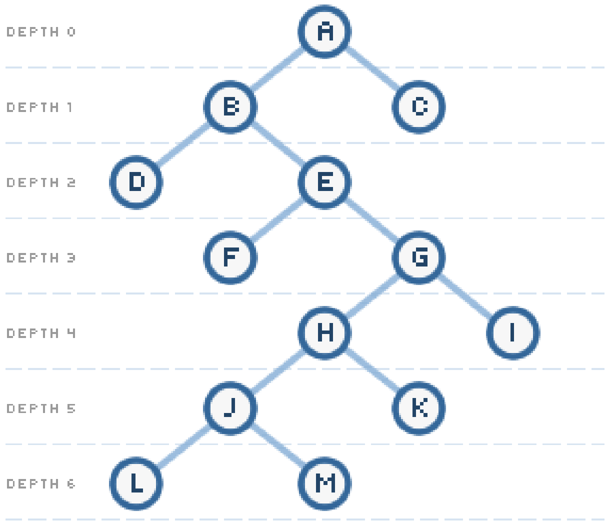
Artificial Intelligence – Lecture 2

Representation

- From AI admirers to AI programmers.
  - Step 1: Represent the problem so that it is computer-friendly.
  - Step 2: Code the problem in a programming language.
  - Step 3: Develop/code an algorithm to find a solution.
  - Step 4: Represent the solution so that it is human-friendly.

- Representation...



Blind search

- If we have no *extra* information to guide the search
  - Depth first search
  - Breadth first search
  - Iterative deepening

- But you can still do some cool things….

The water jug problem:

Suppose you are given 1 jug (3L) and 1 jug (4L). You also have a tap With which you can fill the jugs.

4L

3L

Goal: Get exactly 2L in the 4L jug.

Representation

- Step 1: Representing the problem for a machine.

We represent the amount of water in the jugs with (X,Y)

1.(X,Y) -> (4,Y) Fill the 4 liter jug.

2.(X,Y) -> (X,3) Fill the 3 liter jug.

3.(X,Y) -> (0,Y) Empty the four liter jug

4.(X,Y) if X+Y >= 4 and Y > 0 -> (4,Y-(4-X))
Fill the 4 liter jug with water from

the 3 liter jug.

Representation

Depth first

Breadth first

Water Jug Problem

1. (X,Y: X < 4) -> (4,Y) — Fill the 4-liter jug
2. (X,Y: Y < 3) -> (X,3) — Fill the 3-liter jug
3. (X,Y: X > 0) -> (0,Y) — Empty the 4-liter jug on the ground
4. (X,Y: Y > 0) -> (X,0) — Empty the 3-liter jug on the ground
5. (X,Y: X+Y >= 4 and Y > 0) -> (4,Y-(4-X))
   Fill the 4-liter jug from the 3-liter jug
6. (X,Y: X+Y >= 3 and X > 0) -> (X-(3-Y),3))
   Fill the 3-liter jug from the 4-liter jug
7. (X,Y: X+Y <= 4 and Y > 0) -> (X+Y,0)
   Pour all water from the 3-liter jug into the 4-liter jug
8. (X,Y: X+Y <= 3 and X > 0) -> (0,X+Y))
   Pour all water from the 4-liter jug into the 3-liter jug
9. (X,Y: X > 0) -> (X-D,Y)
10. (X,Y: Y > 0) -> (X,Y-D)

