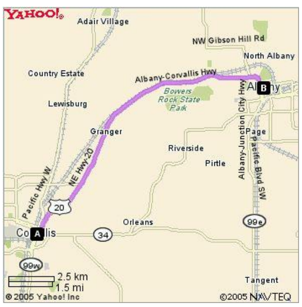
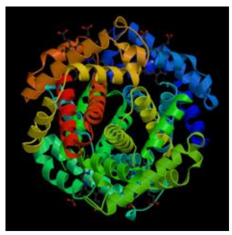
# CS 331: Artificial Intelligence
# Uninformed Search

1

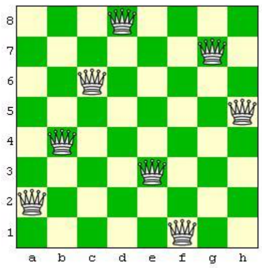
## Real World Search Problems

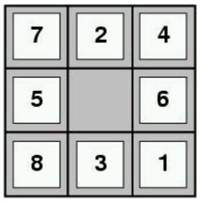


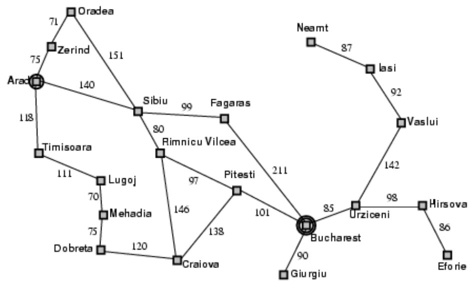



2

# Simpler Search Problems






# Assumptions About Our Environment

- Fully Observable
- Deterministic
- Sequential
- Static
- Discrete
- Single-agent

# Search Problem Formulation

Components of a search problem:

1. A state space S
2. An initial state I ∈ S
3. A non-empty set of goal states G ⊆ S (or a goal test function that tests whether a state is a goal)
4. A description of the actions available in state s, ***actions(s)***.
5. A transition model describing the result of action a applied in state s: ***result(s,a)***
6. A cost function ***cost(s,a,s')*** which returns the non-negative one-step cost of travelling from state ***s*** to ***s***' by applying ***a***. The cost function is only defined if ***s'*** is a successor state of ***s***.



# Search Problem Formulation

Components of a search problem:

1. A state space S
2. An initial state I ∈ S
3. A non-empty set of goal states G ⊆ S (or a goal test function that tests whether a state is a goal)
4. When actions are deterministic, we can combine ***actions(s)*** and ***result(s,a)*** into a successor function ***succ(s)*** that returns the states reachable in one step from s.
5. 
6. A cost function ***cost(s,a,s')*** which returns the non-negative one-step cost of travelling from state ***s*** to ***s***' by applying ***a***. The cost function is only defined if ***s'*** is a successor state of ***s***.

# Search Problem Formulation

A search problem has 5 components:

1. A state space S
2. An initial state I ∈ S
3. A non-empty set of goal states G ⊆ S (or a goal test function that tests whether a state is a goal)
4.
5.
6.

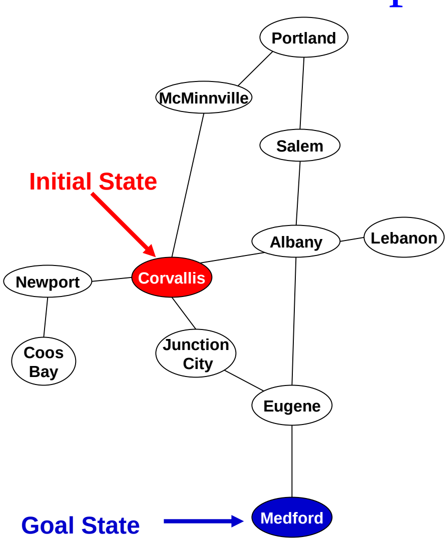
When actions are deterministic, we can combine ***actions(s)*** and ***result(s,a)*** into a successor function ***succ(s)*** that returns the states reachable in one step from s.

Then we can also abbreviate ***cost(s,a,s')*** as ***cost(s,s')***.

state ***s*** to ***s'*** by applying ***a***. The cost function is only defined if ***s'*** is a successor state of ***s***.

