INFORMS Journal on Computing
Vol. 24, No. 3, Summer 2012, pp. 485–499
ISSN 1091-9856 (print) | ISSN 1526-5528 (online)

http://dx.doi.org/10.1287/ijoc.1110.0466

Coloring Graphs Using Two Colors While Avoiding Monochromatic Cycles

Fabrice Talla Nobibon
QuantOM, HEC-Management School, University of Liège, B-4000 Liège, Belgium, fabrice.tallanobibon@ulg.ac.be

Cor A. J. Hurkens
Department of Mathematics and Computer Science, Eindhoven University of Technology,
5600 MB Eindhoven, The Netherlands, wscor@win.tue.nl

Roel Leus, Frits C. R. Spieksma
Operations Research Group, University of Leuven, B-3000 Leuven, Belgium
{roel.leus@econ.kuleuven.be, frits.spieksma@econ.kuleuven.be}

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 microeconomics. 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 that 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 a few seconds; however, the CPU time of the integer-programming algorithm increases with the number of vertices in the graph more clearly than 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, whereas the branch-and-check algorithm takes approximately 10 minutes and the backtracking algorithm less than 5 minutes. Finally, for every algorithm, we also study empirically the transition from a high to a low probability of a YES answer as a function of the number of arcs divided by the number of vertices.

Key words: directed graph; undirected graph; bipartite graph; acyclic graph; phase transition; NP-complete
History: Accepted by Karen Aardal, Area Editor, Design and Analysis of Algorithms; received July 2010; revised December 2010; accepted March 2011. Published online in *Articles in Advance* August 25, 2011.

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. Because 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 be NP-complete (Wu et al. 1996). 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 that maximizes the number of vertices that can be colored using two colors while avoiding monochromatic cycles. Furthermore, 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 the *cycle-identification algorithm*), a backtracking algorithm, and a branch-and-check algorithm. We compare these algorithms based on their CPU times, both on real-life instances coming from microeconomics and on randomly generated graphs. We find that every algorithm solves random graphs

of considerable size within a 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 and branch-and-check algorithms. Furthermore, for every algorithm we study empirically the phase transition of the problem as a function of the number of arcs divided by the number of vertices. When applying the three algorithms to real-life instances stemming from a microeconomics 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, whereas the branch-and-check algorithm solves that instance in about 10 minutes and the backtracking algorithm in less than 5 minutes.

The contributions of this paper include the following:

(1) The proof of the complexity status of the acyclic 2-coloring problem for oriented graphs and the establishment of the nonapproximability of the optimization formulation that 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.

The rest of this paper is organized as follows. In §2, we motivate this problem and present a brief literature review. In §3, we prove the complexity and the nonapproximability results and present some properties of the acyclic 2-coloring problem. In §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 §6, we comment computational results and study empirically the phase transition of the problem. We conclude in §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 microeconomics. We now shortly elaborate on this application. Suppose that there is an economy with k goods and that we are given a data set S consisting of l observations. Each observation i consists of a pair (p^i, x^i) of (positive) prices $p^i = (p^i_1, \ldots, p^i_k)$ and (nonnegative) quantities (also called *bundle*) $x^i = (x^i_1, \ldots, x^i_k)$, with $i = 1, \ldots, l$. 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 data set 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 $p^i x^i \geq p^i x$.

The notion of preference has allowed economic theory to develop a number of properties that reflect rationality of the data set (see Varian 2006 for an overview). As example of such a property, we mention the strong axiom of revealed preference (SARP); a data set 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 $i, j, \ldots, r$ such that $p^s x^s \geq p^s x^i$, $p^i x^i \geq p^i x^j, \ldots, p^r x^r \geq p^r x^t$, 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 $x^j, \ldots,$ of x^r over x^t (and we say that x^s is preferred over x^t), whereas 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 data set S satisfies SARP. It has been shown (Varian 2006) that this question can be answered using graph theory. A directed graph G with l vertices is built by considering each observation i as a vertex. Furthermore, there is an arc from vertex i to vertex j if and only if $p^i x^i \geq p^i x^j$. The data set 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 (Cherchye et al. 2007). Deb (2008a) 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 (2010). To find out whether a given data set satisfies CARP, integer-programming models are proposed in Cherchye et al. (2008), and heuristic approaches, based on acyclic 2-coloring problems for specific directed

