CSCI3230 (ESTR3108)
Fundamentals of Artificial Intelligence

Lecture 9. Uninformed Search

Prof. Qi Dou

Email: qidou@cuhk.edu.hk
Office: Room 1014, 10/F, SHB

Dept. of Computer Science & Engineering
The Chinese University of Hong Kong

Outline

Part 1. Introduction

What is search?

What is a search algorithm?

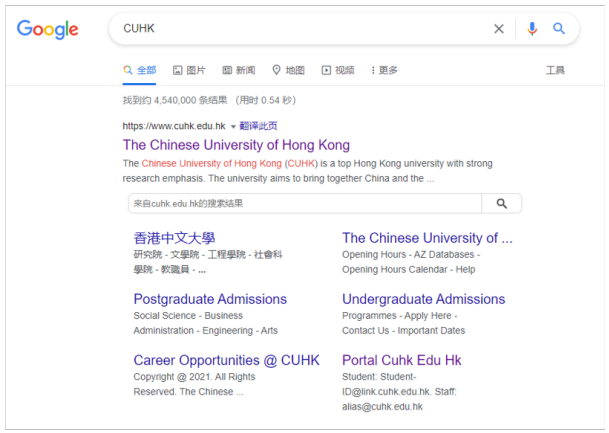
- A search algorithm is an algorithm which solves a search problem. Search algorithms work to retrieve information stored within some data structure, or calculated in the search space of a problem domain, either with discrete or continuous values.

Why do we need search algorithms?

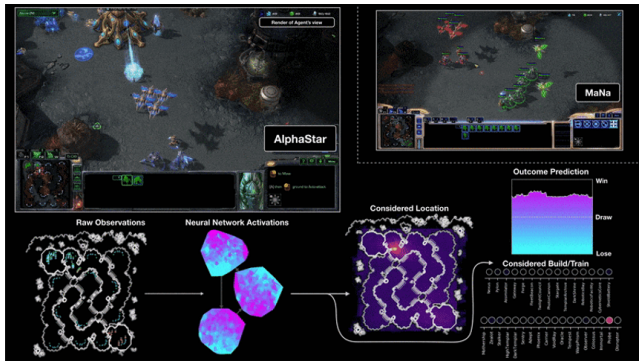
It has many applications:

- Path design: Given the current location and the destination, design the shortest path.
- Search engine: Order relevant search results based on many factors.
- Video game: AlphaStar beats 99.8%of human StarCraft II players.

Path Design



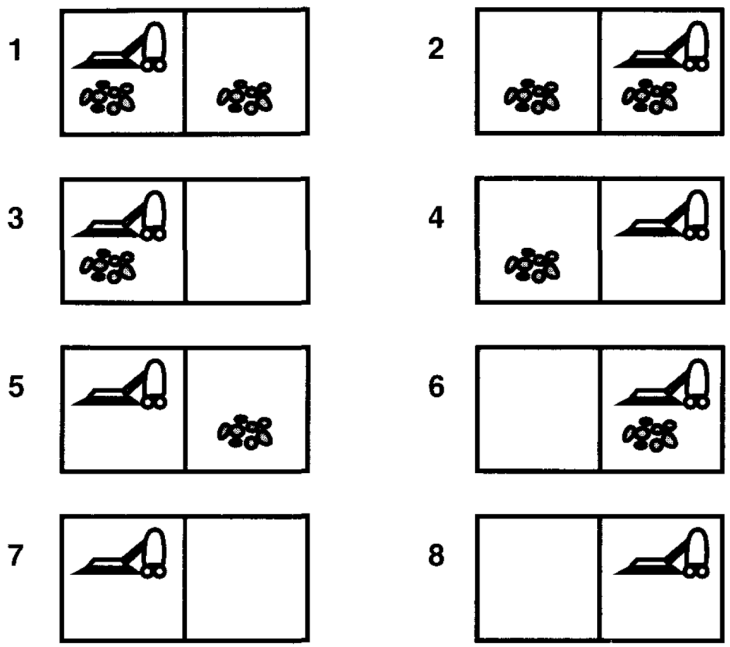
Search Engine



StarCraft II (game with human player named MaNa)

An example: Vacuum Cleaner Problem

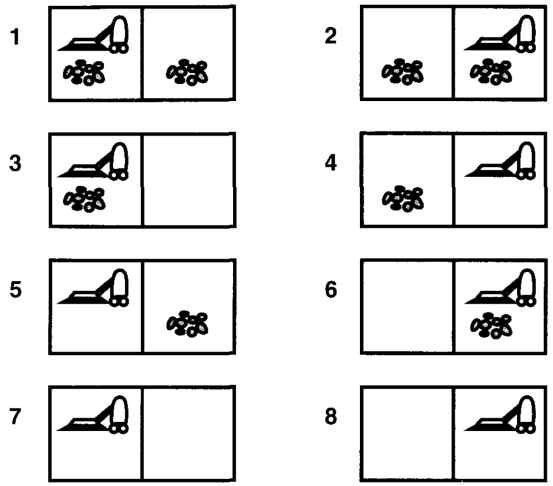
- Vacuum cleaner problem is a well-known search problem.
- In this problem, there are two locations, each of them may or may not contain dirt. The vacuum cleaner is the agent, which may be in one location or the other. So, there are 8 possible states.
- The agent has three possible actions, i.e., Left, Right and Suck.
- The goal of this agent is to clean up the whole area.



An example: Vacuum Cleaner Problem

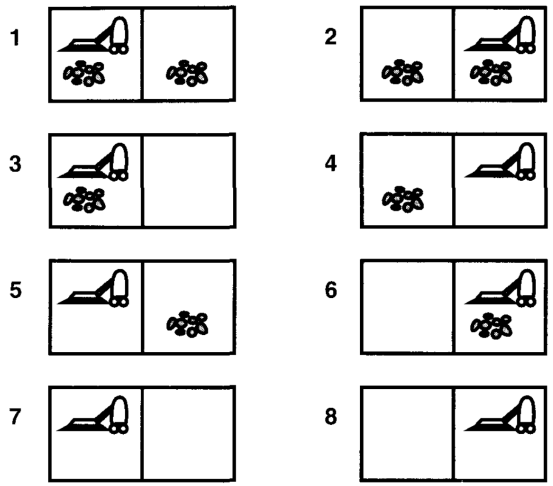
We can change from one state to another state by a sequence of actions.

- Assume that the initial state is $\mathbf{1}$, after $Right$ action, the state is changed from $\mathbf{1}$ to $\mathbf{2}$.
- Assume that the initial state is $\mathbf{6}$, after $Left$ action, the state is changed from $\mathbf{6}$ to $\mathbf{5}$.
- Assume that the initial state is $\mathbf{1}$, after $Suck$ action, the state is changed from $\mathbf{1}$ to $\mathbf{5}$.



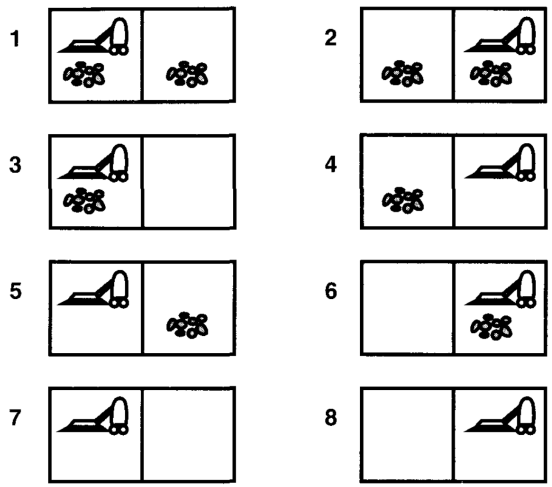
Problem types - knowledge of states and actions

- Single-state problem: complete world state knowledge, complete action knowledge
  - Agent's sensors give it enough information to tell exactly which state the agent is in.
  - And suppose that the agent knows exactly what each of its actions does, then it can calculate exactly which state it will be in after any sequence of actions.
  - For example, if its initial state is 5, then the agent can calculate that the action sequence [Right,Suck] will achieve the goal.



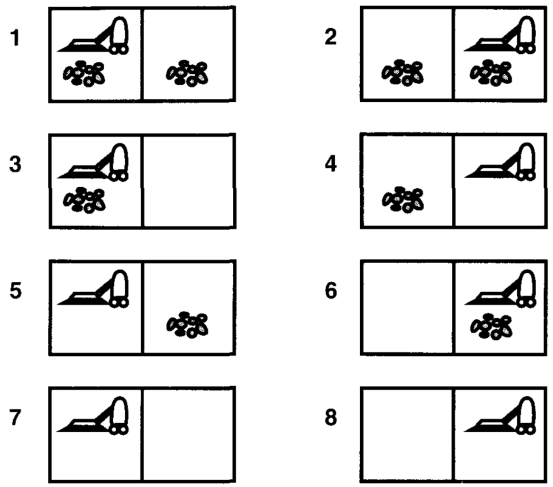
Problem types - knowledge of states and actions

- Multiple-state problem: incomplete world state knowledge, incomplete action knowledge
  - Suppose that the agent knows all the effects of its actions, but has limited access to the world state.
  - For example, in the extreme case, the agent may have no sensors at all. In this case, it only knows its initial state is one of $\{1,2,3,4,5,6,7,8\}$. Because the world is not fully accessible, the agent must reason about sets of states that it might get to, rather than a single state.



Problem types - knowledge of states and actions

- Contingency problem: not possible to define a complete sequence of actions that constitute a solution in advance because information about the intermediary states is unknown.
  - For example, cleaner sometimes deposits dirt on the carpet when there is no dirt existing. If the agent is in state 4, if it sucks, it will reach one of the states $\{2, 4\}$.
  - In other words, the effect of an action is not determined.

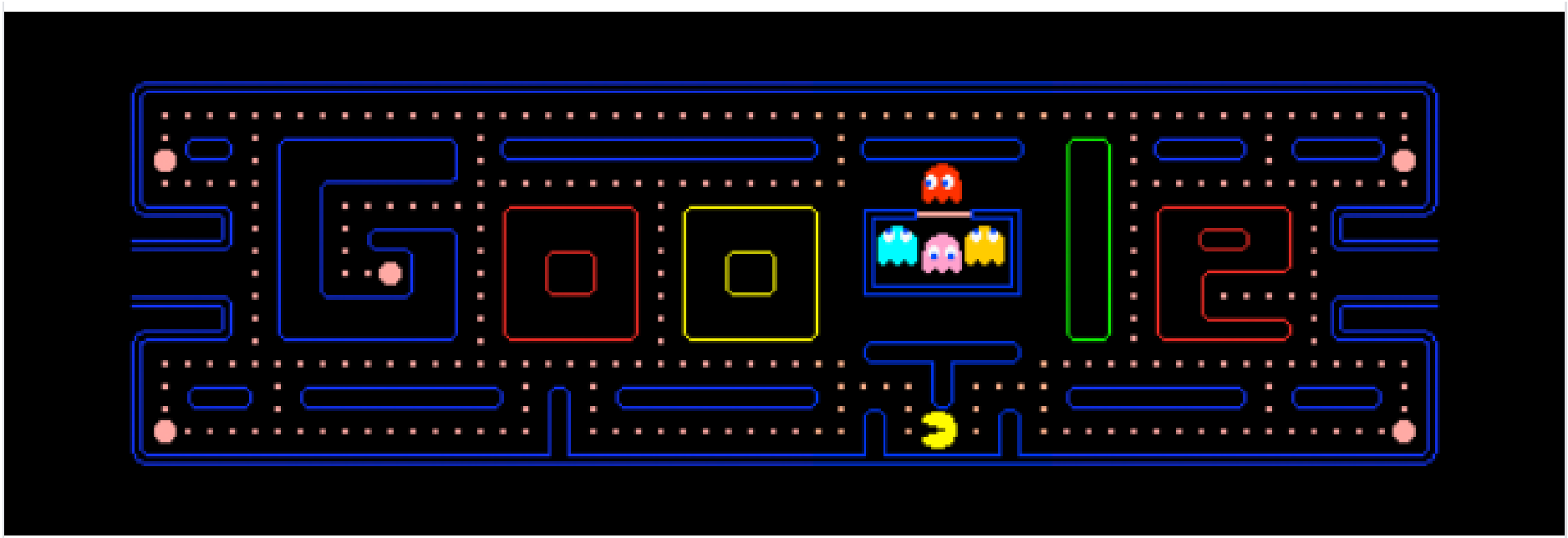


Problem types

- Exploration problem: state space and effects of actions are unknown
  - This is somewhat equivalent to being lost in a strange country with no map at all.
  - In this case, the agent must experiment, gradually discovering what its actions do and what sorts of states exist.
  - This is the hardest task faced by an intelligent agent.
  - It is also the task faced by newborn babies.

