Automatic Solutions of Logic Puzzles

Author: Peter Sempolinski

Automatic Solutions of Logic Puzzles

Peter Sempolinski
Advisor: Prof. Howard Straubing

April 30, 2009

Abstract

The use of computer programs to automatically solve logic puzzles is examined in this work. A typical example of this type of logic puzzle is one in which there are five people, with five different occupations and five different color houses. The task is to use various clues to determine which occupation and which color belongs to each person. The clues to this type of puzzle often are statements such as, "John is not the barber," or "Joe lives in the blue house." These puzzles range widely in complexity with varying numbers of objects to identify and varying numbers of characteristics that need to be identified for each object.

With respect to the theoretical aspects of solving these puzzles automatically, this work proves that the problem of determining, given a logic puzzle, whether or not that logic puzzle has a solution is NP-Complete. This implies, provided that P is not equal to NP, that, for large inputs, automated solvers for these puzzles will not be efficient in all cases.

Having proved this, this work proceeds to seek methods that will work for solving these puzzles efficiently in most cases. To that end, each logic puzzle can be encoded as an instance of boolean satisfiability. Two possible encodings are proposed that both translate logic puzzles into boolean formulas in Conjunctive Normal Form. Using a selection of test puzzles, a group of boolean satisfiability solvers is used to solve these puzzles in both encodings. In most cases, these simple solvers are successful in producing solutions efficiently.

1 The Puzzle

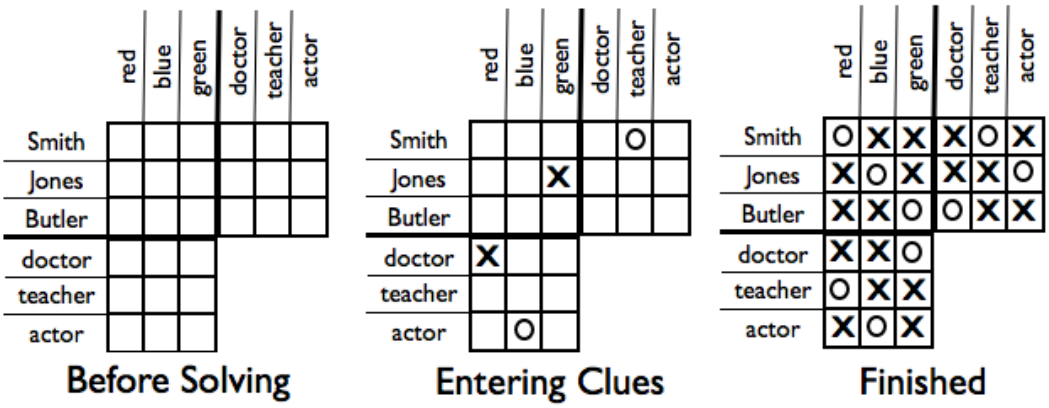
The phrase "logic puzzle," can refer to many of the wide variety of puzzles that have been formulated for people to solve. However, when one hears the phrase, "logic puzzle," typically, one usually envisions a problem somewhat like this: There are three people, named Mr. Smith, Mr. Jones and Mr. Butler. Each of these three people has a different color shirt. The colors of the shirts are red, green and blue. Each of the three people has a different occupation, one is a doctor, one is a teacher and one is an actor. Smith is the Teacher. The actor has a blue shirt. The doctor does not have a red shirt. Jones does not have a green shirt. The task of the solver is to match each person to his occupation and shirt color.

This is just a simple example of the many puzzles that fall under the common name: "Logic Puzzle." These problems are a staple of many puzzle enthusiasts' repertoire. Numerous examples, of varying difficulty and complexity, can be obtained, in various puzzle publications at newsstands and on multiple online puzzle sites, such as:

`<http://www.puzzles.com/projects/LogicProblemsArchive.html>`

In order to solve these puzzles, logical reasoning is used. In the above example, it can be reasoned that if the actor has the blue shirt, then the doctor does not. But, the doctor also does not have the red shirt. Therefore, the doctor has the green shirt and the teacher has the red shirt. But Smith is the teacher, so Smith has the red shirt. Therefore, Jones does not have the red shirt. But Jones also does not have the green shirt. So, Jones has the blue shirt. This leaves Butler with the green shirt. That means he is the doctor. Since Smith is the teacher, that leaves Jones as the actor. As such, the solution is: Mr. Butler, doctor, green shirt; Mr. Jones, actor, blue shirt; Mr. Smith, teacher, red shirt.

Typically, a grid of some sort is provided with these puzzles so that solvers can mark out what pairs of characteristics belong to the same object and which don't. Below, there is a simple example of what such a grid would look like for this puzzle. In this case, "o"s denote that two characteristics match and "x"s denote that two characteristics don't:



This notion of "Logic Puzzle," is actually fairly broad. In practice, many different kinds of clues are used. Sometimes clues say that one person is older than another. Sometimes they indicate that one person is seated next to another at a table. In spite of this variation, it is still possible to define these logic puzzles in a precise way. Once such a description is made, it is possible to examine these puzzles in the context of computational theory. Also, given a standard definition of these puzzles, automated solution finders can be programmed.

However, when we begin to examine the possibility of automatic solutions to these puzzles, we encounter the problem of computational complexity. This is of interest because it determines how feasible it is to use an algorithm to automatically solve instances of this problem. From a computational perspective, some would hope that these puzzles can be solved in polynomial time, since that would mean we could write an optimal solver for them. Of course, some would prefer that these puzzles were not this easy, since much of the point of puzzles is that they should be challenging. Regardless, if it is not the case that an efficient solver exists, if, for example, the problem is NP-Complete, the approach to automated solutions shifts from finding a fast solver, which probably does not exist, to finding good solutions that are effective at solving most instances of the puzzle, typically by using heuristics.

Specifically, with respect to these logic puzzles, first, in order to accurately examine these puzzles from a theoretical perspective, we will carefully define what we mean by "logic puzzle". Second, we will prove that, in the general case, these logic puzzles are NP-complete. Third, since NP-completeness

probably implies that the puzzle is intractable for large inputs in the general case, we will approach the question of how to program automated solvers that at least perform well on most inputs.

2 Logic Puzzle Defined

As stated above, the type of logic puzzle which we are examining is one in which several items and their characteristics must be determined based on a set of clues. Therefore, we will define the following terms. For each puzzle we define m "domains". Each domain is a set, each of which contains n "characteristics". Following the listing of these domains and characteristics, a puzzle contains a set of clues about those domains and characteristics. For example, one puzzle might have the domains of (First Name, Last Name, Occupation). The domains might be: {Aaron, Bob, Clive}, {Amherst, Barth, Chester}, and {Actor, Barber, Carpenter}. The solution to the puzzle is a set of bijections between the domains, one for each pair of domains. Let $f_{i,j} : D_i \rightarrow D_j$ be the bijection mapping domain i to domain j. In order to be a valid solution, the following properties must hold: First, $f_{i,j} \circ f_{j,i} = f_I(x)$ where $f_I(x)$ is the identity function for D_i. Second, $f_{i,j} \circ f_{j,k} = f_{i,k}$. This, in effect, insures that the mappings from one domain to another are consistent definitions of objects. In other words, if Aaron mapped to Barth and Barth mapped to Actor, then, this the same as saying Aaron's last name is Barth and Mr. Barth is the Actor. From this it must follow that Aaron is the actor. Finally, these functions must be consistent with the list of clues provided with the puzzle.

The notion of a clue must therefore be defined. In practice, this can be very difficult, as typical publications often have a wide variety of types of clues. To deal with this, first we define a "term": Let $1 \leq i \leq m$, $1 \leq j \leq n$. The term $a_{i,j}$ denotes the jth object in the ith domain. For example, from above we can say $Bob = a_{FirstName,Bob}$ or use $a_{1,2}$ as shorthand for saying the second object in the first domain. From this, the basic atomic clue is an expression of the statement that two terms correspond to the same object. For example, using the numerical shorthand and the ordering above, the clue, "Mr. Chester is the Barber." is denoted: $a_{2,3} = a_{3,2}$. In terms of the bijection functions that form the solution, $a_{i,j} = a_{k,m} \Leftrightarrow f_{i,k}(j) = m$. That is, term j in domain i is the same object as term m in domain k if and only if the function between domains i and k maps item j to item m.

In order to express more complex clues, boolean combinations of these atomic terms can be taken. In the most basic example, to say that two terms are not the same object is to simply take the negation of the atomic clue that would say that they are. From this, more complex clues can be thought of as boolean combinations of simpler ones. A potential solution can be used to determine the truth values of the literals in this boolean function. If the boolean function evaluates to true, then that clue is consistent with that solution.

In many puzzles, one of the domains is ordered in some way. For example, sometimes "ages" appears as a domain in a puzzle. Clues in the puzzle then take advantage of this ordering by asserting that one person is younger or older than another. Similarly, one might have a domain for houses along a street, with clues stating that one house is somewhere to the left of or to the right of another.

Among the many types of clues that take advantage of ordered domains, the most basic is one that says that one object is greater than another along some ordered domain. We can define this type of clue as a boolean function with three inputs, a domain number and two terms so that a_2 is greater than a_1, (defined in terms of obj, which is the number of objects in the puzzle):
$Order(d, a_1, a_2) = \neg(a_1 = a_2) \wedge \bigwedge_{q=1}^{obj} \bigwedge_{r=q+1}^{obj}((a_2 = a_{d,q}) \rightarrow \neg(a_1 = a_{d,r}))$

Essentially, this formula says, first, that the two objects a_1 and a_2 are not the same and, second, for each possible placement of the second object along the ordered domain, the first object is not greater than the second. This insures that the second object is greater than the first.

For example, given the domains $\{Aaron, Bob, Clive, David, Ernest\}$ and $\{25, 26, 27, 28, 29\}$, corresponding to names and ages, the clue that Aaron is older than Bob is translated the following way, using the numerical shorthand:

$\neg(a_{1,1} = a_{1,2}) \wedge ((a_{1,2} = a_{2,1}) \rightarrow \neg(a_{1,1} = a_{2,2}))$
$\wedge ((a_{1,2} = a_{2,1}) \rightarrow \neg(a_{1,1} = a_{2,3})) \wedge ((a_{1,2} = a_{2,1}) \rightarrow \neg(a_{1,1} = a_{2,4}))$
$\wedge ((a_{1,2} = a_{2,1}) \rightarrow \neg(a_{1,1} = a_{2,5})) \wedge ((a_{1,2} = a_{2,2}) \rightarrow \neg(a_{1,1} = a_{2,3}))$
$\wedge ((a_{1,2} = a_{2,2}) \rightarrow \neg(a_{1,1} = a_{2,4})) \wedge ((a_{1,2} = a_{2,2}) \rightarrow \neg(a_{1,1} = a_{2,5}))$
$\wedge ((a_{1,2} = a_{2,3}) \rightarrow \neg(a_{1,1} = a_{2,4})) \wedge ((a_{1,2} = a_{2,3}) \rightarrow \neg(a_{1,1} = a_{2,5}))$
$\wedge ((a_{1,2} = a_{2,4}) \rightarrow \neg(a_{1,1} = a_{2,5}))$

If we wish, we can re-write this with disjunctions instead of implications:

$\neg(a_{1,1} = a_{1,2}) \wedge (\neg(a_{1,2} = a_{2,1}) \vee \neg(a_{1,1} = a_{2,2}))$
$\wedge (\neg(a_{1,2} = a_{2,1}) \vee \neg(a_{1,1} = a_{2,3})) \wedge (\neg(a_{1,2} = a_{2,1}) \vee \neg(a_{1,1} = a_{2,4}))$
$\wedge (\neg(a_{1,2} = a_{2,1}) \vee \neg(a_{1,1} = a_{2,5})) \wedge (\neg(a_{1,2} = a_{2,2}) \vee \neg(a_{1,1} = a_{2,3}))$

$\wedge (\neg(a_{1,2} = a_{2,2}) \vee \neg(a_{1,1} = a_{2,4})) \wedge (\neg(a_{1,2} = a_{2,2}) \vee \neg(a_{1,1} = a_{2,5}))$
$\wedge (\neg(a_{1,2} = a_{2,3}) \vee \neg(a_{1,1} = a_{2,4})) \wedge (\neg(a_{1,2} = a_{2,3}) \vee \neg(a_{1,1} = a_{2,5}))$
$\wedge (\neg(a_{1,2} = a_{2,4}) \vee \neg(a_{1,1} = a_{2,5}))$

Nearly all of the other types of clues commonly found in these logic puzzles can be expressed as such a boolean function. Among the most common are the following:

$Gap(d, a_1, a_2, gap)$ which is defined such that along the ordered domain d, a_2 is exactly gap greater than a_1.
$Gap(d, a_1, a_2, gap) = \bigvee_{q=1}^{obj-gap}((a_1 = a_{d,q}) \wedge (a_2 = a_{d,q+gap}))$

$Gap2(d, a_1, a_2, gap)$ which is defined that along the ordered domain d, a_2 is exactly gap greater than OR less than a_1.

$$Gap2(d, a_1, a_2, gap) = \bigvee_{q=1}^{obj-gap}((a_1 = a_{d,q}) \wedge (a_2 = a_{d,q+gap})) \vee \bigvee_{q=gap+1}^{obj}((a_1 = a_{d,q}) \wedge (a_2 = a_{d,q-gap}))$$

For any clue, the result is some kind of boolean function. A proposed solution to any puzzle is correct only if all of these functions, for all of the clues, evaluate to true.

3 NP-Completness

The distinction between the time complexity classes of P and NP is one of the most important open questions in computational theory. Generally speaking, P is the class of all problems in which it is possible to devise an efficient polynomial time algorithm that determines if a solution exists. Typically, if it is possible to determine if a solution exists, it is not too difficult to find it. This means that problems in P are usually tractable, even with large inputs. The class NP, however, includes all problems in which, given a potential solution to the problem, it is possible to check, in polynomial time, the validity of that solution against a specific input. Of course, any problem in P is also in NP. If a solution can be found easily, it can be checked easily. However, whether or not there exist problems in NP that are not in P has yet to be proven.

One of the most important results from examining this problem is Cook's Theorem, which proved the existence of NP-Complete problems by proving that boolean satiability is NP-Complete. Boolean satisfiability is the problem

that, given an arbitrary boolean formula that contains a number of unknown variables, does a way exist to assign those variables values, ($true$ or $false$), such that the evaluation of the boolean string will be $true$?[3] If a problem is NP-Complete, this means that it is an NP problem that is at least as hard as any other NP problem. More technically, NP-Complete problems are those to which any other NP problem can be reduced in polynomial time. The significance of this is that, if there was an algorithm that could quickly solve any NP-Complete problem, it would also be able to quickly solve any problem in NP. However, more likely, no such algorithm exists. Therefore, proving that a problem is NP-Complete strongly suggests that no polynomial time algorithm exists for it. The significance of proving that boolean satiability is NP-Complete means that proving other problems are NP-Complete only requires reducing them to boolean satiability. Then, having proven those problems NP-Complete, other problems can be proved NP-Complete by reducing them to any other problem previously proved to be NP-Complete. Of course, the reduction itself must only take polynomial time for the proof to be valid.

As stated above, that task of determining whether a given logic puzzle has a solution is NP-complete. To prove this, we must prove that these puzzles are in NP and that they are NP-hard. To prove the first, we will show that it is possible to devise a polynomial time algorithm that, given a proposed solution to a given puzzle, can determine if that solution is, in fact, a correct solution to that puzzle. The existence of this algorithm implies that the existence of a solution for a logic puzzle can be determined in non-deterministic polynomial time. To prove that these puzzles are NP-hard, we will perform a reduction from the Partial Latin Squares Completion Problem, previously shown to be NP-complete [1]. Note that in the following proof we are only including the simplest form of clues possible. These clues are only to say that two characteristics either do or do not belong to the same object. From this point, the proof can be extend to encompass puzzles of all reasonable types of clues.

3.1 Logic Puzzles are NP

Proving that these puzzles are in NP requires confirming that, given a proposed solution to a logic puzzle, a polynomial time algorithm exists for checking if that solution is a valid solution to the puzzle. To confirm if a solution is valid, we must check two things. First, we must check that the

solution adheres to the general rules for logic puzzles. This is captured, above, in the condition that the solution, "functions" that map between the pairs of domains must be consistent with each other. Second, the solution must be consistent with the clues.

Our algorithm to check a solution to a puzzle is, therefore, in two parts. First, however, we should note that, in a puzzle with n domains and m objects, a solution must contain $n(n-1) = n^2 - n$ functions, between those domains. This is because each of n domains must have a function mapping to all of the other $(n-1)$ domains. Also, each of these functions must contain m mappings, one for each characteristic in the logic puzzle domain that is the domain of that function. In the first part of the algorithm, after checking that the proposed solution is a well-formed description of a set of $n^2 - n$ bijection functions of the correct size, each pair of functions is examined. There are $\frac{(n^2-n)^2-(n^2-n)}{2} = \frac{n^4-2n^3+n^2-n^2+n}{2} = \frac{n^4-2n^3+n}{2}$ such pairs. For each pair, if the two functions can be expressed as: $f_{i,j}$ and $f_{j,i}$ where i and j are unique integers, then, for each of the m inputs of the first function, the solution checker confirms that: $f_{i,j} \circ f_{j,i} = f_I(x)$ where $f_I(x)$ is the identity function for the input domain of $f_{i,j}$. If, however, the pair of functions is in the pattern: $f_{i,j}$ and $f_{j,k}$ where i,j and k are unique integers, then, for each of the m inputs of the first function, it is confirmed that $f_{i,j} \circ f_{j,k} = f_{i,k}$. Since there are a polynomial number of pairs of functions to check and a polynomial number of possible inputs to each function, this step will run in polynomial time.

The second step of checking if a solution is valid involves checking each clue. As we said above, each clue is a boolean formula in which the literals are connections between two objects of the form: $a_{i,j} = a_{k,m}$. To check a clue, first, go through the formula and at each literal of this form, if $f_{i,k}(j) = m$, then replace the $a_{i,j} = a_{k,m}$ with $true$. Otherwise, replace it with $false$. This step can be done in linear time and will result in a simple boolean formula where all of the literals are either $true$ or $false$. This can easily be evaluated in polynomial time. If the formula evaluates to $true$, then the solution adheres to that clue. Each clue can be independently checked in this way.

Since it is possible to check any proposed solution against a puzzle, and do so in polynomial time, these Logic Puzzles are in NP. Intuitively, it is easy to see that this makes sense. When one is doing a puzzle, checking the solution is as easy as comparing the final list of objects with each clue, one

by one.

3.2 Logic Puzzles are NP-hard - a Reduction from Partial Latin Square Completion

The problem of determining, given a logic puzzle and its set of clues, if a valid solution exists, is NP-hard. To prove this, we perform a reduction from the Partial Latin Squares Completion problem, shown by Charles Colbourn to be NP-complete [1]. A Latin Square is a grid of numbers (n by n) in which each row and column contains exactly one instance of the numbers from 1 to n, inclusive. This is an example of a 4 by 4 Latin square:

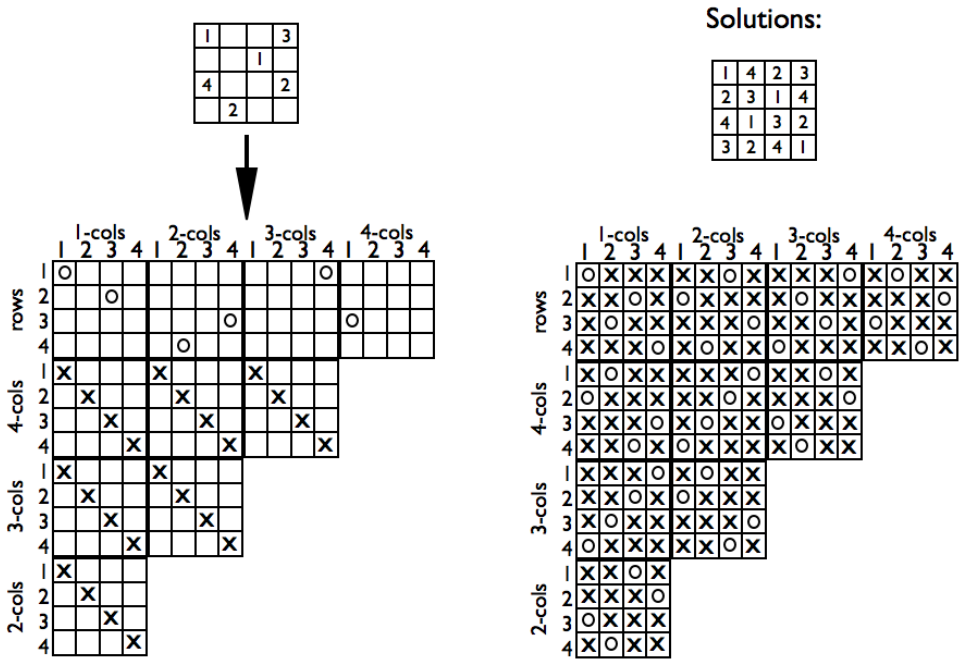
1	2	3	4
2	3	4	1
4	1	2	3
3	4	1	2

A Partial Latin Square is one in which only some of the numbers are filled in. Completion of such a Partial Latin Square entails finding the values for empty spaces that will produce a valid Latin Square. Theorem 3.5 from Colbourn's paper shows that the problem of determining if such a Partial Latin Square can be completed is NP-Complete. Therefore, if we can reduce an arbitrary Partial Latin Square Completion problem to a Logic Puzzle, then we have proved that Logic Puzzles are NP-Hard. Since these logic puzzles contain a set of domains, characteristics and clues, we should be precise in what we mean by, "the size of the input." Specifically, the larger of the number of domains or the number of objects in each domain is the dominating factor of complexity of solving it. As such, the proof below operates on proving NP-completness with respect to the number of domains and objects in the puzzle.

