

Tutorial Results

- Blank: *nothing* submitted
 - 0: *nothing correct* submitted
 - 0.5: *partially correct* solutions submitted
 - 1: *substantially correct* solutions submitted
-
- Note that receiving a 1 does not mean that you have completely correct answers, check the solutions
 - Results for tutorials submitted online will be up on the website (Results link on left side of page) by 5pm on Tuesdays

Solving Problems by Searching

•Russell and Norvig, Chapter 3

Overview: aims

- know how to formulate a problem as a search problem
- be familiar with concepts like start state, goal state(s), action, successor function, search node, search tree, fringe/queue, node expansion, ...
- be familiar with complexity issues: time and space requirements of algorithms
- be familiar with completeness and optimality of search algorithms
- understand several uninformed search techniques and their properties, e.g. breadth-first, depth-first, and iterative deepening

Overview: topics

- Problem solving
- Formal problem specification
- Example problems
- Uninformed search algorithms

Asymptotic Complexity

- $O()$ notation / Big O notation
- Captures how the algorithm will be affected by the size of the input
- Describes the asymptotic upper bound, worst case
- Describes the efficiency of a particular algorithm
- Space and time complexity of search algorithms

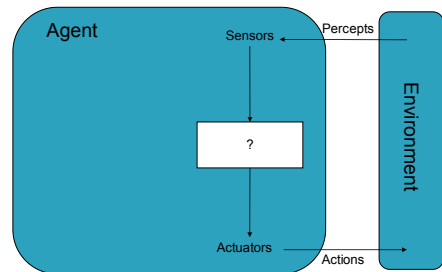
$O()$ notation

- Examples
 - $O(1)$ = constant
 - $O(n)$ = linear (unsorted list)
 - $O(\log n)$ = logarithmic (sorted array, binary search)
 - $O(n^c)$ = polynomial
 - $O(c^n)$ = exponential

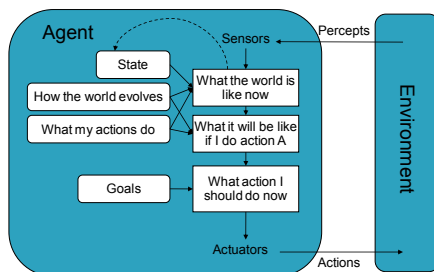
Agents

- Systems that can decide for themselves
- Situated in an environment
- Autonomous action
- Perceives and acts:
 - Percepts – sensory input from the environment
 - Actions – outputs that affect the environment

A Basic Agent



A Goal-Based Agent



A Problem Solving Agent

- Decide what to do by finding sequences of actions that lead to desirable states

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

inputs: *percept*, a percept

static: *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 do**

goal ← FORMULATE-GOAL(*state*)

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

seq ← SEARCH(*problem*)

action ← FIRST(*seq*)

seq ← REST(*seq*)

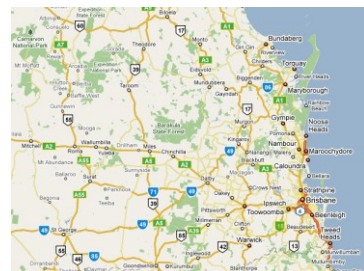
return *action*

Problem solving

- Terminology
 - State – a description of the world
 - Goal - a set of states for which some desirable property holds
 - Actions - cause transitions between world states
- Find out which actions will get it to a goal state: Steps to problem solving
 1. Problem formulation
 - Initial state, what actions and states to consider (a successor function), a goal test, and path costs.
 2. Run search algorithm
 - Examining different possible sequences of actions
 - Returns a solution in the form of an action sequence

Example: Holiday in Queensland (1)

On holiday in Queensland: currently in Roma
Flight leaves tomorrow from Brisbane



Example: Holiday in Queensland (2)

- Formulate goal
Be in Brisbane
- Formulate problem
States: various towns
Actions: drive between towns
- Search: Final solution
Sequence of towns e.g. Roma, Miles, Dalby, Toowoomba, Brisbane

Problem Formulation Example

- States
 - e.g. Towns in Queensland: {"Roma", "Miles", "Brisbane"...}
- Initial state
 - e.g. "Roma"
- Actions and states to consider, as give by a successor function
 - e.g. $S(Roma) = \{ \langle Roma \rightarrow Miles, Miles \rangle, \dots \}$
- Goal test
 - Can be explicit ... e.g. $x = \text{"Brisbane"}$
 - Or implicit ... e.g. $HasInternationalAirport(x)$
- Path cost (additive)
 - e.g. sum of distances, number of towns visited, etc.

