# Comparing Performance of Logic Programming Languages and Solvers With Peg Solitaire

Michael Casolary
Fall 2008
Dr. Joohyung Lee
CSE 598

The game of peg solitaire is one of a handful of well-known yet still challenging puzzles for humans to solve. The basic premise of the game is you are given a board which has several holes arranged in a certain pattern, with pegs placed in every hole except for one. A peg may "jump" another peg by moving from one side of the "jumped" peg to an empty hole immediately on the opposite side of the peg (horizontally or vertically). The "jumped" peg is then removed from the board, creating an additional hole. The goal of the game is to jump pegs in an appropriate manner so that when no valid jumps remain you are left with exactly one peg on the board. Often, an additional challenge in the puzzle description specifies that the final peg must end at a certain position (e.g., the center of the board). There are three "standard" layouts of peg solitaire boards, the "English", "European", and "triangle" layouts (see figures below).

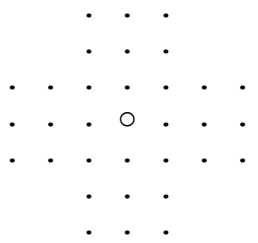

Fig. 1: English board (○ indicates initial hole)

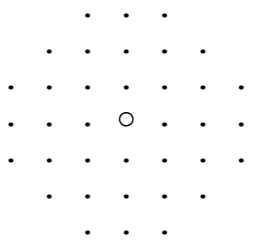

Fig. 2: European board (○ indicates initial hole)

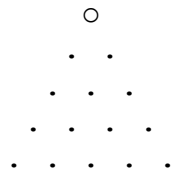

Fig. 3: Triangle board (○ indicates initial hole)

This puzzle and several of its common variations have been extensively studied for their mathematical and combinatorial properties. Of typical interest are the kinds of solutions a given board and/or set of rules have, methods for deriving or arriving at those solutions, and the computational size of the problem (possible states of the board, methods for determining the efficiency of a solution, etc.).

For ease of definition, in this paper the term simple solution means a solution to a given peg solitaire board where at the final stage of said solution there remains only one peg on the board. This is analogous to the basic victory conditions laid out in most presentations of peg solitaire. A true solution is defined as a simple solution where the final peg lands in the initial blank space of a given board (e.g., the center hole for the English and European boards). Other works have demonstrated that the English board and the Triangle board both have true solutions in addition to simple solutions, but the European board does not have a true solution. Whether this was a purposeful trick played on early enthusiasts of the game or an unintentional oversight on the part of the board designers remains a subject of speculation.

At a high level, peg solitaire can be described as a board, rules on how to move (and remove) pegs, and conditions for finding a solution. The formalism of peg solitaire in ASP mimics this categorization by being split into what could be considered four groups: a set of rules defining the size, shape, and initial layout of a game board, a set of rules that generate valid ways to move pegs around the board (i.e., jumping rules), a set of rules defining how pegs are removed from the board, and a set of rules that define the conditions under which a given board would be considered solved. For the sake of ease

of use, the rules governing jumping and removing pegs were decoupled as much as possible from the rules defining the initial board and victory conditions.

In the first attempt to formalize peg solitaire, the jumping rules were generated by creating a set of predicates defining peg movements from every valid space on the board to any valid space on the board. These were then restricted so that at each time step only one peg could be moved to one place, and then a handful of constraints were used to eliminate jumping movements that weren't allowed by the rules (e.g., by default, diagonal jumping is not allowed) or that didn't make sense (e.g., moving a peg onto another peg). A handful of rules defined what it meant to be "jumped" at any given time point, and this definition was used to determine when and how pegs could jump and what happened to pegs that were jumped. By default, these rules supported orthogonal jumping movements of distance 2 (i.e., jumping in a straight line over one peg).

This initial formalization presented several problems that eventually required a redesigning of the formalism to correct the issues. First, this early formalization was very inefficient due to the fact that it generated predicates for movement from every space on the board to all board spaces at every time step, when by the standard rules of peg solitaire the set of possible movement patterns on a given board is actually far smaller than that. Second, none of the rules or constraints in the original definition were mutable. This meant that changing the rules for a particular board setup (e.g., the Triangle board allowing additional ways to jump) would often prove impossible, as the constraints set up to define movement patters almost always clashed with whatever new jumping rules one attempted to use. Finally, even if new jumping rules didn't clash with the default ones, the rules defining being jumped and what that meant for the peg(s) that were jumped were tightly coupled to the rules about jumping, making it very difficult to add new jumping rules.

Due to the problems listed above, it was decided that a redesign of the formalism was needed to make it more efficient, properly elaboration tolerant, and capable of supporting a wider variety of game boards, rules, and conditions. The definitions of game boards and victory conditions could largely be left alone, it was mostly the rules about movement and jumping that needed tweaking.

