COMPX216/Y05337
Artificial Intelligence

Uninformed search

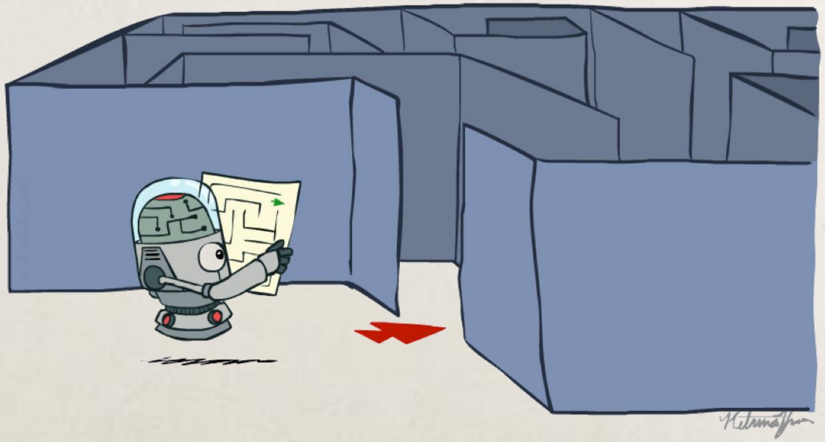
Credits: Prof. Eibe Frank (UoW)

Today: Uninformed search

- Uninformed vs. heuristic search
- Generic best-first search
- The role of the queue
- Priority queues
- The generic best-first search algorithm
- An instantiation: depth-first search
- Actions with different costs: uniform-cost search

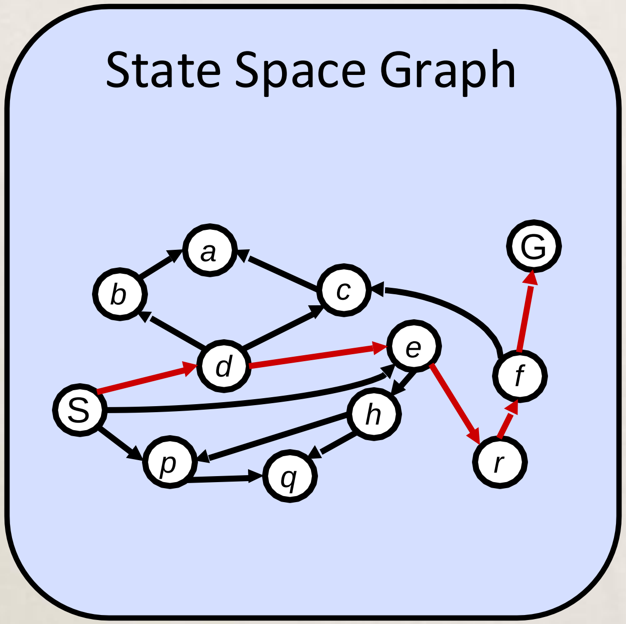
Recap: Search

- Search problem:
  - State space graph is its mathematical representation
  - States (configurations of the world)
  - Actions and costs
  - Transition model (world dynamics)
  - Start state and goal test
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)



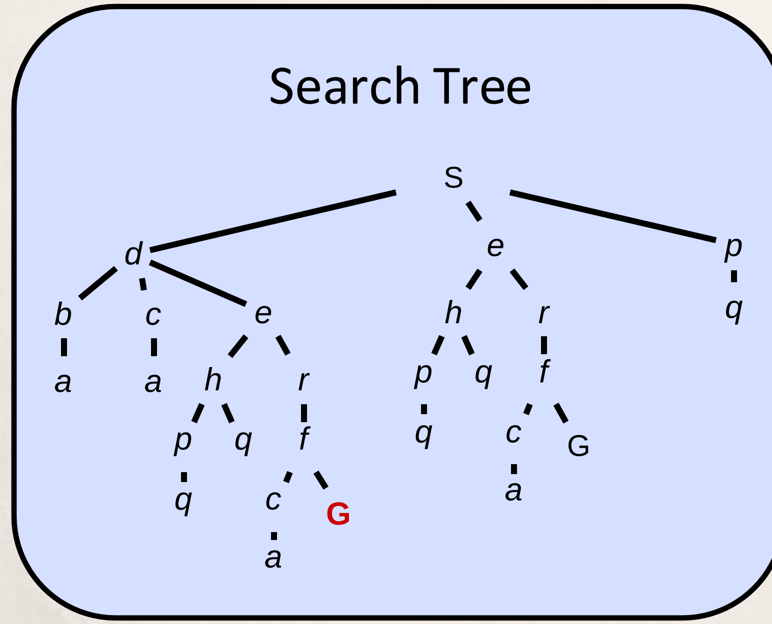
[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Recap: State space graphs vs Search trees



Each NODE in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.



Uninformed vs. heuristic search

- Search algorithms can be distinguished based on whether they make use of a **heuristic function**
- For a given state, the **heuristic function** estimates the cost of the cheapest path from there to a goal state
- In some cases, it is difficult to create a useful heuristic that improves search performance
- **Uninformed search algorithms** explore the state space without the use of such a function
- Breadth-first search is an example of an uninformed search algorithm

Generic best-first search

- Many search algorithms can be cast as instances of a generic **best-first search algorithm**
- This includes uninformed search algorithms as well as heuristic ones
- The algorithms differ in how they define which node amongst the available candidate nodes is "best"
- This definition is used to decide which node in the frontier of the search tree to expand next
- For example, in breadth-first search, the "best" node is the left-most shallowest node in the frontier

