CS4300 Artificial Intelligence

Tom Henderson

SCHOOL OF COMPUTING

THE UNIVERSITY OF UTAH

What's a Problem?

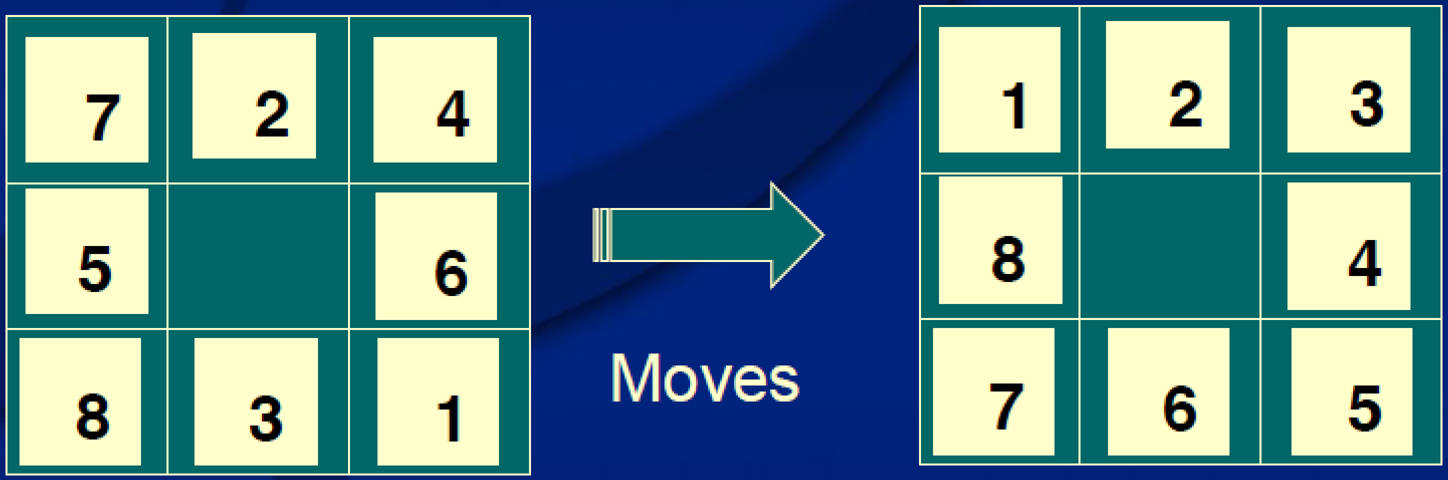
- Initial state
- Actions
- Transition model
- Goal Test
- Path Cost

Does this apply to:

Problem: **Get A in CS5300**

Solution: action sequence from initial to goal state (optimal if path cost is least)

Example: 8-Puzzle



Start State

Goal State

SCHOOL OF COMPUTING

Problem Solving Agent

Persistent state

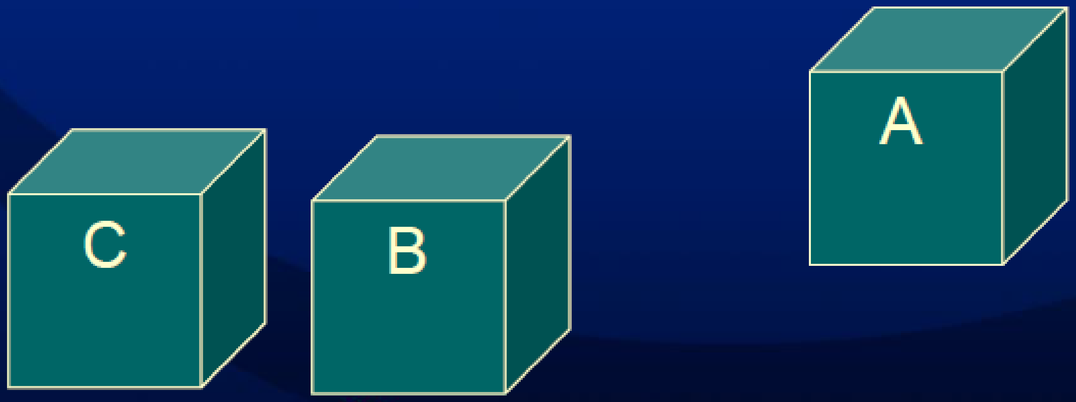
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

