SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Uninformed Search I

PROF. LIM KWAN HUI

50.021 Artificial Intelligence

The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.

Outline & Objectives

- Learn about five different uninformed search algorithms
  - Breadth-First Search
  - Uniform-cost search
  - Depth-First Search
  - Depth-limited search
  - Iterative deepening search
- Understand the trade-offs in properties between these algorithms
- Being able to use these algorithms to solve a search problem

Recap: Types of Search

- Uninformed Search
  - No additional information about states beyond that in the problem definition (AKA blind search)
- Informed Search
  - Uses problem-specific knowledge beyond the definition of the problem itself
- Adversarial Search
  - Used in multi-agent environment where the agent needs to consider the actions of other agents and how they affect its own performance.

Recap: Search Strategies

- The Expand function creates new nodes (and their various fields)
  - using the Actions and Transition Model to create the corresponding states
- A **search strategy** is defined by picking the *order of node expansion*

Expand step: Determines our search strategy

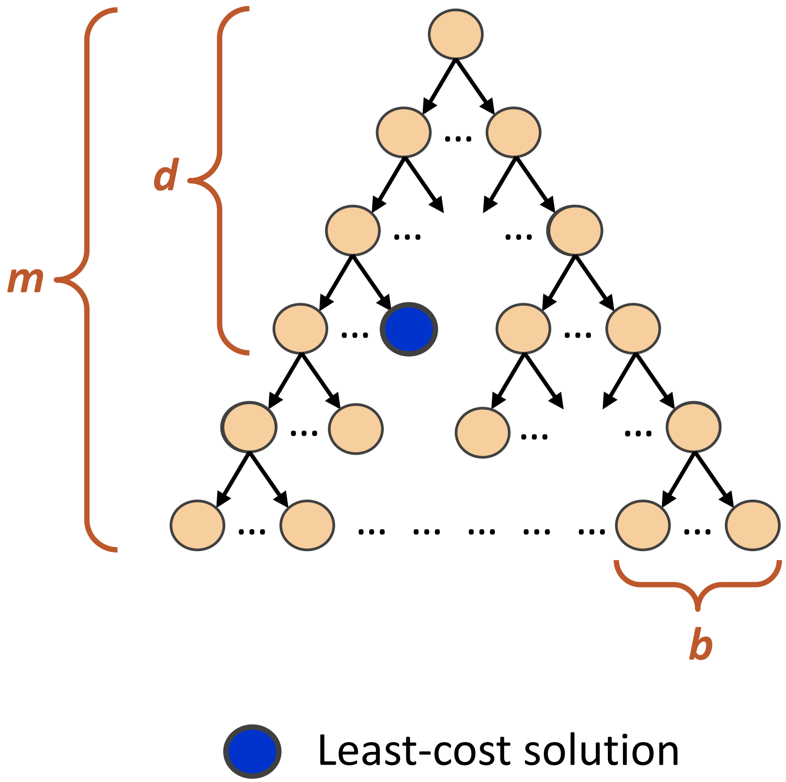
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return**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

Recap: Search Strategies

- Strategies are evaluated along the following dimensions:
  - ***Completeness*** - does it always find a solution if one exists?
  - ***Optimality*** - does it always find a least-cost solution?
  - ***Time complexity*** - number of nodes generated/expanded
  - ***Space complexity*** - maximum number of nodes in memory

Recap: Search Strategies

- Time and space complexity are measured in terms of
  - ***b*** - maximum branching factor of the search tree
  - ***d*** - depth of the least-cost solution
  - ***m*** - maximum depth of the state space (may be infinite)



Recap: Graph vs Tree Search

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

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

Types of Uninformed Search

- Breadth-First Search
- Uniform-cost search
- Depth-First Search
- Depth-limited search
- Iterative deepening search

Search Strategies

- A **search strategy** is defined by picking the *order of node expansion*
- How do we implement this?