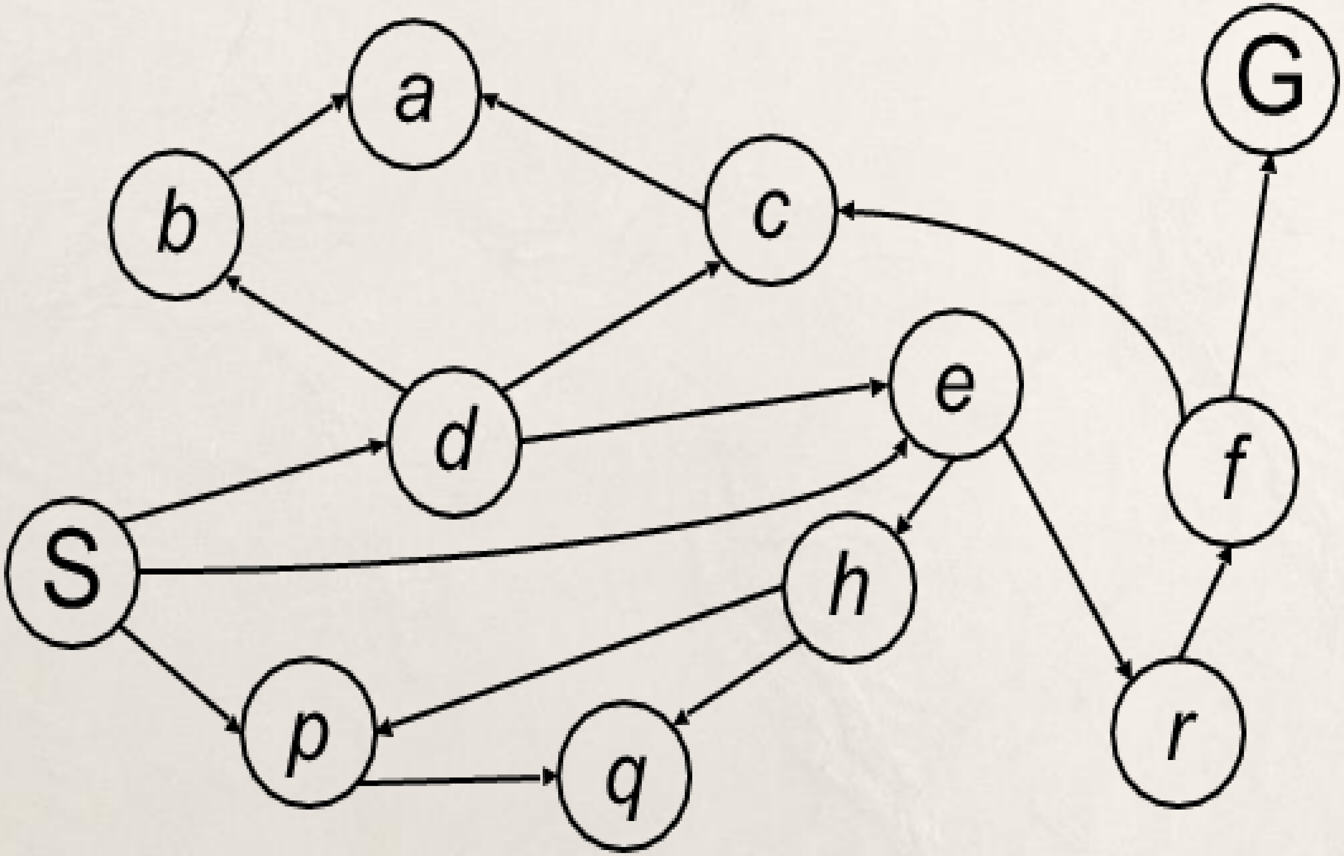
Generic best-first search

```
Function Tree-Search(problem, strategy)
  frontier = {Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = strategy.select_node(frontier)
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Node expansion

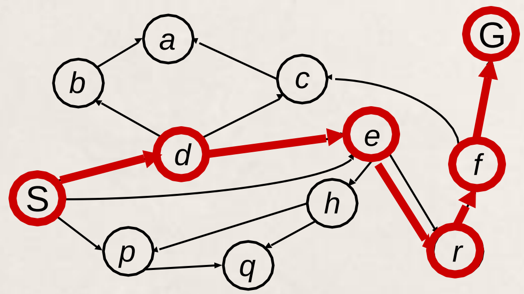
```
Function Expand(node, problem)
    children = {}
    for a : problem.actions(node)
        child.state = Node(node.state.do_action(a))
        child.parent = node
        child.path_cost = node.path_cost + a.cost
        child.action = a
        child.depth = node.depth + 1
        children.add(child)
    return children
```

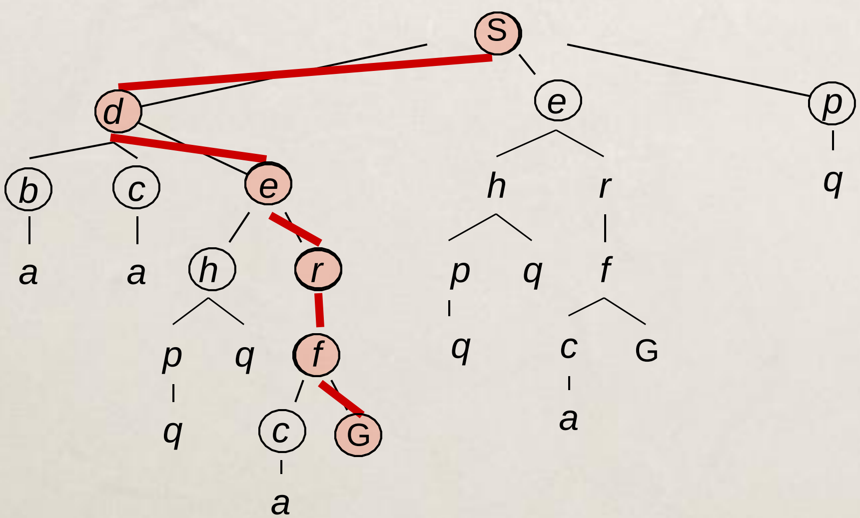
Example: Tree Search



[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Example: Tree Search





s

s → d

s → e

s → p

s → d → b

s → d → c

~~s → d → e~~

s → d → e → h

~~s → d → e → r~~

~~s → d → e → r → f~~

s → d → e → r → f → c

~~s → d → e → r → f → G~~

[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Generic best-first search

- Important ideas:
  - Frontier
  - Expansion
  - Exploration strategy

- Main question: which frontier nodes i.e., "best" to explore?

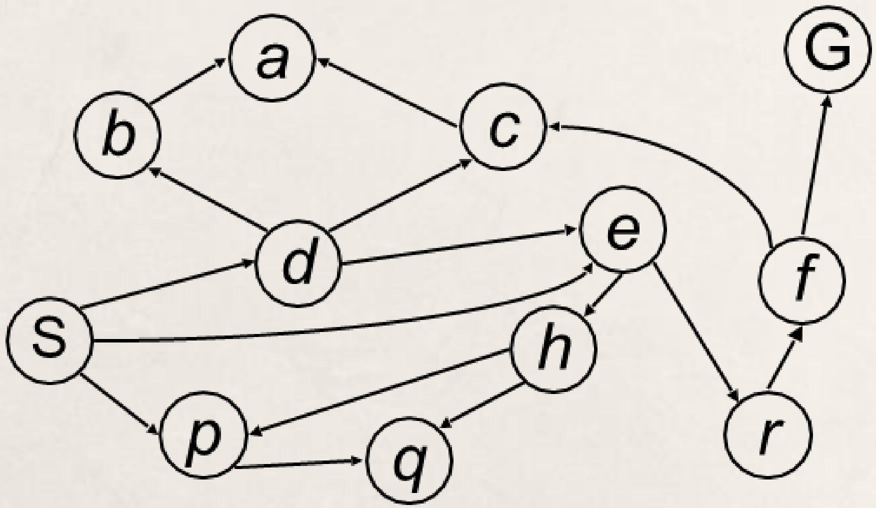
The breadth-first search (BFS) algorithm

- Root node expanded first
- Root children expanded next
- Their children next … etc.
- BFS in general
    - All nodes at a given depth/level are expanded before any nodes in the next level
- Uses a queue (FIFO) for frontier

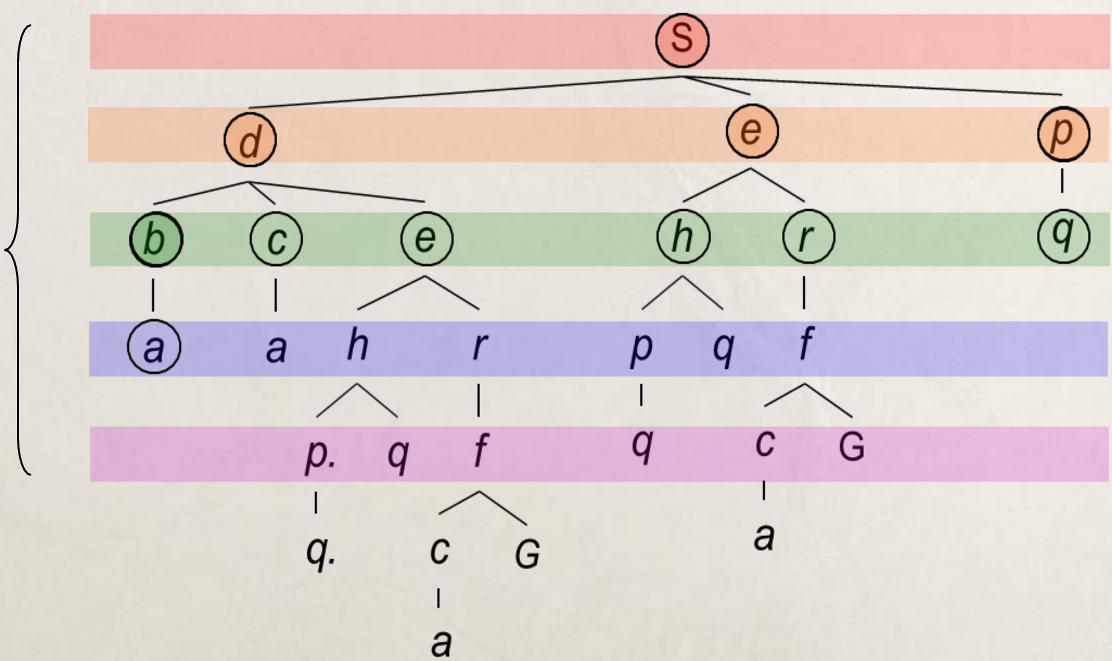
Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Frontier is a FIFO queue



Search Levels/Tiers



[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

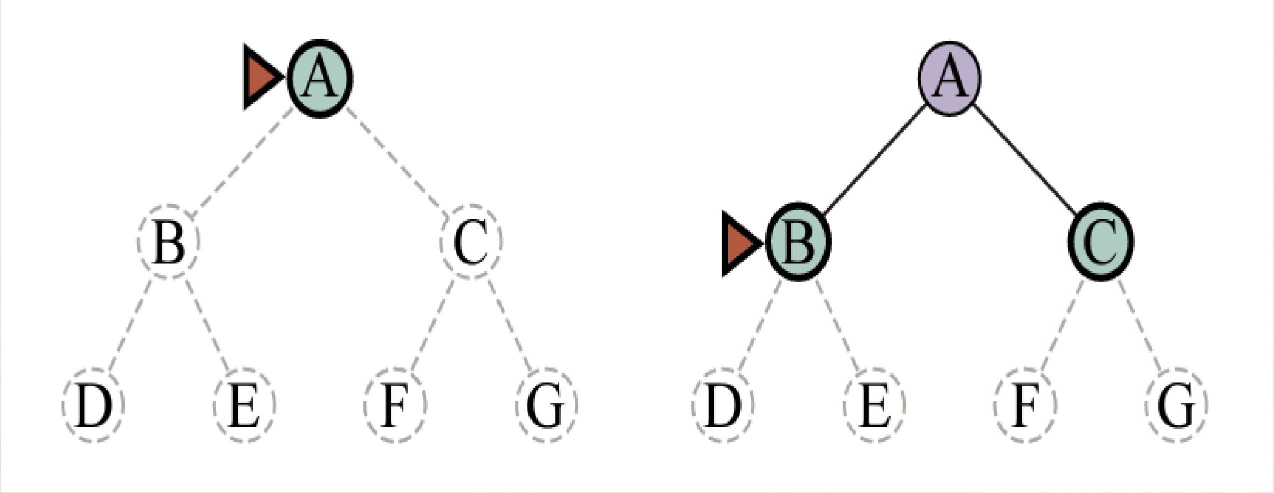
Generic best-first search

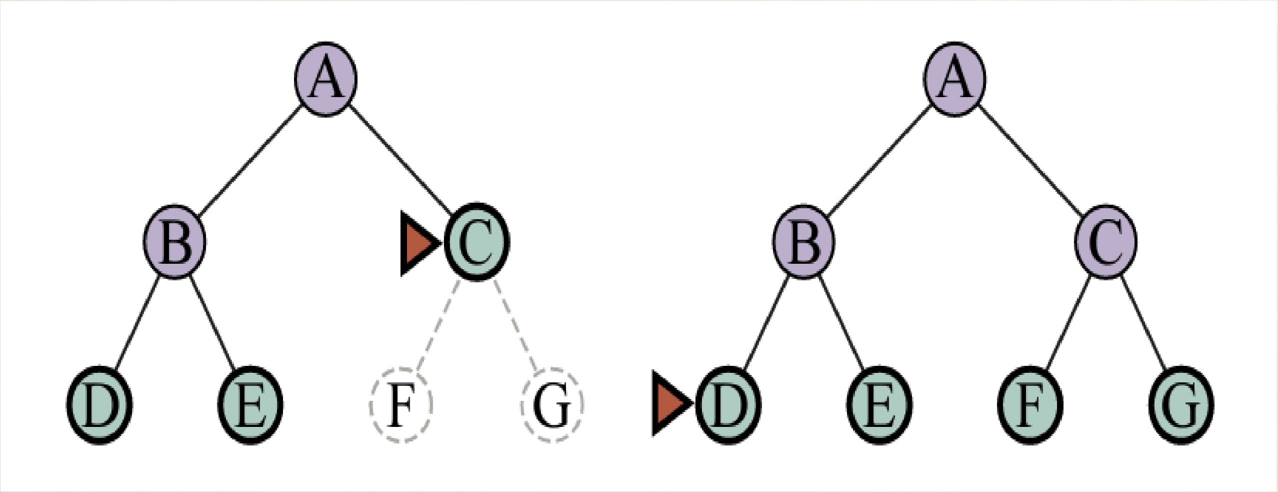
```
Function Tree-Search(problem, strategy)
  frontier = {Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = strategy.select_node(frontier)
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Breadth-first search (BFS)

