Pergamon

Mathl. Comput. Modelling Vol. 25, No. 7, pp. 79–87, 1997

0895-7177/97 $17.00 + 0.00

PII: S0895-7177(97)00050-2

Dynamic Graph Models

F. HARARY AND G. GUPTA
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003-0001, U.S.A.

(Received October 1996; accepted November 1996)

Abstract—Research in graph theory has focused on studying the structure of graphs with the assumption that they are static. However, in many applications, the graphs that arise change with time, i.e., they are dynamic in nature. This is especially true of applications involving graph models in computer science. We present an expository study of dynamic graphs with the main driving force being practical applications. We first develop a formal classification of dynamic graphs. This taxonomy in the form of generalizations and extensions will in turn suggest new areas of application. Next, we discuss areas where dynamic graphs arise in computer science such as compilers, databases, fault-tolerance, artificial intelligence, and computer networks. Finally, we propose approaches that can be used for studying dynamic graphs. The main objective in any study of dynamic graphs should be to

(i) extend results developed for static graph theory to dynamic graphs,
(ii) study the properties that describe how a dynamic graph changes,
(iii) investigate problems and issues in dynamic graph theory that are raised by practical applications of dynamic graphs in computer science.

Keywords—Graph theory, Graph models in computer science, Dynamic graphs.

1. INTRODUCTION

Graph theory [1] is an established area of research in combinatorial mathematics. It is also one of the most active areas of mathematics that has found large number of applications in diverse areas including not only computer science, but also chemistry, physics, biology, anthropology, psychology, geography, history, economics, and many branches of engineering. Graph theory has been especially useful in computer science, since after all, any data structure can be represented by a graph. Furthermore, there are applications in networking, in the design of computer architectures, and in general, in virtually every branch of computer science. However, to date most of the research in graph theory has only considered graphs that remain static, i.e., they do not change with time. A wealth of such literature has been developed for static graph theory. Our purpose is to classify *dynamic graphs*, i.e., graphs that change with time. Dynamic graphs appear in almost all fields of science. This is especially true of computer science, where almost always the data structures (modeled as graphs) change as the program is executed. Very little is known about the properties of dynamic graphs. Given their wide applicability and the frequency with which they appear in almost in all areas, their study is of considerable importance.

One of the main approaches that we advocate for studying graphs is the paradigm of Logic Programming [2,3]. Logic programming is based on a subset of predicate logic, called Horn Logic [4]. Since predicates can also be viewed as relations, logic programming is also known as

relational programming. There is quite a close connection between logic programming and graph theory. A graph G essentially represents a symmetric relation over a set whose elements are the nodes of G. Logic programming is a paradigm of computing based on relations. Every graph can be expressed as a simple logic program. All graph theoretic concepts can also be expressed within the framework of logic programming. Thus, logic programming is an ideal tool for the computational study of structural properties (dynamic or static) of a graph.

2. DYNAMIC GRAPHS

For completeness, we first define a graph, a digraph (directed graph), and a network (node weighted or edge weighted graph, or both). Each one of these can occur as the underlying structure of a dynamic graph, described below.

- A *graph* G is a pair (V, E), where V is a finite set of *vertices* or *nodes*, and E is a set of *edges*, each being an unordered pair $\{u, v\}$ of distinct nodes.
- A *digraph* is a pair (V, A), where V is again a finite set of nodes, and A is a set of *arcs*, ordered pairs (u, v) where $(u, v) \in V \times V$, $u \neq v$.

We now define three kinds of networks: a node network, an edge network, and a network.

- A *node network* (or *node weighted graph*) is a triple (V, E, f) where V is a set of vertices, E is a set of edges $\{u, v\}$, and f is a function, $f : V \rightarrow \mathcal{N}$ where $\mathcal{N}$ is some number system, assigning a value or a weight. Depending on the context, the weights may be real numbers, complex numbers, integers, elements of some group, etc.
- An *edge network* or *edge weighted graph* is a triple (V, E, g) defined similarly.
- A *network* or *fully weighted graph* has weights assigned to both nodes and edges.

These definitions of (static) graphs and networks involve the following entities: V (a set of nodes), E (a set of edges), f (map vertices to numbers), and g (map edges to numbers). A dynamic graph is obtained when any of these four entities changes over time. Thus, there are four basic kinds of dynamic graphs.