Search problems

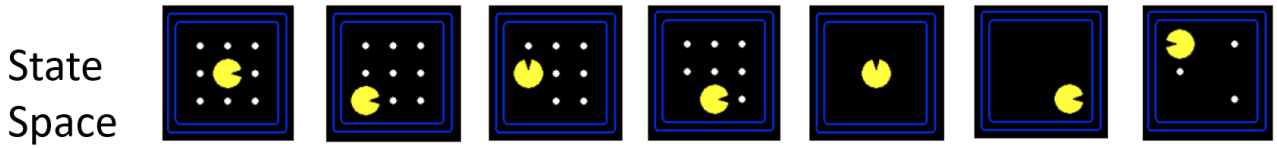
- Pac-Man is a classic video game which was developed in 1980.
- Goal of the game: eat all yellow dots.
- How can we define the problem and finish the goal?



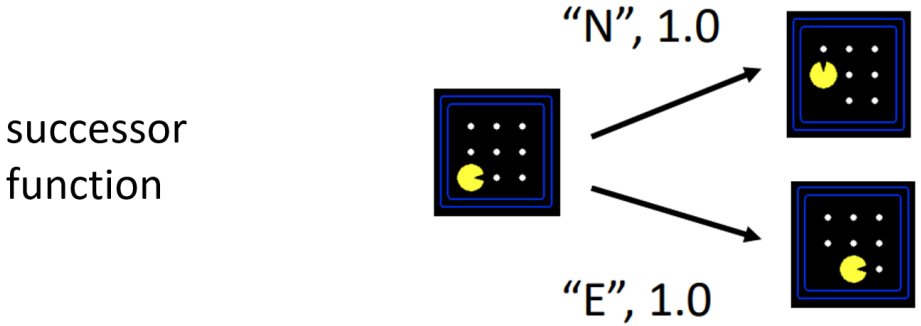
Search problems

A search problem consists of the following components.

- A state space:



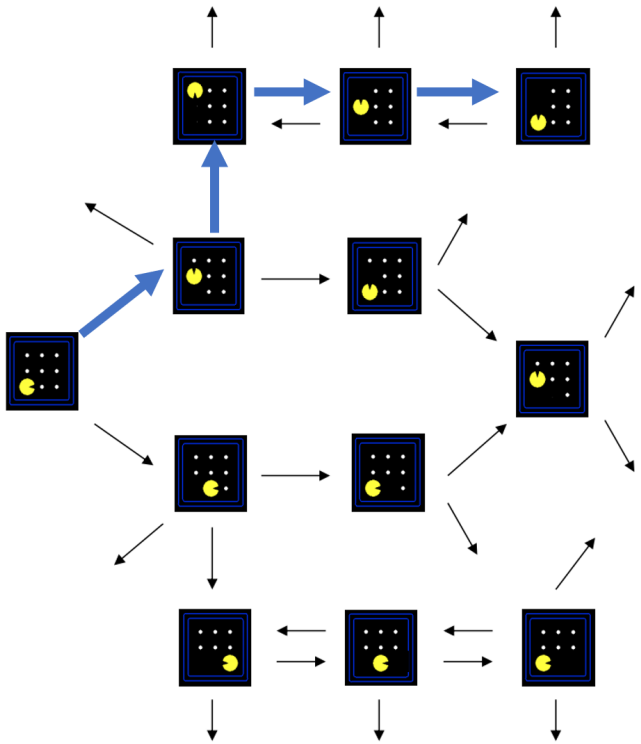
- A successor function (with actions, costs):



- A start state and a goal test.

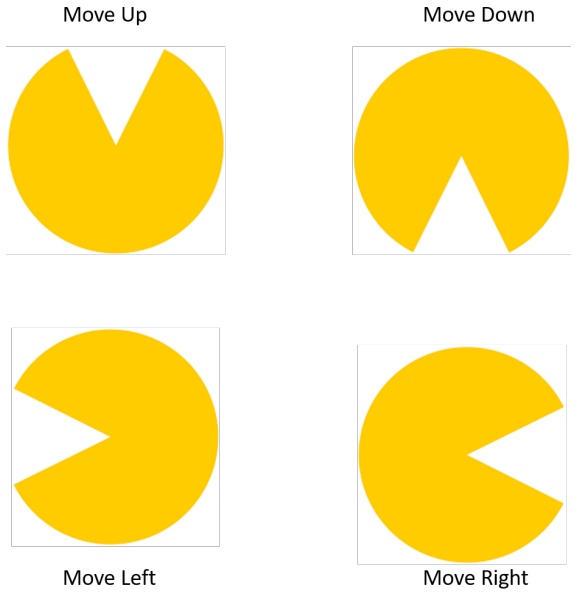
Search problems

- Start state (initial state): indicate which state that the agent starts in.
- State space: the set of all states reachable from the initial state by any sequence of actions.
- Path: a path in the state space is simply any sequence of actions leading from one state to another.



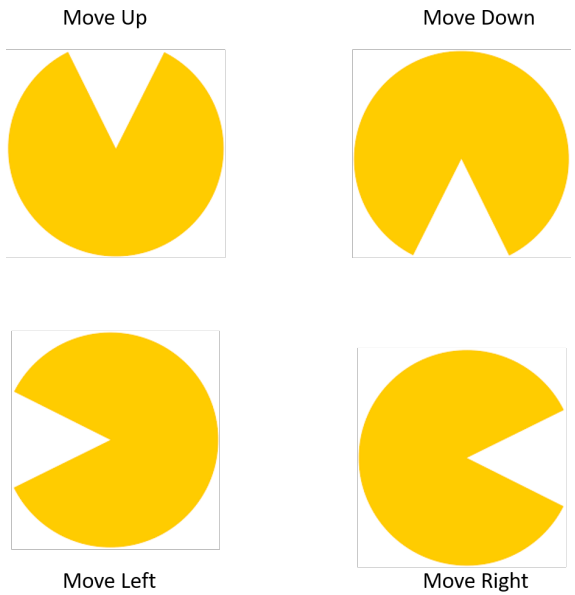
Search problems

- Successor function S: Given a particular state x, $S(x)$ returns the set of states reachable from x by any single action.
- Operator set: the set of possible actions available.



Search problems

- Successor function S: Given a particular state x, $S(x)$ returns the set of states reachable from x by any single action.
- Operator set: the set of possible actions available.



- Path cost function: the sum of the costs of the individual actions along the path. For example, if consider how much time do you need to eat all dots, then, the cost is the time.

Search problems

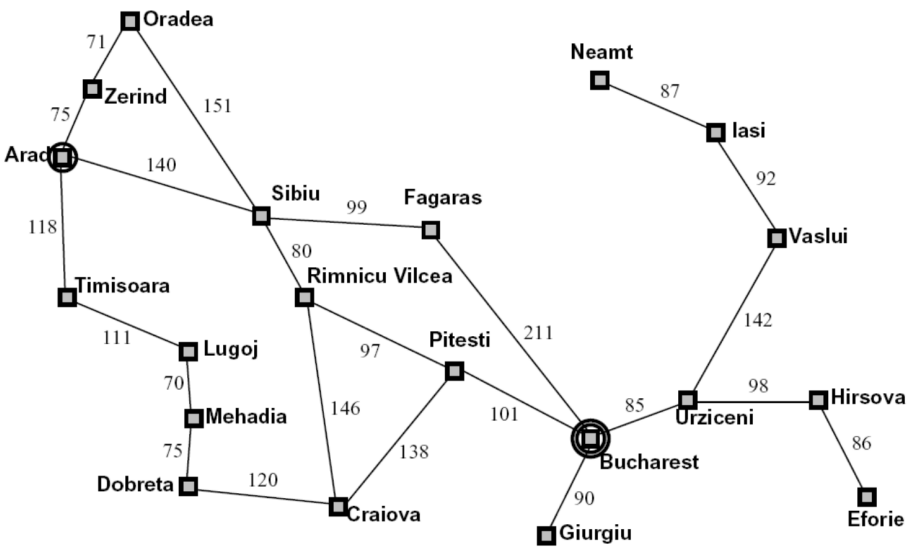
- The goal test: the agent can apply the goal test to a single state to determine if it is a goal state.
- For example, does Pac-Man already eat all dots?

- Sometimes, there is an explicit set of possible goal states, and the test simply checks to see if the agent have reached one of them.
- Sometimes, the goal is specified by an abstract property rather than an explicitly itemized set of states. For example, in chess, the goal is to reach a state called "checkmate".

Search problems

- Together, given the initial state, state space, successor function, and goal test, we can define a search problem.
- In other words, a problem is a collection of information that the agent will use to decide what to do.
- These information will be the input to the search algorithms.
- The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

One more example: Traveling in Romania

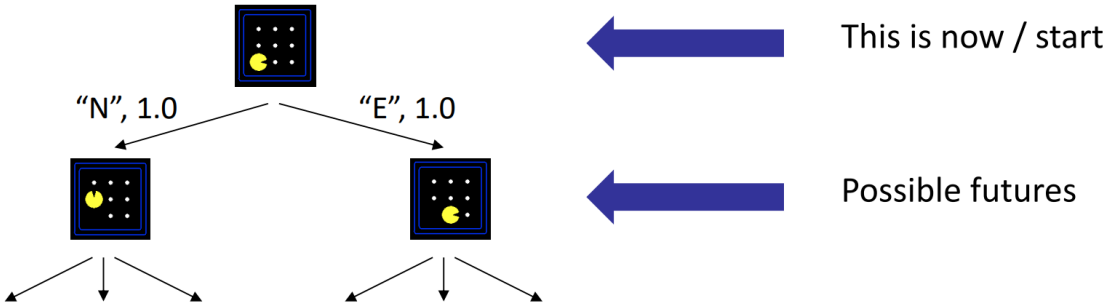


- State space: Cites
- Successor function:
  Go to adjacent city with cost=distance
- Start state: Arad
- Goal test:
  Is state == Bucharest?
- Solution?

Searching for solutions

In order to find the solution to a problem, we can form a search tree from the state space of the problem.

