Boolean Constraint Propagation Networks

Georgi Marinov
Sirma AI Ltd.
18 Benkovski Str
1000 Sofia, Bulgaria
joro@sirma.bg
(359 2) 896-428

Vladimir Alexiev[*][†]
University of Alberta
Dept. of Comp. Sci., 615 GSB
Edmonton, Alberta T6G 2H1
vladimir@cs.ualberta.ca
(403) 433-5910 (home)

Yavor Djonev
SIRMA International
5-563 Riverdale Ave.
Ottawa, Ontario K1S 1S3
70751.2405@CompuServe.com
(613) 526-2160

Abstract

This paper describes a particular inference mechanism which has been successfully used for the implementation of an expert system and a generic shell supporting consulting-type expert systems. The main features of Boolean Constraint Propagation Networks (BCPN) are: the inference flows in all directions, unlike inference modes of forward or backward chaining systems; all possible consequences of a fact are derived as soon as the user enters the fact, therefore the system is very interactive; if the user withdraws an assertion then all propositions depending on it are retracted; the inference architecture is simple and uniform. After a general description of BCPN we give an account of the problems encountered and the approaches we used to solve them. Some possible extensions of the mechanism and its applicability to various areas are also discussed. The current version of BCPN is written in C++ and took about one man-year to develop.

Keywords: constraint propagation, inference engine, knowledge-based system, expert system.

1 Introduction

Expert systems have been around for more than two decades now. There exist a vast variety of architectures for Knowledge-Based Systems, most of which have quite sophisticated and complex control strategies. This variety is motivated by the very different properties which application domains possess (for example backward chaining is probably suitable for a diagnosing ES but unsuitable for an automated design system). Below we describe the type of applications for which BCPN seems an appropriate inference mechanism.

A *consulting-type* ES possesses a large body of knowledge in a particular domain and tries to clarify a certain situation and interpret it according to that knowledge in collaboration with the user. For example many people experience difficulties interpreting taxation laws because of the large volume of these norms and they need help for the task of filling tax returns. The knowledge is not very complex (typically inferences are not very deep) but the knowledge base consists of about 100,000 atomic facts. Such an expert system may interact either directly with the user on a personal computer or indirectly through a public service

[*]Use this author's email address for correspondence.
[†]Supported by a University of Alberta PhD Scholarship.

officer serving as a mediator. Another possible application domain is helping an air travel agent to find the best ticket fares, and so on.

Domains like these are characterized by the fact that it is not known a priori what kind of data the user may possess and therefore it is hard to predict the direction of the inference process. So it seems appropriate to assume that *every* direction is feasible and to give the initiative in the hands of the user. Furthermore, it is very useful to indicate to the user every inference as soon as it is made, so that s/he[1] knows where the problem-solving is going and can adjust his/her line of thought correspondingly. The user is never pressed for an answer: s/he simply states what s/he knows and sees relevant for the task.

The BCPN is constructed automatically by translation from a knowledge representation scheme that is convenient for the knowledge engineers to use to a network of objects. We do not address the issue of translation in this paper.

The rest of the paper is organized as follows. Section 2 describes the basic BCPN architecture and propagation process, the procedure for withdrawing previously stated propositions, the conflict resolution strategy and a goal propagation scheme. Section 3 describes certain problems we encountered with BCPN and some proposed solutions. Section 4 discusses two possible extensions to the basic BCPN architecture. Section 5 contains some concluding remarks and comparison with other well-known approaches.

2 Boolean Constraint Propagation Networks

This section gives a basic description of the BCPN and the inference process in BCPN. We leave some of the more sophisticated matters for subsequent sections.

2.1 The BCPN Architecture

A BCPN (in accordance with [10]) is an undirected bipartite graph consisting of two alternating types of nodes: *propositions* represented by squares and *operations* or *gates* represented by ovals. A sample BCPN is shown on Figure 1.

