Uninformed Search

Chapter 3.1 – 3.5

Building Goal-Based Agents

What are the key questions needing to be addressed?

- **What goal does the agent need to achieve?**
- **What knowledge does the agent need?**
- **What actions does the agent need to do?**

Building Goal-Based Agents

What goal does the agent need to achieve?

- **How do you describe the goal?**
  - as a task to be accomplished
  - as a situation to be reached
  - as a set of properties to be acquired
- **How do you know when the goal is reached?**
  - with a goal test that defines what it means to have achieved/satisfied the goal
- ***Determining the goal is difficult and is usually left to the system designer or user to specify.***

Building Goal-Based Agents

What knowledge does the agent need?

- **The info. needs to be**
  - sufficient to describe all relevant aspects to reaching the goal
  - adequate to describe the world state/situation
- ***We'll use a closed world assumption:***

  All necessary information about a problem domain is accessible in each percept so that each state is a complete description of the world. There is no incomplete information at any point in time.

Building Goal-Based Agents

- **How should the agent's knowledge be represented?**
- **knowledge representation problem:**
  - What information from the raw percept is relevant?
  - How to represent domain knowledge later…

∗ ***Determining what to represent is difficult and is usually left to the system designer to specify.***

5

Building Goal-Based Agents

What actions does the agent need to do?

- **The set of actions/events needs to be**
  - decomposed into primitive steps
    that are discrete, i.e. treated as instantaneous
  - quantified to fully describe initial and final states
    with no uncertainty
  - sufficient to describe all necessary changes

∗ ***The number of actions needed depends on how the world states are represented.***

6

Building Goal-Based Agents

What actions does the agent need to do?

- **Given:**
  - an action (operator/move)
  - a description of the current state of the world
- **Action completely specifies:**
  - if that action can be applied (applicable? legal?)
  - what the exact state of the world will be after the action is performed in the current world (no "history" information needed to compute the new world state).

7

Formalizing Search in a State Space

- **A state space is a *graph*: (V, E)**
  - V is a set of nodes (vertexes)
  - E is a set of arcs (edges)
    each arc is directed from one node to another node
- **Each node is a data structure that contains:**
  - a state description
  - other information such as:
    - link to parent node
    - name of operator that generated this node (from its parent)
    - other bookkeeping data

8

Formalizing Search in a State Space

- **Each arc corresponds to one of the operators:**
  - when the operator is applied to the state associated with the arc's source node
  - then the resulting state is the state associated with the arc's destination node
- **Each arc has a fixed, positive cost:**
  - corresponds to the cost of the operator

9

Formalizing Search in a State Space

- **Each node has a set of successor nodes:**
  - corresponds to all of the legal operators that can be applied at the source node's state
- **Expanding a node means:**
  - generate all of the successor nodes
  - add them and their associated arcs to the state-space search tree

10

Formalizing Search in a State Space

- **One or more nodes are designated as start nodes**
- **A goal test is applied to a node's state to determine if it is a goal node**
- **A solution is a sequence of operators associated with a path in the state space from a start to a goal node:**
  - just the goal state (e.g. cryptarithmetic)
  - a path from start to goal state (e.g. 8-puzzle)
- **The cost of a solution is the sum of the arc costs on the solution path**

11

Formalizing Search in a State Space

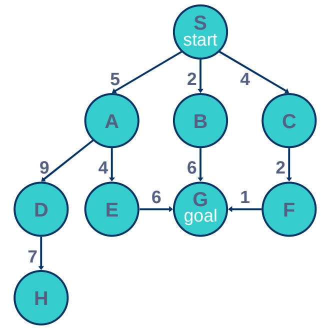
- **State-space search** is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph to include a goal node
  - initially $V=\{S\}$, where S is the start node
  - then S is *expanded*, its successors are generated and those nodes are added to V and the associated arcs are added to E
  - this process continues until a goal node is found

12

Formalizing Search in a State Space

- **Each node implicitly or explicitly represents**
  - a partial solution path from the start node to the given node
  - cost of the partial solution path
- **From this node there are:**
  - many possible paths that have this partial path as a prefix
  - many possible solutions