Representation in python

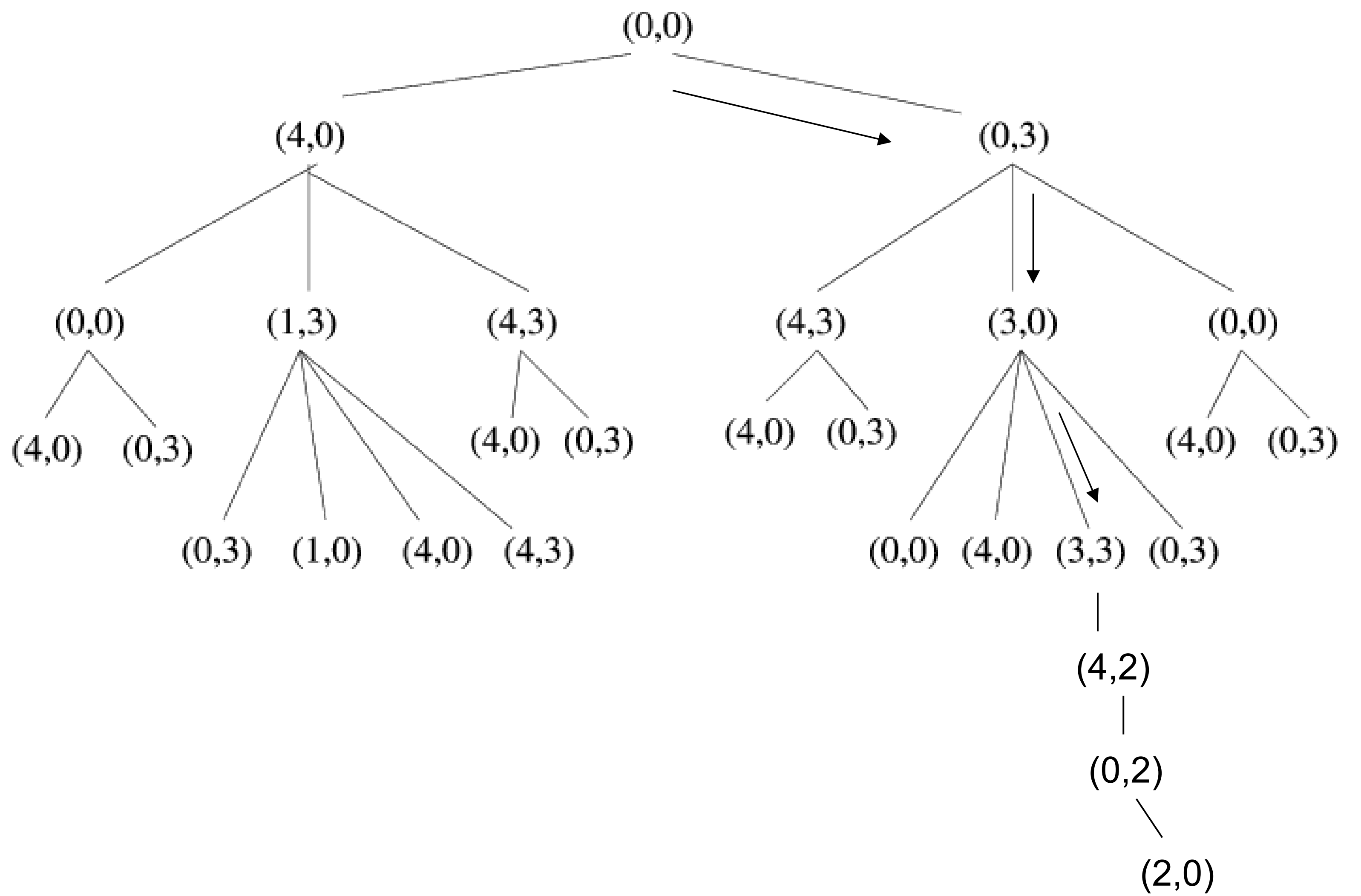

```
# Each state is a tuple (x,y) of water
def nextStates(current_state):
  x,y = current_state
  states = [(4, y),(x, 3),(0, y),(x, 0)]
  if x+y >= 4:
      # Fill 4 liter jug from the 3 liter one
      states = states + [(4,y-(4-x))]
  else:
      # Pour everything from the 3 liter to the 4 liter one
      states = states + [(x+y,0)]
  if x+y >= 3:
      # Fill the 3 liter jug from the four liter one
      states = states + [(x-(3-y),3)]
  else:
      # Pour everything from the 4 litre to the 3 litre jug
      states = states + [(0,x+y)]
  # Remove duplicate states
  return list(set(states))
```

Representation in python


```
>>> nextStates( (0,0) )
[(0, 3), (0, 0), (4, 0)]
>>> nextStates( (0, 3) )
[(3, 0), (0, 3), (0, 0), (4, 3)]
>>> nextStates( (3, 0) )
[(3, 0), (0, 3), (0, 0), (3, 3), (4, 0)]
```