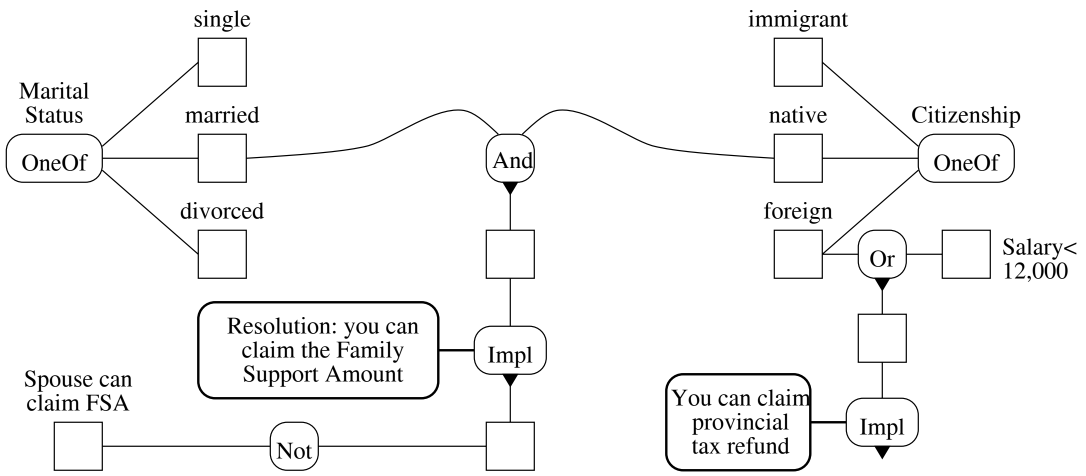

Figure 1: A Boolean Constraint Propagation Network

[1]S/he stands for he/she.

Every proposition corresponds to an atomic assertion (*e.g.* `MaritalStatus=married`) entered by the user (in which case it is called a *fact*) and/or inferred by the system. Propositions can be in one of the three states `True`, `False` and `Unknown`. Unlike *e.g.* Doyle [5], no `Contradiction` state is allowed because contradictions are not tolerated and are removed immediately (see section 2.4). Operations bind the propositions and constrain the possible combinations of states. It is important to note that although the output pins of the gates are marked by arrows, the information can flow in all directions and not only along the arrow.

`Impl` (stands for *implication*) gates represent rules in the knowledge base. When the input proposition of a rule is established to `T`, the rule *fires* and an attached list of *resolutions* is printed and/or a list of *actions* is carried out. These resolutions/actions are not part of the BCPN itself (the logic part of the system) but of the Knowledge Base. KB and BCPN form two separate levels of the system and the inference is independent on the additional features of the KB objects. Every parameter and condition imposed on parameter values have a corresponding proposition but the KB objects have additional properties: set of allowed values, prompt, *etc.* On the other hand, there are intermediate propositions which do not correspond to any KB object and are not directly user-accessible (the ones which do not have a label on the figure). Such propositions are generated during the compilation of boolean expressions representing the relations between KB objects.

2.2 Propagation of Boolean Values

The inference process in BCPN is a process of *propagation*. When the user enters a new fact through the user interface, the corresponding proposition is changed and notifies all operations connected to it about the change. These operations check whether the change affects some of their other pins and if so, notify them appropriately. In turn these propositions change their values and repeat the procedure recursively. A "wave" of changing propositions forms and propagates through the network. In order to avoid this wave going back immediately to its initiator (something like a recoil) and to avoid infinite recursion, there is a flag in each proposition signifying whether it is under propagation. We should note that the user can set a proposition to `U`, that is to withdraw a fact stated earlier. For ease of understanding we explain this in the next section, though it uses basically the same propagation mechanism.

And	a	b	c
1	T	T	T
2	T	F	F
3	T	U	U
4	F	T	F
5	F	F	F
6	F	U	F
7	U	T	U
8	U	F	F
9	U	U	U
10	U	U	F

Or	a	b	c
1	T	T	T
2	T	F	T
3	T	U	T
4	F	T	T
5	F	F	F
6	F	U	U
7	U	T	T
8	U	F	U
9	U	U	U
10	U	U	T

OneOf	a	b	c
1	T	F	F
2	F	T	F
3	F	F	T
4	F	U	U
5	U	F	U
6	U	U	F
7	U	U	U

Impl	a	b
1	T	T
2	F	T
3	F	F
4	F	U
5	U	T
6	U	U

Table 1: The operations `And`, `Or`, `OneOf` and `Impl`

Table 1 shows all allowed value combinations for some of the gates. Of these only `OneOf` needs some explanation: it postulates that exactly one of its pins should be `T` (it is something like a multi-arity `Not`) and is used for parameters with mutually exclusive allowed values.