For starters, instead of using one rule to generate a large set of movement possibilities that were then pared down later, the new formalization uses a set of mutable rules that slowly generate portions of the movement space based on specific rules and restrictions. This method of movement space generation is much easier to work with for several reasons. First, it drastically reduces the number of predicates generated, making solutions smaller and faster to compute. Second, it removes the need to have a series of constraints defining valid movement according to the given rules, which means that changing or adding movement patterns is easier because there are fewer rules and constraints to add. Third, it means that modifying the rules governing movement is much simpler: to add a new way to move, all that needs to be done is to add rules that generate those movement possibilities, and to change the default way to move, one only has to disable the default rules and then replace them as desired.

In a similar vein, the rules governing jumping are mutable and as decoupled from the rest of the rules as possible, making it easier to add to them (the typical other half of adding new movement rules in peg solitaire) or to modify them.

The major improvement with the new formalization was to reduce the number of constraints and streamline and simplify their definitions. Before, there were many constraints that were controlling movement and jumping, and over half of them were tightly coupled to the particular rules for movement and jumping they were controlling. With the new formalization, much of the heavy lifting is now done by the new rules themselves, meaning the number of constraints gets cut down quite a bit. Also, by limiting the generation rules themselves, the remaining constraints now are very generic, and apply to essentially any instance of peg solitaire. For instance, with the old formalization, the constraint forcing a peg to be jumped with every step used the rules governing movement to decide if something was jumped or not. Now, that constraint simply uses the definition of being jumped also defined in the rules to basically make the statement, "The predicate jumped has to be defined for something at each time step," which means that the constraint is totally decoupled from *how* one defines what it means to be jumped. This improvement is reflected in several of the other constraints, making them decoupled from any board-specific definitions, instead functioning as generic rules of play (as they were originally intended to be).

The definition of a given board size, shape, configuration, and solution conditions remained essentially unchanged across both versions, and so will be addressed here. In terms of the ASP formalization, a board can be thought of as a subset of a particular rectangular portion of a Cartesian grid. A board's size is given by constants that define valid ranges of coordinates for pegs and holes. Its shape and initial configuration are determined by defining predicates of peg positions at the initial time step, and by defining predicates that essentially state, "No peg may ever be at this position." This last set of predicates allows for non-rectangular boards (e.g., the English board's cross shape). Victory conditions, if any, can govern where pegs must end up, which pegs must jump where, and other similar restrictions. By default, the basic formalization stipulates that there must be one peg left (assuming one sets the time step parameters properly), so if a simple solution is all that is desired, no additional constraints or rules need to be added.

For boards such as the Triangle board where there additional rules about movement and jumping need to be added before the puzzle can be solved, one may simply define the predicates that "disable" the default movement and jumping rules (if necessary), and then define custom rules for movement and jumping. In the Triangle board's case, the default rules simply needed to be extended to support an additional way of jumping, so the default rules were left in place, and new rules governing movement and jumping were added. As a result of this flexibility, essentially any board that can be somehow represented on a Cartesian grid can be solved using this formalization.

Initially, the plan was to solve the English board using the "default" ASP setup of lparse feeding data into smodels, and then using that data to benchmark other solving systems

against smodels' performance. The European board was determined to be an unsuitable candidate, for one because of its larger search space and solution length, and for another because it lacks a true solution, making it a much less interesting board to examine and use for comparison. However, as is often the case with such setups, ambition exceeded capability. Attempting to solve the English board (even after making the formalization more efficient using the techniques described earlier) proved to be an exercise in futility, as smodels failed to generate even a simple solution for the English board after being run for the better part of a full day on fairly modern hardware. The formalization is sound, so most likely given enough time on powerful enough hardware a solution would be found, however considering that few solvers could be expected to outperform smodels by orders of magnitude (at least without prior evidence suggesting this) and some of the systems that would be benchmarked were rumored to be slower than smodels, it was an obvious decision to abandon the English board and seek a smaller, more tractable board to use as a benchmarking tool. Though the naïve set of possible states of the English board is close to 20 million, probably the main problem encountered when trying to solve the English board comes from the fact that any solution will be 31 steps long. In the past, working with other puzzles involving multiple solution steps in a changing environment demonstrated that the difficulty of finding a solution and time required to find said solution(s) was directly proportional to the size of the problem (i.e., the search space) and the number of steps required to find a solution. Thus, a problem of the size of the English board is definitely in the category of "difficult to solve in any reasonable time frame".

The English board thus abandoned, attention was focused on the final "canonical" board, the Triangle board. It has the distinct advantages of being far smaller than both the English and European boards, and any solution to the Triangle board would only be 13 steps long, almost two thirds shorter than the other two boards. Since computational complexity in terms of answer set solvers tends to be exponential in nature, this meant there was a good chance that the smaller Triangle board could actually be solved.