Depth first

Breadth first



Silly implementation – don't do this

- Just try everything until it works

```
def silly( state , goal ):
  visited_states = [ (state) ]
  while state != goal:
    choices = nextStates(state)
    next = choices[random.randrange(0,len(choices))]
    state = next
    visited_states += [state]
  return visited_states
```

Silly implementation – it's really bad


```
>>> silly( (0,0), (3,0) )
[(0, 0), (0, 3), (0, 0), (0, 3), (0, 0),
(4, 0), (4, 0), (4, 0), (4, 3), (4, 3),
(4, 3), (4, 0), (4, 0), (4, 0), (0, 0),
(0, 3), (3, 0)]
```

Recursive depth-first search


```
def recursiveDF(state, goal, previous):
    if state == goal:
        return previous
    for choice in nextStates(state):
        if choice in previous:
            # We have already been in that state before
            continue
        else:
            solution = recursiveDF(choice, goal, previous+
[choice])
            if solution != []:
                return solution
    return []
```

Recursive depth first


```
>>> recursiveDF( (0,0), (2,0), [(0,0)] )
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2),
(0, 2), (2, 0)]

>>> recursiveDF( (0,0), (0,1), [(0,0)] )
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2),
(0, 2), (2, 0), (2, 3), (4, 1), (0, 1)]
```

Iterative implementation of depth first

- Store a list of states to visit
- If the first state is the goal state, then finished
- Remove first state from the list
  - compute all choices for this state
  - remove choices where we have already been (loop detection!)
    - Store not only current state, but all previous states!
  - add all choices to the beginning of list

Iterative implementation of depth first


```
def dfSearch( start, goal ):
    l = [ [start] ]
    while l != []:
        path = l[0]
        l = l[1:]
        if path[-1] == goal:
            return path
        choices = nextStates( path[-1] )
        for c in choices:
            if c not in path:
                l = [path+[c]] + l
    return []
```

Testing it


```
>>> dfSearch( (0,0), (2,0) )
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]
```

Evaluation criteria

- Complete:
  - Does the algorithm always find a solution if it exists?
- Optimal:
  - Is the solution always “the best” one?
    - eg. length of solution
- Space:
  - How much memory does it take to find a solution?
- Time:
  - How long time does it take to find a solution?

Evaluating depth first

- Complete:
  - Only if we avoid loops and search tree is finite
- Optimal:
  - No, we're satisfied with *any* solution
- Space:
  - Only as much as needed to remember the current path

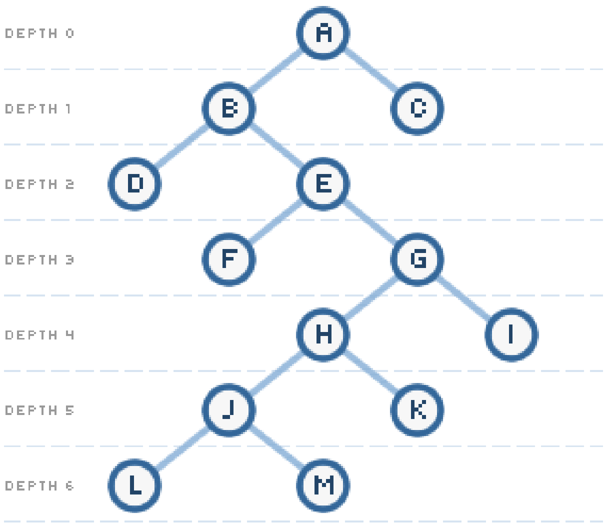
How can we find the "best" solution?

Depth first

Breadth first

- Idea 1: find *all* solutions and compare them
  - This can be quite many....
- Idea 2: explore the tree so that we look for the shortests solutions at each time.

```
Depth first:
   a b d e f g h j ...
Breadth first:
   a b c d e f ...
```



Iterative implementation of breadth first

- Store a queue of states to visit
- If the first state is the goal state, then finished
- Remove first state from the queue
  - compute all choices for this state
  - remove choices where we have already been (loop detection!)
    - Store not only current state, but all previous states!
  - add all choices to end of queue

