3010

ARTIFICIAL INTELLIGENCE

Lecture 1 State space search

Masashi Shimbo

2019-05-09

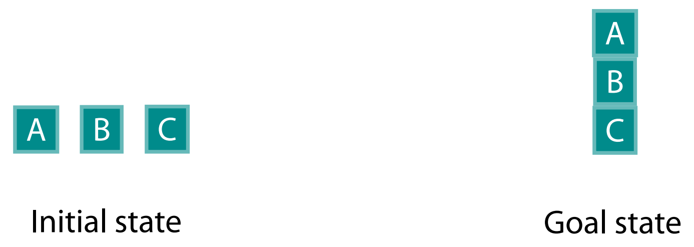
TODAY'S AGENDA

- What is **state space search** in AI?
- State space search algorithm **template**
- **Two** basic search algorithms
  - Breadth-first search
  - Depth-first search

AI SEARCH: SCENARIO

- An AI program (**agent**) wants the environment to be in a particular state (**goal state**)
- The agent usually has many **actions** to choose from
—Taking an action changes the state of the environment
- Thus, the task of an agent is to find an action sequence that changes the current state (**initial state**) of the environment into the goal state

EXAMPLE: "BLOCKS WORLD"

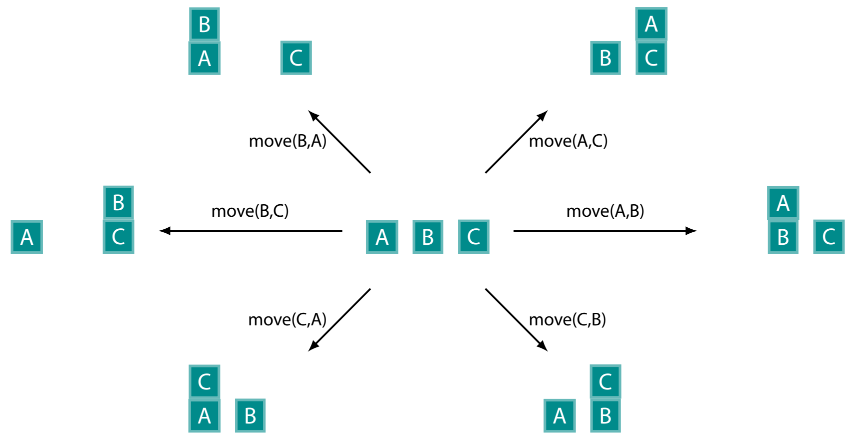


- Current state of the environment is called **initial state**
- Desired state is called **goal state**

AI SEARCH: SCENARIO

- An AI program (**agent**) wants the environment to be in a particular state (**goal state**)
- The agent usually has many **actions** to choose from
  —Taking an action changes the state of the environment
- Thus, the task of an agent is to find an action sequence that changes the current state (**initial state**) of the environment into the goal state

ACTIONS CHANGE THE STATE OF THE ENVIRONMENT



AI SEARCH: SCENARIO

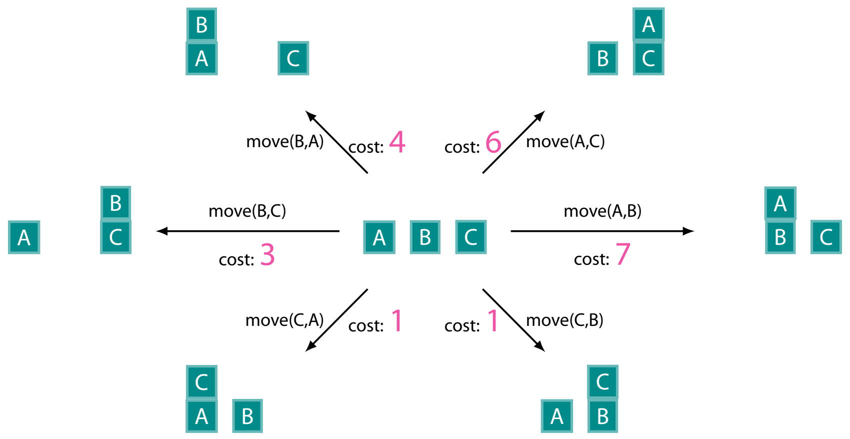
- An AI program (**agent**) wants the environment to be in a particular state (**goal state**)
- The agent usually has many **actions** to choose from
  —Taking an action changes the state of the environment
- Thus, the task of the agent is to find an action sequence that changes the current state (**initial state**) of the environment into the goal state

AI SEARCH: TASK

Find an action sequence (**solution**) that can put the environment in a desired **goal state**

—preferably without incurring too much total cost

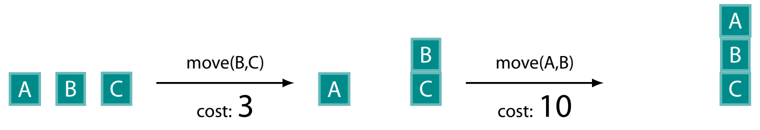
(each action incurs a certain amount of **cost**)



Note: All action costs must be **positive**

SOLUTION

= Action sequence changing the initial state to a goal state



Note: cost of an action sequence is the **sum** of the costs of actions involved

➥ solution above has cost $3 + 10 = 13$

ACTION COSTS—WHAT DO THEY REPRESENT?

Depend on the task you want to solve

But “costs” always represent the quantity you want to **minimize**

➥ the smaller the better

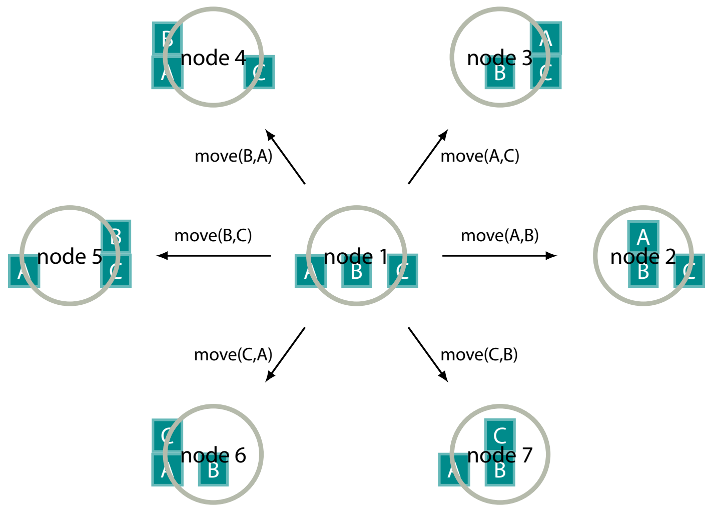
Example:
payment, labor, fuel consumption, time, or combination of these

