Uninformed Search

Chapter 3.1 – 3.4

Models To Be Studied in CS 540

State-based Models

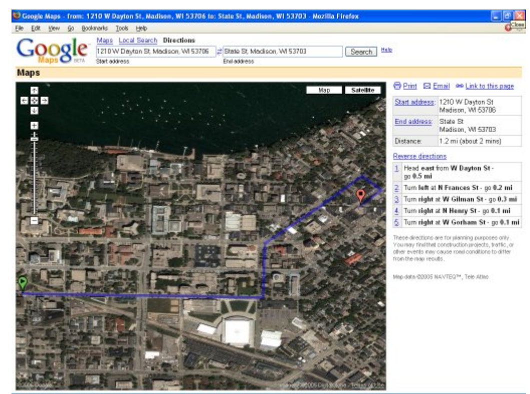
- **Model task as a graph of all possible states**
  - Called a "**state-space graph**"
- A state captures all the relevant information about the past in order to act (optimally) in the future
- Actions correspond to transitions from one state to another
- Solutions are defined as a sequence of steps/actions (i.e., a path in the graph)

Many AI (and non-AI) Tasks can be Formulated as Search Problems

Goal is to find a *sequence of actions*

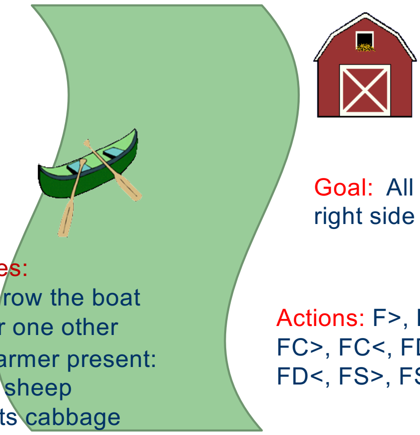
- **Puzzles**
- **Games**
- **Navigation**
- **Assignment**
- **Motion planning**
- **Scheduling**
- **Routing**

Search Example: Route Finding



Actions: go straight, turn left, turn right
Goal: shortest? fastest? most scenic?

Search Example: River Crossing Problem

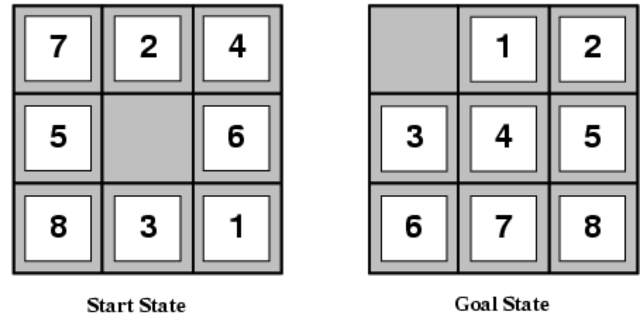


Goal: All on right side of river

Rules:
1) Farmer must row the boat
2) Only room for one other
3) Without the farmer present:
   - Dog bites sheep
   - Sheep eats cabbage

Actions: F>, F<, FC>, FC<, FD>, FD<, FS>, FS<

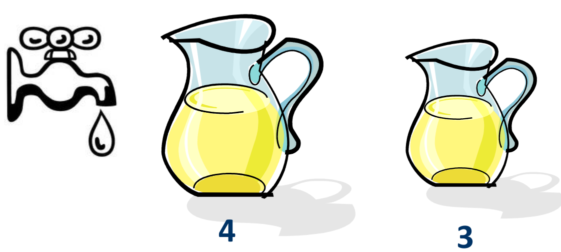
Search Example: 8-Puzzle



Actions: move tiles (e.g., Move2Down)
Goal: reach a certain configuration

Search Example: Water Jugs Problem

Given 4-liter and 3-liter pitchers, how do you get exactly 2 liters into the 4-liter pitcher?

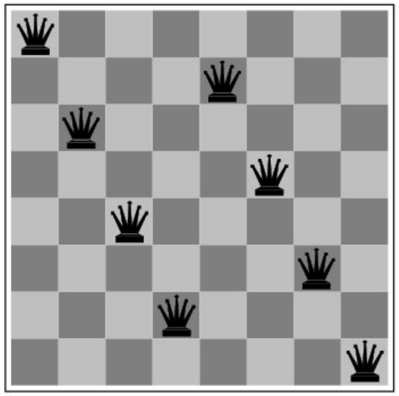


Search Example: Robot Motion Planning

Actions: translate and rotate joints

Goal: fastest? most energy efficient? safest?

Search Example: 8-Queens



What Knowledge does the Agent Need?

- The information needs to be
  - sufficient to describe all relevant aspects for reaching the goal
  - adequate to describe the world ***state*** (aka ***situation***)
- **Fully observable** assumption, also known as the ***closed world assumption***, means
  - *All necessary information about a problem domain is accessible so that each state is a complete description of the world; there is no missing (or noisy) information at any point in time*

How should the Environment be Represented?

- ***Determining what to represent is difficult and is usually left to the system designer to specify***
- Problem ***State*** = representation of all necessary information about the environment
- ***State Space*** (aka **Problem Space**) = *all* possible valid configurations of the environment

What Goal does the Agent want to Achieve?

