CPSC 312

Functional and Logic Programming

November 19, 2015

Project

Congratulations on finishing the first project!

Your Haskell project will be announced over the next few days.

You will have 3 weeks to do this project which means it will be due by the end of the second week of December, a few days after your final exam.

This will be your last work for this class! (I.e. no more assignments - oh, you have the final exam as well)

Assignment

Your one and only Haskell homework is due tomorrow night.

Handin is now open for submissions:
cs312, assign4

Like I said before, you can hand this one in as a pair.
If you aren't and you don't have a teammate, you should start searching for one for your project.
Project MUST be done in a pair. Piazza teammate search can be used.

Assignment

1. Type Declaration: If the problem specification doesn’t explicitly tell you the expected types of the function arguments or results, use what works for you.

Assignment

2. Validation or arguments: Do you need to validate arguments? No, you don't.
The only thing you need to validate with respect to arguments, is making sure you never get a "Non-Exhaustive Patterns" error. Everything else is dealt with by Haskell. For instance:

```
Prelude> 1/0
Infinity

Prelude> sqrt(-1)
NaN
```

From the previous lectures

List Comprehension

Algebraic Types

Topics left:
Search in AI
Higher level functions

Search and Intelligence

"A physical symbol system exercises its intelligence in problem solving by search -- that is, by generating and progressively modifying symbol structures until it produces a solution structure."

Allen Newell and Herbert A. Simon, “Computer Science as Empirical Inquiry: Symbols and Search”

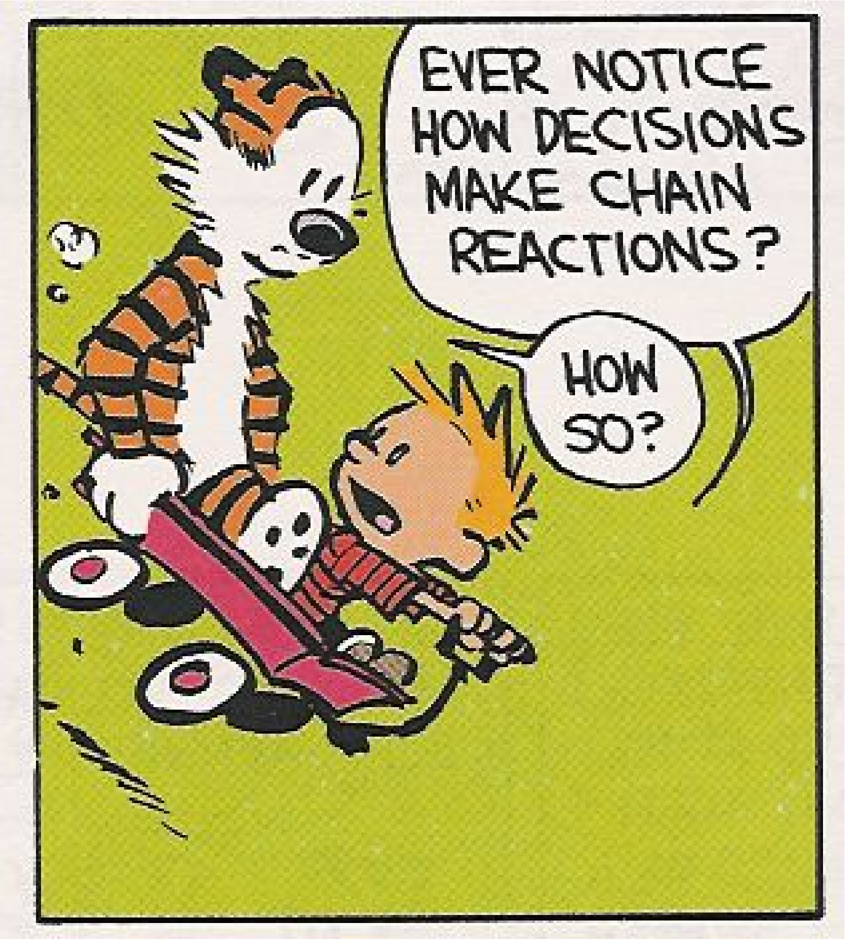
Search and Intelligence

"In order to cope, an organism must either armor itself (like a tree or a clam) and 'hope for the best,' or else develop methods for getting out of harm's way and into the better neighborhoods of the vicinity. If you follow this latter course, you are confronted with the primordial problem that every agent must continually solve: Now what do I do?"

Daniel C. Dennett, "Consciousness Explained"

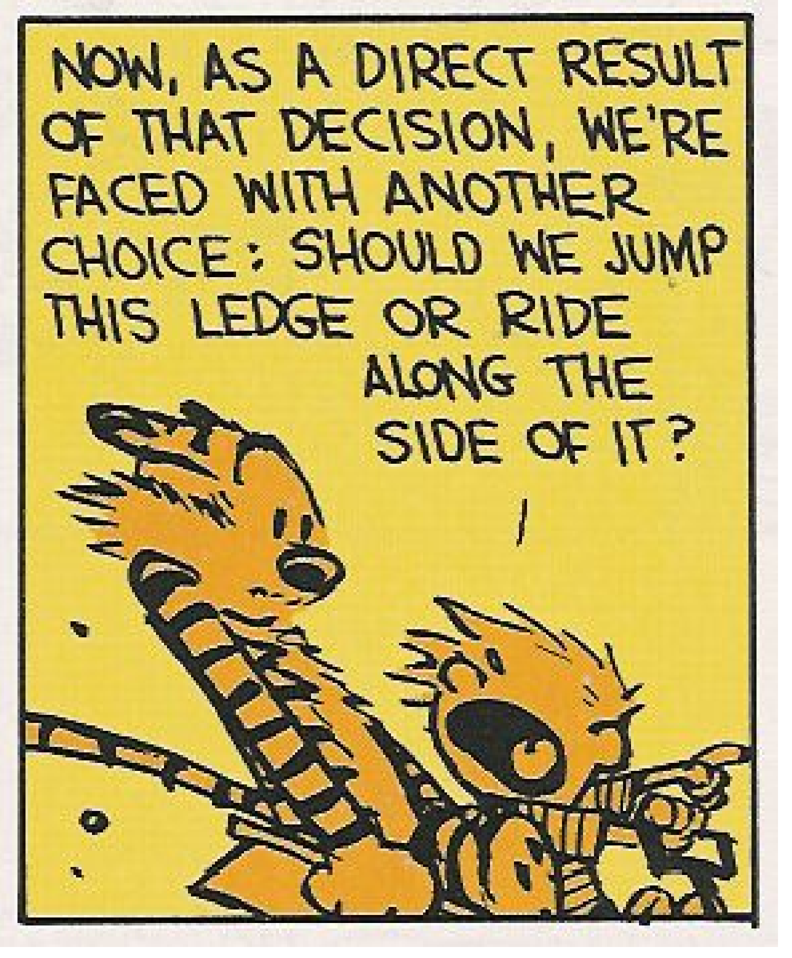

Calvin and Hobbes
by WATTERSON
FASTER! FASTER!


THE TURBO IS POOPED.
THAT'S OK. GRAVITY JUST KICKED IN.


EVER NOTICE HOW DECISIONS MAKE CHAIN REACTIONS?
HOW SO?

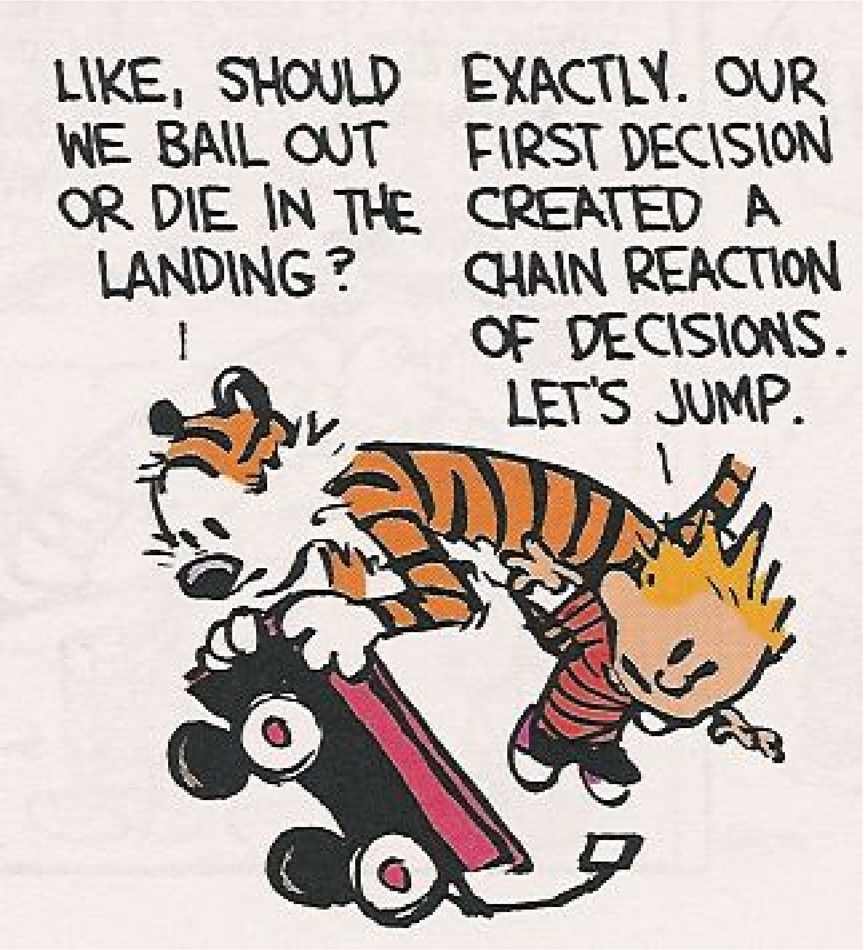

WELL, EACH DECISION WE MAKE DETERMINES THE RANGE OF CHOICES WE'LL FACE NEXT.


TAKE THIS FORK IN THE ROAD FOR INSTANCE. WHICH WAY SHOULD WE GO? ARBITRARILY, I CHOOSE LEFT.


NOW, AS A DIRECT RESULT OF THAT DECISION, WE'RE FACED WITH ANOTHER CHOICE: SHOULD WE JUMP THIS LEDGE OR RIDE ALONG THE SIDE OF IT?


IF WE HADN'T TURNED LEFT AT THE FORK, THIS NEW CHOICE WOULD NEVER HAVE COME UP.
WATTERSON


I NOTE, WITH SOME DISMAY, YOU'VE CHOSEN TO JUMP THE LEDGE.
RIGHT. AND THAT DECI-SION WILL GIVE US NEW CHOICES.


LIKE, SHOULD WE BAIL OUT OR DIE IN THE LANDING?
EXACTLY. OUR FIRST DECISION CREATED A CHAIN REACTION OF DECISIONS. LET'S JUMP.


SEE? IF YOU DON'T MAKE EACH DECISION CAREFULLY, YOU NEVER KNOW *WHERE* YOU'LL END UP. THAT'S AN IMPORTANT LESSON WE SHOULD LEARN SOMETIME.
I WISH WE COULD TALK ABOUT THESE THINGS WITHOUT THE VISUAL AIDS.

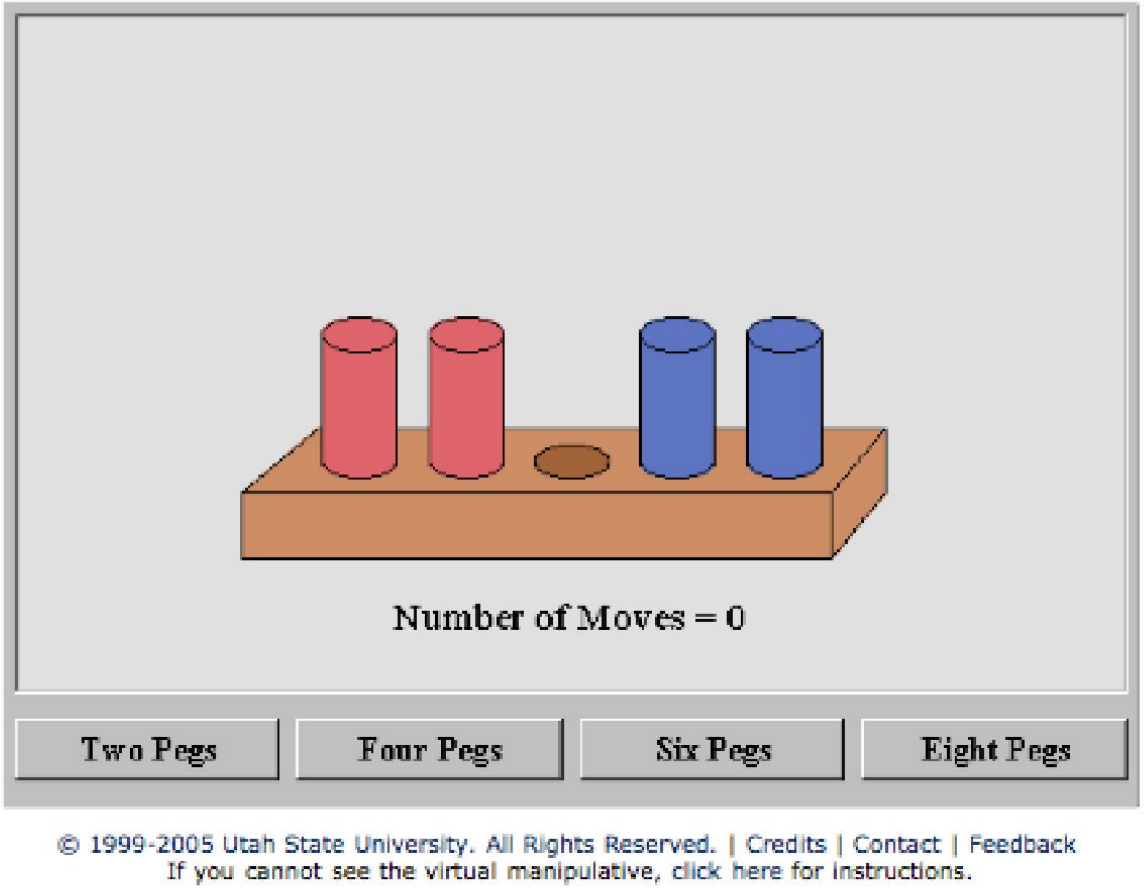
Puzzles and intelligence

In the early days of AI, the problems posed by puzzles and games were thought to be hard problems, requiring human intelligence for their solution. Things like understanding language were thought to be easy by comparison.

Since then, we've figured out that things like using English are hard, and tile puzzles and chess turn out to be easy problems by comparison. Still, the so-called "toy domains" of puzzles and games have remained useful vehicles for exploring the underlying principles of search and how those search techniques can be used in getting intelligent behaviour out of our machines.

The 15-tile puzzle

A simple peg puzzle



http://nlvm.usu.edu/en/nav/frames_asid_182_g_3_t_1.html

State-space search

The kind of search we use to solve these puzzles is often called state-space search. It's what Calvin and Hobbes were doing as they rode down the hill in their wagon, and it's what Newell and Simon said is characteristic of intelligence.

State-space search

A state-space is defined as the set of all possible states generated by the repeated application of a finite set of operators (or operations, or transformations, or moves... they're all the same thing in this context) to some initial state. In performing a state-space search, the intention is usually to find a sequence of operators that gets one from the initial state to some goal state.

State-space search

When applied to the start state, that sequence of operators produces a chain of states from the start state through intermediate states to the goal state. The sequence of operators is really the solution, but it's sometimes easier for us to follow the solution by looking at the chain of states and inferring the operations.

State-space search

Computation itself is just state-space search. The current state of a computation is the collection of variables and their bindings, including the program counter. The operations are given by the instructions in the program. The computer executes the instruction indicated by the program counter, variable bindings are changed, the program counter is incremented, and the computation has progressed to a new state. (You’ll hear more about this in CPSC 421, however, they'll probably talk about it in terms of theoretical finite state automata and Turing machines, not real computing hardware.)

State-space search

So this notion of beginning with some start state and applying the correct sequence of operations to achieve some goal state is fundamental to computer science, not just artificial intelligence.

Simple puzzles give us useful experience with state-space search, representations for states, and operations for transforming one state into another.

For example, a state in a peg puzzle need be nothing more than a string of characters representing pegs and spaces. The operations at some abstract level are just sliding a peg to an adjacent open space, or jumping a peg of another colour into an open space.

