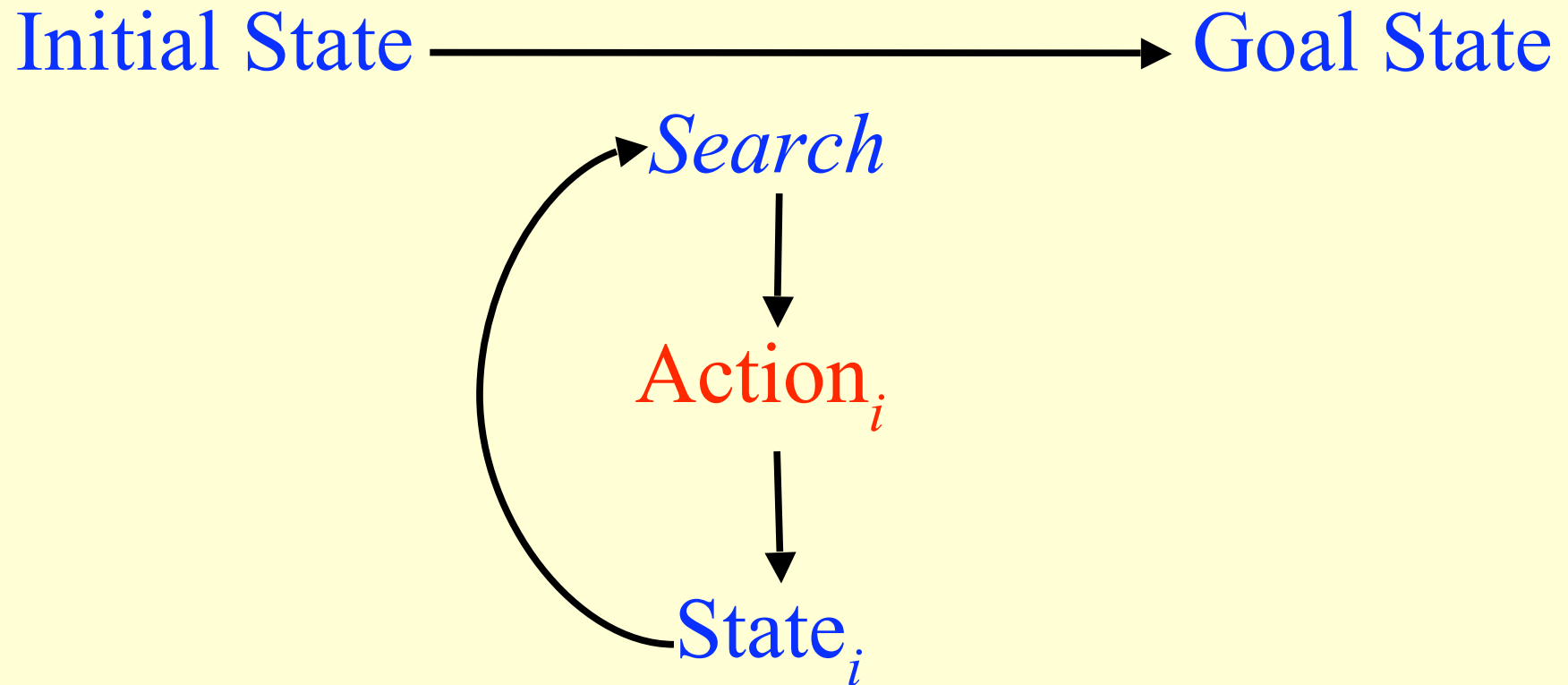


Chapter 3: Solving Problems by Searching

The General Idea



A Simple Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT(p) **returns** an action

inputs: p , a percept

static: s , an action sequence, initially empty

$state$, some description of the current world state

g , a goal, initially null

$problem$, a problem formulation

$state \leftarrow$ UPDATE-STATE($state, p$)

if s is empty **then**

$g \leftarrow$ FORMULATE-GOAL($state$)

$problem \leftarrow$ FORMULATE-PROBLEM($state, g$)

$s \leftarrow$ SEARCH($problem$)

$action \leftarrow$ RECOMMENDATION($s, state$)

$s \leftarrow$ REMAINDER($s, state$)

return $action$

A Simple Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT(*p*) **returns** an action

inputs: *p*, a percept

static: *s*, an action sequence, initially empty
state, some description of the current world state
g, a goal, initially null
problem, a problem formulation