Expand step: Determines our search strategy

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

Search Strategies

- A **search strategy** is defined by picking the *order of node expansion*
- How do we implement this?
  - Using different types of queue structures to represent the frontier

Expand step: Determines our search strategy

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

Search Strategies

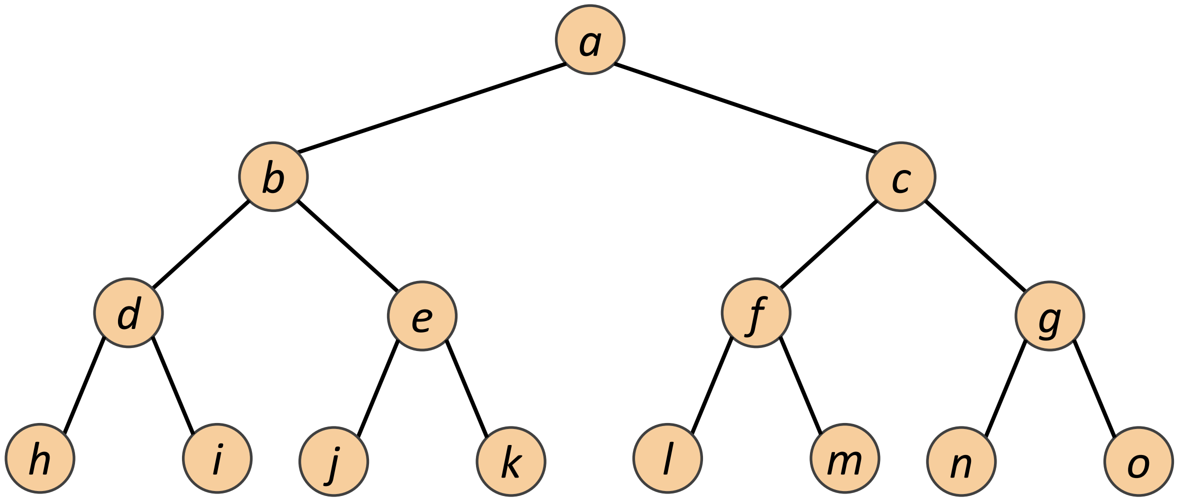
- A **search strategy** is defined by picking the *order of node expansion*
- How do we implement this?
  - Using different types of queue structures to represent the frontier
  - *Order of node expansion = Adding/removal sequence in queue*

Expand step: Determines our search strategy

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

Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

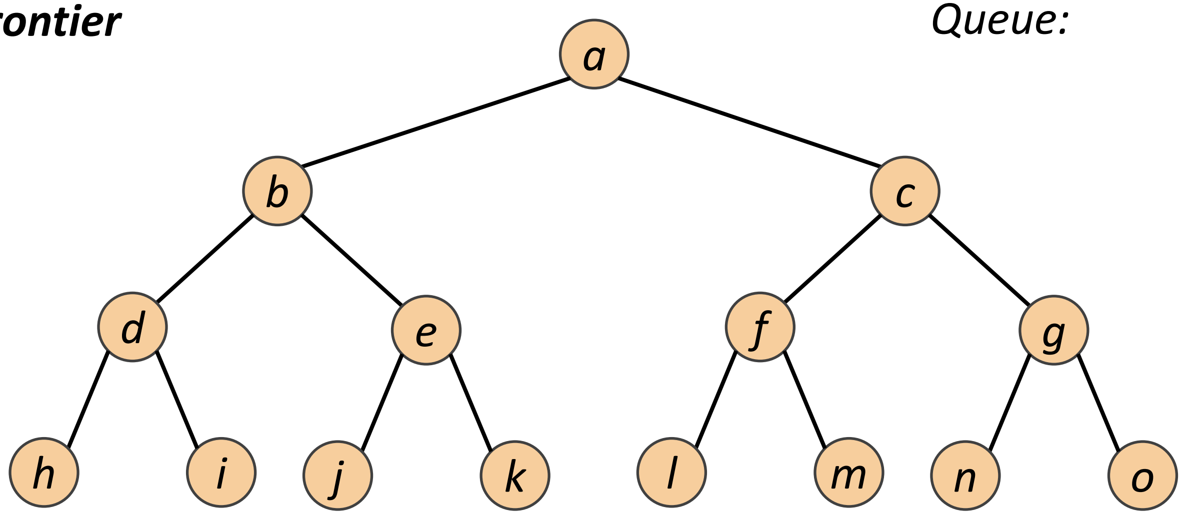


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Initialize frontier

Queue:

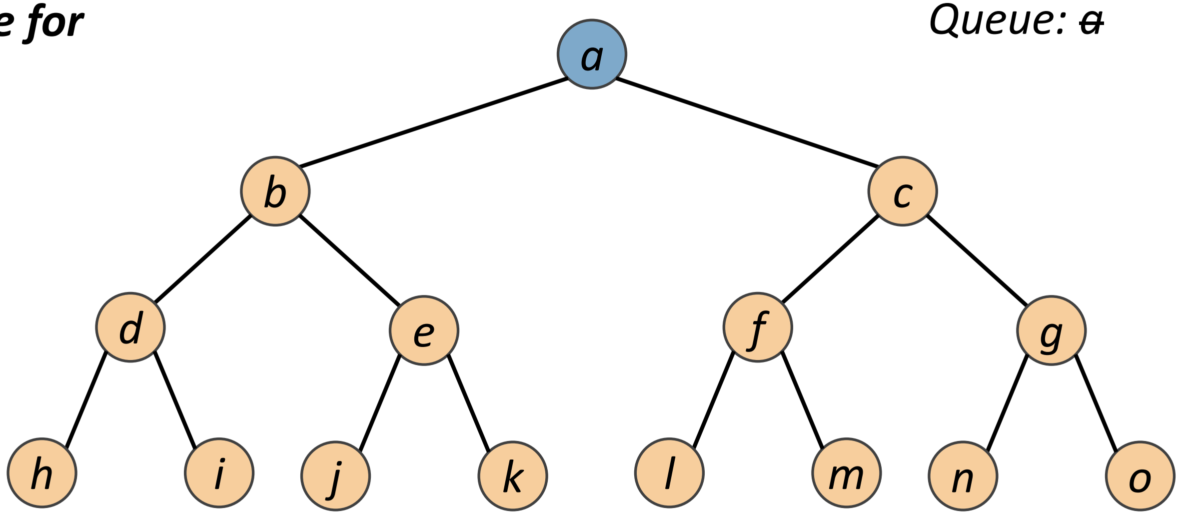


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Check node for goal state

Queue: ~~a~~

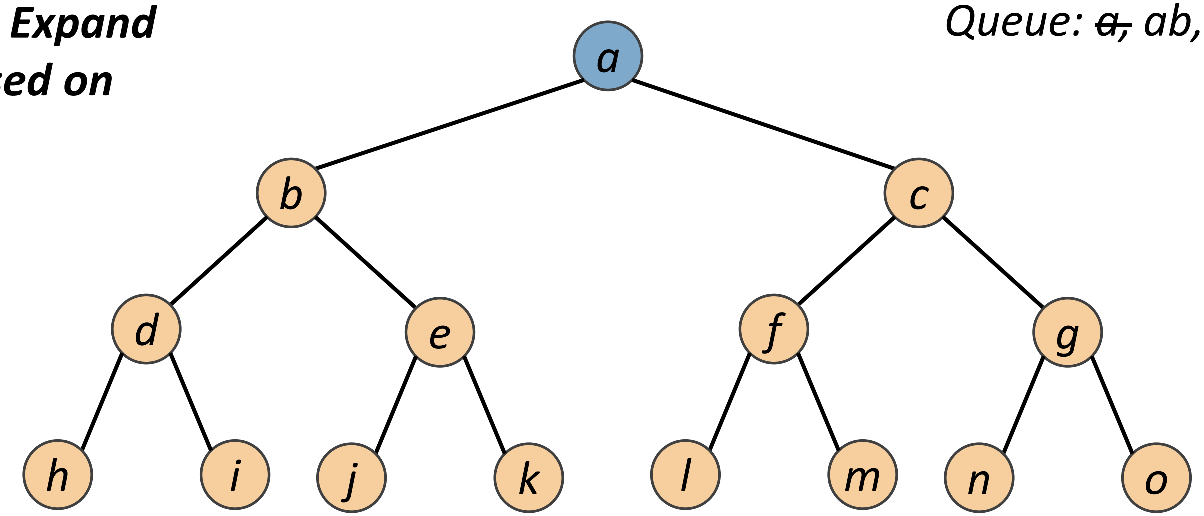


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Not goal? Expand nodes based on strategy

