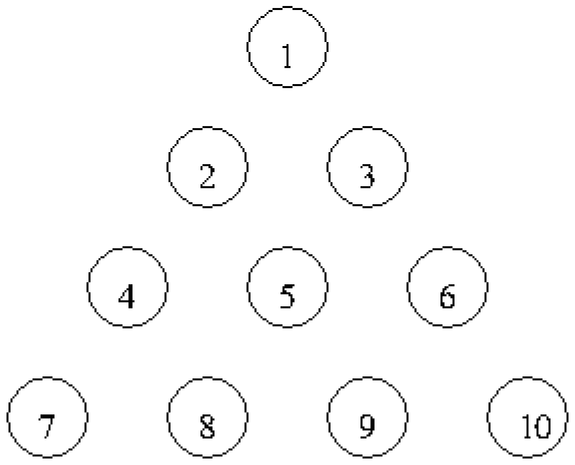
Programming Assignment 2

Assigned: Sept 17.
Due: Oct. 13

Overview

In this assignment, you are to implement the Davis-Putnam algorithm and use it to solve versions of the peg game.

The peg game is a well-known puzzle. You have a board with holes in a geometric pattern and a collection of pegs. Initially, there are pegs in all but one of the holes. The rule is that if there are three holes, A, B, C in a row, C is empty, and there are pegs in A and B, then you can jump the peg in A over B to C, and take out the peg in B. The puzzle is to find a way of carrying out jumps so that there is one peg left on the board.



You will write three programs.

1. An implementation of the Davis-Putnam procedure, which takes as input a set of clauses and outputs either a satisfying valuation, or a statement that the clauses cannot be satisfied.
2. A front end, which takes as input a geometric layout of a board and a starting state and outputs a set of clauses that can be input to (1).
3. A back end, which takes as input the output of (1) and translates it into a solution to the original problem.

Compiling the peg puzzle to propositional logic

The peg game can be expressed propositionally as follows: There are two kinds of atoms:

- Peg(H,I) means that hole H has a peg in it at time I. For instance Peg(5,2) means that hole 5 has a peg at time 2. (Note: in propositional logic, a construction like "Peg(5,2)" is considered a *single symbol* of 7 characters. The parentheses and comma are not punctuation, they are just characters in the symbol, to make it easier for the human reader.)
- Jump(A,B,C,I) means that at time I, the peg in A is jumped to C over B.

The number of time points is necessarily equal to the number of holes minus one, since you start with one hole and end with one peg. Let N be the number of holes. Then for each hole H and I=1..N-1 there should be an atom Peg(H,I) and for each triple of holes in a row A,B,C and I=1..N-2 there should be an atom Jump(A,B,C,I).

Propositional Encoding

There are between seven and nine kinds of propositions, depending on which optional categories are included.

1. **Precondition axioms.** If, at time I, you jump a peg from A to C over B, then A and B must have pegs at time I and C must not have a peg at time I.

   For instance, with the above puzzle,
   `Jump(8,5,3,4) => Peg(8,4) ^ Peg(5,4) ^ ~Peg(3,4)`

2. **Causal axioms.** If you jump from A to C over B at time I, then, at time I+1, A and B are empty and C has a peg.

   For instance, `Jump(8,5,3,4) => ~Peg(8,5) ^ ~Peg(5,5) ^ Peg(3,5)`

3. **Frame axioms.** (Frame axioms assert that state can change only if a relevant action occurs.)
   A. If Peg(H,I) and ~Peg(H,I+1) then either Jump(X,H,Y) or Jump(H,X,Y) for some holes X and Y.
      For instance, `Peg(6,4) ^ ~Peg(6,5) => Jump(10,6,3,4) V Jump(3,6,10,4) V Jump(6,3,1,4) V Jump(6,5,4,4).`

   B. If ~Peg(H,I) and Peg(H,I+1) then Jump(X,Y,H) for some holes X and Y.
      For instance `~Peg(6,4) ^ Peg(6,5) => Jump(1,3,6,4) V Jump(4,5,6,4).`

4. **One action at a time**
   A. No two actions can be executed at the same time.
      For instance `~(Jump(1,2,4,4) ^ Jump(10,6,3,4))`
   B. (Optional) At least one action is executed at each step.
      For instance `Jump(1,2,4,4) V Jump(4,2,1,4) V ... V Jump(10,9,8,4)` (going through all 18 possible jumps).
      (Adding additional constraints that you know are valid never costs much in performance, unless you add an enormous number, and can hugely improve performance, depending on the specifics.)

5. **Starting state** . Specify the truth value of `Peg(H,1)` for each hole H.