STATE SPACE IS ESSENTIALLY A GRAPH

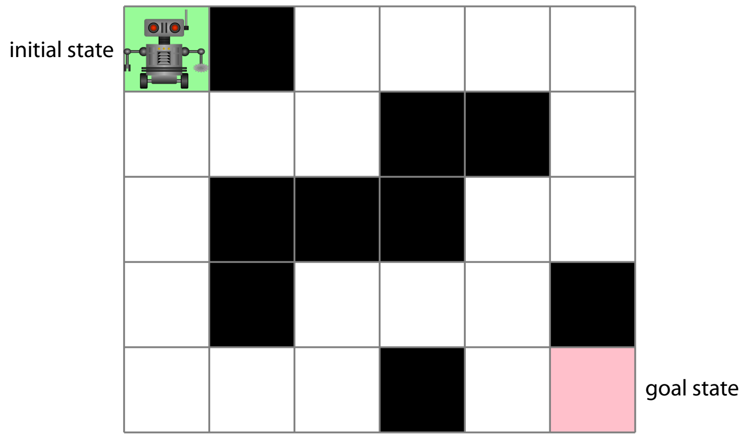
State space (states, actions, costs) can be represented as a graph

state space concept		graph concept
state	$=$	node/vertex
action	$=$	arc/edge
action cost	$=$	arc (edge) weight
action sequence	$=$	path

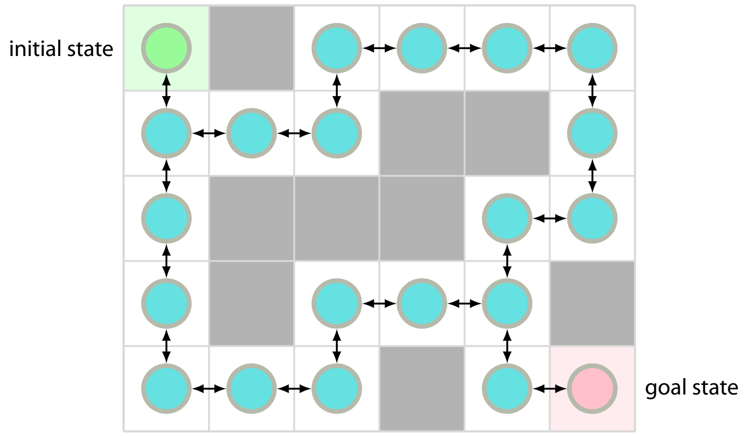
These terms will be used interchangeably in this course



EXAMPLE 2: GRIDWORLD



EXAMPLE 2: GRIDWORLD



STATE SPACE: MORE FORMAL DEFINITION

State space is a labeled graph (V, A, c), where

- V = set of nodes/vertices (states)
- A = set of edges/arcs (actions), $A \subset V \times V$
  - ➥ $(u, v) \in A$ means there is an edge from node u to node v
- c = cost (label) function, $c : A \rightarrow \mathbb{R}_+$
  - ➥ function c associates each edge with a positive cost; e.g., $c(u, v) = 5$

with

- a specific node $s \in V$ called **initial state**
- a set $G \subset V$ of nodes called **goal** (or **terminal**) **states**

(State space) search is a task of finding an action sequence (or path) in state space, from the initial state to a goal state

— preferably the one with the least cost
(= shortest path in the state space graph)

So we are essentially dealing with the **shortest path problem**

FURTHER ASSUMPTIONS

- No outgoing edges exist in goal states
- The number of states (nodes) can be infinite
  —However, for any state, the number of outgoing edges (available actions) is finite (i.e., the graph is **locally finite**)
- Action costs are bounded away from 0. That is,

  $$\exists \varepsilon > 0 \quad \forall (u, v) \in A \quad c(u, v) > \varepsilon$$

  (all action costs are greater than a certain positive number ε)

INFORMATION AVAILABLE TO AGENT

Following functions can be used for designing AI search algorithms:

function $\mathrm{Succ}(v)$**:** set of "successor" nodes of node v

$$\mathrm{Succ}(v) = \{u \mid (v, u) \in A\}$$

i.e., set of nodes that can be reached from v with a single action

N.B. Because the graph is locally finite, $\mathrm{Succ}(v)$ is finite for every v

function $c(v, u)$**:** cost of taking an action at node v that leads to node u

function $\mathrm{IsGoal}(v)$**:** returns "true" if node v is a goal; "false" if not

STATE SPACE IN AI PROBLEMS CAN BE HUGE

- The number of states might even be infinite
- in which case explicit graph representation **does not fit on memory**

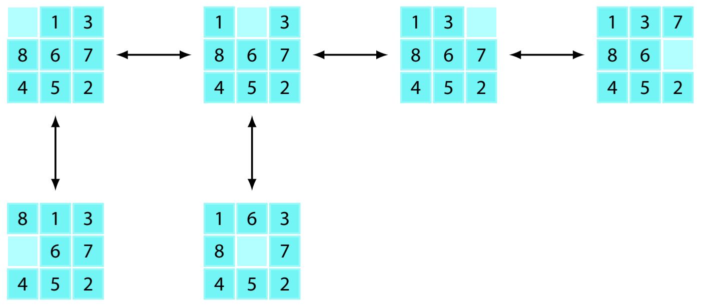
For example …

EIGHT-PUZZLE

	1	3
8	6	7
4	5	2

1	2	3
4	5	6
7	8	

EIGHT-PUZZLE



EIGHT-PUZZLE

	1	3
8	6	7
4	5	2

$9!/2 \simeq 1.8 \times 10^5$ states

1	2	3
4	5	6
7	8	

FIFTEEN-PUZZLE

14	13	15	7
11	12	9	5
6		2	1
4	8	3	10

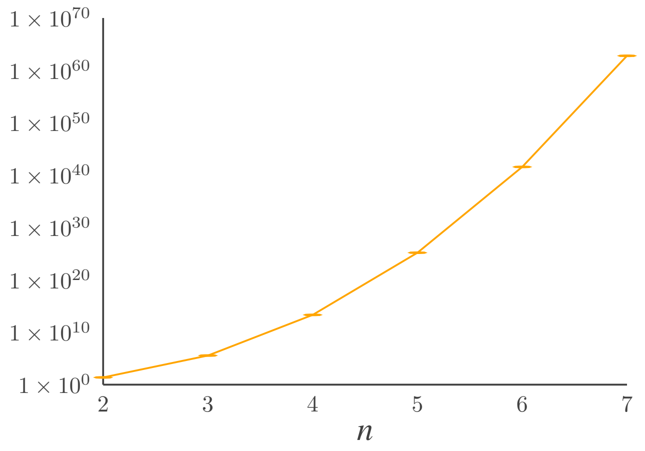
$16!/2 \simeq 1.0 \times 10^{13}$ states