Iterative implementation of breadth first

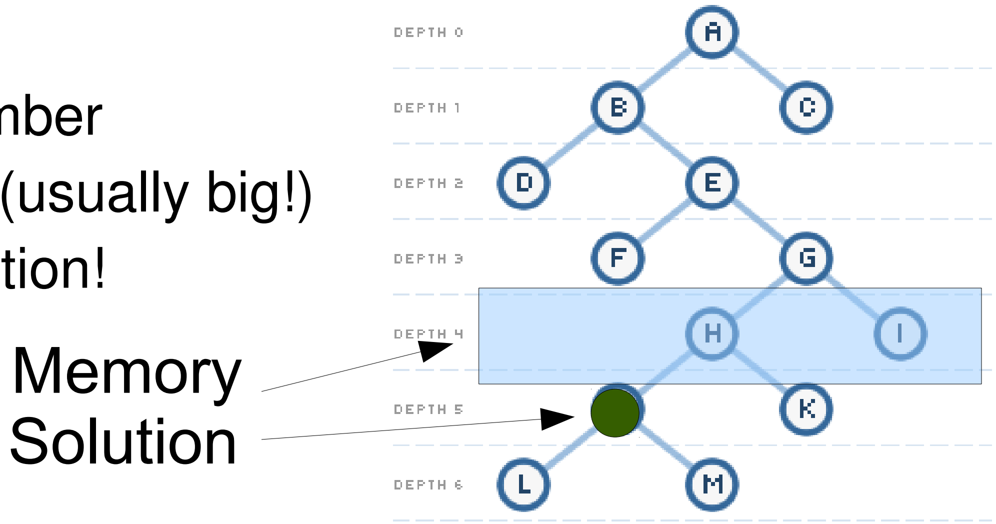

```
def bfSearch( start, goal ):
    l = [ [start] ]
    while 1:
        path = l.pop(0)
        if path[-1] == goal:
            return path
        choices = nextStates( path[-1] )
        for c in choices:
            if c not in path:
                l.append(path+[c])
    return []
```

Testing it


```
>>> bfSearch( (0,0), (2,0) )
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]
```

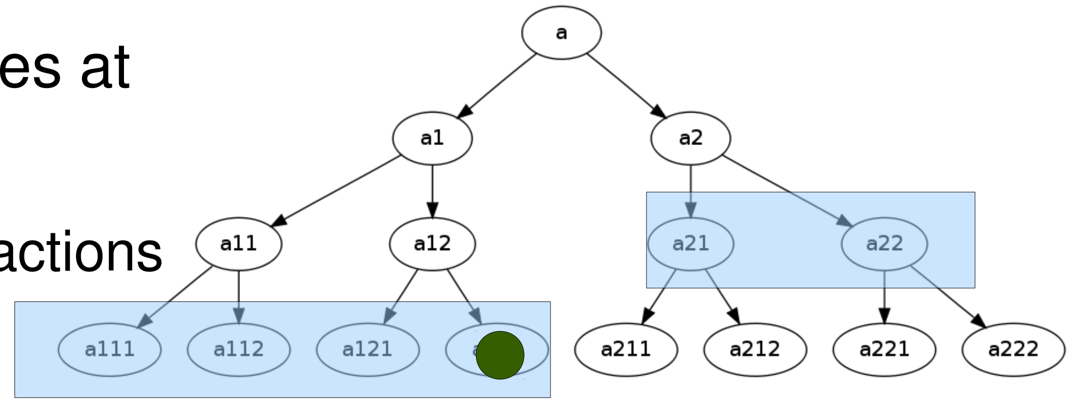
Evaluating breadth first

- Complete:
  - Yes, if a solution exists
- Optimal:
  - Yes, the first one found must have shortest path
- Space:
  - Need to remember <u>the whole row</u> (usually big!) above the solution!