7

# Example: Oregon




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

I = {Corvallis}

G={Medford}

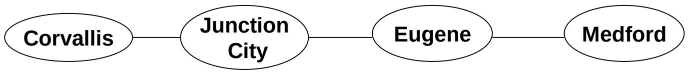
Succ(Corvallis)={Albany, Newport, McMinnville, Junction City}

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

8

# 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)

9

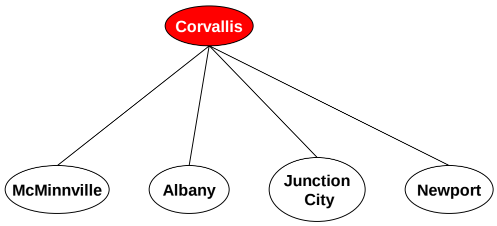
# Search Tree


Start with Initial State.

Is initial state the goal?

10

# Search Tree

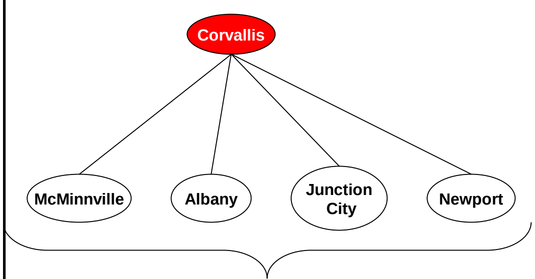



Is initial state the goal?

- Yes, return solution
- No, “expand” initial state by applying Successor() function

11

# Search Tree




Apply Successor() function

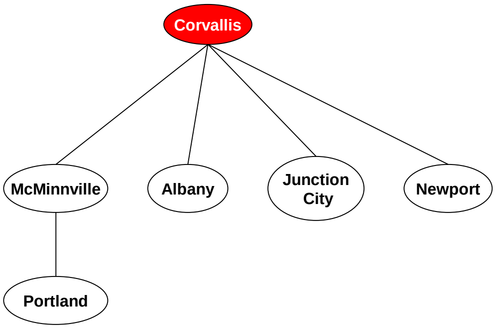
These nodes have not been expanded yet. Call them the frontier. We’ll put them in a queue.

Queue

| |
|---|
| McMinnville |
| Albany |
| Junction City |
| Newport |

12

# Search Tree

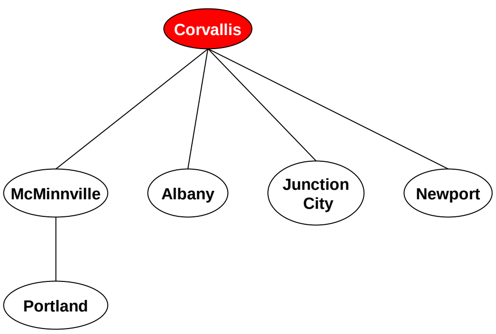



| Queue |
|---|
| Albany |
| Junction City |
| Newport |
| Portland |

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.

13

# 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*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **if** the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

15

# Tree-Search Pseudocode

**function** TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **if** the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

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

16

## Tree-Search Pseudocode

**function** TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **if** the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Note: The expand function also keeps backpointers to the parents of the expanded node so a solution path can be reconstructed.



## Avoiding Repeated States

- Tradeoff between space and time!
- Need an "explored" list which stores every expanded node (memory requirements could make search infeasible)
- If the current node matches a node in the explored list, discard it (i.e., discard the newly discovered path)
- We'll refer to this algorithm as GRAPH-SEARCH

## GRAPH-SEARCH

**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  ***initialize the explored set to be empty***
  **loop do**
    **if** the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    **if** the node contains a goal state **then return** the corresponding solution
    ***add the node to the explored set***
    expand the chosen node, adding the resulting nodes to the frontier
      ***only if not in the frontier or explored set***

19

## 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

20

## 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?



## 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

## Uninformed Search Algorithms

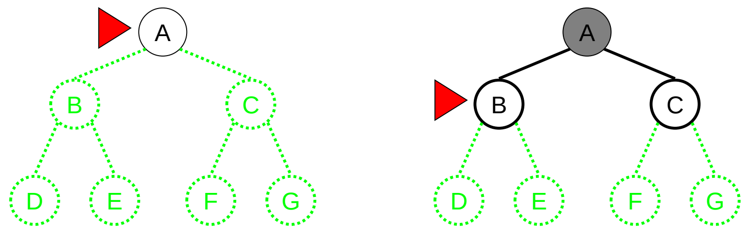
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative Deepening Depth-first Search
- Bidirectional search