The water jug problem (informal)

- There are two unmarked jugs which we know hold 3 litres and 4 litres.
- There is an infinite supply of water.
- The jugs may be filled, emptied, or poured from one into the other
- The aim is to measure out two litres.

Problem Formulation

- A problem is defined by the four items
 1. Initial state
 - The state that the system knows itself to be in
 2. Actions and states to consider, as give by a successor function
 - Given the state x , $S(x)$ returns the states reachable from x by any single action

The initial state and the successor function define the state space of the problem: the set of all states reachable from the initial state by any sequence of actions.
 3. Goal test
 - Determines whether a given state is the goal state
 4. Path cost (additive)
 - The sum of the costs of the individual actions along the path
- A solution is a sequence of action leading from the initial state to the goal state

Selecting a state space

- State space must be abstracted for problem solving
 - Compare "Roma" to the actual trip
 - State may include companions, the radio, what's out the window, the vehicle being used, ... all left out of the state description since they are irrelevant to problem of finding a route to Brisbane (but only we as designers know that)
- Actions must also be abstracted
 - An action has many effects
 - Changing location of vehicle and passengers also take time, uses fuel, generates pollution... we only take into consideration the change in location (but only we know the relevance of changes)
- Consider solution
 - Abstraction is valid... it corresponds to a large number of more detailed paths (none of which will change our perception of the overall/abstract plan)

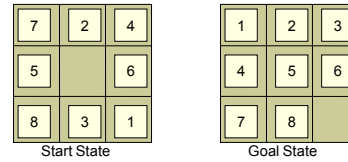
The water jug problem (formal)

- Representation: $(x,y) =$
(volume of water in jug3L, volume of water in jug4L),
 $0 \leq x \leq 3, 0 \leq y \leq 4$.
- Initial state: (0,0)
- Goal states: $(_,2)$ or $(2,_)$
- See if you can work out the actions which would get us from one state to the next state in the Water Jug problem.

Water jug actions

| Actions | Constraints | Meaning |
|----------------------|-----------------------|------------------------------|
| a1 (x,y) → (3,y) | | fill up jug3L |
| a2 (x,y) → (x,4) | | fill up jug4L |
| a3 (x,y) → (0,y) | | empty jug3L |
| a4 (x,y) → (x,0) | | empty jug4L |
| a5 (x,y) → (0,x+y) | $[0 \leq x+y \leq 4]$ | pour all of jug3L into jug4L |
| a6 (x,y) → (x+y,0) | $[0 \leq x+y \leq 3]$ | pour all of jug4L into jug3L |
| a7 (x,y) → (x+y-4,4) | $[x+y > 4]$ | fill up jug4L from jug3L |
| a8 (x,y) → (3,x+y-3) | $[x+y > 3]$ | fill up jug3L from jug4L |

Example: The 8-puzzle



States: The location of each of the 8 tiles and blank

Initial state: Any state

Successor function: Legal states resulting from actions (blank moves *Left, Right, Up, Down*)

Goal state: Current state matches right side above

Path cost: Each step costs 1, path cost is number of steps in the path

Other State Space Problems

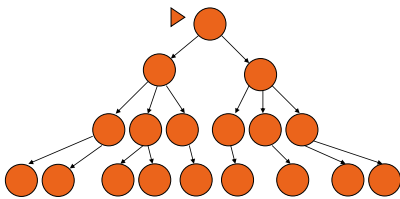
- Exchanging the Knights
- Towers of Hanoi
- Missionaries and Cannibals

Uninformed search algorithms

- Basic Idea
 - Exploration of state space by generating *successors* of expanding states

```
function TREE-SEARCH(problem, strategy) returns a solution or failure
  initialise the search tree using the initial state of problem by putting it in the fringe
  loop do
    if there are no candidates in the fringe then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

Uninformed search algorithms: Basic Idea



Water jug search

```
function TREE-SEARCH(problem, strategy) returns a solution or failure
  initialise the search tree using the initial state of problem

  for the water jug problem – (0,0)