Two pegs

R_B

goal: B_R

Two pegs

R_B

goal: B_R

_RB

Two pegs

```
        R_B        goal: B_R
       /
    _RB
     |
    BR_
```

Two pegs

R_B

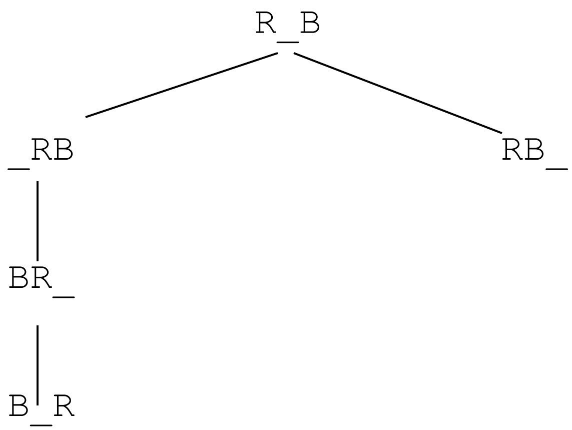
goal: B_R

_RB

BR_

B_R

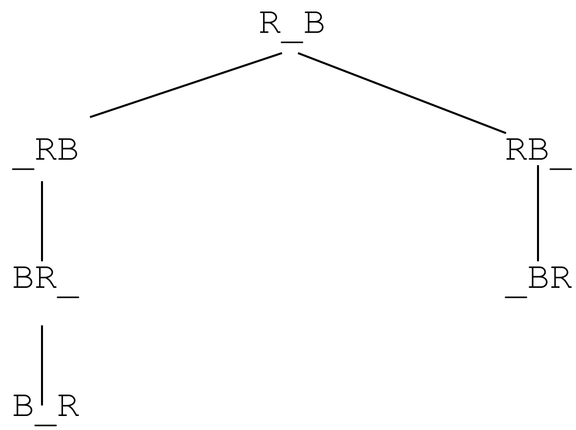
Two pegs



goal: B_R

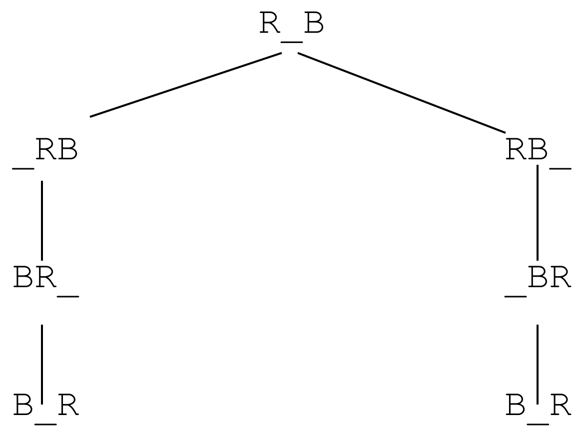
Two pegs

goal: B_R



Two pegs

goal: B_R



Three pegs

RR_B

goal: B_RR

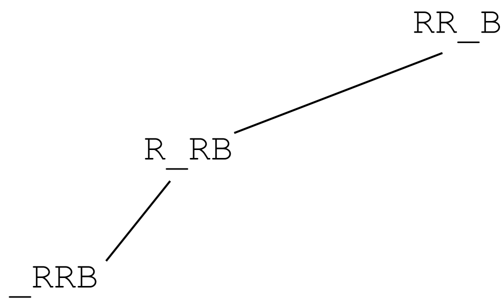
Three pegs

RR_B

goal: B_RR

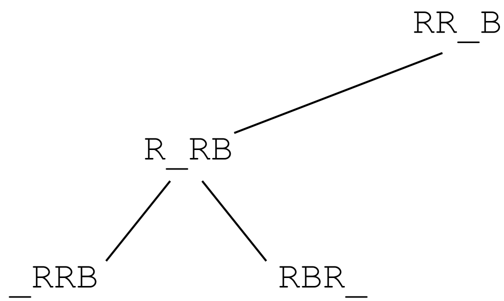
R_RB

Three pegs



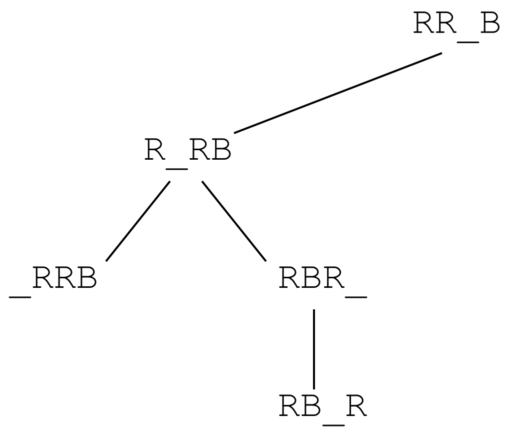
goal: B_RR

Three pegs



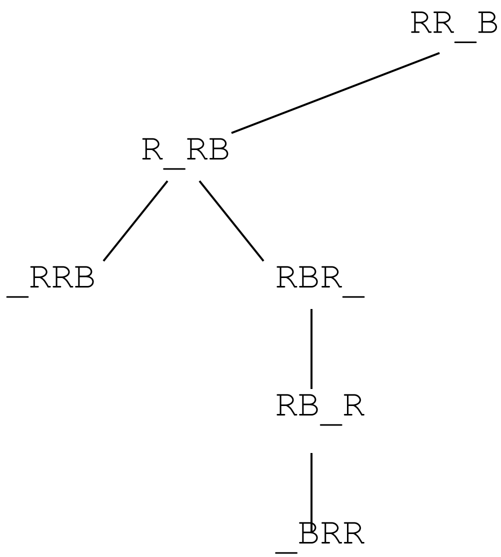
goal: B_RR

Three pegs



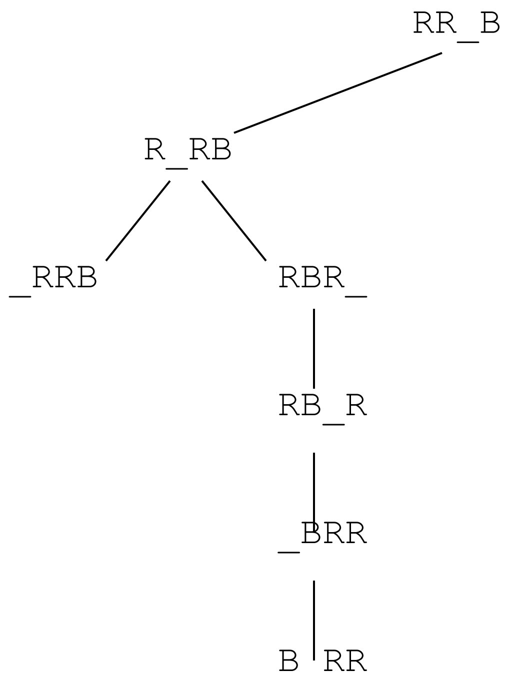
goal: B_RR

Three pegs



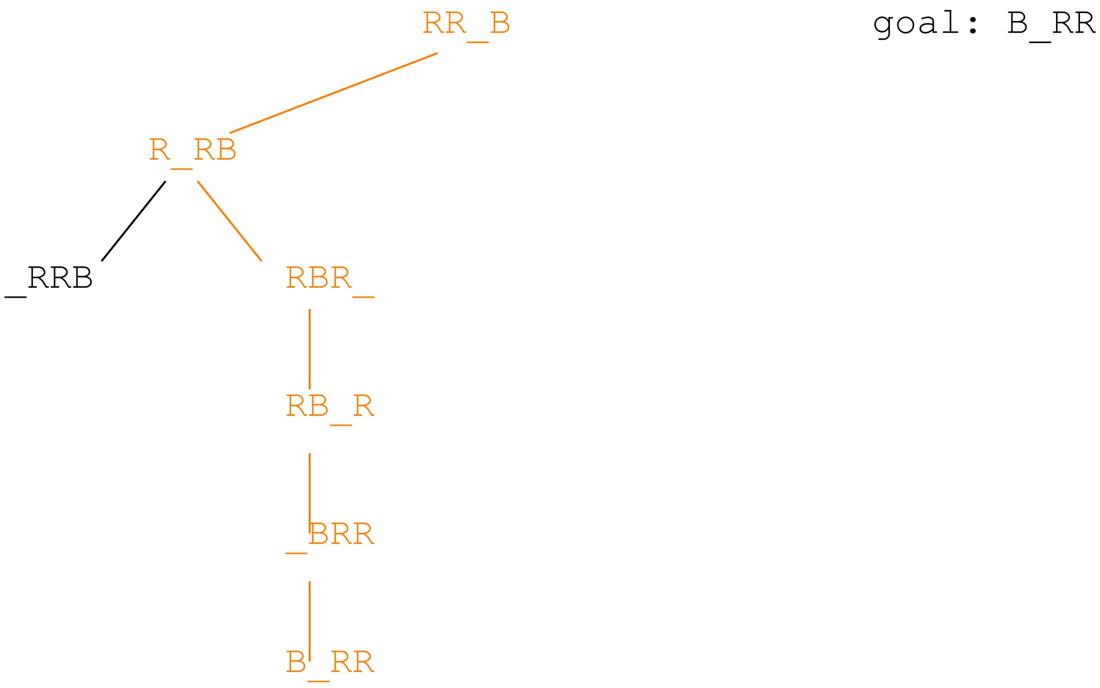
goal: B_RR

Three pegs



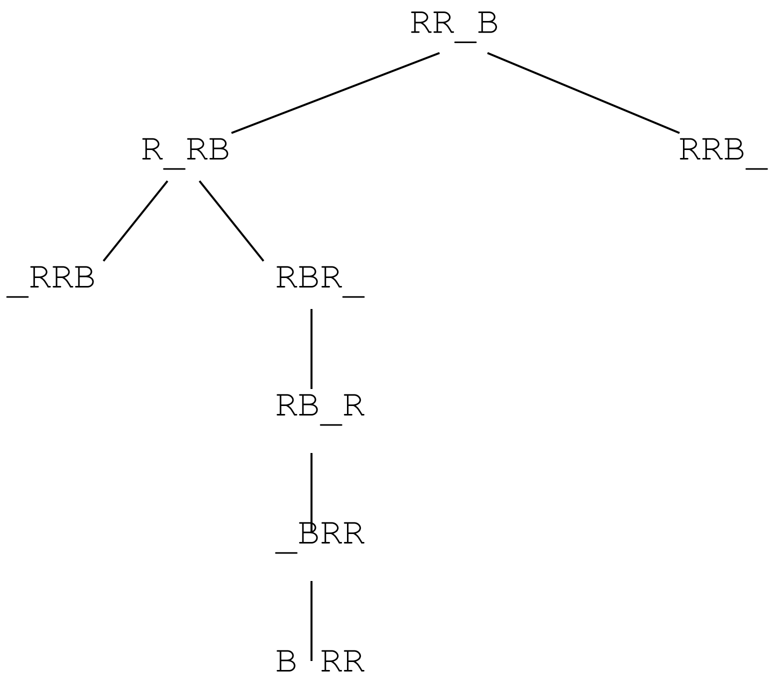
goal: B_RR

Three pegs



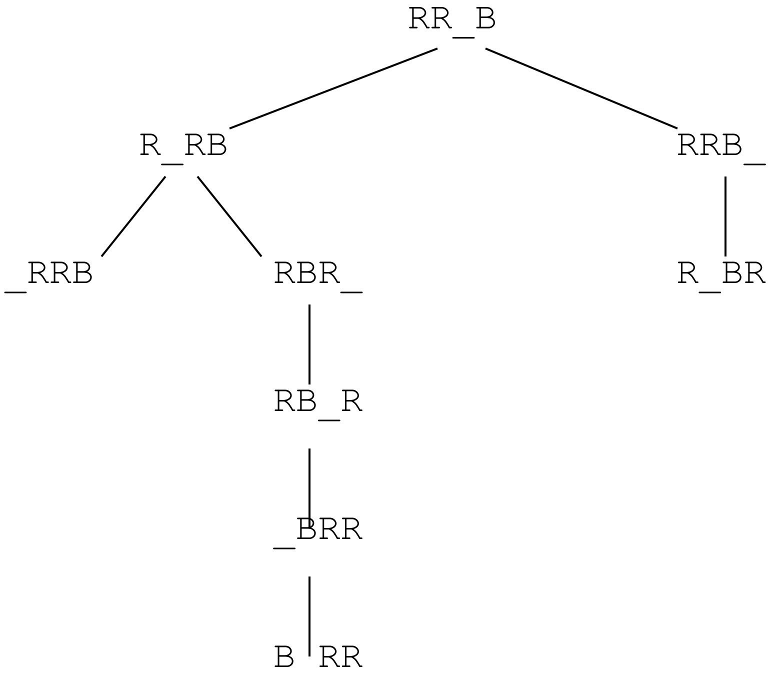
Three pegs

goal: B_RR



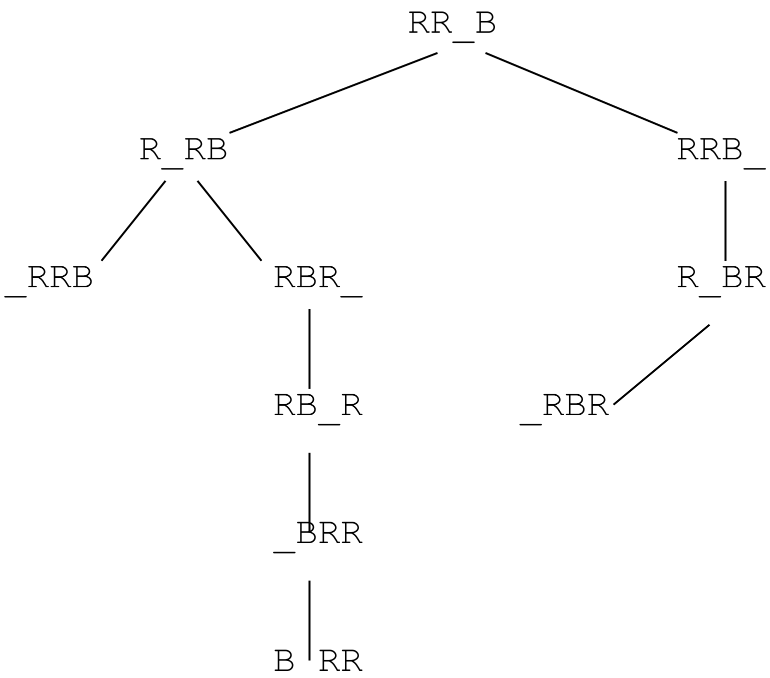
Three pegs

goal: B_RR

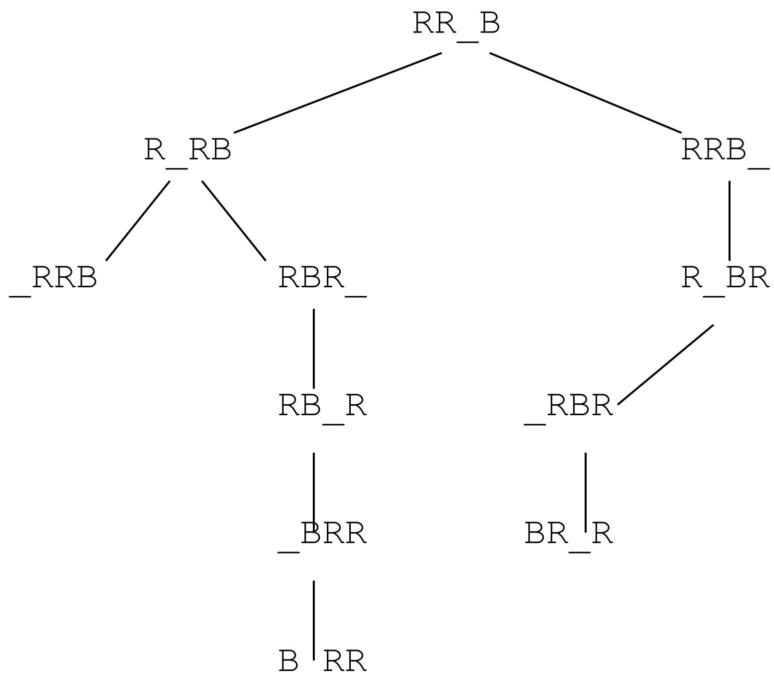


Three pegs

goal: B_RR

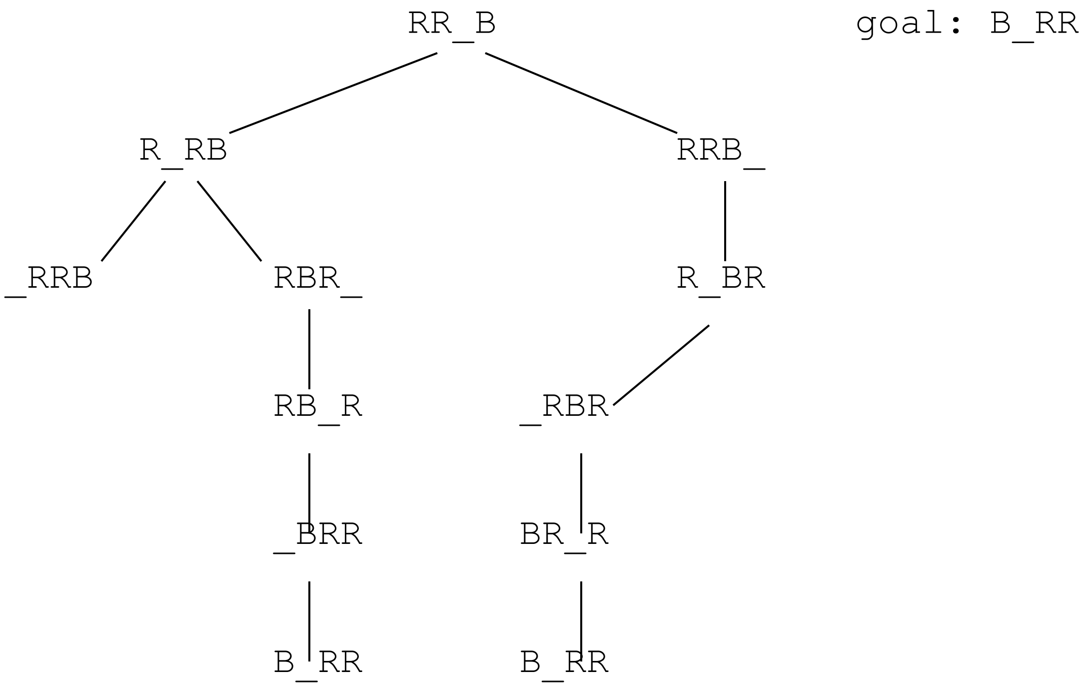


Three pegs

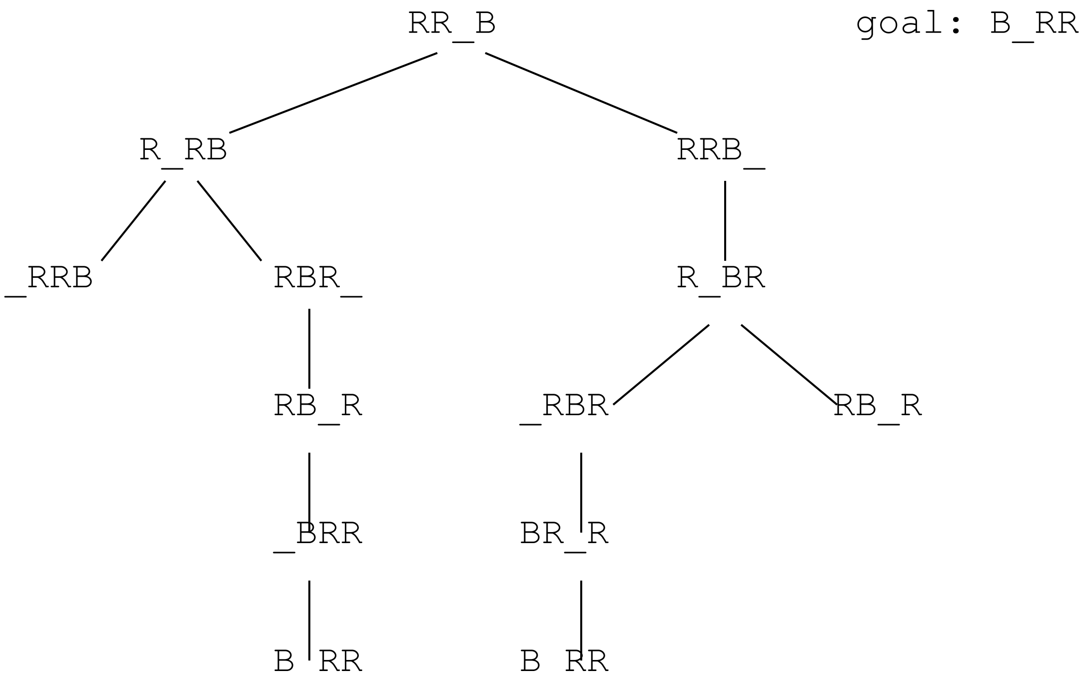


goal: B_RR

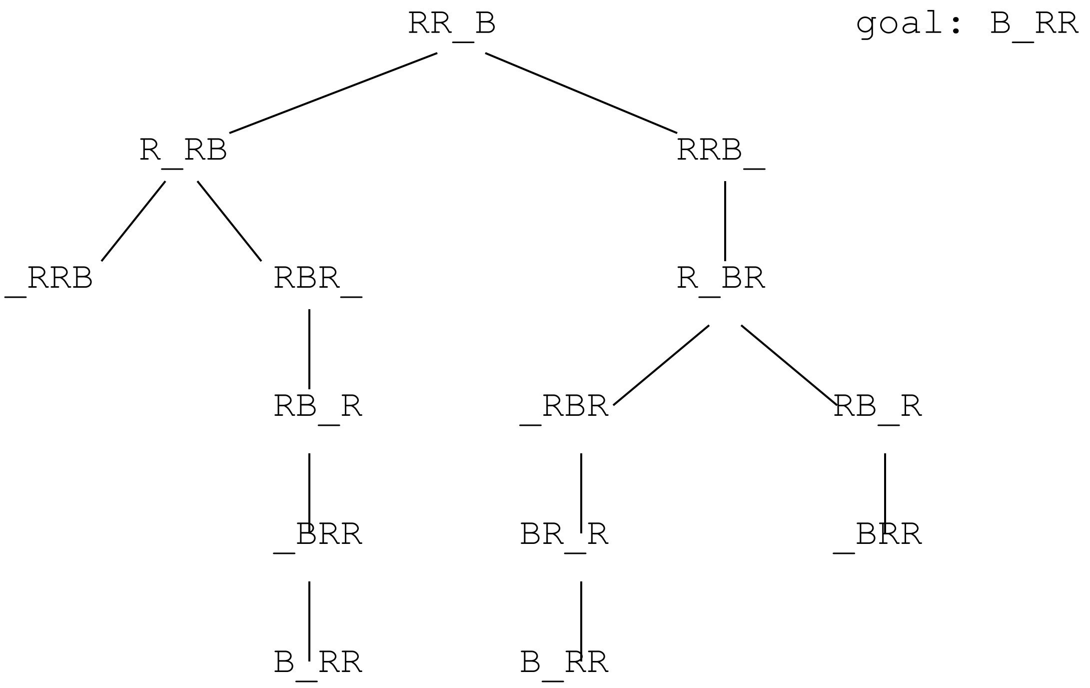
Three pegs



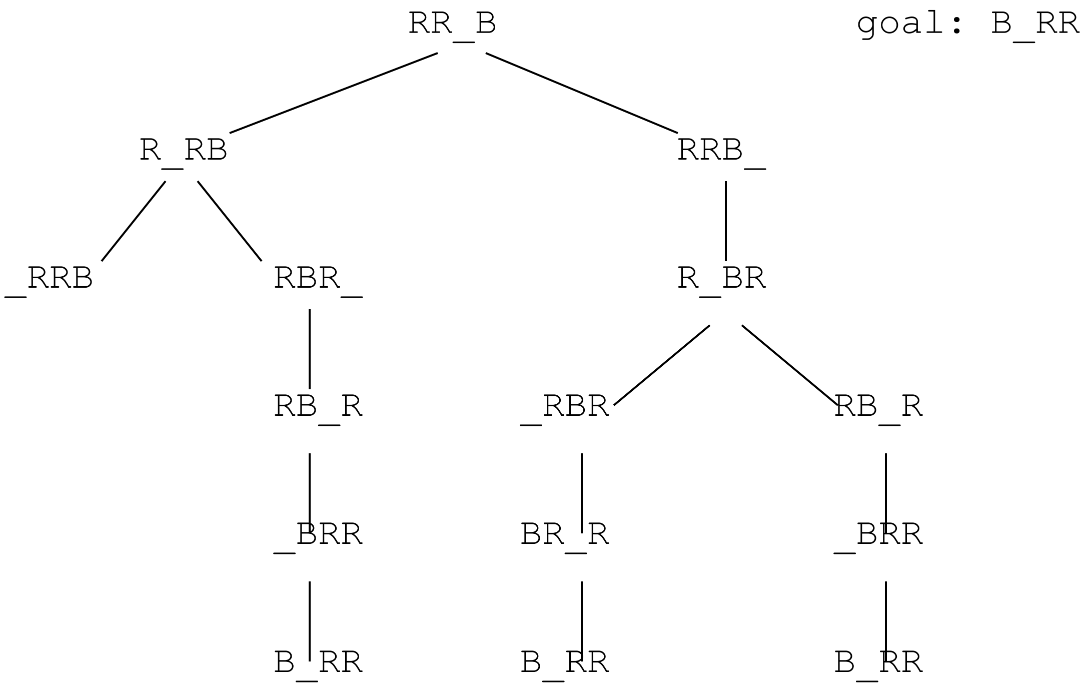
Three pegs



Three pegs



Three pegs


RR_B
goal: B_RR
R_RB
RRB_
_RRB
RBR_
R_BR
RB_R
_RBR
RB_R
_BRR
BR_R
_BRR
B_RR
B_RR
B_RR

Four pegs

RR_BB goal: BB_RR

Four pegs

RR_BB

goal: BB_RR

R_RBB

Four pegs

RR_BB

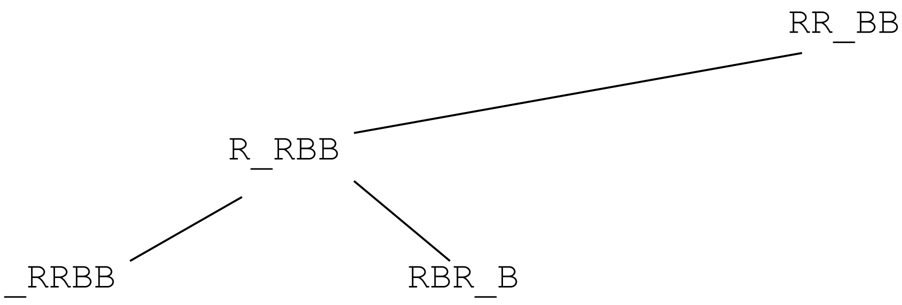
goal: BB_RR

R_RBB

_RRBB

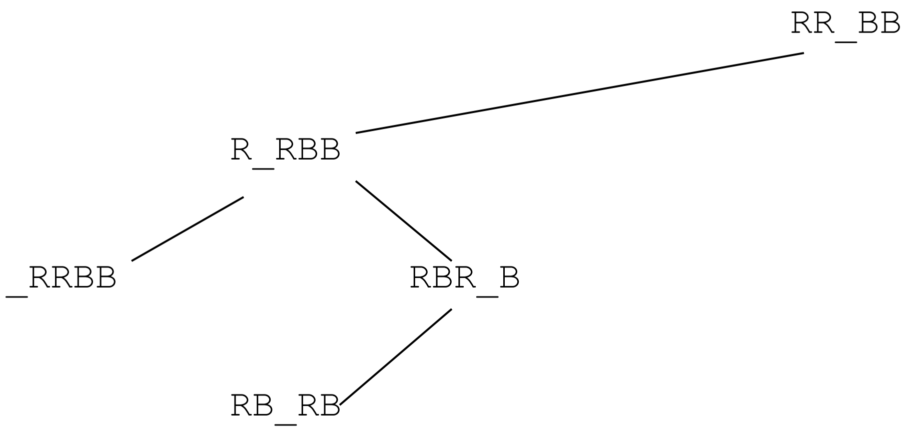
Four pegs

goal: BB_RR

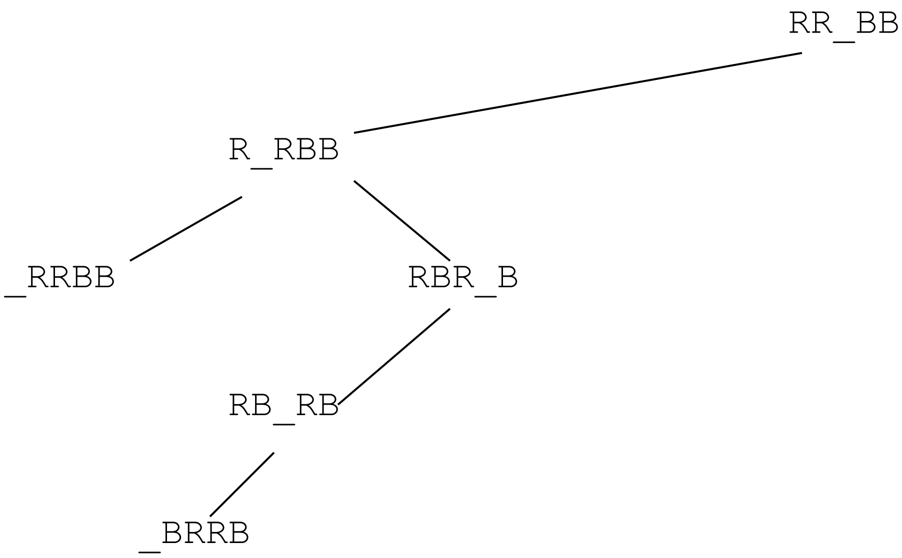


Four pegs

goal: BB_RR

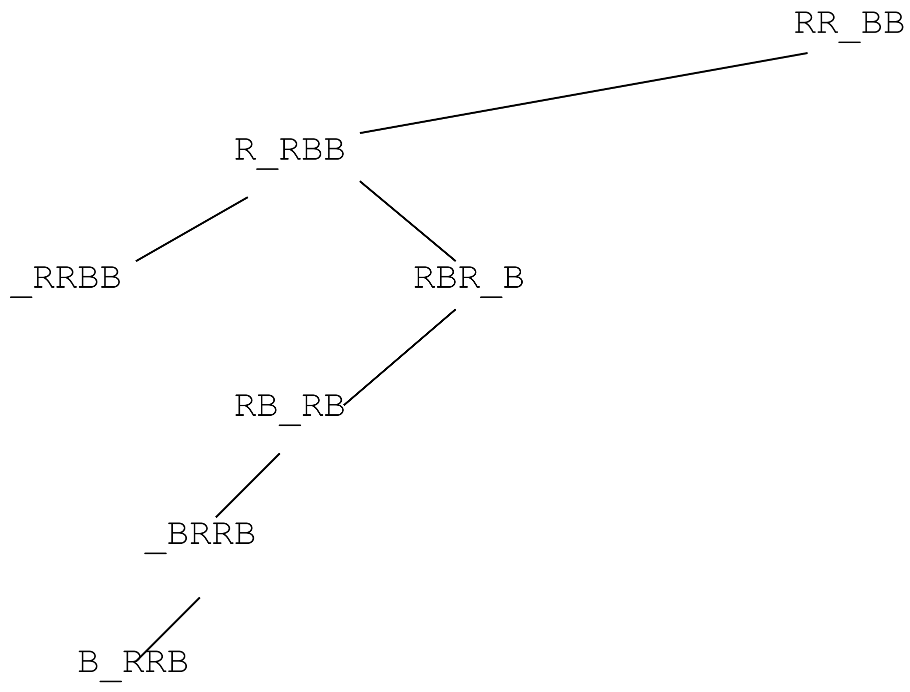


Four pegs



goal: BB_RR

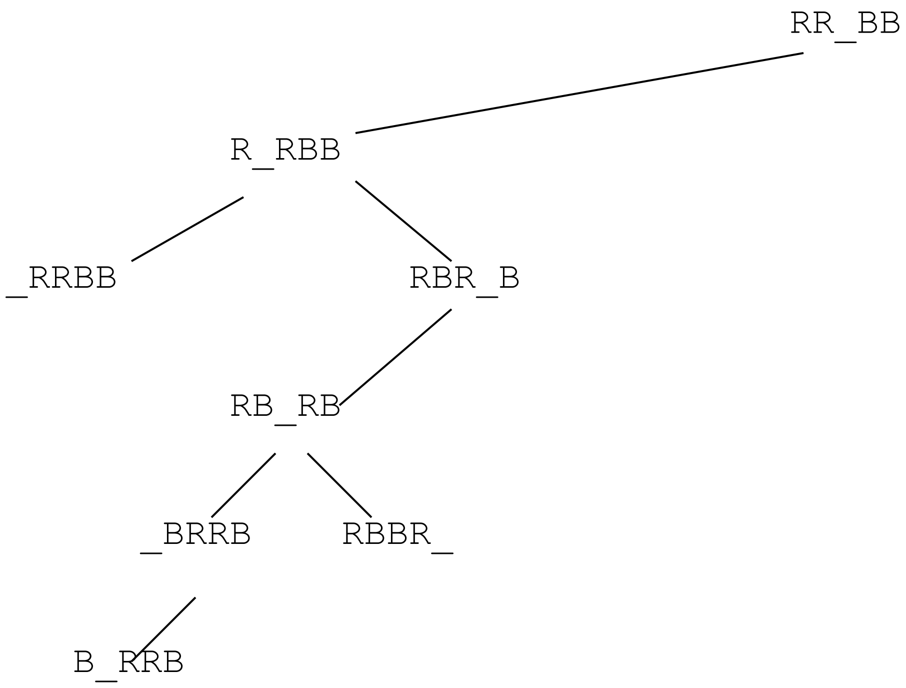
Four pegs



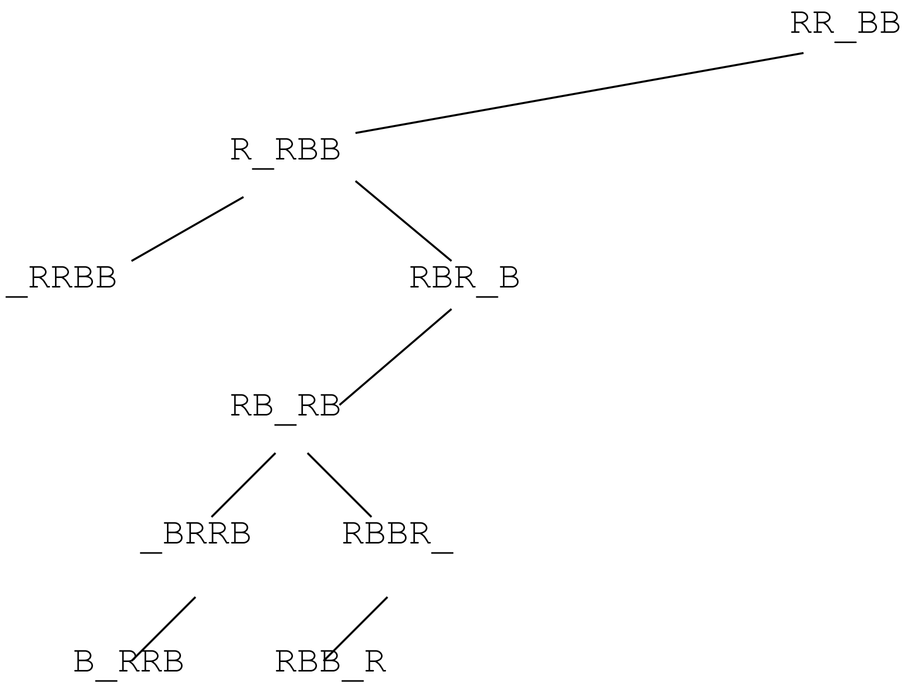
goal: BB_RR

Four pegs

goal: BB_RR

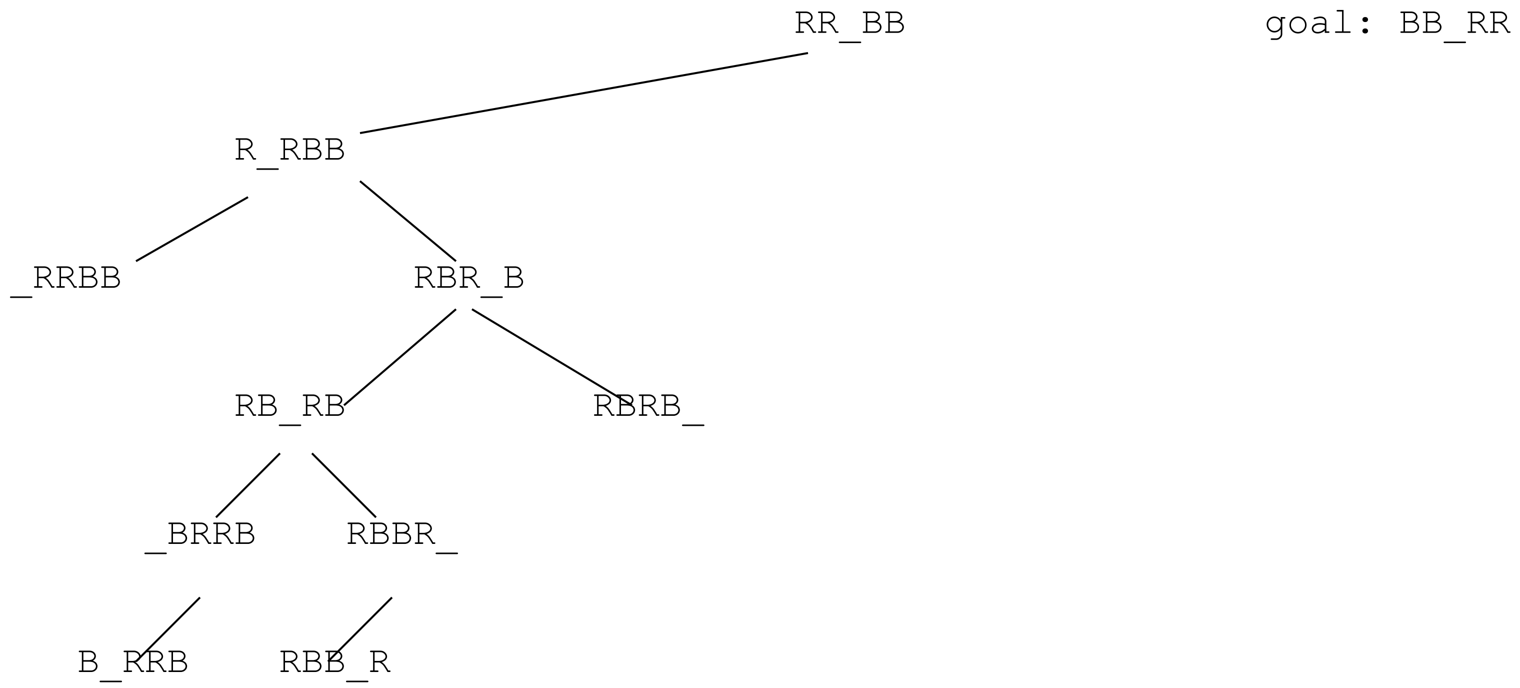


Four pegs

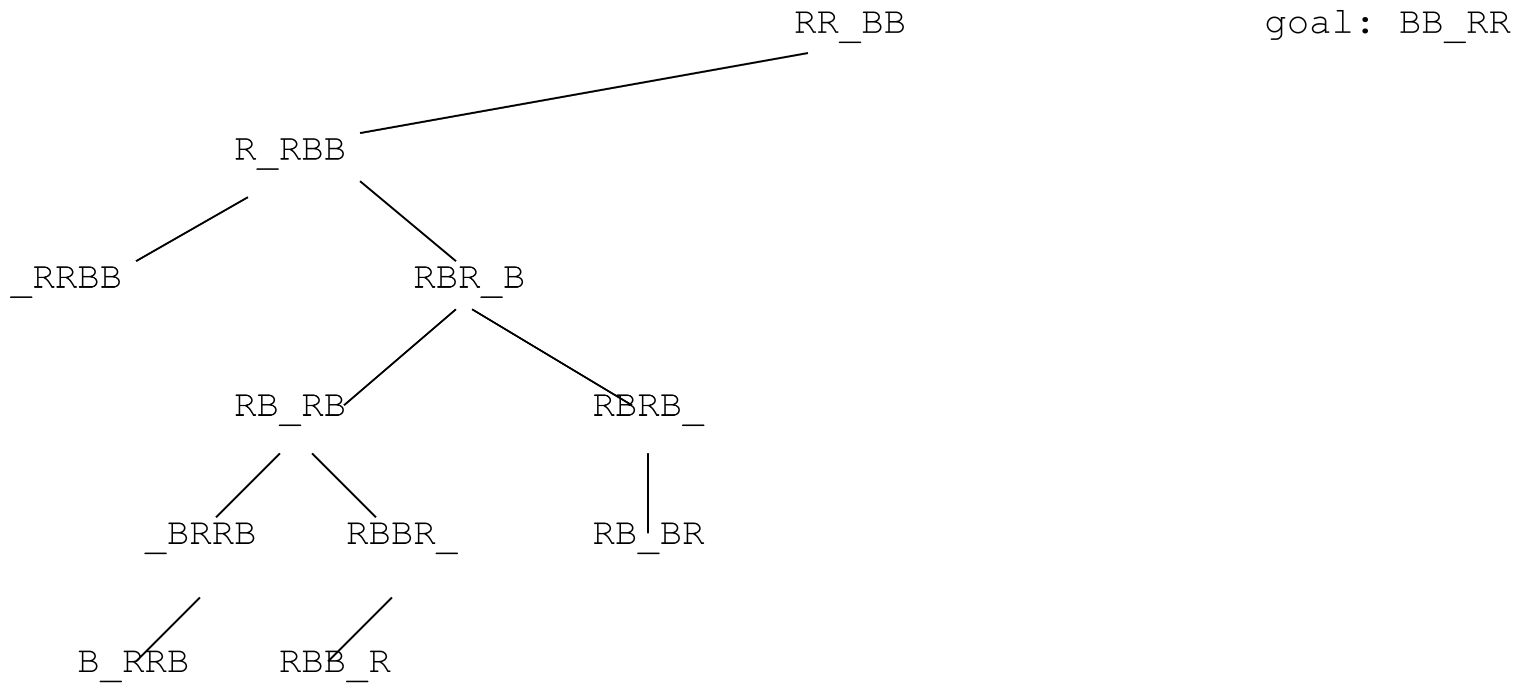


goal: BB_RR

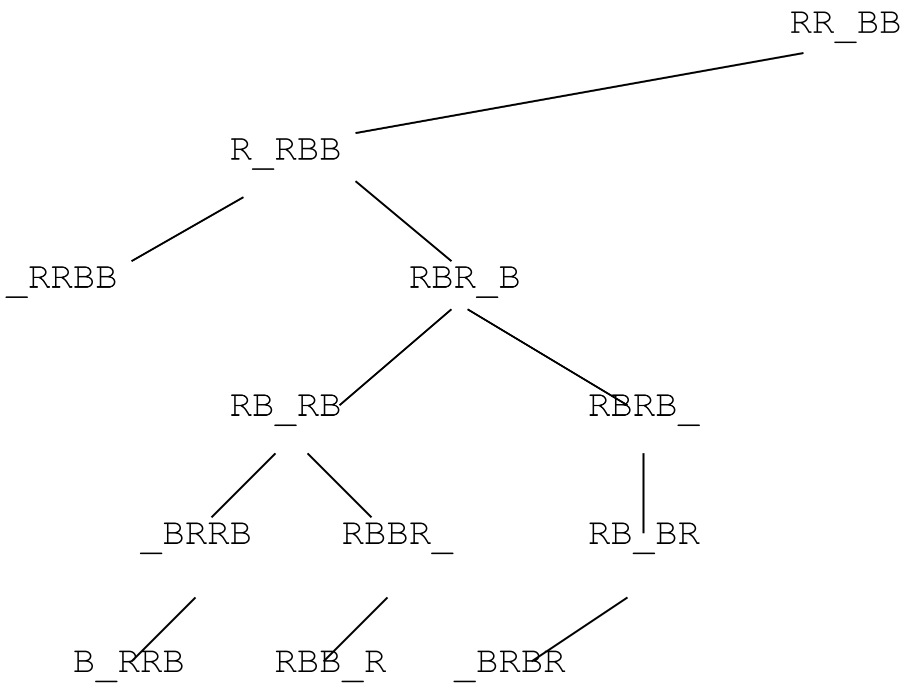
Four pegs



Four pegs

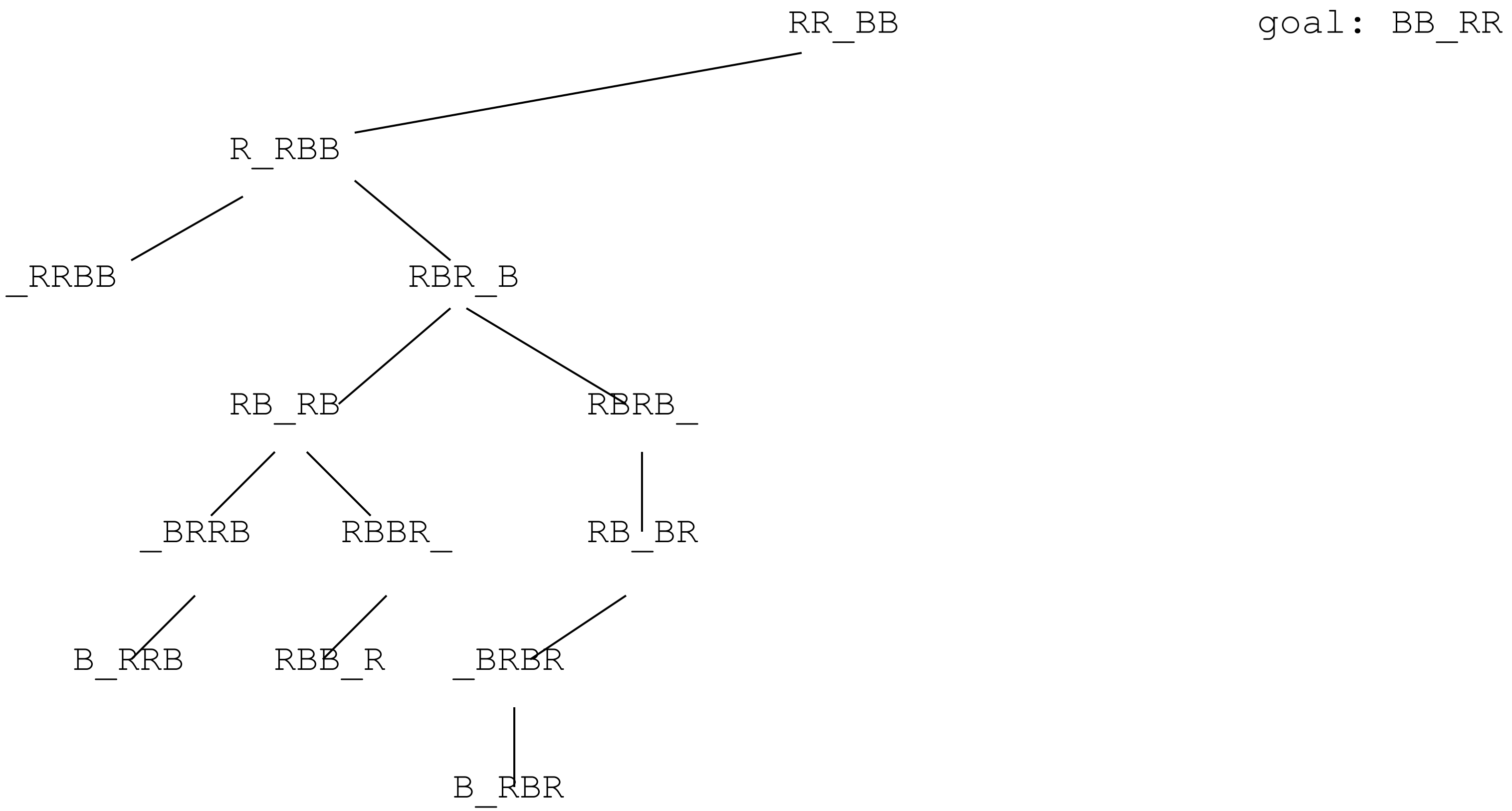

RR_BB
goal: BB_RR
R_RBB
_RRBB
RBR_B
RB_RB
RBRB_
_BRRB
RBBR_
RB_BR
B_RRB
RBB_R

Four pegs



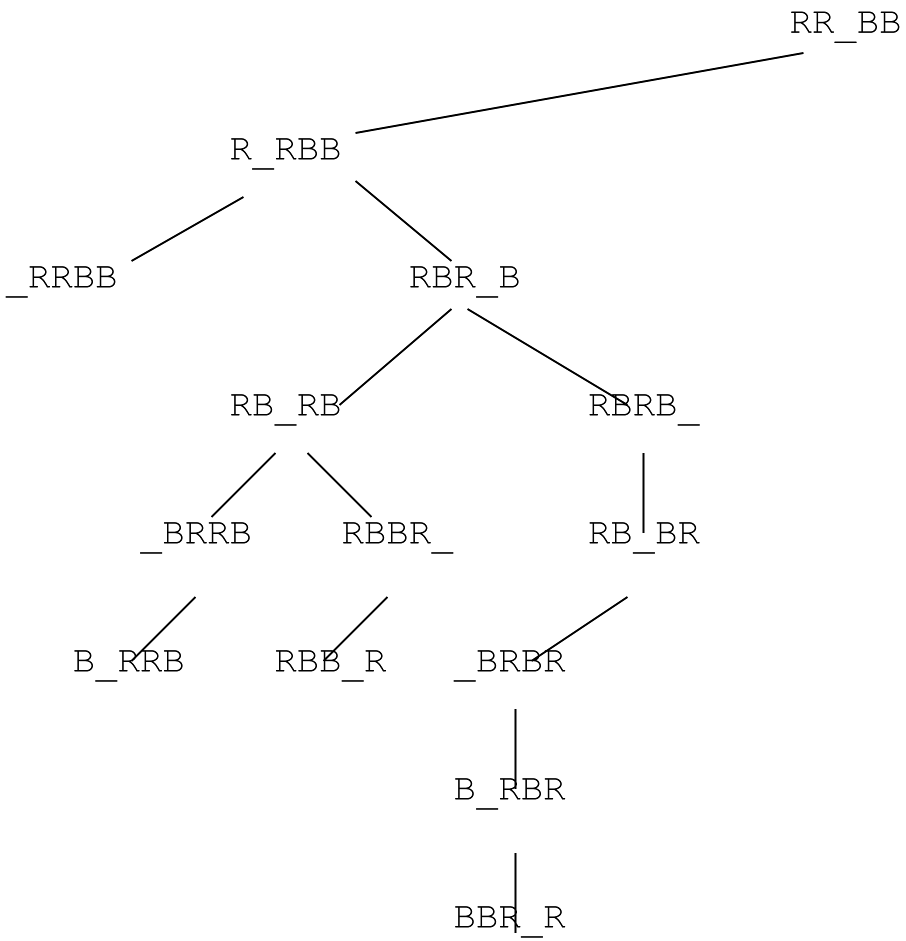
goal: BB_RR

Four pegs

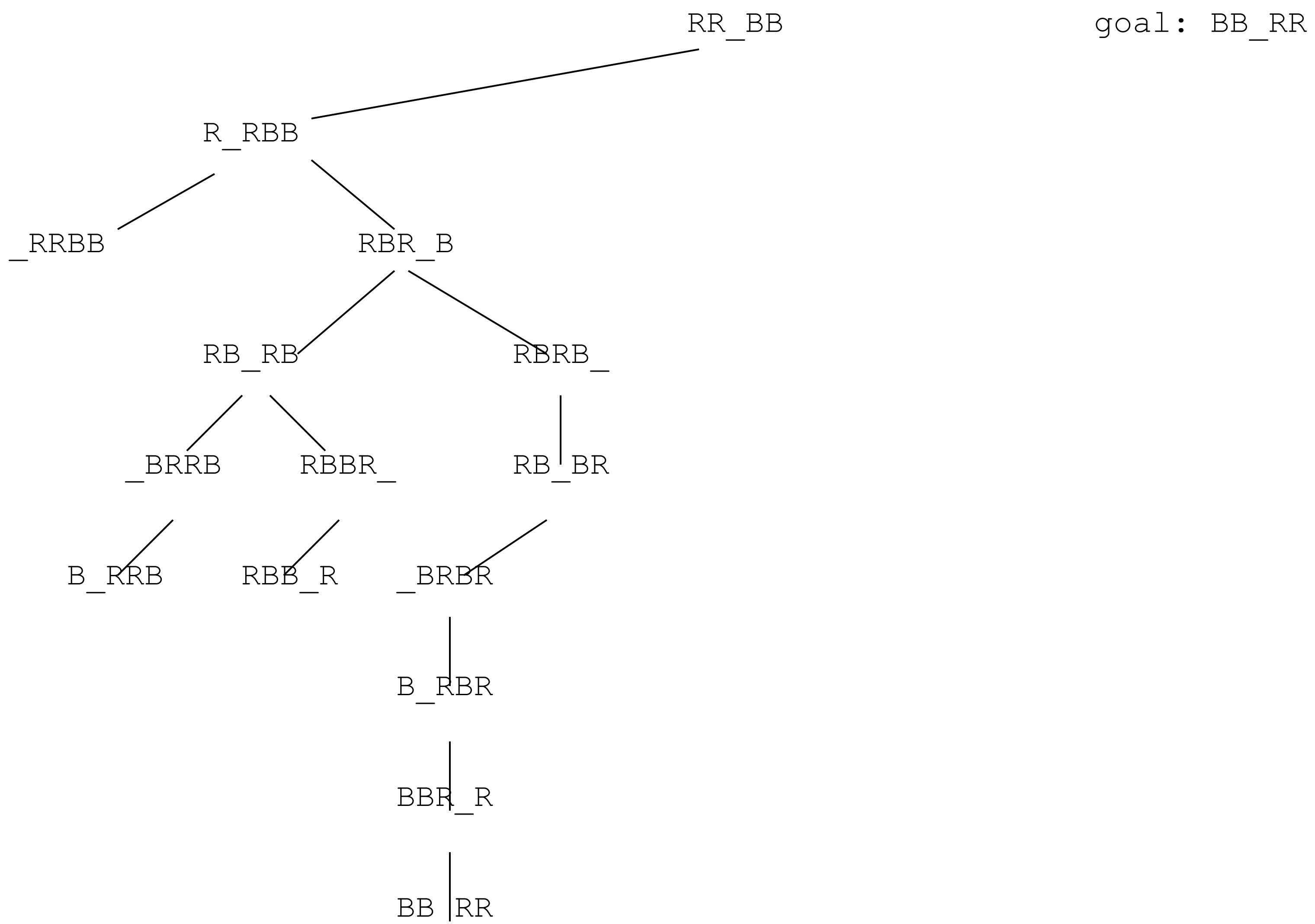

RR_BB
goal: BB_RR
R_RBB
_RRBB
RBR_B
RB_RB
RBRB_
_BRRB
RBBR_
RB_BR
B_RRB
RBB_R
_BRBR
B_RBR

Four pegs

goal: BB_RR



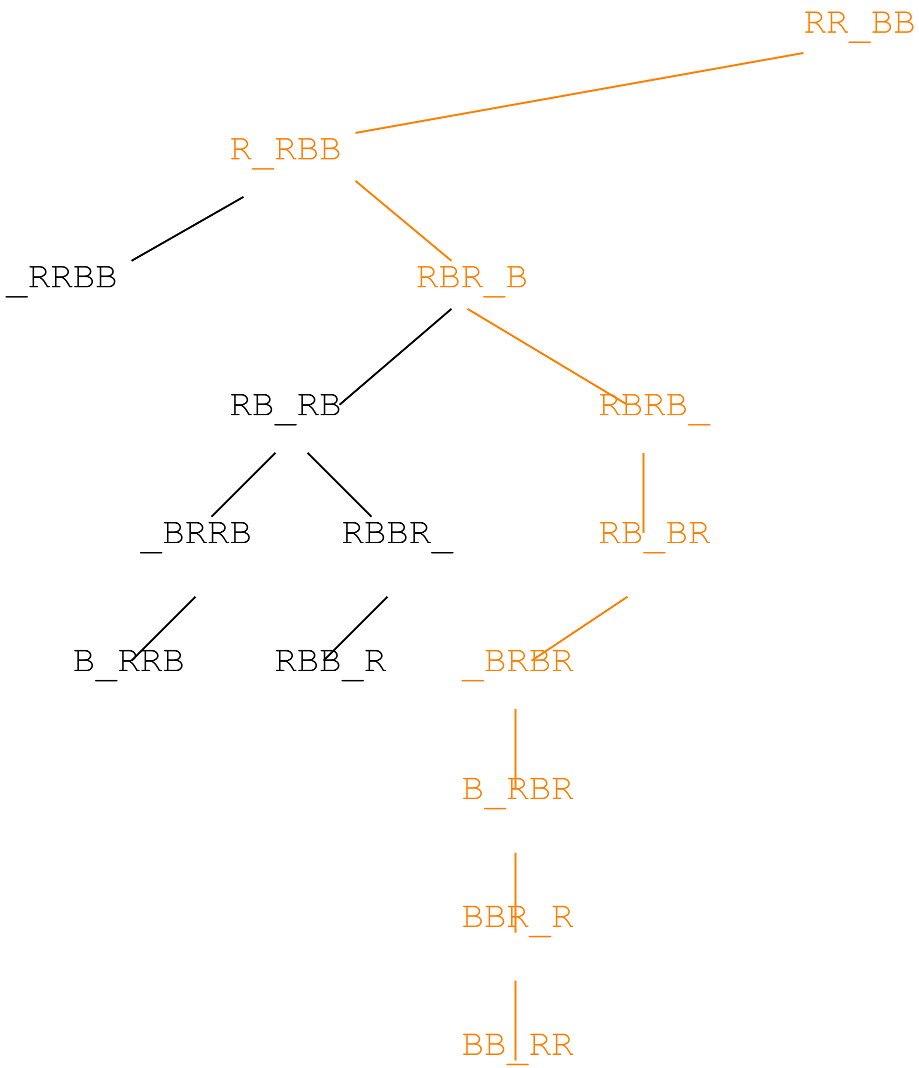
Four pegs



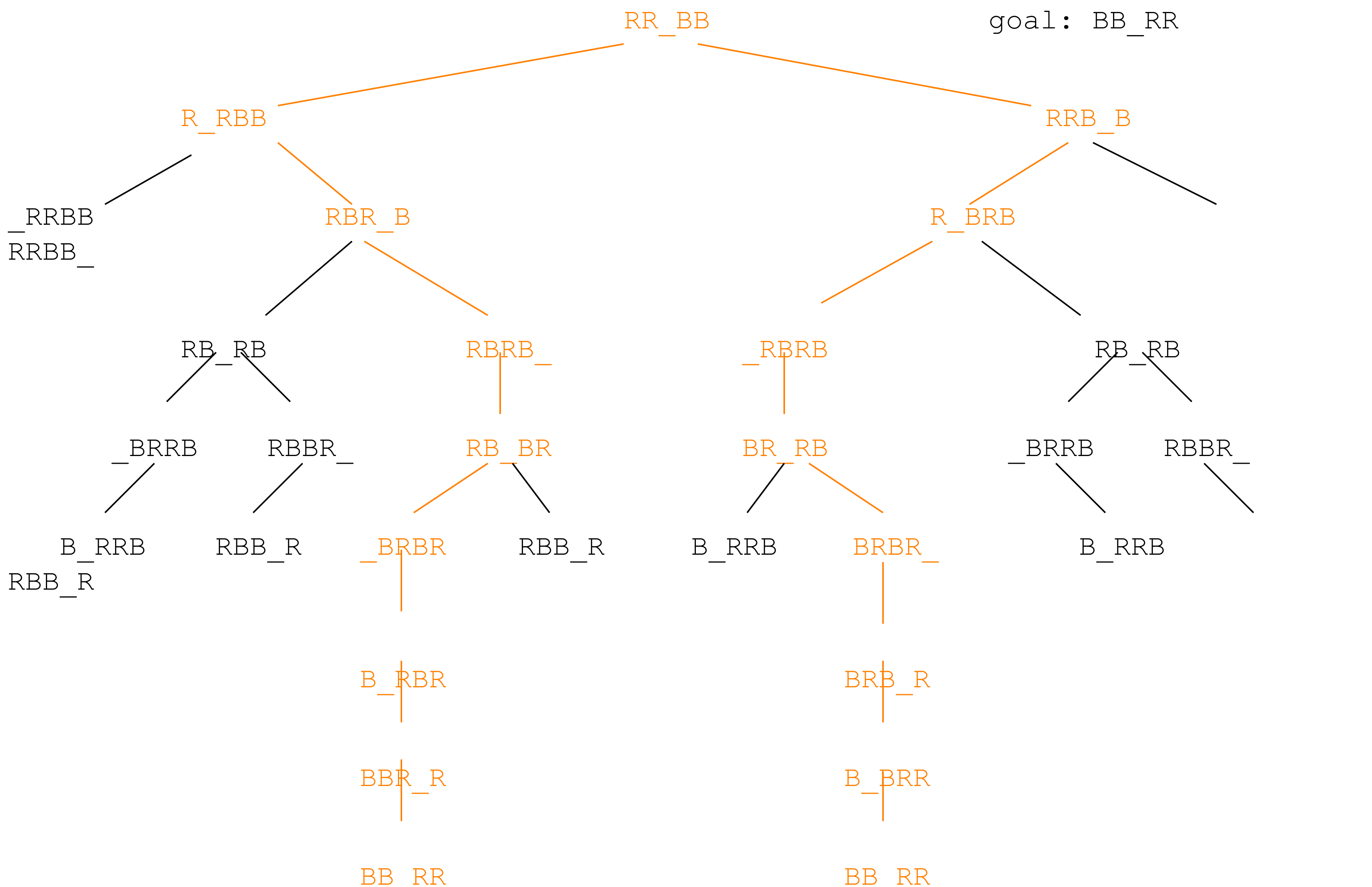
RR_BB
goal: BB_RR
R_RBB
_RRBB
RBR_B
RB_RB
RBRB_
_BRRB
RBBR_
RB_BR
B_RRB
RBB_R
_BRBR
B_RBR
BBR_R
BB_RR