## 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




Not yet reached

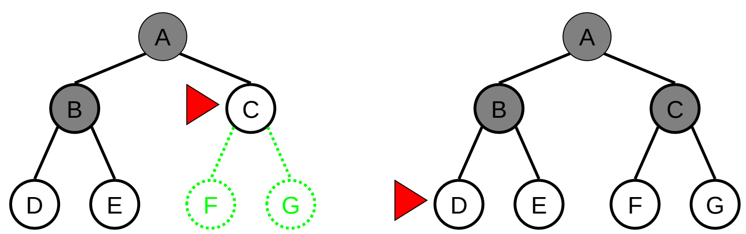
Closed (expanded) nodes

Open nodes (on the frontier)

Current node to be expanded

25

# Breadth First Search Example




Not yet reached

Closed (expanded) nodes

Open nodes (on the frontier)

Current node to be expanded

26

## Evaluating BFS

| | |
|---|---|
| Complete? | |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |



## Evaluating BFS

| | |
|---|---|
| Complete? | Yes provided branching factor is finite |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

## Evaluating BFS

| | |
|---|---|
| Complete? | Yes provided branching factor is finite |
| Optimal? | Yes if step costs are identical |
| 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 | |

# 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

31

# 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.

So, assume the cost of every step to be $\geq \varepsilon$

32

## Evaluating Uniform-cost Search

| | |
|---|---|
| Complete? | |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

33

## Evaluating Uniform-cost Search

| | |
|---|---|
| Complete? | Yes provided branching factor is finite and step costs $\geq \varepsilon$ for small positive $\varepsilon$ |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

34

## Evaluating Uniform-cost Search

| | |
|---|---|
| Complete? | Yes provided branching factor is finite and step costs ≥ ε for small positive ε |
| Optimal? | Yes |
| Time Complexity | |
| Space Complexity | |

35

## Evaluating Uniform-cost Search

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

36

# Evaluating Uniform-cost Search

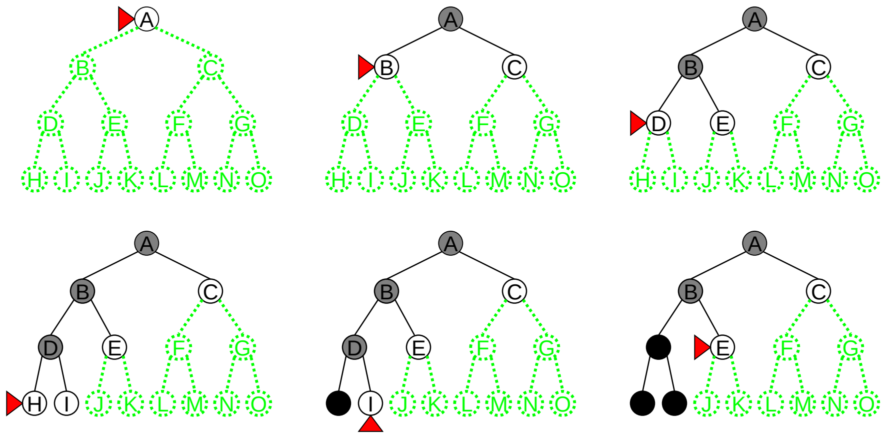
| | |
|---|---|
| Complete? | Yes provided branching factor is finite and step costs $\geq \varepsilon$ for small positive $\varepsilon$ |
| Optimal? | Yes |
| Time Complexity | $O(b^{1+floor(C^*/\varepsilon)})$ where $C^*$ is the cost of the optimal solution |
| Space Complexity | $O(b^{1+floor(C^*/\varepsilon)})$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