← Instance variables

state ← UPDATE-STATE(*state*, *p*)

if *s* is empty **then**

g ← FORMULATE-GOAL(*state*)

problem ← FORMULATE-PROBLEM(*state*, *g*)

s ← SEARCH(*problem*)

← Constructor

action ← RECOMMENDATION(*s*, *state*)

s ← REMAINDER(*s*, *state*)

return *action*

In Python

```
class SimpleProblemSolvingAgent:

    def __init__(self):
        self.s = []
        self.state = None
        self.g = None
        self.problem = None

    def respond(self, p):
        if len(s) == 0: # initially and perhaps later
            self.g = self.formulateGoal()
            self.problem = self.formulateProblem()
            self.s = self.Search()
        action = self.Recommendation()
        self.s = self.Remainder()
        return action
```

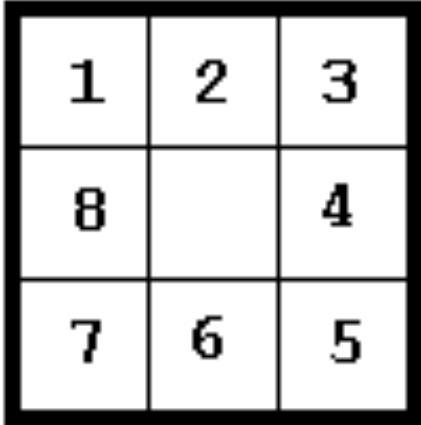
Terminology

- **operator** – an action taken to reach a state
- **state space** – the set of all possible states reachable from the initial state
- **goal test** – does state satisfy goal?
- **path** - sequence of actions leading from one state to another
- **path cost** – cost associated with a particular path (e.g., flying more expensive than driving)
- **search cost** – time/space required to find a solution

Classic (Toy) Problems

The 8-Puzzle

- **States** : location of each tile
- **Operators** : move blank up, right, left, or down
- **Goal test** : state looks like figure at right
- **Path cost** : total number of moves

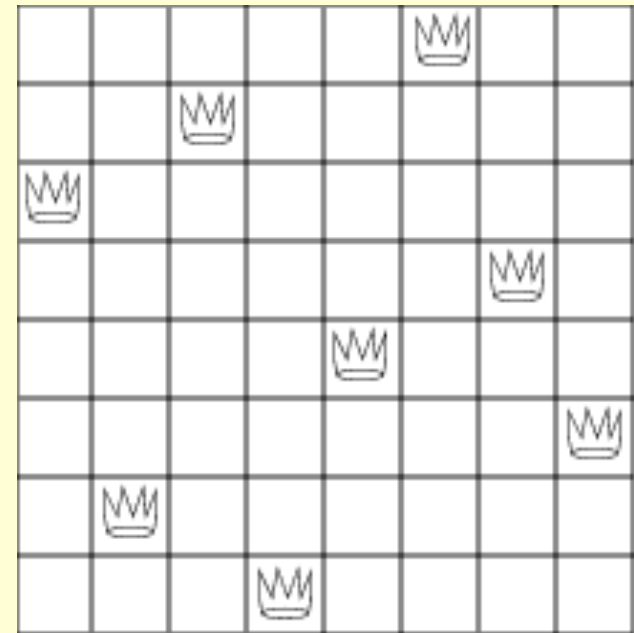


1	2	3
8		4
7	6	5

Classic (Toy) Problems

The 8-Queens Problem

- **Goal test** : 8 queens on board, none under attack
- **Path cost** : zero (???)



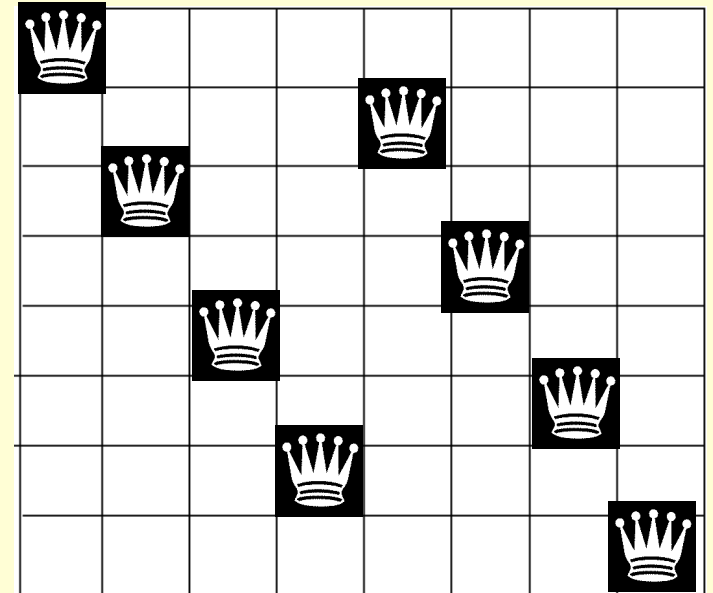
8-Queens Variant #1

- **States** : arrangement of 0-8 queens on board
- **Operators** : add a queen to any square (till there are 8 on the board)
- **Search cost** : 64^8 possible configs to check!