State Space Diagram



The size of a problem is usually described in terms of the number of possible states:

Tic-Tac-Toe: 3^9 states
Checkers: 10^{40} states
Rubik's Cube: 10^{19} states
Chess: 10^{120} states

State-Space Search Algorithm

```
//Note: this algorithm doesn't detect loops in the state space

Node generalSearch (Problem problem, List fringe) {
  fringe.add(new Node(problem.getStartState()));
  while (true) {
    if (fringe.isEmpty()) return new Node("failure");
    Node node = fringe.remove();
    if (problem.isGoal(node.getState())) return node;
    fringe.add(expand node given problem operators);

   //expand: generates all of this node's children nodes
   //Note: the goal test is NOT done when nodes are generated
  }
}
```


Key Issues of State-Space Search Algorithm

- **Search process constructs a "search tree"**
  - root is the start state
  - leaf nodes are:
    - unexpanded nodes (in the nodes list)
    - "dead ends" (nodes that aren't goals and have no successors because no operators were applicable)
    - goal node is last leaf node found
- **Loops in graph may cause "search tree" to be infinite even if state space is small**
- **Changing the nodes "list" to different data structures leads to different search strategies!**

Evaluating Search Strategies

- **Completeness**
  If a solution exists, will it be found?
  – a complete algorithm will find a solution

- **Optimality/Admissibility**
  If a solution is found, is it guaranteed to be optimal?
  – an admissible algorithm will find a solution with minimum cost

19

2/6/2006

Evaluating Search Strategies

- **Time Complexity**
  How long does it take to find a solution?
  – measured for worst or average case
  – measured in number of nodes expanded/tested

- **Space Complexity**
  How much space is used by the algorithm?
  – measured in terms of the maximum size of the *nodes list* during the search

20

2/6/2006

Uninformed Search Strategies

Uninformed Search: strategies that order nodes without using any domain specific information

- **BFS: breadth-first search**
  ∗ *queue (FIFO) used for the nodes "list"*
  – remove from front, add to back
- **DFS: depth-first search**
  ∗ *stack (LIFO) used for the nodes "list"*
  – remove from front, add to front