seq: sequence of actions

solve search problem

E.g., Stack Blocks Problem

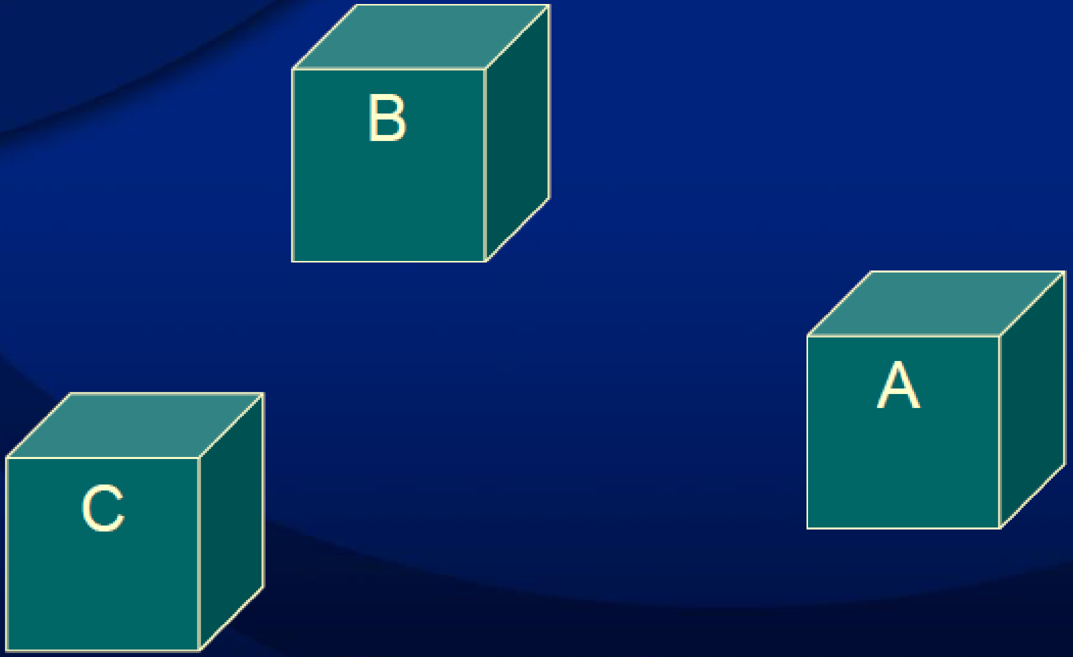
- Given blocks A, B, and C on the table



Stack Blocks Problem

- Figure out a sequence of actions to get goal of: B on A and C on B

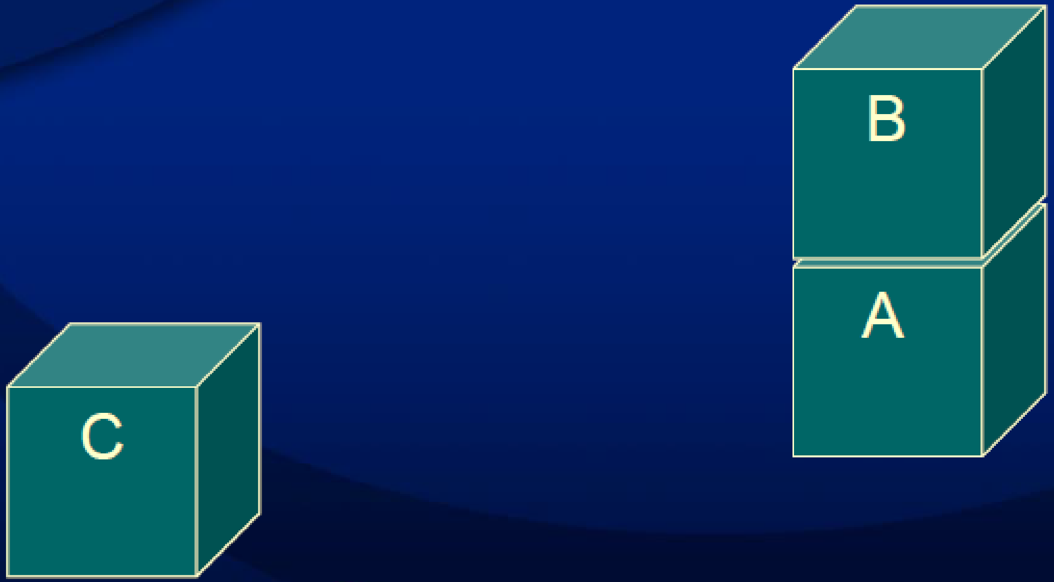
1. Pickup B



Stack Blocks Problem

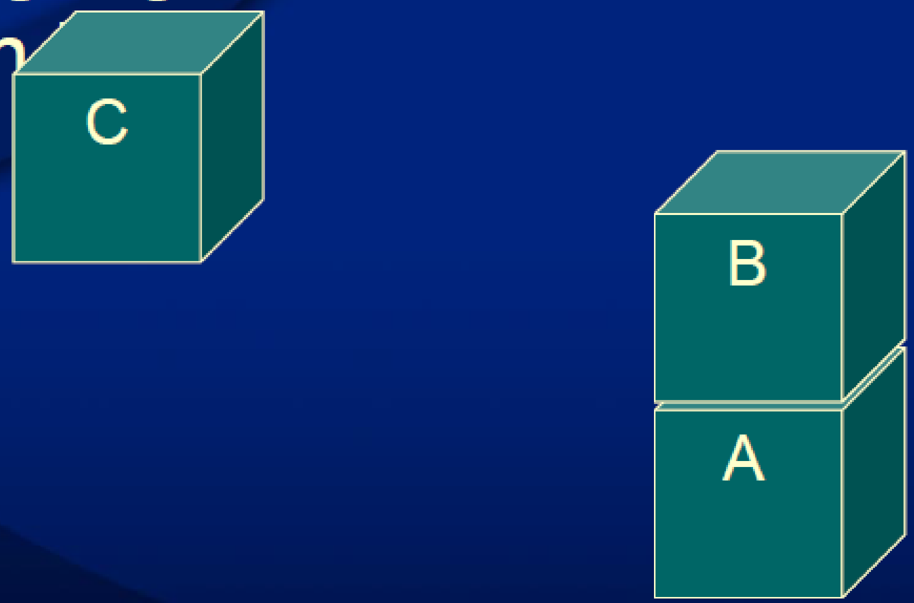
- Figure out a sequence of actions to get goal of: B on A and C on B

1. Pickup B
2. Put B on A



Stack Blocks Problem

- Figure out a sequence of actions to get goal of: B on A and C on



1. Pickup B
2. Put B on A
3. Pickup C

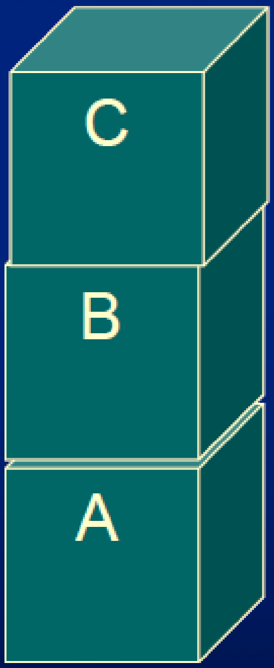
SCHOOL OF COMPUTING

Stack Blocks Problem

- Figure out a sequence of actions to get goal of: B on A and C on B

1. Pickup B
2. Put B on A
3. Pickup C
4. Put C on B

seq



THE UNIVERSITY OF UTAH

SCHOOL OF COMPUTING

Problem Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
   persistent: seq, an action sequence, initially empty
               state, some description of the current world state
               goal, a goal, initially null
               problem, a problem formulation

   state ← UPDATE-STATE(state, percept)
   if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
   action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

Note:

- **assumes actions work!**
- **seq vs percept & state?**

First time in, finds seq

Subsequent times in, returns first in seq

THE UNIVERSITY OF UTAH

Goal Formulation

Artificial Problems: e.g., tic-tac-toe

?? Is this a problem-solving Agent??

Goal: winning board state (3 in a line)
Rules: legal moves specified by game

Problem Formulation: what actions and states

State: board and whose turn
Action: put an X or an O
Search: get from initial state to goal
Solution: action sequence
Execution: run solution

Representation

- State?
- Action?

