IMPROVING SEARCH EFFICIENCY

JEFFREY L. POPYACK

IMPROVING SEARCH EFFICIENCY

- Ordering Alternatives via Heuristics
  - Modifying search methods to include use of heuristics
- Implicit Enumeration
  - Modifying search methods for implicit enumeration
- Symmetry
  - Modifying search methods to take advantage of symmetry
- Forward-Backward Search
- Forward-Backward Search and Symmetry

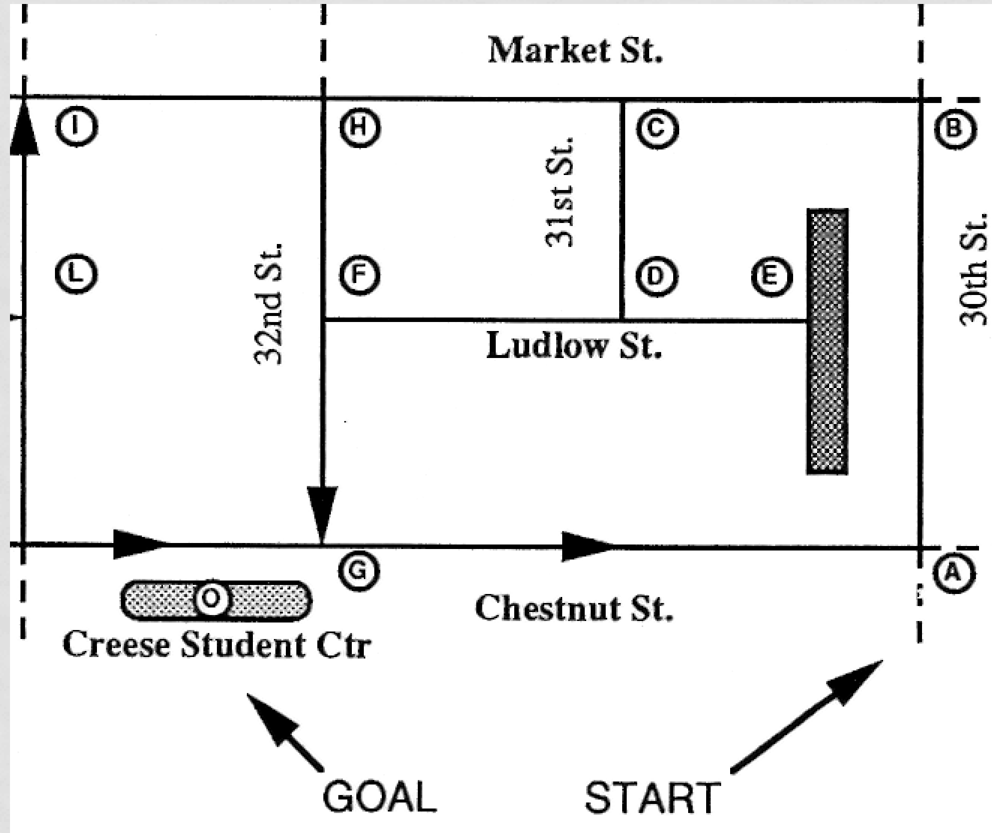
ORDERING ALTERNATIVES: HEURISTICS

RECALL:
Backtracking Example

When at C, choices are { ↓ → ← }

<u>Before</u>: try these in some fixed order (e.g. { ↓ → ← })

<u>Now</u> : use a *heuristic* to suggest a more intelligent order to try these in. ("educated guess")



Drexel Campus, Early 1980's

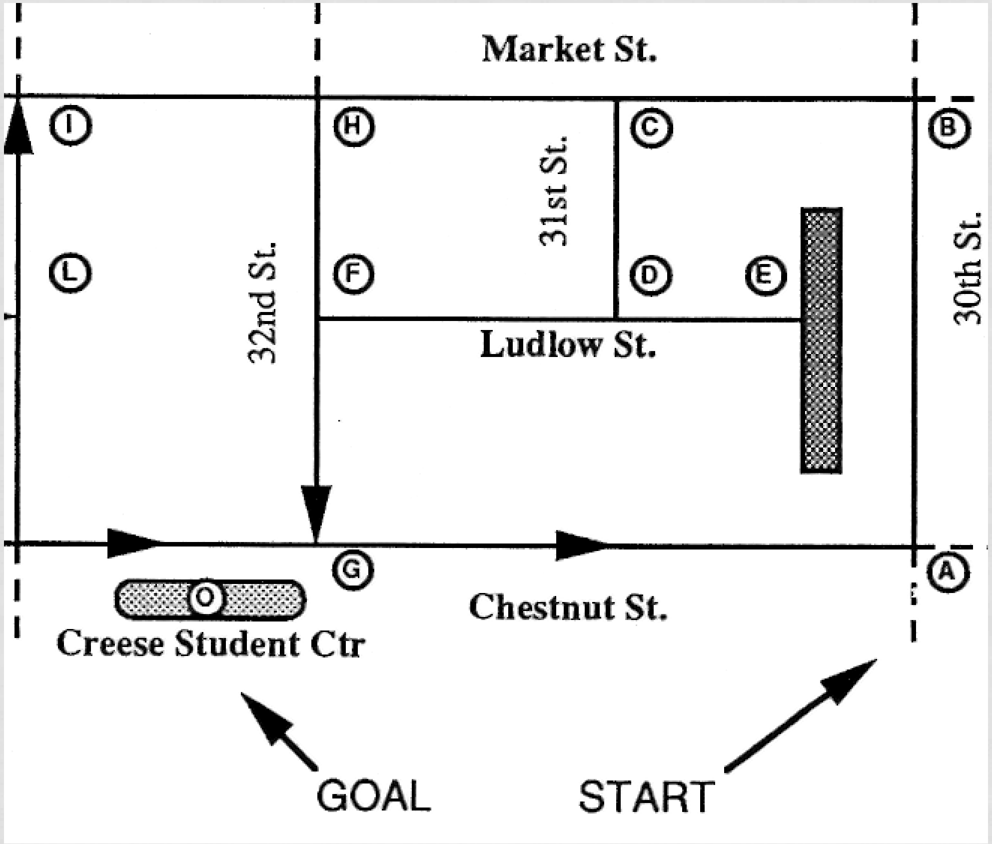
ORDERING ALTERNATIVES: HEURISTICS

RECALL:
Backtracking Example

When at C, choices are { ↓ → ← }

Before: try these in some fixed order (e.g. { ↓ → ← })

Now : use a *heuristic* to suggest a more intelligent order to try these in. ("educated guess")



e.g, let *h(state,rule)* ε R . **Try r_1 before r_2 if *h(state, r_1) < h(state, r_2)***

ORDERING ALTERNATIVES: HEURISTICS

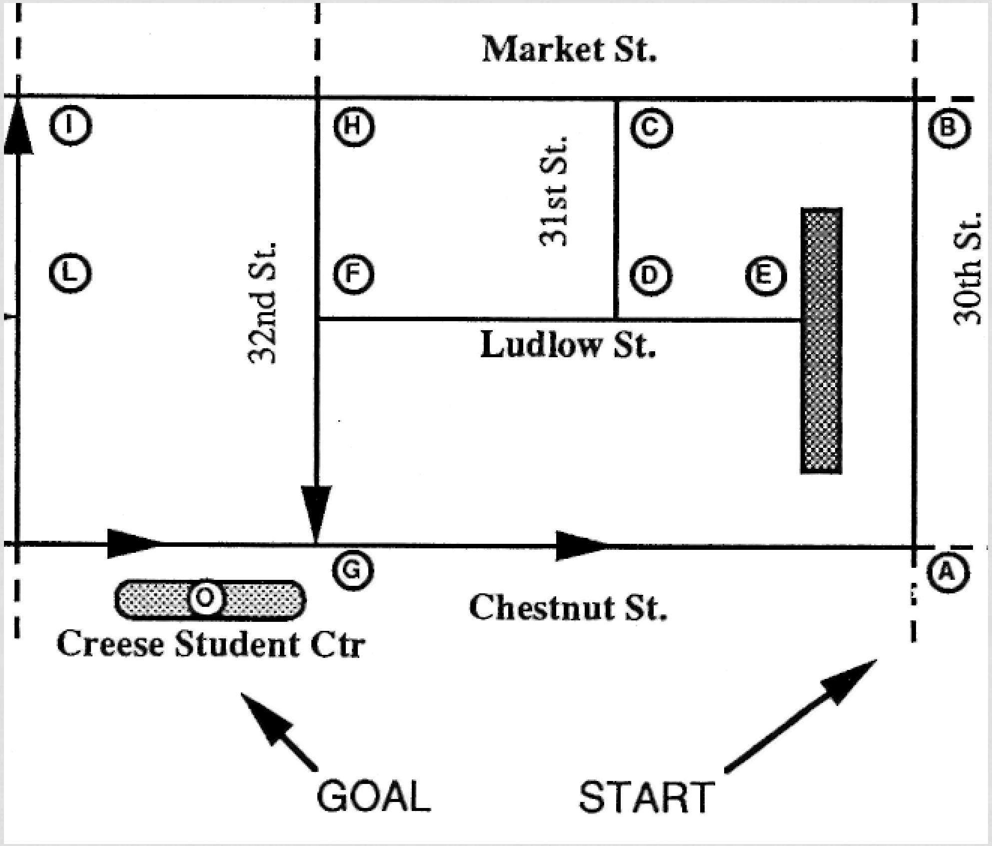
For this problem:

If state = s
& Apply(r_1,s) $\rightarrow$ s_1 ,
& Apply(r_2,s) $\rightarrow$ s_2 ,

let *h(s,r_i) = dist(s_i,goal)*

So, from C, can either choose:

$r_1 = \downarrow$, $r_2 = \rightarrow$, $r_3 = \leftarrow$.
s_1= D, r_2= B, r_3= H.