Four pegs

goal: BB_RR



Four pegs


RR_BB
goal: BB_RR
R_RBB
RRB_B
_RRBB
RRBB_
RBR_B
R_BRB
RB_RB
RBRB_
_RBRB
RB_RB
_BRRB
RBBR_
RB_BR
BR_RB
_BRRB
RBBR_
B_RRB
RBB_R
RBB_R
_BRBR
RBB_R
B_RRB
BRBR_
B_RRB
B_RBR
BRB_R
BBR_R
B_BRR
BB_RR
BB_RR

State-space search

state-space-search (list-of-unexplored-states, goal-state, operators)

1. look at the first (leftmost) unexplored-state
2. if that state is the goal-state, then return success
3. if that state isn't the goal-state, then generate all possible new states from that state by applying the set of operators to that state
4. call state-space-search with this new list of states passed as the unexplored-states argument, and if that succeeds then return success else...
5. call state-space-search with the old list of unexplored-states that remained after you stripped off the first unexplored-state in step 1, and if that succeeds then return success else...
6. return failure

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
   | null unexplored                = []
   | goal == head unexplored        = goal:path
   | (not (null newstates))         = newstates
   | otherwise                      =
      statesearch (tail unexplored) goal path
     where newstates = statesearch
                        (generateNewStates (head unexplored))
                        goal
                        ((head unexplored):path)
```

Here's how that outline can be implemented in Haskell for the peg puzzle. It's not complete as you can tell, we'll see the complete implementation a little later..

Four pegs in Haskell

```
RR_BB                    goal: BB_RR
```

Four pegs in Haskell

`RR_BB`

`goal: BB_RR`

Is this the goal?

Four pegs in Haskell

`RR_BB`

`goal: BB_RR`

Is this the goal?

Can we generate new states?

Four pegs in Haskell

```
                RR_BB            goal: BB_RR
R_RBB                      RRB_B
```

Is this the goal?

Can we generate new states?

Four pegs in Haskell

```
RR_BB                goal: BB_RR

R_RBB         RRB_B
```

Is this the goal?

Can we generate new states?

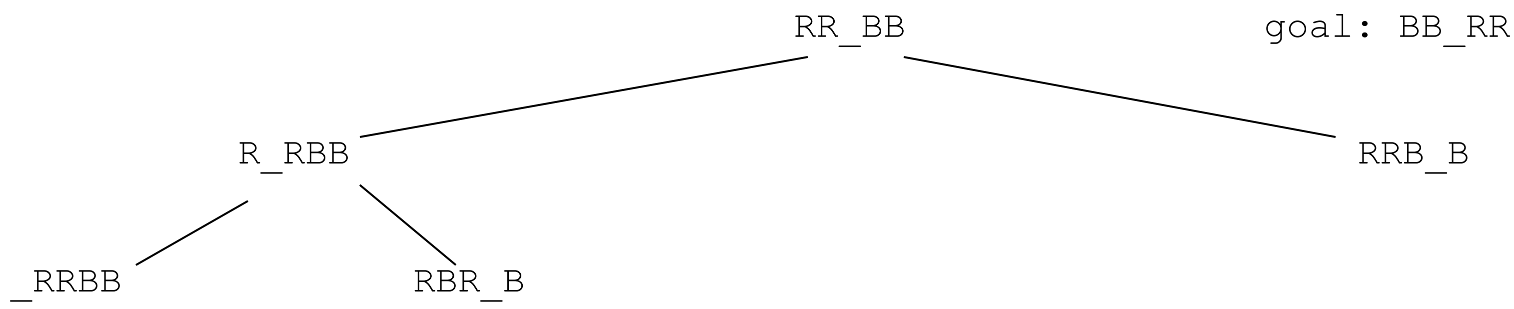
Four pegs in Haskell