Goal Formulation

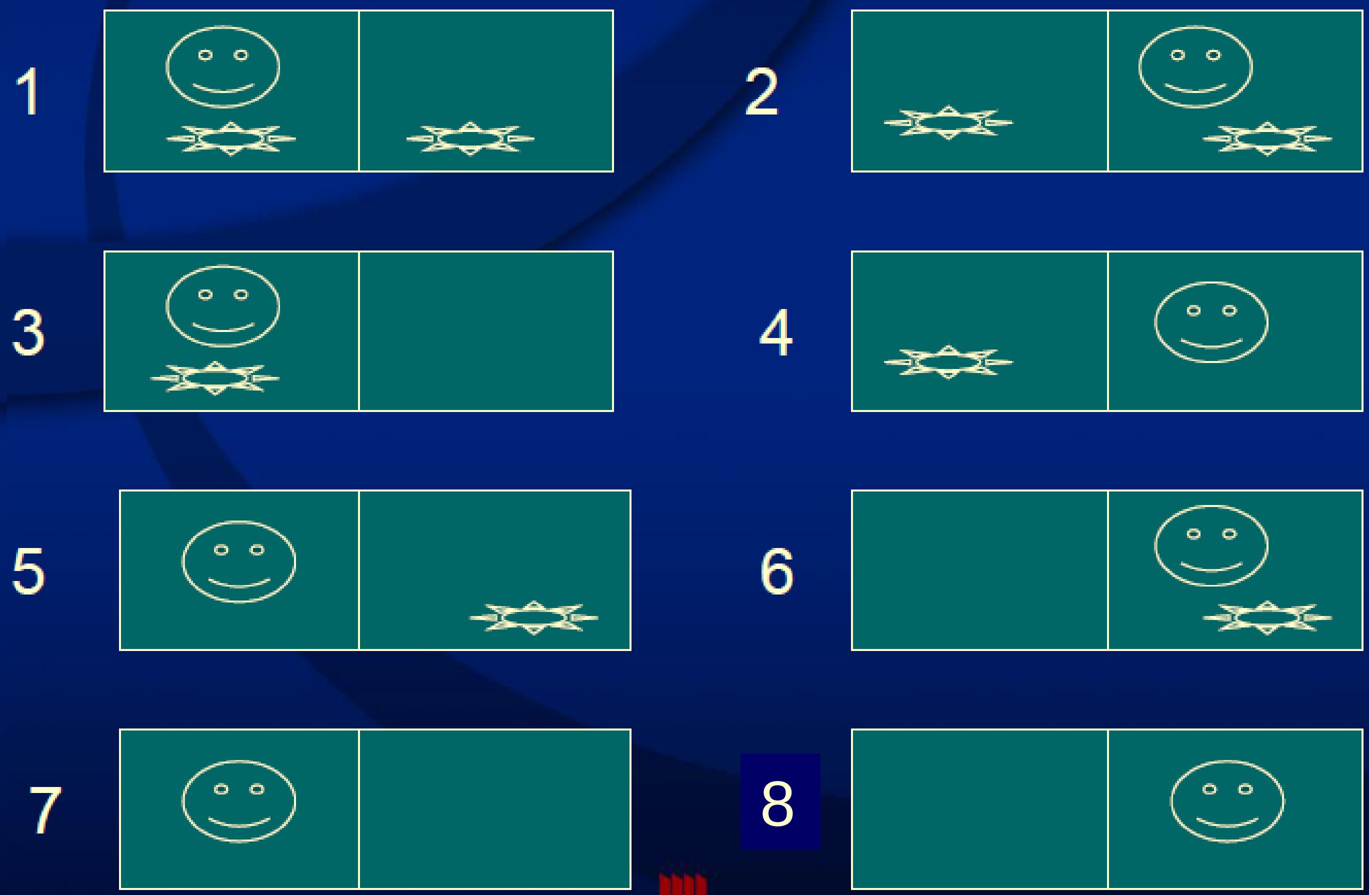
Real World Problems: e.g., clean floor

Goal: no dirt on floor
Rules: physical and social

Problem Formulation: what actions and states

State: enumerable?
Action: move, vacuum (but: knock over table!)
Search: get from initial state to goal
Solution: action sequence
Execution: run solution

Vacuum World States



Representation

State + Actions: crucial issue

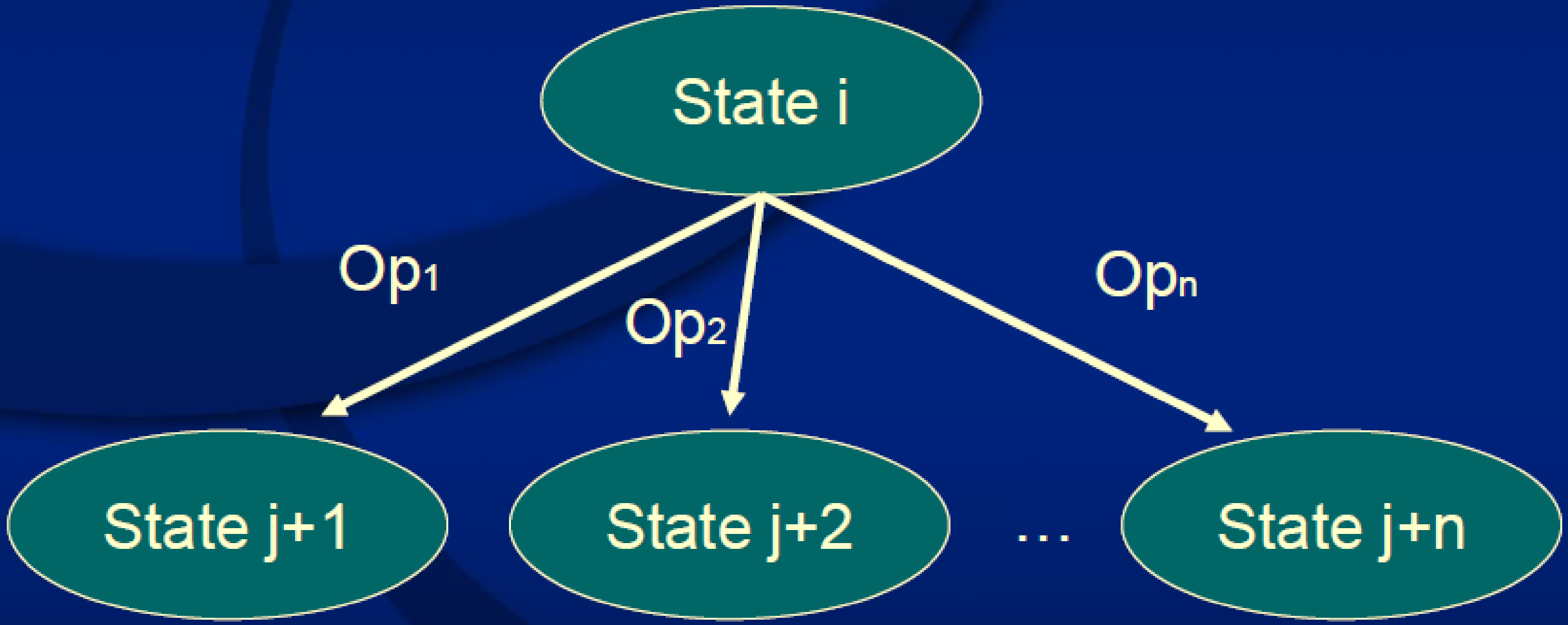
Problem types:

- **Single State**: action is function
- **Multiple states**: several possibilities
- **Contingency**: sensing necessary
- **Exploration**: determine consequences