For benchmarking purposes, all tests were run on the same machine under as similar of conditions as possible. The machine had a 2.4 GHz Pentium 4 processor, 1 GB of RAM, and was running all of the following software using Cygwin on Windows XP. Initially as a control, smodels was used to generate a simple solution and a true solution for the Triangle board. smodels was able to generate a simple solution in 11 seconds, and generated a true solution in 23 minutes. After confirming that the Triangle board was solvable in a reasonable time frame, cmodels was introduced into the mix. cmodels uses SAT solvers to compute solutions to ASP programs, and of the several SAT solvers that cmodels supports, relsat, MiniSat, zChaff, and Simo were used in this benchmark.

Using cmodels with relsat, a simple solution to the Triangle board was found in 18 seconds, and a true solution was found after 24 minutes, which was very similar in performance to smodels. The similarities end with zChaff, which computed a simple solution in 3 seconds and a true solution in 5 seconds. Continuing the trend of blowing relsat and smodels out of the water, MiniSat found a simple solution in only 1.5 seconds, and needed just 3 seconds to find a true solution to the Triangle board. Unfortunately, to make up for such astounding performance, Simo couldn't even find a single simple

solution for the Triangle board. For that matter, even when fed an insultingly simple version of the Triangle board that only took 5 steps to solve, it failed to find a solution even after running for over half an hour. Examining the configuration cmodels uses to call Simo, one possible explanation was that cmodels wasn't passing Simo the right options, and so its learning algorithm was getting stunted. However, after tweaking the parameters and making a best effort attempt at optimizing Simo's performance, it still failed to find a solution to the insultingly easy Triangle board, which suggests that Simo may simply not be an effective solver for this domain.

One problem with all of the SAT solvers used with cmodels was that they were all deterministic solvers. These all shared an algorithmic design that they would methodically search through the entire set of all possible solutions for a given problem in order to find a particular solution. The advantage of this approach is that if solution(s) exist, a deterministic solver will find any solution(s), and is capable of finding all of them if more than one exists. The primary disadvantage with such systems is that their exhaustive search of the solution space can lead down many dead-end paths, meaning that finding solutions can be a slow and painful process. Another approach is to use a stochastic SAT solver, which randomly wanders through a problem's solution space, non-deterministically changing course in an attempt to improve the quality of its current solution and find a valid solution essentially by chance. This bears the advantage that often a stochastic solver will find a solution much faster than a deterministic solver, but comes with the hazard that nondeterministic wandering may not find all solutions to a problem or possibly even fail to find any solution when several may have existed.

In an attempt to bring stochastic SAT solvers into the benchmarking mix, ASSAT was suggested as a possible solving tool. It, like cmodels, uses SAT solvers at the core of its operations, but ASSAT supposedly works with a wider variety of SAT solvers than cmodels, including stochastic solvers like WalkSAT. Unfortunately, ASSAT is an older answer set solver that was not updated to support newer language constructs used by the peg solitaire formalization (such as choice formulas), which meant that trying to get ASSAT to work with the formalization was a pointless and unavoidable exercise in failure and frustration.

In the end, the overall winner in terms of solving speed was definitely cmodels with MiniSat, with zChaff coming in at a close second. The astounding speed with which both solvers managed to find solutions of both kinds for the Triangle board was nothing short of amazing. Most interesting was that the time difference between a simple solution and a true solution for MiniSat and zChaff was only a factor of about 2, whereas the time difference between those solutions for relsat and smodels was closer to a factor of 100. Of all the answer set solvers tested, given its outstanding performance MiniSat would be most likely to produce a solution to the English board before causing the death of the machine it's using to solve the puzzle.

Though this was the first documented attempt to formalize peg solitaire in ASP, other formalizations exist that use different systems to compute solutions. The two most common are solutions written in Java and in C++. In both cases, these formalizations

encode the description of the board, the rules governing movement, jumping, and solving the puzzle, and an algorithm used to compute solutions directly into the program's code. From the solutions examined, the clear advantage of this approach was speed. Java and C++ are both relatively fast languages (in terms of efficiency of instruction execution), and this coupled with solution algorithms specifically engineered to solve instances of peg solitaire made these formalizations blazing fast. Where ASP couldn't even compute a solution to the English board in any reasonable time frame, C++ and Java formalizations were able to do so in usually only a handful of seconds. Thus, in terms of time to find a solution, the "classic" formalizations beat out any ASP solver every time.