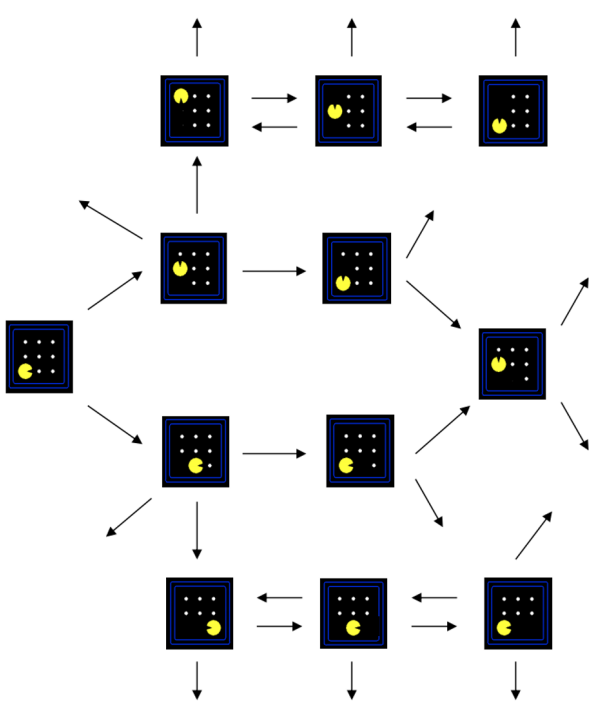
- Search tree is superimposed over the state space.
- Search tree root is a search node corresponding to the initial state.
- Search tree leaf nodes correspond to states that do not have successors in the tree, either because they have not been expanded, or because they are expanded but generated the empty set.
- At each step, the search algorithm chooses one leaf node to expand.

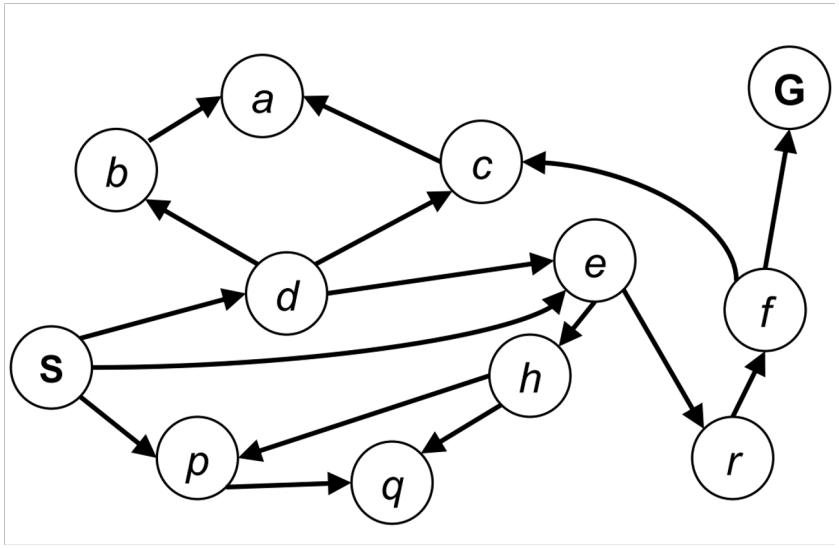


State space graph

State space graph: mathematical representation of a search problem

- Nodes are instances of states
- Arcs represent successors (action results)
- The goal test is a (set of) goal node(s)
- Each state occurs only once

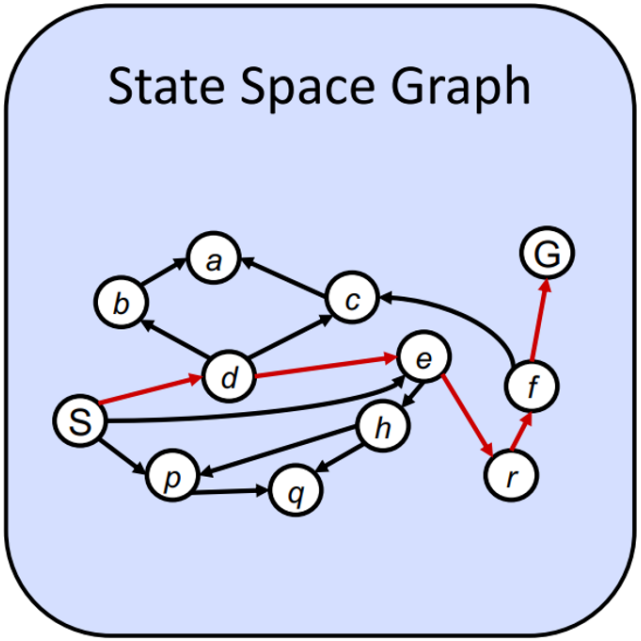


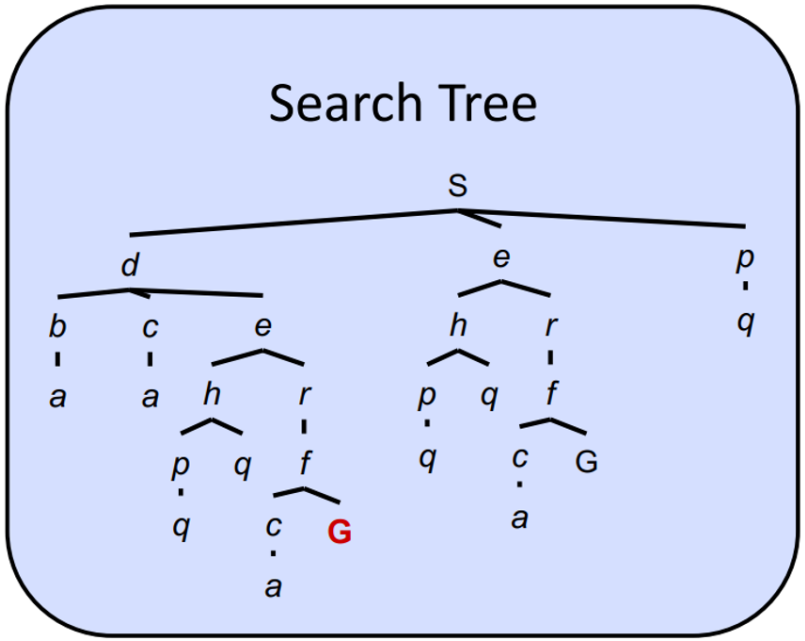


State space graph v.s. search tree

Let's compare the state space graph and search tree

- Assume that the start state is S and the goal state is G
- Each node in the search tree is an entire path in a state space graph

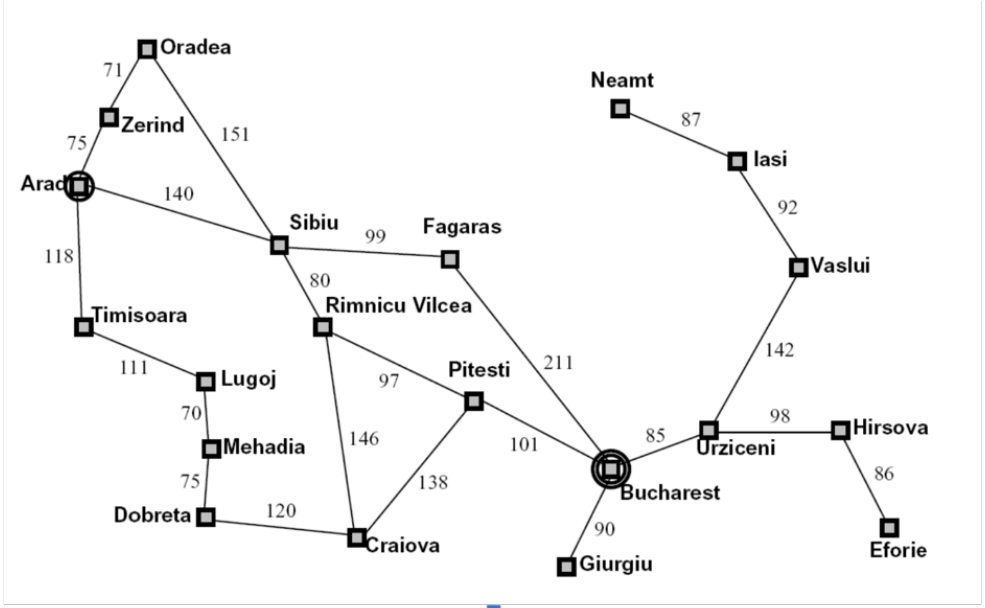




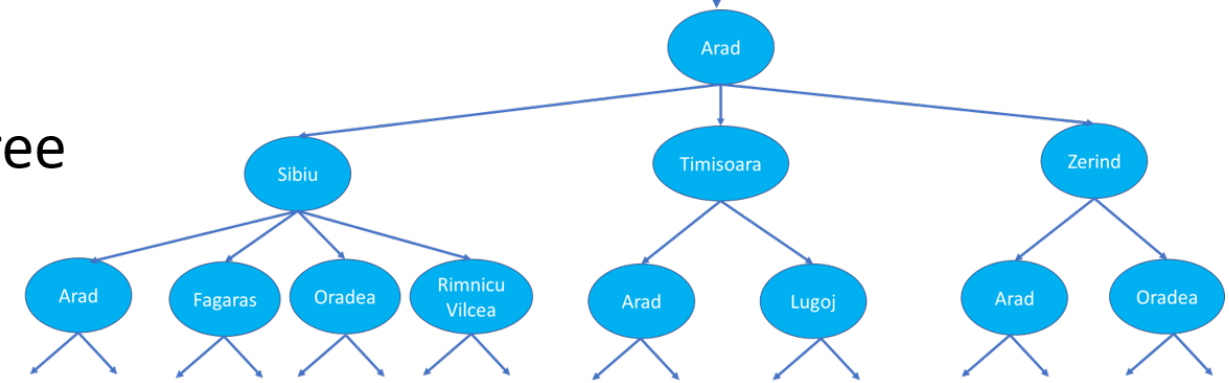
State space graph v.s. search tree

An example: from state space graph to search tree

State Space Graph



Search Tree



General tree search

Key concepts in general tree search

- Fringe: the collection of nodes that are waiting to be expanded
- Expansion: pick one out of fringe and expand it
- Search strategy: determine the order in which nodes are expanded. In other words, it is a function that selects the next node to be expanded from the fringe.

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

Search strategy

- The output of a problem-solving algorithm is either failure or a solution. (Note that some algorithms might get stuck in an infinite loop and never return an output.)
- We typically evaluate search strategies in terms of four criteria:
  - Completeness: Is the strategy guaranteed to find a solution when there is one?
  - Optimality: Does the strategy find the optimal solution when there are several different solutions? (An optimal solution has the lowest path cost among all solutions.)
  - Time complexity: How long does it take to find a solution?
  - Space complexity: How much memory does it need to do the search?

Uninformed search

- Uninformed search (a.k.a. blind search)
  - The term means that they have no additional information about states beyond that provided in the problem definition.
  - All they can do is to generate successors and distinguish a goal state from a non-goal state.
- We cover five uninformed search strategies:
  - breadth-first search
  - uniform cost search
  - depth-first search
  - depth-limited search
  - iterative deepening search
- Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic search strategies.

