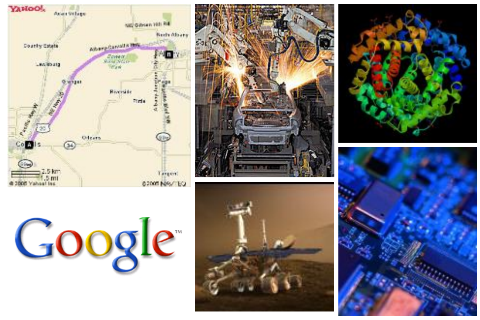
CS 331: Artificial Intelligence
Uninformed Search

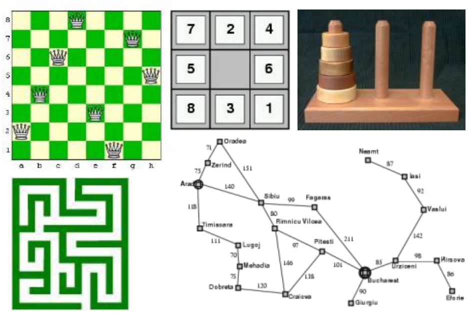
1

Real World Search Problems



2

Simpler Search Problems



3

Assumptions About Our Environment

- Static
- Observable
- Discrete
- Deterministic
- Single-agent

4

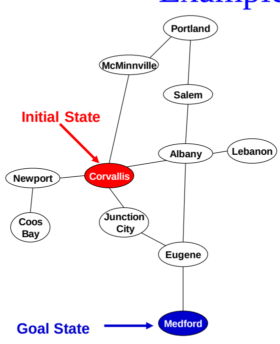
Search Problem Formulation

A search problem has 5 components:

1. A finite set of states S
2. A non-empty set of initial states $I \subseteq S$
3. A non-empty set of goal states $G \subseteq S$
4. A successor function ***succ(s)*** which takes a state ***s*** as input and returns as output the set of states you can reach from state ***s*** in one step.
5. A cost function ***cost(s,s')*** which returns the non-negative one-step cost of travelling from state ***s*** to ***s'***. The cost function is only defined if ***s'*** is a successor state of ***s***.

5

Example: Oregon



S = {Coos Bay, Newport, Corvallis, Junction City, Eugene, Medford, Albany, Lebanon, Salem, Portland, McMinnville}

I = {Corvallis}

G={Medford}

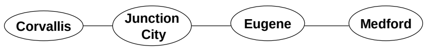
Succ(Corvallis)={Albany, Newport, McMinnville, Junction City}

Cost(s,s') = 1 for all transitions

6

Results of a Search Problem

- Solution
  Path from initial state to goal state



- Solution quality
  Path cost (3 in this case)
- Optimal solution
  Lowest path cost among all solutions (In this case, we found the optimal solution)

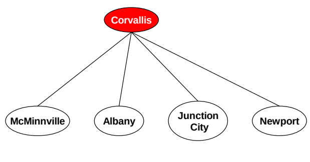
7

Search Tree

Start with Initial State

8

Search Tree

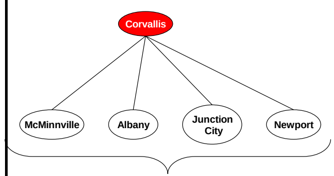


Is initial state the goal?

- Yes, return solution
- No, apply Successor() function

9

Search Tree



Apply Successor() function

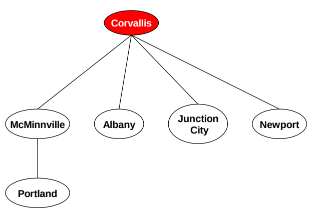
These nodes have not been expanded yet. Call them the fringe. We'll put them in a queue.

Queue

McMinnville
Albany
Junction City
Newport

10

Search Tree



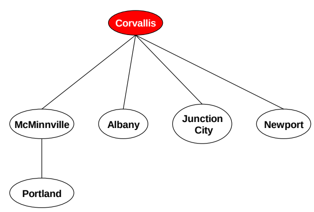
Queue

Albany
Junction City
Newport
Portland

Now remove a node from the queue. If it's a goal state, return the solution. Otherwise, call Successor() on it, and put the results in the queue. Repeat.

11

Search Tree



Queue

Albany
Junction City
Newport
Portland

Things to note:

- Order in which you expand nodes (in this example, we took the first node in the queue)
- Avoid repeated states

Tree-Search Pseudocode

```
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 SOLUTION(node)
     fringe ← INSERTALL(EXPAND(node, problem), fringe)

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
```

13

Tree-Search Pseudocode

```
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 SOLUTION(node)
     fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
  successors ← the empty set
     PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
     DEPTH[s] ← DEPTH[node] + 1
     add s to successors
  return successors
```

Note: Goal test happens after we grab a node off the queue.

14

Tree-Search Pseudocode

```
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
                                                     TION(node)

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
```

Why are these parent node backpointers are important?