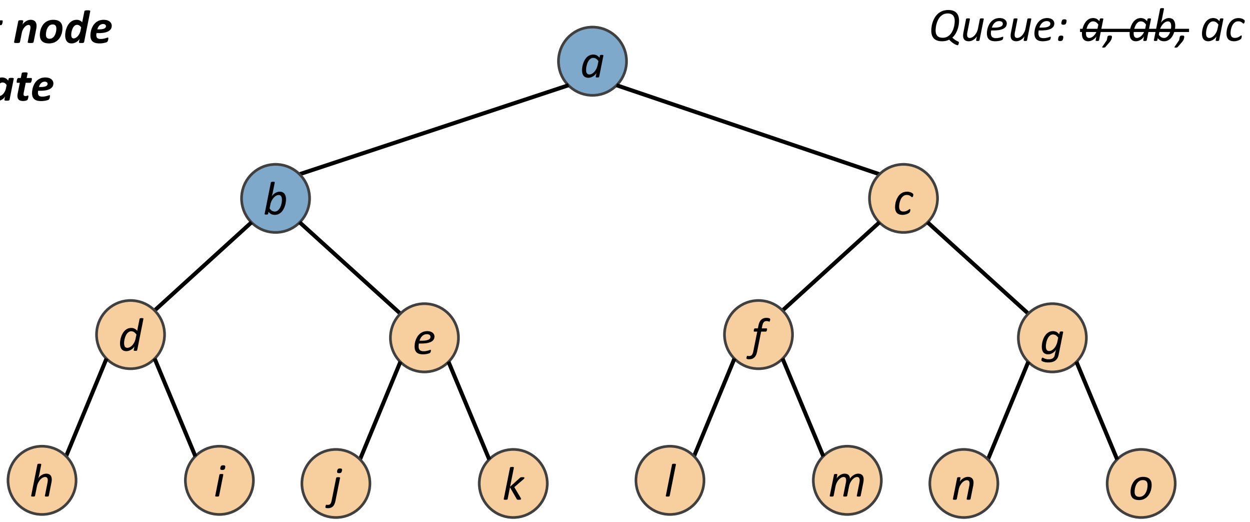
Queue: ~~a,~~ ab, ac



Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Check next node for goal state

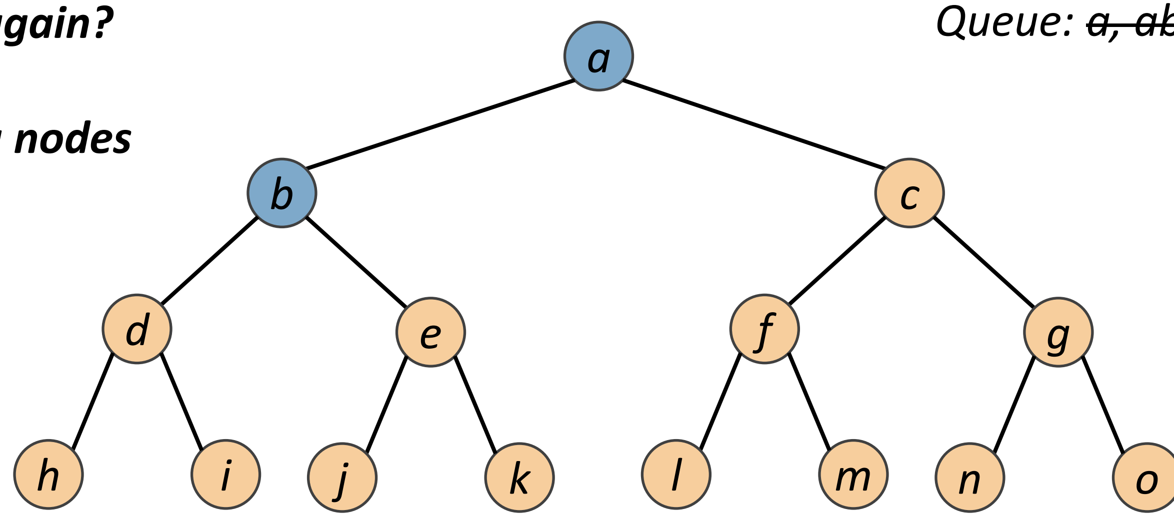


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Not goal again? Continue expanding nodes