Hint: We shouldn't put a queen on an attacked position...

8-Queens Variant #2

- **States** : arrangement of 0-8 queens on board with none under attack
- **Operators** : place a queen in the leftmost empty column s.t. it is not attacked
- **Search cost** : 2057 possible sequences (one at right fails)



Conclusion: A better formulation of the problem can make a big difference!

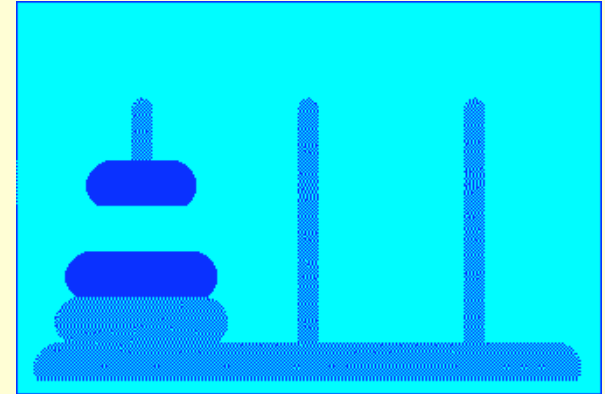
Classic (Toy) Problems

Missionaries & Cannibals

- **States** : $\langle M, C, B \rangle$ on starting side, where $M = \#$ missionaries, $C = \#$ cannibals, $B = \#$ boats
- **Operators** : From each side, take either 1 M, 1 C, 2 M, 2 C, or one of each across.
- **Goal test** : $\langle 0, 0, 0 \rangle$
- **Path cost** : $\#$ of crossings

Classic (Toy) Problems

Towers of Hanoi



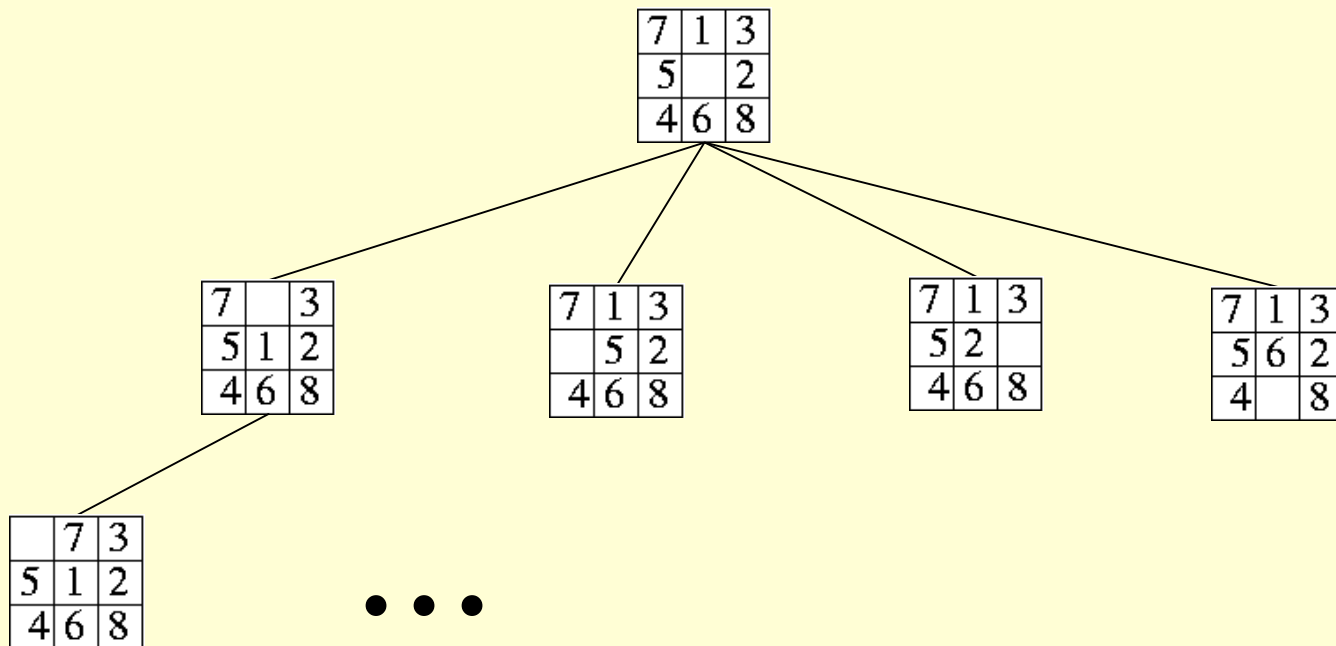
- **States** : $\langle P_1, P_2, P_3 \rangle$ where P_i is configuration of i th peg
- **Operators** : Move a disc from P_i to P_j without putting a larger disc on a smaller one.
- **Goal test** : $\langle 0, 0, P_3 \rangle$
- **Path cost** : # of moves