15

Uninformed Search

- No info about states other than generating successors and recognizing goal states
- Later on we'll talk about informed search – can tell if a non-goal state is more promising than another

16

Evaluating Uninformed Search

- Completeness
  Is the algorithm guaranteed to find a solution when there is one?
- Optimality
  Does it find the optimal solution?
- Time complexity
  How long does it take to find a solution?
- Space complexity
  How much memory is needed to perform the search

17

Complexity

1. Branching factor (b) – maximum number of successors of any node
2. Depth (d) of the shallowest goal node
3. Maximum length (m) of any path in the search space

Time Complexity: number of nodes generated during search

Space Complexity: maximum number of nodes stored in memory

18

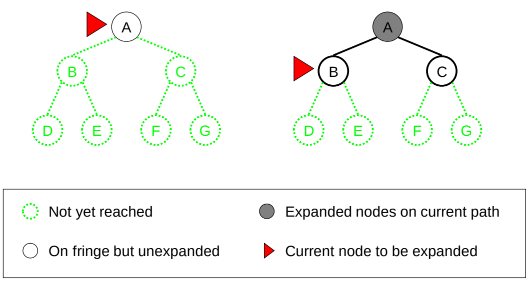
Uninformed Search Algorithms

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative Deepening Depth-first Search
- Bidirectional search

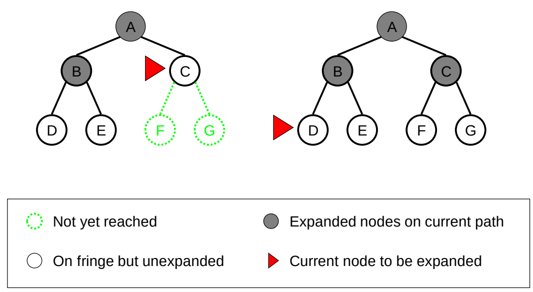
Breadth-First Search

- Expand all nodes at a given depth before any nodes at the next level are expanded
- Implement with a FIFO queue

Breadth First Search Example



Breadth First Search Example



Evaluating BFS

Complete?	
Optimal?	
Time Complexity	
Space Complexity	

Evaluating BFS

Complete?	Yes provided branching factor is finite
Optimal?	Yes if step costs are identical
Time Complexity	$b+b^2+b^3+\ldots+b^d+(b^{d+1}-b)=$ $O(b^{d+1})$
Space Complexity	$O(b^{d+1})$

Exponential time and space complexity make BFS impractical for all but the smallest problems

Uniform-cost Search

- What if step costs are not equal?
- Recall that BFS expands the shallowest node
- Now we expand the node with the lowest path cost
- Uses priority queues

Note: Gets stuck if there is a zero-cost action leading back to the same state.

For completeness and optimality, we require the cost of every step to be ≥ ε

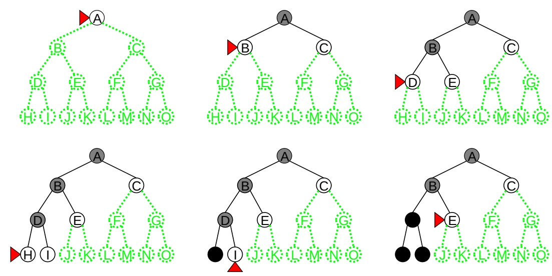
Evaluating Uniform-cost Search

Complete?	Yes provided branching factor is finite and step costs ≥ ε for small positive ε
Optimal?	Yes
Time Complexity	$O(b^{1+\text{floor}(C^*/\varepsilon)})$ where C* is the cost of the optimal solution
Space Complexity	$O(b^{1+\text{floor}(C^*/\varepsilon)})$ where C* is the cost of the optimal solution

Depth-first Search

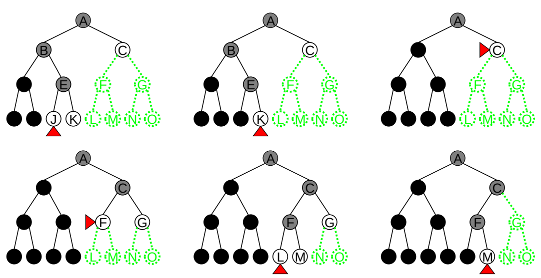
- Expands the deepest node in the current fringe of the search tree
- Implemented with a LIFO queue

Depth-first Search Example



Not yet reached | Expanded nodes on current path | Current node to be expanded
On fringe but unexpanded | Expanded nodes with no descendants in the fringe (can be removed from memory) | Goal state

Depth-first Search Example



Not yet reached | Expanded nodes on current path | Current node to be expanded
On fringe but unexpanded | Expanded nodes with no descendants in the fringe (can be removed from memory) | Goal state

Evaluating Depth-first Search

Complete?	
Optimal?	
Time Complexity	
Space Complexity	

Evaluating Depth-first Search

Complete?	No (Might not terminate if it goes down an infinite path with no solutions)
Optimal?	No (Could expand a much longer path than the optimal one first)
Time Complexity	$O(b^m)$
Space Complexity	O(bm)