To perform the reduction, we have a Partial Latin Square of size n by n. This we will use to produce a logic puzzle with $n + 1$ domains and n objects. The first domain we will call "Row". All the other domains will be "1-Columns", "2-Columns", and so on, up to "n-columns". In this puzzle, each characteristic in each domain is a number from 1 to n. Then, we number the rows and columns of the Partial Latin Square, each from 1 to n. Note that if the Latin Square had n rows and columns, then our logic puzzle has only $n + 1$ domains and n objects. This certainly polynomial within the size of the Latin Square.

In the puzzle, if an object has a particular row characteristic r and a particular characteristic c from the domain "d-columns", this will correspond to the number d appearing in row r and column c in the Latin square. Since only one characteristic can be taken from each domain, each number can only appear once in each row and column, which satisfies one of the conditions to be a Latin Square.

The given entries in a partial Latin square must be mapped to clues in the puzzle. For each given number d, in row r and column c, this corresponds to the clue: row:r is d-columns:c. This means one clue per given number in the Partial Latin Square. We must also encode, as clues, the condition of the Latin Squares that only one number can occupy each place. To do this, for each number d, column c, and each other column e, we have the clue: d-column:c is not e-column:c. In other words, for a given row, placing a number in a column precludes placing different number in the same column. Since there are n columns and n numbers, there will be $(\frac{n^2-n}{2})(n)$ or $(\frac{n^3-n^2}{2})$ such clues. This is number of clues is certainly polynomial in the size of the original Latin Square. Using this method, we can reduce, in polynomial time, any instance of the Partial Latin Squares Completion problem into an instance of our Logic Puzzles. If the our puzzle can be solved, then our Latin Square can be completed. Since Latin Square Completion is known to be NP-Complete, and we already proved above that logic puzzles are in NP, these logic puzzles must be NP-Complete in the number of domains and objects. Below, we show an example of this reduction on a small Partial Latin Square Completion problem, with the corresponding solutions:



3.3 Complex Clues and NP-Completeness

The proof above for NP-Competeness relied only on the two simplest types of clues. However, it also applies to puzzles admitting any of the variety of clues possible under our scheme. This would seem intuitive, since, if the puzzle is intractable when the clues are simple, it must be more intractable when the clues are complex. To prove this more formally, we point out that, since we have proven that logic puzzles are NP-Complete in the number of domains and objects using simple clues, any puzzle with simple clues can be reduced to a puzzle where more complex clues are allowed. Simply put, if we had a polynomial time algorithm for solving puzzles with complex clues, we would also have a polynomial time algorithm for solving puzzles with just simple clues.

4 Automated Solvers for Logic Puzzles

4.1 A Previously Proposed Algorithm

An algorithm proposed by Mark Valentine and Robert H. Davis, published in Information Processing Letters in 1987 [6], seeks to solve a set of logic puzzles like the ones that we are investigating. This algorithm attempted to solve these puzzles by representing objects as tuples of potential values of characteristics for each domain. For example, given the ongoing example, one could have a tuple of {{Aaron, Bob, Clive}, Chester, {Actor, Barber, Carpenter}}. This would signify not knowing anything about Chester. If we determined that Mr. Chester was not Clive, this tuple would become: {{Aaron, Bob}, Chester, {Actor, Barber, Carpenter}}. A solution would involve resolving these tuples into tuples with only one value per domain.

In order to do this, the proposed algorithm ran by comparing tuples against each other pairwise. If it could be determined that two tuples must represent different objects, due to allowed values in one domain being disjoint between the two tuples, any values in one tuple that were single values in their domain could be removed from the other and vice-versa. Similarly, if the two tuples must be the same object, if they both have a domain in which only one characteristic is possible, and it is the same, anything not common to both tuples can be removed from each and the two tuples joined to make a single tuple.

However, since this algorithm only compares tuples in pairs, it is unable to capture any logic in which multiple tuples must be involved. For example, take the following three:
{Azaleas, [250, 275]}
{Boxwoods, [250, 275]}
{Hostas, [200, 250, 275]}

In this case, if you were simply to compare any pair of tuples, you could gain no information. However, looking at all three shows that Azaleas and Boxwoods must each take one of 250 and 275, meaning that Hostas must be 200. Valentine and Davis do not capture this kind of logic nor quite a lot of similar examples of deductions involving looking at more that two objects at once. Even if one were to somehow capture all of the logic that could potentially arise from any comparison of any number of tuples, within a group of n tuples, there are, by looking at the power set of these tuples, 2^n subsets of tuples that would have to be checked in this manner. Also, as

shown above, logic puzzles of this type are NP-complete, which means that it is extremely unlikely that the Valentine and Davis algorithm, as it runs in polynomial time, can be modified to be correct.

Finally, this algorithm only examined three potential types of clues. In practice, this only covered about a fifth of the puzzles that we found when looking through various examples of these logic puzzles. Therefore, another approach is needed for solving these puzzles automatically.

4.2 Logic Puzzles as Boolean Satisfiability Problems

A more successful approach to solving these puzzles can be attempted by taking advantage of the fact that these logic puzzles can be mapped to a boolean satisfiability problem. The reason why SAT-solving works as a good approach is that, in the SAT encoding of these puzzles, first, it is fairly easy to put the puzzle in Conjunctive Normal Form, and, second, when we do, there are two different ways of encoding the puzzle that closely, though not exactly, resemble forms of boolean satisfiability with efficient solvers. The first encoding produces a boolean formula in which most of the clauses have only two literals. Were the entire encoding to have only two literals per clause, we would be able to solve these puzzles in linear time [5]. Of course, not all of the clauses have only two literals, but, since most do, that suggests that various SAT solver heuristics will be reasonably successful in solving these puzzles when they are put in this form. Also, a second encoding of logic puzzles as a boolean formula results in a string of clauses that are mostly clauses with at most one positive literal, also called horn clauses. Again, if these were all horn clauses, then the puzzle would be easily solvable [2]. However, the fact that they are mostly horn clauses suggests that basic SAT solvers may have success in solving some logic puzzles in this encoding as well.

There are two main pieces that are involved in encoding logic puzzles as boolean satisfiability. First, the basic conditions of the puzzle must be encoded in some way. This part of the boolean formula will be similar from puzzle to puzzle, but it will vary in length depending on the number of domains and objects in the puzzle. The second part of encoding the puzzle is more tricky. This involves encoding the clues to the puzzle. We have already defined each clue as a boolean combination of facts about the puzzle solution, so this would seem to be amenable to a SAT encoding of the whole puzzle. However, in an actual puzzle, the clues are given in natural English.

This means that a considerable amount of thought must be used to parse this natural language and determine the exact logical content of any given clue.

While solving the general problem of the natural English wording of clues is a highly complex artificial intelligence problem that is beyond the scope of this work, the issue must be taken into account when attempting to create automatic solvers. The approach we use is to define boolean strings that correspond to eight of the most common and basic types of clues. These are: "A is not B", "A is B", "A is B implies C is D","A is not B implies C is D", "A is B implies C is not D", "A is less than B along domain C", "A is exactly n less than B along domain C" and "A is exactly n less than or greater than B along domain C". For example, if the English was, "John, who is not Mr. Smith, was the teacher," then we can input this into our solver as, "John is not Smith" and "John is teacher." In terms of the actual program input this was done with symbols: "John*Smith" and "John=teacher". Nearly all of the clues that were encountered could be expressed as one of the eight clues or as a logically equivalent list of these clues. Sometimes, fitting a fairly esoteric wording for a clue to these eight basic clues was fairly complex, however, this approach still worked well. This is because this set of possible inputs was close to the direct meaning of about ninety percent of statements in puzzle clues. Most of rest of the clues, with some thought, could be modeled as a list of clues from these input options. While the wording of clues often varied, nearly all of the clues could be captured by this group of input options. The input to the solver program was neither pure logic symbols, which would have made reading and entering actual puzzle inputs very painful, nor was it a natural language reader, which is beyond the scope of current technology.

Given the means of inputting puzzles into the solver, an algorithm generating a boolean formula that corresponds to it must, first, have the ability to generate a part of the boolean string that corresponds to the basic conditions of the puzzle, that each object corresponds to one and only one characteristic in each domain and that no two objects share characteristics. Second, we must have a way of generating a boolean encoding for each type of clue that we have chosen to include among those that we will allow inputed into the solver. Lastly, since most of the SAT solvers that we tested on these strings could be programmed to run fairly quickly and fairly simply on formulas in Conjunctive Normal Form, all the SAT encodings we generate are in Conjunctive Normal Form.

4.3 The First Boolean Satisfiability Encoding

To map these logic puzzles to boolean satisfiability, first, we define a boolean literal $N(A, B)$. In this mapping, we use the first domain of the puzzle as a basis. In this way, A is a characteristic from the first domain and B is a characteristic from any other domain. A *true* truth value for this literal would mean these two characteristics are the same object. A $false$ truth value for this literal means that these two characteristics do not represent the same object. For example, the statement "Aaron is the Barber," is the literal $N(Aaron, Barber)$. Any clues that include something from the first domain can be represented this way with either a single literal or its negation.

First, we encode the general conditions of the puzzle. From when we defined the puzzle, we have that the relationship of the two objects between any two domains is a bijection across the objects of those two domains. Since our literals only include the relationship between the first domain to one of the others, we don't need to worry about the second part of the definition, that the mapping between different domains be consistent. All of our mappings in this encoding are from the first domain.

In order to insure that a function is a bijection, we must, first, insure that every value in the range of the function corresponds to exactly one value in the domain of the function and, second, that every value in the domain of the function corresponds to exactly one value in the range of the function. In our case, this means that we must insure, in comparing the first domain to another domain, first, that each object in the first domain maps to exactly one object in the second domain and, second, that each object in the second domain is mapped to by exactly one object in the first.

For the first of these conditions, to say that each object in the first domain maps to exactly one characteristic in each of the other domains, we use a set of clauses such as $(\neg N(Aaron, Amherst) \vee \neg N(Aaron, Barth))$, or Aaron is not both Barth and Amherst. One clause of this type is included for each pair of characteristics, in each domain (except the first), for each object, or $2(\frac{n^2-n}{2})(m-1)(n) = (n^3 - n^2)(m-1)$ literals. We must also include one clause for each object, for each domain, to the effect that at least one characteristic for that domain applies. This is simply a lengthy disjunction such as: $(N(Aaron, Amherst) \vee N(Aaron, Barth) \vee N(Aaron, Clive))$ The length of these clauses is n literals and there are $n(m-1)$ such clauses.

Next, we must encode the condition that each characteristic in the do-

mains other than the first must correspond to exactly one characteristic in the first domain. The structure of these is similar to above. First, any object having a characteristic implies that all others do not. For example: $(\neg N(Aaron, Chester) \vee \neg N(Bob, Chester))$, that is, Chester is not both Aaron and Bob. One clause of this form must be included for each pair of objects, (characteristics from the first domain,) and each characteristic from all the other domains. This is again $2(\frac{n^2-n}{2})(m-1)(n) = (n^3-n^2)(m-1)$ literals. And, of course, each characteristic corresponds to at least one object from the first domain, is coded as $(N(Aaron, Amherst) \vee N(Bob, Amherst) \vee N(Chester, Amherst))$. Again, like above, the length of these clauses is n literals and there are $n(m-1)$ such clauses.

To summarize, where each literal, $N(a, b)$ where a is a characteristic from the first domain and b is a characteristic from another domain, represents that a and b correspond to the same object, the SAT encoding of the basic conditions of a logic puzzle is: (where obj is the number of objects and dom is the number of domains)

$\bigwedge_{q=2}^{dom} \bigwedge_{r=1}^{obj} \bigwedge_{i=1}^{obj} \bigwedge_{j=i+1}^{obj} (\neg N(a_{1,r}, a_{q,i}) \vee \neg N(a_{1,r}, a_{q,j}))$
$\wedge \bigwedge_{q=2}^{dom} \bigwedge_{r=1}^{obj} \bigvee_{i=1}^{obj} (N(a_{1,r}, a_{q,i}))$
$\wedge \bigwedge_{q=2}^{dom} \bigwedge_{r=1}^{obj} \bigwedge_{i=1}^{obj} \bigwedge_{j=i+1}^{obj} (\neg N(a_{1,i}, a_{q,r}) \vee \neg N(a_{1,j}, a_{q,r}))$
$\wedge \bigwedge_{q=2}^{dom} \bigwedge_{r=1}^{obj} \bigvee_{i=1}^{obj} (N(a_{1,i}, a_{q,r}))$

Next, we must encode the clues of the puzzle. For the simplest type of clues, statements that two characteristics either do, or do not, correspond to the same object, if one of the characteristics involved is from the first domain, then encoding it is just a single literal, corresponding to those two characteristics, set to $true$ if the clues says that the two characteristics correspond to the same object and $false$ if they do not.

Of course, some clues might relate two characteristics not from the first domain. In this case, we point out that, for each object, having the first characteristic implies that that object, either, depending on the clue, does or does not have the other characteristic. Therefore, in our ongoing example, the clue, "Barth is not the carpenter," can be thought of as, "If Aaron is Barth, Aaron is not the carpenter," and "if Bob is Barth, Bob is not the carpenter," and similarly for Chester. As such, in boolean variables, "Barth is not the carpenter," becomes:
$(N(Aaron, Barth) \rightarrow \neg N(Aaron, Carpenter)) \wedge$

$(N(Bob, Barth) \rightarrow \neg N(Bob, Carpenter)) \wedge$
$(N(Chester, Barth) \rightarrow \neg N(Chester, Carpenter))$
or:
$(\neg N(Aaron, Barth) \vee \neg N(Aaron, Carpenter)) \wedge$
$(\neg N(Bob, Barth) \vee \neg N(Bob, Carpenter)) \wedge$
$(\neg N(Chester, Barth) \vee \neg N(Chester, Carpenter))$

In any case, each clue corresponds to a boolean string either as a single literal or a set of clauses $2n$ literals in length.

In the general case, we can define a boolean function $Clue(val, a_1, a_2)$, where d_1 is the first domain, $a_2 \notin d_1$ and $val = true$ if the clue says that a_1 and a_2 correspond to the same object and $val = false$ otherwise, as:

$$\begin{aligned} Clue(val, a_1, a_2) = \; & N(a_1, a_2) \text{ if } a_1 \in d_1 \text{ and } val = true \\ & \neg N(a_1, a_2) \text{ if } a_1 \in d_1 \text{ and } val = false \\ & \textstyle\bigwedge_{i=1}^{obj}(\neg N(a_{1,i}, a_1) \vee N(a_{1,i}, a_2)) \text{ if } a_1 \notin d_1 \text{ and } val = true \\ & \textstyle\bigwedge_{i=1}^{obj}(\neg N(a_{1,i}, a_1) \vee \neg N(a_{1,i}, a_2)) \text{ if } a_1 \notin d_1 \text{ and } val = false \end{aligned}$$

Note that nearly all of the clauses produced, so far, in this way are disjunctions of 2 literals. This means that a significant portion of the boolean string, though not all of it, is in 2-Conjunctive Normal Form.

4.4 Complex Clues in the SAT Encoding

The next type of clue is the simple implication. For arbitrary characteristics, p,q,r and s, we need a way of saying: $(a_p = a_q) \rightarrow (a_r = a_s)$ or $(a_p = a_q) \rightarrow \neg(a_r = a_s)$ or $\neg(a_p = a_q) \rightarrow (a_r = a_s)$. The problem that we encounter is that our first SAT encoding only allows relating characteristics to some characteristic in the first domain. p,q,r and s might be in any domain. One way to solve this problem is to change the definition of the literals to a different SAT encoding, one that allows for the relationship between any two characteristics to be captured as a literal. This possibility will be explored in the next section.

If, however, we preserve the current encoding, we can note that, since we are relating p and q, these can't both be in the first domain. Therefore, for simplicity, we say that only p might be in the first domain. We also do the same for r. Therefore, we have, given d_1 is the first domain, obj is the number of objects, dom is the number of domains, we define:

$Imply(a_p, a_q, a_r, a_s) = (\neg N(a_p, a_q) \vee N(a_r, a_s))$ if $(a_p \in d_1) \wedge (a_r \in d_1)$
$\bigwedge_{i=0}^{obj}(\neg N(a_p, a_q) \vee \neg N(a_{1,i}, a_r) \vee N(a_{1,i}, a_s))$ if $(a_p \in d_1) \wedge (a_r \notin d_1)$
$\bigwedge_{i=0}^{obj}(\neg N(a_{1,i}, a_p) \vee \neg N(a_{1,i}, a_q) \vee N(a_r, a_s))$ if $(a_p \notin d_1) \wedge (a_r \in d_1)$
$\bigwedge_{i=0}^{obj} \bigwedge_{j=0}^{obj}(\neg N(a_{1,i}, a_p) \vee \neg N(a_{1,i}, a_q) \vee \neg N(a_{1,j}, a_r) \vee N(a_{1,j}, a_s))$
if $(a_p \notin d_1) \wedge (a_r \notin d_1)$

Also, we can define $ImplyTF(a_p, a_q, a_r, a_s)$ to mean, $(a_p = a_q) \rightarrow \neg(a_r = a_s)$, and $ImplyFT(a_p, a_q, a_r, a_s)$ to mean, $\neg(a_p = a_q) \rightarrow (a_r = a_s)$ as follows:

$ImplyTF(a_p, a_q, a_r, a_s) = (\neg N(a_p, a_q) \vee \neg N(a_r, a_s))$ if $(a_p \in d_1) \wedge (a_r \in d_1)$
$\bigwedge_{i=0}^{obj}(\neg N(a_p, a_q) \vee \neg N(a_{1,i}, a_r) \vee \neg N(a_{1,i}, a_s))$ if $(a_p \in d_1) \wedge (a_r \notin d_1)$
$\bigwedge_{i=0}^{obj}(\neg N(a_{1,i}, a_p) \vee \neg N(a_{1,i}, a_q) \vee \neg N(a_r, a_s))$ if $(a_p \notin d_1) \wedge (a_r \in d_1)$
$\bigwedge_{i=0}^{obj} \bigwedge_{j=0}^{obj}(\neg N(a_{1,i}, a_p) \vee \neg N(a_{1,i}, a_q) \vee \neg N(a_{1,j}, a_r) \vee \neg N(a_{1,j}, a_s))$
if $(a_p \notin d_1) \wedge (a_r \notin d_1)$

$ImplyFT(a_p, a_q, a_r, a_s) = (N(a_p, a_q) \vee N(a_r, a_s))$ if $(a_p \in d_1) \wedge (a_r \in d_1)$
$\bigwedge_{i=0}^{obj}(N(a_p, a_q) \vee \neg N(a_{1,i}, a_r) \vee N(a_{1,i}, a_s))$ if $(a_p \in d_1) \wedge (a_r \notin d_1)$
$\bigwedge_{i=0}^{obj}(\neg N(a_{1,i}, a_p) \vee N(a_{1,i}, a_q) \vee N(a_r, a_s))$ if $(a_p \notin d_1) \wedge (a_r \in d_1)$
$\bigwedge_{i=0}^{obj} \bigwedge_{j=0}^{obj}(\neg N(a_{1,i}, a_p) \vee N(a_{1,i}, a_q) \vee \neg N(a_{1,j}, a_r) \vee N(a_{1,j}, a_s))$
if $(a_p \notin d_1) \wedge (a_r \notin d_1)$

Using the functions that we have defined, $Imply$, $ImplyTF$, $ImplyFT$ and $Clue$, we can generate CNF formulas for the various types of clues discussed when we defined the logic puzzle. $Order$ refers to a clue that one object is less than another along an ordered domain. Gap refers to a clue which says one object is exactly some gap less than another along an ordered domain. $Gap2$ refers to a clue which says one object is exactly exactly some gap less than or exactly some gap greater than another along an ordered domain.

