

Creating Logic Puzzles

Bart Peintner

Final report for EECS 592, Winter 2001

April 17, 2001

ABSTRACT

A logic puzzle is a form of entertainment that asks the solver to infer relationships between objects given a set of clues. Each clue is a portion of the solution. A good puzzle is one that is challenging - one that requires the solver to make difficult inferences to realize each part of the solution. My goal was to find an algorithm that, given a solution to a puzzle, would generate a set of clues that would make the puzzle difficult for a person to solve. In this paper, I describe the logic puzzle domain, present the algorithm I designed and compare the puzzles it created with previously published puzzles.

1 INTRODUCTION

A logic puzzle is a form of entertainment that asks the solver to infer relationships between objects given a set of clues. The set of clues acts as a knowledge base from which the solver can infer new clues. Since each clue is a part of the solution, the process of continually inferring new clues will eventually lead to a solution.

Logic puzzles, like those published by Dell and PennyPress, exist in many different forms, which means the term 'logic puzzles' may not have the same meaning to all readers. In the subsequent section, I define the form of logic problems examined in this project.

Because Horn propositions can represent the knowledge and relationships needed, solving logic puzzles is simple. Generating these puzzles - interesting, challenging puzzles - requires more effort. In this paper, I examine possible methods of efficiently creating logic puzzles and report the result of implementing two of the proposed methods. One method failed, while the other produced seemingly high-quality puzzles. After placing the puzzles created by my algorithm in direct competition with published logic puzzles and viewing the results, I found that one of the assumptions necessary for my algorithm does not hold. Therefore, the quality of the puzzles fell below my expectations.

1.1 LOGIC PUZZLES

In published forms of logic puzzles, a story always defines the puzzle. The story enumerates the objects involved and establishes the types of relationships that exist between objects. The last sentence usually asks the solver to determine which objects are related. Formally, the story outlines the following elements:

- 1) A set of categories, C .

- 2) A set of objects, O .
Each $c \in C$ is a set of n unique objects, $\{O(c, 1), O(c, 2), \dots, O(c, n)\}$.
- 3) A set of semantic relationships.
These relationships show the relationships between categories. They have no bearing on the 'logic' of the problem, i.e. the puzzle can be solved without them.

The story implies that every object in each category is related to exactly one object in all other categories. This relationship is bi-directional; meaning, if $O(c_1, 1)$ is related to $O(c_3, 4)$, then $O(c_3, 4)$ is related to $O(c_1, 1)$. The previous statement implies that $O(c_1, 1)$ is not related to every object in category 3 besides $O(c_3, 4)$.

Beyond the story, each puzzle contains a set of clues (a knowledge base), KB . These clues give information about the relationships of objects. For example, one clue may state that $O(c_1, 1)$ is not related to $O(c_3, 2)$. Another clue may state that $O(c_2, 3)$ and $O(c_1, 4)$ are related. These clues, taken as a knowledge base, always entail all relationships. Moreover, the knowledge base of clues is always consistent; given the set of clues, only one solution to the puzzle is possible.

See Appendix A for examples of logic problems.

1.2 RELATIONSHIPS USED TO SOLVE LOGIC PROBLEMS

The set of clues, KB , is not sufficient information for solving a logic problem. The relationships residing in the KB must be used in conjunction with implicit relationships that exist in all puzzles. Below, I have enumerated these implicit relationships:

1) Related by elimination.

If object o_1 in category A is not related to every object in category B , except object o_2 , then o_1 must be related to o_2 .

2) Not related by exclusion.

If object o_1 in category A is related to object o_2 in category B , then o_1 is not related to every other object in category B , and o_2 is not related to every other object in category A .

3) Related by transitivity.

If object o_1 is related to object o_2 , and o_2 is related to object o_3 , then object o_1 is related to o_3 .

4) Not related by transitivity.

If object o_1 is related to object o_2 , and o_2 is not related to object o_3 , then object o_1 is not related to o_3 .