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

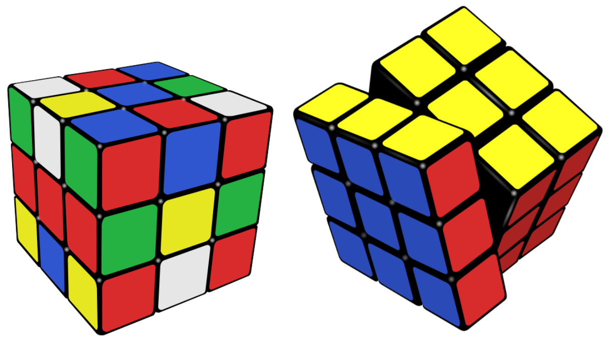
3×3	8-Puzzle	$9!/2 \simeq 1.8 \times 10^5$
4×4	15-Puzzle	$16!/2 \simeq 1.0 \times 10^{13}$
5×5	24-Puzzle	$25!/2 \simeq 7.8 \times 10^{24}$
6×6	35-Puzzle	$36!/2 \simeq 1.9 \times 10^{41}$
7×7	48-Puzzle	$49!/2 \simeq 3.0 \times 10^{62}$

Cf. $1G = 10^9$
$1T = 10^{12}$
$1P = 10^{15}$
$1E = 10^{18}$

NUMBER OF STATES IN $(n^2 - 1)$-PUZZLES



RUBIK'S CUBE



4.3×10^{19} configurations (states)

R. Korf [1997] used Iterative-Deepending A* to find optimal solutions to Rubik's cube

STATE SPACE DOES NOT FIT ON MEMORY—WHAT CAN BE DONE?

Build a **partial** graph representing part of the state space **on the fly**

➥ **Expand** one node at a time, until a goal state is reached

Terminology

"expand" a node
a node is said to be **expanded** if all of its successor nodes are **generated** (see below)

"generate" a node
a node is said to be *generated* if its representation is created and stored on memory

GRADUALLY BUILD A STATE SPACE GRAPH

Expand one node at a time, until a goal state is reached

- Initially, only initial node s is on memory (i.e., generated)
- All search algorithms start by expanding s (i.e., generating all succssors of s)
- Then **choose** one of these nodes to expand next, and repeat

DIFFERENT NODE EXPANSION STRATEGIES
➥ DIFFERENT SEARCH ALGORITHMS

The order in which nodes are expanded determines different search strategies

➥ Many search algorithms differ only on node expansion strategies — otherwise they are quite similar

TODAY'S AGENDA

- What is **state space search** in AI?
- State space search algorithm **template**
- **Two** basic search algorithms
  - Breadth-first search
  - Depth-first search

- Breadth-first search
- Depth-first search
- Dijkstra's algorithm
- A*

Difference in these algorithms lies in the order in which nodes are expanded

Let us assume that the state space is a **uniform-cost tree** rooted at the initial node, and analyze how they differ

TREE-STRUCTURED STATE SPACE

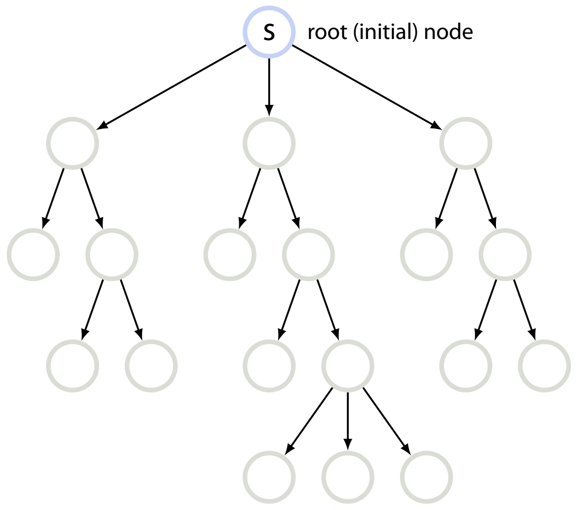
For the ease of discussion, we assume:

- all action costs are identical
- the state space is a **tree** rooted at the initial state

A **(rooted) tree** is a graph such that:

- every node in the graph has exactly one path from the root (initial) node.

ROOTED TREE: EXAMPLE



- Breadth-first search
- Depth-first search
- Dijkstra's algorithm
- A*

If the state space is a tree, these algorithms are instances of the **General Tree Search** algorithm…

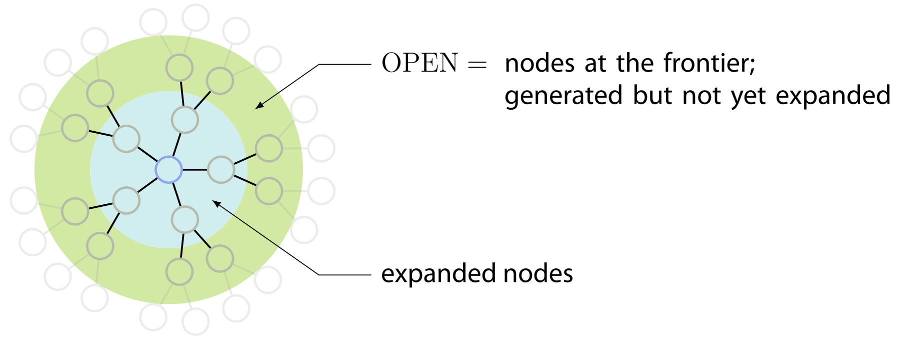
GENERAL TREE SEARCH ALGORITHM (TEMPLATE)

input : initial state s

output : a solution path, if found, or "failure"

```
OPEN ← new List                          # create a list to hold nodes that are generated but not yet expanded
Insert(OPEN, s)                          # initially OPEN only holds the initial state
loop do                                  # repeat the following forever
    if IsEmpty(OPEN) then                # empty OPEN means no goal state is reachable from s
        return "failure"
    v ← RemoveOne(OPEN)                  # pick a node in OPEN
    if IsGoal(v) then return Solution(v, s)   # if it is a goal, return solution path
    Expand(v, OPEN)                      # expand v, i.e., generate all its successors and put it in OPEN
```