$Order(d, a_1, a_2) = Clue(a_1, a_2, false) \wedge \bigwedge_{q=1}^{obj} \bigwedge_{r=q+1}^{obj} ImplyTF(a_2, a_{d,q}, a_1, a_{d,r}))$
In other words, the two objects are not the same and for any possible placement of the second (the greater) object, along the ordered domain, the first object cannot be greater than that.

$Gap(d, a_1, a_2, gap) = Clue(a_1, a_2, false) \wedge \bigwedge_{q=1}^{obj-gap}(Imply(a_1, a_{d,q}a_2, a_{d,q+gap})) \wedge$

$\bigwedge_{q=obj-gap+1}^{obj}(Clue(a_1, a_{d,q}, false))$
Again, the two objects are not the same. Also, for any placement of the lesser object, the greater object must be *gap* places higher. And, the first object cannot be assigned any place in the ordered domain such that there is no space *gap* places higher.

$Gap2(d, a_1, a_2, gap) = Clue(a_1, a_2, false) \wedge$
$\bigwedge_{p=1}^{obj} \bigwedge_{q\neq p \wedge q\neq p-gap \wedge q\neq p+gap}(ImplyTF(a_1, a_{d,p}, a_2, a_{d,q}))$
Again, the two objects are not the same. For every possible placement of the first object, placing the object in that spot in the ordered domain disallows every placement of the second object that is not *gap* spaces lower or higher.

This encoding, while producing a simple encoding for the basic puzzle itself, can potentially produce highly complex clues. The goal of the second encoding is to simplify the clues.

4.5 A Second, Extended SAT Encoding

A second SAT encoding can be created by defining literals $M(a_p, a_q)$ to compare two characteristics in any two domains. This, however, has the disadvantage that the boolean string must now include clauses to insure that mappings from different domains are consistent with each other. For example, if $M(a_{2,p}, a_{3,q})$ and $M(a_{3,q}, a_{4,r})$, then $M(a_{2,p}, a_{4,r})$ must also be true. This fact must be encoded in the general formula for the puzzle.

Formally, where each literal, $M(a, b)$ represents that a and b correspond to the same object, the SAT encoding of the basic conditions of a logic puzzle begins: (where *obj* is the number of objects and *dom* is the number of domains)

$\bigwedge_{p=1}^{dom} \bigwedge_{q=p+1}^{dom} \bigwedge_{r=1}^{obj} \bigwedge_{i=1}^{obj} \bigwedge_{j=i+1}^{obj}(\neg N(a_{p,r}, a_{q,i}) \vee \neg N(a_{p,r}, a_{q,j}))$
$\wedge \bigwedge_{p=1}^{dom} \bigwedge_{q=p+1}^{dom} \bigwedge_{r=1}^{obj} \bigvee_{i=1}^{obj}(M(a_{p,r}, a_{q,i}))$
$\wedge \bigwedge_{p=1}^{dom} \bigwedge_{q=p+1}^{dom} \bigwedge_{r=1}^{obj} \bigwedge_{i=1}^{obj} \bigwedge_{j=i+1}^{obj}(\neg M(a_{p,i}, a_{q,r}) \vee \neg M(a_{p,j}, a_{q,r}))$
$\wedge \bigwedge_{p=1}^{dom} \bigwedge_{q=p+1}^{dom} \bigwedge_{r=1}^{obj} \bigvee_{i=1}^{obj}(M(a_{p,i}, a_{q,r}))$
This above part of the encoding is essentially the same as in the small encoding. However, it is extended to include all possible parings of domains, since we now have literals which represent connections between characteristics of all domains. We also add:

$\bigwedge_{p=1}^{dom} \bigwedge_{q=p+1}^{dom} \bigwedge_{r=1}^{obj} \bigwedge_{i=1}^{obj} (\neg M(a_{p,r}, a_{q,i}) \vee M(a_{q,i}, a_{p,r}))$
This above line represents the reflexive property that was given when we defined the logic puzzle. In other words, to say that John is Smith, that implies Smith is John.

$\bigwedge_{p=1}^{dom} \bigwedge_{i=1}^{obj} \bigwedge_{q=1}^{dom} \bigwedge_{j=1}^{obj} \bigwedge_{r=1}^{dom} \bigwedge_{k=1}^{obj} (\neg M(a_{p,i}, a_{q,j}) \vee \neg M(a_{q,j}, a_{r,k}) \vee M(a_{p,i}, a_{r,k}))$
Finally, this line represents the transitive property we stated when we first defined the puzzle. In other words, if Joe is Jones, and Jones is the convict, then Joe is the convict.

It should be noted that, in this encoding, in all but the second and fourth lines, all of the clauses are horn clauses. That is, nearly all of the clauses contain, at most, one positive literal.

4.6 Clues in the Extended Encoding

In the extended encoding, we can define the clues using the same definitions for *Order*, *Gap* and *Gap*2 as above. However, we will substitute alternate, (and simpler), definitions for *Imply*, *ImplyTF*, *ImplyFT* and *Clue*, using the extended encoding:

$$Clue(val, a_1, a_2) = M(a_1, a_2) \text{ if } (val = true)$$
$$\neg M(a_1, a_2) \text{ if } (val = false)$$

$Imply(a_p, a_q, a_r, a_s) = (\neg M(a_p, a_q) \vee M(a_r, a_s))$
$ImplyTF(a_p, a_q, a_r, a_s) = (\neg M(a_p, a_q) \vee \neg M(a_r, a_s))$
$ImplyFT(a_p, a_q, a_r, a_s) = (M(a_p, a_q) \vee M(a_r, a_s))$
Substituting these CNF formulas for the extended encoding into the definitions of *Order*, *Gap* and *Gap*2 will give CNF formulas for the extended encoding of these clues. As can be seen by the simpler form of these equations, the extended encoding has more simple encodings for clues, but pays for it with larger and more complex encodings for the basic puzzle.

5 Boolean Satisfiability Solvers

After producing the boolean formula encoding for these puzzles, various SAT solving techniques described below were used on a sample of puzzles, to

determine their relative performance. (See appendices for the source code and instructions on use.) The first three of these approaches are modeled after approaches proposed in a paper that used SAT techniques for solving Sudoku puzzles [4]. These SAT heuristics are among some of the simpler approaches for finding satisfying assignments for boolean formulas. The reason why these were chosen was that the puzzle encodings suggested, by their close proximity to easily solved forms of boolean satisfiability, (2-SAT and horn clauses), that simple approaches might work.

- Unit Propagation

  Usually, as a result of some of the clues, there are clauses which contain only one literal. If this is the case, then these variables can be assigned as they appear. Using these assignments, other clauses in the formula that are now satisfied can be removed. Also, other clauses that contain the negation of the assignment of any assigned variable can have just that literal removed from the clause. This might, in turn, reveal further single variable clauses, which can be assigned as well. Sometimes, this is sufficient to solve some simple puzzles. However, this procedure is most useful in that it can be used to simplify the boolean formula in between the steps of the other solvers. An example of this process is given below:
  $(\neg A) \wedge (A \vee B \vee C) \wedge (\neg A \vee \neg B) \wedge (A \vee \neg B)$
  Since $(\neg A)$ is singular, then A must be assigned $false$. With this assignment, the third clause is satisfied and removed. Also, the first literal in the second clause, as well as the first literal in the fourth clause are not true and are also removed. This gives:
  $(B \vee C) \wedge (\neg B)$
  Now, since $(\neg B)$ is singular, this variable can be assigned, $(false)$, and the processes is repeated to get:
  (C)
  This leaves C which is assigned $true$. Therefore, a satisfying assignment to the original formula is $A = false$, $B = false$ and $C = true$.

- Failed Literal Rule

  In this solver, each variable is tested for a $true$ assignment and a $false$ assignment. If any assignment, after unit propagation on the remaining string, shows that such an assignment is impossible, then the opposite assignment is taken and kept in the solution. Variables are tried in this

manner until no more information can be found. For example:
$(\neg A \vee \neg C) \wedge (A \vee B) \wedge (A \vee \neg B) \wedge (C \vee \neg B)$
In this case, nothing can be found by unit propagation since there are no singular variables. So, we try assigning A as $false$:
$(\neg A) \wedge (\neg A \vee \neg C) \wedge (A \vee B) \wedge (A \vee \neg B) \wedge (C \vee \neg B)$
$(B) \wedge (\neg B) \wedge (C \vee \neg B)$
This is a contradiction. Therefore, A must be assigned $true$. Applying this to the original formula gives:
$(\neg C) \wedge (C \vee \neg B)$
With simple unit propagation, from that, we get C assigned $false$ and B assigned $false$.

- Binary Failed Literal Rule

  In this solver, each pair of variables is tested for each possible assignment as well as each single variable. If any assignment of a pair of variables is shown to be impossible, then an extra clause is added to the string which is the disjunction of the opposite assignments of the two variables. For example, if A and B, both assigned true, is shown to be impossible, then $(\neg A \vee \neg B)$ is added to the string. All variables and pairs of variables are tried until no more information can be found. Because of the number of pairs of variables in some of the strings produced by these logic puzzles, this solver has a tendency to take a long time before giving up if it cannot find a solution.

- Simple Backtracking

  This solver is a simple backtracking approach. At each level of the backtracker, a variable is selected. Both assignments of that variable are tried against two recursive calls of the backtracker. Of course, those recursive calls select a different variable and try both assignments of that variable. The base case of the recursion is when the backtracker encounters a satisfying solution or an impossible case. For example, in the following formula:
  $(A \vee C) \wedge (\neg A \vee \neg C \vee \neg B) \wedge (\neg A \vee \neg B \vee C) \wedge (C \vee B)$
  The first step of the backtracker is to chose a variable, which, for the simple backtracker is the first found in the string, A, and assign it the value corresponding to that first literal $true$. This result is fed into a recursive call of the same backtracker:
  $(\neg C \vee \neg B) \wedge (\neg B \vee C) \wedge (C \vee B)$

Next, the backtracker runs again, assigning the first variable: C as $false$. This assignment gives:
$(\neg B) \wedge (B)$
This is a contraction, therefore, the algorithm returns (backtracks) to the previous string:
$(\neg C \vee \neg B) \wedge (\neg B \vee C) \wedge (C \vee B)$
Now, the other possible assignment C as $true$ is tried:
$(\neg B)$ Unit propagation will give B as $false$ at this point. This completes the satisfying assignment. In order to speed up the algorithm, unit propagation is tried at each level of the search. This works well for our logic puzzles since, in a 2-SAT clause, assigning a single variable in such a clause leaves a single variable. Therefore, when there are many such clauses, as in the case of our encoded puzzles, assigning a single variable has the potential to yield a high number of variables implied directly from it.

- Backtracking to remove non-horn clauses

  The structure of this algorithm is much like the backtracker above. However, at each level, the backtracker searches for the first case of a literal that is not part of a horn clause. If it finds none, then it knows that the formula contains only horn clauses and will run the the efficient solver for horn clauses. This approach is designed to reduce the depth of search for the backtracker. It does this in two ways, first, when a formula of horn clauses is found, then that equation is solved immediately, decreasing the depth that the search is required to go. Second, by seeking out the first non-horn clause as the variable to assign, and by that reduce or remove the horn clause, the chance of getting to the point in which there are only horn clauses increases.

- Backtracking to remove non-2SAT clauses

  This is like the backtracker above. However, it searches for non-2-SAT clauses instead of horn clauses. Also, at the base case the solver, if the formula is 2-SAT, uses a basic 2-SAT solver. This basic 2-SAT solver works on the same principle as Tarjan's solver, namely that, if all the clauses are 2-SAT, then all the relationships between variables can be captured as direct implications [5]. However, for the sake of reducing program code complexity and avoiding the overhead of producing

graph nodes, the solver does not produce a graph of these implications. Rather, it reuses the code from unit propagation, which, when the clauses are only 2-SAT, is logically equivalent to finding everything implied by assigning a variable. Tarjan's algorithm operates on the idea that, so long as a variable assignment does not imply either its opposite assignment or a contradictory pair of assignments, then that variable assignment can be used in a satisfying assignment. So, the basic SAT solver simply chooses a variable to assign a value. If the unit propagation yields no contradiction, then that assignment is kept. If there is a contradiction, the opposing assignment is kept for that variable. Of course, if both assignments are contradictory, then that formula has no satisfying assignment. While not in linear time, like Tarjan's approach, this approach still is a small polynomial and, in practice, runs quickly.

- Backtracking to remove non-horn clauses, search for best

  This backtracker is like the other "horn clause" backtracker above. However, it searches the entire string looking for the variable that is found in the most non-horn clauses and uses that. It does this by counting, for each variable, all instances of that variable for non-horn clauses. The variable that has the highest number of such instances is chosen. This approach designed to increase the chance that assigning the selected variables will lead all to horn clauses by choosing variables that are most likely to remove non-horn clauses.

- Backtracking to remove non-2-SAT clauses, search for best

  This is similar to above, except that all non-2-SAT clauses are checked.

The first three solvers always terminate in polynomial time. However, they might not yield a solution. The last five solvers do not terminate in polynomial time. However, they always yield a solution.

6 Results

In order to test our approaches to solving these puzzles, we ran each of the solvers on 50 puzzles of various difficulty from three sources. First, we used the simple puzzle that was given as an example in Valentine and Davis paper [6]. Second, we ran these solvers on the fairly well known "Zebra

Puzzle," also, erroneously, called Einstein's puzzle [8]. Third, the remaining 48 examples were taken from the 2008 January edition of Dell Magazine's "Logic Puzzles." [7]

6.1 Puzzle Difficulty

A good way of assessing the difficulty of the various puzzles that were solved can be found in comparing the lengths of the boolean formulas generated by each puzzle. On average, the small encoding generated 1081 clauses and the large encoding generated 19335 clauses. Also, often, these boolean formulas could be greatly simplified by the Unit Propagation Method described above. This, on average, removed about 58% of the clauses in the formula.

6.2 Capability to Solve

The first four methods that we used to solve these puzzles always terminated in polynomial time, however, they did not always produce a solution. In the table below, we can see, out of 50, the number of puzzles each of these puzzle solvers actually solved:

Encoding	Val&Davis	Unit Prop.	Failed Lit.	Bin. Failed Lit.
Small	11	4	36	44
Extended	XX	14	45	45

As can be seen from the table, resorting to the large encoding increased the strength of these puzzle solvers. This makes sense. The increased number of clauses and variables in the large encoding meant that the logical implications of any variable assignment had more ways of being determined by more basic operations on the boolean formula.

6.3 Time to solve

However, the increased power of the solvers using the large encoding also included a cost of greatly increasing the average time it took to solve these puzzles.

Encoding	Val&Davis	Unit Prop.	Failed Lit.	Bin. Failed Lit.
Small	0.2638ms	0.1473ms	5.9595ms	212.6779ms
Extended	0.2638ms	13.2661ms	1212.7794ms	1213.1963ms

Also, it should be noted that the use of the solvers that searched the boolean formula in greater detail both increased the time and increased the chance of solving.

The last five solvers were variations on backtracking. The average time spent by each of these solvers is given below:

Encoding	Backtrack	First non-horn	First non-2SAT	Best non-horn	Best non-2SAT
Small	8.2986ms	9.2070ms	5.8291ms	124.5304ms	34.1519ms
Extended	493.5578ms	87.0748ms	1689.5214ms	150.0675ms	1339.4121ms

From this table, it can be seen that there was a slight difference in the run time of each of these solvers, depending on the encoding. The small encoding ran slightly, though not significantly, faster when targeting non-2-SAT clauses. Similarly, the large encoding was solved faster with the solvers that targeted non-horn clauses. This, intuitively, makes sense, since most of the clauses in the small encoding were 2-SAT clauses and most of the clauses in the large encoding were horn clauses.

6.4 Search depth comparisons

Finally, scanning the entire formula for the "best" non-horn or non-2-SAT clause seemed not to be very helpful. From the table below, showing average of the highest number of literals that each backtracker needed to guess at any given time, it can be seen that searching the whole string for variables to guess assignments did not significantly affect the depth of the backtracker's search.

Encoding	Simple	First non-horn	First non-2SAT	Best non-horn	Best non-2-SAT
Small	5.57	3.39	6.35	5.53	8.76
Big	7.98	8.59	5.61	11.59	7.56

7 Further Work

7.1 Reading Puzzles for Clues

We have already mentioned the difficulty of reading the natural language of these puzzles and determining how to encode the clues into their equivalent

logical forms. Writing a program that would encode a larger variety of clues would make it easier to input a larger number of puzzles to test these solvers. Also, currently, puzzles are entered as individual text files. Having a user interface in which to enter these puzzles in a more intuitive way would also ease entering large numbers of puzzles.

7.2 Generating Hard Puzzles

In addition to solving logic puzzles, it is of interest to try and figure out how to generate instances of these puzzles automatically. This task, however, runs into problems that were not encountered in merely solving the puzzle. The most cumbersome problem was the fact that, given a set of clues, one had to insure that there was only one solution to the puzzle composed of those clues. Part of the reason that these basic solvers, especially the backtrackers, tend to work well for these puzzles is that there is only one solution for each of these puzzles. For this reason, randomly assigning variables tends to quickly create contradictions from which conclusions could be drawn. However, when repeatedly checking sets of clues that do not have unique solutions, this requires a much deeper search and takes a much longer time. This is even worse when the clues are deliberately meant to form a difficult puzzle. In that case, the puzzle solvers are supposed to take a long time and we are running them repeatedly to gauge how poorly the automatic solvers do. Therefore, in order to generate puzzles, an approach must be found to quickly determine if the solution to a given set of clues is unique.

Bibliography

[1] Colbourn, Charles. "The Complexity of Completing Partial Latin Squares." Discrete Applied Mathematics. 8 (1984): 25-30.

[2] Dasgupta, Sanjoy, Christos Papadimitriou and Umesh Vazirani. Algorithms. New York: McGrawHill, 2008. 144-145.

[3] Garey, Michael R. and David S. Johnson. Computers and Intractability. New York: W.H. Freeman and Company, 1979.

[4] Lynce, Ines and Joel Ouaknine. "Sudoku as a SAT Problem." Proceedings of the 9th Symposium on Artificial Intelligence and Mathematics. (2006).

[5] Aspvall, Bengt, Michael F. Plass and Robert Endre Tarjan. "A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas." Information Processing Letters. 8.3 (1987): 121-123.

[6] Valentine, Mark and Robert H. Davis. "The Automated Solution of Logic Puzzles." Information Processing Letters. 24 (1987): 317-324.

Sample Puzzle Sources:

[7] "Dell Logic Puzzles" Dell Magazines. Canada: January 2008.

[8] "Zebra Puzzle." Wikipedia, The Free Encyclopedia. 18 Apr 2009, 18:59 UTC. 22 Apr 2009 <http://en.wikipedia.org/w/index.php?title=Zebra_Puzzle&oldid=284661589>.

Appendix - Instructions for using automated solver

This code solves puzzles by reading specially formatted puzzle files, as described below. The automated solver runs in two modes. The first is if one runs "LogicPuzzles run all", this will cause the program to seek a set of files, all named puzzleN.txt, where N is some number, and solve all such puzzles it can find, starting with N = 1 and incrementing N until it can find the corresponding file. On each file, the program will use all the solvers. Then, statistics relating to the solving will be outputted.

More usefully, one can input a file name as an argument. This file must be a text file of the proper puzzle format. If it is, the solver will attempt to solve that puzzle using the algorithm number specified in the file. If it succeeds, it will output the solution. If it determines that there is no solution, it will say so.