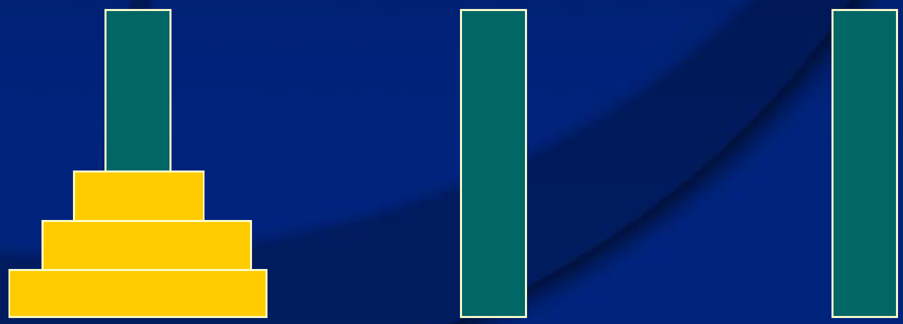
Problem Definition

- Initial state
- Operator (successor function)
- Goal test
- Path cost

State Space



Towers of Hanoi

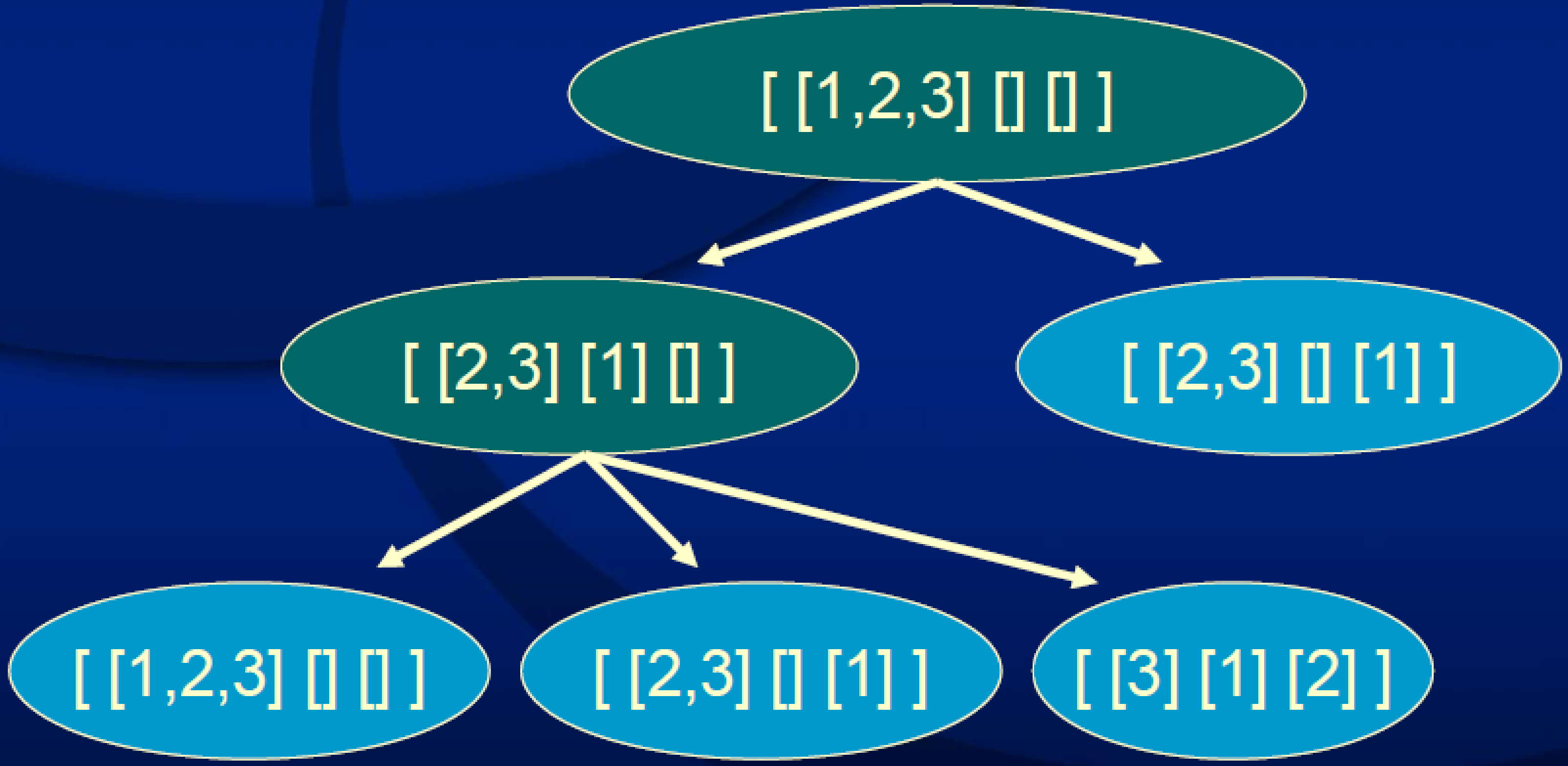


Initial state

Move 1 ring at a time
  * only smaller on top

Goal state

State Space



Initial State

Choice of representation:

- Easy to understand
- Easy to make operations
- Easy to recognize goal
- Easy to calculate cost

Vector of 3 vectors: one for each tower

Operator

Move from one tower to another:

- Move 1 to 2 (Meaning?)
- Move 1 to 3
- Move 2 to 1
- Move 2 to 3
- Move 3 to 1
- Move 3 to 2

Operator

Example:

Move 1 to 2:

[[1 2 3] [ ] [ ]] → [[2 3] [1] [ ]]

SCHOOL OF COMPUTING

Operator

Example:

Move 2 to 3:

[[1 2 3] [ ] [ ]] → [[1 2 3] [] [ ]]

THE UNIVERSITY OF UTAH

Goal Test

Goal state:

[[ ] [ ] [1 2 3]]

Path Cost

Common:

- 1 (for each operation)
- Distance
- Power, etc.

Search Cost

Search cost comprised of:

- Solution found (if no, then infinite)
- Path cost (intrinsic to problem) [online]
- Search cost (time, memory) [offline]

Example Problems

Standard problems (know these!):

- 8-puzzle
- 8 queens
  - Packing
  - Covering
- Cryptarithmetic
- Missionaries and Cannibals

Missionaries & Cannibals

State: [M,C,B] number on wrong side of river

Start
[3 3 1]

Goal
[0 0 0]

1M 1C 2M 2C 1M, 1C

[2 3 0] [3 2 0] [1 3 0] [3 1 0] [2 2 0]

[3 3 1] ← duplicate

Missionaries & Cannibals

State: [M1,C1,B1;M2,C2,B2]

General Search

Search strategy: how to expand nodes

function General-Search(problem,strategy)
  returns solution
Loop do
  if no candidates to expand **then return** fail
  choose leaf for expansion using *strategy*
  if node contains goal state **then** return solution
  else expand node and add to search tree
end

SCHOOL OF COMPUTING

Search Tree Data Structure

Datatype node

components:

- State
- Parent
- Operator
- Depth
- Path_cost

Search Strategies

- Completeness: find a solution?
- Time complexity: how long?
- Space complexity: how much memory
- Optimality: best solution?