Queue: ~~a, ab,~~ ac, abd, abe

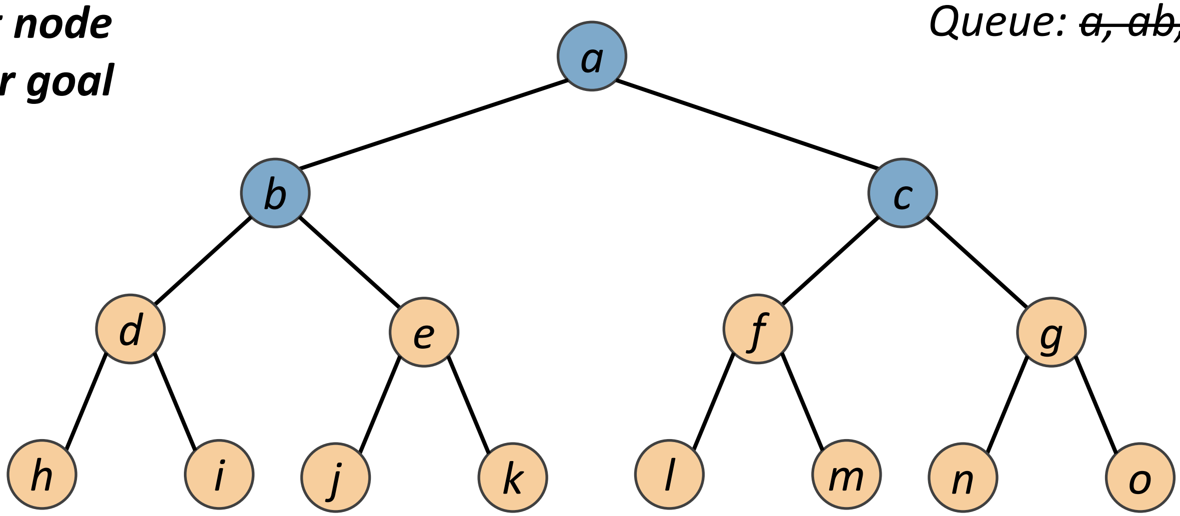


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Check next node (again!) for goal state

Queue: ~~a, ab, ac,~~ abd, abe

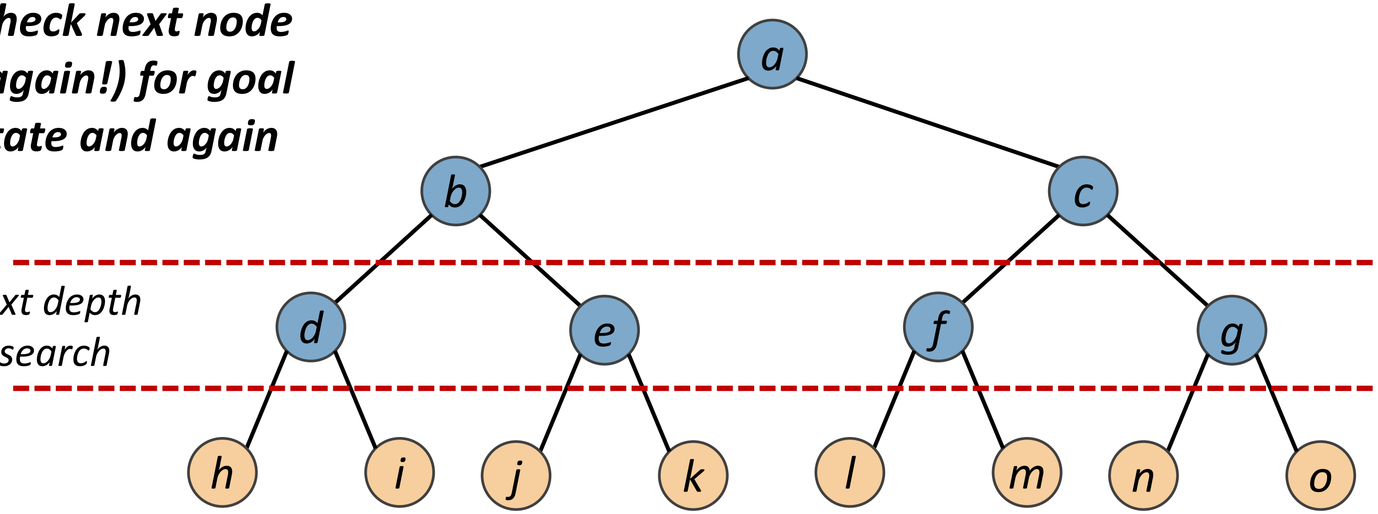


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Check next node (again!) for goal state and again