The last (tenth) lines of the `And` and `Or` tables do not represent a valid combination for a three-valued logic gate because if both inputs of a gate are in the third (`U`) state then so should be the output (which situation is already accounted for in line 9). But BCPN is different from Three Valued Logic: if the output of an `And` is established to `F` then BCPN cannot deduce anything better than `U` for either of the inputs. We just know that one of them must be `F` but we cannot tell which one. We will see later in Section 3.1 that these "illogical" situations cause problems. This is due to the fact that `And` and `Or` are not reversible operations: they lose information.

For efficiency reasons and to be able to easily increase the arity of operations, we have not implemented the gates using these tables but rather like explicit algorithms. For example the algorithm for `And` is like this: "If one of the inputs is `F` then set the output to `F`; if all inputs are `T` then set the output to `T`; if the output is `T` then set all inputs to `T`; if all pins except one have values, then set it to `F`" (the last two clauses propagate from output to input).

The propagation stops in a certain direction under one of the following cases:

1. It enters a region of all-`U` propositions and has insufficient information to continue through this region.

2. It tries to set a proposition with a certain value but discovers that the proposition already has that same value.

3. It reaches a node that is already under propagation.

The first case is significant for large knowledge bases. Such KBs are naturally divided into modules (*topics*) which are loaded in memory as indivisible units. Propagation often will not pass the boundary of a topic because topics are linked weakly. In a typical consultation the user enters just a small amount of facts and only a small part of the KB and the underlying BCPN are activated. This is what makes the approach scalable.

2.3 Propagation of `Unknown`

The system supports withdrawal of previously stated assertions. All consequences of the cancelled fact should also be cancelled. To facilitate this, dependency information (*justifications*) is maintained, similar to Doyle's TMS [5]. First, all facts entered by the user get marked as such. Second, *arrows* which link propositions and operations are introduced and BCPN becomes a directed graph (Figure 2).

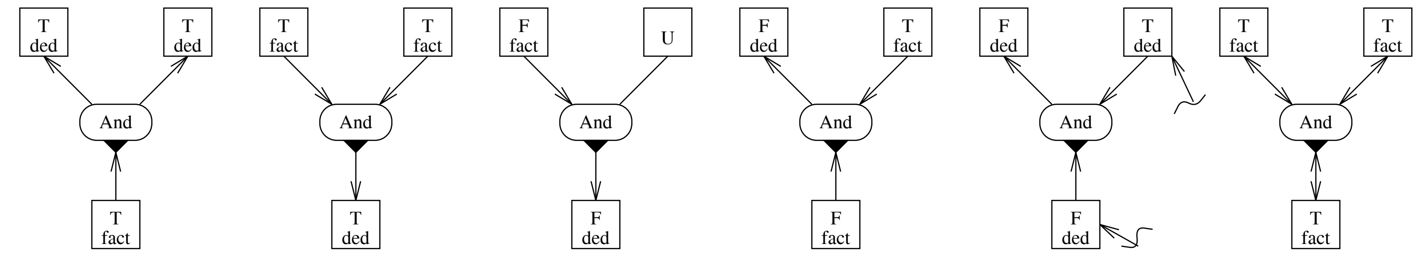

Figure 2: Some of the possible support states for `And`

There is an arrow from an operation to a proposition if the operation has set (or tried to set) that proposition's value; there is an arrow going the other way if the value of the proposition caused (or tried to cause) a change at some of the other pins of the gate. It is possible to have arrows going both ways in the case the user enters (confirms) already inferred data (*e.g.* the rightmost gate on Figure 2).

An important design consideration for BCPN is that the order of presentation of facts should have no impact on the BCPN state, only the set of facts should. We paid particular attention to save this property and this is why the clauses (or tried to set) in the previous paragraph are there (see the second propagation-stopping condition in the previous subsection).

The dependency arrows can always be computed anew starting from the founding facts but it is more efficient to cache them explicitly.