Breadth-First Search

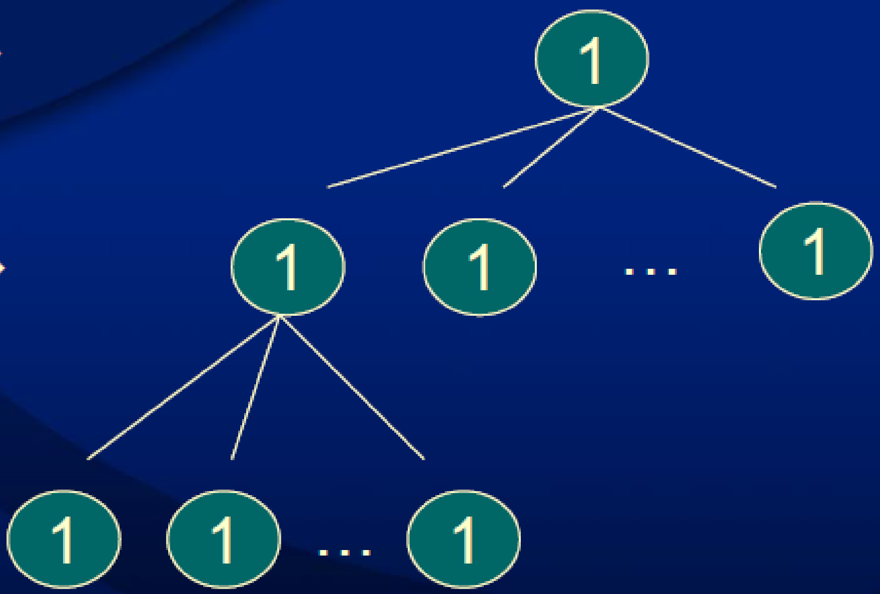
function Breadth-first-search(problem)

  returns result

(1) All this level →

(2) All this level →

(3) All this level →



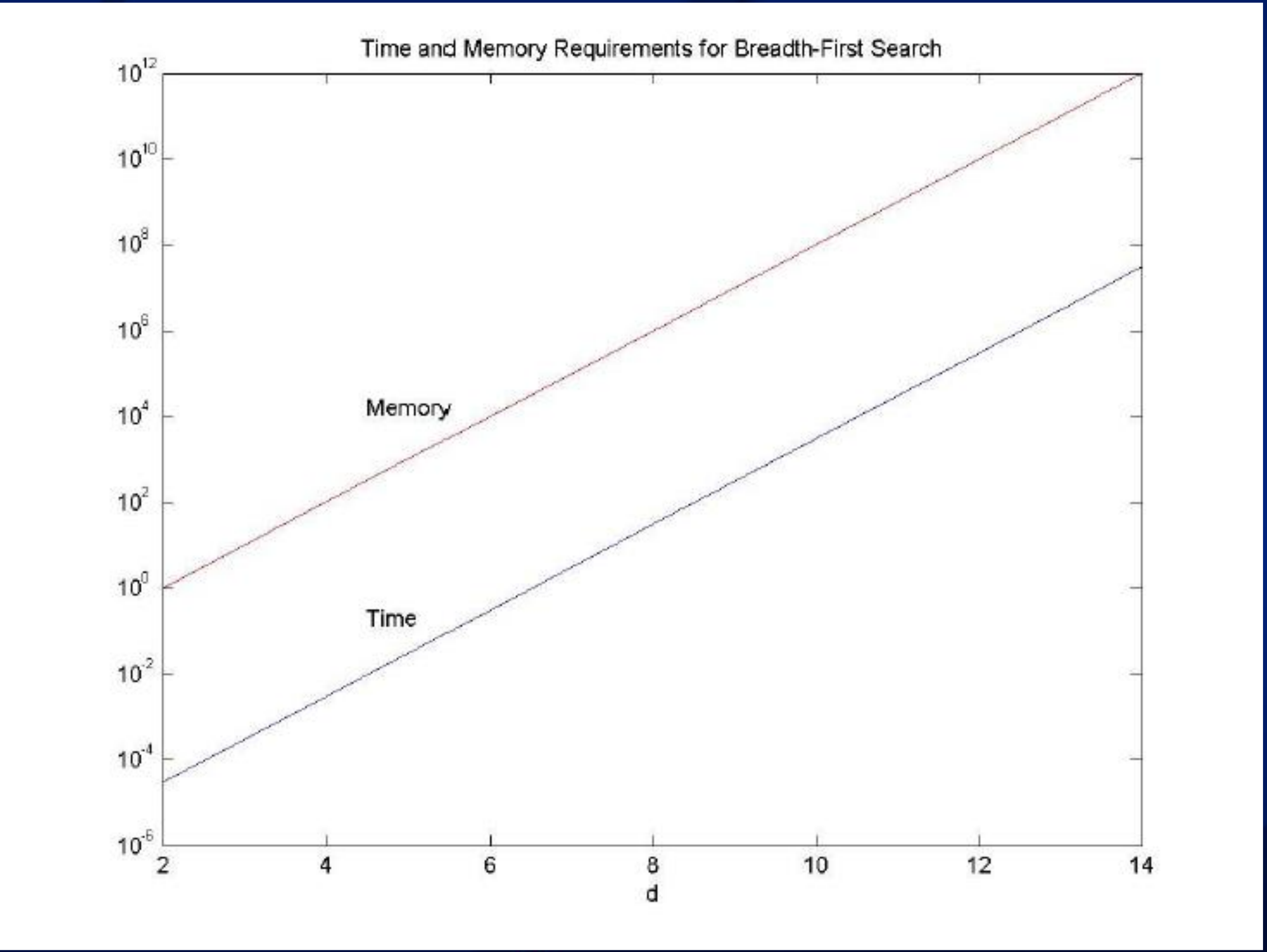
Breadth-First Search

Complexity is high (10 branches):

Depth	Time	Memory
12	13 days	1 Pb
16	350 years	10 Eb

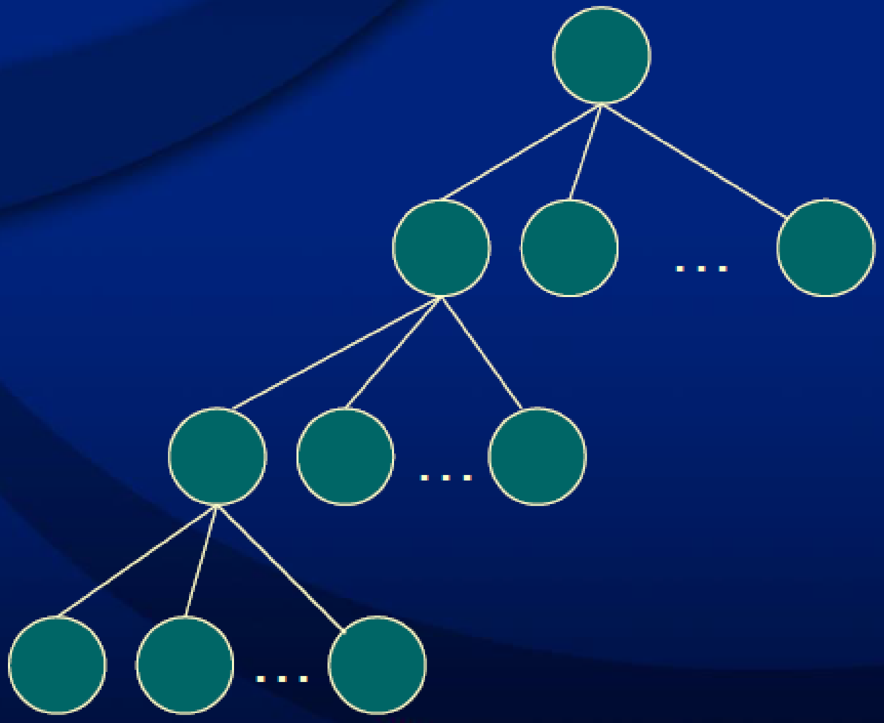
Uniform Cost: expand lowest cost path

Time & Memory: (D,log(D))