  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy

    How do we move to another state
    for the water jug problem?

    if we fill jug x with water the associated action is
    (x,y) → (3,y)

    ∴ (0,0) → (3,0)

    if the node contains a goal state then return corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

Search Strategies

- A strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
 - Completeness – does it always find a solution if one exists?
 - Time complexity – number of nodes expanded
 - Space complexity – maximum number of nodes in memory
 - Optimality – does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b – maximum branching factor of the search tree
 - d – depth of the least-cost solution
 - m – maximum depth of the state space

Breadth-first Procedure (1)

1. Set L to be a list of the initial nodes in the problem
2. Let n be the first node on L . If L is empty, fail.
3. If n is a goal state, stop and return it and the path from the initial node to n .
4. Otherwise, remove n from L and add to the end of L all of n 's children, labeling each with its path from the initial node. Return to step 2.

The water jug breadth-first search list

- List L is initial nodes
 $L = \langle \text{nil}; (0,0) \rangle$
- $n = \langle \text{nil}; (0,0) \rangle$
- Goal test – the state in n is not goal state
- How do we find all of n 's children?
Use valid operations on the state in n
Can apply
a1 – fill jug3L
a2 – fill jug4L
- Remove n from L and add n 's children to the end of L
 $L = \langle \text{a1}; (3,0) \rangle, \langle \text{a2}; (0,4) \rangle$

Uninformed search strategies

- Uninformed strategies use only the information available in the problem definition
 - Breadth-first search
 - Depth-first search
 - Depth-limited search
 - Iterative deepening search

Breadth-first Procedure (2)

- Tree is examined from the top down
- Every node at depth d is examined before any node at depth $d+1$ is
- Can implement a breadth-first search by adding new nodes to the end of the list maintained.

The water jug breadth-first search list

- Remove n from L and add n 's children to the end of L
 $L = \langle \text{a1}; (3,0) \rangle, \langle \text{a2}; (0,4) \rangle$
- $n = \langle \text{a1}; (3,0) \rangle$
- Goal test – the state in n is not goal state
- How do we find all of n 's children?
Use valid operations on the state in n
Can apply
a2 – fill jug4L
a3 – empty jug3L
a4 – transfer all contents of jug3L to jug4L
- Remove n from L and add n 's children to the end of L
 $L = \langle \text{a2}; (0,4) \rangle, \langle \text{a2}; (3,4) \rangle, \langle \text{a3}; (0,0) \rangle, \langle \text{a4}; (0,3) \rangle$

The water jug breadth-first search list

- Remove n from L and add n 's children to the end of L
 $L = \{ \langle a2: (0,4) \rangle, \langle a2: (3,4) \rangle, \langle a3: (0,0) \rangle, \langle a4: (0,3) \rangle \}$
 - $n = \langle a2: (0,4) \rangle$
 - Goal test – the state in n is not goal state
 - How do we find all of n 's children?
 Use valid operations on the state in n
 - Can apply
 - a1 – fill jug3L
 - a4 – empty jug4L
 - a8 – add jug4L to jug3L until jug3L is full
 - Remove n from L and add n 's children to the end of L
 $L = \{ \langle a2: (3,4) \rangle, \langle a3: (0,0) \rangle, \langle a4: (0,3) \rangle, \langle a1: (3,4) \rangle, \langle a4: (0,0) \rangle, \langle a8: (3,1) \rangle \}$
- and so on...

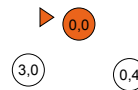
Tree representation of breadth-first search

- Expand shallowest unexpanded node
 - Root node is expanded first
 - Then all nodes generated by the root node
 - Then all nodes generated by these nodes
 - ...
- Tree is examined from top down
 - Every node at depth d is examined before any node at depth $d + 1$
- Implementation:
 - Fringe is a FIFO queue, i.e., new successors go at end

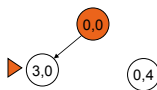
Tree representation of breadth-first search



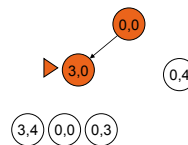
Tree representation of breadth-first search



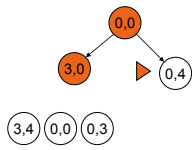
Tree representation of breadth-first search



