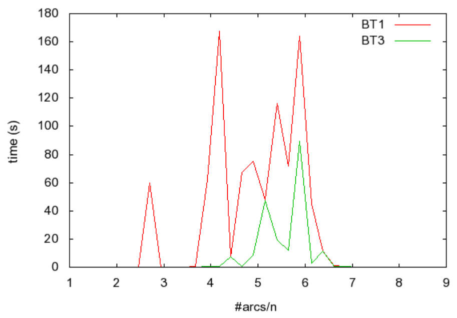
### 6.2.1 Comparison of different implementations of each algorithm

In this section, we examine different implementations of every algorithm for the set of 50-vertex graphs generated during the first phase (these are 400 graphs in total); a time limit of ten minutes is used to stop each algorithm and when this happens, we output undecided. The three algorithms are subsequently compared based on their best (chosen) implementation both on randomly generated graphs (Section 6.2.2) and on real-life instances (Section 6.2.3). In Section 6.2.4, we study empirically the phase transition [17, 20] of the acyclic 2-coloring problem as function of the number of arcs divided by $n$. Throughout this section, the CPU time is expressed in seconds.

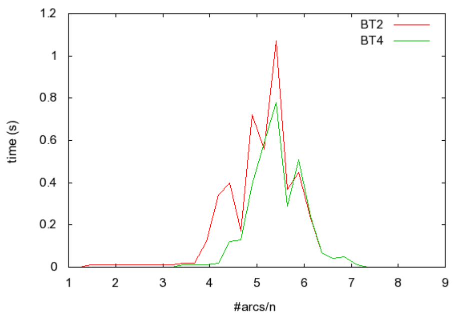
#### Cycle identification algorithm

Figure 3 displays the average CPU time as function of the number of arcs divided by the number of vertices, for four different implementations of the cycle identification algorithm. The first implementation, identified by **C1**, is the implementation of this algorithm as described by the pseudocode CycleId($G$). In this first implementation, however, when a monochromatic cycle is find for one color class, the corresponding pair of constraints is added to the IP problem and the problem is solved again. More precisely, for this implementation, if a monochromatic cycle is find for the class of vertices colored red while we have not yet investigated the existence of such monochromatic cycle in the class of blue vertices, we will not search for monochromatic cycle in that class anymore. The second implementation, **C2**, is similar to implementation **C1**, excepted that whether a monochromatic cycle is found for the first color class or not, we search for a monochromatic cycle in the second color class. The third implementation, **C3**, considers the implementation **C1** with in addition the use of dominance rules while the fourth implementation, **C4**, adds the dominance rules to **C2**.

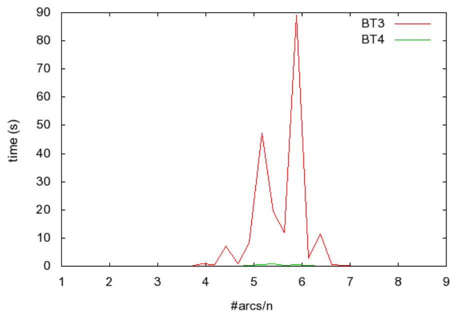
A comparison of different plots of CPU time display in Figure 3 shows that the first



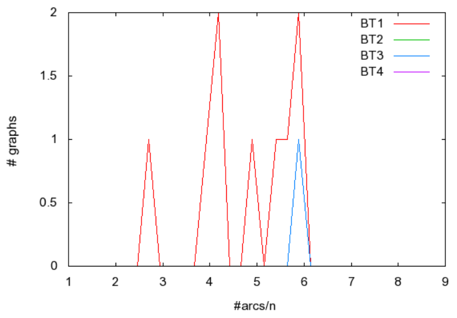
(a) CPU time of BT1 and BT3



(b) CPU time of BT2 and BT4



(c) CPU time of BT3 and BT4



(d) Number of undecided instances

Figure 4: CPU time of four different implementations of the backtracking algorithm for 50-vertex random graphs generated during the first phase.

implementation, **C1**, has an average CPU time slightly higher than that of the third implementation (**C3**). Similarly, the average CPU time of **C2** is higher than that of **C4**. These two observations imply that the use of dominance rules reduces the average CPU time of the algorithm. As for the comparison between **C3** and **C4**, there is no clear indication that one dominates the other. However, the plot of **C4** is usually below that of **C3**. In the rest of this paper, implementation **C4** is adopted for the cycle identification algorithm; meaning that whenever we refer to this algorithm, we imply that the implementation **C4** is used.

Notice that, we have also implemented a variant where at each iteration, the pair of constraints corresponding with only one monochromatic cycle (of minimal length) is added to the IP instance. Further, we have tried different objective functions: one where we minimize the number of red vertices, and one where we balance the number of red and blue vertices (by randomly drawing each objective coefficient out of $\{-1, 1\}$).

## Backtracking algorithm

Figure 4 displays the average CPU time as function of the number of arcs divided by the number of vertices, for four different implementations of the backtracking algorithm. The first implementation, identified by **BT1**, is the implementation of this algorithm as described by the pseudocode BT($RED$, $BLUE$, $G$) with in addition the use of the propagation rule. In this implementation, the branching strategy choose the first uncolored vertex encountered while going from the vertex 1 to vertex $n$. The second implementation, **BT2**, is similar to implementation **BT1**; however, the branching strategy choose an uncolored vertex with the highest degree. The third implementation, **BT3**, considers the implementation **BT1** with in addition the use of dominance rules while the fourth implementation, **BT4**, adds the dominance rules to **BT2**.

A comparison of different plots of CPU time display in Figure 4(a) and Figure 4(b) shows that the first implementation, **BT1**, has an average CPU time higher than that of the third implementation (**BT3**). Similarly, the average CPU time of **BT2** is usually higher than that of **BT4**; enforcing the positive effect of the use of dominance rules. On the other hand, **BT4** has an average CPU time much more smaller than that of **BT3** (see Figure 4(c)). Further using **BT4**, all the instances are solved within a time limit of 10 minutes while there is one instance not decided after the time limit when we use **BT3** (see Figure 4(d)). To conclude, the implementation **BT4** is used for the rest of experiments when we applied the backtracking algorithm.