graphs, are described in Talla Nobibon et al. (2010b). 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 the start and end vertices are the same. For a vertex $p \in V$, the *outdegree* of p is the number of arcs leaving p, and 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 a 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 an *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 $[v_0, v_1, \ldots, v_l]$ is called a *chain* of length l if $v_{i-1}v_i \in A$ or $v_iv_{i-1} \in A$ for $i = 1, \ldots, l$. 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 $[v_0, v_1, \ldots, v_l]$ is called a *path* from v_0 to v_l if $v_{i-1}v_i \in A$ for $i = 1, \ldots, l$. A *vertex-induced subgraph* (subsequently called an *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 $[v_0, v_1, \ldots, v_l, v_0]$ is called a *cycle* of length $l + 1$ in $G = (V, A)$ if $v_{i-1}v_i \in A$ for $i = 1, \ldots, l$ and $v_lv_0 \in A$. 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 $[v_0, v_1, \ldots, v_l, v_0]$ in G is *monochromatic* if there exists $i \in \{1, \ldots, k\}$ such that $v_0, v_1, \ldots, v_l \in V_i$. 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 (2008a, b) 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 (2000) for undirected graphs by computing an upper bound on the acyclic chromatic number $a(G)$. Talla Nobibon et al. (2010b) 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. (1996) study the partition of a graph into two induced forests. Thomassen (2008) studies 2-list-coloring planar graphs without monochromatic triangles. Broersma et al. (2006) investigate the coloring problem on planar graphs while avoiding monochromatic subgraphs. Several authors have studied the acyclic coloring problem for planar graphs (Goddard 1991, Raspaud and Wang 2008, Aifeng and Jinjiang 1991, Roychoudhury and Sur-Kolay 1995). For a general k, Chen (2000) 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 (Alekseev et al. 2004). Applications of acyclic k-coloring for undirected graphs include wireless spectrum estimation (Khanna and Kumaran 1998), game theory (Bartnicki et al. 2008), and logic (Bench-Capon 2002).

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 that 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 §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; see Ahuja et al. 1993). 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 the Online Supplement (available at http://dx.doi.org/10.1287/ijoc.1110.0466). □

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 as 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 (1993) prove a nonapproximability 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^{ϵ} 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, let G_2 be its associated 2-undirected graph, and let G_s be 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, and the NO part is straightforward. □

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

Proof. The proof is straightforward. □

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 exist 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 and B&C algorithms are implicit enumeration algorithms built to solve the acyclic 2-coloring problem, whereas the cycle-identification

algorithm is based on an IP formulation of the problem. We also present two dominance rules that 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 1 if vertex i is colored red and 0 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 §6.2. To complete the IP formulation, we add for each cycle $\mathscr{C}$ in G the pair of constraints $1 \le \sum_{i \in \mathscr{C}} x_i \le |\mathscr{C}| - 1$, where $|\mathscr{C}|$ is the number of vertices in $\mathscr{C}$. 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 a 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 that 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 $RED = \varnothing$ and $BLUE = \varnothing$ 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. Furthermore, 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 that 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 (Thorsteinsson 2001).

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 $A \backslash \{pq\}$ 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: $p(pq_1)$, $(pq_1)p$, $q(pq_1)$, $(pq_1)q$, $(pq_1)(pq_2)$, and $(pq_2)(pq_1) \in A'$.

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. Stated 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)). Furthermore, we add a node (pq_1) in F^{pq} to enforce that the vertices p, q, and (pq_2) have the same color. Note 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, whereas (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 $\mathscr{C}$ in F^{pq}. $\mathscr{C}$ cannot contain (pq_1) because all its neighbors have a different color. $\mathscr{C}$ 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 $\mathscr{C}$ into cycle $\mathscr{C}'$ by replacing $x \to (pq_2) \to y$ by $x \to p \to q \to y$. This would be a monochromatic cycle in G.

($\Rightarrow$) Suppose that F^{pq} or H^{pq} can be partitioned into two acyclic subgraphs. Clearly, a partition of H^{pq} into