Part 2. Breadth-first search

Breadth-first search

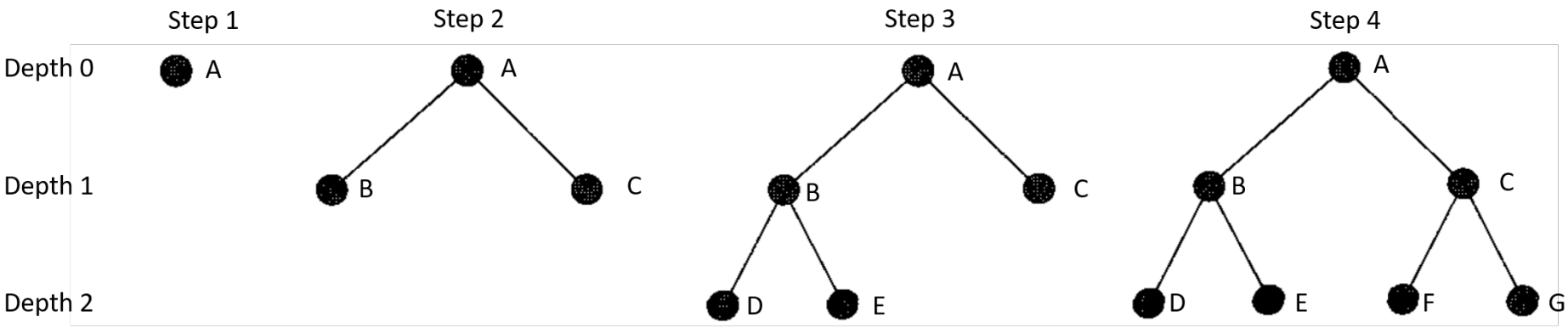
Breadth-first search is a simple search strategy

- The root node is expanded first, then all the successors of the root node are expanded next, and then their successors, and so on.
- In general, all the nodes at depth d in the search tree are expanded, before the nodes at depth $d + 1$

Breadth-first search - an example

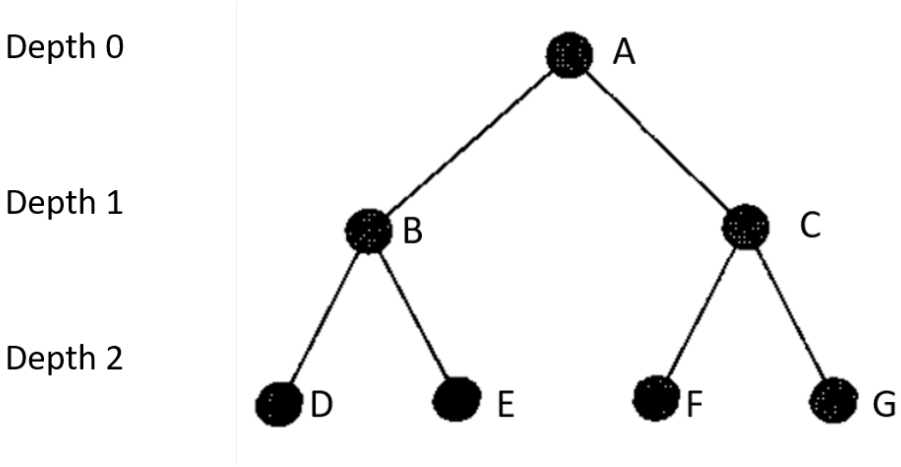
Assume that the destination is node G

1. Start the breadth-first search algorithm from the initial node A.
2. Expand node B and C from initial node A.
3. Expand node D and E from node B. The paths ABD and ABE are not the solution.
4. Expand node F and G from node C. The node G is our destination.
5. The search algorithm can be finished.



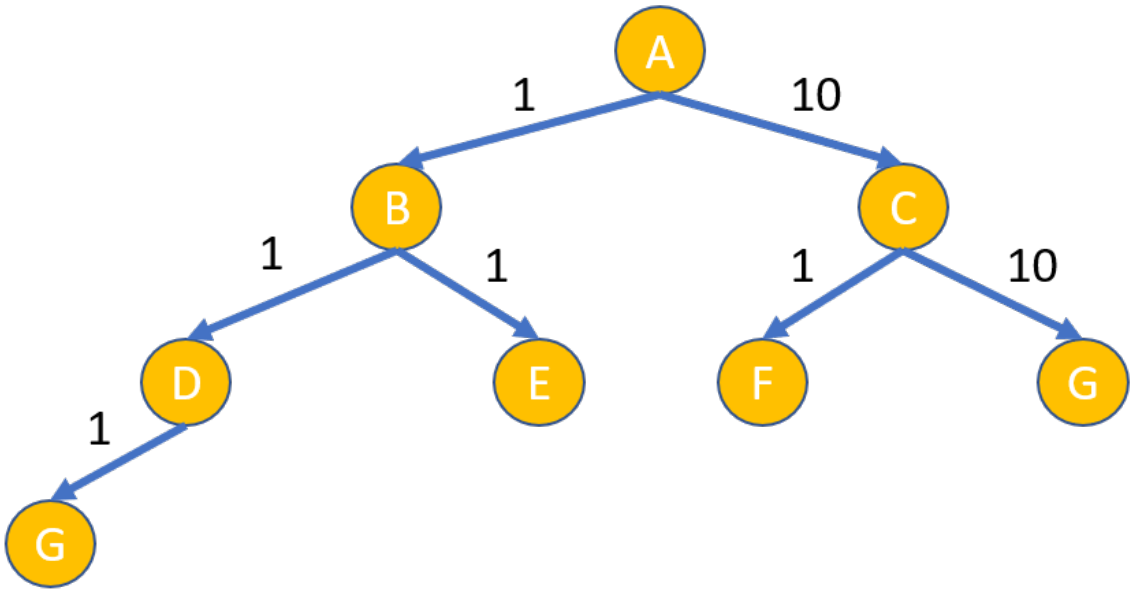
Breadth-first search analysis: completeness

- It is complete. If the shallowest goal node is at some finite depth d, we will eventually find it after expanding all shallower nodes.
- See the following example:
  - Search at depth 0, if no solution,
  - Search at depth 1, if no solution,
  - Search at depth 2, find the goal node.
  - Depth d must be finite if a solution exists. If branching factor is finite, we can finally go to the depth d.



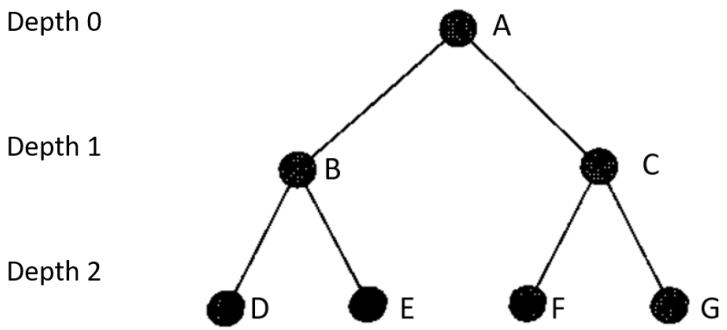
Breadth-first search analysis: optimality

- The shallowest goal node is not necessary the optimal one.
- Breadth-first search is optimal if the path cost is a non-decreasing function of the depth of the node. For example, when all actions have the same cost, breadth-first search is optimal.
- Otherwise, see an example in which some costs are decreasing
  - If use breadth-first search, the solution is ACG while the cost is 20.
  - However, the optimal solution is $ABDG$, in which the cost is 3.



Breadth-first search analysis: time complexity

- Time complexity: consider a hypothetical state space where every state has b successors.
  - The root of the search tree generates b nodes at the first level.
  - Each node generates b more nodes, for a total of b^2 at the second level.
  - Each node generates b more nodes, yielding b^3 nodes at the third level, and so on.
- Suppose that the solution is at depth d, in the worst case, we would expand all but the last node at level d (goal node is not expanded):
  - When depth is 0, yielding b^0 node, including A
  - When depth is 1, yielding b^1 nodes, including B, C
  - When depth is 2, yielding b^2 nodes, including D, E, F, G
  - When depth is d, the time complexity is $1+ b+b^2+b^3+...+b^d=O(b^d)$.



Breadth-first search analysis: space complexity

- Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node.
- So, the space complexity is $O(b^d)$, i.e., the same as time complexity
- In general, the memory requirement is a bigger problem for breadth-first search than is the execution time.

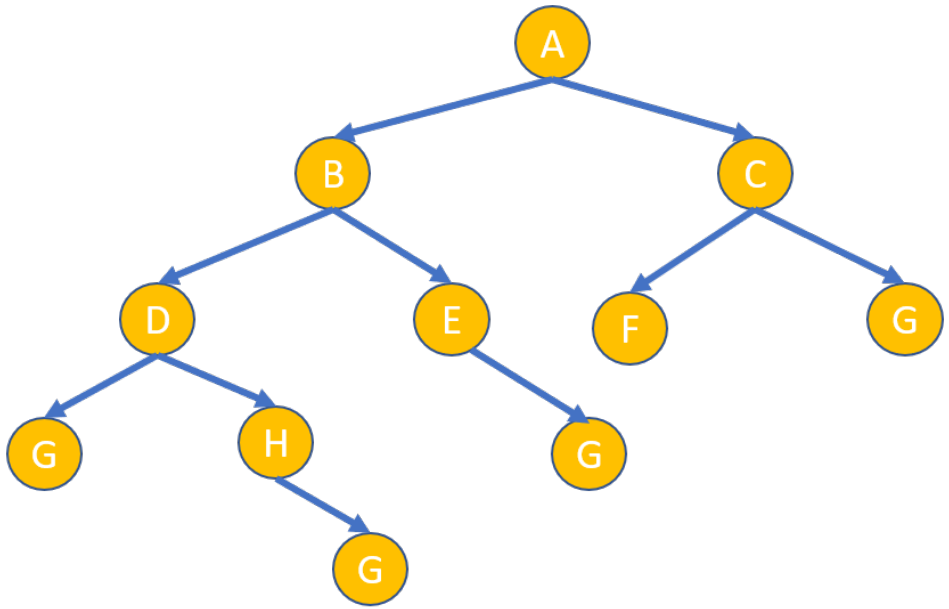
Depth	Node	Time	Memory
2	1100	0.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

Time and memory requirements for breadth-first search. The number shown assume branching factor b=10; 10,000 nodes/second ; 1000 bytes/node

Quiz

If node G is a goal node, what is the solution if use breadth-first search?

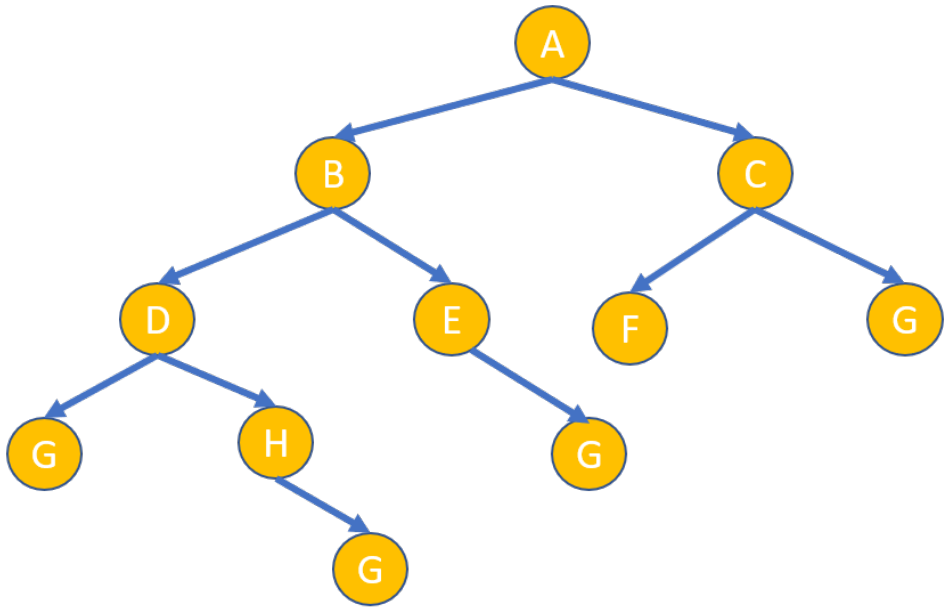
A. ABDG
B. ABDHG
C. ABEG
D. ACG



Quiz

If node G is a goal node, what is the solution if use breadth-first search?