The puzzle files must obey the following grammar, where EOL is end of a line and $FILE$ is the start symbol:

$FILE \rightarrow ALGORITHM\_NUMBER\ EOL$
$DOMAINLINE\ .\ .\ .\ DOMAINLINE$
'Clues:' EOL
$CLUELINES$

$ALGORITHM\_NUMBER \rightarrow 1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17$

$DOMAINLINE \rightarrow DOM{:}CHR{:}CHR{:}\ .\ .\ .\ {:}CHR\ EOL$

$CLUELINES \rightarrow ACLUE\ EOL\ CLUELINES \mid ACLUE\ EOL$

$ACLUE \rightarrow NOT\_CLUE \mid$
$CHR = CHR \mid$
$DOM\$CHR{:}CHR \mid$
$DOM\$CHR{:}CHR{:}NUM \mid$
$CHR{>}CHR{>}CHR{>}CHR \mid$
$CHR|CHR|CHR|CHR \mid$
$CHR\}CHR\}CHR\}CHR \mid$
$DOM\%CHR{:}CHR{:}NUM$

$NOT\_CLUE \rightarrow CHR * CHR \mid CHR * NOT\_CLUE$

DOM corresponds to some string giving the name of a domain.
CHR corresponds to some string giving the name of a charateristic.
In $FILE$ the number of $DOMAINLINE$ is exactly the number of domains.
In $DOMAINLINE$ the number of CHR is exactly the number of objects.

The first part of the file, above 'Clues:', is used to list all of the domains and characteristics in the puzzle. The second part is used to enumerate all of the clues in the puzzle. For each $ACLUE$, the options are, in the order given in the grammar:

- Any characteristic listed is not the same object as any other characteristic listed.
- $CHR1$ is the same object as $CHR2$
- Ordered by DOM, $CHR1$ is less than $CHR2$.
  (The order is the order the CHR are given in the $DOMAINLINE$)
- Ordered by DOM, $CHR1$ is exactly NUM less than $CHR2$.
- If $CHR1$ is the same object as $CHR2$, then $CHR3$ is $CHR4$
- If $CHR1$ is the same object as $CHR2$, then $CHR3$ is not $CHR4$
- If $CHR1$ is not the same object as $CHR2$, then $CHR3$ is $CHR4$
- Ordered by DOM, $CHR1$ is either exactly NUM less than or NUM more than $CHR2$.

Lastly, the number in the first line of the file corresponds to the desired algorithm. These are:

1: Valentine, Mark and Robert H. Davis Algorithm
2: Small encoding, Unit propagation only.
3: Large encoding, Unit propagation only.
4: Small encoding, Failed literal rule.
5: Large encoding, Failed literal rule.

6: Small encoding, Binary failed literal rule.
7: Large encoding, Binary failed literal rule.
8: Small encoding, Simple backtracking
9: Large encoding, Simple backtracking
10: Small encoding, backtracking to remove non-horn clauses
11: Large encoding, backtracking to remove non-horn clauses
12: Small encoding, backtracking to remove non-2SAT clauses
13: Large encoding, backtracking to remove non-2SAT clauses
14: Small encoding, backtracking to remove non-horn clauses, search for best
15: Large encoding, backtracking to remove non-horn clauses, search for best
16: Small encoding, backtracking to remove non-2SAT clauses, search for best
17: Large encoding, backtracking to remove non-2SAT clauses, search for best

Appendix - Sample puzzle input file

For the simple puzzle given in the first section, solved with the simple backtracker, the puzzle file would be:

```
8
Name:Smith:Jones:Butler
Shirt:red:green:blue
Job:doctor:teacher:actor
Clues:
Smith=teacher
actor=blue
doctor*red
Jones*green
```

After running: LogicPuzzles puzzleFile.txt, the result is:

```
method: Simple backtracking encoding: Small time: 0.032000ms
Smith, red, teacher
Jones, blue, actor
Butler, green, doctor
```

Appendix - Notes on Code Implementation

- While the first drafts of the implementation of these solvers was in java, the final programming language used was C. Of course, if one

represents CNF-clauses and literals as objects, java code lends itself well to implementing these algorithms. However, with upwards of 50,000 literals used in the larger encodings, this means many objects are being created and garbage collected by the Java Virtual Machine. Porting the code to C resulted in a speedup by a factor of 1000 by removing that overhead.

- The C code is specifically designed to avoid excessive memory allocation and freeing, in order to reduce memory costs and avoid memory leaks. Because of this, CNF-boolean strings are stored as a pair of arrays, one of integers, and the other of bools, and not as a linked data structure. This can be done since an analysis of the puzzle's clues and number of domains and objects is able to determine the exact size of the string. This requires only that malloc is called once per string of clauses. Each variable is represented as a number. The end of a CNF clause is signaled by the number -1. For each variable, the bool value at the same spot in the array of bools is its value in the string. For example, if the arrays are: 5,6,8,-1,3,-1 and true,true,false, true, false, true, this means $(5 \wedge 6 \wedge \neg 8) \vee (\neg 3)$. (bool values at the same index as the -1s are ignored.)

Appendix - Code Listing

The following is a listing of the C code used for solving these puzzles:

logicPuzzles.c

```
#include "logicIncludes.h"
#include "fileRead.h"
#include "clauseMaker.h"
#include "clauseOutputter.h"
#include "timer.h"
#include "solvers.h"
#include "dataCollect.h"
#include "ValDavis.h"

int main(int argc, char *argv[])
{
  if(argc == 3)
  {
    char * word1 = argv[1];
    char * word2 = argv[2];
    if((word1[0] == 'r') && (word1[1] == 'u') && (word1[2] == 'n') &&
       (word2[0] == 'a') && (word2[1] == 'l') && (word2[2] == 'l'))
    {
      runDataCollect();
      return EXIT_SUCCESS;
    }
    printf("Usage: LogicPuzzles <filename> \n       LogicPuzzles run all\n");
    return EXIT_SUCCESS;
  }
  if( argc != 2 )
  {
    printf("Usage: LogicPuzzles <filename> \n       LogicPuzzles run all\n");
    return EXIT_SUCCESS;
  }

  char * filename = argv[1];
  FILE *inputFile = fopen(filename, "r");

  if (inputFile == NULL)
  {
    printf("Unable to read file.\n");
    return EXIT_SUCCESS;
  }
```

```
  Puzzle thePuzzle = getPuzzle(inputFile);
  fclose(inputFile);

  if (thePuzzle.algorithm == 1)
  {
    timerSet();
    ValSolution sol = runValDavis(thePuzzle);
    outputSolutionVal(thePuzzle, sol, timerGet());
    freeVal(sol);
    return EXIT_SUCCESS;
  }
  else
  {
    thePuzzle.bigEncode = false;
    if (thePuzzle.algorithm % 2 == 1)
    {
      thePuzzle.bigEncode = true;
      thePuzzle.algorithm--;
    }
    thePuzzle.algorithm /= 2;
  }

  ClauseList clauses = getClauses(thePuzzle);
  timerSet();

  //outputClauses(thePuzzle,clauses);
  Solution solved = solveSAT(clauses, thePuzzle.algorithm);

  //debug text below:
  //outputSolutionVars(thePuzzle, solved);
  //outputCutClauses(thePuzzle,clauses, solved);

  outputSolution(thePuzzle, solved, timerGet());

  freePuzzle(thePuzzle);
  clauseFree(clauses);
  freeSolution(solved);
  return EXIT_SUCCESS;
}
```

backtracker.h

```
#ifndef BACKTRACKER_H_
#define BACKTRACKER_H_

#include "logicIncludes.h"

Solution backtrack(ClauseList c, int method);
void recursedBacktrack (ClauseList c, Solution * ret, int nowDepth);
int depthStat();

#endif /* BACKTRACKER_H_ */
```

backtracker.c

```
#include "solvers.h"
#include "backtracker.h"
#include "specialSAT.h"
#include "clauseMaker.h"
#include "timer.h"

int searchType;
int searchDepth;

Solution backtrack(ClauseList c, int method)
{
  Solution ret = solSetup(c);

  runUnitCheck(c, &ret);

  if (ret.impossible)
  {
    return ret;
  }
  if (ret.solved)
  {
    return ret;
  }

  ClauseList nc = condence(c,&ret);

  searchType = method;
  searchDepth = 0;
  recursedBacktrack(nc, &ret, 0);

  return ret;
}

void recursedBacktrack (ClauseList c, Solution * ret, int nowDepth)
{
  if (timerGet() > (float)120000.0){ return;}//STOP if taking more than 2 minutes
  int itr = 0;
  int toUse = -1;
  int best = 0;
  int myDepth = nowDepth + 1;
  if (myDepth > searchDepth) searchDepth = myDepth;
  int * references;
```

```
switch (searchType)
{
case 0:
  itr += ret->skipRecord[itr];
  while ((toUse == -1) && (itr < c.numSpots))
  {
    if (c.clauseNumbers[itr] != -1)
    {
      if (ret->assignments[c.clauseNumbers[itr]] == 0)
      {
        toUse = c.clauseNumbers[itr];
      }
    }
    itr++;
    itr += ret->skipRecord[itr];
  }
  break;
case 1: // find not horn clause
  itr += ret->skipRecord[itr];
  int numPos = 0;
  int lastPosVar = -1;
  while ((toUse == -1) && (itr < c.numSpots))
  {
    if (c.clauseNumbers[itr] == -1)
    {
      if (numPos > 1)
      {
        toUse = lastPosVar;
      }
      numPos = 0;
    }
    else
    {
      if (ret->assignments[c.clauseNumbers[itr]] == 0)
      {
        if (c.clauseVals[itr] == 1)
        {
          numPos++;
          lastPosVar = c.clauseNumbers[itr];
        }
      }
    }
    itr++;
    itr += ret->skipRecord[itr];
  }
```

```
  break;
case 2: // find not 2SAT clause
  itr += ret->skipRecord[itr];
  int numClauses = 0;
  while ((toUse == -1) && (itr < c.numSpots))
  {
    if (c.clauseNumbers[itr] == -1)
    {
      if (numClauses > 2)
      {
        toUse = lastPosVar;
      }
      numClauses = 0;
    }
    else
    {
      if (ret->assignments[c.clauseNumbers[itr]] == 0)
      {
        numClauses++;
        lastPosVar = c.clauseNumbers[itr];
      }
    }
    itr++;
    itr += ret->skipRecord[itr];
  }
  break;
case 3: // find most horn clause literal
  references = safeAlloc(c.numVars * sizeof(int));
  for (itr = 0; itr< c.numVars; itr++)
  {
    references[itr] = 0;
  }

  itr = 0;

  itr += ret->skipRecord[itr];
  numPos = 0;
  lastPosVar = -1;
  while (itr < c.numSpots)
  {
    if (c.clauseNumbers[itr] == -1)
    {
      if (numPos > 1)
      {
        int backItr = itr - 1;
```

```
        while ((backItr > -1) && (c.clauseNumbers[backItr] != -1))
        {
          if ((ret->assignments[c.clauseNumbers[backItr]] == 0))
          {
            references[c.clauseNumbers[backItr]] += 1;
          }
          backItr--;
        }
      }
      numPos = 0;
    }
    else
    {
      if (ret->assignments[c.clauseNumbers[itr]] == 0)
      {
        if (c.clauseVals[itr] == 1)
        {
          numPos++;
          lastPosVar = c.clauseNumbers[itr];
        }
      }
    }
    itr++;
    itr += ret->skipRecord[itr];
  }

  for (itr = 0; itr< c.numVars; itr++)
  {
    if (references[itr] > best)
    {
      best = references[itr];
      toUse = itr;
    }
  }

  free(references);

  break;
case 4: // find most >2SAT clause literal
  references = safeAlloc(c.numVars * sizeof(int));
  for (itr = 0; itr< c.numVars; itr++)
  {
    references[itr] = 0;
  }
```

```
  itr = 0;
  itr += ret->skipRecord[itr];
  numClauses = 0;
  while (itr < c.numSpots)
  {
    if (c.clauseNumbers[itr] == -1)
    {
      if (numClauses > 2)
      {
        int backItr = itr - 1;
        while ((backItr > -1) && (c.clauseNumbers[backItr] != -1))
        {
          if ((ret->assignments[c.clauseNumbers[backItr]] == 0))
          {
            references[c.clauseNumbers[backItr]] += 1;
          }
          backItr--;
        }
      }
      numClauses = 0;
    }
    else
    {
      if (ret->assignments[c.clauseNumbers[itr]] == 0)
      {
        numClauses++;
      }
    }
    itr++;
    itr += ret->skipRecord[itr];
  }

  for (itr = 0; itr< c.numVars; itr++)
  {
    if (references[itr] > best)
    {
      best = references[itr];
      toUse = itr;
    }
  }

  free(references);

  break;
}
```

```
if (toUse == -1) // hit bottom of search
{
  switch (searchType)
  {
  case 1:
    hornSolve(c, ret);
    break;
  case 2:
    SAT2resolve(c, ret);
    break;
  case 3:
    hornSolve(c, ret);
    break;
  case 4:
    SAT2resolve(c, ret);
    break;
  }
}
else
{
  Solution tryout = cloneSol(*ret);
  tryout.assignments[toUse] = 1;

  runUnitCheck(c, &tryout);

  if (tryout.solved)
  {
    freeSolution(*ret);
    *ret = tryout;
    return;
  }
  if (tryout.impossible)
  {
    freeSolution(tryout);
    ret->assignments[toUse] = -1;

    runUnitCheck(c, ret);
    if (ret->impossible)
    {
      return;
    }
    if (ret->solved)
    {
      return;
```

```
      }
      recursedBacktrack(c, ret, myDepth);
      return;
    }
    recursedBacktrack(c, &tryout, myDepth);
    if (tryout.solved)
    {
      freeSolution(*ret);
      *ret = tryout;
      return;
    }
    if (tryout.impossible)
    {
      freeSolution(tryout);
      ret->assignments[toUse] = -1;
      runUnitCheck(c, ret);
      if (ret->impossible)
      {
        return;
      }
      if (ret->solved)
      {
        return;
      }
      recursedBacktrack(c, ret, myDepth);
      return;
    }
  }
  return;
}

int depthStat()
{
  return searchDepth;
}
```

ValDavis.h

```
#ifndef VALDAVIS_H_
#define VALDAVIS_H_

#include "logicIncludes.h"

ValSolution runValDavis(Puzzle p);
void outputSolutionVal(Puzzle thePuzzle, ValSolution sol, double time);
ValSolution initSolution(Puzzle p);
void checkSolved (ValSolution * sol);
void freeVal(ValSolution s);
int getTupleNumber(int dom, int obj);
void joinTuple(int t1, int t2);
void excludeTuple(int t1, int t2);
void tupleCompare(int t1, int t2);
void applyClue(int gap, int dom, int t1, int t2);
bool doTry();

#endif /* VALDAVIS_H_ */
```