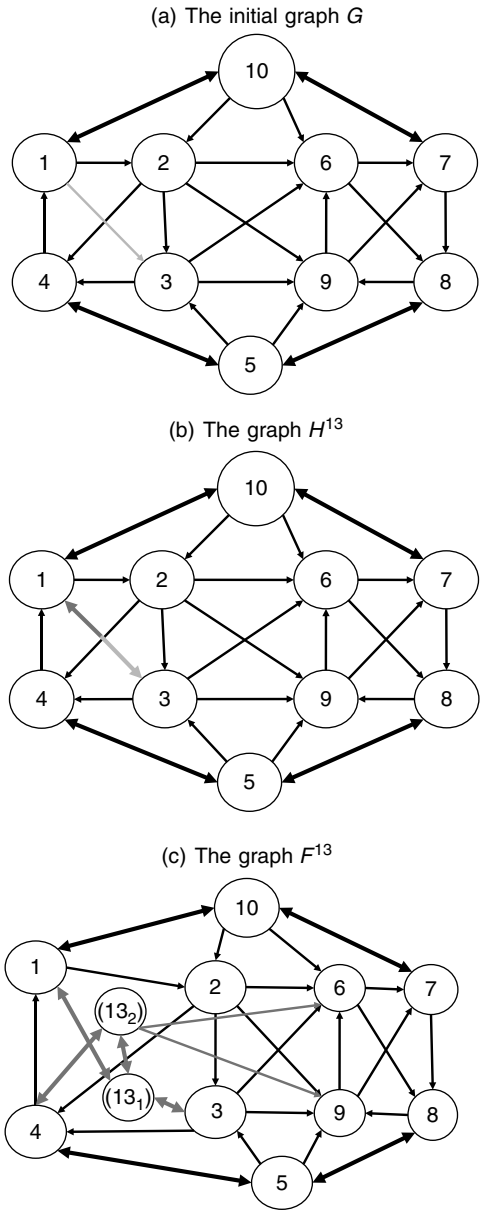

Figure 1 Illustration of the Construction of H^{13} and F^{13}

Note. In the graphs, a double-direction arc ($\leftrightarrow$) represents a cycle of length two between the considered vertices.

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 $G\backslash\{pq\}$ (the graph G minus the arc pq) into two acyclic subgraphs because $G\backslash\{pq\}$ is a subgraph of F^{pq}. Consequently, if there is a monochromatic cycle $\mathscr{C}$ in G, then $\mathscr{C}$ 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.

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})

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 cycles* in G_s, denoted by $L(G)$, 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 thus 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 Propositions 5, 7, and 8. □

Example 2. Figure 2 illustrates the application of the B&C algorithm. The initial graph G, shown in 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}, shown in 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 in 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}. By selecting the graph H^{68}, given in Figure 2(d), the associated 2-undirected graph depicted in Figure 2(e) is bipartite, and the single directed H_s^{68} depicted in 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 that 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 indegree of p is less than or equal to* 1, *then the vertex p can be removed from G without changes in the final outcome.*

Proof. Let $G = (V, A)$ be the directed graph, and let p be a vertex of G with an outdegree or indegree less than or equal to 1. 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 0, 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 1, 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 because 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

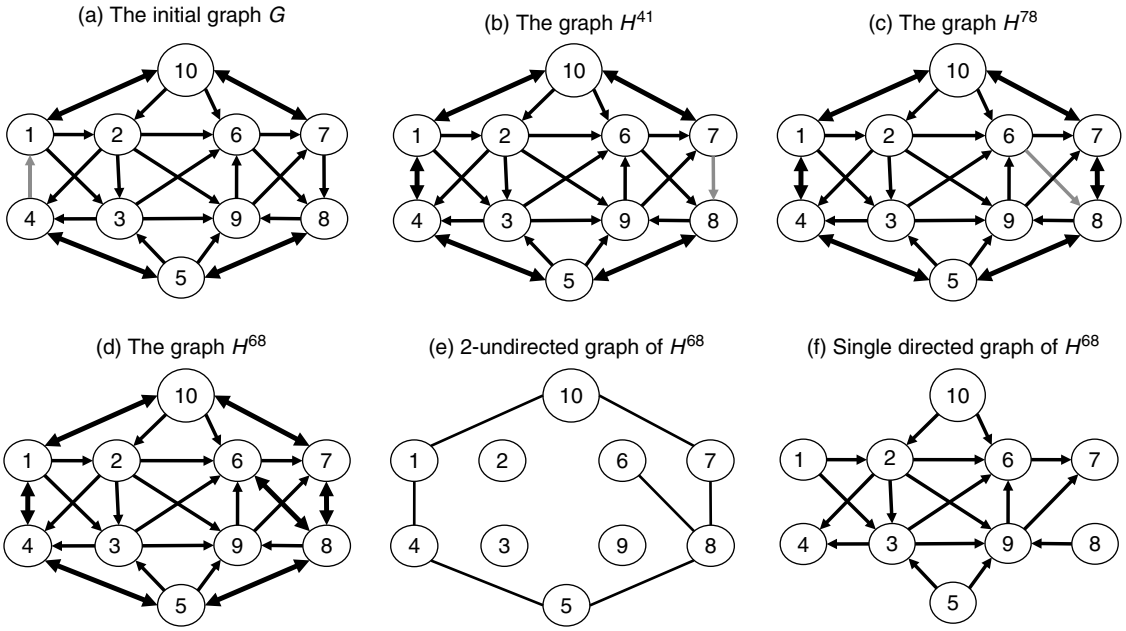

Figure 2 Illustration of the B&C Algorithm

one vertex, then the repeated application of the other rule may further remove additional arcs or vertices. For both the cycle-identification and the backtracking algorithms, 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 (DAGs). The second class of graphs is the class of line graphs (LGs). Talla Nobibon et al. (2010b) 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., Apollonio and Franciosa (2007). 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. Considering that a line graph is a YES instance of the acyclic 2-coloring problem and 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 a degree of at most i; at least one vertex has a 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.*