That having been said, the C++ and Java formalizations had some major flaws. The worst of these was that in designing a specific solution algorithm, most of the formalizations limited themselves to solving only one board (sometimes in only one configuration). The most flexible formalizations examined were capable of handling pegs in arbitrary number and positions on a given board, but even those could not handle changing the board size or shape (even English to European or vice-versa in some cases). Continuing the fixed solution trend, none of these formalizations were easily modifiable to change the rules of movement, jumping, or solving the puzzle. Indeed, their algorithms depended upon these rules not changing, to the point that a simple search did not turn up a single C++ or Java formalization that could handle both an English board and a Triangle board with the same algorithm.

From the standpoint of ease of use and development, the C++ and Java formalizations presented another problem. Since their lengthy solution algorithms were encoded alongside the definition of the puzzle, it made understanding the inner workings of the formalization very difficult. By comparison, the "default" rules for peg solitaire take up less than 15 lines of ASP code, making deciphering their content and purpose far easier. This also made it easier to change the rules of peg solitaire in the ASP formalization, since a handful of lines of code could completely redefine the rules of movement or jumping. In the Triangle board case, all that was needed was four lines of ASP code to add rules for new ways to move and jump, and it was done and ready to go. What's more, because these extra rules resided in a separate file from the original definition, the "default" rules remained untouched and could still be used to compute solutions for versions of peg solitaire similar to the English board without any tweaking or retooling of the "default" rules. Thus, though the C++ and Java formalizations definitely emerge the victor in terms of speed of computation, the ASP formalization most certainly wins in the category of ease of use and flexibility.

After running the ASP solvers through the gauntlet and optimizing the ASP formalization of peg solitaire, another possibility presented itself in the form of formalizing peg solitaire in C+ (another logic programming language) to provide cross-comparison capabilities. C+ is a causal logic language with second-order logic capabilities that was designed to reason about actions. It provides high-level language constructs with the goal of making it easier to define logic problems in a language more human readable than other languages. C+'s primary solver, CCalc, uses many of the same SAT solvers that cmodels does to compute solutions, so it appeared to be an interesting and worthwhile

exercise to translate the problem into the language of C+ and then examine its performance versus ASP using the same SAT solvers. This theoretically would directly demonstrate which language was more efficient in generating its low-level formalizations. In addition, once translated, it would be interesting to compare the two formalizations to see how different rules were represented in both languages.

After some work brushing up on C+ and smoothing over the idiosyncrasies of each language, a translation of the peg solitaire formalization was completed in CCalc that produced the same solutions (after a fashion) as the ASP formalization. Fortunately, much of the original formalization translated over with little change. Most of the predicates remained the same (there were still predicates for jumping, for being jumped, and predicates describing where pegs were, weren't, and couldn't be), and most of the constraints made the transition largely unchanged but for minor syntactic differences. However, some things did not carry over as easily. Since C+ actions (movement, in our case) are defined as general actions and are then given rules defining when that action can take place, the rules that slowly built valid jumping movements in ASP had to be thrown out and replaced with a set of constraints more closely resembling the original set of movement constraints used in the first version of the ASP formalization. However, thanks to the way CCalc grounds variables, a few tweaks could be made to make the constraints easier to understand and fewer in number. In addition, since C+ doesn't have as rich of an ability to define predicate set sizes as ASP, the constraint about jumping had to change in the translation. In ASP, there were simply rules that defined what it meant to be jumped, and then a constraint stating that at least one peg had to be considered jumped at each time step. In C+, the constraint had to change to state that there must be at least one peg that was considered jumped each time a peg moved.

The most drastic change was in the rules governing the victory conditions. In ASP, it was simply enough to state that (after a fashion) a peg must be removed from the board each step and that a solution would have a certain number of steps. This would automatically generate a simple solution to any remotely normal peg solitaire board. Adding a constraint governing the position of the final peg was all that was needed to generate a true solution. In C+, it proved to be very difficult to encode the statement "A peg must move and jump another peg each time step," as forcing an action to happen is treated differently than forcing a fluent to be true. Therefore, an additional set of fluents and rules were created that described how many pegs were remaining on the board, and that when a peg is jumped that number decreases. With that in place, the new "default" victory condition became that the number of pegs left on the board was exactly 1. This goaded CCalc into solving the puzzle, because the only way to reduce the number of pegs was to jump pegs according to the rules.

With the translation complete, C+ appears to just barely emerge as the winner over ASP in terms of readability and understandability of the formalization. After ignoring the "setup" portion of the description where all the fluents and actions are defined, that C+ tends to use more plain English in its description ("x causes y", "constraint p after q", etc.) seems to make it slightly easier for a layperson to read. That said, the additional complexity of the varying relationship among actions, fluents, and their effect (or lack

thereof) on future time steps made developing the CCalc formalization trickier than ASP, because there was more "hidden semantics" in CCalc than there was in ASP. However, some of that difficulty could be attributed to there being no quick reference sheet or formal tutorial for CCalc and that CCalc has a nasty tendency to return cryptic and unintuitive errors if there is a problem with the syntax or semantics of a given formalization.