```
                    RR_BB                    goal: BB_RR

R_RBB                                 RRB_B
```

Is this the goal?

Can we generate new states?

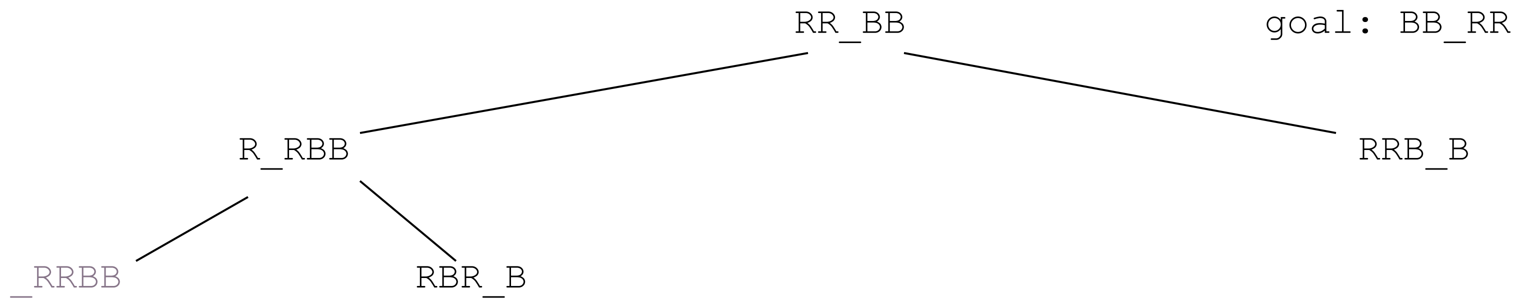
Four pegs in Haskell



Is this the goal?

Can we generate new states?

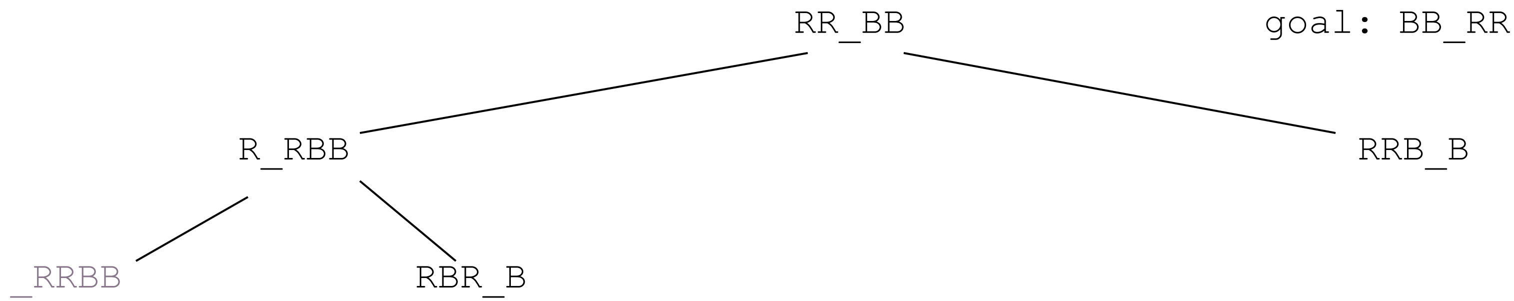
Four pegs in Haskell



Is this the goal?

Can we generate new states?

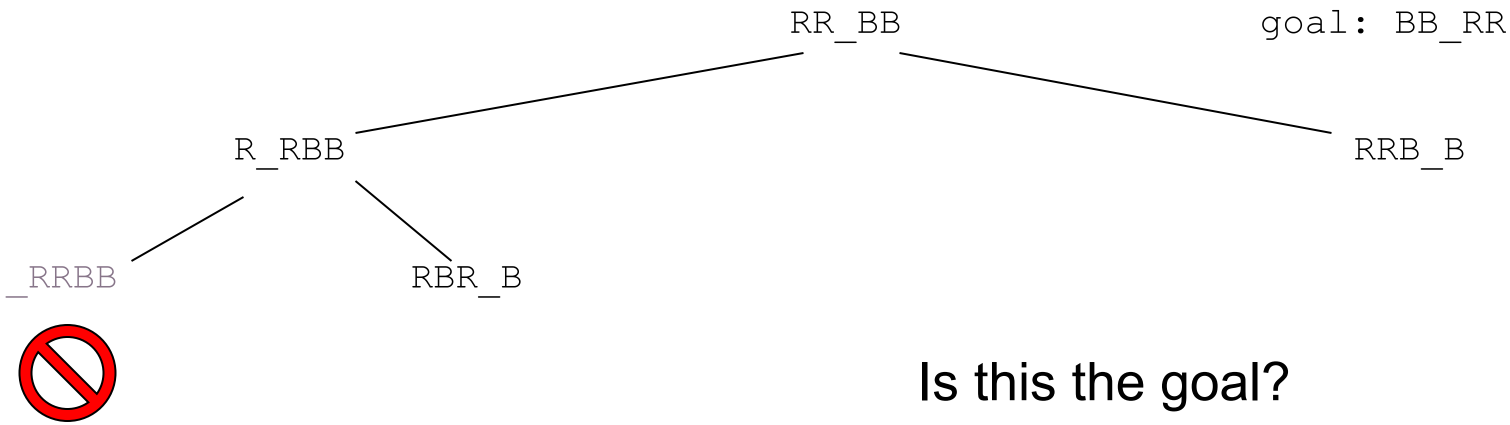
Four pegs in Haskell



Is this the goal?

Can we generate new states?

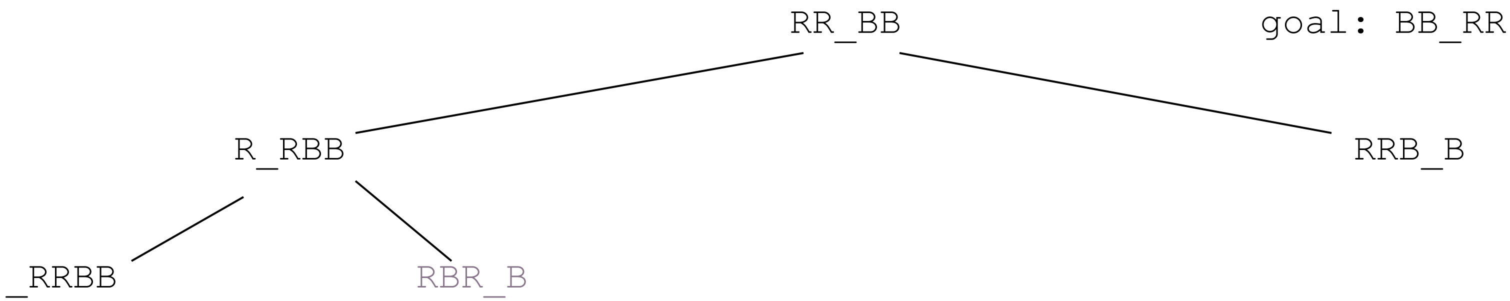
Four pegs in Haskell



Is this the goal?

Can we generate new states?

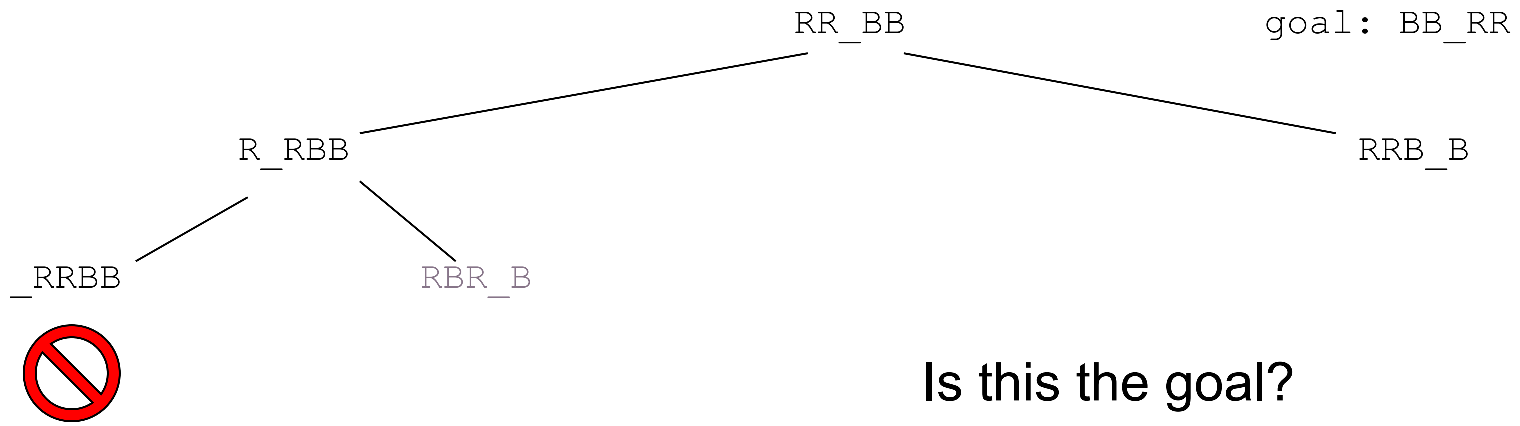
Four pegs in Haskell



Is this the goal?

Can we generate new states?

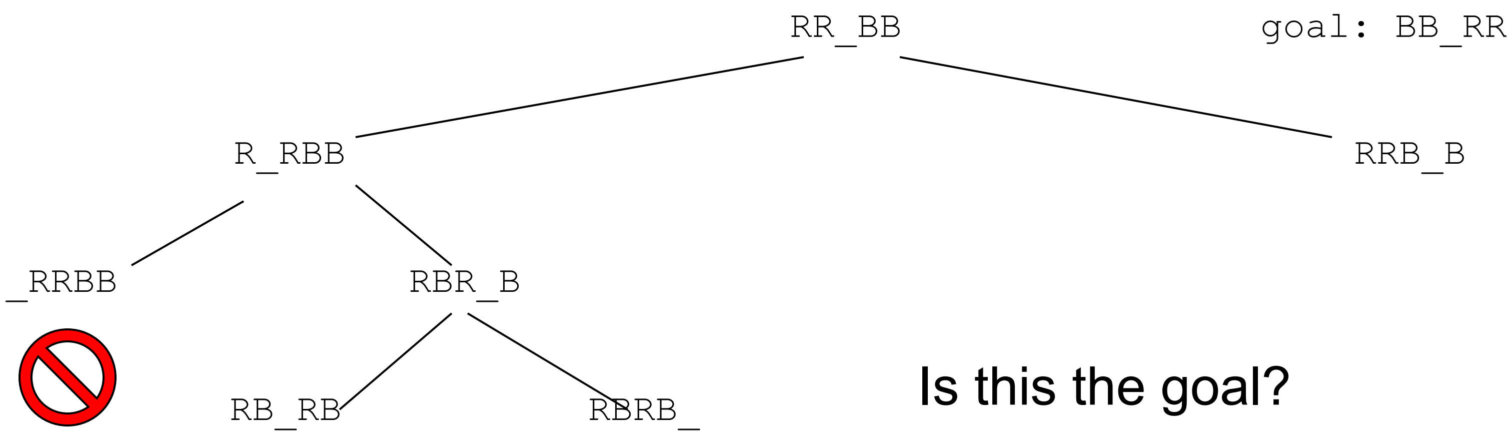
Four pegs in Haskell



Is this the goal?

Can we generate new states?

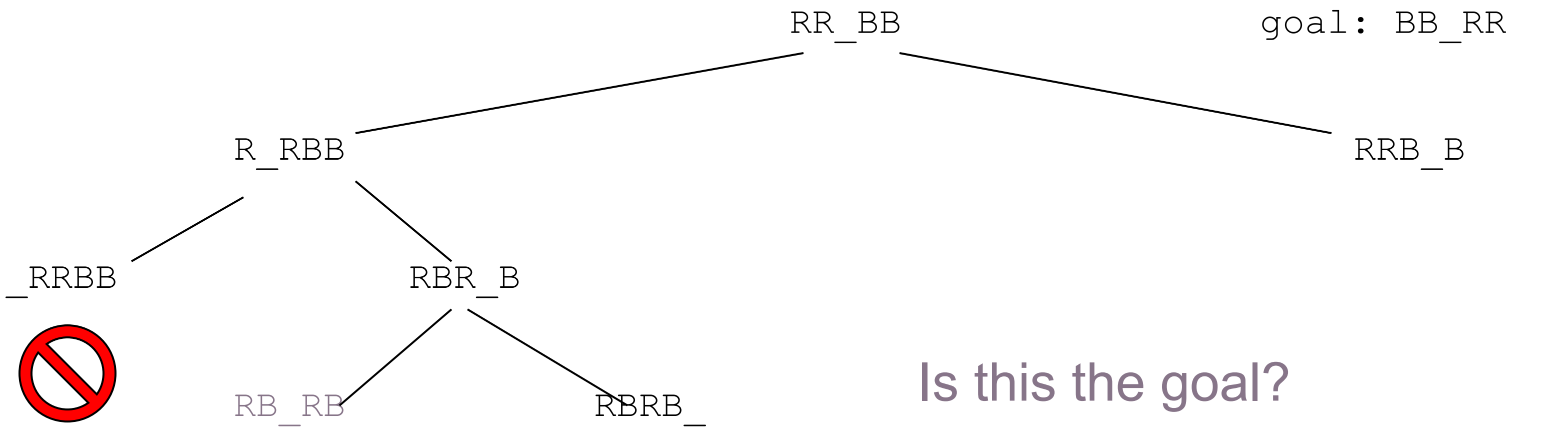
Four pegs in Haskell



Is this the goal?

Can we generate new states?

Four pegs in Haskell

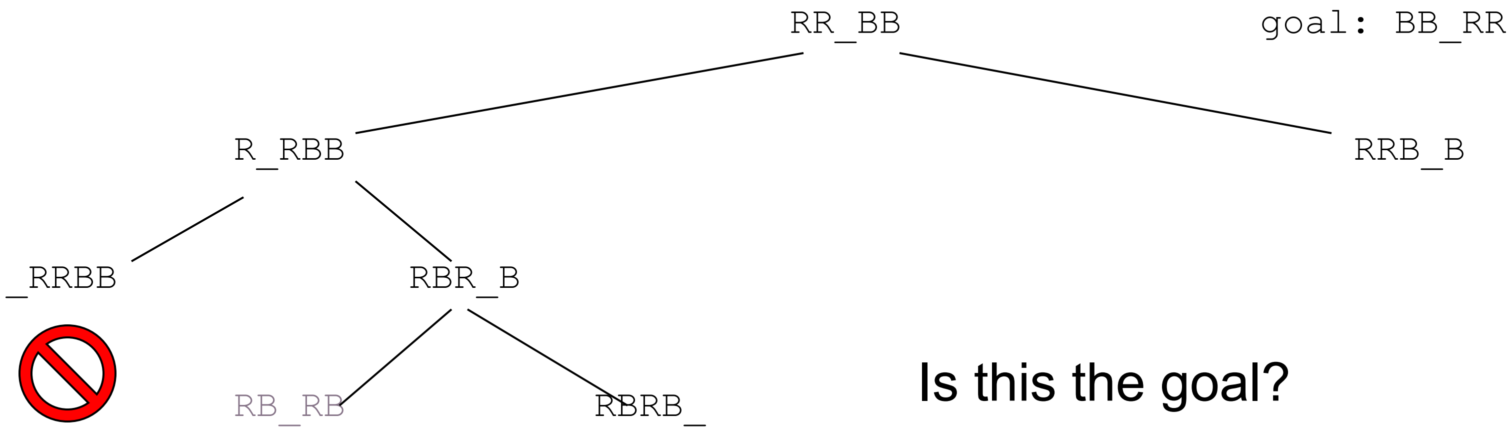


goal: BB_RR

Is this the goal?

Can we generate new states?

Four pegs in Haskell

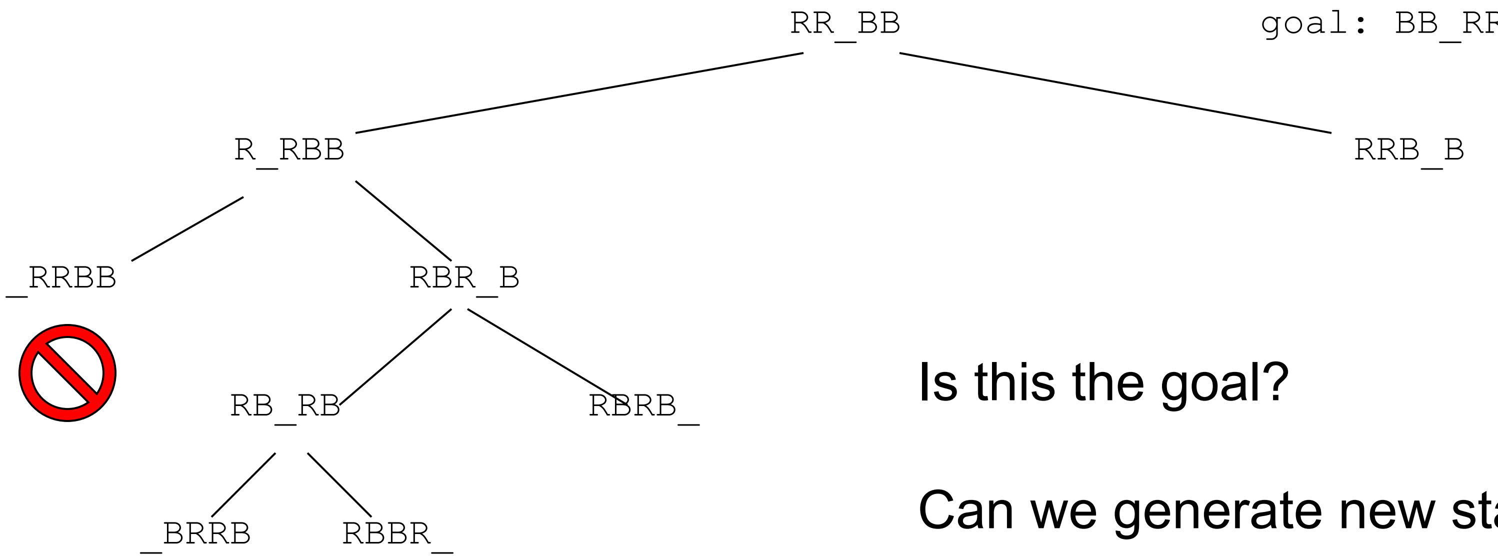


goal: BB_RR

Is this the goal?

Can we generate new states?

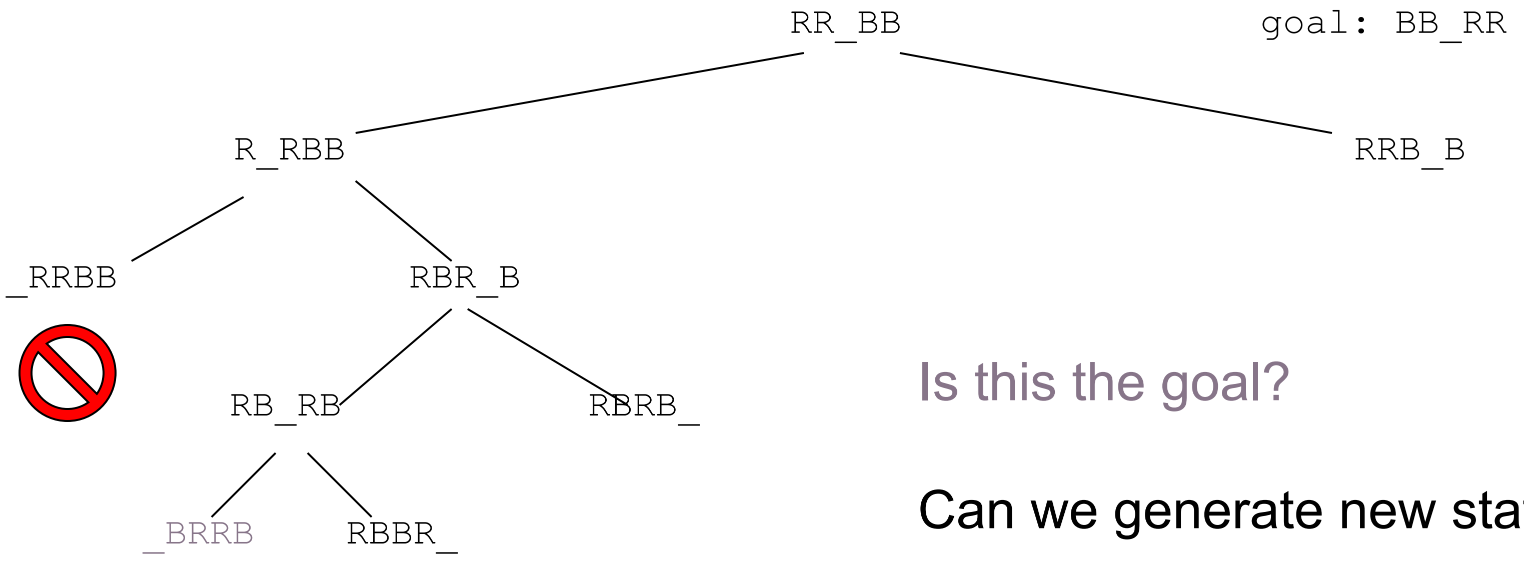
Four pegs in Haskell



Is this the goal?

Can we generate new states?

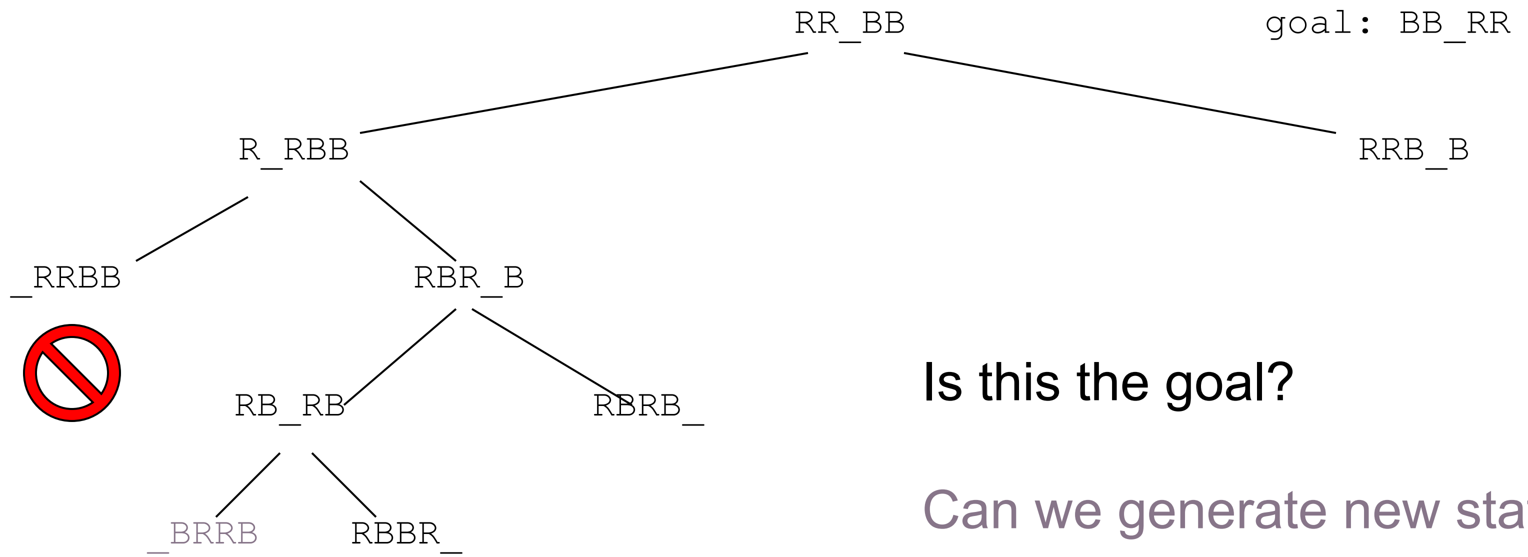
Four pegs in Haskell



Is this the goal?

Can we generate new states?

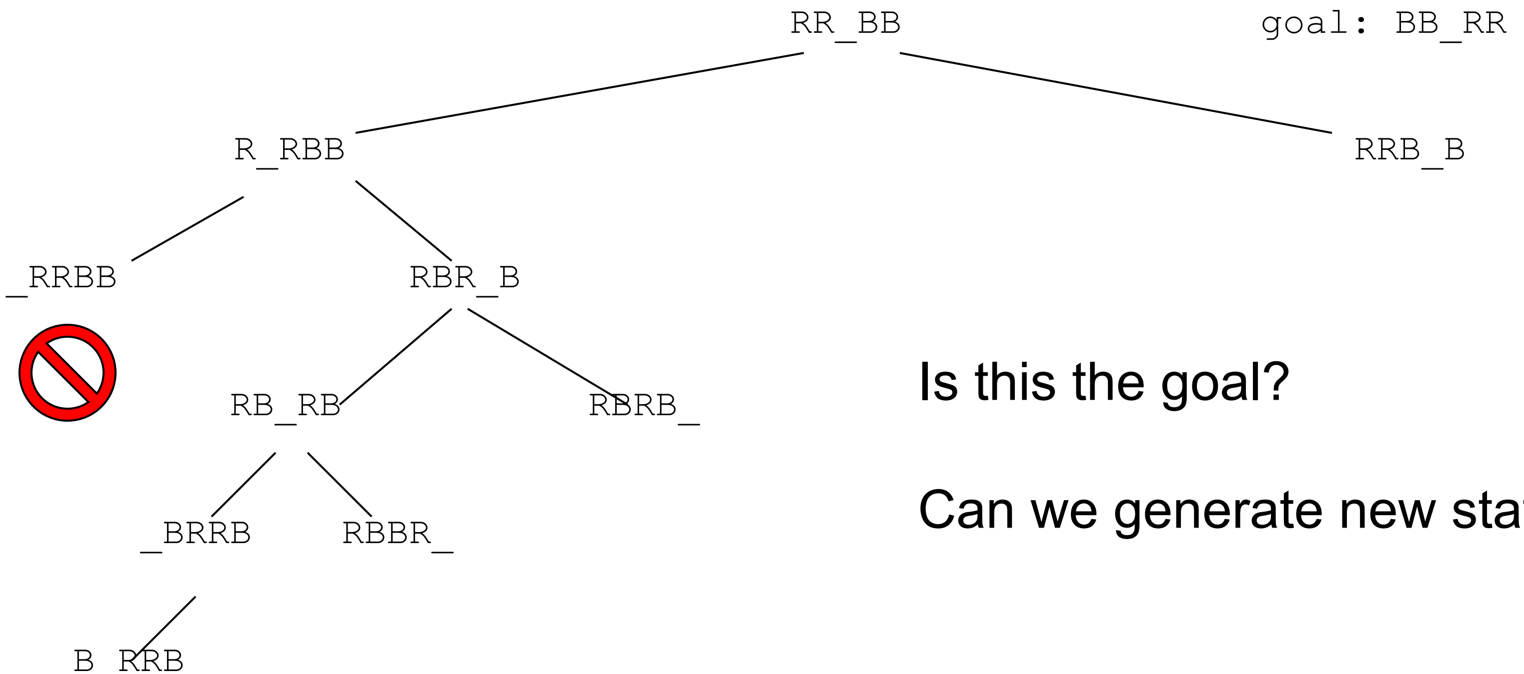
Four pegs in Haskell



Is this the goal?

Can we generate new states?

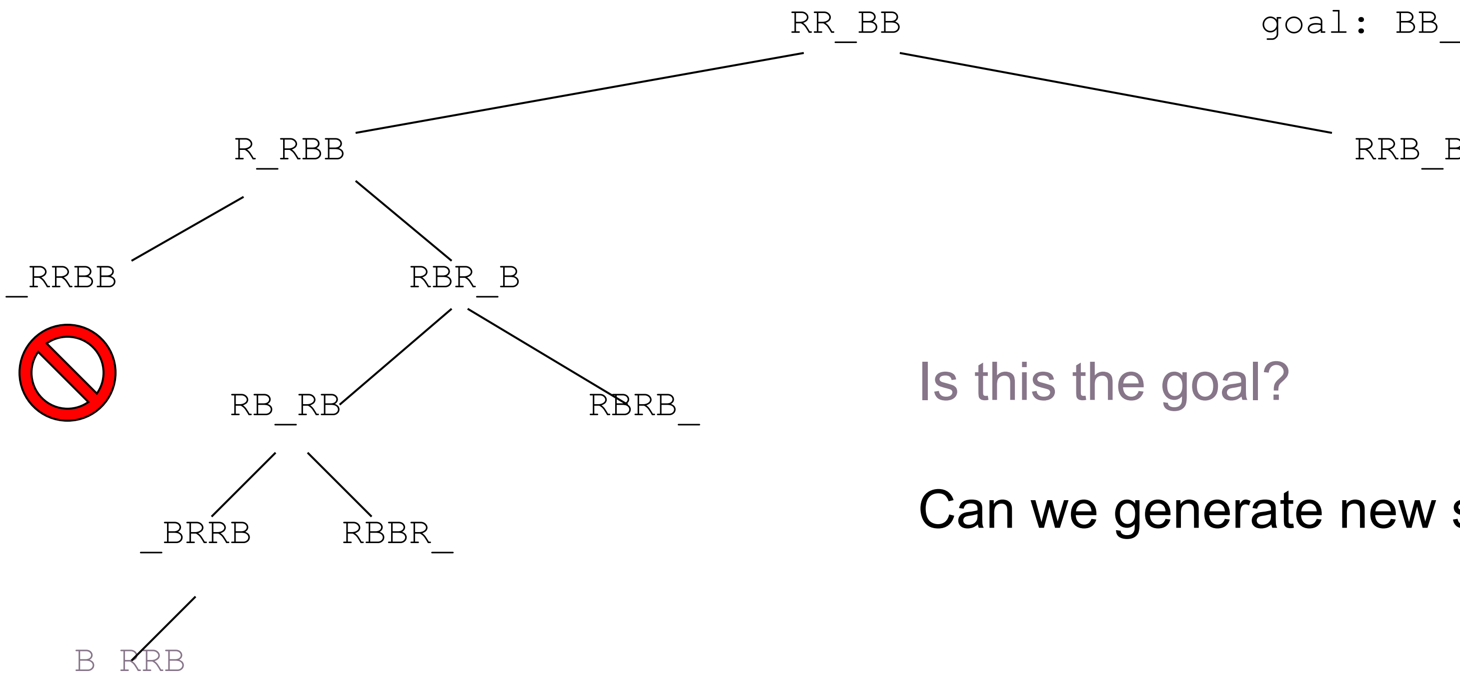
Four pegs in Haskell



Is this the goal?

Can we generate new states?

Four pegs in Haskell

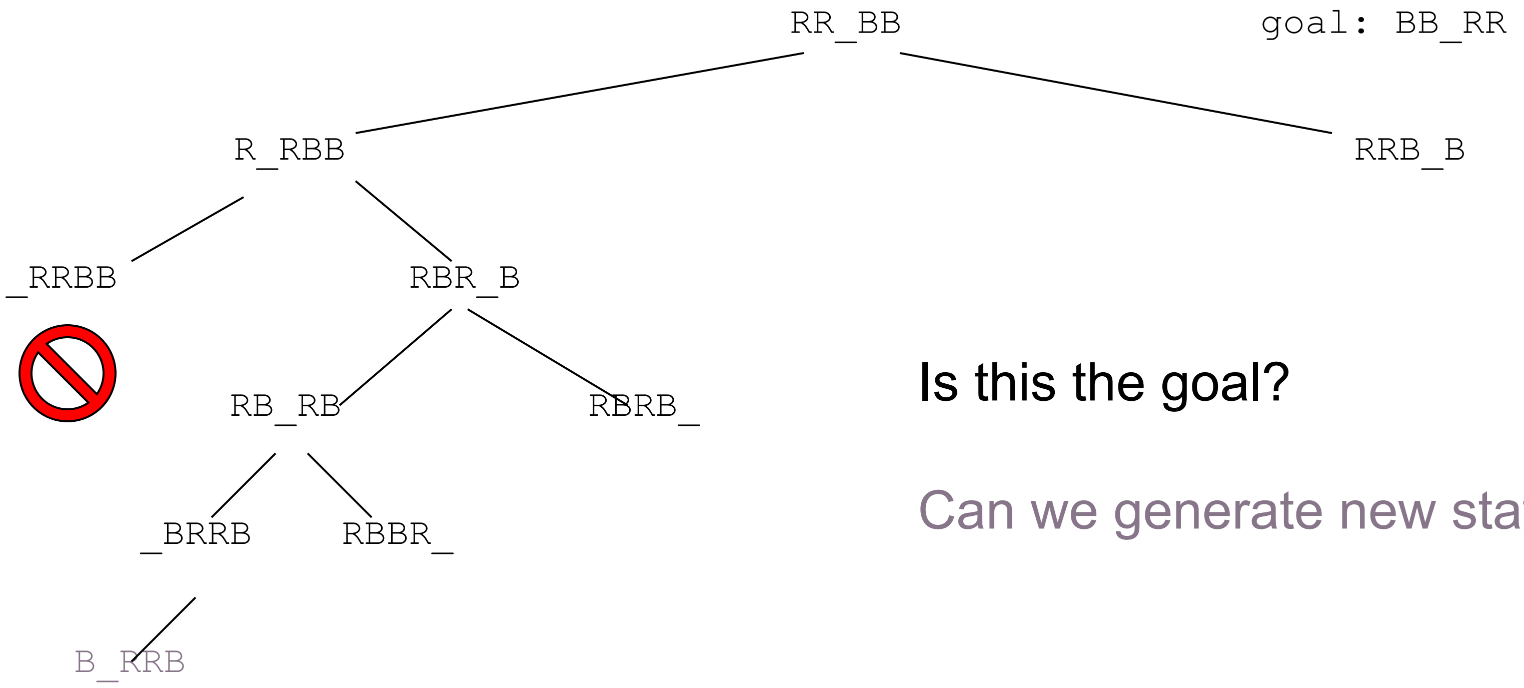


goal: BB_RR

Is this the goal?

Can we generate new states?

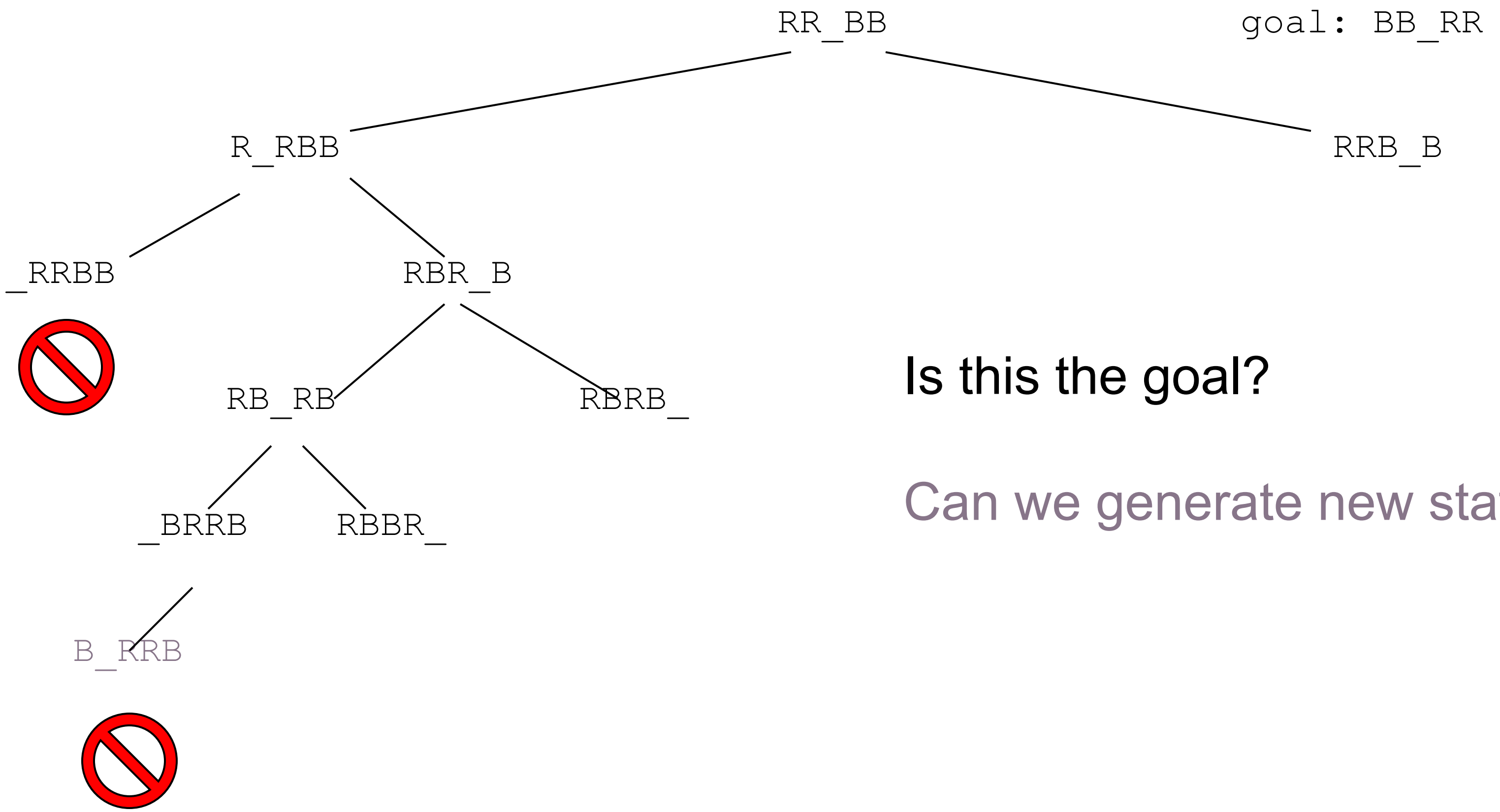
Four pegs in Haskell



Is this the goal?

Can we generate new states?

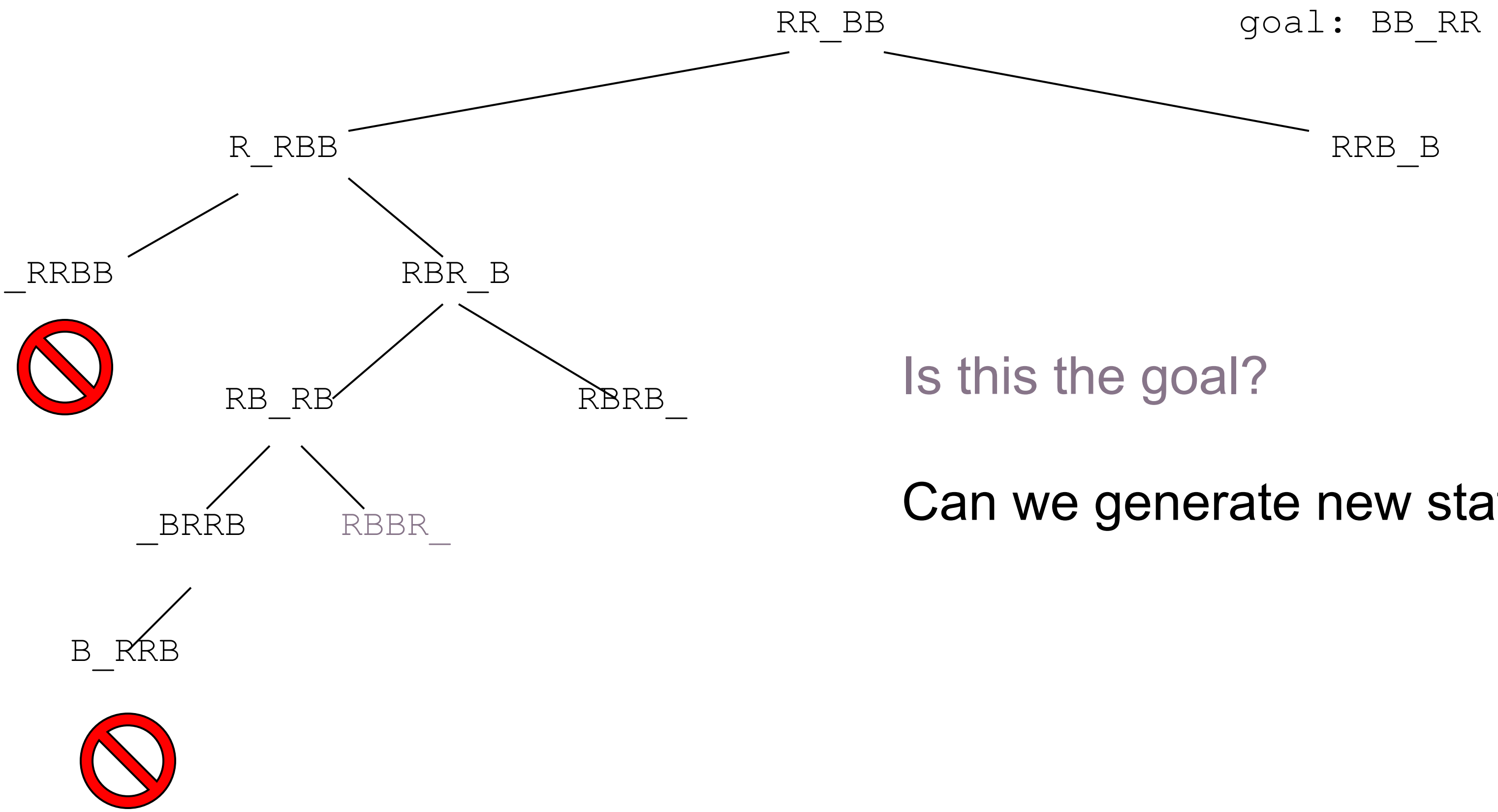
Four pegs in Haskell



Is this the goal?

Can we generate new states?

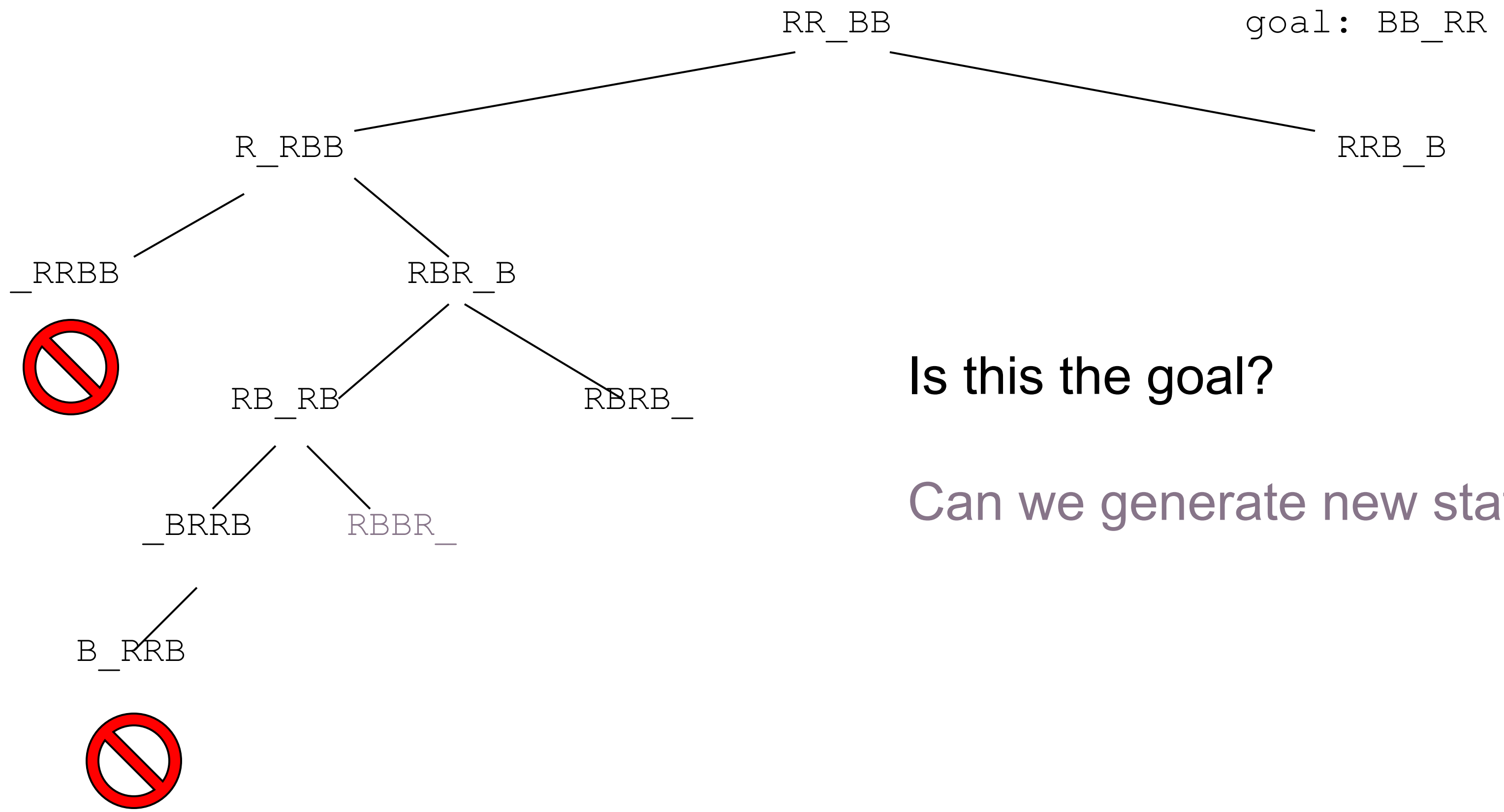
Four pegs in Haskell



Is this the goal?

Can we generate new states?

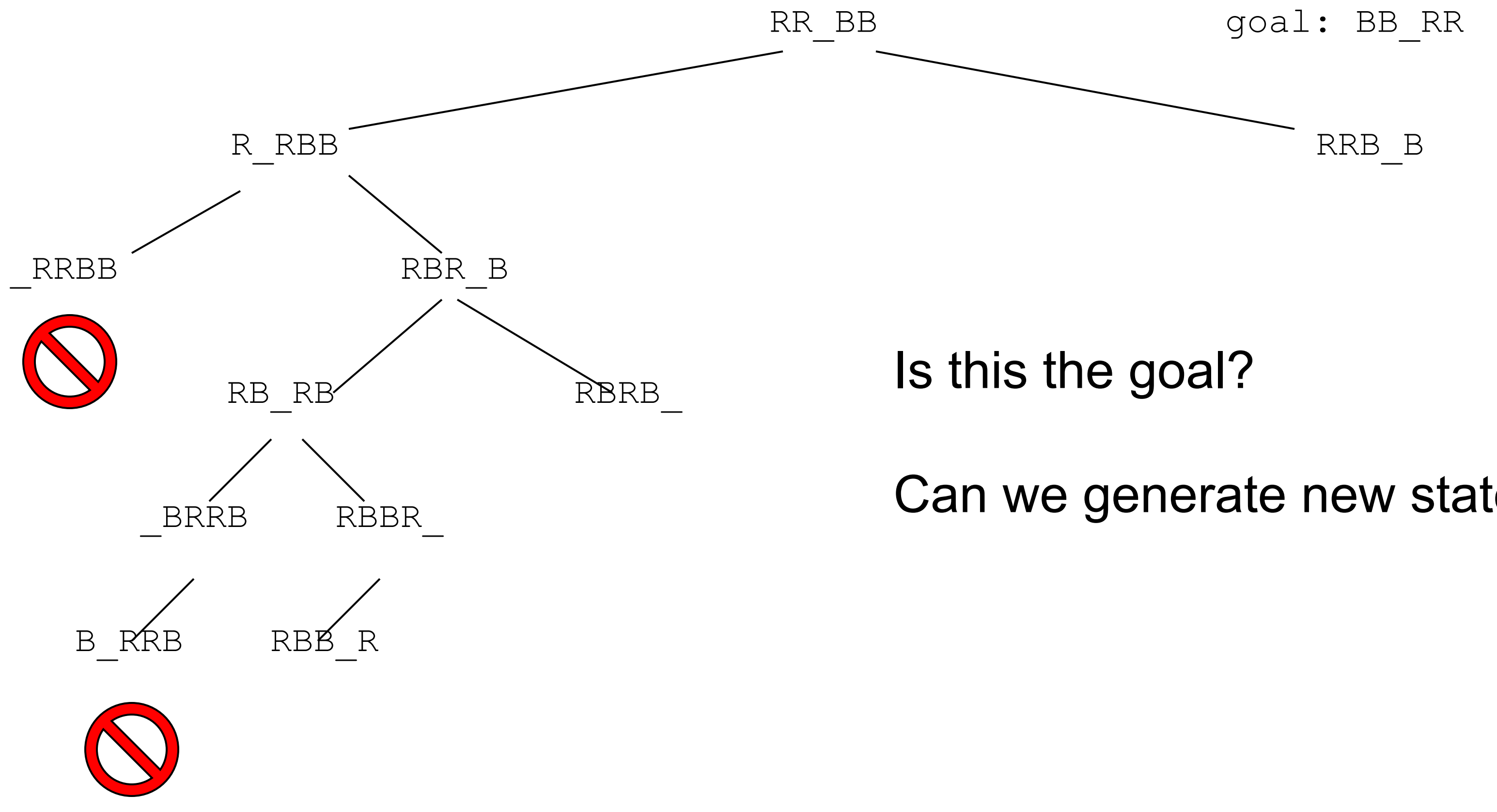
Four pegs in Haskell



Is this the goal?

Can we generate new states?

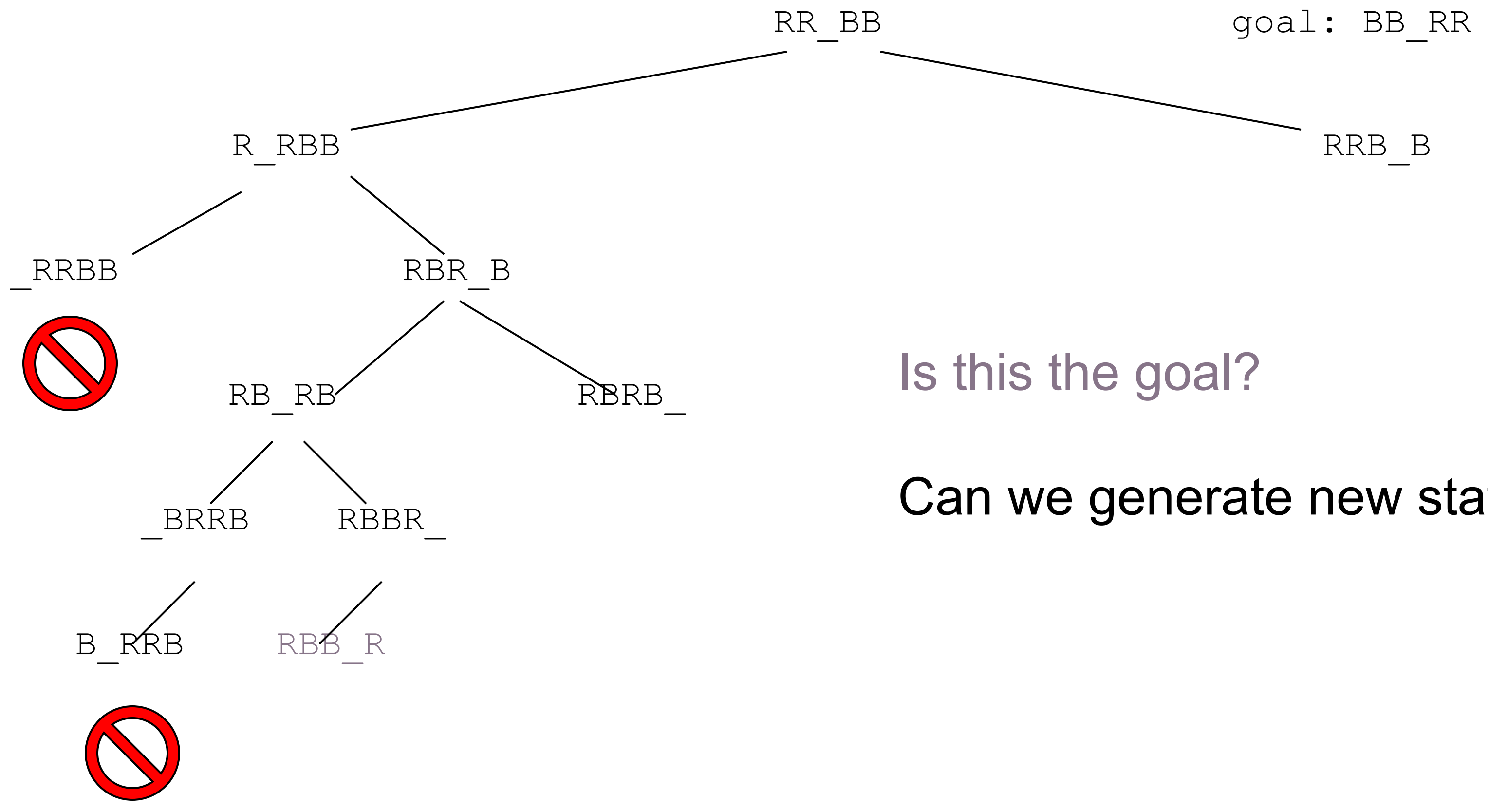
Four pegs in Haskell



Is this the goal?

Can we generate new states?

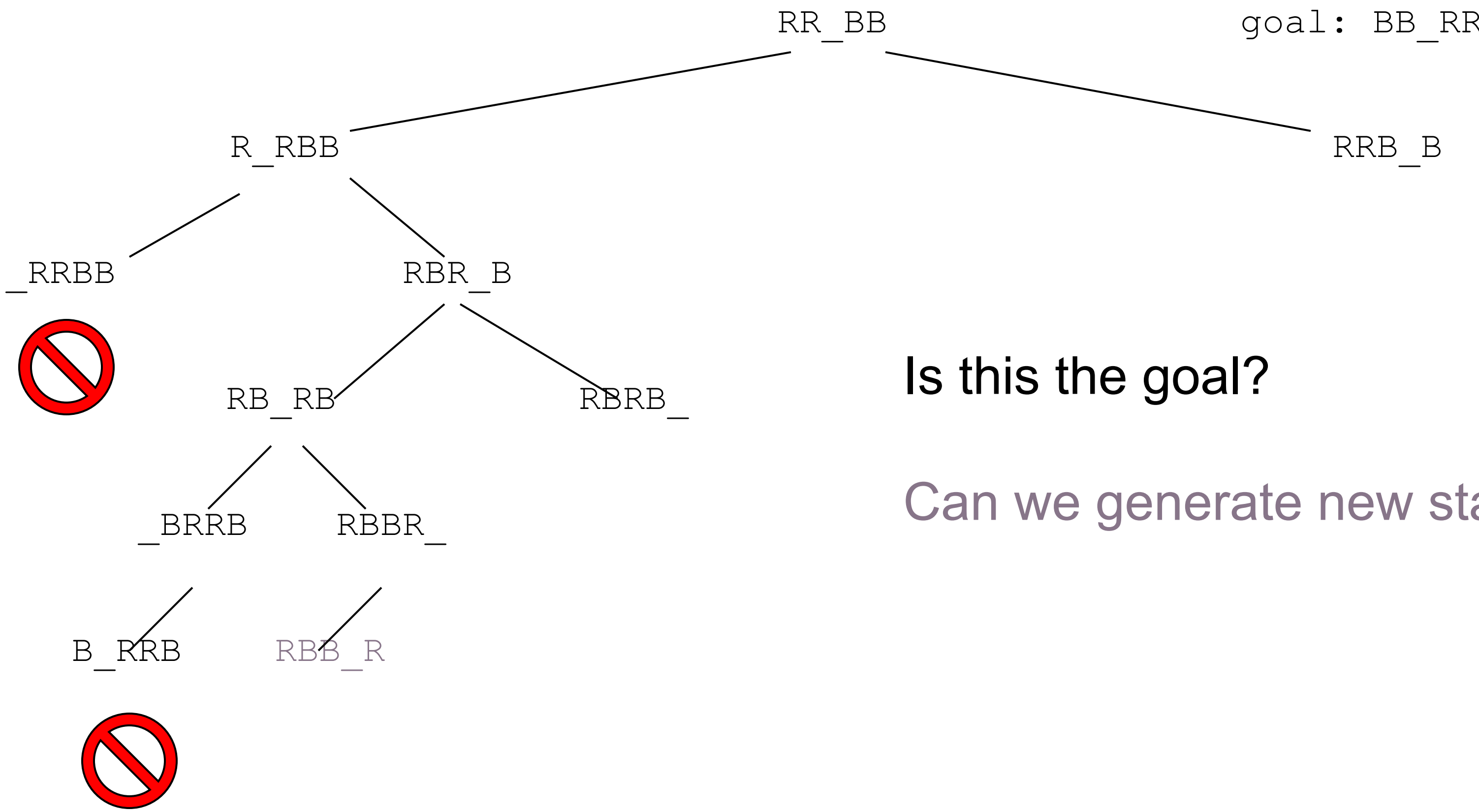
Four pegs in Haskell



Is this the goal?

Can we generate new states?

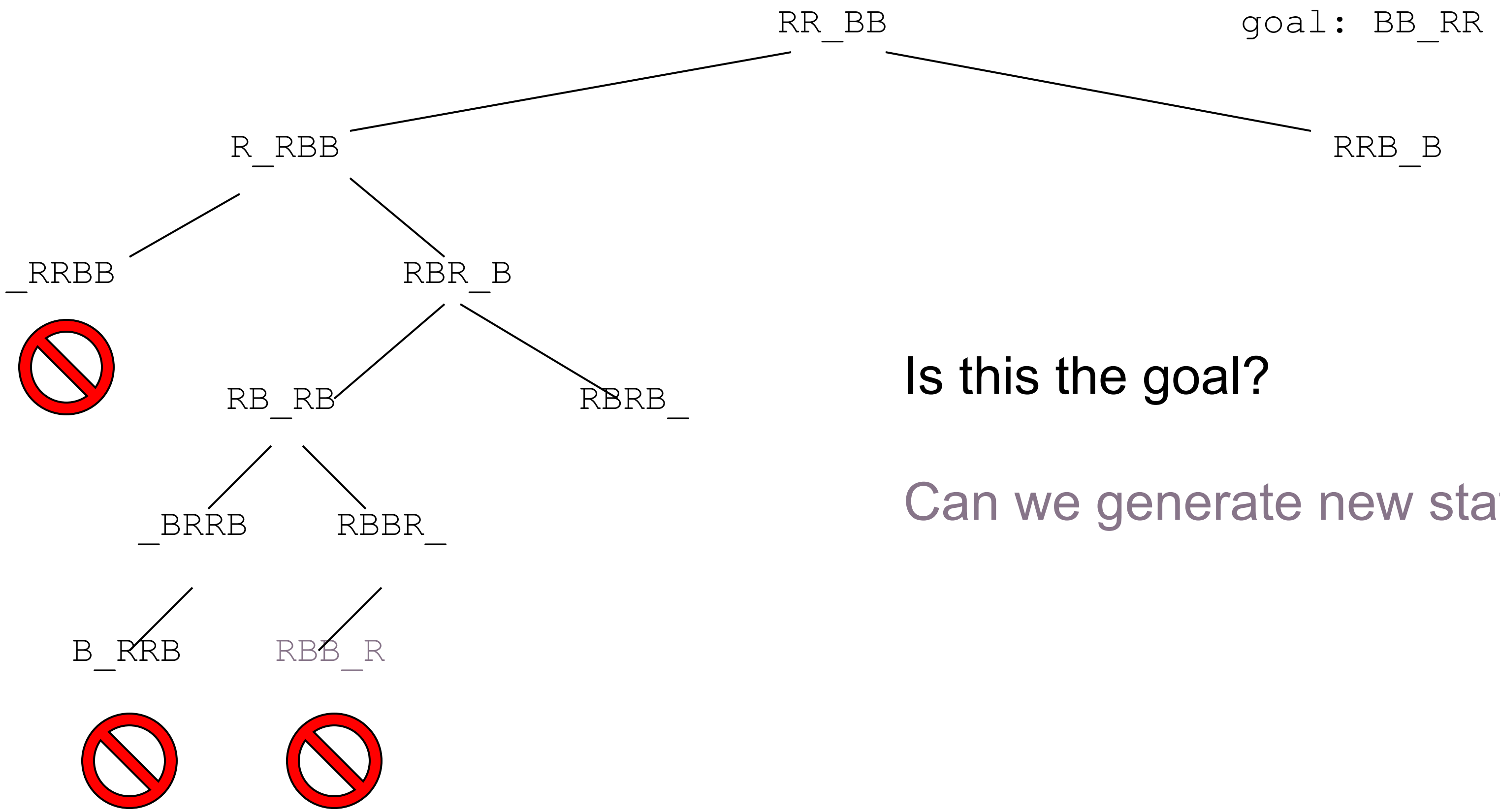
Four pegs in Haskell



Is this the goal?

Can we generate new states?