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

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

(b) *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 (Raspaud and Wang 2008) and for undirected outerplanar graphs (Aifeng and Jinjiang 1991, Goddard 1991). □

5. Implementation Issues

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

5.1. Bipartiteness, Acyclicness, and Strongly Connected Components

An adapted breadth-first-search algorithm (Cormen et al. 2005) 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 (Ahuja et al. 1993) is used for testing the acyclicness of G_s and any induced subgraph $G_s(A)$, where A is a subset of vertices. Tarjan's (1972) algorithm is used to identify the SCCs of a given graph.

5.2. 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 for a monochromatic cycle only in the single directed graphs induced by the color classes. Given a color class, we use the Floyd-Warshall algorithm (Ahuja et al. 1993, Cormen et al. 2005) to find (if there exist) monochromatic cycles that 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 mixed-integer programming solver of CPLEX; once a feasible solution is found, we stop the solver.

5.3. Backtracking Algorithm

5.3.1. 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, and the second choice is an uncolored vertex with the highest degree; ties are broken arbitrarily.

5.3.2. 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 that 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.

5.3.3. 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.

5.4. Branch-and-Check Algorithm

5.4.1. Branching Strategy. This branching strategy selects a single arc pq that is such that there is a cycle in G_s containing that arc. Before 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, $G_2^2, \ldots, G_2^l$ of G_2, assuming that it has l such components. Next, because 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 ($i = 1, 2, \ldots, l$), 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.

Our branching strategy is such that an arc pq occurs at most once for branching. Assuming that this property holds, an alternative construction for $F^{pq} = (V', A')$ (which we did not use in our implementation) is the following. We set $V' = V \cup \{(pq_1)\}$ and $A' = A \backslash \{pq\} \cup \{p(pq_1), q(pq_1), (pq_1)p, (pq_1)q\}$. Notice that the correctness of this construction depends on the branching strategy.

In case 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 $|V'| = |V| - 1$, and 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. We used this simplified construction of F^{pq} in our implementation.

5.4.2. 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 that 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.

5.4.3. 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 a Pentium® processor with 3.16 GHz clock speed and 3.21 GB of 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 data sets, and we subsequently discuss the computational results.

6.1. Data

The three algorithms were tested both on real-life graphs stemming from a microeconomics application and on randomly generated graphs. We first present the real-life instances, and we next 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 in this section come from the study of rationality of consumption behavior described in §2. We refer to Cherchye et al. (2008) for more details about the data sets containing the prices and quantities describing the expenditures of the household and to Talla Nobibon et al. (2010b) for the translation of those data sets into directed graphs. Table 1 reports the 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 the instances as much as possible, 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 $n = 1{,}000$; the step size for subsequent values of n is 0.15%. For $n = 5{,}000$ the lower density is 0.05, and the step size 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.

Table 2 Densities of the Graphs Generated in the Second Phase

	Density (D)			
n	From (%)	To (%)	Step (%)	Total
50	8	15.75	0.250	32
100	3.05	8.95	0.050	119
200	2.01	3.99	0.010	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

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 and NO instances exist. For a given n, we generate additional graphs with the densities given in Table 2.

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 examine different implementations of every algorithm for the set of 50-vertex graphs generated during the first phase (these are 400 graphs in total) in §6.2.1. The three algorithms are subsequently compared based on their best (chosen) implementation both on randomly generated graphs (see §6.2.2) and on real-life instances (see §6.2.3). In §6.2.4, we study empirically the phase transition (Hogg 1985, Monasson et al. 1999) of the acyclic 2-coloring problem as a function of the number of arcs divided by n. Throughout this section, the CPU time is expressed in seconds.

Table 1 Properties of the Real-Life Instances

Instance	1	2	3	4	5	6	7	8	9	10	11	12
No. of vertices (n)	22	48	68	95	118	139	226	279	294	410	755	4,384
No. of arcs (m)	53	169	297	513	699	985	1,979	2,012	2,427	3,660	10,113	124,321
No. of 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

6.2.1. Implementation Used for Every Algorithm. Different implementations of each algorithm have been compared. In this section, we describe the particular implementations that have been chosen for the experiments.

The implementation of the cycle-identification algorithm is the following: First, the dominance rules are applied to reduce the size of the initial graph. At each iteration, after solving the IP problem, we search monochromatic cycles in both color classes. If they exist, we add all the pairs of constraints corresponding to cycles of length 3 in each color class. If there is no such cycle, we search and (if found) add all the pairs of constraints corresponding to cycles of length 4 in each color class. If there is no cycle of length 4, then we add the pair of constraints corresponding to one monochromatic cycle of any length found in any color class, starting with the class of red vertices.