Drexel Campus, Early 1980's

ORDERING ALTERNATIVES: HEURISTICS

For this problem:

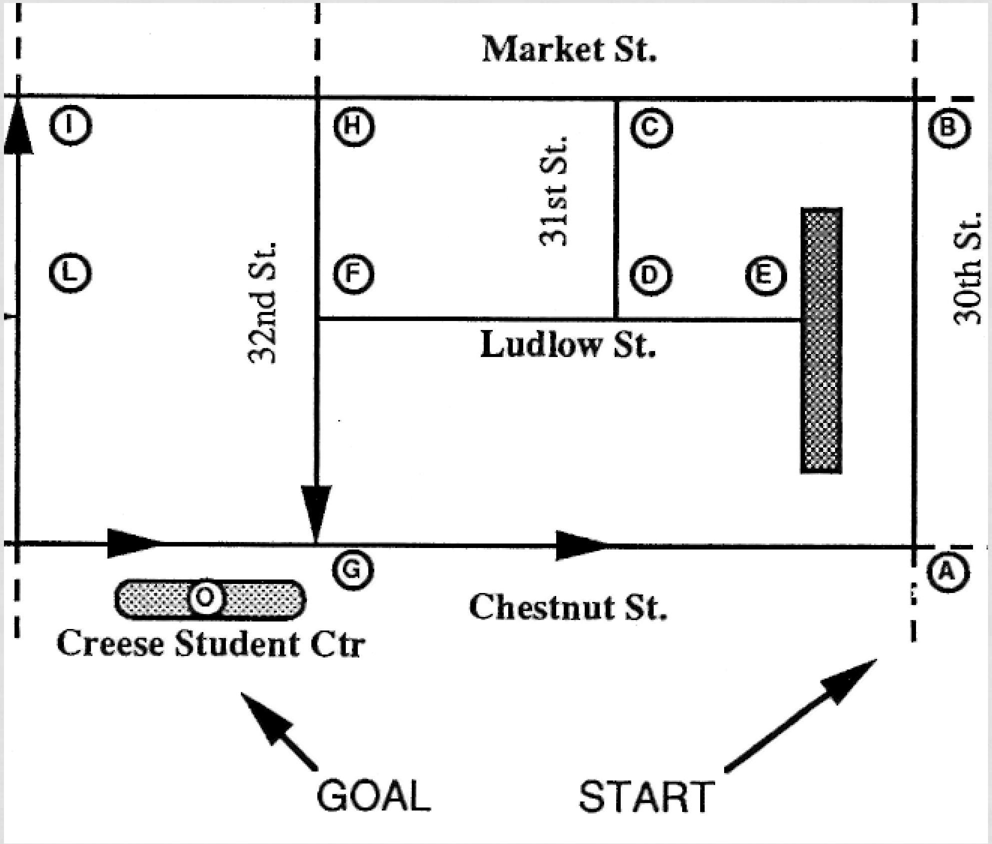
If state = s
& Apply(r_1,s) → s_1 ,
& Apply(r_2,s) → s_2 ,

let $h(s,r_i) = dist(s_i, goal)$

So, from C, can either choose:

$r_1 = \downarrow$, $r_2 = \rightarrow$, $r_3 = \leftarrow$.
s_1= D, r_2= B, r_3= H.

$h(s,r_1) = dist(\ D, goal\ )$
$h(s,r_2) = dist(\ B, goal\ )$
$h(s,r_3) = dist(\ H, goal\ )$



Drexel Campus, Early 1980's

ORDERING ALTERNATIVES: HEURISTICS

For this problem:

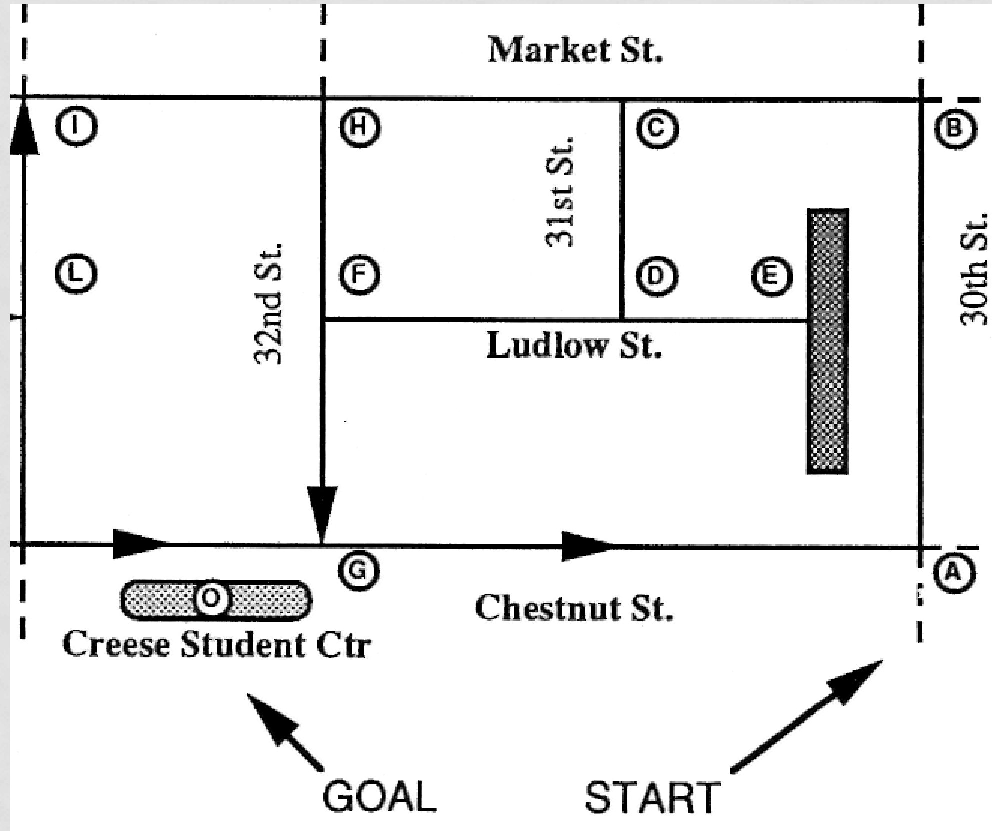
If state = s
& Apply(r_1,s) → s_1 ,
& Apply(r_2,s) → s_2 ,

let *$h(s,r_i) = dist(s_i, goal)$*

So, from C, can either choose:

$r_1 = \downarrow$, $r_2 = \rightarrow$, $r_3 = \leftarrow$.
s_1= D, r_2= B, r_3= H.

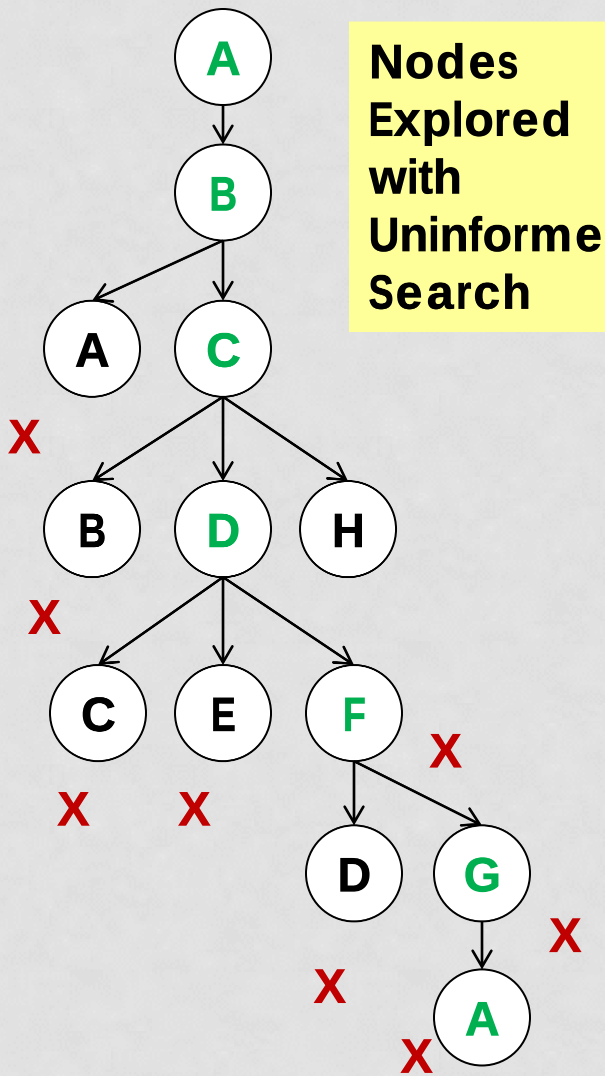
$h(s,r_1) = dist($ D $, goal)$
$h(s,r_2) = dist($ B $, goal)$
$h(s,r_3) = dist($ H $, goal)$