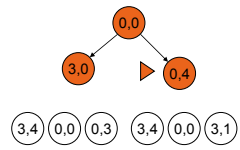
Tree representation of breadth-first search



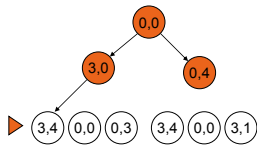
Tree representation of breadth-first search



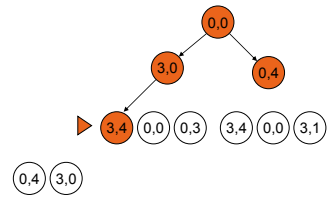
Tree representation of breadth-first search



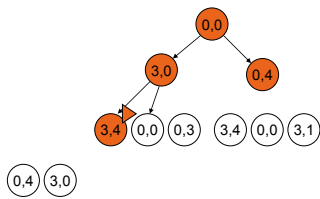
Tree representation of breadth-first search



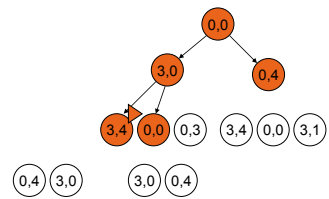
Tree representation of breadth-first search



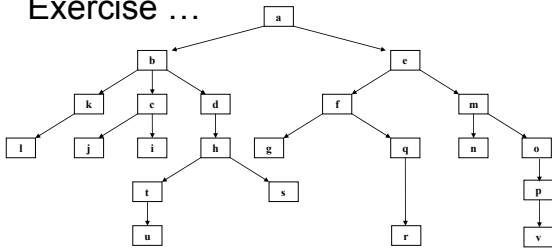
Tree representation of breadth-first search



Tree representation of breadth-first search



Exercise ...

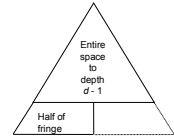


List (in order) the nodes that would be visited if a *breadth-first* search were performed.

abekcdfmljihgqnotsrpuv

Properties of breadth-first search

- **Completeness**
 - yes (if b is finite)
- **Time**
 - $1 + b + b^2 + b^3 + \dots + b^d = O(b^{d+1})$
 - i.e. exponential in d
- **Space**
 - $O(b^d)$
 - keeps every node in memory
- **Optimal**
 - yes - if cost = 1 per step
 - not guaranteed optimal otherwise



Space is the big problem – can easily generate nodes at 10MB/sec

Depth-first Search Procedure (1)

1. Set L to be a list of the initial nodes in the problem.
2. Let n be the first node on L . If L is empty, fail
3. If n is a goal state, stop and return it and the path from the initial node to n .
4. Otherwise, remove n from L and add to the front of L all of n 's children, labeling each with its path from the initial node. Return to step 2.

Depth-first Search Procedure (2)

- Expanding a search tree so that the terminal nodes are examined from left to right
- You always explore a child of the most recently expanded node
- If this node has no children, the procedure backs up a minimum amount before choosing another node to examine.
- Can be implemented by pushing the children of a given node onto the front of list L and always choosing the first node on L as the one to expand.

The water jug depth-first search list

- List L is initial nodes
 $L = \langle \text{nil}; (0,0) \rangle$
- $n = \langle \text{nil}; (0,0) \rangle$
- Goal test – the state in n is not goal state
- How do we find all of n 's children?
Use valid operations on the state in n
Can apply
a1 – fill jug3L
a2 – fill jug4L
- Remove n from L and add n 's children to the **front** of L
 $L = \langle \text{a1}; (3,0); \text{a2}; (0,4) \rangle$

The water jug depth-first search list

- Remove n from L and add n 's children to the front of L
 $L = \langle \text{a1}; (3,0); \text{a2}; (0,4) \rangle$
- $n = \langle \text{a1}; (3,0) \rangle$
- Goal test – the state in n is not goal state
- How do we find all of n 's children?
Use valid operations
Can apply
a2 – fill jug4L
a3 – empty jug3L
a4 – transfer all contents of jug3L to jug4L
- Remove n from L and add n 's children to the **front** of L
 $L = \langle \text{a2}; (3,4); \text{a3}; (0,0); \text{a4}; (0,3); \text{a2}; (0,4) \rangle$

The water jug depth-first search list

- Remove n from L and add n 's children to the front of L
 $L = \langle a2:(3,4), a3:(0,0), a4:(0,3), a2:(0,4) \rangle$
 - $n = a2:(3,4)$
 - Goal test – the state in n is not goal state
 - How do we find all of n 's children?
 Use valid operations
 - Can apply
 $a3$ – empty jug3L
 $a4$ – empty jug4L
 - Remove n from L and add n 's children to the **front** of L
 $L = \langle a3:(0,4), a4:(0,3), a3:(0,0), a4:(0,3), a2:(0,4) \rangle$
- and so on...