Other implementations investigated include 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. We have also 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 $\{-1, 1\}$). These implementations, however, performed worse than the implementation described above. We refer to Talla Nobibon et al. (2010a) for more details.

For the backtracking algorithm, we have chosen the implementation that starts by applying dominance rules to reduce the initial graph, uses the propagation rule at each iteration, and branches on an uncolored vertex with the highest degree. In Talla Nobibon et al. (2010a), we try three other implementations that do not output better results.

The B&C algorithm is implemented as follows: the dominance rules are applied once at the root node to reduce the size of the initial graph; the branch-pruning criterion is used; and for branching, we choose an arc pq with vertex p having the highest degree possible. This implementation was the best of the six different implementations studied in Talla Nobibon et al. (2010a).

In Talla Nobibon et al. (2010a), we plot the variance of the average CPU time 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; furthermore, the value of these high variances is several orders of magnitude greater than that of the corresponding average CPU times. It turns out that among the instances generated, only a few require the algorithm to run for more than a fraction of a second. 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 3 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 3(a) shows the average CPU time for the 50-vertex graphs. BnC usually reports a higher CPU time than the other algorithms. However, the highest average CPU time is less than 1.2 seconds. CycleId usually uses, on average, the smallest CPU time. For 100-vertex graphs (Figure 3(b)), we see that the average CPU time of CycleId is usually between that of BnC and that of BT, with BT using, in most cases, the smallest average time. For the large graphs (with more than 100 vertices; see Figures 3(c)–3(f)), the average CPU time reported for CycleId increases with the value of n, whereas those of BnC and BT are stable, comparable, and usually below 1 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.

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 BT reports the best CPU time for 5 instances out of 12, whereas CycleId achieves the best CPU time for 6 instances, and BnC has the best CPU time for 9 instances. For the largest instance with 4,384 vertices, however, BT spends less than 5 minutes, compared to approximately 10 minutes for BnC and approximately 30 minutes for CycleId.

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

Figure 4 presents the probability of a YES answer as well as the average CPU time of every algorithm as function of the parameter. Figure 4(a) shows the probability of a YES answer as a function of the parameter. The plots in Figure 4(a) are Bézier approximations (Farin 2006) 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, where the value of the