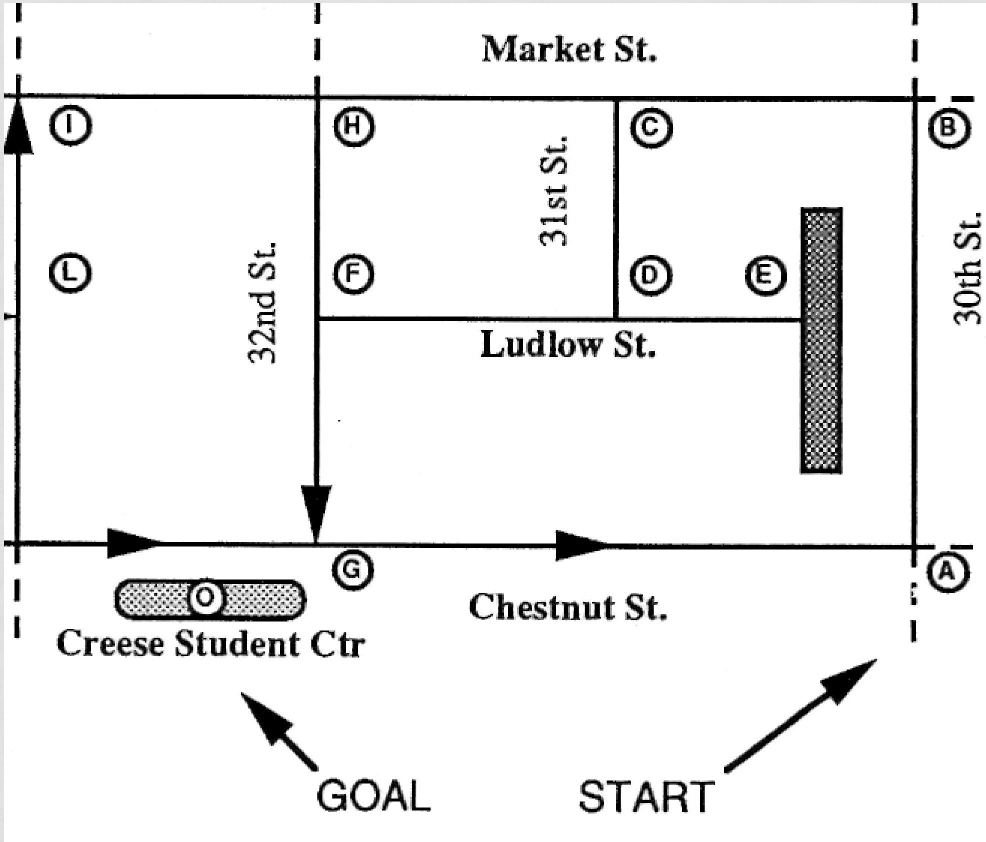
Drexel Campus, Early 1980's

Here, *$h(s,r_3) < h(s,r_1) < h(s,r_2)$* *(probably)*

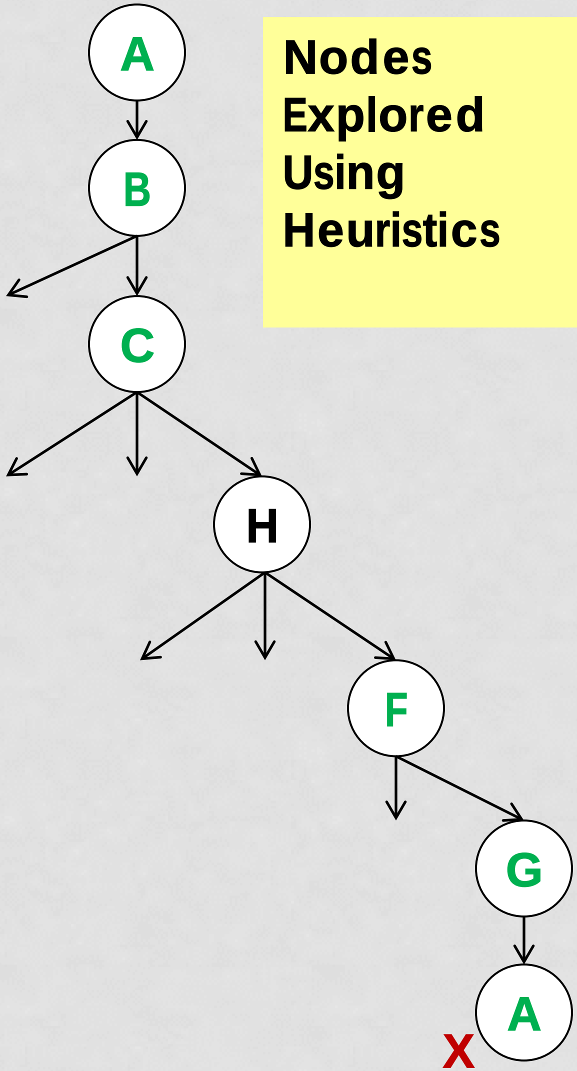
SEARCH COMPARISON



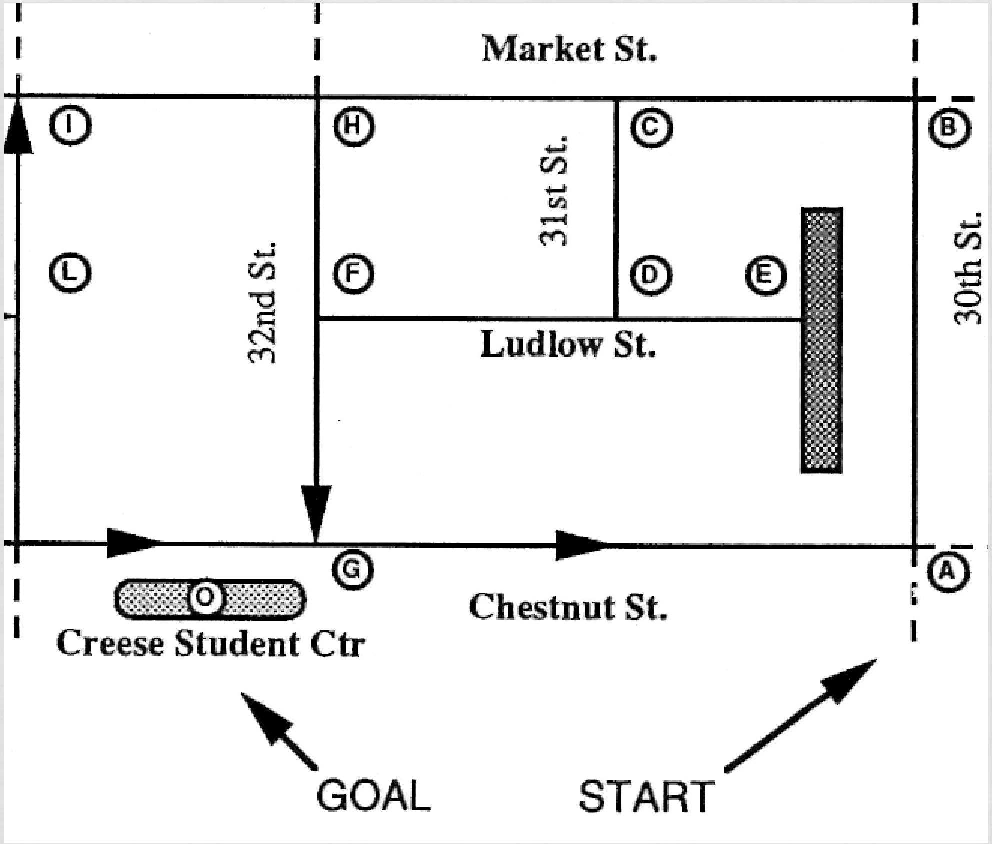
Nodes Explored with Uninformed Search



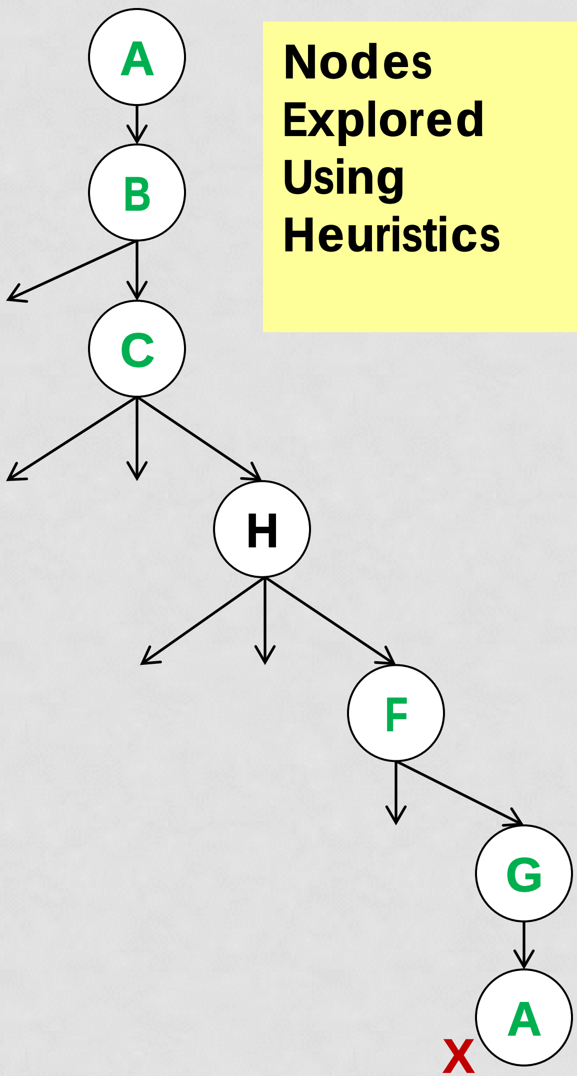
SEARCH COMPARISON



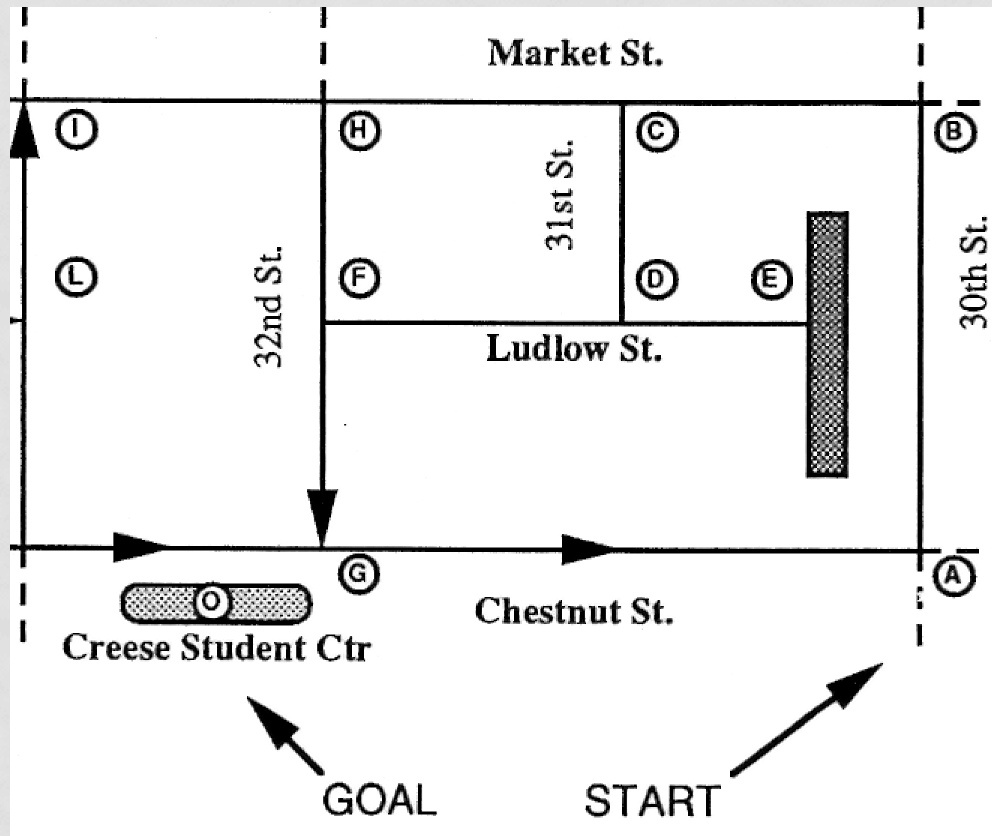
Nodes Explored Using Heuristics



SEARCH COMPARISON



Nodes Explored Using Heuristics



<u>Note</u> : More often than not, heuristic gives good choices. From H, you would try F first, even though we eventually discover it was a bad choice.

MORE EXAMPLES

- “Losing Your Marbles” problem:

Each of three baskets contains a certain number of marbles. You may move from one basket into another basket as many marbles as are already there, thus doubling the quantity in the basket that received the marbles.

You must find a sequence of moves that will yield the same number of marbles in the three baskets (or decide that no such sequence exists).

- What would be a good heuristic here?
- Given a state ***s=(a,b,c)***, and two applicable rules, e.g, $r_1=(-c,0,c)$, $r_2= (0,b,-b)$, which is better?

MORE EXAMPLES

- Possible heuristics:
- ***IDEA***: show preference to rules where the resulting numbers of marbles are close to each other.
- ***Heuristic***:
  Let ***a+b+c = 3n***, and ***s' = ApplyRule(r,s) = (a',b',c')***.
  h(r,s) = |a'-n| + |b'-n| + |c'-n|
- ***Rationale***:
  h(r,s) = 0 at goal, ***h(r,s)>0*** elsewhere.
- Exercise: Starting at (5,6,7), determine moves and the order they will be evaluated. Do this tor two moves
- Discussion: which is better, (5,6,7) or (3,6,9) ?