Real-World Problems

- **Route finding** : go from New York to San Francisco for under \$200 with the minimum # of stopovers.
- **Traveling Salesman Problem** : Visit every major city in a region without visiting any city twice.
- **VLSI layout** : lay out circuit to minimize area and connection lengths
- **Robot navigation** : continuous route-finding
- **Assembly sequencing**: putting together complex objects

3.3 Searching for Solutions

- Many search spaces can be represented as trees.
- Each node in the tree represents a state.
- Each branch in the tree represents an action.



A General Tree-Search Algorithm

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

- Search tree is just a list (stack, queue) - expanding a node means getting its children and replacing it with them
- **INSERTALL** function implements a strategy (more later)
- Let's look at the **EXPAND** function....

A General Tree-Search Algorithm: Expanding Nodes

```
function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

STATE[*node*], *action*, *result*

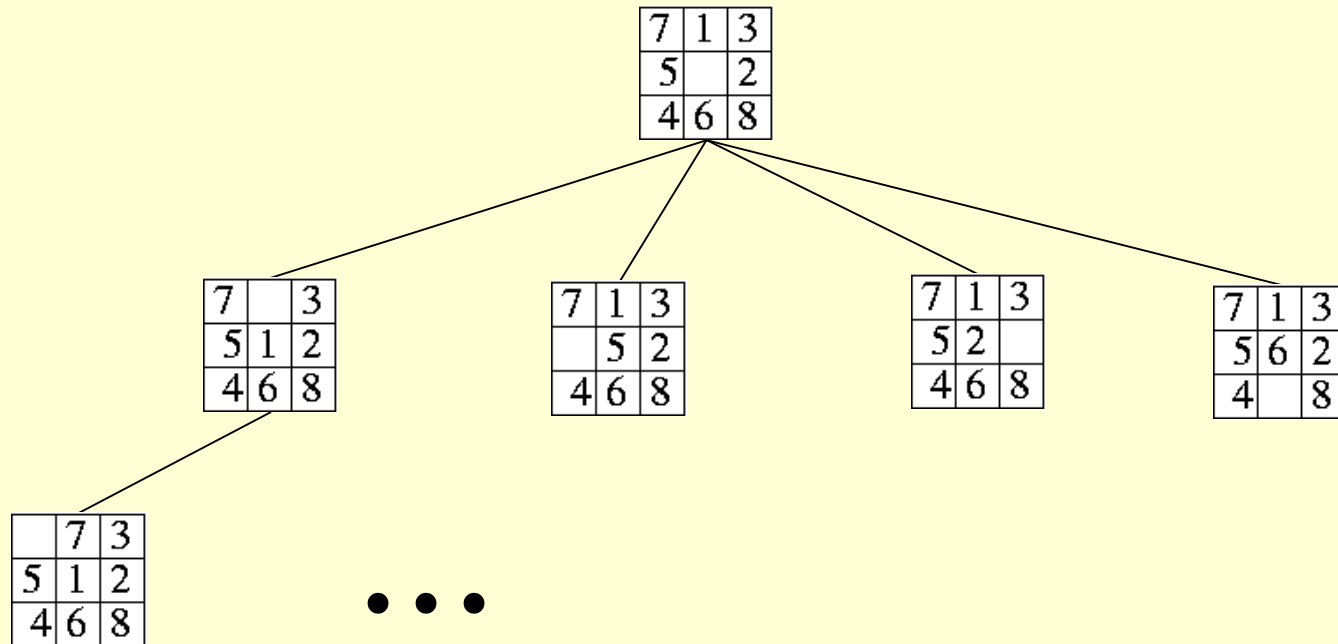
Now we can turn to search strategies...

3.4 Search Strategies

Issues:

- **Completeness:** if there's a solution, will strategy find it? (c.f. Gödel's Incompleteness Theorem; Perceptron Convergence Theorem)
- **Time complexity:** How long does it take?
- **Space complexity:** How much memory needed?
- **Optimality:** Will we get the best solution?

Breadth-First Search



- This can be implemented by having **INSERTALL** put the new nodes at the end of the list - *i.e.*, a queue (FIFO)

Breadth-First Search

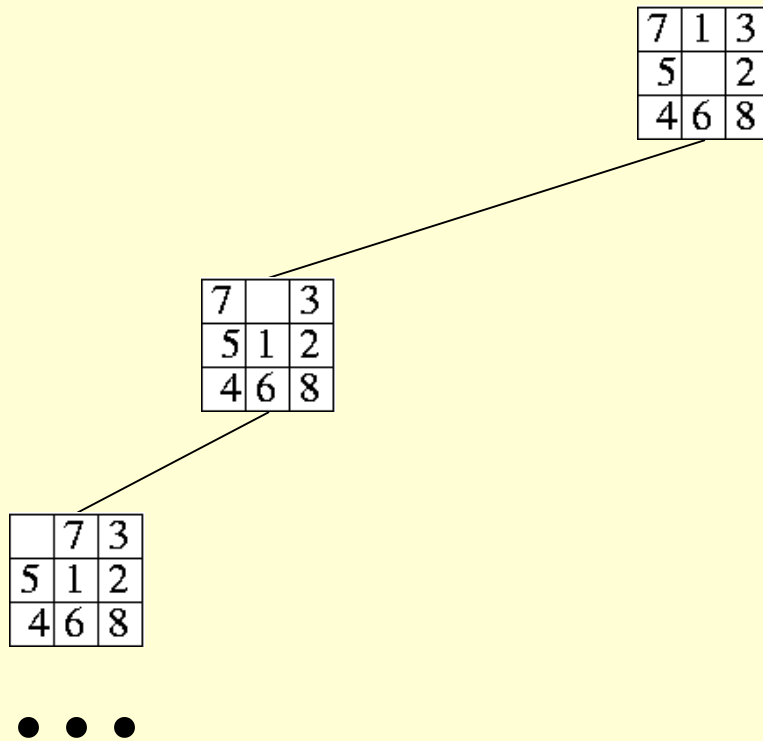
- **Complete?** Yes (considers all paths of a given length)
- **Optimal?** Yes (always finds shortest path), provided path cost increases with path length
- **Time complexity:** $O(b^d)$, for branching factor b , depth d
- **Space complexity:** same

Breadth-First Search