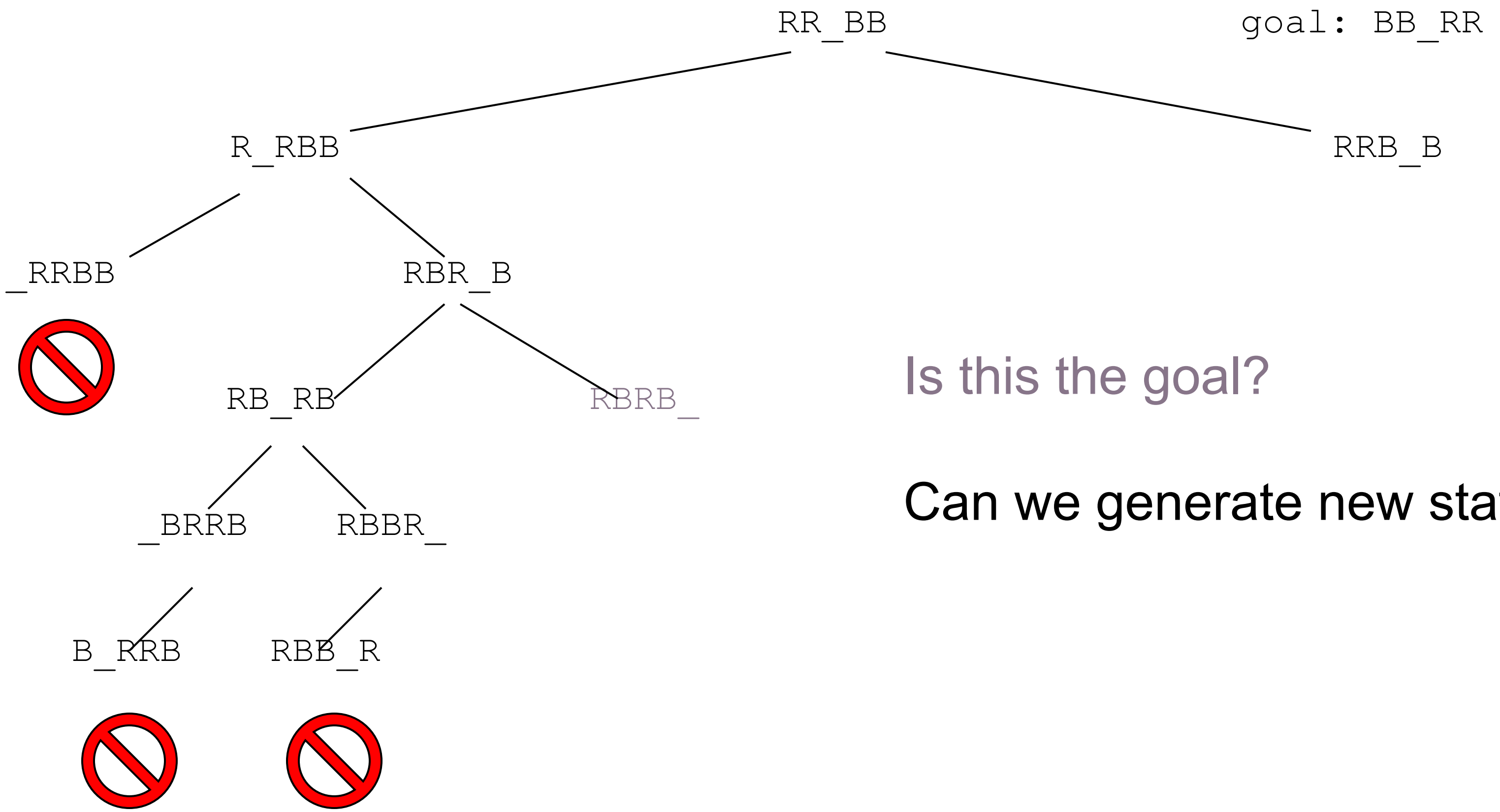
Four pegs in Haskell



Is this the goal?

Can we generate new states?

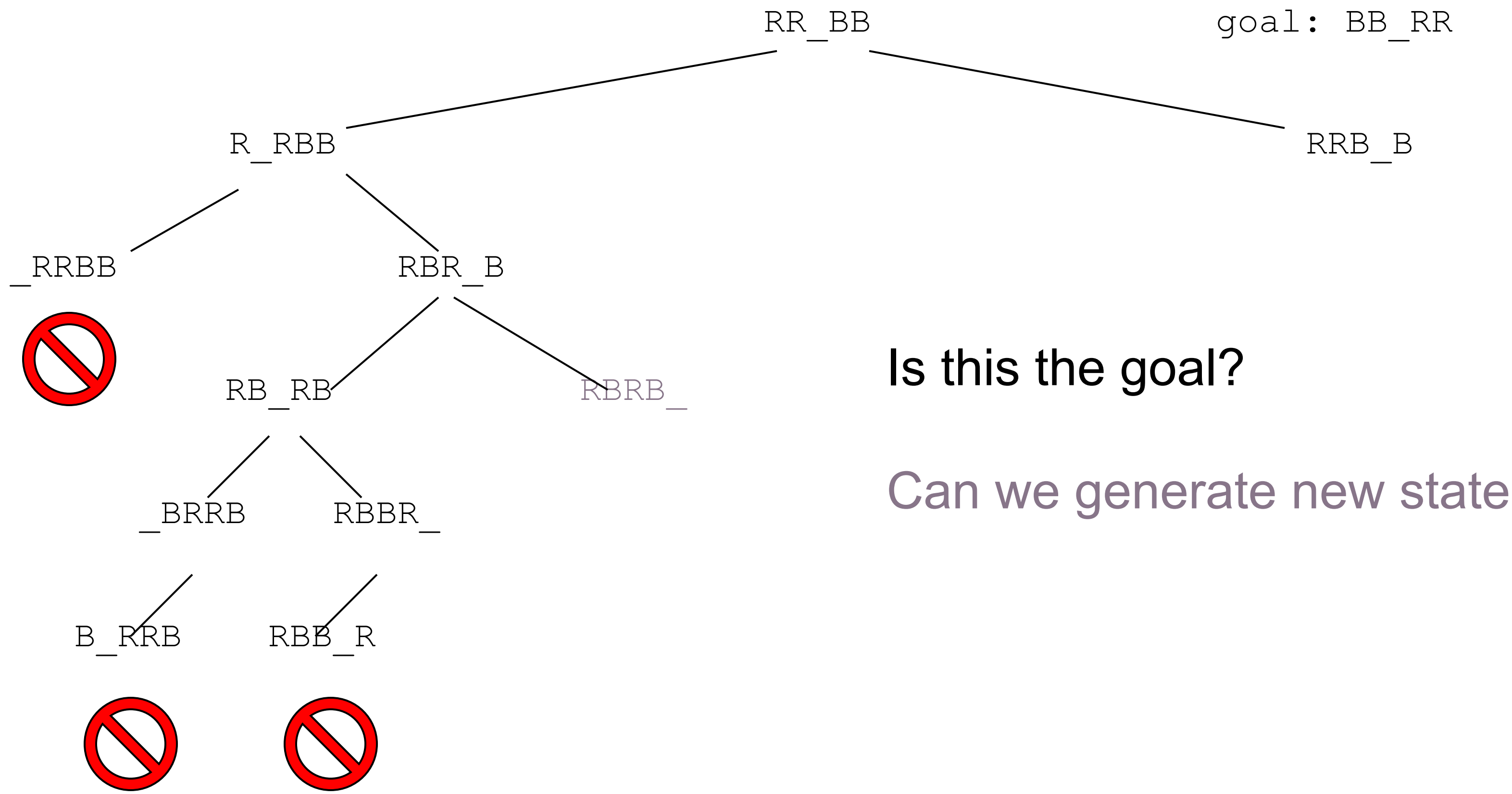
Four pegs in Haskell



Is this the goal?

Can we generate new states?

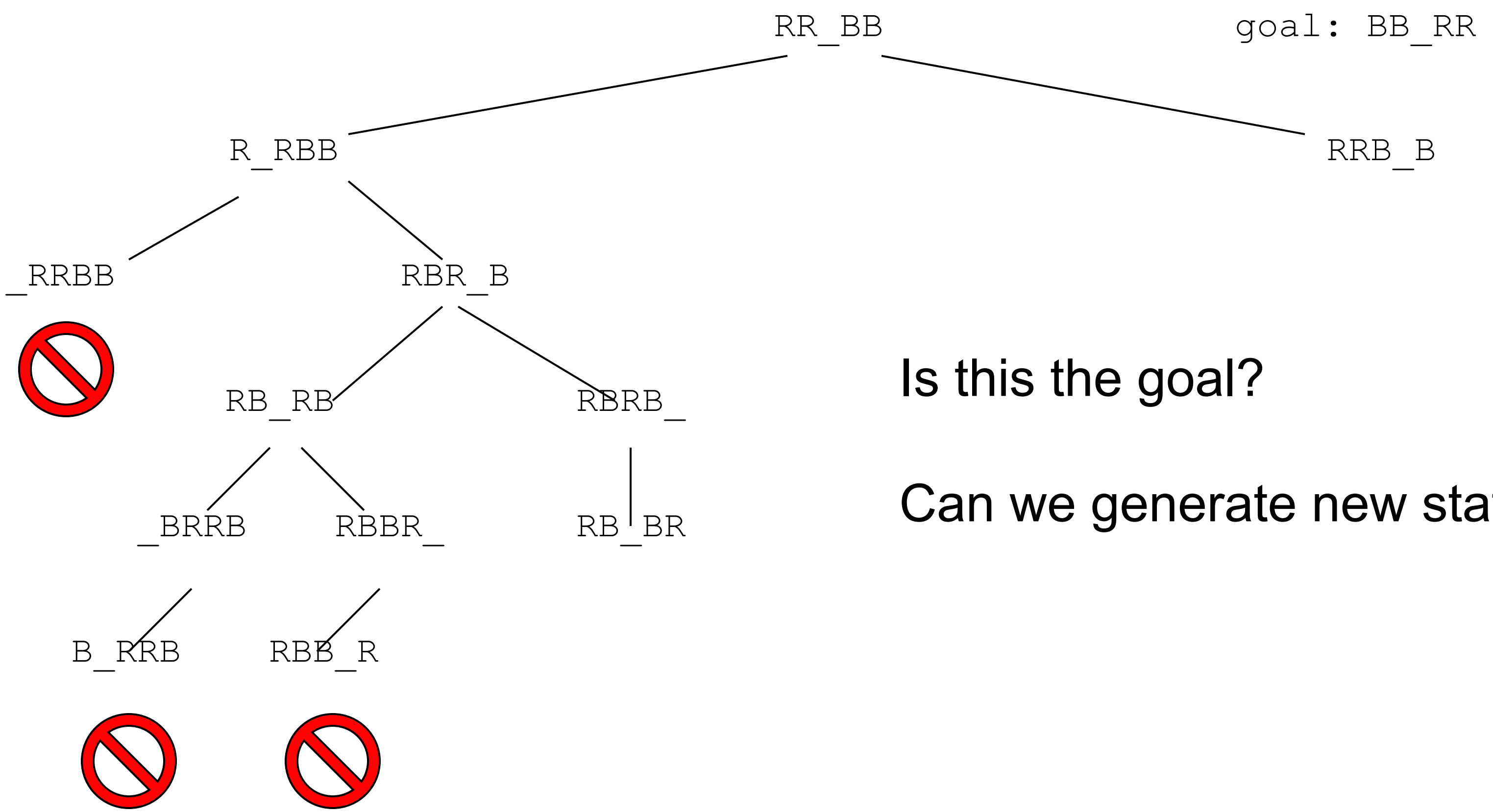
Four pegs in Haskell



Is this the goal?

Can we generate new states?

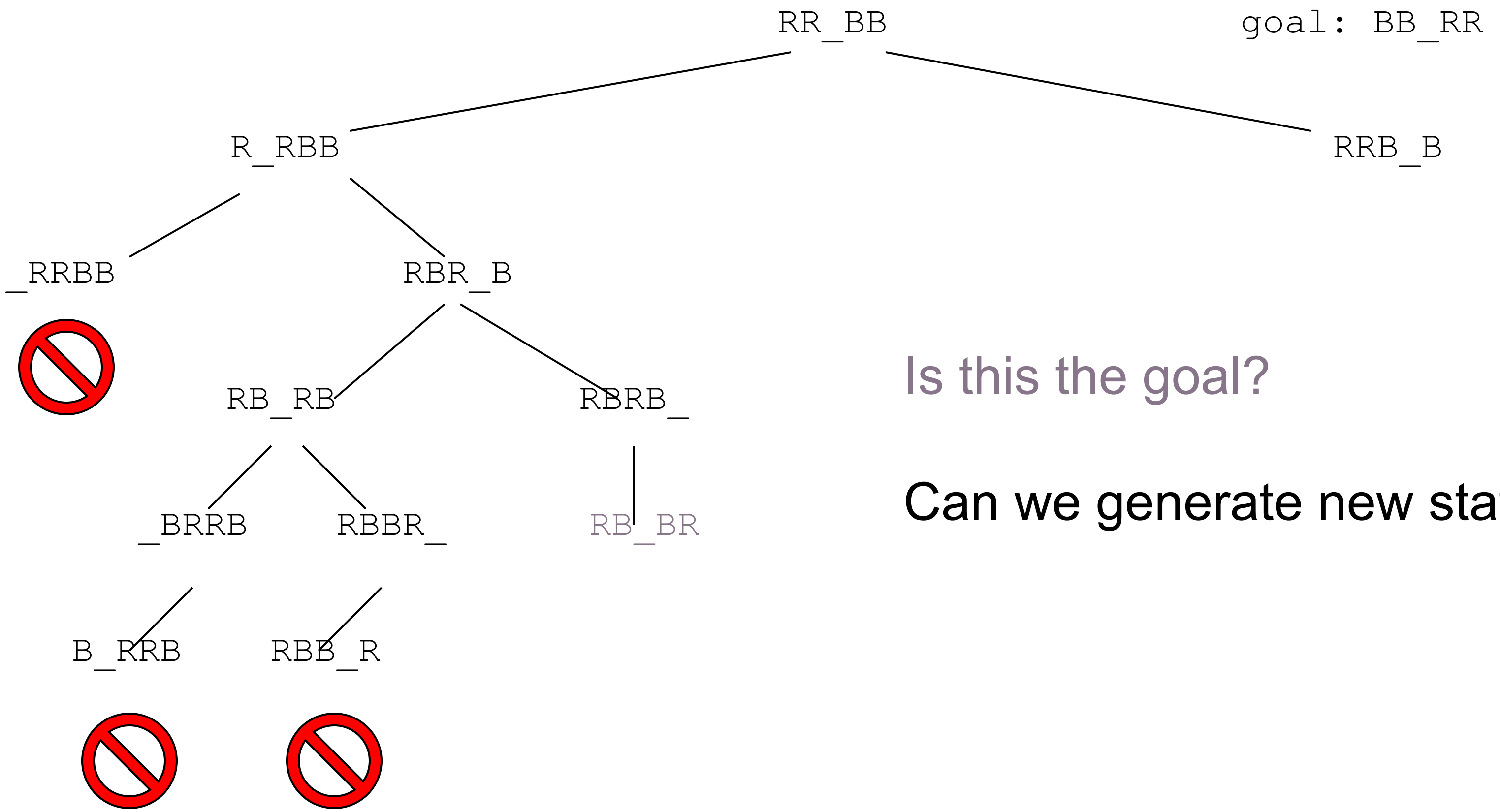
Four pegs in Haskell



Is this the goal?

Can we generate new states?

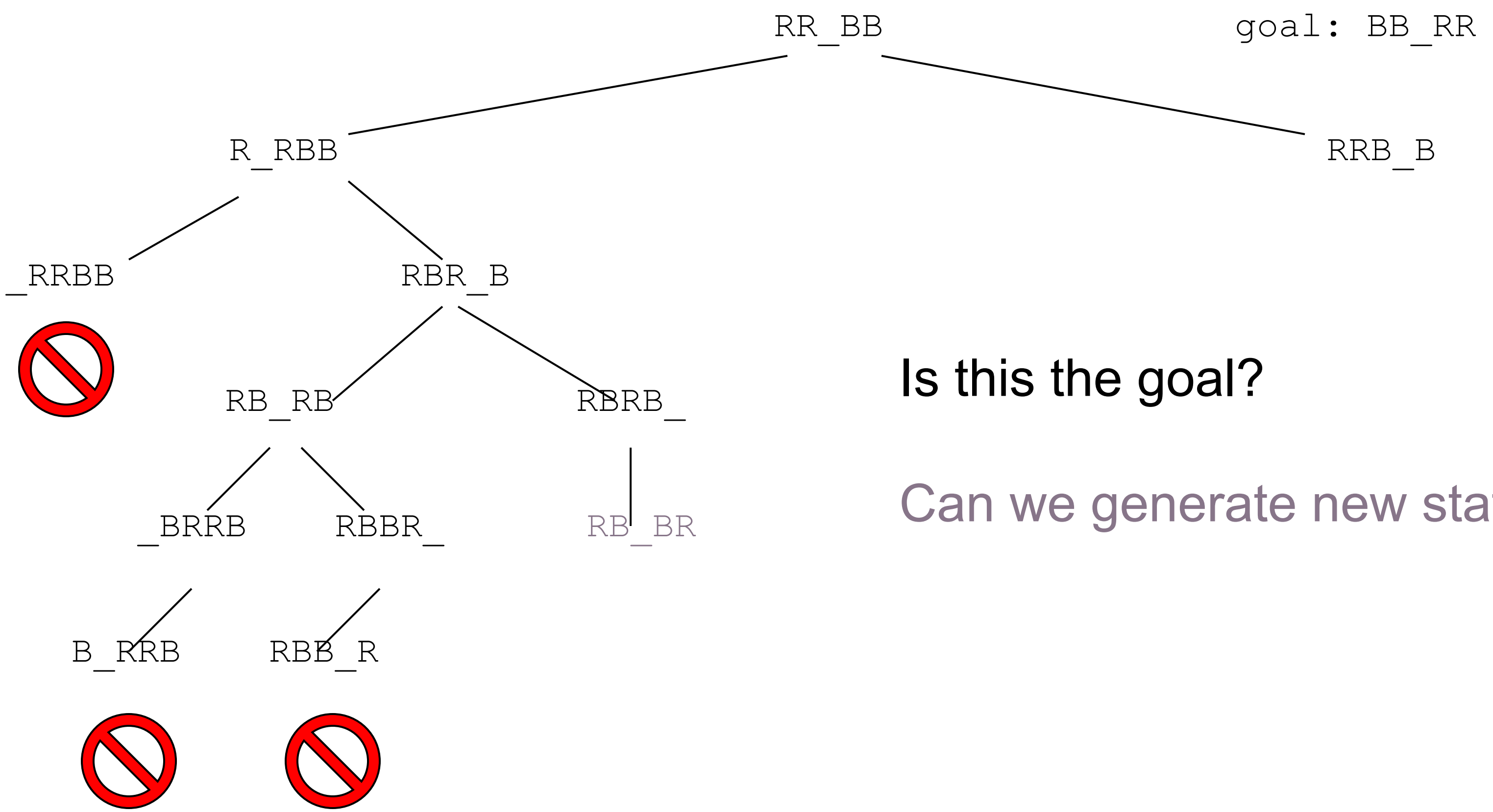
Four pegs in Haskell



Is this the goal?

Can we generate new states?

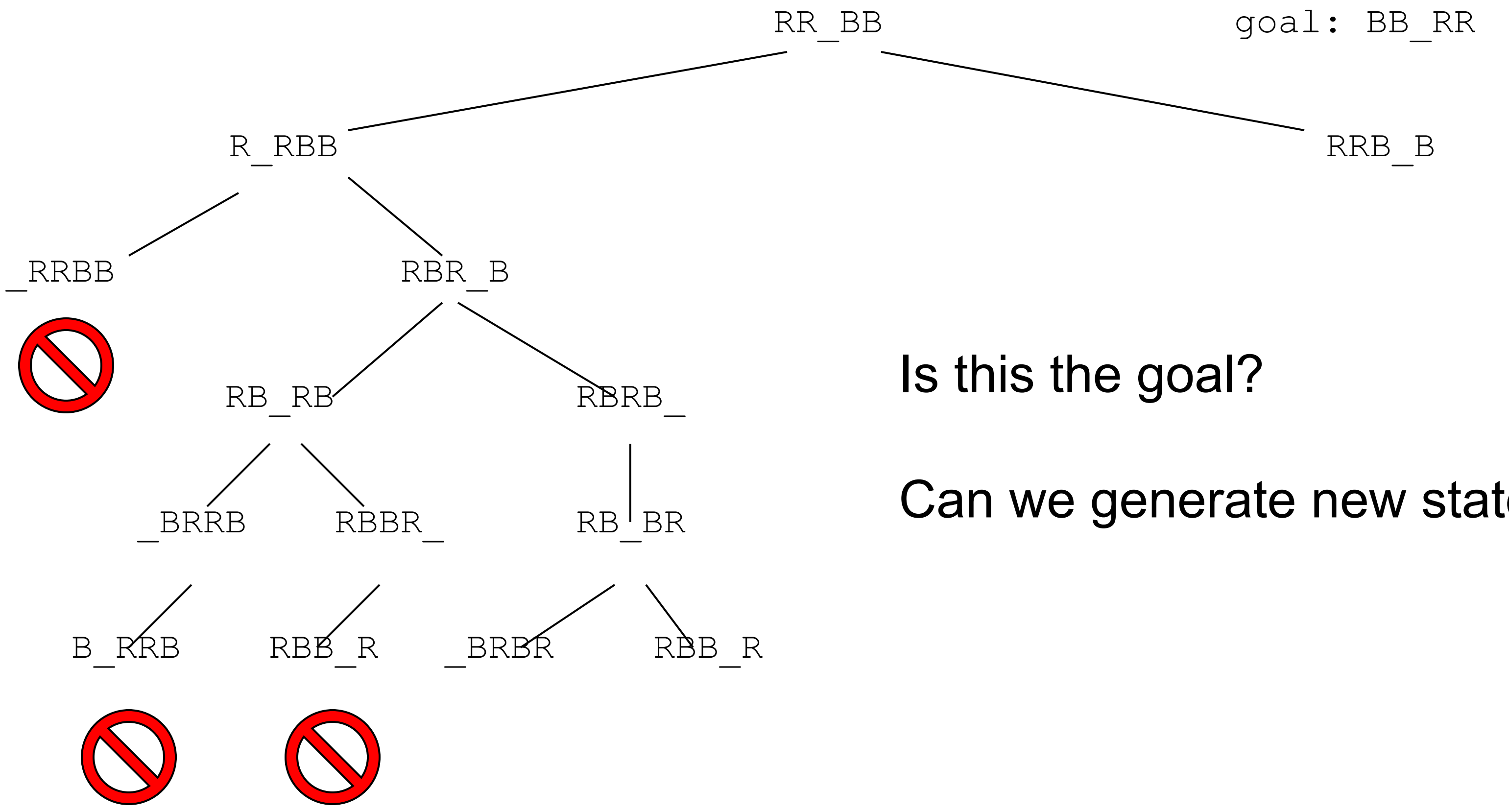
Four pegs in Haskell



Is this the goal?

Can we generate new states?

Four pegs in Haskell

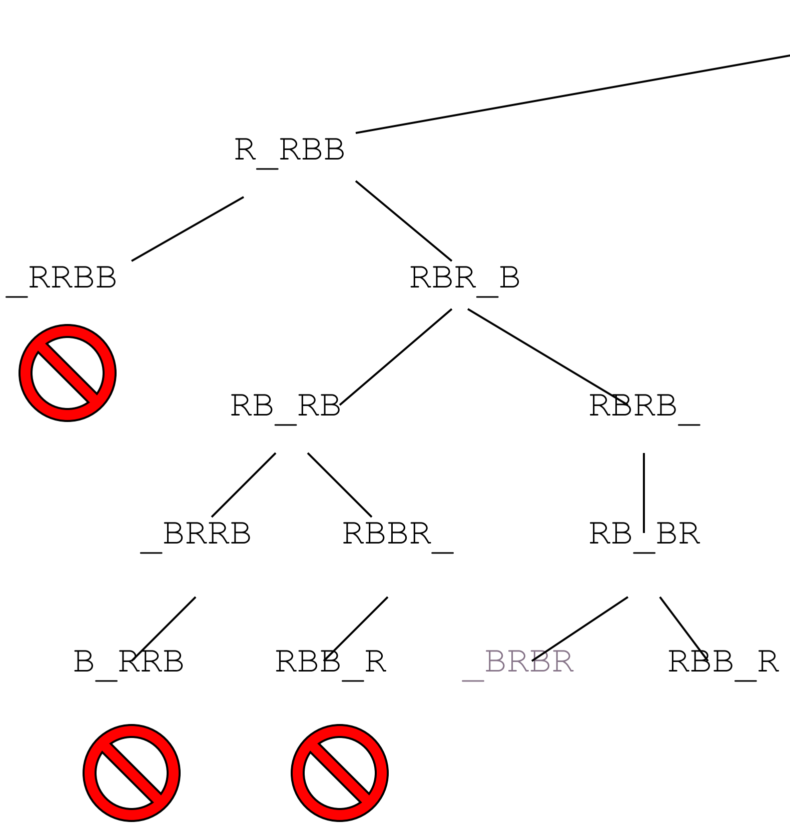


Is this the goal?

Can we generate new states?

Four pegs in Haskell

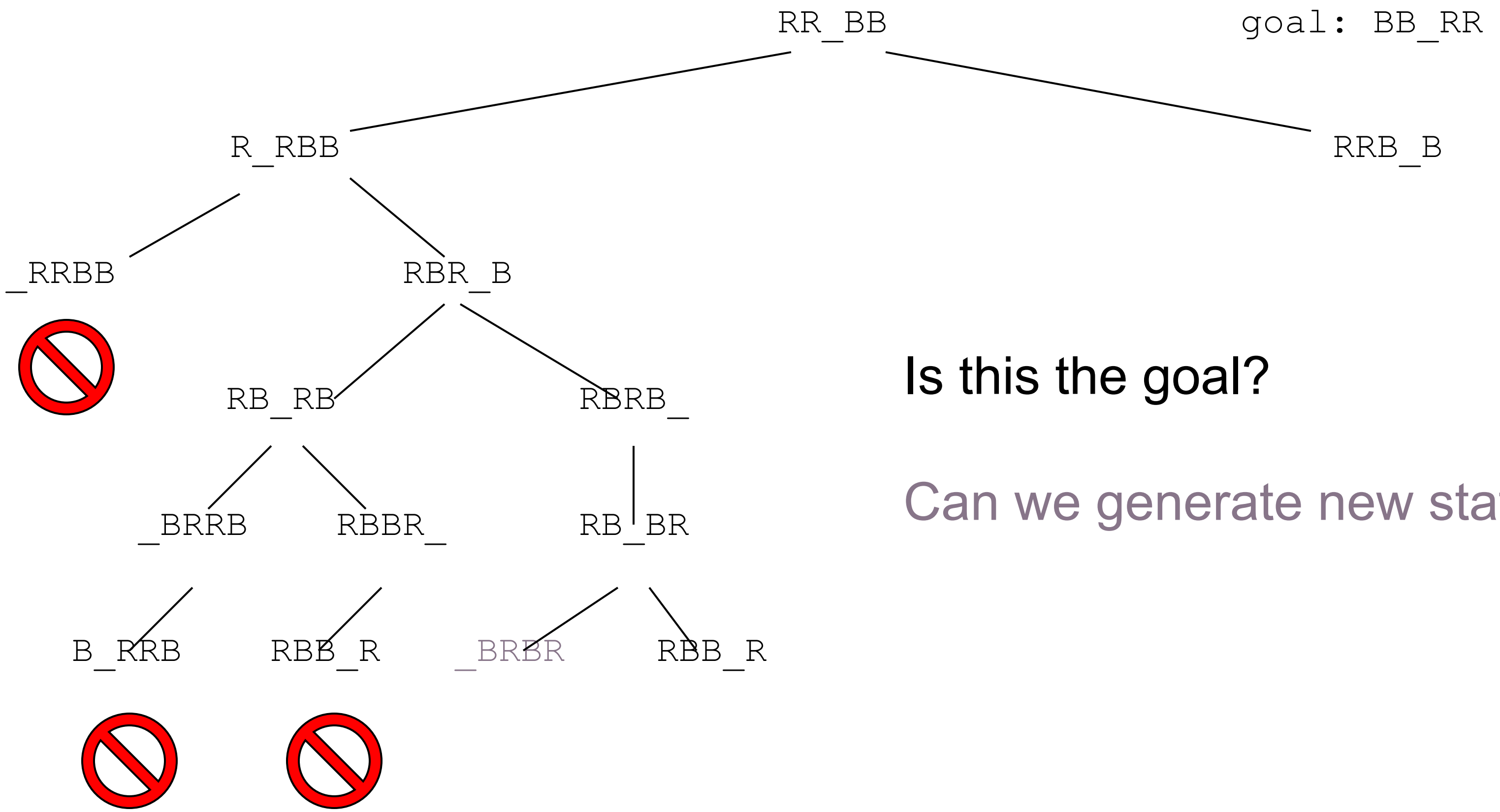
goal: BB_RR



Is this the goal?

Can we generate new states?

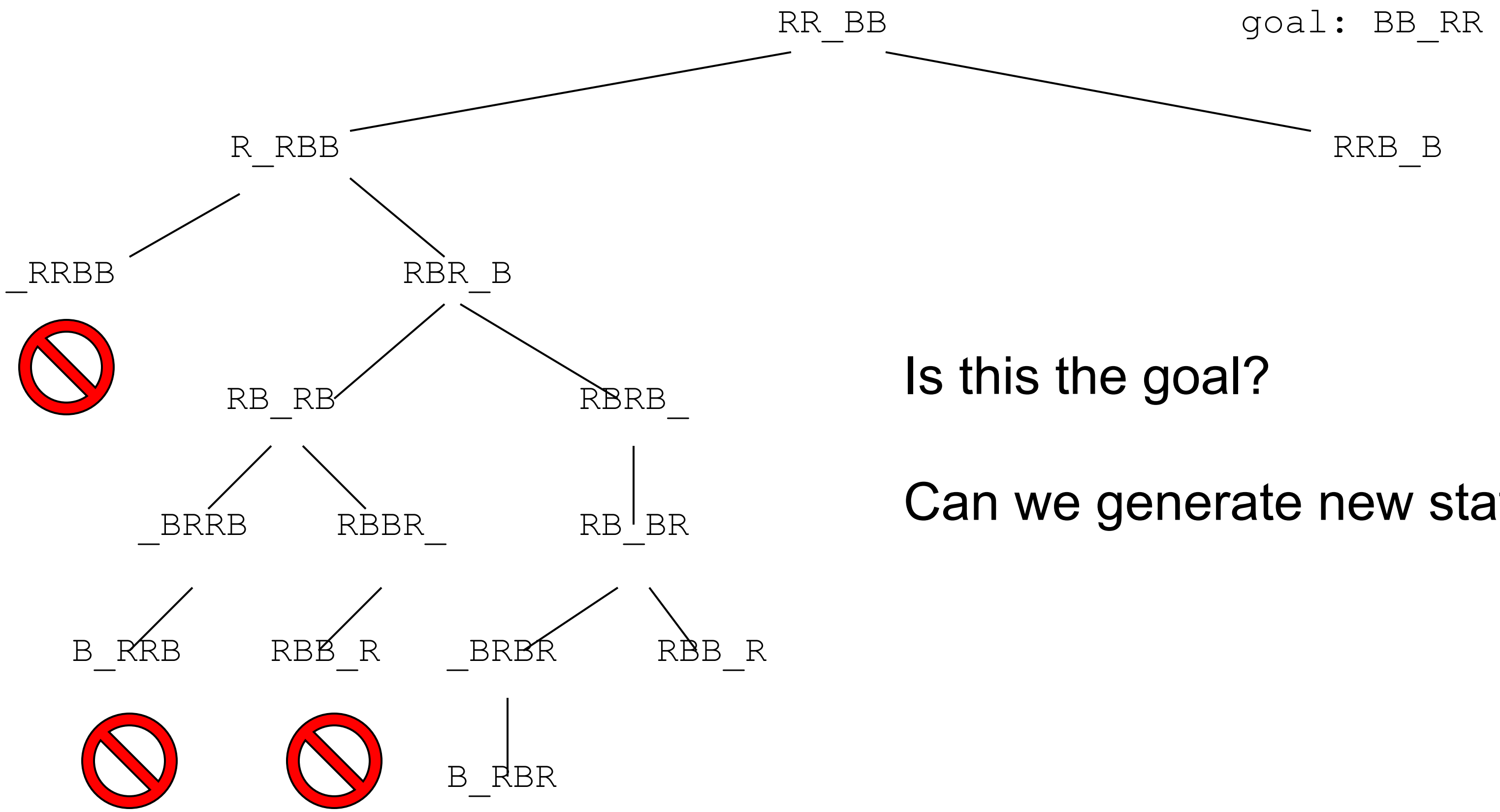
Four pegs in Haskell



Is this the goal?

Can we generate new states?

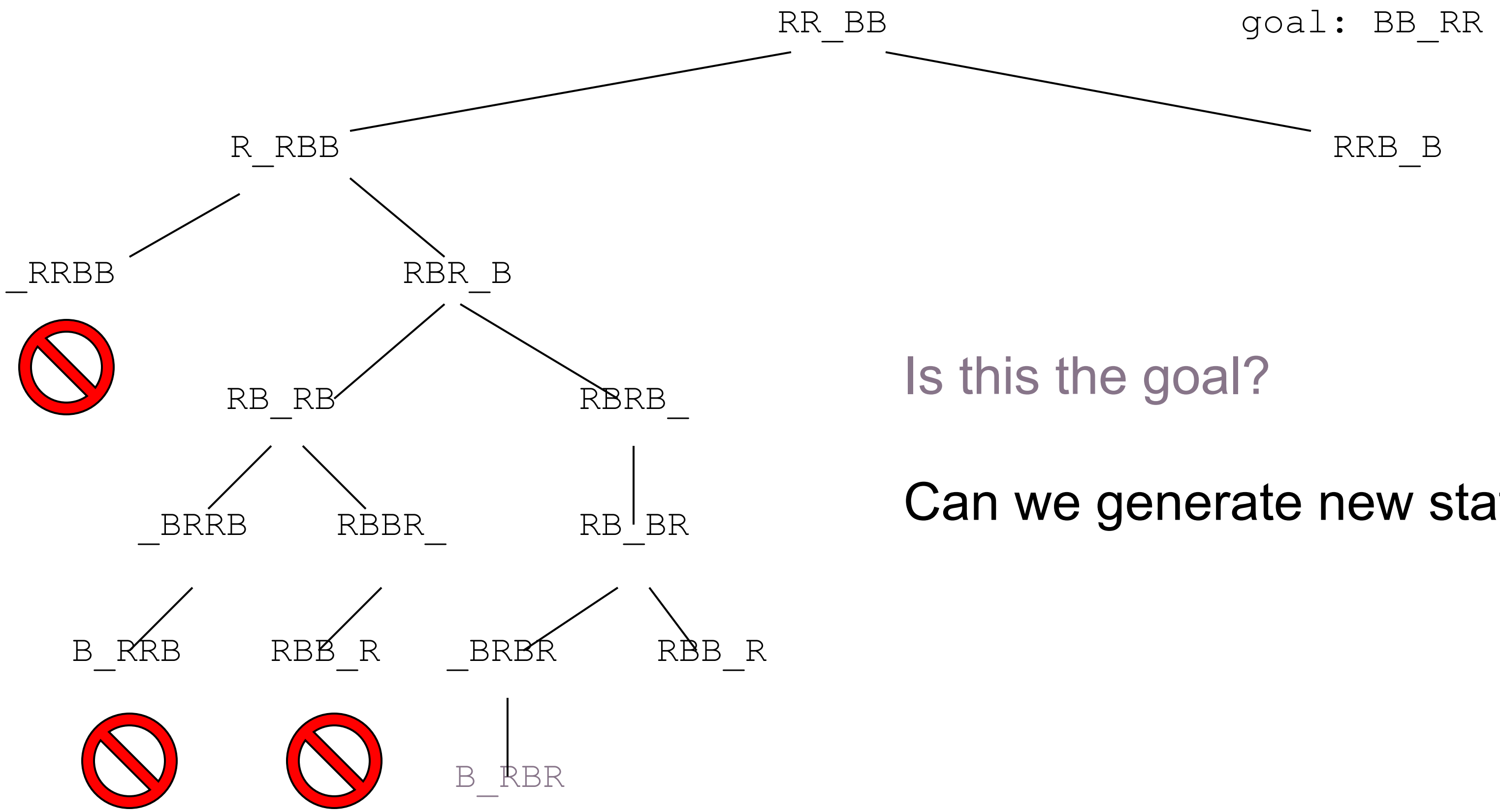
Four pegs in Haskell



Is this the goal?

Can we generate new states?

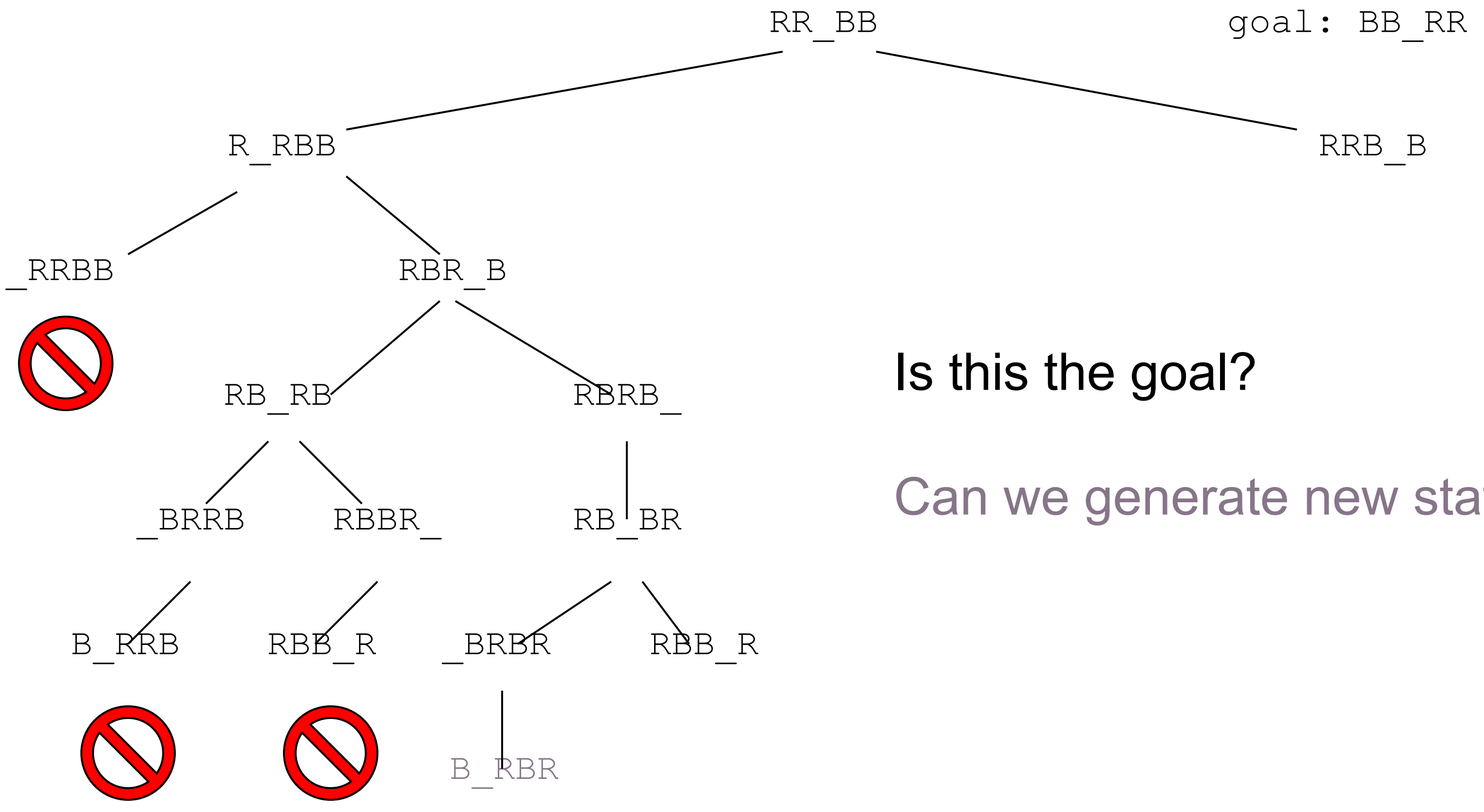
Four pegs in Haskell



Is this the goal?

Can we generate new states?

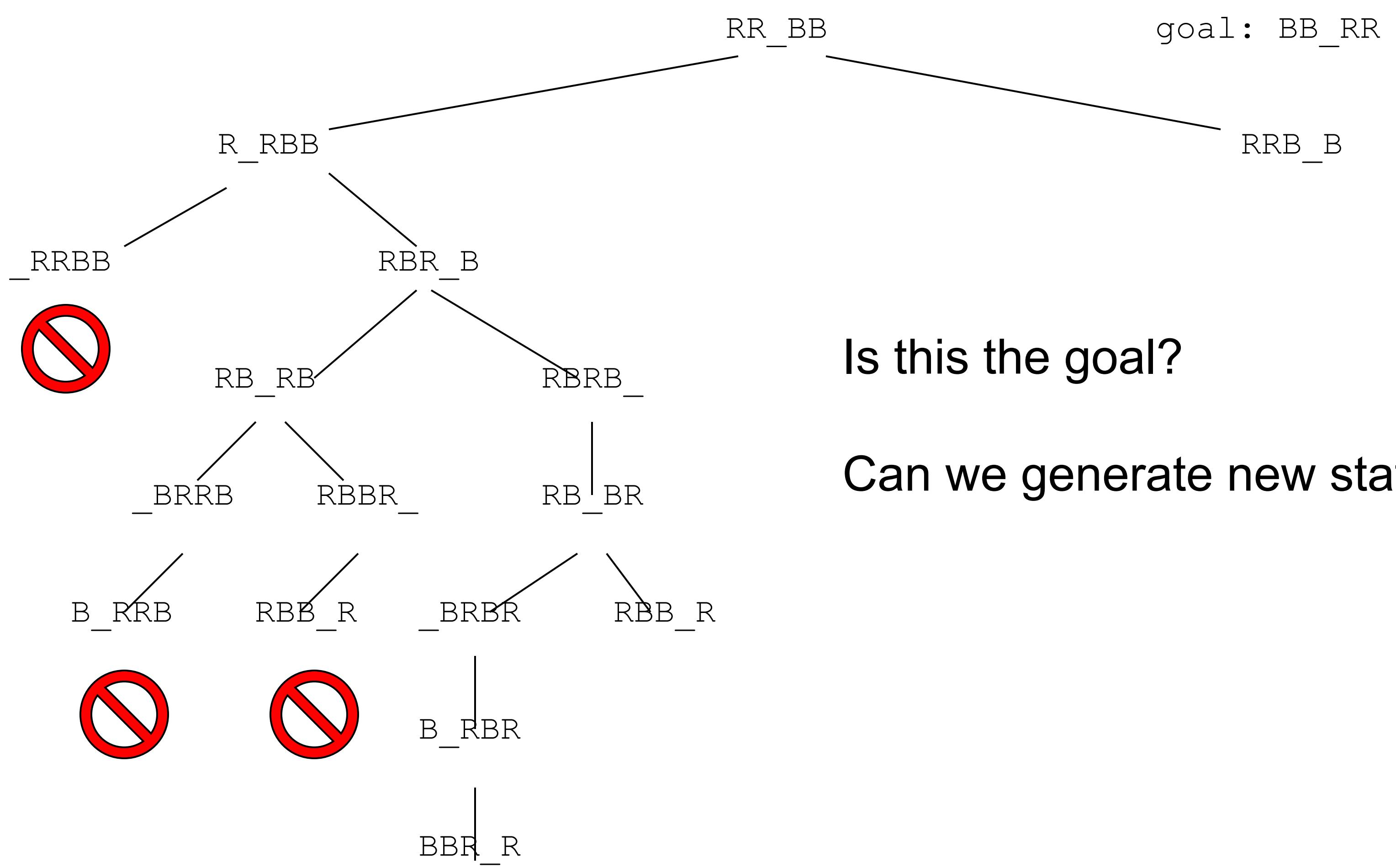
Four pegs in Haskell



Is this the goal?

Can we generate new states?

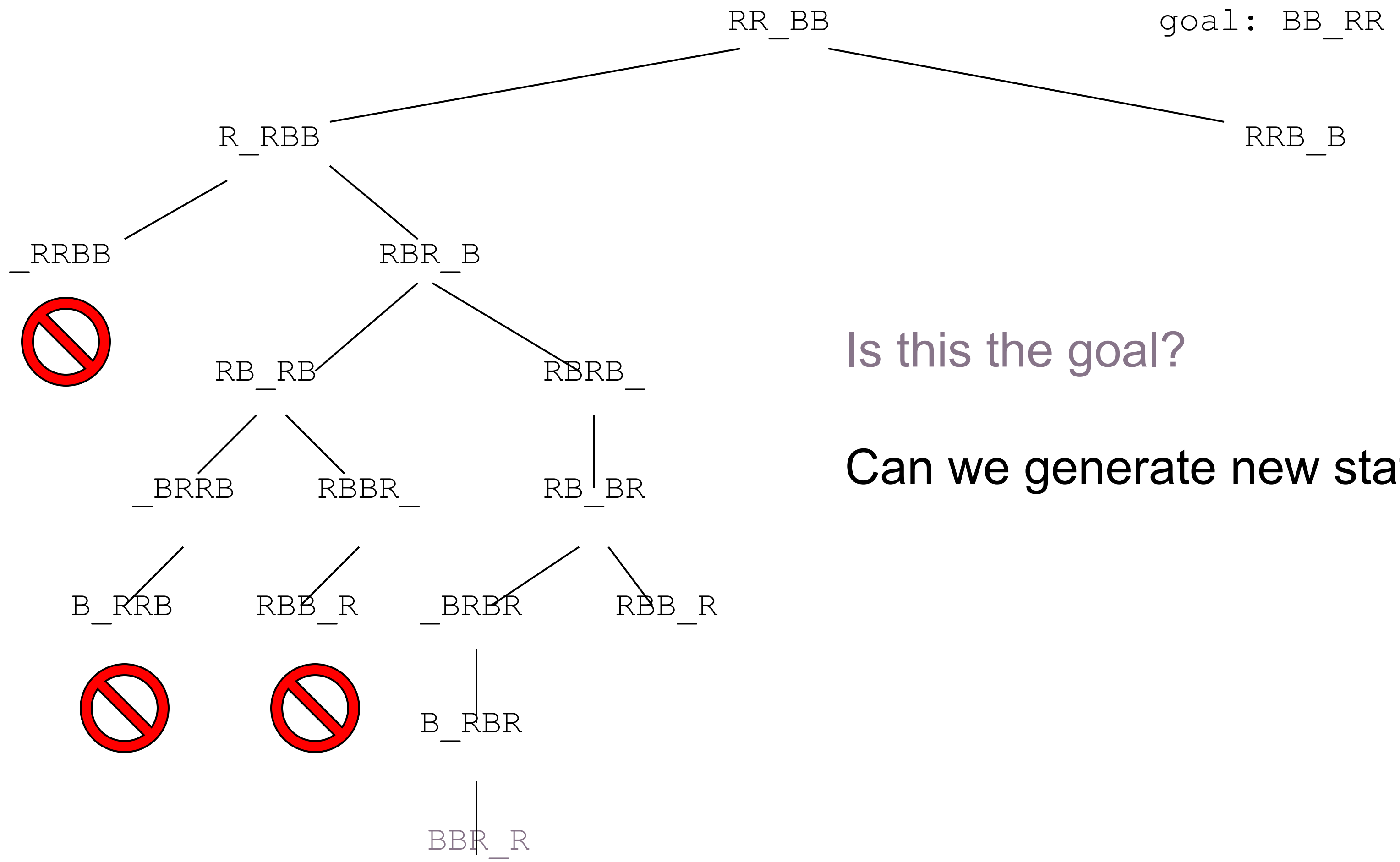
Four pegs in Haskell



Is this the goal?

Can we generate new states?

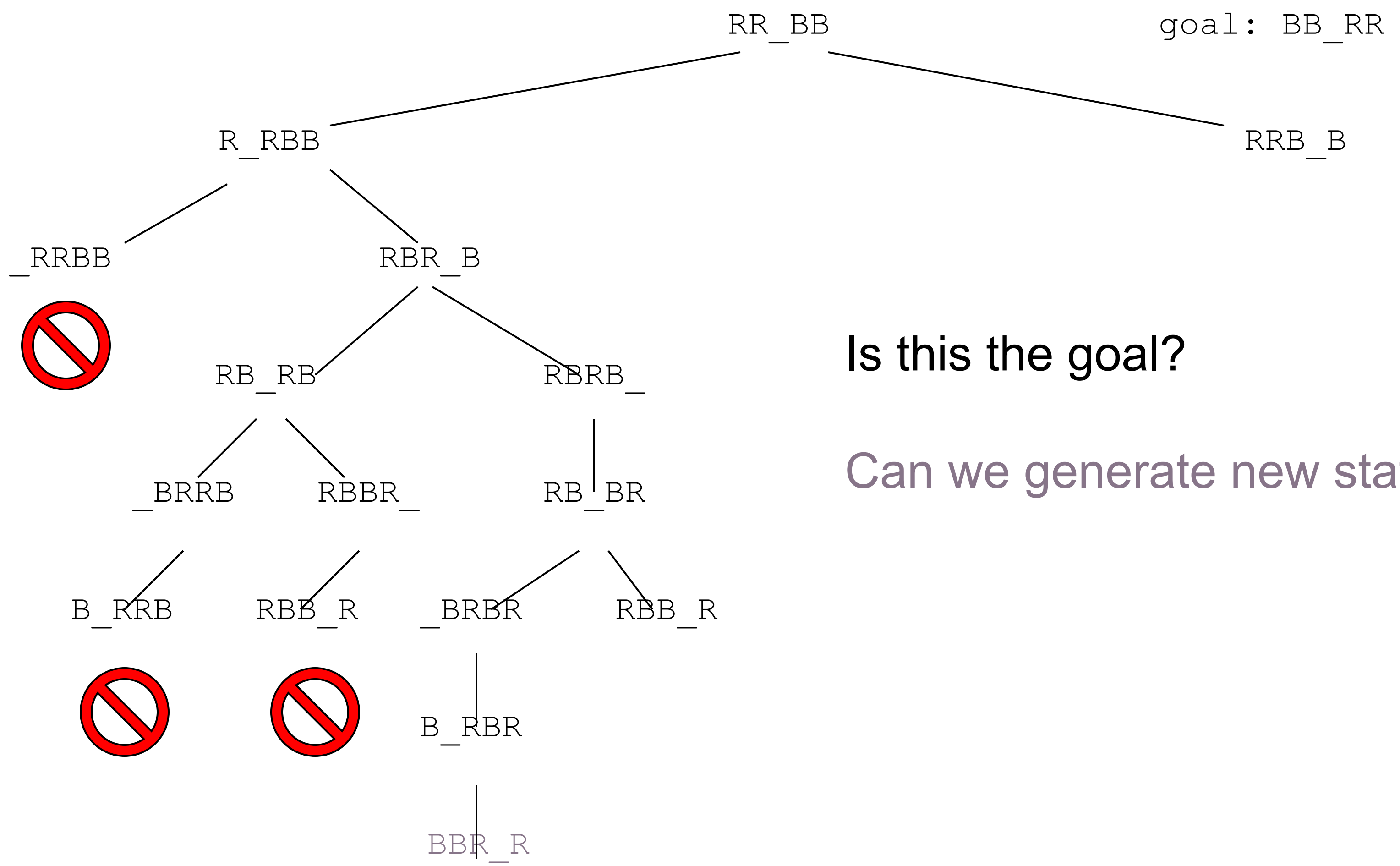
Four pegs in Haskell



Is this the goal?

Can we generate new states?

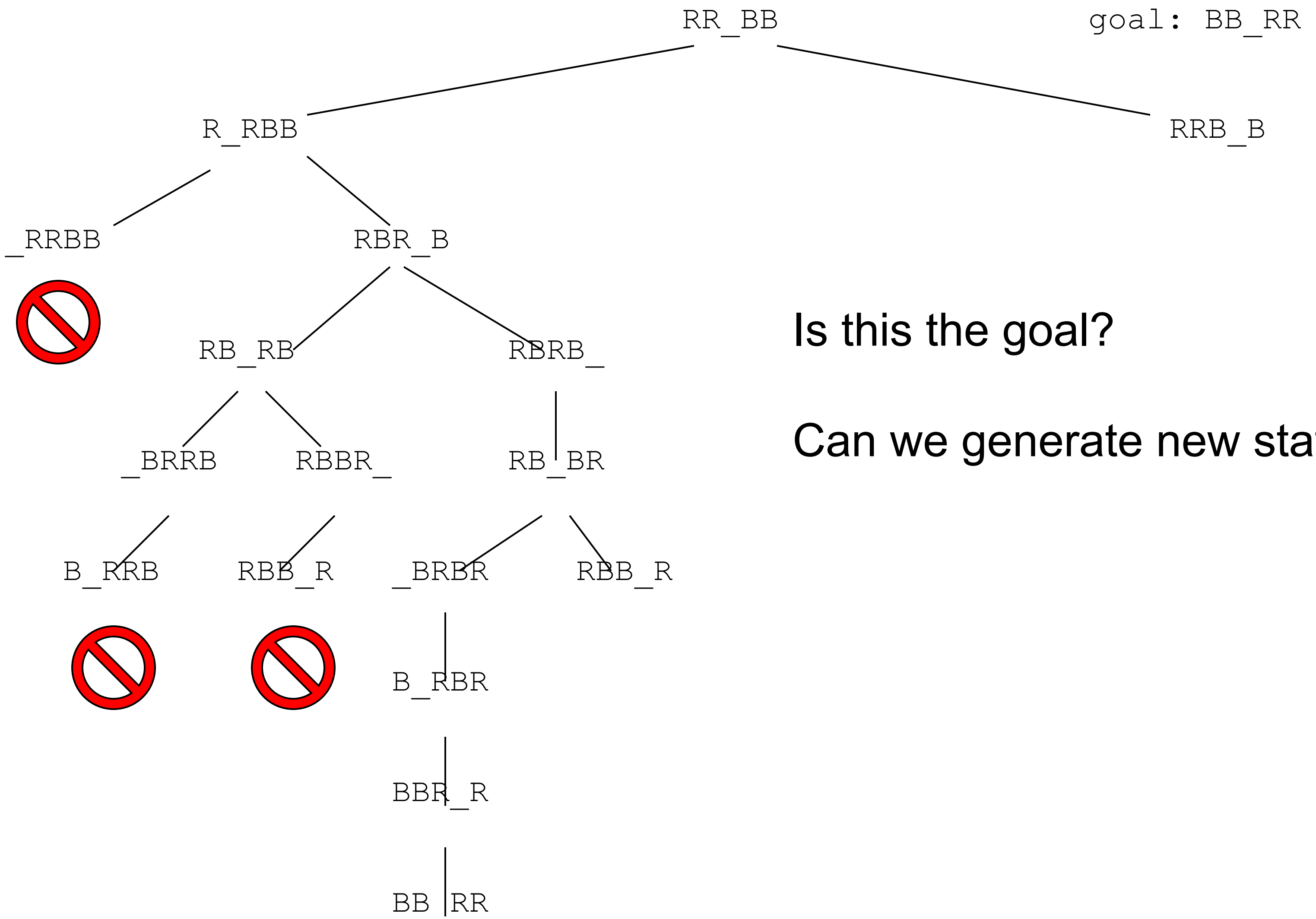
Four pegs in Haskell



Is this the goal?

Can we generate new states?

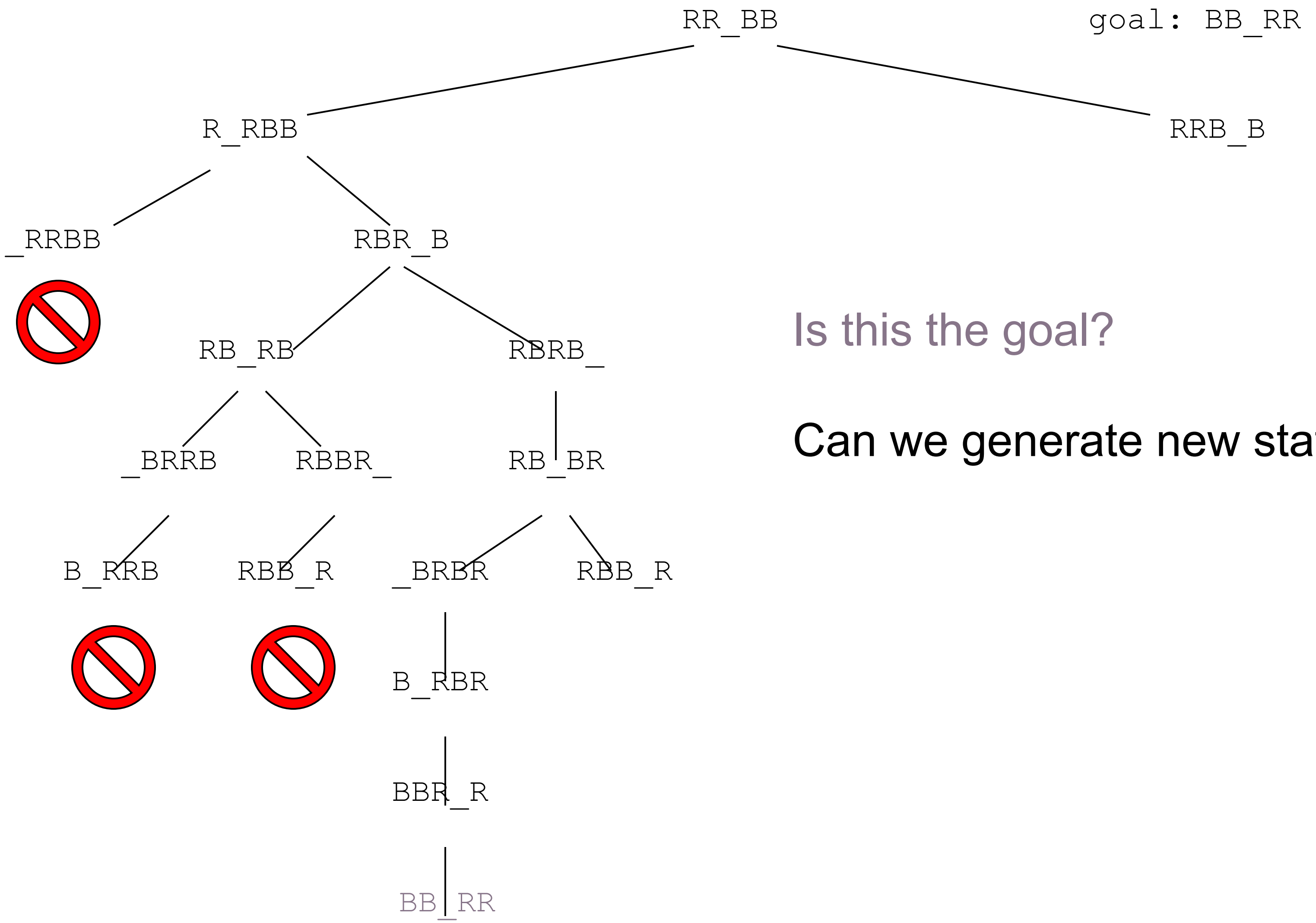
Four pegs in Haskell



Is this the goal?

Can we generate new states?

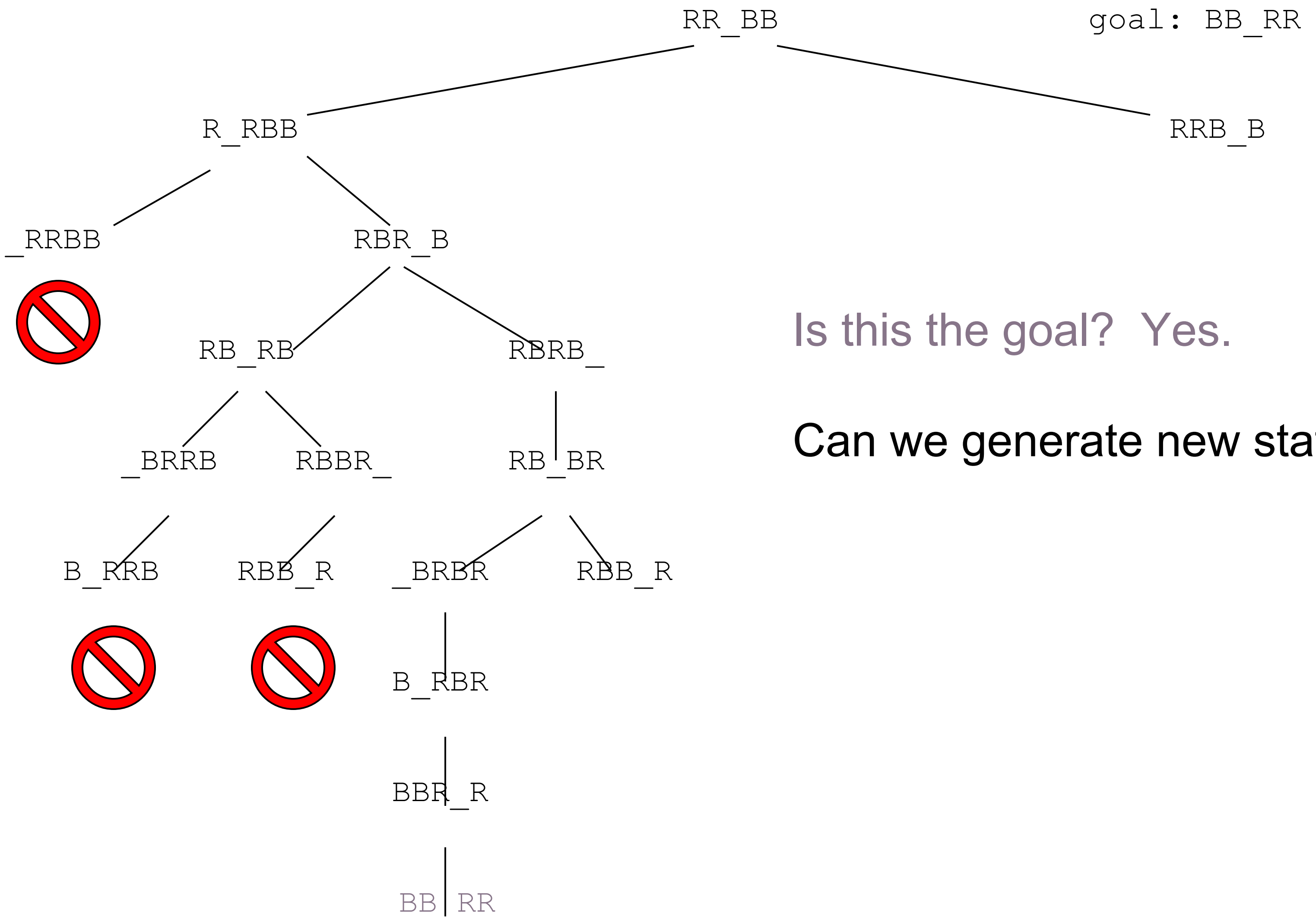
Four pegs in Haskell



Is this the goal?

Can we generate new states?

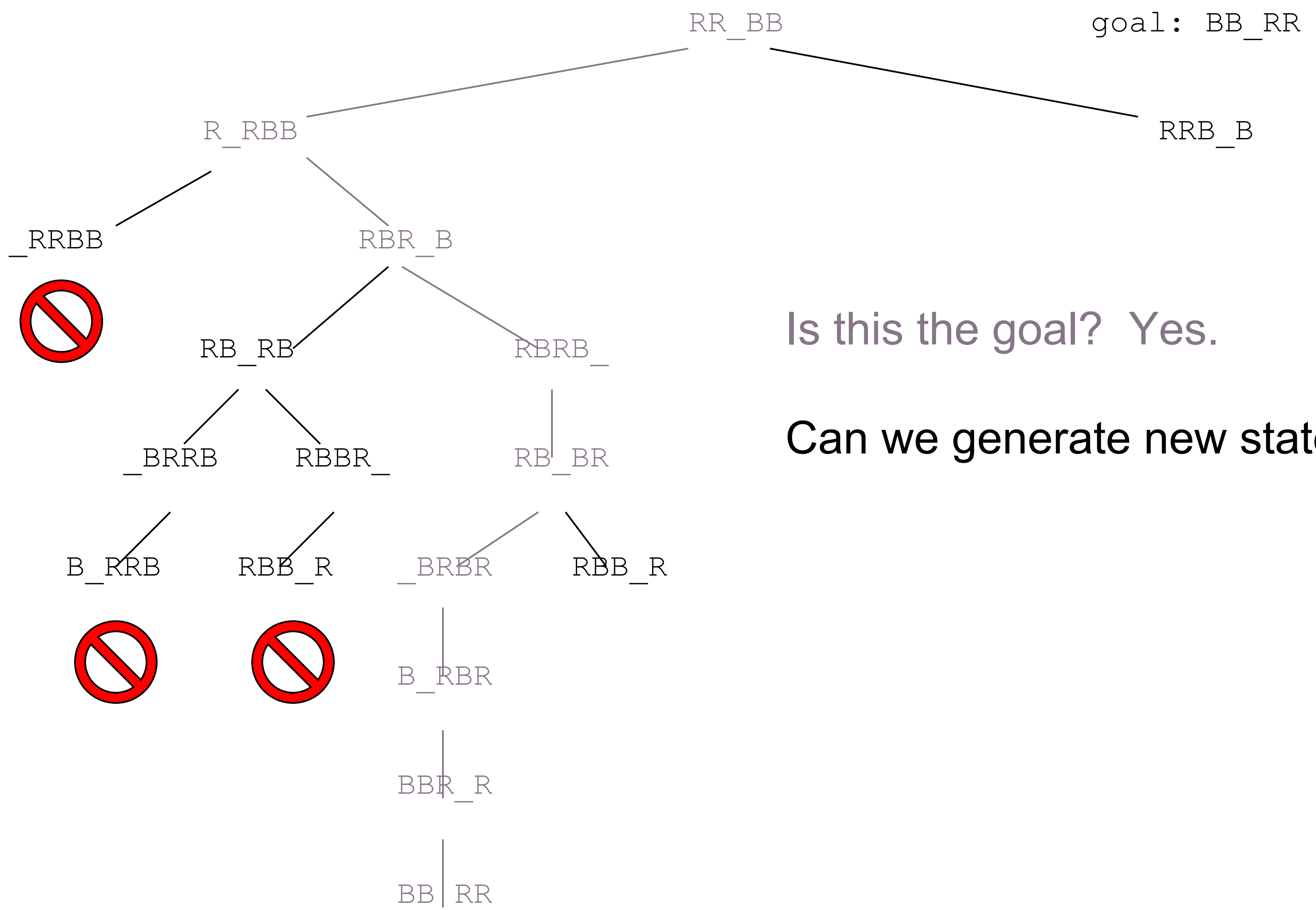
Four pegs in Haskell



Is this the goal? Yes.

Can we generate new states?

Four pegs in Haskell



Is this the goal? Yes.

Can we generate new states?

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
   | null unexplored                 = []
   | goal == head unexplored         = goal:path
   | (not (null newstates))          = newstates
   | otherwise                       =
        statesearch (tail unexplored) goal path
     where newstates = statesearch
                       (generateNewStates (head unexplored))
                       goal
                       ((head unexplored):path)
```