Next depth to search

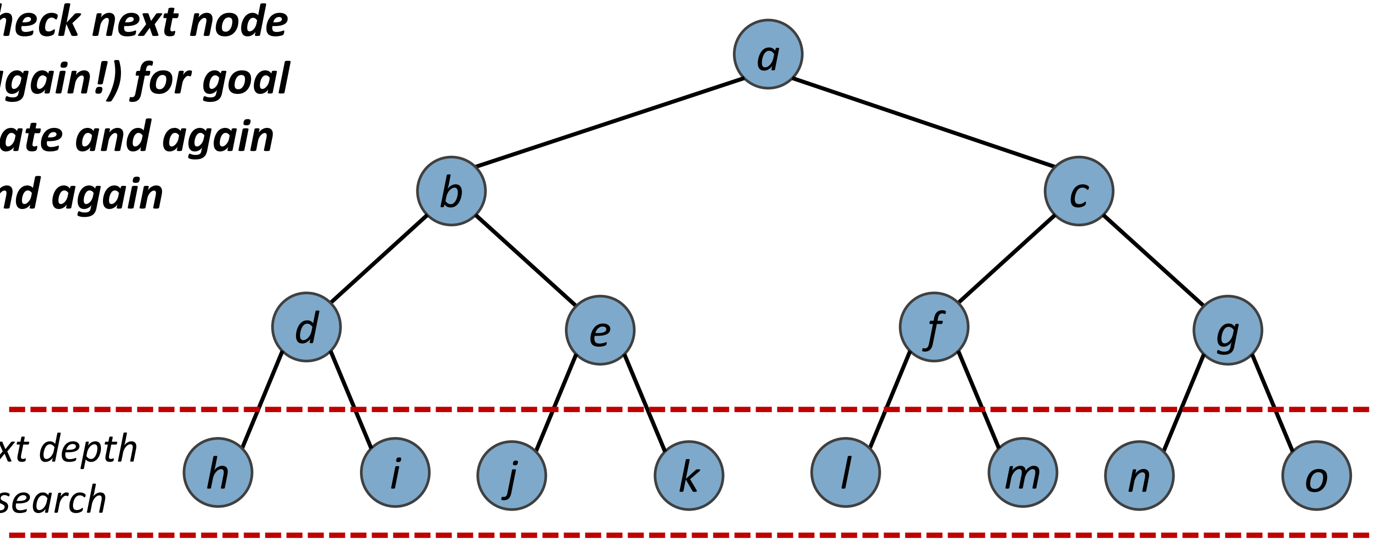


Breadth-First Search

- General idea: Expand the *shallowest* unexpanded node
- Implementation: Using a *First-In First-Out (FIFO)* queue

Check next node (again!) for goal state and again and again

Next depth to search



Breadth-First Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Properties of Breadth-First Search

- Completeness: Yes (if *b* is finite)
- Optimality: Yes (if *cost=1* per step)

  (Not optimal in general)
- Time complexity:
- Space complexity:

Properties of Breadth-First Search

- Completeness: Yes (if b is finite)
- Optimality: Yes (if *cost=1* per step)
  (Not optimal in general)
- Time complexity: $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space complexity: $O(b^d)$
  (keeps every node in memory)

Two Issues with BFS

- Memory requirements are a bigger problem for Breadth-First Search than is the execution time
- Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances

Depth	Nodes	Time	Memory
2	10^2	0.11 milliseconds	107 kilobytes
4	10^4	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabytes
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabytes
14	10^{14}	3.5 years	99 petabytes
16	10^{14}	350 years	10 exabytes