ValDavis.c

```
#include "ValDavis.h"
#include "clauseOutputter.h"

ValSolution theSol;
int tuplesToGo;
bool * dirtyList;
bool invalid;
int * tupleCluesDom;
int * tupleCluesGap;
int * tupleClues2;
int * tupleClues1;
Puzzle * theP;

ValSolution runValDavis(Puzzle p)
{
  theSol = initSolution(p);
  theP = &p;
  invalid = false;

  tuplesToGo = theSol.numTuples - theSol.numObj;
  dirtyList = safeAlloc(theSol.numTuples * sizeof(bool));
  tupleClues1 = safeAlloc(theSol.numTuples * sizeof(int));
  tupleClues2 = safeAlloc(theSol.numTuples * sizeof(int));
  tupleCluesGap = safeAlloc(theSol.numTuples * sizeof(int));
  tupleCluesDom = safeAlloc(theSol.numTuples * sizeof(int));
  int tuple;
  for (tuple = 0; tuple < theSol.numTuples; tuple++)
  {
    dirtyList[tuple] = false;
    tupleClues1[tuple] = -1;
    tupleClues2[tuple] = -1;
    tupleCluesGap[tuple] = -1;
    tupleCluesDom[tuple] = -1;
  }

  int i;
  for (i = 0; i<p.numClues; i++)
  {
    if (p.clueList[i].clueType == 0)
    {
      int tuple1 = getTupleNumber(p.clueList[i].obj1dom,p.clueList[i].obj1val);
      int tuple2 = getTupleNumber(p.clueList[i].obj2dom,p.clueList[i].obj2val);
```

```
    excludeTuple (tuple1, tuple2);
  }
  else if(p.clueList[i].clueType == 1)
  {
    int tuple1 = getTupleNumber(p.clueList[i].obj1dom,p.clueList[i].obj1val);
    int tuple2 = getTupleNumber(p.clueList[i].obj2dom,p.clueList[i].obj2val);

    joinTuple(tuple1, tuple2);
  }
  else if(p.clueList[i].clueType == 2)
  {
    int tuple1 = getTupleNumber(p.clueList[i].obj1dom,p.clueList[i].obj1val);
    int tuple2 = getTupleNumber(p.clueList[i].obj2dom,p.clueList[i].obj2val);

    // assign clue to tuples
    tupleClues1[i] = tuple1;
    tupleClues2[i] = tuple2;
    tupleCluesGap[i] = p.clueList[i].orderDist;
    tupleCluesDom[i] = p.clueList[i].orderedDom;
  }
  else
  {
    theSol.invalid = true;
    free(dirtyList);
    free(tupleClues1);
    free(tupleClues2);
    free(tupleCluesGap);
    free(tupleCluesDom);
    return theSol;
  }

  //outputSolutionVal(p,theSol,-1);
}

if (theSol.impossible)
{
  free(dirtyList);
  free(tupleClues1);
  free(tupleClues2);
  free(tupleCluesGap);
  free(tupleCluesDom);
  return theSol;
}

while (doTry()) {};
```

```
  if (tuplesToGo < 1)
  {
    checkSolved(&theSol);
  }

  free(dirtyList);
  free(tupleClues1);
  free(tupleClues2);
  free(tupleCluesGap);
  free(tupleCluesDom);
  return theSol;
}

bool doTry()
{
  int base = 0;
  while ((base < theSol.numTuples) &&
         ((dirtyList[base] == false) || (theSol.tupleIgnores[base] != -1)))
  {
    base++;
  }
  if (base >= theSol.numTuples)
  {
    return false;
  }
  dirtyList[base] = false;

  int tuple;

  for (tuple = 0; tuple < theSol.numTuples; tuple++)
  {
    if ((theSol.tupleIgnores[tuple] == -1) && (base != tuple))
    {
      tupleCompare(base, tuple);
    }
  }

  return true;
}

void tupleCompare(int t1, int t2)
{
  //check for applicable "related" clues, if found: apply them
  bool foundClue = false;
```

```
int clue;
for (clue = 0; clue < theSol.numTuples; clue++)
{
  if (tupleClues1[clue] != -1)
  {
    if ((tupleClues1[clue] == t1) && (tupleClues2[clue] == t2))
    {
      foundClue = true;
      applyClue(tupleCluesGap[clue],tupleCluesDom[clue],t1,t2);
    }
    else if ((tupleClues1[clue] == t2) && (tupleClues2[clue] == t1))
    {
      foundClue = true;
      applyClue(tupleCluesGap[clue],tupleCluesDom[clue],t2,t1);
    }
  }
}

if (foundClue)
{
  excludeTuple(t1, t2);
  return;
}

int domain;
int obj;
for (domain = 0; domain < theSol.numDom; domain++)
{
  int numTaken1 = 0;
  int numTaken2 = 0;
  int lastSingle1;
  int lastSingle2;
  bool diff = true;

  for (obj = 0; obj < theSol.numObj; obj++)
  {
    if (theSol.tupleList[t1*theSol.tupleSize + domain*theSol.numObj + obj])
    {
      numTaken1++;
      lastSingle1 = domain*theSol.numObj + obj;
    }
    if (theSol.tupleList[t2*theSol.tupleSize + domain*theSol.numObj + obj])
    {
      numTaken2++;
      lastSingle2 = domain*theSol.numObj + obj;
```

```
      }
      if (theSol.tupleList[t1*theSol.tupleSize + domain*theSol.numObj + obj] &&
          theSol.tupleList[t2*theSol.tupleSize + domain*theSol.numObj + obj])
      {
        diff = false;
      }
    }
    if (!diff && (numTaken1 == 1) && (numTaken2 == 1) &&
        (lastSingle1 == lastSingle2))
    {
      joinTuple(t1, t2);
      return;
    }
    else if (diff)
    {
      excludeTuple(t1, t2);
      return;
    }
  }
}

void excludeTuple(int t1, int t2)
{
  if (t1 == t2)
  {
    theSol.impossible = true;
    return;
  }
  int domain;
  int obj;
  for (domain = 0; domain < theSol.numDom; domain++)
  {
    int numTaken1 = 0;
    int numTaken2 = 0;
    int lastSingle1;
    int lastSingle2;

    for (obj = 0; obj < theSol.numObj; obj++)
    {
      if (theSol.tupleList[t1*theSol.tupleSize + domain*theSol.numObj + obj])
      {
        numTaken1++;
        lastSingle1 = domain*theSol.numObj + obj;
      }
      if (theSol.tupleList[t2*theSol.tupleSize + domain*theSol.numObj + obj])
```

```
      {
        numTaken2++;
        lastSingle2 = domain*theSol.numObj + obj;
      }
    }
    if ((numTaken1 == 1) && (theSol.tupleList[t2*theSol.tupleSize + lastSingle1]))
    {
      theSol.tupleList[t2*theSol.tupleSize + lastSingle1] = false;
      dirtyList[t2] = true;
    }
    if ((numTaken2 == 1) && (theSol.tupleList[t1*theSol.tupleSize + lastSingle2]))
    {
      theSol.tupleList[t1*theSol.tupleSize + lastSingle2] = false;
      dirtyList[t1] = true;
    }
  }
}

void joinTuple(int t1, int t2)
{
  if (t1 == t2)
  {
    return;
  }
  int domain;
  int obj;
  for (domain = 0; domain < theSol.numDom; domain++)
  {
    for (obj = 0; obj < theSol.numObj; obj++)
    {
      theSol.tupleList[t1*theSol.tupleSize + domain*theSol.numObj + obj] =
        theSol.tupleList[t1*theSol.tupleSize + domain*theSol.numObj + obj] &&
        theSol.tupleList[t2*theSol.tupleSize + domain*theSol.numObj + obj];
    }
  }
  theSol.tupleIgnores[t2] = t1;
  tuplesToGo--;
  dirtyList[t1] = true;

  int clue;
  for (clue = 0; clue < theSol.numTuples; clue++)
  {
    if (tupleClues1[clue] == t2)
    {
      tupleClues1[clue] = t1;
```

```
    }
    if (tupleClues2[clue] == t2)
    {
      tupleClues2[clue] = t1;
    }
  }
}

int getTupleNumber(int dom, int obj)
{
  int temp = dom*theSol.numObj + obj;
  while (theSol.tupleIgnores[temp] != -1)
  {
    temp = theSol.tupleIgnores[temp];
  }

  return temp;
}

void outputSolutionVal(Puzzle thePuzzle, ValSolution sol, double time)
{
  printf("method: Val-Davis ");

  if (time > -1)
  {
    printf("time: %fms\n", time);
  }
  else
  {
    printf("\n");
  }

  if (sol.invalid)
  {
    puts("Unable to solve. Algorithm does not hand this type of clue");
    return;
  }

  if (sol.impossible)
  {
    puts("No solution assignment exists. Tuple Dump:");
  }
  else if (!sol.solved)
  {
    puts("Puzzle Not Solved. Tuple Dump:");
```

```
}

int tuple;
int domain;
int obj;

for (tuple = 0; tuple < sol.numTuples; tuple++)
{
  if (sol.tupleIgnores[tuple] == -1)
  {
    printf("{");
    bool firstDom = true;
    for (domain = 0; domain < sol.numDom; domain++)
    {
      if (firstDom)
      {
        firstDom = false;
      }
      else
      {
        printf(", ");
      }
      int numTaken = 0;
      for (obj = 0; obj < sol.numObj; obj++)
      {
        if (sol.tupleList[tuple*sol.tupleSize + domain*sol.numObj + obj])
        {
          numTaken++;
        }
      }
      if (numTaken != 1)
      {
        printf("[");
      }

      bool first = true;
      for (obj = 0; obj < sol.numObj; obj++)
      {
        if (sol.tupleList[tuple*sol.tupleSize + domain*sol.numObj + obj])
        {
          if (first)
          {
            first = false;
          }
          else
```

```
            {
              printf(", ");
            }
            outputTextUnit(thePuzzle,domain, obj);
          }
        }

        if (numTaken != 1)
        {
          printf("]");
        }
      }
      printf("}\n");
    }
  }
}

void checkSolved (ValSolution * sol)
{
  int tuple;
  int domain;
  int obj;
  sol->solved = true;

  for (tuple = 0; tuple < sol->numTuples; tuple++)
  {
    if (sol->tupleIgnores[tuple] == -1)
    {
      for (domain = 0; domain < sol->numDom; domain++)
      {
        int numTaken = 0;
        for (obj = 0; obj < sol->numObj; obj++)
        {
          if (sol->tupleList[tuple*sol->tupleSize + domain*sol->numObj + obj])
          {
            numTaken++;
          }
        }
        if (numTaken == 0)
        {
          sol->solved = false;
          sol->impossible = true;
          return;
        }
        if (numTaken != 1)
```

```
        {
          sol->solved = false;
        }
      }
    }
  }
}

ValSolution initSolution(Puzzle p)
{
  ValSolution ret;
  ret.solved = false;
  ret.impossible = false;
  ret.numTuples = p.numDom * p.numObj;
  ret.numDom = p.numDom;
  ret.numObj = p.numObj;
  ret.invalid = false;
  ret.tupleSize = p.numDom * p.numObj;
  ret.limNumber = ret.numTuples * ret.tupleSize;

  ret.tupleList = safeAlloc(ret.limNumber * sizeof(bool));
  ret.tupleIgnores = safeAlloc(ret.numTuples * sizeof(int));

  int tuple;
  int domain;
  int obj;

  for (tuple = 0; tuple < ret.numTuples; tuple++)
  {
    int myDom = (tuple / ret.numObj) % ret.numDom;
    int myObj = tuple % ret.numObj;
    ret.tupleIgnores[tuple] = -1;

    for (domain = 0; domain < ret.numDom; domain++)
    {
      for (obj = 0; obj < ret.numObj; obj++)
      {
        if ((myDom == domain) && (myObj != obj))
        {
          ret.tupleList[tuple*ret.tupleSize + domain*ret.numObj + obj] = false;
        }
        else
        {
          ret.tupleList[tuple*ret.tupleSize + domain*ret.numObj + obj] = true;
        }
```

```
      }
    }
  }

  return ret;
}

void applyClue(int gap, int dom, int t1, int t2)
{
  int obj;
  for (obj = 0; obj < theSol.numObj; obj++)
  {
    if (theSol.tupleList[t1*theSol.tupleSize + dom*theSol.numObj + obj])
    {
      if (obj + gap >= theSol.numObj)
      {
        theSol.tupleList[t1*theSol.tupleSize + dom*theSol.numObj + obj] = false;
        dirtyList[t1] = true;
      }
      else if (theSol.tupleList[t2*theSol.tupleSize + dom*theSol.numObj + obj+gap] == false)
      {
        theSol.tupleList[t1*theSol.tupleSize + dom*theSol.numObj + obj] = false;
        dirtyList[t1] = true;
      }
    }

    if (theSol.tupleList[t2*theSol.tupleSize + dom*theSol.numObj + obj])
    {
      if (obj - gap < 0)
      {
        theSol.tupleList[t2*theSol.tupleSize + dom*theSol.numObj + obj] = false;
        dirtyList[t2] = true;
      }
      else if (theSol.tupleList[t1*theSol.tupleSize + dom*theSol.numObj + obj - gap] == false)
      {
        theSol.tupleList[t2*theSol.tupleSize + dom*theSol.numObj + obj] = false;
        dirtyList[t2] = true;
      }
    }
  }
}

void freeVal(ValSolution s)
{
  free(s.tupleIgnores);
```

```
  free(s.tupleList);
}
```

clauseMaker.h

```
#ifndef CLAUSEMAKER_H_
#define CLAUSEMAKER_H_

#include "logicIncludes.h"

int * pointMover;
bool * boolMover;

ClauseList getClauses(Puzzle p);
ClauseList getBigClauses(Clue * clues, int numClues);
ClauseList getSmallClauses(Clue * clues, int numClues);
void setLiteral(int n1, int n2, int n3, bool val);
void clauseDone();
void smallIsOrNot(int o1d, int o1c, int o2d, int o2c, bool isSame);
int getSmallVarNum(int obj1, int dom2,int obj2);
void smallIf(int o1d1, int o1c1, int o1d2, int o1c2,
             int o2d1, int o2c1, int o2d2, int o2c2, bool term1, bool term2);
void smallIsOrNotCount(int d1,int d2);
void smallIfCount(int o1d1, int o1d2, int o2d1, int o2d2);
int getBigVarNum(int dom1, int obj1, int dom2,int obj2);
void setLiteralBig(int n1, int n2, int n3, int n4, bool val);
void clauseFree(ClauseList l);
void bigIf(int o1d1, int o1c1, int o1d2, int o1c2,
           int o2d1, int o2c1, int o2d2, int o2c2, bool term1, bool term2);

#endif /* CLAUSEMAKER_H_ */
```