- **How do you know when the goal is reached?**
  - with a **goal test** that defines what it means to have achieved the goal
  - or, with a set of **goal states**
- ***Determining the goal is usually left to the system designer or user to specify***

What Actions does the Agent Need?

- **Discrete and Deterministic task assumptions imply**

- **Given:**
  - an ***action*** (aka ***operator*** or ***move***)
  - a description of the current state of the world

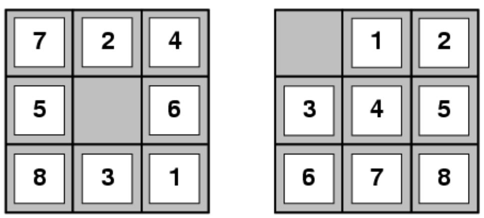
- **Action completely specifies:**
  - **if** that action *can* be applied (i.e., is it legal)
  - **what** the exact state of the world will be after the action is performed in the current state (no "history" information needed to compute the successor state)

What Actions does the Agent Need?

- **A finite set of actions/operators needs to be**
  - decomposed into atomic steps that are discrete and indivisible, and therefore can be treated as instantaneous
  - sufficient to describe all necessary changes

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

Search Example: 8-Puzzle



Start State Goal State

- **States = configurations**
- **Actions = up to 4 kinds of moves: up, down, left, right**

Water Jugs Problem

Given 4-liter and 3-liter pitchers, how do you get exactly 2 liters into the 4-liter pitcher?

State: (*x, y*) for # liters in 4-liter and 3-liter pitchers, respectively
Actions: empty, fill, pour water between pitchers
Initial state: (0, 0)
Goal state: (2, *)

Action / Successor Functions

1. $(x, y \mid x < 4) \rightarrow (4, y)$ "Fill *4*"
2. $(x, y \mid y < 3) \rightarrow (x, 3)$ "Fill *3*"
3. $(x, y \mid x > 0) \rightarrow (0, y)$ "Empty *4*"
4. $(x, y \mid y > 0) \rightarrow (x, 0)$ "Empty *3*"
5. $(x, y \mid x+y \geq 4 \text{ and } y > 0) \longrightarrow (4, y - (4 - x))$
   "Pour from *3* to *4* until *4* is full"
6. $(x, y \mid x+y \geq 3 \text{ and } x > 0) \longrightarrow (x - (3 - y), 3)$
   "Pour from *4* to *3* until *3* is full"
7. $(x, y \mid x+y \leq 4 \text{ and } y > 0) \longrightarrow (x+y, 0)$
   "Pour all water from *3* to *4*"

Formalizing Search in a State Space

- **A state space is a directed *graph*: (V, E)**
  - V is a set of nodes (vertices)
  - 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 action that generated this node (from its parent)
    - other bookkeeping data

Formalizing Search in a State Space

- **Each arc corresponds to one of the finite number of actions:**
  - when the action 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 action

Formalizing Search in a State Space

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

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 actions 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

Search Summary

- Solution is an ordered sequence of primitive actions (steps)
  $f(x) = a_1, a_2, \ldots, a_n$ where x is the input
- Model task as a graph of all possible states and actions, and a solution as a path
- A state captures all the relevant information about the past

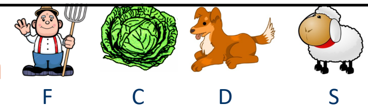
Sizes of State Spaces*

Problem	# Nodes
Tic-Tac-Toe	10^3
Checkers	10^{20}
Chess	10^{50}
Go	10^{170}

* Approximate number of legal states

What are the Components of Formalizing Search in a State Space?

Formalizing Search



A search problem has five components:

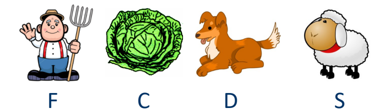
S, I, G, actions, cost

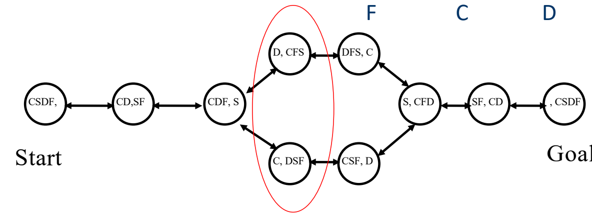
1. **State space** S **:** all valid configurations
2. **Initial states** $I \subseteq S$: a set of start states $I = \{(FCDS,)\} \subseteq S$
3. **Goal states** $G \subseteq S$: a set of goal states $G = \{(,FCDS)\} \subseteq S$
4. An **action function** ***successors*(*s*)** $\subseteq S$ **:** states reachable in one step (one arc) from *s*

   successors((FCDS,)) = {(CD,FS)}
   successors((CDF,S)) = {(CD,FS), (D,FCS), (C,FSD)}
5. A **cost function cost(*s, s'*):** The cost of moving from *s* to *s'*

- The goal of search is to find a solution path from a state in *I* to a state in *G*