Other relationships exist, but they are derived from these four relationships. Relationships 1 and 2 are converses. Relationships 3 and 4 are, in essence, the same rule. A single, more general rule could encompass both of them. Figure 1 shows these relationships in Horn clause logic. The most confusing relationship is relationship 1. Because the relationship requires an object to be 'not related' to all objects but one in a category, the size of the categories, S , must be encoded in the clause.

Key: [;] – disjunction [,] – conjunction
 [\=] – does not unify [:-] – implied by

1. Related by elimination.

```
rel(G,B) :-
    cat(X,B),
    cat(X,A), A\=B,
    ((S < 3) ; cat(X,C), C\=B, C\=A ),
    ((S < 4) ; cat(X,D), D\=C, D\=B, D\=A ),
    ((S < 5) ; cat(X,E), E\=D, E\=C, E\=B, E\=A ),
    cat(Y,G), Y\=X,
    notrel(A,G),
    ((S < 3) ; notrel(C,G) ),
    ((S < 4) ; notrel(D,G) ),
    ((S < 5) ; notrel(E,G) ),
```

2) Not related by exclusion.

```
notrel(G,B) :-
    cat(X,A),
    cat(X,B), A\=B,
    cat(_,G),
    rel(A,G).
```

3) Related by transitivity.

```
rel(B,G) :-
    rel(B,T),
    rel(T,G).
```

4) Not related by transitivity.

```
notrel(B,G) :-
    rel(B,T),
    notrel(T,G).
```

Figure 1: Horn clause relationships implicit in logic puzzles

1.3 SOLVING LOGIC PUZZLES

Using these relationships and the current KB, new facts can be inferred and added to KB. It is important to note the recursive nature of the above relationships and their corresponding sentences in Figure 1. In theory, each relationship can be queried and answered from the original knowledge base and implicit relationships. However, this recursive method has a high computational cost, and supports the possibility of an infinitely deep recursion if the relationships are not ordered correctly.

When an inference procedure is inefficient, it is necessary to reduce the expressiveness of the language or limit the number of inference steps (Selman and Kautz, 1996). Since we cannot reduce the expressiveness of our language (Horn clauses are very basic), we must limit the number of inference steps. Normally, doing so destroys the decidedness of the queries. When a query returns negative, one cannot know if the KB does not entail the query or if the inference procedure simply failed to find it.

Logic puzzles possess a special property that makes this limit acceptable. In fact, the limit is preferred. When solving logic problems, the answer to every query is entailed in the KB. This allows for setting the inference depth as low as desired (even to 1), at the cost of iterations through the space of possible queries. This space, for logic puzzles, stays small. The iteration is the process of inferring new sentences until the answer to the query is found.

GIVEN: A set of objects, A set of relationships between those objects (clues)
RETURN: A set of relationships for every pair of objects in different categories
METHOD

1. Place list of unknown relationships in KB.
2. Remove one unknown relationship, UR. If none exist, then done.
3. Query 'related(UR)?'
4. If true, then add UR to KB as related(UR). Goto 2.
5. Query 'notrelated(UR)?'
6. If true, then add UR to KB as notrelated(UR).
7. Goto 2.

Figure 2: Algorithm for solving a logic puzzle

When people solve a logic problem, we normally employ a grid summarizing known relationships - a visual representation of the knowledge base and of facts unknown. Each square in the grid marks the intersection of two objects in different categories. After reading each clue or making an inference, the square representing the newly found relationship is marked as "related" or "not related", usually with a "•" and an "X", respectively. A puzzle is considered solved when all squares - that is, all relationships - have been marked. See Appendix A for an example of this relationship grid.

To solve a logic problem efficiently using Horn clauses, the sentences of Figure 1 must be converted to non-recursive sentences. Then, after creating a list of unknown relationships, query each relationship as 'related' and 'not related'. If the current KB entails the query, add that fact to the KB; otherwise, put that fact at the end of the list and proceed with the next fact. This algorithm is shown in Figure 2.

This procedure is guaranteed to find the solution. In the worst case, n^2 inferences will occur, where n is the number of unassigned relationships that exist before solving. The expected case requires substantially fewer inferences.