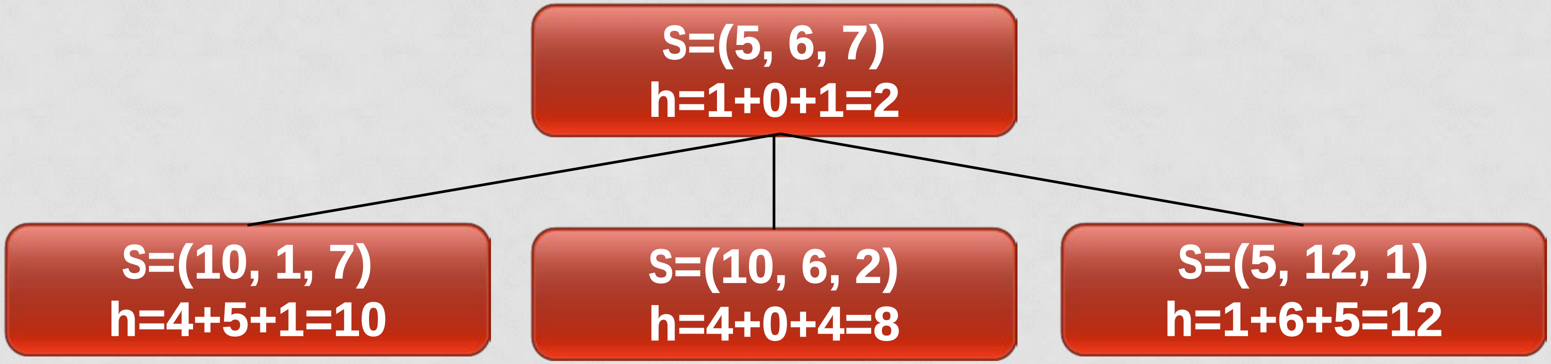
MORE EXAMPLES

Exercise: Starting at (5,6,7), determine moves and the order they will be evaluated. Do this tor two moves

S=(5, 6, 7)
h=1+0+1=2

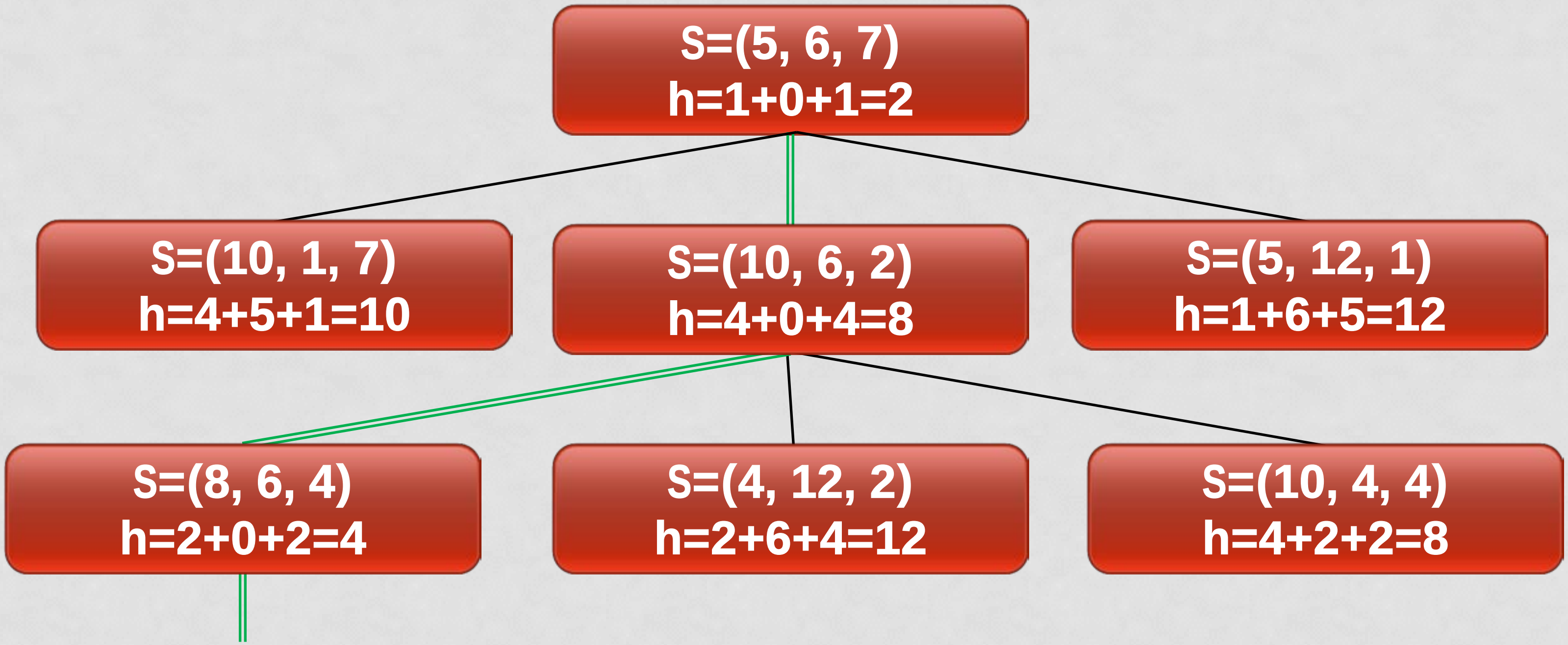
MORE EXAMPLES

Exercise: Starting at (5,6,7), determine moves and the order they will be evaluated. Do this tor two moves



MORE EXAMPLES

Exercise: Starting at (5,6,7), determine moves and the order they will be evaluated. Do this tor two moves



NOW WHAT?

Is this heuristic any good?

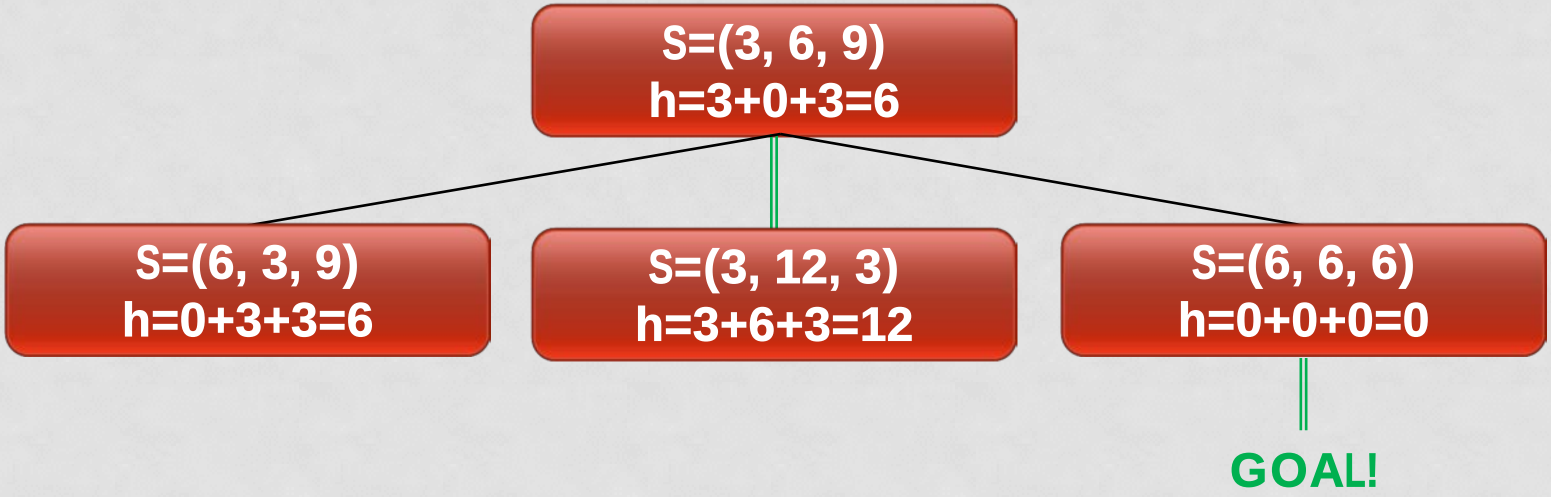
MORE EXAMPLES

Discussion: which is better, (5,6,7) or (3,6,9) ?

S=(3, 6, 9)
h=3+0+3=6

MORE EXAMPLES

Discussion: which is better, (5,6,7) or (3,6,9) ?



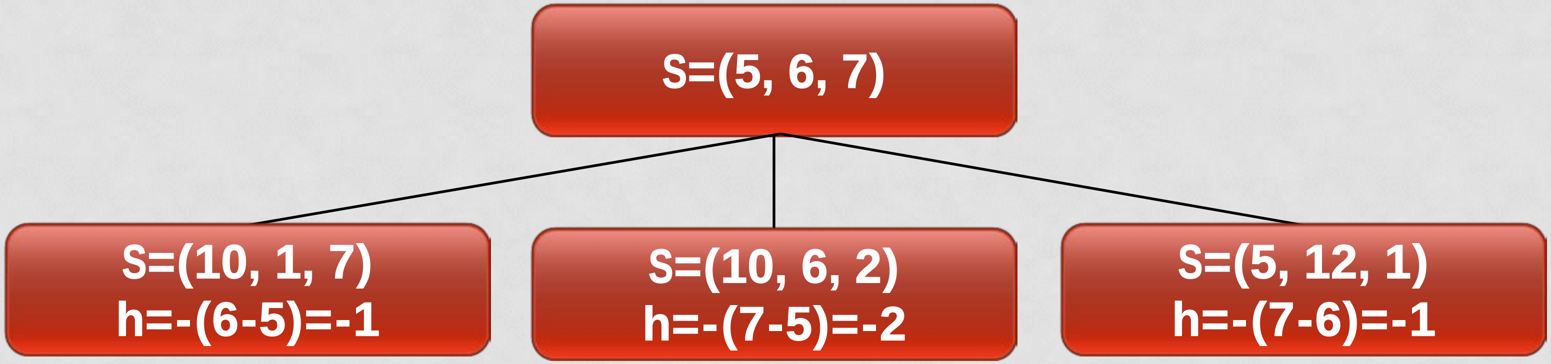
Yet this heuristic preferred (5,6,7)/h=2 to (3,6,9)/h=6.
And all offspring of (5,6,7) were less preferable than (5,6,7).

MORE EXAMPLES

- Another possible heuristic:
- ***IDEA***: show preference to rule where marbles are removed from the bucket with the largest amount, and added to the bucket with the smallest amount.
- ***Heuristic***:
  If $\boldsymbol{s=(a_1,a_2,a_3)}$, and $\boldsymbol{r_{ij}}$ means "move the amount a_j from bucket *i* and place it in bucket *j*", use $\boldsymbol{h(r,s) = -(a_j - a_i)}$.
  e.g., s=(6,9,3): $h(r_{13},s)= -3$, $h(r_{21},s)= -3$, $h(r_{23},s)= -6$.
- ***Rationale***:
  Largest and smallest amounts need to move toward the middle. ***h*** is the difference in the amounts in the buckets, and will be lowest when the difference is highest.