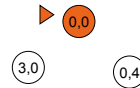
Tree representation of depth-first search

- Expand deepest unexpanded node
 - Depth-first search always expands one of the nodes at the deepest level of the tree
 - Only when the search hits a dead-end (a non-goal node with no expansion) does the search go back up and expand nodes at shallower levels
- Expanding a search tree so that the terminal nodes are examined from left to right
 - Always explore a child of the most recently expanded node
 - If no children, back up a minimum amount then choose another node
- Implementation:
 - Fringe = LIFO queue, i.e. put successors at front

Tree representation of depth-first search



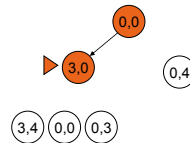
Tree representation of depth-first search



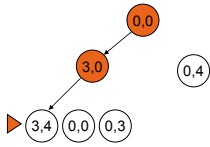
Tree representation of depth-first search



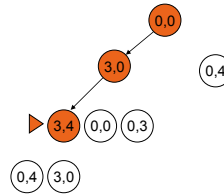
Tree representation of depth-first search



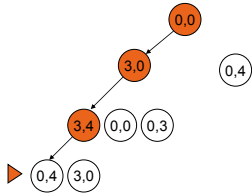
Tree representation of depth-first search



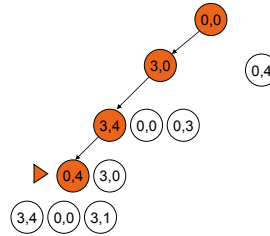
Tree representation of depth-first search



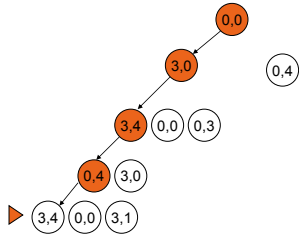
Tree representation of depth-first search



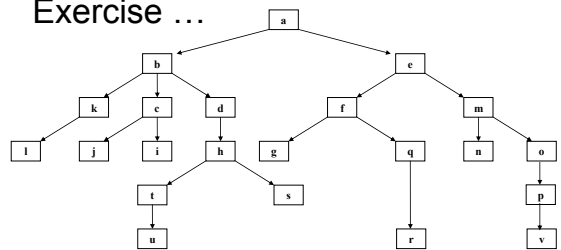
Tree representation of depth-first search



Tree representation of depth-first search

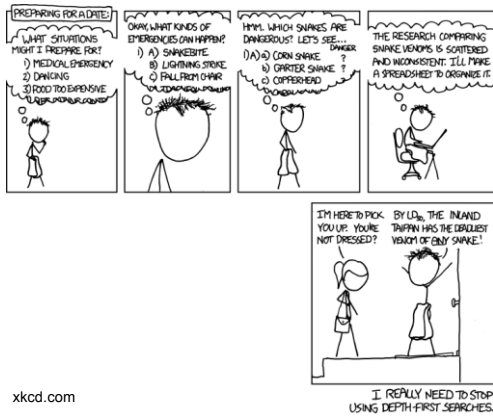


Exercise ...



List (in order) the nodes that would be visited if a *depth-first* search were performed.

abklcjdhutusefgqrmnopv



xkcd.com

Properties of depth-first search

- Completeness
 - no: fails in infinite-depth spaces, spaces with loops
 - modify to avoid repeated states along path
 - \Rightarrow complete in finite spaces
- Time
 - $O(b^m)$
 - Terrible if m is much larger than d
 - But if solutions are dense, may be much faster than breadth-first
- Space
 - $O(bm)$
 - Linear space
- Optimal
 - no