A. ABDG
B. ABDHG
C. ABEG
D. ACG



Answer: D

Part 3. Uniform-cost search

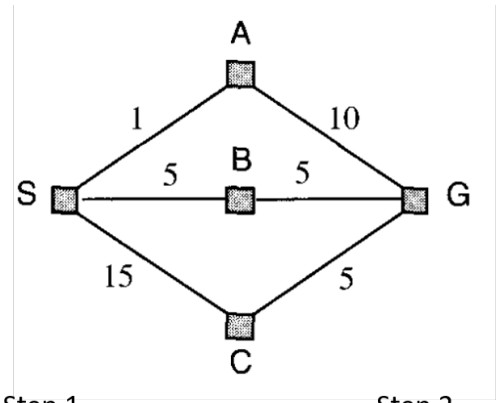
Uniform-cost search

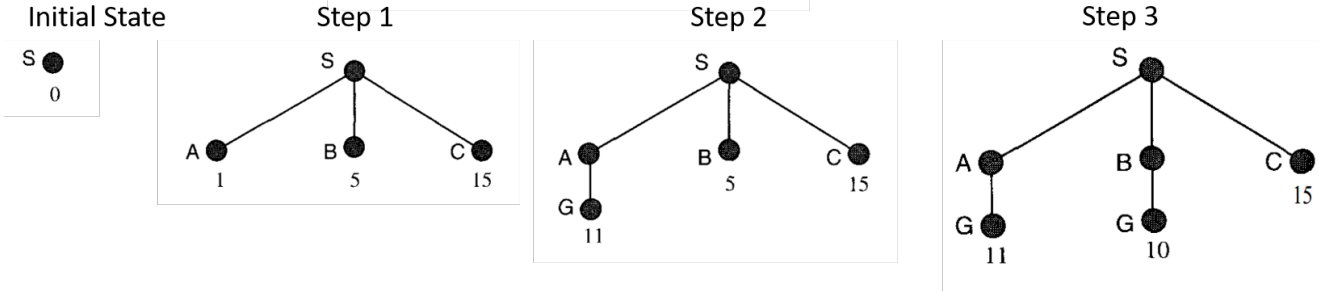
Uniform-cost search is an extension to breadth-first search

- Instead of expanding the shallowest node, uninform-cost search expands the node with the lowest path cost.
- Uniform-cost search does not care about the number of steps a path has, but only about their total cost.
- If all step costs are equal, this is identical to breadth-first search.

Uniform-cost search - an example

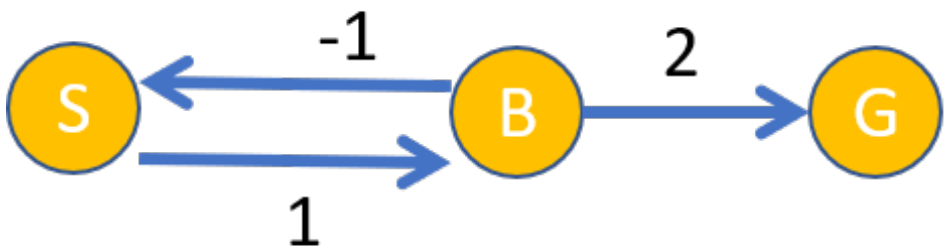
- Expand the initial node S, yielding paths to A, B, and C.
- Since A has lowest-cost, so it is expanded next, generate path SAG.
- The algorithm does not yet recognize this as a solution, because the cost is 11 (current lowest-cost is 5 for path SB)
- Expand path SB, generating SBG, which is now the lowest-cost path. It is goal-checked and returned as the solution.





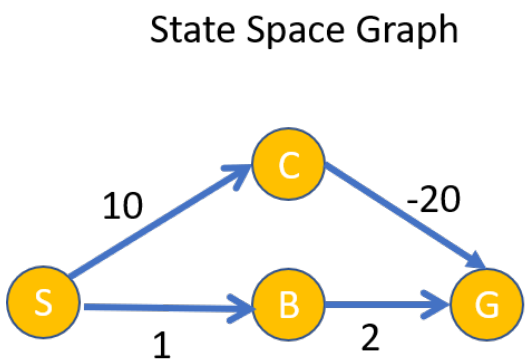
Uniform-cost search analysis: completeness

- It is complete provided the cost of every step is greater than or equal to some small positive constant ϵ.
- If there exists some negative costs, it is not guaranteed to find a goal node. See an example:
  - We start from S. There is one node B available. Therefore, we expand the node B
  - After that, there are two nodes available, including node A and node G. We find that $S \rightarrow B \rightarrow S$ cost is 1-1=0, which is lower than $S \rightarrow B \rightarrow G$ with cost 3. Therefore, we expand node A.
  - It is a loop so that we can not reach the goal node G.



Uniform-cost search analysis: optimality

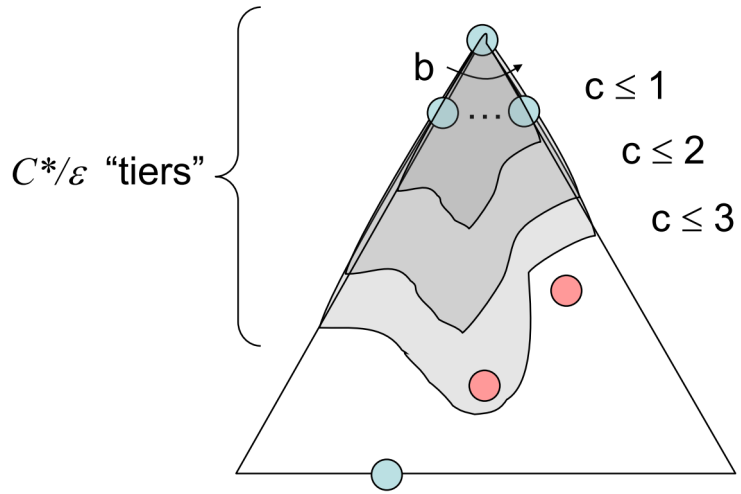
- It is optimal provided the cost of every step is greater than or equal to some small positive constant ϵ. This means that the cost of a path always increases as we go along the path.
- If there exists some negative costs, it is not guaranteed that the solution is optimal. See an example:
  - We start from the node S. then expand B because the cost is lowest.
  - Then, we expand the node G to get solution path SBG. However, the solution SBG is not optimal, as the optimal solution is SCG



Uniform-cost search analysis: time complexity

- Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of b and d.
- Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ, then the "effective depth" is roughly $\frac{C^*}{\epsilon}$.

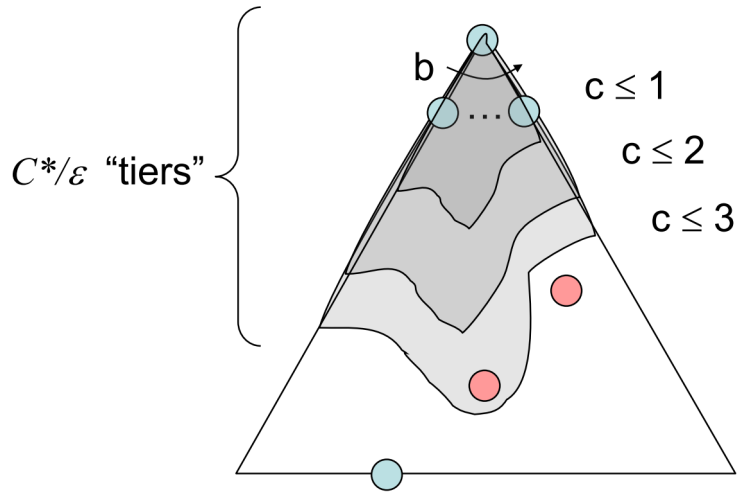
- Time complexity is $O(b^{\frac{C^*}{\epsilon}})$
  - When depth is 0, yielding b^0 node
  - When depth is 1, yielding b^1 nodes
  - When depth is 2, yielding b^2 nodes
  - When depth is $\frac{C^*}{\epsilon}$, time complexity is $1 + b + b^2 + b^3 + ... + b^{\frac{C^*}{\epsilon}} = O(b^{\frac{C^*}{\epsilon}})$.



Uniform-cost search analysis: space complexity

- Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of b and d.
- Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least ϵ, then the "effective depth" is roughly $\frac{C^*}{\epsilon}$.

- Space complexity is $O(b^{\frac{C^*}{\epsilon}})$
  - Similar to breadth-first search, it stores every node that is generated.
  - Because it is either part of the fringe or is an ancestor of a fringe node.

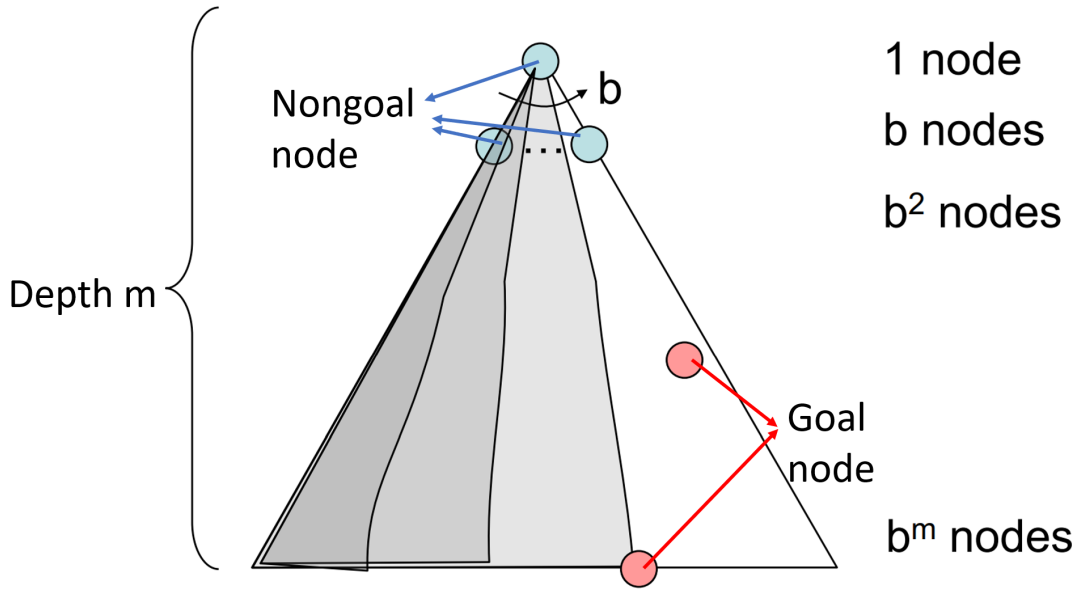


Part 4. Depth-first search

Depth-first search

Depth-first search always expands the deepest node in the current fringe of the search tree.

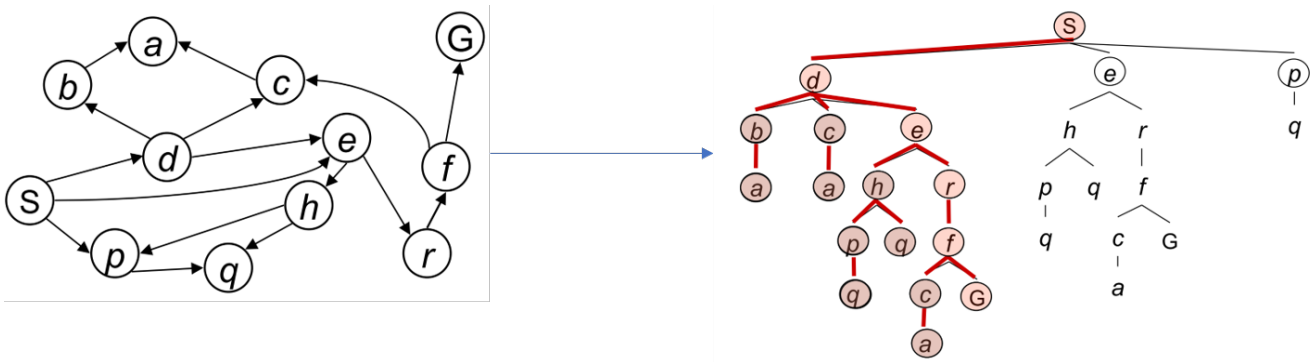
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- In other words, only when the search hits a dead end (a non-goal node with no expansion) does the search go back, and expand nodes at shallower level, and so on.