## B&C algorithm

Figure 5 presents the comparison of six different implementations of the B&C algorithm. The first implementation, **BnC1**, is the B&C algorithm as described by the pseudocode BnC($G$), with in addition the use of the branch pruning criterion and the branching arc $pq$ is the first found. The second implementation, **BnC2**, is similar to **BnC1**, excepted that the branching strategy chooses an arc $pq$ with vertex $p$ having the highest degree. The third implementation, **BnC3**, considers implementation **BnC1** and incorporates the dominance rules at the root node of the branching tree to reduce the initial graph. The fourth implementation, **BnC4**, is **BnC2** plus the use of dominance rules at the root node of the branching tree. The fifth implementation, **BnC5**, considers **BnC1** with the use of dominance rules at every node of the branching tree while the last implementation, **BnC6**, considers **BnC2** with the use of dominance rules at every node of the branching tree.

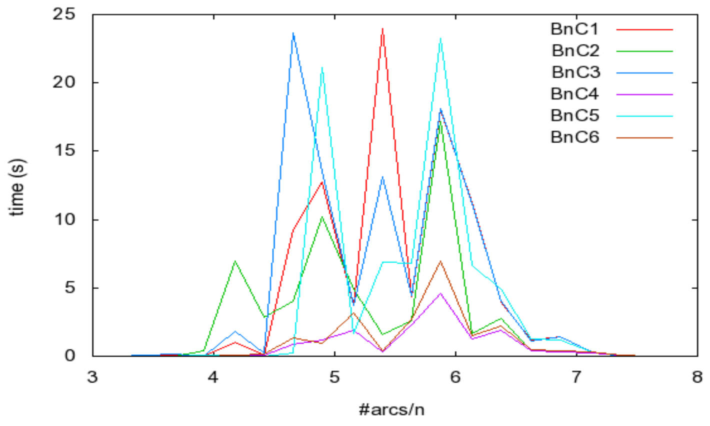
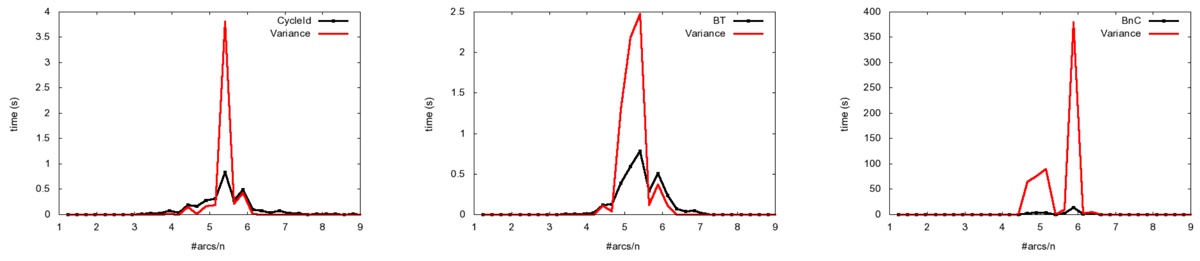


Figure 5: CPU time of six different implementations of the B&C algorithm for 50-vertex random graphs generated during the first phase.

The comparison of the six implementations based on the CPU time is the following. The three implementations using the branching strategy which selects the first arc $pq$ encountered, (**BnC1**, **BnC3** and **BnC5**) have relative higher CPU time compared to the CPU time of



(a) Variance of CycleId (b) Variance of BT (c) Variance of BnC

Figure 6: Variance of every algorithm for 50-vertex graphs generated during the first phase.

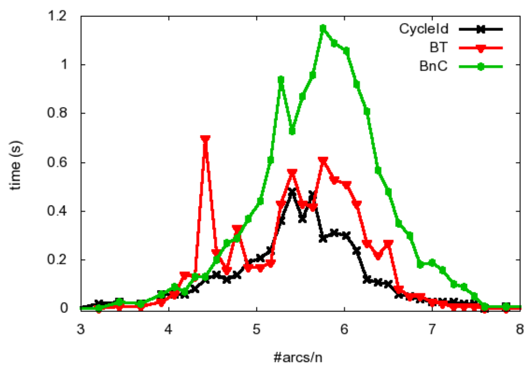
implementations where the arc $pq$ is chosen in such a way the $p$ has the highest degree (**BnC2**, **BnC4** and **BnC6**). Among these last implementations, **BnC4** usually spends the smallest CPU time. In the rest of this paper, we use the implementation **BnC4** of the B&C algorithm for remaining experiments.

Figure 6 displays the variance of the average CPU time of the best implementation of every algorithm as function of the number of arcs divided by $n$; we do this for the 50-vertex graphs generated during the first phase. We find that for every algorithm, a high variance is coupled with a high average CPU time; further, the value of these high variances is several orders of magnitude greater than that of the corresponding average CPU times. This means that among the instances generated, only a few require the algorithm to run for more than a fraction of seconds. In other words, among the instances generated only a few are hard.

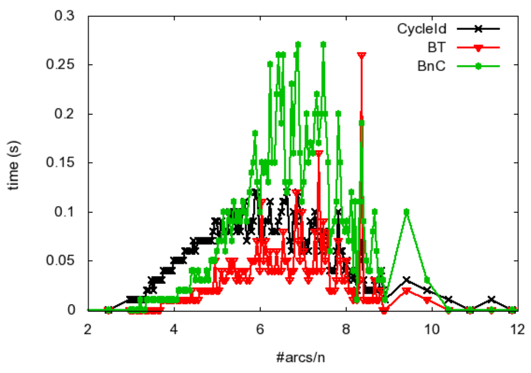
### 6.2.2 Solving random instances

We compare the three algorithms based on their best implementation on random graphs. In Figure 7 we plot, for every value of $n$, the average CPU time of every algorithm as function of the number of arcs divided by $n$. Figure 7(a) shows the average CPU time for the 50-vertex graphs. The B&C algorithm (BnC) usually reports a higher CPU time than the other algorithms. However, the highest average CPU time is less than 1.2 seconds. The cycle-identification algorithm (CycleId) usually uses, on average, the smallest CPU time. For 100-vertex graphs (Figure 7(b)), we see that the average CPU time of CycleId is usually between that of BnC and that of the backtracking algorithm (BT), with BT using, in most cases, the smallest average time. For the large graphs (with more than 100 vertices, see Figures 7(c), 7(d), 7(e) and 7(f)), the average CPU time reported for CycleId increases with the value of $n$, while those of BnC and BT are stable, comparable and usually below one second.