State Space = A Directed Graph

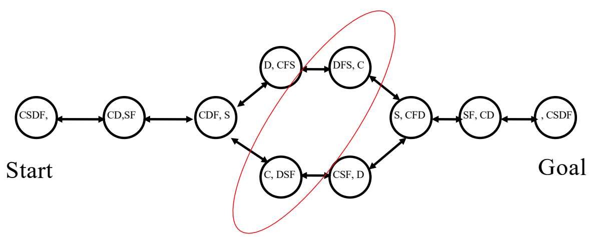




- In general, there will be many generated, but un-expanded, states at any given time during a search
- One has to choose which one to "expand" next

Different Search Strategies

- **The generated, but not yet expanded, states define the *Frontier* (aka *Open* or *Fringe*) set**
- **The essential difference is, which state in the Frontier to expand next?**



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*, in the form of a *search tree*, to include a goal node:**

***TREE SEARCH Algorithm*:**

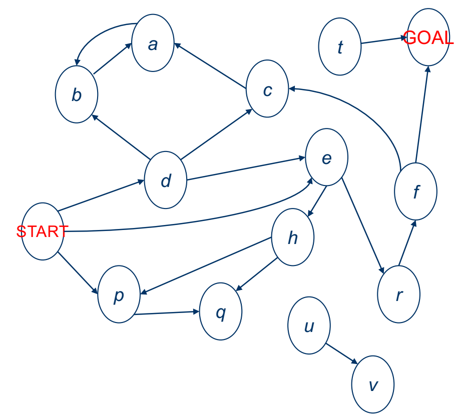
Frontier = {*S*}, where *S* is the start node
Loop do
  if *Frontier* is empty **then return** failure
  pick a node, *n*, from *Frontier*
  if *n* is a goal node **then return** solution
  Generate all *n*'s successor nodes and add them all to *Frontier*
  Remove *n* from *Frontier*

called **"expanding"** node *n*

Formalizing Search in a State Space

- **This algorithm does *NOT* detect a goal when the node is generated**
- **This algorithm does *NOT* detect loops (i.e., repeated states) in state space**
- **Each node implicitly represents**
  - a partial solution path from the start node to the given node
  - cost of the partial solution path
- **From this node there may be**
  - many possible paths that have this partial path as a prefix
  - many possible solutions

A State Space Graph



What is the corresponding search tree?

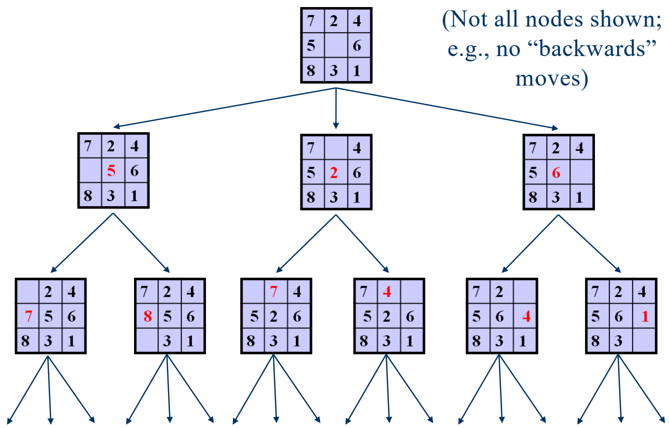
Uninformed Search on Trees

- **Uninformed means we *only* know:**
  - The goal test
  - The ***successors***() function
- But ***not*** which non-goal states are better
- For now, also assume state space is a **tree**
  - That is, we won't worry about *repeated states*
  - We will fix this later

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 Frontier list)
    - "dead ends" (nodes that aren't goals and have no successors because no operators were possible)
    - 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 *Frontier* ordering leads to different search strategies**

8-Puzzle State-Space Search Tree



Uninformed Search Strategies

Uninformed Search: strategies that order nodes *without* using any domain specific information, i.e., don't use any information stored in a state

- **BFS: breadth-first search**
  - *Queue (FIFO) used for the Frontier*
  - remove from front, add to back
- **DFS: depth-first search**
  - *Stack (LIFO) used for the Frontier*
  - remove from front, add to front

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*, in the form of a *search tree*, to include a goal node:**

***TREE_SEARCH Algorithm*:**

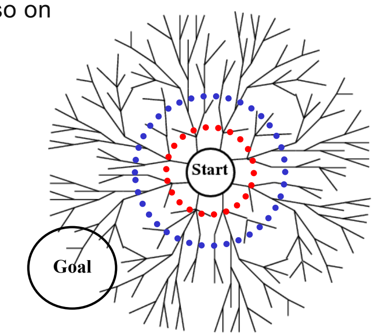
Frontier = {*S*}, where *S* is the start node
Loop do
  if *Frontier* is empty **then return** failure
  pick a node, *n*, from *Frontier*
  if *n* is a goal node **then return** solution
  Generate all *n*'s successor nodes and add them all to *Frontier*
  Remove *n* from *Frontier*

called "**expanding**" node *n*

Breadth-First Search (BFS)

Expand the shallowest node in the tree first:

1. Examine states one step away from the initial state
2. Examine states two steps away from the initial state
3. and so on