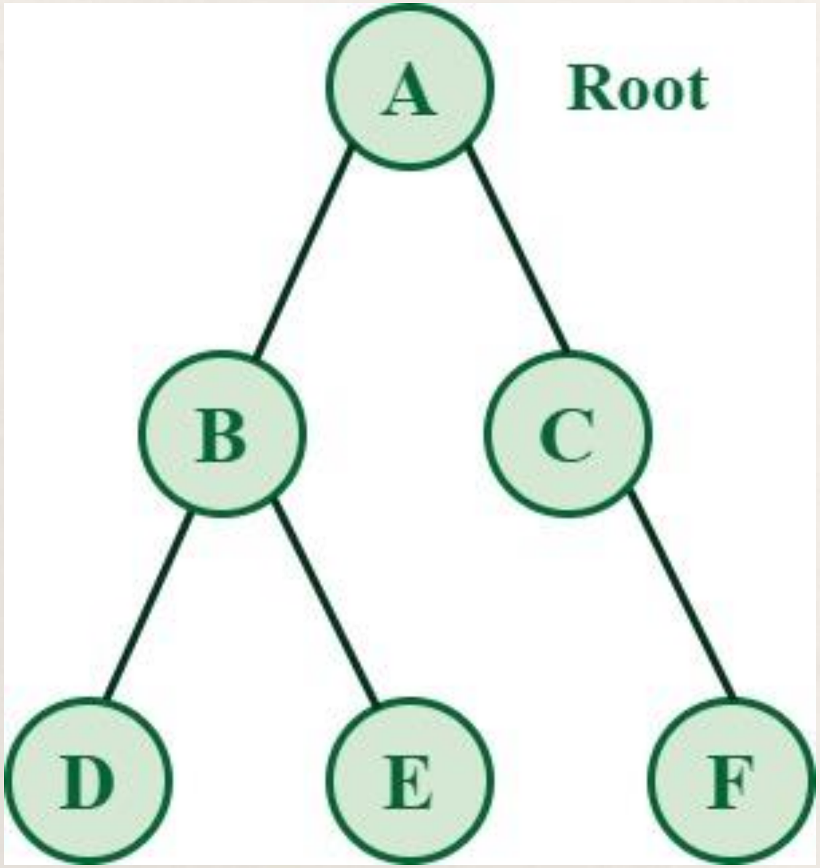
```
Function Tree-Search(problem)
  frontier = Queue{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop()
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

The breadth-first search algorithm

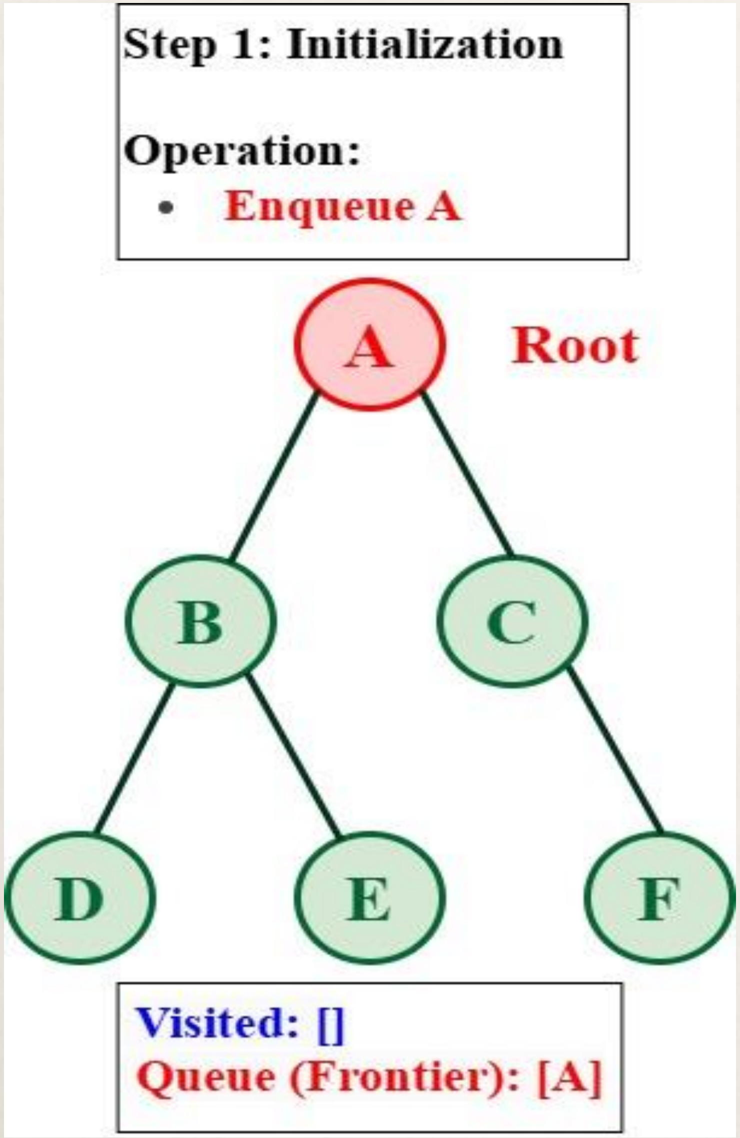




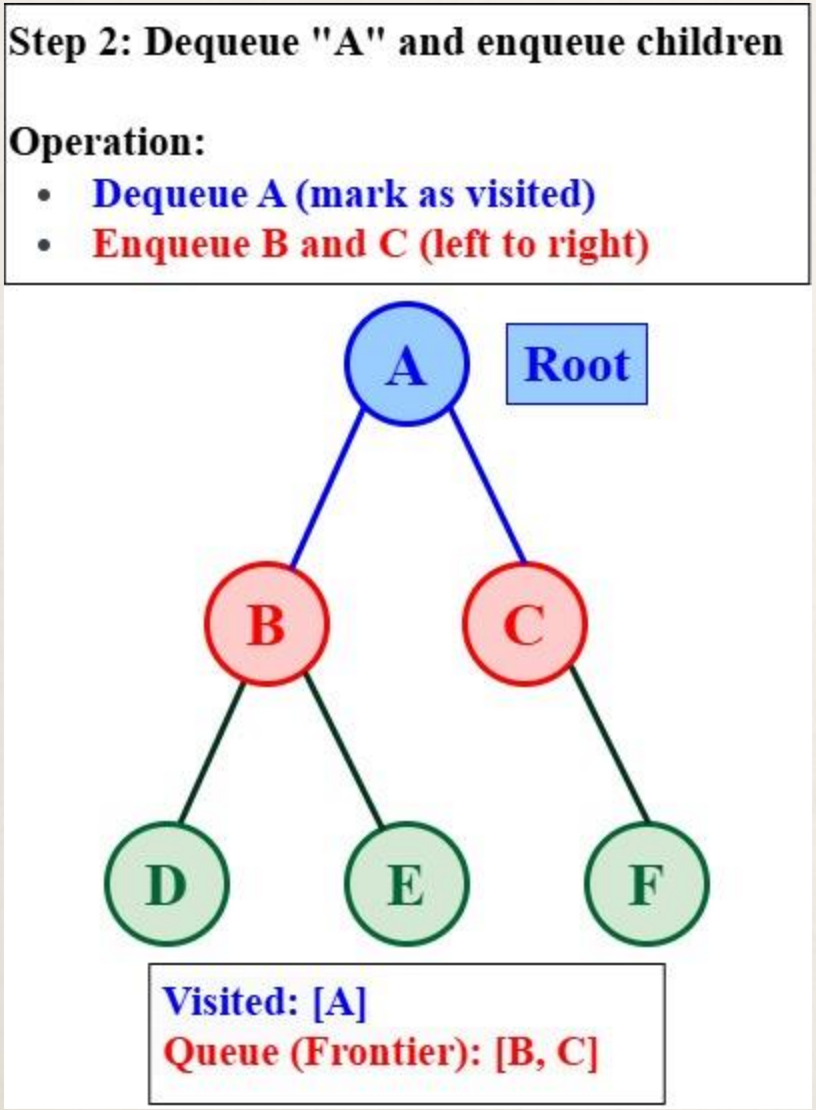
Breadth-first search traversal



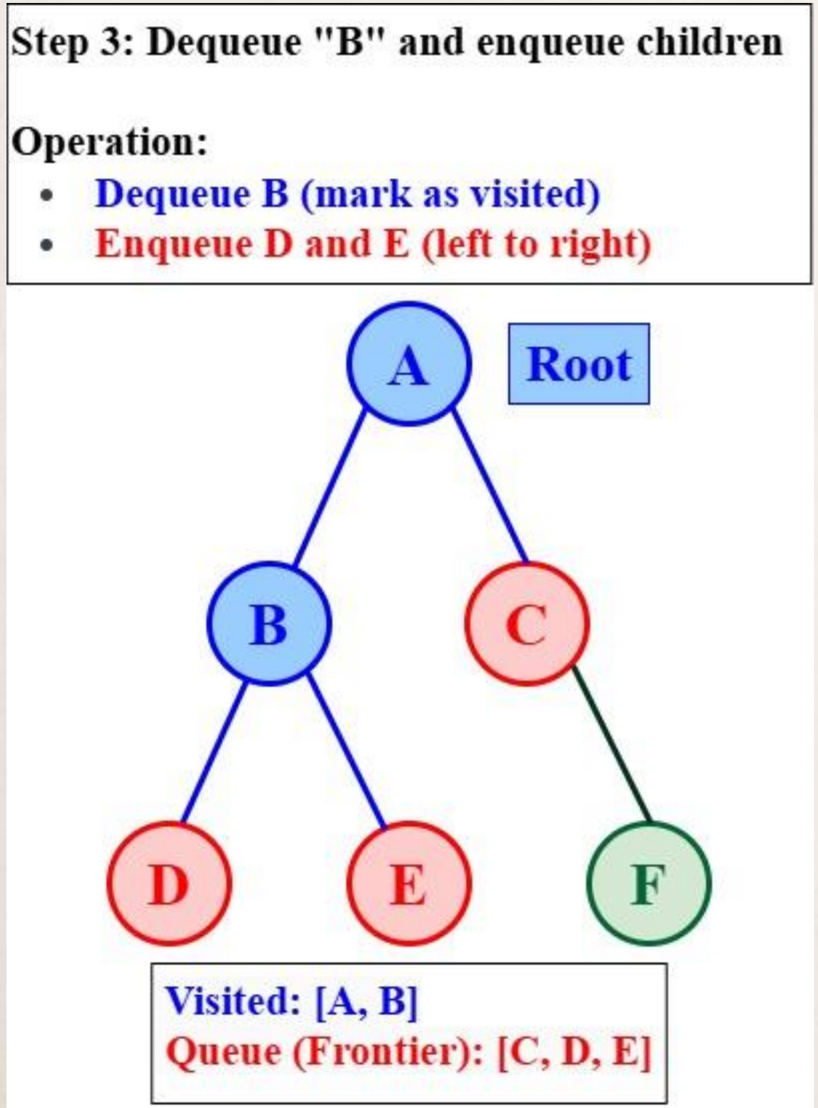
Breadth-first search traversal



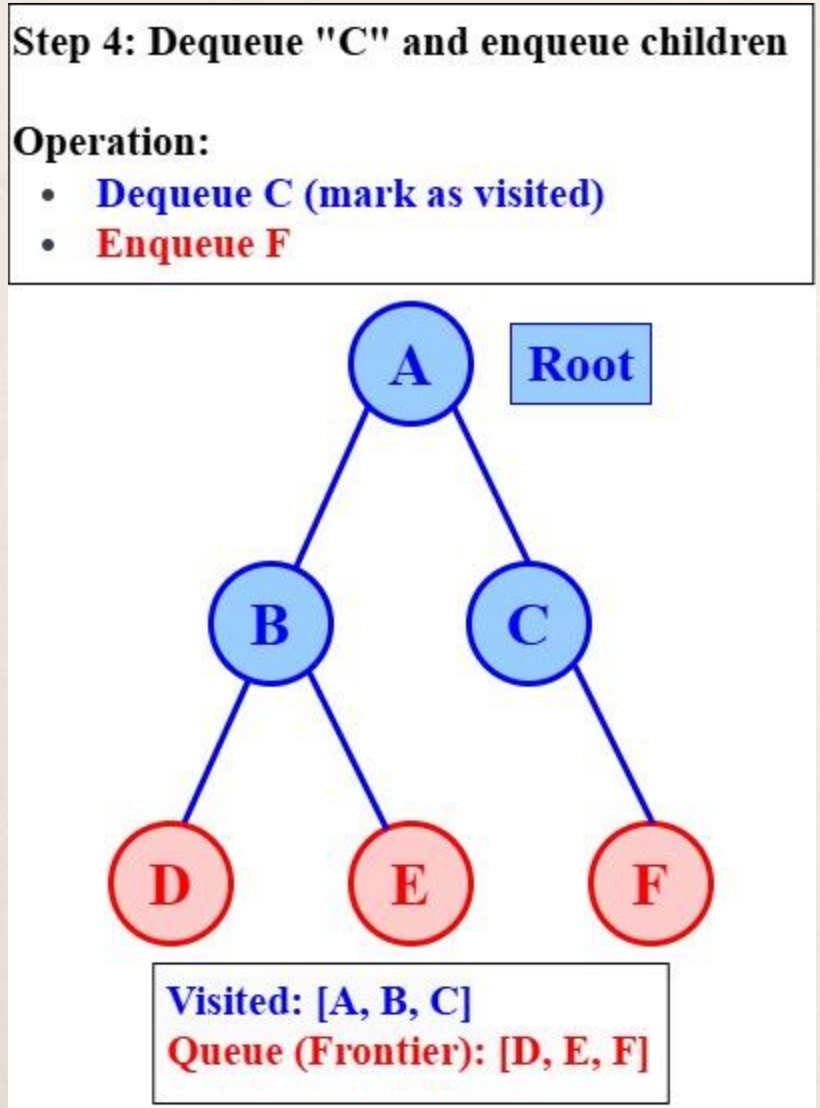