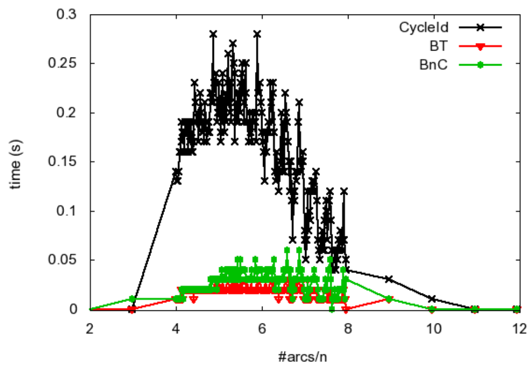
Notice that the CycleId could suffer from the fact that it looks for an optimal solution to the IP and not a feasible one (even though the search is halted as soon as a feasible solution is found). This might partially explain the relatively poor performance when compared with the other algorithms, which are specially designed to find a feasible solution.



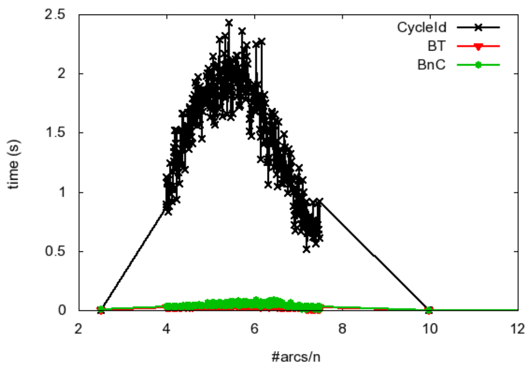
(a) CPU time 50-vertex graphs



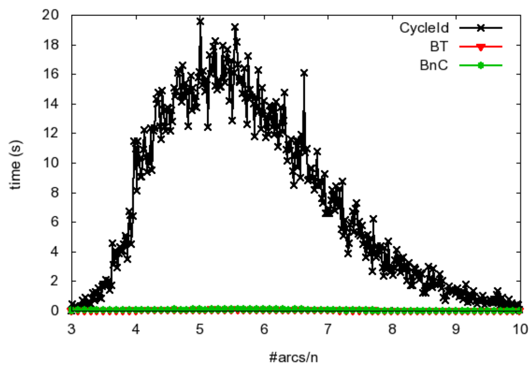
(b) CPU time 100-vertex graphs



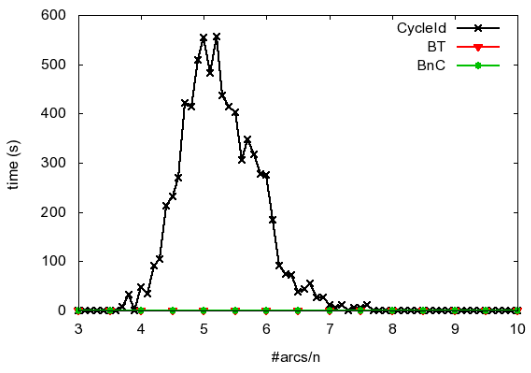
(c) CPU time 200-vertex graphs



(d) CPU time 500-vertex graphs



(e) CPU time 1 000-vertex graphs



(f) CPU time 5 000-vertex graphs

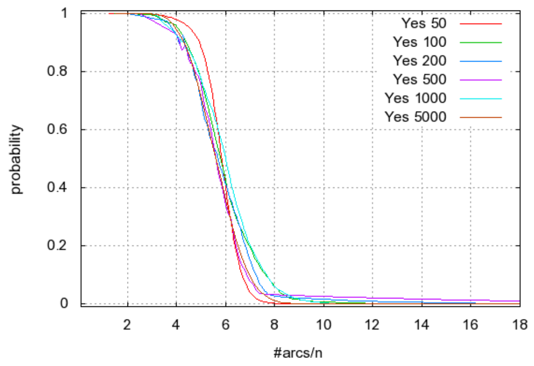
Figure 7: Average CPU time of every algorithm for random graphs.

### 6.2.3 Solving real-life instances

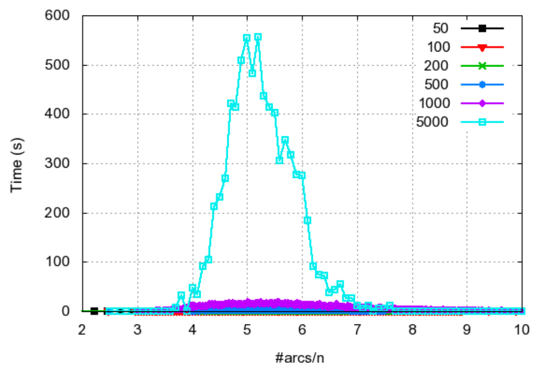
Table 3 reports the CPU time of every algorithm when applied to real-life instances. We see that the backtracking algorithm (BT) reports the best CPU time for five instances out of 12, while the cycle-identification algorithm (CycleId) achieves the best CPU time for six instances and the B&C algorithm (BnC) has the best CPU time for nine instances. For the largest instance with 4384 vertices, however, BT spends less than five minutes, compared to about ten minutes for BnC and about 30 minutes for CycleId.