For evaluation of performance, CCalc was hooked up to the SAT solvers relsat, MiniSat, and zChaff. Though CCalc does support WalkSAT, an issue arose that caused any call from CCalc to WalkSAT to fail, and it appeared to be a problem of a disconnection between the format of data CCalc was preparing and the format of data WalkSAT expected. With all of the functioning SAT solvers, CCalc required about three and a half minutes to ground all the variables for the Triangle board and prepare a low-level formalization for the SAT solvers. Using relsat turned out to be a bad idea, as it could not even compute a simple solution to the Triangle board in a reasonable time frame (though unlike Simo, it could solve the insultingly simple version of the Triangle board after a few seconds). zChaff fared slightly better, but still had unexpected trouble computing solutions, taking 22 minutes to find a simple solution to the Triangle board, and 18 minutes to find a true solution. MiniSat was the again the star performer, finding a simple solution to the Triangle board in 4 seconds, and a true solution in 15 seconds. Thus, the overall "winner" for C+ was MiniSat by a long shot.

Compared to the SAT solver performance of cmodels with ASP, CCalc was overall slightly slower than ASP, and thus ASP wins in terms of speed of computation. These results were not unexpected, as CCalc uses a higher-level language than ASP, and so would be expected to have a greater high-level to low-level conversion penalty. The main mystery of the CCalc to ASP comparison is why MiniSat was able to weather the translation with little slowdown, while zChaff and relsat took respective nose dives. Of particular interest is the fact that it actually took zChaff *longer* to compute a simple solution to the Triangle board than a true solution, something that should never happen since there are many more simple solutions than there are true solutions, and every true solution is a simple solution.

There are a few areas where further research and investigation may yield interesting results. The CCalc formalization created for this paper was largely an afterthought, done as a proof of concept and a way to pass the time while waiting for the ASP solvers to produce their results. As a result, the translation of peg solitaire to C+, while functional and about equally as capable as the ASP formalization, still has some rough edges related to issues translating some aspects of ASP smoothly into C+. Further effort to eliminate these workarounds and hacks would be potentially useful for revealing some of the more subtle differences and similarities between ASP and C+, and possibly help form a body of work that could be used to demonstrate methods of translating ASP formalizations to C+ (or perhaps the reverse). Also, there are likely inefficiencies in the CCalc formalization that could be reduced or eliminated with some further tweaking, potentially improving the performance of that language and its solvers with respect to peg solitaire solutions.

As another topic of further interest, both ASP and C+ have solver systems that supposedly work with stochastic SAT solvers, and though it proved to be too difficult and annoying to get them working for this paper, successful integration of stochastic SAT solvers into the benchmarks would be very interesting. In the past, stochastic SAT solvers have proven very useful when trying to find solutions to large or long problems, their only weakness being their tendency to give up if a solution can't be found quickly enough. Comparing the speed of a stochastic solver at solving peg solitaire and determining how many "hints" must be given to such solvers before they can find solutions to given peg solitaire boards would be an interesting area of possible research.

**References:**


Bayardo, Roberto. "relsat." Google Code. Accessed December 2008. http://code.google.com/p/relsat/

Beasley, John (ed). Games and Puzzles Journal #28. http://www.gpj.connectfree.co.uk/gpjj.htm

"DSA555 - Peg jump / solitaire solver." Arek Zen. Accessed December 2008. http://www.zenebo.com/word/c-and-c-plus-plus/dsa555-peg-jump-solitaire-solver/

Niklas Eén and Niklas Sörensson. "MiniSat Page." Accessed December 2008. http://minisat.se/

Enrico Giunchiglia, *et al*. "Satisfiability Internal Module Object oriented." Accessed December 2008. http://www.mrg.dist.unige.it/~sim/simo/

Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, Hudson Turner. "Nonmonotonic Causal Theories." Artificial Intelligence, 2004.

Henry Kautz and Bart Selman. "Stochastic Local Search for Satisfiability." Accessed December 2008. http://www.cs.rochester.edu/u/kautz/walksat/

Fangzhen Lin and Yuting Zhao. "ASSAT: Answer Set by SAT solvers." Accessed December 2008. http://assat.cs.ust.hk/

Sharad Malik, *et al*. "Boolean Satisfiability Research Group at Princeton." Accessed December 2008. http://www.princeton.edu/~chaff/zchaff.html

Lierler, Yulia *et al*. "CMODELS - Answer Set programming System." Accessed December 2008. http://www.cs.utexas.edu/~tag/cmodels/

Vladimir Lifschitz, *et al*. "CCalc." Accessed December 2008. http://www.cs.utexas.edu/~tag/cc/

O'Brien, Dan. "Peg Board Puzzle Solution Page." Accessed December 2008. http://www.danobrien.ws/PegBoard.html