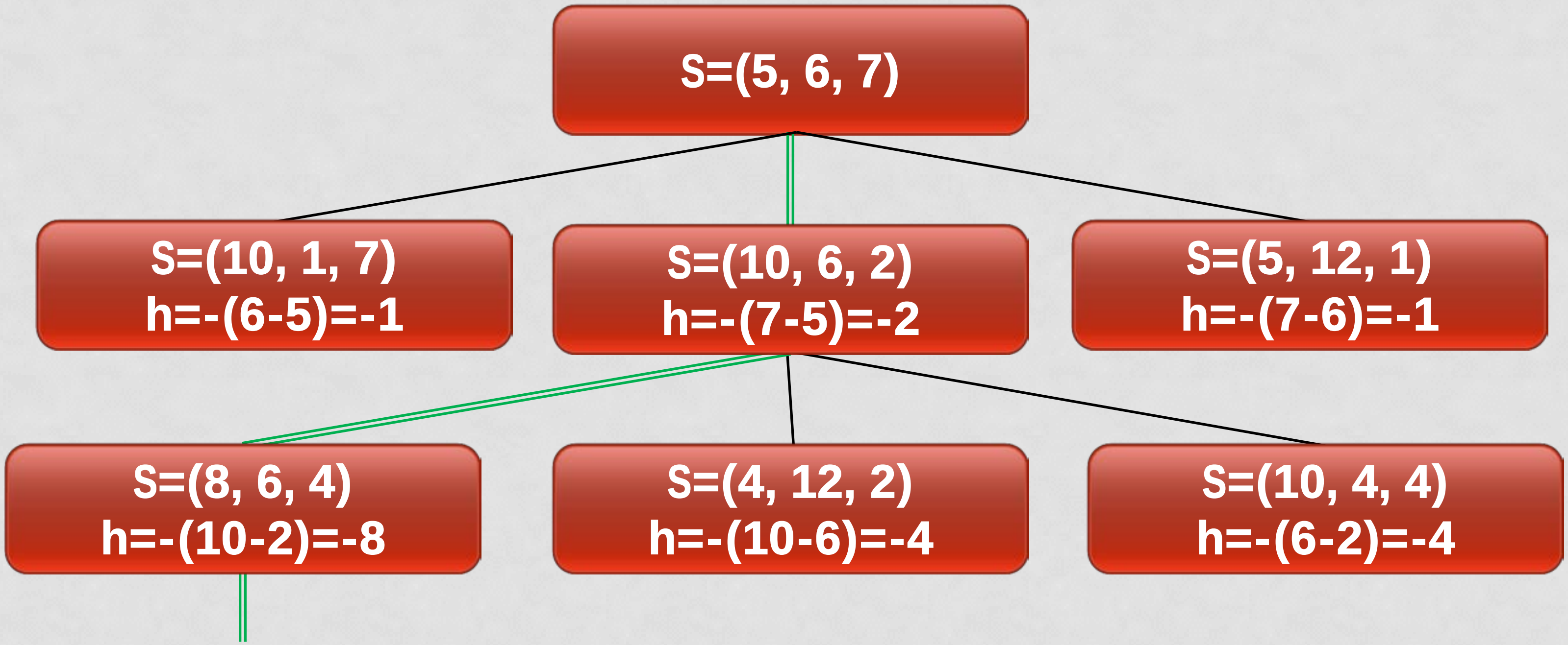
MORE EXAMPLES

Exercise: Starting at (5,6,7), determine moves and the order they will be evaluated. Do this tor two moves



MORE EXAMPLES

Exercise: Starting at (5,6,7), determine moves and the order they will be evaluated. Do this tor two moves



NOW WHAT?

Is this heuristic any better?

```
backTrack ( stateList )
=======================
  state = first element of stateList
  if state is a member of the rest of stateList, return 'FAILED-1
  if deadEnd?(state) return 'FAILED-2
  if goal(state), return NULL
  if length(stateList) > depthBound, return 'FAILED-3

  ruleSet = applicableRules(state)
  if ruleSet == NULL, return 'FAILED-4

  for each rule r in ruleSet,
    newState = applyRule(r,state)
    newStateList = addToFront(newState,stateList)
    path = backTrack(newStateList)
    if path ≠ 'FAILED return append(path,r)

  return 'FAILED-5
```

```
backTrack ( stateList )
=======================
  state = first element of stateList
  if state is a member of the rest of stateList, return 'FAILED-1
  if deadEnd?(state) return 'FAILED-2
  if goal(state), return NULL
  if length(stateList) > depthBound, return 'FAILED-3

  ruleSet = applicableRules(state)
  if ruleSet == NULL, return 'FAILED-4

  for each rule r in ruleSet,
    newState = applyRule(r,state)
    newStateList = addToFront(newState,stateList)
    path = backTrack(newStateList)
    if path ≠ 'FAILED return append(path,r)

  return 'FAILED-5
```

Here is where the heuristic is applied. Put these rules in a presumed "best to worst" order.

OTHER EFFICIENCIES

Implicit Enumeration

- We can save effort if we can eliminate rules from consideration without checking them explicitly.

A farmer is taking a fox, goose, and bag of corn to market. He must cross a river and the boat is only large enough to transport himself and one item at a time. If left unattended, the fox will eat the goose and/or goose will eat the corn

Plan a way for the farmer to get them across

- There are 8 possible rules:
  (←), (← Fox), (← Goose),(← Corn),
  (→), (→ Fox), (→ Goose),(→ Corn)
  where → means "Farmer & Boat go left to right, etc.

OTHER EFFICIENCIES

Implicit Enumeration

- In this implementation of applicableRules(state), we check each rule in allRules explicitly:

```
applicableRules(state)
   result = [ ]
   for r in allRules:
      if preCondition(state,r)
         result ← result + r
   return result
```

- How can we do this more efficiently?
  In this case, *implicit enumeration* means getting the same result without checking each rule explicitly.

OTHER EFFICIENCIES

- Note that although there are 8 possible rules, at most 4 of these can be applicable at any one time, namely the rules that begin with the farmer and boat moving in the correct direction:
  (←), (← Fox), (← Goose),(← Corn)
  or
  (→), (→ Fox), (→ Goose),(→ Corn)
- By simply checking the direction, half the rules can be discarded, without ever considering them (that is, the loop can iterate over a list of 4, not 8).
- Can you think of other ways to encode this?

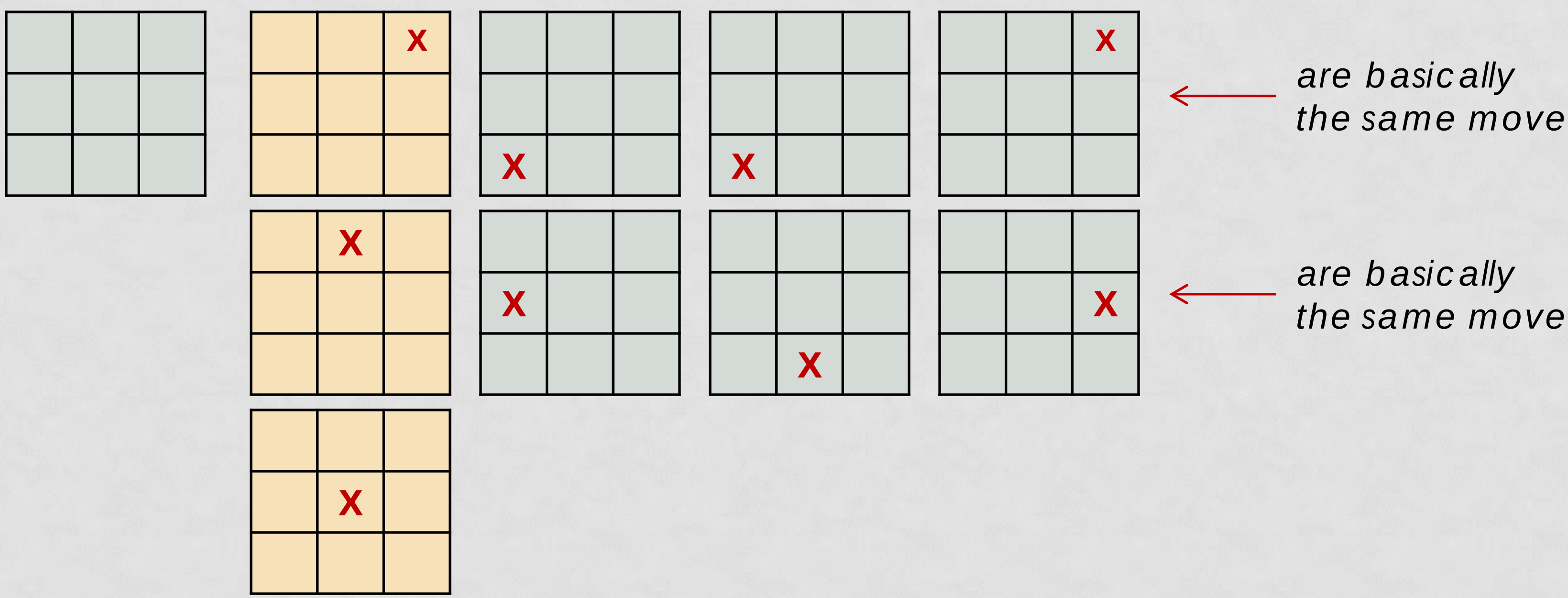
OTHER IDEAS: SYMMETRY

**Sometimes a problem has *inherent symmetry* –
so that some states are similar enough to another state that there is no need trying both:**

OTHER IDEAS: SYMMETRY

**Sometimes a problem has *inherent symmetry* –
so that some states are similar enough to another state
that there is no need trying both:**

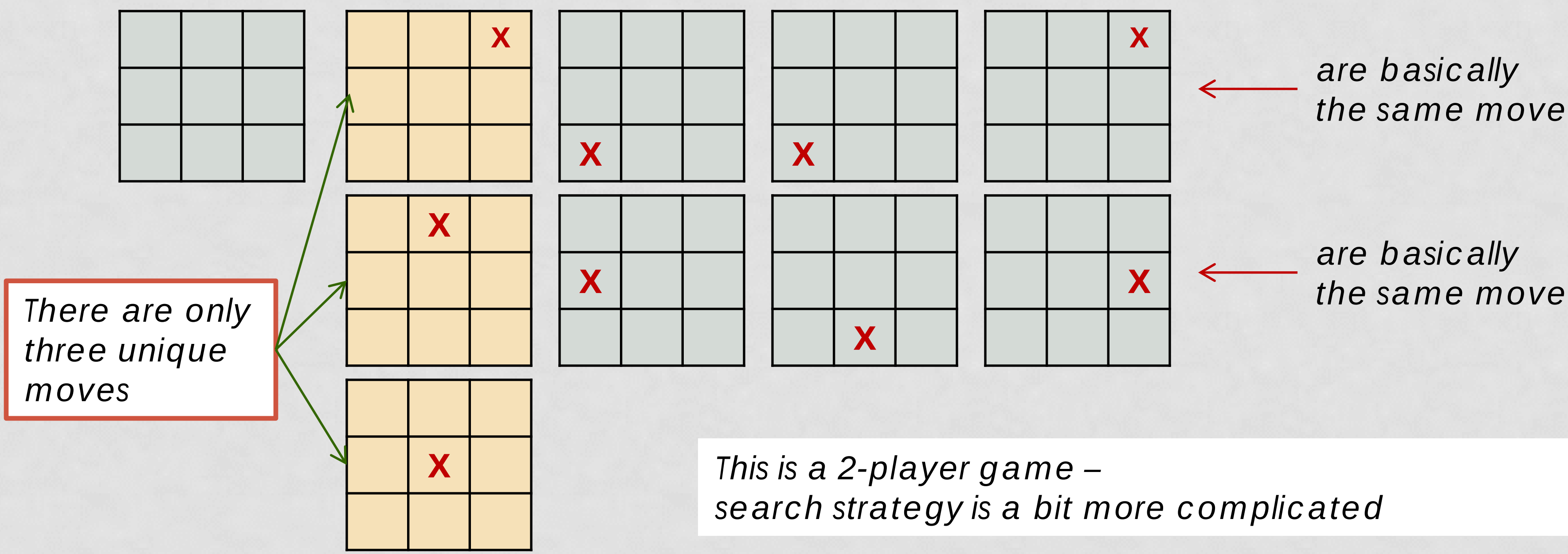
Tic-Tac-Toe **1ST move:9 possibilities (can put an X in any of the 9 squares)**