21

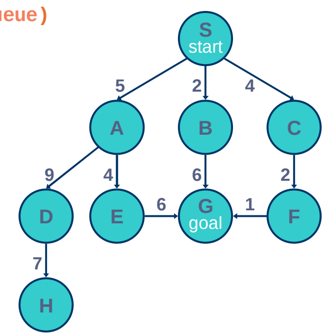
2/6/2006

Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 0, expanded: 0

expnd. node	nodes list
	{S}



22

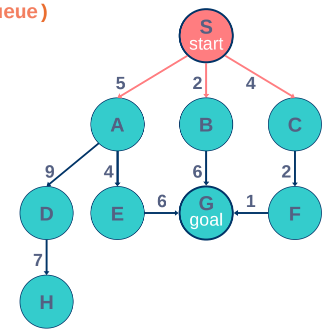
2/6/2006

Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 1, expanded: 1

expnd. node	nodes list
	{S}
S not goal	{A,B,C}

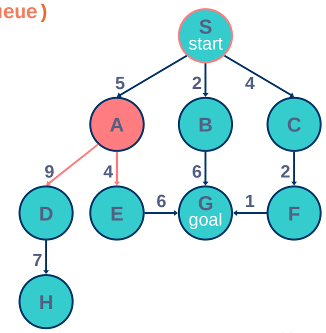


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 2, expanded: 2

expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{B,C,D,E}

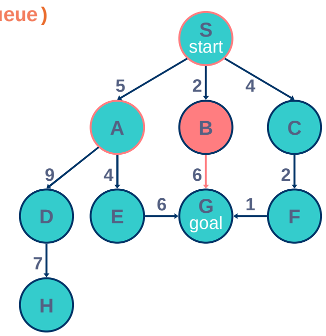


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 3, expanded: 3

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}

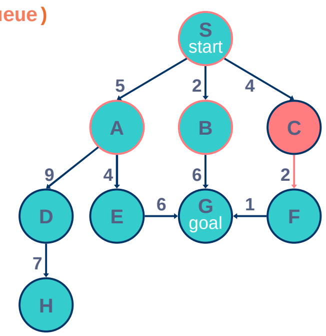


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 4, expanded: 4

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}

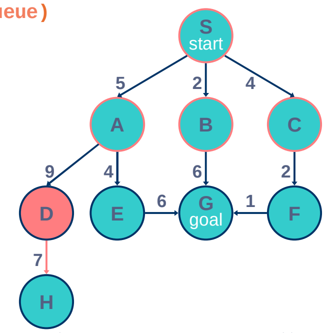


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 5, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D not goal	{E,G,F,H}

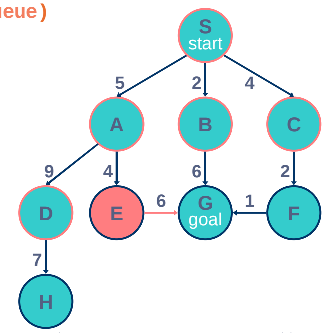


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 6, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E not goal	{G,F,H,G}

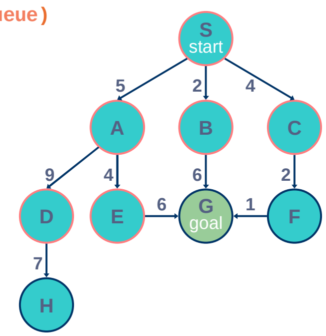


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G goal	{F,H,G} no expand

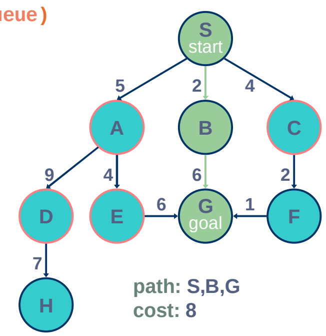


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}



path: S,B,G
cost: 8

Breadth-First Search (BFS)

- **Complete**
- **Optimal/Admissible**
  - if all operators (i.e. arcs) have the same cost
  - otherwise, not optimal but does guarantee finding solution of shortest length (i.e. fewest arcs)

Breadth-First Search (BFS)

- **Time and space complexity: $O(b^d)$ (i.e., exponential)**
  - d is the depth of the solution
  - b: the branching factor at each non-leaf node

∗ ***Will take a long time to find solutions with a large number of steps because must look at all shorter length possibilities first***

Breadth-First Search (BFS)

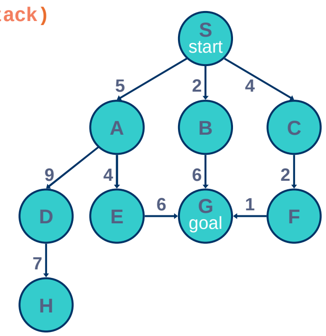
- **A complete search tree has a total of nodes:**
  $1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1) / (b-1)$
  - d: the tree's depth
  - b: the branching factor at each non-leaf node
- **For example: $d = 12$, $b = 10$**
  $1 + 10 + 100 + ... + 10^{12} = (10^{13} - 1)/9 = O(10^{12})$
  - If BFS expands 1000 nodes/sec and each node uses 100 bytes of storage, then BFS will take 35 years to run in the worst case, and it will use 111 terabytes of memory!