OPEN AND EXPANDED NODES DURING SEARCH


OPEN = nodes at the frontier; generated but not yet expanded
expanded nodes

- At the outset, list OPEN contains the initial state s only
- In each iteration:
  - A state is picked up (and removed) from OPEN (RemoveOne)
  - It is Expanded — all successor states are generated (= kept on memory)
  - The generated states are added to OPEN (with Insert)

Different implementations of

- data structure List
- function RemoveOne
- function Insert

lead to different search strategies

GENERAL TREE SEARCH ALGORITHM (TEMPLATE)

```
input   : initial state $s$
output  : a solution path, if found, or "failure"

OPEN ← new List                  # create a list to hold nodes that are generated but not yet expanded
Insert(OPEN, $s$)                # initially OPEN only holds the initial state
loop do                          # repeat the following forever
    if IsEmpty(OPEN) then        # empty OPEN means no goal state is reachable from $s$
        return "failure"
    $v$ ← RemoveOne(OPEN)        # pick a node in OPEN
    if IsGoal($v$) then return Solution($v, s$)   # if it is a goal, return solution path
    Expand($v$, OPEN)            # expand $v$, i.e., generate all its successors and put it in OPEN
```

procedure Expand(v, OPEN)

```
input   : state $v$ to expand
input   : OPEN—list to store successors of $v$

foreach $u \in$ Succ($v$) do
    Reserve memory to store Parent of node $u$    # "generate" $u$
    Parent[$u$] ← $v$                              # remember that $v$ is the parent of $u$
    Insert(OPEN, $u$)                              # put $u$ in OPEN, because $u$ is generated but not yet expanded
```

Recall that Succ(v) is the function that returns the set of successor nodes of v:

$$\mathrm{Succ}(v) = \{u \mid (v,u) \in A\}$$

GENERAL TREE SEARCH ALGORITHM (TEMPLATE)

input : initial state s
output : a solution path, if found, or "failure"

```
OPEN ← new List                      # create a list to hold nodes that are generated but not yet expanded
Insert(OPEN, $s$)                    # initially OPEN only holds the initial state
loop do                              # repeat the following forever
    if IsEmpty(OPEN) then            # empty OPEN means no goal state is reachable from $s$
        return "failure"
    $v$ ← RemoveOne(OPEN)            # pick a node in OPEN
    if IsGoal($v$) then return Solution($v$, $s$)   # if it is a goal, return solution path
    Expand($v$, OPEN)                # expand $v$, i.e., generate all its successors and put it in OPEN
```

function Solution(t, s)

Backtrack Parent[v] to obtain a path from s to t from the initial state s to a goal state t

input : goal state t
input : initial state s
output : list P of actions saved in a "stack"

```
$P$ ← new Stack
$v$ ← $t$
while $v \neq s$ do
    Push($P$, $v$)
    $v$ ← Parent[$v$]
return $P$
```

WHEN WE DISCUSS SEARCH ALGORITHMS, THE FOLLOWING PROPERTIES ARE OF INTEREST:

A search procedure is said to be

Complete
if it never fails to find a solution (provided that one exists)

Admissible
if it always finds a cheapest solution

OTHER RELEVANT EVALUATION METRICS

Time complexity
The amount of time a search procedure takes to find a solution

Space complexity
The amount of memory it needs to find a solution

TODAY'S AGENDA

- What is **state space search** in AI?
- State space search algorithm **template**
- **Two** basic search algorithms
  - Breadth-first search
  - Depth-first search

BREADTH-FIRST SEARCH

Our first tree search algorithm

Use a **(FIFO; "first-in first-out") queue** to order node expansion in the general tree search algorithm

FIFO QUEUE

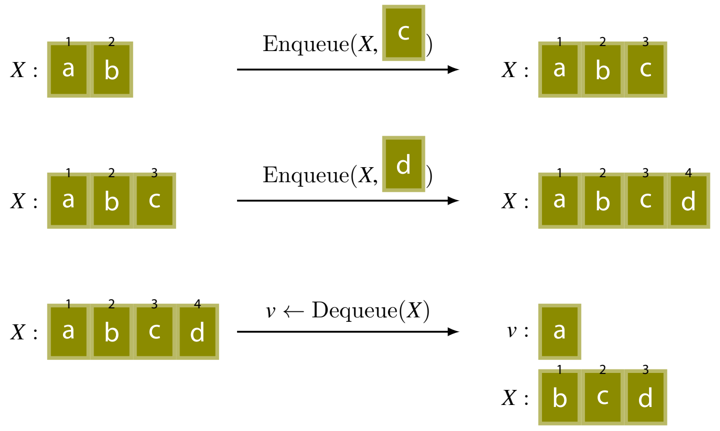
Procedures/functions to manipulate FIFO queue X:

function $\mathrm{IsEmpty}(X)$
  Return true if list (=queue) X is empty

procedure $\mathrm{Enqueue}(X, v)$
  Put item v at the **end** of list X.

function $\mathrm{Dequeue}(X)$
  Return the **first** item in X, after removing it.

FIFO QUEUE: EXAMPLES



GENERAL TREE SEARCH ALGORITHM (TEMPLATE)

input : initial state s
output : a solution path, if found, or “failure”

```
OPEN ← new List                        # create a list to hold nodes that are generated but not yet expanded
Insert(OPEN, s)                        # initially OPEN only holds the initial state
loop do                                # repeat the following forever
    if IsEmpty(OPEN) then              # empty OPEN means no goal state is reachable from s
        return “failure”
    v ← RemoveOne(OPEN)                # pick a node in OPEN
    if IsGoal(v) then return Solution(v, s)   # if it is a goal, return solution path
    Expand(v, OPEN)                    # expand v, i.e., generate all its successors and put it in OPEN
```

BREADTH-FIRST SEARCH

input : initial state s
output : a solution path, if found, or “failure”