OTHER IDEAS: SYMMETRY

**Sometimes a problem has *inherent symmetry* –
so that some states are similar enough to another state
that there is no need trying both:**

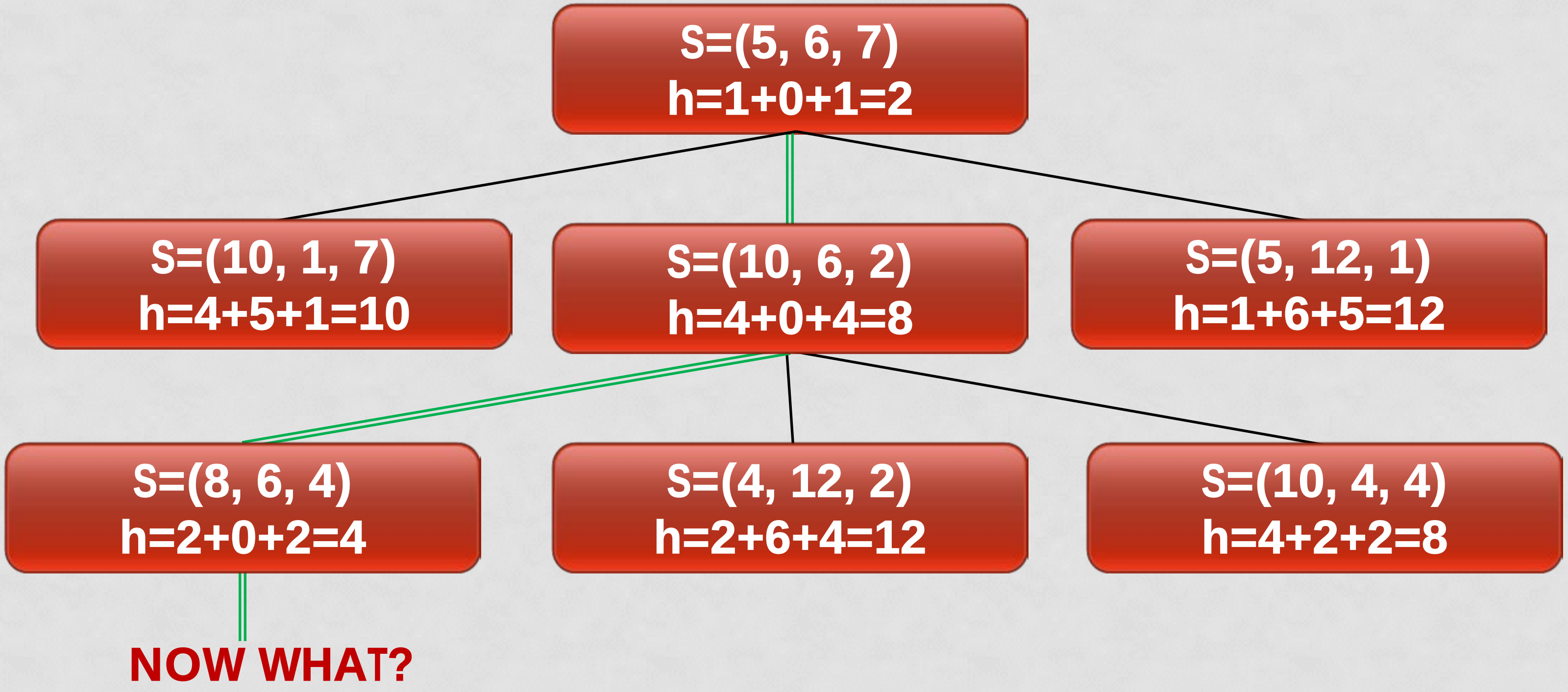
Tic-Tac-Toe **1ST move:9 possibilities (can put an X in any of the 9 squares)**



*This is a 2-player game –
search strategy is a bit more complicated*

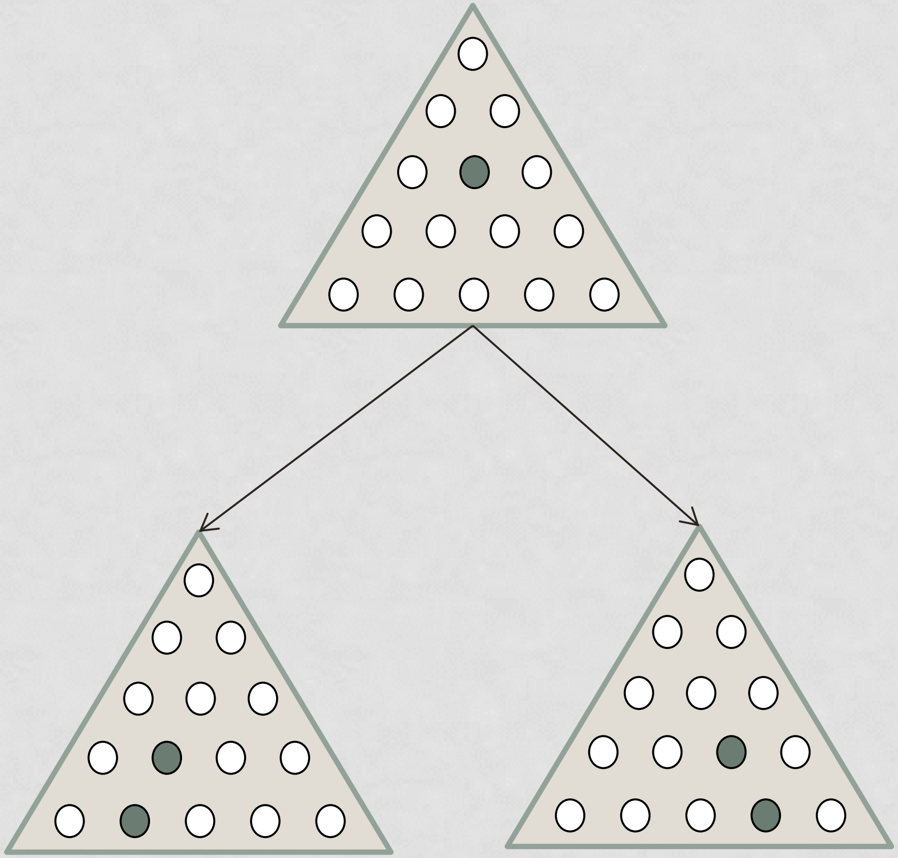
MARBLES, REVISITED

Symmetry: Children of (8,6,4) are (4,6,8), (2,12,4) and (8,2,8). Note that (4,6,8) is symmetric to (8,6,4) [its parent!], and (2,12,4) is symmetric to (4,12,2).



PEGBOARD PROBLEM

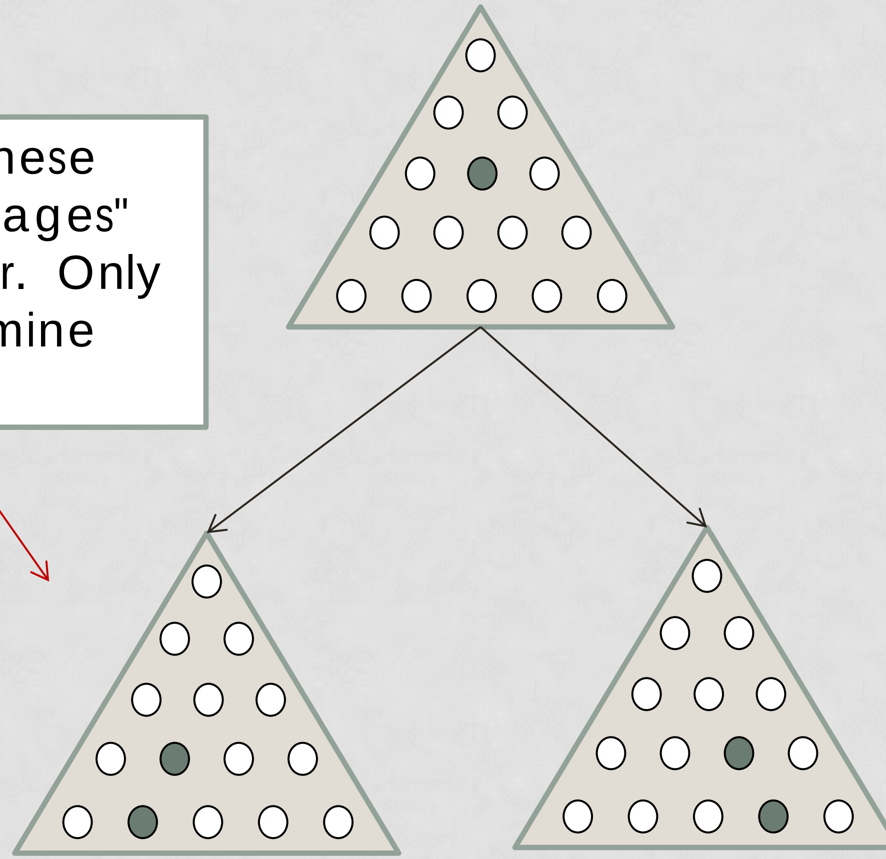
n holes
n-1 tees (pegs)
Using checkers-style jumps, find a sequence of jumps that results in ***1*** peg remaining.
(extra credit if that peg is in the original empty spot)



http://3.bp.blogspot.com/-nyB4nyBCMfY/UzHF_YQ6J5I/AAAAAAAAAOw/qEJzMmOPqN8/s1600/PegBoard.jpg

PEGBOARD PROBLEM

Notice that these are "mirror images" of each other. Only need to examine one of them.



n holes
n-1 tees (pegs)
Using checkers-style jumps, find a sequence of jumps that results in ***1*** peg remaining.
(extra credit if that peg is in the original empty spot)