1.4 PROPERTIES OF A CHALLENGING LOGIC PUZZLE

A challenging puzzle possesses one or more of the following properties:

- 1) The objects in the puzzle are similar or confusing.

Distinct objects with clear semantic ties make it easier for human solvers to make inferences. For example, consider a puzzle that asks you to match objects in the following categories: name, age, and occupation. If we replace the names, ages, and occupations with distinct ten digit numbers, the puzzle becomes harder to solve. Logically and mechanically, the process of solving the puzzle does not change, but our reduced ability to associate and distinguish objects limits our speed of inference.

- 2) The puzzle contains a limited amount of redundant logic.

Clues that provide multiple paths of inference to a single fact can be viewed as partially redundant. When multiple ways of inferring a fact exist, finding a single path of inference for that fact becomes easier.

3) The inferences in the puzzle are "hard".

A limited number of inference types exist in logic puzzles (listed in section 1.2). The degree of difficulty in making these inferences depends on the inference engine. For Prolog (Horn clauses), the 'not related by exclusion' inference is the most challenging simply because it must satisfy more terms than other inference types. For people, the transitive inferences are the most difficult. This is not to say that people are inept at transitive inferences. We simply find the elimination and exclusion cases extremely easy. These cases are visually evident from the relationship grid described in the previous section. In fact, after solving one or two puzzles, inferences (1) and (2) become automatic. We "just know" that after marking a square as "related", we can mark every other object in the same category as "not related."

In this project, property (1) is not analyzed. I attempted to neutralize this property in my tests (see section 4.1). My work in this project focused on property (2). My attempt to limit the amount of redundant logic is described in detail in the remaining sections. Property (3) was not considered in the design of the algorithms. Omitting this property from consideration seriously affected the outcomes of my tests.

2 POSSIBLE METHODS FOR CREATING LOGIC PUZZLES

After a lengthy introduction describing how to solve logic problems, reminding the reader of my original purpose may be helpful. This project focused on finding a minimal set of clues that entail the solution to a logic puzzle. The clues should give as little information as possible, while still entailing the solution. The second property of section 1.4 describes why it makes sense to do this. In short, this work aimed to create hard-to-solve problems.

To make a problem hard to solve, we want to maximize the number and difficulty of the inferences required to arrive at the solution. Since each clue is an element of the solution, and each inference creates an element of the solution, the number of inferences required to solve the problem equals the difference between the number of elements in the solution and the number of clues. Therefore, minimizing the number of clues maximizes the number of inferences required. At the same time, we want to maximize the difficulty of the average inference.

When searching for ways to minimize the number of clues, I found a large amount of literature addressing the converse of this problem. The demand for methods to prove facts in a KB, infer knowledge from a KB, and expand a KB is very high. This makes sense, since countless applications for such methods exist. However, *minimizing* knowledge bases is not a topic of hot research in AI.

This problem has been proven NP-hard (Garey & Johnson, 1979; Pudlák, 1975), so hope of finding the perfect solution must be abandoned. As with most NP-hard problems, we must find a

way to tractably achieve a close-to-optimal solution. The following subsections summarize the two methods I implemented while trying to find a close-to-optimal solution.

2.1 ABDUCTION

Abduction is "the generation of explanations for a set of events from a given domain theory." (Console, Dupre, Torasso, 1991) Translated into the puzzle domain, abduction is the generation of explanations for an element of a solution using the clues given. A few authors use the term "explanation" when describing this process (Huffman & Laird, 1995; Freuder, Likitvivanavong & Wallace, 2000). Another concept, inverse entailment, works toward the same goal, but in other ways (Muggleton, 1995).

The standard example for describing abduction involves a simple KB (Console et al., 1991):

$$\text{KB} = \{ \begin{array}{l} \text{rained last night} \rightarrow \text{grass is wet,} \\ \text{sprinkler was on} \rightarrow \text{grass is wet,} \\ \text{grass is wet} \rightarrow \text{grass is cold and shiny,} \\ \text{grass is wet} \rightarrow \text{shoes are wet} \end{array} \}$$

Given the fact *grass is cold and shiny*, what are the possible causes? The process of abduction enumerates the possible causes of a given fact using the knowledge base. In the example above, *rained last night* and *sprinkler was on* are the two possible explanations of *grass is cold and shiny*. Notice that *grass is wet* is not a possible cause because it is not an *abducible* atom, that is, an atom that is not derived from other atoms (Console et al., 1991).