| Instance | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CycleId | 0.00 | 0.00 | 0.01 | 0.03 | 0.06 | 0.09 | 0.11 | 0.20 | 0.28 | 0.72 | 3.97 | 1812.24 |
| BT | 0.00 | 0.00 | 0.02 | 0.03 | 0.06 | 0.09 | 0.36 | 0.28 | 0.31 | 0.28 | 3.45 | 283.72 |
| BnC | 0.00 | 0.00 | 0.01 | 0.02 | 0.05 | 0.09 | 0.59 | 0.05 | 0.28 | 0.11 | 3.84 | 612.41 |

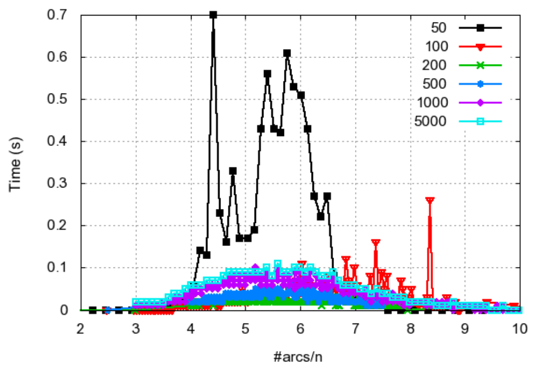
Table 3: CPU time of every algorithm for the real-life instances



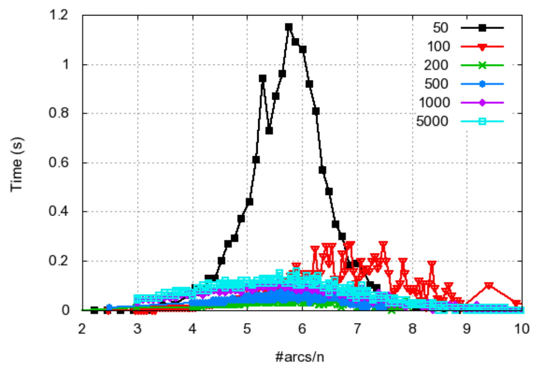
(a) Probability of YES answer



(b) Average CPU time CycleId



(c) Average CPU time BT



(d) Average CPU time BnC

Figure 8: Probability of YES answer and average CPU time of every algorithm.

### 6.2.4 Phase transition analysis

In this section, we investigate the transition from a high to a low YES probability as function of the number of arcs divided by $n$ (subsequently called *parameter* in this section). Further, we show how the CPU time of every algorithm varies as function of the parameter.

Figure 8 presents the probability of a YES answer as well as the average CPU time of every algorithm as function of the parameter. Figure 8(a) shows the probability of YES answer as function of the parameter. The plots in Figure 8(a) are Bézier approximations [14] of the real plots. This approximation is used mainly to render the plots smoother. For every value of $n$, the plot has three regions. In the first region, where the value of the parameter is between 0 and 3, almost all the generated instances have a YES answer. The second region, with the value of the parameter between 3 and 8, is called *critical interval* and contains classes of graphs for which both YES instances and NO instances are present. The last region, with the value of the parameter greater than 8, contains graphs for which the probability of YES is almost zero. Overall, we remark that the five plots are similar and that the *threshold* value of the parameter, for which the probability of YES answer is equal to $\frac{1}{2}$, is almost the same for every $n$ and is close to 5.75.

The plots in Figures 8(b), 8(c) and 8(d) are obtained using the data that were used to generate the plots in Figure 7, but here the plots are grouped by algorithm. Figure 8(b) plots the average CPU time of CycleId for every value of $n$. The plots respect the three regions described above. For the first and the third region, the average CPU time is very close to zero while in the critical interval, we have a non-negligible CPU time, showing an easy-hard-easy transition. Further, CycleId has an average CPU time which increases with the value of $n$, which probably occurs simply because when $n$ increases the IP instance becomes more difficult to solve. Figure 8(c) plots the average CPU time of BT for every value of $n$. The easy-hard-easy transition is also observed here. However, unlike CycleId, BT spends more time in deciding 50-vertex and 100-vertex instances in the critical interval than in deciding instances with more vertices. This decrease in CPU time as the value of $n$ increases stops beyond $n = 200$. The high variability of average CPU time is due to the fact that for very few instances, the algorithm requires more than one second to decide. In other words, among the instances generated there are very few hard instances. In Figure 8(d), the plots of the average CPU time of BnC for every value of $n$ exhibit characteristics similar to those observed for BT. A possible explanation for this decrease in average CPU time is the following: when the value of $n$ increases, the size (number of edges) of the undirected graph $G_2$ increases, making the bipartiteness test used by both BT and BnC more efficient in detecting NO instances. At the same time, both the propagation rule (used by BT) and the branch-pruning criterion (used by BnC) become stronger, reducing the number of possible nodes to investigate in order to arrive at a YES answer. In general, for every value of $n$ and irrespective of the algorithm used, the highest average CPU time is usually obtained for values of the parameter around the threshold value.

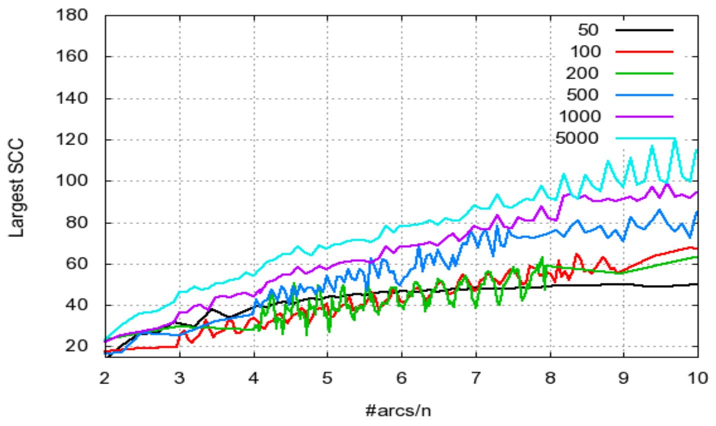


Figure 9: Average size of the largest strongly connected component as function of the parameter.