When a proposition is to be set to `U` (that is, unset), it first checks whether it is a fact. If this is not so (it is *deduced*) then nothing happens, because even if the user says "I don't not know what this value is," the system knows it. If however it is a fact, then its "fact" flag is cleared. Then the proposition checks whether there are some propositions which support it (through neighboring operations) and who themselves are supported by still existing facts (they are "well-founded"). This is done by propagating the question "Are there any facts supporting you?" recursively. It would not be enough to check locally for outside support because there may be a loop of propositions supporting one another (circular support) which used to be justified by the withdrawn fact and now is to be broken. The same fact-gathering mechanism is used for conflict resolution, as described in the next subsection.

If there are no supporting facts then the proposition eventually becomes `U` and notifies all neighbor operations of the change. If some of them cease to support some of their other pins then they will propagate the wave of `U`s further.

This is a bit complicated. We considered the simpler approach to reset the network and restate all remaining facts anew. This is not obviously infeasible because all the propagation is localized in a small part of the network. But if the user has stated, say, 25 facts then s/he would have had to experience a 25 times bigger delay than the normal interaction delay, which may be intolerable.

2.4 Conflict Resolution

When the system or the user tries to change a proposition from a definite (non-`U`) state to the opposite state, a conflict situation occurs. It has to be resolved by withdrawing some of the clashing facts. The system cannot do this by itself because it does not know which of these are actually true / of higher priority. (Unlike ATMS [3], the system never makes assumptions itself.) What it can do is to present the user with the conflicting facts and ask him/her to choose. These facts are being collected starting from the point of conflict by (almost) the same support-collection algorithm described in the previous subsection. The difference is that here we need not only an answer to the question "Does anybody support you?" but the set of support itself. The set of support has and And/Or-tree structure: the `T`s on the inputs of an `And` gate support the `T` on the output in a different manner then they would in an `Or` gate. Namely, in the `And` case the `T`s are both needed, while in the `Or` case either one suffices. Therefore if one wishes to turn the output to `U` one will have to unset either of the inputs in the former case and both of them in the latter case.

This structure can be represented either by the traditional And/Or-tree or by putting different types of lines in the margin, like for a system of equations (Figure 3). The latter would make the interface simpler, but if we want to avoid duplication of propositions (for

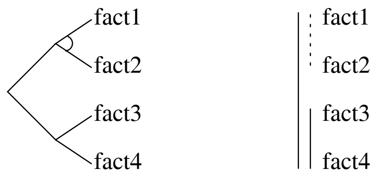

Figure 3: Alternative representations of And/Or structure

example `fact1` and `fact3` might be the same proposition) then we will have to adopt the tree scheme and generalize it to an acyclic graph.

In either case, the user simply points which branch (or both) of an Or node s/he would like to cancel. The point of conflict corresponds to the root of the tree and is an Or node: one of the two "parties" supporting the two conflicting values should resign.

2.5 Goal Propagation

The goal of a consultation is to reach some conclusion. All possible conclusions (outcomes of a consultation) in the domain under consideration are known a priori: they are the resolutions attached to `Impl` operations (rules). Therefore it is possible to advise the user what (additional) information s/he should supply in order to reach certain conclusion(s). Namely, this is the And-Or tree of (yet not established) facts which would support a `T` in the input proposition of the `Impl`. Collecting this tree is different from the collection process for conflict resolution because the facts are not stated yet and therefore no support arrows exist. Fortunately such a structure can be generated based solely on the logical properties of the gates and the structure of the BCPN. The process stops at propositions which are linked to the higher level (KB) and thus accessible to the user. If the user does not know the value of such terminal proposition, s/he may opt to ask further what other propositions should be set to infer the value of the terminal proposition.

This "goal propagation" corresponds in some sense to backward propagation in traditional ES.

3 Problems with BCPN

In this section we describe the problems we have identified in BCPN and possible approaches for their solution.

3.1 Employing Constraint Satisfaction

The most important problem is that sometimes propagation alone is not sufficient to infer everything deducible. An example is given on Figure 4 (this situation presents two `Or` gates in state 10 from Table 1). Propositions d and e say that $a \vee b =$ `T` and $b \vee c =$ `T`. The `OneOf` gate says that a and c cannot both be `T`. Thus, the only possibility is $b =$ `T` and $a = c =$ `F`. But this cannot be inferred by propagation of the two `T` values in d and e.