In the context of logic puzzles, abduction can help minimize a set of clues that entail the solution. Given a set of un-minimized clues (the abducible atoms), an abduction method can generate explanations for each element of the solution. Analyzing these explanations, one can determine which sets of atoms can be removed, still leaving the solution entailed. Removing the largest set should leave us with a minimal set of clues.

The problem with this method lies in the large number of possible explanations. Most situations allowing abduction have this problem. Although many techniques have been developed to minimize the number of explanations (Kakas, Kowalski, Toni, 1992), most of these techniques do not apply in the general case. Most techniques rely on exploiting the structure of the domain. One technique divides the knowledge base into sets of related facts; when an explanation is needed, only the relevant sets are explored. Another technique tests each explanation on "integrity constraints". These constraints eliminate explanations that are possible logically, but fail to hold semantically (Kakas et al. 1992). In logic problems, exploiting the structure of the domain does not improve the situation. Every element is related in some way to all other elements. This strongly-related property of the logic puzzle domain causes the number of explanations to blow up very quickly.

I attempted to implement a modified version of abduction using Prolog. The algorithm, described below, is summarized in Figure 3.

- GIVEN:** A set of relationships defining the logic puzzle's solution
RETURN: A set of clues that entail the logic puzzle's solution
METHOD
1. Add random clues to KB until solution is entailed.
 2. For each clue, try solving the puzzle without it. If solvable, add clue to UNNEEDED
 3. If UNNEEDED empty, then done.
 4. Solve the puzzle, recording explanations for every inference required.
 5. Find the clue in UNNEEDED used the least # of times while solving and throw it out.
 6. Empty UNNEEDED and repeat at 2.

Figure 3: Modified abduction algorithm

The algorithm begins by generating an un-minimized set of clues that entail the solution. This step is trivial. The algorithm simply adds random clues to the KB until the solution is entailed. Then, an attempt is made at solving the puzzle with one clue removed from the KB. If the puzzle can be solved, then the removed clue may not be needed. This clue is added to the UNNEEDED list. This step, step 2 in Figure 3, is repeated for each clue.

In step 4, the puzzle is solved. While solving, the algorithm records the explanation for each inference. After the puzzle is solved, which clues were used and how often is known. Step 5 selects the clue used the least often and throws it out. I repeat the process until all clues are in the required list.

This method did not work. The puzzles created were solved very easily. The algorithm was based on a faulty heuristic assuming clues used less often were more likely to be unneeded. This experiment gave me the intuitive reason as to why this problem is NP-hard. Probably, no incremental solution exists for this problem. In other words, every combination of clues must be tested to find the optimal solution.

2.2 DYNAMIC PROGRAMMING

The lessons learned from the abduction experiment sent my search in a different direction than I had originally envisioned. I decided to abandon my initial goal of constructing a minimal set of clues logically and focus on finding an efficient method to test all combinations of clues. I had two reasons for adopting this strategy:

- 1) The lack of structure in the logic puzzle domain causes problems for abduction and similar procedures. Most techniques designed to make abduction tractable do not apply.
- 2) The abduction experiment and the NP-hardness of the problem suggest that all combinations must be tested before an optimal solution can be found.

Using dynamic programming techniques (Cormen, Leiserson, & Rivest, 1990), it is possible to take a set of clues that entail the solution and minimize them. Recall the assumption from above that minimizing the number of clues effectively maximizes the number of inferences. The dynamic programming approach I took begins by finding every clue that can be removed without making the problem unsolvable. In other words, when that clue is removed, the remaining clues still entail the solution. Using this set of clues, the algorithm finds every set of two clues that can

be removed simultaneously. The algorithm continues increasing the number of clues in the set until the set cannot get any larger. Removing the largest set leaves the minimal set of clues.