Depth-first search - an example

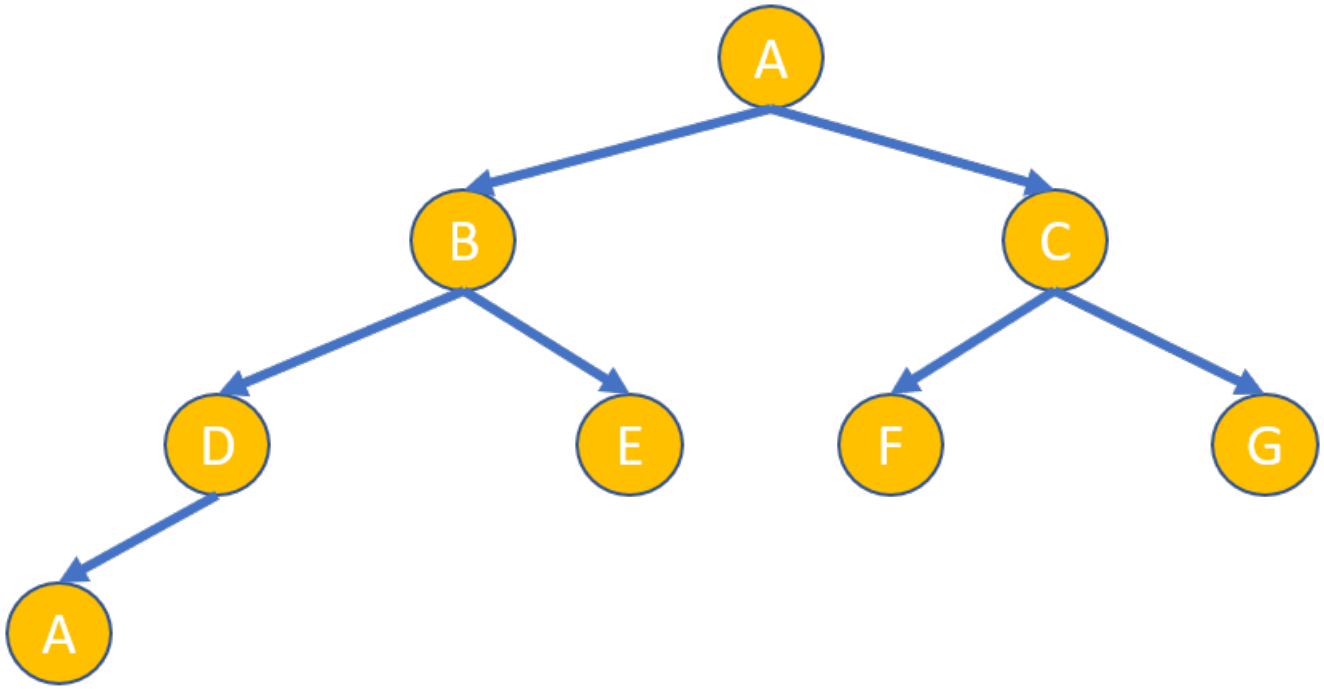
Assume that the goal is node G.

- It first adds node d, e, p to the fringe from the initial state.
- Then, it chooses node d as the next one and adds node node b, c, e to the fringe.
- Then, it expands node a from node b. As node a can not expand other nodes, so go back to node b. As node b only expands a, so go back to node d.
- Repeat the above operations for nodes c and e.
- Finally, find a path $S \rightarrow$ d $\rightarrow$ e $\rightarrow$ r $\rightarrow$ f $\rightarrow G$ which reaches the goal.
- Only red paths are explored by depth-first search algorithm.



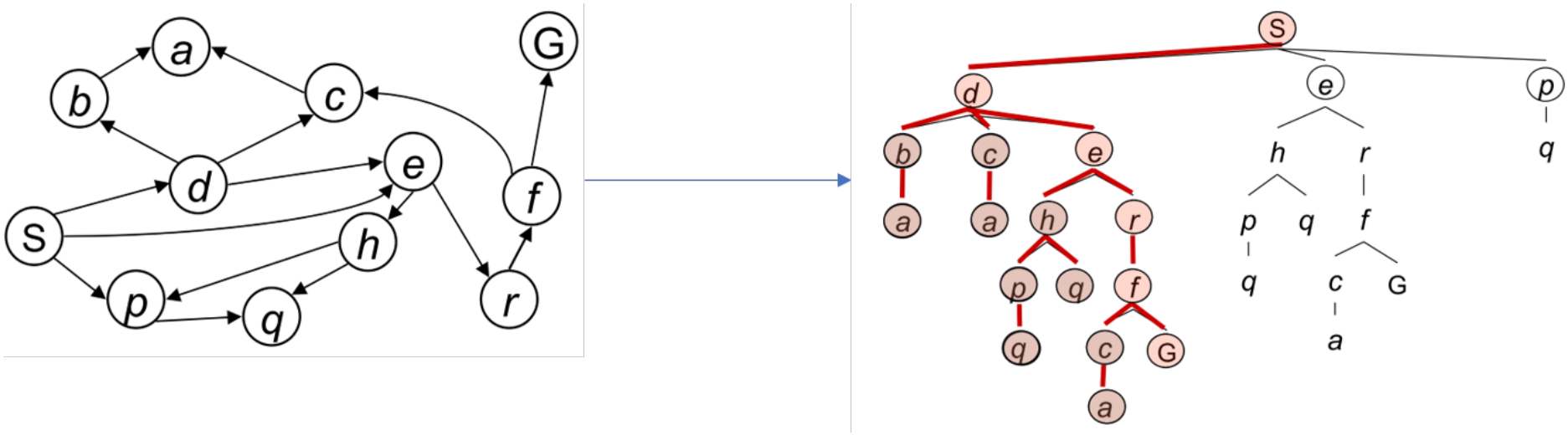
Depth-first search analysis: completeness

- It is complete only if we prevent cycles.
- Suppose that the maximum search depth is m, then m could be infinite. In this case, depth-first search is not complete.
- See an example: after reaching the node D, the next node is A. Therefore, there is a cycle so that the depth m will be infinite. Hence, we can not find a solution.



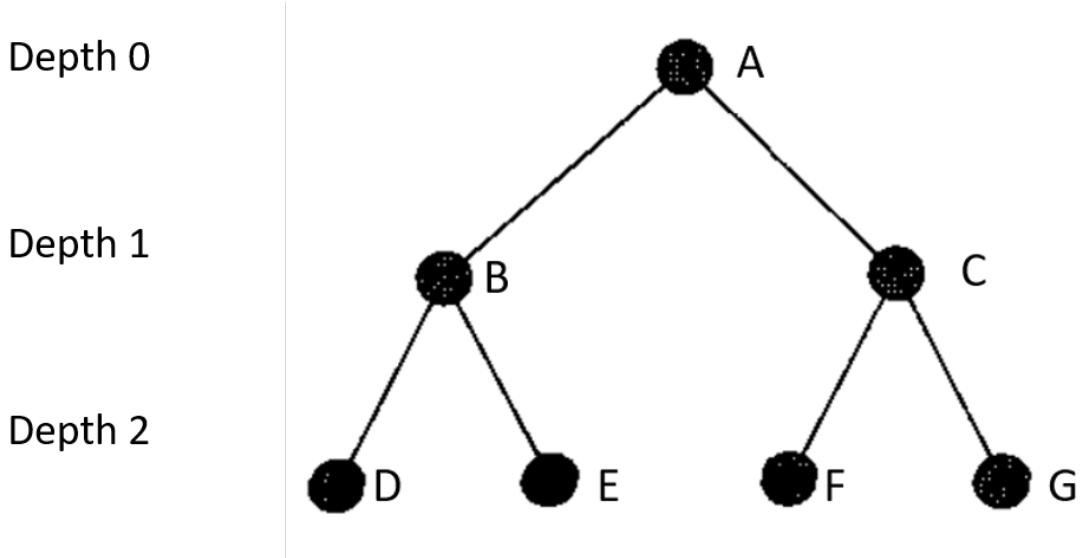
Depth-first search analysis: optimality

- It is not optimal, because it finds the "leftmost" solution regardless of depth or cost.
- See an example: the search result is S $\rightarrow$ d $\rightarrow$ e $\rightarrow$ r $\rightarrow$ f $\rightarrow$ G. However, S $\rightarrow$ e $\rightarrow$ r $\rightarrow$ f $\rightarrow$ G is a better solution.



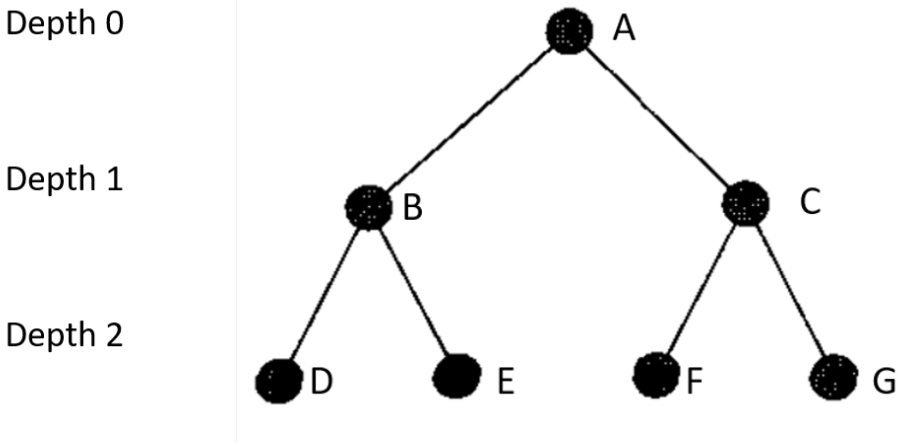
Depth-first search analysis: time complexity

- Time complexity, consider a hypothetical state space where every state has b successors.
- In the worst case, before reaching the goal node G, we traverse the whole search tree:
  - Try path ABD but not find solution.
  - Then, try ABE but not find solution.
  - Then, try ACF but not find solution.
  - Finally, try ACG and get solution.
  - If the maximum depth is m, the time complexity is $O(b^m)$



Depth-first search analysis: space complexity

- Depth-first search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
- The length of the longest path is m. For m nodes down the path, you will have to store b nodes extra for each of the m nodes. Therefore, space complexity is $O(bm)$
- For example, when reach the path ABD, we store A, B, D, E, C



Quiz

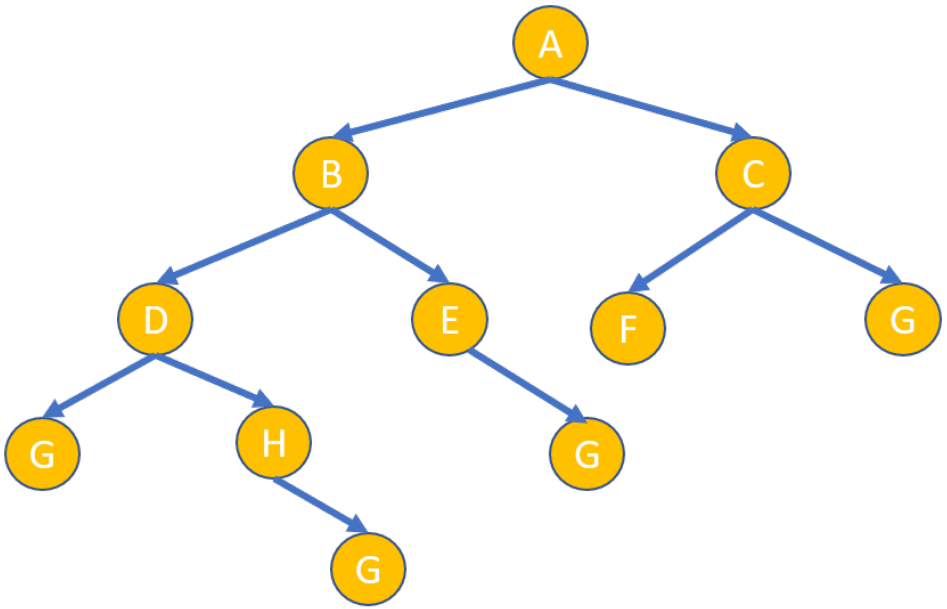
If node G is a goal node, what is the solution if use depth-first search?