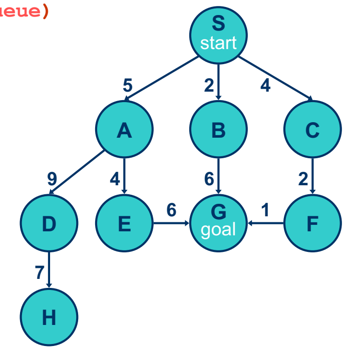


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 0, expanded: 0

expnd. node	Frontier list
	{S}

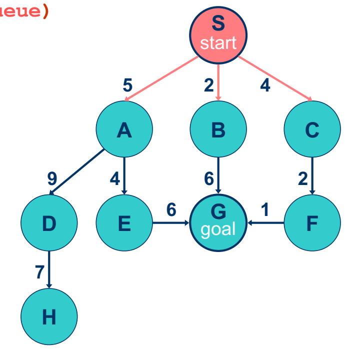


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 1, expanded: 1

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

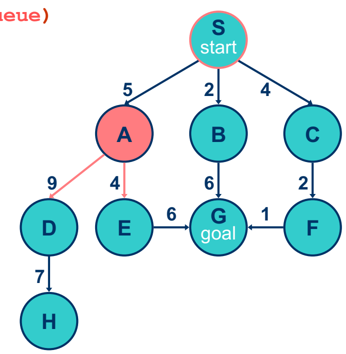


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 2, expanded: 2

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

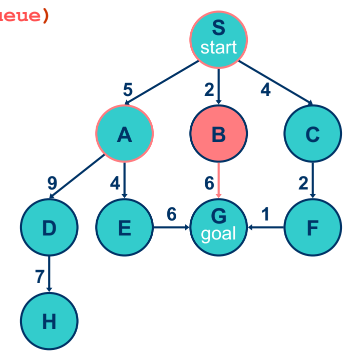


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 3, expanded: 3

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

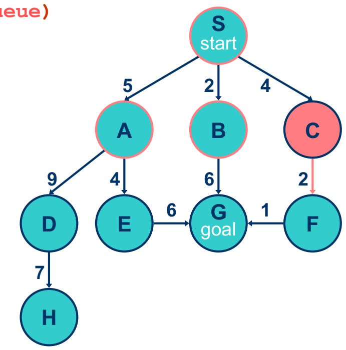


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 4, expanded: 4

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

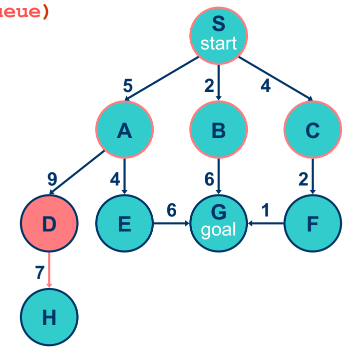


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 5, expanded: 5

expnd. node	Frontier 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}

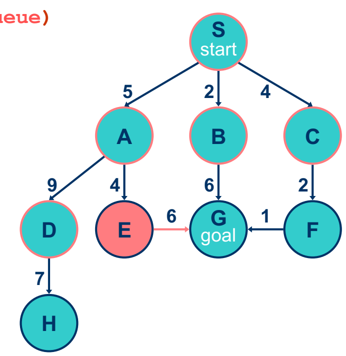


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 6, expanded: 6

expnd. node	Frontier 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}

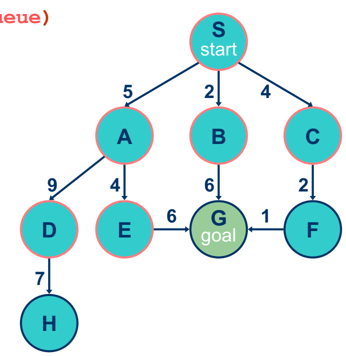


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

expnd. node	Frontier 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

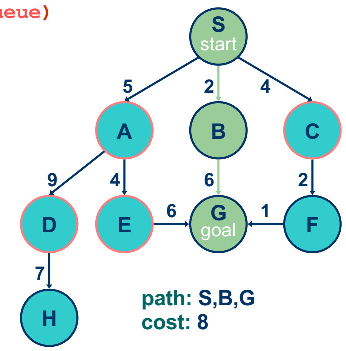


Breadth-First Search (BFS)

`generalSearch(problem, queue)`

# of nodes tested: 7, expanded: 6

expnd. node	Frontier 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

Evaluating Search Strategies

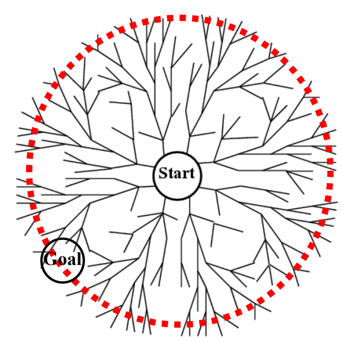
- **Completeness**
  If a solution exists, will it be found?
  - a complete algorithm will find ***a*** solution (not all)
- **Optimality / Admissibility**
  If a solution is found, is it guaranteed to be optimal?
  - an admissible algorithm will find a **solution with minimum cost**

Evaluating Search Strategies

- **Time Complexity**
  How long does it take to find a solution?
  - usually measured for worst case
  - measured by counting **number of nodes expanded, including goal node, if found**
- **Space Complexity**
  How much space is used by the algorithm?
  - measured in terms of the **maximum size of *Frontier*** during the search

What's in the Frontier for BFS?