In order to further understand the difference in the behavior of the three algorithms, we plot in Figure 9 the average size of the largest SCC as function of the parameter. We observe that for a given value of the parameter, the average size of the largest SCC is a slowly increasing function of $n$ (the number of vertices). This may explain the fact that the BT and BnC are little affected by the value of $n$ compared to CycleId.

## 7. Summary and conclusions

This text studies the problem of coloring the vertices of a directed graph using two colors such that no monochromatic cycle occurs. We were motivated to consider this problem by an application in the study of rationality of consumption behavior in households with multiple members. We prove that the problem is NP-complete for arbitrary oriented graphs and that the existence of a constant-factor approximation algorithm is unlikely for an optimization formulation which maximizes the number of vertices that can be colored using two colors while avoiding monochromatic cycles. We present a integer-programming algorithm based on cycle identification, a backtracking algorithm and a branch-and-check algorithm to solve the problem exactly. We compare the three algorithms based on their CPU time, both on real-life instances and on random graphs. For the latter set, graphs with up to 5 000 vertices are solved in few seconds by every algorithm. We also study empirically the phase transition of the problem. We find that the acyclic 2-coloring problem exhibits an easy-hard-easy transition and that hard instances are difficult to generate. For real-life instances coming from the study of rationality of consumption behavior, all the instances are decided using every algorithm and the largest instance with 4384 vertices is solved using the backtracking algorithm in less than five minutes, while the branch-and-check algorithm spends about ten minutes to decide that instance and the cycle-identification algorithm about 30 minutes.

An important research direction that might be pursed in the future is the study of the acyclic 2-coloring problem for some special graphs, including directed planar graphs. Further, it might be interesting to investigate in more detail the optimization variants of the acyclic 2-coloring problem.

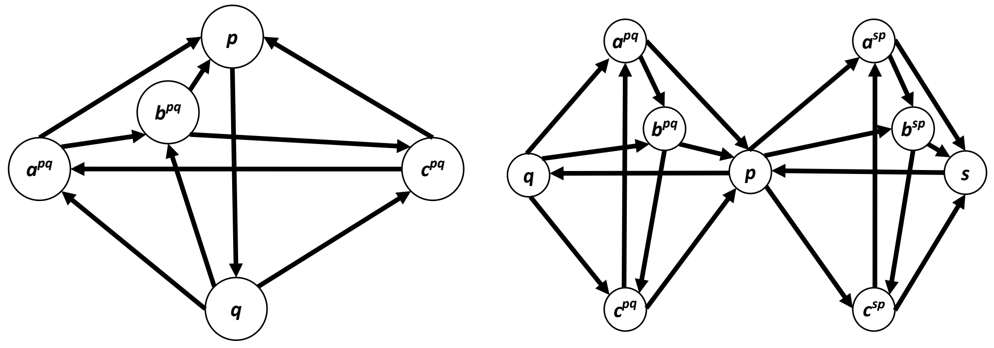
## Appendix: proof of Theorem 1

The proof is a refinement of Deb's proof [12] for arbitrary directed graphs $G$ to oriented graphs. It uses a reduction from the Not-All-Equal-3Sat problem defined as follows.
INSTANCE: Set $X = \{x_1, \ldots, x_{n^*}\}$ of $n^*$ variables, collection $C = \{C_1, \ldots, C_{m^*}\}$ of $m^*$ clauses over $X$ such that each clause $C_\ell \in C$ has $|C_\ell| = 3$, $\ell = 1, \ldots, m^*$.
QUESTION: Is there a truth assignment for $X$ such that each clause in $C$ has at least one true literal and at least one false literal?
Garey and Johnson [15] proved that the Not-All-Equal-3Sat problem is NP-complete.

The proof is structured as follows. First, we build an oriented graph $G = (V, A)$ given the instance of the Not-All-Equal-3Sat problem. Next, we argue the equivalence of a yes-instance of Not-All-Equal-3Sat and the oriented graph $G$ having a partition into two acyclic subgraphs.

In our construction of $G$, we use a gadget called a *p-q block*, which is a (sub)graph $\mathcal{B}_{pq} = (V_{pq}, A_{pq})$ with five vertices and ten arcs defined by: $V_{pq} = \{p, q, a^{pq}, b^{pq}, c^{pq}\}$ and $A_{pq} = \{pq,\ qa^{pq},\ qb^{pq},\ qc^{pq},\ a^{pq}p,\ a^{pq}b^{pq},\ b^{pq}p,\ b^{pq}c^{pq},\ c^{pq}p,\ c^{pq}a^{pq}\}$. The illustration of $\mathcal{B}_{pq}$ is depicted in Figure 10(a). In Figure 10(b), we draw two blocks sharing one vertex $p$; these are the $p$-$q$ block and the $s$-$p$ block. In the block $\mathcal{B}_{pq}$, the vertices $a^{pq}$, $b^{pq}$, $c^{pq}$ are called *block vertices* because they are used to build the block $\mathcal{B}_{pq}$. All the arcs in $A_{pq}$ are called *block arcs*. In our construction of the oriented graph $G$, there is no arc going from a block vertex $a^{pq}$, $b^{pq}$ or $c^{pq}$ to vertices other than $p$ and $q$.

Observe that in any feasible coloring of the block $\mathcal{B}_{pq}$, the vertices $p$ and $q$ must always have different colors. Indeed, if $p$ and $q$ are assigned the same color then the three block vertices $a^{pq}$, $b^{pq}$, and $c^{pq}$ all must have the same color and therefore will form a monochromatic cycle. To obtain a feasible coloring of $\mathcal{B}_{pq}$, it suffices to assign different colors to $p$ and $q$, and make sure that the block vertices $a^{pq}$, $b^{pq}$, and $c^{pq}$ do not have the same color.



(a) $p$-$q$ block (b) Two blocks sharing vertex $p$

Figure 10: Illustration of a single $p$-$q$ block and two blocks sharing one vertex.