31

Depth-limited Search

- Solves infinite path problem by using predetermined depth limit *l*
- Nodes at depth *l* are treated as if they have no successors
- Can use knowledge of the problem to determine *l* (but in general you don't know this in advance)

32

Evaluating Depth-limited Search

Complete?	No (If shallowest goal node beyond depth limit)
Optimal?	No (If depth limit > depth of shallowest goal node and we expand a much longer path than the optimal one first)
Time Complexity	$O(b^l)$
Space Complexity	O(b*l*)

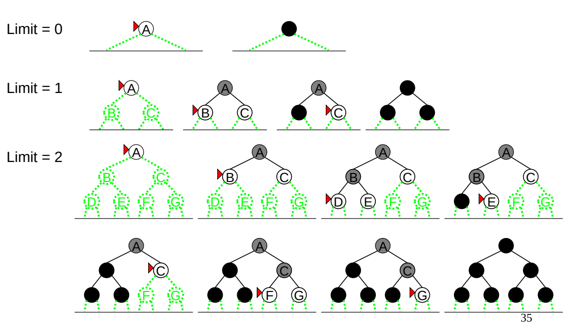
33

Iterative Deepening Depth-first Search

- Do DFS with depth limit 0, 1, 2, … until a goal is found
- Combines benefits of both DFS and BFS

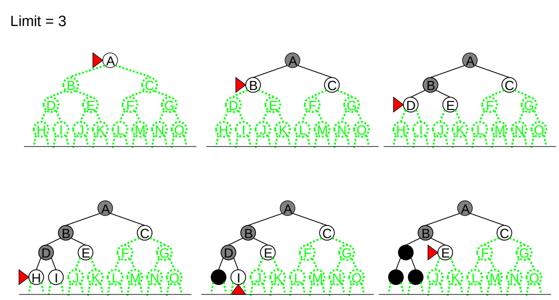
34

Iterative Deepening Depth-first Search Example



35

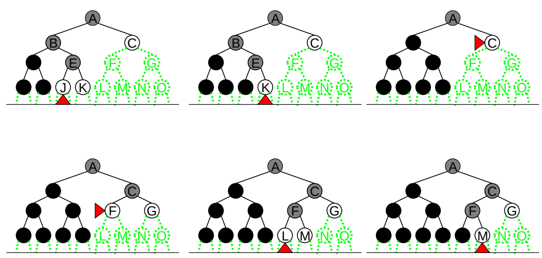
IDDFS Example



36

IDDFS Example

Limit = 3 (Continued)



37

Evaluating Iterative Deepening Depth-first Search

Complete?	
Optimal?	
Time Complexity	
Space Complexity	

38

Evaluating Iterative Deepening Depth-first Search

Complete?	Yes provided branching factor is finite
Optimal?	Yes if the path cost is a nondecreasing function of the depth of the node
Time Complexity	$O(b^d)$
Space Complexity	O(bd)

39

Isn't Iterative Deepening Wasteful?

- Actually, no! Most of the nodes are at the bottom level, doesn't matter that upper levels are generated multiple times.
- To see this, add up the 4th column below:

Depth	**# of nodes**	**# of times generated**	**Total # of nodes generated at depth d**
1	b	d	(d)b
2	b^2	d-1	$(d-1)b^2$
:	:	:	:
d	b^d	1	$(1)b^d$

40

Is Iterative Deepening Wasteful?

Total # of nodes generated by iterative deepening:

$$(d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$$

Total # of nodes generated by BFS:

$$b + b^2 + \ldots + b^d + (b^{d+1}-b) = O(b^{d+1})$$

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

41

Bidirectional Search

- Run one search forward from the initial state
- Run another search backward from the goal
- Stop when the two searches meet in the middle

42

Bidirectional Search

- Needs an efficiently computable Predecessor() function
- What if there are several goal states?
  - Create a new dummy goal state whose predecessors are the actual goal states
- Problematic if no efficient way to generate the set of all goal states and check for them in the forward search eg. "All states that lead to checkmate by move m_1"

43

Evaluating Bidirectional Search

Complete?	Yes provided branching factor is finite and both directions use BFS
Optimal?	Yes if the step costs are all identical and both directions use BFS
Time Complexity	$O(b^{d/2})$
Space Complexity	$O(b^{d/2})$ (At least one search tree must be kept in memory for the membership check)

44

Avoiding Repeated States

- Tradeoff between space and time!
- Need a closed list which stores every expanded node (memory requirements could make search infeasible)
- If the current node matches a node on the closed list, discard it (ie. discard the newly discovered path)
- We'll refer to this algorithm as GRAPH-SEARCH
- Is this optimal? Only for uniform-cost search or breadth-first search with constant step costs.

45

GRAPH-SEARCH

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
  closed ← an empty set
  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 SOLUTION(node)
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

46

Things You Should Know

- How to formalize a search problem
- How BFS, UCS, DFS, DLS, IDS and Bidirectional search work
- Whether the above searches are complete and optimal plus their time and space complexity
- The pros and cons of the above searches

47