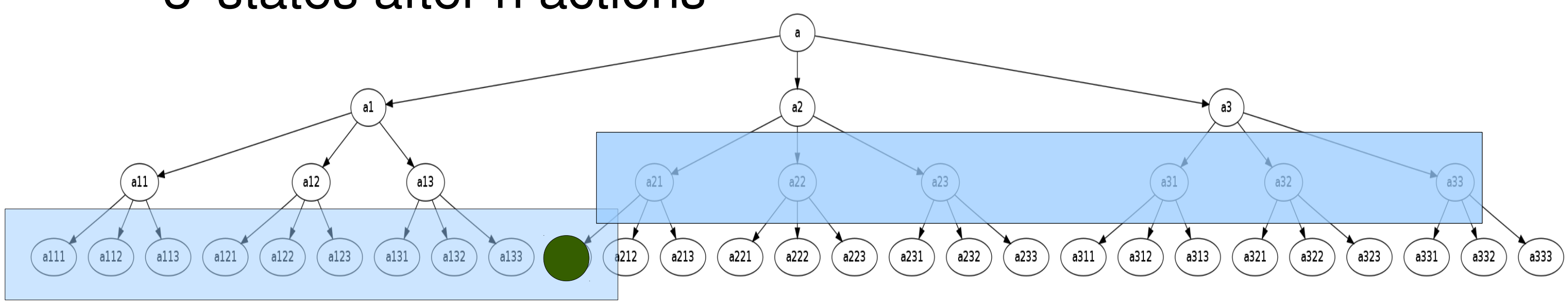


Branching factor

- If we have 2 choices at each state
  - 2^n states after n actions



- If we have 3 choices at each state
  - 3^n states after n actions



Analysing breadth first search

- If the solution exists at depth n, then breadth first search takes $O(B^n)$ time where B is the *branching factor* of the problem, and uses $O(B^n)$ space
- If the <u>found</u> solution exists at depth n, then depth first search takes $O(n)$ time where B is the *branching factor* of the problem, and uses $O(B^n)$space.

The problem

- Depth first
  - Not optimal
  - Uses O(n) space
- Breadth first
  - Optimal
  - Uses O(B^n) space
- Can we combine the advantages of both approaches?

Iterative deepening (IDA)

- Let M be a *maximum depth*.
- Run depth first, but only until the given depth.
- Repeat for increasing values of M. M=1, M=2 ...

Iterative deepening


```
def idaSearch( start, goal ):
    M = 0 ; l=[]
    while 1:
        if l == []:
          M = M+1 ; l = [[start]]
        path = l.pop()
        if path[-1] == goal:
            return path
        if len(path) > M:
          continue
        choices = nextStates( path[-1] )
        for c in choices:
            if c not in path:
                l = [path+[c]] + l
    return []
```

Iterative deepening : water jugs problem


```
>>> idaSearch( (0,0), (2,0) )
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]
>>> idaSearch( (0,0), (0,1) )
[(0, 0), (4, 0), (1, 3), (1, 0), (0, 1)]
```

```
How many nodes where visited?
>>> idaSearch ( (0,0), (0,2) )
73 nodes visited
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)]
>>> dfSearch ( (0,0), (0,2) )
25 nodes visited
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)]
```

Analysing iterative deepening

Depth first

Breadth first

- Completeness
  - Yes, will return the *"best"* (shortest) solution
- Space complexity
  - O*(n)*
- Time complexity
  - Each iteration of *M* takes: $O(B^M)$ time
  - Total time: $O(B^1)+O(B^2)+\ldots+O(B^N) = O(B^N)$
    - That's the same complexity as both df and breadth first!
  - In *practice*, if we skip the big-O-notation
    - IDA takes *B* times longer
    - Use breadth first if we have enough memory, otherwise IDA
    - Using too much memory causes swapping – which is slow!

Heuristic search

- Basic idea
  - Use some *domain knowledge* to create a *heuristic* that tells how close to the goal a solution is.
    - Example: In navigation, count the *distance* to the destination
  - Heuristic does not have to be perfect, only give a rough guide to how good/bad a solution is
  - When searching, expand first the nodes that have a good heuristic value

A* search

- Use a cost function: $f(n) = g(n) + h(n)$
  - $g(n)$: cost from root node to this node
  - $h(n)$: admissible heuristic cost from this node to goal
  - Admissible heuristic: must never overestimate the distance to the goal
- For each node, keep track of cost $f(n)$
- Expand the node n that have the lowest cost
- Compute cost of children. Insert *sorted* into list of nodes
  - Sort explicitly (inefficient) or,
  - Iterate over list and insert at "*right*" place

A* search

- The efficiency of A* depends on the heuristic
- Provides a good way of combining *domain knowledge* with general search.
- Provides *optimal* solutions iff heuristic is *admissible*
- Time complexity
  - In worst case: $O(B^n)$
- Space complexity
  - In worst case: $O(B^n)$