In the first step of the proof, we aim at building an oriented graph $G = (V, A)$ from an arbitrary instance of the Not-All-Equal-3Sat problem. We first determine the set $V$ of vertices followed by the set $A$ of arcs.

Consider an arbitrary instance of the Not-All-Equal-3Sat problem. We build the set $V$ of vertices as follows. For each variable $x_i \in X$, we have five vertices: $x_i$, $\bar{x}_i$, $a^{x_i\bar{x}_i}$, $b^{x_i\bar{x}_i}$ and $c^{x_i\bar{x}_i}$, where the last three vertices are block vertices; they are used to build the block $\mathcal{B}_{x_i\bar{x}_i}$. Therefore, in our oriented graph there will not be an arc going from one of these three vertices to a vertex other than $x_i$ and $\bar{x}_i$. The vertices $x_i$ and $\bar{x}_i$ are called *variable* vertices. Hence, if $|X| = n^*$, we have $5n^*$ vertices corresponding to variables in the Not-All-Equal-3Sat instance. For each clause $C_\ell = \left(x_1^\ell \vee x_2^\ell \vee x_3^\ell\right) \in C$, we define 12 vertices among which nine block vertices. The three vertices $x_1^\ell$, $x_2^\ell$, and $x_3^\ell$ are called *literal* vertices. There are block vertices associated with $x_1^\ell$, $x_2^\ell$, and $x_3^\ell$, respectively. For the literal $x_1^\ell$ there is a variable $x_i \in X$ such that either $x_1^\ell = x_i$ or $x_1^\ell = \bar{x}_i$. On the one hand, if $x_1^\ell = x_i$ then using the block vertices $a^{x_1^\ell\bar{x}_i}$, $b^{x_1^\ell\bar{x}_i}$ and $c^{x_1^\ell\bar{x}_i}$, we build the block $\mathcal{B}_{x_1^\ell\bar{x}_i}$. On the other hand, if $x_1^\ell = \bar{x}_i$ then we use the block vertices $a^{x_1^\ell x_i}$, $b^{x_1^\ell x_i}$ and $c^{x_1^\ell x_i}$ to build the block $\mathcal{B}_{x_1^\ell x_i}$. The block vertices associated with the literal $x_2^\ell$ and $x_3^\ell$ are defined similarly. Notice that for each literal $x_r^\ell \in C^\ell$ $(r = 1, 2, 3)$ we have four vertices, namely the literal vertex $x_r^\ell$ and three block vertices. If there are $m^*$ clauses, we have $12m^*$ vertices coming from clauses. In total, the set $V$ contains $5n^* + 12m^*$ vertices.

To complete the definition of our oriented graph $G$, we now specify the set $A$ of arcs. We distinguish two types of arcs, depending on whether they are block arcs or not.

1. **Block arcs:** For each variable $x_i \in X$, there is a block $\mathcal{B}_{x_i\bar{x}_i}$, which requires ten block arcs. Hence, if $|X| = n^*$, we have $10n^*$ such block arcs. Further, for each clause $C_\ell = \big(x_1^\ell \vee x_2^\ell \vee x_3^\ell\big) \in C$ there are three blocks, one associated with each literal. Hence, for the $m^*$ clauses there are $30m^*$ block arcs.
2. **Other arcs:** For each clause $C_\ell = \big(x_1^\ell \vee x_2^\ell \vee x_3^\ell\big) \in C$ there are three arcs which are not block arcs. These are $x_1^\ell x_2^\ell$, $x_2^\ell x_3^\ell$, and $x_3^\ell x_1^\ell$, which form a cycle containing the literal vertices $x_1^\ell$, $x_2^\ell$, and $x_3^\ell$. Hence, for the $m^*$ clauses there are $3m^*$ such arcs.

In total, we have $|A| = 33m^* + 10n^*$. This completes the definition of our oriented graph $G$. Clearly, the above reduction can be done in polynomial time and the obtained graph is an oriented graph.

To illustrate the reduction, we consider the following example of the Not-All-Equal-3Sat problem. The set of variables is $X = \{x_1, x_2, x_3\}$, and there are two clauses $C_1 = (x_1 \vee x_2 \vee x_3)$ and $C_2 = (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$; that is $x_1^1 = x_1$, $x_2^1 = x_2$, $x_3^1 = x_3$, $x_1^2 = \bar{x}_1$, $x_2^2 = x_2$ and $x_3^2 = \bar{x}_3$. Notice that the truth assignment $x_1 = x_2 = 1$ and $x_3 = 0$ is a solution to this Not-All-Equal-3Sat instance. For this example, the set of vertices corresponding to variables is $\{x_1,\ \bar{x}_1,\ a^{x_1\bar{x}_1},\ b^{x_1\bar{x}_1},\ c^{x_1\bar{x}_1},\ x_2,\ \bar{x}_2,\ a^{x_2\bar{x}_2},\ b^{x_2\bar{x}_2},\ c^{x_2\bar{x}_2},\ x_3,\ \bar{x}_3,\ a^{x_3\bar{x}_3},\ b^{x_3\bar{x}_3},\ c^{x_3\bar{x}_3}\}$, and the set of vertices stemming from clauses is $\{x_1^1,\ a^{x_1^1\bar{x}_1},\ b^{x_1^1\bar{x}_1},\ c^{x_1^1\bar{x}_1},\ x_2^1,\ a^{x_2^1\bar{x}_2},\ b^{x_2^1\bar{x}_2},\ c^{x_2^1\bar{x}_2},\ x_3^1,\ a^{x_3^1\bar{x}_3},\ b^{x_3^1\bar{x}_3},\ c^{x_3^1\bar{x}_3},\ x_1^2,\ a^{x_1^2x_1},\ b^{x_1^2x_1},\ c^{x_1^2x_1},\ x_2^2,\ a^{x_2^2\bar{x}_2},\ b^{x_2^2\bar{x}_2},\ c^{x_2^2\bar{x}_2},\ x_3^2,\ a^{x_3^2x_3},\ b^{x_3^2x_3},\ c^{x_3^2x_3}\}$. The set $A$ of arcs obtained by the reduction contains the arcs $x_1^1x_2^1$, $x_2^1x_3^1$, $x_3^1x_1^1$, $x_1^2x_2^2$, $x_2^2x_3^2$, $x_3^2x_1^2$, which are not block arcs, and the block arcs used to build the blocks $\mathcal{B}_{x_1\bar{x}_1}$, $\mathcal{B}_{x_2\bar{x}_2}$, $\mathcal{B}_{x_3\bar{x}_3}$, $\mathcal{B}_{x_1^1\bar{x}_1}$, $\mathcal{B}_{x_2^1\bar{x}_2}$, $\mathcal{B}_{x_3^1\bar{x}_3}$, $\mathcal{B}_{x_1^2x_1}$, $\mathcal{B}_{x_2^2\bar{x}_2}$, and $\mathcal{B}_{x_3^2x_3}$.