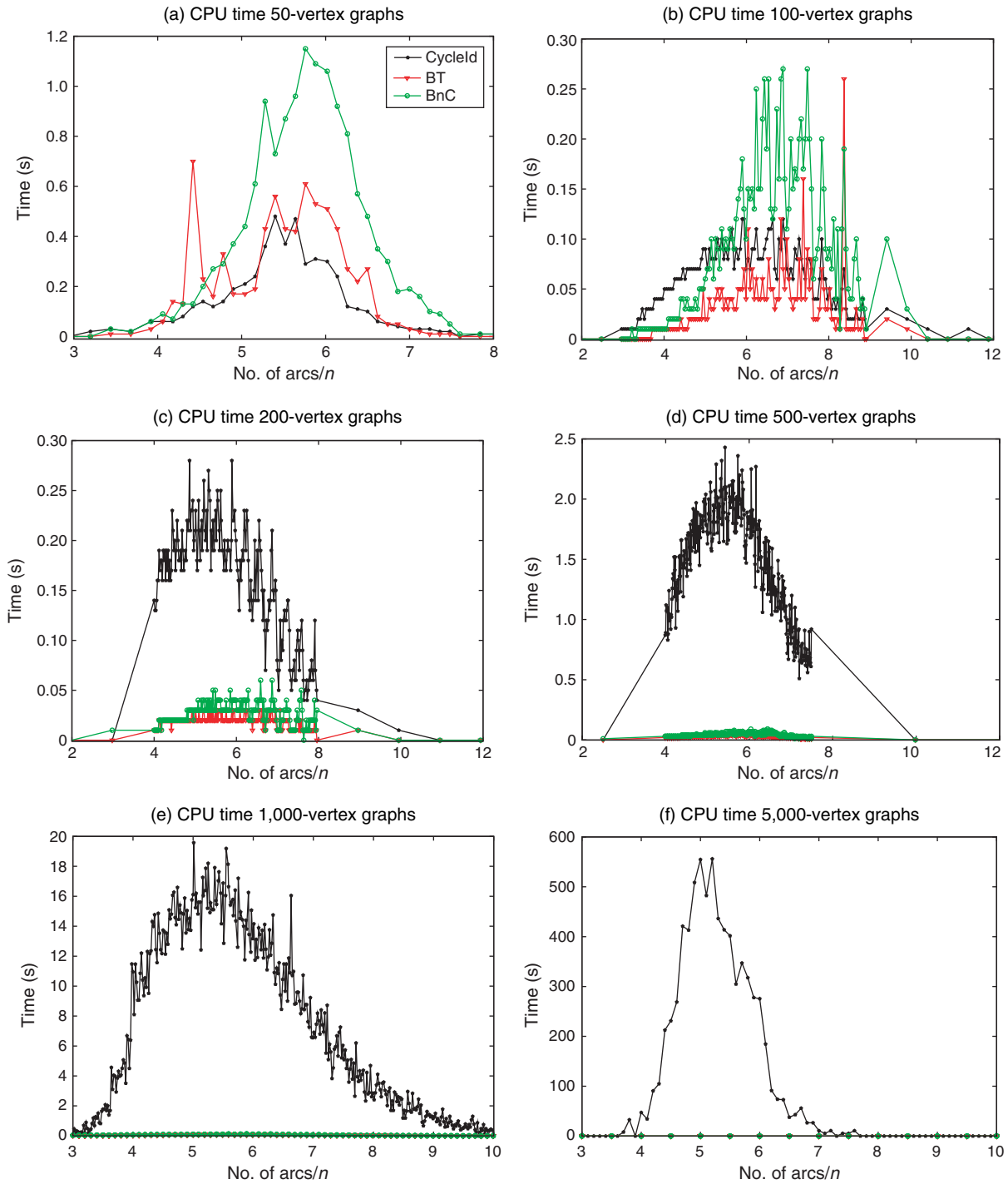

Figure 3 **Average CPU Time of Every Algorithm for Random Graphs**

Table 3 **CPU Time of Every Algorithm for the Real-Life Instances**

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	1,812.24
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