Assuming branching factor b=10; 1 million nodes/second; 1000 bytes/node

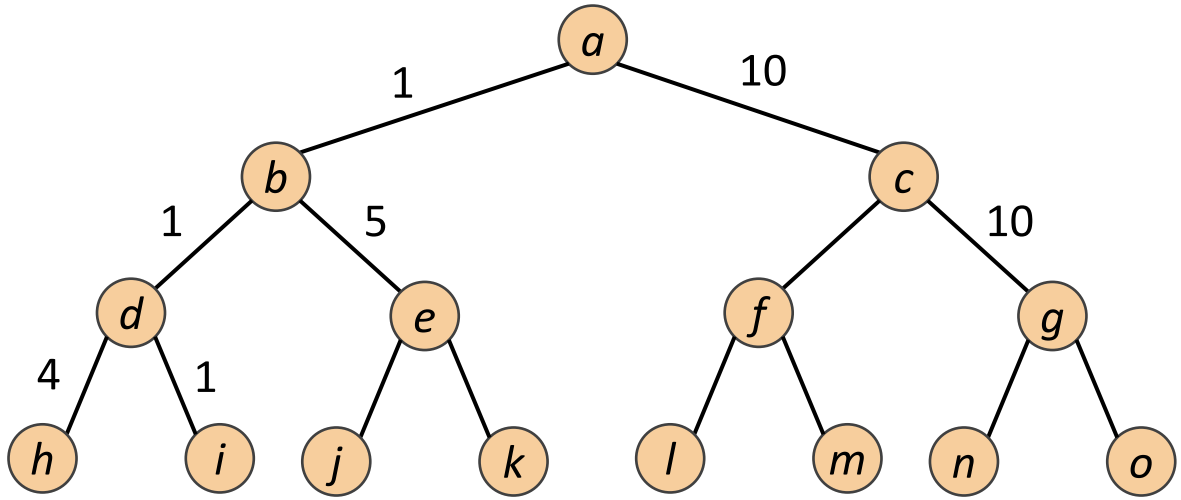
Properties of Breadth-First Search

- Completeness: Yes (if *b* is finite)
- Optimality: Yes (if *cost=1* per step)
  (Not optimal in general)
- Time complexity: $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space complexity: $O(b^d)$
  (keeps every node in memory)

What if step cost!=1 or is non-uniform?

Uniform Cost Search

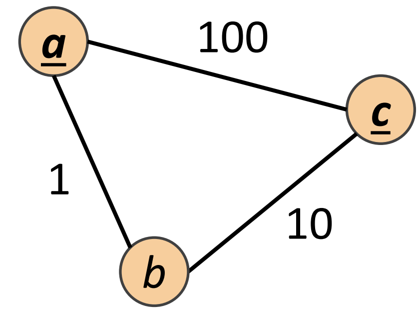
- General idea: Expand unexpanded node *n* with the *lowest path cost g(n)*
- Implementation: Using a *priority* queue ordered by *path cost g*



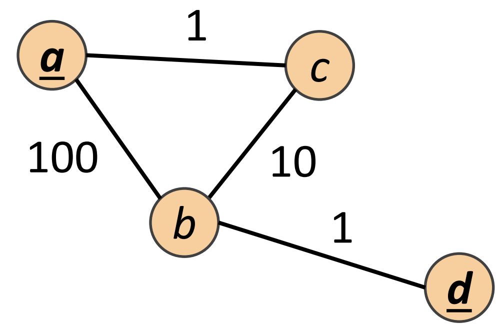
Uniform Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Uniform Cost Search




```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```



Uniform Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Properties of Uniform Cost Search

- Completeness: Yes

  (if *step cost* > *ε,* some positive constant)
- Optimality: Yes
- Time complexity:
- Space complexity:

Properties of Uniform Cost Search

- Completeness: Yes
  (if *step cost* > ε, some positive constant)
- Optimality: Yes
- Time complexity: $O(b^{C^*/\varepsilon})$
  Given C^* = cost of optimal solution
- Space complexity: $O(b^{C^*/\varepsilon})$

Summary: Uninformed Search

- Breadth-First Search
- Uniform-cost search
- Depth-First Search
- Depth-limited search
- Iterative deepening search