A ABDG

B ABDHG

C ABEG

D ACG



Quiz

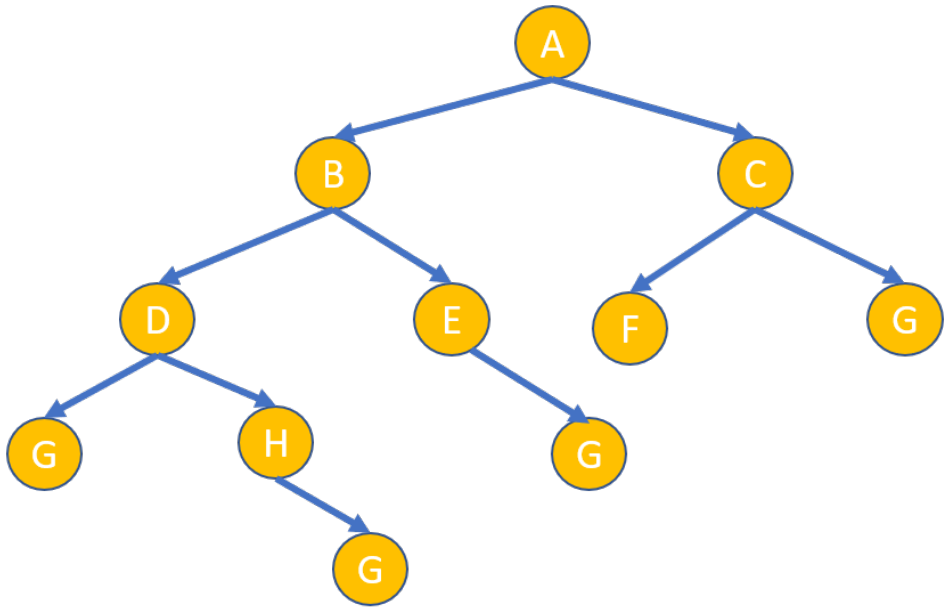
If node G is a goal node, what is the solution if use depth-first search?

A ABDG

B ABDHG

C ABEG

D ACG

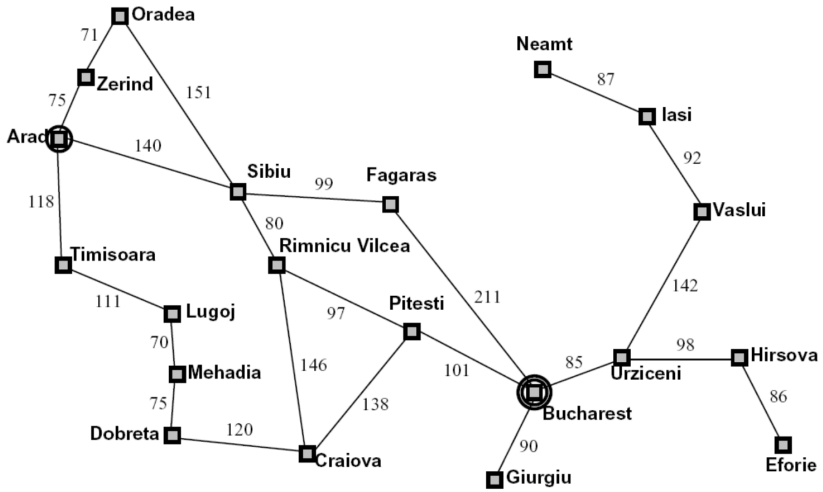


Answer: A

Part 5. Depth-limited search

Depth-limited search

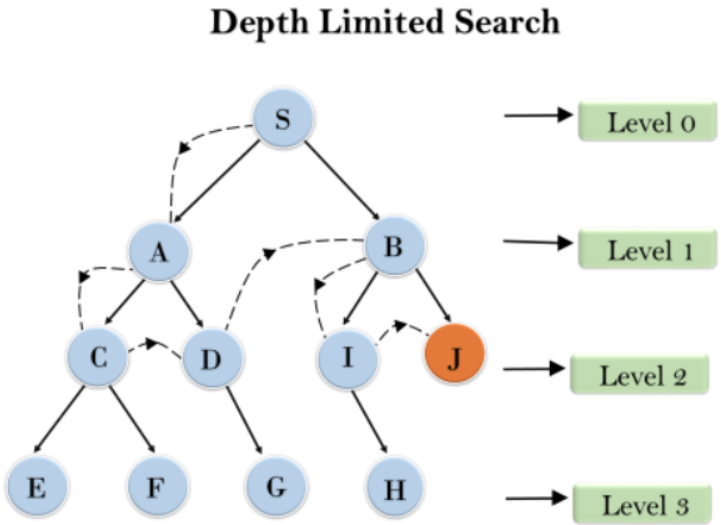
- Depth-limited search supplies depth-first search with a pre-determined depth limit ℓ.
- Nodes at depth ℓ are treated as if they have no successors.
- The depth limit solves the infinite-path problem of depth-first search.
- Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania, there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so $\ell = 19$ is a possible choice for depth limit.



Depth-limited search - an example

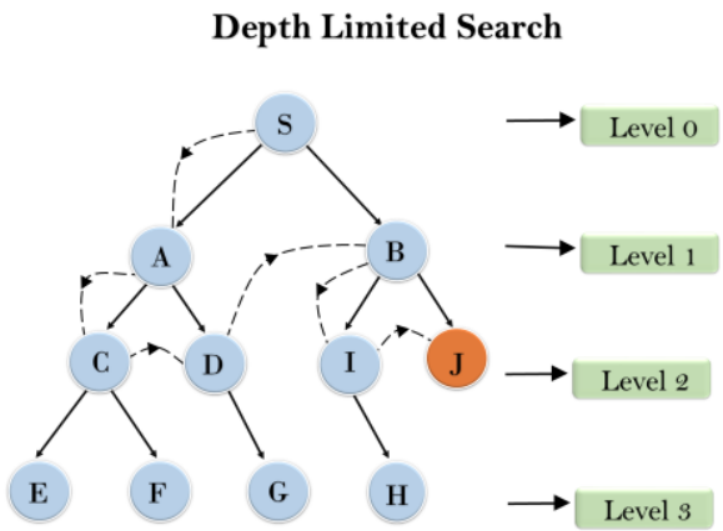
Assume that the goal node is the node J, the depth limit ℓ is 2.

- First, start from root node S and expand node A.
- Then, expand node C. As node C is not the goal node, but its depth is 2, we go back to node A.
- Then, we expand node D. As node D is not the goal node, but its depth is 2, we go back to node S and expand node B.
- Repeat the above steps, until reach the goal node J.



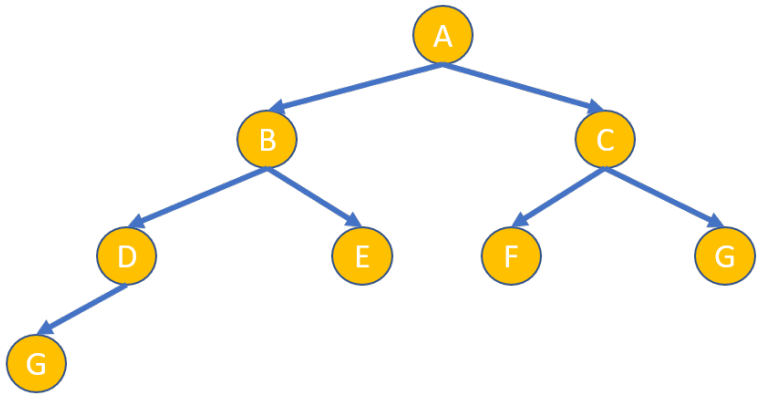
Depth-limited search analysis: completeness

- It introduces a source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit.
- This is not unlikely when d is unknown.



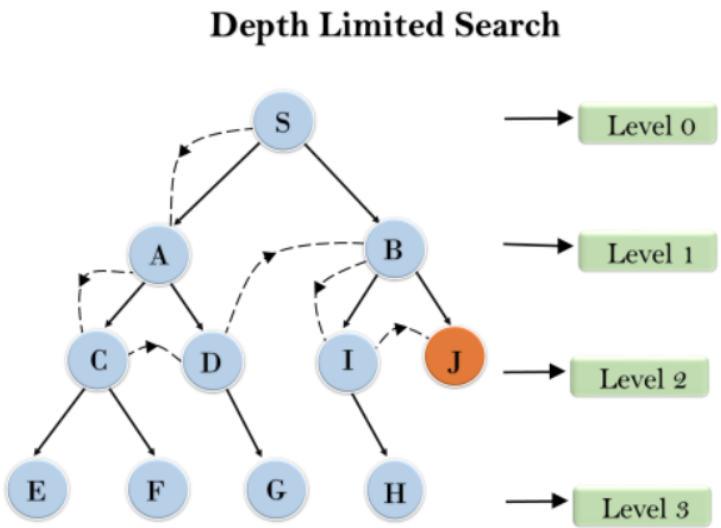
Depth-limited search analysis: optimality

- It is not optimal if we choose $\ell > d$, because it finds the "leftmost" solution regardless of depth or cost.
- The following is an example. The optimal solution depth d is 2. If ℓ is larger than 2, the solution will be $ABDG$. However, the optimal solution is ACG



Depth-limited search analysis: time & space complexity

- Time complexity: consider a hypothetical state space when every state has $\boldsymbol{b}$ successors, depth limit is $\boldsymbol{\ell}$.
- Similarly to depth-first search, in the worst case, before reaching the goal node $\boldsymbol{J}$, we traverse the whole search tree with depth $\boldsymbol{\ell}$.
- The time complexity is $O(\boldsymbol{b}^{\boldsymbol{\ell}})$
- The space complexity is $O(\boldsymbol{b\ell})$