- **If goal is at depth *d*, how big is the Frontier (worst case)?**



Breadth-First Search (BFS)

- **Complete?**
  - Yes
- **Optimal / Admissible?**
  - **Yes**, ***if*** all operators (i.e., arcs) have the same constant cost, or costs are positive, non-decreasing with depth
  - 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 is the branching factor at each non-leaf node
- Very slow to find solutions with a large number of steps because must look at *all* shorter length possibilities first

Breadth-First Search (BFS)

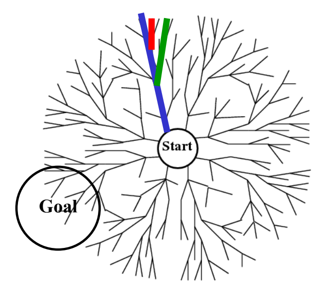
- **A complete search tree has a total # of nodes =**
  $\mathbf{1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1) / (b-1)}$
  - d: the tree's depth
  - b: the branching factor at each non-leaf node
- **For example:** $d = 12$, $b = 10$
  $1 + 10 + 100 + ... + 10^{12} = (10^{13} - 1)/9 = O(10^{12})$
  - If BFS expands 1,000 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

Expand the ***deepest*** node first

1. Select a direction, go deep to the end
2. Slightly change the end
3. Slightly change the end some more…

Use a Stack to order nodes in *Frontier*

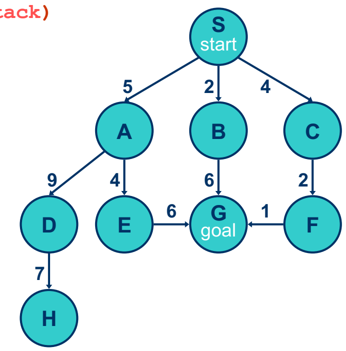


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 0, expanded: 0

expnd. node	Frontier
	{S}

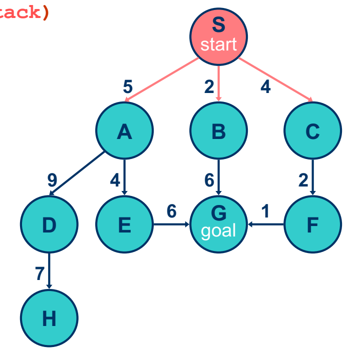


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 1, expanded: 1

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

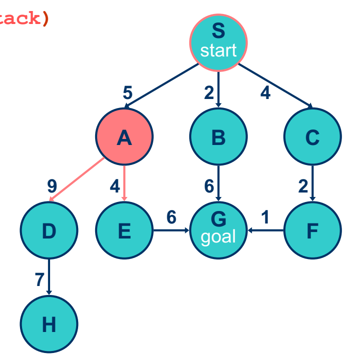


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 2, expanded: 2

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

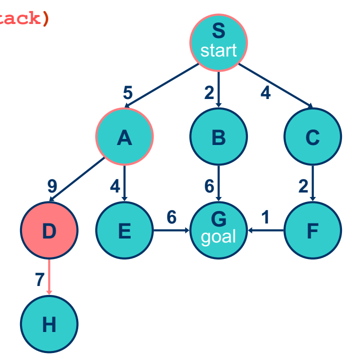


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 3, expanded: 3

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

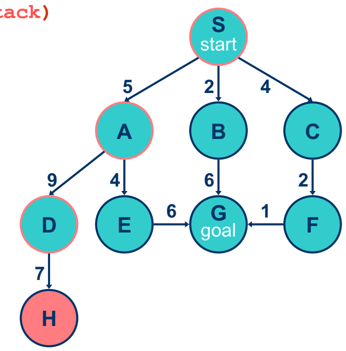


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 4, expanded: 4

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

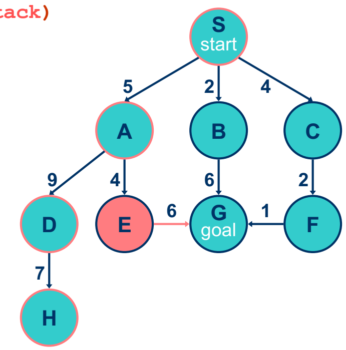


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 5, expanded: 5

expnd. node	Frontier
	{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}

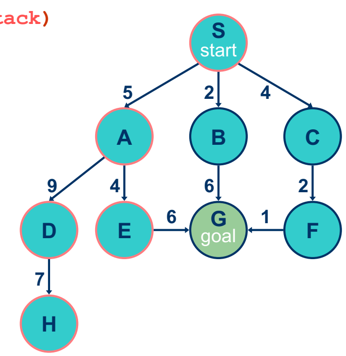


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

expnd. node	Frontier
	{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

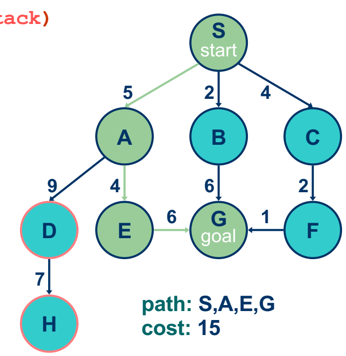


Depth-First Search (DFS)