clauseMaker.c

```
#include "clauseMaker.h"

int doms;
int objs;

int literals;
int clauses;

ClauseList getClauses(Puzzle p)
{
  doms = p.numDom;
  objs = p.numObj;

  if (p.bigEncode)
  {
    return getBigClauses(p.clueList, p.numClues);
  }
  return getSmallClauses(p.clueList, p.numClues);
}

ClauseList getBigClauses(Clue * clues, int numClues)
{
  ClauseList ret;

  ret.numVars = doms * doms * objs * objs;

  //Step 1: count clauses
  clauses = 0;
  literals = 0;
  // For equiv
  clauses += 2*doms*objs*(doms-1)*objs;
  literals += 4*doms*objs*(doms-1)*objs;

  // objA is objB and objA is objC implied objB is objC
  int a,b,c,d,e,f;
  for (a = 0; a<doms; a++)
  {
    for (b = 0; b<objs; b++)
    {
      for (c = 0; c<doms; c++)
      {
        for (d = 0; d<objs; d++)
        {
```

```
        for (e = 0; e<doms; e++)
        {
          for (f = 0; f<objs; f++)
          {
            if ((a != c) && (a != e) && (c != e))
            {
              clauses += 2;
              literals += 6;
            }
          }
        }
      }
    }
  }

for (a = 0; a<doms; a++)
{
  for (b = 0; b<objs; b++)
  {
    for (c = 0; c<doms; c++)
    {
      if (a != c)
      {
        clauses += 1;
        literals += objs;
      }
    }
  }
}

for (a = 0; a<doms; a++)
{
  for (b = 0; b<objs; b++)
  {
    for (c = 0; c<doms; c++)
    {
      if (a != c)
      {
        for (d = 0; d<objs; d++)
        {
          for (e = d+1; e<objs; e++)
          {
            clauses += 1;
            literals += 2;
```

```
          }
        }
      }
    }
  }
}

int i;
int itr;
for (i = 0; i<numClues; i++)
{
  Clue aClue = clues[i];
  switch (aClue.clueType)
  {
  case 0: case 1: clauses++;literals++;
    break;
  case 2: clauses++;literals++;
    int it;
    for (it = 0; it < objs; it++)
    {
      if (it + aClue.orderDist < objs)
      {
        clauses += 1;literals += 2;
      }
      else
      {
        clauses += 2;literals += 2;
      }
    }
    break;
  case 3:
    clauses++;literals++;
    for (itr = 0; itr < objs; itr++)
    {
      int j;
      for (j = itr-1; j > -1; j--)
      {
        clauses += 1;literals += 2;
      }
    }
    clauses += 2;literals += 2;
    break;
  case 4: case 5: case 6:clauses += 1;literals += 2;
    break;
  case 7:
```

```
    clauses++;literals++;
    for (itr = 0; itr < objs; itr++)
    {
      int lowNum = itr - aClue.orderDist;
      int highNum = itr + aClue.orderDist;
      if (lowNum < 0)
      {
        if (highNum >= objs)
        {
          clauses+=2;literals+=2;
        }
        else
        {
          clauses += 2;literals += 4;
        }
      }
      else
      {
        if (highNum >= objs)
        {
          clauses+=2;literals+=4;
        }
        else
        {
          int j;
          for (j = 0; j < objs; j++)
          {
            if ((j != lowNum) && (j != itr) && (j != highNum))
            {
              clauses+=2;literals+=4;
            }
          }
        }
      }
    }
    break;
  default:
    printf("Invalid Clue Type.");
    exit(EXIT_SUCCESS);
  }
}

ret.numClauses = clauses;
ret.numLiterals = literals;
```

```
//Step 2: allocate and make them
ret.numSpots = ret.numClauses + ret.numLiterals;
ret.clauseNumbers = safeAlloc(ret.numSpots * sizeof(int));
ret.clauseVals = safeAlloc(ret.numSpots * sizeof(bool));

pointMover = ret.clauseNumbers;
boolMover = ret.clauseVals;

for (i = 0; i<numClues; i++)
{
  Clue aClue = clues[i];
  switch (aClue.clueType)
  {
  case 0: setLiteralBig(aClue.obj1dom,aClue.obj1val,
                        aClue.obj2dom,aClue.obj2val, false);clauseDone();
    break;
  case 1: setLiteralBig(aClue.obj1dom,aClue.obj1val,
                        aClue.obj2dom,aClue.obj2val, true);clauseDone();
    break;
  case 2:
    setLiteralBig(aClue.obj1dom,aClue.obj1val,
                  aClue.obj2dom,aClue.obj2val,false);clauseDone();
    int it;
    for (it = 0; it < objs; it++)//6 is gap, 5 is domain
    {
      if (it + aClue.orderDist < objs)
      {
        bigIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,it,
              aClue.obj2dom,aClue.obj2val,aClue.orderedDom,it + aClue.orderDist,
              true,true);
      }
      else
      {
        setLiteralBig(aClue.obj1dom,aClue.obj1val,
                      aClue.orderedDom,it,false);clauseDone();
        setLiteralBig(aClue.obj2dom,aClue.obj2val,
                      aClue.orderedDom,objs-it-1,false);clauseDone();
      }
    }
    break;
  case 3:
    setLiteralBig(aClue.obj1dom,aClue.obj1val,
                  aClue.obj2dom,aClue.obj2val,false);clauseDone();
    for (itr = 0; itr < objs; itr++)
    {
```

```
    int j;
    for (j = itr-1; j > -1; j--)
    {
      bigIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
            aClue.obj2dom,aClue.obj2val,aClue.orderedDom,j,true,false);
    }
  }
  setLiteralBig(aClue.obj1dom,aClue.obj1val,
                aClue.orderedDom,objs-1,false);clauseDone();
  setLiteralBig(aClue.obj2dom,aClue.obj2val,
                aClue.orderedDom,0,false);clauseDone();
  break;
case 4: bigIf(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,
              aClue.obj3dom,aClue.obj3val,aClue.obj4dom,aClue.obj4val,true, true);
  break;
case 5: bigIf(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,
              aClue.obj3dom,aClue.obj3val,aClue.obj4dom,aClue.obj4val,true, false);
  break;
case 6: bigIf(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,
              aClue.obj3dom,aClue.obj3val,aClue.obj4dom,aClue.obj4val,false, true);
  break;
case 7:
  setLiteralBig(aClue.obj1dom,aClue.obj1val,
                aClue.obj2dom,aClue.obj2val,false);clauseDone();
  for (itr = 0; itr < objs; itr++)
  {
    int lowNum = itr - aClue.orderDist;
    int highNum = itr + aClue.orderDist;
    if (lowNum < 0)
    {
      if (highNum >= objs)
      {
        setLiteralBig(aClue.obj1dom,aClue.obj1val,
                      aClue.orderedDom,itr,false);clauseDone();
        setLiteralBig(aClue.obj2dom,aClue.obj2val,
                      aClue.orderedDom,itr,false);clauseDone();
      }
      else
      {
        bigIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
              aClue.obj2dom,aClue.obj2val,aClue.orderedDom,highNum,true,true);
        bigIf(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,
              aClue.obj1dom,aClue.obj1val,aClue.orderedDom,highNum,true,true);
      }
    }
```

```
      else
      {
        if (highNum >= objs)
        {
          bigIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
                aClue.obj2dom,aClue.obj2val,aClue.orderedDom,lowNum,true,true);
          bigIf(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,
                aClue.obj1dom,aClue.obj1val,aClue.orderedDom,lowNum,true,true);
        }
        else
        {
          int j;
          for (j = 0; j < objs; j++)
          {
            if ((j != lowNum) && (j != itr) && (j != highNum))
            {
              bigIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
                    aClue.obj2dom,aClue.obj2val,aClue.orderedDom,j,true,false);
              bigIf(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,
                    aClue.obj1dom,aClue.obj1val,aClue.orderedDom,j,true,false);
            }
          }
        }
      }
    }
    break;
  default:
    printf("Invalid Clue Type.");
    exit(EXIT_SUCCESS);
  }
}

for (a = 0; a<doms; a++)
{
  for (b = 0; b<objs; b++)
  {
    for (c = 0; c<doms; c++)
    {
      if (a != c)
      {
        for (d = 0; d<objs; d++)
        {
          setLiteralBig(a,b,c,d,false);
          setLiteralBig(c,d,a,b,true);
          clauseDone();
```

```
          setLiteralBig(c,d,a,b,false);
          setLiteralBig(a,b,c,d,true);
          clauseDone();
        }
      }
    }
  }
}

// objA is objB and objA is objC implied objB is objC
for (a = 0; a<doms; a++)
{
  for (b = 0; b<objs; b++)
  {
    for (c = 0; c<doms; c++)
    {
      for (d = 0; d<objs; d++)
      {
        for (e = 0; e<doms; e++)
        {
          for (f = 0; f<objs; f++)
          {
            if ((a != c) && (a != e) && (c != e))
            {
              setLiteralBig(a,b,c,d,false);
              setLiteralBig(a,b,e,f,false);
              setLiteralBig(c,d,e,f,true);
              clauseDone();
              setLiteralBig(a,b,c,d,false);
              setLiteralBig(a,b,e,f,true);
              setLiteralBig(c,d,e,f,false);
              clauseDone();
            }
          }
        }
      }
    }
  }
}

// only one of each domain for obj
for (a = 0; a<doms; a++)
{
  for (b = 0; b<objs; b++)
  {
```

```
      for (c = 0; c<doms; c++)
      {
        if (a != c)
        {
          for (d = 0; d<objs; d++)
          {
            setLiteralBig(a,b,c,d,true);
          }
          clauseDone();
        }
      }
    }
  }

  for (a = 0; a<doms; a++)
  {
    for (b = 0; b<objs; b++)
    {
      for (c = 0; c<doms; c++)
      {
        if (a != c)
        {
          for (d = 0; d<objs; d++)
          {
            for (e = d+1; e<objs; e++)
            {
              setLiteralBig(a,b,c,d,false);
              setLiteralBig(a,b,c,e,false);

              clauseDone();
            }
          }
        }
      }
    }
  }

  return ret;
}

ClauseList getSmallClauses(Clue * clues, int numClues)
{
  ClauseList ret;

  ret.numVars = doms * (objs) * objs;
```

```
//Step 1: count clauses
clauses = 0;
literals = 0;
// For 3+ SAT Clauses
clauses += 2*(doms-1)*objs;
literals += 2*(doms-1)*objs*objs;
// For std 2SAT Clauses
clauses += objs*(doms-1)*(objs*objs - objs);
literals += objs*(doms-1)*(objs*objs - objs)*2;

int i;
int itr;
for (i = 0; i<numClues; i++)
{
  Clue aClue = clues[i];
  switch (aClue.clueType)
  {// finish this
  case 0: case 1: smallIsOrNotCount(aClue.obj1dom, aClue.obj2dom);
    break;
  case 2: smallIsOrNotCount(aClue.obj1dom,aClue.obj2dom);
    int it;
    for (it = 0; it < objs; it++)
    {
      if (it + aClue.orderDist < objs)
      {
        smallIfCount(aClue.obj1dom,aClue.orderedDom,aClue.obj2dom,aClue.orderedDom);
      }
      else
      {
        smallIsOrNotCount(aClue.obj1dom,aClue.orderedDom);
        smallIsOrNotCount(aClue.obj2dom,aClue.orderedDom);
      }
    }
    break;
  case 3:
    smallIsOrNotCount(aClue.obj1dom,aClue.obj2dom);
    for (itr = 0; itr < objs; itr++)
    {
      int j;
      for (j = itr-1; j > -1; j--)
      {
        smallIfCount(aClue.obj1dom,aClue.orderedDom,aClue.obj2dom,aClue.orderedDom);
      }
    }
```

```
  smallIsOrNotCount(aClue.obj1dom,aClue.orderedDom);
  smallIsOrNotCount(aClue.obj2dom,aClue.orderedDom);
  break;
case 4:case 5:case 6:smallIfCount(aClue.obj1dom,aClue.obj2dom,aClue.obj3dom,aClue.obj4dom);
  break;
case 7:
  smallIsOrNotCount(aClue.obj1dom,aClue.obj2dom);
  int itr;
  for (itr = 0; itr < objs; itr++)
  {
    int lowNum = itr - aClue.orderDist;
    int highNum = itr + aClue.orderDist;
    if (lowNum < 0)
    {
      if (highNum >= objs)
      {
        smallIsOrNotCount(aClue.obj1dom,aClue.orderedDom);
        smallIsOrNotCount(aClue.obj2dom,aClue.orderedDom);
      }
      else
      {
        smallIfCount(aClue.obj1dom,aClue.orderedDom,aClue.obj2dom,aClue.orderedDom);
        smallIfCount(aClue.obj2dom,aClue.orderedDom,aClue.obj1dom,aClue.orderedDom);
      }
    }
    else
    {
      if (highNum >= objs)
      {
        smallIfCount(aClue.obj1dom,aClue.orderedDom,aClue.obj2dom,aClue.orderedDom);
        smallIfCount(aClue.obj2dom,aClue.orderedDom,aClue.obj1dom,aClue.orderedDom);
      }
      else
      {
        int j;
        for (j = itr+1; j < objs; j++)
        {
          if ((j != lowNum) && (j != itr) && (j != highNum))
          {
            smallIfCount(aClue.obj1dom,aClue.orderedDom,
                         aClue.obj2dom,aClue.orderedDom);
            smallIfCount(aClue.obj2dom,aClue.orderedDom,
                         aClue.obj1dom,aClue.orderedDom);
          }
        }
```

```
        }
      }
    }
    break;
  default:
    printf("Invalid Clue Type.");
    exit(EXIT_SUCCESS);
  }
}

ret.numClauses = clauses;
ret.numLiterals = literals;

//Step 2: allocate and make them
ret.numSpots = ret.numClauses + ret.numLiterals;
ret.clauseNumbers = safeAlloc(ret.numSpots * sizeof(int));
ret.clauseVals = safeAlloc(ret.numSpots * sizeof(bool));

pointMover = ret.clauseNumbers;
boolMover = ret.clauseVals;

for (i = 0; i<numClues; i++)
{
  Clue aClue = clues[i];
  switch (aClue.clueType)
  {
  case 0: smallIsOrNot(aClue.obj1dom,aClue.obj1val,
                       aClue.obj2dom,aClue.obj2val, false);
    break;
  case 1: smallIsOrNot(aClue.obj1dom,aClue.obj1val,
                       aClue.obj2dom,aClue.obj2val, true);
    break;
  case 2:
    smallIsOrNot(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,false);
    int it;
    for (it = 0; it < objs; it++)//6 is gap, 5 is domain
    {
      if (it + aClue.orderDist < objs)
      {
        smallIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,it,
                aClue.obj2dom,aClue.obj2val,
                aClue.orderedDom,it + aClue.orderDist,true,true);
      }
      else
      {
```

```
      smallIsOrNot(aClue.obj1dom,aClue.obj1val,
                   aClue.orderedDom,it,false);
      smallIsOrNot(aClue.obj2dom,aClue.obj2val,
                   aClue.orderedDom,objs-it-1,false);
    }
  }
  break;
case 3:
  smallIsOrNot(aClue.obj1dom,aClue.obj1val,
               aClue.obj2dom,aClue.obj2val,false);
  for (itr = 0; itr < objs; itr++)
  {
    int j;
    for (j = itr-1; j > -1; j--)
    {
      smallIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
              aClue.obj2dom,aClue.obj2val,aClue.orderedDom,j,true,false);
    }
  }
  smallIsOrNot(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,objs-1,false);
  smallIsOrNot(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,0,false);
  break;
case 4: smallIf(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,
                aClue.obj3dom,aClue.obj3val,aClue.obj4dom,aClue.obj4val,true, true);
  break;
case 5: smallIf(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,
                aClue.obj3dom,aClue.obj3val,aClue.obj4dom,aClue.obj4val,true, false);
  break;
case 6: smallIf(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,
                aClue.obj3dom,aClue.obj3val,aClue.obj4dom,aClue.obj4val,false, true);
  break;
case 7:
  smallIsOrNot(aClue.obj1dom,aClue.obj1val,aClue.obj2dom,aClue.obj2val,false);
  for (itr = 0; itr < objs; itr++)
  {
    int lowNum = itr - aClue.orderDist;
    int highNum = itr + aClue.orderDist;
    if (lowNum < 0)
    {
      if (highNum >= objs)
      {
        smallIsOrNot(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,false);
        smallIsOrNot(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,false);
      }
      else
```

```
        {
          smallIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
                  aClue.obj2dom,aClue.obj2val,aClue.orderedDom,highNum,true,true);
          smallIf(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,
                  aClue.obj1dom,aClue.obj1val,aClue.orderedDom,highNum,true,true);
        }
      }
      else
      {
        if (highNum >= objs)
        {
          smallIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
                  aClue.obj2dom,aClue.obj2val,aClue.orderedDom,lowNum,true,true);
          smallIf(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,
                  aClue.obj1dom,aClue.obj1val,aClue.orderedDom,lowNum,true,true);
        }
        else
        {
          int j;
          for (j = itr+1; j < objs; j++)
          {
            if ((j != lowNum) && (j != itr) && (j != highNum))
            {
              smallIf(aClue.obj1dom,aClue.obj1val,aClue.orderedDom,itr,
                      aClue.obj2dom,aClue.obj2val,aClue.orderedDom,j,true,false);
              smallIf(aClue.obj2dom,aClue.obj2val,aClue.orderedDom,itr,
                      aClue.obj1dom,aClue.obj1val,aClue.orderedDom,j,true,false);
            }
          }
        }
      }
    }
    break;
  default:
    printf("Invalid Clue Type.");
    exit(EXIT_SUCCESS);
  }
}

int a, b, c, d;
for (a = 1; a < doms; a++)
{
  for (b = 0; b < objs; b++)
  {
    for (c = 0; c < objs; c++)
```

```
      {
        setLiteral(c,a,b,true);
      }
      clauseDone();
      for (c = 0; c < objs; c++)
      {
        setLiteral(b,a,c,true);
      }
      clauseDone();
    }
  }

  for (a = 0; a < objs; a++)
  {
    for (b = 1; b < doms; b++)
    {
      for (c = 0; c < objs; c++)
      {
        for (d = c+1; d < objs; d++)
        {
          setLiteral(a,b,c,false);
          setLiteral(a,b,d,false);
          clauseDone();
          setLiteral(c,b,a,false);
          setLiteral(d,b,a,false);
          clauseDone();
        }
      }
    }
  }
  return ret;
}

int getSmallVarNum(int obj1, int dom2,int obj2)
{
  return obj2 + (dom2-1)*objs + obj1*doms*objs;
}

int getBigVarNum(int dom1, int obj1, int dom2,int obj2)
{
  return obj2 + dom2*objs + obj1*doms*objs + dom1*objs*doms*objs;
}

void smallIsOrNot(int o1d, int o1c, int o2d, int o2c, bool isSame)
{
```

```
  if ((o1d == 0) && (o2d == 0))
  {
    if ((o1c != o2c) && (isSame))
    {
      printf("Your clues contradit themselves.\n");
      exit(EXIT_SUCCESS);
    }
    return;
  }
  if (o1d == 0)
  {
    setLiteral(o1c,o2d,o2c,isSame);
    clauseDone();
  }
  else if (o2d == 0)
  {
    setLiteral(o2c,o1d,o1c,isSame);
    clauseDone();
  }
  else
  {
    int i;
    for (i = 0; i < objs; i++)
    {
      setLiteral(i,o2d,o2c,isSame);
      setLiteral(i,o1d,o1c,false);
      clauseDone();
    }
  }
}

void smallIsOrNotCount(int d1,int d2)
{
  if ((d1 == 0) && (d2 == 0))
  {
    return;
  }
  if (d1 == 0)
  {
    literals++;
    clauses++;
  }
  else if (d2 == 0)
  {
    literals++;
```

```
    clauses++;
  }
  else
  {
    literals += 2*objs;
    clauses += objs;
  }
}

void smallIf(int o1d1, int o1c1, int o1d2, int o1c2,
             int o2d1, int o2c1, int o2d2, int o2c2, bool term1, bool term2)
{
  //~o:o1d1:o1c1 v ~o:o1d2:o1c2 v ~o2:o2d1:o2c1 v o2:o2d2:o2c2
  //~o1c1:o1d2:o1c2 v ~o2:o2d1:o2c1 v o2:o2d2:o2c2
  //~o:o1d1:o1c1 v ~o:o1d2:o1c2 v o2c1:o2d2:o2c2
  //~o1c1:o1d2:o1c2 v o2c1:o2d2:o2c2

  // first check for idiocy
  if (o1d1 == o1d2)
  {
    printf("Your ordered/conditional clues make no sense.");
    exit(EXIT_SUCCESS);
  }
  if (o2d1 == o2d2)
  {
    printf("Your ordered/conditional clues make no sense.");
    exit(EXIT_SUCCESS);
  }
  int i;
  if (o1d1 == 0)
  {
    if (o2d1 == 0)
    {
      setLiteral(o1c1,o1d2,o1c2,!term1);
      setLiteral(o2c1,o2d2,o2c2,term2);
      clauseDone();
      return;
    }

    if (o2d2 == 0)
    {
      setLiteral(o1c1,o1d2,o1c2,!term1);
      setLiteral(o2c2,o2d1,o2c1,term2);
      clauseDone();
      return;
```

```
  }

  //~o1c1:o1d2:o1c2 v ~o2:o2d1:o2c1 v o2:o2d2:o2c2
  for (i = 0; i < objs; i++)
  {
    setLiteral(o1c1,o1d2,o1c2, false);
    setLiteral(i,o2d1,o2c1, !term1);
    setLiteral(i,o2d2,o2c2, term2);
    clauseDone();
  }
  return;
}

if (o1d2 == 0)
{
  if (o2d1 == 0)
  {
    setLiteral(o1c2,o1d1,o1c1,!term1);
    setLiteral(o2c1,o2d2,o2c2,term2);
    clauseDone();
    return;
  }

  if (o2d2 == 0)
  {
    setLiteral(o1c2,o1d1,o1c1,!term1);
    setLiteral(o2c2,o2d1,o2c1,term2);
    clauseDone();
    return;
  }

  for (i = 0; i < objs; i++)
  {
    setLiteral(o1c2,o1d1,o1c1, false);
    setLiteral(i,o2d1,o2c1, !term1);
    setLiteral(i,o2d2,o2c2, term2);
    clauseDone();
  }
  return;
}

if (o2d1 == 0)//~o:o1d1:o1c1 v ~o:o1d2:o1c2 v o2c1:o2d2:o2c2
{
  for (i = 0; i < objs; i++)
  {
```

```
      setLiteral(i,o1d1,o1c1, false);
      setLiteral(i,o1d2,o1c2, !term1);
      setLiteral(o2c1,o2d2,o2c2, term2);
      clauseDone();
    }
    return;
  }

  if (o2d2 == 0)
  {
    for (i = 0; i < objs; i++)
    {
      setLiteral(i,o1d1,o1c1, false);
      setLiteral(i,o1d2,o1c2, !term1);
      setLiteral(o2c2,o2d1,o2c1, term2);
      clauseDone();
    }
    return;
  }

  for (i = 0; i < objs; i++)
  {
    int j;
    for (j = 0; j < objs; j++)
    {//~o:o1d1:o1c1 v ~o:o1d2:o1c2 v ~o2:o2d1:o2c1 v o2:o2d2:o2c2
      setLiteral(i,o1d1,o1c1, false);
      setLiteral(i,o1d2,o1c2, !term1);
      setLiteral(j,o2d1,o2c1, false);
      setLiteral(j,o2d2,o2c2, term2);
      clauseDone();
    }
  }
  return;
}

void bigIf(int o1d1, int o1c1, int o1d2, int o1c2,
           int o2d1, int o2c1, int o2d2, int o2c2, bool term1, bool term2)
{
  // first check for idiocy
  if (o1d1 == o1d2)
  {
    printf("Your ordered/conditional clues make no sense.");
    exit(EXIT_SUCCESS);
  }
  if (o2d1 == o2d2)
```

```
  {
    printf("Your ordered/conditional clues make no sense.");
    exit(EXIT_SUCCESS);
  }
  setLiteralBig(o1d1,o1c1,o1d2,o1c2,!term1);
  setLiteralBig(o2d1,o2c1,o2d2,o2c2,term2);
  clauseDone();
}

void smallIfCount(int o1d1, int o1d2, int o2d1, int o2d2)
{
  if (o1d1 == 0)
  {
    if (o2d1 == 0)
    {
      literals += 2;
      clauses++;
      return;
    }

    if (o2d2 == 0)
    {
      literals += 2;
      clauses++;
      return;
    }

    literals += 3*objs;
    clauses += objs;
    return;
  }

  if (o1d2 == 0)
  {
    if (o2d1 == 0)
    {
      literals += 2;
      clauses++;
      return;
    }

    if (o2d2 == 0)
    {
      literals += 2;
      clauses++;
```

```
      return;
    }

    literals += 3*objs;
    clauses += objs;
    return;
  }

  if (o2d1 == 0)
  {
    literals += 3*objs;
    clauses += objs;
    return;
  }

  if (o2d2 == 0)
  {
    literals += 3*objs;
    clauses += objs;
    return;
  }

  literals += 4*objs*objs;
  clauses += objs*objs;

  return;
}


void setLiteral(int n1, int n2, int n3, bool val)
{
  *pointMover = getSmallVarNum(n1, n2, n3); pointMover++;
  *boolMover = val; boolMover++;
}

void setLiteralBig(int n1, int n2, int n3, int n4, bool val)
{
  *pointMover = getBigVarNum(n1, n2, n3, n4); pointMover++;
  *boolMover = val; boolMover++;
}

void clauseDone()
{
  *pointMover = -1;
  pointMover++;
```

```
  boolMover++;
}

void clauseFree(ClauseList l)
{
  free(l.clauseNumbers);
  free(l.clauseVals);
}
```

clauseOutputter.h

```
#ifndef CLAUSEOUTPUTTER_H_
#define CLAUSEOUTPUTTER_H_

#include "logicIncludes.h"

void outputDebugVars(Solution s);
void outputClauses(Puzzle p, ClauseList c);
void outputCutClauses(Puzzle p, ClauseList c, Solution s);
void outputSolution(Puzzle p, Solution s, float time);
bool printCutClause(int *loc, Puzzle p, ClauseList c, Solution s, bool firstClause);
void printClause(int *loc, Puzzle p, ClauseList c, bool firstClause);
void outputLiteral (int varNum, bool val, Puzzle p);
void outputSolutionVars(Puzzle p, Solution s);
int getNum(int obj1, int dom2,int obj2);
void outputTextUnit(Puzzle p, int d, int o);
void outputDebugClauses(ClauseList c, Solution s);
bool printDebugClause(int *loc, ClauseList c, Solution s, bool firstClause);

#endif /* CLAUSEOUTPUTTER_H_ */
```

clauseOutputter.c

```
#include "clauseOutputter.h"
#include "clauseMaker.h"

bool useBigEncode;

void outputDebugVars(Solution s)
{
  puts("Assigned Variables:");
  bool first = true;
  int itr = 0;
  while (itr < s.numVars)
  {
    if (s.assignments[itr] == -1)
    {
      if (first)
      {
        first = false;
      }
      else
      {
        printf(", ");
      }
      printf("~%d",itr);
    }
    else if (s.assignments[itr] == 1)
    {
      if (first)
      {
        first = false;
      }
      else
      {
        printf(", ");
      }
      printf("%d",itr);
    }

    itr++;
  }
  printf("\n");
}

void outputSolutionVars(Puzzle p, Solution s)
```

```
{
  puts("Assigned Variables:");
  bool first = true;
  int itr = 0;
  while (itr < s.numVars)
  {
    if (s.assignments[itr] == -1)
    {
      if (first)
      {
        first = false;
      }
      else
      {
        printf(", ");
      }
      outputLiteral (itr, false, p);
    }
    else if (s.assignments[itr] == 1)
    {
      if (first)
      {
        first = false;
      }
      else
      {
        printf(", ");
      }
      outputLiteral (itr, true, p);
    }

    itr++;
  }
  printf("\n");
}

void outputClauses(Puzzle p, ClauseList c)
{
  puts("Boolean Formula:");
  int itr = 0;
  bool first = true;
  while (itr < c.numSpots)
  {
    printClause(&itr, p, c, first);
    first = false;
```

```
    itr++;
  }
  printf("\n");
}

void outputLiteral (int varNum, bool val, Puzzle p)
{
  if (val == false)
  {
    printf("~");
  }
  if (p.bigEncode)
  {
    printf("%i#",varNum/(p.numObj*p.numDom*p.numObj));
    varNum %= p.numObj*p.numDom*p.numObj;
  }
  printf("%i#",varNum/(p.numDom*p.numObj));
  varNum %= p.numDom*p.numObj;
  if (!p.bigEncode)
  {
    varNum+=p.numObj;
  }
  printf("%i#",varNum/(p.numObj));
  varNum %= p.numObj;
  printf("%i",varNum);
}

void outputCutClauses(Puzzle p, ClauseList c, Solution s)
{
  if (s.solved)
  {
    puts("Formula Satisfied\n");
    return;
  }

  puts("Boolean Formula:");
  int itr = 0;
  bool first = true;
  while (itr < c.numSpots)
  {
    if (printCutClause(&itr, p, c, s, first))
    {
      first = false;
    }
```

```
    itr++;
  }
  printf("\n");
}

void outputDebugClauses(ClauseList c, Solution s)
{
  if (s.solved)
  {
    puts("Formula Satisfied\n");
    return;
  }

  puts("Boolean Formula:");
  int itr = 0;
  bool first = true;
  while (itr < c.numSpots)
  {
    if (printDebugClause(&itr, c, s, first))
    {
      first = false;
    }

    itr++;
  }
  printf("\n");
}

void printClause(int *loc, Puzzle p, ClauseList c, bool firstClause)
{
  bool first = true;
  while (c.clauseNumbers[*loc] != -1)
  {
    if (first)
    {
      first = false;
      if (!firstClause)
      {
        printf(" ^ \n");
      }
      printf("(");
    }
    else
    {
```

```
      printf(" v ");
    }

    outputLiteral (c.clauseNumbers[*loc], c.clauseVals[*loc], p);

    (*loc)++;
  }
  printf(")");
}

bool printCutClause(int *loc, Puzzle p, ClauseList c, Solution s, bool firstClause)
{
  bool first = true;
  int scan = *loc;
  bool bad = false;
  while (c.clauseNumbers[scan] != -1)
  {
    if (s.assignments[c.clauseNumbers[scan]] == 1)
    {
      if (c.clauseVals[scan] == true)
      {
        bad = true;
      }
    }
    else if (s.assignments[c.clauseNumbers[scan]] == -1)
    {
      if (c.clauseVals[scan] == false)
      {
        bad = true;
      }
    }
    scan++;
  }

  if (bad)
  {
    *loc = scan;
    return false;
  }

  while (c.clauseNumbers[*loc] != -1)
  {
    if (s.assignments[c.clauseNumbers[*loc]] == 0)
    {
      if (first)
```

```
      {
        first = false;
        if (!firstClause)
        {
          printf(" ^ \n");
        }
        printf("(");
      }
      else
      {
        printf(" v ");
      }

      outputLiteral (c.clauseNumbers[*loc], c.clauseVals[*loc], p);
    }
    (*loc)++;
  }
  if (!first)
  {
    printf(")");
  }

  return true;
}

bool printDebugClause(int *loc, ClauseList c, Solution s, bool firstClause)
{
  bool first = true;
  int scan = *loc;
  bool bad = false;
  while (c.clauseNumbers[scan] != -1)
  {
    if (s.assignments[c.clauseNumbers[scan]] == 1)
    {
      if (c.clauseVals[scan] == true)
      {
        bad = true;
      }
    }
    else if (s.assignments[c.clauseNumbers[scan]] == -1)
    {
      if (c.clauseVals[scan] == false)
      {
        bad = true;
      }
```

```
    }
    scan++;
  }

  if (bad)
  {
    *loc = scan;
    return false;
  }

  while (c.clauseNumbers[*loc] != -1)
  {
    if (first)
    {
      first = false;
      if (!firstClause)
      {
        printf(" ^ \n");
      }
      printf("(");
    }
    else
    {
      printf(" v ");
    }

    if (!c.clauseVals[*loc]) printf("~");
    printf("%d",c.clauseNumbers[*loc]);
    (*loc)++;
  }
  if (!first)
  {
    printf(")");
  }

  return true;
}

void outputSolution(Puzzle p, Solution s, float time)
{
  switch (p.algorithm)
  {
  case 1:printf("method: Singular Variables only ");break;
  case 2:printf("method: Failed literal rule ");break;
  case 3:printf("method: Binary failed literal rule ");break;
```

```
case 4:printf("method: Simple backtracking ");break;
case 5:printf("method: Backtracking for horn ");break;
case 6:printf("method: Backtracking for 2SAT. ");break;
case 7:printf("method: Backtracking for horn (better)");break;
case 8:printf("method: Backtracking for 2SAT. (better)");break;
}

if (p.bigEncode)
{
  printf("encoding: Large ");
}
else
{
  printf("encoding: Small ");
}

if (time != -1)
{
  printf("time: %fms\n", time);
}
else
{
  printf("\n");
}

if (s.impossible)
{
  puts("No solution assignment exists.");
  return;
}

if (!s.solved)
{
  puts("Puzzle Not Solved");
  return;
}

useBigEncode = false;
if (p.bigEncode)
{
  useBigEncode = true;
}

int itr, j, k;
```

```
  for (itr = 0; itr<p.numObj;itr++)
  {
    outputTextUnit(p, 0, itr);
    for (j = 1; j < p.numDom; j++)
    {
      printf(", ");
      for (k = 0; k < p.numObj; k++)
      {
        if (s.assignments[getNum(itr,j,k)] == 1)
        {
          outputTextUnit(p, j, k);
        }
      }
    }
    printf("\n");
  }
}

int getNum(int obj1, int dom2,int obj2)
{
  if (useBigEncode)
  {
    return getBigVarNum(0, obj1, dom2,obj2);
  }
  return getSmallVarNum(obj1, dom2,obj2);
}

void outputTextUnit(Puzzle p, int d, int o)
{
  printf("%s",p.textPtrs[d*(p.numObj+1)+o+1]);
}
```