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

- Memory requirements dominate (though 111 MB no big deal nowadays!)
- Time is still very expensive

Depth-First Search



- This can be implemented by having **INSERTALL** put the new nodes at the front of the list - *i.e.*, a stack (LIFO)

Depth-First Search

- **Complete?** No (can get stuck in loops or “infinite” paths)
- **Optimal?** No (best solution may be higher up in another branch)
- **Time complexity:** $O(b^m)$, for branching factor b , max depth m
- **Space complexity:** $O(bm)$ (versus b^d for breadth-first, where d = depth of shallowest goal)
- **Conclusion:** Avoid for search trees with large or infinite maximum depth

Implementation: Python

- Search class with abstract `insertAll` (strategy) method
- `DepthFirstSearch` subclass implements `insertAll` as stack (insert at front of list)
- `BreadthFirstSearch` implements it as queue (insert at back of list)
- Don't need to pass in fringe, just problem
- Data-structure translation example:

`STATE[s] ← result` translates to `s.state = result`

Implementation: LISP/Scheme (FYI)

- As in Python, use list for stack/queue
- Pass in strategy function as a parameter
- Use recursion instead of do-loop

Depth-limited Search

- Depth-first search with a depth limit
- Often know limit; e.g., for route, total number of cities
- **Complete?** Yes
- **Optimal?** No (best solution may be higher up in another branch)
- **Time complexity:** $O(b^l)$, for branching factor b , depth limit l
- **Space complexity:** $O(bl)$

Depth-limited Search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

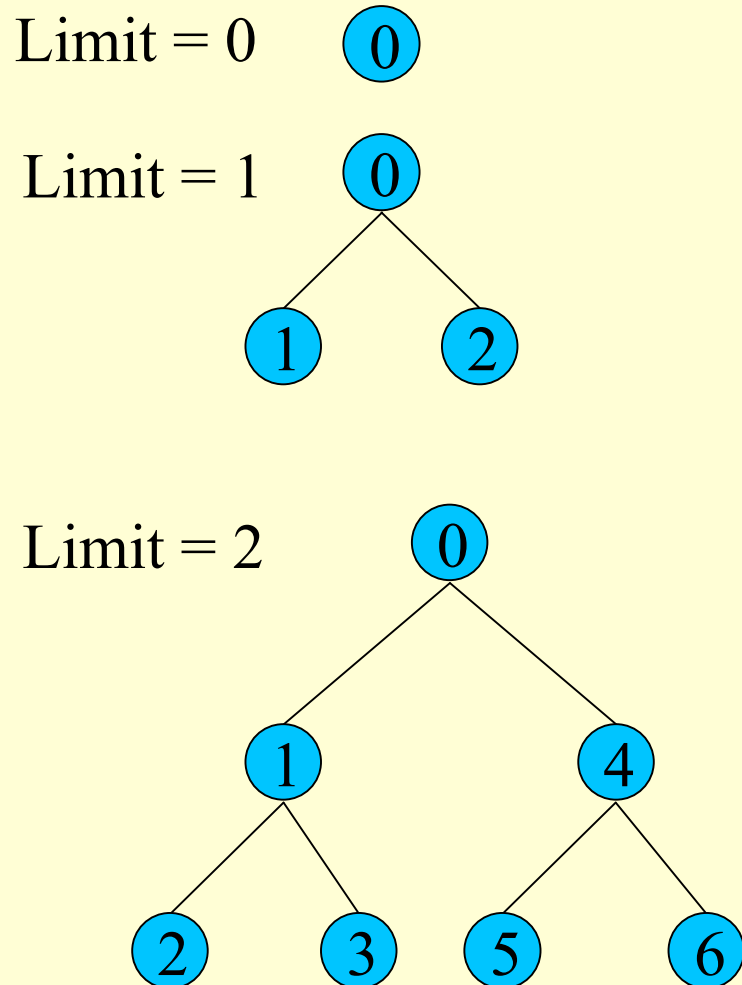
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Iterative-deepening Search

- Depth-limited search with increasing limits
- **Complete? Yes**
- **Optimal? Yes**
- **Time complexity: $O(b^l)$**
- **Space complexity: $O(bl)$**

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
  end
```