- In a *node-dynamic* graph or digraph, the set V varies with time. Thus, some nodes may be added or removed. When nodes are removed, the edges (or arcs) incident with them are also eliminated.
- In an *edge-dynamic* (or *arc-dynamic*) graph or digraph, the set E varies with time. Thus, edges may be added or deleted from the graph or digraph.
- In a node weighted dynamic graph, the function f varies with time; thus, the weights on the nodes also change.
- In an edge or arc weighted dynamic graph or digraph, the function g varies with time.
- In fully weighted dynamic graph, both functions f and g may vary with time.

All combinations of the above types can occur. For example, a computer network with changing bandwidth (edge-weight), changing topology (edges being added or deleted), changing computing power (node-weights changing), and computers representing nodes crashing, and recovering represents a dynamic graph that entails all the above basic types.

3. APPLICATION AREAS OF DYNAMIC GRAPHS

Dynamic graphs are ubiquitous in computer science and other real world applications. We now list some of these.

3.1. Computer Programming Languages

The computation models (or the operational semantics) of many computer languages are given in terms of graphs. Given a program, it is transformed into a graph. This graph changes with time by operations specified in the language, ultimately reducing to a simpler graph that constitutes the

answer the program was asked to compute. Logic programming languages, functional languages, and data-flow languages fall in this class. In logic programming languages [2], such as Prolog [3], execution is based on the depth-first search of the query's *execution tree* (also called *SLD tree*) [4]. To evaluate a query and answer it, the SLD tree of the query is dynamically constructed during the execution. Once the construction of the SLD tree is complete, the answer is found. Knowing properties of dynamic trees that are constructed during execution, it turns out, is very useful in parallel implementations of Prolog [5]. Such research will benefit from the answer to the following question about dynamic trees: "given a dynamic rooted tree, and given two of its nodes, is there an algorithm that determines in *constant time* whether the two nodes lie in the same path from the root?"

The computation model of functional programming languages is also expressed in terms of graphs. *Graph reduction* is one of the most common ways of implementing functional languages. Functional programs are translated into expression graphs, and program execution consists of reducing this graph until an irreducible graph is obtained. This final graph (usually a single node containing a value) constitutes the answer.

Data-flow languages are yet another class of languages whose operational semantics are based on graphs. The execution model of data-flow languages is built around the notion of *data-flow graphs*. A given program is translated into a data-flow graph and the execution of this program involves reducing this graph dynamically to obtain an answer.

3.2. Artificial Intelligence

Dynamic graphs arise naturally in Artificial Intelligence. The tree search that is involved in problem solving in AI essentially involves building trees. Game tree search [6] also involves building and pruning trees dynamically. Semantic nets provide yet another kind of dynamic graph that changes when more information is added.

3.3. Networks

Computer networks of all kinds are dynamic. This is because the weights on the edges, which denote the amount of traffic on the link corresponding to the edge, change dynamically. Also, the nodes [7] go down (fail) and so do links [8], resulting in the network becoming a dynamic graph. With computer networks (such as the internet) becoming widely available, the need to study dynamic graphs is even more compelling, as large computer networks can be abstractly modeled as dynamic graphs.

A network of communicating processes [9], such as one that is created by many parallel languages (e.g., Occam [10]), is also a dynamic graph, because processes establish links to communicate with other processes dynamically (edge dynamic graphs). Also, the processes spawn and die with time (node dynamic graph).

A network of constraints is also a dynamic graph. Such networks arise in Artificial Intelligence and Operations Research. A network of constraints is obtained as follows. Each constraint constitutes a node in the network. If the input variables of a constraint c_1 obtain their value from output variables of constraint c_2, then an arc will be placed from c_2 to c_1 indicating this dependence, resulting in a digraph. Nodes and arcs are deleted as constraints become solved. Graph models for dependence of statistical variables are similar.

3.4. Data Structures

Data structures used in computer programming are inherently dynamic. Thus, linked lists, binary trees, etc., are modeled by dynamic graphs. All types of specialized trees such as AVL trees, heaps, B-trees [11] are dynamic in nature. For instance, B-trees [12] are intensively used in relational databases to build efficient indices. Then given a key, the corresponding record can be accessed very quickly from the database. The B-tree changes as records are added to and deleted from the database.