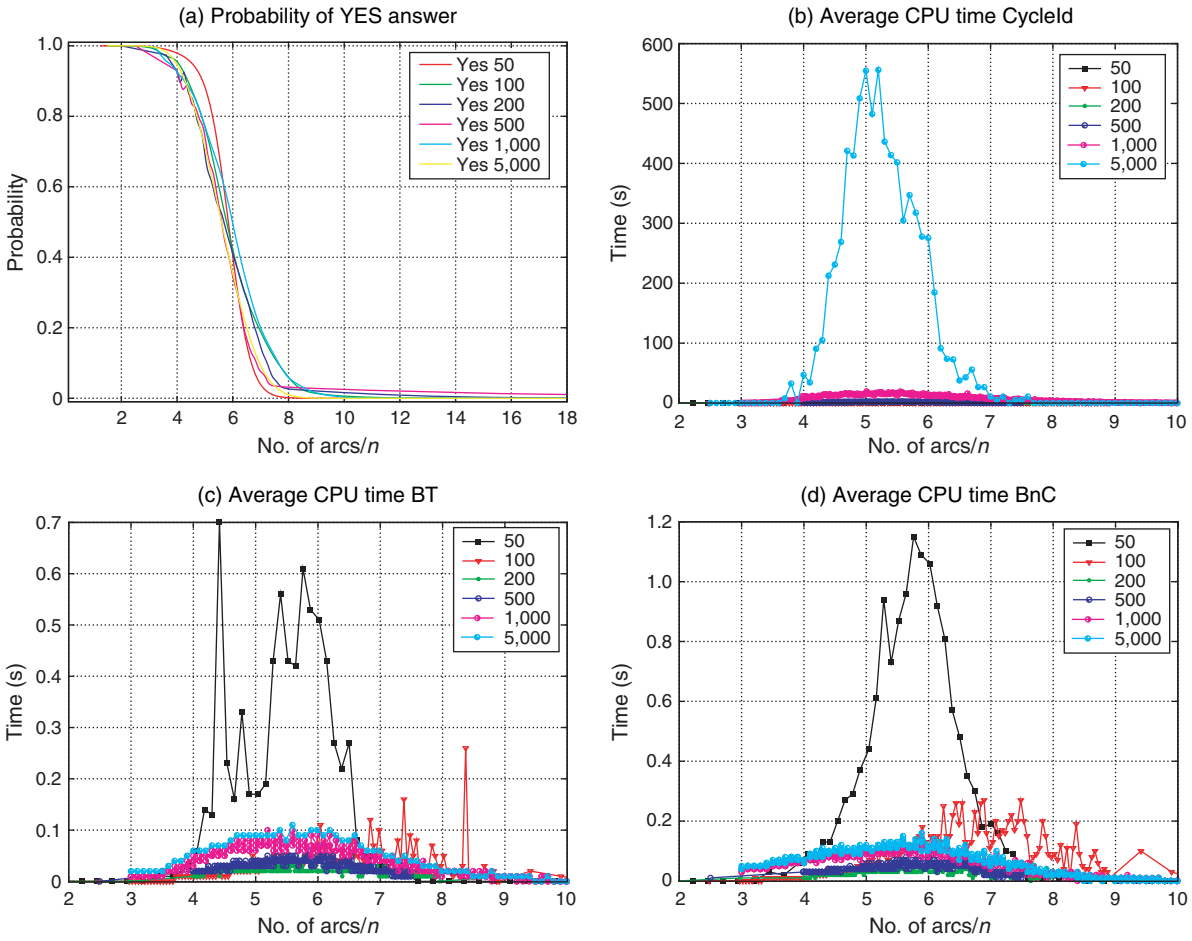

Figure 4 Probability of a YES Answer and Average CPU Time of Every Algorithm

parameter is between 3 and 8, is called *critical interval* and contains classes of graphs for which both YES and NO instances are present. The last region, where the value of the parameter is greater than 8, contains graphs for which the probability of YES is almost 0. Overall, we note that the five plots are similar and that the *threshold* value of the parameter for which the probability of YES answer is equal to 0.5 is almost the same for every n and is close to 5.75.

The plots in Figures 4(b)–4(d) are obtained using the data that were used to generate the plots in Figure 3, but here, the plots are grouped by algorithm. Figure 4(b) plots the average CPU time of CycleId for every value of n. The plots reflect the three regions described above. For the first and third regions, the average CPU time is very close to 0, whereas in the critical interval, we have a nonnegligible CPU time, showing an easy–hard–easy transition. Furthermore, CycleId has an average CPU time that increases with the value of n, which probably occurs simply because when n increases, the IP instance becomes more difficult to solve. Figure 4(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- 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 because for very few instances, the algorithm requires more than 1 second to decide. In other words, among the instances generated, there are very few hard instances. In Figure 4(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.

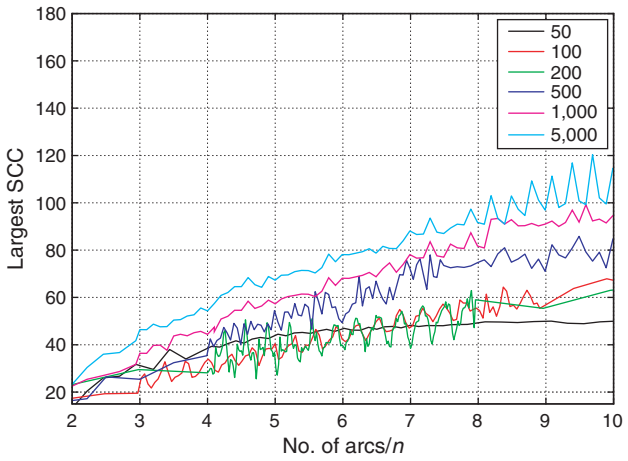

Figure 5 **Average Size of the Largest Strongly Connected Component as a Function of the Parameter**

To further understand the difference in the behavior of the three algorithms, we plot in Figure 5 the average size of the largest SCC as a function of the parameter. Observe that our parameter equals $D(n-1)$, which can be seen as the expected degree of a vertex. This parameter is used by Karp (1990) to study the size of the strongly connected components in random directed graphs where (in contrast to our setting) loops are allowed. In line with Karp's theoretical findings, we find that almost all strongly connected components consist of a single vertex. We also 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 why BT and BnC are little affected by the value of n compared with CycleId. We mention, however, that the average size of the largest SCC obtained in our experiements is usually smaller than the theoretical figure provided by Karp (1990). Perhaps this is caused by the fact that we ensure that our instances are connected and cyclic.

7. Summary and Conclusions

This paper 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 that 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 a 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 4,384 vertices is solved using the backtracking algorithm in less than 5 minutes, whereas the branch-and-check algorithm spends approximately 10 minutes to decide that instance, and the cycle-identification algorithm spends approximately 30 minutes.

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

Electronic Companion

An electronic companion to this paper is available as part of the online version at http://dx.doi.org/10.1287/ijoc.1110.0466.

Acknowledgments

The authors thank the referees and the associate editor for their constructive comments, and in particular for suggesting the alternative branching strategy. A preliminary version of some of the results of this paper appeared in Talla Nobibon et al. (2010a).

References

Ahuja, R. K., T. L. Magnanti, J. B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Upper Saddle River, NJ.

Aifeng, Y., Y. Jinjiang. 1991. On the vertex arboricity of planar graphs of diameter two. *Discrete Math.* **307**(19–20) 2438–2447.

Alekseev, V. E., A. Farrugia, V. V. Lozin. 2004. New results on generalized graph coloring. *Discrete Math. Theoret. Comput. Sci.* **6**(2) 215–222.

Apollonio, N., P. G. Franciosa. 2007. A characterization of partial directed line graphs. *Discrete Math.* **307**(21) 2598–2614.