"Peg Solitaire." Wikipedia. Accessed December 20008. http://en.wikipedia.org/wiki/Peg_solitaire

"The Peg Solitaire Puzzle Game." Accessed December 2008. http://blackdog4kids.com/games/puzzles/peggy/index.html

Simons, Patrick. "Computing the Stable Model Semantics." Accessed December 2008. http://www.tcs.hut.fi/Software/smodels/

# Appendix A: Performance Comparison of Solvers

| | Triangle Board, Simple Solution | Triangle Board, True Solution |
|---|---|---|
| **ASP: smodels** | 11 seconds | 23 minutes |
| **ASP: cmodels w/ relsat** | 18 seconds | 24 minutes |
| **ASP: cmodels w/ zChaff** | 3 seconds | 5 seconds |
| **ASP: cmodels w/ MiniSat** | 1.5 seconds | 3 seconds |
| **ASP: cmodels w/ Simo** | -* | -* |
| **C+: CCalc w/ relsat**** | -* | -* |
| **C+: CCalc w/ zChaff**** | 22 minutes | 18 minutes |
| **C+: CCalc w/ MiniSat**** | 4 seconds | 15 seconds |

* Unable to generate solution within time frame of 15 minutes (simple solution) or 1 hour (true solution)
** Does not include average 3.5 minute grounding, completion, and preparation time

# Appendix B: ASP Code Describing Peg Solitaire

## B.1: Peg Solitaire Description, Default Rules

```
% Peg jumping puzzle.
% Use predicate peg(x,y,t) to say a peg exists at x,y at time t.
% Use predicate -peg(x,y,t) to say a space x,y is blank at time t.
% Use predicate nopegs(x,y) to indicate a portion of the board where pegs cannot go (to
support non-square board layouts).
% Predicate jump(X,Y,X1,Y1,T) indicates that the move for time T is to have the peg at
X,Y jump to X1,Y1.
% Predicate jumped(X,Y,T) indicates that at time T some peg jumped over X,Y, eliminating
the peg at X,Y.

xCoord(1..x).
yCoord(1..y).
timePoint(1..t).

#domain xCoord(X;X1), yCoord(Y;Y1), timePoint(T).

% ---JUMP GENERATION---

% At each time step, we should move a peg from P,Q to R,S, jumping a peg in the process.
% Define basic set of valid jumps.  For starters, can only jump in straight lines.
% Mutable with ab(jumpDef).
{jump(P,Q,R,S,T): abs(P-R)==2: Q==S: xCoord(P;R): yCoord(Q;S)} :- not ab(jumpDef).
{jump(P,Q,R,S,T): P==R: abs(Q-S)==2: xCoord(P;R): yCoord(Q;S)} :- not ab(jumpDef).
% There must be one (and only one) jump per time step.
:- 2{jump(P,Q,R,S,T): xCoord(P;R): yCoord(Q;S)}.
:- {jump(P,Q,R,S,T): xCoord(P;R): yCoord(Q;S)}0.

% ---RULES---

% Defined: jumped(P,Q,T) means the peg at P,Q got jumped at time T.
% Mutable with ab(jumpedDef).
jumped(P,Q,T) :- jump(X,Y,X1,Y1,T), abs(X-P)==1, abs(X1-P)==1, Y==Q, Y1==Q, peg(P,Q,T),
xCoord(P), yCoord(Q), not ab(jumpedDef).
jumped(P,Q,T) :- jump(X,Y,X1,Y1,T), abs(Y-Q)==1, abs(Y1-Q)==1, X==P, X1==P, peg(P,Q,T),
xCoord(P), yCoord(Q), not ab(jumpedDef).

% Jumping a peg causes the jumping peg to move from X,Y to X1,Y1
% and causes the jumped peg to disappear.
-peg(X,Y,T+1) :- jump(X,Y,X1,Y1,T).
peg(X1,Y1,T+1) :- jump(X,Y,X1,Y1,T).
% Remove a peg if it got explicitly jumped.
-peg(X,Y,T+1) :- jumped(X,Y,T).

% Inertia: Pegs that don't jump and aren't jumped just stay there.
peg(X,Y,T+1) :- {jump(X,Y,R,S,T): xCoord(R): yCoord(S)}0, not jumped(X,Y,T), peg(X,Y,T).
```

```
% Inertia: Blank spaces stay blank unless a peg jumps in to them.
-peg(X1,Y1,T+1) :- {jump(P,Q,X1,Y1,T): xCoord(P): yCoord(Q)}0, -peg(X1,Y1,T).

% Invalid board locations never have pegs on them.
-peg(X,Y,T) :- nopegs(X,Y).

% ---CONSTRAINTS---

% No jumping with a nonexistent peg.
:- jump(X,Y,X1,Y1,T), -peg(X,Y,T).

% We can't jump to a space already occupied.
:- jump(X,Y,X1,Y1,T), peg(X1,Y1,T).

% We must jump one peg each time step.
:- {jumped(P,Q,T): xCoord(P): yCoord(Q)}0.
:- 2{jumped(P,Q,T): xCoord(P): yCoord(Q)}.

% We can't jump an empty spot.
:- jumped(X,Y,T), -peg(X,Y,T).

% We can't jump somewhere if it isn't a valid place to jump on the board.
:- jump(X,Y,X1,Y1,T), nopegs(X1,Y1).

#hide xCoord(_), yCoord(_), timePoint(_), nopegs(_,_).
#hide peg(_,_,_).
```