dataCollect.h

```
#ifndef DATACOLLECT_H_
#define DATACOLLECT_H_

#include "fileRead.h"
#include "clauseMaker.h"
#include "clauseOutputter.h"
#include "timer.h"
#include "solvers.h"
#include "logicIncludes.h"

void runDataCollect();
void runSeveralSolvers(Puzzle p, int place, theDATA data);

#endif /* DATACOLLECT_H_ */
```

dataCollect.c

```
#include "dataCollect.h"
#include "backtracker.h"
#include "ValDavis.h"

const int NUM_BACKTRAKS = 5;
const int NUM_STATS = 11;

void runDataCollect()
{
  int i,j,k;
  char * puz;
  puz = safeAlloc(64*sizeof(char));
  i = 0;
  bool puzCountDone = false;

  while (!puzCountDone)
  {
    i++;
    sprintf(puz, "puzzle%d.txt",i);
    FILE *inputFile = fopen(puz, "r");
    if (inputFile == NULL)
    {
      puzCountDone = true;
    }
    fclose(inputFile);
  }

  int NUM_PUZZLES = i-1;

  if (NUM_PUZZLES == 0)
  {
      printf("No Puzzles to do./n");
      exit(EXIT_SUCCESS);
  }

  const int NUM_METHODS = 9;

  theDATA theData;
  //initalize the DATA
  theData.numMethods = NUM_METHODS;
  theData.numPuzzles = NUM_PUZZLES;
  theData.puzStats = safeAlloc(NUM_PUZZLES * (NUM_BACKTRAKS*2+NUM_STATS) * sizeof(int));
  theData.ValDavisValid = safeAlloc(NUM_PUZZLES * sizeof(bool));
```

```
theData.fails = safeAlloc(NUM_PUZZLES * NUM_METHODS * 2 * sizeof(bool*));
theData.sucesses = safeAlloc(NUM_PUZZLES * NUM_METHODS * 2 * sizeof(bool*));
theData.times = safeAlloc(NUM_PUZZLES * NUM_METHODS * 2 * sizeof(float*));

for (i = 1; i <= NUM_PUZZLES; i++)
{
  sprintf(puz, "puzzle%d.txt",i);
  FILE *inputFile = fopen(puz, "r");
  if (inputFile == NULL)
  {
    printf("Unable to read file.");
    exit(EXIT_SUCCESS);
  }

  Puzzle thePuzzle = getPuzzle(inputFile);
  fclose(inputFile);

  runSeveralSolvers(thePuzzle,i-1,theData);
  freePuzzle(thePuzzle);
}
free(puz);

puts("Puzzles Stats:");
printf("puzzle\t");
k = 0;
int datas = (NUM_BACKTRAKS*2+NUM_STATS);
printf("domains\tobjects\tclues\tclauS\tlitS\tclauSC\tlitSC\t");
printf("clauB\tlitB\tclauBC\tlitBC\td1S\td2S\td3S\td4S\td5S\td1B\td2B\td3B\td4B\td5B");

printf("\n");
for (i = 0; i < NUM_PUZZLES; i++)
{
  printf("%d\t",i+1);
  for (j = 0; j < datas; j++)
  {
    printf("%d\t",theData.puzStats[k]);
    k++;
  }
  printf("\n");
}
printf("\n");


puts("Puzzles Solved, small encode:");
```

```
printf("method\t");
k = 0;
for (i = 1; i <= NUM_METHODS; i++)
{
  printf("%d\t",i);
}
printf("\n");
for (i = 0; i < NUM_PUZZLES; i++)
{
  printf("%d\t",i+1);
  if (theData.ValDavisValid[i])
  {
    if (theData.fails[k])
    {
      printf("II\t");
    }
    else
    {
      printf("%d\t",theData.sucesses[k]);
    }
  }
  else
  {
    printf("XX\t");
  }
  k += 2;
  for (j = 1; j < NUM_METHODS; j++)
  {
    if (theData.fails[k])
    {
      printf("II\t");
    }
    else
    {
      if (theData.fails[k])
      {
        printf("II\t");
      }
      else
      {
        printf("%d\t",theData.sucesses[k]);
      }
    }
    k += 2;
  }
```

```
  printf("\n");
}
printf("\n");

puts("Puzzles Solved, big encode:");
printf("method\t");
k = 1;
for (i = 1; i <= NUM_METHODS; i++)
{
  printf("%d\t",i);
}
printf("\n");
for (i = 0; i < NUM_PUZZLES; i++)
{
  printf("%d\t",i+1);
  for (j = 0; j < NUM_METHODS; j++)
  {
    if (theData.fails[k])
    {
      printf("II\t");
    }
    else
    {
      printf("%d\t",theData.sucesses[k]);
    }
    k += 2;
  }
  printf("\n");
}
printf("\n");

puts("Times, small encode:");
printf("method\t");
k = 0;
for (i = 1; i <= NUM_METHODS; i++)
{
  printf("%d\t",i);
}
printf("\n");
for (i = 0; i < NUM_PUZZLES; i++)
{
  printf("%d\t",i+1);
  for (j = 0; j < NUM_METHODS; j++)
  {
    if (theData.times[k] > (float)120000)
```

```
      {
        printf(">120000ms\t");
      }
      else
      {
        printf("%f\t",theData.times[k]);
      }
      k += 2;
    }
    printf("\n");
  }
  printf("\n");

  puts("Times, big encode:");
  printf("method\t");
  k= 1;
  for (i = 1; i <= NUM_METHODS; i++)
  {
    printf("%d\t",i);
  }
  printf("\n");
  for (i = 0; i < NUM_PUZZLES; i++)
  {
    printf("%d\t",i+1);
    for (j = 0; j < NUM_METHODS; j++)
    {
      if (theData.times[k] > 120000.0)
      {
        printf(">120000ms\t");
      }
      else
      {
        printf("%f\t",theData.times[k]);
      }
      k += 2;
    }
    printf("\n");
  }

  free(theData.puzStats);
  free(theData.times);
  free(theData.sucesses);
}

void runSeveralSolvers(Puzzle p, int place, theDATA data)
```

```
{
  int i;
  p.bigEncode = false;
  ClauseList clausesS = getClauses(p);
  p.bigEncode = true;
  ClauseList clausesB = getClauses(p);

  int statsoff = (NUM_BACKTRAKS*2+NUM_STATS) * place;
  data.puzStats[statsoff] = p.numDom;
  data.puzStats[statsoff + 1] = p.numObj;
  data.puzStats[statsoff + 2] = p.numClues;

  int * toStats = getStats(&clausesS);
  data.puzStats[statsoff + 3] = toStats[0];
  data.puzStats[statsoff + 4] = toStats[1];
  data.puzStats[statsoff + 5] = toStats[2];
  data.puzStats[statsoff + 6] = toStats[3];

  toStats = getStats(&clausesB);
  data.puzStats[statsoff + 7] = toStats[0];
  data.puzStats[statsoff + 8] = toStats[1];
  data.puzStats[statsoff + 9] = toStats[2];
  data.puzStats[statsoff + 10] = toStats[3];

  int q = place * data.numMethods * 2;

  timerSet();
  ValSolution solvedV = runValDavis(p);
  float time = timerGet();

  data.ValDavisValid[place] = !solvedV.invalid;

  data.sucesses[q] = solvedV.solved;
  data.fails[q] = solvedV.impossible;
  data.times[q] = time;
  q++;
  data.sucesses[q] = solvedV.solved;
  data.fails[q] = solvedV.impossible;
  data.times[q] = time;
  q++;
  freeVal(solvedV);

  for (i = 1; i<data.numMethods;i++)
  {
    timerSet();
```

```
    Solution solvedS = solveSAT(clausesS, i);
    float time1 = timerGet();

    if (i > 3)
    {
      data.puzStats[statsoff + NUM_STATS + 2*(i-4)] = depthStat();
    }

    timerSet();
    Solution solvedB = solveSAT(clausesB, i);
    float time2 = timerGet();

    data.sucesses[q] = solvedS.solved;
    data.fails[q] = solvedS.impossible;
    data.times[q] = time1;
    q++;
    data.sucesses[q] = solvedB.solved;
    data.fails[q] = solvedB.impossible;
    data.times[q] = time2;
    q++;

    freeSolution(solvedS);
    freeSolution(solvedB);

    if (i > 3)
    {
      data.puzStats[statsoff + NUM_STATS + 2*(i-4) + 1] = depthStat();
    }
  }

  clauseFree(clausesS);
  clauseFree(clausesB);
}
```

fileRead.h

```
#ifndef FILEREAD_H_
#define FILEREAD_H_

#include "logicIncludes.h"

Puzzle getPuzzle(FILE *filepointer);
int getDataDom(Puzzle puzzle, char * toFind);
int getDataObj(Puzzle puzzle, char * toFind);
int getDom(Puzzle puzzle, char * toFind);
void freePuzzle(Puzzle p);

#endif /* FILEREAD_H_ */
```

fileRead.c

```
#include "fileRead.h"

Puzzle getPuzzle(FILE *filepointer)
{
  Puzzle ret;

  // get algorithm number
  ret.algorithm = 0;
  char aChar = fgetc(filepointer);
  while (aChar != '\n')
  {
    if ((aChar < 58) && (aChar > 47))
    {
      ret.algorithm *= 10;
      ret.algorithm += (int)aChar-48;
    }
    else
    {
      printf("First line should be valid algorithm number.\n");
      exit(EXIT_SUCCESS);
    }
    aChar = fgetc(filepointer);
  }

  int dataSize = 1024;
  int pos = 0;
  ret.dataText = safeAlloc(dataSize * sizeof(char));
  char copyScan;
  while(0 == feof(filepointer))
  {
    copyScan = fgetc(filepointer);
    if (pos >= dataSize)
    {
      dataSize += 1024;
      ret.dataText = safeReAlloc(ret.dataText,dataSize * sizeof(char));
    }
    ret.dataText[pos] = copyScan;

    pos++;
  }
  ret.dataText[pos-1] = '\0';

  char * cluePt = strstr(ret.dataText,"\nClues:\n");
```

```
if (cluePt == NULL)
{
  printf("Use a line \"Clues:\" to indicate start of clues.\n");
  exit(EXIT_SUCCESS);
}

int countSemis = 0;
int countLines = 1;

char * scanner = ret.dataText;
while (scanner < cluePt)
{
  if (*scanner == ':')
  {
    countSemis++;
    *scanner = '\0';
  }
  if (*scanner == '\n')
  {
    countLines++;
    *scanner = '\0';
  }
  scanner++;
}

if (countSemis % countLines != 0)
{
  printf("Invalid Puzzle File.");
  exit(EXIT_SUCCESS);
}

ret.numDom = countLines;
ret.numObj = countSemis / countLines;

int limNum = (ret.numDom * (ret.numObj + 1));
ret.textPtrs = safeAlloc(limNum * sizeof(char*));
scanner = ret.dataText;
ret.textPtrs[0] = ret.dataText;
int itr = 1;
while (scanner < cluePt)
{
  if (*scanner == '\0')
  {
    if (itr >= limNum)
    {
```

```
      printf("Invalid Puzzle File.");
      exit(EXIT_SUCCESS);
    }
    ret.textPtrs[itr] = scanner + 1;
    itr++;
  }
  scanner++;
}
*cluePt = '\0';
cluePt += 8;//move pointer to start of clue list; C,l,u,e,s,:,\n +1

ret.numClues = 1;
scanner = cluePt;
while (*scanner != '\0')
{
  if (*scanner == '\n')
  {
    ret.numClues++;
  }
  scanner++;
}
char * endPoint = scanner;

if (*(scanner-1) == '\n')
{
  ret.numClues--;
  *(scanner-1) = '\0';
  endPoint--;
}

scanner = cluePt;
while (scanner < endPoint)
{
  if (*scanner == '\n')
  {
    *scanner = '\0';
  }
  scanner++;
}

scanner = cluePt;
while (scanner < endPoint)
{
  char * lineStart = scanner;
  while (*scanner != '\0')
```

```
  {
    scanner++;
  }
  char * nextLine = scanner + 1;

  if (strchr(lineStart,'*') != NULL)//This is not that
  {
    lineStart = strchr(lineStart,'*')+1;
    int numDot = 2;
    while (strchr(lineStart,'*') != NULL)
    {
      numDot++;
      lineStart = strchr(lineStart,'*') + 1;
    }
    ret.numClues += (numDot*numDot - numDot)/2 - 1;
  }

  scanner = nextLine;
}

//Last step, make clue structure
ret.clueList = safeAlloc(ret.numClues * sizeof(Clue));

itr = 0;

scanner = cluePt;
while (itr < ret.numClues)
{
  char * lineStart = scanner;
  while (*scanner != '\0')
  {
    scanner++;
  }
  char * nextLine = scanner + 1;

  char * usePoints [4];

  if (strchr(lineStart,'*') != NULL)//This is not that
  {
    char * ptr1;
    ptr1 = strchr(lineStart,'*');
    *ptr1 = '\0';
    ptr1++;
    while (strchr(ptr1,'*') != NULL)
    {
```

```
  ptr1 = strchr(ptr1,'*');
  *ptr1 = '\0';
  ptr1++;
}

char * str1 = lineStart;

char * str2 = str1;
while (*str2 != '\0')
{
  str2++;
}
str2++;

while (str1 < nextLine)
{
  while (str2 < nextLine)
  {
    if (getDataDom(ret,str1) != getDataDom(ret,str2))
    {
      ret.clueList[itr].clueType = 0;

      ret.clueList[itr].obj1dom = getDataDom(ret,str1);
      ret.clueList[itr].obj1val = getDataObj(ret,str1);
      ret.clueList[itr].obj2dom = getDataDom(ret,str2);
      ret.clueList[itr].obj2val = getDataObj(ret,str2);

      itr++;
    }
    else
    {
      ret.numClues--;
    }
    while (*str2 != '\0')
    {
      str2++;
    }
    str2++;
  }
  while (*str1 != '\0')
  {
    str1++;
  }
  str1++;
  str2 = str1;
```

```
    while (*str2 != '\0')
    {
      str2++;
    }
    str2++;
  }
  itr--;
}
else if (strchr(lineStart,'=') != NULL)//This is that
{
  ret.clueList[itr].clueType = 1;
  usePoints[0] = lineStart;
  usePoints[1] = strchr(lineStart,'=') + 1;
  *(usePoints[1] - 1) = '\0';
}
else if (strchr(lineStart,'$') != NULL)//Ordered Domain
{
  ret.clueList[itr].clueType = 3;
  usePoints[0] = strchr(lineStart,'$') + 1;
  *(usePoints[0] - 1) = '\0';

  ret.clueList[itr].orderedDom = getDom(ret, lineStart);

  if (strchr(usePoints[0],':') == NULL)
  {
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
  usePoints[1] = strchr(usePoints[0],':') + 1;
  *(usePoints[1] - 1) = '\0';

  char * temp = strchr(usePoints[1],':');

  if (temp != NULL)
  {
    *temp = '\0';
    temp++;
    ret.clueList[itr].clueType = 2;// A specific gap used
    ret.clueList[itr].orderDist = strtol(temp, NULL, 10);
    if (ret.clueList[itr].orderDist == 0)
    {
      printf("Bad Clue type.");
      exit(EXIT_SUCCESS);
    }
  }
```

```
}
else if (strchr(lineStart,'%') != NULL)//Ordered Domain, offset by an amount up OR down
{
  ret.clueList[itr].clueType = 7;
  usePoints[0] = strchr(lineStart,'%') + 1;
  *(usePoints[0] - 1) = '\0';

  ret.clueList[itr].orderedDom = getDom(ret, lineStart);

  if (strchr(usePoints[0],':') == NULL)
  {
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
  usePoints[1] = strchr(usePoints[0],':') + 1;
  *(usePoints[1] - 1) = '\0';

  char * temp = strchr(usePoints[1],':');

  if (temp != NULL)
  {
    *temp = '\0';
    temp++;
    ret.clueList[itr].orderDist = strtol(temp, NULL, 10);
    if (ret.clueList[itr].orderDist == 0)
    {
      printf("Bad Clue type.");
      exit(EXIT_SUCCESS);
    }
  }
  else
  {
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
}
else if (strchr(lineStart,'>') != NULL)//These imply those
{
  ret.clueList[itr].clueType = 4;
  usePoints[0] = lineStart;
  usePoints[1] = strchr(lineStart,'>') + 1;
  *(usePoints[1] - 1) = '\0';

  if (strchr(usePoints[1],'>') == NULL)
  {
```

```
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
  usePoints[2] = strchr(usePoints[1],'>') + 1;
  *(usePoints[2] - 1) = '\0';

  if (strchr(usePoints[2],'>') == NULL)
  {
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
  usePoints[3] = strchr(usePoints[2],'>') + 1;
  *(usePoints[3] - 1) = '\0';
}
else if (strchr(lineStart,'|') != NULL)//These imply not those
{
  ret.clueList[itr].clueType = 5;
  usePoints[0] = lineStart;
  usePoints[1] = strchr(lineStart,'|') + 1;
  *(usePoints[1] - 1) = '\0';

  if (strchr(usePoints[1],'|') == NULL)
  {
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
  usePoints[2] = strchr(usePoints[1],'|') + 1;
  *(usePoints[2] - 1) = '\0';

  if (strchr(usePoints[2],'|') == NULL)
  {
    printf("Bad Clue type.");
    exit(EXIT_SUCCESS);
  }
  usePoints[3] = strchr(usePoints[2],'|') + 1;
  *(usePoints[3] - 1) = '\0';
}
else if (strchr(lineStart,'}') != NULL)//Not These imply those
{
  ret.clueList[itr].clueType = 6;
  usePoints[0] = lineStart;
  usePoints[1] = strchr(lineStart,'}') + 1;
  *(usePoints[1] - 1) = '\0';

  if (strchr(usePoints[1],'}') == NULL)
```

```
      {
        printf("Bad Clue type.");
        exit(EXIT_SUCCESS);
      }
      usePoints[2] = strchr(usePoints[1],'}') + 1;
      *(usePoints[2] - 1) = '\0';

      if (strchr(usePoints[2],'}') == NULL)
      {
        printf("Bad Clue type.");
        exit(EXIT_SUCCESS);
      }
      usePoints[3] = strchr(usePoints[2],'}') + 1;
      *(usePoints[3] - 1) = '\0';
    }
    else
    {
      printf("Bad Clue type.");
      exit(EXIT_SUCCESS);
    }

    if (ret.clueList[itr].clueType != 0)
    {
      ret.clueList[itr].obj1dom = getDataDom(ret,usePoints[0]);
      ret.clueList[itr].obj1val = getDataObj(ret,usePoints[0]);
      ret.clueList[itr].obj2dom = getDataDom(ret,usePoints[1]);
      ret.clueList[itr].obj2val = getDataObj(ret,usePoints[1]);
    }

    if ((ret.clueList[itr].clueType > 3) && (ret.clueList[itr].clueType < 7))
    {
      ret.clueList[itr].obj3dom = getDataDom(ret,usePoints[2]);
      ret.clueList[itr].obj3val = getDataObj(ret,usePoints[2]);
      ret.clueList[itr].obj4dom = getDataDom(ret,usePoints[3]);
      ret.clueList[itr].obj4val = getDataObj(ret,usePoints[3]);
    }

    scanner = nextLine;
    itr++;
  }

  return ret;
}

int getDataDom(Puzzle puzzle, char * toFind)
```

```
{
  int i,j;
  for (i = 0; i < puzzle.numDom; i++)
  {
    for (j = 0; j < puzzle.numObj; j++)
    {
      if (0 == strcmp(toFind,puzzle.textPtrs[i*(puzzle.numObj+1)+j+1]))
      {
        return i;
      }
    }
  }
  printf("Clue text not found:%s\n",toFind);
  exit(EXIT_SUCCESS);
  return -1;
}

int getDataObj(Puzzle puzzle, char * toFind)
{
  int i,j;
  for (i = 0; i < puzzle.numDom; i++)
  {
    for (j = 0; j < puzzle.numObj; j++)
    {
      if (0 == strcmp(toFind,puzzle.textPtrs[i*(puzzle.numObj+1)+j+1]))
      {
        return j;
      }
    }
  }
  printf("Clue text not found.%s\n",toFind);
  exit(EXIT_SUCCESS);
  return -1;
}

int getDom(Puzzle puzzle, char * toFind)
{
  int i;
  for (i = 0; i < puzzle.numDom; i++)
  {
    if (0 == strcmp(toFind,puzzle.textPtrs[i*(puzzle.numObj+1)]))
    {
      return i;
    }
  }
```

```
  printf("Clue text not found.%s\n",toFind);
  exit(EXIT_SUCCESS);
  return -1;
}

void freePuzzle(Puzzle p)
{
  free(p.dataText);
  free(p.clueList);
}
```