3.5. Compilers

The optimization phase in a compiler makes heavy use of dynamic graphs. A *flow graph* is built from the *blocks* of the program to be compiled to capture the flow of control in a program. A block is a sequence of program statements with exactly one point where control enters and exactly one point where it leaves [13]. During the optimization phase of a compiler, flow of data from one block to another is analyzed and the results used to improve program execution performance. The flow graph is a dynamic graph, in which the edge labels (data flows) change as flow analysis proceeds. Flow analysis is thus a dynamic activity that involves a dynamic graph. Dynamic data-flow graphs also arise in compile-time analysis tools that are based on the technique of abstract interpretation [14].

3.6. Databases

The widely used data structure for maintaining indices to large databases is a B-tree [12]. A B-tree permits fast access to the disk block containing the desired record with a given *key*. Each node in the B-tree can store a fixed maximum number of keys. Insertion of a key may cause the node to have more keys than it can hold, and the node may have to be split into two nodes. Likewise, deletion a key may cause a node to have too few keys stored in that node, in which case it is better to merge this node with another node to create a single one. Thus, as records are inserted and deleted from the database, the B-tree changes. Thus, a B-tree is another example of a dynamic graph.

3.7. Fault Tolerance

Dynamic graphs are essential to the area of fault tolerance. Fault tolerance deals with maintaining the structure of a parallel computer architecture in the presence of node (processor) or edge (link) failures. Reliable fault tolerance is essential for maintaining a robust computer network or a multiprocessor system [15]. If properties of the network in presence of node and edge failures are known, then its behavior in case of a fault can be detected and fixed more easily [7,8,16].

Another dynamic aspect in the realm of fault tolerance is network reconfiguration [15]. Given a computer network, one or more of whose nodes or edges fail, then it must be reconfigured so as to maintain the same communication pattern between jobs running on various nodes.

3.8. Connectivity in Graphs

The (*node*) *connectivity* of a connected graph G is the smallest number of nodes whose removal results in a disconnected graph. The edge connectivity is defined similarly. Connectivity is a dynamic concept because changes in the graph structure are made by removing nodes or edges in order to disrupt the possibility of communication. The most important result known about connectivity is the classical 1927 theorem due to Menger [17].

THEOREM. *For any two nodes of a connected graph which are not adjacent, the minimum number of edges whose removal disconnects them from each other equals the maximum number of edge-disjoint paths joining them.*

This basic minimax result of graph theory was later independently rediscovered by Ford and Fulkerson in 1956, who formulated it in terms of networks and called it the maximum flow/minimum cut theorem [18,19]. Their theorem and algorithm for implementing it constitute one of the cornerstones of Operations Research.

More generally, for other graph theoretic properties P, one can define conditional connectivity [20]. The (node) conditional connectivity of a graph G with respect to a graph property P is the smallest number of nodes which need to be removed from G to a get a subgraph having property P. The conventional property P is for a graph to be disconnected. Other candidates for

graph properties include: bipartite, acyclic, Hamiltonian, Eulerian, having a specified maximum degree.

Conditional connectivity offers an exceptionally promising field for further research on dynamic graphs as every theorem on connectivity (see [1, Chapter 5]) can be investigated for each type of conditional connectivity.

3.9. Signed Graphs

Signed graphs can also be dynamic. A *signed graph* is an edge weighted graph except that the function g maps edges to the set $\{+1, -1\}$. Signed graphs have been used in studying interpersonal networks [21]. A signed graph is *balanced* if every cycle has even number of negative edges. The more balanced a network, the more stable the relationships among different people (or countries) involved [22]. Imbalance can result in friction and instability [23,24]. Personal networks can be highly dynamic, as preference of people (or countries) for each other changes with time. However, the following result is known [25].

THEOREM. *For a signed graph, the smallest number of (signed) edges that need to be removed to get balance equals the smallest number of edges whose sign needs to be changed to obtain balance, and the same set of edges works for both transformations.*

3.10. Graph Games

Dynamic graphs also occur in analyzing various *graph games*. A typical graph game will involve adding edges under certain constraints, until one of the two players involved accomplishes the goal of the game. For example, consider the following two player game.

The playing board consists of five isolated nodes. These can be labeled for convenience by the numbers 1, 2, 3, 4, 5. The two players are A and B, with A always making the first move using a green color, while B uses red (in the figures, a dashed line represents a green edge while a solid line represents a red edge). The game begins with A drawing a green edge between two of the nodes. The result is one green edge and three isolated nodes, no matter where A moves. Hence, this first move has no effect at all on the outcome of the game. Then B draws a red edge joining two nodes not already adjacent. Now A draws another green edge joining two nodes that are not already adjacent, and so forth. The player who first completes a triangle with all three edges in his own color is the loser. It was recently proven by Harary and Shader [26] that with perfect play this triangle avoidance game will end in a draw.