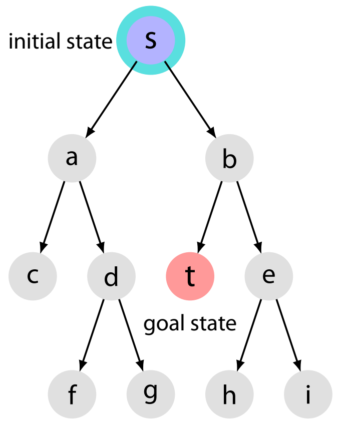
```
OPEN ← new Queue                       # create a list to hold nodes that are generated but not yet expanded
Enqueue(OPEN, s)                       # initially OPEN only holds the initial state
loop do                                # repeat the following forever
    if IsEmpty(OPEN) then              # empty OPEN means no goal state is reachable from s
        return “failure”
    v ← Dequeue(OPEN)                  # pick a node in OPEN
    if IsGoal(v) then return Solution(v, s)   # if it is a goal, return solution path
    Expand(v, OPEN)                    # expand v, i.e., generate all its successors and put it in OPEN
```

procedure Expand(v, OPEN)

input : state v to expand
input : OPEN—list to store successors of v

```
foreach u ∈ Succ(v) do
    Reserve memory to store Parent of node u    # “generate” u
    Parent[u] ← v                               # remember that v is the parent of u
    Enqueue(OPEN, u)                            # put u in OPEN, because u is generated but not yet expanded
```

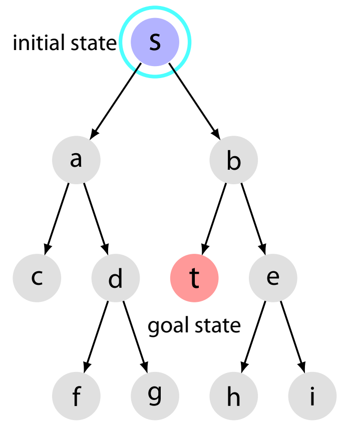
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: s

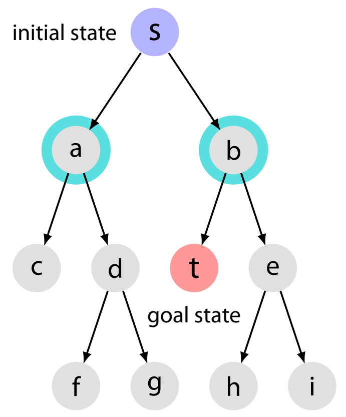
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: s

OPEN:

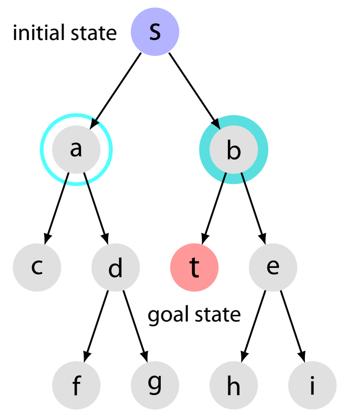
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: a b

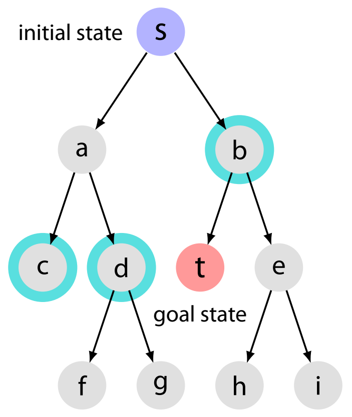
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: a

OPEN: b

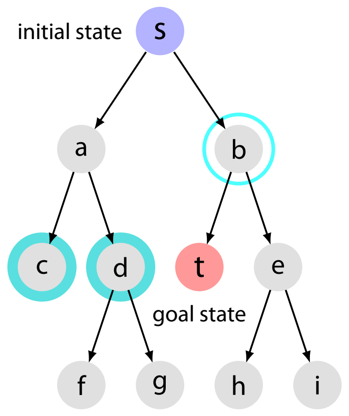
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: b c d

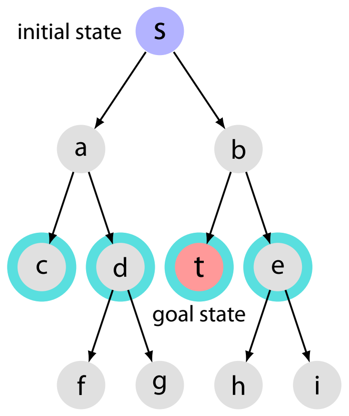
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: b

OPEN: c d

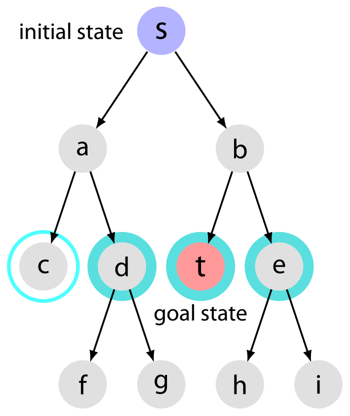
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: c d t e

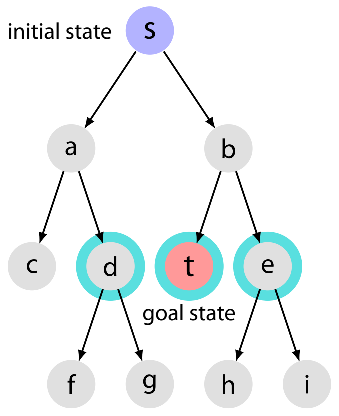
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: c

OPEN: d t e

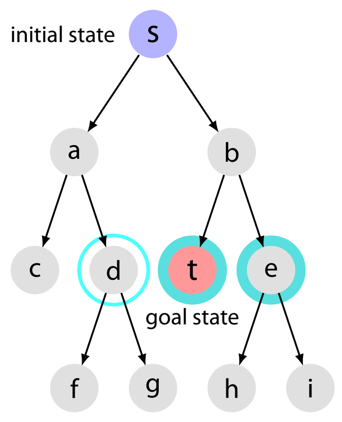
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: d t e

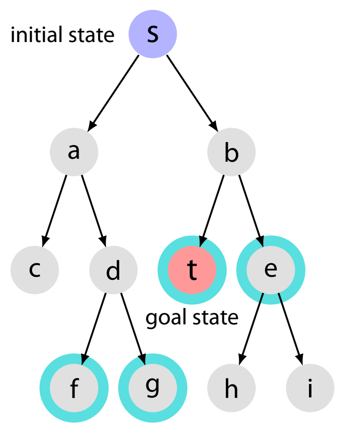
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: d

OPEN: t e

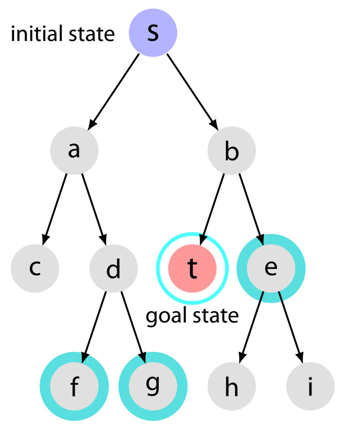
BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: t e f g

BREADTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: t

OPEN: e f g

BREADTH-FIRST SEARCH: PROPERTIES

BFS is complete

BFS never fails to find a goal, even if the state space is infinite.