Depth-First Search (DFS)

```
generalSearch(problem, stack)
```

\# of nodes tested: 0, expanded: 0

expnd. node	nodes list
	{S}

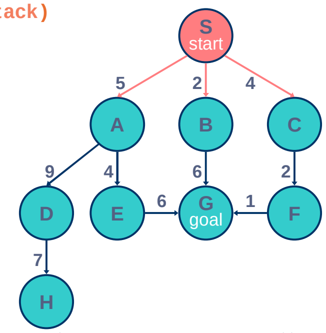


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 1, expanded: 1

expnd. node	nodes list
	{S}
S not goal	{A,B,C}

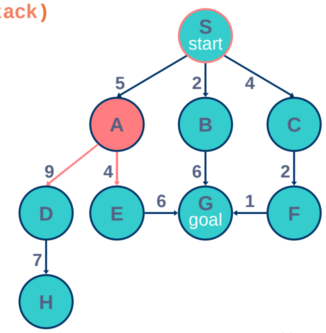


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 2, expanded: 2

expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{D,E,B,C}

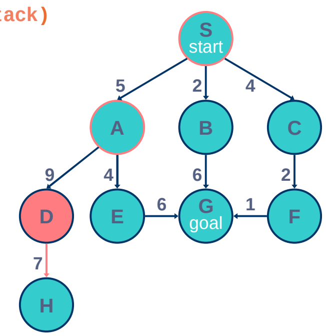


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 3, expanded: 3

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D not goal	{H,E,B,C}

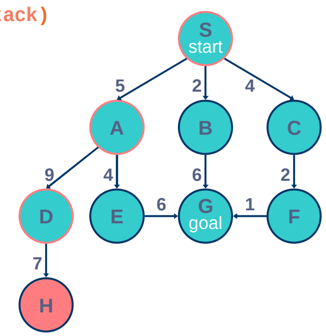


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 4, expanded: 4

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H not goal	{E,B,C}

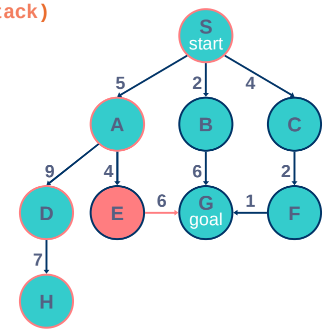


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 5, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E not goal	{G,B,C}



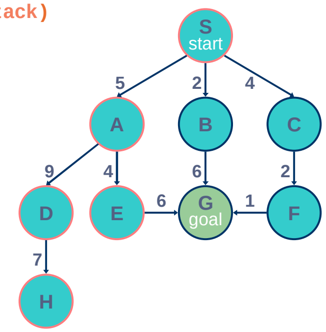
39

Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G goal	{B,C} no expand



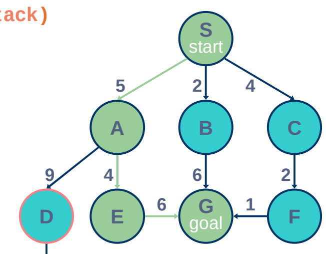
40

Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G	{B,C}



path: S,A,E,G
cost: 15

41

Depth-First Search (DFS)

- **May not terminate without a depth bound i.e., cutting off search below a fixed depth, D**
- **Not complete**
  - with or without cycle detection
  - and, with or without a depth cutoff
- **Not optimal/admissible**

∗ ***Can find long solutions quickly if lucky***

42

Depth-First Search (DFS)

- **Time complexity:** $O(b^d)$ **exponential**
  Space complexity: $O(bd)$ **linear**
  - d is the depth of the solution
  - b: the branching factor at each non-leaf node
- **Does chronological backtracking**
  - when search hits a dead end, backs up one level at a time
  - problematic if the mistake occurs because of a bad operator choice near the top of search tree

Uninformed Search Strategies

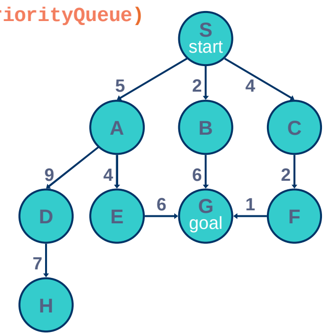
- **UCS: uniform-cost search**
  - ∗ *priority queue used to order nodes, sort by path cost*
  - let $g(n) =$ cost of path from start node s to current node n
  - sort nodes by increasing value of g
  - only uninformed search that worries about costs