Part 6. Iterative deepening depth-first search

Iterative deepening depth-first search

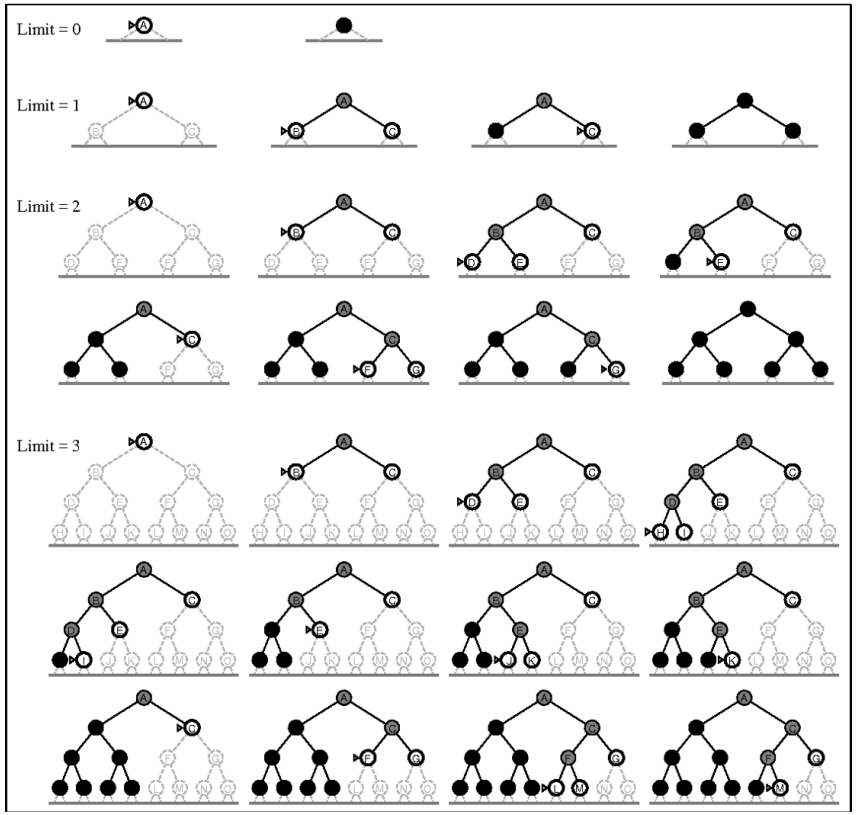
- Iterative deepening search is a general strategy, which is often used in combination with depth-first search, that finds the best depth limit.
- Gradually increase the depth limit (e.g., first 0, then 1, then 2, and so on) until a goal is found.
- The solution is found when depth limit reaches d, which is the depth of the shallowest goal node.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH (problem, depth)
      if result ≠ cutoff then return result
```

Iterative deepening depth-first search - an example

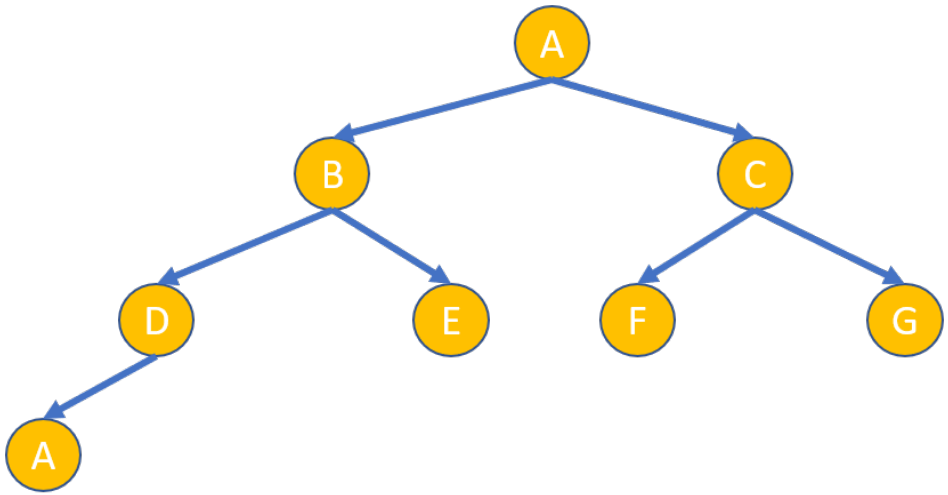
- Try depth limit 1, and apply depth-first search. If no solution,
- Try depth limit 2, and apply depth-first search. If no solution.
- Try depth limit d, and finally get solution if a solution exists.



Four iterations of iterative deepening

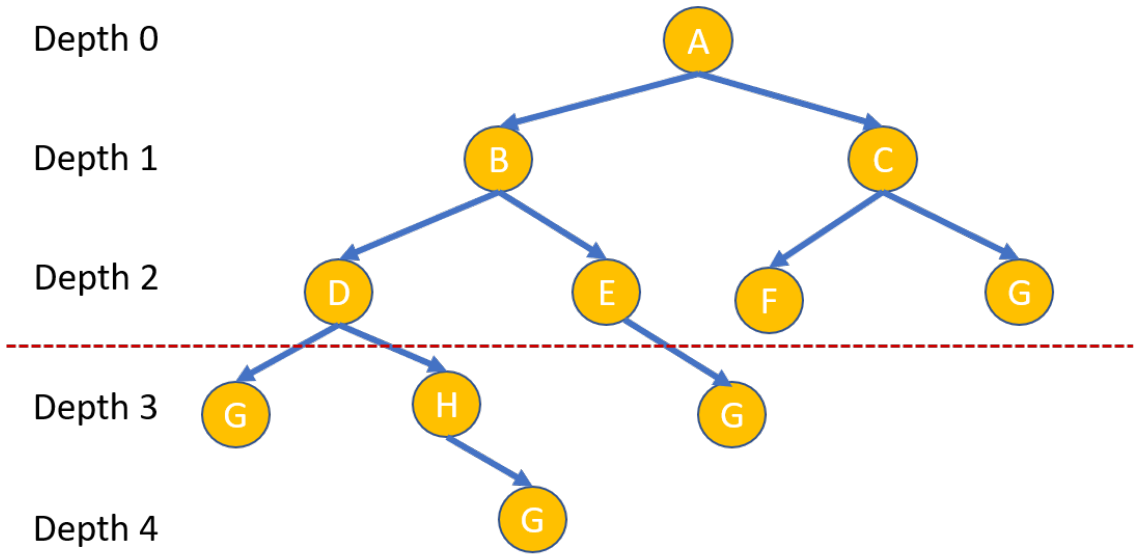
Iterative deepening analysis: completeness

- Iterative deepening combines the benefits of depth-first search and breadth-first search.
- Like breadth-first search, it is complete when branching factor is finite.
- See an example, node G is a goal node, though it has cycle, we can find a solution when depth is 2.
  - Run a depth-first search with depth limit 0. If no solution,
  - Run a depth-first search with depth limit 1. If no solution,
  - Run a depth-first search with depth limit 2, find a solution.



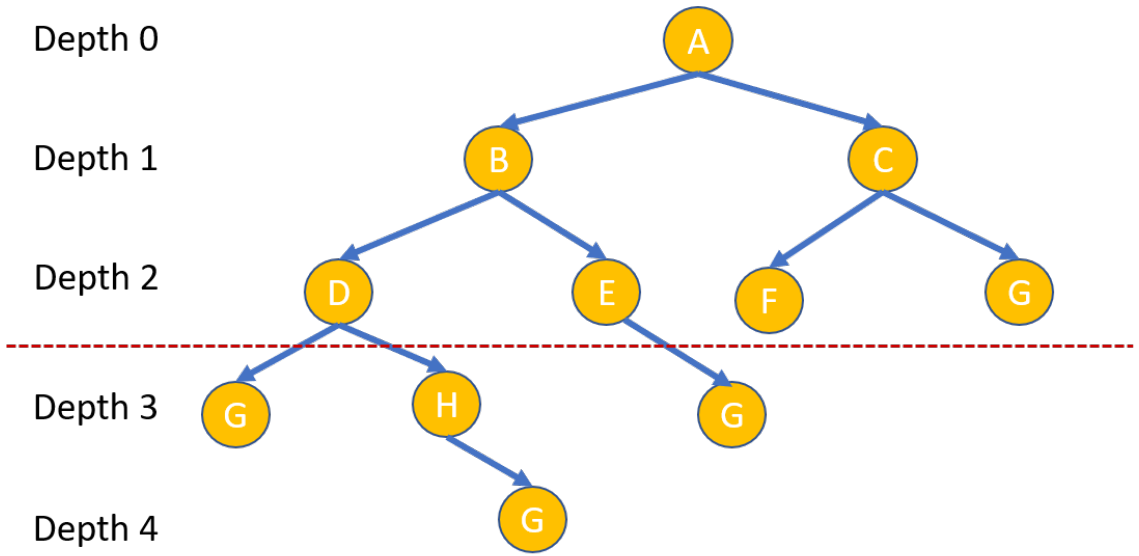
Iterative deepening analysis: optimality

- Like breadth-first search, it is optimal when the path cost is a non-decreasing function of the depth of the node.
- See an example, node G is a goal node. When we have one goal node at depth 2, depth 3 and depth 4 :
  - Run a depth-first search with depth limit 1. If no solution
  - Run a depth-first search with depth limit 2, find a solution.
  - When get the first solution, the depth is smallest one, so it is optimal.



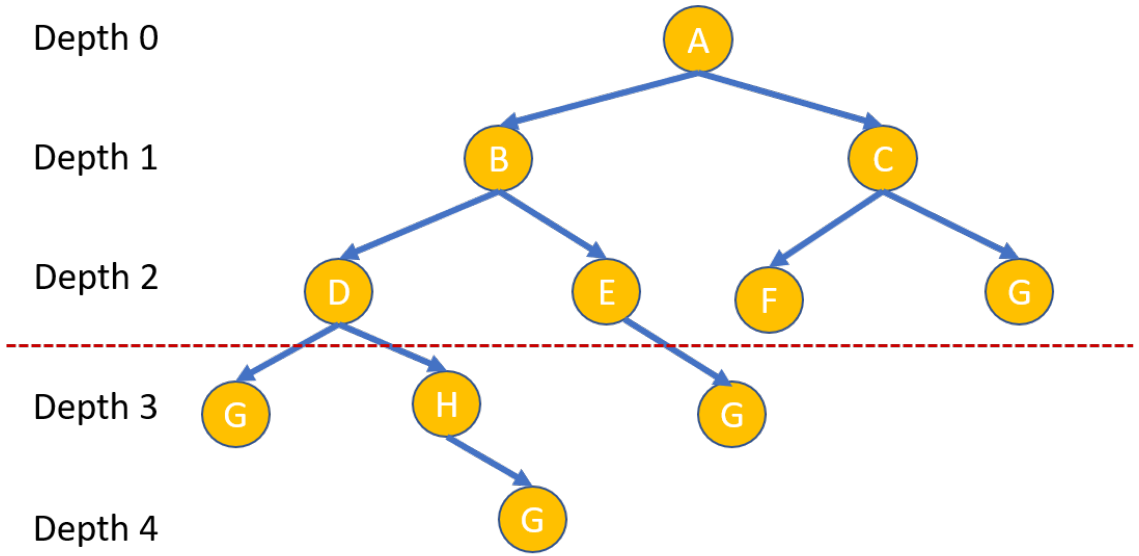
Iterative deepening analysis: time complexity

- Time complexity: the nodes on bottom level (depth d) are expanded once, those on the next to bottom level are expanded twice, and so on, up to the child of the root, which are expanded d times.
- See an example, node G is a goal node, the solution depth is 2.
  - Node A at depth 0 will be counted 3 times
  - Node B and C at depth 1 will be counted 2 times
  - Node D, E, F and G at depth 2 will be counted 1 time
  - Time complexity is $(d+1)b^0 + db + (d-1)b^2 + ... + b^d = O(b^d)$



Iterative deepening analysis: space complexity

- Like depth-first search, memory requirement of iterative deepening analysis is modest.
- It needs to store only a single path from the root to a leaf node, along with remaining unexpanded sibling nodes for each node on the path
- The space complexity is $O(\boldsymbol{bd})$, when $\boldsymbol{d}$ is the solution depth.



Part 7. Summary

Summary

- Concepts of the searching problem
- Concepts of the searching algorithms
- Uninformed search
  - breadth-first search
  - uniform-cost search
  - depth-first search
  - depth-limited search
  - iterative deepening

Reading materials

- Chapter 3, Artificial Intelligence A Modern Approach by Stuart J. Russell and Peter Norvig
- Slides, CS 188 Introduction to Artificial Intelligence Summer 2021