This algorithm, if implemented exactly as described above, has the potential to become very costly in terms of computation time. If you have a set of n clues to minimize, and the optimal solution removes k of those clues, then the minimum number of sets tested throughout the computation will be

$$\sum_{i=1}^{k-1} \binom{k}{i}.$$

Proof.

Since all k items can be removed with the solution still entailed, then any subset $c \subseteq k$ can be removed with the solution still entailed. This is intuitively obvious, but the formal expression of this idea is stated as:

$$(KB-k) \models \text{soln} \rightarrow [(KB-k) \cup (k-c)] \models \text{soln}.$$

The above statement is only true if $k-c$ is consistent with $KB - k$. In logic puzzles, this is always true. Adding any clue to the knowledge base always results in a consistent KB.

Therefore, since all possible subsets of size i must be checked in step i , k choose i subsets must be checked at each step. Each "check" requires that the system attempt to solve the puzzle with the given knowledge base.

The summation of all levels increases rapidly with respect to k . If $k = 10$, then the given logic problem will have to be solved a minimum of 1022 times. In addition to the k clues that are not needed, other clues not in the final removal set appear in the removal sets at lower levels. This can substantially increase the number of times a problem must be solved.

To keep this calculation reasonable, we need to try to reduce k with other methods between the dynamic program steps. In addition, we would like to reduce the average amount of time needed to solve the puzzle for each check. The algorithm below tackles both of these issues by caching a partial solution created by the set of clues known to be needed. The algorithm described below shows how this "known" set of clues is found at each step.

3 DYNAMIC PROGRAMMING ALGORITHM

Figure 4 summarizes the algorithm explained below. The algorithm was implemented in Prolog.

The algorithm takes, as input, a set of un-minimized clues that entail the solution. Each clue, X , is inserted as *active*(X) into the KB. The KB also contains the sentence, $RC([\emptyset])$, meaning no removal combinations are known thus far. A removable combination, RC , is a set of clues such that the KB, without RC , still entails the solution.

Step 1 of the algorithm in Figure 4 takes an RC with i elements from the KB and tries to add another clue to the set, producing a new RC of size $i+1$ ($i = 0$ for the first pass through the

GIVEN: A set of relationships defining the logic puzzle's solution, and a set of un-minimized clues

RETURN: A maximum size set of clues that can be safely removed

METHOD

1. For all removable combinations in KB, rc,
KB = add_removable_combination(rc, KB)
2. Determine which clues are not found in new RCs and retire them.
3. Partially solve the problem using only retired relationships, retiring each new relationship inferred.
4. Remove any RCs that include a newly inferred relationship and add that relationship to the set of Unneeded Clues.
5. If retired set contains the solution, then done. Return the union of any removable combination in KB and the set of all UnneededClues.
6. If RCs exist, save one as CurrentLargest. Goto 1.
7. Return the union of the CurrentLargest and the set of all UnneededClues.

Figure 4: Dynamic programming algorithm

algorithm). A single RC can produce many RCs. Step 1 is repeated for each RC of size i currently in the KB.

To test if a combination is removable, the algorithm temporarily removes the set of elements from the knowledge base and tries to solve the puzzle. If the puzzle can be solved, then the set is deemed removable. I call this process an *RC check*. After all RCs have been processed, step 2 determines which clues are present in the new RCs. Clues not found in the new RCs are now considered necessary. The algorithm marks these clues as *retired* and removes them from the *active* list.

Step 3 reduces the average amount of time required for each RC check. Using the retired set of clues, new relationships are inferred and added to the KB as retired clues. Making these inferences now eliminates the need to do so in each future RC check.

Step 4 reduces the number of combinations processed. If one of the newly inferred relationships found in step 3 can be found in the set of RCs, then that relationship must be one of the clues. Since the clue has been inferred from a retired clue, the clue is known to be redundant. This clue is added to the set of Unneeded Clues and the RCs that contain it are removed.