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
    | null unexplored                 = []
    | goal == head unexplored         = goal:path
    | (not (null newstates))          = newstates
    | otherwise                       =
        statesearch (tail unexplored) goal path
      where newstates = statesearch
                        (generateNewStates (head unexplored))
                        goal
                        ((head unexplored):path)
```

Note 0: This top-level program could easily be used as the top-level program for solving all sorts of puzzles. All that's needed is supporting functions to implement the appropriate knowledge representation for the puzzle (e.g., tiles and location) and the operators (e.g., slide tile left, right, up, down).

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
   | null unexplored                  = []
   | goal == head unexplored          = goal:path
   | (not (null newstates))           = newstates
   | otherwise                        =
      statesearch (tail unexplored) goal path
     where newstates = statesearch
                       (generateNewStates (head unexplored))
                       goal
                       ((head unexplored):path)
```

Note 1: We've added a parameter for passing the list of Strings that represents the chain of states from the start state to the state currently being explored. When a goal state is found, the list will contain the goal state and all the intermediate states back to the initial state.

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
    | null unexplored                = []
    | goal == head unexplored        = goal:path
    | elem (head unexplored) path   = statesearch (tail unexplored)
    | (not (null newstates))         = newstates
    | otherwise                      =
        statesearch (tail unexplored) goal path
      where newstates = statesearch
                        (generateNewStates (head unexplored))
                        goal
                        ((head unexplored):path)
```

Note 2: With the simple peg puzzle, there is no possibility of cycles -- paths that loop back on themselves -- because the pegs can't move backwards. But a more general search algorithm would check for cycles like this.

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
    | null unexplored                = []
    | goal == head unexplored        = goal:path
    | (not (null newstates))         = newstates
    | otherwise                      =
        statesearch (tail unexplored) goal path
      where newstates = statesearch
                         (generateNewStates (head unexplored))
                         goal
                         (head unexplored):path
```

Note 4: Haskell error messages are tragically unhelpful. Can you find the bug in this code? Here's most of the
error message...

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
   | null unexplored                 = []
   | goal == head unexplored         = goal:path
   | (not (null newstates))          = newstates
   | otherwise                       =
        statesearch (tail unexplored) goal path
     where newstates = statesearch
                       (generateNewStates (head unexplored))
                       goal
                       (head unexplored):path
```

```
    Couldn't match expected type `String' with actual type `Char'
     Expected type: [[String]]
       Actual type: [String]
     In the first argument of `head', namely `unexplored'
     In the third argument of `statesearch', namely `(head unexplored)'
Failed, modules loaded: none.
```

State-space search in Haskell

```
-- PegPuzzle.hs

pegpuzzle start goal = reverse (statesearch [start] goal [])

statesearch :: [String] -> String -> [String] -> [String]
statesearch unexplored goal path
   | null unexplored                = []
   | goal == head unexplored        = goal:path
   | (not (null newstates))         = newstates
   | otherwise                      =
      statesearch (tail unexplored) goal path
     where newstates = statesearch
                        (generateNewStates (head unexplored))
                        goal
                        ((head unexplored):path)
```

missing parentheses

State-space search in Haskell

The peg puzzle program works for any number of red pegs, any number of blue pegs, and apparently any number of empty spaces, as long as you start and end with the same number of each and there actually is a way to get from the start state to the goal state.

I have posted a commented version of this program on Piazza (General Resources).

State-space search in Haskell

```
generateNewStates :: String -> [String]
generateNewStates currState =
    concat  [generateNewRedSlides currState,
             generateNewRedJumps currState,
             generateNewBlueSlides currState,
             generateNewBlueJumps currState]
```

State-space search in Haskell

```
generateNewStates :: String -> [String]
generateNewStates currState =
    concat  [generateNewRedSlides currState,
             generateNewRedJumps currState,
             generateNewBlueSlides currState,
             generateNewBlueJumps currState]

generateNewRedSlides currState =
   generateNew currState 0 "R_" "_R"
```

State-space search in Haskell

```
generateNewStates :: String -> [String]
generateNewStates currState =
    concat  [generateNewRedSlides currState,
             generateNewRedJumps currState,
             generateNewBlueSlides currState,
             generateNewBlueJumps currState]

generateNewRedSlides currState =
   generateNew currState 0 "R_" "_R"

generateNewRedJumps currState =
   generateNew currState 0 "RB_" "_BR"
```

State-space search in Haskell

```
generateNewStates :: String -> [String]
generateNewStates currState =
    concat  [generateNewRedSlides currState,
             generateNewRedJumps currState,
             generateNewBlueSlides currState,
             generateNewBlueJumps currState]

generateNewRedSlides currState =
   generateNew currState 0 "R_" "_R"

generateNewRedJumps currState =
   generateNew currState 0 "RB_" "_BR"

generateNewBlueSlides currState =
   reverseEach (generateNew (reverse currState) 0 "B_" "_B")
```

State-space search in Haskell

```
generateNewStates :: String -> [String]
generateNewStates currState =
    concat  [generateNewRedSlides currState,
             generateNewRedJumps currState,
             generateNewBlueSlides currState,
             generateNewBlueJumps currState]

generateNewRedSlides currState =
   generateNew currState 0 "R_" "_R"

generateNewRedJumps currState =
   generateNew currState 0 "RB_" "_BR"

generateNewBlueSlides currState =
   reverseEach (generateNew (reverse currState) 0 "B_" "_B")

generateNewBlueJumps currState =
   reverseEach (generateNew (reverse currState) 0 "BR_" "_RB")
```

State-space search in Haskell

```
reverseEach listOfLists
    | null listOfLists       = []
    | otherwise              = (reverse (head listOfLists)):
                               (reverseEach (tail listOfLists))
```

State-space search in Haskell

```
generateNew currState pos oldSegment newSegment
   | pos + (length oldSegment) > length currState    = []
   | segmentEqual currState pos oldSegment           =
        (replaceSegment currState pos newSegment):
        (generateNew currState (pos + 1) oldSegment newSegment)
   | otherwise                                       =
        (generateNew currState (pos + 1) oldSegment newSegment)

segmentEqual currState pos oldSegment   =
   (oldSegment == take (length oldSegment) (drop pos currState))

replaceSegment oldList pos segment
   | pos == 0   = segment ++ drop (length segment) oldList
   | otherwise  =
        (head oldList):
        (replaceSegment (tail oldList)(pos - 1) segment)
```

State-space search

state-space-search (list-of-unexplored-states, goal-state, operators)

1. look at the first (leftmost) unexplored-state
2. if that state is the goal-state, then return success
3. if that state isn't the goal-state, then generate all possible new states from that state by applying the set of operators to that state
4. call state-space-search with this new list of states passed as the unexplored-states argument, and if that succeeds then return success else...
5. call state-space-search with the old list of unexplored-states that remained after you stripped off the first unexplored-state in step 1, and if that succeeds then return success else...
6. return failure

State-space search

state-space-search (list-of-unexplored-states, goal-state, operators)

1. look at the first (leftmost) unexplored-state
2. if that state is the goal-state, then return success
3. if that state isn't the goal-state, then generate all possible new states from that state by applying the set of operators to that state

(By the way, in step 3, you'd like to check all the new states to see if you've explored them before. You do that by keeping track of the sequence of states that was generated in going from the very first state to where you are now, and then comparing that list to the set of new states you just generated. If there are any duplicates, be sure to eliminate them from the set of new states. We don't need to check for these cycles in the simple peg puzzle, but we will keep track of the sequence of states that's generated in getting from the start state to the currently-explored state, because that's what we'll eventually return as a solution.)

Problem-solving as list manipulation

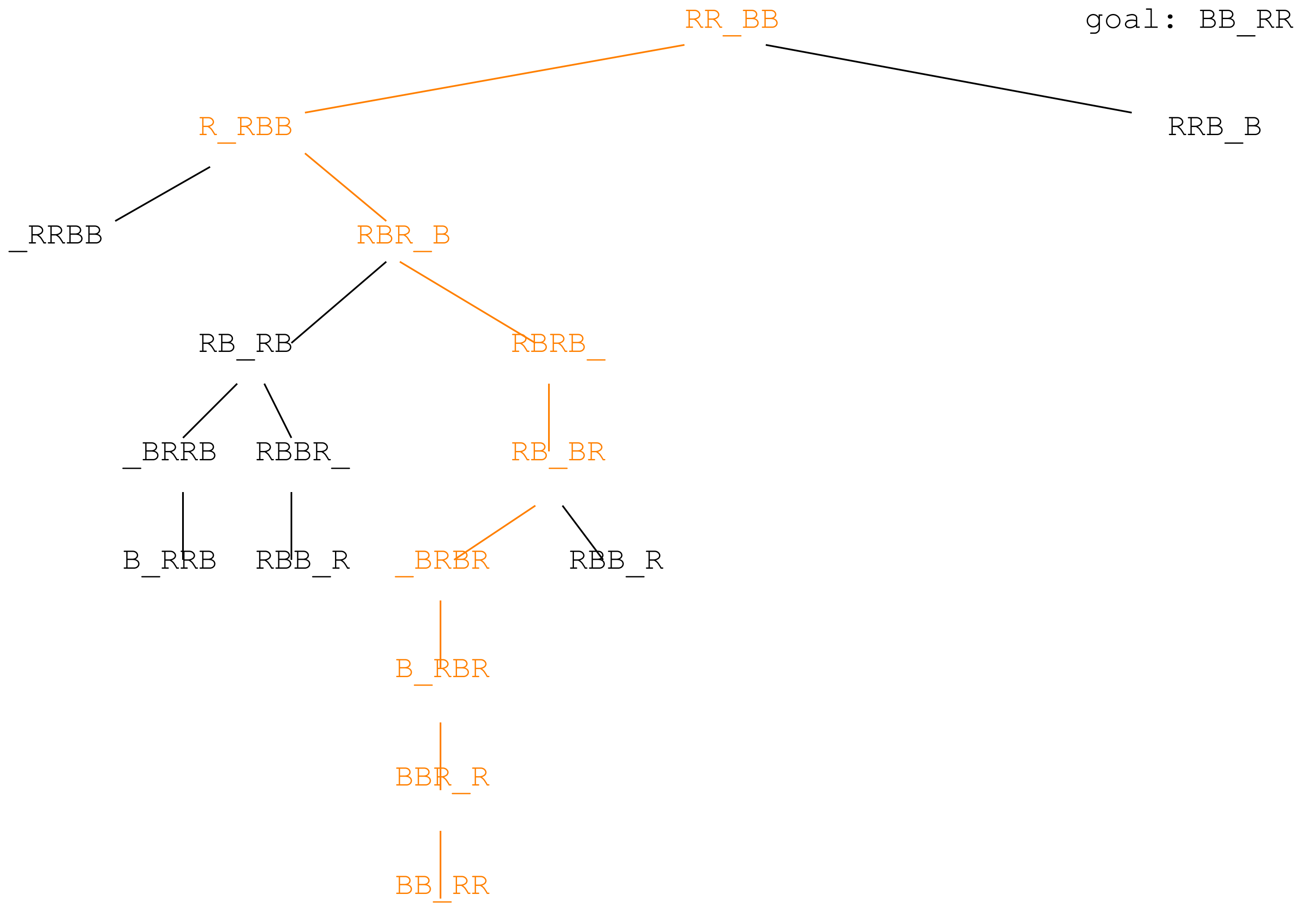
The list data structure is prominent in functional programming languages, at the very least because these languages rely on recursion and lists are recursively-defined data structures.

Problem-solving as list manipulation

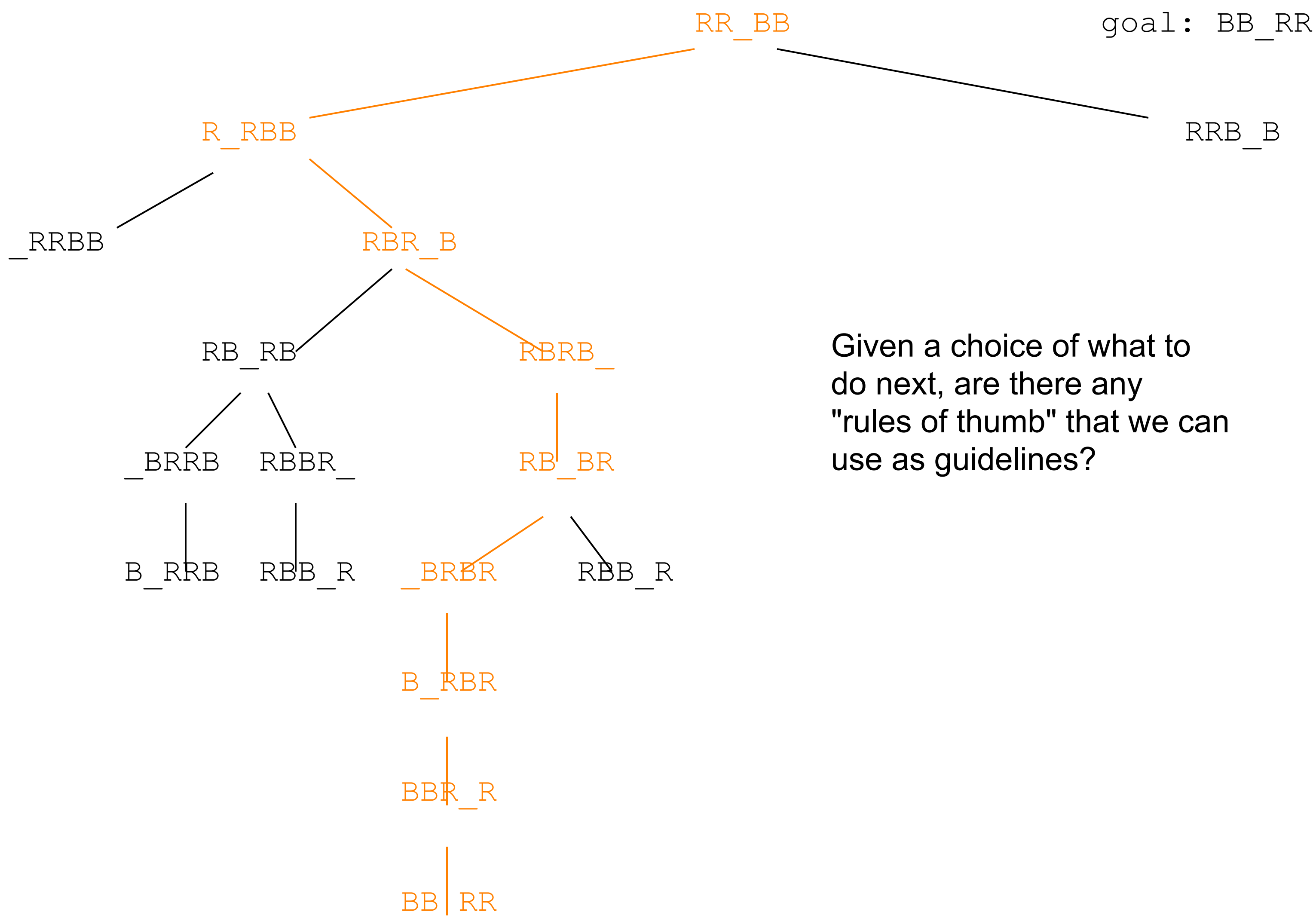
All the list manipulation in the peg puzzle program is done with cons (:), head, and tail. (Note that ++, reverse, take, and drop are all defined in terms of cons, head, and tail).

In a list-based programming world, all you need are cons, head, tail, relational operators and a conditional -- with those five things, you can do anything.

Four pegs again

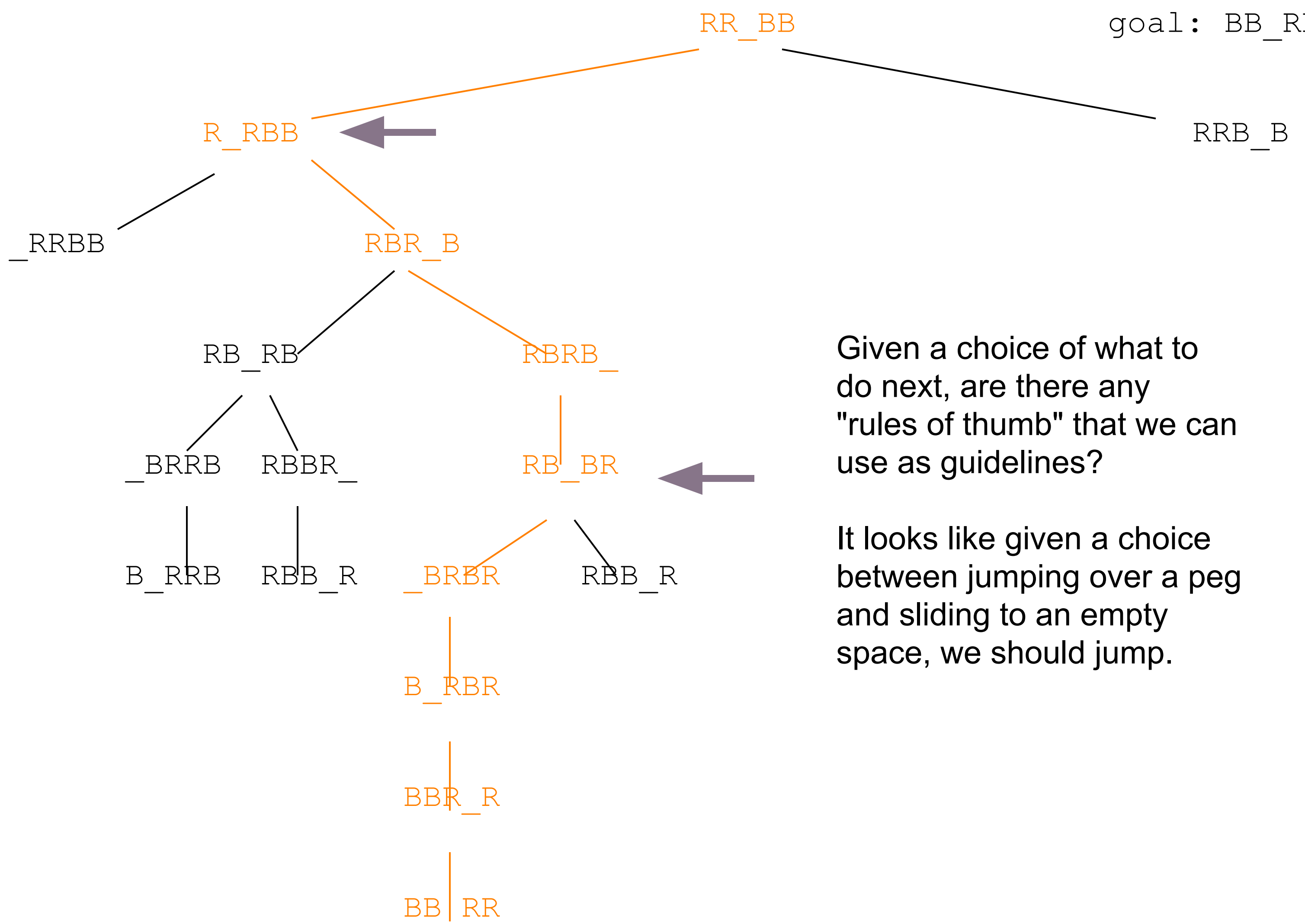


Four pegs again



Given a choice of what to do next, are there any "rules of thumb" that we can use as guidelines?

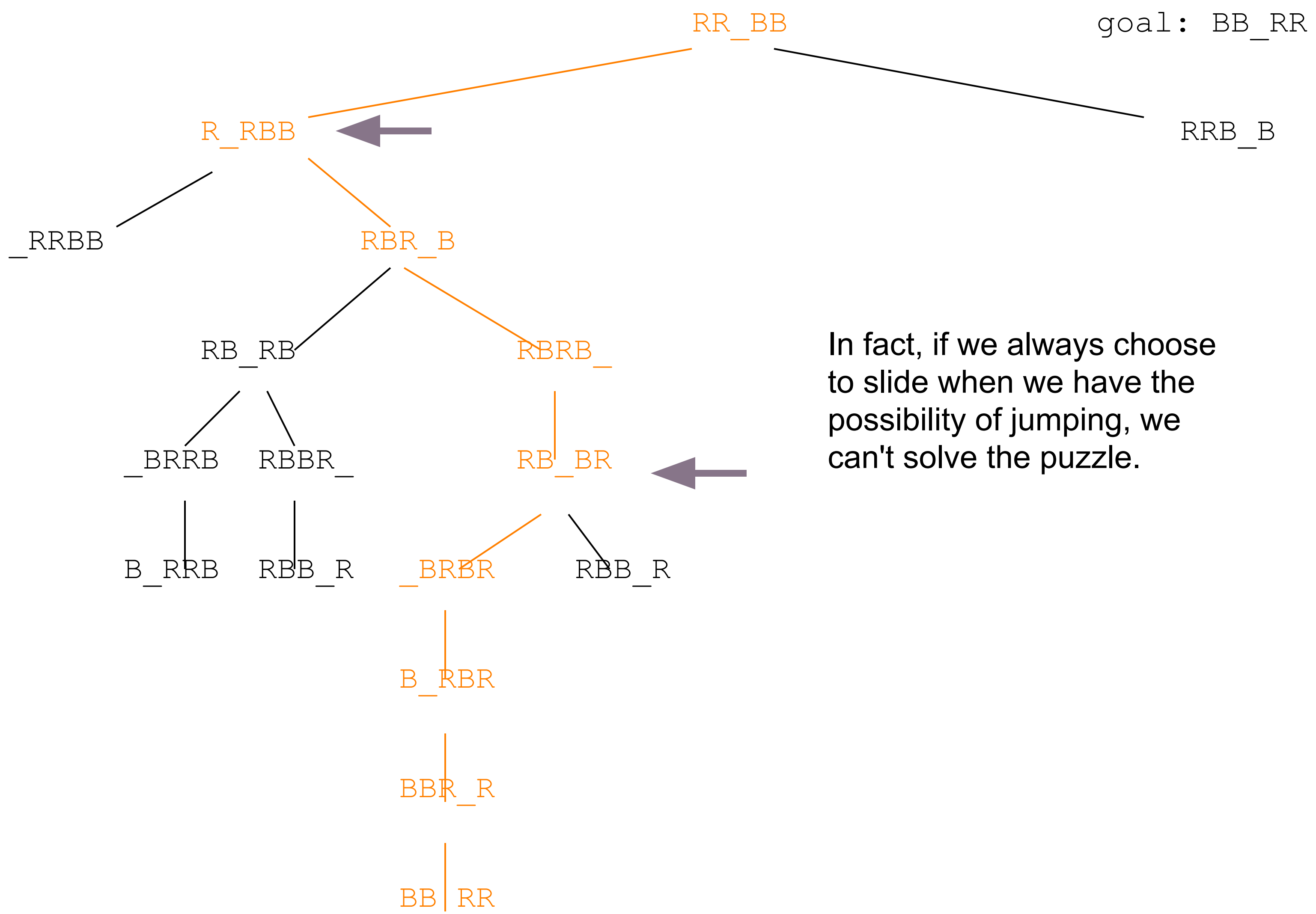
Four pegs again



Given a choice of what to do next, are there any "rules of thumb" that we can use as guidelines?

It looks like given a choice between jumping over a peg and sliding to an empty space, we should jump.

Four pegs again



In fact, if we always choose to slide when we have the possibility of jumping, we can't solve the puzzle.

This leads us to...

...a brief exploration of everything any computer scientist should know about artificial intelligence even if they never do any AI work or even take an AI class.

The two most important principles of AI

The two most important principles of AI

Intelligence is search (Newell and Simon)

The two most important principles of AI

Intelligence is search (Newell and Simon)

Search is to be avoided (Feigenbaum)

The two most important principles of AI

Intelligence is search (Newell and Simon)

Search is to be avoided (Feigenbaum)

This leads to the inevitable question...

...can we make search less expensive?

Usually we don’t have time to spare. We want to find the path to goal with as little effort as possible.

We can use problem-specific knowledge to tell us how to choose the next node or state for exploration. This knowledge provides an estimate of the nearness of a given node to the goal node - sometimes called the “goodness” of a node.

Can we make search less expensive?

This knowledge about the problem domain used to make “educated guesses” about which node to choose next for further exploration is called heuristic knowledge.

Can we make search less expensive?

This knowledge about the problem domain used to make “educated guesses” about which node to choose next for further exploration is called heuristic knowledge.

Good heuristic knowledge reduces the search significantly in many cases, but isn’t necessarily guaranteed to do the job in all cases.

Let's look at heuristic knowledge in the context of a slightly more complex problem...

The 15-tile puzzle

The 8-tile puzzle

Tile puzzle -- exhaustive depth-first search