`generalSearch(problem, stack)`

# of nodes tested: 6, expanded: 5

expnd. node	Frontier
	{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}



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***

Depth-First Search (DFS)

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

Uniform-Cost Search (UCS)

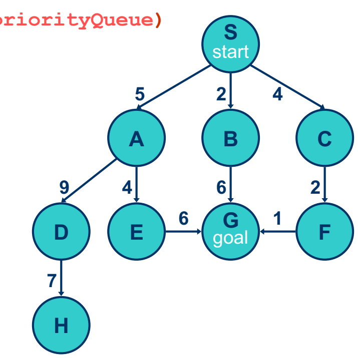
- Use a **Priority Queue** to order nodes in *Frontier*, sorted by path cost
- Let $g(n) =$ cost of path from start node s to current node n
- Sort nodes by increasing value of g

Uniform-Cost Search (UCS)

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

\# of nodes tested: 0, expanded: 0

expnd. node	Frontier list
	{S}

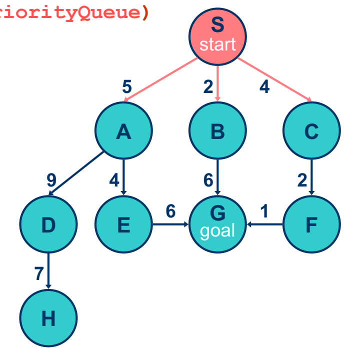


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

\# of nodes tested: 1, expanded: 1

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

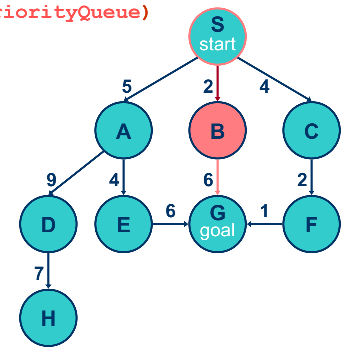


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

\# of nodes tested: 2, expanded: 2

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

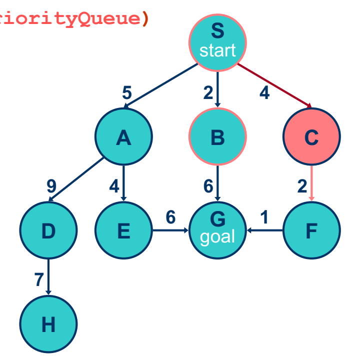


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

\# of nodes tested: 3, expanded: 3

expnd. node	Frontier 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}

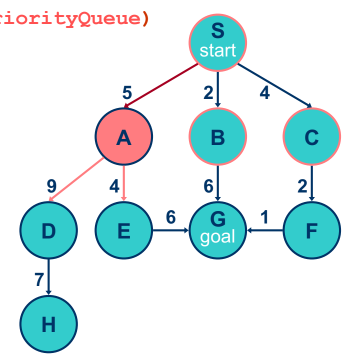


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

\# of nodes tested: 4, expanded: 4

expnd. node	Frontier 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}

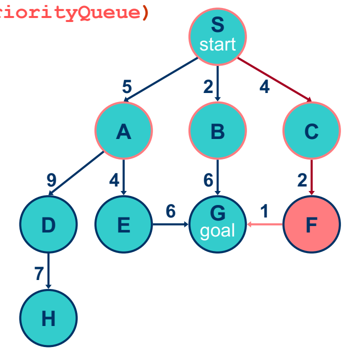


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 5, expanded: 5

expnd. node	Frontier 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}

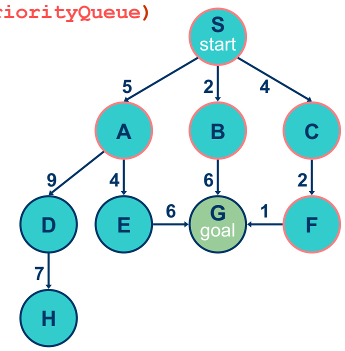


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

expnd. node	Frontier 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

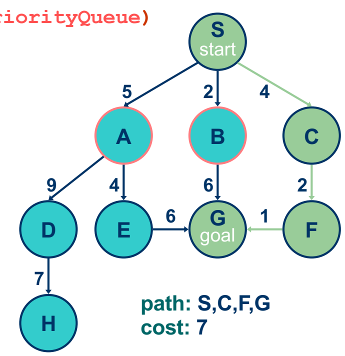


Uniform-Cost Search (UCS)

generalSearch(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

expnd. node	Frontier 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 literature**
- **Similar to *Branch and Bound Algorithm* in Operations Research literature**
- **Complete**
- **Optimal / Admissible**
  - requires that the goal test is done when a node is ***removed*** from the *Frontier* rather than when the node is generated by its parent node

Uniform-Cost Search (UCS)