BFS is *not* admissible

BFS returns the first shallowest solution path it finds. In general, the shallowest solution path might not be the one with the least cost.
It is, however, admissible if all actions have an equal cost.

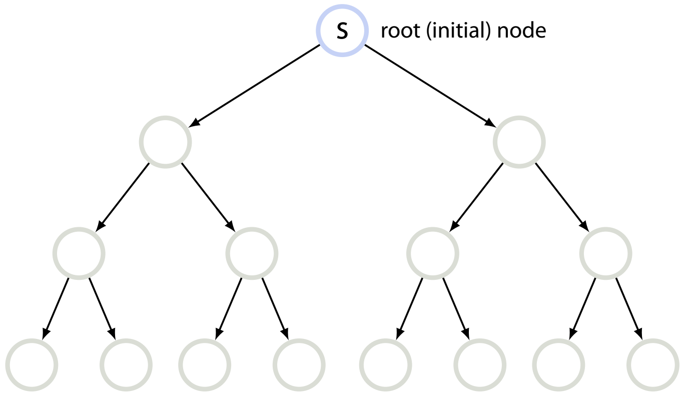
BREADTH-FIRST SEARCH: COMPLEXITY

For discussion, assume a tree with uniform branching factor b

Branching factor (of a node)
Number of successors at a given node

Tree with uniform branching factor b
tree in which all internal nodes have exactly b successors

TREE WITH UNIFORM BRANCHING FACTOR $b = 2$



WORST CASE SPACE COMPLEXITY

$d =$ depth of a shallowest goal state

Space complexity is dependent on the number of generated nodes

- because once a node is generated, it is kept on memory
- Recall that generated nodes = nodes placed in OPEN

Space complexity
In the worst case, all nodes up to depth d, and some nodes at depth $(d+1)$ are generated. Thus,

$$1 + b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

WORST CASE TIME COMPLEXITY

$d =$ depth of a shallowest goal state

Time complexity
Same as space complexity
—because node generation dominates the runtime

$$O(b^{d+1})$$

PROBLEM WITH BREADTH-FIRST SEARCH

Memory inefficient

DEPTH-FIRST SEARCH

Uses Stack (LIFO list; “last-in first-out” list) for node expansion ordering

STACK

LIFO (“Last-in first-out”) buffer

Procedures/functions to manipulate stack S:

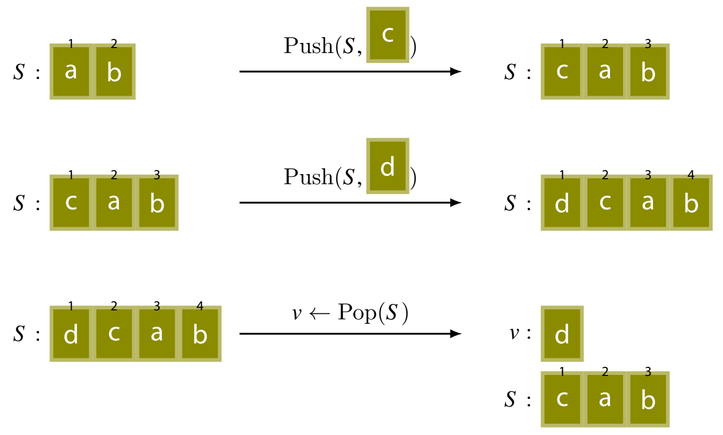
function $\mathrm{IsEmpty}(S)$
Return true if stack S is empty

procedure $\mathrm{Push}(S, v)$
Insert item v at the **beginning** of list S

function $\mathrm{Pop}(S)$
Return the **first** item in S after removing it

function $\mathrm{Inspect}(S)$
Return the **first** item in S, without removing the item
(Equivalent to $v \leftarrow \mathrm{Pop}(S)$ followed by $\mathrm{Push}(S, v)$)

STACK: EXAMPLES



GENERAL TREE SEARCH ALGORITHM (TEMPLATE)

input : initial state s
output : a solution path, if found, or “failure”

```
OPEN ← new List                          # create a list to hold nodes that are generated but not yet expanded
Insert(OPEN, $s$)                        # initially OPEN only holds the initial state
loop do                                  # repeat the following forever
    if IsEmpty(OPEN) then                # empty OPEN means no goal state is reachable from $s$
        return “failure”
    $v$ ← RemoveOne(OPEN)                # pick a node in OPEN
    if IsGoal($v$) then return Solution($v$, $s$)   # if it is a goal, return solution path
    Expand($v$, OPEN)                    # expand $v$, i.e., generate all its successors and put it in OPEN
```

DEPTH-FIRST SEARCH

Input : initial state s
Output : a solution path, if found, or “failure”

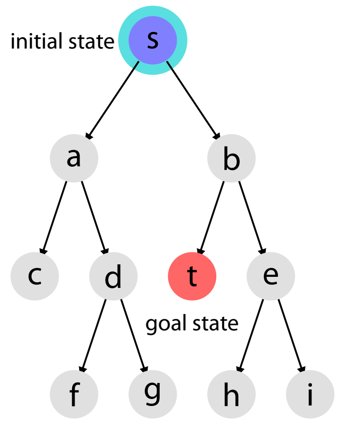
```
OPEN ← new Stack                         # create a list to hold nodes that are generated but not yet expanded
Push(OPEN, $s$)                          # initially OPEN only holds the initial state
loop do                                  # repeat the following forever
    if IsEmpty(OPEN) then                # empty OPEN means no goal state is reachable from $s$
        return “failure”
    $v$ ← Pop(OPEN)                      # pick a node in OPEN
    if IsGoal($v$) then return Solution($v$)   # if it is a goal, return solution path
    Expand($v$, OPEN)                    # expand $v$, i.e., generate all its successors and put it in OPEN
```

procedure Expand(v, OPEN)

input : state v to expand
input : OPEN—list to store successors of v

```
foreach $u \in$ Succ($v$) do
    Reserve memory to store Parent of node $u$    # “generate” $u$
    Parent[$u$] ← $v$                             # remember that $v$ is the parent of $u$
    Push(OPEN, $u$)                               # put $u$ in OPEN, because $u$ is generated but not yet expanded
```