logicIncludes.h

```
#ifndef LOGICINCLUDES_H_
#define LOGICINCLUDES_H_

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>

typedef struct aClue{
  int clueType;
  int obj1dom;
  int obj1val;
  int obj2dom;
  int obj2val;

  int obj3dom;
  int obj3val;
  int obj4dom;
  int obj4val;
  int orderedDom;
  long orderDist;
} Clue;

typedef struct puzzleData{
  int algorithm;
  bool bigEncode;
  int numClues;
  int numDom;
  int numObj;
  Clue * clueList;
  char * dataText;
  char ** textPtrs;
} Puzzle;

typedef struct Clauses{
  int * clauseNumbers;
  bool * clauseVals;
  int numSpots;
  int numClauses;
  int numLiterals;
  int numVars;
} ClauseList;
```

```
typedef struct Solved{
  int * skipRecord;
  int numVars;
  int numSkips;
  short * assignments;
  bool solved;
  bool impossible;
} Solution;

typedef struct CollectedData{
  int numPuzzles;
  int numMethods;
  float * times;
  bool * sucesses;
  int * puzStats;
  bool * fails;
  bool * ValDavisValid;
} theDATA;

typedef struct tuplesolve
{
  bool solved;
  bool impossible;
  int numTuples;
  int tupleSize;
  int numDom;
  int numObj;
  int limNumber;
  bool * tupleList;
  int * tupleIgnores;
  bool invalid;
} ValSolution;

void * safeAlloc(int size);
void * safeReAlloc(void* ptr, int size);

#endif /* LOGICINCLUDES_H_ */
```

logicIncludes.c

```
#include "logicIncludes.h"

void * safeAlloc(int size)
{
  void * ret = malloc(size);
  if (ret == 0)
  {
    printf("Heap Space Exhausted");
    exit(EXIT_SUCCESS);
  }
  return ret;
}

void * safeReAlloc(void * ptr,int size)
{
  void * ret = realloc(ptr, size);
  if (ret == 0)
  {
    printf("Heap Space Exhausted");
    exit(EXIT_SUCCESS);
  }
  return ret;
}
```

solvers.h

```
#ifndef SOLVERS_H_
#define SOLVERS_H_

#include "logicIncludes.h"

Solution solveSAT(ClauseList c, int method);
void freeSolution(Solution s);
Solution solSetup(ClauseList c);
Solution cloneSol (Solution s);
Solution solveUnitProp(ClauseList clauses);
void runUnitCheck(ClauseList c, Solution *s);
void safeRunUnitCheck(ClauseList c, Solution *s);
Solution solveFailedLiteral(ClauseList clauses);
void outputDebugVars(Solution s);
Solution solveBinFailedLiteral(ClauseList clauses);
Solution solveSimpleBacktracking(ClauseList c);
Solution solveHornBacktracking(ClauseList c);
Solution solve2SATBacktracking(ClauseList c);
ClauseList condence(ClauseList c, Solution *s);
Solution solveSmartHornBacktracking(ClauseList c);
Solution solveSmart2SATBacktracking(ClauseList c);
int * getStats(ClauseList * org);

#endif /* SOLVERS_H_ */
```

solvers.c

```
#include "solvers.h"
#include "clauseMaker.h"
#include "backtracker.h"
#include "clauseOutputter.h"
#include "timer.h"

int stats [4];
const float DROP_TIME = 120000.0;

Solution solveSAT(ClauseList c, int method)
{
  switch(method)
  {
  case 1: return solveUnitProp(c);
  case 2: return solveFailedLiteral(c);
  case 3: return solveBinFailedLiteral(c);
  case 4: return solveSimpleBacktracking(c);
  case 5: return solveHornBacktracking(c);
  case 6: return solve2SATBacktracking(c);
  case 7: return solveSmartHornBacktracking(c);
  case 8: return solveSmart2SATBacktracking(c);
  default: puts("This algorithm not yet implemented."); exit(EXIT_SUCCESS);
  }
}

void freeSolution(Solution s)
{
  free(s.assignments);
  free(s.skipRecord);
}

Solution solSetup(ClauseList c)
{
  Solution ret;
  ret.impossible = false;
  ret.numVars = c.numVars;

  ret.solved = false;

  ret.assignments = safeAlloc(ret.numVars*sizeof(short));

  int i;
  for (i = 0; i < ret.numVars; i++)
```

```
  {
    ret.assignments[i] = 0;
  }

  ret.numSkips = c.numSpots;
  ret.skipRecord = safeAlloc((c.numSpots+1) *sizeof(int));
  for (i = 0; i <= c.numSpots; i++)
  {
    ret.skipRecord[i] = 0;
  }

  return ret;
}

Solution cloneSol (Solution s)
{
  Solution ret;
  ret.impossible = s.impossible;
  ret.numVars = s.numVars;

  ret.solved = s.solved;

  ret.assignments = safeAlloc(ret.numVars*sizeof(short));

  int i;
  for (i = 0; i < ret.numVars; i++)
  {
    ret.assignments[i] = s.assignments[i];
  }

  ret.numSkips = s.numSkips;
  ret.skipRecord = safeAlloc((s.numSkips+1) *sizeof(int));
  for (i = 0; i <= ret.numSkips; i++)
  {
    ret.skipRecord[i] = s.skipRecord[i];
  }
  return ret;
}

int * getStats(ClauseList * org)
{
  Solution ret = solSetup(*org);

  runUnitCheck(*org, &ret);
```

```
stats[0] = org->numClauses;
stats[1] = org->numLiterals;

if ((ret.impossible) || (ret.solved))
{
  stats[2] = 0;
  stats[3] = 0;
  freeSolution(ret);
  return stats;
}

int itr = 0;
int clauses = 0;
int literals = 0;
bool sated = false;

while (itr < org->numSpots)
{
  if (org->clauseNumbers[itr] == -1)
  {
    if (!sated)
    {
      clauses++;
    }
    else
    {
      sated = false;
    }
  }
  else
  {
    if (ret.assignments[org->clauseNumbers[itr]] == 0)
    {
      literals++;
    }
    else if (ret.assignments[org->clauseNumbers[itr]] == 1)
    {
      if (org->clauseVals[itr] == true)
      {
        sated = true;
      }
    }
    else
    {
      if (org->clauseVals[itr] == false)
```

```
        {
          sated = true;
        }
      }
    }
    itr++;
  }

  stats[2] = clauses;
  stats[3] = literals;

  freeSolution(ret);
  return stats;
}

Solution solveUnitProp(ClauseList clauses)
{
  Solution ret = solSetup(clauses);

  runUnitCheck(clauses, &ret);
  return ret;
}

Solution solveFailedLiteral(ClauseList oc)
{
  Solution ret = solSetup(oc);

  runUnitCheck(oc, &ret);

  if (ret.impossible)
  {
    return ret;
  }
  if (ret.solved)
  {
    return ret;
  }

  ClauseList clauses = condence(oc,&ret);

  int itr = 0;
  bool progress = true;
  while (progress)
  {
    itr = 0;
```

```
progress = false;
while (itr < ret.numVars)
{
  if (timerGet() > DROP_TIME){ return ret;}//STOP if taking more than 2 minutes
  if (ret.assignments[itr] == 0)
  {
    Solution tryout = cloneSol(ret);

    tryout.assignments[itr] = 1;

    runUnitCheck(clauses, &tryout);

    if (tryout.impossible)
    {
      freeSolution(tryout);
      progress = true;
      ret.assignments[itr] = -1;
      runUnitCheck(clauses, &ret);
      if (ret.impossible)
      {
        clauseFree(clauses);
        return ret;
      }
      //itr++;
    }
    else
    {
      Solution tryout2 = cloneSol(ret);
      tryout2.assignments[itr] = -1;
      runUnitCheck(clauses, &tryout2);
      if (tryout2.impossible)
      {
        progress = true;
        freeSolution(tryout2);
        freeSolution(ret);
        ret = tryout;
      }
      else
      {
        freeSolution(tryout);
        freeSolution(tryout2);
      }

      //itr++;
    }
```

```
      }
      //else
      //{
        itr++;
      //}
    }
  }

  clauseFree(clauses);

  return ret;
}

Solution solveBinFailedLiteral(ClauseList oc)
{
  Solution ret = solveFailedLiteral(oc);
  if (ret.impossible)
  {
    return ret;
  }
  if (ret.solved)
  {
    return ret;
  }

  free(ret.skipRecord);
  ret.numSkips = oc.numSpots;
  ret.skipRecord = safeAlloc((ret.numSkips+1) * sizeof(int));
  int qq;
  for (qq = 0; qq < ret.numSkips+1; qq++)
  {
    ret.skipRecord[qq] = 0;
  }
  runUnitCheck(oc, &ret);

  ClauseList c = condence(oc,&ret);

  ret.numSkips *= 4;
  ret.skipRecord = safeReAlloc(ret.skipRecord, (ret.numSkips+1) * sizeof(int));

  for (qq = 0; qq < ret.numSkips+1; qq++)
  {
    ret.skipRecord[qq] = 0;
  }
```

```
c.clauseNumbers = safeReAlloc(c.clauseNumbers, (c.numSpots*4)*sizeof(int));
c.clauseVals = safeReAlloc(c.clauseVals, (c.numSpots*4)*sizeof(bool));
int itr = 0;
bool progress = true;
while (progress)
{
  itr = 0;
  progress = false;
  while (itr < ret.numVars)
  {
    if (timerGet() > DROP_TIME){ return ret;}//STOP if taking more than 2 minutes
    if (ret.assignments[itr] == 0)
    {
      Solution tryout = cloneSol(ret);

      tryout.assignments[itr] = 1;
      safeRunUnitCheck(c, &tryout);

      if (tryout.impossible)
      {
        freeSolution(tryout);
        progress = true;
        ret.assignments[itr] = -1;
        safeRunUnitCheck(c, &ret);
        if (ret.impossible)
        {
          clauseFree(c);
          return ret;
        }
        //itr++;
      }
      else
      {
        Solution tryout2 = cloneSol(ret);
        tryout2.assignments[itr] = -1;
        safeRunUnitCheck(c, &tryout2);
        if (tryout2.impossible)
        {
          progress = true;
          freeSolution(tryout2);
          freeSolution(ret);
          ret = tryout;
        }
        else
        {
```

```
int q ;
for (q = itr+1; q < c.numVars; q++)
{
  if (ret.assignments[q] == 0)
  {
    Solution * inUse = &tryout;
    int blah;
    for (blah = 0; blah < 2; blah++)
    {
      if (inUse->assignments[q] == 0)
      {
        //try q both true and false, if fail, add the binary
        Solution tryout3 = cloneSol(*inUse);

        tryout3.assignments[q] = 1;
        safeRunUnitCheck(c, &tryout3);
        if (tryout3.impossible)
        {
          progress = true;

          c.clauseNumbers[c.numSpots] = itr;
          c.clauseNumbers[c.numSpots+1] = q;
          c.clauseNumbers[c.numSpots+2] = -1;

          c.clauseVals[c.numSpots] = blah;
          c.clauseVals[c.numSpots+1] = 0;
          c.numSpots += 3;
        }

        freeSolution(tryout3);

        tryout3 = cloneSol(*inUse);

        tryout3.assignments[q] = -1;
        safeRunUnitCheck(c, &tryout3);
        if (tryout3.impossible)
        {
          progress = true;

          c.clauseNumbers[c.numSpots] = itr;
          c.clauseNumbers[c.numSpots+1] = q;
          c.clauseNumbers[c.numSpots+2] = -1;

          c.clauseVals[c.numSpots] = blah;
          c.clauseVals[c.numSpots+1] = 1;
```

```
                    c.numSpots += 3;
                  }
                  freeSolution(tryout3);
                }
                inUse = &tryout2;
              }
            }
          }
          freeSolution(tryout);
          freeSolution(tryout2);
        }
      }
    }
    itr++;
  }
}

  clauseFree(c);
  return ret;
}

void safeRunUnitCheck(ClauseList c, Solution *s)
{
  if (s->numSkips < c.numSpots)
  {
    s->numSkips = c.numSpots;
  }
  runUnitCheck(c,s);
}

void runUnitCheck(ClauseList c, Solution *s)
{
  int itr = 0;
  int numOutstanding = 0;
  bool sated = false;
  int varOutstanding;
  bool valOutstanding;
  bool clauseUnsated = false;
  int startSpot = 0;

  while (itr < c.numSpots)
  {
    itr += s->skipRecord[itr];
    if (itr < c.numSpots)
    {
```

```
if (c.clauseNumbers[itr] == -1)
{
  if ((numOutstanding == 1) && (sated == false))
  {
    if (valOutstanding)
    {
      s->assignments[varOutstanding] = 1;
    }
    else
    {
      s->assignments[varOutstanding] = -1;
    }

    s->skipRecord[startSpot] = (itr - startSpot) + 1 + s->skipRecord[itr+1];

    startSpot = 0;
    itr = -1;
    clauseUnsated = false;
  }
  else if ((numOutstanding == 0) && (sated == false))
  {
    s->impossible = true;
    return;
  }
  else if (sated == false)
  {
    clauseUnsated = true;
    startSpot = itr + 1;
  }
  else
  {
    s->skipRecord[startSpot] = (itr - startSpot) + 1 + s->skipRecord[itr+1];
  }
  numOutstanding = 0;
  sated = false;
}
else
{
  if (s->assignments[c.clauseNumbers[itr]] == 0)
  {
    numOutstanding++;
    varOutstanding = c.clauseNumbers[itr];
    valOutstanding = c.clauseVals[itr];
  }
  else if (s->assignments[c.clauseNumbers[itr]] == -1)
```

```
        {
          if (c.clauseVals[itr] == false)
          {
            sated = true;
          }
        }
        else if (s->assignments[c.clauseNumbers[itr]] == 1)
        {
          if (c.clauseVals[itr] == true)
          {
            sated = true;
          }
        }
        else
        {
          printf("Bizarre Error in Unit Prop. Should not happen.");
          exit(EXIT_SUCCESS);
        }
      }

      itr++;
    }
  }

  if (clauseUnsated == false)
  {
    s->solved = true;
  }
}

Solution solveSimpleBacktracking(ClauseList c)
{
  return backtrack(c, 0);
}

Solution solveHornBacktracking(ClauseList c)
{
  return backtrack(c, 1);
}

Solution solve2SATBacktracking(ClauseList c)
{
  return backtrack(c, 2);
}
```

```
Solution solveSmartHornBacktracking(ClauseList c)
{
  return backtrack(c, 3);
}

Solution solveSmart2SATBacktracking(ClauseList c)
{
  return backtrack(c, 4);
}

ClauseList condence(ClauseList c, Solution *s)
{
  ClauseList ret;
  int itr1 = 0;
  int itr = 0;
  itr += s->skipRecord[itr];
  while (itr < c.numSpots)
  {
    itr1++;
    itr++;
    itr += s->skipRecord[itr];
  }

  ret.numSpots = itr1;
  ret.numClauses = 0;
  ret.numLiterals = 0;
  ret.numVars = c.numVars;

  ret.clauseNumbers = safeAlloc(ret.numSpots * sizeof(int));
  ret.clauseVals = safeAlloc(ret.numSpots * sizeof(bool));

  itr1 = 0;
  itr = 0;
  itr += s->skipRecord[itr];
  while (itr < c.numSpots)
  {
    ret.clauseNumbers[itr1] = c.clauseNumbers[itr];
    ret.clauseVals[itr1] = c.clauseVals[itr];
    itr1++;
    itr++;
    itr += s->skipRecord[itr];
  }
  free(s->skipRecord);
  s->skipRecord = safeAlloc((ret.numSpots+1) * sizeof(int));
  for (itr = 0; itr <= ret.numSpots; itr++)
```

```
  {
    s->skipRecord[itr] = 0;
  }
  s->numSkips = ret.numSpots;

  return ret;
}
```

specialSAT.h

```
#ifndef SPECIALSAT_H_
#define SPECIALSAT_H_

#include "logicIncludes.h"

void hornSolve(ClauseList c, Solution * ret);
void SAT2resolve(ClauseList c, Solution * ret);

#endif /* SPECIALSAT_H_ */
```

specialSAT.c

```
#include "specialSAT.h"
#include "solvers.h"

void hornSolve(ClauseList c, Solution * ret)
{
  //Of course, if there are no singles,
  //and everything is a horn clause, then setting everything to negatives is fine.
  int i;
  for (i = 0; i<ret->numVars; i++)
  {
    if (ret->assignments[i] == 0)
    {
      ret->assignments[i] = -1;
    }
  }
  ret->solved = true;
}

void SAT2resolve(ClauseList c, Solution * ret)
{
  //ASSUMES 2-SAT
  int itr = 0;

  while (itr < ret->numVars)
  {
    if (ret->assignments[itr] == 0)
    {
      Solution tryout = cloneSol(*ret);

      tryout.assignments[itr] = 1;

      runUnitCheck(c, &tryout);

      if (tryout.impossible)
      {
        freeSolution(tryout);
        ret->assignments[itr] = -1;
        runUnitCheck(c, ret);
        if (ret->solved)
        {
          return;
        }
        if (ret->impossible)
```

```
        {
          return;
        }
      }
      else
      {
        freeSolution(*ret);
        *ret = tryout;
        if (ret->solved)
        {
          return;
        }
      }
    }
    itr++;
  }
}
```

timer.h

```
#ifndef TIMER_H_
#define TIMER_H_

#include "logicIncludes.h"

void timerSet();
float timerGet();

#endif /* TIMER_H_ */
```

timer.c

```
#include "timer.h"

unsigned long long setTime;

void timerSet()
{
  setTime = (unsigned long long)clock();
}

float timerGet()
{
  unsigned long long temp = ((unsigned long long)clock() - setTime) * 1000;
  return ((double)temp)/ CLOCKS_PER_SEC;
}
```