```
2 8 3              1 2 3
 1 _ 4      goal:  8 _ 4
 7 6 5              7 6 5
```

Tile puzzle

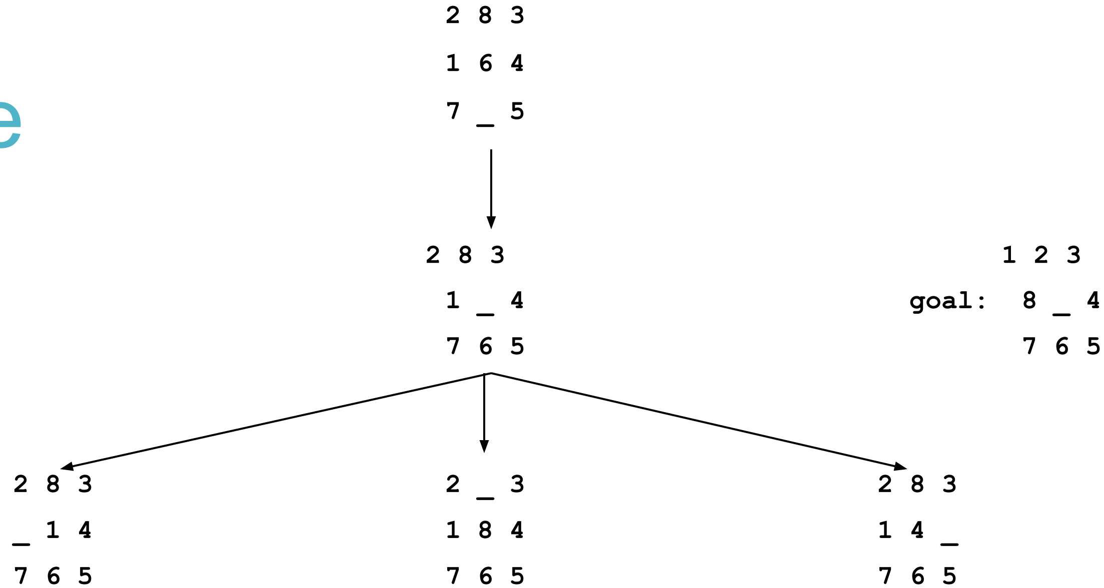
```
2 8 3
1 6 4
7 _ 5
```

$\downarrow$

```
2 8 3
1 _ 4
7 6 5
```

```
       1 2 3
goal:  8 _ 4
       7 6 5
```

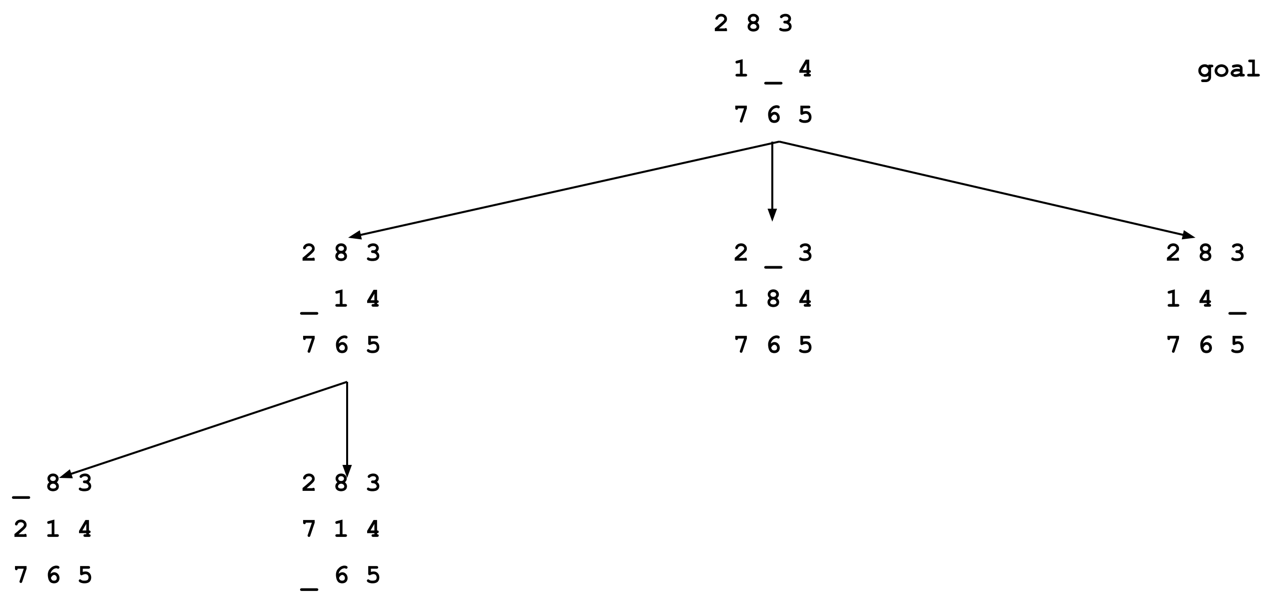
Tile puzzle



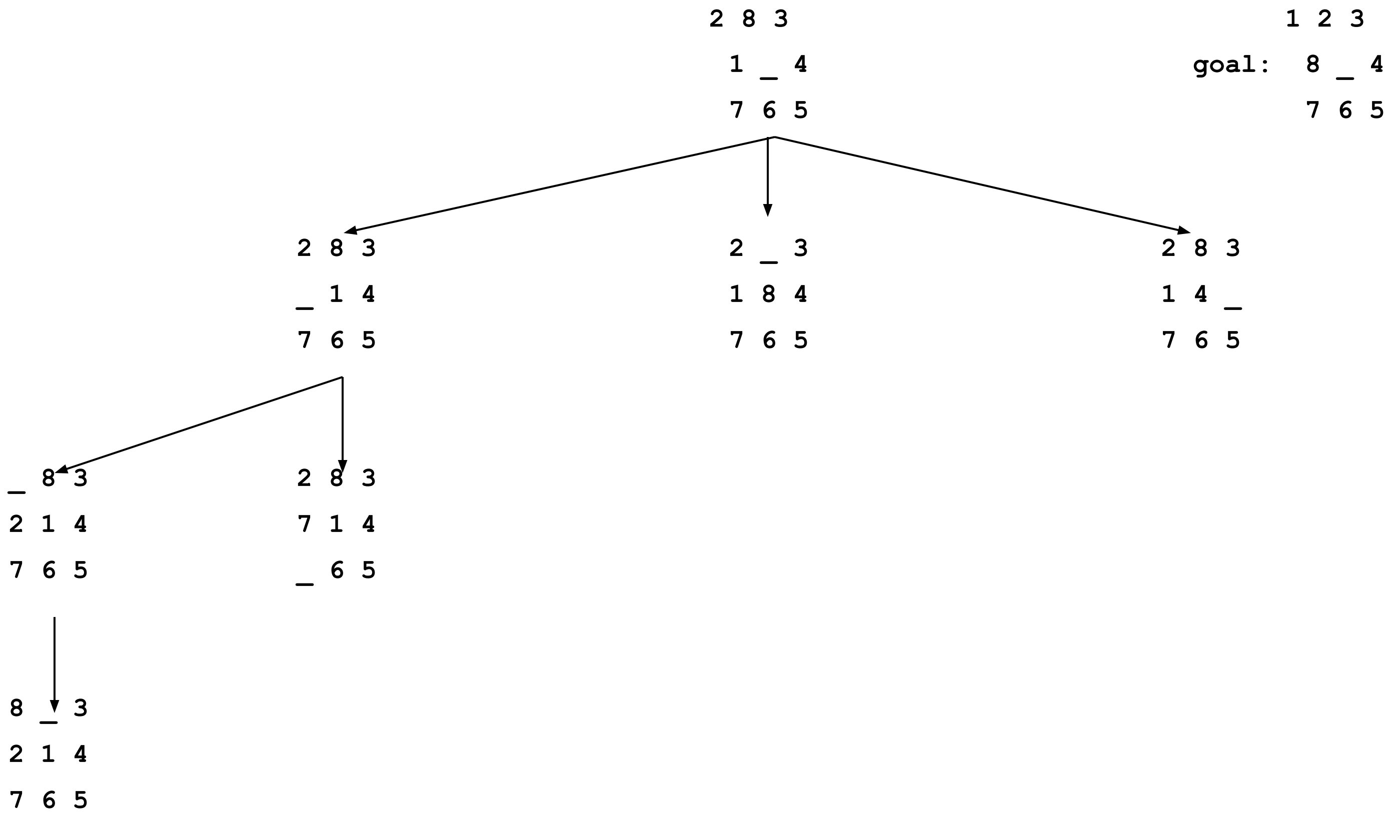
2 8 3
1 6 4
7 _ 5
2 8 3
1 _ 4
7 6 5
1 2 3
goal: 8 _ 4
7 6 5
2 8 3
_ 1 4
7 6 5
2 _ 3
1 8 4
7 6 5
2 8 3
1 4 _
7 6 5

Tile puzzle -- exhaustive depth-first search

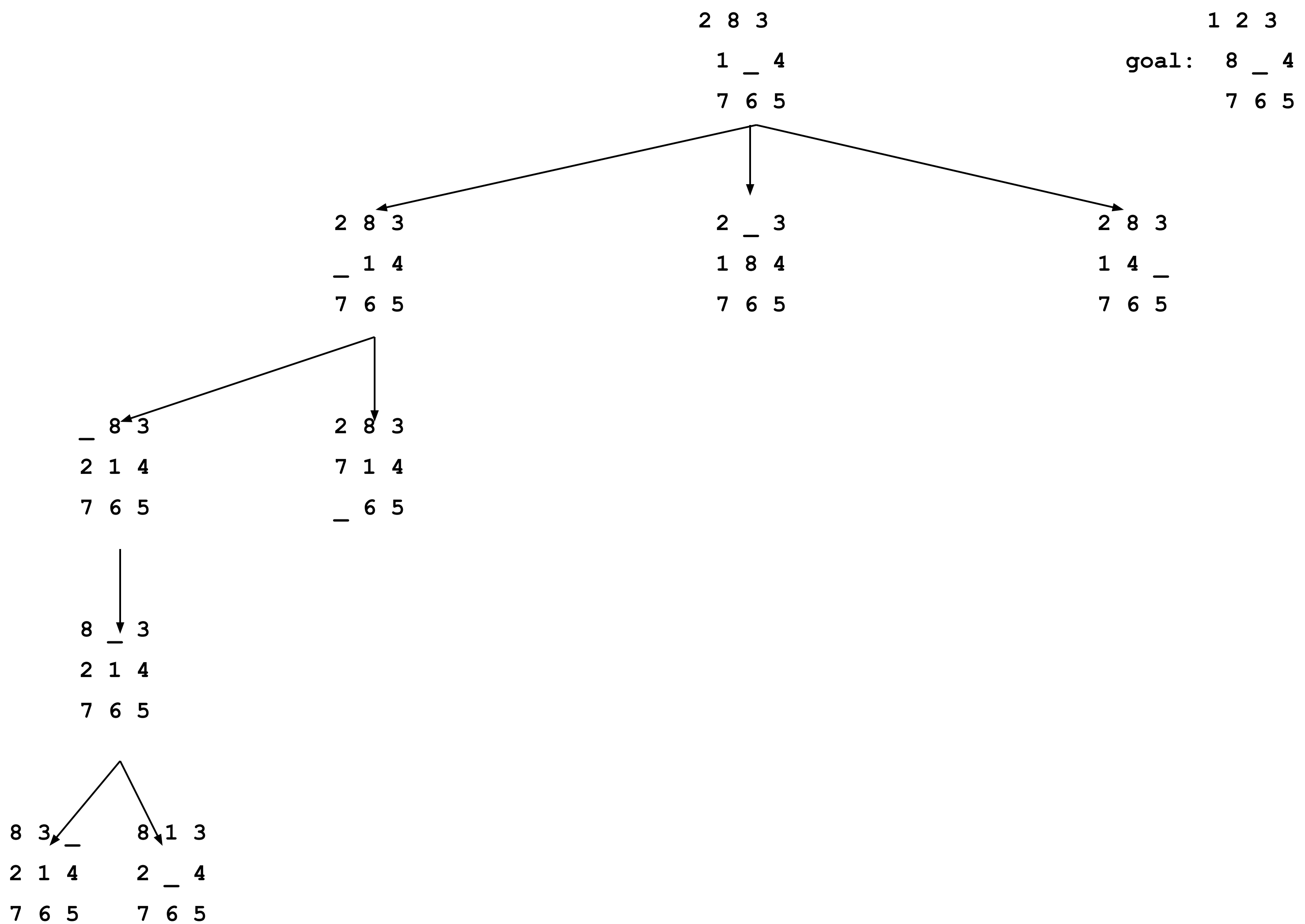
```
        1 2 3
goal:   8 _ 4
        7 6 5
```



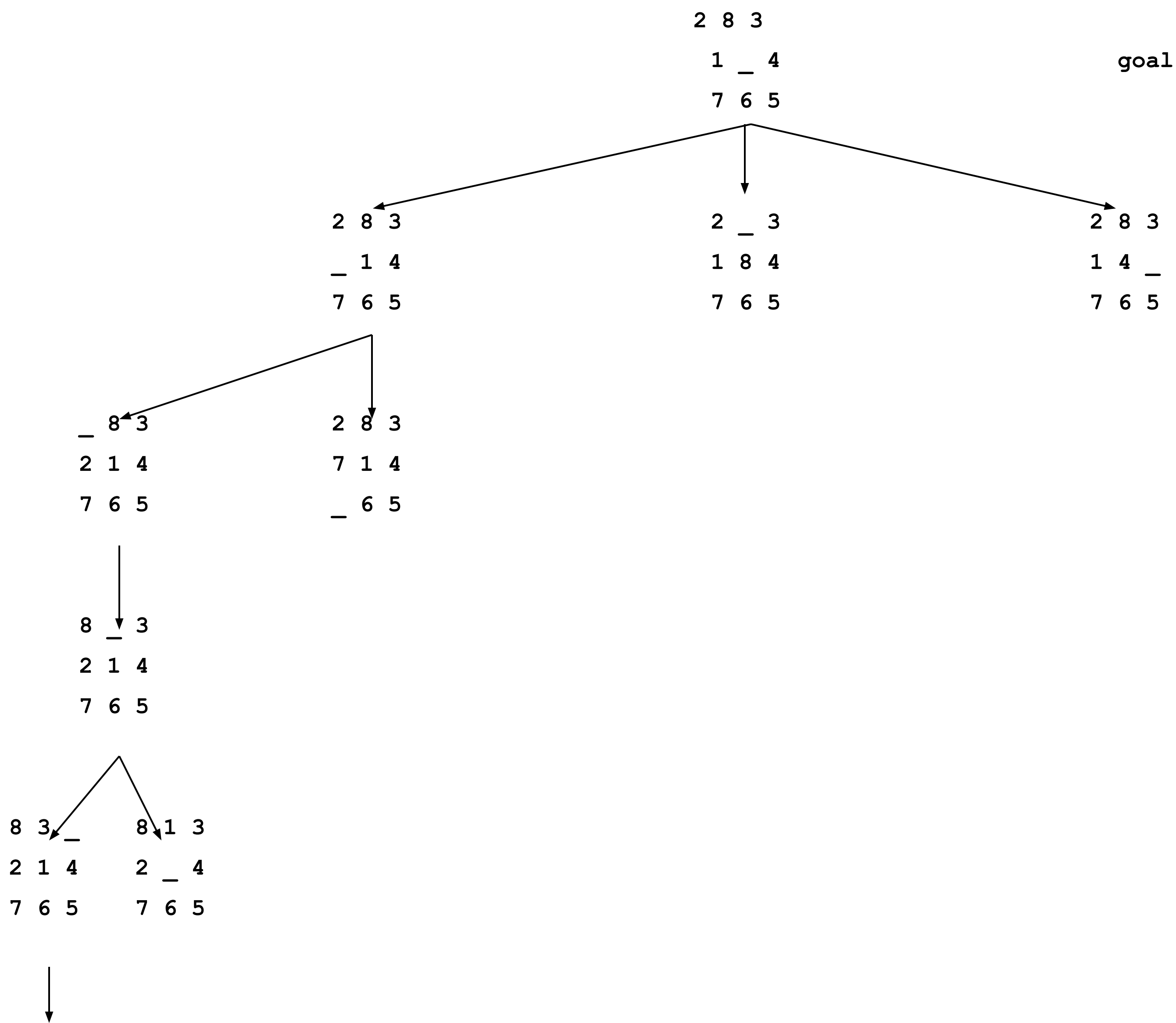
Tile puzzle -- exhaustive depth-first search



Tile puzzle -- exhaustive depth-first search



Tile puzzle -- exhaustive depth-first search



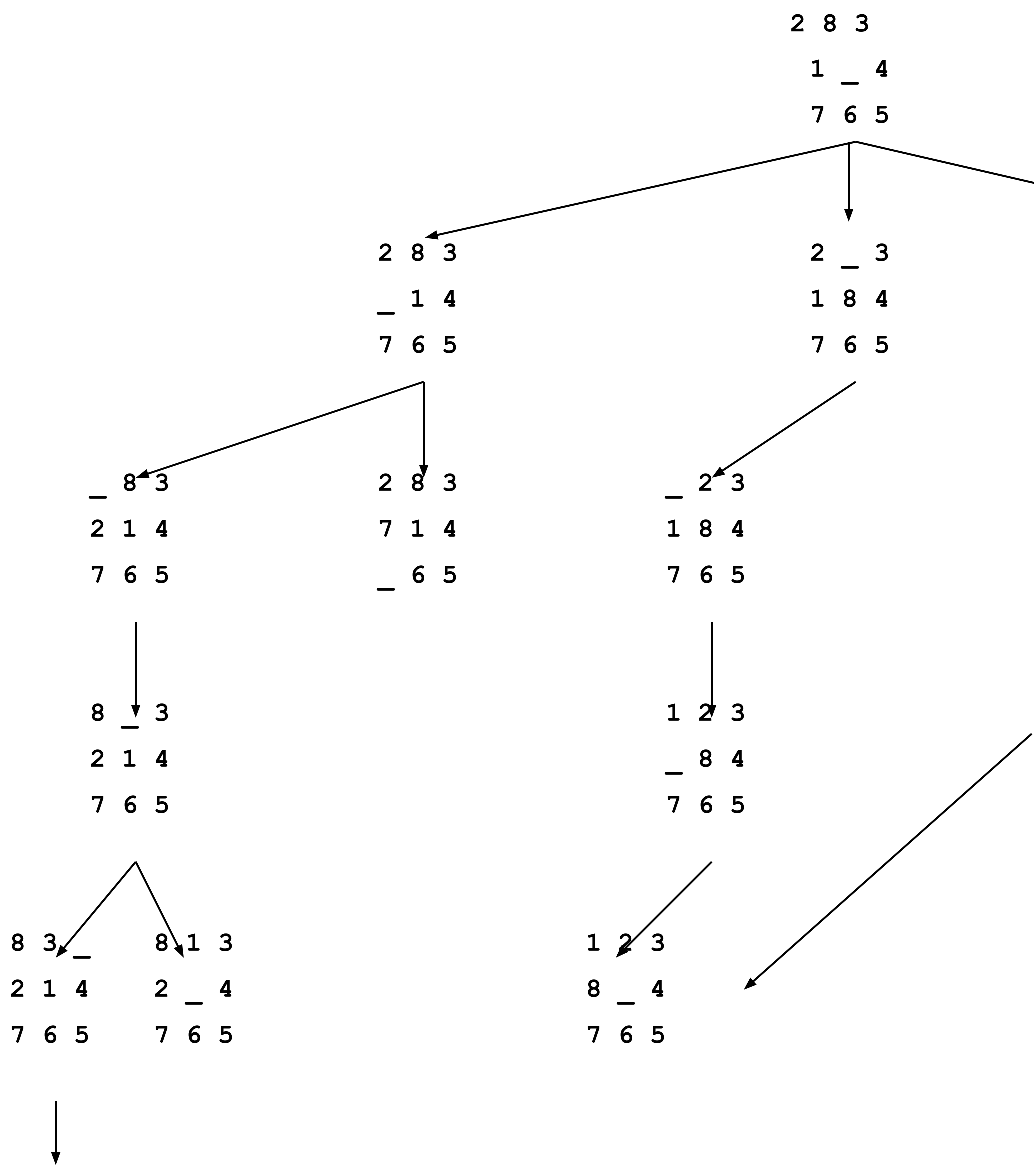

```
       1 2 3
goal:  8 _ 4
       7 6 5
```

Tile puzzle -- exhaustive depth-first search

```
                          2 8 3                    1 2 3
                          1 _ 4             goal:  8 _ 4
                          7 6 5                    7 6 5

          2 8 3           2 _ 3           2 8 3
          _ 1 4           1 8 4           1 4 _
          7 6 5           7 6 5           7 6 5

  _ 8 3   2 8 3
  2 1 4   7 1 4
  7 6 5   _ 6 5

  8 _ 3
  2 1 4
  7 6 5

8 3 _   8 1 3
2 1 4   2 _ 4
7 6 5   7 6 5
```

The problem with this approach is...

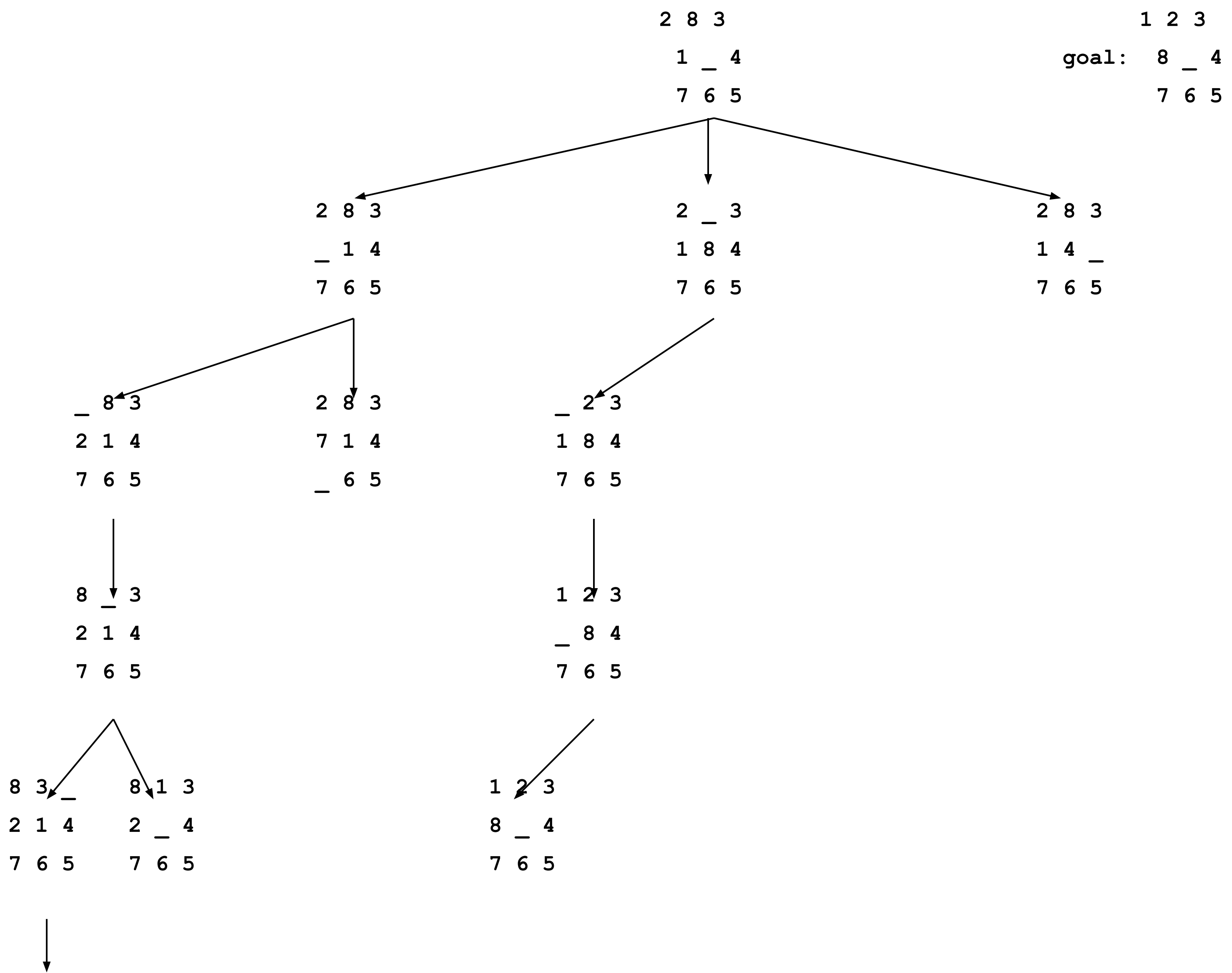
Tile puzzle -- exhaustive depth-first search




```
       1 2 3
goal:  8 _ 4
       7 6 5
```

The problem with this approach is the solution is over here, only 4 moves from the start state

What's a good heuristic here?



Best-first search - tiles out of place

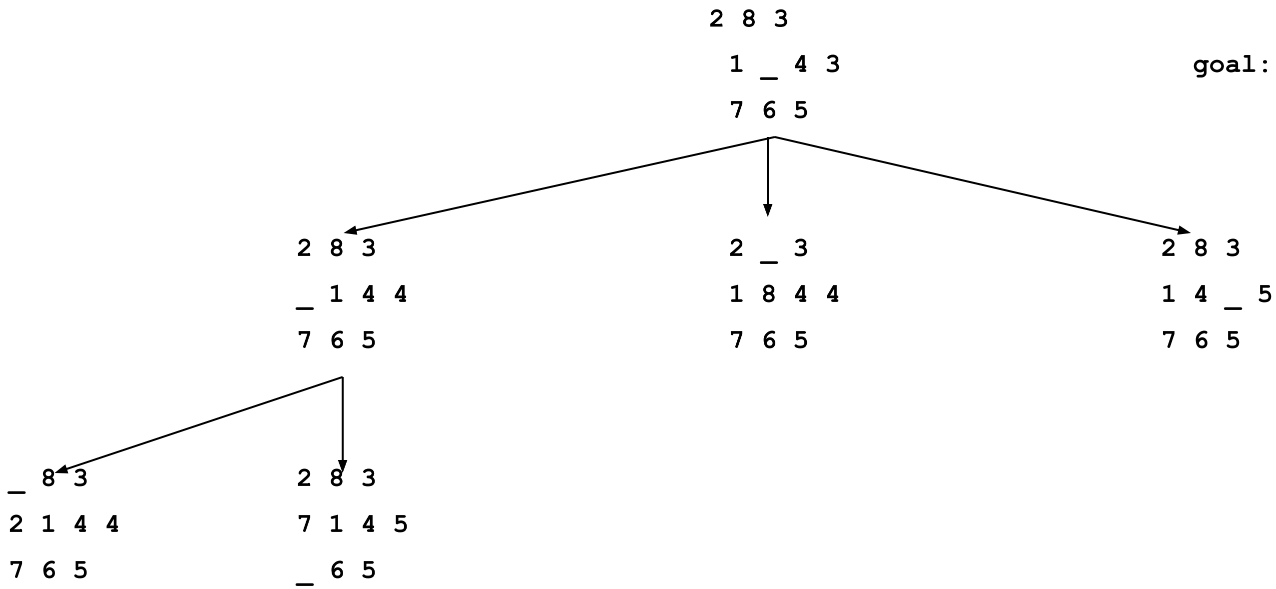
```
2 8 3
 1 _ 4 3
 7 6 5
```

```
      1 2 3
goal:  8 _ 4
       7 6 5
```

Best-first search - tiles out of place

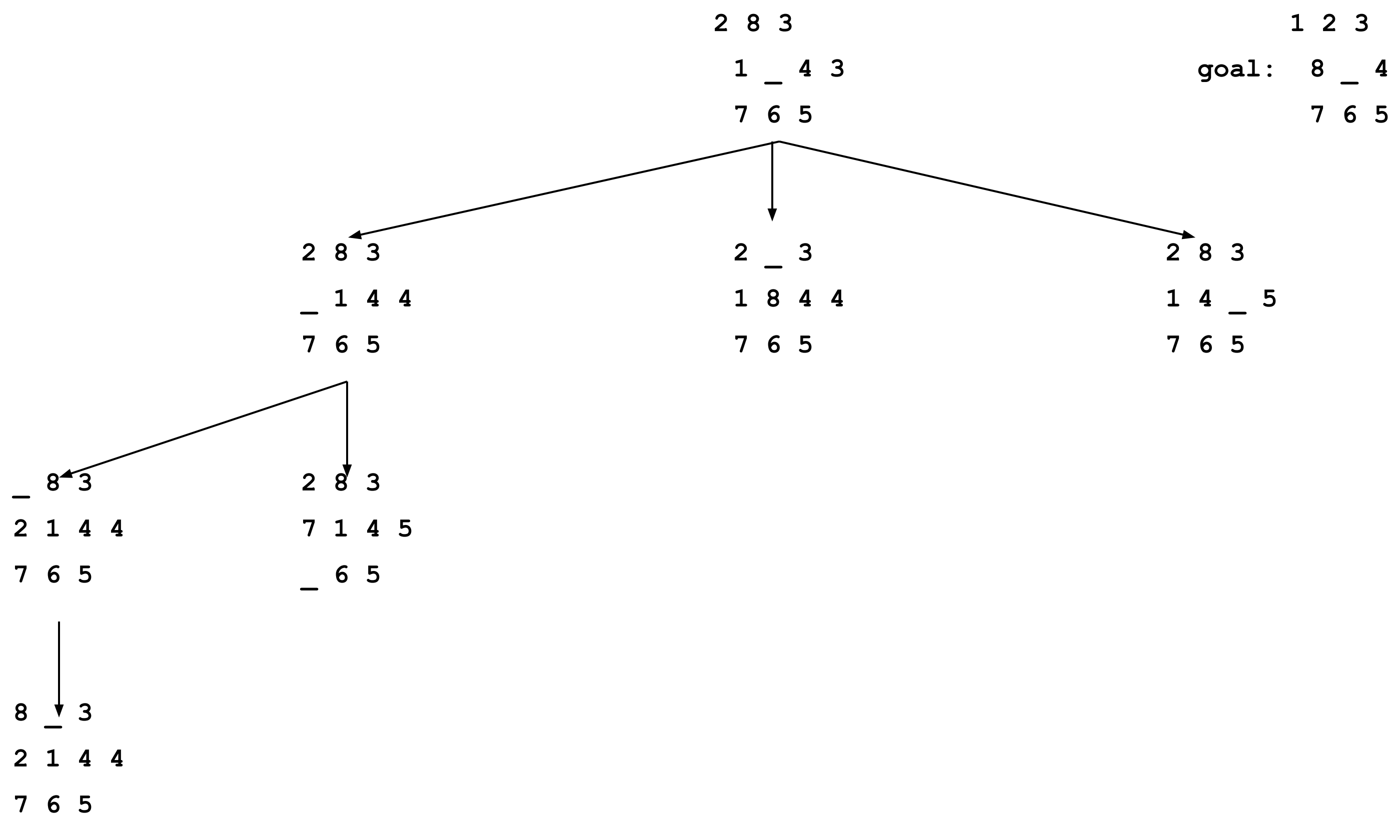
```
                         2 8 3                         1 2 3
                          1 _ 4 3              goal:  8 _ 4
                          7 6 5                        7 6 5

2 8 3                     2 _ 3                 2 8 3
_ 1 4 4                   1 8 4 4               1 4 _ 5
7 6 5                     7 6 5                 7 6 5
```

Best-first search - tiles out of place

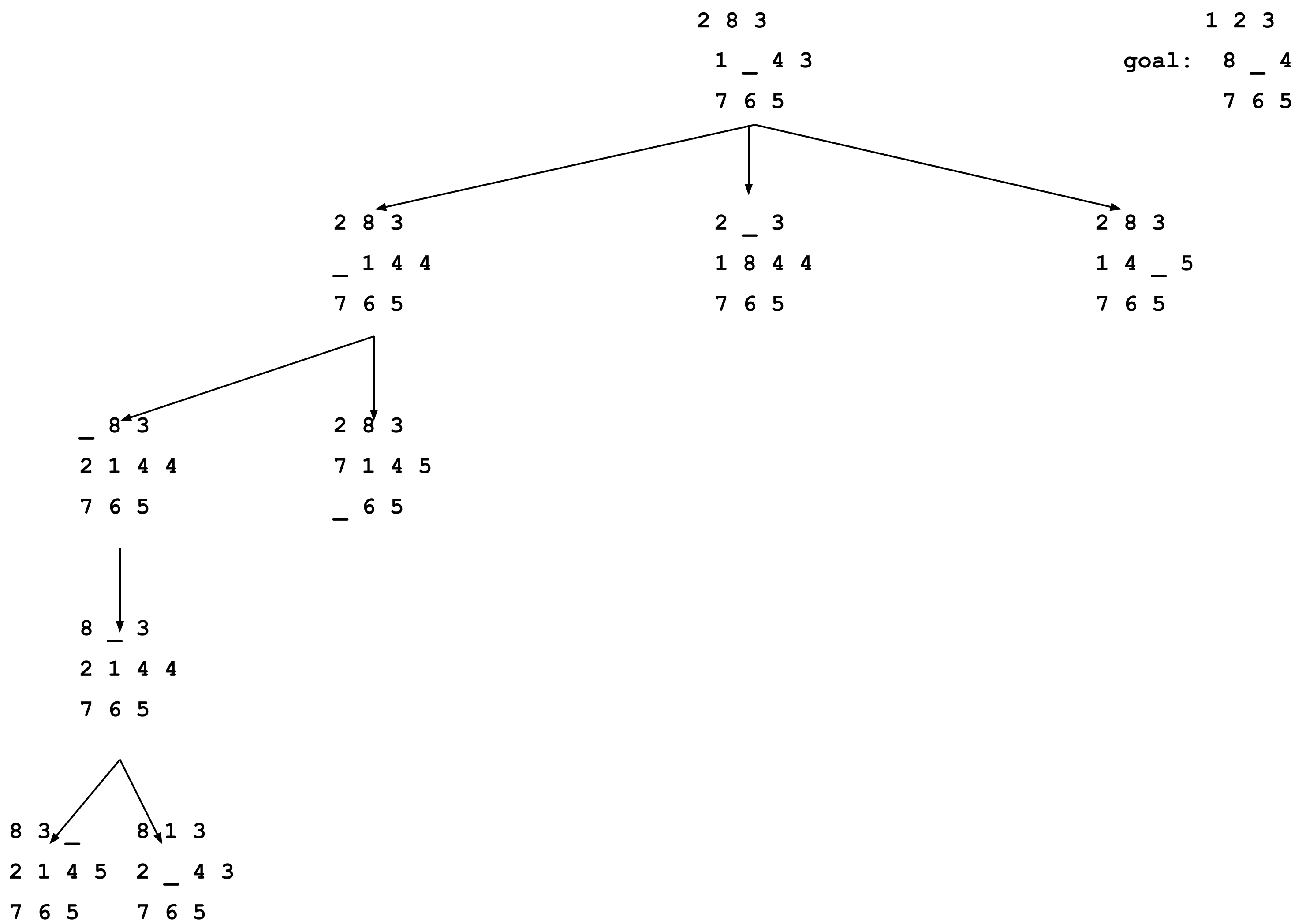