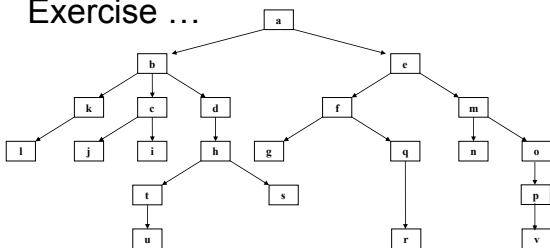
Depth-first / breadth-first comparison

- Breadth-first
 - requires more memory
 - may look at a large portion of the tree before finding the solution
 - guarantees a minimal-cost solution
- Depth-first
 - more efficient in time and space
 - may get trapped in (infinite?) blind-alleys, or loops
 - order of rules is important
 - may not find the shallowest solution

Depth-limited search

- = depth-first search with depth limit l
 - i.e. nodes at depth l have no successors
- The root node has a depth of 0

Exercise ...



List (in order) the nodes that would be visited if a *depth-limited* search were performed to a depth of 3.

abklcjdhfgqmmo

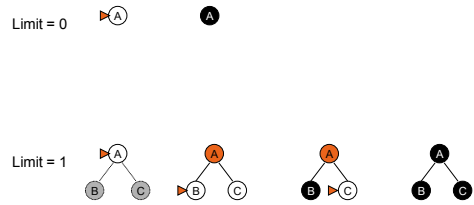
Properties of depth-limited search

- Completeness
 - Not if $l < d$
- Time
 - $O(b^l)$
- Space
 - $O(bl)$
- Optimal
 - Not if $l > d$

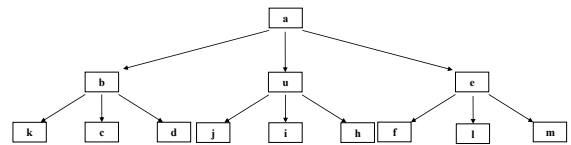
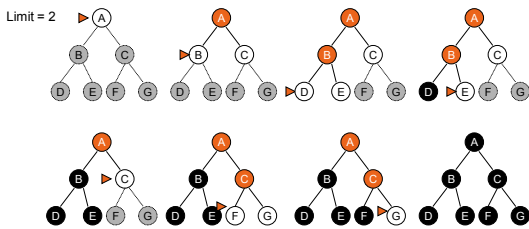
Iterative deepening search

- Iterative deepening depth-first search
- Gradually increase the depth limit used in a depth-limited search

Iterative deepening search



Iterative deepening search



List (in order) the nodes that would be visited if an *iterative deepening search* were performed

a abue abkcduijheflm

Properties of iterative deepening search

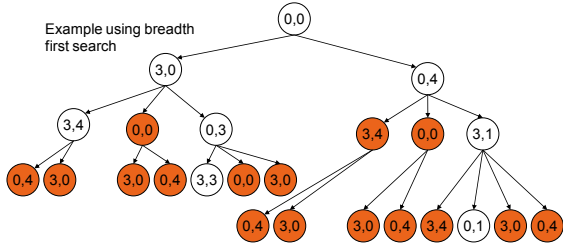
- Completeness
 - yes
- Time
 - $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + 2b^{d-1} + b^d = O(b^d)$
- Space
 - $O(bd)$
- Optimal
 - yes, if step cost = 1
 - not guaranteed otherwise

Comparing uninformed search strategies

| | Breadth-First | Depth-First | Depth-Limited | Iterative Deepening |
|-----------|----------------|-------------|---------------|---------------------|
| Complete? | • Yes | • No | • No | • Yes |
| Time | • $O(b^{d+1})$ | • $O(b^m)$ | • $O(b^l)$ | • $O(b^d)$ |
| Space | • $O(b^{d+1})$ | • $O(bm)$ | • $O(bl)$ | • $O(bd)$ |
| Optimal? | • Yes | • No | • No | • Yes |

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



Dealing with repeated states

- Do not return to the state you just came from
 - Expand function doesn't allow generation of a successor that is the same state as the parent node
- Do not create paths with cycles in them
 - Expand function doesn't allow generation of a successor that is the same state as any of the ancestor nodes
- Do not generate any state that was ever generated before
 - Expand function doesn't allow generation of a successor that is the same as any existing node

Summary

- Problem formulation requires abstracting away real-world details to define a state space that can be feasibly explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other informed algorithms