Breadth-first search traversal



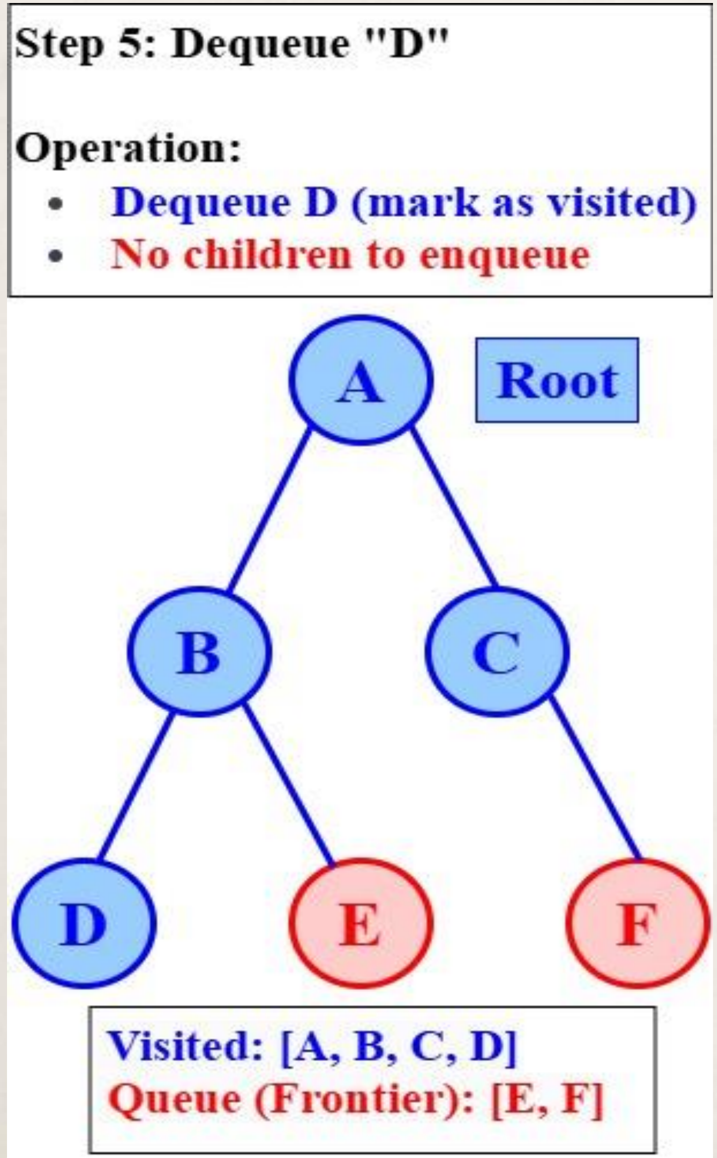
Breadth-first search traversal



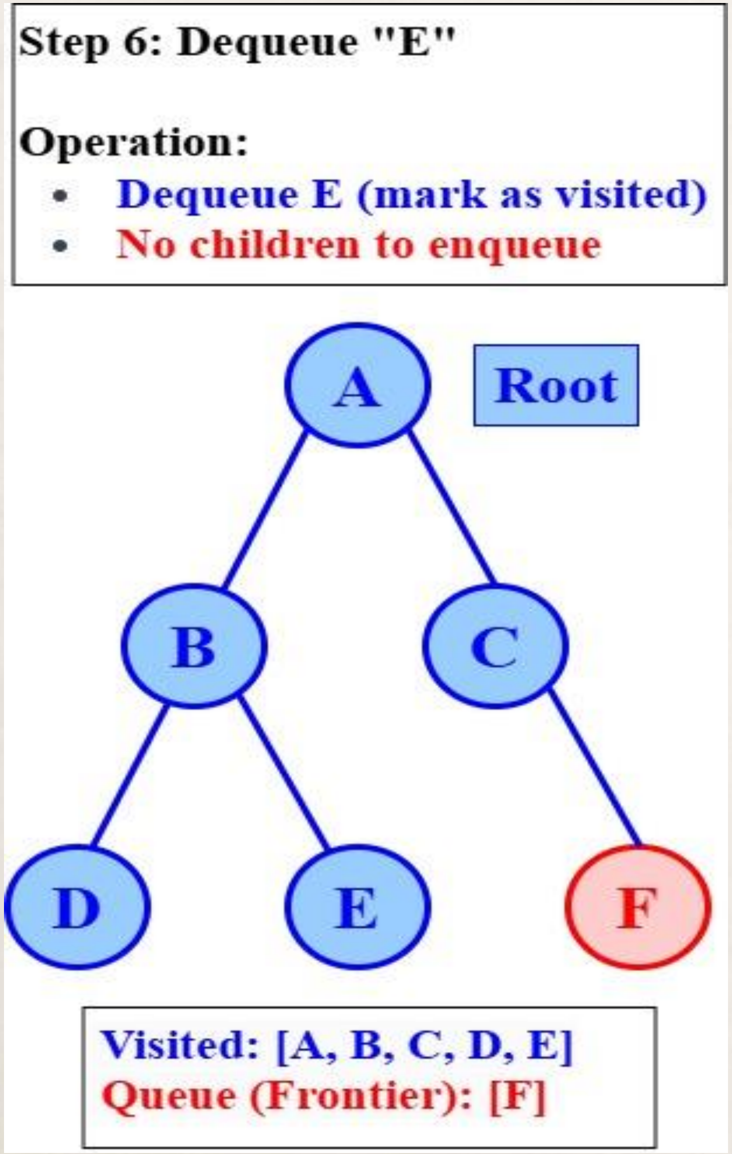
Breadth-first search traversal



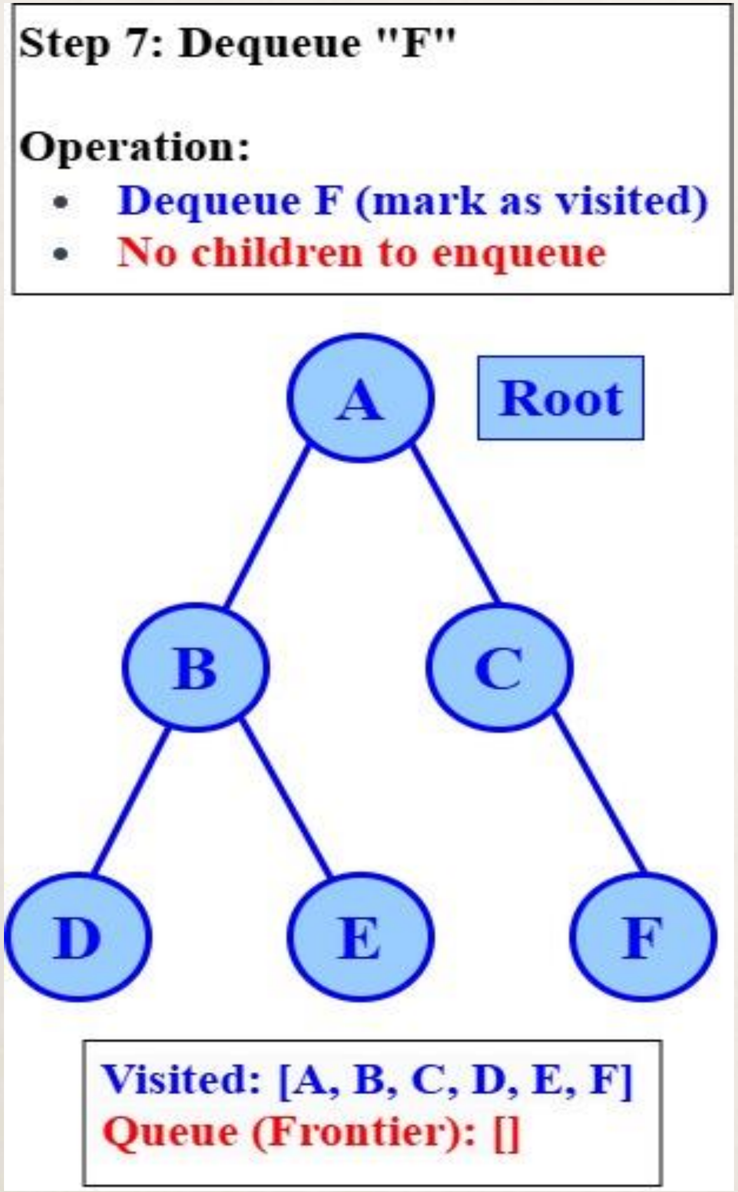
Breadth-first search traversal



Breadth-first search traversal



Breadth-first search traversal



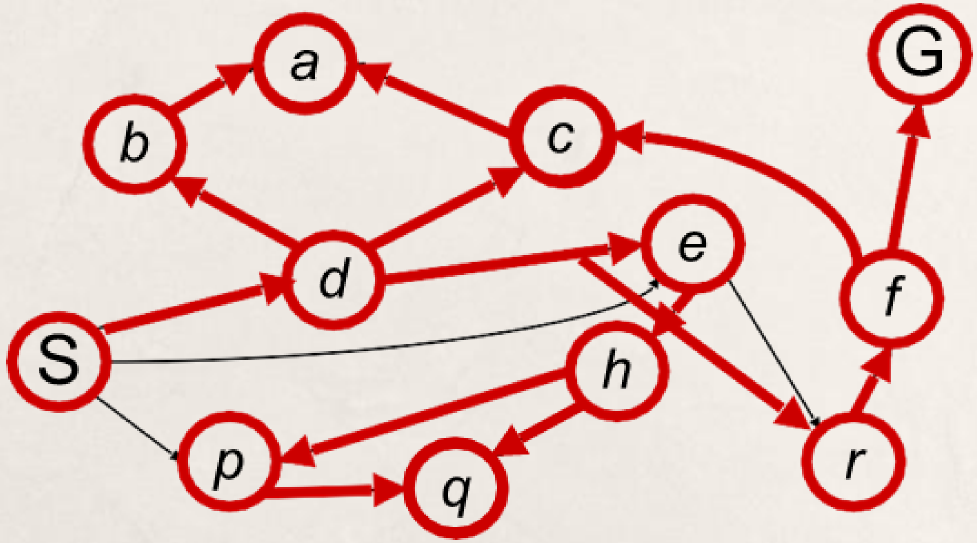
The depth-first search (DFS) algorithm

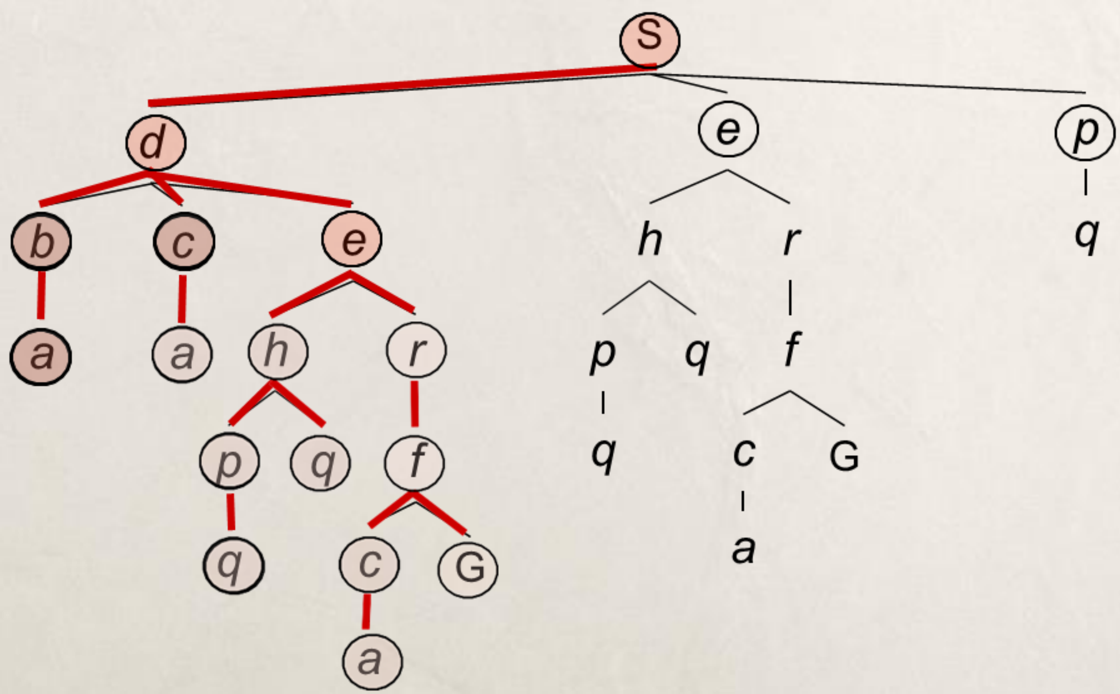
- Expands the deepest node on the frontier
- Search goes immediately to the deepest level of the search tree to a leaf node
  - Leaf node has no children
- If a leaf is reached, it is discarded and the search 'backs up' to previous depth
- Uses a stack (LIFO) for frontier

Depth-First Search

Strategy: expand a deepest node first

Implementation: Frontier is a LIFO stack





[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Generic best-first search