- **Time and space complexity: $O(b^d)$ (i.e., exponential)**
  - d is the depth of the solution
  - b is the branching factor at each non-leaf node
- **More precisely, time and space complexity is $O(b^{C^*/\varepsilon})$ where all edge costs are ε, $\varepsilon > 0$, and C^* is the best goal path cost**

Iterative-Deepening Search (IDS)

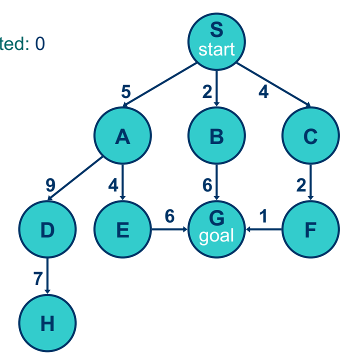
- **requires modification to DFS search algorithm:**
  - do DFS to depth 1
    and treat all children of the start node as leaves
  - if no solution found, do DFS to depth 2
  - repeat by increasing "depth bound" until a solution found
- **Start node is at depth 0**

Iterative-Deepening Search (IDS)

```
deepeningSearch(problem)
```

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

expnd. node	Frontier
	{S}

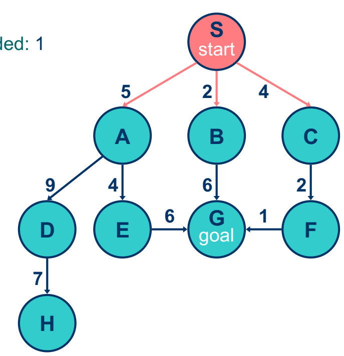


Iterative-Deepening Search (IDS)

```
deepeningSearch(problem)
```

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

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

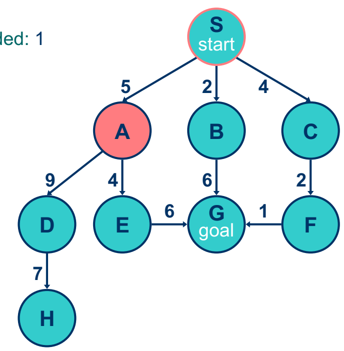


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

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

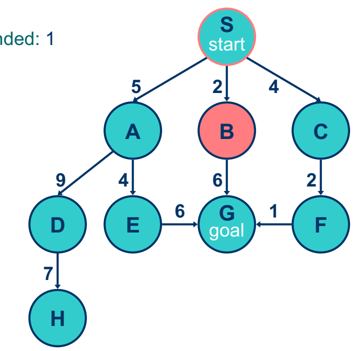


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

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

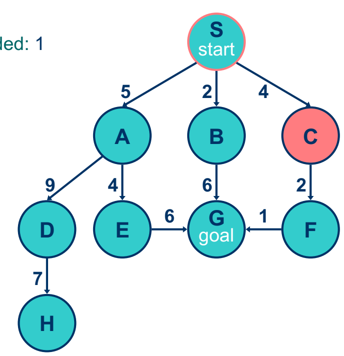


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

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

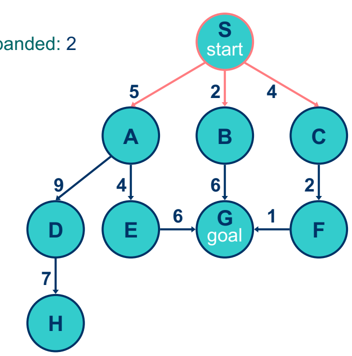


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

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

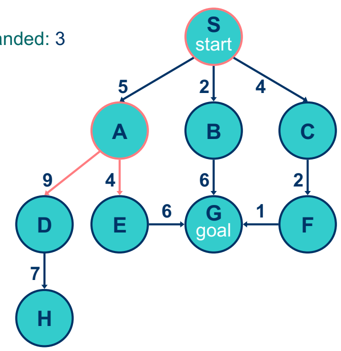


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

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

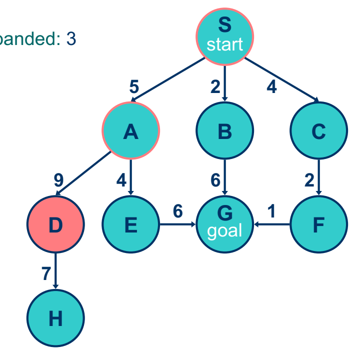


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

expnd. node	Frontier
	{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

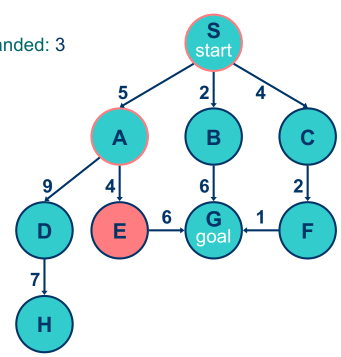


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

expnd. node	Frontier
	{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

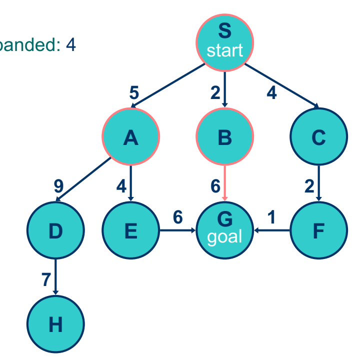


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

expnd. node	Frontier
	{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}

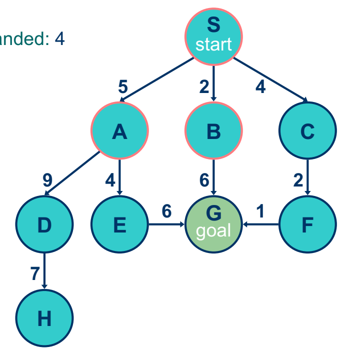


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

expnd. node	Frontier
	{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

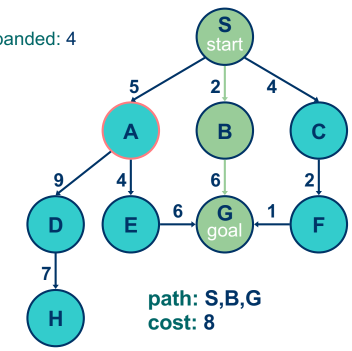


Iterative-Deepening Search (IDS)

deepeningSearch(problem)

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

expnd. node	Frontier
	{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}



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

Iterative-Deepening Search (IDS)