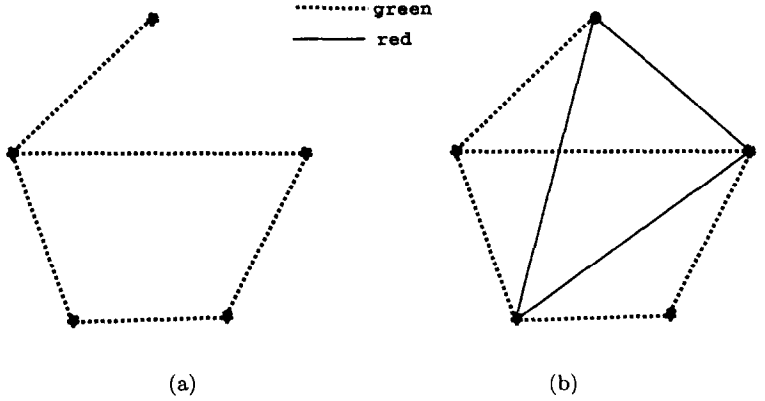

Figure 1. A makes a green C_4 in (a); B is forced into a red triangle in (b).

The following observation presents a tactic for winning this game.

LEMMA. *If with his first four moves A completes a quadrilateral C_4 in green, while B draws a star $K_{1,4}$ joining the fifth node with the four nodes of C_4, then A will lose.*

PROOF. On his fifth move and last move, A must draw a diagonal of his green C_4.

THEOREM. *Except for the procedure of Lemma 1, if one of the players forms a quadrilateral in his color, then the other player must lose.*

PROOF. As shown in Figure 1, if the five (say) green edges are as in Figure 1a, then the remaining edges include a triangle (Figure 1b).

4. APPROACHES TO STUDYING DYNAMIC GRAPHS

While dynamic graphs are ubiquitous in computer science, very little is known about their properties. We suggest two possible approaches that can be used. The first one utilizes known results from static graph theory and the second is a computational approach based on the paradigm of logic programming.

4.1. Knowledge from Static Graph Theory

A dynamic graph can be viewed as a discrete sequence of static graphs. The properties of each static snapshot of a dynamic graph can be studied using the knowledge that has been developed for static graphs [1]. We give an example using Markov chains.

By definition [27], a *finite stationary Markov chain* consists of n states v_1 to v_n, one being distinguished as the initial state, together with a probability matrix $P = [p_{ij}]$ such that $0 \leq p_{ij} \leq 1$, for all i, j and for each i, $\Sigma_{j=1}^{n} p_{ij}$. Each number p_{ij} is the conditional probability that if the present state is v_i, then the next state will be v_j. In a stationary Markov chain, all the numbers p_{ij} are constant over time. Otherwise, the chain is nonstationary (dynamic), which is in general, more appropriate as a model for real world phenomena. Stationary Markov chains have been studied and applied intensively [27]. Nonstationary chains were used as a mathematical model by Harary and Lipstein [28] in their study of brand loyalty and brand switching. A panel of grocery shoppers reported each month which brand of a product (toothpaste, for example) was purchased. The proportion of those who bought brand i last time bought brand j this time was taken as p_{ij} forming matrix P. Thus, the entries of matrix P changed each time period (one month in this example). From the sequence of matrices $P_1, P_2, \ldots$, predictions involving eigenvalues were made for the likelihood that a new brand introduced into the marketplace would succeed in surviving. This exemplifies a succession of digraph networks with the same node set but with changing arc values.

One can also study dynamic graphs by specifying the properties that remain invariant with time. This approach is especially useful in studying fault tolerance of computer architectures and networks. One question that arises in this area is to determine the minimum number of (spare) nodes and edges that must be added to a given graph G, such that if a node or an edge fails (is removed), then the remaining subgraph contains G up to isomorphism. Consider a 4-node architecture whose interconnection network is a path P_4. Adding an extra node and connecting it to the extremities of the path (to obtain a cycle C_5), will ensure that if any one node fails, a path P_4 will still remain. This is the case of node fault tolerance. For an example of edge fault tolerance, consider a ring architecture whose structure is pentagon C_5. We need to add three edges (see Figure 2), to ensure that if any one edge fails then the resulting graph contains a C_5 [8].