If the retired set equals the solution or if no RCs of greater size can be produced, then the algorithm is complete.

4 TEST RESULTS

This section describes the tests used to compare the difficulty of the puzzles created by the algorithms to published versions of similar puzzles. The results of the tests are summarized.

4.1 TESTS

To test the level of difficulty of puzzles the algorithm creates, I selected three logic puzzles of varying size and difficulty from the March 2001 edition of “Original Logic Problems” by PennyPress. I randomly chose a solution for all three puzzles - a solution that differed from the published solution - and set these solutions as input to my puzzle creator.

Using the clues given as output and the clues given in the published version of the puzzle, I created a document that contained two versions of each puzzle. With the exception of these clues (and the solution), each version was identical. Each had the same story, the same logic diagram, and the same presentation of the clues.

I feel that an identical style and presentation of the clues is important. If you recall property (1) in Section 1.4, the presentation of objects and relationships make a difference in a person’s ability to make inferences. To nullify this effect, I reduced the complex sentences presented in the published version to straightforward relationships that matched the output from the puzzle creator.

I distributed the three sets of puzzles to 55 people and asked them to record the amount of time it took to solve each puzzle. 18 people responded. Not every participant solved every problem. The test was designed to simply determine which puzzle required a longer time for the average person to solve. One pair of these puzzles is attached as Appendix A.

4.2 TEST DATA

Table 1 shows the properties and solving statistics for the six puzzles. *Objects / Category* and *# Relationships* indicate the size of the puzzle given. *# Clues* refers to the number of clues explicitly given to the solver. *# Positive Clues* refers to the number of clues stating that two objects are related. (A negative clue states that two objects are not related.) The importance of this metric stems from property (3) of Section 1.4. The “exclusion” inferences following a positive relation were found to be easy for people. In fact, I suggested that they became ‘automatic’ for people after solving a few puzzles.

The last four rows in Table 1 give statistics on how long it took the participants to solve the puzzles. The important statistic, *# with greater solve time*, shows how many participants found the problem more difficult than its mirror problem.

Notice that in puzzle set 1, the puzzle generated by the algorithm seemed to outperform the PennyPress puzzles. The reverse is true for sets 2 and 3. After interviewing several participants, I found that puzzle 1a served as a warm-up puzzle. After solving 1a, the participants had increased their abilities, making 1b seem a lot easier. For this reason, puzzle set 1 is shaded, indicating that I do not consider the results from this set useful.

Since positive clues often trigger ‘automatic’ inferences, the facts these inferences uncover are essentially free clues. Table 2 shows the number of automatic clues in the puzzles tested.

Table 1: Properties and solving statistics of the test puzzles

Test Puzzle #		1a	1b	2b	2a	3a	3b
Created by		Algorithm	PennyPress	Algorithm	PennyPress	Algorithm	PennyPress
Puzzle Properties	Objects/Category	4	4	5	5	5	5
	# Relationships	96	96	150	150	150	150
	# Clues	7	10	12	16	13	17
	# Positive Clues	6	5	9	5	8	6
Test Results	Maximum solve time	38	24	18	45	30	90
	Minimum solve time	3	2.75	2.75	5	2.5	4
	Average solve time	16.2	9.4	7.5	20.0	13.2	23.4
	# w/ greater solve time	17	3	1	17	4	15

Table 2: Comparison of the number of Automatic clues in the test puzzles

Test Puzzle #	2b	2a	3a	3b
Created by	Algorithm	PennyPress	Algorithm	PennyPress
Objects/Category	5	5	5	5
# Relationships	150	150	150	150
# Actual Clues	12	16	13	17
# Positive Clues	9	5	8	6
# Automatic clues	72	46	74	66

5. ANALYSIS OF RESULTS

5.1 VALIDITY OF TESTS

The goal of my research was to create puzzles that were difficult for people. Therefore, testing the puzzles on people was an obvious necessity. Testing performance relative to published puzzles may not be completely valid, however, since published puzzles vary in difficulty.