```
Function Tree-Search(problem, strategy)
  frontier = {Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = strategy.select_node(frontier)
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Breadth-first search (BFS)

```
Function Tree-Search(problem)
  frontier = Queue{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop()
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Depth-first search (BFS)

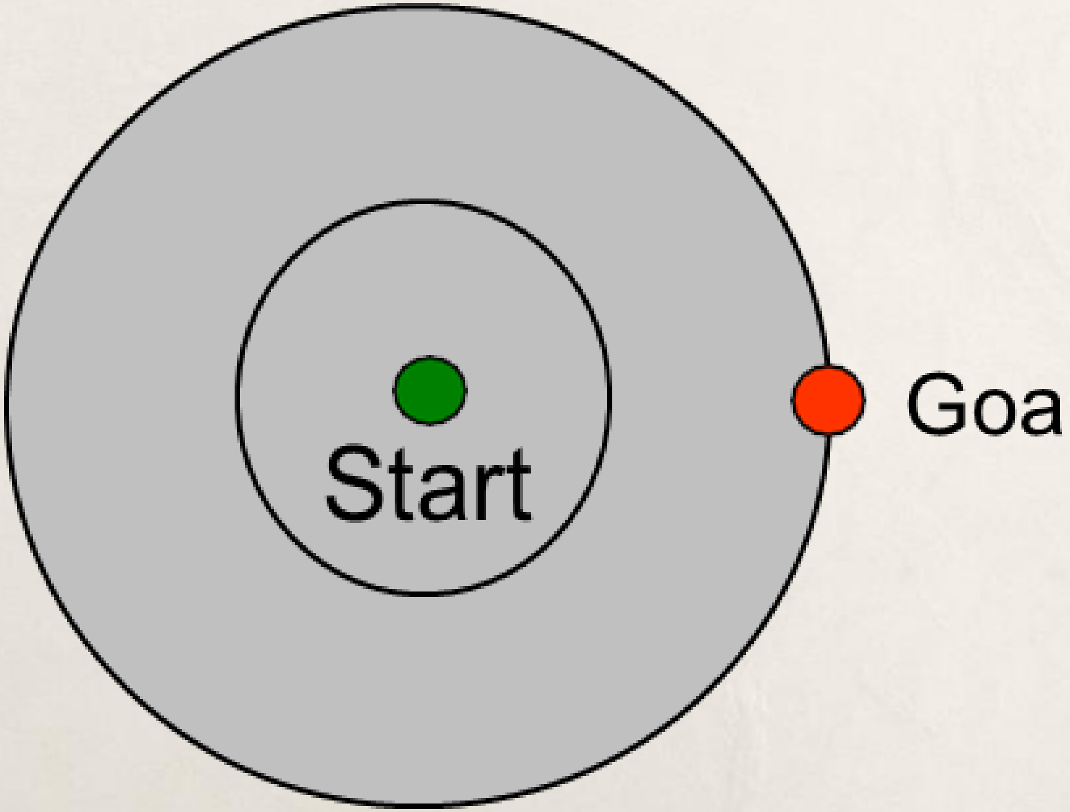
```
Function Tree-Search(problem)
  frontier = Stack{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop()
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