If more than one component of a dynamic graph change with time, then the invariants can be studied much in the fashion of partial derivatives. Consider a fully weighted dynamic graph. We may want to keep V, E, and function f constant, and study the invariants as function g changes. Likewise, V, E, and function g can be kept constant, and the invariant properties can be studied as function f changes, etc.

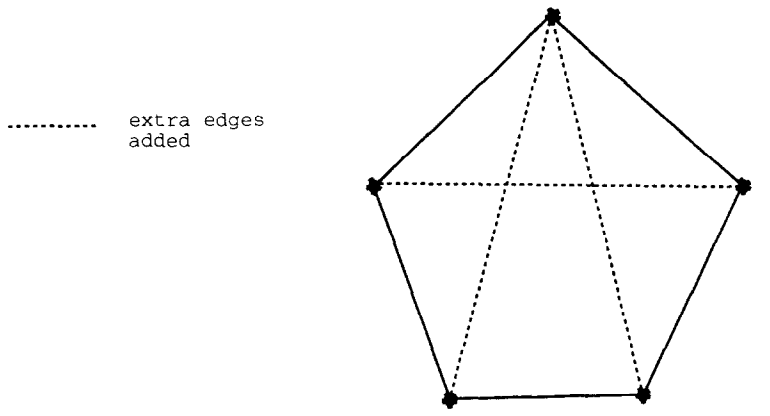

Figure 2. Edge fault tolerance for C_5.

4.2. Logic Programming (LP)

Dynamic graphs can be modeled using a logic program. The logic program can then be executed to discover interesting properties of the dynamic graph being modeled.

LP is a paradigm of programming based on a subset of first-order logic, called Horn Logic. A logic program consists of facts and rules.

A fact is of the form: $q(t_1, \ldots, t_n)$, where q is a predicate or a relation over *terms* t_1 through t_n. A fact states that the predicate $q(t_1, \ldots, t_n)$ is unconditionally true.

A *rule* is of the form $p(t_1, \ldots, t_n) \Leftarrow r(u_1, \ldots, u_l) \wedge \cdots \wedge s(v_1, \ldots, v_m)$ where p, r, and s are predicates. Rules are used for stating conditional truths. The rule above states that the predicate $p(t_1, \ldots, t_n)$ is true if the conjunction of predicates $r(u_1, \ldots, u_l)$ through $s(v_1, \ldots, v_m)$ is true.

The operational semantics of logic programming are defined by interpreting predicates p and q as program procedures. Thus, to solve p, we need to call procedures r through s, which in turn may call other procedures. Once rules and facts have been defined, the program can be queried to check for facts that logically follow from it. To take a very simple example, we can define graph reachability as a logic program. Facts will be used to represent the digraph, and the rules will be used to define the notion of reachability. LP can also be used for programming graph algorithms, as well as to study graph theoretic properties such as finding perfect matchings of a graph and determining its chromatic numbers.

LP can be used for studying dynamic graphs. The dynamic aspect of the graph can be empirically modeled by a logic program. There are several reasons for choosing LP to study graphs.

1. Logic programming is a relational programming paradigm, and given that graphs are relations, they can be naturally modeled within LP.
2. Due to its high level, declarative nature and its closeness to logic, logic programs can be developed and debugged much more easily [3].
3. *Nondeterminism* is a central feature of logic programming. Given a goal, there may be many ways of reaching it. The execution mechanism of logic programming will find all possible ways. Presence of nondeterminism makes it an ideal language for programming combinatorial searches. Given that many graph theoretic properties are combinatorial in nature, logic programming is an ideal tool for studying them.
4. LP is highly suitable for metaprogramming [3]. Given that graphs are modeled as logic programs, a dynamic graph will be modeled by a logic program that changes as execution proceeds. Metaprogramming capabilities of LP can be used for dynamically changing this logic program representation of a graph to model dynamic graphs.

For example, logic programming can be used to verify the well-known fact that the 2-color, triangle avoidance game described in Section 3.10 can end in a draw. One can write a more general logic program that will verify if a 2-color, K_n avoidance game can end in a draw. This

logic programming formulation can also be used for computing Ramsey numbers [29]. In fact, *constraint logic programming* [30,31], a generalization of logic programming, can be used to program the above more efficiently.