Since those taking my tests had a widely varying level of experience with logic programs, the absolute solving times have no meaning. The tests only measured which test required more time to solve in each pair of puzzles. Based on the results, I believe the varying experience of the participants did affect the first two puzzles. If the solver had no previous experience solving logic puzzles, the large improvement in their skills between the first few puzzles will skew the results. Therefore, I will ignore puzzle set 1.

5.2 INTERPRETATION OF DATA

All puzzles created by the algorithm contained fewer clues than the PennyPress versions. Thus, the goal of maximizing the number of inferences seems satisfied. The PennyPress versions required a fewer number of inferences to solve. However, the PennyPress puzzles took, on

average, much longer to solve. From these two facts, it follows that the average inference in the PennyPress puzzle was more difficult. I believe this is due to the larger number of positive clues given in my puzzles. Table 2 shows that the number of automatic clues is much less for the PennyPress puzzles. Had I attempted to minimize the number of automatic clues, I believe the results would have been much better.

Originally, I expected the puzzles created by the above algorithm to compete well with the published puzzles. I assumed that the published puzzles would be harder on average, just not to such a high degree. The weakness in the puzzles my algorithm generated lies in the neglect of property (3) described in Section 1.4. The algorithm I created only maximizes the number of inferences, not their difficulty. After I realized my error, the results did not seem so surprising.

6. CONCLUSIONS

My experiments with abduction do not conclusively show that abduction is ill suited for logic puzzle generation. However, research shows that most techniques designed to make abduction tractable rely on the properties of structured domains; a logic puzzle does not have a structured domain.

The dynamic programming approach described above produces a minimal set of clues from a given un-minimized set of clues that entail the solution. The number of clues produced numbered less than similar published puzzles. Steps in the algorithm were necessary to make the algorithm tractable. The problem with this approach is that all types of inferences are treated equally. Since the algorithm does not take the *type* of inferences into account, the puzzles created were not as difficult as published versions.

When generating the input to the dynamic programming algorithm (the un-minimized set of clues), positive relationships could be disallowed. Forcing the dynamic programming algorithm to minimize a set of negative relationships would result in fewer automatic inferences and a more difficult problem. Time constraints forced me to leave this hypothesis as future work.

7 SUMMARY

Solving logic puzzles is efficient and straightforward using Horn clauses. The four types of inferences needed to solve a puzzle were described. Creating logic puzzles - at least challenging ones - is not as simple. Three properties of challenging puzzles were found that served as a guide for creating challenging logic puzzles. One property suggests that finding a set of clues containing a small amount of logical redundancy increases the puzzle's difficulty. Finding the set that contain the least logical redundancy is NP-hard. The author's attempt to use abduction to produce close-to-optimal sets of clues failed. A dynamic programming approach produces puzzles that require the maximum number of inferences while solving, but does not take the difficulty of each inference into consideration. Further improvements to the algorithm presented could achieve results of higher quality.

APPENDIX A: ONE PAIR OF LOGIC PUZZLES USED IN TEST

Note: The puzzle on this page is an adaptation of PennyPress's version of this puzzle. The logic puzzle on the next page was created by my algorithm.

SET IN STONE

Today, five men, each of whom has a wife with an April birthday, spent the day searching for the perfect gifts. After seeing several exquisite items in the windows of local jewelry stores, each man purchased a piece of jewelry set with a different gemstone. Each piece was bought in a different store. From the information provided, match each husband (Adrian, Bruce, Dave, Hank, or Tom) with the wife (Janet, Odette, Rhonda, Thelma, or Ursula) for whom he is buying a gift, the store (one is Suchy's Fine Jewelry) where each made his purchase, and the type of gemstone (one is a sapphire) set in the piece each man bought.