Uniform-Cost Search (UCS)

```
generalSearch(problem, priorityQueue)
```

\# of nodes tested: 0, expanded: 0

expnd. node	nodes list
	{S}

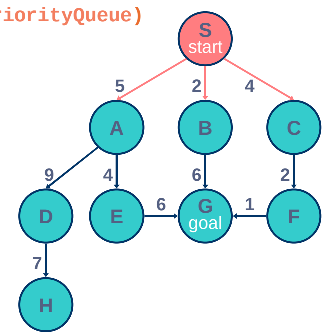


Uniform-Cost Search (UCS)

```
generalSearch(problem, priorityQueue)
```

\# of nodes tested: 1, expanded: 1

expnd. node	nodes list
	{S:0}
S not goal	{B:2,C:4,A:5}

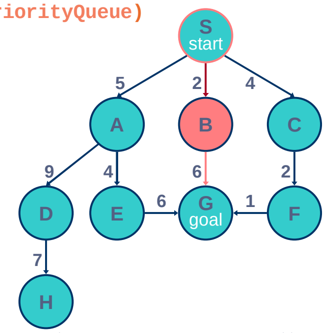


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 2, expanded: 2

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}

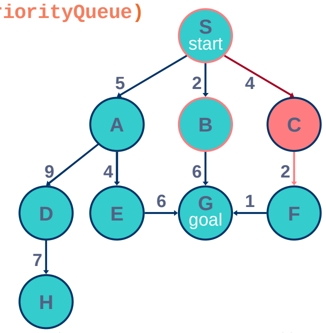


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 3, expanded: 3

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}

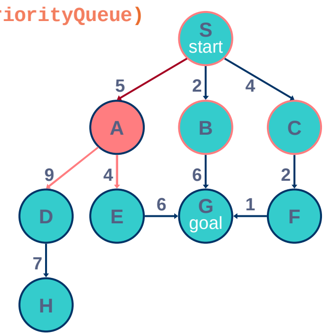


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 4, expanded: 4

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4, D:5+9}

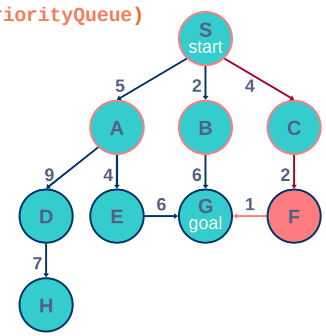


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 5, expanded: 5

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:4+2+1,G:8,E:9, D:14}

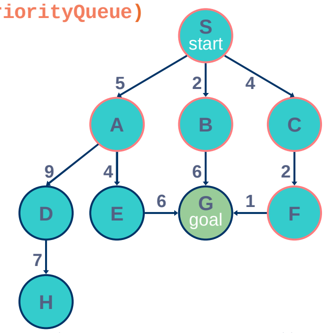


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand

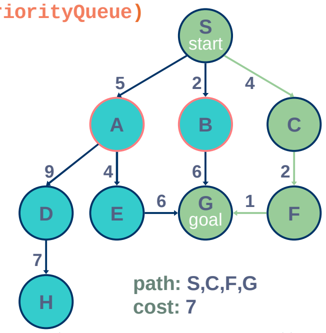


Uniform-Cost Search (UCS)

`generalSearch(problem, priorityQueue)`

# of nodes tested: 6, expanded: 5

expnd. node	nodes list
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{G:8,E:9,D:14}



path: S,C,F,G
cost: 7

Uniform-Cost Search (UCS)

- **Called *Dijkstra's Algorithm* in the algorithms lit.**
- **Similar to *Branch and Bound Algorithm* in operations research literature**
- **Complete**
- **Optimal/Admissible**
  - requires that the goal test being applied when a node is removed from the nodes list
  - rather than when the node is first generated while its parent node is expanded

Uniform-Cost Search (UCS)

- **Time and space complexity: $O(b^d)$ (i.e., exponential)**
  - d is the depth of the solution
  - b: the branching factor at each non-leaf node

Uninformed Search Strategies

- IDS: **depth-first, iterative-deepening search**
  requires modification to search algorithm:
  - do DFS to depth 1
    treat all children of the start node as leafs
  - if no solution found, do DFS to depth 2
  - repeat by increasing depth until a solution found

∗ **Start node is at depth 0**

Depth-First, Iterative-Deepening Search (IDS)