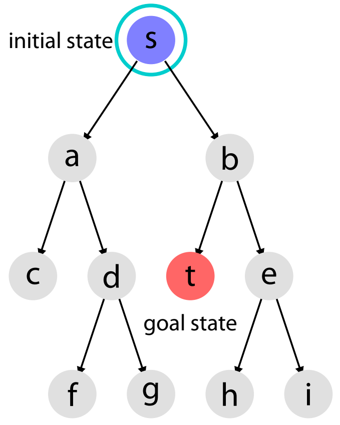
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: s

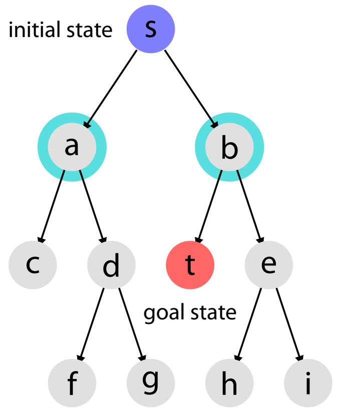
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: s

OPEN:

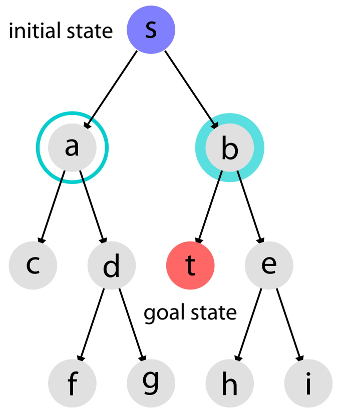
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: a b

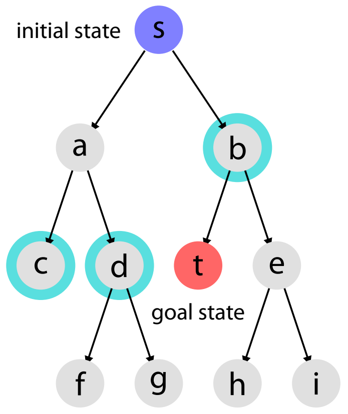
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: a

OPEN: b

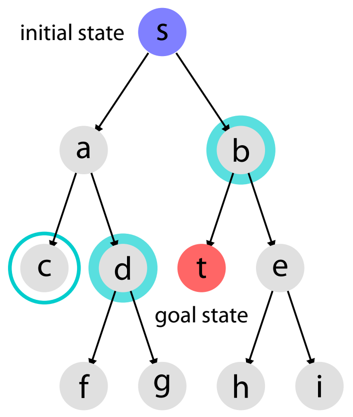
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: c d b

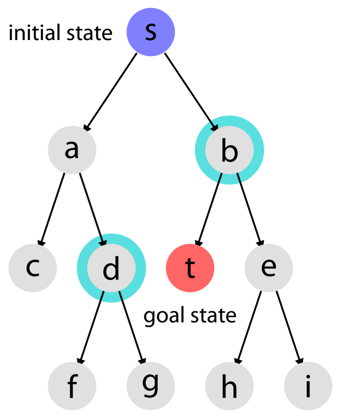
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: c

OPEN: d b

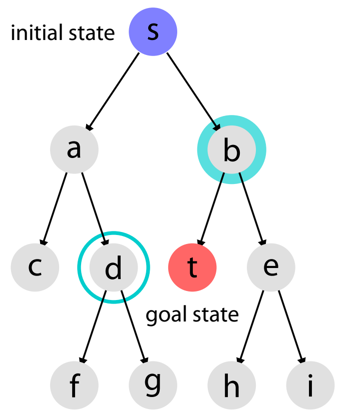
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: d b

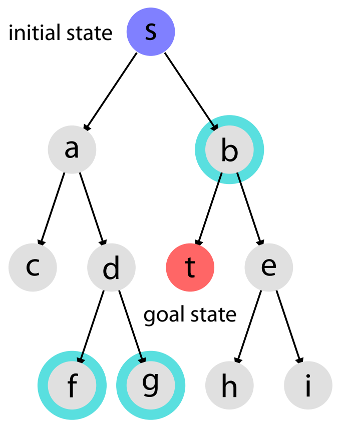
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: d

OPEN: b

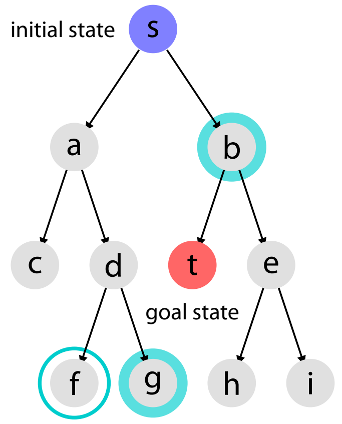
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: f g b

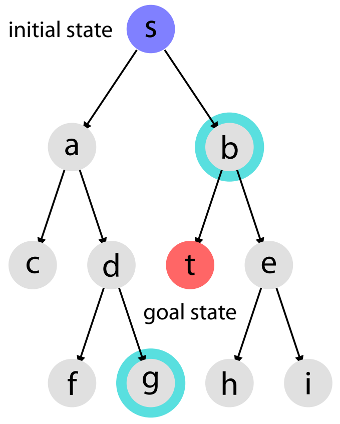
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: f

OPEN: g b

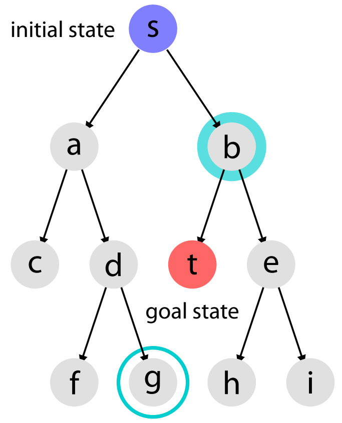
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: g b

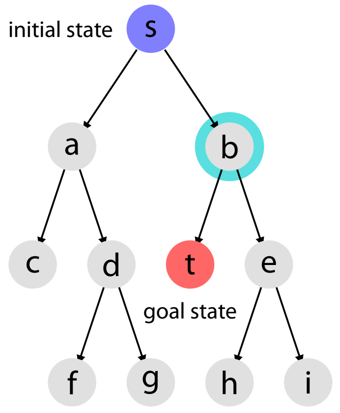
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: g