## B.2: English Board Description

```
% English board, standard configuration
% Must stop in the center

#const x=7.
#const y=7.
#const t=31.

nopegs(1,1). nopegs(2,1). peg(3,1,1). peg(4,1,1). peg(5,1,1). nopegs(6,1). nopegs(7,1).
nopegs(1,2). nopegs(2,2). peg(3,2,1). peg(4,2,1). peg(5,2,1). nopegs(6,2). nopegs(7,2).
peg(1,3,1). peg(2,3,1). peg(3,3,1). peg(4,3,1). peg(5,3,1). peg(6,3,1). peg(7,3,1).
peg(1,4,1). peg(2,4,1). peg(3,4,1). -peg(4,4,1). peg(5,4,1). peg(6,4,1). peg(7,4,1).
peg(1,5,1). peg(2,5,1). peg(3,5,1). peg(4,5,1). peg(5,5,1). peg(6,5,1). peg(7,5,1).
nopegs(1,6). nopegs(2,6). peg(3,6,1). peg(4,6,1). peg(5,6,1). nopegs(6,6). nopegs(7,6).
nopegs(1,7). nopegs(2,7). peg(3,7,1). peg(4,7,1). peg(5,7,1). nopegs(6,7). nopegs(7,7).

% The final move must end with the last peg landing in the center of the board.
:- not peg(4,4,32).
```

## B.3: Triangle Board Description

```
% The "classic IQ test"
%       X
%       XX
%       XXX
%       XXXX
%       .XXXX

#const x = 5.
#const y = 5.
#const t = 13.

peg(1,1,1). nopegs(2,1). nopegs(3,1). nopegs(4,1). nopegs(5,1).
peg(1,2,1). peg(2,2,1). nopegs(3,2). nopegs(4,2). nopegs(5,2).
peg(1,3,1). peg(2,3,1). peg(3,3,1). nopegs(4,3). nopegs(5,3).
peg(1,4,1). peg(2,4,1). peg(3,4,1). peg(4,4,1). nopegs(5,4).
-peg(1,5,1). peg(2,5,1). peg(3,5,1). peg(4,5,1). peg(5,5,1).

% Define additional valid jump conditions (can jump like a backslash: \, down to right,
or up to left).
{jump(P,Q,R,S,T): R-P==2: S-Q==2: xCoord(P;R): yCoord(Q;S)}.
{jump(P,Q,R,S,T): P-R==2: Q-S==2: xCoord(P;R): yCoord(Q;S)}.
```

```
% Define an additional condition under which a peg counts as being "jumped" (can jump
like a backslash: \, down to right, or up to left).
jumped(P,Q,T) :- jump(X,Y,X1,Y1,T), P-X==1, X1-P==1, Q-Y==1, Y1-Q==1, peg(P,Q,T),
xCoord(P), yCoord(Q).
jumped(P,Q,T) :- jump(X,Y,X1,Y1,T), X-P==1, P-X1==1, Y-Q==1, Q-Y1==1, peg(P,Q,T),
xCoord(P), yCoord(Q).

% Additional victory condition: Final peg must land in the initial hole.
:- not peg(1,5,14).
```