Bartnicki, T., J. A. Grytczuk, H. A. Kierstead. 2008. The game of arboricity. *Discrete Math.* **308**(8) 1388–1393.

Bench-Capon, T. J. M. 2002. Value-based argumentation frameworks. S. Benferhat, E. Giunchiglia, eds. *Proc. 9th Internat. Workshop Non-Monotonic Reasoning, Toulouse, France*, 443–454.

Broersma, H., F. V. Fomin, J. Kratochvil, G. J. Woeginger. 2006. Planar graph coloring avoiding monochromatic subgraphs: Trees and paths make it diffcult. *Algorithmica* **44**(4) 343–361.

Chen, Z.-Z. 2000. Efficient algorithms for acyclic colorings of graphs. *Theoret. Comput. Sci.* **230**(1–2) 75–95.

Cherchye, L., B. De Rock, F. Vermeulen. 2007. The collective model of household consumption: A nonparametric characterization. *Econometrica* **75**(2) 553–574.

Cherchye, L., B. D. Rock, J. Sabbe, F. Vermeulen. 2008. Nonparametric tests of collectively rational consumption behavior: An integer programming procedure. *J. Econometrics* **147**(2) 258–265.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, C. Stein. 2005. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Deb, R. 2008a. Acyclic partitioning problem is NP-complete for $k = 2$. Private communication, April 28. Yale University, New Haven, CT.

Deb, R. 2008b. An efficient nonparametric test of the collective household model. Working paper, Yale University, New Haven, CT.

Farin, G. 2006. Class A Bézier curves. *Comput. Aided Geometric Design* **23**(7) 573–581.

Goddard, W. 1991. Acyclic colorings of planar graphs. *Discrete Math.* **91**(1) 91–94.

Hogg, T. 1985. Refining the phase transition in combinatorial search. *Artificial Intelligence* **81**(1–2) 127–154.

Karp, R. M. 1990. The transitive closure of a random digraph. *Random Structures Algorithms* **1**(1) 73–93.

Khanna, S., K. Kumaran. 1998. On wireless spectrum estimation and generalized graph coloring. *Proc. INFOCOM'98, San Francisco*, IEEE, Piscataway, NJ, 1273–1283.

Lund, C., M. Yannakakis. 1993. The approximation of maximum subgraph problems. G. Goos, J. Hartmanis, eds. *Automata, Languages, Programming* (*ICALP* 93). Lecture Notes in Computer Science, Vol. 700. Springer, Berlin, 40–51.

Monasson, R., R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky. 1999. Determining computational complexity from characteristic "phase transitions." *Nature* **400**(6740) 133–137.

Raspaud, A., W. Wang. 2008. On the vertex-arboricity of planar graphs. *Eur. J. Combinatorics* **29**(4) 1064–1075.

Roychoudhury, A., S. Sur-Kolay. 1995. Efficient algorithms for vertex arboricity of planar graphs. P. S. Thiagarajan, ed. *Proc. 15th Conf. Foundations Software Tech. Theoret. Comput. Sci.* Lecture Notes in Computer Science, Vol. 1026. Springer, Berlin, 37–51.

Talla Nobibon, F., F. Spieksma. 2010. On the complexity of testing the collective axiom of revealed preference. *Math. Soc. Sci.* **60**(2) 123–136.

Talla Nobibon, F., C. Hurkens, R. Leus, F. Spieksma. 2010a. Exact algorithms for coloring graphs while avoiding monochromatic cycles. B. Chen, ed. *Proc. 6th Internat. Conf. Algorithmic Aspects in Information and Management* (*AAIM* 2010). Lecture Notes in Computer Science, Vol. 6124. Springer, Berlin, 229–242.

Talla Nobibon, F., L. Cherchye, B. D. Rock, J. Sabbe, F. Spieksma. 2010b. Heuristics for deciding collectively rational consumption behavior. *Comput. Econom.* ePub ahead of print June 10.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1**(2) 146–160.

Thomassen, C. 2008. 2-list-coloring planar graphs without monochromatic triangles. *J. Combin. Theory* **98**(6) 1337–1348.

Thorsteinsson, E. S. 2001. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. T. Walsh, ed. *Principles Practice Constraint Programming* (*CP* 2001). Lecture Notes in Computer Science, Vol. 2239. Springer, Berlin, 16–30.

Varian, H. R. 2006. Revealed preference. M. Szenberg, L. Ramrattan, A. A. Gottesman, eds. *Samuelsonian Economics and the Twenty-First Century*. Oxford University press, Oxford, UK, 99–115.

Wu, Y., J. Yuan, Y. Zhao. 1996. Partition a graph into two induced forests. *J. Math. Stud.* **29**(1) 1–6.