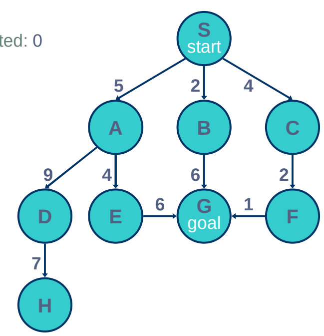
```
Node deepeningSearch (Problem problem) {
  int depth = 1;
  Stack DSnodes = new Stack();
  while (true) {  // while not solved
    Node node = DFS_depthBound(problem, DSnodes, depth);
    // DFS_depthBound limits DFS search to level <= depth
    if (node isn't "failure") return node; // solved
    depth++; // look deeper
  }
}
```


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes expanded: 0, tested: 0

expnd. node	nodes list
	{S}

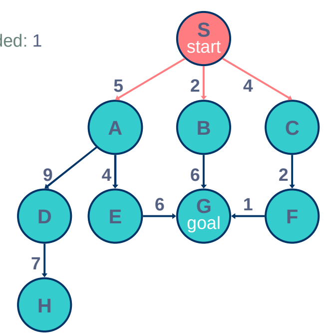


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 1, expanded: 1

expnd. node	nodes list
	{S}
S not goal	{A,B,C}

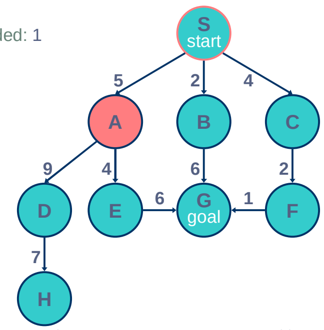


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 2, expanded: 1

expnd. node	nodes list
	{S}
S	{A,B,C}
A not goal	{B,C} no expand

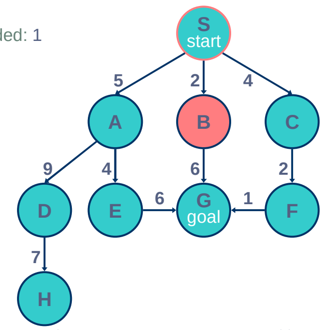


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 3, expanded: 1

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B not goal	{C} no expand

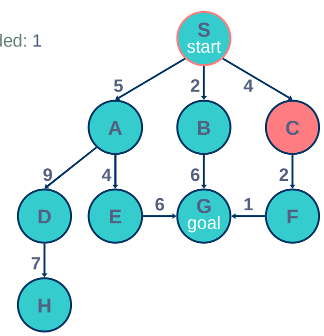


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 1, # of nodes tested: 4, expanded: 1

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C not goal	{ } no expand-FAIL

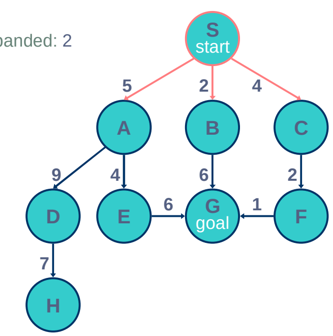


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 4(1), expanded: 2

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S no test	{A,B,C}

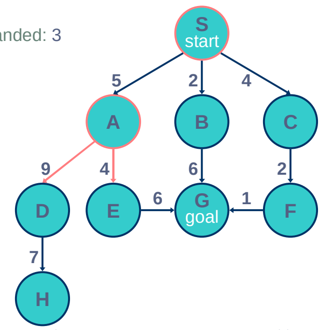


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 4(2), expanded: 3

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A no test	{D,E,B,C}

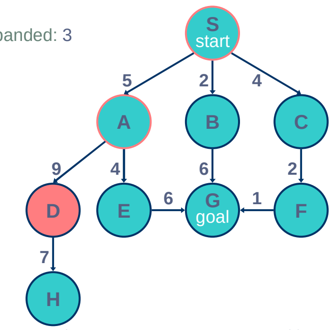


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 5(2), expanded: 3

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D not goal	{E,B,C} no expand