6. **Ending state:** Exactly one peg remains. This involves two kinds of axioms.
   A. No two holes have a peg. For each pair of holes H,J, assert that `~(Peg(H,N-1) ^ Peg(J,N-1).`
   B. (Optional) At least one peg remains at time N-1. `Peg(1,N-1) V Peg(2,N-1) V ... V Peg(N,N-1).`

Specifications

Input / Output.

All three programs take their input from a text file produce their output to a text file. (If you want, you may use standard input and output.)

Davis-Putnam

The input to the Davis-Putnam procedure has the following form: An atom is denoted by a natural number: 1,2,3 ... The literal P is the same number as atom P; the literal ~P is the negative. A clause is a line of text containing the integers of the corresponding literals. After all the clauses have been given, the next line is the single value 0; anything further in the file is ignored in the execution of the procedure and reproduced at the end of the output file. (This is the mechanism we will use to allow the front end to communicate to the back end.)

The output from the Davis-Putnam procedure has the following form: First, a list of pairs of atom (a natural number) and truth value (either T or F). Second, a line containing the single value 0. Third, the back matter from the input file, reproduced.

Example: Given the input

```
1 2 3
-2 3
-3
0
This is a simple example with 3 clauses and 3 atoms.
```

Davis-Putnam will generate the output

```
1 T
2 F
3 F
0
This is a simple example with 3 clauses and 3 atoms.
```

This corresponds to the clauses

```
P V Q V R.
~Q V R.
~R.
```

If the clauses have no solution, then Davis-Putnam outputs a single line containing a 0, followed by the back-matter in the input file.

Note: Your implementation of Davis-Putnam must work on *any* set of clauses, not just those that are generated by the peg program.

Front end

The front end takes as input a specification of a puzzle and a starting state and generates as output a set of clauses to be satisfied.

The format of the input contains the following elements:

- First line: The number of holes and the hole that is empty at time 1.
- Remaining lines: The encoding of the puzzle as a set of triples. Each line is a triple of three numbers, for holes that lie in a row.

Note: The puzzle does not have to be geometrically feasible. The only requirement is that jumps are symmetric; if the input contains a line I,J,K then it is possible, both to jump from I to K over J and from K to I over J.

Thus, the encoding of the above puzzle is the following:

```
10 1
1 2 4
2 4 7
3 5 8
4 5 6
7 8 9
8 9 10
1 3 6
3 6 10
2 5 9
```

You may assume that the input is correctly formatted. You do not have to do any error checking on the input. The output consists of

- 1. A set of clauses suitable for inputting to Davis-Putnam as described above.

- 2. A key to allow the back end to translate the numbers used for propositional atoms in the clauses into the correct sequence of jumps. The format of this is up to you. One possibility is the one I've used in my Sample output (for the simple example below); this has the advantage of being easily read by a human reader and so useful in debugging.

Back-end

The back end takes as input the output that Davis-Putnam generates when run on the output of the front end. It generates as output the sequence of jumps that solves the problem.

If the input indicates that the clauses have no solution, the back end should output the message "NO SOLUTION".

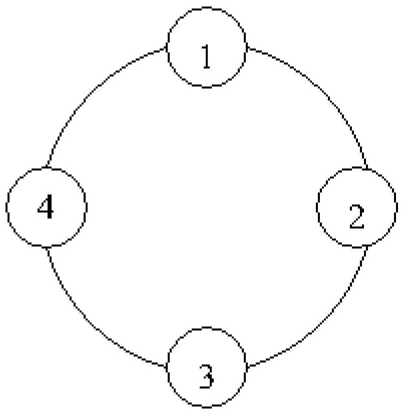
Another example for Davis-Putnam

The following is the input-output pair just for the Davis-Putnam module --- *not* the front and back ends --- corresponding to the example in the class notes.

- Input for Davis-Putnam
- Output from Davis-Putnam

A Simpler Puzzle Example

It would probably be a mistake to begin your testing using the above example, with its 234 propositional atoms and few thousand propositions. Rather, I would suggest that you start by working on the following simple puzzle.



By the way, there is no solution for the 10-peg puzzle, above. There is a solution for the 15-peg triangular peg puzzle.

Input for front end for this small puzzle
Propositions for this small puzzle in symbolic form. Note: There is no need for your program to generate any kind of output or data structure in this format. This is just to help you check your work.
Output for front end for this small puzzle
These do not include the optional axioms 4.B.
Output from Davis-Putnam for this small puzzle.
Output from back end for this small puzzle.

Deliverable

You should upload to NYU Classes. (a) the source code; (b) instructions for running it, if there's anything at all non-obvious about it. Nothing else.

Grading

The Davis-Putnam program is worth 60% of the grade; the front end is worth 35%; the back end is worth 5%. In each of these, a program that does not compile will get a maximum of 10%; a correct program will get 90%; the remaining 10% is for being well-written and well commented.