Depth-First Search

function Depth-first-search(problem)
returns result



Depth-First Search

- Depth limited: fix deepest level
- Iterative deepening: keep increasing depth limit
- Bi-directional search: go from goal to start, as well as start to goal

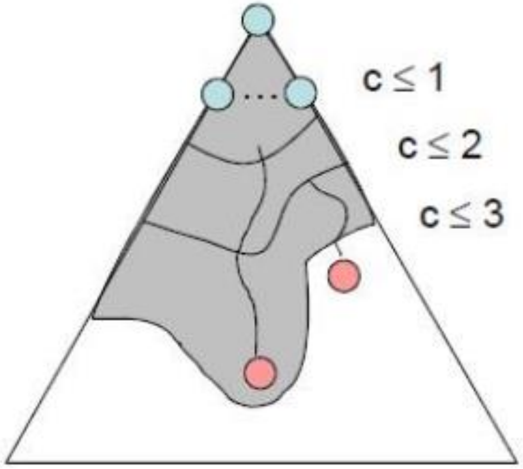
Avoid generating duplicate states!!

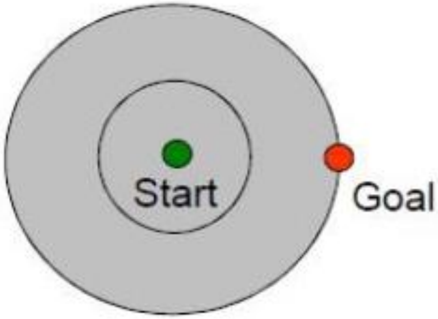
Uniform Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

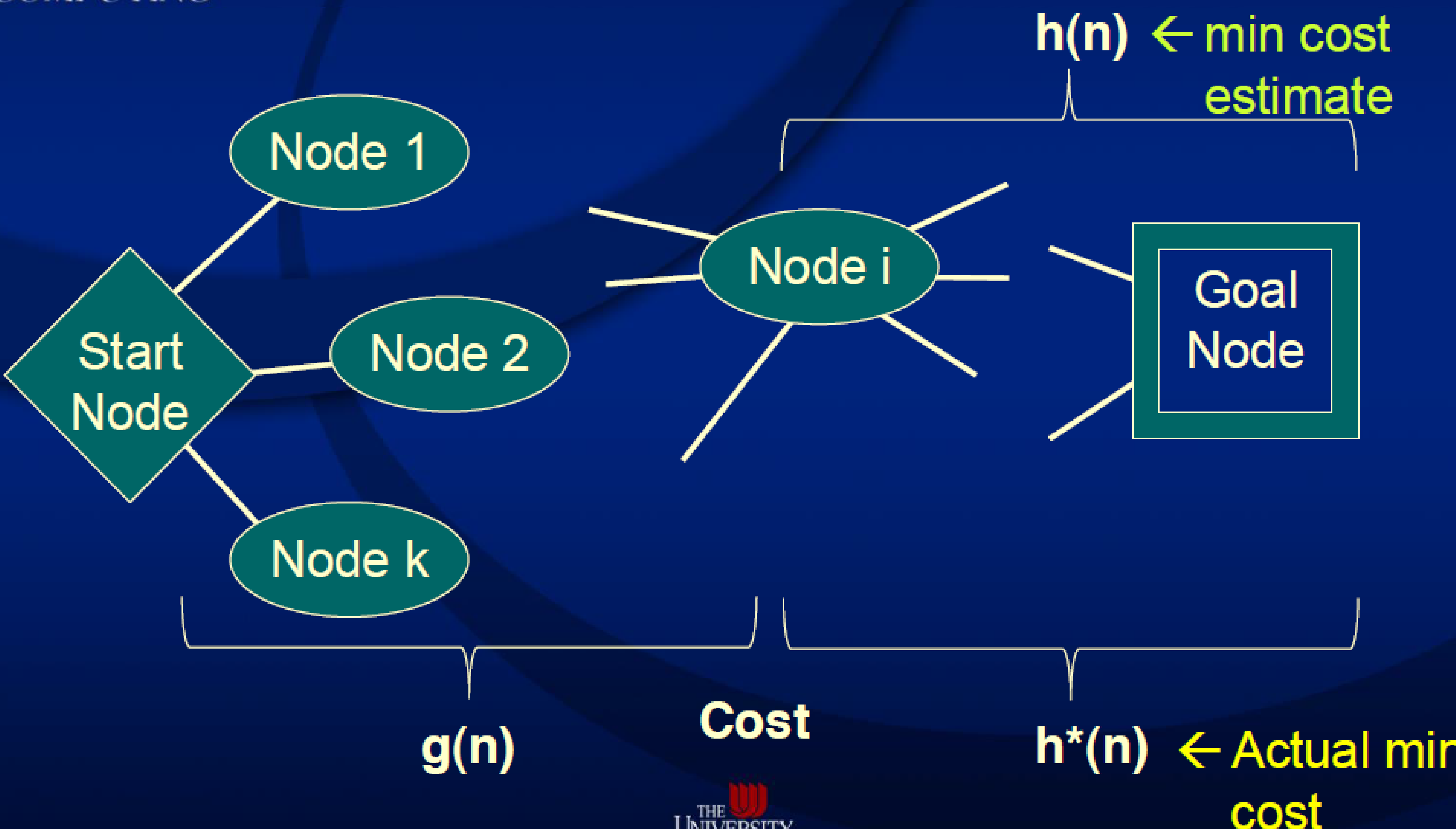
Uniform Cost Search

- Strategy: expand lowest path cost
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location





Going from Start to Goal



Evaluation Function

- Cost from start to goal
- Each search method:
  - Prioritizes nodes for expansion
    - Based on f(n)
- Two parts:
  - Cost from start to node n   g(n)
  - Cost from node n to goal   h(n) estimate