# Depth-first Search

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

# Depth-first Search Example




Not yet reached

On frontier but unexpanded

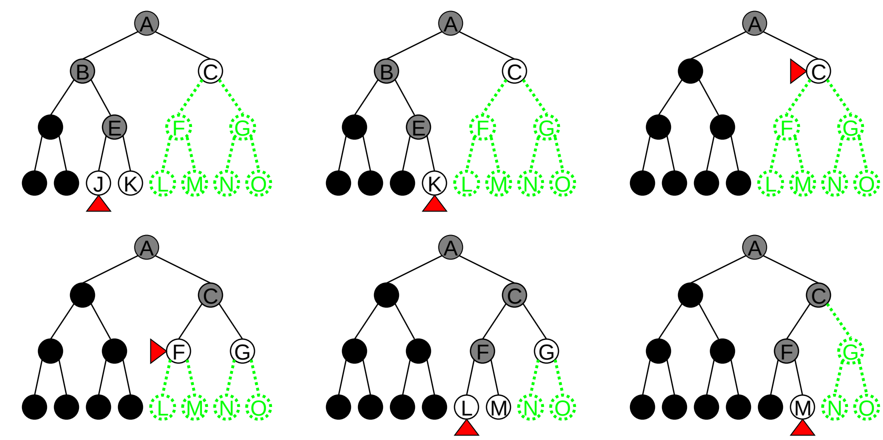
Expanded nodes on current path

Expanded nodes with no descendants in the frontier (can be removed from memory)

Current node to be expanded

Ⓜ Goal state


# Depth-first Search Example



Not yet reached

On frontier but unexpanded

Expanded nodes on current path

Expanded nodes with no descendants in the frontier (can be removed from memory)

Current node to be expanded

Ⓜ Goal state

## Evaluating Depth-first Search

| | |
|---|---|
| Complete? | |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

41

## Evaluating Depth-first Search

| | |
|---|---|
| Complete? | Yes for finite m. No if there is an infinitely long path with no solutions. |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

42

## Evaluating Depth-first Search

| | |
|---|---|
| Complete? | Yes for finite m. No if there is an infinitely long path with no solutions. |
| Optimal? | No (Could expand a much longer path than the optimal one first) |
| Time Complexity | |
| Space Complexity | |



## Evaluating Depth-first Search

| | |
|---|---|
| Complete? | Yes for finite m. No if there is an infinitely long path with no solutions. |
| Optimal? | No (Could expand a much longer path than the optimal one first) |
| Time Complexity | $O(b^m)$ |
| Space Complexity | |

# Evaluating Depth-first Search

| | |
|---|---|
| Complete? | Yes for finite m. No if there is an infinitely long 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) |

45

# 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)

46

## Evaluating Depth-limited Search

| | |
|---|---|
| Complete? | |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

47

## Evaluating Depth-limited Search

| | |
|---|---|
| Complete? | No (If shallowest goal node beyond depth limit) |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