The uniform-cost search (UCS) algorithm

- BFS is optimal when all action costs are equal because it expands the shallowest node
- Uniform-cost search is optimal for any cost
- UCS expands node with lowest path cost
- If all costs are equal, equivalent to BFS
- UCS is Dijkstra's algorithm for a single goal

The uniform-cost search (UCS) algorithm

- Explores options in every "direction"

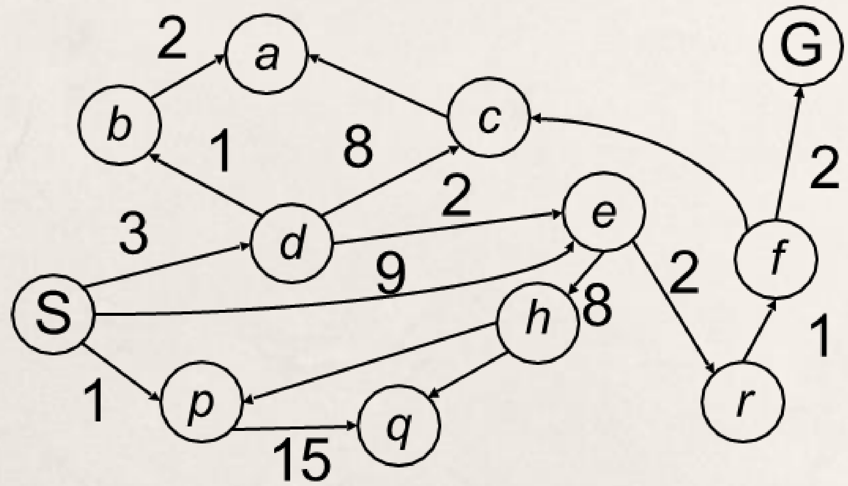


[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

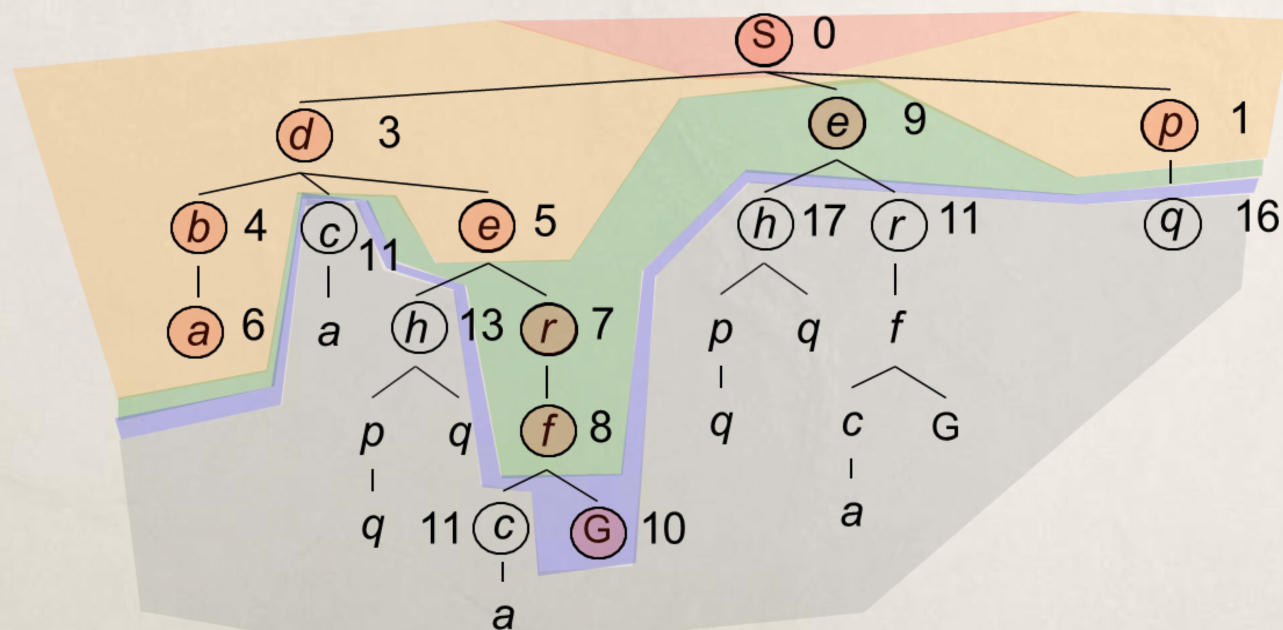
Uniform-cost search

Strategy: expand a cheapest node first

Implementation: Frontier is a priority queue Priority (cumulative cost)