Compare Search Methods

- Uniform: $f(n) = g(n)$
- Greedy: $f(n) = h(n)$
- A*: $f(n) = g(n) + h(n)$
  - Where $h(n) <= h^*(n)$

hanoi

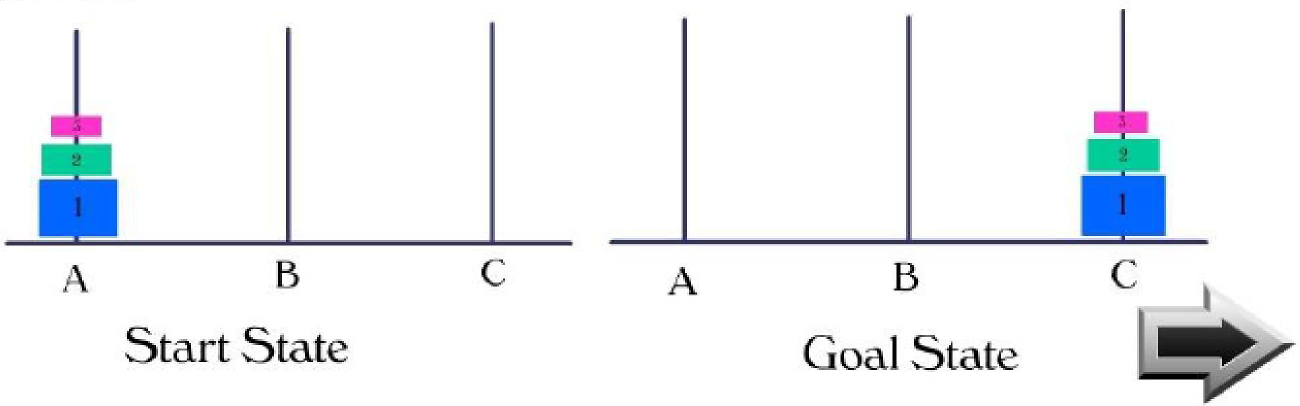
www-g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/hanoi.html

A* heuristic for 3 peg Towers of hanoi

Towers Of Hanoi

The Towers of Hanoi problem will now be used to demonstrate the benefits of a carefully chosen heuristic

The problem starts with all three disks on the left hand peg. To solve the problem the three disks must be transferred to the right hand peg, with the largest disk on the bottom and the smallest on the top. The rules are that only one disk may be moved at a time and no disk may be placed on a peg on top of a smaller disk.



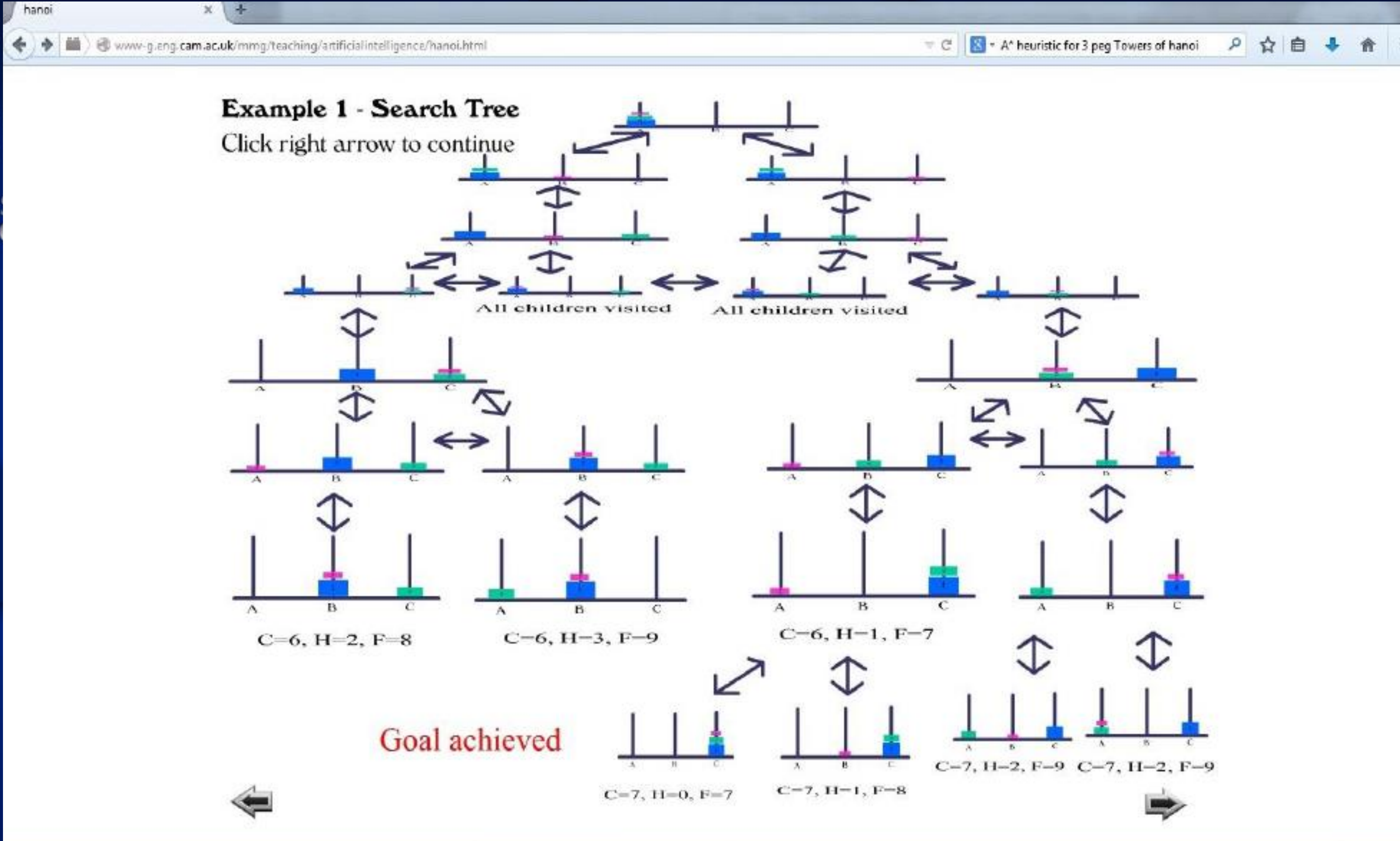
From: http://www-g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/hanoi.html

Towers of Hanoi

- Initial State:
- Goal Test:
- Successor Function:
- Cost Function:

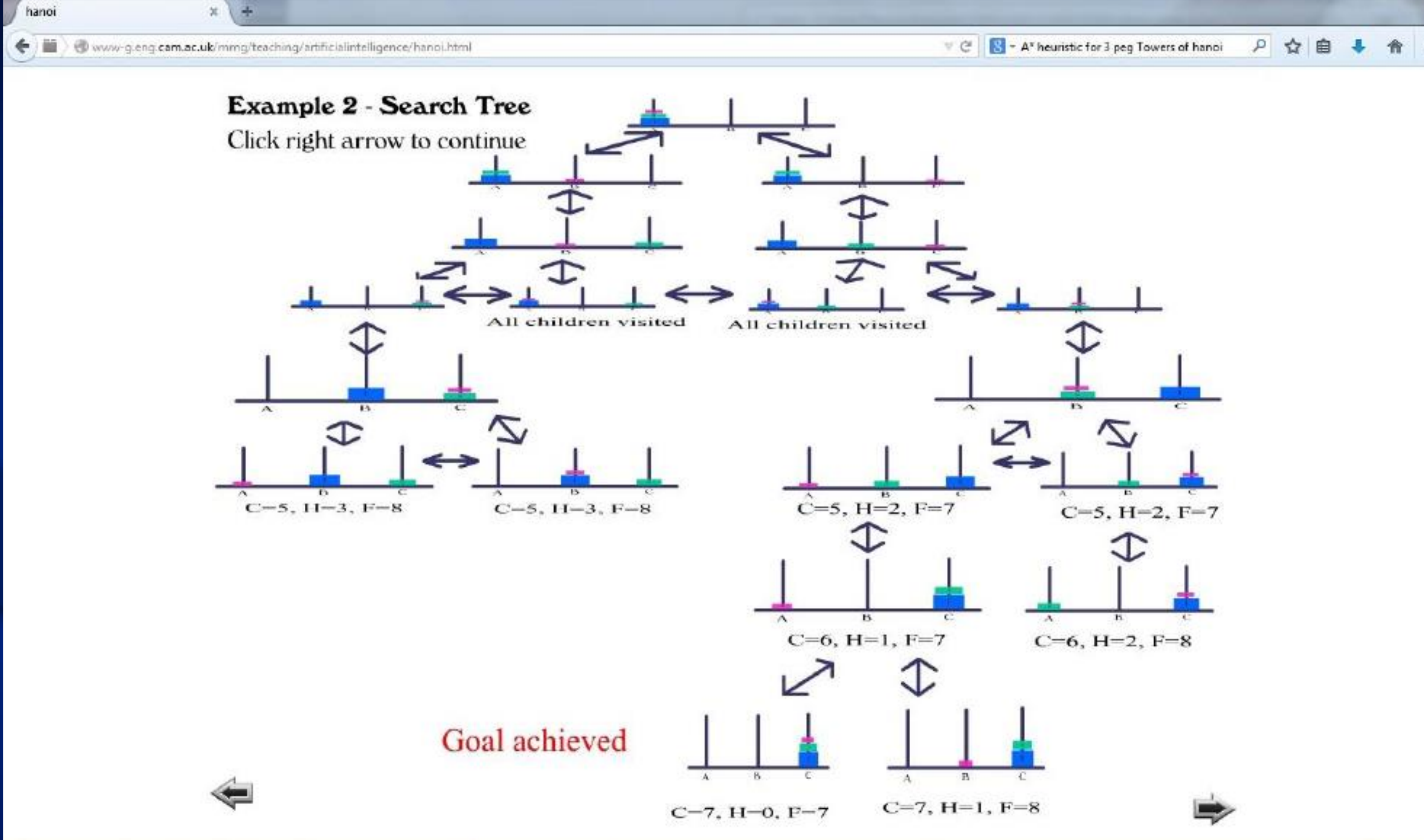
Solution: Breadth-First

1. {[1 2 3] [ ] [ ]}
2. {[2 3] [1] [ ], [2 3] [ ] [1]}
3. {[3][1][2],[3][2][1]}
4. {[3][ ][1 2],[3][1 2] [ ], [1 3] [ ] [2],[1 3][2] [ ]}
5. {[ ][3] [1 2],[][13][2}
6. {[1][3][2],[1][2][3]],[ ] [1 2] [3],[][2][13]}
7. {[1][23][],[2][13][],[1][ ][2 3],[1][2][3]}
8. {[][23][1],[][123][],[12][3][],[2][3][1],[ ] [ ][1 2 3],[][1][23]}



Heuristic: Number Disks not on goal (23 of 27 expanded)

From: http://www-g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/hanoi.html



Heuristic: # Disks not on right disk (19 of 27 expanded)

From: http://www-g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/hanoi.html

Breadth-First Search

```
[nn4,sol4] = CS5300_BFS_Hanoi

nn4 = 1x77 (50 dups) struct array with fields:

    state
    parent
    children

sol4 =    1    3    7    15    25    32    45    56
```

Breadth-First Search

```
>> CS5300_Hanoi_show_sol(nn4,sol4);
Solution for Towers of Hanoi


 1 2 3 ---  [ ]  ---     [ ]
   2 3 ---  [ ]  ---      1
     3 ---    2  ---      1
     3 ---  1 2  ---     [ ]
    [ ] --- 1 2  ---      3
     1 ---    2  ---      3
     1 ---  [ ]  ---    2 3
    [ ] --- [ ]  ---  1 2 3
>>
```

1 2 3 4	--- [ ]	--- [ ]
2 3 4	--- 1	--- [ ]
3 4	--- 1	--- 2
3 4	--- [ ]	--- 1 2
4	--- 3	--- 1 2
1 4	--- 3	--- 2
1 4	--- 2 3	--- [ ]
4	--- 1 2 3	--- [ ]
[ ]	--- 1 2 3	--- 4
[ ]	--- 2 3	--- 1 4
2	--- 3	--- 1 4
1 2	--- 3	--- 4
1 2	--- [ ]	--- 3 4
2	--- 1	--- 3 4
[ ]	--- 1	--- 2 3 4
[ ]	--- [ ]	--- 1 2 3 4

Solution for 4 disks

Breadth-First Search

Need to pick best node for expansion:

evaluation function orders nodes (priority queue)

Best-First Search

function Best-First-Search(problem, eval-fn)
  returns result

queueing-fn = a function sorted by eval-fn

return General-Search(problem,queueing-fn)

Measure: estimate cost of path to closest goal

Greedy Search

Minimize estimated cost to reach goal

heuristic function: estimates cost

- $h(n)$ = estimated cost of cheapest path from node n to goal

Greedy Search

function Greedy-Search(problem)
    returns result

return Best-First-Search(problem,h)

Example Heuristic

Route finding

H(n) = straight-line distance from n to goal

Not optimal or complete

Uniform Cost

$g(n) = depth(n)$

Optimal and complete, but inefficient

Use: $f(n) = g(n) + h(n)$

where f estimates cost of cheapest solution through n

Admissible Heuristic

Complete and optimal if:

h never over-estimates cost to goal

(e.g., $h(n) = 0$ works!)

A* Search

function A*-Search(problem)
  returns result
return Best-First-Search(problem,g+h)

- A* is optimally efficient: expands fewest nodes of any algorithm
- # nodes is exponential in solution length

Performance

Effective branching factor: b*

$$N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

Solve for b*, given N and d

(How to solve?)

Heuristics for 8-Puzzle

- h1(n) = number misplaced tiles
- h2(n) = sum (goal_i-tile_i)
- humans = ?

181,440 reachable states

2 days at 1/sec nonstop

Iterative Refinement

Start with complete configuration and make modifications to improve quality

Random restart: run from several random initial states and take best

Wumpus World

- Initial State: [1,1,0]
  - All states: [x,y,d] d ϵ {0,1,2,3}
- Actions: {FORWARD,RIGHT,LEFT}
- Transition Model: (x,y,d) → (x',y',d')
- Goal: Gold in [x_G, y_G] & state=[x_G, y_G,d]
- Cost: each action costs 1 unit

Wumpus World

- More on transition model
  - ([x,y,d],FORWARD) → [x’,y’,d]

  Where [x’,y’] is the neighbors cell in direction d, or if none, then x’=x, y’=y
  - ([x,y,d],RIGHT) → [x,y,rem(d+3,4)]
  - ([x,y,d],LEFT) → [x,y,rem(d+1,4)]

Questions?