```
      1 2 3
goal:  8 _ 4
       7 6 5
```

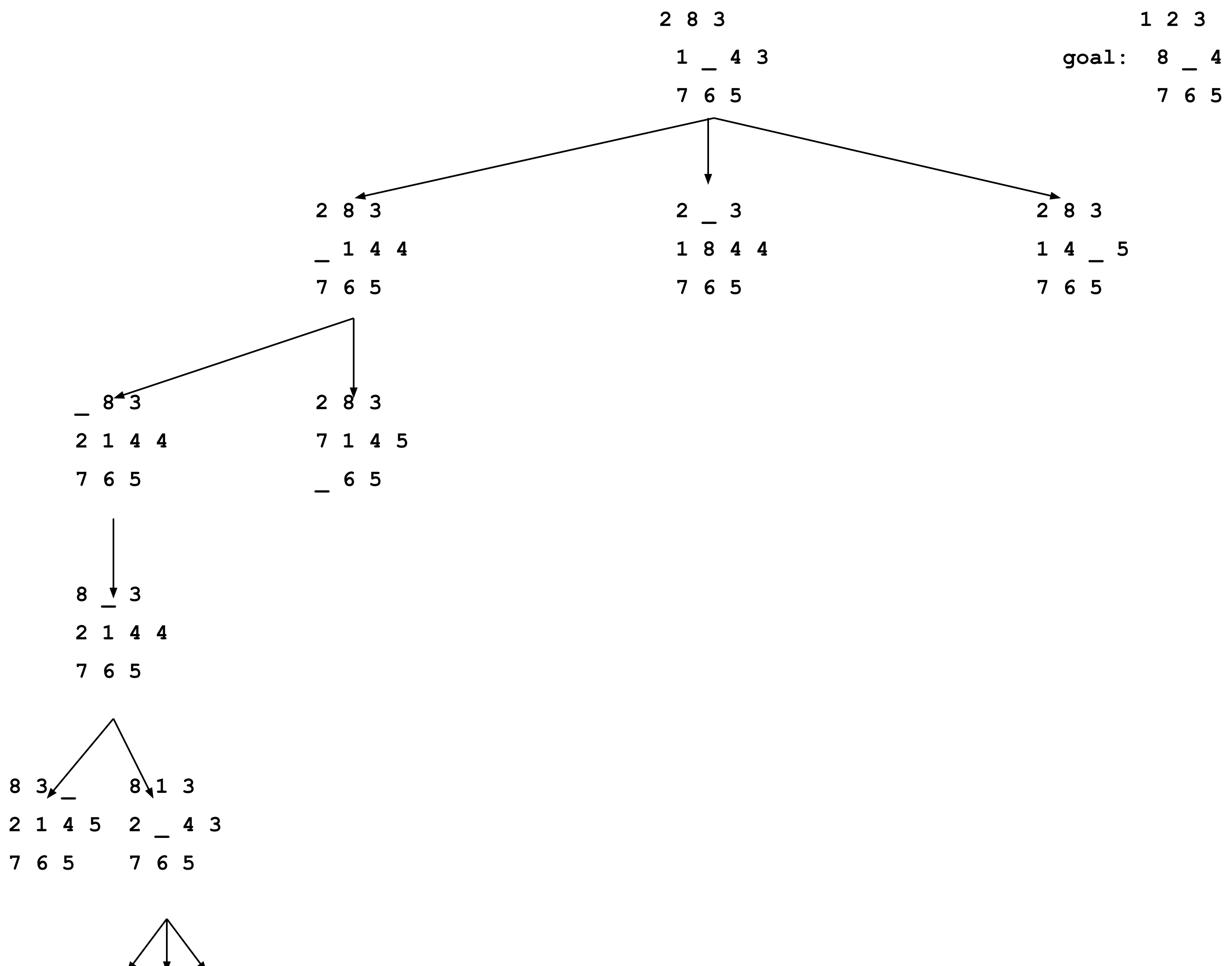
Best-first search - tiles out of place



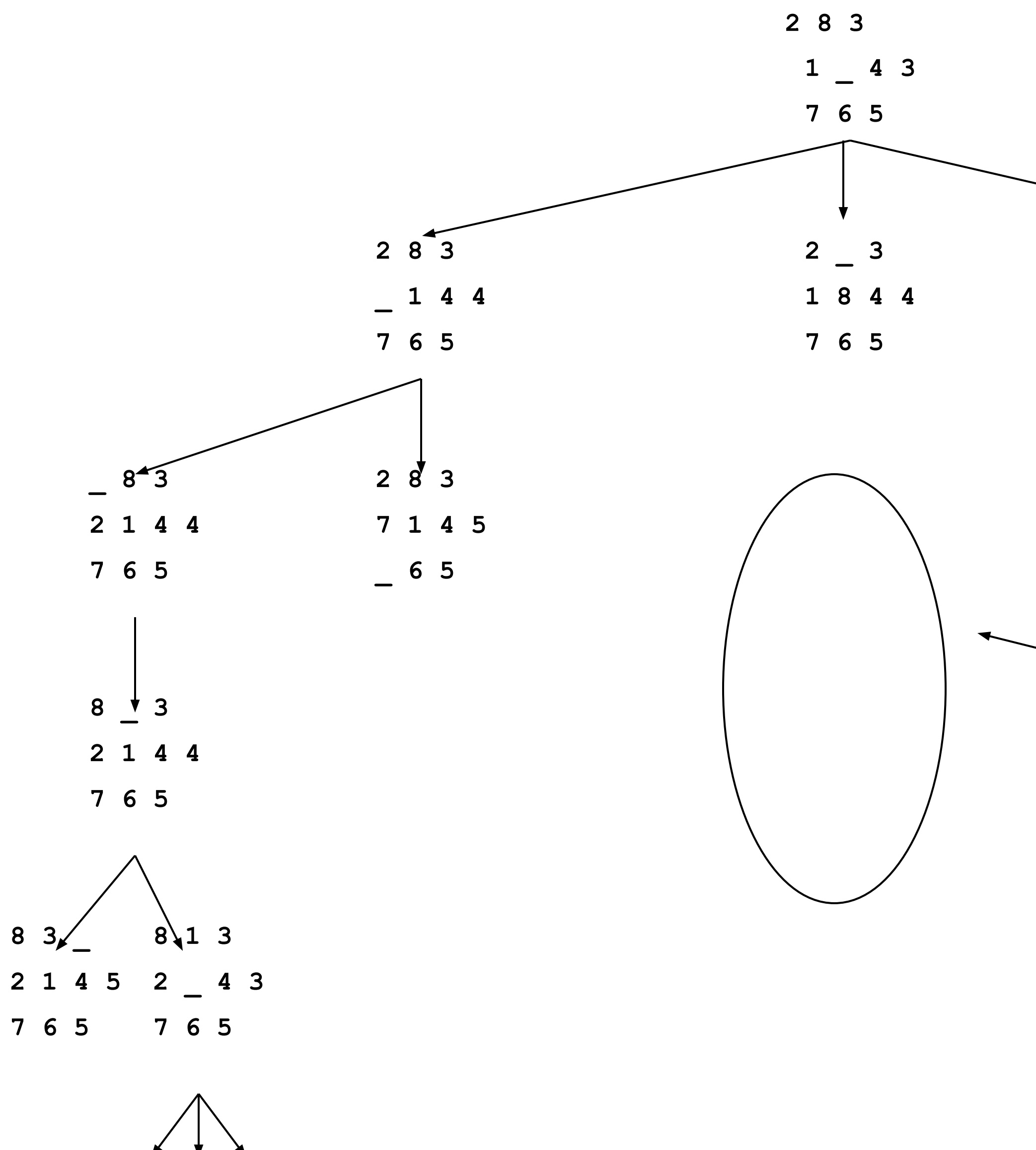
Best-first search - tiles out of place



Best-first search - tiles out of place



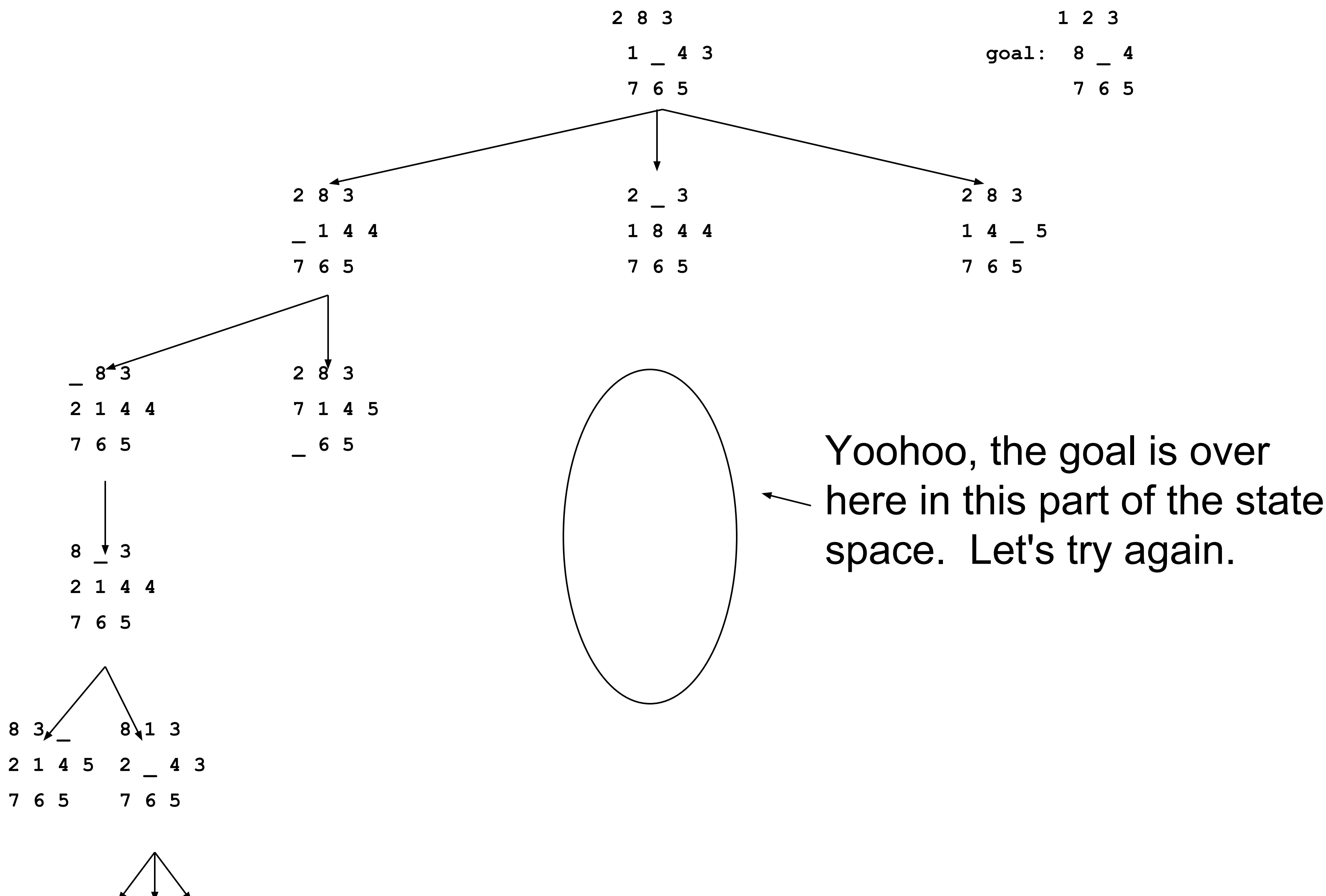
Best-first search - tiles out of place



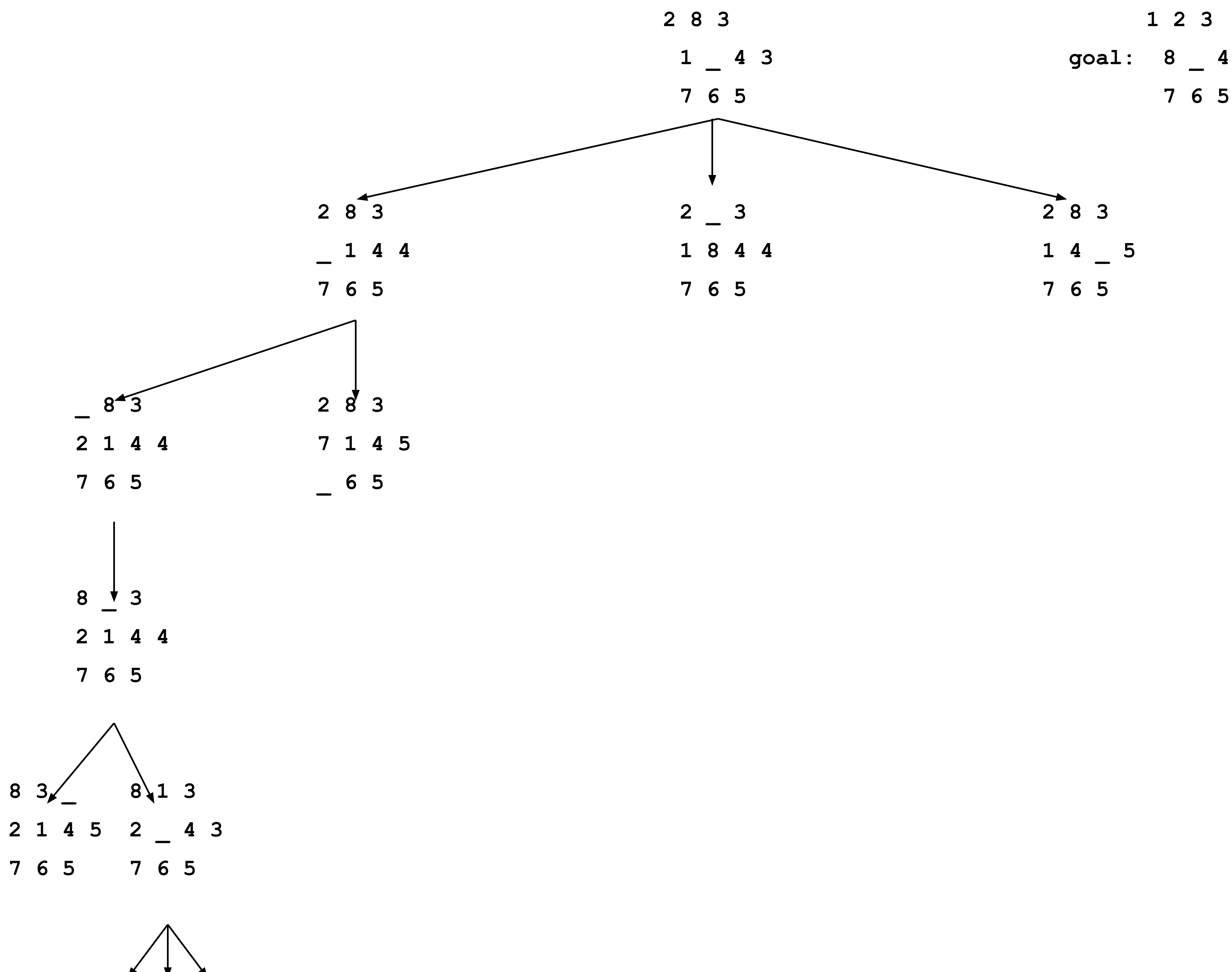

```
      1 2 3
goal: 8 _ 4
      7 6 5
```

Yoohoo, the goal is over here in this part of the state space. Let's try again.

How about a better heuristic?



How about a better heuristic?



Best-first search - Manhattan distance

```
2 8 3                 1 2 3
 1 _ 4         goal:   8 _ 4
 7 6 5                 7 6 5
```

Best-first search - Manhattan distance

Manhattan district in New York City: streets based on grid system of roughly equal-size blocks

shortest distance between two points by taxicab is the sum of the absolute values of the differences of their coordinates

the distance from 8th Avenue and 42nd Street to 5th Avenue and 55th Street is 3 + 13 = 16 city blocks

also called rectilinear distance or city block distance

Best-first search - Manhattan distance

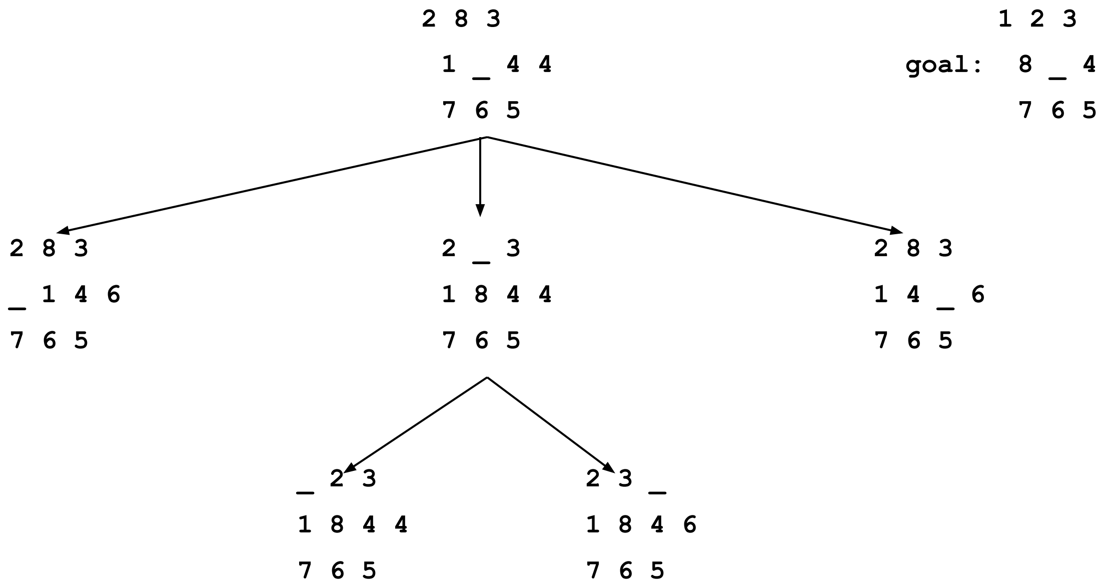
```
 2 8 3                      1 2 3
  1 _ 4 4           goal:   8 _ 4
  7 6 5                     7 6 5
```

For the tile puzzle, we want to know, for each tile, what's the "Manhattan distance" between where the tile is now and where it needs to be to satisfy the goal state. In the example above, three tiles are out of place. The '1' and '2' tiles are a Manhattan distance of 1 away from where they need to be. The '8' tile is a Manhattan distance of 2 away. So the total Manhattan distance is 4.

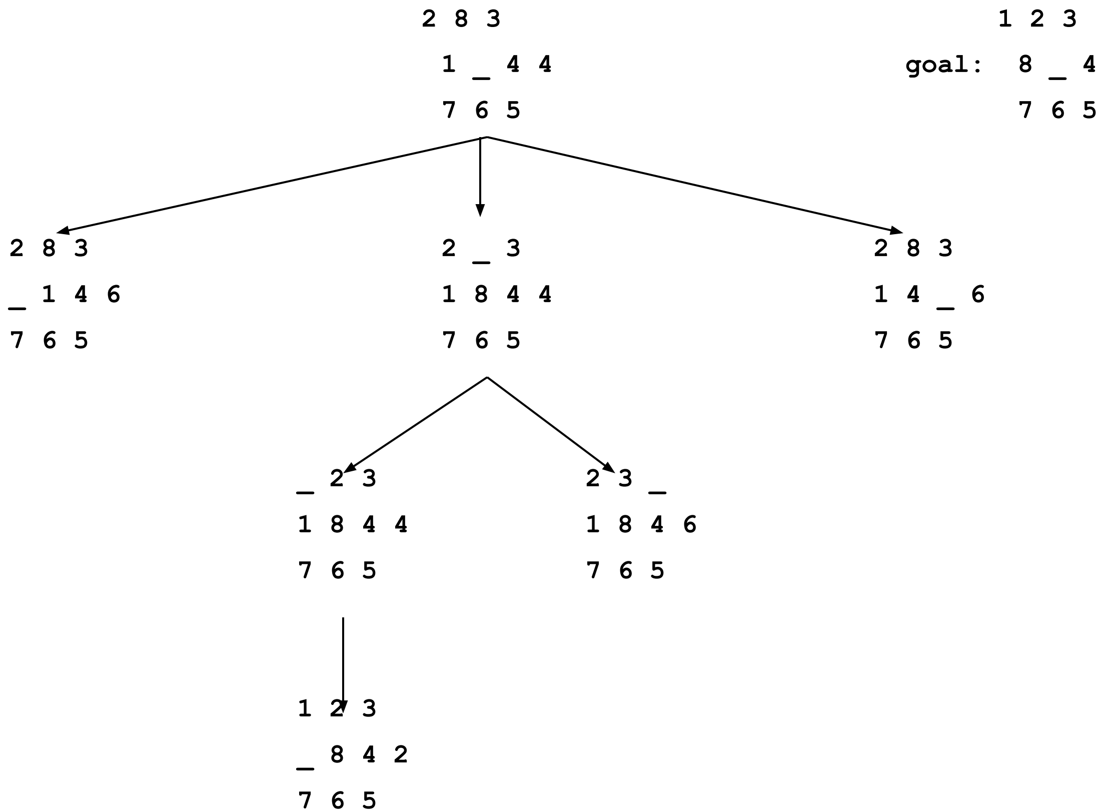
Best-first search - Manhattan distance

```
                         2 8 3                         1 2 3
                          1 _ 4 4               goal:  8 _ 4
                          7 6 5                         7 6 5

2 8 3                     2 _ 3                   2 8 3
_ 1 4 6                   1 8 4 4                 1 4 _ 6
7 6 5                     7 6 5                   7 6 5
```

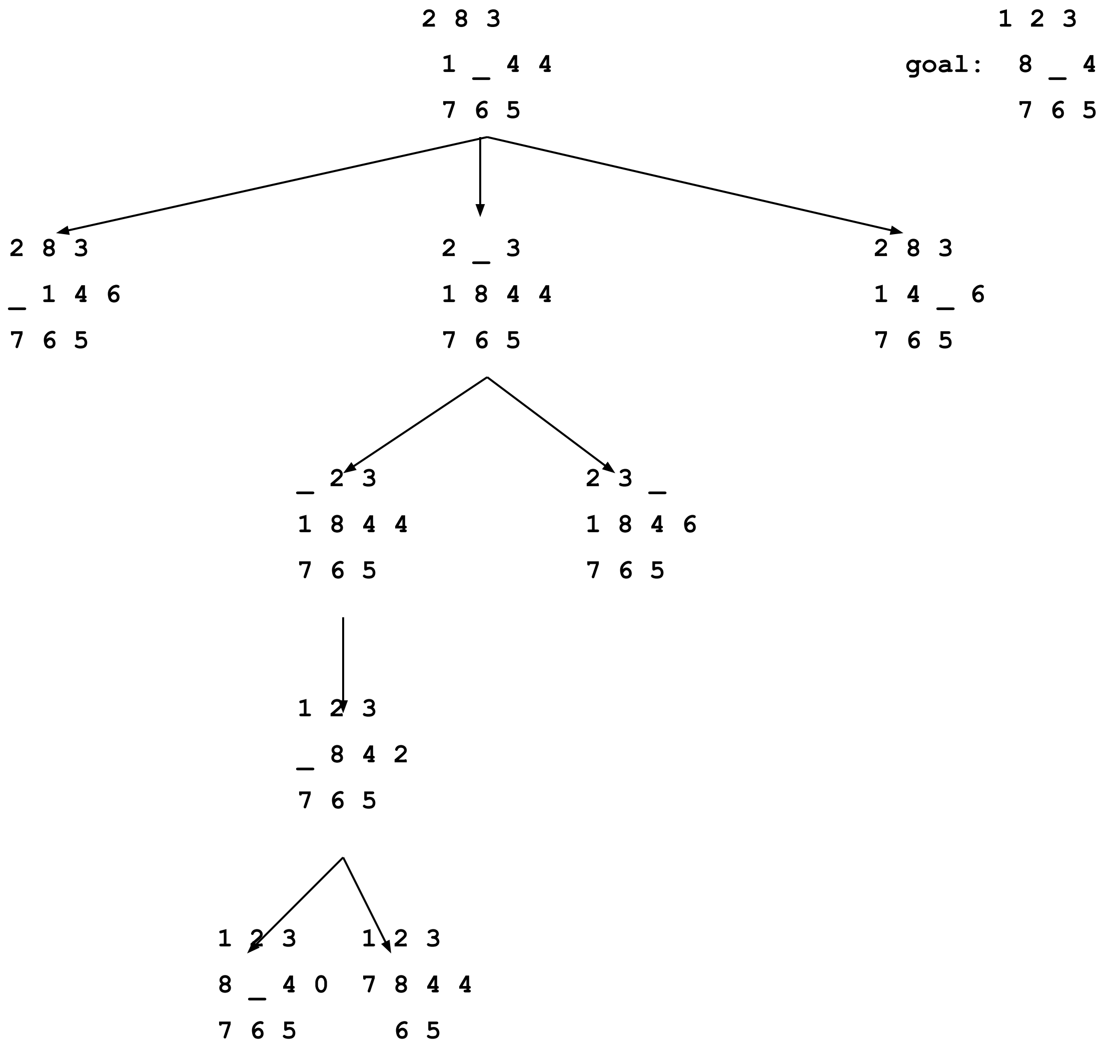
Best-first search - Manhattan distance



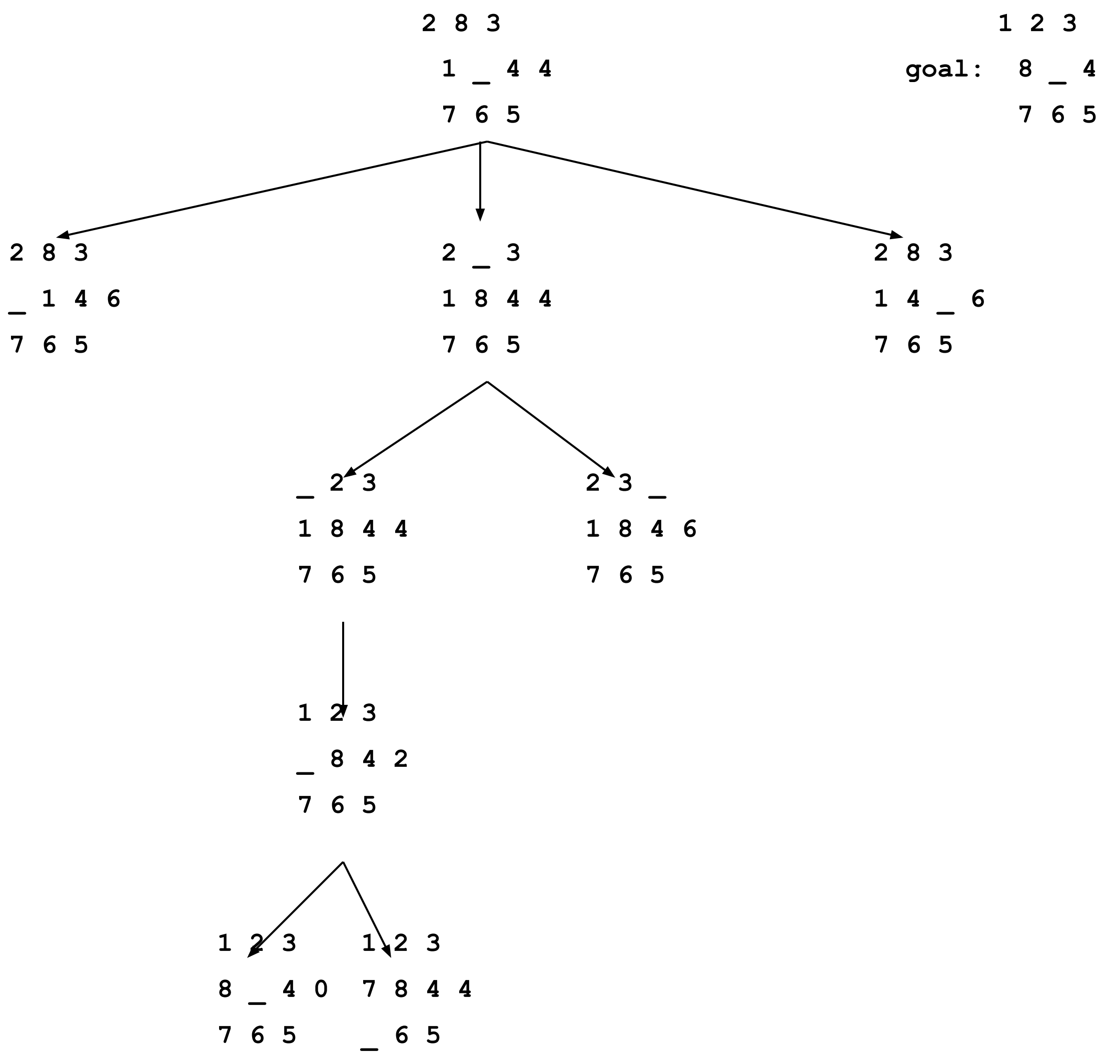
Best-first search - Manhattan distance



Best-first search - Manhattan distance



Best-first search - Manhattan distance



Quantifying goodness

You've now seen a couple of real examples of attempts to quantify the nearness of a state to the goal state - the goodness of the state. Crafting these heuristics is a skill that takes practice, and even with lots of practice your heuristics will still be wrong some of the time.

But it's something to think about. Ponder some heuristics for other puzzles you might be familiar with. Your opportunity to put heuristics in your Haskell programs is coming up real soon.

Questions?

The moral of the story so far...

We can write an evaluation function that, when applied to some state, uses knowledge about the problem domain to calculate a quantitative value that represents an estimate of that state's nearness to a goal.

That quantitative value can then be used to answer that question: Now what do I do?

Search in the real world

The sort of heuristic state-space search we've seen only gets us so far in the real world, because the real world can be a hostile place.

Consequently, we often find ourselves in situations where we're trying to find the path to our goal while somebody else is trying to prevent us from getting there.

Search in the real world

Real examples include

- the world of commerce
- the athletic field
- the battlefield
- organic chem lab when pre-med students are enrolled

Search in the real world

Real examples include

- the world of commerce
- the athletic field
- the battlefield
- organic chem lab when pre-med students are enrolled
- the simplest example is the two-player game, which leads us to...

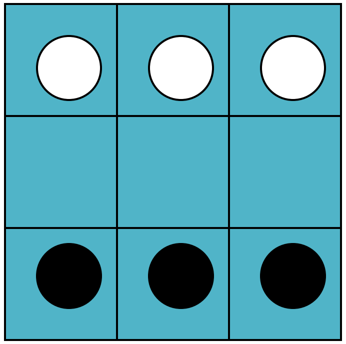
The Joy of Hex

The game of hexapawn

The Joy of Hex

The game of hexapawn

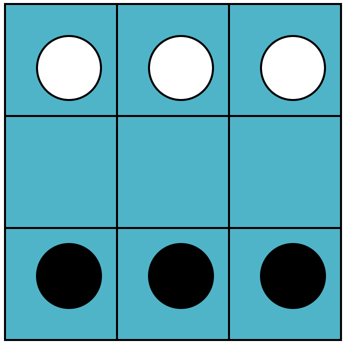
- 3 x 3 board
- 3 pawns on each side



The Joy of Hex

The game of hexapawn

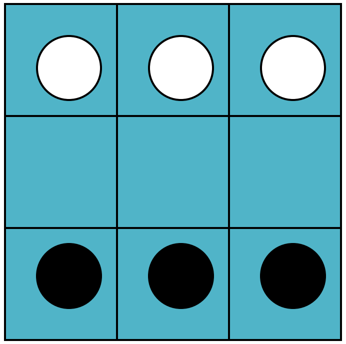
- 3 x 3 board
- 3 pawns on each side
- movement of pawns:



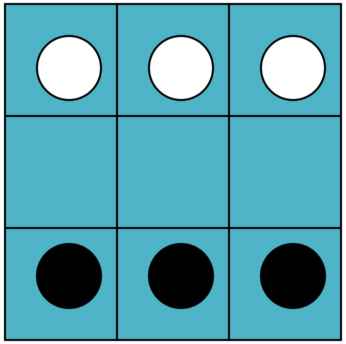
The Joy of Hex

The game of hexapawn

- 3 x 3 board
- 3 pawns on each side
- movement of pawns:
  - white moves first



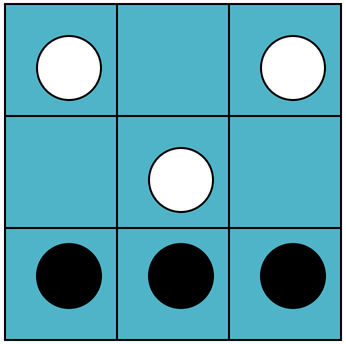
The Joy of Hex



The game of hexapawn

- 3 x 3 board
- 3 pawns on each side
- movement of pawns:
  - white moves first
  - pawn can move straight ahead one space if that space is empty

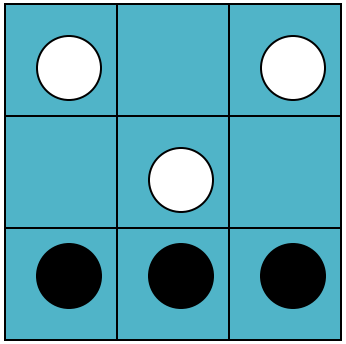
The Joy of Hex



The game of hexapawn

- 3 x 3 board
- 3 pawns on each side
- movement of pawns:
  - white moves first
  - pawn can move straight ahead one space if that space is empty

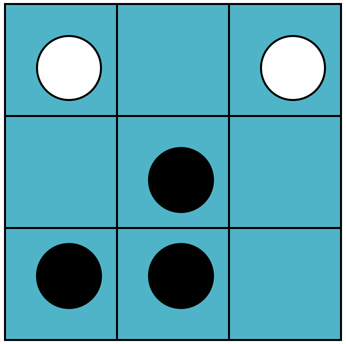
The Joy of Hex



The game of hexapawn

- 3 x 3 board
- 3 pawns on each side
- movement of pawns:
  - white moves first
  - pawn can move straight ahead one space if that space is empty
  - pawn can move diagonally one space forward to capture opponent's pawn occupying that space

The Joy of Hex

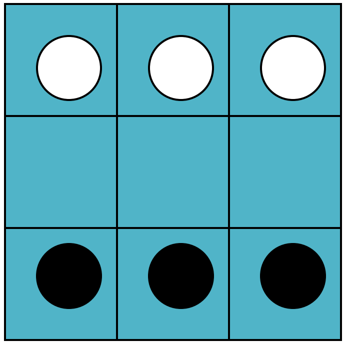


The game of hexapawn

- 3 x 3 board
- 3 pawns on each side
- movement of pawns:
  - white moves first
  - pawn can move straight ahead one space if that space is empty
  - pawn can move diagonally one space forward to capture opponent's pawn occupying that space

The Joy of Hex

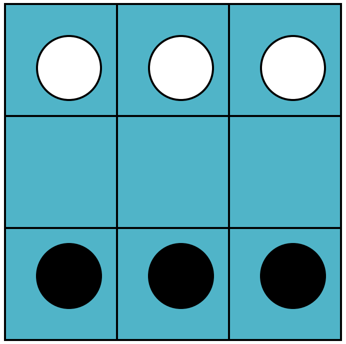
The game of hexapawn



- 3 ways to win:

The Joy of Hex

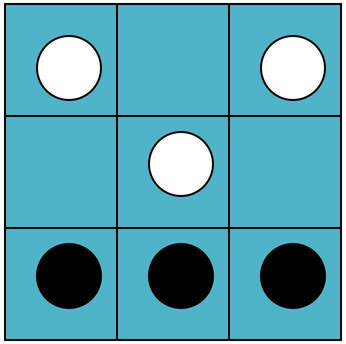
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns

The Joy of Hex

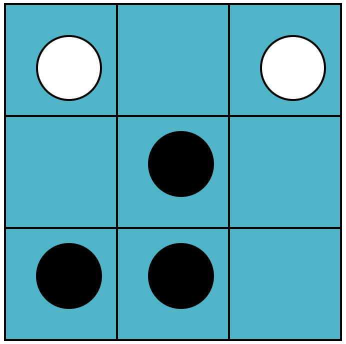
The game of hexapawn



- 3 ways to win:
  - capture all your opponent's pawns

The Joy of Hex

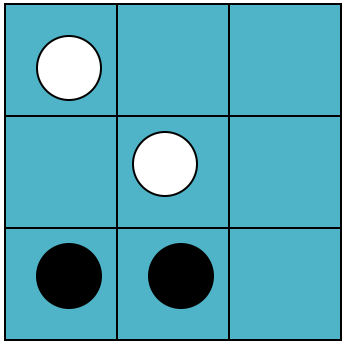
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns

The Joy of Hex

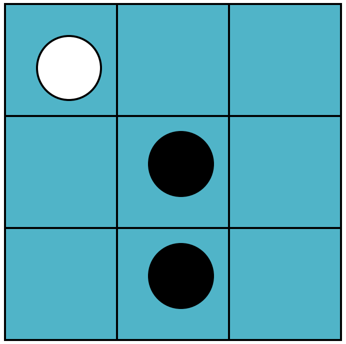
The game of hexapawn



- 3 ways to win:
  - capture all your opponent's pawns

The Joy of Hex

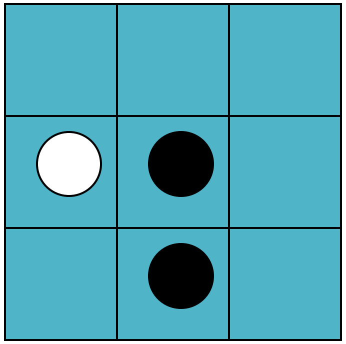
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns

The Joy of Hex

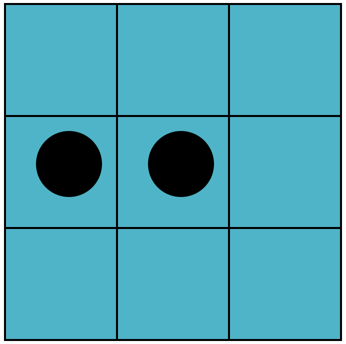
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns

The Joy of Hex

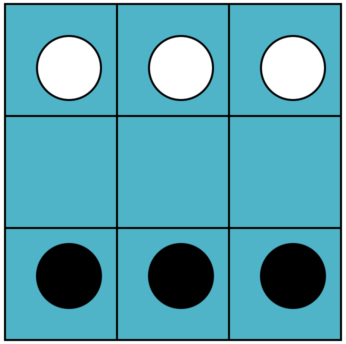
The game of hexapawn



- 3 ways to win:
  - capture all your opponent's pawns

The Joy of Hex

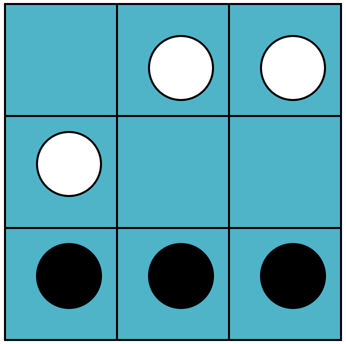
The game of hexapawn



- 3 ways to win:
  - capture all your opponent's pawns
  - one of your pawns reaches the opposite end of the board

The Joy of Hex

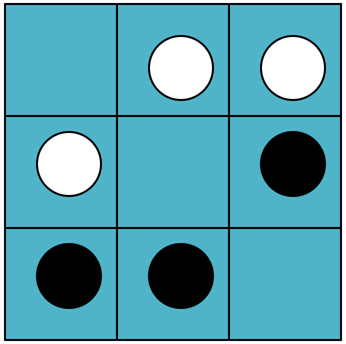
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns
  - one of your pawns reaches the opposite end of the board

The Joy of Hex

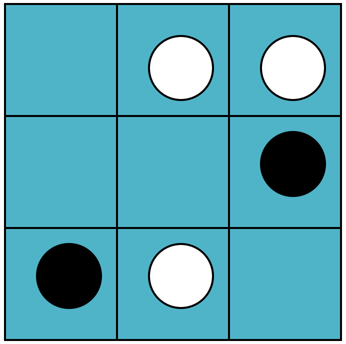
The game of hexapawn



- 3 ways to win:
  - capture all your opponent's pawns
  - one of your pawns reaches the opposite end of the board

The Joy of Hex

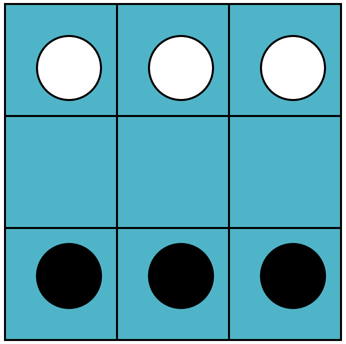
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns
  - one of your pawns reaches the opposite end of the board

The Joy of Hex

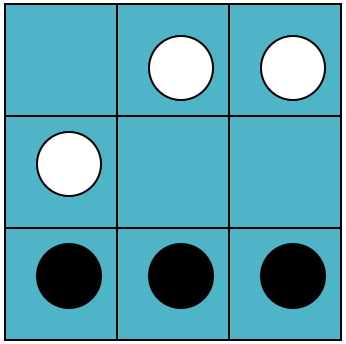
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns
  - one of your pawns reaches the opposite end of the board
  - it’s your opponent’s turn but your opponent can’t move

The Joy of Hex

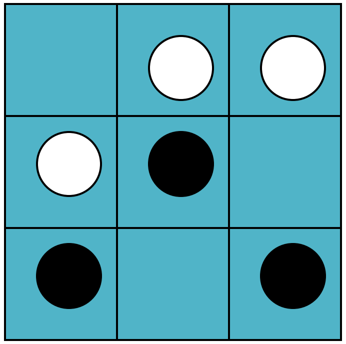
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns
  - one of your pawns reaches the opposite end of the board
  - it’s your opponent’s turn but your opponent can’t move

The Joy of Hex

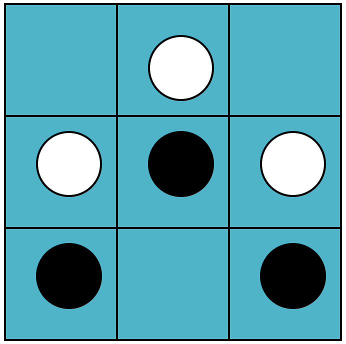
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns
  - one of your pawns reaches the opposite end of the board
  - it’s your opponent’s turn but your opponent can’t move

The Joy of Hex

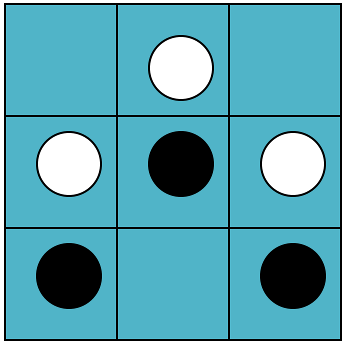
The game of hexapawn



- 3 ways to win:
  - capture all your opponent’s pawns
  - one of your pawns reaches the opposite end of the board
  - it’s your opponent’s turn but your opponent can’t move

The Joy of Hex

The game of hexapawn



- 3 ways to win:
  - capture all your opponent's pawns
  - one of your pawns reaches the opposite end of the board
  - it's your opponent's turn but your opponent can't move

Now let's look at the search tree...
(we're pushing the black pawns)

Questions?