5. CONCLUSIONS

We have presented an expository classification of dynamic graphs. Dynamic graphs arise in virtually all areas of science. This is specially true of computer science, where all data structures can be regarded as implementations of dynamic graphs. Dynamic graphs are also found in chemistry, anthropology, psychology, and all other areas to which graph theory can be fruitfully applied. Thus, their largely neglected study can be of utmost practical importance. We suggest two approaches to studying dynamic graphs:

(i) modeling a dynamic graph as a sequence of static graphs and studying the properties of this sequence;

(ii) modeling a dynamic graph using logic programming, a declarative and highly expressive programming paradigm, and examining the properties of this graph via execution of the logic program that models it.

It is not clear how one should go about systematically studying dynamic graphs. We hope that this paper will spur interest in the study of dynamic graphs and will induce the development of new applications of dynamic graphs as well as novel approaches to studying them.

REFERENCES

1. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, (1969).
2. R.A. Kowalski, Predicate logic as a programming language, *Proc. IFIPS Congress*, (1974).
3. L. Sterling and E. Shapiro, *The Art of Prolog*, Second edition, MIT Press, Cambridge, MA, (1994).
4. J.W. Lloyd, *Foundations of Logic Programming*, Second edition, Springer-Verlag, Berlin, (1987).
5. G. Gupta and B. Jayaraman, Analysis of OR-parallel execution models, In *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, Vol. 15, pp. 659–680, (1993).
6. E. Rich and K. Knight, *Artificial Intelligence*, McGraw-Hill, New York, (1983).
7. F. Harary and J.P. Hayes, Node fault tolerance in graphs, *Networks* (to appear).
8. F. Harary and J.P. Hayes, Edge fault tolerance in graphs, *Networks* **23**, 135–142, (1993).
9. C.A.R. Hoare, Communicating sequential processes, In *Communications of the ACM*, Vol. 21, pp. 666–677, (1978).
10. D. May, Programming in occam, *SIGPLAN Notices* **18**, 69–79, (1983).
11. T. Cormen, C. Lieserson and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, (1989).
12. J.D. Ullman, *Principles of Database Systems*, Computer Science Press, (1988).
13. A. Aho, J.D. Ullman and R. Sethi, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, (1986).
14. P. Cousot and R. Cousot, Abstract interpretation: A unified model for static analysis of programs for construction of approximation of fix-point, In *Conference Record of the 4^{th} ACM Symposium on Principles of Programming Languages*, pp. 238–252, (1977).
15. F. Harary and M. Malek, Quantifying fault recovery in multiprocessor systems, *Mathl. Comput. Modelling* **17** (11), 97–104, (1993).
16. J.P. Hayes, A graph model for fault-tolerant computer systems, *IEEE Trans. Comput.* **C-25**, 879–884, (1976).
17. K. Menger, Zur allgemeinen Kruventheorie, *Fund. Math.* **10**, 96–115, (1927).
18. L.R. Ford and D.R. Fulkerson, Maximal flow through a network, *Canadian J. Math.* **8**, 399–404, (1956).
19. L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, (1962).
20. F. Harary, Conditional connectivity, *Networks* **13**, 347–357, (1983).
21. F. Harary, On the notion of balance of a signed graph, *Michigan Math. J.* **2**, 143–146, (1953).
22. F. Harary and D. Cartwright, Structural balance: A generalization of Heider's theory, *Psychol. Review* **63**, 277–293, (1956).
23. F. Harary, A structural analysis of the situation in the Middle East in 1956, *J. Conflict Resolution* **5**, 167–178, (1961).
24. F. Harary, Graphing conflict in international relations, *Papers Peace Sci. Soc. Intl.* **27**, 1–9, (1977).
25. F. Harary and A. Blass, Deletion versus alteration in finite structures, *J. Combin. Inform. System Sci.* **7**, 139–142, (1982).
26. F. Harary and L. Shader, Perfect play in 2-color triangle avoidance on five nodes gives a draw, *J. Recreational Math.* (to appear).

27. W. Feller, *An Introduction to Probability Theory and its Applications*, Vol. 1, (Second edition), Wiley, New York, (1957).
28. F. Harary and B. Lipstein, The dynamics of brand loyalty: A Markovian approach, *Operations Research* **10**, 19–40, (1962).
29. F. Harary, Generalized Ramsey Theory I to XIII, In *The Theory and Applications of Graphs*, (Edited by G. Chartrand *et al.*), pp. 373–390, Wiley, New York, (1981).
30. P. van Hentenryck, *Constraint Handling in Prolog*, MIT Press, Cambridge, MA, (1987).
31. F. Benhamou and A. Colmerauer, Editors, *Constraint Logic Programming: Selected Research*, MIT Press, Cambridge, MA, (1993).