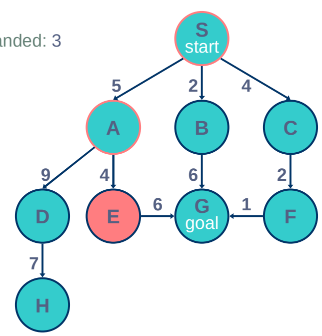


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 6(2), expanded: 3

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E not goal	{B,C} no expand

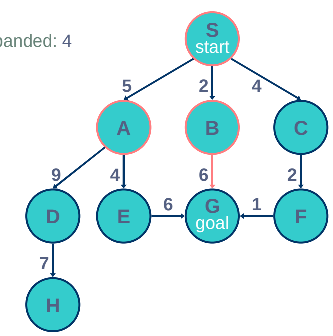


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 6(3), expanded: 4

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E	{B,C}
B no test	{G,C}

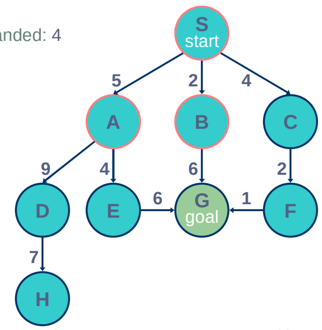


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 7(3), expanded: 4

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E	{B,C}
B	{G,C}
G goal	{C} no expand

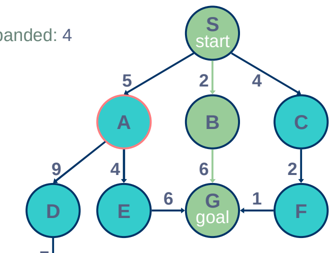


Depth-First, Iterative-Deepening Search (IDS)

`deepeningSearch(problem)`

depth: 2, # of nodes tested: 7(3), expanded: 4

expnd. node	nodes list
	{S}
S	{A,B,C}
A	{B,C}
B	{C}
C	{ }
S	{A,B,C}
A	{D,E,B,C}
D	{E,B,C}
E	{B,C}
B	{G,C}
G	{C}



path: S,B,G
cost: 8

Depth-First, Iterative-Deepening Search (IDS)

- **Has advantages of BFS**
  - completeness
  - optimality as stated for BFS
- **Has advantages of DFS**
  - limited space
  - in practice, even with redundant effort it still finds longer paths more quickly than BFS

Depth-First, Iterative-Deepening Search (IDS)

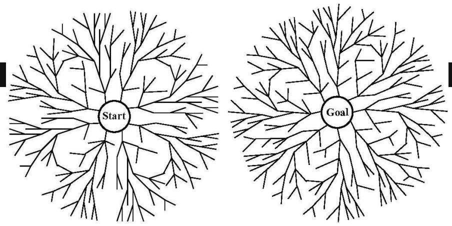
- **Space complexity: $O(bd)$ linear like DFS**
- **Time complexity is a little worse than BFS or DFS**
  - because nodes near the top of the search tree are generated multiple times (redundant effort)
- **Worst case time complexity: $O(b^d)$ exponential**
  - because most nodes are near the bottom of tree

Depth-First, Iterative-Deepening Search (IDS)

How much redundant effort is done?

- **The number of times the nodes are generated:**
  $1b^d + 2b^{(d-1)} + \ldots + db \le b^d / (1 - 1/b)^2 = O(b^d)$
  - d: the solution's depth
  - b: the branching factor at each non-leaf node
- **For example:** $b = 4$
  $4^d / (1 - ¼)^2 = 4^d / (.75)^2 = 1.78 \times 4^d$
  - in the worst case, 78% more nodes are searched (redundant effort) than exist at depth d
  - as b increases, this % decreases

Which Direction Should We Search?



Our choices: Forward, backwards, or bidirectional

The issues: How many start and goal states are there?
Branching factors in each direction
How much work is it to compare states?

General Search with Open and Close

```
//Note: this algorithm does detect loops in the state space
Node generalSearch (Problem problem, List OPEN) {
  OPEN.add(new Node(problem.getStartState()));
  List CLOSE = new List(); //initially empty, just a List DS
  while (true) {
    if (OPEN.isEmpty()) return new Node("failure");
    Node node = OPEN.remove(); //removes front node
    if (problem.isGoal(node.getState())) return node;
    CLOSE.add(node); //remember it was expanded
    OPEN.add(expand node given problem operators);
    //expand: only add successors not already in OPEN or CLOSE
  }
}
```