This is not a purposely contrived configuration, it appears naturally when representing numeric values (see Section 4.2; do not pay attention to the numbers on the figure yet).

Therefore some other methods should be employed to amend this deficiency. We generally call them *Constraint Satisfaction* (as opposed to Constraint Propagation). An enormous

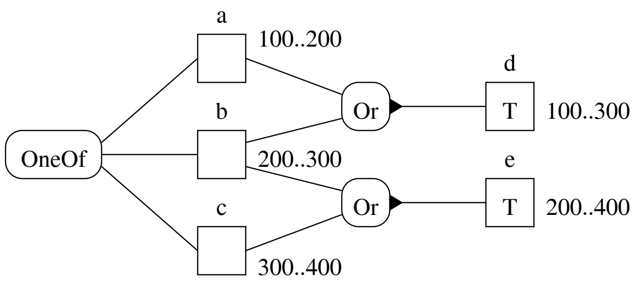

Figure 4: How to deduce that b=`T` from the given information?

amount of research has been performed in this area in connection with Constraint Satisfaction Problems (CSP) (*e.g.* see [8, 6, 9]) and Constraint Logic Programming (CLP). The main problem is to localize the constrained configuration both spatially (in a small part of the BCPN) and temporally (at a certain moment in time).

Because of the large size of the network, standard backtracking approaches to CSP are inappropriate. Some approaches from CSP seem applicable, *e.g.* interchangeability [7], articulation nodes [4]. Also relevant are approaches from CLP, *e.g.* arbitrary boolean expressions are supported and solved in Prolog III [2] (see also [1]). But the allowable systems of constraints in Prolog III are much smaller than the size of BCPN, and the scalability of that method is questionable.

An approach in the spirit of ATMS [3] would be the system to assume each one of `T` and `F` for a proposition and see whether both set some other proposition(s) to some definite values. However it is infeasible to trigger such tests before the constrained situation is localized, and the approach gives no hint in this respect.

We came up with a simple solution presented on Figure 5. An auxiliary `And` gate is added

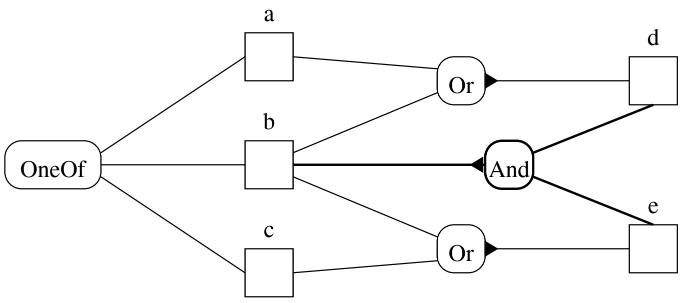

Figure 5: A simple solution: adding an `And` gate

during compilation which "injects" the correct value into *b* as soon as *d* and *e* are set to `T`. This solution eliminates the temporal localization problem and does not impose any changes to the inference mechanism. But it increases the size of the BCPN and furthermore it is not quite clear (yet) how to identify where such additional gates are needed (except for some standard configurations like intersected numeric intervals; see Section 4.2). In other words, the spatial localization problem remains.

We hope this is not very much of a problem because it occurs rarely and because it only concerns the completeness of the deduction but not its soundness. As a consolation, the strength of BCPN is in the propagation, not in the satisfaction.

3.2 BCPN Verification and Optimization

Typically BCPN contains a lot of redundancies: a proposition can be set from another proposition by many different (and sometimes implying each other) paths. Such redundancies are not immediately obvious to the knowledge engineers and they create a lot of unneeded links "just to be sure". A similar problem is a closed loop of propositions which imply each other. In this case they all can be reduced to a single node. A third related problem is a permanent conflict: two chains of propositions which would always imply opposite values at a certain node. This is an inconsistency in the KB which should be detected at compile time. Other things to check for are unconnected nodes, not fully connected operations, *etc.*

At runtime similar redundancies and inconsistencies are normally present and cause no harm because they are conditional upon a certain combination of "controlling" propositions. For example, if an input of an `And` (the "controlling" one) is `T` then the `And` transmits transparently and bidirectionally between its other input and its output which creates conditions for redundant/inconsistent loops. But static redundancies/inconsistencies should be detected by a preprocessing verifier and reported to the knowledge engineer. This program should also try to factor out common subnetworks. Its job may also include *introducing* redundancies to implement the solution mentioned in the previous subsection.