In the last step of our proof, we show that the oriented graph $G$ obtained by the above reduction can be partitioned into two acyclic subgraphs if and only if the instance of the Not-All-Equal-3Sat problem is a YES instance. The goal here is to prove that partitioning the oriented graph $G$ built from the instance of the Not-All-Equal-3Sat problem into two acyclic subgraphs is at least as hard as that instance of the Not-All-Equal-3Sat problem.

$\Rightarrow\big)$ If the graph $G$ can be vertex-partitioned into two acyclic subgraphs $G_1$ and $G_2$, then for each variable $x_i \in X$, if the associated vertex $x_i \in G_1$, then we set the variable $x_i = 1$; otherwise $x_i = 0$. This is a truth assignment for $X$ since each variable in $X$ receives either value 0 or value 1. We now prove that this truth assignment is such that each clause in $C$ has at least one true literal and at least one false literal. We argue by contradiction. Suppose that there exists a clause $C_\ell = (x_1^\ell \vee x_2^\ell \vee x_3^\ell)$ $(\ell \in \{1, \ldots, m^*\})$ in $C$ which is such that either $x_1^\ell = x_2^\ell = x_3^\ell = 1$ or $x_1^\ell = x_2^\ell = x_3^\ell = 0$. Without loss of generality, let us assume that $x_1^\ell = x_2^\ell = x_3^\ell = 1$. We are going to investigate each literal in $C_\ell$ individually. The first literal $x_1^\ell$ is either $x_i$ or $\bar{x}_i$ for a given variable $x_i \in X$. We will argue that in both cases, the associated vertex, $x_1^\ell$, belongs to $G_1$.

On the one hand, if the literal $x_1^\ell = x_i$ then the variable $x_i = 1$. This implies, from the assignment of values to variables, that the associated vertex $x_i \in G_1$. In the construction of $G$, there is a block $\mathcal{B}_{x_i\bar{x}_i}$ which makes sure that the vertices $x_i$ and $\bar{x}_i$ are not in the same subgraph. Since the vertex $x_i \in G_1$ this implies that the vertex $\bar{x}_i \in G_2$. Next, the presence of the block $\mathcal{B}_{x_1^\ell\bar{x}_i}$ in $G$ (which exists by construction) and the fact that the vertex $\bar{x}_i \in G_2$ imply that the vertex $x_1^\ell \in G_1$.

On the other hand, if $x_1^\ell = \bar{x}_i$ then $\bar{x}_i = 1$ implies that the variable $x_i = 0$ and hence the associated vertex $x_i \in G_2$. The block $\mathcal{B}_{x_1^\ell x_i}$ in $G$ and the fact that the vertex

$x_i \in G_2$ imply that the vertex $x_1^\ell \in G_1$.

We conclude that whether the literal $x_1^\ell$ is the variable $x_i$ or its negation $\bar{x}_i$, as long as its value equals 1 the associated vertex $x_1^\ell \in G_1$. Notice that for case $x_1^\ell = 0$ we would conclude that the vertex $x_1^\ell \in G_2$.

By applying a similar reasoning to the literal $x_2^\ell$, we obtain that the associated vertex $x_2^\ell \in G_1$, while the application of that reasoning to the literal $x_3^\ell$ leads to $x_3^\ell \in G_1$. We obtain that the vertices $x_1^\ell \in G_1$, $x_2^\ell \in G_1$ and $x_3^\ell \in G_1$; This implies that $G_1$ contains the cycle $[x_1^\ell, x_2^\ell, x_3^\ell]$. This contradicts the hypothesis that $G_1$ is acyclic.

$\Leftarrow\big)$ Conversely, suppose that there is a truth assignment for $X$ which is such that each clause in $C$ has at least one true literal and at least one false literal. Consider the subgraphs $G_1$ and $G_2$ defined as follows. For each variable $x_i \in X$, if $x_i = 1$ then the variable vertex $x_i \in G_1$ and the variable vertex $\bar{x}_i \in G_2$. Otherwise, if the variable $x_i = 0$ then the vertex $\bar{x}_i \in G_1$ and the vertex $x_i \in G_2$. Further, for the block vertices (used to build the block $\mathcal{B}_{x_i\bar{x}_i}$) we make sure that they are not all three in the same subgraph. For example, we may put the block vertex $a^{x_i\bar{x}_i} \in G_1$, the block vertex $b^{x_i\bar{x}_i} \in G_1$ and the block vertex $c^{x_i\bar{x}_i} \in G_2$. This ensures that each vertex coming from a variable in $X$ is either in $G_1$ or in $G_2$. We now deal with vertices stemming from clauses.