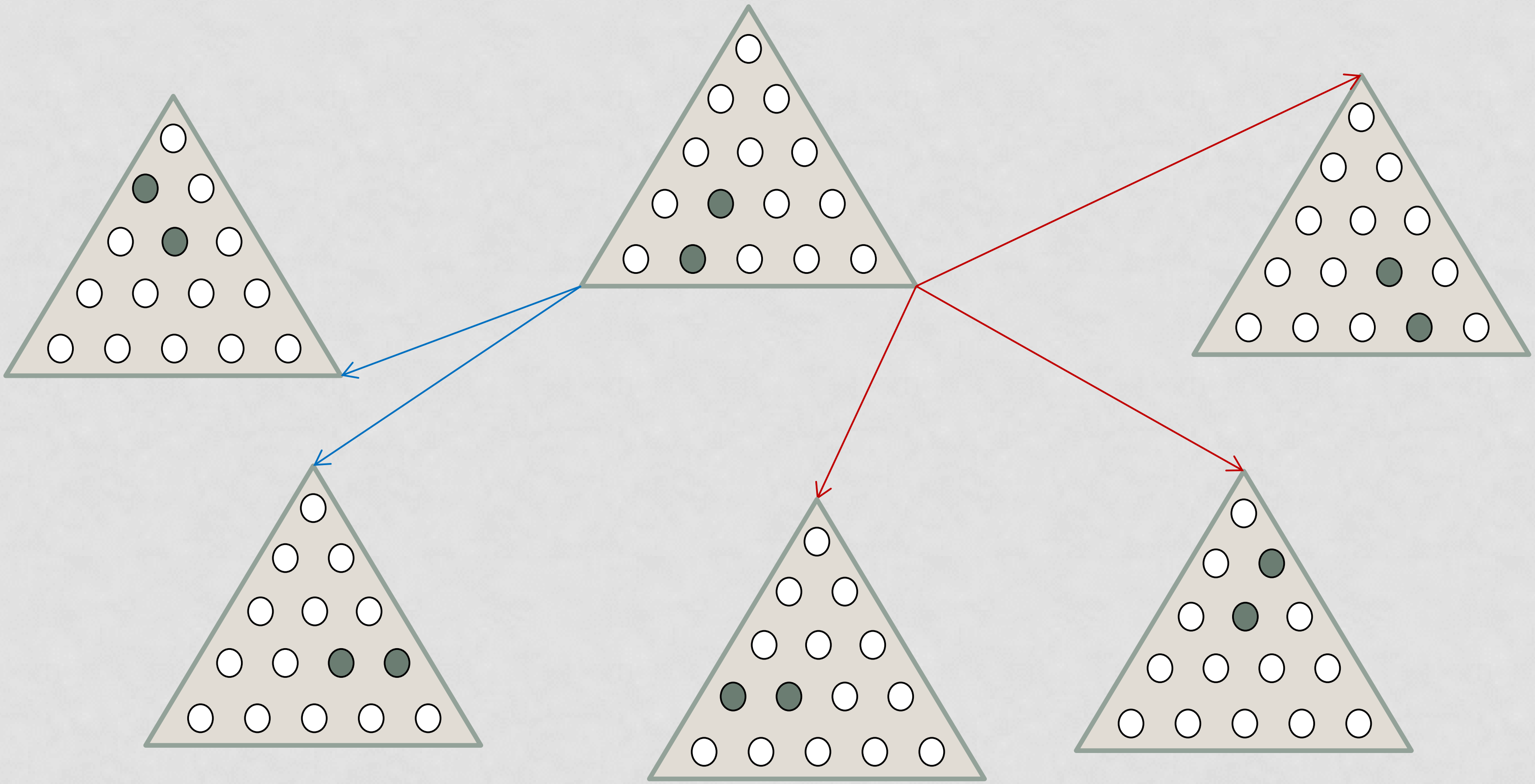
http://3.bp.blogspot.com/-nyB4nyBCMfY/UzHF_YQ6J5I/AAAAAAAAAOw/qEJzMmOPqN8/s1600/PegBoard.jpg

PEGBOARD PROBLEM

```
for each rule r in ruleSet,
    newState = applyRule(r,state)
    newStateList = addToFront(newState,stateList)
    path = backTrack(newStateList)
    if path != 'FAILED
        return append(path,r)
```

If we have already seen a symmetric variant of **newState**,
no need to add it to the list

PEGBOARD PROBLEM REVISITED

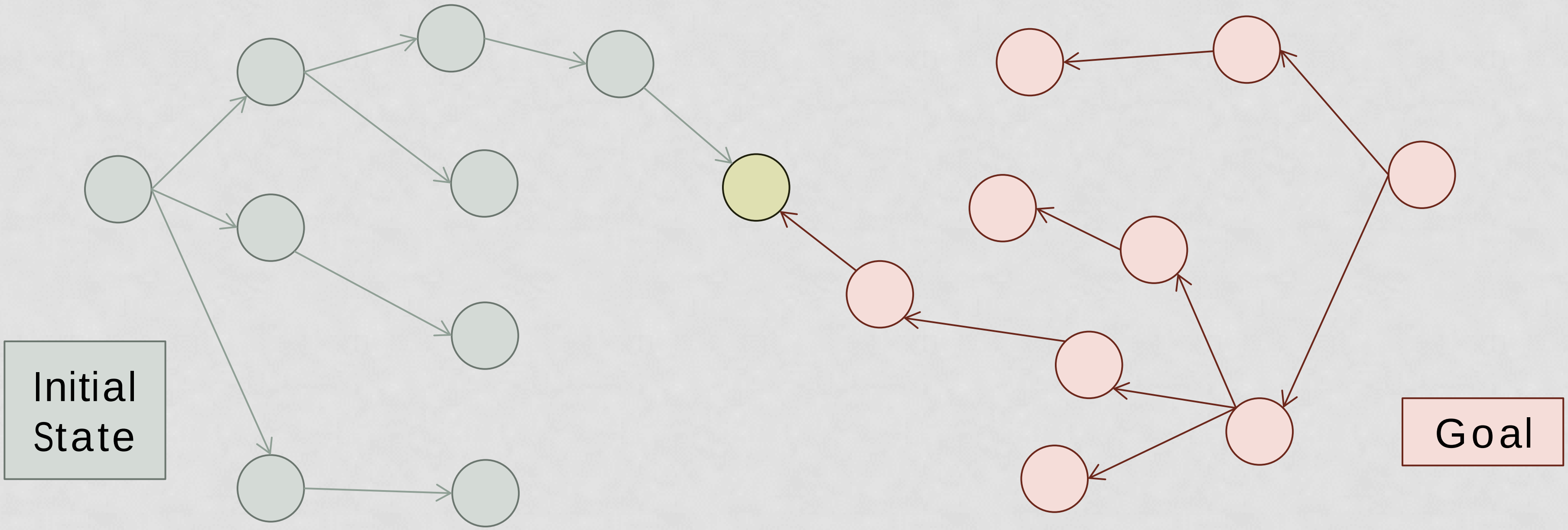


Rotational Symmetry:
rotating left or right

Reflectional Symmetry:
flipping on an axis

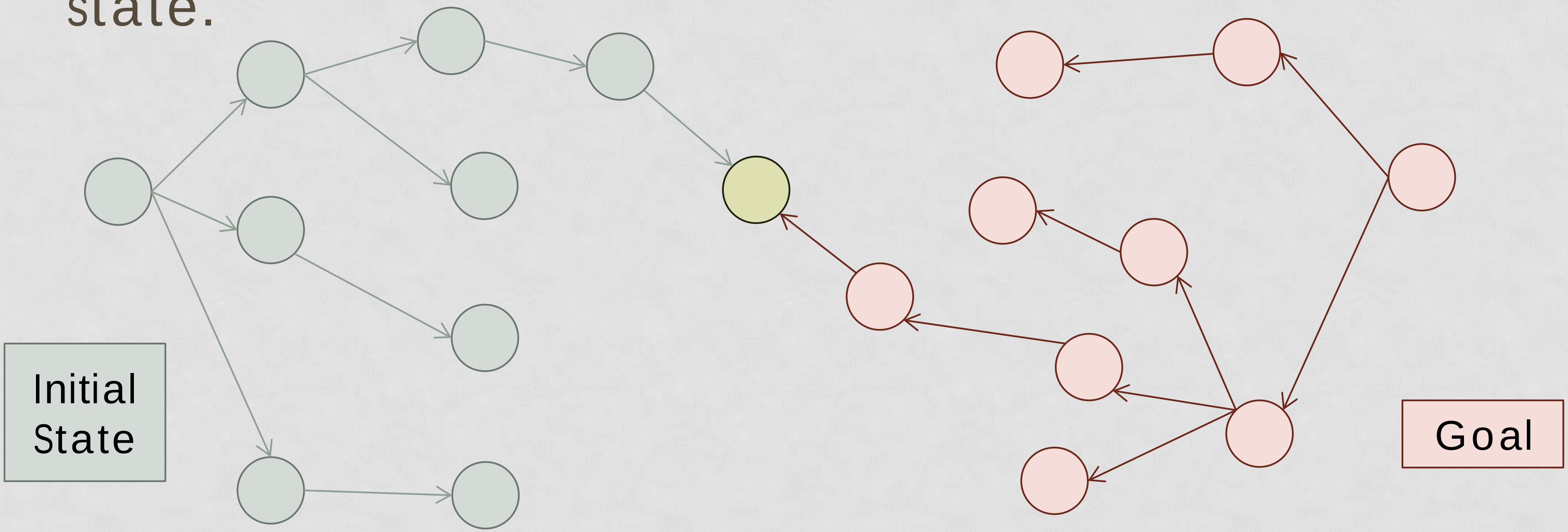
FORWARD-BACKWARD SEARCH

Alternate between applying moves forward through the tree from the Initial State and applying moves backward through the tree from a Goal State:

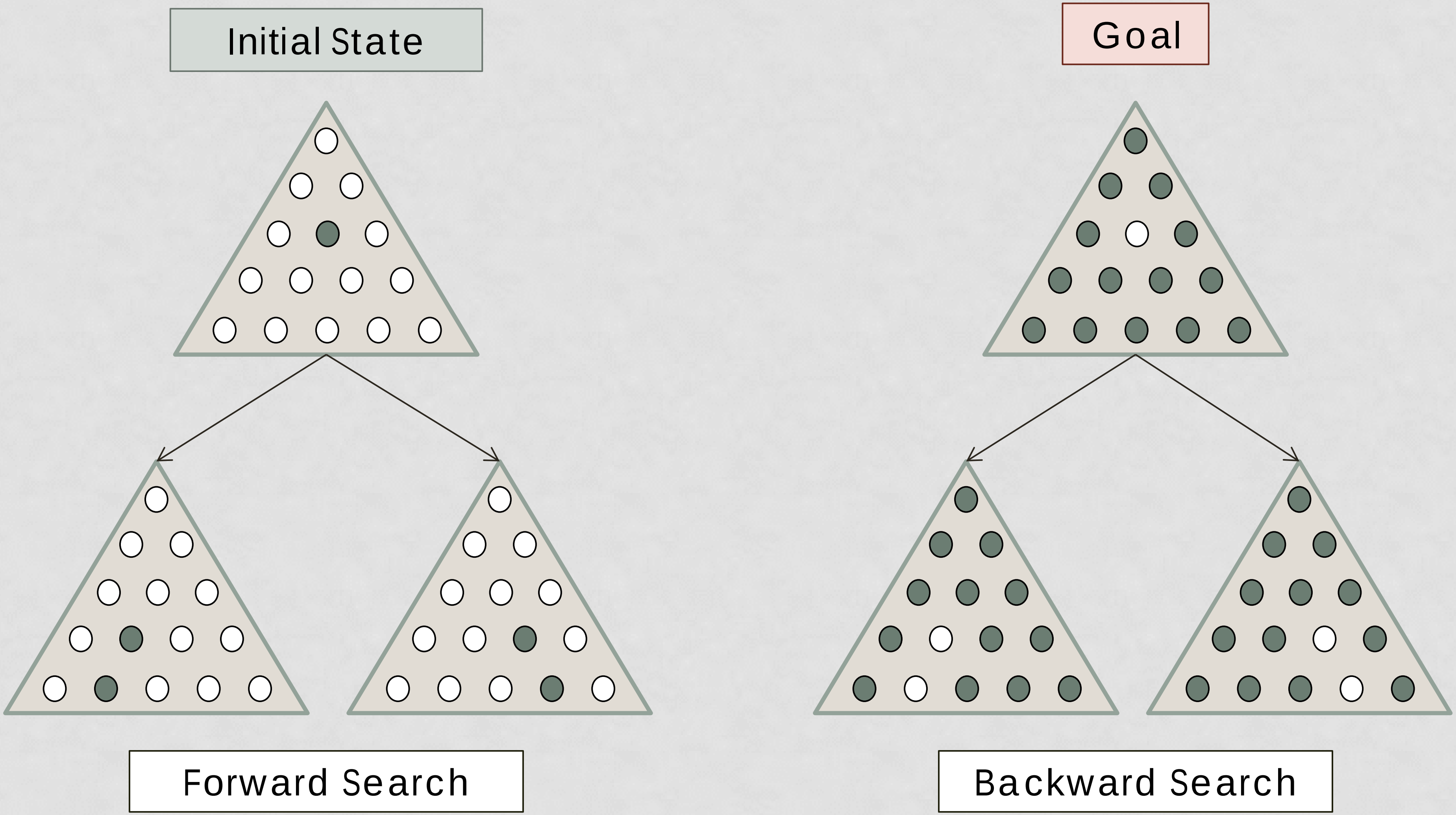


FORWARD-BACKWARD SEARCH

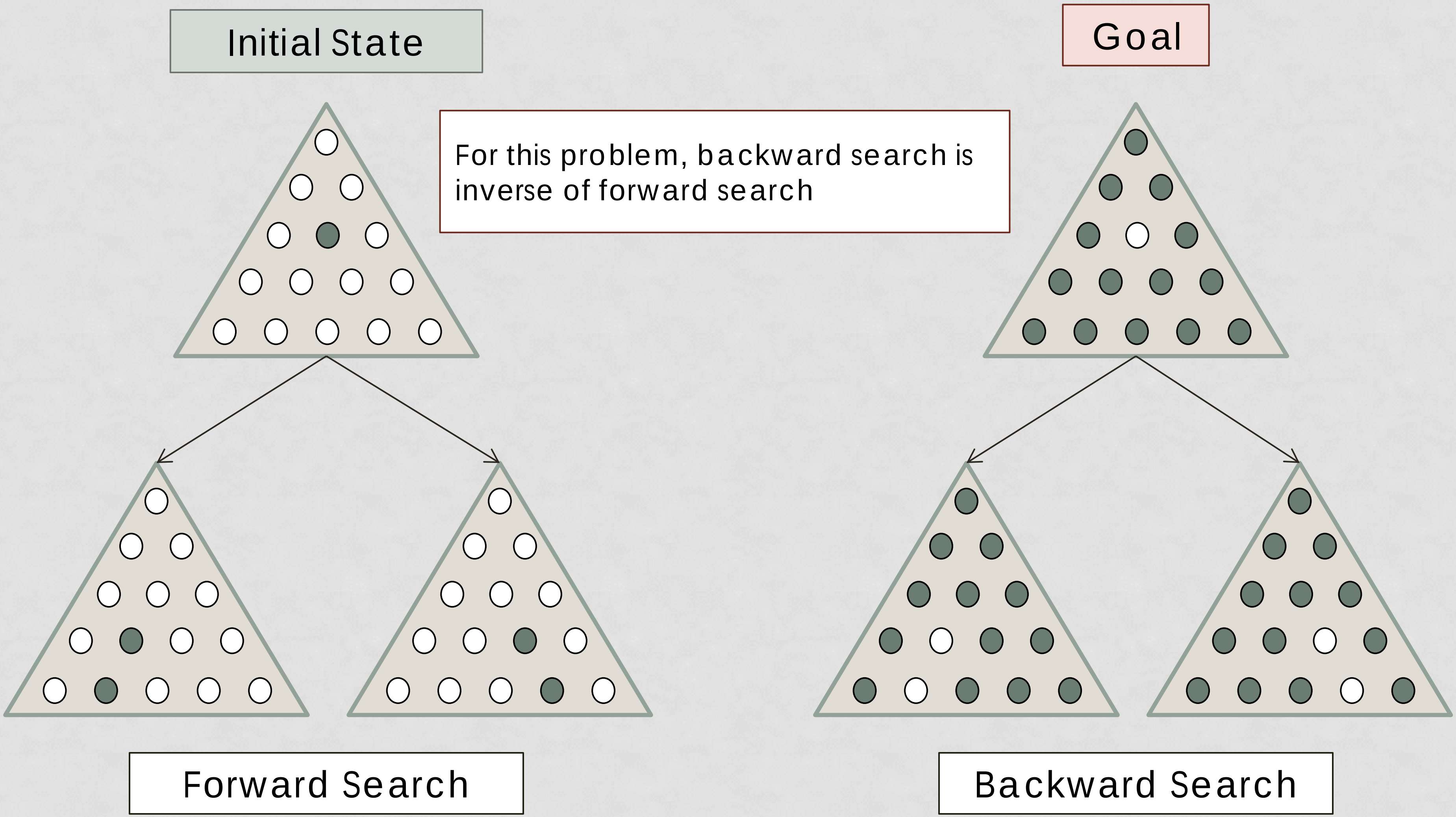
For Forward-Backward Search, the “**state**” is a pair of search trees, one beginning at “Initial State” and another beginning at “Goal”. The **goal()** function returns true if both search trees produce a common state.



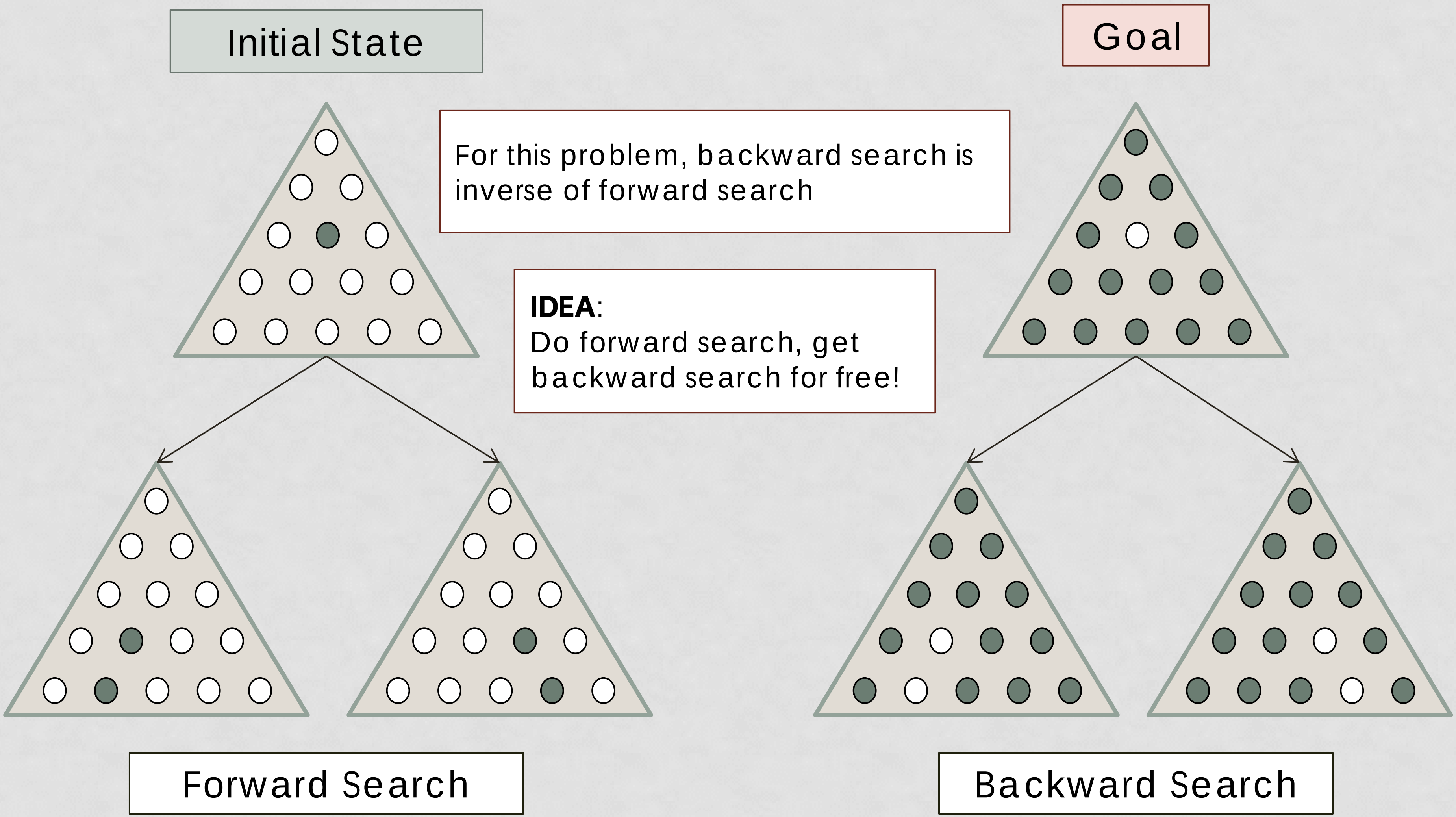
FORWARD-BACKWARD SEARCH WITH SYMMETRY



FORWARD-BACKWARD SEARCH WITH SYMMETRY



FORWARD-BACKWARD SEARCH WITH SYMMETRY



FORWARD-BACKWARD SEARCH WITH SYMMETRY

For instance, using binary representation:

Initial State: **111101111111111** = $(2^{15}-1) - 2^{10}$

Goal State: **000010000000000** = 2^{10}

Children of Initial State:

A = **111111101110111** = $(2^{15}-1) - 2^{7} - 2^{3}$

B = **111111110111101** = $(2^{15}-1) - 2^{6} - 2^{1}$

A' = **000000010001000** = $2^{7} + 2^{3}$

B' = **000000001000010** = $2^{6} + 2^{1}$

Initial State = $(2^{15}-1)$ – Goal State

A = $(2^{15}-1)$ – A'

B = $(2^{15}-1)$ – B'

While building the forward search tree requires some computation to find and apply applicable rules,etc., …

Building the backward search tree requires only simple bit operations applied to the forward tree.