The Water Jugs Problem

There are two jugs, one holds 4 litres, the other holds 3 litres. Both are initially empty. You can fill either jug from a tap but if you do, you have to fill it completely. You are also allowed to fill one jug from another, or empty out either jug on the ground.

The problem is to get exactly 2 litres in the jug of capacity 3 litres.

problem state: two element list showing amount in each jug e.g. [1 2]

```
                                           ---
                                          |0 0|
                                           ---
                                            |
                          -------------------------------------
                         |                                     |
                        ---                                   ---
                       |4 0|                                 |0 3|
                        ---                                   ---
                         |                                     |
                 -----------------                     -----------------
                |                 |                   |                 |
               ---               ---                 ---               ---
              |1 3|             |4 3|               |3 0|             |4 3|
               ---               ---                 ---               ---
                |                 |                   |                 |
      ---------------------       |             -------------           |
     |          |          |      |            |             |          |
    ---        ---        ---    ---          ---           ---        ---
   |0 3|      |1 0|      |4 3|  |0 3|        |4 0|         |3 3|      |4 0|
    ---        ---        ---    ---          ---           ---        ---
     |          |          |      |            |             |          |
   ------       |          |      |          ------        ------       |
  |      |      |          |      |         |      |      |      |      |
 ---    ---    ---        ---    ---       ---    ---    ---    ---    ---
|3 0|  |4 3|  |0 1|      |0 3|  |3 0|     |1 3|  |4 3|  |4 2|  |4 3|  |1 3|
 ---    ---    ---        ---    ---       ---    ---    ---    ---    ---
  |             |          |      |         |                    |      |
  |           ------       |      |       ------                 |      |
  |          |      |      |      |      |      |                |      |
 ---        ---    ---    ---    ---    ---    ---              ---    ---
|3 3|      |4 1|  |0 3|  |3 0|  |3 3|  |1 0|  |4 3|            |4 0|  |1 0|
 ---        ---    ---    ---    ---    ---    ---              ---    ---
```

```
define isgoal(state) -> result;
  state matches [= 2] -> result
enddefine;

define samestate(state1, state2) -> result;
  state1 = state2 -> result
enddefine;
```

```
define nextfrom(state) -> states;
   vars X Y result;
   state --> [?X ?Y];
   [%    if Y /= 0 and X /= 4 and X+Y > 4 then [4 ^(X+Y-4)] endif;
         if Y /= 0 and X /= 4 and X+Y =< 4 then [^(X+Y) 0] endif;
         if Y /= 3 and X /= 0 and X+Y > 3 then [^(X+Y-3) 3] endif;
         if Y /= 3 and X /= 0 and X+Y =< 3 then [0 ^(X+Y)] endif;
         if X /= 0 then [0 ^Y] endif;
         if Y /= 0 then [^X 0] endif;
         if X /= 4 then [4 ^Y] endif;
         if Y /= 3 then [^X 3] endif;
     %] -> states;
enddefine;


nextfrom([1 3]) =>
** [[4 0] [0 3] [1 0] [4 3]]
```

The Tower Problem

You have a collection of toy bricks of various heights. The problem is build a tower from these bricks to exactly a given height.

problem state: 3 element list, two sublists and an integer.
First sublist is blocks used up so far, second sublist is blocks left to be used. Final integer is target height of tower e.g.

[[1 3 4] [5 9] 17]

```
define isgoal(state) -> result;
  sumlist(state(1)) = state(3) -> result;
enddefine;

define sumlist(numlist) -> total;
   lvars n;
   0 -> total;
   for n in numlist do n + total -> total endfor
enddefine;

isgoal([[1 3 4] [5 9] 17]) =>
** <false>
isgoal([[1 3 4 9] [5] 17]) =>
** <true>
```

```
define samestate(state1, state2) -> result;
  sort(state1(1)) = sort(state2(1)) and
  sort(state1(2)) = sort(state2(2)) and
  state1(3) = state2(3) -> result;
enddefine;

samestate([[1 3 4][5 9] 17],[[1 4 3][9 5] 17]) =>
** <true>
```

```
define nextfrom(state) -> states;
  vars sofar, left, target, early, late block;
  state --> [?sofar ?left ?target];
  [%  for block in left do
      left --> [??early ^block ??late];
      [[^^sofar ^block][^^early ^^late] ^target]
      endfor
  %] -> states
enddefine;

nextfrom([[1 3 4][5 9] 17]) ==>
** [[[1 3 4 5] [9] 17] [[1 3 4 9] [5] 17]]
nextfrom([[1][2 3 4 5] 20]) ==>
** [[[1 2] [3 4 5] 20]
    [[1 3] [2 4 5] 20]
    [[1 4] [2 3 5] 20]
    [[1 5] [2 3 4] 20]]
```

```
define search(state) -> result;
    lvars alternatives, considered, templist;
    [^state] -> alternatives;
    [] -> considered; false -> result;
    until alternatives = [] do
        dest(alternatives) -> alternatives -> state;
        state :: considered -> considered;
        if isgoal(state) then state -> result; return endif;
        nextfrom(state) -> templist;
        for state in templist do
            unless isoneof(state, alternatives) or
                   isoneof(state, considered)
            then insert(state, alternatives) -> alternatives;
            endunless
        endfor
    enduntil;
enddefine;
```

```
define isoneof(state, list) -> result;
    lvars prevstate;
    false -> result;
    for prevstate in list do
        if samestate(state, prevstate)
        then true -> result; return;
        endif;
    endfor;
enddefine;
```

```
define insert(newstate, list) -> result;
    vars state, rest;
    if list matches [?state ??rest] and
        isbetter(state, newstate)
    then state :: insert(newstate, rest) -> result
    else newstate :: list -> result
    endif
enddefine;
```

```
define isbetter(oldstate, newstate) -> result;
   ;;; depth-first
   false -> result;
enddefine;

define isbetter(oldstate, newstate) -> result;
   ;;; breadth-first
   true -> result;
enddefine;
```

```
/* TOWER PROBLEM */
/* using the depth first definition of isbetter */

search([[][1 3 4 5 9] 17]) =>
** [[9 5 3] [1 4] 17]
search([[][1 3 4 5 9] 100]) =>
** <false>

/* using the breadth first definition of isbetter */

search([[][1 3 4 5 9] 17]) =>
** [[3 5 9] [1 4] 17]
search([[][1 3 4 5 9] 100]) =>
** <false>
```

```
/* JUGS PROBLEM */
/* using the depth first definition of isbetter */

search([0 0]) =>
** [4 2]

/* using the breadth first definition of isbetter */

search([0 0]) =>
** [4 2]
```

```
define search_all(state) -> result;
    lvars alternatives, considered, templist;
    [^state] -> alternatives; [] -> considered;
    [% until alternatives = [] do
        dest(alternatives) -> alternatives -> state;
        state :: considered -> considered;
        if isgoal(state) then state endif;  ;;; leave on stack
        nextfrom(state) -> templist;
        for state in templist do
            unless isoneof(state, alternatives) or
                   isoneof(state, considered)
            then insert(state, alternatives) -> alternatives;
            endunless
        endfor
    enduntil; %] -> result
enddefine;
```

```
/* jugs - all solutions*/

search_all([0 0]) =>
** [[4 2] [0 2]]

/* tower - all solutions*/

search_all([[][1 3 4 5 9] 17]) =>
** [[[9 5 3] [1 4] 17] [[9 4 3 1] [5] 17]]
```