Let us consider the vertices coming from a clause $C_\ell = (x_1^\ell \vee x_2^\ell \vee x_3^\ell)$ $(\ell \in \{1, \ldots, m^*\})$ in $C$. We deal with each literal vertex separately. The first literal vertex, $x_1^\ell$, is associated with the first literal $x_1^\ell$ in $C_\ell$, the latter is either $x_i$ or $\bar{x}_i$ $(i \in \{1, \ldots, n\})$. Since the corresponding variables vertices ($x_i$ and $\bar{x}_i$) are either in $G_1$ or in $G_2$, we proceed as follows. If the literal $x_1^\ell = x_i$ then we put the literal vertex $x_1^\ell$ in the same subgraph as the variable vertex $x_i$. Otherwise (the literal $x_1^\ell = \bar{x}_i$), the literal vertex $x_1^\ell$ is in the same subgraph as the variable vertex $\bar{x}_i$. In each case, the block vertices (used in the block $\mathcal{B}_{x_1^\ell\theta}$, where $\theta$ is either $x_i$ or $\bar{x}_i$) are distributed in such a way that they are not all three in the same subgraph as sketched before. A similar distribution is done for the literal vertex $x_2^\ell$ and for the literal vertex $x_3^\ell$. This completes the definition of $G_1$ and $G_2$. Clearly $G_1$ and $G_2$ form a partition of $G$ since each vertex in $G$ is either in $G_1$ or in $G_2$.

We now prove that $G_1$ and $G_2$ are acyclic. We also argue by contradiction. Suppose, without loss of generality, that $G_1$ contains a cycle. Notice that this cycle cannot contain both $p$ and $q$ for any $p$-$q$ block present in $G$. Further, that cycle cannot be contained in two blocks sharing one vertex. Therefore, if there is a cycle in $G_1$ then there exists a clause $C^\ell \in C$ such that the cycle uses the literal vertices $x_1^\ell$, $x_2^\ell$, and $x_3^\ell$. Therefore, $x_1^\ell$, $x_2^\ell$, $x_3^\ell \in G_1$ and hence all the literals of the clause $C^\ell$ have the same value. This contradicts the fact that the truth assignment for $X$ was such that each clause of $C$ has at least one false and at least one true literal.

This concludes the proof that the instance of the Not-All-Equal-3Sat is a YES instance only if the oriented graph $G$ can be partitioned into two acyclic subgraphs, and hence completes the proof of Theorem 1.

## References

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] Y. Aifeng and Y. Jinjiang. On the vertex arboricity of planar graphs of diameter two. *Discrete Mathematics*, 307:2438–2447, 1991.

[3] V.E. Alekseev, A. Farrugia, and V.V. Lozin. New results on generalized graph coloring. *Discrete Mathematics and Theoretical Computer Science*, 6:215–222, 2004.

[4] N. Apollonio and P.G. Franciosa. A characterization of partial directed line graphs. *Discrete Mathematics*, 307:2598–2614, 2007.

[5] T. Bartnicki, J.A. Grytczuk, and H.A. Kierstead. The game of arboricity. *Discrete Mathematics*, 308:1388–1393, 2008.

[6] T.J.M. Bench-Capon. Value-based argumentation frameworks. *In Proceedings of Non Monotonic Reasoning 2002*, pages 444–453, 2002.

[7] H.J. Broersma, F.V. Fomin, J. Kratochvil, and G.J. Woeginger. Planar graph coloring avoiding monochromatic subgraphs: Trees and paths make it difficult. *Algorithmica*, 4:343–361, 2006.

[8] Z. Chen. Efficient algorithm for acyclic colorings of graphs. *Theoretical Computer Science*, 230:75–95, 2000.

[9] L. Cherchye, B. De Rock, and F. Vermeulen. The collective model of household consumption: a nonparametric characterization. *Econometrica*, 75:553–574, 2007.

[10] L. Cherchye, B. De Rock, J. Sabbe, and F. Vermeulen. Nonparametric tests of collectively rational consumption behavior: an integer programming procedure. *Journal of Econometrics*, 147:258–265, 2008.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2005.

[12] R. Deb. Acyclic partitioning problem is NP-complete for $k = 2$. Private communication, Yale University, United States, 2008.

[13] R. Deb. An efficient nonparametric test of the collective household model. Working paper, Yale University, United States, 2008.

[14] G. Farin. Class A Bézier curves. *Computer Aided Geometric Design*, 23:573–581, 2006.

[15] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[16] W. Goddard. Acyclic colorings of planar graphs. *Discrete Mathematics*, 91:91–94, 1991.

[17] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81:127–154, 1985.

[18] S. Khanna and K. Kumaran. On wireless spectrum estimation and generalized graph coloring. *Proceedings of INFOCOM'98*, pages 1273–1283, 1998.

[19] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. *Lecture Notes in Computer Science*, 700:40–51, 1993.

[20] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.

[21] A. Raspaud and W. Wang. On the vertex-arboricity of planar graphs. *European Journal of Combinatorics*, 29:1064–1075, 2008.

[22] A. Roychoudhury and S. Sur-Kolay. Efficient algorithms for vertex arboricity of planar graphs. *Lecture Notes in Computer Science*, 1026:37–51, 1995.

[23] F. Talla Nobibon, L. Cherchye, B. De Rock, J. Sabbe, and F.C.R. Spieksma. Heuristics for deciding collectively rational consumption behavior. *Computational Economics*, 2010, DOI 10.1007/s10614-010-9228-9.

[24] F. Talla Nobibon and F.C.R. Spieksma. On the complexity of testing the collective axiom of revealed preference. *Mathematical Social Sciences*, 60:123–136, 2010.

[25] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 2:146–160, 1972.

[26] C. Thomassen. 2-list-coloring planar graphs without monochromatic triangles. *Journal of Combinatorial Theory*, 98:1337–1348, 2008.

[27] E.S. Thorsteinsson. Branch-and-check: a hybrid framework integrating mixed integer programming and constraint logic programming. *Lecture Notes in Computer Science*, 2239:16–30, 2001.

[28] H. Varian. Revealed preference. *Samuelsonian Economics and the 21st Century*, 2006.

[29] Y. Wu, J. Yuan, and Y. Zhao. Partition a graph into two induced forests. *Journal of Mathematical Study*, 1:1–6, 1996.