Cost contours



[This slide was created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Generic best-first search

```
Function Tree-Search(problem, strategy)
  frontier = {Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = strategy.select_node(frontier)
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Breadth-first search (BFS)

```
Function Tree-Search(problem)
  frontier = Queue{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop()
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Depth-first search (BFS)

```
Function Tree-Search(problem)
  frontier = Stack{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop()
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Uniform-cost search (UCS)

```
Function Tree-Search(problem)
  frontier = {Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = min_cost_cum_path_value(frontier)
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

The magic of the queue

- A queue is used to store the nodes from the search tree that are in the frontier (i.e., that haven been generated but not yet expanded)
- It turns out that different notions of which node is "best" can be implemented simply by using different types of queues
- The rest of the generic search algorithm can remain the same to get a variety of well-known (informed and uniformed) search algorithms
- For example, by using a first-in first-out (**FIFO**) queue, we get breadth-first search, where the shallowest nodes are expanded first
- Using a last-in first-out (**LIFO**) queue (aka **stack**), we get **depth-first search**, another uninformed search that expands the deepest nodes first
- In Python, we can use a **dequeue** (implemented as a doubly-linked list under the hood) to simulate FIFO and LIFO queues
    - FIFO: pop off last element in the dequeue for expansion
    - LIFO: pop off first element in the dequeue for expansion

Priority queues

- To get an even wider variety of search algorithms, we can use a **priority queue** so that we can go beyond LIFO and FIFO
    - This is implemented as a **heap** data structure under the hood
- The priority queue enables us to pop-off, at any time, the node with the highest priority in the queue
- Priority is defined by providing an evaluation function **f**
- In our case, the function **f** takes a node in the frontier as its argument and returns a numeric value that indicates its priority
- In the following, we assume that lower values returned by this function indicate greater priority
- For example, to get breadth-first search, we can use the depth of a node as the numeric value returned by the function

Avoiding repeated states

- One of the most important search ideas
- Especially important with reversible actions
- Most infinite loops/wasted time can be avoided by not returning to identical states
- We can remember nodes expanded
  - Don't re-expand them
- Can lead to exponential savings in the number of nodes generated

"reached" list

- Store states we have already expanded
- `"reached"` list stores expanded states
  - Often implemented as a hash table/dictionary for efficient lookup
- Check if a node is reached before expanding
- `"frontier"` stores unexpanded nodes
  - Often implemented as a priority queue
  - With $O(\log(n))$ insertion and extraction time instead of the faster FIFO queue, which is $O(1)$

Generic best-first search

```
Function Tree-Search(problem, strategy)
  frontier = {Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = strategy.select_node(frontier)
    if (node.state is goal) return solution
    else frontier.add(Expand(node, problem))
```

Generic best-first search updated

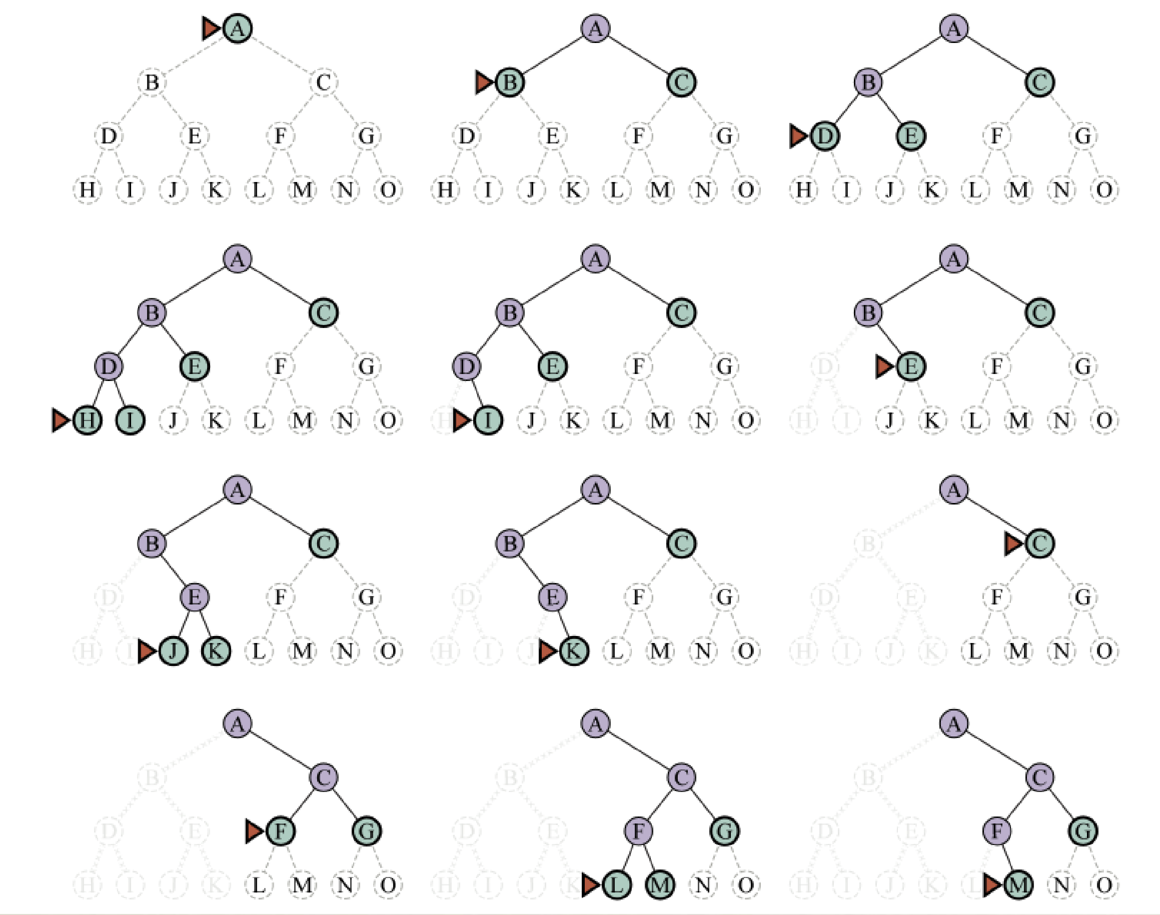
```
Function Tree-Search-Updated(problem, strategy)
  reached = {} // add reached list
  frontier = PQ{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop() // based on a priority/criteria
    if (node.state is goal) return solution
    if (node.state in reached) continue // check if reached
    reached.add(node.state) // mark node reached
    frontier.add(Expand(node, problem))
```

Generic best-first search updated

```
Function Tree-Search-Updated(problem, strategy)
  reached = {}
  frontier = PQ{Node(problem.initial_state)}
  while (true)
    if (frontier.empty) return fail
    node = frontier.pop()
    if (node.state is goal) return solution
    if (node.state in reached)
    reached.add(node.state)
    frontier.add(Expand(node, problem))
```

Depth-first search

- An implementation of depth-first search can be obtained using the negative of a node's depth as the value returned by **f**



Uniform-cost search (Dijkstra's algorithm)

- When actions have different costs, it does not make sense to make decisions based solely on the depth of nodes in the search tree
- We can use a priority queue where nodes with smaller path costs are preferred, i.e., nodes that can be reached with lower cost
- In this case, **f** returns the path cost of the node it is applied to

References

- Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu)
- Dr. David Churchill, Department of Computer Science Memorial University of Newfoundland (http://www.cs.mun.ca/~dchurchill)