Iterative-deepening Search



- *Q*: Doesn't repetition waste time?
- *A*: Not really: most nodes are at bottom.
- So only slightly more expensive than depth-limited search.

Uniform-Cost Search

- Expand “cheapest” node first
- So order fringe nodes by cost
- A subclass of “best-first” search
- Other famous best-first is A*
- **Complete?** Yes, assuming cost is well defined
- **Optimal?** Yes
- **Time complexity:** $O(b^x)$
- **Space complexity:** $O(b^x)$

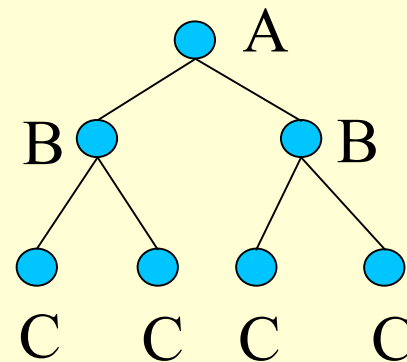
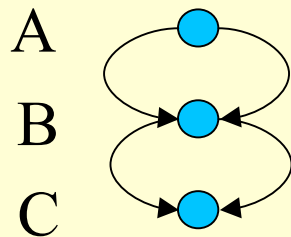
Where x is proportional to cost of optimal solution

Comparing Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Postscript: Avoiding Repeated States

- For some problems (8 Queens), not an issue
- Others (Missionaries & Cannibals; Mazes) are reversible and so can repeat states indefinitely.
- Even without reverses, can have exponential search from linear path:



Avoiding Repeated States: Solutions

- Don't return to a state that you just came from (cheap but not guaranteed).
- Don't create paths with cycles (more expensive, but better).
- Don't generate a state that's been generated before (most expensive, but guaranteed).
- If you do need to repeat a state (e.g., in maze), maintain a list of visited states and put non-visited states at front of fringe (e.g. by sorting).