# Appendix C: CCalc Code Describing Peg Solitaire

## C.1: Peg Solitaire Description & Default Rules

```
% Peg Solitaire, base definition.

:- sorts
 xCoord;
 yCoord;
 pegVals.

:- objects
 1..numPegs :: pegVals.

:- variables
 X,X1,P :: xCoord;
 Y,Y1,Q :: yCoord;
 C,C1 :: pegVals.

:- constants
 pegsLeft :: inertialFluent(pegVals);
 peg(xCoord, yCoord) :: inertialFluent;
 nopegs(xCoord, yCoord) :: inertialFluent;

 jumped(xCoord, yCoord) :: simpleFluent;

 jumpToX :: inertialFluent(xCoord);
 jumpToY :: inertialFluent(yCoord);

 abJumpedDef :: inertialFluent;
 abJumpDef :: inertialFluent;

 jump(xCoord, yCoord, xCoord, yCoord) :: exogenousAction.

% Jumping moves a peg from X,Y to X1,Y1
jump(X,Y,X1,Y1) causes -peg(X,Y).
jump(X,Y,X1,Y1) causes peg(X1,Y1).

% A hack to make sure we only jump once per round.
jump(X,Y,X1,Y1) causes jumpToX=X1.
jump(X,Y,X1,Y1) causes jumpToY=Y1.

% Jumping over a peg causes it to be jumped.
% Default orthogonal rules, mutable by unsetting abJumpedDef.
jump(X,Y,X1,Y1) causes jumped(P,Q) if abs(X-P)=1 & abs(X1-P)=1 & Y=Q & Y1=Q & peg(P,Q) &
abJumpedDef.
jump(X,Y,X1,Y1) causes jumped(P,Q) if abs(Y-Q)=1 & abs(Y1-Q)=1 & X=P & X1=P & peg(P,Q) &
abJumpedDef.

% Jumped pegs disappear.
caused -peg(P,Q) if jumped(P,Q).
jump(X,Y,X1,Y1) causes pegsLeft=C1 if pegsLeft=C & C1=C-1.

% Must jump orthogonally by default.
% Mutable by unsetting abJumpDef.
nonexecutable jump(X,Y,X1,Y1) if (abs(X-X1)=\=2 & Y=Y1) & abJumpDef.
nonexecutable jump(X,Y,X1,Y1) if (abs(Y-Y1)=\=2 & X=X1) & abJumpDef.
nonexecutable jump(X,Y,X1,Y1) if (X=\=X1 & Y=\=Y1) & abJumpDef.

% We have to jump a peg when we make a jump.
```

```
constraint [\/ P \/ Q | jumped(P,Q)] after jump(X,Y,X1,Y1).
% We can't jump if there's no peg there.
nonexecutable jump(X,Y,X1,Y1) if -peg(X,Y).
% We can't jump to a spot if a peg is already there.
nonexecutable jump(X,Y,X1,Y1) if peg(X1,Y1).
% We can't jump somewhere if it's not a valid part of the board.
nonexecutable jump(X,Y,X1,Y1) if nopegs(X1,Y1).

default abJumpedDef.
default abJumpDef.

caused -peg(X,Y) if nopegs(X,Y).
caused -nopegs(X,Y) if peg(X,Y).
default -jumped(X,Y).
0: [/\ P /\ Q | -jumped(P,Q)].
0: jumpToX=1, jumpToY=1.
0: pegsLeft=numPegs.

:- show peg(X,Y);jumped(X,Y).
```

## C.2: Triangle Board Description

```
% The "classic IQ test"
%       X
%       XX
%       XXX
%       XXXX
%       .XXXX

:- macros
 numPegs -> 14.

:- include 'pegjump'.

% Define how big the board is.
:- objects
 1..5 :: xCoord;
 1..5 :: yCoord.

% Add on to jumped rules, now we can jump diagonally like a backslash (up & left or down
& right).
jump(X,Y,X1,Y1) causes jumped(P,Q) if P-X=1 & X1-P=1 & Q-Y=1 & Y1-Q=1 & peg(P,Q).
jump(X,Y,X1,Y1) causes jumped(P,Q) if X-P=1 & P-X1=1 & Y-Q=1 & Q-Y1=1 & peg(P,Q).

% Disable default jump rules, we need to make our own.
caused -abJumpDef.

% Make our own jumping rules.  Orthogonal movement and diagonal "backslash" movement
supported.
nonexecutable jump(X,Y,X1,Y1) if (abs(X-X1)=\=2 & Y=Y1).
nonexecutable jump(X,Y,X1,Y1) if (abs(Y-Y1)=\=2 & X=X1).
nonexecutable jump(X,Y,X1,Y1) if (X=\=X1 & Y=\=Y1 & abs((X-X1)+(Y-Y1))=\=4).

:- query
 label :: 1;
 maxstep :: numPegs-1;
 0: peg(1,1), nopegs(2,1), nopegs(3,1), nopegs(4,1), nopegs(5,1);
 0: peg(1,2), peg(2,2), nopegs(3,2), nopegs(4,2), nopegs(5,2);
 0: peg(1,3), peg(2,3), peg(3,3), nopegs(4,3), nopegs(5,3);
 0: peg(1,4), peg(2,4), peg(3,4), peg(4,4), nopegs(5,4);
 0: -peg(1,5), peg(2,5), peg(3,5), peg(4,5), peg(5,5);
 0: -nopegs(1,5);
 % "True solution" victory condition: final peg must end up in the starting free hole.
 maxstep: peg(1,5);
 % For any solution, there should only be one peg left when we're done.
 maxstep: pegsLeft=1.
```