Currently we only implemented a simple common-subexpression eliminator which takes into account only some of the algebraic properties of the operations (commutativity, associativity) but not their logical properties.

4 Possible Extensions to BCPN

4.1 Temporal Reasoning

Temporal reasoning can be incorporated in BCPN with relative ease. One can attach to each proposition a history list holding its values from the past with time tags attached to them. It would be profitable to have distributed (nonuniform) time because typically only a small number of gates triggers at a given propagation cycle (which corresponds to an instant of the simulated time). This is similar to the approach adopted in modern electrical simulators where the time granularity is finer in the more active areas of the circuit and coarser in the passive ones. The user should be able to inspect and modify these history lists, though it would be more convenient for him/her to do so in a uniform manner (by setting the global time). If a user decides to "change the history" in some past moment then all history lists affected should be reevaluated from that moment onward. Hopefully this reevaluation will not have to go up to the present (at least not for all propositions) because the old and the changed histories may converge at an earlier point. More importantly, it will not spoil the aforementioned spatial locality of change.

The only real problem is to incorporate *rules* that access histories. Some language should be adopted for describing such temporal rules (*e.g.* "If your salary two years ago was $10,000 then ...") and they are to be implemented in the inference engine. An obvious way to go without it is to define a separate parameter `Salary2YearsAgo`, but this is clumsy.

4.2 True Numeric Variables

Currently BCPN supports only numeric variables which are reduced to a set of ranges connected by `OneOf`. Now the problem from Section 3.1 (Figure 4) appears naturally if there are two independent scales over which a numeric variable has some relevance. For we will have to intersect the two scales to get the atomic intervals a, b and c and then `Or` them to reconstruct the original intervals d and e. If the user has stated that the parameter is both between 100...300 and 200...400, then the program should be able to intersect d and e and deduce that the value is in b.

5 Concluding Remarks

We described a simple but powerful inference architecture which is useful for building highly-interactive Knowledge-Based Systems in simply-structured yet very large domains.

In contrast to the TMS approach of Doyle [5], the BCPN module provides not only truth maintenance (actually, justification maintenance), but also truth inference (propagation). The knowledge that the BCPN module has about logic gates allows it to optimize justification maintenance and use basically the same algorithms for both justification maintenance and propagation (the similarity of `T`/`F`-propagation and `U`-propagation mentioned in Section 2).

During our development we found that the standard constraint satisfaction approaches are unsuitable for BCPN due to its large size and relative sparseness of constrained situations.

An important property of BCPN is that since the density of connections does not depend on the KB size, the speed of the propagation (nodes/second) is independent on the KB size. Furthermore real KBs are "chunked" in a number of loosely connected parts so the average "free run" of the propagation is also independent on the KB size. This makes the propagation cycle and response times independent of the KB size, thus the approach is well-scalable.

BCPN is the base of the INTELLIGENT WORKSTATION project of *Sirma International*. The current version of BCPN is written in C++ and took some one man-year to develop.

Acknowledgment

Vesselin Kirov of Sirma AI participated actively in the development of Goal Directed Activation. One of the authors is grateful to the University of Alberta for providing the equipment to produce this paper.

References

[1] W. Büttner and H. Simonis. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation*, 4(3):191–206, 1987.

[2] A. Colmerauer. An introduction to Prolog III. *Commun. ACM*, 33(7):69–90, 1990.

[3] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[4] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1988.

[5] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[6] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, 1982.

[7] E. C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings AAAI'91*, pages 227–233, 1991.

[8] A. K. Mackworth, J. A. Mulder, and W. S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[9] I. Rivin and R. Zabih. An algebraic approach to constraint satisfaction problems. In *Proceedings IJCAI'89*, pages 284–289, 1989.

[10] P. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, MA, second edition, 1984.