48

# 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 | |
| Space Complexity | |

49

# 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 | |

50

## 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 | $O(bl)$ |

51

## 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

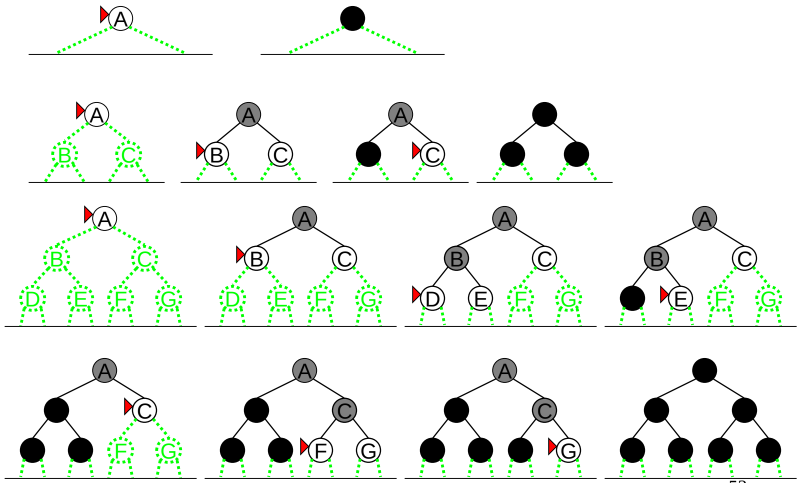
52

# Iterative Deepening Depth-first Search Example

Limit = 0

Limit = 1

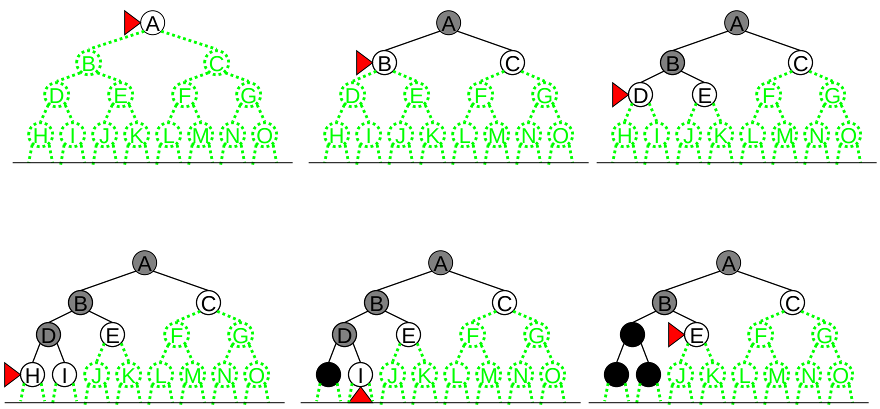
Limit = 2




53

# IDDFS Example

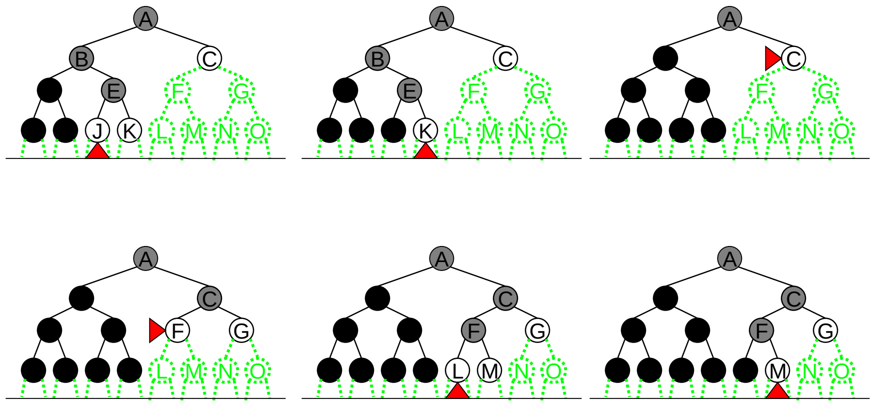
Limit = 3




54

# IDDFS Example

Limit = 3 (Continued)




55

# Evaluating Iterative Deepening Depth-first Search

| | |
|---|---|
| Complete? | |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

56

## Evaluating Iterative Deepening Depth-first Search

| | |
|---|---|
| Complete? | Yes provided branching factor is finite |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

57

## 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 | |
| Space Complexity | |

58

## 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 | $(d)b + (d-1)\, b^2 + \ldots + (1)\, b^d = O(b^d)$ |
| Space Complexity | |

59

## 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 | $O(bd)$ |

60

## 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 | $(1)b^d$ |



## Is Iterative Deepening Wasteful?

Total # of nodes generated by iterative deepening:

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

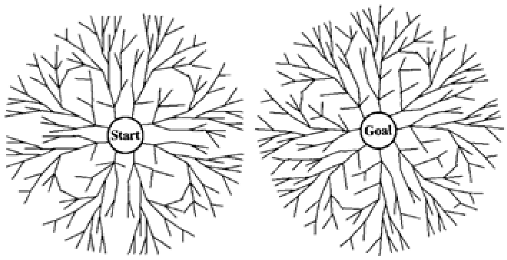
Total # of nodes generated by BFS:

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

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

# 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






# 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
- Difficult when the goal is an abstract description like “no queen attacks another queen”

## Evaluating Bidirectional Search

| | |
|---|---|
| Complete? | |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

65

## Evaluating Bidirectional Search

| | |
|---|---|
| Complete? | Yes provided branching factor is finite and both directions use BFS |
| Optimal? | |
| Time Complexity | |
| Space Complexity | |

66

## 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 | |
| Space Complexity | |

67

## 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 | |

68

## 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) |



## 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