		WIFE					STORE					GEMSTONE				
		Janet	Odette	Rhonda	Thelma	Ursula	Banks & Bailey's	Bennett's	Harvey's	Lux and Bond's	Suchy's	Emerald	Opal	Ruby	Sapphire	Topaz
Husband	Adrian															
	Bruce															
	Dave															
	Hank															
	Tom															
Gemstone	Emerald															
	Opal															
	Ruby															
	Sapphire															
	Topaz															
Store	Bank & B's															
	Bennett's															
	Harvey's															
	Lux & B's															
	Suchy's															

- Janet is not married to Adrian or Tom.
- Hank is married to Ursula.
- Adrian is not married to Thelma.
- Dave did not shop at Bank's & Bailey's or Lux & Bond's.
- Adrian did not shop at Harvey's.
- Tom bought the emerald.
- Dave did not buy the Ruby.
- Hank did not buy the Topaz.
- Janet did not get the Topaz.
- Rhonda received the Opal.
- The Ruby was bought at Bennett's.
- The Emerald was not bought at Bank's & Bailey's.
- Thelma received the gift from Bank's & Bailey's.
- Janet did not receive a gift from Harvey's.

2a.

1..1.1 Time: _____

Completed? _____

SET IN STONE

Today, five men, each of whom has a wife with an April birthday, spent the day searching for the perfect gifts. After seeing several exquisite items in the windows of local jewelry stores, each man purchased a piece of jewelry set with a different gemstone. Each piece was bought in a different store. From the information provided, match each husband (Adrian, Bruce, Dave, Hank, or Tom) with the wife (Janet, Odette, Rhonda, Thelma, or Ursula) for whom he is buying a gift, the store (one is Suchy's Fine Jewelry) where each made his purchase, and the type of gemstone (one is a sapphire) set in the piece each man bought.

		WIFE					STORE					GEMSTONE				
		Janet	Odette	Rhonda	Thelma	Ursula	Banks & Bailey's	Bennett's	Harvey's	Lux and Bond's	Suchy's	Emerald	Opal	Ruby	Sapphire	Topaz
Husband	Adrian															
	Bruce															
	Dave															
	Hank															
	Tom															
Gemstone	Emerald															
	Opal															
	Ruby															
	Sapphire															
	Topaz															
Store	Bank & B's															
	Bennett's															
	Harvey's															
	Lux & B's															
	Suchy's															

- Adrian bought his gift at Banks and Bailey's.
- Bruce bought his gift at Suchy's.
- Dave did not buy his gift at Lux and Bond's.
- The sapphire was bought at Suchy's.
- Dave bought the Opal.
- Thelma is not married to Hank.
- Hank bought his gift at Bennett's.
- The Ruby was not bought at Lux and Bond's.
- Adrian and Ursula are married.
- Tom and Odette are married.
- Ursula received the Topaz.
- Rhonda married Dave.

2b.

1.1.2 Time: _____

Completed? _____

REFERENCES

- Console, L., Dupre, D. T., and Torasso, P. (1991). "On the Relationship between Abduction and Deduction", *Journal of Logic and Computation*, 1(5),661-690.
- Cormen, T.H., Leiserson, C.E. & Rivest, R.L. (1990). "Introduction to Algorithms", MIT press, Cambridge, MA.
- Freuder, E., Likitvivatanavong, C. & Wallace, R.J. (2000). "A Case Study in Explanation and Implication", CP2000 Analysis and Visualization of Constraint Programs and Solvers Workshop. Available: <http://www.pms.informatik.uni-muenchen.de/ereignisse/02paper.ps>.
- Garey, M.R. & Johnson, D.S. (1979). "Computers and Intractability: A Guide to the Theory of NPCompleteness". W.H. Freeman and Company, New York.
- Huffman, S.B. and Laird, J.E. (1995) "Flexibly Instructable Agents", *JAIR*, 3, 271-324.
- Kakas, A. C., Kowalski, R. A. & Toni, F. (1992) "Abductive Logic Programming". *Journal of Logic and Computation*,2(6), 719-770.
- Muggleton, S. (1995). "Inverse entailment and PROLOG". *New Generation Computing*, 13, 245-286.
- Pudlák, P. (1975), "Polynomially complete problems in the logic of automated discovery," in *Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 32*, Springer, Berlin, 358-361.
- Selman, B. & Kautz, H. (1996). "Knowledge compilation and theory approximation", *Journal of the ACM*, 43(2), 193-224.