- **Space complexity:** $O(bd)$ **(i.e., 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

Iterative-Deepening Search (IDS)

How much redundant effort is done?

- **The number of times the nodes are generated:**
  $1b^d + 2b^{(d-1)} + ... + db \leq b^d / (1 - 1/b)^2 = O(b^d)$
  - 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

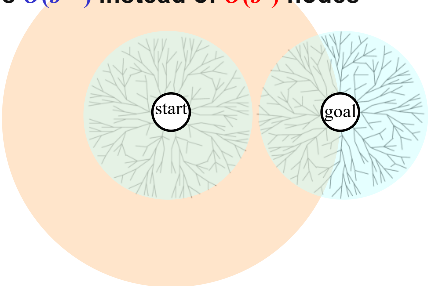
Iterative-Deepening Search

- **Trades a little time for a huge reduction in space**
  - lets you do breadth-first search with (more space efficient) depth-first search
- **"Anytime" algorithm: good for response-time critical applications, e.g., games**

- An "anytime" algorithm is an algorithm that can return a valid solution to a problem even if it's interrupted at any time before it ends. The algorithm is expected to find better and better solutions the longer it runs.

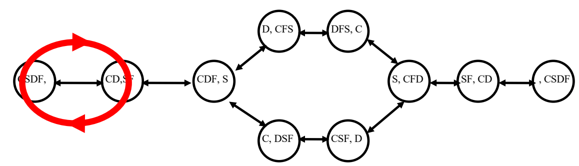
Bidirectional Search

- **Breadth-first search from both start and goal**
- **Stop when Frontiers meet**
- **Generates $O(b^{d/2})$ instead of $O(b^d)$ nodes**



If State Space is *Not* a Tree

- **The problem: *repeated states***



- **Ignoring repeated states: wasteful (BFS) or impossible (DFS). Why?**

- **How to prevent these problems?**

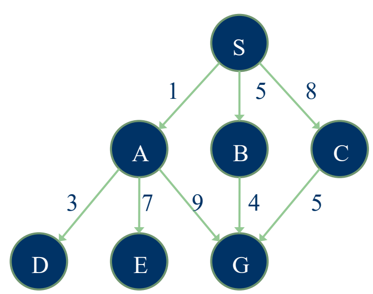
If State Space is *Not* a Tree

- **We have to remember already-expanded states (called *Explored* (aka *Closed*) set) too**
- **When we pick a node from *Frontier***
  - Remove it from *Frontier*
  - Add it to *Explored*
  - Expand node, generating all successors
  - For each successor, *child*,
    - If *child* is in *Explored* or in *Frontier*, throw *child* away // for BFS and DFS
    - Otherwise, add it to *Frontier*
- Called **Graph-Search algorithm** in Figure 3.7 and **Uniform-Cost-Search** in Figure 3.14

```
function Uniform-Cost-Search (problem)
loop do
  if Empty?(frontier) then return failure
  node = Pop(frontier)
  if Goal?(node) then return Solution(node)
  Insert node in explored
  foreach child of node do
    if child not in frontier or explored then
      Insert child in frontier
    else if child in frontier with higher cost then
      Remove that old node from frontier
      Insert child in frontier
```

This is the algorithm in Figure 3.14 in the textbook; note that if *child* is **not** in *frontier* but **is** in *explored*, this algorithm will throw away *child*

Example



How are nodes expanded by

- Depth First Search
- Breadth First Search
- Uniform Cost Search
- Iterative Deepening

Are the solutions the same?

Nodes Expanded by:

- **Depth-First Search: S A D E G**
  Solution found: S A G
- **Breadth-First Search: S A B C D E G**
  Solution found: S A G
- **Uniform-Cost Search: S A D B C E G**
  Solution found: S B G
- **Iterative-Deepening Search: S A B C S A D E G**
  Solution found: S A G