OPEN: b

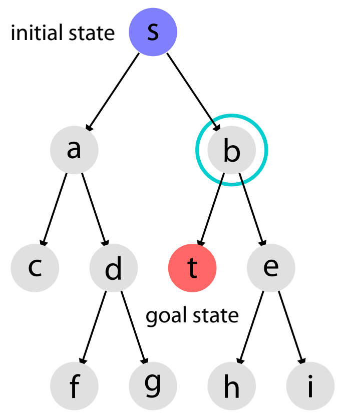
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: b

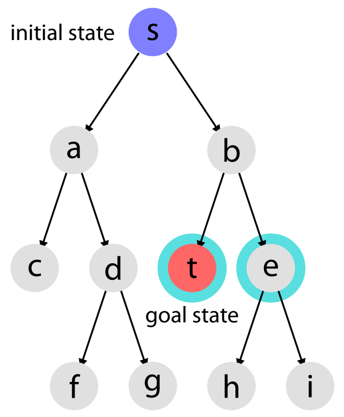
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: b

OPEN:

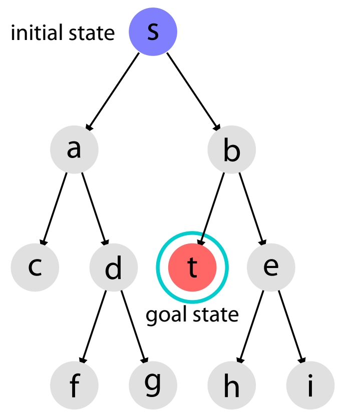
DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state:

OPEN: t e

DEPTH-FIRST SEARCH: A SAMPLE RUN



Expanded state: t

OPEN: e

DEPTH-FIRST SEARCH IS NOT COMPLETE, NOR IS ADMISSIBLE

May not terminate if the state space is infinite

It returns the first (leftmost) solution however bad its quality is.

DEPTH-FIRST SEARCH

Additional improvement

With small modifications, depth-first search can be memory-efficient

➥ suitable for search in huge state space

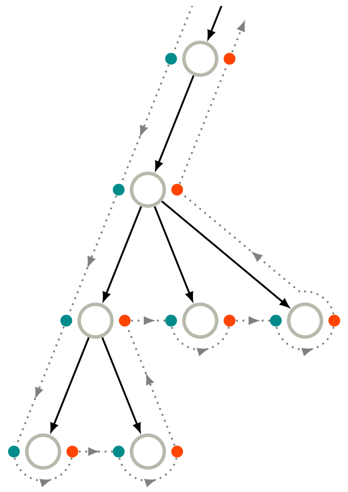
Idea:
As soon as all descendants of a node have been expanded, the node can be removed from memory

IDEA: VISIT EACH NODE TWICE

- once when it is expanded
  — when there is still a chance that a goal exists below the node
- once when all its descendant has been expanded and we are backtracking
  — when all the subtree below the node has been exhaustively searched (but goal was not found)

We keep track of the number of visits to v in $\mathrm{Count}[v]$

- First, when v is generated, let $\mathrm{Count}[v] \leftarrow 0$
  ➥ v in OPEN has $\mathrm{Count}[v] = 0 \implies$ it's the first pass for v
- After it is expanded, let $\mathrm{Count}[v] \leftarrow 1$
  ➥ v in OPEN has $\mathrm{Count}[v] = 1 \implies$ it's the second pass for v



● **first pass (forward search)**

- keep the node in OPEN
- set Count to 1 (from 0)
- expand the node

● **second pass (backtracking)**

- Detected by checking whether $\mathrm{Count} = 1$
- second pass means no goal was found in the subtree underneath (so no use keeping this node)
- release memory for the node
- remove the node from OPEN

DEPTH-FIRST SEARCH—original version

Input : initial state s
Output : a solution path, if found, or "failure"

```
OPEN ← new Stack
Push(OPEN, $s$)
loop do
    if OPEN is empty then return "failure"
    $v$ ← Pop(OPEN)
    if IsGoal($v$) then return Solution($v$)
    Expand($v$, OPEN)
```

DEPTH-FIRST SEARCH—memory saving version

Input : initial state s
Output : a solution path, if found, or "failure"

```
Count[$s$] = 0
OPEN ← new Stack
Push(OPEN, $s$)
loop do
    if OPEN is empty then return "failure"
    $v$ ← Inspect(OPEN)                          # We don't remove $v$ from the stack yet
    if IsGoal($v$) then return Solution($v$)
    if Count[$v$] = 0 then # first time $v$ is visited
        Expand($v$, OPEN)
        Count[$v$] ← 1
    else # second time $v$ is visited
        Pop(OPEN)                                # remove $v$ from the stack
        Release memory associated with $v$       # i.e., Parent[$v$] and Count[$v$]
```

procedure Expand(v, OPEN)

input : state v to expand
input : OPEN—where to store successors of v

```
foreach $u \in$ Succ($v$) do
    Reserve memory to store Parent[$u$] and Count[$u$]
    Parent[$u$] ← $v$
    Count[$u$] ← 0
    Push(OPEN, $u$)
```

MEMORY USED BY DEPTH-FIRST SEARCH (memory saving version)

- As soon as all descendants of a node have been expanded, the node is removed from memory
  ➥ required memory is linear in the number of nodes in OPEN
- For a tree with uniform branching factor b, when depth-first search expands a node at depth k, at most $O(bk)$ nodes are stored in OPEN
  ➥ required memory is linear in the depth of the state space

WORST CASE COMPLEXITY OF DEPTH-FIRST SEARCH

Time complexity
In the worst case all nodes in the state space will be expanded. hence, $O(b^m)$ where m is the maximum depth of any node.

Space complexity
Using the memory-saving version,

$$O(bm)$$

SUMMARY

Breadth-first vs. depth-first tree search

Search algorithm	Breadth-first	Depth-first
Complete?	yes	no
Admissible?	yes*	no
Time complexity	$O(b^{d+1})$	$O(b^m)$
Space complexity	$O(b^{d+1})$	$O(bm)$

*admissible if action costs are all identical

b :	branching factor
d :	depth of the shallowest goal state
m :	maximum depth of the state space

NEXT WEEK

Dijkstra's shortest-path algorithm

- Dealing with loops (general graph search)
- Taking non-uniform costs into account